A MATLAB to C vectorizing compiler exploiting custom instructions of targeted processors

by
Latifis Ioannis

Department of Informatics and Telecommunications,
University of Peloponnese

A dissertation submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science

February 2017

ABSTRACT

This Ph.D. dissertation presents a MATLAB to C vectorizing compiler exploiting custom instructions of the target processor such as SIMD instructions or scalar math instructions which are supported by hardware. The compiler generates ANSI C code and the derived custom instructions are represented via intrinsic C functions enabling the compatibility of the compiler to any target processor.

Firstly, a parameterized target processor model which is used for the specification of the specialized instruction set is presented. The processor model allows the description of the available customized instructions, operations for packing data to vectors ready for SIMD processing and native data structures such as vector data types. The processor model is used by an instruction selection algorithm to map the MATLAB source code with the available hardware modules of the target architecture. It is also utilized by the type inference compilation stage to infer the result type of the MATLAB function calls that are mapped to customized instructions. The parametrized processor model forms a multi-target MATLAB-to-C compilation framework allowing the generation of optimized code for any target architecture.

Next, the compiler's infrastructure is described with emphasis to the parts that support the data parallel execution and the generation of the vectorized C code. Initially, an instruction selection algorithm is presented that matches the MATLAB operations and functions with the customized instructions of the target architecture. The algorithm selects a suitable intrinsic from that have been specified in the processor model. Subsequently, packing and unpacking statements are introduced in the intermediate representation to enable data parallel execution. The process concerns the insertion of packing (and unpacking) instructions to convert data in packed form (and vice versa) ready for SIMD execution. Afterwards local value numbering is performed to remove the redundant packing/unpacking statements that have been inserted from the previous compilation stage. The code generator is also discussed in the dissertation focusing on the production of the vectorized code and the fixed point code generation. The vectorized code generation includes the production of for-loops that correspond to SIMD blocks as well as the generation of packing instructions and the matched SIMD instructions with vector semantics. The code generation of fixed point data types concerns the production of additional C code to handle the fixed point arithmetic such as shift operations. The compilation framework supports the data parallel processing and it leads to the vectorized code generation exploiting the customized instructions of the target architecture. The generated code achieves significant speed-up against other approaches that don't produce vectorized code or any processor's special instruction.

Finally, a comprehensive evaluation of the generated code by the MATLAB compiler is presented on diverse architectures. The experimental environment includes two ASIP processors (one of them supporting SIMD) that have been developed in IMEC research center and four processors from ARM and x86 architectures. The benchmark composed by eight representative DSP applications using different SIMD widths as well as a variety of data types including floating or fixed point and packed or unpacked types. The performance of the generated code by the compiler has been compared against the generated code by MathWorks Coder showing that the proposed approach achieves substantial speed-up across all target architectures especially when using SIMD processing. Further experiments have been conducted to evaluate the performance of the auto-vectorization of popular C compilers including Clang/LLVM, GCC and MSVC, on the generated code by MathWorks Coder and the scalarized code that have been produced by the MATLAB compiler. The experimental results show that the auto-vectorizing C compilers vectorize only a small percentage of the benchmark's loops concluding that the processor's SIMD units cannot be fully leveraged by auto-vectorization. In contrast, the approach developed in this thesis handles a much broader class of applications.

ΠΕΡΙΛΗΨΗ

Στην παρούσα διδακτορική διατριβή παρουσιάζεται ένας μεταγλωττιστής της γλώσσας MATLAB ο οποίος παράγει κώδικα C αξιοποιώντας εντολές ειδικού σκοπού του εκάστοτε επεξεργαστή στόχου όπως εντολές επεξεργασίας πολλαπλών δεδομένων (SIMD) και βαθμωτές (scalar) εντολές μαθηματικών συναρτήσεων/λειτουργιών που υποστηρίζονται από το υλικό. Ο μεταγλωττιστής παράγει ANSI C κώδικα και οι εντολές ειδικού σκοπού αναπαρίστανται από εσωτερικές (intrinsic) συναρτήσεις επιτρέποντας τη συμβατότητα του μεταγλωττιστή σε οποιοδήποτε επεξεργαστή (και το μεταγλωττιστή του).

Αρχικά, παρουσιάζεται ένα παραμετρικό μοντέλο επεξεργαστή, το οποίο χρησιμοποιείται για την περιγραφή των εντολών ειδικού σκοπού. Το μοντέλο αυτό επιτρέπει την περιγραφή εξειδικευμένων εντολών, λειτουργίες εισαγωγής δεδομένων σε διανύσματα (packing) για παράλληλη επεξεργασία και εσωτερικές δομές δεδομένων όπως οι τύποι δεδομένων των διανυσμάτων ενός επεξεργαστή. Το μοντέλο επεξεργαστή χρησιμοποιείται από έναν αλγόριθμο επιλογής εντολών (instruction selection) με σκοπό να αντιστοιχήσει τον πηγαίο κώδικα εισόδου με τις διαθέσιμες μονάδες του υλικού της αρχιτεκτονικής-στόχου. Το μοντέλο του επεξεργαστή χρησιμοποιείται επίσης από το στάδιο της εξαγωγής τύπων δεδομένων (type inference) ώστε να εξάγεται ο τύπος του αποτελέσματος των συναρτήσεων που αναπαριστόνται από εντολές ειδικού σκοπού. Το παραμετρικό μοντέλο επεξεργαστή διαμορφώνει ένα πλαίσιο μεταγλώττισης υποστηρίζοντας διαφορετικούς επεξεργαστές-στόχους κάτι που έχει ως αποτέλεσμα την παραγωγή βελτιστοποιημένου κώδικα για οποιαδήποτε αρχιτεκτονική.

Στην συνέχεια, περιγράφεται η υποδομή του μεταγλωττιστή με έμφαση στα τμήματα υποστήριξης της παράλληλης επεξεργασίας δεδομένων και παραγωγής διανυσματικού C κώδικα. Αρχικά, παρουσιάζεται ο αλγόριθμος επιλογής εντολών ο οποίος αντιστοιχίζει τις εντολές και συναρτήσεις MATLAB με τις εντολές ειδικού σκοπού της αρχιτεκτονικής-στόχου. Ο αλγόριθμος επιλέγει την κατάλληλη από τις εντολές που περιλαμβάνονται στο μοντέλο του επεξεργαστή. Στη συνέχεια, λειτουργίες εισαγωγής και εξαγωγής δεδομένων σε μεταβλητές διανύσματα εισάγονται στην ενδιάμεση αναπαράσταση για να επιτρέψουν την παραγωγή κώδικα παράλληλης επεξεργασίας. Η διαδικασία αφορά την εισαγωγή των συγκεκριμένων λειτουργιών ώστε να μετατρέπονται τα δεδομένα σε κατάλληλη μορφή για παράλληλη επεξεργασία. Στη συνέχεια, ένας αλγόριθμος αρίθμησης των τοπικών μεταβλητών (local value numbering) εκτελείται με σκοπό την αφαίρεση από την ενδιάμεση αναπαράσταση των περιττών λειτουργιών εισαγωγής/εξαγωγής δεδομένων σε διανύσματα οι οποίες εισάγονται από το προηγούμενο στάδιο μεταγλώττισης. Τέλος, περιγράφεται το στάδιο παραγωγής κώδικα, εστιάζοντας στην παραγωγή κώδικα για παράλληλη επεξεργασίας δεδομένων και την παραγωγή κώδικα για τύπους δεδομένων σταθερής υποδιαστολής. Η παραγωγή διανυσματικού κώδικα περιλαμβάνει την παραγωγή βρόχων οι οποίοι αναπαριστούν τα τμήματα του κώδι-

κα που περιλαμβάνουν διανυσματικές εντολές ειδικού σκοπού, την παραγωγή λειτουργιών εισαγωγής/εξαγωγής δεδομένων σε διανύσματα καθώς επίσης και τις εντολές παράλληλης επεξεργασίας. Η παραγωγή κώδικα για τύπους δεδομένων σταθερής υποδιαστολής αφορά την παραγωγή επιπλέον C κώδικα για τη διαχείριση της αριθμητικής δεδομένων σταθερής υποδιαστολής (π.χ. λειτουργίες ολίσθησης). Ο μεταγλωττιστής υποστηρίζει την παράλληλη επεξεργασία δεδομένων και επιτρέπει την παραγωγή διανυσματικού κώδικα αξιοποιώντας τις εντολές ειδικού σκοπού του εκάστοτε επεξεργαστή-στόχου. Ο παραγόμενος κώδικας επιτυγχάνει αξιοσημείωτη βελτίωση ως προς το χρόνο εκτέλεσης έναντι άλλων τεχνικών οι οποίες δεν παράγουν διανυσματικό κώδικα και/ή δεν αξιοποιούν εντολές ειδικού σκοπού ενός επεξεργαστή.

Στην παρούσα διδακτορική διατριβή παρουσιάζεται επίσης η αξιολόγηση του μεταγλωττιστή σε διαφορετικές αρχιτεκτονικές. Για την αξιολόγηση του μεταγλωττιστή χρησιμοποιήθηκαν δύο επεξεργαστές ειδικών εφαρμογών (ASIP) εκ των οποίων ο ένας περιλαμβάνει εντολές επεξεργασίας πολλαπλών δεδομένων, και τέσσερις επεξεργαστές αρχιτεκτονικών ARM και x86. Για την αξιολόγηση του μεταγλωττιστή, χρησιμοποιήθηκαν οκτώ εφαρμογές ψηφιακής επεξεργασίας σήματος χρησιμοποιώντας διαφορετικά πλάτη παραλληλίας, τύπους δεδομένων κινητής και σταθερής υποδιαστολής καθώς επίσης και τύπους/μορφές δεδομένων για παράλληλη επεξεργασία. Η απόδοση ως προς το χρόνο εκτέλεσης του παραγόμενου κώδικα συγκρίθηκε με την απόδοση του κώδικα που παράγεται από τον MathWorks Coder. Ο προτεινόμενος μεταγλωττιστής επιτυγχάνει σημαντική βελτίωση της απόδοσης (ταχύτητας) σε όλες τις εξεταζόμενες αρχιτεκτονικές και ειδικά όταν χρησιμοποιούνται εντολές επεξεργασίας πολλαπλών δεδομένων. Επιπλεόν πραγματοποιήθηκαν δοκιμές για την αξιολόγηση της αυτόματης διανυσματοποίησης κώδικα (auto-vectorization) από δημοφιλείς μεταγλωττιστές της γλώσσας C όπως οι Clang/LLVM, GCC και MSVC. Οι δοκιμές πραγματοποιήθηκαν στους κώδικες οι οποίοι παράγονται από τον MathWorks Coder και τον προτεινόμενο μεταγλωττιστή. Τα πειραματικά αποτελέσματα έδειξαν ότι οι C μεταγλωττιστές διανυσματοποιουν ένα μικρό ποσοστό των βρόχων που μπορούν πραγματικά να διανυσματοποιηθούν οδηγώντας στο συμπέρασμα ότι οι μονάδες των επεξεργαστών που υποστηρίζουν την παράλλήλη επεξεργασία δεδομένων δεν μπορούν να αξιοποιηθούν πλήρως από αυτούς. Αντίθετα ο προτεινόμενος μεταγλωττιστής υποστηρίζει μια ευρύτερη κατηγορία περιπτώσεων.

ACKNOWLEDGEMENTS

I would like to thank all people who have contributed to the completion of my doctoral studies.

Initially, I would like to thank my supervisor, professor and rector of University of Peloponnese, Konstantinos Masselos about his contribution and help at my Ph.D studies as well as his useful advices all these years.

I would like also to thank and express my appreciation to the supervisor of my research activity at Inter-University Micro-Electronics Center, professor Francky Catthoor for his support and his guidance of my Ph.D.

Special thanks to Grigoris Dimitroulakos, who provided me with his knowledge of compilers construction and his support not only during the period of my Ph.D studies, but also during the elaboration of my undergraduate thesis.

Many thanks to my colleagues at Inter-University Micro-Electronics Center, Karthick Parashar and Hans Cappelle for helping me to carry out the experiments of my research and their guidance on technical issues of my Ph.D.

Finally, I would like to thank my parents and my brother for their invaluable support during the time of undergraduate and postgraduate studies.

CONTENTS

1 INTRODUCTION 40
1.1 Need for Applications Development in High Level Languages . . . 41
1.2 Field of Application 42
1.3 Advantage of Generating C Code Exploiting Custom Instructions . 43
1.3.1 Tools and Methodologies That Do Not Exploit Custom Instructions of Targeted Processors 44
1.3.2 An Example of MATLAB Code That Cannot Be Automatically Vectorized 45
1.4 Brief Review of Related Work 47
1.5 Brief Presentation of the Developed Compiler/Innovative Features 48
1.5.1 Presentation of Compiler's Infrastructure 48
1.5.2 Innovations and Contributions of the Compiler 51
1.6 Thesis Organization 52
2 RELATED WORK 54
2.1 Type Inference Approaches 55
2.2 MATLAB-to-C Compilation 56
2.3 MATLAB Compilation to Hardware 59
2.4 MathWorks Commercial Tools 61
2.5 Auto-vectorization 62
2.5.1 Auto-vectorizing C compilers 63
2.5.2 Limitations of Auto-vectorization 64
2.5.3 Auto-vectorization Evaluation 65
2.5.4 Comparison against auto-vectorizing C compilers 66
2.6 Comparison of the Proposed Compiler Against State-of-the-art . . 66
2.6.1 MATLAB to FORTRAN 68
2.6.2 MATLAB Just-in-time Compilation 68
2.6.3 Aspect oriented Approaches 69
2.7 Compiling MATLAB to Other Efficient Execution Environments . 70
3 COMPILER'S FRONT-END 71
3.1 MATLAB Input Code 71
3.1.1 MATLAB Language Subset That Is Not Supported 72
3.1.2 Built-in functions 73
3.1.3 Annotations 73
3.1.3.1 Annotations of Variable Decelerations 74
3.1.3.2 Other Annotations 75
3.2 Parametrized Processor Model 77

3.2.1 Description of Customized Instructions in XML 79
3.2.1.1 Description of Operands'/Parameters' Types . . . 80
3.2.1.2 Description of Customized Instructions 81
3.2.1.3 Description of Derived C Types 83
3.2.1.4 Description of fixed point semantics 84
3.2.1.5 Description of packing/unpacking operations . . . 86
3.2.1.6 Description of Other Auxiliary Semantics - Operations 90
3.3 Front-end 91
3.4 Conclusion 92
4 COMPILER'S MIDDLE-LAYER 95
4.1 Abstract Syntax Tree Representation 95
4.2 Type Inference 98
4.2.1 Function Calls Type Inference 98
4.2.2 Intrinsics Type Inference 100
4.2.3 Intrinsics Type Inference Example 100
4.2.4 Key Contribution of Type Inference 101
4.3 Instruction Selection 101
4.3.1 Instruction Selection Example 104
4.3.2 Generated Custom Instructions of benchmark 104
4.4 Support for Data Parallel Execution - AST Decomposition 108
4.4.1 AST Decomposition 109
4.4.2 Packing/Unpacking statements Introduction 113
4.5 Redundant Packing/Unpacking Elimination 115
4.5.1 Inserted/Removed Packing and Unpacking Statements of Benchmark 119
4.6 Conclusion 119
5 COMPILER'S BACK-END (CODE GENERATION) 121
5.1 Structure of Generated C Code 122
5.2 Code Generation For Control Flow Statements 124
5.3 Fixed Point Code Generation 127
5.4 Code Generation of Derived Data Types 129
5.5 Code Generation For MATLAB Operations - customized instructions 131
5.6 Scalarized Code Generation 133
5.7 Code Generation of Packing/Unpacking Statements 134
5.8 Vectorized Code Generation 136
5.9 Code Generation statistics of benchmark 138
5.10 Conclusion 139
6 EVALUATION OF THE COMPILER 141
6.1 Executive Summary 141

6.1.1 Executive Summary of Performance on the ARM Architectures . . . 142
6.1.2 Executive Summary of Performance on the x86 Architectures . . . 144
6.1.3 Executive Summary of Performance on Application Specific Instruction Set Processors . . . 145
6.2 Experimental Setup . . . 146
6.2.1 Experimental Environment and Configurations . . . 147
6.2.2 Abbreviations of Diagrams and Tables . . . 148
6.2.3 Benchmark Characteristics . . . 149
6.2.4 Architectures Selection . . . 151
6.3 Results from ARM Architectures . . . 152
6.3.1 Presentation of ARM Architectures . . . 153
6.3.2 Performance of Generated Code on Raspberry PI 2 . . . 155
6.3.3 Performance of Generated Code on Raspberry PI 3 . . . 168
6.4 Results from x86 Architectures . . . 179
6.4.1 Presentation of x86 Architectures . . . 179
6.4.2 Performance of Generated Code on Intel Sandy Bridge (i7-3820) . . . 182
6.4.3 Performance of Generated Code on Intel Ivy Bridge (i7-3770) . . . 195
6.5 Results from BoT ASIP . . . 207
6.5.1 Presentation of BoT Architecture . . . 207
6.5.2 Presentation and Discussion of Results on BoT Architecture 209
6.6 Results from tinyBoT ASIP . . . 213
6.6.1 Presentation and Discussion of Results on TinyBoT Architecture . . . 214
6.7 Comparison of the Proposed Compiler at the Different Architectures 217
6.7.1 Comparison of Generated Code by MathWorks and Proposed Compiler on ARM Processors . . . 217
6.7.2 Comparison of Generated Code by MathWorks and Proposed Compiler on x86 Processors . . . 219
6.7.3 Comparison of the Proposed Compiler on TinyBoT and BoT ASIPs . . . 222
6.8 Comparison Against MathWorks Coder . . . 224
6.8.1 Comparison of Generated Code Without Intrinsics Against MathWorks Coder on ARM and x86 Processors . . . 224
6.8.2 Comparison of Generated Code Without Intrinsics Against MathWorks Coder on TinyBoT ASIP . . . 228
6.9 Examination of Clang/LLVM Aggressive Auto-vectorization Options 230
6.10 Auto-vectorization Evaluation for C compilers . . . 234

6.10.1 Report of Successfully Auto-vectorized Loops by C Compilers . . . 234
6.10.2 Comparison of Auto-vectorizing C Compilers . . . 238
6.11 Comparison of C Compilers on the Generated Code . . . 243
6.12 Compilation Times . . . 246
6.13 Conclusion . . . 247
7 CONCLUSION AND FUTURE WORK . . . 249
7.1 Contribution of Dissertation . . . 250
7.2 Future Work . . . 251
8 APPENDIX . . . 255
8.1 Compiler Options . . . 256
8.2 Compilation Options Used in The Experiments . . . 257
8.3 XML description examples of the target architectures . . . 261
8.3.1 XML description of Bot Processor . . . 261
8.3.2 XML description of tinyBot Processor . . . 262
8.3.3 XML description of ARM and x86 Architectures . . . 263
8.4 Comprehensive Results of Aggressive Clang Optimization Options 265
8.4.1 Results Compiling with Aggressive Clang Optimization Options on Raspberry PI 2 . . . 265
8.4.2 Results Compiling with Aggressive Clang Optimization Options on i7-3770 . . . 265
8.5 Performance of Vectorized Code Against Compiler's Scalarized Code 272
8.5.1 Performance of Vectorized Code Against Compiler's Scalarized Code on Raspberry PI 2 . . . 272
8.5.2 Performance of Vectorized Code Against Compiler's Scalarized Code on Raspberry PI 3 . . . 277
8.5.3 Performance of Vectorized Code Against Compiler's Scalarized Code on x86 Desktop with i7-3820 Processor . . . 282
8.5.4 Performance of Vectorized Code Against Compiler's Scalarized Code on x86 Desktop with i7-3770 Processor . . . 287
8.6 Performance of MathWorks and Compiler's Scalarized Code . . . 292
8.6.1 Performance of MathWorks Generated Code, Compiler's Scalarized Generated Code and Non-vectorized Generated Code on Raspberry PI 2 . . . 292
8.6.2 Performance of MathWorks Generated Code, Compiler's Scalarized Generated Code and Non-vectorized Generated Code on Raspberry PI 3 . . . 300
8.6.3 Performance of MathWorks Generated Code, Compiler's Scalarized Generated Code and Non-vectorized Generated Code on Desktop with i7-3820 Processor . . . 307

8.6.4 Performance of MathWorks Generated Code, Compiler's Scalarized Generated Code and Non-vectorized Generated Code on Desktop with i7-3770 Processor 314
8.7 Performance of Generated Code on x86 Using Alternative Options 321
8.7.1 Performance of Generated Code on x86 Architectures Producing x64 Code 321
8.7.1.1 Performance of Generated Code on Intel i7-3820 Producing x64 Code 322
8.7.1.2 Performance of Generated Code on Intel i7-3770 Producing x64 Code 324
8.7.2 Performance of Generated Code on x86 Architectures Producing x86 Code with SSE 326
8.7.2.1 Performance of Generated Code on Intel i7-3820 Producing x86 with SSE Extension 327
8.7.2.2 Performance of Generated Code on Intel i7-3770 Producing x86 with SSE Extension 329
8.7.3 Performance of Generated Code on x86 architectures Producing x64 Code with SSE 331
8.7.3.1 Performance of Generated Code on Intel i7-3820 Producing x64 with SSE Extension 332
8.7.3.2 Performance of Generated Code on Intel i7-3770 Producing x64 with SSE Extension 334

LIST OF FIGURES

Figure 1 Compiler infrastructure. 49
Figure 2 Dependence graph of chapters. 53
Figure 3 Parametrized processor model. 77
Figure 4 Example of Parametrized processor model. 79
Figure 5 UML diagram of classes representing parse tree. 93
Figure 6 UML diagram of classes representing AST tree. 97
Figure 7 Control flow diagram for type inference of functions calls. 99
Figure 8 Type inference example. 101
Figure 9 Instruction Selection example. 104
Figure 10 AST Decomposition example. 110
Figure 11 Redundant packing/unpacking Elimination example. . . . 118
Figure 12 Dependence graph of back-end components. 121
Figure 13 Control flow diagram for the generation of fixed point addition.128
Figure 14 Speed-up of fixed point generated code on Raspberry PI 3 143
Figure 15 Speed-up of floating point generated code on Raspberry PI 3143
Figure 16 Speed-up of fixed point generated code on Core i7-3770 . . 144
Figure 17 Speed-up of floating point generated code on Core i7-3770 145
Figure 18 Speed-up of generated code on ASIPs 146
Figure 19 Speed-up comparing with MathWorks compiler on PI 2 using Clang/LLVM with packed fixed point data 157
Figure 20 Speed-up comparing with MathWorks compiler on PI 2 using GCC with packed fixed point data 158
Figure 21 Speed-up comparing with MathWorks compiler on PI 2 using MSVC with packed fixed point data 158
Figure 22 Speed-up comparing with MathWorks compiler on PI 2 using Clang/LLVM with unpacked fixed point data 159
Figure 23 Speed-up comparing with MathWorks compiler on PI 2 using GCC with unpacked fixed point data 160
Figure 24 Speed-up comparing with MathWorks compiler on PI 2 using MSVC with unpacked fixed point data 160
Figure 25 Speed-up comparing with MathWorks compiler on PI 2 using Clang/LLVM with packed floating point data 161
Figure 26 Speed-up comparing with MathWorks compiler on PI 2 using GCC with packed floating point data 162
Figure 27 Speed-up comparing with MathWorks compiler on PI 2 using MSVC with packed floating point data 162

Figure 28 Speed-up comparing with MathWorks compiler on PI 2 using Clang/LLVM with unpacked floating point data 163
Figure 29 Speed-up comparing with MathWorks compiler on PI 2 using GCC with unpacked floating point data 163
Figure 30 Speed-up comparing with MathWorks compiler on PI 2 using MSVC with unpacked floating point data 164
Figure 31 Performance of vectorized code with unpacked fixed point data types versus packed fixed point data types using Clang/LLVM on Raspberry PI 2 165
Figure 32 Performance of vectorized code with unpacked fixed point data types versus packed fixed point data types using GCC on Raspberry PI 2 166
Figure 33 Performance of vectorized code with unpacked fixed point data types versus packed fixed point data types using MSVC on Raspberry PI 2 166
Figure 34 Performance of vectorized code with unpacked floating point data types versus packed floating point data types using Clang/LLVM on Raspberry PI 2 167
Figure 35 Performance of vectorized code with unpacked floating point data types versus packed floating point data types using GCC on Raspberry PI 2 167
Figure 36 Performance of vectorized code with unpacked floating point data types versus packed floating point data types using MSVC on Raspberry PI 2 167
Figure 37 Speed-up comparing with MathWorks compiler on PI 3 using Clang/LLVM with packed fixed point data 168
Figure 38 Speed-up comparing with MathWorks compiler on PI 3 using GCC with packed fixed point data 169
Figure 39 Speed-up comparing with MathWorks compiler on PI 3 using MSVC with packed fixed point data 170
Figure 40 Speed-up comparing with MathWorks compiler on PI 3 using Clang/LLVM with unpacked fixed point data 170
Figure 41 Speed-up comparing with MathWorks compiler on PI 3 using GCC with unpacked fixed point data 171
Figure 42 Speed-up comparing with MathWorks compiler on PI 3 using MSVC with unpacked fixed point data 171
Figure 43 Speed-up comparing with MathWorks compiler on PI 3 using Clang/LLVM with packed floating point data 172
Figure 44 Speed-up comparing with MathWorks compiler on PI 3 using GCC with packed floating point data 172

Figure 45 Speed-up comparing with MathWorks compiler on PI 3 using MSVC with packed floating point data 172
Figure 46 Speed-up comparing with MathWorks compiler on PI 3 using Clang/LLVM with unpacked floating point data 173
Figure 47 Speed-up comparing with MathWorks compiler on PI 3 using GCC with unpacked floating point data 173
Figure 48 Speed-up comparing with MathWorks compiler on PI 3 using MSVC with unpacked floating point data 174
Figure 49 Performance of vectorized code with unpacked fixed point data types versus packed fixed point data types using Clang/LLVM on Raspberry PI 3 175
Figure 50 Performance of vectorized code with unpacked fixed point data types versus packed fixed point data types using GCC on Raspberry PI 3 176
Figure 51 Performance of vectorized code with unpacked fixed point data types versus packed fixed point data types using MSVC on Raspberry PI 3 176
Figure 52 Performance of vectorized code with unpacked floating point data types versus packed floating point data types using Clang/LLVM on Raspberry PI 3 177
Figure 53 Performance of vectorized code with unpacked floating point data types versus packed floating point data types using GCC on Raspberry PI 3 177
Figure 54 Performance of vectorized code with unpacked floating point data types versus packed floating point data types using MSVC on Raspberry PI 3 177
Figure 55 Speed-up comparing with MathWorks compiler on i7-3820 using Clang/LLVM with packed fixed point data 184
Figure 56 Speed-up comparing with MathWorks compiler on i7-3820 using GCC with packed fixed point data 184
Figure 57 Speed-up comparing with MathWorks compiler on i7-3820 using MSVC with packed fixed point data 184
Figure 58 Speed-up comparing with MathWorks compiler on i7-3820 using Clang/LLVM with unpacked fixed point data 186
Figure 59 Speed-up comparing with MathWorks compiler on i7-3820 using GCC with unpacked fixed point data 186
Figure 60 Speed-up comparing with MathWorks compiler on i7-3820 using MSVC with unpacked fixed point data 186
Figure 61 Speed-up comparing with MathWorks compiler on i7-3820 using Clang/LLVM with packed floating point data 188

Figure 62 Speed-up comparing with MathWorks compiler on i7-3820 using GCC with packed floating point data 188
Figure 63 Speed-up comparing with MathWorks compiler on i7-3820 using MSVC with packed floating point data 188
Figure 64 Speed-up comparing with MathWorks compiler on i7-3820 using Clang/LLVM with unpacked floating point data 190
Figure 65 Speed-up comparing with MathWorks compiler on i7-3820 using GCC with unpacked floating point data 190
Figure 66 Speed-up comparing with MathWorks compiler on i7-3820 using MSVC with unpacked floating point data 190
Figure 67 Performance of vectorized code with unpacked fixed point data types versus packed fixed point data types using Clang/LLVM on desktop with i7-3820 191
Figure 68 Performance of vectorized code with unpacked fixed point data types versus packed fixed point data types using GCC on desktop with i7-3820 192
Figure 69 Performance of vectorized code with unpacked fixed point data types versus packed fixed point data types using MSVC on desktop with i7-3820 193
Figure 70 Performance of vectorized code with unpacked floating point data types versus packed floating point data types using Clang/LLVM on desktop with i7-3820 194
Figure 71 Performance of vectorized code with unpacked floating point data types versus packed floating point data types using GCC on desktop with i7-3820 194
Figure 72 Performance of vectorized code with unpacked floating point data types versus packed floating point data types using MSVC on desktop with i7-3820 194
Figure 73 Speed-up comparing with MathWorks compiler on i7-3770 using Clang/LLVM with packed fixed point data 196
Figure 74 Speed-up comparing with MathWorks compiler on i7-3770 using GCC with packed fixed point data 197
Figure 75 Speed-up comparing with MathWorks compiler on i7-3770 using MSVC with packed fixed point data 197
Figure 76 Speed-up comparing with MathWorks compiler on i7-3770 using Clang/LLVM with unpacked fixed point data 198
Figure 77 Speed-up comparing with MathWorks compiler on i7-3770 using GCC with unpacked fixed point data 198
Figure 78 Speed-up comparing with MathWorks compiler on i7-3770 using MSVC with unpacked fixed point data 199

Figure 79 Speed-up comparing with MathWorks compiler on i7-3770 using Clang/LLVM with packed floating point data 200
Figure 80 Speed-up comparing with MathWorks compiler on i7-3770 using GCC with packed floating point data 200
Figure 81 Speed-up comparing with MathWorks compiler on i7-3770 using MSVC with packed floating point data 200
Figure 82 Speed-up comparing with MathWorks compiler on i7-3770 using Clang/LLVM with unpacked floating point data 202
Figure 83 Speed-up comparing with MathWorks compiler on i7-3770 using GCC with unpacked floating point data 202
Figure 84 Speed-up comparing with MathWorks compiler on i7-3770 using MSVC with unpacked floating point data 202
Figure 85 Performance of vectorized code with unpacked fixed point data types versus packed fixed point data types using Clang/LLVM on desktop with i7-3770 203
Figure 86 Performance of vectorized code with unpacked fixed point data types versus packed fixed point data types using GCC on desktop with i7-3770 204
Figure 87 Performance of vectorized code with unpacked fixed point data types versus packed fixed point data types using MSVC on desktop with i7-3770 205
Figure 88 Performance of vectorized code with unpacked floating point data types versus packed floating point data types using Clang/LLVM on desktop with i7-3770 206
Figure 89 Performance of vectorized code with unpacked floating point data types versus packed floating point data types using GCC on desktop with i7-3770 206
Figure 90 Performance of vectorized code with unpacked floating point data types versus packed floating point data types using MSVC on desktop with i7-3770 206
Figure 91 BoT architecture. 208
Figure 92 Speed up comparing with MathWorks compiler on BoT using packed data 210
Figure 93 Speed up comparing with MathWorks compiler on BoT using unpacked data 211
Figure 94 Performance of vectorized code with unpacked data types versus packed data types on BoT 211
Figure 95 Executions times of vectorized code with SIMD width 8 comparing to SIMD width 4 on BoT using packed data 212
Figure 96 Executions times of vectorized code with SIMD width 8 comparing to SIMD width 4 on BoT using unpacked data 212

Figure 97 Execution times of FFT comparing with optimized versions on BoT 213
Figure 98 Speed up comparing with MathWorks compiler on TinyBoT.214
Figure 99 Execution times of FFT comparing with optimized versions on tinyBoT 216
Figure 100 Speed-up of MathWorks and scalarized with no intrinsics generated code on PI 3 compared to PI 2 217
Figure 101 Speed-up of vectorized generated code on PI 3 compared to PI 2 218
Figure 102 Difference between PI 2 and PI 3 of the speed-up achieved by the vectorized generated code against the MathWorks generated code 219
Figure 103 Speed-up of MathWorks and scalarized with no intrinsics generated code on i7-3770 compared to i7-3820 220
Figure 104 Speed-up of vectorized generated code on i7-3770 compared to i7-3820 221
Figure 105 Difference between i7-3770 and i7-3820 of the speed-up achieved by the vectorized generated code against the MathWorks generated code 221
Figure 106 Execution times of TinyBoT versus BoT with packed data types223
Figure 107 Execution times of TinyBoT versus BoT with unpacked data types 223
Figure 108 Speed-up of MathWorks generated code compared to scalarized generated code without intrinsics 225
Figure 109 Speed-up of MathWorks floating point generated code compared to scalarized floating point generated code without intrinsics on PI 2 using MSVC 225
Figure 110 Speed-up of MathWorks fixed point generated code compared to scalarized fixed point generated code without intrinsics on PI 3 using Clang/LLVM 226
Figure 111 Speed-up of MathWorks floating generated code compared to scalarized floating point generated code without intrinsics on i7-3820 using GCC 227
Figure 112 Performance of MathWorks code and generated code without intrinsics on tinyBoT. 229
Figure 113 Normalized execution times of MathWorks fixed point generated code, compiled with aggressive Clang auto-vectorization options232
Figure 114 Normalized execution times of MathWorks fixed point generated code, compiled with aggressive superword-level parallelism Clang options 232

Figure 115 Normalized execution times of scalarized floating point generated code, compiled with aggressive Clang auto-vectorization options . . . 232
Figure 116 Normalized execution times of scalarized floating point generated code, compiled with aggressive superword-level parallelism Clang options . . . 233
Figure 117 Auto-vectorization speed-up of MathWorks generated code 238
Figure 118 Auto-vectorization speed-up of scalarized without intrinsics generated code . . . 238
Figure 119 GCC auto-vectorization speed-up of MathWorks floating point generated code on i7-3820 . . . 240
Figure 120 MSVC auto-vectorization speed-up of scalarized floating point generated code on i7-3820 . . . 240
Figure 121 GCC auto-vectorization speed-up of scalarized fixed point generated code on PI 3 . . . 240
Figure 122 Auto-vectorization speed-up of MathWorks fixed point generated code . . . 241
Figure 123 Auto-vectorization speed-up of MathWorks floating point generated code . . . 242
Figure 124 Auto-vectorization speed-up of scalarized fixed point generated code . . . 242
Figure 125 Auto-vectorization speed-up of scalarized floating point generated code . . . 242
Figure 126 Performance of the MathWorks generated code by C compilers244
Figure 127 Performance of the scalarized generated code by C compilers244
Figure 128 Performance of the vectorized (SIMD width 4) generated code by C compilers . . . 245
Figure 129 Performance of the vectorized (unpacked data with SIMD width 4) generated code by C compilers . . . 245
Figure 130 Performance of the vectorized (SIMD width 8) generated code by C compilers . . . 245
Figure 131 Performance of the vectorized (unpacked data with SIMD width 8) generated code by C compilers . . . 246
Figure 132 Compilations times. . . . 246
Figure 133 Dependence graph of experiments. . . . 255
Figure 134 Normalized execution times of MathWorks fixed point generated code on PI 2, compiled with aggressive Clang auto-vectorization options . . . 266
Figure 135 Normalized execution times of MathWorks fixed point generated code on PI 2, compiled with aggressive superword-level parallelism Clang options . . . 266

Figure 136 Normalized execution times of compiler's scalarized fixed point generated code on PI 2, compiled with aggressive Clang auto-vectorization options 266
Figure 137 Normalized execution times of compiler's scalarized fixed point generated code on PI 2, compiled with aggressive superword-level parallelism Clang options 267
Figure 138 Normalized execution times of MathWorks floating point generated code on PI 2, compiled with aggressive Clang auto-vectorization options 267
Figure 139 Normalized execution times of MathWorks floating point generated code on PI 2, compiled with aggressive superword-level parallelism Clang options 267
Figure 140 Normalized execution times of compiler's scalarized floating point generated code on PI 2, compiled with aggressive Clang auto-vectorization options 268
Figure 141 Normalized execution times of compiler's scalarized floating point generated code on PI 2, compiled with aggressive superword-level parallelism Clang options 268
Figure 142 Normalized execution times of MathWorks fixed point generated code on i7-3770, compiled with aggressive Clang auto-vectorization options 268
Figure 143 Normalized execution times of MathWorks fixed point generated code on i7-3770, compiled with aggressive superword-level parallelism Clang options 269
Figure 144 Normalized execution times of compiler's scalarized fixed point generated code on i7-3770, compiled with aggressive Clang auto-vectorization options 269
Figure 145 Normalized execution times of compiler's scalarized fixed point generated code on i7-3770, compiled with aggressive superword-level parallelism Clang options 269
Figure 146 Normalized execution times of MathWorks floating point generated code on i7-3770, compiled with aggressive Clang auto-vectorization options 270
Figure 147 Normalized execution times of MathWorks floating point generated code on i7-3770, compiled with aggressive superword-level parallelism Clang options 270
Figure 148 Normalized execution times of compiler's scalarized floating point generated code on i7-3770, compiled with aggressive Clang auto-vectorization options 270

Figure 149 Normalized execution times of compiler's scalarized floating point generated code on i7-3770, compiled with aggressive superword-level parallelism Clang options 271
Figure 150 Performance of vectorized code compared to scalarized code using packed fixed point data types, compiled with Clang on Raspberry PI 2 273
Figure 151 Performance of vectorized code compared to scalarized code using unpacked fixed point data types, compiled with Clang on Raspberry PI 2 273
Figure 152 Performance of vectorized code compared to scalarized code using packed floating point data types, compiled with Clang on Raspberry PI 2 273
Figure 153 Performance of vectorized code compared to scalarized code using unpacked floating point data types, compiled with Clang on Raspberry PI 2 274
Figure 154 Performance of vectorized code compared to scalarized code using packed fixed point data types, compiled with GCC on Raspberry PI 2 274
Figure 155 Performance of vectorized code compared to scalarized code using unpacked fixed point data types, compiled with GCC on Raspberry PI 2 274
Figure 156 Performance of vectorized code compared to scalarized code using packed floating point data types, compiled with GCC on Raspberry PI 2 275
Figure 157 Performance of vectorized code compared to scalarized code using unpacked floating point data types, compiled with GCC on Raspberry PI 2 275
Figure 158 Performance of vectorized code compared to scalarized code using packed fixed point data types, compiled with MSVC on Raspberry PI 2 275
Figure 159 Performance of vectorized code compared to scalarized code using unpacked fixed point data types, compiled with MSVC on Raspberry PI 2 276
Figure 160 Performance of vectorized code compared to scalarized code using packed floating point data types, compiled with MSVC on Raspberry PI 2 276
Figure 161 Performance of vectorized code compared to scalarized code using unpacked floating point data types, compiled with MSVC on Raspberry PI 2 276

Figure 162 Performance of vectorized code compared to scalarized code using packed fixed point data types, compiled with Clang on Raspberry PI 3 278
Figure 163 Performance of vectorized code compared to scalarized code using unpacked fixed point data types, compiled with Clang on Raspberry PI 3 278
Figure 164 Performance of vectorized code compared to scalarized code using packed floating point data types, compiled with Clang on Raspberry PI 3 278
Figure 165 Performance of vectorized code compared to scalarized code using unpacked floating point data types, compiled with Clang on Raspberry PI 3 279
Figure 166 Performance of vectorized code compared to scalarized code using packed fixed point data types, compiled with GCC on Raspberry PI 3 279
Figure 167 Performance of vectorized code compared to scalarized code using unpacked fixed point data types, compiled with GCC on Raspberry PI 3 279
Figure 168 Performance of vectorized code compared to scalarized code using packed floating point data types, compiled with GCC on Raspberry PI 3 280
Figure 169 Performance of vectorized code compared to scalarized code using unpacked floating point data types, compiled with GCC on Raspberry PI 3 280
Figure 170 Performance of vectorized code compared to scalarized code using packed fixed point data types, compiled with MSVC on Raspberry PI 3 280
Figure 171 Performance of vectorized code compared to scalarized code using unpacked fixed point data types, compiled with MSVC on Raspberry PI 3 281
Figure 172 Performance of vectorized code compared to scalarized code using packed floating point data types, compiled with MSVC on Raspberry PI 3 281
Figure 173 Performance of vectorized code compared to scalarized code using unpacked floating point data types, compiled with MSVC on Raspberry PI 3 281
Figure 174 Performance of vectorized code compared to scalarized code using packed fixed point data types, compiled with Clang on desktop with i7-3820 processor 283

Figure 175 Performance of vectorized code compared to scalarized code using unpacked fixed point data types, compiled with Clang on desktop with i7-3820 processor 283
Figure 176 Performance of vectorized code compared to scalarized code using packed floating point data types, compiled with Clang on desktop with i7-3820 processor 283
Figure 177 Performance of vectorized code compared to scalarized code using unpacked floating point data types, compiled with Clang on on desktop with i7-3820 processor 284
Figure 178 Performance of vectorized code compared to scalarized code using packed fixed point data types, compiled with GCC on on desktop with i7-3820 processor 284
Figure 179 Performance of vectorized code compared to scalarized code using unpacked fixed point data types, compiled with GCC on on desktop with i7-3820 processor 284
Figure 180 Performance of vectorized code compared to scalarized code using packed floating point data types, compiled with GCC on on desktop with i7-3820 processor 285
Figure 181 Performance of vectorized code compared to scalarized code using unpacked floating point data types, compiled with GCC on on desktop with i7-3820 processor 285
Figure 182 Performance of vectorized code compared to scalarized code using packed fixed point data types, compiled with MSVC on on desktop with i7-3820 processor 285
Figure 183 Performance of vectorized code compared to scalarized code using unpacked fixed point data types, compiled with MSVC on on desktop with i7-3820 processor 286
Figure 184 Performance of vectorized code compared to scalarized code using packed floating point data types, compiled with MSVC on on desktop with i7-3820 processor 286
Figure 185 Performance of vectorized code compared to scalarized code using unpacked floating point data types, compiled with MSVC on on desktop with i7-3820 processor 286
Figure 186 Performance of vectorized code compared to scalarized code using packed fixed point data types, compiled with Clang on desktop with i7-3770 processor 288
Figure 187 Performance of vectorized code compared to scalarized code using unpacked fixed point data types, compiled with Clang on desktop with i7-3770 processor 288

Figure 188 Performance of vectorized code compared to scalarized code using packed floating point data types, compiled with Clang on desktop with i7-3770 processor 288
Figure 189 Performance of vectorized code compared to scalarized code using unpacked floating point data types, compiled with Clang on on desktop with i7-3770 processor 289
Figure 190 Performance of vectorized code compared to scalarized code using packed fixed point data types, compiled with GCC on on desktop with i7-3770 processor 289
Figure 191 Performance of vectorized code compared to scalarized code using unpacked fixed point data types, compiled with GCC on on desktop with i7-3770 processor 289
Figure 192 Performance of vectorized code compared to scalarized code using packed floating point data types, compiled with GCC on on desktop with i7-3770 processor 290
Figure 193 Performance of vectorized code compared to scalarized code using unpacked floating point data types, compiled with GCC on on desktop with i7-3770 processor 290
Figure 194 Performance of vectorized code compared to scalarized code using packed fixed point data types, compiled with MSVC on on desktop with i7-3770 processor 290
Figure 195 Performance of vectorized code compared to scalarized code using unpacked fixed point data types, compiled with MSVC on on desktop with i7-3770 processor 291
Figure 196 Performance of vectorized code compared to scalarized code using packed floating point data types, compiled with MSVC on on desktop with i7-3770 processor 291
Figure 197 Performance of vectorized code compared to scalarized code using unpacked floating point data types, compiled with MSVC on on desktop with i7-3770 processor 291
Figure 198 Performance of MathWorks fixed point generated code compared to MathWorks non-vectorized code compiling with Clang on Raspberry PI 2 292
Figure 199 Performance of compiler's scalarized fixed point code compared to compiler's non-vectorized code compiling with Clang on Raspberry PI 2 293
Figure 200 Performance of compiler's scalarized fixed point code compared to MathWorks fixed point generated code compiling with Clang on Raspberry PI 2 294

Figure 201 Performance of MathWorks floating point generated code compared to MathWorks non-vectorized code compiling with Clang on Raspberry PI 2 294
Figure 202 Performance of compiler's scalarized floating point code compared to compiler's non-vectorized code compiling with Clang on Raspberry PI 2 294
Figure 203 Performance of compiler's scalarized floating point code compared to MathWorks floating point generated code compiling with Clang on Raspberry PI 2 295
Figure 204 Performance of MathWorks fixed point generated code compared to MathWorks non-vectorized code compiling with GCC on Raspberry PI 2 295
Figure 205 Performance of compiler's scalarized fixed point code compared to compiler's non-vectorized code compiling with GCC on Raspberry PI 2 295
Figure 206 Performance of compiler's scalarized fixed point code compared to MathWorks fixed point generated code compiling with GCC on Raspberry PI 2 296
Figure 207 Performance of MathWorks floating point generated code compared to MathWorks non-vectorized code compiling with GCC on Raspberry PI 2 296
Figure 208 Performance of compiler's scalarized floating point code compared to compiler's non-vectorized code compiling with GCC on Raspberry PI 2 296
Figure 209 Performance of compiler's scalarized floating point code compared to MathWorks floating point generated code compiling with GCC on Raspberry PI 2 297
Figure 210 Performance of MathWorks fixed point generated code compared to MathWorks non-vectorized code compiling with MSVC on Raspberry PI 2 297
Figure 211 Performance of compiler's scalarized fixed point code compared to compiler's non-vectorized code compiling with MSVC on Raspberry PI 2 297
Figure 212 Performance of compiler's scalarized fixed point code compared to MathWorks fixed point generated code compiling with MSVC on Raspberry PI 2 298
Figure 213 Performance of MathWorks floating point generated code compared to MathWorks non-vectorized code compiling with MSVC on Raspberry PI 2 298

Figure 214 Performance of compiler's scalarized floating point code compared to compiler's non-vectorized code compiling with MSVC on Raspberry PI 2 298
Figure 215 Performance of compiler's scalarized floating point code compared to MathWorks floating point generated code compiling with MSVC on Raspberry PI 2 299
Figure 216 Performance of MathWorks fixed point generated code compared to MathWorks non-vectorized code compiling with Clang on Raspberry PI 3 301
Figure 217 Performance of compiler's scalarized fixed point code compared to compiler's non-vectorized code compiling with Clang on Raspberry PI 3 301
Figure 218 Performance of compiler's scalarized fixed point code compared to MathWorks fixed point generated code compiling with Clang on Raspberry PI 3 301
Figure 219 Performance of MathWorks floating point generated code compared to MathWorks non-vectorized code compiling with Clang on Raspberry PI 3 302
Figure 220 Performance of compiler's scalarized floating point code compared to compiler's non-vectorized code compiling with Clang on Raspberry PI 3 302
Figure 221 Performance of compiler's scalarized floating point code compared to MathWorks floating point generated code compiling with Clang on Raspberry PI 3 302
Figure 222 Performance of MathWorks fixed point generated code compared to MathWorks non-vectorized code compiling with GCC on Raspberry PI 3 303
Figure 223 Performance of compiler's scalarized fixed point code compared to compiler's non-vectorized code compiling with GCC on Raspberry PI 3 303
Figure 224 Performance of compiler's scalarized fixed point code compared to MathWorks fixed point generated code compiling with GCC on Raspberry PI 3 303
Figure 225 Performance of MathWorks floating point generated code compared to MathWorks non-vectorized code compiling with GCC on Raspberry PI 3 304
Figure 226 Performance of compiler's scalarized floating point code compared to compiler's non-vectorized code compiling with GCC on Raspberry PI 3 304

Figure 227 Performance of compiler's scalarized floating point code compared to MathWorks floating point generated code compiling with GCC on Raspberry PI 3 304
Figure 228 Performance of MathWorks fixed point generated code compared to MathWorks non-vectorized code compiling with MSVC on Raspberry PI 3 305
Figure 229 Performance of compiler's scalarized fixed point code compared to compiler's non-vectorized code compiling with MSVC on Raspberry PI 3 305
Figure 230 Performance of compiler's scalarized fixed point code compared to MathWorks fixed point generated code compiling with MSVC on Raspberry PI 3 305
Figure 231 Performance of MathWorks floating point generated code compared to MathWorks non-vectorized code compiling with MSVC on Raspberry PI 3 306
Figure 232 Performance of compiler's scalarized floating point code compared to compiler's non-vectorized code compiling with MSVC on Raspberry PI 3 306
Figure 233 Performance of compiler's scalarized floating point code compared to MathWorks floating point generated code compiling with MSVC on Raspberry PI 3 306
Figure 234 Performance of MathWorks fixed point generated code compared to MathWorks non-vectorized code compiling with Clang on desktop with i7-3820 308
Figure 235 Performance of compiler's scalarized fixed point code compared to compiler's non-vectorized code compiling with Clang on desktop with i7-3820 308
Figure 236 Performance of compiler's scalarized fixed point code compared to MathWorks fixed point generated code compiling with Clang on desktop with i7-3820 308
Figure 237 Performance of MathWorks floating point generated code compared to MathWorks non-vectorized code compiling with Clang on desktop with i7-3820 309
Figure 238 Performance of compiler's scalarized floating point code compared to compiler's non-vectorized code compiling with Clang on desktop with i7-3820 309
Figure 239 Performance of compiler's scalarized floating point code compared to MathWorks floating point generated code compiling with Clang on desktop with i7-3820 309

Figure 240 Performance of MathWorks fixed point generated code compared to MathWorks non-vectorized code compiling with GCC on desktop with i7-3820 310
Figure 241 Performance of compiler's scalarized fixed point code compared to compiler's non-vectorized code compiling with GCC on desktop with i7-3820 310
Figure 242 Performance of compiler's scalarized fixed point code compared to MathWorks fixed point generated code compiling with GCC on desktop with i7-3820 310
Figure 243 Performance of MathWorks floating point generated code compared to MathWorks non-vectorized code compiling with GCC on desktop with i7-3820 311
Figure 244 Performance of compiler's scalarized floating point code compared to compiler's non-vectorized code compiling with GCC on desktop with i7-3820 311
Figure 245 Performance of compiler's scalarized floating point code compared to MathWorks floating point generated code compiling with GCC on desktop with i7-3820 311
Figure 246 Performance of MathWorks fixed point generated code compared to MathWorks non-vectorized code compiling with MSVC on desktop with i7-3820 312
Figure 247 Performance of compiler's scalarized fixed point code compared to compiler's non-vectorized code compiling with MSVC on desktop with i7-3820 312
Figure 248 Performance of compiler's scalarized fixed point code compared to MathWorks fixed point generated code compiling with MSVC on desktop with i7-3820 312
Figure 249 Performance of MathWorks floating point generated code compared to MathWorks non-vectorized code compiling with MSVC on desktop with i7-3820 313
Figure 250 Performance of compiler's scalarized floating point code compared to compiler's non-vectorized code compiling with MSVC on desktop with i7-3820 313
Figure 251 Performance of compiler's scalarized floating point code compared to MathWorks floating point generated code compiling with MSVC on desktop with i7-3820 313
Figure 252 Performance of MathWorks fixed point generated code compared to MathWorks non-vectorized code compiling with Clang on desktop with i7-3770 315

Figure 253 Performance of compiler's scalarized fixed point code compared to compiler's non-vectorized code compiling with Clang on desktop with i7-3770 315
Figure 254 Performance of compiler's scalarized fixed point code compared to MathWorks fixed point generated code compiling with Clang on desktop with i7-3770 315
Figure 255 Performance of MathWorks floating point generated code compared to MathWorks non-vectorized code compiling with Clang on desktop with i7-3770 316
Figure 256 Performance of compiler's scalarized floating point code compared to compiler's non-vectorized code compiling with Clang on desktop with i7-3770 316
Figure 257 Performance of compiler's scalarized floating point code compared to MathWorks floating point generated code compiling with Clang on desktop with i7-3770 316
Figure 258 Performance of MathWorks fixed point generated code compared to MathWorks non-vectorized code compiling with GCC on desktop with i7-3770 317
Figure 259 Performance of compiler's scalarized fixed point code compared to compiler's non-vectorized code compiling with GCC on desktop with i7-3770 317
Figure 260 Performance of compiler's scalarized fixed point code compared to MathWorks fixed point generated code compiling with GCC on desktop with i7-3770 317
Figure 261 Performance of MathWorks floating point generated code compared to MathWorks non-vectorized code compiling with GCC on desktop with i7-3770 318
Figure 262 Performance of compiler's scalarized floating point code compared to compiler's non-vectorized code compiling with GCC on desktop with i7-3770 318
Figure 263 Performance of compiler's scalarized floating point code compared to MathWorks floating point generated code compiling with GCC on desktop with i7-3770 318
Figure 264 Performance of MathWorks fixed point generated code compared to MathWorks non-vectorized code compiling with MSVC on desktop with i7-3770 319
Figure 265 Performance of compiler's scalarized fixed point code compared to compiler's non-vectorized code compiling with MSVC on desktop with i7-3770 319

Figure 266 Performance of compiler's scalarized fixed point code compared to MathWorks fixed point generated code compiling with MSVC on desktop with i7-3770 319
Figure 267 Performance of MathWorks floating point generated code compared to MathWorks non-vectorized code compiling with MSVC on desktop with i7-3770 320
Figure 268 Performance of compiler's scalarized floating point code compared to compiler's non-vectorized code compiling with MSVC on desktop with i7-3770 320
Figure 269 Performance of compiler's scalarized floating point code compared to MathWorks floating point generated code compiling with MSVC on desktop with i7-3770 320
Figure 270 Performance of vectorized generated code (x64) with packed fixed point data types compared to MathWorks generated code on i7-3820 322
Figure 271 Performance of vectorized generated code (x64) with unpacked fixed point data types compared to MathWorks generated code on i7-3820 322
Figure 272 Performance of vectorized code (x64) with unpacked fixed point data types versus packed fixed point data types on desktop with i7-3820 322
Figure 273 Performance of vectorized generated code (x64) with packed floating point data types compared to MathWorks generated code on i7-3820 323
Figure 274 Performance of vectorized generated code (x64) with unpacked floating point data types compared to MathWorks generated code on i7-3820 323
Figure 275 Performance of vectorized code (x64) with unpacked floating point data types versus packed floating point data types on desktop with i7-3820 323
Figure 276 Performance of vectorized generated code (x64) with packed fixed point data types compared to MathWorks generated code on i7-3770 324
Figure 277 Performance of vectorized generated code (x64) with unpacked fixed point data types compared to MathWorks generated code on i7-3770 324
Figure 278 Performance of vectorized code (x64) with unpacked fixed point data types versus packed fixed point data types on desktop with i7-3770 324

Figure 279 Performance of vectorized generated code (x64) with packed floating point data types compared to MathWorks generated code on i7-3770 325
Figure 280 Performance of vectorized generated code (x64) with unpacked floating point data types compared to MathWorks generated code on i7-3770 325
Figure 281 Performance of vectorized code (x64) with unpacked floating point data types versus packed floating point data types on desktop with i7-3770 325
Figure 282 Performance of vectorized generated code (x86-SSE) with packed fixed point data types compared to MathWorks generated code on i7-3820 327
Figure 283 Performance of vectorized generated code (x86-SSE) with unpacked fixed point data types compared to MathWorks generated code on i7-3820 327
Figure 284 Performance of vectorized code (x86-SSE) with unpacked fixed point data types versus packed fixed point data types on desktop with i7-3820 327
Figure 285 Performance of vectorized generated code (x86-SSE) with packed floating point data types compared to MathWorks generated code on i7-3820 328
Figure 286 Performance of vectorized generated code (x86-SSE) with unpacked floating point data types compared to MathWorks generated code on i7-3820 328
Figure 287 Performance of vectorized code (x86-SSE) with unpacked floating point data types versus packed floating point data types on desktop with i7-3820 328
Figure 288 Performance of vectorized generated code (x86-SSE) with packed fixed point data types compared to MathWorks generated code on i7-3770 329
Figure 289 Performance of vectorized generated code (x86-SSE) with unpacked fixed point data types compared to MathWorks generated code on i7-3770 329
Figure 290 Performance of vectorized code (x86-SSE) with unpacked fixed point data types versus packed fixed point data types on desktop with i7-3770 329
Figure 291 Performance of vectorized generated code (x86-SSE) with packed floating point data types compared to MathWorks generated code on i7-3770 330

Figure 292 Performance of vectorized generated code (x86-SSE) with unpacked floating point data types compared to MathWorks generated code on i7-3770 330
Figure 293 Performance of vectorized code (x86-SSE) with unpacked floating point data types versus packed floating point data types on desktop with i7-3770 330
Figure 294 Performance of vectorized generated code (x64-SSE) with packed fixed point data types compared to MathWorks generated code on i7-3820 332
Figure 295 Performance of vectorized generated code (x64-SSE) with unpacked fixed point data types compared to MathWorks generated code on i7-3820 332
Figure 296 Performance of vectorized code (x64-SSE) with unpacked fixed point data types versus packed fixed point data types on desktop with i7-3820 332
Figure 297 Performance of vectorized generated code (x64-SSE) with packed floating point data types compared to MathWorks generated code on i7-3820 333
Figure 298 Performance of vectorized generated code (x64-SSE) with unpacked floating point data types compared to MathWorks generated code on i7-3820 333
Figure 299 Performance of vectorized code (x64-SSE) with unpacked floating point data types versus packed floating point data types on desktop with i7-3820 333
Figure 300 Performance of vectorized generated code (x64-SSE) with packed fixed point data types compared to MathWorks generated code on i7-3770 334
Figure 301 Performance of vectorized generated code (x64-SSE) with unpacked fixed point data types compared to MathWorks generated code on i7-3770 334
Figure 302 Performance of vectorized code (x64-SSE) with unpacked fixed point data types versus packed fixed point data types on desktop with i7-3770 334
Figure 303 Performance of vectorized generated code (x64-SSE) with packed floating point data types compared to MathWorks generated code on i7-3770 335
Figure 304 Performance of vectorized generated code (x64-SSE) with unpacked floating point data types compared to MathWorks generated code on i7-3770 335

Figure 305 Performance of vectorized code (x64-SSE) with unpacked floating point data types versus packed floating point data types on desktop with i7-3770 335

LIST OF TABLES

Table 1 Comparison of MATLAB to C compilers. 59
Table 2 Comparison of MATLAB compilation to hardware. 61
Table 3 MATLAB built-in functions 74
Table 4 Pragma functions for variable declaration 76
Table 5 XML specification of operands/result type. 81
Table 6 XML specification of customized instructions. 82
Table 7 XML specification of derived C types. 84
Table 8 XML specification of shift operations. 85
Table 9 XML specification of reciprocal operations. 85
Table 10 XML specification of packing/unpacking attributes. 87
Table 11 Packing operations. 87
Table 12 Unpacking operations. 88
Table 13 Matched intrinsics of scalarized fixed point generated code on ARM/x86 106
Table 14 Matched intrinsics of SIMD fixed point generated code on ARM/x86 107
Table 15 Inserted and removed packing and unpacking statements of benchmark. 119
Table 16 Generated code statistics of benchmark. 139
Table 17 Different experimental setups on CPU architectures 148
Table 18 Benchmarks characteristics. 150
Table 19 Comparison between BoT and processors in the CEVA DSP family. 152
Table 20 Reference values (exec. time in µs) used for normalization of results on PI 2 157
Table 21 Reference values (exec. time in µs) used for normalization of vectorized results on PI 2 165
Table 22 Reference values (exec. time in µs) used for normalization of results on PI 3 169
Table 23 Reference values (exec. time in µs) used for normalization of vectorized results on PI 3 175
Table 24 Reference values (exec. time in µs) used for normalization of results on i7-3820 183
Table 25 Reference values (exec. time in µs) used for normalization of vectorized results on i7-3820 192

Table 26 Reference values (exec. time in µs) used for normalization of results on i7-3770 196
Table 27 Reference values (exec. time in µs) used for normalization of vectorized results on i7-3770 204
Table 28 Reference values (instructions count) used for normalization of results at BoT 210
Table 29 Reference values (instructions count) used for normalization at additional FFT results 213
Table 30 Reference values (instructions count) used for normalization of results at tinyBoT 215
Table 31 Reference values (instructions count) used for normalization at additional FFT results 216
Table 32 Reference values (instructions count) used for normalization at ASIPs comparison 222
Table 33 Reference values (exec. time in µs) used for normalization of MathWorks versus scalarized code comparison 226
Table 34 Reference values (instructions count) used for normalization of baseline experiment at tinyBoT 229
Table 35 Reference values (exec. time in µs) used for normalization of aggressive Clang options examination 231
Table 36 Successfully auto-vectorized loops of MathWorks fixed point generated code 235
Table 37 Successfully auto-vectorized loops of MathWorks floating point generated code 236
Table 38 Successfully auto-vectorized loops of scalarized fixed point generated code 236
Table 39 Successfully auto-vectorized loops of scalarized floating point generated code 237
Table 40 Reference values (exec. time in µs) used for normalization of auto-vectorization results 239
Table 41 Compilation options 257
Table 42 Compilation options of Clang/LLVM 258
Table 43 Compilation options of GCC 259
Table 44 Compilation options of MSVC 260
Table 45 Reference values (exec. time in µs) used for normalization of aggressive Clang options examination 265
Table 46 Reference values (exec. time in µs) used for normalization of results on PI 2 272
Table 47 Reference values (exec. time in µs) used for normalization of results on PI 3 277

Table 48 Reference values (exec. time in µs) used for normalization of results on desktop with i7-3820 processor 282
Table 49 Reference values (exec. time in µs) used for normalization of results on desktop with i7-3770 processor 287
Table 50 Reference values (exec. time in µs) used for normalization of results on Raspberry PI 2 293
Table 51 Reference values (exec. time in µs) used for normalization of results on Raspberry PI 3 300
Table 52 Reference values (exec. time in µs) used for normalization of results on desktop with i7-3820 processor 307
Table 53 Reference values (exec. time in µs) used for normalization of results on desktop with i7-3770 processor 314
Table 54 Reference values (exec. time in µs) used for normalization of aggressive Clang options examination 321
Table 55 Reference values (exec. time in µs) used for normalization of aggressive Clang options examination 326
Table 56 Reference values (exec. time in µs) used for normalization of aggressive Clang options examination 331

LISTINGS

1.1 MATLAB example code that cannot be automatically vectorized 46
1.2 Generated C code by MathWorks compiler 46
1.3 Generated SIMD code by current compiler 46
1.4 LLVM pseudocode using as input the generated code from MathWorks compiler 47
1.5 LLVM pseudocode generated by SIMD C code 47
1.6 FFT-32 stage 1, MATLAB code annotated for SIMD generation 50
1.7 FFT-32 stage 1, SIMD C code 50
1.8 Snippet of parametrized processor model for code example 50

3.1 Example of pragmas for definition of SIMD block. 76
3.2 Example of pragmas for handling the fixed point arithmetic. 77
3.3 Example of types XML specification. 80
3.4 Example of customized instructions XML specification 82
3.5 Example of MATLAB code with matching operation and function. 83
3.6 Generated C code with matched operation and function. 83
3.7 Example of derived C types XML specification 84
3.8 Example of MATLAB code with variable declarations 84
3.9 Generated C code with derived C types 84
3.10 Example of shift and reciprocal XML specification. 86
3.11 Example of MATLAB code with shift and reciprocal operations. 86
3.12 Generated C code with shift and reciprocal operations. 86
3.13 Example of packing/unpacking XML specification 88
3.14 Function C prototypes of packing/unpacking operations 88
3.15 Example of MATLAB code with packing/unpacking operations at scalarized code generation. 89
3.16 Generated C code with packing/unpacking operations at scalarized code generation. 89
3.17 Example of MATLAB code with packing/unpacking operations at vectorized code generation. 90
3.18 Generated C code with packing/unpacking operations at vectorized code generation. 90
3.19 Example of auxilary semantics XML specification 91

4.1 Customized instructions example 111
4.2 Decomposed Customized instructions example 111

4.3 Function example . . . 112
4.4 Decomposed Function example . . . 112
4.5 Array Concatenation example . . . 113
4.6 Decomposed Array Concatenation example . . . 113
4.7 Packing/unpacking example . . . 115
4.8 Packing/unpacking example after AST decomposition . . . 115

5.1 Compiler's output example (MATLAB file). . . . 122
5.2 Compiler's output example (C header file). . . . 123
5.3 Compiler's output example (defines file). . . . 123
5.4 Compiler's output example (C source file). . . . 123
5.5 Code generation of for-loop example (MATLAB source). . . . 125
5.6 Code generation of for-loop example (C generated code). . . . 125
5.7 Code generation of if-else example (MATLAB source). . . . 126
5.8 Code generation of if-else example (C generated code). . . . 126
5.9 Code generation of while statement example (MATLAB source). . . . 127
5.10 Code generation of while statement example (C generated code). . . . 127
5.11 Fixed point operations example (MATLAB source). . . . 129
5.12 Fixed point operations example (C generated code). . . . 129
5.13 Derived C data types example (MATLAB source). . . . 130
5.14 Derived C data types example (XML processor model) . . . 130
5.15 Derived C data types example (C generated code). . . . 131
5.16 Customized instructions - MATLAB operations example (MATLAB source).132
5.17 Customized instructions - MATLAB operations example (XML processor model) . . . 132
5.18 Customized instructions - MATLAB operations example (C generated code).132
5.19 Scalarized MATLAB code example (MATLAB source). . . . 134
5.20 Scalarized MATLAB code example (C generated code). . . . 134
5.21 Packing/Unpacking code generation example (C generated code). . . . 136
5.22 Vectorized MATLAB code example (MATLAB source). . . . 138
5.23 Vectorized MATLAB code example (C generated code). . . . 138

8.1 Snippet of parametrized processor model for the BoT architecture . . . 262
8.2 Snippet of parametrized processor model for the tinyBoT architecture . . 263
8.3 Snippet of parametrized processor model for the ARM and x86 architectures264

Chapter 1

Introduction

MATLAB [MATLAB, 2016] is among very popular, perhaps the most preferred language for algorithmic and system modeling with several million users worldwide both in industry and academia in different scientific and technical disciplines. In the context of embedded systems and Systems-on-Chip MATLAB is used for the development of executable specification. Consequently, MATLAB compilation to implementation code (e.g. C, VHDL etc) targeting software or hardware is of major importance to bridge the gap between specification and implementation and reduce design time, effort and cost of computer systems and systems-on-chip.

On the other hand, modern processors have been extended beyond the boundaries of traditional architectures which provide only a conventional instruction set of limited operations/tasks. In recent years, the capabilities of cutting edge processors have been enlarged to provide a more efficient environment for the execution of applications which demand intensive processing power. Several instruction-set architectures (ISA) have been expanded to provide parallel computing in the form of Single Instruction, Multiple Data (SIMD). Popular examples of SIMD extensions are the NEON [NEON, 2016] technology of ARM architecture and the SSE [SSE, 2016] technology of the x86 architecture.

Other classes of processors offered for advanced computing are the Domain-Specific Instruction set Processor (DSIP) and Application Specific Instruction Set Processors (ASIP) which offer a very interesting trade-off between implementation efficiency (area and power for given performance) and flexibility/programmability (to implement different algorithms with non overlapping execution, to accommodate future updates etc.). DSIP processors composed by heterogeneous functional units of generic hardware blocks with optional massive parallelism to enable real-time processing. Contrary to DSIP platforms, the instruction set of ASIP processors is more customized to benefit a specific set of applications constraining their reusability [Fasthuber et al., 2013]. DSIPs and ASIPs are in most cases instantiated as components in Systems-on-Chip (including field programmable ones) in embedded systems.

In the current thesis a MATLAB to C vectorizing compiler is presented exploiting custom instructions, e.g. for SIMD processing and instructions for complex arithmetic present in several processors and ASIPs. The compiler matches the MATLAB input code (functions and operations) with the available instruction set of the target processor and generates ANSI C code representing the custom instructions via specialized

intrinsic functions. The instruction set extension of the target processor is described in a parameterized way using a target processor independent architecture description thereby allowing support for any processor. The information contained in the target processor description is used for the type inference of functions mapped with specialized intrinsics to determine the type of function call. Furthermore, the compiler is able to produce different data types such as fixed point, floating point or integers as well as complex and non-complex types. The generated code is evaluated on various popular processors and ASIP processors from DSP domain. In the evaluation, the generated code by the compiler is compared against the MathWorks Coder [MathWorks Coder, 2016] generated code in combination with the state of the art optimizers/C compilers (Clang/LLVM [Lopes and Auler, 2014], GCC [GCC, 2016], MSVC [MSVC, 2016]). The evaluation comprises the comparison of the generated code performance by the MATLAB compilers as well as the auto-vectorization techniques which are applied on the generated code.

1.1 Need for Applications Development in High Level Languages

The development of an application demanding an optimized implementation of the algorithm in a given time frame is a difficult procedure. High level languages, like MATLAB, are indicated for the rapid prototyping of the application avoiding the detailed specification of the algorithm where strong programming skills and a lot of effort/time are required. Especially, due to MATLAB semantics the language is recommended for the development of digital signal processing applications (a well supported domain by hardware to meet real-time processing constraints). By using high level languages, the algorithm developer focus on the design of the application without addressing issues which are related to the target architecture. The high level specification alone does not help as the insight required for efficient porting of algorithm to different architectures is not captured. Furthermore, the abstract specification of the algorithm allows more opportunities to the compilation flow for depended to architecture optimizations, leading to a more efficient execution code.

Currently algorithm developers specify and explore an algorithm first at the MATLAB level. Where performance (such as latency and throughput) of the code matters, the limitations of existing compilers requires developers to spend effort writing code in low level languages such as C or VHDL for every target type. Especially, the difficulty of manual translation of algorithm in a lower level language is raised in relation with the always increasing application complexity and technology capabilities. This design flow is too time-consuming and erroneous and thus there is strong demand to effectively raise the abstraction level.

The aforementioned reasons create the need for design automation in the form of compilers from high level specifications to implementation code (e.g. C or VHDL etc) to meet time related to market constraints of embedded systems markets. The greatest advantage of the proposed compiler is that it raises the abstraction of algorithm development by effectively supporting generation of target specific optimized C code automatically. Thus the compiler allows algorithm developers to focus on a higher level of abstraction (MATLAB coding). Utilizing the proposed Matlab to C compiler, developers can avoid spending a lot of effort worrying about the right usage of specific custom instructions at a low level abstraction such as the C language. Moreover, the MATLAB code can be re-used for the compilation of the application to other architectures with only minor changes at the specification of customized instructions.

The most challenging part of the MATLAB compilation is the addressing of the type inference problem, the conversion of floating point MATLAB algorithm to fixed point representation and the code generation system for complex instructions in order to implement an efficient translation path from the MATLAB input code to a lower level representation. The majority of research activity have focused at the efficient compilation of the MATLAB code resolving the specific issues without leveraging the architecture's features. For this reason, the research presented in the current thesis has focused on exploitation of the target processor/ASIP where substantial speed-up can be achieved.

1.2 Field of Application

MATLAB programming has a wide range of application in fields like math, statistics, Digital Signal Processing (DSP), image processing and computer vision, finance and biology. Although the compiler can be used for the compilation of applications that are derived from these fields, the current study focuses at the application of the compiler in the DSP domain. The compiler is evaluated within the domain of wireless -signal processing applications. Popular kernels such as: CFO, FFT, mean, FIR, CORDIC and QR-decomposition. The selected kernels offer various code characteristics such as data dependencies, control flow, array dimensions and diversity of indexing instances and SIMD operations. These properties are representative for the complete evaluation of the compiler's generated code. Furthermore, the specific domain in most of the cases allows parallel computing which is the factor with the greatest impact regarding the speed-up which can be achieved via the generation of vectorized C code. Thus, the selected kernels also allow the evaluation of auto-vectorizing C compilers on the generated code by the MATLAB compilers. The experimental applications are discussed more detail at section 6.2.

The generated code have been executed and evaluated on various ASIPs and general purpose processors. More specifically, two different ASIP processors (one of them supporting SIMD), two processors of ARM architecture and two processors of x86 architecture have been employed. The two ASIPs are derived from IMEC ADRES template [Mei

et al., 2003] which is used for wireless signal processing applications delivering high performance and energy efficiency. The ASIP processor supporting SIMD processing was primary selected due to the VLIW paradigm which is used in this class of processors and helps in instantiating several heterogeneous functional units. Several instantiations of the VLIW data-paths are typically dedicated to SIMD data-paths to benefit from the data-level parallelism (DLP). Secondly, the ASIP processors provide flexibility and expandability of the architecture regarding the different application scenarios. This characteristic meets the re-targetability of the compiler which matches the MATLAB code with the specified custom instructions of the target processor.

The ARM and x86 are the most dominant commercial ISAs providing instruction set extensions for SIMD processing accelerating the MATLAB generated code. The generated code may be executed on these processors using different operating systems and different C compilers. Thus, the benchmarking of the compiler can be carried out using different configurations making the benchmarking results more reliable. Finally, popular compilers like Clang/LLVM, GCC, or MSVC, which are used in the benchmarking, perform many optimizations (such as auto-vectorization) at the C generated code and in combination with the MathWorks Coder constitute the most competitive solution against the proposed vectorizing compiler.

1.3 Advantage of Generating C Code Exploiting Custom Instructions of Targeted Processors

The majority of the compilers don't automatically generate code producing instructions from ISA extensions (ISE) of the target processor. The conventional compilers generate code using only the basic ISA and they don't produce the highest quality code could be achieved in relation with the available architecture capabilities. Furthermore, the instruction level parallelism (ILP) is provided in VLIW architectures cannot be efficiently exploited due to lack of advanced instruction selection algorithms to match the input code with target processor/ASIP function units. This disadvantage has a greater impact especially in the embedded systems domain. The embedded processors (like ASIPs) provide complicated, expendable and expensive, in the context of time, tasks/operations/instructions via hardware modules which aren't exploited by existing C compilers. There is a considerable interest in academic community [Murray and Franke, 2012], [Arnold and Corporaal, 2001], [Scharwaechter et al., 2007], [Almer et al., 2012], [Li et al., 2009], [Manilov et al., 2015], [Leupers and Bashford, 2000], [Leupers and Marwedel, 1996] and [Clark et al., 2006] with regard to the efficient instruction selection (in a low level representation) from the custom instructions of the current processor/architecture.

Additionally, auto-vectorization techniques transform the intermediate representation code (loop unrolling) into a suitable form and subsequently match the converted internal code with the available SIMD instructions of the target architecture. Despite that

the optimization looks promising, most of the popular vectorizing C compilers such as GCC, Intel C compiler (ICC), IBM XLC don't apply it extensively. According to the study in [Maleki et al., 2011], only a small amount of loops can be automatically vectorized by commercial vectorizing C compilers. Although the loops theoretically can be vectorized, compilers lack accurate analysis and they fail to perform certain compiler transformations in order to enable vectorization [Maleki et al., 2011].

The above research studies/commercial tools try to exploit custom and SIMD instructions in a low level representation such as C code or an intermediate representation of input C code. On the contrary, the current study introduces a new approach which matches the input code with the available custom and SIMD instructions at a high level representation. The proposed methodology is simpler for instruction selection because the MATLAB code include compact information related to vectorization, since the MATLAB statements may express vector or array operations. Thus, the high level vectorized representation can be directly translated to lower level vectorized C code. Moreover, due to its abstract specification, MATLAB code consists of complicated operations/-functions, usually involving advanced math structures such as complex numbers and multi-dimensional arrays. In embedded systems domain, operations such as complex numbers and their polar coordinate representations are provided by many ASIP processors as application-specific instructions. Therefore, there is a direct correspondence between the MATLAB operations and custom instructions which are offered for the generation of C code exploiting the custom instruction of the targeted processor.

On the contrary, traditional approaches such as the use of MathWorks compilation tools in combination with a C compiler/optimizer (ex. Clang/LLVM) attempting to vectorize the generated code, wouldn't be an optimal solution. After scalarization, the vectorization information contained in MATLAB code may sometimes be eliminated incapacitating the auto-vectorizer to fully exploit the vectorized MATLAB operations/-functions (such as a complex number or a matrix as a C-struct). In that way, the vectorization information would be distributed among the nodes of the intermediate representation tree (internal representation of input code) and it would be hard to export the necessary information for vectorization.

1.3.1 Tools and Methodologies That Do Not Exploit Custom Instructions of Targeted Processors

MathWorks Embedded Coder [MathWorks Embedded Coder, 2016] provides a processor architecture description model similar to the one used by the compiler for the description of the target processor customized instructions. The main disadvantage of MathWorks Embedded Coder is that it supports customization of operations and functions only for scalar and array data types. Embedded Coder neither generates vectorized code nor exploits the features of the target vectorized architectures (e.g. native vector

data types, packing/unpacking elements to/from vector operations etc). Furthermore, MATLAB code including operations with indexing cannot be always efficiently compiled using Embedded Coder and employing array (instead of vector) operations to describe customized C instructions. In such cases Embedded Coder first generates temporary arrays to store sub-array references (related to indexed references) and then the specialized functions are called. The intermediate storage of indexed references (frequently used in MATLAB coding) generates performance overheads in the generated code and may cause even worse performance compared to that of the corresponding code generated by MathWorks Coder [MathWorks Coder, 2016].

The fact that using MathWorks Coder combined with an auto-vectorizing C compiler such as LLVM [Lopes and Auler, 2014] is not an efficient approach for generation high performance code is already well argued in [Maleki et al., 2011]. MATLAB-to-C compiler scalarizes MATLAB array statements and thus largely eliminates information and opportunities (present in MATLAB source code) for vectorization by LLVM or other vectorizing compiler. Furthermore, MathWorks Coder doesn't apply any optimizations or code transformations to ensure that generated loop-nested statements are suitable for vectorization. An auto-vectorizing compiler such as LLVM cannot vectorize any for-loop. Even if LLVM is directed by the developer to vectorize any for-loop, this may not be beneficial and might lead to even worse performance.

1.3.2 An Example of MATLAB Code That Cannot Be Automatically Vectorized

The LLVM auto-vectorizer cannot vectorize any loop especially those, which process data non-sequentially. The example described in the code-snippet in listing 1.1 below presents a representative case where an automatic vectorizing compiler such as LLVM cannot vectorize a given code, although this is still possible. The example code constitutes a common case of MATLAB code which is frequently used in MATLAB programming. The attempt to vectorize MATLAB code in listing 1.1 for Cortex A9 processor with MathWorks compiler (generated code is presented in listing 1.2) in combination with Clang/LLVM compiler with auto-vectorization enabled fails to vectorize for loop corresponding to MATLAB array statement in line 5 of listing 1.2 as LLVM reports. Using the arguments *-mllvm -force-vector-width=8* in clang/LLVM, the compiler is forced to vectorize the for-loop producing the code shown in listing 1.4 (readable pseudocode of LLVM). The code includes operations of packing (and unpacking) elements to vectors. The intermediate result of *'out'* variable after the addition is unpacked and then packed again to be used as an operand in the multiplication. This proves that LLVM doesn't perform code elimination of redundant packing/unpacking operations as applied by our compiler.

Compiling the code of listing 1.1 for ARMv7-A architecture introducing to the parametrized processor model SIMD instructions for multiplication and addition on 32-bit integers, the compiler produces the code shown in listing 1.3. The generated code for this example qualifies that data are packed in array of vector inversely (packing in rows instead of packing in columns). Then, the generated C code is compiled with clang to produce the LLVM code which is shown in listing 1.5. The LLVM code consist of packed data types and only SIMD addition and multiplication are performed. Finally, compiling the MATLAB code of listing 1.1 and declaring the input parameters as unpacked variables, the generated code by current compiler in combination with clang is similar to the code in listing 1.4 but without the intermediate redundant packing and unpacking operations.

```
function [out] = matlabCoder_ex( in1, in2 ) %in1, in2 are 512x16384
  out = int32(zeros(512,16384));
  for k=1:2:512
    out(k,:) = in1(k,:)+in2(k,:);
    out(k,:) = out(k,:).*in2(k,:);
  end
end
```

Listing 1.1: MATLAB example code that cannot be automatically vectorized

```
void matlabCoder_ex(const int in1[8388608], const int in2[8388608], int out[8388608]){
  int k;  int b_k;  int i0;
  for (k = 0; k < 256; k++) {
    b_k = k << 1;
    for (i0 = 0; i0 < 16384; i0++) {
      out[b_k+(i0 << 9)]=in1[b_k+(i0 << 9)] + in2[b_k + (i0 << 9)];
      out[b_k + (i0 << 9)] *= in2[b_k + (i0 << 9)];
    }
  }
}
```

Listing 1.2: Generated C code by MathWorks compiler

```
void matlabCoder_ex(vect_t out[8388608/SW], vect_t in1[8388608/SW], vect_t in2[8388608/SW]){
  for(k=1; k < 513; k=k+2){
    for (si0 = 0; si0 < 16384; si0 = si0 + SW){
      out[(si0+(k-1)*16384)/4]=vaddq_s32(in1[(si0+(k-1)*16384)/SW],in2[(si0+(k-1)*16384)/SW]);
      out[(si0+(k-1)*16384)/4]=vmulq_s32(out[(si0+(k-1)*16384)/SW],in2[(si0+(k-1)*16384)/SW]);
    }
  }
}
```

Listing 1.3: Generated SIMD code by current compiler

```
for loop
  %1 = load the 4 next elements of  in1
  %2 = insert elements %1 to vector
  %3 = load the 4 next elements of  in2
  %4 = insert elements %3 to vector
  %add_res = SIMD_addition %2, %4
  %5 = extract elements of vector %add_res
  store 4 elements of %5 to  out
  %6 = load the 4 next elements of  in2
  %7 = insert elements %6 to vector
  %mul_res = SIMD_multiplication %add_res, %7
  %8 = extract elements of vector %mul_res
  store 4 elements of %8 to  out
end for

```

Listing 1.4: LLVM pseudocode using as input the generated code from MathWorks compiler

```
for loop
  %1 = load i vector from  in1
  %2 = load i vector from  in2
  %add_res = SIMD addition %1, %2
  store %add_res to i vector of  out
  %3 = load i vector from  in2
  %mul_res = SIMD multiplication %add_res, %3
  store %mul_res to i vector of  out
end for
```

Listing 1.5: LLVM pseudocode generated by SIMD C code

1.4 Brief Review of Related Work

Several approaches have been presented for the compilation of MATLAB to languages that provide a more efficient execution environment. One of the first approaches is FALCON [De Rose and Padua, 1999], [DeRose, L. A., 1996] a MATLAB to FORTRAN 90 compiler with main key contribution the static and dynamic inference mechanism supported by a sophisticated symbolic value propagation algorithm. MaJIC [Almási and Padua, 2002], is a just-in-time compiler applying ahead-of-time compilation in MATLAB codes, using a type inference technique similar to FALCON. MATCH [Banerjee et al.,], [Banerjee et al., 1999] and [Banerjee, 2003] compiler and its commercial version, AccelDSP [Banerjee et al., 2004] target high-level synthesis [Micheli, 1994] and translate MATLAB code to a register transfer level HDL supporting fixed point arithmetic. Both tools provide a framework of notations named as directives, that the user can insert in MATLAB code to bridge the gap between MATLAB source and the available computa-

tional structures. OTTER [Quinn et al., 1998b], [Quinn et al., 1998a], MENHIR [Chauveau and Bodin, 1999] and RTExpress [Benincasa et al., 1998] are MATLAB compilers producing SPMD-style C code for parallel code execution (MENHIR can produce C or FORTRAN) relying on libraries such as ScaLapack and MPI message-passing libraries. The MAT2C [Joisha and Banerjee, 2007], a similar to MathWorks compiler [MATLAB compiler, 2016], uses MAJICA [Joisha and Banerjee, 2003a] type inference tool attaining better performance from MathWorks compiler (MCC) [MATLAB compiler, 2016]. MEGHA [Prasad et al., 2011] uses a heuristic algorithm to map data parallel regions of the program (kernels) to heterogeneous processors (CPU and GPU). In [Shei et al., 2011], fragments of MATLAB code that incur high overheads, are compiled and scalarized using a modified typed fusion algorithm [Kennedy and Allen, 2002] aiming at parallelizing the scalarized statements. MATISSE [Bispo et al., 2014], [Bispo et al., 2015] compiler focuses on MATLAB to C efficient compilation performing optimizations and transformations on MATLAB code supported by LARA [Cardoso et al., 2012] aspect-Oriented programming language.

1.5 Brief Presentation of the Developed Compiler and Its Innovative Features

The MATLAB-to-C compiler generates optimized C code exploiting the instruction set architecture extensions (ISE) presented in the given ASIP platforms/processors. MATLAB expressions matching the extended instruction set of the targeted processor are exposed in C code in the form of intrinsic functions. Such functions can be exploited by any C compiler supporting the target processor at a later stage including the popular GCC and LLVM/Clang and also (retargetable) ASIP compilers (i.e. Target Suite Tool [ASIP Designer, 2016]). In this way the compiler allows mapping specialized operations to specific hardware modules present in ASIPs and advanced processors. The compiler is target processor independent since the selection of instructions is made using a parameterized processor model. For the type inference of functions mapped to hardware, the parametrized processor model is employed to resolve the function's type result using the specification of the associated specialized instructions. The compiler's back-end may produce either scalarized or SIMD-style C depending on the target processor. SIMD support can be configured with respect to the blocks that are eligible for SIMD code generation and the preferred vector size. Furthermore, the compiler is able to produce different data types such as fixed point, floating point or integers as well as complex and non-complex types. The subsection 1.5.1 present briefly the compiler's architecture while in subsection 1.5.2 the innovations of compiler are discussed.

1.5.1 Presentation of Compiler's Infrastructure

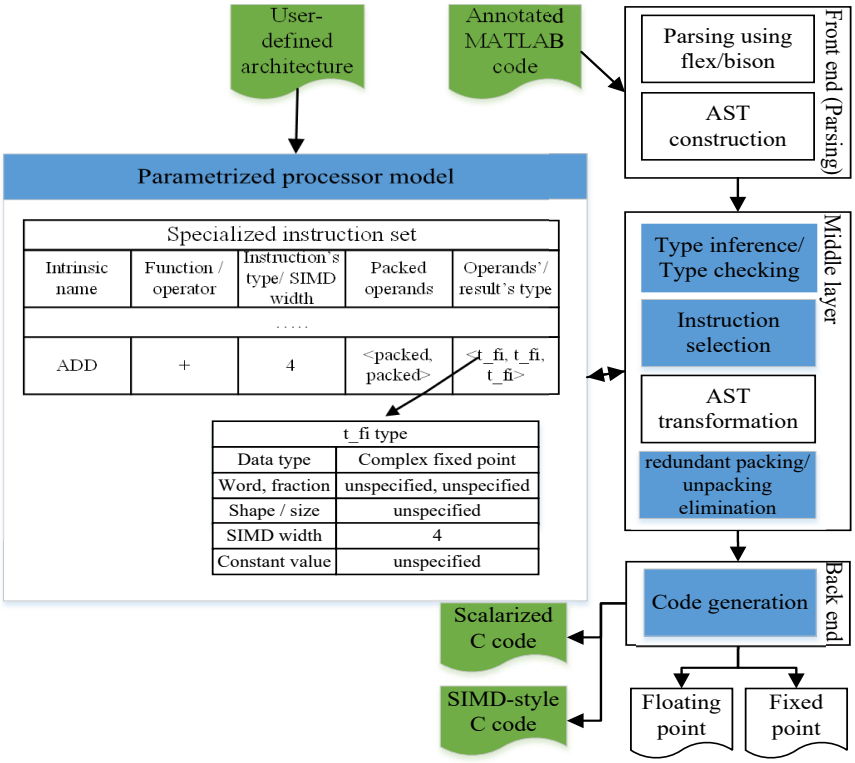

Figure 1: Compiler infrastructure.

Figure 1 depicts the compiler's flow highlighting the innovative contributions. The inputs to the compiler are the annotated application MATLAB code and the instruction set architecture of the target processor in XML. Annotations of MATLAB code are introduced by the developer in the form of pragma functions that comply to MATLAB syntax. MATLAB code with annotations beyond MATLAB syntax could not be used by MathWorks tools thus making very difficult the application development in MATLAB. Pragmas are used to declare the program's parameters as to the data type, the array shape/size and the fixed point attributes (word and fraction length) in the case of fixed point variable. Pragma functions are also used for the selection of the parts of the input code (SIMD blocks) that will be translated into SIMD style allowing the developer to select the preferred vector size of the block's SIMD operations. Listing 1.6 shows the MATLAB code of the first stage of a 32-point FFT annotated for SIMD code generation. The *dec_fixpc_p* pragma functions declare vector variables of complex fixed point data type with word length of 16 and fraction length of 12 while the last two parameters are referred to the dimensions of the variables. The *startSIMD* and *stopSIMD* pragma functions define the SIMD block while the parameter (value 2) of *startSIMD* indicates the

selected SIMD width. Listing 1.7 presents the generated SIMD code of the MATLAB code depicted in listing 1.6 using the parametrized processor model of listing 1.8. The output includes the SIMD block for-loop with step *'SW'* (can be defined by user at C with any value) and loop condition 4 as the operands' size for the specialized instructions. The complex fixed point addition, subtraction and multiplication have been mapped to *ADD*, *SUB*, *VMUL* intrinsic functions which has been specified in the parametrized processor model, while flattening has been performed for the generation of the vector indices.

```
dec_fixpc_p('in','16','12',4,8);  %variable declerations
dec_fixpc_p('twd_fac','16','12',4,1);
dec_fixpc_p('out','16','12',4,8);
startSIMD('2');
for k=1:size(in,1) %stage 1
  out(:,k) = in(:,k) +in(:,k+4);
  out(:,k+4) = (in(:,k) - in(:,k+4)).* twd_fac;
end
... %code for stage 2 and 3
stopSIMD();
```

Listing 1.6: FFT-32 stage 1, MATLAB code annotated for SIMD generation

```
for(si=0; si < 4; si=si+SW){ //SIMD block for-loop
  for(k=1; k < 5; k=k+1){
    out[(si+(k-1)*4)/SW]=ADD(in[(si+(k-1)*4)/SW],in[(si+(k+3)*4)/SW]);
    tmp0[si/SW]=SUB(in[(si+((k-1)*4))/SW],in[(si+((k + 3)*4))/SW]);
    out[(si+((k + 3)*4))/SW]=VMUL(tmp0[si/SW], twd_fac [si/SW]);
  }
  ... //code of the rest FFT stages is continued here
}
```

Listing 1.7: FFT-32 stage 1, SIMD C code

```
<!---types--->
<type id="f_c" dt="fixp" complex="true"></type>
<!--Customized operations-->
<instruction name="ADD" type="SIMD" op="+" pack="p,p" SIMD_width ="true" op_types="f_c,f_c">
</instruction>
<instruction name="SUB" type="SIMD" op="-" pack="p,p" SIMD_width ="true" op_types="f_c,f_c">
</instruction>
<instruction name="VMUL" type="SIMD" op="*" pack="p,p" SIMD_width ="true" op_types="f_c,f_c">
</instruction>
```

Listing 1.8: Snippet of parametrized processor model for code example

The parameterized processor model describes the instruction set architecture of the target processor including custom instructions. The model includes a list with information about the instructions. For each instruction the function name or operation type, the corresponding instruction name in C, the instruction type (scalar, SIMD or array),

the operands needed to be packed for SIMD processing and the operands' (and result's) types are provided. For the instructions which can be mapped without the requirement of any of the type's attributes, the corresponding attributes can remain unspecified.

Listing 1.8 shows an example of the parametrized processor model using XML specification. Firstly, the operand type *f_c* is defined which is used in the below instructions specification to determine the intrinsic's type operands. Then three customized SIMD instructions are described for the addition, subtraction and multiplication MATLAB operations. The *pack* attribute denotes that the instructions' operands must be in packed form and the *op_types* attribute associates the operands' types with the predefined *f_c* type.

In the compiler's middle-layer, type inference analysis is performed using an approach similar to the one described in [De Rose and Padua, 1999] only for compile-time type detection while the type inference of the function calls mapped to custom instructions is based on the parameterized processor model. For the latter, the type checking of operands' types and the type inference of function call's result is performed involving the specified types of the custom instruction in parametrized processor model. In the next step, the instruction selection pass utilizes the parametrized processor model to map available specialized instructions to function calls and MATLAB operations. To achieve this, the traversal passes to the model, the operands/parameters types, the operation type or function name and information on whether or not the current function or operation is part of a SIMD block. After instruction selection, a multi-pass stage is applied to transform the AST for data parallelization execution. This first pass involves the decomposition of complex MATLAB array expressions to simpler ones retaining the vectorized form of the SIMD operands. In the next step, new statements are created to transfer only the unpacked SIMD operand values' to packed data structure (ready for SIMD processing). Similar statements are introduced in the low-level IR, only for SIMD operation result variable's which have been declared as unpacked data types, transferring the packed result's values to the unpacked data structure. In the next step redundant packing/unpacking elimination is performed to remove the redundant intermediate packing/un-packing statements. After this optimization the SIMD operands are packed only once (unlike to multi reads) and only the results which are used outside the SIMD block are unpacked.

The compiler's back-end consists of the code generation which generates the special instructions as intrinsic functions. The output depends on the developer's SIMD block declarations and may be scalarized where MATLAB array expressions (operations) are translated to C loop nests or SIMD-style C where each SIMD block is implemented as a C for-loop. The for-loops corresponding to SIMD blocks use the specified by the developer vector width as step and the array size of the SIMD block operands as for loop condition. The derived data types can be floating point, integer or fixed point. For the latter, extra C code is generated for handling fixed point arithmetic such as shifting to adjust the operand's fraction length.

1.5.2 Innovations and Contributions of the Compiler

The compiler constitutes an innovation since no compilation frameworks supporting the generation of customized and vectorized C code for any target processor (in particularly useful in ASIP context) currently exist. The proposed approach introduces a multitarget MATLAB-to-C compilation framework exploiting the entire instruction set of a given target processor through the use of a parameterized target processor model. The compiler generates vectorized (SIMD) or scalarized code supporting different data types (including floating point, integer or fixed point) compatible with any standard C/C++ compiler supporting the target architecture and its custom instructions. The compiler is able to generate C code including vectoring semantics such as native vector types, packing/unpacking operations and indexing corresponds to vector arrays. The main contribution of the work presented in this thesis includes:

- A target processor independent MATLAB to vectorizing C compiler that exploits custom instructions present in the target instruction set.
- The description of a parametrized processor model that can be used for the specification of the target architecture ISE. The processor model is exploited by the MATLAB compiler to generate high performance C code.
- An instruction selection algorithm for efficient matching of complex MATLAB operations/functions to custom instructions.
- A methodology for the support of data parallel execution and the generation of vectorized C code.
- A type inference approach for type checking and inference of function calls which are matched to custom instructions using the parametrized processor model.
- Evaluation of the compiler on a variety of ASIPs and processors comparing against MathWorks Coder.

1.6 Thesis Organization

The remainder of this thesis is organized as follows:

- Chapter 2 overviews the existing works of approaches for MATLAB compilation to a more efficient execution environment. Furthermore, auto-vectorization techniques are discussed and an evaluation of auto-vectorizing compilers from literature is presented.
- In chapter 3 the compiler's front-end is presented. The compiler's input is described focusing on the annotations of MATLAB code and the parametrized processor model. The infrastructure of the compiler's front-end is presented as well.

- In chapter 4 the compiler's middle-layer is presented. In this chapter, the internal stages of the compiler's to prepare the input MATLAB code for code generation are described.
- In chapter 5 the compiler's back-end is presented. The code generation of MATLAB input code is described and generated C code examples by the compiler are presented.
- Chapter 6 discusses the evaluation of the compiler. The performance of the compiler's generated code by the compiler is examined comparing to the performance of the generated code by the MathWorks Coder on various processors. A brief discussion of the execution performance of the generated code by both compilers using different compilation techniques among the different architectures/processors is discussed as well.
- Chapter 7 discusses the final conclusion of this thesis. In addition, topics for future work are discussed in the domain of MATLAB compilation.
- The chapter 8 includes the appendix presenting technical details and supplementary diagrams of the compiler's evaluation results.

Figure 2 shows a dependence graph of the chapters.

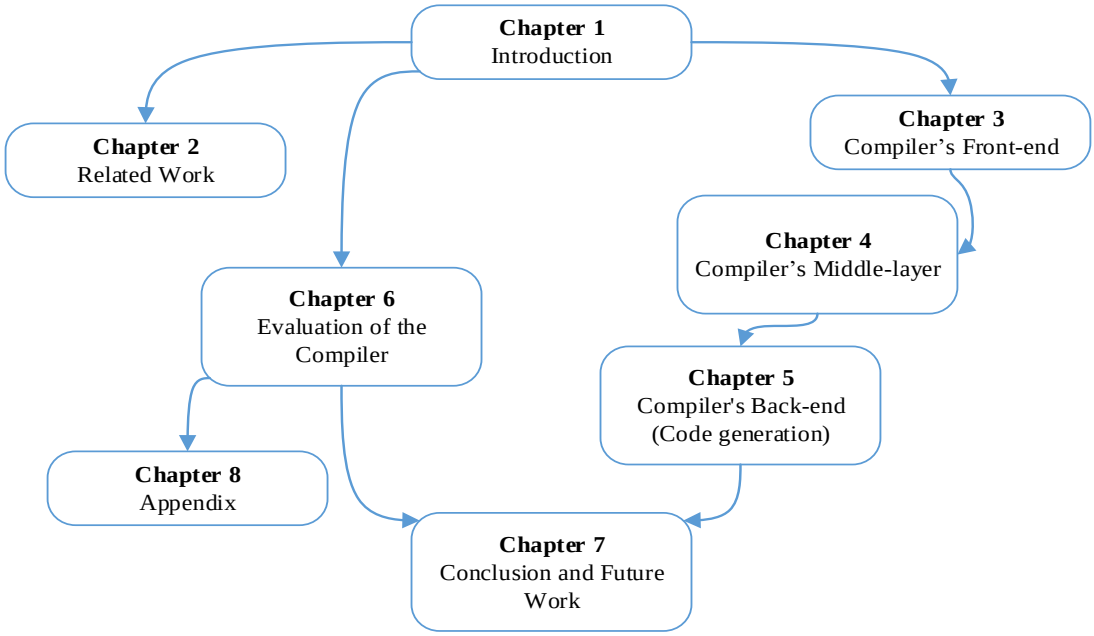

Figure 2: Dependence graph of chapters.

Chapter 2

Related work

The research activity in the domain of MATLAB compilation focused on the type inference analysis and the translation of MATLAB code to lower level representations aiming improvement of performance. Due to MATLAB type-less semantics, sophisticated analysis is required for the type inference of MATLAB code. Furthermore, MATLAB is a scripting language accompanied with an interpretation system. This programming environment comprises performance constraints, especially for a language intended for numerical computing. Thus, there is the need for the compilation of MATLAB to implementation code targeting software or hardware.

Modern architectures include vector units that have been designed to accelerate the processor's performance. The vector extensions can be leveraged either explicitly by programming with SIMD instructions or implicitly by utilizing an auto-vectorizing C compiler. Although, the development of application code with vector instructions can considerably accelerate the performance, it requires effort, deep knowledge of the SIMD architecture and comprises portability constraints. On the other hand, auto-vectorization is a promising optimization method, that transforms loops to vectorized code. There is a considerable interest in academic community about the application of auto-vectorization techniques on C code. However, auto-vectorization is not always applicable due to algorithmic constraints (data dependencies) as well as the low level semantics of the C language, where much data type information is hidden in detailed complex syntax constructs. Hence, as advocated in Chapter 1, we propose to work from the Matlab specification level instead.

This chapter gives an overview of the existing work on the compilation of MATLAB language or alternative to MATLAB languages such as Octave and Scilab. The studies concern the type inference and the translation of MATLAB language to C or FORTRAN providing a more efficient environment such as the execution of generated code on GPUs, high performance parallel computing architectures or reconfigurable multi-Core embedded systems. Furthermore, compilers are also presented for the mapping of MATLAB to heterogeneous computing systems and aspect oriented optimization approaches. At the end of chapter, studies concerning auto-vectorization issues as well as studies evaluating the effectiveness of auto-vectorization are discussed as well.

2.1 Type Inference Approaches

In [De Rose and Padua, 1999] and [De Rose and Padua, 1996] De Rose and Padua present a technique which combines a static and a dynamic inference methodology to translate MATLAB code to FORTRAN. Initially, the source code is transformed to static single statement (SSA) form. Then static analysis operates on SSA to resolve the variables types using a repetitive forward propagation algorithm. Additionally, constant value propagation and an on-demand backward propagation of types are applied. Furthermore, the type inference mechanism generates code that applies dynamic analysis at run time to infer the type of variables with unknown attributes. More specifically, the code that is generated allocates variables with appropriate arrays dimensions while extra variables are produced to handle any possible data type.

The MAGICA [Joisha and Banerjee, 2003a] type inference engine is a tool that determines compile-time the variable types of MATLAB expressions and scripts. The tool infers the value range, data type and array shape of the variables of the given input MATLAB code. It uses its own internal system of types representation, using semantics and mechanisms from Mathematica [Mathematica, 2014]. The MAGICA tool supports a large set of the MATLAB language with a variety of built-in functions. It can be used as add-on, integrated in compilers' front-ends for type-related optimizations, code generation or code annotation and visualization. In studies [Joisha et al., 2001] and [Joisha and Banerjee, 2006], Joisha and Banerjee describe their array inference methodology using an algebraic system to determine the shape of MATLAB variables/arrays. The model is able to cope with unknown array shapes/dimensions in compile-time where symbolic types are propagated for the resolution of the subsequent arrays. The new type inference mechanism have been adopted by the MAGICA tool.

In [Chauhan et al., 2003], a translation system which generates FORTRAN or C linear algebra libraries from MATLAB code is presented. The translator's type inference scheme accepts the MATLAB prototype with a variety of specifications about the library's input parameters. The type inference algorithm transforms the input MATLAB code in graph performing a n-clique algorithm to statically deduce the variables types. Subsequently, the slice-hoisting [Chauhan and Kennedy, 2003] method (based on [De Rose and Padua, 1999]) is applied for the dynamic size inference of the types which have not been resolved. The idea of slice-hoisting technique is to identify slices of code that participate in the computing of the arrays size and hoist them before the first use of the corresponding array variable in order to preallocate the entire array from the beginning of its use. The result of this type inference scheme is a set of specialized variants of library procedures depending on the types that have been inferred.

In [Shei et al., 2009] a type inference approach which specializes the MATLAB input code to different type-based versions is discussed. The approach incorporates constant propagation using data flow analysis to evaluate the statement's type for the variables that can be determined in compile-time enhancing the resolution of variables' types. The

disambiguation of types is applied via partial evaluation [Jones et al., 1993] distinguishing the types can be inferred in compile-time. For variables that cannot be resolved, fall-back code is produced to handle parts of code involving unknown type of variables in run-time. The proposed type inference approach is indicated for optimization applications such as recognizing code patterns, MATLAB vectorization [Birkbeck et al., 2007] and better mapping of operations to the underlying libraries [McFarlin and Chauhan, 2007].

The existing approaches provide adequate solutions for the resolution of the types of MATLAB code. Most of them introduce advanced algorithms for the static type inference in compile-time as well as methods for run-time analysis of the MATLAB variables that haven't been resolved. However, the existing approaches qualify the requirements of the proposed compilation framework where the type inference of the function calls mapped to customized instructions are based on the prototype of the underlying target processor/ASIP instructions. The proposed MATLAB compiler uses a variation of that described in [De Rose and Padua, 1999]. The FALCON project approach was chosen to be implemented in the current compilation framework because it provides the fundamental and well defined solution of the type inference problem. However, the deployed type inference mechanism has been extended according to the specifications of current compiler scheme. The proposed type inference mechanism resolves the output of function calls mapped with customized instructions depending on the hardware implementation. This is achieved thought the description of the customized instruction at the parametrized processor model.

2.2 MATLAB-to-C Compilation

Otter [Quinn et al., 1998b] and [Quinn et al., 1998a] is a MATLAB to C compiler producing SPMD-style C code, suitable for parallel processing using the MPI communication protocol. The compiler transforms the AST inserting additional statements for the manipulation of calls to the run-time library for parallel processing. The compiler's output is C code with calls to the run-time library for the distribution/communication of the data to be processed in parallel along the architecture's nodes. Otter is equipped with an MPI run-time library implementing the initialization of the parallel environment and the communication between platform's nodes. Otter's run-time library also implements parallel operations through calls to third-party numerical libraries such as the linear algebra ScaLAPACK library. The generated code by the Otter compiler can be applied on a distributed-memory multicomputer, a cluster of PCs, a symmetric multiprocessor, or a network of symmetric multiprocessors.

RTExpress [Benincasa et al., 1998] is a MATLAB compilation framework for the execution of MATLAB code on high performance computers. The RTExpress utilizes a target balancing tool to allow the manually parallel partitioning of MATLAB code providing a variety of parallel paradigms: task parallel, pipelined, data parallel, round robin, and mixed mode parallelization. The RTExpress environment acts in combination

with the MCC compiler. More specifically, it modifies the MATLAB input code for parallel execution preventing the generation of for-loops by MCC. After the generation of C code by MCC, RTExpress links the generated code with its Parallel Function Library, MPI Library, Third Party Libraries, and vendor specific libraries.

Menhir [Chauveau and Bodin, 1999] is a multi-target compiler for MATLAB, generating C or FORTRAN code. The compiler allows the generation of sequential or parallel code which is relied on libraries such as ScaLAPACK. Menhir provides a target description system addressing targets programming languages such as FORTRAN and C, or a linear algebra library. Menhir's target system may generate various code styles defining MATLAB language semantics such as: data structures, matrix accessing, MATLAB operators and functions, memory management operations and the form of output statements.

In [Allen, 2005], an automation path from MATLAB to C targeting DSP platforms is presented. The study focus on the conversion of floating point to fixed point arithmetic. It introduces the mode inference to infer the mode of fixed point arithmetic as to the overflow/rounding mode and fixed point precision. Furthermore, it discusses issues concerning the application of generated code on DSPs, the generation of array statements with loop-carried dependencies and the loop-fusion optimization in order to improve the reuse of array references.

MAT2C [Joisha and Banerjee, 2007] is a MATLAB to C translator system similar to the MCC [MATLAB compiler, 2016] MathWorks compiler. The MAT2C uses MAGICA [Joisha and Banerjee, 2003a] tool to statically infer the types of input MATLAB code. In case that the types cannot be resolved, a fall-back code is produced for run-time resolution. MAT2C performs several optimizations such as dead-code elimination, array copy propagation, common-subexpression elimination, constant folding, code selection and array storage coalescence. The translator is able to produce C/C++ code of standard C library or MathWorks math library. Furthermore, MAT2C compiler also applies an optimization [Joisha and Banerjee, 2003b] on the generated code for the memory allocation problem of MATLAB array variables.

MEGHA [Prasad et al., 2011] is a MATLAB compiler framework which enables synergistic execution on heterogeneous processors. MEGHA's output is a combination of C++ and CUDA code. The compiler identifies regions of MATLAB source code as kernels (fragments of code that can be run together on GPU) assigning them to heterogeneous processors (CPU or GPU). For the mapping of kernels to heterogeneous processors, kernels and their associated dependencies are represented in the form of graph. Then, a heuristic algorithm is applied to solve the scheduling problem, minimizing the memory overhead of transferring data from/to GPU. Finally, the compiler implements optimizations such as the transformation of kernels' intermediate arrays to scalar variables (common technique in CUDA programming) and parallel loop reordering to increase the locality.

MATISSE [Bispo et al., 2014], [Bispo et al., 2015] is multi-target MATLAB to C compiler performing optimizations and transformations. The compiler can also produce MATLAB code for validation, testing and monitoring purposes. Another target system of MATISSE is the generation of pragma directives which accelerate the generated code via openCL (OpenACC). MATISSE incorporates LARA [Cardoso et al., 2012], an aspect oriented programming language developed for code instrumentation and analysis, type inference assistance and hardware synthesis optimizations. MATISSE implements type inference with the assistance of LARA which provides information about the variable types. LARA is also utilized by MATISSE for source to source code transformations.

In [Shei et al., 2011], a compilation framework which compiles MATLAB array statements to scalarized C++ code is presented. The compiler uses a modified version of Allen and Kennedy's typed fusion algorithm [Kennedy and Allen, 2002] to group vector/array statements which don't contain dependences among them. Subsequently, the clustered statements are translated to C++ performing scalarization and loop fusion. The generated loop-nested code is always parallelizable, since is produced by vectorized statements. Thus, the compiler's output can be leveraged by high performance parallel architectures.

Table 1 gives an overview of the basic characteristics of existing MATLAB to C compilers by comparing features that are presented in the proposed compiler. The various approaches address a wide range of MATLAB compilation for high performance computing. However, all these compilers produce sequential or parallel C code in SPMD-style targeting mainly distributed architectures. More specifically, the Otter [Quinn et al., 1998b], RTExpress [Benincasa et al., 1998] and Menhir [Chauveau and Bodin, 1999] MATLAB compilers produce parallel code in the style of SPMD targeting the compilation of MATLAB code for parallel distributed architectures. MEGHA [Prasad et al., 2011] compiler generates CUDA C code for the acceleration of performance on GPU cores. The MAT2C [Joisha and Banerjee, 2007], MATISSE [Bispo et al., 2014] and the compilers in [Allen, 2005] and [Shei et al., 2011] generate sequential C code targeting conventional processors. None of the existing approaches produces vectorized C code in the form of SIMD intrinsics. Some of these approaches take advantage of computing parallel capabilities. However, none of them produce code that is suitable/optimized for embedded systems generating specialized instructions of the target processor/ASIP. Finally, only the Menhir and Matisse compilers support a multi-target compilation environment for the customization of the generated code, although they don't provide a description model to exploit the capabilities of the target architecture.

On the contrary, the proposed approach introduces a multi-target MATLAB-to-C compilation framework expendable and flexible to cover any target processor especially those presented in embedded systems. The proposed compiler uses a parameterized target processor model and exploits the entire instructions set of the processor. The compiler generates scalarized or vectorized C code in the form of SIMD intrinsics. Finally,

the compilation framework support the code generation of floating point and fixed point data types (which are commonly used on DSP cores).

	C code style	ISE code generation	multi -target	fixed point support
Otter [Quinn et al., 1998b]	SPMD-style C	-	-	-
RTExpress [Benincasa et al., 1998]	parallel C	-	-	-
Menhir [Chauveau and Bodin, 1999]	parallel /sequential C	-	√	-
[Allen, 2005]	sequential C	-	-	√
MAT2C [Joisha and Banerjee, 2007]	sequential C	-	-	-
MEGHA [Prasad et al., 2011]	CUDA SPMD-style	-	-	-
MATISSE [Bispo et al., 2014]	sequential C	-	√	-
[Shei et al., 2011]	sequential C	-	-	-

Table 1: Comparison of MATLAB to C compilers.

2.3 MATLAB Compilation to Hardware

MATCH [Banerjee et al., 1999], [Banerjee et al.,], [Haldar et al., 2001b] and [Haldar et al., 2001a] compiler and its commercial version, AccelDSP [Banerjee, 2003] and [Banerjee et al., 2004] automatically map MATLAB code to distributed, heterogeneous computing systems composed by field-programmable gate arrays (FPGAs), embedded processors and digital signal processors. MATCH translates the MATLAB code assigned to DSP or Embedded processors into equivalent C code while the MATLAB code mapped to FPGA's is translated in Register Transfer Level (RTL) to VHDL code. The compiler provides a set of directives to: explicitly declare the variables' type/shape, describe the available computing systems and pre-defined libraries, indicate optimizations (unroll, pipeline, map array variables to embedded RAMs) and indications to map the MATLAB code to computing components. However, MATCH is able to automatically map the MATLAB code into the available computing systems of a target architecture. To achieve this, it uses a mixed integer linear programming formulation with primary objective to optimize the performance under resources constraints or vice versa. AccelDSP converts the floating point MATLAB code to fixed point equivalent using an auto-quantization algorithm [Banerjee et al., 2003] and [Roy and Banerjee, 2004]. The algorithm computes the value range of variables using quantization of MATLAB operations. Then, it calculates the difference between the initial floating point code and the examined fixed point version. The procedure is repeated, increasing the fixed point

bit-width, until the difference of floating and fixed point values is smaller than the predefined error metric value.

In [Khoury et al., 2011] a framework is presented as extension to Octave's interpretation system for the parallel execution on Cell Broadband Engine architecture following a MIMD style. The framework exploits the Cell architecture as to the data, instruction-level, pipeline and task parallelism. It performs partitioning of the matrix instructions as sub-operations which process different amount of data and assigns (scheduling) them for processing on the available matrix execution units. The scheduler assigns at the Cell processor elements different instructions or the same operation with a different partition of data under the restrictions of data and control dependencies.

The ALMA project [Stripf et al., 2012], [Becker et al., 2012] and [Stripf et al., 2013] is a toolchain approach for the compilation of the Scilab language to reconfigurable multi-Core embedded systems. The toolchain produces automatically parallel code supporting any reconfigurable architecture. To achieve this, a Architecture Description Model (ADL) has been developed to describe the target architecture. The information is used by the compiler for parallelization tasks and the generation of target-depended code. The compiler's framework performs fine-grain and coarse-grain parallelism using the GeCoS [Floc'h et al., 2013] compiler infrastructure.

Math2Mat [Thoma et al., 2012] is a translation system of Octave/MATLAB to VHDL focusing on the efficient generation of hardware with a pipelined structure. The main goal of Math2Mat is the efficient combination of basic blocks in order to generate MATLB operations or functions of the source code. The tool is accompanied by automated verification methods using the SystemVerilog language to verify the generated design.

Table 2 presents the major features of the existing works regarding the application of MATLAB into hardware. The MATCH compiler [Banerjee et al., 1999] targets to the generation of code including embedded processors while its output may be of fixed point data types. However, the tool focus at the assignment of the source code sections for execution to the heterogeneous computing systems. The tool is not designed to leverage any hardware components special capabilities, thus any available specialized instructions or SIMD operations cannot be exploited. In [Khoury et al., 2011], a framework for the extension of the Octave's interpretation system is presented. The framework internally transforms the source code to allow parallelization in different granularities such as data parallelism (SIMD) or instruction level parallelism having main objective the generation of MIMD code. Although, the tool is a component of an interpretation system - not a standalone compiler. Moreover, its use is restricted allowing the parallel computing of Octave code only at cell broadband engine architecture. The ALMA project [Stripf et al., 2012] presents a toolchain compilation framework for the application of Scilab code to embedded Multi-core systems. The tool consists of an architecture description model for the description of the target processor including the specification of available SIMD instructions. The ALMA project and the proposed compiler present similar characteristics and the two approaches are closely related. However, there is a significant

difference between the two tools. The compilation framework of ALMA project transform the array-syntax source code to a scalarized lower level representation. Then, the intermediate representation code is vectorized using the GeCos tool included the ALMA compilation framework. The approach is more similar to that of using the MathWorks Coder in combination with an auto-vectorizing C compiler than the proposed approach which directly vectorizes the array MATLAB statements.

	code style	**hardware**	**fixed point support**	**SIMD support**
MATCH [Banerjee et al., 1999]	C/VHDL	FPGA, DSP, embedded processors	√	-
[Khoury et al., 2011]	internal shared library's functions	Cell Broadband Engine	-	√
ALMA project [Stripf et al., 2012]	binary code	embedded Multicore systems	√	√
Math2Mat [Thoma et al., 2012]	VHDL	FPGA	-	-

Table 2: Comparison of MATLAB compilation to hardware.

Several approaches have been also presented in the bibliography discussing methodology flows for the translation of MATLAB into hardware. The following studies do not address automatic implementation of MATLAB into hardware, and therefore, are briefly discussed. In [Krukowski and Kale, 1999], a methodology system is presented for the conversion of Simulink/Matlab to VHDL aiming at the prototyping of DSP applications on FPGA platforms. In [Weijers et al., 2006], a methodology flow for the manual translation of MATLAB models into real-time hardware prototype is discussed. [Bhatt and McCain, 2005], addresses the design of FPGAs using the MATLAB environment in combination with high level synthesis tools.

2.4 MathWorks Commercial Tools

MathWorks provide a suite of tools for the generation of C or HDL code from MATLAB language. MathWorks MATLAB Coder [MathWorks Coder, 2016] is the MathWorks MATLAB to C (and C++) compiler. The generated code can be used as source code, static or dynamic library in a C++ application project or it can be integrated in a MATLAB application (using MEX functions) to accelerate the performance. The compiler supports code generation of several MATLAB toolboxes, as well as a large set of MATLAB language. The tool allows the specification of the type of MATLAB input variables but it also generates code for variables with unknown shape/size on compile-time. For the latter case, MathWorks Coder produces code with dynamic memory allocation. MathWorks Coder may incorporate Fixed-Point Designer [Fixed-Point Designer, 2016] to convert the initial floating point MATLAB code to a fixed point implementation.

MathWorks Embedded Coder [MathWorks Embedded Coder, 2016] provides a framework of configuration options and optimizations. The tool can be leveraged by MathWorks Coder and Simulink Coder for the deployment of generated code on an embedded system. By using the tool, the characteristics of the target architecture can be defined such as the primitive types or byte ordering. Embedded Coder also provides the code replacement library for the description of the instructions which are provided by the hardware (or software) of the target device. The code replacement library provides a framework for the specification of customized instructions as to the MATLAB operation/function are corresponded, the instruction's type (scalar or array), types of operands and the C function prototypes. MathWorks Coder uses the code replacement library to match the MATLAB code with the available customized instructions.

Simulink Coder [Simulink Coder, 2016] and HDL Coder [HDL Coder,] are available for the deployment of MATLAB applications on FPGAs and SoCs. Simulink Coder generates C and C++ code from Simulink diagrams and MATLAB functions. HDL Coder generates synthesizable Verilog and VHDL code from Simulink models and MATLAB functions. The C or HDL output can be used by a subsequent compiler/design tool to apply the MATLAB application on an FPGA board or SoC. A variety of compilers/design tools may be used to automatically synthesize the generated code by Simulink and HDL coders on FPGA boards such as the Vivado Suite tool [Altera, 2016] and SoC Embedded Design Suite [SoC Embedded Design Suite, 2016] for XILINX FPGA boards [XILINX, 2016] and Altera FPGAs/SoCs [Altera, 2016] respectively.

The MathWorks tools present deficiencies for the compilation of MATLAB code targeting a vectorized architectures. Compiling MATLAB to C for vectorized processor architectures using the Mathworks Coder [MathWorks Coder, 2016] and an auto-vectorization compiler (i.e. LLVM) wouldn't lead to the optimal solution. The information related to vectorization which should be represented in MATLAB would be partly counteracted or even eliminated (due to internal transformations and other optimizations) during translation to scalarized C code; the latter would not allow the full exploit of the vectorized MATLAB operations by the auto-vectorizer. MathWorks Embedded Coder [MathWorks Embedded Coder, 2016] provides the user with an environment to compile MATLAB code targeting embedded systems. Embedded Coder uses an architecture description model similar to the one used by the proposed compiler for the customization of the generated C code. The major disadvantage is the lack of support for vector operations since only instructions for scalars and arrays are supported and vectorized C code cannot be generated. Even if array (instead of vector) operations are used for the customization of the generated code, code including operations with indexing cannot be efficiently compiled. Furthermore, Embedded Coder generates code storing sub-array references to an intermediate temporary array and then the specialized function (corresponding to customized instruction) is called. Such code results in most cases to a worse performance than the corresponding code generated from the Mathworks Coder [MathWorks Coder, 2016].

2.5 Auto-vectorization

Auto-vectorization is a special case of automatic parallelization which converts scalar to vector implementation. Auto-vectorization approaches are differentiated in two categories - the loop level parallelism and the superword level parallelism. The specific optimization is mostly applied by the auto-vectorizing C compiler. However, C language's characteristics incapacitate the auto-vectorization of C code. These limitations will be discussed for the entire group of research activities because they are common for all projects. Additionally, several recent evaluation studies suggest that despite the progress of auto-vectorization techniques, many existing C code cannot be vectorized by state of the art auto-vectorizing C compilers.

2.5.1 Auto-vectorizing C compilers

According to the description of GCC auto-vectorization procedure [Naishlos, 2004], Auto-vectorizers construct a strongly connected components graph to detect loop carried dependencies. In the case of identification of dependencies, the compiler optionally perform loop distribution (or any transformation such as loop interchange, scaling, skewing, reversal) to allow parallelism. If there aren't dependencies (or can be eliminated) auto-vectorizers apply a set of analyses on each loop to determine if loops are eligible to be vectorized. The analysis includes evaluation regarding issues that are discussed below (subsection 2.5.2) and further restrictions that incapacitate auto-vectorization (ex. function calls). In case that the loop specified as vectorizable, a set of code transformations are applied for the vectorization of the loop. There are two different techniques for the vectorization of a loop. The first is the unrolling of loop by the vector's width and the replacement of unrolled statements by a vector instruction. The latter is the strip-mining of the loop and the vectorization of the new loop replacing the scalar to vector statements. In the next stage, the loop bound and step are changed accordingly to the vector's width. In case of uncountable loop, additional code is inserted after the vectorized code as well as code for run-time check may be generated. The loops that include unaligned data structures, non-unit stride, or if-else statements can be vectorized using techniques which are discussed in [Eichenberger et al., 2004], [Wu et al., 2005], [Nuzman et al., 2006], [Ren et al., 2006], [Kennedy and McKinley, 1990], [Shin et al., 2005] and [Shin, 2007].

In [Larsen and Amarasinghe, 2000], the concept of Superword Level Parallelism (SLP) is introduced presenting a novel technique for the parallelization of basic blocks including isomorphic statements (execute same operation) instead of vectorizing loops. The main objective of the proposed algorithm is to detect statements of same operation that can be executed in parallel. The source operands are packed into registers for SIMD processing. Then, the isomorphic statements whose operands have been packed, are transformed to

vector instructions. The algorithm benefits from the reuse of packed operands and the intermediate packed results which can be used directly as a source in next computations.

There is a considerable interest in academic community regarding auto-vectorization approaches. The study in [Li et al., 2006], discusses the translation of binary programs written for one architecture to be executed at another architecture focusing on the efficient use of SIMD registers. In [Trifunovic et al., 2009], the impact of interactions between loop transformations and vectorization is explored using a polyhedral model for the representation of cost model. In [Nuzman et al., 2011] a split vectorization framework is presented aiming at the efficient execution of SIMD code on disparate architectures. In [Guelton et al., 2014], a vectorizer for the Python language is presented. The compilation framework uses a Python to C++ compiler (named Pythran) to generate vectorized code producing functions of a SIMD C library.

2.5.2 Limitations of Auto-vectorization

Auto-vectorizing C code is a highly complex task [Bik, 2004], [Allen and Johnson, 1988] and [Naishlos, 2004] due to language semantics and re-targetability issues [Nuzman and Henderson, 2006]. Cases that prevent or make harder the auto-vectorization of C loops are presented below:

- C applications usually include pointer variables. Multiple pointers may refer to same memory block (aliasing) preventing auto-vectorization. Most of the commercial compilers perform alias analysis to certify that references in loops do not access the same region of memory. In [Sui et al., 2016], an inter-procedural loop oriented pointer analysis is presented analyzing arrays/pointers in order to enable auto-vectorization.
- Unaligned data structures (or aligned with different size compared to the vector alignment) cannot be efficiently vectorized. Memory data must be aligned on a natural vectors size boundary to increase efficiency of data loads/stores to/from the processor. Unaligned data incur considerable overhead or even may prevent auto-vectorization. Even if the memory data are aligned, misaligned references may be presented in the code. The studies in [Eichenberger et al., 2004] and [Wu et al., 2005] discuss a compilation scheme to vectorize loops in the presence of misaligned memory generating data reorganization instructions to align data in registers.
- Loops with non-unit stride or with indirect addressing usually lead to non-contiguous memory accessing which prevent massive loading/storing of data from/to memory for SIMD processing. To achieve auto-vectorization in the presence of interleaved data without performance penalty, a) data reordering techniques are applied [Nuzman et al., 2006] and b) data permutation merging optimizations are performed [Ren et al., 2006].

- Uncountable loop bounds obstacle auto-vectorization or enlarge the generated code. The auto-vectorizer cannot evaluate the loop bound either at compile time or at run-time; subsequently aborts auto-vectorization of the loop or generates SIMD code with special epilog code. The additional code consists of loop iterations (placed at the end of a vectorized loop) that are not a multiple of the vector width.
- Nested loops with vectorizable outer-most loops must be interchanged. Usually inner-most loops cannot be vectorized [Prieto et al., 2005] or outer-most loops exhibit greater data-level parallelism and locality [Nuzman and Zaks, 2008]. However, many auto-vectorizing C compiler don't perform that loop transformation to enable auto-vectorization.
- Loops of single basic blocks are more straightforward to be vectorized than multi basic blocks such as if-then-else which require the generation of vectorized code using vector predicate statements [Kennedy and McKinley, 1990], [Shin et al., 2005] and [Shin, 2007]. Auto-vectorizers generate a vector instruction for the execution of the if-statement condition. Then they generate SIMD instructions for both if and else scopes avoiding the generation of if-else statement. Finally, they insert at the end of if-else statement a conditional SIMD instruction for merging the results of different control flow paths.
- The fact that auto-vectorizing C compilers commonly support multi platforms complicates the cost analysis algorithm to determine whether or not the examined loop is beneficial to be vectorized. The different architectures implement various vector widths and vector instructions with different acceleration (or overhead for packing/unpacking). This information usually isn't available at C compilers.

2.5.3 Auto-vectorization Evaluation

The evaluation study in [Smith, 1991] discusses the performance of auto-vectorization for the Convex and Cray Standard C compilers. The auto-vectorizing compilers are evaluated on two test suites - a collection of kernels have been developed by compiler researches and a test suite from real applications. The compilers vectorize approximately more than the half loops of test suites but different types of loops from one another. This suggests that compilers fail to vectorize all the valid for vectorization loops. Furthermore, they fail to vectorize loops including characteristics such as: break statements, pointers, induction variables, function calls and structure assignments. The study concludes that there are many areas where the auto-vectorization can be improved.

In [Ren et al., 2005], the performance of the vectorization on multimedia applications is explored. The study evaluates the performance of a multimedia benchmark whose characteristics are discussed in detail in [Ren et al., 2003]. According to the study [Ren et al., 2005], the Intel compiler v8.0 (ICC) vectorizes only 17 loops of the total 160 which are included in the benchmark. However, 23 of 34 core procedures (a subset of benchmark selected to experiment with) can be manually vectorized by re-writing the applications

code with SSE intrinsics. The manually optimized code achieves considerable speed-up up to 239%. On the contrary, the Intel vectorizer achieves similar speed-up comparing with the manually optimized counterparts only for one application while for the rest benchmark cases, the speed-up is marginal or even negative.

The evaluation in [Maleki et al., 2011] discusses the effectiveness of three vectorizing C compilers on:

- A synthetic benchmark which is used for the evaluation of auto-vectorization performance.
- A collection of real applications.
- A multimedia applications benchmark.

The study shows that the GCC, Intel C compiler (ICC) and the IBM XLC vectorizing C compilers automatically vectorize only the 46%, 70% and 54% of the synthetic benchmark respectively and only the 21%, 30% and 18% of the real applications respectively. According to this study, compilers lack accurate analysis and they don't implement transformations to enable vectorization. Despite the small amount of loops that can be automatically vectorized, in theory, a vectorizing C compiler could vectorize the 81% of the synthetic benchmark and the 48% of the collection of real applications [Maleki et al., 2011].

2.5.4 Comparison against auto-vectorizing C compilers

The subsection discusses the limitations for auto-vectorization of the code that is deriving from MathWorks tools. Auto-vectorizing C compilers vectorize the loops of C language encountering difficulties such as unaligned variables, aliasing of memory references, non-continuous memory accessing and uncountable loops. The proposed vectorization scheme provides a higher level of programming (than C) translating the MATLAB source code to vectorized code producing native vector semantics. Thus, a MATLAB application can be efficiently compiled to be executed on a vectorized architecture without invoking an auto-vectorizing C compiler and thus leading to sub-optimal results. Auto-vectorizing C compilers doesn't always vectorize all kinds of C code especially the MathWorks generated code which isn't suitable for auto-vectorization. The generated code by MathWorks Coder doesn't ensure that the derived data types are aligned to vector's natural vectors size. Furthermore, MATLAB variables which represent complex numbers, are translated to C by a structure with two variables containing the real and imaginary part. For arrays, this leads to the non-sequential storing of real and imaginary part in memory which could be loaded instantly for SIMD processing. Additionally, the generated code may include non-unit stride addressing which could add performance overhead during packing/unpacking of data. Moreover, MathWorks Coder usually fuses the generated loops of array MATLAB statements and inlines the implementation of

functions leading to loops with large scopes. Taking into account that auto-vectorizers doesn't vectorize all the operations and built-in functions, it is more likely to find such a restriction preventing the auto-vectorization of a loop with large scope. Finally, C compilers perform loop-based auto-vectorization. Thus, even if loops can be vectorized, intermediate packing/unpacking is still required between vectorized loops.

2.6 Comparison of the Proposed Compiler Against State-of-the-art

This section discusses the comparison of the proposed MATLAB compiler against other research activities. The proposed compilation framework is distinguished from the other approaches due to its multi-platform scheme supporting the generation of either scalarized or vectorized C code. Below, the differences between the proposed approach and other existing works are discussed.

Several approaches have been proposed for the compilation of MATLAB to implementation code targeting software or hardware. The compilation frameworks for production of software focus on the generation of optimized code [Joisha and Banerjee, 2007], [Bispo et al., 2015] or the generation of code which is applicable on parallel distributed architectures [Quinn et al., 1998a], [Benincasa et al., 1998] and GPUs [Prasad et al., 2011]. Furthermore, the mapping of MATLAB code to hardware translates MATLAB functions and operations to a suitable form that can be executed by heterogeneous computing systems [Banerjee et al., 1999] (FPGA, DSP), reconfigurable multi-Core embedded systems [Stripf et al., 2012] or the Cell broadband engine [Khoury et al., 2011].

The proposed approach suggests a multi-target compilation framework that take advantage of the ISE extensions or other hardware function units of any target architecture. Unlike the compilers generating SPMD-style (or MIMD [Khoury et al., 2011]) code for parallel distributed architectures [Quinn et al., 1998a], [Benincasa et al., 1998] and GPUs [Prasad et al., 2011], the proposed compiler generates SIMD code in the form of C intrinsics - suitable for embedded systems and programming of ASIPs. The proposed compiler introduces a parametrized processor model for the generation of code, which is optimized for the target architecture. Menhir [Chauveau and Bodin, 1999] compiler and ALMA project [Stripf et al., 2012] introduce description models which assist the generation of code as well. However, Menhir's target system description is proposed to form the generated code (ex. FORTRAN or C). The compilation framework of ALMA project uses an architecture description model to specify the characteristics of target architecture and define the available SIMD instructions. However, the ALMA's and the proposed compiler's description models are different. The current processor model provides a complete framework for the description of the specialized instruction set of a processor; these include scalar, SIMD or array instructions instead of ALMA's architecture description model which allows only for the specification of the available SIMD

instructions. Furthermore, the list of SIMD intrinsics is used by GeCoS [Floc'h et al., 2013] (internal component of the compilation framework) to vectorize the code. The GeCos framework operates on a low level representation following an auto-vectorization approach relevant to that of auto-vectorizing C compilers. In conclusion, the proposed compiler vectorizes directly the source code translating the MATLAB array statements to SIMD instructions, while ALMA compilation framework translates the Scilab input code into scalarized code from which auto-vectorization is performed.

The compilation of MATLAB code targeting vectorized architectures could be achieved by using the MathWorks Coder [MathWorks Coder, 2016] in combination with a auto-vectorizing C compiler such Clang/LLVM. Furthermore, the MathWorks Embedded Coder [MathWorks Embedded Coder, 2016] could be also used for the description of the available specialized instructions of the target processor/ASIP. However, the approach would lead to sub-optimal results due to the deficiencies of auto-vectorization techniques. The MATLAB code is translated to scalarized code performing several transformations/optimizations. Thus, the generated code isn't suitable for vectorization by the C compiler. The generated code may include unaligned structures, pointers (aliasing problem) and loops with complicated indexing that could prevent vectorization. Finally, the usage of Embedded Coder generating specialized instructions of the target architecture is limited only for scalar and array operations. The tool doesn't support SIMD instructions, and therefore the capabilities of vectorized architecture cannot be exploited.

Finally, the [Birkbeck et al., 2007] describes a tool for source to source transformation vectorizing the loop-based Matlab code. The proposed approach doesn't overlap with the current work which generates SIMD C code. However, the tool could be used as an extension to the current compilation framework to vectorize the MATLAB source code allowing the proposed MATLAB compiler to generate vectorized C code.

2.6.1 MATLAB to FORTRAN

This section presents existing works of the compilation of MATLAB code to FORTRAN language. There is no overlap between the proposed compiler and the research activities of MATLAB to FORTRAN compilation, thus no comparison is discussed in the section.

FALCON project [Rose et al., 1995], [De Rose and Padua, 1996], [De Rose and Padua, 1999] and [De Rose and Padua, 2003] is a MATLAB to FORTRAN compiler. The key contribution is the static type inference system and the fall-back scheme to resolve array shapes, when they have not been statically determined. The static inference mechanism extracts information about the variables types from constant values, operators, built-in functions and input files. Functions in MATLAB can be called with different type arguments. To facilitate the type inference analysis, FALCON inlines the calls to user-defined functions in the MATLAB code. Regarding the dynamic type inference, FALCON pro-

duces additional variables storing information about the variables hasn't been resolved in compile-time. Furthermore, additional code is generated for the statements involving unknown type variables to perform type checking and memory management.

CMC [Kawabata et al., 2004] is a MATLAB to FORTRAN compiler supporting programs with sparse matrix computations. The compiler provides annotations for the specification of sparse structures. Depending on the annotated MATLAB code, CMC performs inference of variable attributes based on the sparsity/density of arrays, and generates appropriate data structures and FORTRAN code. CMC also applies a set of optimizations such as loop-invariant code motion, copy propagation, common subexpression elimination, dead-code elimination and strength reduction of operations.

Mc2FOR [Li and Hendren, 2014] is an automatic translation system from MATLAB to FORTRAN. The MATLAB translator is a part of McLAB project [Hendren et al., 2011], a toolkit with a MATLAB to FORTRAN translator (McFOR), a MATLAB JIT (McVM) [Chevalier-Boisvert et al., 2010], and an aspect oriented language (AspectMatlab) [Aslam et al., 2010] for source level transformation and program profiling. Mc2FOR uses its own type analysis to infer the type shape/dimensions and range values of the MATLAB variables. The translator generates code for run-time checking of array bounds. This feature can be disabled in the generation to obtain higher performance.

2.6.2 MATLAB Just-in-time Compilation

The section presents just-in-time compilers for the MATLAB language. The just-in-time compilation approach is differentiated from the traditional compilation framework that is described in the thesis and no comparison was made about the tools against the proposed compiler.

A variety of MATLAB just-in-time (JIT) compilers [Almási and Padua, 2002], [Chevalier-Boisvert et al., 2010] and [D'Elia and Demetrescu, 2016] have been suggested in the bibliography. MaJIC [Almási and Padua, 2002] is a JIT compiler which also performs ahead of time compilation attempting to guess the most possible context will occur at run-time. During run-time, pre-compiled parts of MATLAB code that can be used are instantly executed gaining higher performance, otherwise JIT compilation is performed. In studies [Chevalier-Boisvert et al., 2010], [Casey et al., 2010], [Hendren et al., 2011], [Lameed and Hendren, 2013a], and [D'Elia and Demetrescu, 2016] two different MATLAB JIT compilers are presented which are based on the LLVM JIT framework. Additionally, several works have been presented based on the McVM JIT compiler [Chevalier-Boisvert et al., 2010] regarding the optimization of MATLAB feval function [Lameed and Hendren, 2013b], the elimination of redundant array copies [Lameed and Hendren, 2011] and the profiling of MATLAB code as to the range of loop-bounds [Aslam and Hendren, 2010].

2.6.3 Aspect oriented Approaches

The section discusses aspect oriented approaches which are used for the optimization of source code as well as for the profiling of programs. The approaches are not related to the proposed compilation framework, instead they could be used as extensions to improve the compilation framework.

Cardoso et al. have been explored aspect oriented approaches for MATLAB compilation. In [Cardoso et al., 2006], aspect oriented rules are discussed for the application of MATLAB code in DSP processors with fixed point arithmetic. The aspect rules concern the insertion of fixed point behavior in MATLAB code. The approach discusses a set of rules for customization to add in the source MATLAB code quantization statements, casting assignments, and code for run-time monitoring of fixed point arithmetic. In [Cardoso et al., 2010] domain-specific aspect language is described for the specification of transformations on MATLAB code. The approach enables program's transformations/optimizations, such as loop unrolling or function inlining. Using the aspect language, transformations are applied for specific variable types/sizes or input values. The approach can be also used for the profiling and monitoring of MATLAB applications. The overall work of aspect oriented approaches for MATLAB compilation is discussed in detail in [Cardoso et al., 2013].

[Aslam et al., 2010] and [Hendren, 2011] are discussing aspect oriented approaches for the MATLAB language. The study in [Aslam et al., 2010] presents the AspectMatlab, an aspect oriented language intended for profiling programs and monitoring of data. AspectMatlab was inspired from other aspect languages and follows the traditional design of patterns (for the specification of matching) and actions (the code to be inserted). The study in [Hendren, 2011] discusses further the aspect oriented approach for the run-time checking of the variable types against those that have been specified.

2.7 Compiling MATLAB to Other Efficient Execution Environments

The section discusses research activities with regard to the improvement of the MATLAB environment performance. The studies concern the translation of MATLAB to other languages as well as source to source transformations that optimize the execution performance. The activities are not related with the proposed compiler, therefore no comparison is discussed in the section.

[Birkbeck et al., 2007] present a tool that vectorizes scalarized MATLAB statements using an extension of Allen & Kennedy's [Allen and Kennedy, 2001] *codegen* algorithm. The *codegen* algorithm constructs a data dependence graph which is used by the loop vectorizer to examine each loop of the source code and determine if it can be vectorized. Then, the loop-based code with no loop-carried dependencies is converted to the equiv-

alent array-based one. The vectorization tool additionally provides an extensible loop pattern database with user-defined patterns for the transformation of scalarized code to more efficient equivalent vectorized code.

In [McFarlin and Chauhan, 2007], an algorithm is presented for the mapping of Octave code with functions from a target library such as BLAS. The proposed selection instruction algorithm uses empirical data to select the appropriate library's function. The data which are evaluated is the performance of function for a target architecture depending on the operands' size. The algorithm is able to map operations (to functions) with more than two operands. This is achieved by performing an iterative procedure using the already mapped operations as operands for the mapping of higher-level operations to library's functions.

Further research implementations have been presented in the bibliography for the compilation of MATLAB to more efficient execution environments. However, the following studies are not relevant to the current thesis and thus, they are briefly discussed. In [Menon and Pingali, 1999], a case study is discussed for source to source transformations of MATLAB code in order to enhance the execution performance. The transformations regard the vectorization of MATLAB for-loops, the pre-allocation of variables and the optimization of MATLAB expressions with high overhead. [DeRose et al., 1995] present another part of De Rose study which translates the MATLAB to FORTRAN code generating directives which enable parallelism. MIX10 [Kumar and Hendren, 2014] is source to source compiler translating MATLAB to X10, a language providing high performance parallel computing. Velociraptor [Garg and Hendren, 2014] is a compiler toolkit infrastructure for the development of JIT compilers targeting CPU and GPU from high level languages such as MATLAB and Python. The study in [Jahanzeb et al., 2014] presents a translator from MATLAB to Modelica, an object oriented language for system modeling. The M2M [Zhang et al., 2013] translates the MATLAB language to MapReduce, a parallel programming model for clouding computing.

Chapter 3

Compiler's Front-end

The architecture of a compiler is typically composed of a front-end, back-end and optionally a middle-layer. The division of compiler's infrastructure to separate parts aims at the implementation of different compilation tasks individually. The compiler's front-end performs scanning and parsing of the source code producing its parse tree and subsequently the Abstract syntax tree (AST).

In the beginning of the section, the subset of MATLAB language that isn't supported is listed. Then, the components of the compiler's front-end are presented. The MATLAB compiler has been developed in C++ and follows the compilers formal design. It consists of a front-end performing the parsing of MATLAB language, a middle-layer applying transformations/optimizations at AST and a back-end generating the C output code. Regarding the compiler's front-end, the compiler performs lexical and syntax analysis of the annotated MATLAB source code producing the parse tree of the input code and eventually the AST which is derived from parse tree. Typical front-end designs are well known in compiler theory therefore the syntax and lexical analysis as well as the production of parse tree and AST are discussed briefly in the current thesis. However, the novel parts namely the annotations of MATLAB code and the parametrized processor model are thoroughly discussed. Finally, the compiler's options constitute technical content and they are presented in the appendix (section 8.1).

3.1 MATLAB Input Code

The compiler's grammar complies with the MATLAB language syntax. Some of the main features which are supported by the compiler is the majority of MATLAB operations, any kind of user defined functions and function calls, basic built-in MATLAB functions (ex. zeros), all the types of control flow statements, array concatenations statements, all the MATLAB primitive types, struct variables, global variables and arrays of any shape/dimensions. Nonetheless, the compiler supports a large subset of MATLAB language but features which aren't considered relevant in embedded systems context, have not been implemented.

3.1.1 MATLAB Language Subset That Is Not Supported

The list below show the restrictions of the MATLAB compiler:

- Dynamically typed variables are not supported. The type/dimension of variables is inferred during compile time with the assistance of annotations. In many cases the compiler's user have to insert pragma functions to declare the static type and dimensions of a variable.
- The first appearance of a non-inferred variable must not include indexing (ex. *myArray(1:10,x)*).
- The compiler doesn't generate code for bounds checking. Indexing that exceeds the boundaries of an array possibly will cause a runtime error.
- The compiler doesn't perform dependence analysis and it isn't able to detect loop-carried dependencies of MATLAB array statements. For example the *res(2:10) = res(1:11)* statement would be translated incorrectly by the compiler. Thus, the user must explicitly transforms the source code using temporary variables to eliminate such data dependencies.
- Control flow statements such *If-else* and *while* with array dimension conditions aren't supported for SIMD code generation. However, *for-loops* as well as *If-else* and *while* of scalar conditions can be included in SIMD blocks. In that case, the control-flow statements are translated to scalarized code. However, this kind of statements may be included in SIMD blocks. In that case they are translated to scalarized code.
- The compiler vectorizes only statements/expressions of one dimension. Although the variables can be multi-dimensional, the operands and the result of a SIMD operation/function must be one-dimensional references. For example *res(1:10,1:32)= simdFun(op1(1:10,1:32))* is not supported for vectorized code generation. The user must explicitly insert a for-loop accessing the non-vectorized dimension: *for k=1:10 res(k,1:32)= simdFun(op1(k,1:32)); end.*
- The size of SIMD operands must be a multiple of block's SIMD width. No special epilog code is generated for the processing of the data are remaining.
- The functions that are mapped to customized instructions must contain only one output variable.
- Matrix division '/' isn't supported. Although, the parametrized processor model can be used to describe a C implementation for that operation.
- The compiler infers the fraction length of the intermediate expression's results of an assignment which includes more than one fixed point operations/functions in order to achieve accurate results. Although, it is strongly recommended (specifically for multiplication and division operations) the user to use MATLAB assignments with only one operation adjusting manually (*set_fixp* pragma) the fraction length of the result.

- Targeting BoT (and tinyBoT) the floating point constants are automatically translated to fixed point values. This isn't applied for integer constants. In order to convert integer values to fixed point use zero decimal instead. For example: *myFixp = [1.2 1.0]*.
- For high performance generated code the array variables are passed by reference in functions. This doesn't comply with the MATLAB language specification. User should manually write MATLAB code to copy the data of the array if these are changed inside a function.
- Switch statements are supported only scalar expressions.
- Structures are supported. However, array structures are not fully supported. Moreover, a field of a structure must have same type/dimensions among all the cells of an array structure.
- Type checking and type inference isn't completely supported for struct references.
- Cells are not supported because of their complexity. Supporting cells would cause inefficient generated C code.
- All the relative features of object oriented programming are not supported.
- Function handles are not supported.

3.1.2 Built-in functions

The compiler supports a small set of MATLAB built-in functions. Only essential built-in functions are supported due to engineering effort that is needed for the code generation of their implementation. However, the compiler's infrastructure is flexible and additional built-in functions may be inserted. Table 3 shows the available built-in functions by category. Math built-in functions are not supported for fixed point data types while bit-wise built-in functions are supported only for integer data types. All built-in functions are supported for complex and non-complex variables. For the math built-in functions, the compiler also generates code including the *'stdlib.h'* and *'math.h'* libraries. Therefore, if the target architecture doesn't support these libraries, the generated code cannot be executed. In that case, the user can alternatively use the parametrized processor model to overwrite the built-in functions specifying customized implementations of the built-in functions.

3.1.3 Annotations

The compiler provides a set of annotations for the declaration of variables as well as the indication of the parts of the input code (SIMD blocks) that will be translated into SIMD style. The annotations are in the form of pragma functions that comply with MATLAB syntax. The compatibility of pragmas with MATLAB syntax facilitate the development of an application since the annotated MATLAB code can be used

Type functions	Math	bit-wise	complex	Dimension	Array construction
int8	floor	bitand	complex	size	zeros
uint8	round				
int16	abs	bitor	real		
uint16	sqrt				
int32	log	bitxor		length	
uint32	log2				
int64	exp	bitcmp	imag		ones
uint64	sin			ndims	
double	cos	bitshift	conj		
char	power				
logical	atan2	bitsra	isreal	numel	rand
struct	mod				

Table 3: MATLAB built-in functions

both by the MATLAB compiler and the MathWorks tools. The MATLAB compiler is accompanied with MATLAB files including the pragmas as MATLAB functions of empty body. Thus, using the MATLAB code with pragmas in MathWorks environment, the MathWorks syntax analyzer handles pragmas as MATLAB functions which doesn't effect the execution of the code due to their empty implementation.

3.1.3.1 Annotations of Variable Decelerations

The compiler provides a set of pragmas for the declaration of program's parameters as to the data type, the shape/dimensions, as well as including indications to determine if a variable's data elements are in immediate consumable form (packed) for SIMD processing. Moreover, a set of options is provided to specify the fixed point attributes (word and fraction length) in the case of a fixed point variable and the integer class in case of an integer variable. The variable declaration pragmas start with the prefix *dec_*, continue with the data type definition such as *fixp_*, *dbl_* or *int_*, followed by the *c* if the variable is complex and they suffix with *_p* if the variable is a vector. The parameters of the pragmas consist of the variable's name and its dimensions while in the case of fixed point or integer definition extra parameters are provided for the specification of word and fraction length or the integer class respectively.

Table 4 shows the complete set of pragmas for variable declarations. Declarations among complex, non-complex or packed variables are provided for floating point, fixed point and integers types. Moreover, pragmas are available for the variable declaration of logical, character and structure types. The pragma for structure declaration doesn't provide attributes for the definition of the structure's fields and should be accompanied with the built-in *struct* function.

The first attribute of pragmas is a string constant which denotes the variable's name while the last parameter is a list of the variable's dimensions. Each dimension may be a number constant or any MATLAB expression/reference for which a constant value have been determined via constant propagation. Finally, for the declaration of scalar variables the dimension attributes may be skipped. Regarding the declaration of fixed point variables, two additional attributes are required for the specification of variable's word and fraction length which are given in the form of string literal. Similarly, for the pragmas of integer declaration an extra attribute is available for the determination of integer class. The value of that parameter can be any of the: *uint8*, *int8*, *uint16*, *int16*, *uint32*, *int32*, *uint64*, *int64* strings.

The pragmas concerning the declaration of vector variables consist of two additional attributes: the specification of SIMD width and optionally the indication of the packing dimension. The first attribute regards the number of elements which are stored in the vector unit while the second denotes the dimension of which data are packed. For instance, declaring a 16x32 matrix of packed data with packing dimension of *'2'*, the compiler assumes that the data are packed by the second dimension (by rows in MATLAB context). The packing dimension attribute isn't necessary at declaration of packed variables and it may be avoided. In this case, the default packing dimension is the last dimension of variable (for any array shape) which is the most commonly used. The attribute composed by two number constants in the form of a string literal separated by '/' character.

The example below declares a complex fixed point variable *myFixp* of *10* elements with word length of *16* and fraction length of *8*.

dec_fixpc('myFixp','16','8',1,10)

The following example declares a floating point matrix of *4x64* dimensions with packed data types. The vector's size is *4* and the data are packed by default at the second dimension.

dec_dbl_p('myVector','4',4,64)

The example below declares a fixed point array of *4x64* dimensions with packed data types. The vector's size is *8* and the data are packed at the second dimension.

dec_fixp_p('myFixpVector','16','12','8/2',10,32,10)

3.1.3.2 Other Annotations

Pragma functions are also used for the indication of SIMD blocks as well as the specification of word and fraction length of the assignment's fixed point result. The

Description	Pragmas	Attributes	Pragmas for vectors	Attributes for vectors
Fixed point	dec_fixp dec_fixpc	*'var_name'* *'word_l'* *'fraction_l'* *(dim1,dim2,...)*	dec_fixp_p dec_fixpc_p	*'var_name'* *'word_l'* *'fraction_l'* *'SIMD_width/pack_dim'* *(dim1,dim2,...)*
Floating point	dec_dbl dec_dblc	*'var_name'* *(dim1,dim2,...)*	dec_dbl_p dec_dblc_p	*'var_name'* *'SIMD_width/pack_dim'* *(dim1,dim2,...)*
Integer	dec_int dec_intc	*'var_name'* *'int_type'* *(dim1,dim2,...)*	dec_int_p dec_intc_p	*'var_name'* *'int_type'* *'SIMD_width/pack_dim'* *(dim1,dim2,...)*
Logical	dec_log	*'var_name'* *(dim1,dim2,...)*	-	- -
Character	dec_chr	*'var_name'* *(dim1,dim2,...)*	-	- -
Structure	dec_stc	*'var_name'* *(dim1,dim2,...)*	-	-

Table 4: Pragma functions for variable declaration

pragmas *startSIMD* and *endSIMD* can be used for the selection of the parts of the input code (SIMD blocks) that will be translated into SIMD style allowing the developer to select the preferred vector size of the block's SIMD operations. The parameters of the pragmas are the preferred SIMD width and the array size of the SIMD block operands. According to the code snippet below (listing 3.1), the array size of the SIMD block operands is *32* as the second parameter of *startSIMD*. Compiling the code snippet, the compiler will generate SIMD instructions of vector size *4* as this is indicated at the first attribute of the pragma.

```
1 startSIMD(4,32)
2 res = vect(1,1:32) + vect(1,33:64)
3 endSIMD()
```

Listing 3.1: Example of pragmas for definition of SIMD block.

Fixed point arithmetic demands the modification of variable's fraction (and word) length during the execution of an application in order to enhance the precision of the algorithm result. Pragma *set_fixp* allows the modification of the resulted fixed point

attributes of the next assignment's operation. The pragma consist of three attributes: the name of the variable which is effected, the new word length and the new fraction length. The example below (listing 3.2) shows the usage of pragma. Although, the *myFixp* variable has been declared with fraction length of *8*, after the execution of the fixed point multiplication the fraction length of *myFixp* variable will be *12*.

```
1 dec_fixpc('myFixp','16','8',1,10)
2 ...
3 set_fixp('myFixp','16','12')
4 myFixp = myFixp * 0.0364;
```

Listing 3.2: Example of pragmas for handling the fixed point arithmetic.

3.2 Parametrized Processor Model

The parameterized processor model (Fig. 3) describes the specialized instruction set architecture of the target processor. The model consists of a primary list with information about the custom instructions and a secondary list with types which are associated with the custom instruction operands and result. In more detail, the custom instructions description includes: a) the operation type or function name with the corresponding intrinsic C name, b) the instruction's type which can be scalar type, array type or an

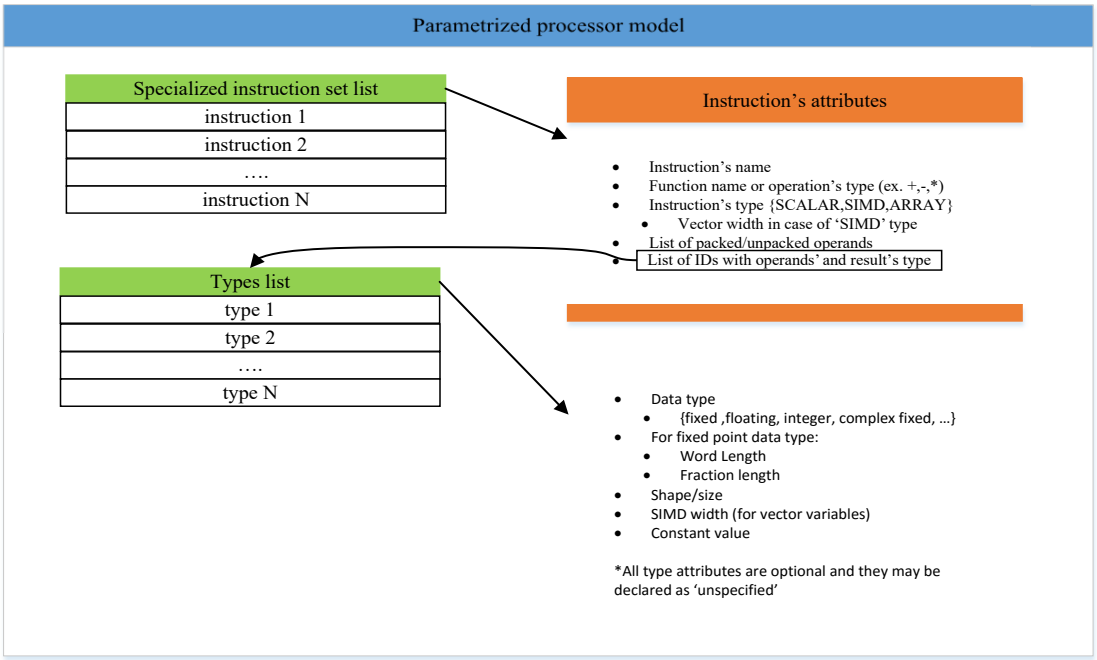

Figure 3: Parametrized processor model.

integer that represents the vector width in the case of SIMD instruction's type, c) a boolean value for each operand indicating which operands are required to be in packed format ready for SIMD processing and d) an ID (type name) for each operand and the result (this corresponds to a name of a type in the secondary list of types). The secondary list includes a) the shape/size, b) the SIMD width (in case of a vector type) c) the data type which can be any of those supported and d) the constant value(s) of operand. In case of fixed point type the word and fraction length may be defined as well. The parametrized processor model allows declaring any of the type attributes as unspecified in case the specific attribute is not required. Unspecified attributes for the result allow the dependent on the instruction type, inference of function calls as described below. On the other hand, unspecified attributes for operands enable an abstract and flexible description of instructions where instruction selection mechanism can match custom instructions with operations or function calls regardless of the unspecified attributes. This technique aims at having only one custom instruction required for different operand types of the same operation or function. For instance, all the element-wise operations can be described leaving unspecified the shape/size of operands since one appropriate custom instruction could be matched for any shape/size. One more case which demonstrates the importance of the unspecified attributes is the custom instructions on the fixed point type which can be applied without any constraint of fraction or word length (ex. addition, subtraction).

An example of a custom instruction is shown in figure 4. The record shown in the parametrized processor list is a vector addition of SIMD width 4. Additionally, the instruction requires that the two operands should be in packed form while the types of the operands and the result are specified by the *t_fi* type. The *t_fi* type denotes a complex fixed point vector of SIMD width 4 with any word and fraction length, any shape/size and unspecified constant values.

Different implementations of a MATLAB function or operation in hardware are supported. For instance, an architecture may support different instructions for the multiplication of complex operands and for the multiplication of a complex with a non-complex operand. In such cases, the parametrized processor model supports multiple entries for a function or an operation corresponding to different custom instructions – hardware implementations. In case that a function or operation of the MATLAB source code could be matched with more than one custom instructions in the parametrized processor list, the first in the list will be selected in the instruction selection phase. Thus, the processor model allows the user to choose the priority in which customized instructions will be selected.

The parametrized processor model is described using XML given as input to the compiler. The developer describes the available customized instructions using XML and defines information related to the target processor such as the native vector types in C or the packing and unpacking instructions (for each supported vector type). The XML language provides an appropriate and sufficient structure for the specification of the cus-

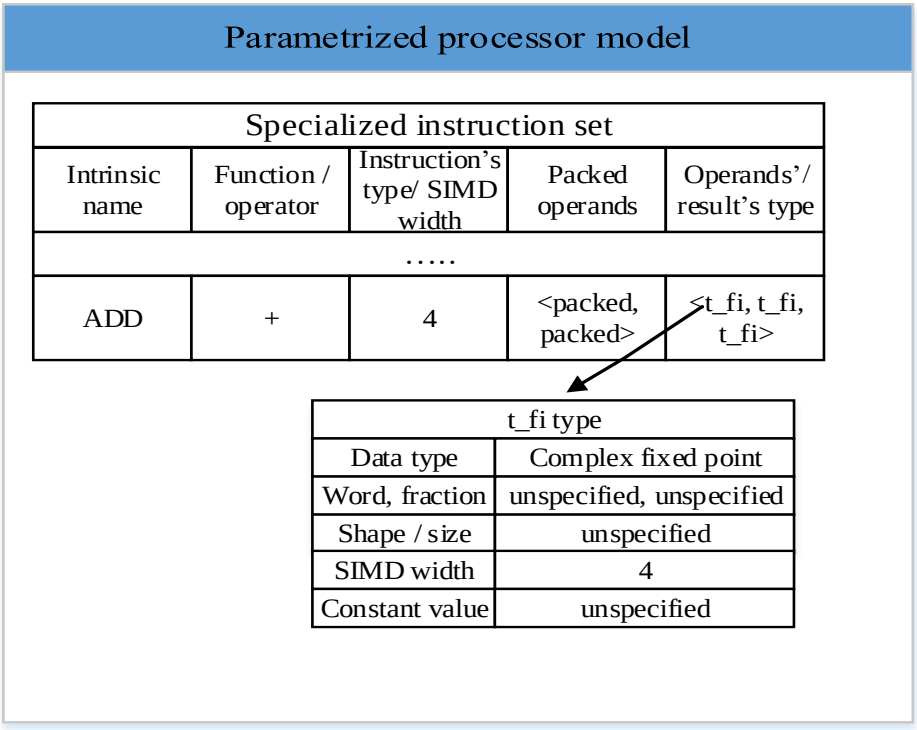

Figure 4: Example of Parametrized processor model.

tom instructions facilitating the compiler to extract only the relevant information. On the contrary, if a processor/architecture description language was used, a new parser would be required to extract only the relevant information (in some cases that could be impossible due to information absence), and more effort would be required by the compiler's user to design the instruction space (as to the binary encoding of instructions). Furthermore, processor/architecture description languages are used internally from specific tools (e.g Synopsys Processor Designer [ASIP Designer, 2016] uses nML Processor Modeling Language) limiting the compiler to be used in combination with other C/C++ compilers.

3.2.1 Description of Customized Instructions in XML

The parametrized processor model is described using the XML language. The description of the architecture primarily includes the specification of the processor's customized instructions and their operands/result types. Additional information about the target architecture is described in the model as well. The information concerns auxiliary semantics such as the name of derived C data types, additional C libraries and C header files to be included and other special operations such as shift instructions which are used in fixed point arithmetic, packing/unpacking operations and instructions to read/store

the real or imaginary part of complex variables. Such information is depended on the target architecture and is required for the portability of the generated code to the various architectures. The user can encapsulate the code related to architecture at header C files and describes the semantics/functions of the header files in the parametrized processor model. Then, the compiler is able to use that information to produce code which is applied on the target architecture. Thus, the schema allows the compatibility of the compiler to any target architecture. Below a detailed description of the parametrized processor model regarding the XML specification is given. The section 8.3 in appendix presents XML description examples of the target architectures.

3.2.1.1 Description of Operands'/Parameters' Types

The parametrized processor model allows the description of the types which are associated with the custom instruction operands and result. Table 5 shows the attributes which are specified for a type. The *id* attribute is the type's name which is used at custom instructions specification to identify an associated type. The *dt* attribute denotes the data type and can be any of these are shown in the table. Specifying fixed point data types, word and fraction length may be also declared with number constants in parenthesis. If word and fraction length aren't declared, they are considered as unspecified attributes. In the case that only word (or fraction) length is required to be specified the other attribute can be declared as unspecified denoting with *'any'* string literal. Attributes *dim* and *complex* represent the dimensions of the type and whether or not the type is complex. The *dim* is a list of integers separated by comma or space while the *complex* is the *'true'* or *'false'* value. Constant values may be also specified for a type. The specification of constant values implies that the instruction's operand of this type is a constant number or any expression/reference for which a constant value have been determined via constant propagation. Constant values (*const*) are specified using MATLAB syntax with a constant number or an array concatenation expression. The attributes of a type may be declared with XML tags (*t1* in listing 3.3) or XML elements (*t2* in listing 3.3) while missing attributes are considered as unspecified.

Listing 3.3 shows two examples of type specification (*t1* and *t2*). The first example specifies a complex fixed point type with fraction length of *12*. The word length of type as well as its dimensions are missing and they are considered as unspecified attributes. Finally, the last line of listing declares a square matrix (2x2) of complex fixed point type (with any word/fraction length) and specific constant values.

```
<type id="t1">
   <dt>fixp(any,12)</dt>
   <complex>true</complex>
</type>
<type id="t2" dt="fixp" dim="2" complex="true" const="[1+1i 1; 2i 2]"> </type>
```

Listing 3.3: Example of types XML specification.

Attributes	Specific values	Description
id	-	The name of type
dt	*fixp(WL,FL)*, *double*, *int8*, *int16*, *int32*, *int64*, *uint8*, *uint16*, *uint32*, *uint64*	In case of *fixp* the word *WL* and fraction *FL* lengths are specified, too
dim	-	A list of dimensions
complex	*true*, *false*	Specify if the type is complex
const	-	Specify any constant values

Table 5: XML specification of operands/result type.

3.2.1.2 Description of Customized Instructions

Table 6 presents the customized instructions specification. The *name* attribute is the name of the generated intrinsic and the *type* attribute represents the type of instruction among: *scalar*, *SIMD* and *array* types. For the specification of a customized instructions which are mapped with MATLAB functions, the *func* attribute is assigned with a MATLAB function's name while the customized instructions implementing MATLAB operations are set with the appropriate operator at *op* attribute. The *pack* list is used to determine the intrinsic's operands which must be in packed format. According to that argument, the compiler generates additional code to pack the unpacked operands. The *SIMD_width* is used to specify the intrinsic's SIMD width and must be the same as that it has been defined in *startSIMD* pragma for the matching of instruction with a MATLAB function/operation included in a SIMD block. Customized instructions which operate on any vector's width, are set as *true* instead of determining the exact SIMD width. The *op_types* defines the operands' types of customized instruction. The attribute consists of a list separated by comma with the name of the types which have been also declared in the XML file. For an operand more than one types can be specified. At this case the multiple types are included in parenthesis and they are separated by the character '|'. Regarding the customized instructions mapped with MATLAB functions, the additional *res_type* attribute is specified. The attribute is used in the type inference stage to infer the type of the function's call result as it is discussed in subsection 4.2.2. Skipping the declaration of the result's type, the MATLAB function call fully inherits the type of its first parameter.

Attributes	Specific values	Description
name	-	Intrinsic name
type	scalar, SIMD, array	Intrinsic type
func	-	Function name
op	$+$, $-$, $*$, $.*$, $/$, $\backslash$, $./$, $.\backslash$, $\hat{}$, $.\hat{}$, $:$, $<$, $<=$, $>$, $>=$, $==$, $!=$, $\&$, $\vert$, $\&\&$, $\vert\vert$, $'$, $.'$	Operator
pack	*p*, *u*	List of packed/unpacked operands
SIMD_width	*false*, *true*, *any number*	SIMD width
op_types	-	List of operand types
res_type	-	Results type

Table 6: XML specification of customized instructions.

```
<type id="t1" dt="int32" complex="false"></type>
<type id="t2" dt="fixp(any,8)"  complex="false"></type>
<type id="t3" dt="fixp(any,12)" complex="false"></type>
<instruction>
  <name>SUB</name>
  <type>SCALAR</type>
  <op>-</op>
  <op_types>t1,t1</op_types>
</instruction>
<instruction name="RSH_v4" type="SIMD" func="bitsra" pack="p,u" SIMD_block ="4"
op_types="(t2|t3),t1"></instruction>
```

Listing 3.4: Example of customized instructions XML specification

Listing 3.4 presents the specification of two customized instructions. The first one concerns the definition of the intrinsic *SUB* which implements the subtraction of two scalar 32-bit integers. The second instruction describes the hardware implementation, named *RSH_v4*, of the *bitsra* MATLAB function for a non-complex fixed point vector with SIMD width 4. For the purposes of the example it is assumed that the instruction operates with fixed points of fraction length *8* and *12* which are expressed by the types *t2* and *t3* respectively. The function consists of two parameters (the sifted vector and the shifting value) requiring the first one to be in packed form and the second one to be unpacked. Finally, no result type is needed for the customized instruction because result's type fully depended on the first parameter's type.

The listing 3.6 shows the generated C code of the MATLAB code in listing 3.5 using the parametrized processor model described in listing 3.4. In the MATLAB code a scalar integer and an unpacked fixed point matrix are defined. Moreover, the MATLAB code includes a subtraction between scalar integers and a function call of *bitrsa* enclosed in SIMD block with vector width of 4. The compiler matches the MATLAB operation and function with the available instructions of the parametrized processor

model as this is shown in listing 3.6. For the generation of vector intrinsic, additional code is produced to pack only the first operand's data. Similarly, additional code is generated to unpack the packed result of *RHS_v4* instruction. Finally, declaring *op1* and *res* as packed variables, the packing/unpacking operations wouldn't be generated.

```
function [res] = cust_instr_ex(op1, myInt)
  dec_fixp('op1','32','8',1,32);
  dec_int('myInt','int32');
  myInt = 32-myInt; %matching with SUB
  startSIMD(4,32);
  res = bitsra(op1, myInt); %matching with RSH_v4
  endSIMD();
end
```

Listing 3.5: Example of MATLAB code with matching operation and function.

```
void cust_instr_ex_1(_fixp_t res[32], _fixp_t op1[32],D_SINT32 myInt){
  fixp32x4_t vec_0, vec_1;
  D_SINT32 i2, i1;
  myInt = SUB(32,myInt); //matched operation
  D_SINT32 si0;
  for(si0=0; si0 < 32; si0=si0+4){
    for(i2=si0; i2 < si0+4; i2=i2+1){
      vec_0 = insert_i_4(vec_0, i2-si0, op1[i2]);
    }
    vec_1 = RSH_v4(vec_0, myInt); //matched function
    for(i1=si0; i1 < si0+4; i1=i1+1){
      res[i1] = extract_i_4(i1-si0, vec_1);
    }
  }
}
```

Listing 3.6: Generated C code with matched operation and function.

3.2.1.3 Description of Derived C Types

The parametrized processor model allows the declaration of the generated C data types for an architecture. The derived types may vary among data types (and different fraction or word length), dimensions, complex/non-complex type, or different SIMD widths. For their description (Table 7), an ID of a specified type is assigned in the *type* attribute while the SIMD width can be also defined in case of packed variable (vector) declaration. Finally, the *name* attribute is set with name of the derived C type.

During code generation, the compiler produces C variable declarations using the derived C data types which have been matched with the types of variables in the parametrized processor model. However, for the non matched variables, default C type names are produced.

Attributes	Specific values	Description
name	-	C data type
type	-	The name of a specified types
SIMD_width	-	SIMD width for vectors

Table 7: XML specification of derived C types.

```
<type id="t1" dt="fixp" complex="true"></type>
<derived_type name="cfixp_t" type="t1"></derived_type>
<derived_type name="cfixp32x4_t" type="t1" SIMD_width="4"></derived_type>
```

Listing 3.7: Example of derived C types XML specification

Listing 3.7 shows an example of two derived type declarations. Both of them use the *t1* type which is complex fixed point with the difference that the second declaration (*cfixp32x4_t*) refers to a vector definition with SIMD width 4.

Listing 3.9 shows the generated C code of MATLAB code in listing 3.8 using the snippet of parametrized processor model in listing 3.7. The first variable in MATLAB code is a complex fixed point while the second variable is a vector with SIMD width 4 of complex fixed point type. The compiler generates declaration statements for the two variables which are declared in MATLAB code using the names of the derived types in the XML specification.

```
function [] = der_types_ex()
  dec_fixpc('var1','32','8',1,32);
  dec_fixpc_p('var2','32','8','4',1,32);
end
```

Listing 3.8: Example of MATLAB code with variable declarations

```
void der_types_ex(){
  cfixp_t var1[32];
  cfixp32x4_t var2[8];
}
```

Listing 3.9: Generated C code with derived C types

3.2.1.4 Description of fixed point semantics

The fixed point arithmetic requires shifting operations to adjust the fraction length of operands and result. Moreover, the majority of processors doesn't provide any instruction for the division of integers (fixed points are based in integers). Although, most of

them provide instructions for the calculation of the reciprocal of a number which can be used as follows: a division is replaced by a multiplication of the dividend and the reciprocal of the divisor.

The parametrized processor model allows the description of such operations concerning the fixed point arithmetic. Table 8 shows the attributes used for the description of the right and left shift operations. Attribute *op_types* is used to specify the operand's type of shift operation and *type* attribute denotes the intrinsic's type concerning scalar or SIMD processing. The *SIMD_width* attribute is used in case of SIMD shift operation to define the vector's width. Table 9 presents the description of the reciprocal operation. The description composed by the name of the instruction and attributes concerning the operation's type as those in the shift operation description.

Reciprocal instructions are generated only targeting BoT and tinyBoT ASIPs (using option *-arch*). Furthermore, the compiler doesn't automatically transform division to multiplication using the reciprocal of the divisor. To achieve that, customized instructions of multiplication intrinsics must be included in the parametrized processor model for the description of the division operators.

Attributes	**Specific values**	**Description**
leftShift	-	Left shift instruction name
rightShift	-	Right shift instruction name
type	scalar, SIMD	Intrinsic type
SIMD_width	-	SIMD width for vectors
op_types	-	Operand type

Table 8: XML specification of shift operations.

Attributes	**Specific values**	**Description**
recip	-	Reciprocal instruction name
type	scalar, SIMD	Intrinsic type
SIMD_width	-	SIMD width for vectors
op_types	-	Operand type

Table 9: XML specification of reciprocal operations.

Listing 3.10 presents specification examples of the shift and reciprocal operations. The examples include the description of scalar and SIMD shift/reciprocal instructions operating on fixed point data type.

```
<type id="t1" dt="fixp" complex="false"></type>
<shift leftShift="LSH" rightShift="RSH" type="SCALAR" op_types="t1"></shift>
<shift leftShift="V4_LSH" rightShift="V4_RSH" type="SIMD" SIMD_block ="4" op_types="t1"></shift>
<reciprocal recip="RECIP" type="SCALAR" op_types="t1"></reciprocal>
<reciprocal recip="V4_RECIP" type="SIMD" SIMD_block ="4" op_types="t1"></reciprocal>
```

Listing 3.10: Example of shift and reciprocal XML specification.

Listing 3.12 shows the generated code of MATLAB code in listing 3.11 using the shift and reciprocal operations of listing 3.10. The compiler generates the scalar and vector customized instructions which have been specified for the *'./'* operator in the parametrized processor model and produces additional code to calculate the divisor's reciprocal (*RECIP* and *v4_RECIP*). The fraction length of the fixed point multiplication's result equals to the sum of the fraction length of operands. Thus, the compiler generates right shift instructions to adjust the fraction length of the multiplication's result from *16* to *8*.

```
function [scRes, vecRes] = division_ex(sc1,sc2, vec1,vec2)
  dec_fixp('sc1','32','8');
  dec_fixp('sc2','32','8');
  dec_fixp_p('vec1','32','8','4',1,4);
  dec_fixp_p('vec2','32','8','4',1,4);
  scRes = sc1 ./ sc2;
  startSIMD(4,4)
  vecRes = vec1 ./ vec2;
  endSIMD();
end
```

Listing 3.11: Example of MATLAB code with shift and reciprocal operations.

```
void division_ex_1(_fixp_t &scRes,fixp32x4_t vecRes, _fixp_t sc1,_fixp_t sc2,fixp32x4_t vec1,
    fixp32x4_t vec2){
  scRes=RSH(mul_fxp(sc1,RECIP(sc2)),8);
  vecRes=V4_RSH(v4_mul_fxp(vec1,V4_RECIP(vec2)),8);
}
```

Listing 3.12: Generated C code with shift and reciprocal operations.

3.2.1.5 Description of packing/unpacking operations

The parametrized processor model allows the specification of packing and unpacking operations of different data types and SIMD width. Table 10 presents the attributes of operations while tables 11 and 12 show the different packing and unpacking operations are provided by the parametrized processor model. The *insert* and *extract* operations are used in the vectorized generation code to pack/unpack the SIMD instruction's operands

and result to/from vectors. The *insert_ptr* and *extract_ptr* operations are used for the same purpose for packing/unpacking a memory segment (by using a pointer) in case that data stored in continuous memory locations. The rest operations are used in the scalarized generation code to insert or extract a scalar to/from a vector. The different packing/unpacking operations concern the various shapes and packing dimensions of packed array variables. Finally, the *expand* operation is used for the expanding of a scalar to a vector.

Attributes	**Specific Values**	**Description**
SIMD_width	*false*, *true*, *any number*	SIMD width
op_types	-	List of operand types

Table 10: XML specification of packing/unpacking attributes.

Architectures with SIMD processing provide only basic instructions of packing/unpacking operations. Most of the packing/unpacking operations included in a parametrized model cannot be directly mapped to a hardware intrinsic. However, the compiler's user can write his own implementation (C function prototypes are included in listing 3.14) of the specified packing/unpacking operations and include them in the generated code (including additional code in the compilation of generated code is described at 3.2.1.6). Listing 3.13 shows the description of packing and unpacking operations for fixed point vectors of SIMD width 4. The prototypes of C functions corresponding to the packing/unpacking operations of the parametrized processor example are shown in the listing 3.14.

Pack Operations	**Description**
insert	Insert element to vector given the offset of an array of vectors
insert_ptr	Insert massively elements to vector
insertToVector	Similar to *insert*. It is used at scalar processing with vectors
insertToVector_RowsPack	Pack in rows at scalar processing with vectors
insertToVector_firstDimPack	Pack in the 1st dimension of 3D array at scalar processing
insertToVector_SecondDimPack	Pack in the 2st dimension of 3D array at scalar processing
expand	Expand a scalar to vector

Table 11: Packing operations.

Unpack operations	Description
extract	Extract element from vector given the offset of an array of vectors
extract_ptr	Extract massively elements from vector
extractFromVector	Similar to *extract*. It is used at scalar processing with vectors
extractFromVector_RowsPack	Unpack from rows at scalar processing with vectors
extractFromVector_firstDimPack	Unpack from the 1st dimension of 3D array at scalar processing
extractFromVector_SecondDimPack	Unpack from the 2st dimension of 3D array at scalar processing

Table 12: Unpacking operations.

```
<type id="t1" dt="fixp(any,any)" complex="false"></type>
<derived_type name="fixp_t" type="t1" ></derived_type>
<derived_type name="fixp32x4_t" type="t1" SIMD_width="4"></derived_type>
<pack op_types="t1" SIMD_width ="4"  insert="insert_i_4" insert_ptr="insert_all_ptr_i_4"
     insertToVector="insert_v_i4" insertToVector_RowsPack="insert_v_r_i4"
     insertToVector_firstDimPack="insert_v_3d_1_i4" insertToVector_SecondDimPack="insert_v_3d_2_i4"
      expand="insert_all_i_4"></pack>
<unpack op_types="t1" SIMD_width ="4">
  <extract>extract_i_4</extract>
  <extract_ptr>extract_all_ptr_i_4</extract_ptr>
  <extractFromVector>extract_v_i4</extractFromVector>
  <extractFromVector_RowsPack>extract_v_r_i4</extractFromVector_RowsPack>
  <extractFromVector_firstDimPack>extract_v_3d_1_i4</extractFromVector_firstDimPack>
  <extractFromVector_SecondDimPack>extract_v_3d_2_i4</extractFromVector_SecondDimPack>
</unpack>
```

Listing 3.13: Example of packing/unpacking XML specification

```
inline fixp32x4_t insert_i_4(fixp32x4_t vect, int offset, fixp_t val);
inline fixp32x4_t insert_all_ptr_i_4(fixp32x4_t res, fixp_t *val);
inline void insert_v_i4(fixp32x4_t *res, int offset, fixp_t x, int SIMDW);
inline void insert_v_r_i4(fixp32x4_t *res, int offset, fixp_t x, int rowD, int colD, int SIMDW);
inline void insert_v_3d_1_i4(fixp32x4_t *res, int offset, fixp_t x, int dim1, int dim2, int dim3,
     int SIMDW);
inline void insert_v_3d_2_i4(fixp32x4_t *res, int offset, fixp_t x, int dim1, int dim2, int dim3,
     int SIMDW);
inline fixp32x4_t insert_all_i_4(fixp_t val);

inline fixp_t extract_i_4(int offset, fixp32x4_t vect);
inline void _extract_all_ptr_i_4(fixp_t *val, fixp32x4_t vect);
inline fixp_t extract_v_i4(int offset, fixp32x4_t *x, int SIMDW);
inline fixp_t extract_v_r_i4(int offset, fixp32x4_t *x, int rowD, int colD, int SIMDW);
inline fixp_t extract_v_3d_1_i4(int offset, fixp32x4_t *x, int dim1, int dim2, int dim3, int SIMDW);
inline fixp_t extract_v_3d_2_i4(int offset, fixp32x4_t *x, int dim1, int dim2, int dim3, int SIMDW);
```

Listing 3.14: Function C prototypes of packing/unpacking operations

Listing 3.16 shows examples of producing packing and unpacking operations for scalarized code generation compiling the MATLAB code of listing 3.15. The MATLAB code includes pragmas for the declaration of a scalar variable and the declarations of packed arrays of different shapes and packing dimensions. Then, there are scalar assignments inserting and extracting an element from/to a vector. In the generated code (listing 3.16) different packing/unpacking operations are produced depended on the shape and the packing dimension of the packed variables. For the assignment of line 6, the instruction assigned as *extractFromVector* (listing 3.13) is produced because the *vec* variable has been declared with default packing dimension, while for the assignment in line 7, it is produced the specified as *insertToVector_RowsPack* instruction. Finally, the instruction assigned as *insertToVector_SecondDimPack* packing operation is produced for the line 8 of MATLAB code because the second dimension of *vec3d* has been declared as packing dimension.

```
1 function [] = pack_unpack_scalar_ex()
2   dec_fixp('sc','32','8');
3   dec_fixp_p('vec','32','8','4',32,2);
4   dec_fixp_p('vecR','32','8','4/2',2,32);
5   dec_fixp_p('vec3d','32','8','4/2',2,32,6);
6   sc = vec(4,2);
7   vecR(2,4) = sc;
8   vec3d(2,8,6) = sc;
9 end
```

Listing 3.15: Example of MATLAB code with packing/unpacking operations at scalarized code generation.

```
1 void pack_unpack_scalar_ex_1(){
2   fixp_t sc;
3   fixp32x4_t vec[16];
4   fixp32x4_t vec3d[96];
5   fixp32x4_t vecR[16];
6   sc = extract_v_i4(4-1+(2-1)*32, vec, 4);
7   insert_v_r_i4(vecR, 2-1+(4-1)*2, sc, 32,2,4);
8   insert_v_3d_2_i4(vec3d, 2-1+(8-1)*2+(6-1)*64, sc, 6,32,2,4);
9 }
```

Listing 3.16: Generated C code with packing/unpacking operations at scalarized code generation.

Listing 3.18 presents an example producing packing/unpacking operations for vectorized code generation. For the example, the MATLAB code of listing 3.17 has been used which includes a declaration of unpacked matrix and an addition inside SIMD block scope. In the C code, the compiler produces additional code to pack the unpacked operands and unpack the packed result generating the instructions has been specified in the parametrized processor model of listing 3.13. The reference *var(1:8,1)* accesses

data sequentially, therefore the instruction specified as *insert_ptr* is produced while for the reference *var(1:2:16,1)* (non-sequential data accessing), the instruction assigned to *insert* packing operation is generated.

```
function [] = pack_unpack_simd_ex()
  dec_fixp('var','32','8',16,1);
  startSIMD(4,8)
  vecRes = var(1:2:16,1) + var(1:8,1);
  endSIMD();
end
```

Listing 3.17: Example of MATLAB code with packing/unpacking operations at vectorized code generation.

```
void pack_unpack_simd_ex_1(){
  fixp32x4_t vec_0;
  fixp32x4_t vec_1;
  fixp32x4_t vec_2;
  fixp_t var[16];
  fixp_t vecRes[8];
  D_SINT32 i2;
  D_SINT32 si0;
  for(si0=0; si0 < 8; si0=si0+4){
    for(i2=(si0*2 + 1); i2 < (si0*2 + 1)+8; i2=i2+2){
      vec_0 = insert_i_4(vec_0, (i2-(si0*2 + 1))/2, var[i2-1]);
    }
    vec_1 = insert_all_ptr_i_4(vec_1, &((var)[16+si0]));
    vec_2=v4_add_i_rr(vec_0,vec_1);
    _extract_all_ptr_i_4(&((vecRes)[si0]), vec_2);
  }
}
```

Listing 3.18: Generated C code with packing/unpacking operations at vectorized code generation.

3.2.1.6 Description of Other Auxiliary Semantics - Operations

The parametrized processor model allows the description of auxiliary semantics and operations such as accessing real or imaginary part of a complex variable instructions, specification of vector assignments and names of header C files which are included in the generated code.

Listing 3.19 shows the auxiliary semantics supported by parametrized processor model. The XML tag *header_files* is used to define the header files to be included in the generated code. For each header file specification, an *'#include'* statement with the name of file is inserted in the generated code. The XML tag *setConstant* defines the instruc-

tions used by the compiler to generate assignments of a number constant to the real and imaginary part (or both parts) of a complex variable. Similarly, the *getPart* tag specifies operations to get the real or imaginary part of a variable. The operation may be used on unpacked variables (*scalar* type) or packed variables (*SIMD* type) for different data types while using a SIMD operation of this purpose outside SIMD block, the *SIMD_width* attribute must be set to zero. The *vectorAssign* tag is used to specify instructions of assignments between vectors. For the most of the architectures, simple statements such as *vec1 = vec2* assign the content of a vector to the other one. Nonetheless, the parametrized processor model provides the specification of these instructions for reasons of applicability of the compiler to any architecture. Finally, definition of print operations is supported by the parametrized processor model among scalar, array and packed array variables of different data types.

```
<type id="t1" dt="fixp(any,any)" complex="false"></type>
<header_files type="types.h">
  <common>common.h</common>
</header_files>
<setConstant setReal="SET_RE" setImag="SET_IM" setReIm="SET_RE_IM" op_types="t1"></setConstant>
<getPart getReal="getRe" getImag="getIm" op_types="t1" type="SCALAR" ></getPart>
<getPart getReal="getRe_v" getImag="getIm_v" op_types="t1" type="SIMD" SIMD_width ="0"></getPart>
<vectorAssign assign="_copy_vector" SIMD_width ="true" op_types="t1"></vectorAssign>
<print printScalar="printFXPc" printArray="printFXPAc" printPackedArray="printFXPVc" op_types="t1">
    </print>
```

Listing 3.19: Example of auxilary semantics XML specification

3.3 Front-end

The compiler's front-end performs syntax analysis using a lexical and syntax analyzer that have been generated by the flex/bison tools [Levine, 2009] for MATLAB grammar. No simplifying assumptions about the input syntax have been made. The compiler scans the input MATLAB code using the generated by Flex lexical analyzer which feeds with tokens the parser. At the syntax analysis stage, the parse syntax tree is constructed consisting of nodes of terminal and non-terminal symbols that have been determined in the grammar. The tree's nodes are connected according to the production rules of grammar. More specifically, a node representing a non-terminal symbol is constituted by descendants which are the derived symbols of the corresponding syntax rule.

In the next stage, the parse tree is transformed to an Abstract Syntax Tree. The parse tree is composed by nodes for each symbol of grammar. Most of these nodes are redundant incapacitating any tree traversals. During the generation of the AST only nodes including useful information about the input program are constructed. The AST facilitates any traversals for transformations/optimization and code generation.

The parse tree is represented by 122 classes each of them referring to a different symbol of MATLAB grammar. Figure 5 shows a part of UML class diagram regarding the parse tree representation. Each grammar symbol is a derived class of *CParseTree_SyntaxElement* base class. The *CParseTree_translation_unit* class represents the root of parse tree. Class *CParseTree_TOKEN* represents terminal symbols and the rest derived classes are referred to non-terminal grammar symbols. Each derived class includes a specific number of class constructor overloads corresponding to the syntax production rules of the symbol. The base class includes member variables about the descendants of the node, its symbol type (terminal or non-terminal), a symbol enumeration and the location of the token at input file (line and column). Methods which traverse the parse tree (ex. *ParseTreeToASTGenerationPass*) are polymorphic (*virtual* functions). Through polymorphism there are different implementations of the same traversal function among the derived classes allowing the specialization of actions at parse tree nodes of different symbols.

3.4 Conclusion

In this chapter the front-end of the MATLAB compiler was presented. The compiler's front-end parses the source MATLAB code producing the parse tree which is subsequent transformed to AST. Another part of the front-end is the compiler's input specification which is the annotated MATLAB code and the processor model in XML. The annotations in source code are used for the declaration of the type of MATLAB variables as well as the definition of the blocks that are eligible for SIMD code generation specifying the preferred vector size. The parameterized processor model allows the generation of code that exploits the capabilities of the current architecture. Through the use of processor model, the specialized instruction set and native semantics of an architecture are described which are leveraged by the compiler to generate C code for the specific target processor.

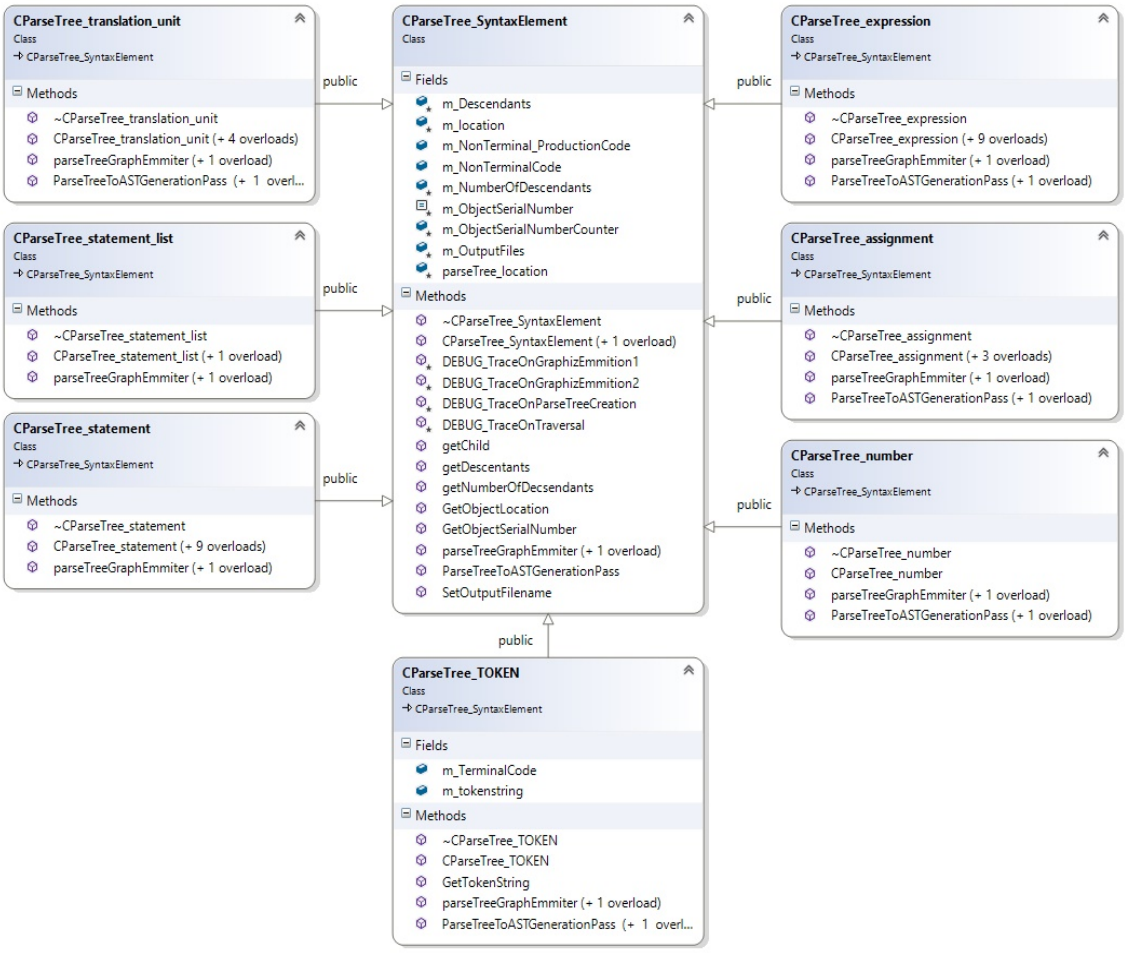

Figure 5: UML diagram of classes representing parse tree.

Chapter 4

Compiler's Middle-layer

The middle-layer is the part of the compiler's infrastructure that converts the intermediate representation to an appropriate form for code generation. Furthermore, dependence analysis and program optimizations are performed by the compiler to improve the performance of the code that is generated.

The middle-layer of the MATLAB compiler initially performs type inference analysis on AST representation to resolve the type of MATLAB variables. In this stage, the parametrized processor model is invoked to determine the result's type of MATLAB function calls that later are mapped with customized instructions. Subsequently, instruction selection is applied to match the MATLAB operations and functions with the customized instructions of the target architecture. The instruction selection algorithm utilizes the parametrized processor model for the mapping process. In the next step of the compiler's middle-layer, a multi-pass stage is applied to transform the AST for data parallel execution. Firstly, complex MATLAB array expressions are decomposed to simpler ones retaining the vectorized form of the SIMD operands. Afterwards, new statements are introduced in the AST for the packing of the unpacked data which are involved in SIMD instructions. Similarly, unpacking statements are inserted to transfer the packed result's values to the unpacked data structure. Finally, data flow analysis is performed and packing/unpacking elimination follows to remove the redundant intermediate packing/un-packing statements. The majority of the middle-layer stages including the type inference, instruction selection and introduction of packing/unpacking statements constitute the main contributions of the thesis.

4.1 Abstract Syntax Tree Representation

The AST is composed by classes that represent the syntax elements of MATLAB language. Figure 6 presents a part of UML class diagram regarding the AST representation. The *AST_SyntaxElement* is the base AST class which includes member variables such as the list of node's descendants. The AST classes that represent syntax elements are divided to a) statements such assignments and control flow statements using the *AST_StatementElement* as base class and b) syntax elements composing MATLAB expressions which derived from *AST_ExpressionElement* class. The classes of AST which represent the MATLAB syntax elements (ex. *AST_REFERENCE*) may be consti-

tuted by other derived classes creating a multilevel of inheritance in the class hierarchy model. For instance, the *AST_EXPRESSION* class represents a MATLAB expression node at AST. The class is base of classes such as the *AST_EXPRESSION_MUL* and *AST_EXPRESSION_TRANSPOSE* which express the multiplication and transpose operations respectively. Furthermore, a reference syntax element may be specialized to a variable reference or a function reference using the *AST_VAR_REFERENCE* or *AST_FUNCTION_REFERENCE* respectively deriving from *AST_REFERENCE* class. The hierarchical structure of classes allows implementation of specialized actions depending on the type of syntax element among the different compilation stages.

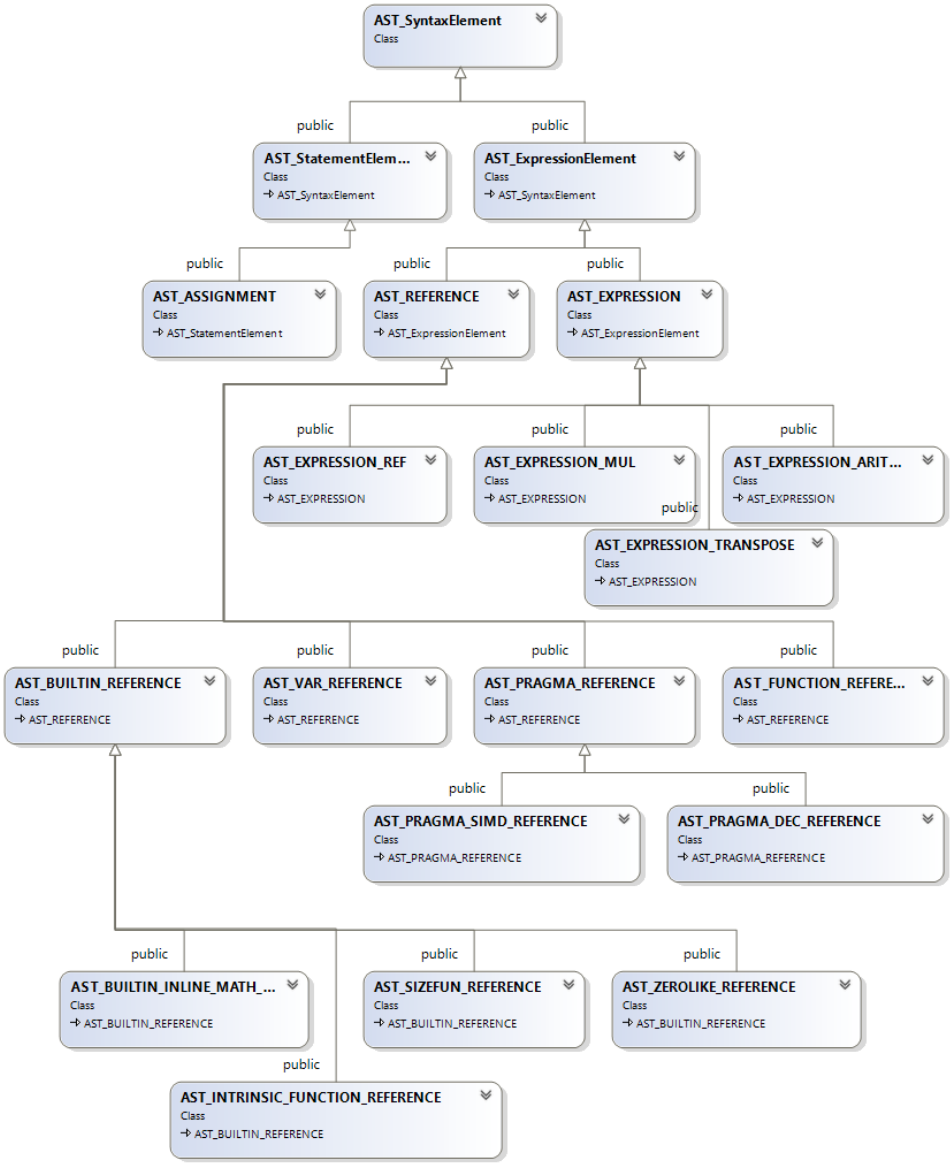

Figure 6: UML diagram of classes representing AST tree.

4.2 Type Inference

The type inference mechanism is a variation of the method described in [De Rose and Padua, 1999]. The technique presented in De Rose and Padua work combines a static and a dynamic inference methodology to translate MATLAB code to FORTRAN. The static analysis mechanism transforms the input code to static single statement (SSA) form where repetitive forward propagation of types takes place. To improve the quality of the generated code, value propagation analysis and an on-demand backward propagation of types are applied as well. For the variables not being inferred during compilation time, dynamic analysis is performed generating extra code to determine at run time the type of variables with unknown attributes. More specifically, code is generated for properly allocating arrays dimensions and extra variables are produced to handle any possible data type. Our work uses a simplification of the static inference mechanism presented in [De Rose and Padua, 1999] with only one forward repetition for variables type inference. The proposed approach doesn't currently support runtime type inference. More specifically, in the compiler type inference involves a special AST pass initiating from the leaf nodes (variables, constants) towards the root node which resolves the type of every MATLAB expression in the input program. The type of the unspecified variables derives from the assignment's right-side expression while for the left-side typed variables, type checking is performed instead of type inference. In the case of undetermined variables, such as the input parameters of the primary function, the user can declare them using pragma functions.

4.2.1 Function Calls Type Inference

One part of the type inference mechanism is the type analysis of function calls. When the type inference traversal reach an AST node that represents a function call, the compiler determines the type of function that is called (special instruction, built-in or user-defined) and performs type inference appropriately. Figure 7 shows the control flow diagram for the type inference of functions calls. Firstly, the compiler checks if the evaluated function will subsequently be mapped with a customized instruction. In that case, the parametrized processor model is invoked to infer the type of the function's result as discussed in section 4.2.2. In case that a function call cannot be matched to any custom instruction in the processor model, the compiler determines whether the function is a built-in function or a user defined function. In case of a built-in function call the result type is inferred according to a rule (similar to those used for MATLAB operations) specified internally in the compiler. If the function call refers to a user defined function, the type inference process is suspended and a new type inference instance is applied to determine the type of the current function call. More specifically, for each user defined MATLAB function the compiler reserves (in Symbol Table structure) pairs of function's

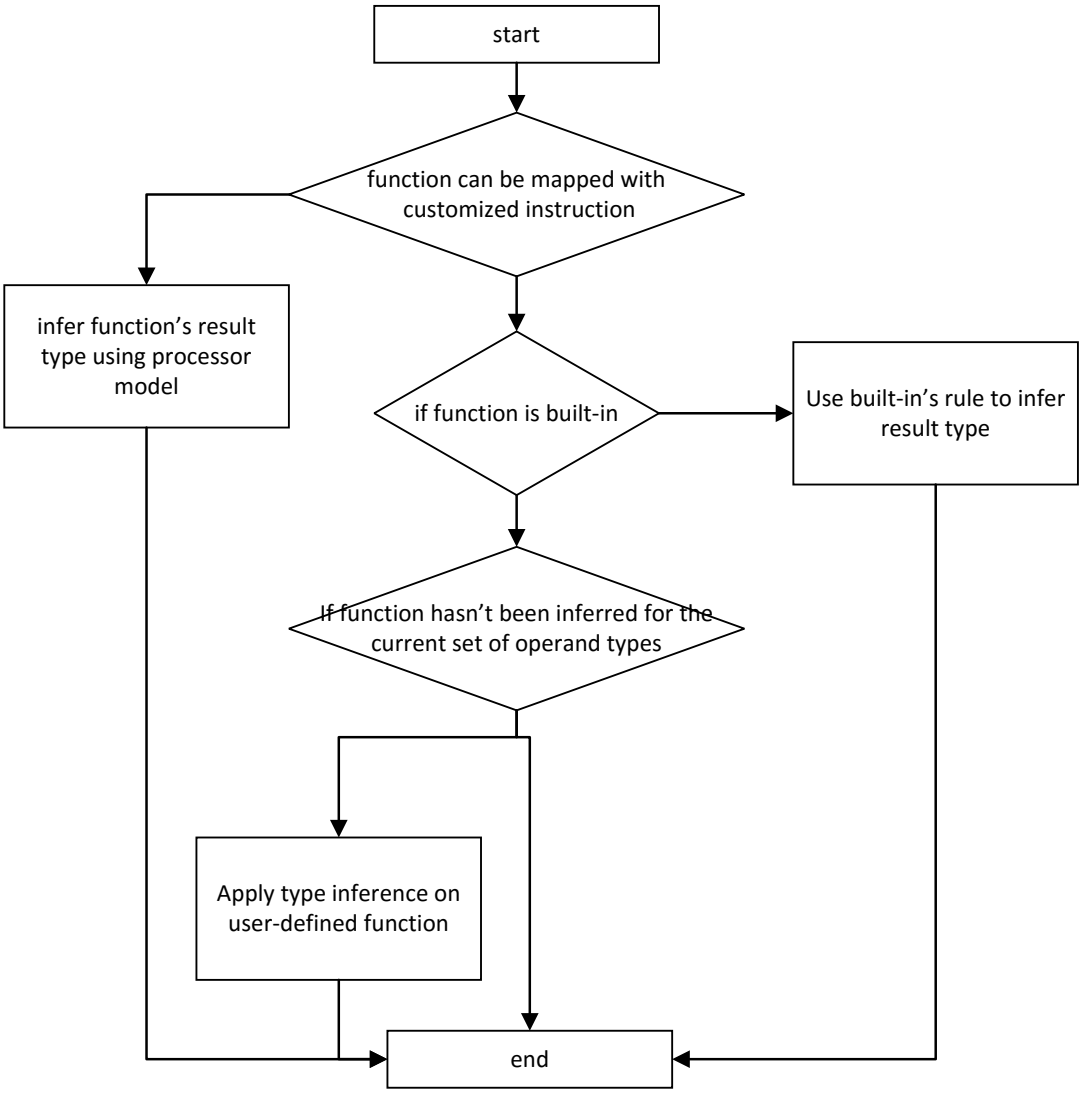

Figure 7: Control flow diagram for type inference of functions calls.

input parameter types and inferred result types. Each time a user defined function is inferred by the compiler, the pair of the input and output parameters is stored in a list. Thus, the compiler can use this information during type inference phase (searching for the result type of a function based on the parameter types) avoiding the type analysis of function calls with same parameter types. Finally, if none of the above cases occurred, an error message of undefined function is shown to the user.

4.2.2 Intrinsics Type Inference

The type inference of function calls to be mapped to custom instructions is based on the parameterized processor model. When the type inference traversal reaches a function call node, the processor model is used to infer the type of the function call getting as input the types of the function's parameters. The processor model is used to apply a forward matching (discussed in section 4.3) to identify the appropriate instruction. When a custom instruction is identified, type inference is applied using the types of the input parameters. The unspecified attributes of the custom instruction result type are determined by inheriting the type attributes from the first parameter of the function call. The fully specified result type of the custom instruction can be used as the result type of the function call contributing for the type inference of the higher AST expressions. Type inference of function calls mapped to custom instructions can be used extensively on MATLAB functions during compilation since the majority of them perform element-wise processing on the input matrices returning output with the same type as the input type.

Forward matching is necessary only for function calls at the type inference stage because the parametrized processor model is used in the type inference of function calls which will be subsequently matched to custom instructions. At this stage no information of the matched custom instructions is stored in the AST. This is performed during instruction selection stage where the complete matching of both functions and operations is implemented. Integrating the instruction selection phase in the type inference phase would not be a good option as far as the compiler's design is concerned. The code would be too complicated and difficult to maintain/extend.

4.2.3 Intrinsics Type Inference Example

Figure 8 shows the type inference process of *atan* function using the *ATAN* custom instruction in the processor model. The scalar instruction can be used for complex fixed point variables of fraction length 12 and word length 16 with any shape/size (unspecified shape/size). The type of the function's output is a complex fixed point variable with word length of 16 and fraction length of 15 (instead 12) while the array shape/size depends on the input parameter's shape/size (unspecified shape/size). During type inference of function call, the result type inherits from the input parameter the shape/size which is a vector of 32 elements while the attributes of data type, word and fraction length and SIMD width (zero due to scalar) are copied from the specification of instruction's result type. Thus, the result type of the function has been fully determined using a partial specification in the processor model in combination with the input parameter type of the function call.

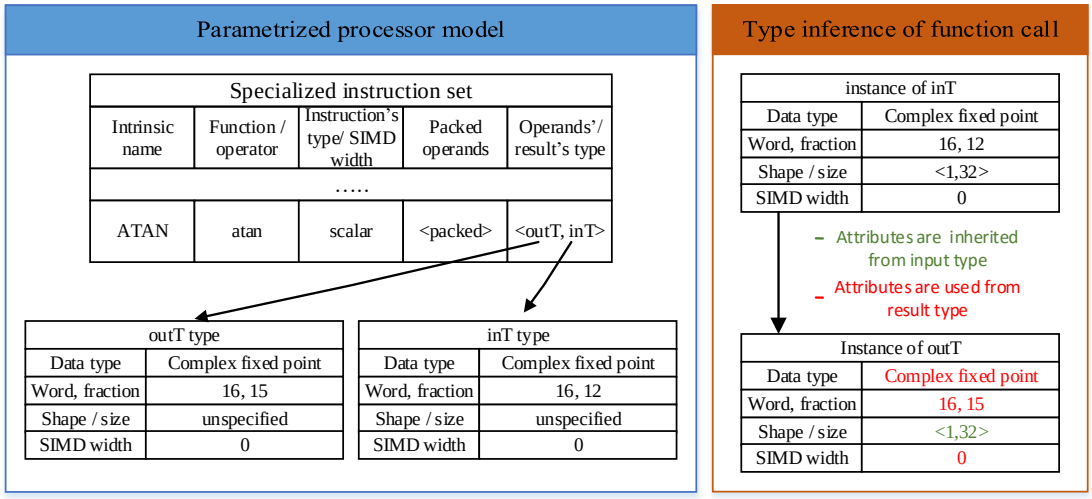

Figure 8: Type inference example.

4.2.4 Key Contribution of Type Inference

The type inference mechanism developed in the compilation framework is composed by methodologies that are already existed in the literature. However, the type inference of function calls that are mapped to customized instructions according to their description in the processor model is one of the main contributions of this thesis regarding the type checking/inference.

4.3 Instruction Selection

The next compiler stage is one of the contributions presented in the thesis applying an instruction selection algorithm to map the MATLAB code with the customized instructions described in the processor model. This stage is a post-order traversal (algorithm 1) which acts on AST nodes of function calls and MATLAB operations utilizing the parametrized processor model to search for available custom instructions of the target processor. To achieve this, the traversal passes to the parametrized processor model, the operands/parameters types, the operation type or function name and information on whether or not the current function or operation is required to be included inside a SIMD block. In the next step (algorithm 2), the parametrized processor model matches the provided information with the appropriate available custom instruction. The process of matching involves the sequential comparison of each record contained in the custom instruction list with the provided criteria until the first match is found. In a first step, the MATLAB operation/function is compared against the instruction's name in order to be matched with an appropriate custom instruction. Afterwards, the instruction's type is

examined, to check whether a SIMD instruction is required or not. If *SIMD_width* variable is zero (current node not in SIMD block), then a scalar or array type instruction should be identified. Otherwise the examined instruction must have the same SIMD width as the SIMD block width (same value of *SIMD_width*). After examining the instruction's type, the input parameters of the function/operation and the instruction under consideration are evaluated/compared (only for the specified type attributes). The first instruction matching the function/operation, is returned to the instruction selection traversal process and the custom instruction is assigned to the current AST node.

ALGORITHM 1: Instruction selection traversal (AST_instrSelect_trav)

```
   Data: SIMD_width
   Result: AST with matched custom instructions
1  if current_node == 'StartSIMD' then
2  |  SIMD_width ← currrent_node.getSIMD_width();
3  else if current_node == 'StopSIMD' then
4  |  SIMD_width ← 0;
5  foreach descendant D_i of current_node do
6  |  D_i.AST_instrSelect_trav(SIMD_width);
7  |  input_param_list.add_type(D_i);
8  if current_node.isFunctionCall() || current_node.isOperation() then
9  |  matched_instruction ←
   |   instrSelection(node_name, SIMD_width, input_param_list);
```

ALGORITHM 2: Instruction Selection procedure

Data: node_name, SIMD_width, input_param_list
Result: Returns the matched custom instruction

```
1  foreach instruction instr_i in proccesor_model_list do
2      if node_name == instr_i.getName() then
3          if (instr_i.isScalar() || instr_i.isArray()) && SIMD_width! = 0 then
4              continue;
5          else if instr_i.getSIMD_width() != SIMD_width then
6              continue;
7      foreach type t_j in input_param_list do
8          if SIMD_width ! = 0 && t_j.getSIMD_width() ! = instr_i.getSIMD_width()
            then
9              continue;
10         else if instr_i.par_j.Type_isSpecified() && t_j.getType() ! = instr_i.par_j.getType()
            then
11             continue;
12         else if instr_i.par_j.Size_isSpecified_j() && t_j.getSize() ! = instr_i.par_j.getSize()
            then
13             continue;
14         else if instr_i.par_j.ConstantValue_isSpecified_j() && t_j.getConstantValue() ! =
            instr_i.par_j.getConstantValue() then
15             continue;
16         return intr_i;
17 return NULL;
```

4.3.1 Instruction Selection Example

Figure 9 presents an example of the instruction selection stage. The figure presents the matching of the add operator and the cosine MATLAB function with the vectorized instructions available in the parametrized processor list. In the first step, the traversal collects the parameters types and the SIMD block width. In the next step, instruction selection is performed to identify an available instruction according to the provided criteria.

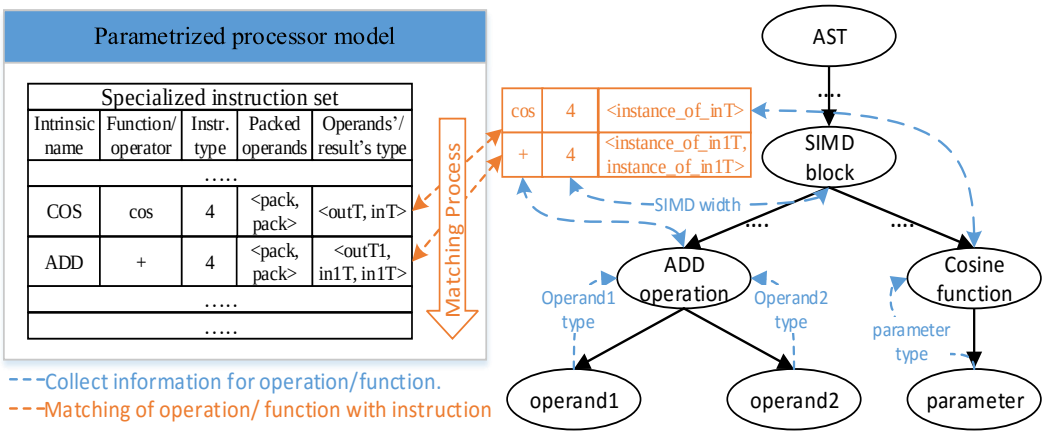

Figure 9: Instruction Selection example.

4.3.2 Generated Custom Instructions of benchmark

This subsection discusses the custom instructions which are generated during the compilation of the benchmark on the ARM and x86 targeted architectures. The compiler may generate code including custom instructions which represent native intrinsic functions of the current architecture or any other C function such as built-in and user defined functions. Although the generated code for the ASIPs have been mapped to intrinsic functions, the MATLAB operations and functions have been matched to custom instructions which are represented by C user defined functions for the ARM and x86 targets. The names of the corresponding user defined functions between the two architectures (ARM and x86) are the same, but their implementations are different for ARM and x86. Finally, the C functions are included in header files ('common_ARM.h' and 'common_x86.h') which are also given as input to the C compiler for the compilation of the generated code for a specific target.

The mapping of MATLAB code to user defined C functions for ARM and x86 has been performed for two reasons. Primary, several MATLAB operations and functions

cannot be directly mapped to C code for these architectures. For example, the current processors don't provide math operations such as the calculation of sine and exponential in vector mode. Thus, it was required to be written C vectorized code implementing these operations in the form of user defined functions. Furthermore, even if a direct mapping of a MATLAB operation/function to an intrinsic could be performed, the MATLAB code has been mapped to user defined functions used as wrappers to intrinsic functions. The use of wrappers enables the portability of the generated code on various architectures. Therefore, the generated code can be compiled/executed in both architectures using the appropriate version of the common header file. The described technique avoids the description of the parametrized processor model for each architecture.

Tables 13 and 14 present the generated custom instructions for the set of benchmarks targeting ARM and x86 architectures with fixed point data types. More specifically, the tables present the number of custom instructions that have been generated per benchmark producing scalarized (Table 13) and vectorized code (Table 14) (using packed data types and SIMD width 4). The custom instructions generated for other experiments such as targeting to ASIPs or using floating point data types are similar, and they are not presented in the thesis. In table 13, the custom instructions of rows 1-8 are math operations, the rows 9-17 represent complex arithmetic instructions, the rows 18-24 are auxiliary custom instructions for save/read to/from the real and imaginary part of complex variables and the two last rows are shifting operations which are used in fixed point arithmetic. For all the benchmarks, 175 custom instructions have been generated with most of them appeared in the FFT's and QR-decomposition applications.

Similarly in table 14, the first 8 rows are vectorized math functions and the following rows are complex arithmetic operations with vectors (rows 9-20). The rows 21-26 are packing and unpacking operations for vectors. Although the benchmark compiled for the current measurement with packed data types, packing/unpacking operations were generated to expand scalar operands to vectors for SIMD processing as well as instructions to access/write vectors at benchmark parts of scalar processing with vectors. The rest of the table rows are custom instructions for assigning vector-to-vector commands (27-28), vectorized shifting operations (29-30), reading the real part/imaginary of a vector (31-32) and other scalarized instructions (33-37). Although vectorized code is generated, scalarized instructions are required for some parts of benchmark. For instance, the mean application calculates in scalar mode the average value of the vector which has been occurred by previous vectorized processing. For all the benchmarks, the custom instructions that have been generated are 347, with most of them appearing in the FFT's and QR-decomposition. The scalarized generated code includes less custom instructions than the vectorized generated code, because the vectorized code includes packing and unpacking operations. Furthermore, some operations such as scalarized addition of real numbers have been generated without the use of any custom instruction reducing the number of generated instructions.

	cfo	cordic	fft32	fft64-v1	fft64-v2	fft128	fir	mean	qr-dec
cexp_i	1								4
angleR_i		1							
angle_i									5
sin_i									2
cos_i									2
abs_i									6
conj_i									3
scaleDown_i		1						1	
cadd_i			4	6	8	9		1	3
cadd_i_rc									1
csub_i			4	6	8	11			
cminus_i				1	2	2			
cmul_i	1		4	4	6	10			1
cmul_i_rc									6
cmul_i_cr									3
cdiv_i_cr									1
SET_RE_IM_F				2	2	5			3
SET_RE_IM_I				3	2	3			8
getRe_f				1					
getIm_f				1					
getRe_i									2
getIm_i									2
_rshift	6	5					1		5
_lshift	1								
OVERALL:	**8**	**7**	**12**	**24**	**28**	**40**	**1**	**2**	**53**

Table 13: Matched intrinsics of scalarized fixed point generated code on ARM/x86

	cfo	cordic	fft32	fft64-v1	fft64-v2	fft128	fir	mean	qr-dec
v4__abs__i									6
v4__angle__i									5
v4__angleR__i		1							
v4__cos__i									2
v4__sin__i									2
v4__cexp__i 1	1								4
v4__conj__i									3
scaleDown__i		1							
v4__sub__i			4	2	4	4			
v4__sub__i__rr		2							5
v4__minus__i__r									1
v4__add__i			4	2	4	5		1	3
v4__add__i__rc									1
v4__add__i__rr	2	1					1		1
v4__mul__i	1		4	3	5	7			1
v4__mul__i__rr	3	5					1		1
v4__mul__i__rc									6
v4__mul__i__cr									3
v4__div__i__rr									4
v4__div__i__cr									1
__insert__v__ic4				8	8	8		1	2
__insert__v__i4	1	2							
insert__all__ic__4				3	1	5			13
insert__all__i__4	4	6					2		5
__extract__v__ic4				16	16	18		1	2
__extract__v__i4		3							2
__cp__vect	7			2	12	6	1		17
__cp__vectR__to__c4									3
V4__ASR__i	1								4
V4__ASL__i	1								
v4__getIm__r									2
v4__getIm__i									2
SET__RE__IM__F				2	2	6			
SET__RE__IM__I				1		4		2	13
cadd__i				4	4	4		1	
cdiv__i__cr								1	
__rshift	1								
OVERALL:	**22**	**21**	**12**	**43**	**56**	**67**	**5**	**7**	**114**

Table 14: Matched intrinsics of SIMD fixed point generated code on ARM/x86

4.4 Support for Data Parallel Execution - AST Decomposition

After instruction selection, a multi-pass stage is applied to transform the AST for data parallelization execution. This compiler stage prepares the AST for SIMD code generation, introducing statements in the intermediate representation to convert data in packed form (and vice versa) ready for SIMD execution. Currently the compiler doesn't apply dependence-analysis to determine whether vectorization should be applied to a code segment annotated by the user for this purpose. Furthermore, no enabling transformations such as loop-peeling, loop-skewing or loop-unswitching are applied before vectorization. The data parallelization steps applied by the compiler include:

- A first pass of AST which involves the decomposition of complex MATLAB array expressions to simpler ones retaining the vectorized form of the SIMD operands. After this pass, each statement includes only one operation or function call facilitating the implementation of the following steps.
- In the next step, information related to the instruction which has been assigned to the current node (operation or function) during instruction selection is used to determine which operands need packing. Not all operands of a SIMD instruction need to be packed vectors. For instance, a vector shifting instruction may include an input vector in packed form and a non-packed scalar constant defining the shifting factor. According to custom instruction's prototype, new statements are generated to transfer only the unpacked SIMD operand values' to packed data structure (ready for SIMD processing). Similar statements are introduced in the low-level IR, only for SIMD operation result variables which have been declared as unpacked data types, transferring the packed result's values to the unpacked data structure. Packing/unpacking statements are also inserted for packed operands and results with non-continuous memory accessing.
- In the final step, an implementation of the local value numbering [Cooper and Torczon, 2012] is performed to remove the redundant intermediate packing/unpacking statements. The redundant packing/unpacking elimination removes packing/unpacking statements existing among SIMD operations by exploiting the fact that an intermediate packed variable can be used at a subsequent SIMD operation without being unpacked first and then packed again.

The introduction of packing/unpacking statements is one of the major contributions of the thesis. The methodology allows the generation of C code that executes data in parallel using unpacked data types for the compilation of MATLAB input code. In combination with the existing local value numbering algorithm, the output code is optimal packing unpacked data only at their first invocation in SIMD processing and vice versa.

4.4.1 AST Decomposition

The AST decomposition stage transforms the AST representation to a form that is suitable for code generation. The complex MATLAB statements are converted to simple statements that are subsequently translated to C code by the compiler's back-end. The AST decomposition stage (algorithm 3) is an AST traversal examining statements that require decomposition. When a statement determined as complex, it is decomposed in two or more new statements introducing temporary variables to link the new statements. Then, type inference is applied to assign the type of the decomposed expressions to new temporary variables. After type inference, the AST decomposition traversal is called recursively using the decomposed sub-AST for further simplification. Finally, the simplified statements are replaced with the initial complex statement constructing the new decomposed AST.

Figure 10 presents an example of the AST decomposition algorithm. The input statement is decomposed to two new statements which are given as input to a new recursive instance of the AST decomposition algorithm. The new traversal examines the input statements and decomposes further the already simplified first statement. Then, the input sub-AST is replaced with the decomposed statements and it is returned to the first invocation of AST decomposition algorithm. The decomposed statements replace the complex initial statement leading to the final result of the algorithm - the decomposed AST.

Some of the statements are subject to decomposition include: expressions with more than one operation that mapped to customized instructions, array concatenation statements, and functions calls with references with indexing. These cases are frequently appeared in MATLAB code. However a number of other cases are also decomposed but they aren't presented in the thesis. Examples of AST decomposition exposing various cases are presented below.

ALGORITHM 3: AST decomposition traversal (AST_decomp_trav)

```
   Data: AST_statements
   Result: Decomposed AST
1  foreach statement S_i of AST_statements do
2  |  if decomposition_required(S_i) then
3  |  |  decomposed_statements ← decompose(S_i);
4  |  |  foreach statement D_j of decomposed_statements do
5  |  |  |  type_inference(D_j);
6  |  |  decomposed_statements ← AST_decomp_trav(decomposed_statements);
7  |  |  AST_statements → insert(decomposed_statements);
8  |  |  AST_statements → remove_statement(S_i);
9  |  |  S_i ← decomposed_statements.last;
10 return AST_statements;
```

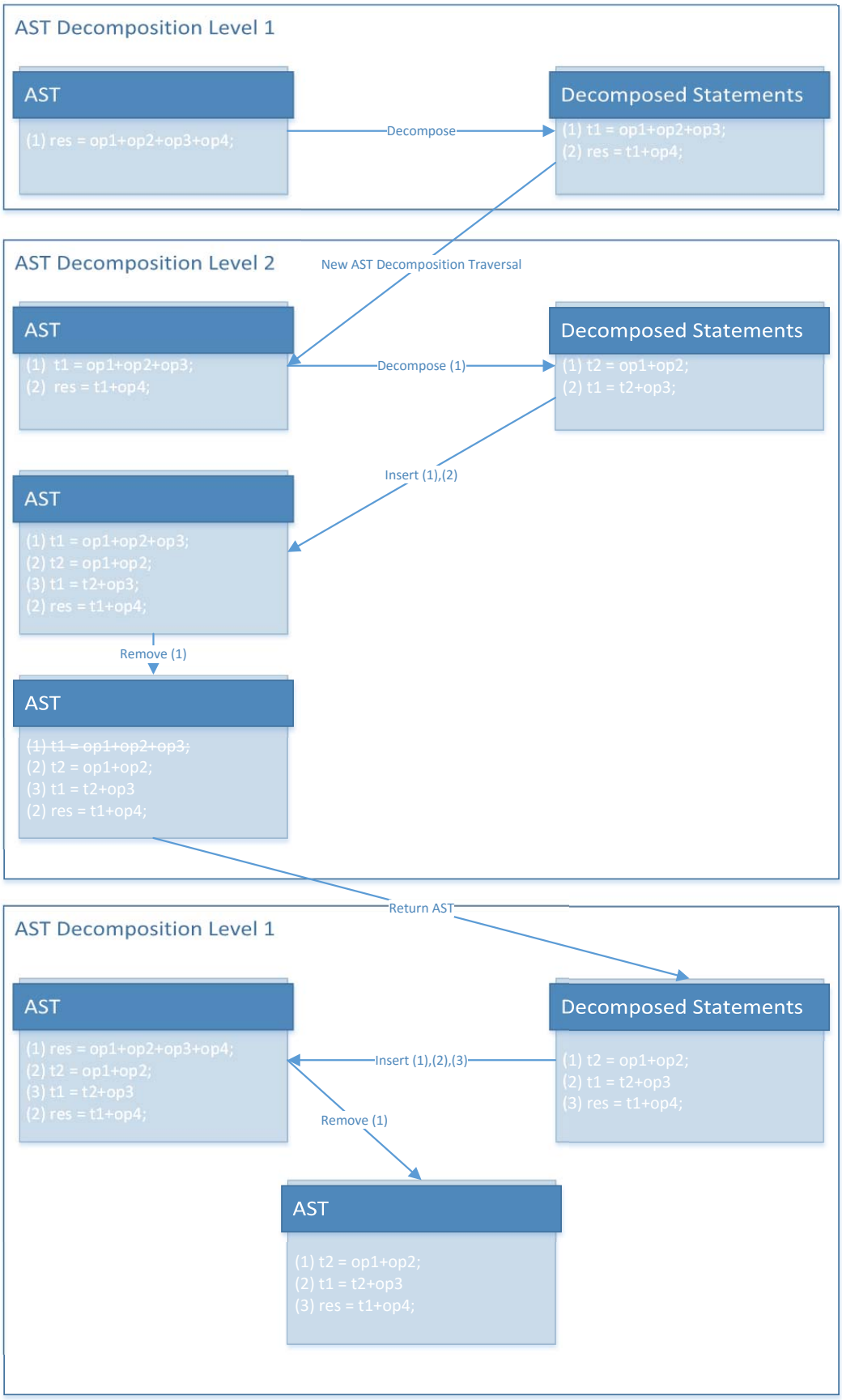

Figure 10: AST Decomposition example.

Customized instructions Decomposition The MATLAB statements including operations that are mapped to customized instructions are decomposed to simplify the code generation process. Thus, each statement represent a custom instruction consisting of only one MATLAB operation/function. The listing 4.1 presents a code example with a MATLAB assignment including three operations that are mapped to customized instructions. The AST decomposition process decomposes the statement as it is presented in listing 4.2. Two new statements are introduced for the calculation of the element-wise multiplications before the multi-operation statement. Furthermore, the decomposed operations are replaced by temporary variables.

```
1 function [res] = CustomInstrExample(op1,op2,op3,op4)
2   dec_dblc('op1',1,8);
3   dec_dblc('op2',1,8);
4   dec_dblc('op3',1,8);
5   dec_dblc('op4',1,8);
6   res = op1.*op2 + op3.*op4;
7 end
```

Listing 4.1: Customized instructions example

```
1 %Matlab code generation from AST
2 function [res] = CustomInstrExample(op1,op2,op3,op4)
3   dec_dblc('op1',1,8)
4   dec_dblc('op2',1,8)
5   dec_dblc('op3',1,8)
6   dec_dblc('op4',1,8)
7   tmp_0 = op1*op2;
8   tmp_1 = op3*op4;
9   res = tmp_0+tmp_1;
10 end
11
```

Listing 4.2: Decomposed Customized instructions example

Function Statement Decomposition AST decomposition stage creates temporary statements for the function parameters that cannot be passed directly to a function call at C. Function parameters which are decomposed, maybe a reference of array variable with indexing or array operations. However, function parameters that comply with C syntax and decomposition isn't required are scalar references, scalar operations or array references without indexing. The listing 4.3 shows a MATLAB code example that calls a user-defined function of two output and three input parameters. The function call in line 4 is decomposed (in listing 4.4) as follows: a) a new statement is introduced before the function call that calculates the array addition (*op+op*) and the last function input parameter is replaced be the introduced temporary array b) the last output parameter (*res2(1:2:16)*) is replaced by a new temporary variable and a new statement is inserted

after the function call to transfer the data from the temporary variable to *res2*. Finally, the rest function parameters don't need decomposition. These parameters are two array references and one scalar reference of array that in C can be passed to the function by reference and by value respectively.

```
1  function [res1 res2] = FunctionExample(op)
2    dec_dbl('op',1,8);
3    dec_dbl('res2',1,16);
4    [res1 res2(1:2:16)] = myFun(op(1), op, op+op);
5  end
6
7  function [res1, res2] = myFun(par1,par2,par3)
8    res1 = par1+par2;
9    res2 = par1+par3;
10 end
```

Listing 4.3: Function example

```
1  %Matlab code generation from AST
2  function [res1,res2] = FunctionExample(op)
3    dec_dbl('op',1,8)
4    dec_dbl('res2',1,16)
5    _tmp_0 = op+op;
6    [res1 _tmp_1 ] = myFun(op(1),op,_tmp_0);
7    res2(1:2:16) = _tmp_1;
8  end
9
10 function [res1,res2] = myFun(par1,par2,par3)
11   res1 = par1+par2;
12   res2 = par1+par3;
13 end
```

Listing 4.4: Decomposed Function example

Array Concatenation Statement Decomposition The MATLAB array concatenation statements are decomposed to multiple simplified assignments. Each of the new statements correspond to the assignment of an element inside the brackets to left-side variable. Listing 4.5 presents the source code of a MATLAB function including an array concatenation statement (line 4). The statement consists of the concatenation of four elements with dimensions 1x8 creating a new matrix of 2x16. Listing 4.6 shows the decomposed MATLAB code of the source code presented in listing 4.5. The array concatenation statement of listing 4.5 is converted to four statements. Each of them assign an element from array concatenation to a different part (generating references with indexing) of the left-side matrix.

```
1 function [res] = arrayConExample(op1,op2)
2   dec_dbl('op1',1,8);
3   dec_dbl('op2',1,8);
4   res = [ op1+op2 op1; op1-op2 op2];
5 end
```

Listing 4.5: Array Concatenation example

```
1 %Matlab code generation from AST
2 function [res] = arrayConExample(op1,op2)
3   dec_dbl('op1',1,8)
4   dec_dbl('op2',1,8)
5   res(1,1:8) = op1+op2;
6   res(1,9:16) = op1;
7   res(2,1:8) = op1-op2;
8   res(2,9:16) = op2;
9 end
```

Listing 4.6: Decomposed Array Concatenation example

4.4.2 Packing/Unpacking statements Introduction

The introduction of packing/unpacking statements is an AST traversal (algorithm 4) that acts on the user defined SIMD blocks. The process is applied at the statements correspond to SIMD instructions inserting packing/unpacking statements to enable data parallel execution. For the statements of SIMD block, the instruction's operands and result are examined to determine if packing/unpacking of data is needed. To achieve this, the customized instructions that have been stored in the AST nodes during instruction selection phase are retrieved to specify the operands need to be packed. According to that specification, packing statements are introduced in the AST. More specifically, different packing commands are inserted depending to the references are involved in the SIMD instructions. For scalar references or constant values, statements that expand the scalar value to vector are introduced (line 12 of listing 4.8). For references of unpacked data with sequential indexing, a simple pack command is inserted to transfer massively the unpacked data to the vector structure (line 7 of listing 4.8). Finally, references with non-contiguous memory accessing are packed inserting a for-loop that inserts one data element to a vector at each repetition (lines 13-15 of listing 4.8). The last case is applied on references of unpacked or even packed data. The re-packing of packed references is required to reorder the vector elements to an appropriate sequence. After the introduction of packing statements, unpacking instructions are also inserted in the AST to transfer the packed instruction's result to unpacked variables. There are two types of unpacking statements - the instant transferring of vector to memory and the

ALGORITHM 4: Packing/Unpacking statements Introduction (Packing_introduction)

```
   Data: AST
   Result: AST with packing/unpacking statements
1  foreach simd_block B_i of AST do
2      foreach statement S_j of B_i do
3          AST_decomp_trav(S_j);
4          foreach operand P_k of S_j do
5              if needPacking(P_k) then
6                  if isScalar(P_k) || isConstantValue(P_k) then
7                      insertExpand(P_k);
8                  else if (sequentialIndex(P_k) || noIndex(P_k)) && isUnpackedRef(P_k)
                    then
9                      insertMassivePacking(P_k);
10                 else if nonSequentialIndex(P_k) then
11                     insertPacking(P_k);
12         if (sequentialIndex(resultOf(S_j)) || noIndex(resultOf(S_j))) &&
            isUnpackedRef(resultOf(S_j)) then
13             insertMassiveUnpacking(resultOf(S_j));
14         else if nonSequentialIndex(resultOf(S_j) then
15             insertUnpacking(resultOf(S_j));
```

unpacking of data using a for-loop for non-sequential accessing of memory. Similarly to the re-packing of packed references, unpacking is also applied on packed result reference with non-sequential indexing.

The packing/unpacking statements introduction requires the use of vector variables to transfer data from/to memory. The procedure generates a new vector variable for each textually different reference that pack/unpacked. The textually same references (same variable name and indexing) use the same vector for packing/unpacking. The re-use of same vectors assists the elimination of redundant packing/unpacking statements as it is described below.

Listing 4.7 presents a MATLAB code example with an invocation to *scaleUp* function and an array addition inside a specified SIMD block. The *scaleUp* function performs bitwise shift left on the input vector (first parameter) by the scalar value which is given as the second parameter. Finally, the variables that are used in the example are unpacked data type. Listing 4.8 shows the transformed code after the introduction of packing/unpacking statements by compiling the code of listing 4.7 using a processor model that maps the *scaleUp* and array addition to SIMD instructions. The first parameter of *scaleUp* function is an unpacked reference with sequential indexing and a packing instruction is inserted (line 7) to transfer massively the data to the vector. The second parameter of the function is unpacked scalar, thus packing is avoided. The array addition of listing 4.7 consists of unpacked operands and results. The first operand is a scalar reference, therefore an expand instruction is inserted before the operation (line

12) to fill a vector with the value. The second operand and result are references with non-continuous accessing to memory. Consequently for-loop statements are inserted before and after of the addition to pack and unpack the data respectively.

```
function [res1 res2] = packingExample(op)
  dec_fixpc('op','16','12',1,16);

  startSIMD(4,8)
  res1 = scaleUp(op(1:8),2);
  res2(1:2:16) = op(1)+op(1:2:16);
  endSIMD()
end
```

Listing 4.7: Packing/unpacking example

```
%Matlab code generation from AST
function [res1,res2] = packingExample(op)
  dec_fixpc('op','16','12',1,16)

  startSIMD(4,8)
  %res1 = scaleUp(op(1:8),2);
  vec_0 = pack(op(1:8));
  vec_1 = scaleUp(vec_0,2);
  res1 = unpack(vec_1);

  %res2(1:2:16) = op(1)+op(1:2:16);
  vec_2 = expand(op(1));
  for k=1:2:16
    vec_3 = pack_element(vec_3, op, k);
  end
  vec_4 = vec_2+vec_3;
  for k=1:2:16
    res2 = unpack_element(vec_4, res2, k);
  end
  endSIMD()
end
```

Listing 4.8: Packing/unpacking example after AST decomposition

4.5 Redundant Packing/Unpacking Elimination

Redundant code elimination is a common local optimization that is applied in basic blocks to remove computations that have been previously calculated. The algorithm is applied to remove/replace the redundant statements performing analysis to determine whether or not re-computation of a value is required. The analysis comprises the examination of the statements in order to check if the values of the variables have been modified since the last execution of the same computation. In case that the values of the variables involved in the examined statement haven't been modified, the redundant

computation is replaced by the assignment of the variable that holds the result of the previous calculation.

The MATLAB compiler uses an implementation of the local value numbering described in [Cooper and Torczon, 2012] to remove the redundant packing/unpacking statements that are inserted in the AST from the previous compiler's stage. The idea of value numbering algorithm is that same numbers are assigned at the references of same variables only if they have provably equal values [Cooper and Torczon, 2012]. Therefore, if the value of variable is changed (variable at the right side of assignment) a new number is assigned at the subsequent references of this variable. The result of value numbering process is the AST of basic block composed by expressions having numbering identification to determine which re-computations are redundant.

During redundant packing/unpacking elimination stage (algorithm 5), the compiler applies local value numbering to remove the packing/unpacking statements that are redundant following the method described in [Cooper and Torczon, 2012]. Firstly, the compiler assigns numbers at references and vector variables of SIMD block. Subsequently, it is examined if the vector of a packing/unpacking statement is existed in hash table. In case that vector isn't found, a new record is inserted using as key, the name of vector. The value of the new entry contains a pair of values including the text of expression which is packed/unpacked with its numbering and the type of instruction (pack or unpack). In case that an entry is existed in hash table for the examined vector, one of the following conditions is applied:

- If it is pack instruction and the reference that is packed has the same numbering with a record of hash table, then the statement is redundant. In that case the current packing statement is removed. The condition eliminates the duplicates of packing instruction of SIMD operands whose the values haven't been modified.
- If it is pack instruction and its input reference is included in the hash table with the same numbering of a reference that is used as result of unpacking instruction, then the statement is redundant. In that case of redundancy, the compiler similarly removes both the current packing statement and the unpacking statement found in the hash table. The condition eliminates the unpacking-packing instructions of intermediate results of SIMD block. Thus, the unpacking of data that are subsequently are re-packed to be used as operand to another SIMD instruction is avoided.
- If it is unpack instruction and there is a reference with same numbering in the hash table for another unpacking instruction, then the statement that is found is redundant and it is removed. The condition eliminates multi-unpacking of same data in the SIMD block and preserves that data are in packed form until the last use of them.
- In case that the text of expressions are different, the entry of the hash table is updated with the text of the expression of the current statement. For the case of

unpack-pack instructions elimination (second condition), the type of pack instruction is also changed in the hash table from unpack to pack.

ALGORITHM 5: Elimination of redundant packing/unpacking statements (Packing_elimination)

```
   Data: AST
   Result: AST including only necessary packing/unpacking statements
 1 hashTable = init();
 2 foreach simd_block B_i of AST do
 3     foreach statement S_j of B_i do
 4         localValueNumbering(S_j);
 5         if isPacking(S_j) || isUnpacking(S_j) then
 6             foundRec ← hashTable.find(getVector(S_j));
 7             if foundRec.empty() then
 8                 newRec.setKey(getVector(S_j));
 9                 newRec.setValue(pair < getExpression(S_j), getPackType(S_j) >);
10                 hashTable.insert(newRec);
11             else
12                 if isPacking(S_j) && foundRec.isPacking() then
13                     if getExpression(S_j) == foundRec.getExpression() then
14                         remove_from_AST(S_j);
15                     else
16                         foundRec.updateExpression(getExpression(S_j));
17                         hashTable.modify(foundRec);
18                 else if isPacking(S_j) && foundRec.isUnpacking() then
19                     if getExpression(S_j) == foundRec.getExpression() then
20                         remove_from_AST(S_j);
21                         remove_from_AST(foundRec.getAST_statement());
22                     else
23                         foundRec.updateExpression(getExpression(S_j));
24                         foundRec.updatePackingType(getPackType(S_j));
25                         hashTable.modify(foundRec);
26                 else if isUnpacking(S_j) && foundRec.isUnpacking() then
27                     if getExpression(S_j) == foundRec.getExpression() then
28                         remove_from_AST(foundRec.getAST_statement());
29                     else
30                         foundRec.updateExpression(getExpression(S_j));
31                         hashTable.modify(foundRec);
```

The compiler supports for-loops statements as well as if-else and while statements of scalar conditions inside a SIMD block. In that case, the compiler applies conservative analysis during local value numbering and assumes that the statements in for-loop are always executed regardless the run-time execution path of the control-flow statement.

Figure 11 presents an example of redundant packing/unpacking elimination. Firstly, the MATLAB source code is shown comprising by three SIMD additions and two scalar assignments. The second box of the figure shows the intermediate code highlighting the vectors that have been introduced after the packing/unpacking statements introduction stage. The compiler generates a new vector variable for each textually different reference Therefore, two different vectors are produced for the packing of *op2* variable which is used with different indexing in the SIMD operations. The third box of the figure shows the intermediate code applying local value numbering. The example demonstrates a case that unpacking-packing is redundant and another case that re-packing is required because value of the variable is modified. The red highlighted statements are redundant and they are eliminated because the second condition (described above) is satisfied. The variables *r1* and *r2* are firstly unpacked and subsequently are re-packed using the same numbering (7 and 12 respectively), therefore they are removed. On the other hand, the variable *op1* is packed twice using the same textually expression, although it isn't considered as redundant. This is due to the fact that the two packing references have different numbering because the value of the variable i used for indexing is modified.

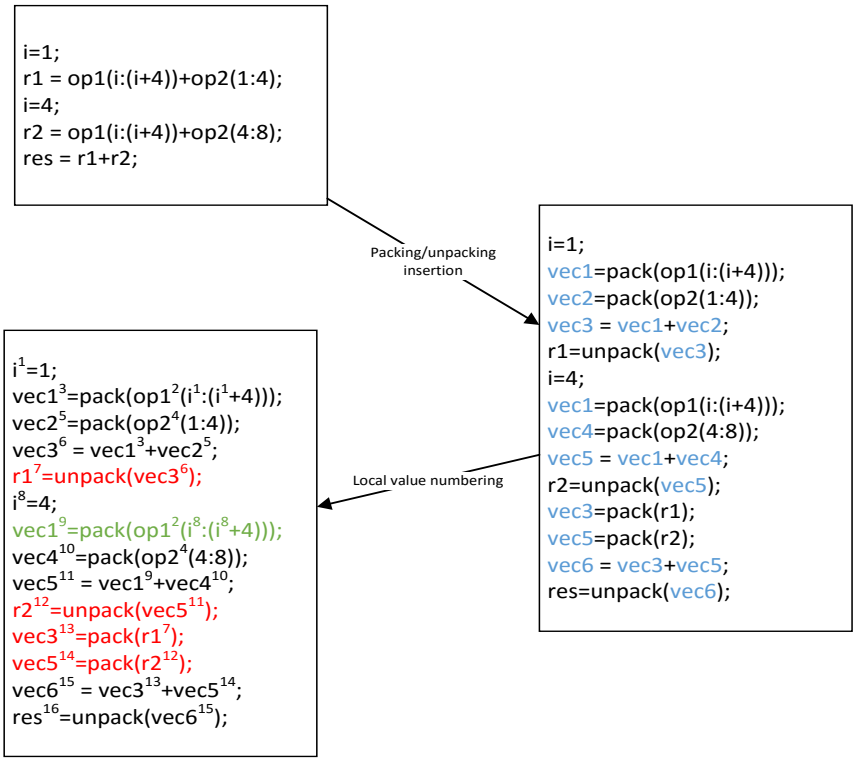

Figure 11: Redundant packing/unpacking Elimination example.

4.5.1 Inserted/Removed Packing and Unpacking Statements of Benchmark

This subsection discusses the performance of the compiler's stage regarding the support for parallel execution. The table 15 presents the number of packing/unpacking statements inserted in AST and the number of these which has been removed from packing/unpacking elimination stage for the benchmark. The results show that the 53% of the statements introduced in AST are redundant and finally are removed. This metric shows the significance of elimination of packing/unpacking statements stage for the performance of the vectorized generated code using unpacked data types. For the applications with large SIMD blocks, a higher percentage of packing/unpacking statements are eliminated. The large SIMD blocks increase the reusability of the intermediate packed results, thus more packing/unpacking statements are redundant and can be eliminated. On the other hand, application of single operation included in SIMD blocks (FIR, mean) doesn't reuse any packed data and no elimination is performed.

	removed	inserted
cfo	12	20
cordic	5	14
fft32	9	28
fft64-v1	13	22
fft64-v2	47	60
fft128	30	57
fir	0	6
mean	0	3
qr-dec	66	134
OVERALL:	182	344

Table 15: Inserted and removed packing and unpacking statements of benchmark.

4.6 Conclusion

In this chapter the middle-layer of the compiler's infrastructure was presented. In middle-layer, the compiler prepares the AST for the generation of C code. Firstly, type inference is performed to infer/check the type of MATLAB variables. The processor model is utilized for the resolution of the type of function calls that are matched with

custom instructions. Subsequently, instruction selection is applied to map the MATLAB operations and functions with the customized instructions that are described in the processor model. Finally, a multi-pass stage is performed to transform the AST to support the generation of data parallel code. The stage comprises the decomposition of the complicated MATLAB statements following by the introduction of packing/unpacking statements. Next, the elimination of the redundant statements is applied removing the packing/unpacking operations that has been inserted in the previous stage.

The middle-layer constitutes the most significant part of the compiler enabling the generation of data parallel code as well as the generation of optimized code (instruction selection) for a target architecture exploiting its specialized instructions. A number of contributions are presented in the chapter including: a) a type inference mechanism for function calls according to the processor model, b) an instruction selection algorithm for the map of MATLAB code with modules of hardware, c) and a methodology for the insertion of packing/unpacking statements to enable the data parallel execution of the generated code.

Chapter 5

Compiler's Back-end (Code generation)

The compiler's back-end consists of the code generator which produces the output of the compiler. The code generator translates the intermediate source code representation to the target program which may be in the form of another programming language or assembly/binary code.

The back-end of the MATLAB compiler generates ANSI C code which is compatible with any C/C++ compiler supporting the target architecture. The style of the generated C code may be scalarized or vectorized including the custom instructions of the target processor/ASIP in the form of intrinsic functions. The back-end generates the intrinsic functions that have been matched at the instruction selection stage while equivalent C code is produced for the MATLAB operations that haven't been mapped with hardware. The derived data types of the generated C code can be floating point, integer or fixed point. For the latter, extra C code is generated for handling the fixed point arithmetic. The figure 12 shows the association of the different parts of compiler's back-end.

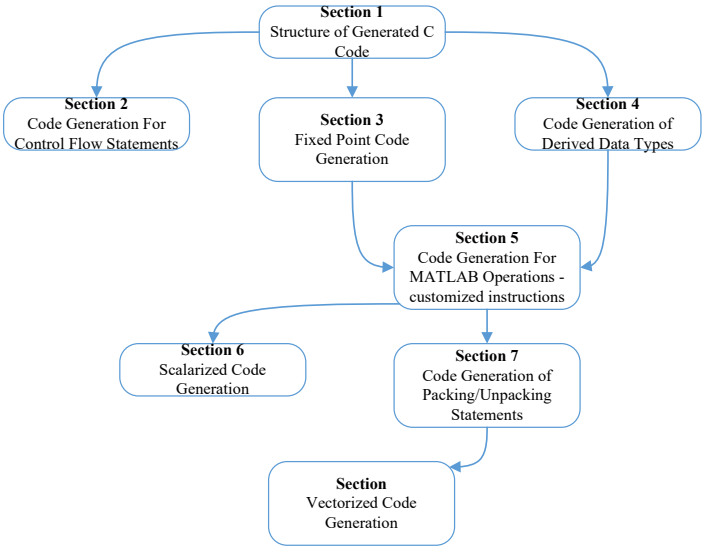

Figure 12: Dependence graph of back-end components.

5.1 Structure of Generated C Code

The code generator produces a C source file including the translated C code. For each MATLAB function of the input source files, one or more C implementations are produced depending on the different parameter types that have been determined during the type inference stage. The code generator produces two additional C header files. The first includes the deceleration of the functions, the additional header files which have been specified in the parametrized processor model and the definitions of the C derived variables which have been declared in MATLAB as global variables. The second header file include *#define* directives that define data names for the C primitive types.

The listings 5.2, 5.3 and 5.4 shows the output of the compiler using the input MATLAB code of listing 5.1. The MATLAB example includes two definitions of variables which one of the them is specified as global variable *arr*, and two function calls myFunc using different type of parameter. Listing 5.4 shows the translated C code including the definition of *CodeOutput* function and two implementations of *myFunc* function - one for each different type that is called in MATLAB code. The Listing 5.2 file is the C header file consisting of a)the declaration of the three generated functions, b)the definition of *arr* variable which has been declared as global variable in MATLAB code and c)an *include* statement of *common.h* file which has been specified in the processor model as additional header file. Finally, the listing 5.3 comprises *#define* preprocessor directives that allow the definition of C types. The derived variables are generated using the specified data names of the file instead of producing variables with the direct primitive types.

```
1  function [] = codeOutput(op, arr)
2    global arr;
3    dec_dbl('op');
4    dec_dbl('arr',2,3);
5    res1 = myFunc(op);
6    res2 = myFunc(arr);
7  end
8  function [res] = myFunc(op)
9    res = 1-op;
10 end
```

Listing 5.1: Compiler's output example (MATLAB file).

```
#ifndef _EX_C_CODE
#define _EX_C_CODE
#include "ex_Defines.h"
#include "common.h"
void codeOutput_2(D_DBL_TY op,D_DBL_TY arr[3][2]);
void myFunc_3(D_DBL_TY &res, D_DBL_TY op);
void myFunc_4(D_DBL_TY res[3][2], D_DBL_TY op[3][2]);
D_DBL_TY arr[3][2];

#endif
```

Listing 5.2: Compiler's output example (C header file).

```
#ifndef _EX_DEFINES
#define _EX_DEFINES
#define D_SINT8 signed char
#define D_UINT8 unsigned char
#define D_SINT16 signed short
#define D_UINT16 unsigned short
#define D_SINT32 signed int
#define D_UINT32 unsigned int
#define D_SINT64 signed long long
#define D_UINT64 unsigned long long
#define D_LOG_TY bool
#define D_DBL_TY double
#define D_CHR_TY char
#endif
```

Listing 5.3: Compiler's output example (defines file).

```
#include "ex_C_code.h"

void codeOutput_2(D_DBL_TY op,D_DBL_TY arr[3][2]){
  D_DBL_TY res1;
  D_DBL_TY res2[3][2];
  myFunc_3(res1 ,op);
  myFunc_4(res2 ,arr);
}
void myFunc_3(D_DBL_TY &res, D_DBL_TY op){
  res = 1 - op;
}
void myFunc_4(D_DBL_TY res[3][2], D_DBL_TY op[3][2]){
  D_SINT32 i1 ,i2;
  for(i1=0; i1 < 3; i1=i1+1){
    for(i2=0; i2 < 2; i2=i2+1){
      res[i1][i2] = 1 - op[i1][i2];
    }
  }
}
```

Listing 5.4: Compiler's output example (C source file).

5.2 Code Generation For Control Flow Statements

The compiler fully supports the code generation of control flow *for-loop*, *if-else* and *while* statements and partially supports the code generation of *switch-case* statements. The MATLAB language provides *if-else* and *while* statements consisting of conditions of scalar or array dimensions. In MATLAB, a condition of array dimensions is true if all the element-wise operations satisfy the condition. Moreover, MATLAB is deployed with for-loops statements which mostly are used with an expression of colon operation to represent the range of iterations and the increment of loop. The code generator prints C for-loops using the operands of the colon operation for the lower-upper bounds and the increment of the loop. In case of known values in compile time, the operands of colon operations are simply printed to represent the translated for-loop as follows:

- The first colon operand is used as the initializing expression of for-loop statement.
- The last colon operand composes the for-loop condition evaluating if the loop counter is greater than the upper bound.
- In case that colon operation is composed by three operands, the second one is used to express the step of for-loop statement. If the colon operation consists of only two operands, the unit is used as increment of the for-loop statement.

However, MATLAB for-loop statements using variable colon operands, are translated to C for-loops consisting of a complex condition to terminate accordingly the execution of the for-loop. The condition examines if the loop counter exceeds the upper bound of for-loop statement considering that the colon operands may be positive or negative values.

The *if-else* and *while* statements composed by conditions of scalar dimensions, can be directly translated to C code. However, additional code is generated for MATLAB control flow statements of conditions with array dimension. For the C code generation of an array conditional *if-else* statement, nested for-loops are printed to perform the operations which are included in the condition assigning the result to a scalar variable. The for-loops are terminated if all the elements of the array have been accessed or the scalar variable is false. The checking of the scalar variable implements the behavior of MATLAB control flow statements where a condition must be satisfied across all the elements of the evaluated arrays. If one of the element-wise operations is false, the scalar variable is set to zero to determine that the condition is not satisfied. Finally, the code generator prints a *if* statement using as condition, the scalar variable that was used to store the result of MATLAB *if-else* statement condition. Therefore, the C code of *if* statement scope is executed only if the scalar variable is true, namely only if all the element-wise operations of array condition are true.

The array conditional *while* statements are generated similarly to *if-else* statements. Firstly, the *while* statement is generated in C using the *true* value as condition in order to infinitely be executed. In the beginning of the *while*, the nested for-loops implementing

the MATLAB condition are generated storing the result in scalar variable as well. Subsequently, an *if* statement is generated that exits (break) the loop only if the MATLAB condition of *while* statement is not satisfied.

The listing 5.6 presents the generated C code of the MATLAB example in listing 5.5 showing the generation of the for-loop MATLAB statements. The outer MATLAB for-loop is composed by a colon operation with constant values while the inner for-loop includes a colon operation of variables with unknown values in compile time. The code generator produces the outer for-loop using the values 1,10 and 2 for the lower bound, the upper bound and the step of for-loop respectively. Generating the inner for-loop, a more complicated statement is produced examining in runtime the value of *x2* whether or not is negative. Depending on the signedness of the variable, the for-loop counter is checked to terminate the loop or continue the execution.

```
1  function [res] = for_loop_example(x1,x2,x3)
2    dec_int('x1','int32');
3    dec_int('x2','int32');
4    dec_int('x3','int32');
5    dec_dbl('res');
6    res=0;
7    for k= 1:2:10
8      for l=x1:x2:x3
9        res = res+k*l;
10     end
11   end
12 end
```

Listing 5.5: Code generation of for-loop example (MATLAB source).

```
1  void for_loop_example_1(D_DBL_TY &res, D_SINT32 x1,D_SINT32 x2,D_SINT32 x3){
2    D_SINT32 k;
3    D_SINT32 l;
4    res = 0;
5    for(k=1; k < 11; k=k+2){
6      for(l=(x1); ((((x2) < 0) && (l >=(x3))) || (((x2)>0) && (l <= (x3)))); l=l+(x2)){
7        res = res + k * l;
8      }
9    }
10 }
```

Listing 5.6: Code generation of for-loop example (C generated code).

The listing 5.8 presents the generated code of MATLAB source code in listing 5.7. The MATLAB code includes an array conditional *if-else* statement of three scopes (*if*, *if-else*, *else*). The generated C code in listing 5.8 contains the nested for-loops that perform the operations (comparing the equality of zero with the subtraction of *op1* and *op2* variables) of the *if* MATLAB condition. The generated for-loop iterates the elements of condition operands (16 elements) and is terminated if the variable *conditionB* is false,

namely the MATLAB array condition is false. Finally, an *if* statement is generated including the variable *conditionB*, as a condition.

```
function [res] = if_else_example(op1,op2)
  dec_dbl('op1',1,16);
  dec_dbl('op2',1,16);
  dec_dbl('res');

  if( (op1-op2) == 0)
    res=1;
  elseif op1<op2
    res=-1;
  else
    res=0;
  end
end
```

Listing 5.7: Code generation of if-else example (MATLAB source).

```
void if_else_example_1(D_DBL_TY &res, D_DBL_TY op1[16],D_DBL_TY op2[16]){
  D_SINT32 i1;
  D_SINT32 conditionB = 1;
  for(i1=0; i1 < 16 && conditionB; i1=_i1+1){
    conditionB = ((op1[_i1] - op2[_i1]) == 0);
  }
  if( conditionB ){
    res = 1;
  }
  else{
    conditionB = 1;
    for(i1=0; i1 < 16 && conditionB; i1=i1+1){
      conditionB = op1[i1] < op2[i1];
    }
    if( conditionB ){
      res = -1;
    }
    else{
      res = 0;
    }
  }
}
```

Listing 5.8: Code generation of if-else example (C generated code).

The listing 5.10 shows the generated code of the MATLAB example shown in listing 5.9. The MATLAB code includes an array conditional *while* statement of the *greater than* operation. The generated C code (listing 5.10) includes a for-loop that performs the relational operation assigning in *conditionB* the result. Next, an *if* statement is produced that examines the *conditionB* and terminates the *while* loop only if the condition is not satisfied.

```
function [res] = while_example(op)
  dec_dbl('op',1,16);
  dec_dbl('res');

  while( op > 0.1)
    op = op ./ 1.1;
  end
  res = op;
end
```

Listing 5.9: Code generation of while statement example (MATLAB source).

```
void while_example_1(D_DBL_TY &res, D_DBL_TY op[16]){
  D_SINT32 i1 ,i3;
  D_SINT32 conditionB;
  while(1){
    conditionB = 1;
    for(i1=0; i1 < 16 && conditionB; i1=i1+1){
      conditionB = (op[i1] > 0.1);
    }
    if( !conditionB )
      break;
    for(i3=0; i3 < 16; i3=i3+1){
      op[i3] = (op[i3] / 1.1);
    }
  }
  for(i1=0; i1 < 16; i1=i1+1){
    res = op[i1];
  }
}
```

Listing 5.10: Code generation of while statement example (C generated code).

5.3 Fixed Point Code Generation

The code generator produces fixed point operations generating additional C code for the handling of the fixed point arithmetic. For operations such as addition and subtraction where the fraction length of the two operands must be the same before the operation execution, shifting instructions are produced to adjust the fraction lengths in case they are different. Additionally, for the fixed point operations producing a result with a different fraction length than the declared variable's fraction length, shifting instructions are generated, too.

More specifically, the flow diagram of figure 13 presents the principle followed by the compiler to generate an addition of fixed point numbers (subtraction or bitwise logical operations). Firstly, the code generator produces extra C code to adjust the fraction length of the operands if they are not identical. In case that the operand's fraction

lengths are different, a left bitwise shift operation is generated to modify the scaling of the operand with the lower fraction length. Next, if the fraction length of the fixed point addition is different from the fraction length of the variable presenting the result, an extra shift operation is generated to adjust accordingly the scaling.

Concerning the fixed point multiplication (and division), the code generator produces code to modify the operation's intermediate fraction length to that of result's variable. The multiplication of two fixed point numbers gives their product with fraction length, the summary of the operand's fraction lengths. Thus, the compiler produces a shift operation to scale from the intermediate fraction length (sum of operand's fraction length's) to the fraction length of the result variable. Overflows are commonly appeared in fixed point arithmetic especially when the operations are implemented in embedded systems with hardware constraints such as small world length. Quantization operations (ex. rounding, ones-complement) may be applied by the hardware module units in fixed point arithmetic as well. Most of these architectures provide special instructions for the multiplication of integers performing scaling to reduce the precision of the result and prevent overflow. Therefore, the compiler considers that the underlying customized instructions that are mapped with the fixed point multiplications, return a result of half precision (fraction length).

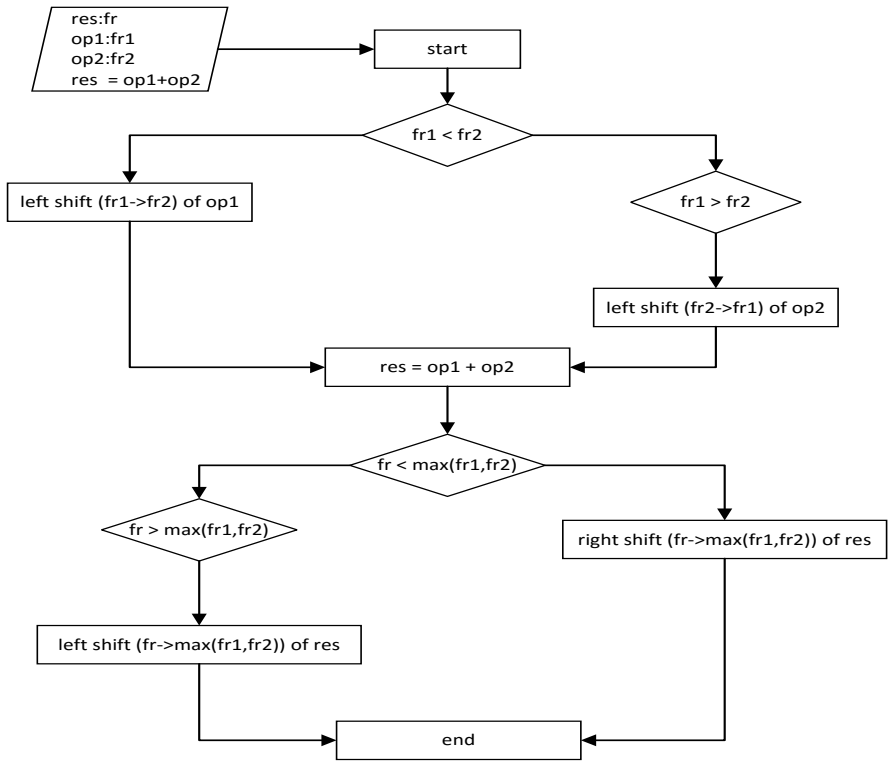

Figure 13: Control flow diagram for the generation of fixed point addition.

The listing 5.12 presents the generated code of the fixed point operations which are shown in MATLAB code of listing 5.11. The MATLAB code includes an addition and a multiplication of fixed point numbers with different fraction lengths. The generated code consists of the fixed point operations in C performing shifting operations to adjust the fraction lengths accordingly. The *op1* variable is shifted to left by 2 to be aligned with the fraction length of variable *op2*. Furthermore, the result of fixed point addition (fraction length 8) is shifted to right by 4 in order to be adjusted to the fraction length of the *res* variable. The fixed point multiplication is generated shifting to the right the intermediate result (fraction length of 10) by 6 bits in order to achieve 4 bits fractional precision, as the fraction length of *res* variable. The fixed point operations are printed via C macros encapsulating the appropriate instructions instead of producing directly C operators. The C macros are also generated by the compiler in a separate header file. For demonstration purposes of the example, the macros are shown in the beginning of listing 5.12.

```
function [res] = fixp_example(op1,op2)
  dec_fixp('op1','32','6');
  dec_fixp('op2','32','8');
  dec_fixp('res','32','4');
  res = op1+op2;
  res = op1*res;
end
```

Listing 5.11: Fixed point operations example (MATLAB source).

```

#define _rshift(op1,op2) (op1 >> op2)
#define _lshift(op1,op2) (op1 << op2)
#define _add_op(op1,op2) (op1 + op2)
#define _mul_op(op1,op2) (op1 * op2)

void fixp_example_1(_fixp_t &res, _fixp_t op1,_fixp_t op2){
  res=_rshift(_add_op(_lshift(op1,2),op2),4);
  res=_rshift(_mul_op(op1,res),6);
}
```

Listing 5.12: Fixed point operations example (C generated code).

5.4 Code Generation of Derived Data Types

The code generator prints the local C variables in the beginning of the function scope while global defined variables are printed in the C header file. Each MATLAB variable is represented by an equivalent variable in C retaining the shape that has been inferred (or declared) in type inference stage. The complex variables are represented with a

C structure data types including two variables of simple data types for the real and imaginary part of the complex number. The code generator utilizes the processor model to match the derived C data types with the target specific C data types of the current architecture. In case of type mismatch a default data name is printed according to that have been defined in the generated header file with the *#define* directives (ex. listing 5.3).

The Listing 5.15 presents the generated derived C data types of input code in listing 5.13 using the processor model shown in listing 5.14. The MATLAB code includes the declaration of five variables of different types and dimensions. The generated code in listing 5.15 uses the information of the specified derived data types of XML configuration (listing 5.14) and generates variables of target specific data types. The MATLAB variable *dbl_var* is a non-complex floating point scalar for which no description is existed in the processor model. Therefore, the default data name *D_DBL_TY* is produced. The *arr_var* and *arr_var_512_512* variables are complex floating point arrays of 2x3 and 512x512 dimensions respectively. The *arr_var* variable is matched with the *cfl_t* type and the *c_float* C data types is generated for the variable. The *arr_var_512_512* is matched with the C data type *cfl_512x512_t* due to dimensions specification and the *c_float_512x512* is generated for that variable. Finally, the *vector_var* is a packed variable matching with the derived type *cfl_vector* which describes a complex floating point vector of SIMD width 4.

```
function [] = derivedTypesExample()
  dec_dbl('dbl_var');
  dec_dblc('complex_var');
  dec_dblc('arr_var', 2,3);
  dec_dblc('arr_var_512_512', 512,512);
  dec_dblc_p('vector_var','4', 1,1024);
end
```

Listing 5.13: Derived C data types example (MATLAB source).

```
<PROCESSOR_MODEL>
  <type id="cfl_512x512_t" dt="double" dim="512" complex="true"></type>
  <type id="cfl_t" dt="double" complex="true"></type>
  <type id="vec_t" dt="double" complex="true"></type>

  <derived_type name="c_float_512x512" type="cfl_512x512_t" ></derived_type>
  <derived_type name="c_float" type="cfl_t" ></derived_type>
  <derived_type name="cfl_vector" SIMD_width="4" type="vec_t" ></derived_type>
</PROCESSOR_MODEL>
```

Listing 5.14: Derived C data types example (XML processor model)

```
1 void derivedTypesExample_1(){
2   c_float arr_var[3][2];
3   c_float_512x512 arr_var_512_512[512][512];
4   c_float complex_var;
5   D_DBL_TY dbl_var;
6   cfl_vector vector_var[256];
7 }
```

Listing 5.15: Derived C data types example (C generated code).

5.5 Code Generation For MATLAB Operations - customized instructions

The code generator prints the intrinsic functions that have been matched at the instruction selection stage while equivalent C code is produced for the MATLAB operations that haven't been mapped with any function unit of the target processor/ASIP. During code generation stage, the specialized instructions that have been assigned at the AST nodes, are printed using as parameters the operands of the MATLAB operation. For the assignments that haven't been mapped with customized instructions, C code is generated that implements the MATLAB operation. The compiler supports all the MATLAB operations and it is able to produce code for any operation of any data type. For the majority of element-wise MATLAB operations with non-complex numbers a direct translation of MATLAB to C take place. However, additional C code is produced for several MATLAB operations such as element-wise operations of complex number, array multiplication or transpose operation. For the element-wise operations of complex numbers two C assignments are produced calculating the real and imaginary part of the complex number. For array multiplication MATLAB statements, a segment of C code including three nested loops is generated to calculate the matrix product. Finally, the statements including a transpose operation are translated to a loop-nested code inversing the data of the input matrix.

The listing 5.18 presents the generated C code of MATLAB example in listing 5.16 using the processor model described in listing 5.17. The first statement of MATLAB code (line 8) includes a floating point multiplication of complex numbers which is matched with the customized instruction *cmul_f* in the processor model. The second and third assignments of MATLAB code are integers' multiplication of non-complex and complex numbers respectively which aren't mapped with any instruction in the XML. The code generator produces the customized instruction *cmul_f* for the first MATLAB assignment using its operands as intrinsic's parameters. For the second and third MATLAB statements, C code is produced to calculate the multiplication operations. For the complex multiplication, two statements are printed. The first calculates the real part of the com-

plex multiplication and the second generated statement calculates the imaginary part.

```
function [] = customizedInstructionsExample()
  dec_dblc('op_f1');
  dec_dblc('op_f2');
  dec_int('op_i1','int32');
  dec_int('op_i2','int32');
  dec_intc('op_ic1','int32');
  dec_intc('op_ic2','int32');
  res_f = op_f1.*op_f2;
  res_i = op_i1.*op_i2;
  res_ic = op_ic1.*op_ic2;
end

```

Listing 5.16: Customized instructions - MATLAB operations example (MATLAB source).

```
<PROCESSOR_MODEL>
<type id="cfl_t" dt="double" complex="true"></type>
<type id="cint_t" dt="int" complex="true"></type>
<derived_type name="c_float" type="cfl_t" ></derived_type>
<derived_type name="c_int" type="cint_t" ></derived_type>

<instruction name="cmul_f" type="SCALAR" op="*" op_types="cfl_t,cfl_t"></instruction>
</PROCESSOR_MODEL>
```

Listing 5.17: Customized instructions - MATLAB operations example (XML processor model)

```
void customizedInstructionsExample_1(){
  c_float op_f1;
  c_float op_f2;
  D_SINT32 op_i1;
  D_SINT32 op_i2;
  c_int op_ic1;
  c_int op_ic2;
  c_float res_f;
  D_SINT32 res_i;
  c_int res_ic;
  res_f = cmul_f(op_f1, op_f2);
  res_i = op_i1 * op_i2;
  res_ic.re = ((op_ic1.re * op_ic2.re) - (op_ic1.im * op_ic2.im));
  res_ic.im = ((op_ic1.im * op_ic2.re) + (op_ic1.re * op_ic2.im));
}
```

Listing 5.18: Customized instructions - MATLAB operations example (C generated code).

5.6 Scalarized Code Generation

The code generator produces scalarized C code for the MATLAB statements that are not included in SIMD blocks. The MATLAB array statements are translated to C loop nests. For each statement of the input MATLAB code, a loop nest is generated with a same number of loops as the number of dimensions of expressions/variables which are involved in the statement. For statements that include references with different indexing additional counters are generated to access the operand arrays. More specifically, for references that include index of colon operation with non-unit stride, a new counter is produced to traverse the array by the specified colon step. In case that two references access different range of elements, then C code is generated adding the starting point of MATLAB indexing in the corresponding C counter. Finally, the flattened MATLAB indexes (MATLAB reference with missing subscripts) are translated accordingly to C code that flattens the multidimensional derived arrays.

The listing 5.20 presents the generated code of source code shown in listing 5.19. The MATLAB code includes two functions demonstrating array statements of one dimension with different indexing and three-dimensional array statements of flattened index. The code generator produces the MATLAB statement of line 5 (listing 5.19) printing a for-loop with counter *i1* that accesses the elements of *res1* array. Two additional counters (*i3* and *i4*) are generated to traverse the references *op1* and *op2* which consist of different indexing. The *i3* counter is initialized to 17 as the MATLAB colon operation indicates while the counter *i4* is increased by 2 corresponding to the MATLAB indexing. Moreover, the MATLAB statement of line 6 is generated by the code generator printing a for-loop and only one counter. The counter is used in both references although their MATLAB indexing is different. The MATLAB reference *op1* accesses its last 16 elements, thus the C counter is incremented by 17. Concerning the code generation of second MATLAB function, the compiler produces two nested loops, one for each MATLAB subscript. The multidimensional derived C variables are transformed to flattened arrays using pointer notation and printing only one subscript. The C index includes the translated first subscript of MATLAB code while the second one is multiplied by the first dimension of the array.

```
function [res1 res2] = scalarized_example(op1,op2)
  dec_dbl('op1',1,32);
  dec_dbl('op2',1,32);
  dec_dbl('res1',1,16);
  res1 = op1(17:32)+op2(1:2:32);
  res2 = op1(17:32);
end

function [res_3d] = scalarized_3d_example(arr_3d,ind)
  dec_dbl('arr_3d',8,16,32);
  dec_dbl('res_3d',8,16,32);
  dec_int('ind','int32',1,16);
  res_3d(4:end,ind) = arr_3d(4:end,ind);
end
```

Listing 5.19: Scalarized MATLAB code example (MATLAB source).

```
void scalarized_example_2(D_DBL_TY res1[16],D_DBL_TY res2[16], D_DBL_TY op1[32],D_DBL_TY op2[32]){
  D_SINT32 i3 ,i4 ,i1;
  i3 = 17;
  i4 = 1;
  for(i1=0; i1 < 16; i1=i1+1){
    res1[i1] = op1[(i3-1)] + op2[(i4-1)];
    i3=i3+1;
    i4=i4+2;
  }
  for(i1=0; i1 < 16; i1=i1+1){
    res2[i1] = op1[(i1+17-1)];
  }
}
void scalarized_3d_example_3(D_DBL_TY res_3d[32][16][8], D_DBL_TY arr_3d[32][16][8],D_SINT32 ind
     [16]){
  D_SINT32 i1 ,i4 ,i2;
  for(i2=0; i2 < 16; i2=i2+1){
    for(i1=4; i1 < 9; i1=i1+1){
      (**res_3d)[(i1-1) +(ind[i2]-1)*8] = (**arr_3d)[(i1-1) +(ind[i2]-1)*8];
    }
  }
}
```

Listing 5.20: Scalarized MATLAB code example (C generated code).

5.7 Code Generation of Packing/Unpacking Statements

The code generator prints packing/unpacking instructions according to the statements that have been introduced in the AST during the packing/unpacking introduction stage. During code generation, the processor model is utilized to retrieve the name of the intrinsic function corresponding to the packing/unpacking instruction type (ex. expand, packing memory by pointer) and the type (floating or fixed point) of the variable which

is packed/unpacked. To achieve this, an implementation of the instruction selection algorithm discussed in section 4.3 is deployed to match the packing/unpacking instruction and its types with an available instruction of the target architecture.

The *expand* instruction is printed with an intrinsic function that is assigned to a vector variable and accepts the value which fill the vector as input parameter. The packing memory by pointer instruction is produced similarly to the *expand* instruction, printing the corresponding intrinsic function. The code generator follows the prototype of packing commands of common vectorized architectures which composed by two input parameters returning the filled vector. The first input parameter is a pointer to the vector which is filled with the unpacked elements and the second parameter is a pointer to the memory of the first data that is inserted in the vector. Finally, the packing instruction inserting a scalar in a vector, is generated in a nested for-loop that repeatedly inserts the unpacked data to the vector. The intrinsic function composed by the input parameters of

- The vector variable.
- The vector's index where the data is stored (index from 0 to SIMD width).
- The value that is stored in the vector.

The intrinsic function returns the vector including the inserting value. The sequence below shows the for-loop that are generated for the iterative packing/unpacking statements in combination with the loop that correspond to the SIMD block.

$$\sum_{n=1}^{n} SD(n) \sum_{m=n*s}^{n*s+SD*s} s(m)$$

The nested loops pack or unpack the elements of an array SIMD operand in the following manner: the inner loop inserts *SIMD width* number of data in the vector while the SIMD block for-loop iterates the next *SIMD width* elements of the array reference that is packed/unpacked. More detailed, the outer sequence is a for-loop with upper bound (n) which is the array size of the SIMD block operands, while the for-loop step is the SIMD width (SD) of the vector instructions. The inner sequence represent the for-loops that are generated to iteratively insert/extract data to/from vectors. The lower bound of the for-loop is the current iteration of outer loop (n), multiplied by the stride (s) (index) of reference that is packed/unpacked. The upper bound is equal to the lower bound plus the product of stride (s) and the SIMD width (SD). Finally, the step of the inner loop is the stride of the reference index. The resulted generated loops are similar to the scalarizing of array MATLAB operations and subsequently performing loop strip-mining on the scalarized for-loops.

The listing 5.21 presents the generated code using as input the code of listing 4.7 in chapter 4. The functions in lines 6 and 8 insert and extract four scalars to/from the vector instantly. The *insert_all_ic_4* function in line 9 fills the vector *vec_2* with the first element of *op* variable. The two last packing/unpacking instructions of the example insert and extract iteratively data to/from vectors generating for-loops. The MATLAB

references involved in the packing/unpacking, contain indexing with colon operation of non-unit stride. Therefore, the generated for-loops composed by a lower bound of the SIMD block for-loop counter (*si0*) multiplied by two (the step of colon operations). The upper bound of generated loops allows four repetitions (as the SIMD width) and the increment of loop is 2 as the stride of the reference index that is packed/unpacked.

```
1  void packingExample_1(cfixp_t res1[8],cfixp_t res2[16], cfixp_t op[16]){
2    cfixp32x4_t vec_0, vec_1, vec_3, vec_4;
3    D_SINT32 i2 ,i3 ,i1;
4    D_SINT32 si0;
5    for(si0=0; si0 < 8; si0=si0+SW){
6      vec_0 = insert_all_ptr_ic_4(vec_0, &((op)[si0]));
7      vec_1 = v4_lsl_i(vec_0, 2);
8      extract_all_ptr_ic_4(&((res1)[si0]), vec_1);
9      vec_2 = insert_all_ic_4(op[0]);
10     for(i3=(si0*2 + 1); i3 < (si0*2 + 1)+SW*2; i3=i3+2){
11       vec_3 = insert_ic_4(vec_3, ((i3-(si0*2 + 1))/2), op[i3-1]);
12     }
13     vec_4=v4_add_i(vec_2,vec_3);
14     for(i1=(si0*2 + 1); i1 < (si0*2 + 1)+SW*2; i1=i1+2){
15       res2[i1-1] = _extract_ic_4(((i1-(si0*2 + 1))/2), vec_4);
16     }
17   }
18 }
```

Listing 5.21: Packing/Unpacking code generation example (C generated code).

5.8 Vectorized Code Generation

The generation of vectorized code differentiates from the production of scalarized code. Furthermore, the generation of vectorized code presented in the section is a novel methodology for the production of C code with vector semantics from the MATLAB array-syntax form and it constitutes a contribution of the thesis. The code generator translates the MATLAB array statements included in SIMD blocks to vectorized C code. SIMD code generation incorporates the production of for-loops that correspond to SIMD blocks. The for-loops composed by a string literal "SW" for-loop step and the array size of the SIMD block operands for-loop condition. Instead of "SW", the constant width of the SIMD block could be used as well. However, the use of SIMD block width would lead to a more restrictive and non-portable (to other architectures) code. By using "SW", the compiler's user is able to execute the generated code with his preferred SIMD width using, a *#define* directive in C to specify the value of "SW". The MATLAB variables which have been declared with packed format and they are SIMD operands, are generated as follows:

- Firstly, the index of a MATLAB reference is translated to C. This comprises the generation of C code for each subscript of the reference. SIMD operands that are eligible for SIMD code generation are constrained to be composed by only one subscript of vector dimension. However, more subscripts are allowed to represent a MATLAB reference that is vectorized with the restriction that the additional subscripts are scalar indexing. Finally, the subscript that is vectorized, is always a colon operation or index represented by other reference/expression. Different kinds of subscript expressions with vector dimension don't ensure that access the data of SIMD operand sequentially and re-packing of data would have been introduced in the previous compilation stage.
- Next, flattening of index is performed in case that the MATLAB variable is a multidimensional array. The index flattening comprises the generation of only one C array subscript which is represented by the summary of the translated MATLAB subscripts, multiplied accordingly by the variable dimensions.
- Finally, the generated C index is added with the counter of the for-loop that corresponds to SIMD block and subsequently is divided by the current SIMD width. The addition of the SIMD for-loop counter allows the accessing of the next *SIMD width* elements at each loop iteration. The division of the generated indexing by the SIMD width ("*SW*" value) maps the translated indexing to the C vector structure. Avoiding the division, the derived code of index would point to the corresponding element of a C array consisting of scalar elements. The C vector structures that are generated for the packed variables are arrays of vectors. Therefore, dividing the C index by the SIMD width, it is scaled down to the corresponding vector element of the derived C vector-typed array.
- In case that a MATLAB reference doesn't include indexing, the generated SIMD indexing consists of only the counter of for-loop divided by the SIMD width value.
- Finally, no indexing is generated for packed variables whose dimensions coincide with the SIMD width (ex. declaring an 1x4 MATLAB variable which is packed to vector of 4 width).

Listing 5.23 shows the generated SIMD code using as input MATLAB code, the example is shown in listing 5.22. The MATLAB code includes an addition and a multiplication in SIMD block which are mapped to *v4_add_f_rr* and *v4_mul_f_rr* respectively. The SIMD operands are variables of two and three dimensions. The code of listing 5.23 shows the vectorized code consisting of the generated for-loop that is corresponded to SIMD block and the for-loop of the MATLAB code in line 8. The C expressions of flattened index are added by the counter of the SIMD block for-loop *s0* and then they are divided by the vector's width *SW*. The generated C variable which corresponds to the MATLAB packed variable is an array of vectors where each vector stores N elements (where N the vector's width 4). Dividing the translated (from MATLAB to C) index by the vector width the real index of C array vector is computed.

```
function [res res_3d] = vectorized_example(in, arr_3d)
  dec_dbl_p('in','4',16,8);
  dec_dbl_p('res','4',16,8);
  dec_dbl_p('arr_3d','4',4,8,16);
  dec_dbl_p('res_3d','4',4,8,16);

  startSIMD(4);
  for k=1:size(in,1)
    res(:,k) = in(:,k) +in(:,k+4);
    res_3d(2,2,:) = arr_3d(2,4,:).*arr_3d(2,8,:);
  end
  endSIMD();
  end

```

Listing 5.22: Vectorized MATLAB code example (MATLAB source).

```
void vectorized_example_1(float32x4_t res[128/SW],float32x4_t res_3d[512/SW], float32x4_t in[128/SW
    ],float32x4_t arr_3d[512/SW]){
  D_SINT32 k;
  D_SINT32 si0;
  for(si0=0; si0 < 16; si0=si0+SW){
    for(k=1; k < 17; k=k+1){
      res[(si0+(k-1)*16)/SW] = v4_add_f_rr(in[(si0+(k-1)*16)/SW], in[(si0+(k + 3)*16)/SW]);
      res_3d[(si0+(1*128+1*16))/SW] = v4_mul_f_rr(arr_3d[(si0+(1*128+3*16))/SW], arr_3d[(si0
          +(1*128+7*16))/SW]);
    }
  }
}
```

Listing 5.23: Vectorized MATLAB code example (C generated code).

5.9 Code Generation statistics of benchmark

The table 16 shows statistics of the benchmark's generated code producing scalarized and vectorized C code. The first two rows of table show the scalar and SIMD generated instructions. The row *derived C data types* presents the different data types that are generated for the target architecture. The *packing/unpacking operations* show the operations generated for the insertion/extraction of data to/from vectors producing vectorized C code. The *fixed-point shifting operations* enumerate the shifting operations that are generated to handle fixed-point arithmetic. The *MATLAB operations* row shows the generated statements for the MATLAB operations that haven't been mapped with any customized instructions when producing scalarized code. The *SIMD block for-loops* row presents the C for-loops correspond to the SIMD blocks while the *scalarized generated for-loops* enumerate the generated for-loops producing scalarized code. Finally, the last

three rows of the table present the generated *if-else statements*, the *for-loops statements* and the *user defined functions* respectively.

	cfo	cordic	fft32	fft64-v1	fft64-v2	fft128	fir	mean	qr-dec
custom instructions	2	2	12	24	56	40		2	57
SIMD custom instructions	7	10	12	7	13	16	2	1	49
derived C data types	2	1	1	1	1	1	1	1	2
packing/unpacking operations	8	9	19	13	13	27	6	3	68
fixed-point shifting operations	7	5					1		5
MATLAB operations	6	10					2		10
SIMD block for-loops	1	4	1	2	1	5	1	1	12
scalarized generated for-loops	8	13	11	9	25	23	3		75
if-else statements	1								
for-loop statements		3	3	5	2	9	2	1	12
user defined functions	1		1	3	3	4	1	1	3

Table 16: Generated code statistics of benchmark.

5.10 Conclusion

In this chapter, the compiler's back-end is presented. The back-end consists of the code generator producing vectorized or scalarized ANSI C code. The vectorized code generation includes the production of for-loops that correspond to SIMD blocks as well as the generation of packing instructions and the matched SIMD instructions with vector semantics. The scalarized code generation translates the array MATLAB statements to equivalent C nested loops. The compiler is able to produce the MATLAB operations via the matched customized instructions while equivalent C code is produced for the

MATLAB operations that haven't been mapped in instruction selection stage. Furthermore, for the generation of fixed point operations additional C code is generated for handling the fixed point arithmetic. Finally, the compiler generates C code with any of the primitive MATLAB data types including floating point, fixed point or integers.

Chapter 6

Evaluation of the Compiler

In previous chapters a MATLAB to C vectorizing compiler was presented. In this chapter the performance of the generated code by the MATLAB compiler is evaluated. The generated code by the compiler is compared against the generated code by MathWorks Coder using eight representative DSP benchmarks on various target processors. The set of target architectures, where the generated C code has been mapped, consists of an ASIP named BoT supporting SIMD processing, an ASIP named tinyBoT providing scalar custom instructions, two computer boards of ARM architecture and two different desktops computers of x86 architecture.

Additional experiments are presented in the chapter regarding the performance of the generated code and the performance of auto-vectorizing C compilers. The experiments include evaluation of the generated code by the compiler on the different architectures without the use of any custom instructions. Additionally, the auto-vectorization performance applied by Clang/LLVM, GCC and MSVC is examined on the scalarized generated code by the two MATLAB compilers. Furthermore, a comparison between the C compilers on the generated code is also presented. At the end of the chapter, the compilation times of the two MATLAB compilers are discussed briefly, as well.

6.1 Executive Summary

In this section an executive summary of the performance of the generated code by the compiler is presented. The section is indicated for readers who are interest for a brief discussion of the compiler's experimental results avoiding the detailed result's presentation which is followed in the below sections. The compiler has been evaluated using eight representative DSP algorithms (which are enumerated in section 1.2) concerning different input stream sizes. The benchmarks have been mapped on:

- The raspberry PI 2 and PI 3 single-board computers (SBC) [Raspberry Pi, 2016] which include a quad-core ARM Cortex-A7 (ARMv7-A) processor and a 64-bit quad-core Cortex-A53 (ARMv8-A) processor correspondingly.
- Two different x86-based desktops, one of them including an Intel Core i7-3820 from Sandy Bridge micro-architecture processor, while the second one includes an Intel (Core i7-3770) from Ivy Bridge micro-architecture processor.

- Two different ASIP processors (one of them supporting SIMD) customized for baseband signal processing.

The benchmarking have been conducted using both floating point and fixed point data types on ARM and x86 architectures while for ASIPs only fixed point data types has been used since no floating point arithmetic is supported on the specific ASIPs. Moreover, all experiments relating to the generation of vectorized code have been carried out using different configurations concerning the SIMD processing width where widths of 4 and 8 elements have been applied as well as the form of data between unpacked and packed data types. However, the FFT algorithms have been evaluated using only SIMD width 4 in order to limit the data shuffling appeared in that application. The performance (speed) of the generated C code has been compared to that of the code generated by the MathWorks coder. The following subsections show a summary of the speed-up compared against the MathWorks coder generated code on the above targets with different configurations concerning the SIMD width and the form of data (packed vs unpacked).

6.1.1 Executive Summary of Performance on the ARM Architectures

Figures 14 and 15 presents the minimum, maximum and average speed-up of the generated code by the compiler against the MathWorks Coder generated code on a Raspberry PI 3 using fixed point data types (fig. 14) and floating point data types (fig. 15). For this experiment the Raspberry computer board uses Rasbian [Rasbian, 2016] operating system and Clang/LLVM has been used for the compilation of generated C code by both MATLAB compilers. The experiments with floating point has been conducted only using SIMD width 4 because the NEON technology of ARM architecture doesn't provide SIMD width of 8 for floating point.

The compiler's generated code achieves a substantial speed-up among the different SIMD configurations for both data types. More specifically, the benchmarks with fixed point data types achieve a speed-up up to 106.6x and 54.1x and an average speed-up of 12.8 and 9x for packed data types with SIMD width 4 and 8 respectively while the speed-up for unpacked data types is up to 46.8x and 24.5x with average speed-up of 8.2x and 7.3x. Additionally, the speed-up of floating point benchmark for the current experiment is up to 13.3x and 15.6x (4.6x and 3.9x on average) for packed and unpacked data types with SIMD width 4.

The fixed point benchmarks achieve a higher speed-up compared to the floating point benchmarks because the generated code produced by MathWorks Coder is more complicated than the corresponding generated code for floating point data types. The fixed point generated code includes shift operations for fixed point arithmetic or overflow handling operations which slow up the execution performance of the generated code

compared to the vectorized code by the compiler. Furthermore, in the experiment with fixed point data types the versions with SIMD width 4 achieve a higher maximum and average speed-up than the SIMD width 8 versions due to the impressive performance of FFTs which are evaluated only with SIMD width 4.

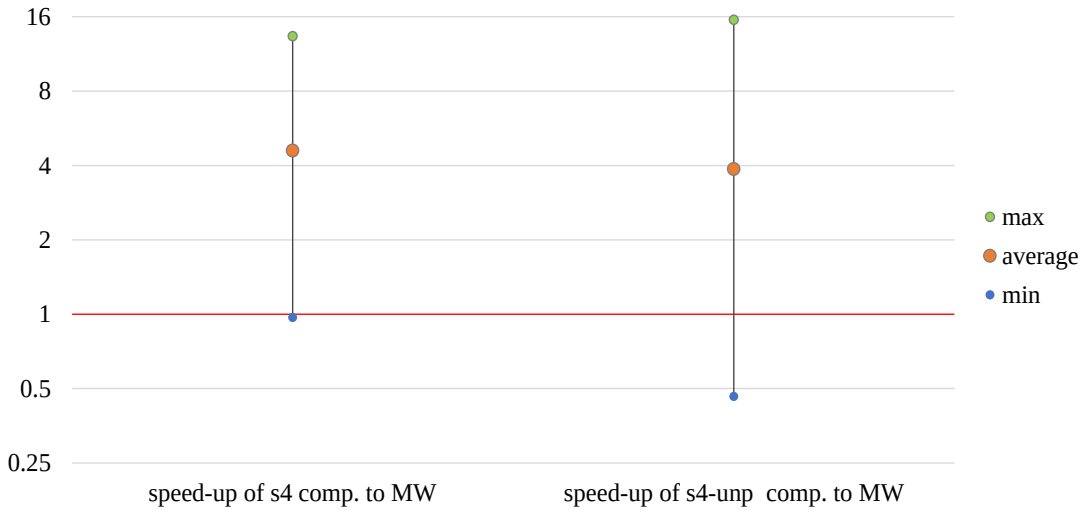

Figure 14: Speed-up of fixed point generated code on Raspberry PI 3

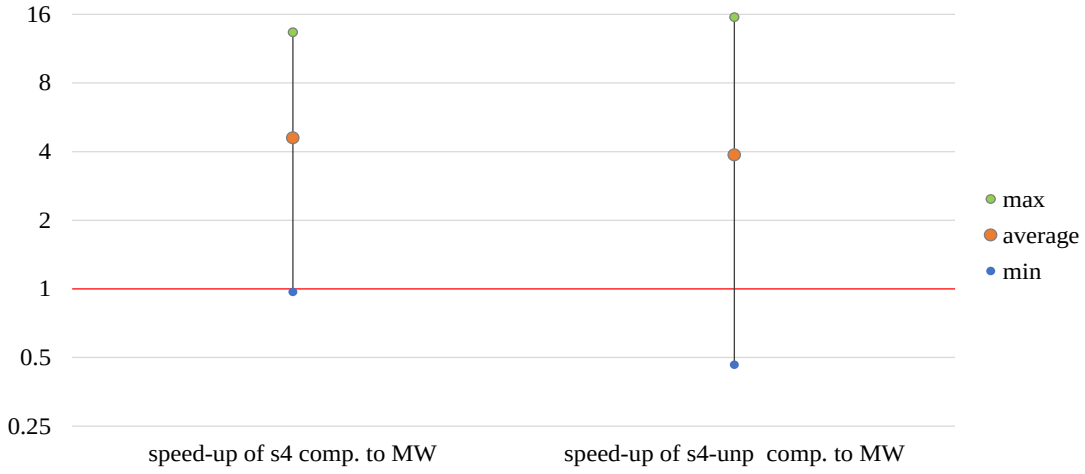

Figure 15: Speed-up of floating point generated code on Raspberry PI 3

6.1.2 Executive Summary of Performance on the x86 Architectures

Figures 16 and 17 presents the minimum, maximum and average speed-up of the generated code by the compiler compared to the generated code by the MathWorks Coder on a Core i7-3770 desktop using fixed point data types (fig. 16) and floating point data types (fig. 17). For the experiment, Linux Ubuntu [Linux, 2016] operating system have been used and the generated C code has been compiled with Clang/LLVM.

The fixed point generated code by the compiler achieves a speed-up up to 270.7x and 7.5x with 19.3x and 2.9x on average for SIMD width of 4 and 8 with packed data types. The speed-up achieved for unpacked fixed point data types with SIMD width 4 and 8 is up to 66.1x and 43.2x with an average speed-up of 8.4x and 8.3x. The speed-up achieved for the floating point generated code is up to 4.7x and 3.9x with average speed-up of 1.7x and 1.8x for SIMD width 4 and 8 using packed data types. The unpacked floating point data type benchmarks achieve a speed-up up to 8.6x and 8.2x with average speed-up of 1.8x and 1.9x.

Similarly to the Raspberry results, the fixed point benchmark achieve a substantial higher speed-up due to the fixed point arithmetic generated by MathWorks compiler. Finally, comparisons between different SIMD width configuration shouldn't be considered because the SIMD width 8 version doesn't include the FFT benchmarks which effect the presented summary speed-ups.

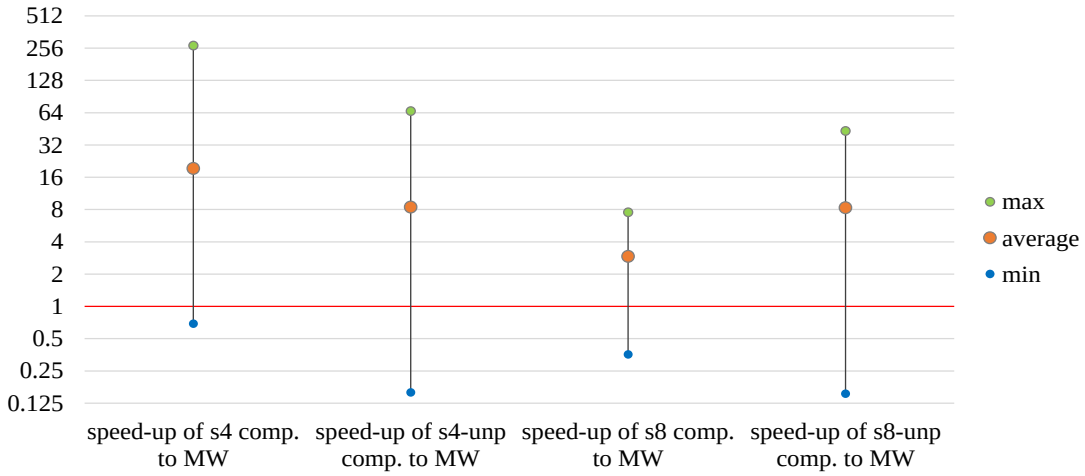

Figure 16: Speed-up of fixed point generated code on Core i7-3770

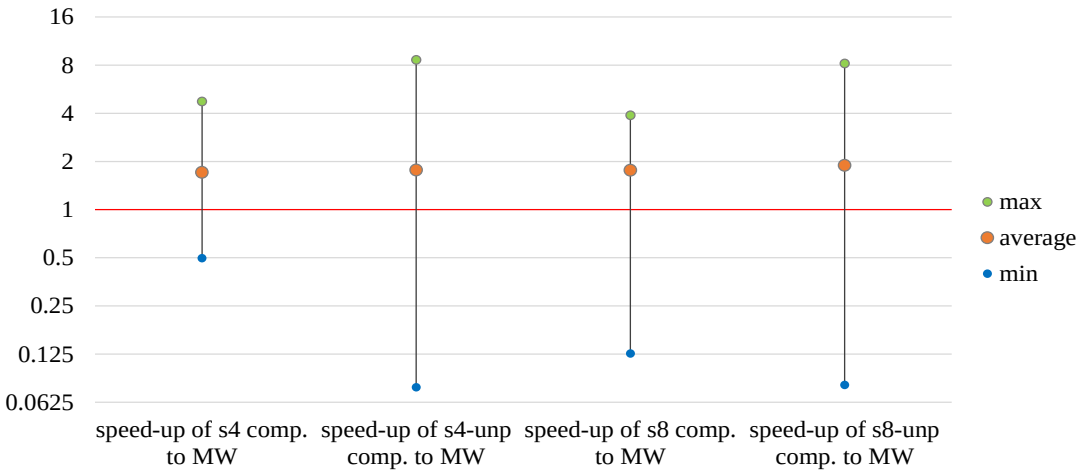

Figure 17: Speed-up of floating point generated code on Core i7-3770

6.1.3 Executive Summary of Performance on Application Specific Instruction Set Processors

Figure 18 shows the minimum, maximum and average speed-up of the compiler's generated code against the MathWorks Coder generated code on the two ASIPs. The C application codes generated by both compilers have been mapped to the ASIPs using the compiler of the Synopsys ASIP Designer [ASIP Designer, 2016]. The values which are lower than 1 represent a negative speed-up (slow up). The *'speed-up of scalarized code with intr. comp. to MW'* values refers to the speed-up achieved on the ASIP with scalar customized instructions. The rest data values of the figure present the speed-up values of the generated code on ASIP supporting SIMD processing with different configurations. More specifically, the speed-ups including the *'s4'* (in horizontal axis) indicate that SIMD width 4 has been used (the *'s8'* refers to SIMD width is 8 respectively) and the speed-ups including the *'unp'* refer to the compilation of the MATLAB code using unpacked data.

The generated code by the compiler achieves a speed-up between 2x-74x and 8.3x on average compared to the generated code by the MathWorks coder for the ASIP with scalar customized instructions. Furthermore, the speed-up achieved on the ASIP with SIMD processing is up to 39x and 41.3x (10.1x and 14.3x on average) using SIMD width 4 and 8 respectively while the speed-up on the same ASIP using unpacked data types is up to 81.8x and 97.1x with average values of 10.1x and 17.4x using SIMD width 4 and 8 respectively.

The generated code by the compiler achieves a better execution performance than the MathWorks generated code on the two ASIPs because custom instructions such as complex number operations and mathematical functions have been mapped on specialized hardware in contrast to the code generated by MathWorks coder that does not include custom instructions. Moreover, the speed-ups (of different configurations) on ASIP with

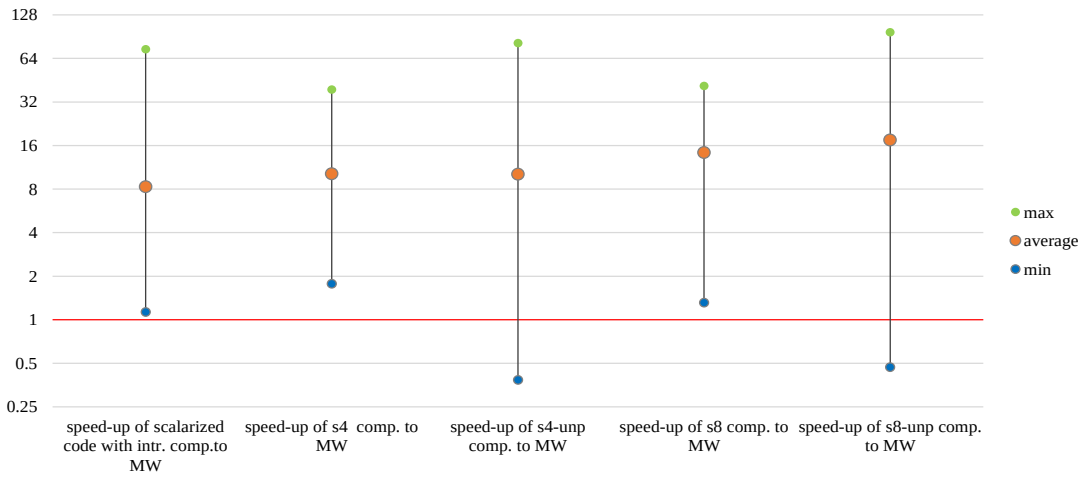

Figure 18: Speed-up of generated code on ASIPs

SIMD processing are in general higher than the speed-up achieved on ASIP supporting scalar processing due to the parallel computing which is applied on the ASIP with the SIMD instructions.

6.2 Experimental Setup

The compiler has been used to generate code for a number of application benchmarks for the target processors. MathWorks coder 2.8 [MathWorks Coder, 2016] has also been used to generate code for the application benchmarks for the targeted processors to allow comparisons. In the case of fixed point application codes, the initial floating point MATLAB code have been automatically modified with fixed Point Designer [MATLAB fi, 2016] to add fixed point behavior in the source code using fi objects [MATLAB fi, 2016]. The compilation of applications source code has been on a MS Windows 10, Intel i-Core7 and 8 GB RAM machine.

The benchmarks include five examples specifically from the signal processing domain and three more general mathematical/ statistical examples that are also popularly used in signal processing. More specifically the benchmark consists of:

- The FFT algorithm is used to calculate the DFT [Burrus and Parks, 1991] used in several wireless baseband algorithms such as IEEE-802.11 [IEEE-802.11, 2009] and 3GPP-LTE [3GPP-LTE, 2016] standards. Often, for these algorithms to adapt to different bandwidth conditions requires using a different FFT size.
- Carrier Frequency offset (CFO) [Stuber et al., 2004] is similarly a requirement in almost all OFDM/A based communication systems.

- The digital finite-impulse-response (FIR) filter is among the most popular choice for band-limiting signals and in general signal conditioning.
- The average computation algorithm (mean) is perhaps the most commonly used statistical parameter and it is required to be computed as a part of various algorithms including wireless communication and signal processing.
- CORDIC algorithm [Andraka, 1998] has been used as an effective computational approximation to trigonometric operations especially when using complex numbers. They are vastly deployed on systems implemented in hardware [Andraka, 1998], routinely used as a software subroutine in several algorithms.
- QR decomposition is a popular Matrix inversion technique that has wide applicability in many signal processing and learning algorithms [Anderson et al., 1992].

The compiler has been evaluated against MathWorks Coder on multiple scenarios concerning the benchmarks' input stream sizes, data types and algorithm's implementation. The benchmarks' sizes arise from their application at DSP domain while different input sizes have been selected to evaluate the performance of generated code relating to the volume of data which are processed. More specifically, the CFO algorithm has been tested with a signal stream of 32, 64 and 128 samples. The mean has been evaluated with input array sizes of 32, 64, 128 and 1024. The FIR has been evaluated using 32 input streams with 64,128 and 256 lengths each for processors while only input matrix of 32x256 has been evaluated on ASIPs. The CORDIC application has been also used for benchmarking (assuming 64 samples as input). QR decomposition has been used assuming 32, 64 and 128 arrays of 2x2 size each as input for processors. For the two ASIPs, QR decomposition has been examined using only the version with a 3 dimensional array of 2x2x32 dimensions. Three different algorithms of FFT have been also evaluated. More specifically, an FFT consisted by 3 stages of radix-2 with 32 input samples, an FFT of 3 radix-4 stages with 64 input samples and a mixed radix (a radix-8 followed by 2 radix-4 stages) FFT with 128 input samples have been used. For benchmarking FFT-64 two different implementations of the algorithm have been evaluated - a standard one (FFT64-v1) and an optimized one (FFT64-v2) for data locality. Finally, the benchmarks have been evaluated using floating point and fixed point types for the ARM and x86 architectures. For benchmarking on ASIPs only fixed point was used since the specific ASIPs don't support floating point arithmetic.

6.2.1 Experimental Environment and Configurations

The C application codes generated by both the compiler and the MathWorks Coder have been mapped to the targeted ASIPs using the compiler of the Synopsys ASIP Designer [ASIP Designer, 2016]. The tool was also used for the simulation of the two ASIPs and the execution of the benchmarks reporting the cycle counts of the executed code. For benchmarking of the generated code on general purpose processors, a variety

of C compilers and operating systems have been used. Table 17 presents the different experimental setups that have been used for the targeted processors. Although in ASIPs the cycle counts can be easily exported, the cycle counting of processors on operating systems is might be impossible at several instances. Thus, the benchmarking of the processors has been conducted by counting the execution time of each benchmark using the *clock* C function. Using an operating system command such as *time* at linux OS, it could affect the results due to the fact that the time required for the loading of the executable program by the operating system spends a large amount of time regarding the execution time of the application. Furthermore, due to the small execution times of some benchmarks and execution time deviations which have been observed on these environments, each benchmark have been executed 5 times and each of the executions includes 10000 execution repetitions of the algorithm (in order to achieve time that is counted by *clock* C function). The average execution time of each benchmark has been calculated and has been implemented for the results shown in this chapter.

Architecture	**OS**	**C compiler**
ARM	Rasbian	Clang/LLVM
ARM	Rasbian	GCC
ARM	Windows IoT	MSVC (Visual Studio)
x64	Linux Ubuntu	Clang/LLVM
x64	Linux Ubuntu	GCC
x64	Windows 10	MSVC (Visual Studio)

Table 17: Different experimental setups on CPU architectures

All experiments have been carried out using SIMD processing widths of 4 and 8 on unpacked and packed data types except FFT algorithms, where only SIMD processing width of 4 has been explored. Using SIMD processing width of 8 on FFT is not meaningful since more data shuffling would be needed thus dramatically decreasing performance. The floating point benchmark on ARM architecture has been evaluated only by using SIMD width value of 4, because the NEON extension doesn't provide instructions of floating point vectors with SIMD width 8. Moreover, due to the incompatibility of performance value ranges, the experimental results referring to performance have been normalized per each benchmark and input size. The related to performance result tables describe the cycles/ execution times that correspond to a performance value of 1 for each benchmark. The header of each column describes the experimental scenario and the corresponding figure.

6.2.2 Abbreviations of Diagrams and Tables

The subsection presents the abbreviations appeared in diagrams of current chapter and appendix and discusses their meaning. The abbreviations concern the configuration of the experiment as to data types, SIMD configuration, MATLAB compiler, C compiler and targeted processor. The abbreviations of diagrams are:

- *MW* (or *MathWorks*) refers to the generated code by MathWorks Coder.
- *MC* refers to the scalarized generated code by compiler without using customized instructions of hardware.
- *fx* indicates that fixed point data types have been used.
- *fl* is similar to *fx* but for floating point data types.
- *s4* is mentioned when SIMD width of 4 has been used for the generation of vectorized C code.
- *s8* is similar to *s4* for SIMD width of 8.
- *unp* is used when unpacked data types have been used for compilation of MATLAB code.
- *Clang*, *GCC* and *MSVC* are abbreviations to indicate the C compiler which has been used for the compilation of the generated C code.
- *PI2*, *PI3*, *i7-3770* and *i7-3820* are abbreviations to specify the target processor where the generated C code has been executed.

6.2.3 Benchmark Characteristics

The selection of applications used as benchmarks has been made focusing on significant code properties that are important to evaluate. Benchmarks include a variety of different codes with respect to data dependencies, control flow, array dimensions, diversity of indexing instances and SIMD operations (ex. scalar or constant operands). In general, the selected applications use the majority of instructions supported by the targeted architectures with a large variety regarding indexing of MATLAB references. References have been evaluated with or without indexing accessing data sequentially or non-sequentially (indexing with colon expressions). No references including array indexing using other reference e.g. *'myArrayRef(1,IndexArrayRef(:))'* have been evaluated, although they supported by the compiler. However, they are not used in SIMD processing since data of array references used for indexing *'IndexArrayRef(:)'* are unknown at compile time. Thus, these data must be packed (or even repacked in case of packed data) at run time, limiting opportunities for SIMD processing. Evaluation of MATLAB code including control flow structures such as for-loops and if-else statements combined with SIMD blocks has been also a focus of benchmarking. A subset of the selected benchmarks include both control flow and data-dependencies. Control flow dependencies limit vectorization opportunities. The code segments carrying such dependencies are

transformed by the compiler to scalarized code (SIMD block annotations are not used in this case). The compiler has been evaluated for its efficiency to handle control flow dependencies and for exploring the overheads introduced when packing and unpacking operations are used to generate scalarized code with packed data variables. Finally, the code with loop-carried dependencies has been manually modified to a multi input stream version enabling vectorization. For example the initial MATLAB code of FIR includes a one-dimensional vector of 8192 elements. MATLAB code has been manually modified to a multi input stream of 32 vectors with 26 elements each (32x256 matrix). The variable references of MATLAB input code have been manually modified from *'in(k)'* to *'in(:,k)'*, thus allowing vectorization of the first dimension by the compiler.

Table 18 shows the characteristics which have been considered about the selection of the benchmarks for the evaluation of the compiler. FFT algorithms have been selected to evaluate code generation for two dimensional arrays. Furthermore, FFT benchmarks include a variety of different indexing cases as well as for-loop statements and scalar operations involving vector variables (data shuffling). CFO algorithm has been chosen as a case using one dimensional arrays. The mean, FIR and CORDIC benchmarks are included in the benchmark set due to the data-dependencies they include (CORDIC includes data and control flow dependencies). FIR includes two nested for-loops and CORDIC includes two nested for-loops and one if-else statement. QR decomposition has been chosen because of several features. The most interesting feature is the three dimensional structure of the algorithm making code parallelization challenging. QR decomposition uses a large number of mathematical functions (sine, cosine, Inverse-Tangent, exponential function, complex conjugate operation and the absolute value function) which are mapped to custom instructions. QR decomposition also includes advanced MATLAB operations such as array concatenation, array multiplication and matrix transpose.

6.2.4 Architectures Selection

A number of different instruction set architectures ranging from general purpose to application specific ones has been selected to run and evaluate the code generated by the compiler. The selection has been made to address as much as possible a wide range of state of the art processor features broadly used today and covering a big market share. The specific choice of processors (and the experimental environment in general) offers a wide space that allows the thorough evaluation of the compiler's applicability, flexibility and extendibility as well as the performance and the portability of the generated code.

ARM architecture provides three different profiles including the A-profile for high performance applications such as mobile and enterprise, the R-profile for embedded systems demanding high performance, and the energy efficient M-profile for embedded and IoT applications. ARM is a reduced instruction set computing (RISC) architecture with a uniform register file operating only on registers (no direct interaction with memory). ARM generally provides a fixed-length 32-bit instruction set and recently introduced the

FFT	two dimensional arrays scalar operations with vector variables indexing for-loops
CFO	one dimensional arrays trigonometric functions scalar operands in SIMD operations (expanding)
CORDIC	data and control flow dependencies nested for-loops and if-else statements
FIR	data-dependencies nested for-loops
MEAN	data-dependencies
QR-decomposition	three dimensional arrays mathematic and trigonometric functions array concatenation (with vectors) array multiplication (with vectors) matrix transpose - scalar processing involving vector variables

Table 18: Benchmarks characteristics.

64-bit ARMv-8 architecture. The newest ARM architectures provide the NEON general-purpose SIMD engine (supported only in Cortex-A series) to accelerate multimedia and signal processing algorithms. The NEON technology is a 128-bit SIMD architecture extension performing packed SIMD processing on integer and single precision floating point data types.

The x86 is a complex instruction set computer (CISC) architecture, compatible with the Intel IA-32 and x86 AMD processors. The x86 was firstly appeared in Intel 80386 processor and continued in subsequent Intel processors. The x86 architecture includes variable length instructions which can operate with memory data. The x86 architecture provides the Streaming SIMD Extension (SSE) which was subsequently expanded to SSE2, SSE3, SSSE3, and SSE4 as well as the Advanced Vector Extensions (AVX) SIMD extension (and its AVX2, AVX-512 expansions). The extensions provide SIMD processing on floating point and integer arithmetic. The different SIMD extensions vary on the bit width of vectors and the instruction set extension which is supported.

ASIPs is another class of processors intended for advanced computing with a wide range of available processors at the market. Keeping in view, the cost-performance trade-off for various signal processing applications, several architectures from CEVA, Broadcom, Qualcomm, Texas Instruments etc. are used in the embedded system design community. These processors share several features that differentiate them with other class of processors such as general purpose processors or other Micro-controllers. One of the predominant common trait is that all these processors function in the VLIW

paradigm which helps in instantiating several heterogeneous functional units. This makes possible the exploitation of the instruction level parallelism (ILP) in algorithms. Among the many instantiations of VLIW data-paths, a large part of functional units is typically dedicated to SIMD data-paths to benefit from the data-level parallelism (DLP). Thirdly, from flexibility perspective, all of them can either provide extensible processor architecture or provide a family of architectures to help the cause of catering to the different application scenarios. In this study, BoT and TinyBoT ASIPs derived from ADRES [Mei et al., 2003] template, have been chosen to experiment with.

To put this in perspective, the BoT architecture is specially tuned to dimensions in a much more narrow/niche domain. The BoT architecture has been developed at research center IMEC and is used for wireless baseband processing meeting high throughput and low power consumption. The processor provides high performance operations using VLIW desgin of 10 slots performing parallel processing up to 8 elements on a large set of trigonometric and complex arithmetic instructions. The BoT architecture described in subsection 6.5.1, is a realization of such an ASIP for wireless baseband application. The table 19 below, provides a brief comparison between the BoT and processors in the CEVA DSP family.

	VLIW -width	**SIMD -width**	**register-file -depth**	**bit -manipulations**	**Spl. instructions**
CEVA DSP series (XC4000) series	Upto 8	4 parallel complex -numbers	Upto 512 bits wide	40 bit shifts, Pack-unpack	Predicated ALU, high precision MAC instruction
BoT	10	8 parallel complex -numbers	512 bits wide, 8 parallel 32 bits complex number	Pack-unpack, multi-word shuffle	Multi-cycle pipelined trigonometric and complex-arithmetic instructions

Table 19: Comparison between BoT and processors in the CEVA DSP family.

6.3 Results from ARM Architectures

This section presents the performance of the compiler's generated code compared to that of the code generated by the MathWorks Coder on two different ARM processors. For the compilation of generated code by both MATLAB compilers, Clang/LLVM, GCC and MSVC auto-vectorizing C compilers have been used. The benchmark has been

executed on the Raspberry computer boards using Rasbian [Rasbian, 2016] and Windows IoT [Windows IoT, 2016] operating systems.

6.3.1 Presentation of ARM Architectures

ARM (Advanced RISC Machine) [ARM, 2016] is a family of reduced instruction set computing (RISC) architectures for computer processors widely applied in markets such as mobile, embedded systems and the emerging domains of Internet of Things (IoT) - wearables [ARM markets, 2016]. The ARM architecture [Furber, 2000] consists of a large uniform register file with load/store instructions operating only on register content. The instruction fields are uniform with fixed-length to simplify instruction decoding and ease pipelining, at the cost of decreased code density. Additionally, the ARM architectures provides: -an Arithmetic Logic Unit (ALU) with Barrel shifter to maximize data processing instructions -register auto-increment addressing modes for optimization of program loops -load/store Multiple instructions and conditional execution instructions to maximize the data and execution throughput. These enhancements allow ARM processors to achieve a good balance of high performance, small code size, low power consumption, and small silicon area and maintaining the RISC design strategy [ARM reference manual, 2016].

ARM provides several industry-leading technologies included in the ARM processors. The big.LITTLE processing is a power-optimization technology combining high-performance ARM CPU cores (big cores) with efficient power consumption ARM CPU cores (LITTLE cores) for the execution of high and low intensity workloads to deliver peak-performance capacity and a good balance between performance and power efficiency. DSP and advanced SIMD (NEON) instruction set extensions, are technologies aiming to increase the processing capability for multimedia applications. ARM provides processors with hardware floating point unit to increase the floating point arithmetic and the Jazelle optimization for the acceleration of multi-tasking Java Virtual Machine (JVM) execution. Finally, technologies such as the TrustZone technology for low cost security optimization and the virtualization extension for implementation of efficient virtual machine hypervisors are provided as well. The most related to the current study technology is the advanced SIMD NEON, which is discussed in the following subsections.

Advanced SIMD Extension (NEON) The Advanced SIMD extension [NEON reference manual, 2016] (NEON) is a general-purpose SIMD engine supported only by Cortex-A processors for acceleration of multimedia and signal processing applications such as video encode/decode, 2D/3D graphics, gaming and image processing. NEON performs parallel computing of 128-bit width of same data types among which: signed/unsigned 8-bit, 16-bit, 32-bit, 64-bit integer and single precision floating point. The SIMD engine includes its own independent pipeline and register file while for some ARM devices implementing both NEON and ARM floating point architecture, the register bank

is shared between them. There are 32 registers of 64-bits each, but they may also be used as 16 registers of 128-bits each. NEON technology can be exploited by auto-vectorizing C compilers, by coding with assembly SIMD instructions, by using C intrinsic functions or by using other technologies which incorporates NEON such as openMAX library.

Programming with Advanced SIMD Extension (NEON) The definitions of NEON semantics such as the vector data types and the intrinsic functions are included in the *'arm_neon.h'* header file. The union *'__n128'* represents the ARM Advanced SIMD 128-bit type including all the supported data types mentioned in previous paragraph. For the benchmarking of the compiler targeting ARM, signed integers of 32-bit and 16-bit per vector's element have been used to implement fixed point vectors of SIMD width 4 and 8 respectively. For the floating point vector with SIMD width 4, single precision floating point of 32-bit per vector's element has been used, while floating point vector with SIMD width 8 is not provided in C by NEON. For the benchmarking of the compiler, several intrinsics of the aforementioned data types have been used such as loading and storing elements from/to vectors, arithmetic SIMD instructions and bit shift SIMD operations.

ARMv7 Architecture ARMv7 [ARM reference manual, 2016] is a 32-bit RISC architecture implementing the *Application*, *Real-time* and *Microcontroller* profiles. The fixed-length instruction set includes load/store instructions, arithmetic instructions such as addition, subtraction, 32-bit or 64-bit multiplication and optionally division with barrel shifter, while the majority of them may be conditionally executed. The architecture consists of 16 32-bit registers including the stack pointer, link register and program counter. Most of the technologies described in 6.3.1 are supported, and the ARM floating point architecture may be implemented with different options regarding the trapping of floating point exceptions and the floating point precision (half versus single precision). To improve code density ARMv7 provides two additional instruction sets: Thump, which is a 16-bit encoding instruction set supporting a subset of ARM instructions and Thump-2, which is a variable-length instruction set extending the 16-bit Thump instruction set with additional 32-bit instructions.

ARMv-8 Architecture ARMv8 [ARMv8 reference manual, 2016] is an optional 64-bit architecture which is backward compatible with previous ARM architectures. The ARMv8 provides a 32-bit instruction set (A32) largely similar to that of ARMv7 supporting Thump (T32) and Thump-2 as well. Additionally, ARMv8 introduces a new instruction set (A64) with similar functionality to A32 and T32. The most significant improvements of A64 is the insertion of new instructions with 64-bit operands and the extension of the register file from 16 to 31 64-bit registers. Furthermore, the registers for SIMD processing have been enlarged from 16 to 32, and the floating point instructions are fully complaint to IEEE 754, applying single or double precision floating point arithmetic.

Raspberry PI 2 Raspberry PI 2 Model B [Raspberry Pi 2, 2016] is a single-board computer of the second generation Raspberry PI, using the BCM2836 Broadcom SoC. It is composed by a 900Mhz quad-core cortex-A7 processor of ARMv7, 1GB RAM shared with the Broadcom VideoCore IV GPU and integrated 512KB level 2 cache memory. There are many available operating systems for Raspberry PI 2 such as the Rasbian [Rasbian, 2016] (a configured version of Debian) and Windows IoT Core [Windows IoT, 2016] which has been used for the purpose of the study. Cortex-A7 processor may be used as a power-efficient standalone multicore processor cluster or as a LITTLE core in ARM big.LITTLE architecture combined with Cortex-A15 and Cortex-A7 big cores. The processor is an eight stage pipeline microarchitecture executing instructions in-order. The implemented ARM technologies of Cortex-A7 are: NEON, Thump-2, Jazelle, hardware virtualization and floating point unit with version *'VFPv4'*.

Experimental configuration on Raspberry PI 2 For the conduction of the experiments on Raspberry PI 2, Rasbian 4.4 OS has been used in combination with Clang 3.7/LLVM 3.7 and GCC 4.9 as well as Windows IoT 10 with MSVC (Visual studio 2015, version 14.0).

Raspberry PI 3 Raspberry PI 3 Model B [Raspberry Pi 3, 2016] is a single-board computer of the third generation Raspberry PI using the BCM2837 Broadcom SoC. Raspberry PI 3 consist of a 1.2GHz 64-bit quad-core Cortex-A53 processor of ARMv8 and 1GB LPDDR2 RAM of 900 MHz with 32kB Level 1 and 512kB Level 2 cache memory. Cortex-A53 achieves significantly higher performance than the Cortex-A7 and it may be also used either as standalone multicore processor cluster or a LITTLE core in combination with Cortex-A72 processor as a big core in ARM big.LITTLE architecture. Cortex-A53 is an eight stage in-order pipelined processor and it supports both 32-bit and 64-bit instruction sets. Finally, Cortex-A53 implements ARM technologies, similar to that presented in Cortex-A7.

Experimental configuration on Raspberry PI 3 For the conduction of the experiments on Raspberry PI 3, Rasbian 4.9 OS has been used in combination with Clang 3.5/LLVM 3.5 and GCC 4.9 as well as Windows IoT 10 with MSVC (Visual studio 2015, version 14.0).

6.3.2 Performance of Generated Code on Raspberry PI 2

Performance using packed fixed point types This section presents the performance of the generated code by the compiler using various data types and different C compilers on the Raspberry PI 2. Figures 19, 20 and 21 show the normalized execution times (Table 20) of vectorized generated code with packed fixed point types compared to the generated code by MathWorks Coder using Clang/LLVM, GCC and MSVC C com-

pilers correspondingly. The speed-up achieved by the vectorized code compared to that of MathWorks compiling with clang/LLVM and executing on Rasbian OS, is between 1.1x-44.6x and 7.6x on average for SIMD width 4 and between 2.1x-43.4 with average speed-up of 7.57x for SIMD width 8. Similarly, using GCC for compilation of the generated code, and executing the output code on Rasbian OS the speed-up of vectorized code against MathWorks generated code is from 2.4x up to 76.4x and 14.2x on average for SIMD width 4, while the speed-up for SIMD width 8 is between 2.4x-45.7 and 13x on average. Finally, the speed-up achieved by the vectorized generated code using the MSVC compiler and Windows IoT OS is from 2.9x up to 39.3 and 9.1x on average for SIMD width 4, and the speed-up achieved for SIMD width 8 is from 3.7x up to 20.4x with 10.8x on average.

Discussion of performance using packed fixed point types The FFT-32 benchmark achieves the highest speed-up using any of the three C compilers with SIMD width 4. The highest speed-up with SIMD width 8, is obtained at QR-decomposition-32 benchmark for Clang and GCC, while using MSVC the highest speed-up obtained at FIR application. The vectorized code with SIMD width 4 achieves better performance than the vectorized code with SIMD width 8 for the MEAN application using small input stream sizes, especially compiling with GCC compiler. The MSVC doesn't exploit efficiently the SIMD instructions processing 8 elements for CFO application and it obtains the same performance as using SIMD width of 4 elements. Finally, the Clang compiles efficiently the MathWorks generated code of FIR application (comparing the reference values in Table 20) obtaining similar performance to that of vectorized code with SIMD width 4. Although, the vectorized code with SIMD width 8 achieves a speed-up of 2.2x compared to MathWorks generated code for FIR application.

	Fig. 19, Fig. 22	Fig. 20, Fig. 23	Fig. 21, Fig. 24	Fig. 25, Fig. 28	Fig. 26, Fig. 29	Fig. 27, Fig. 30
fft32	94.92	87.75	48.07	4.01	4.61	2.90
fft64-v1	31.33	26.16	52.33	15.24	15.84	15.60
fft64-v2	156.19	123.78	99.87	11.33	10.63	13.30
fft128	345.61	311.50	220.20	30.42	33.43	31.77
cfo-32	9.05	9.35	22.30	37.69	36.89	18.07
cfo-64	63.95	64.83	41.60	73.80	74.94	34.13
cfo-128	126.47	132.69	82.30	148.68	153.63	66.83
cordic-64	1178.05	2081.48	4106.57	719.76	645.27	628.17
fir-32x64	175.52	425.50	1695.80	466.88	403.74	396.20
fir-32x128	400.62	928.03	3667.17	1035.53	890.86	863.70
fir-32x256	805.31	1931.84	7743.83	2176.11	1878.44	1799.53
mean-32	1.28	2.41	1.97	0.94	1.09	1.53
mean-64	2.36	4.62	3.40	1.82	2.07	2.90
mean-128	4.53	9.04	6.37	3.51	4.06	6.17
mean-1024	35.61	70.75	48.63	27.23	32.02	46.40
qr-decomp-32	3134.28	2690.74	1388.73	353.98	363.93	168.17
qr-decomp-64	812.29	832.95	730.57	705.69	734.44	344.10
qr-decomp-128	1644.79	1666.91	1445.83	1445.23	1484.21	704.80

Table 20: Reference values (exec. time in µs) used for normalization of results on PI 2

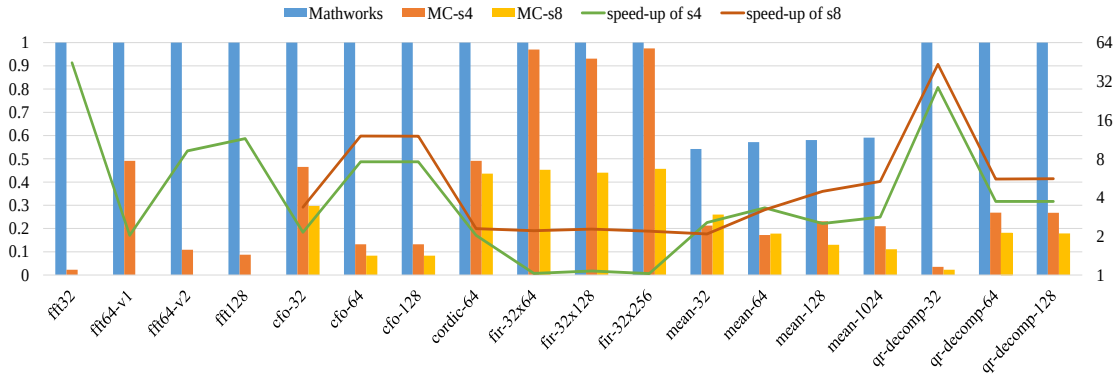

Figure 19: Speed-up comparing with MathWorks compiler on PI 2 using Clang/LLVM with packed fixed point data

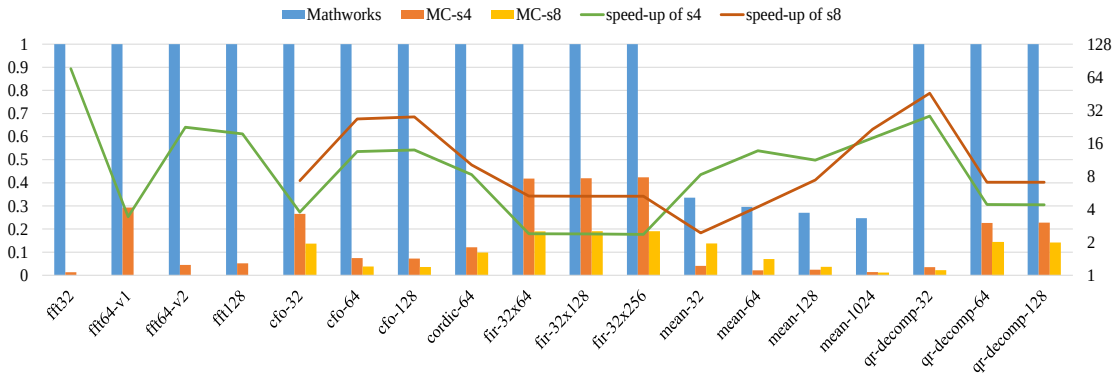

Figure 20: Speed-up comparing with MathWorks compiler on PI 2 using GCC with packed fixed point data

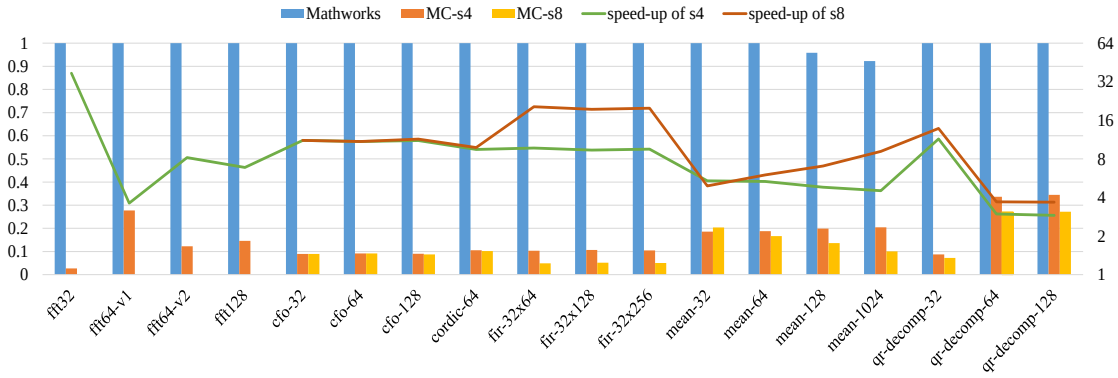

Figure 21: Speed-up comparing with MathWorks compiler on PI 2 using MSVC with packed fixed point data

Performance using unpacked fixed point types Figures 22, 23 and 24 present the normalized execution times (Table 20) of the compiler's vectorized generated code with unpacked fixed point types compared to the generated code by MathWorks Coder. The speed-up achieved compiling the generated code by both MATLAB compilers with Clang and executing the output code on Rasbian OS (Fig. 22) is up to 28.4x, with average speed-up of 6.3x for SIMD width 4, and between up to 19.8x with average speed-up of 6.3x for SIMD width 8. Using GCC for compilation and executing, the output code on the Rasbian OS (Fig. 23), the speed-up gained from the vectorized code compared to MathWorks generated code is up to 38x with 7.3x on average for SIMD width 4 and up to 18.2x with 6.9x on average for SIMD width 8. The Windows IoT OS and MSVC configuration (Fig. 24) achieves a speed-up between 1x-19.2x and 7.8x on average for SIMD width 4, and a average speed-up of 11x up to 26.6x for SIMD width of 8.

Discussion of performance using unpacked fixed point types The FFT-32 with unpacked data types obtains the highest speed-up for SIMD width of 4, while CFO, CORDIC and FIR applications achieve the highest speed-up for SIMD width 8 by using Clang, GCC and MSVC respectively. The worst performance of the compiler's generated code is obtained at MEAN application where the vectorized code compiled with Clang and GCC obtains a negative speed-up (worse performance) compared to MathWorks generated code and similar performance as that of MathWorks generated code compiling with MSVC. The reduced performance of vectorized code with unpacked data types at MEAN application is due to the packing/unpacking overhead which cannot be eliminated by the acceleration of the SIMD instruction which is included in the SIMD block.

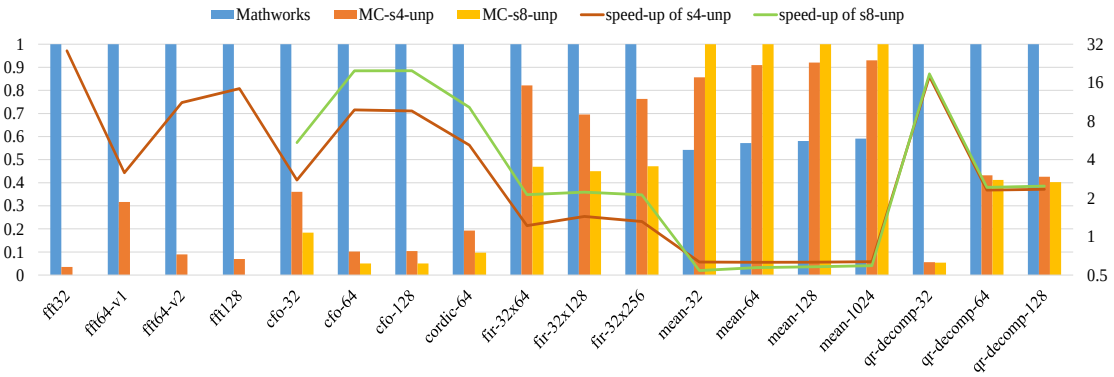

Figure 22: Speed-up comparing with MathWorks compiler on PI 2 using Clang/LLVM with unpacked fixed point data

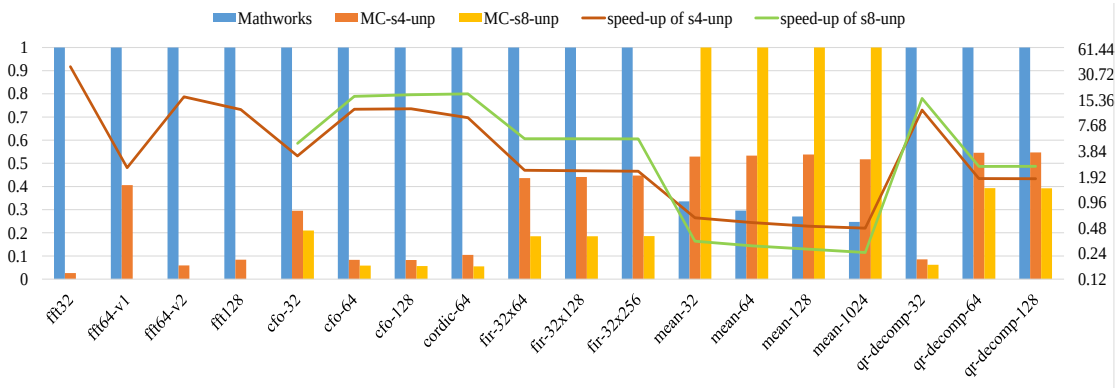

Figure 23: Speed-up comparing with MathWorks compiler on PI 2 using GCC with unpacked fixed point data

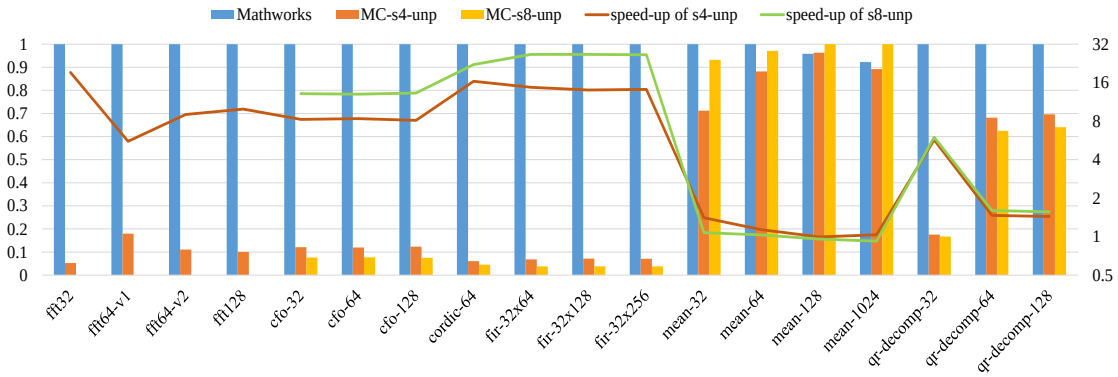

Figure 24: Speed-up comparing with MathWorks compiler on PI 2 using MSVC with unpacked fixed point data

Performance using packed floating point types Figures 25, 26 and 27 present the normalized execution times (Table 20) of the vectorized generated code with packed (only of SIMD width 4) floating point types compared to MathWorks generated code using Clang, GCC and MSVC respectively. The compler's vectorized generated code compared to that of MathWorks achieves an average speed-up of 3.3x up to 8.2x using Clang and Rasbian OS, a speed-up between 1.5x-12.3x with average of 4.4x compiling with GCC and a maximum speed-up of 4.9x with 2x on average using MSVC and Windows IoT OS.

Discussion of performance using packed floating point types The vectorized code achieves the highest speed-up for CFO application compiling the generated code with any of the three C compilers. The Clang and MSVC doesn't exploit efficiently the SIMD-style generated code by the compiler for FFT-64 and FFT-128 benchmarks leading to worse performance compared to MathWorks generated code. Finally, the MSVC compiles efficiently the generated code of MEAN application by MathWorks obtaining a better performance compared to the vectorized generated code.

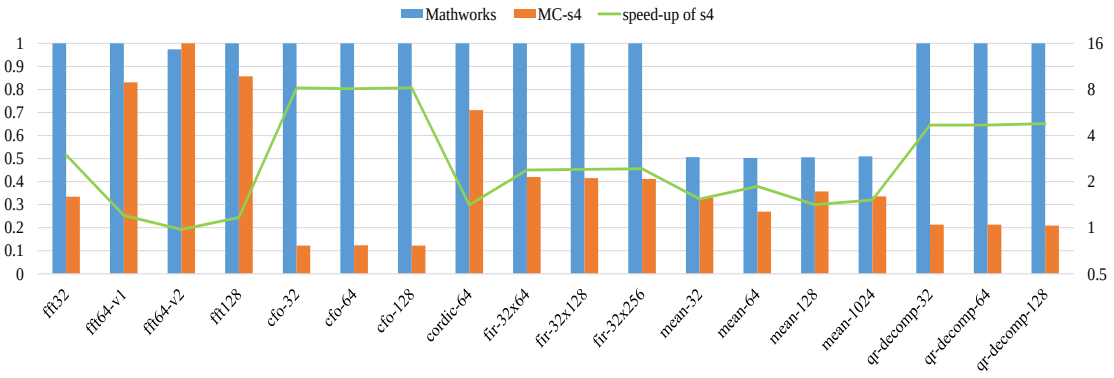

Figure 25: Speed-up comparing with MathWorks compiler on PI 2 using Clang/LLVM with packed floating point data

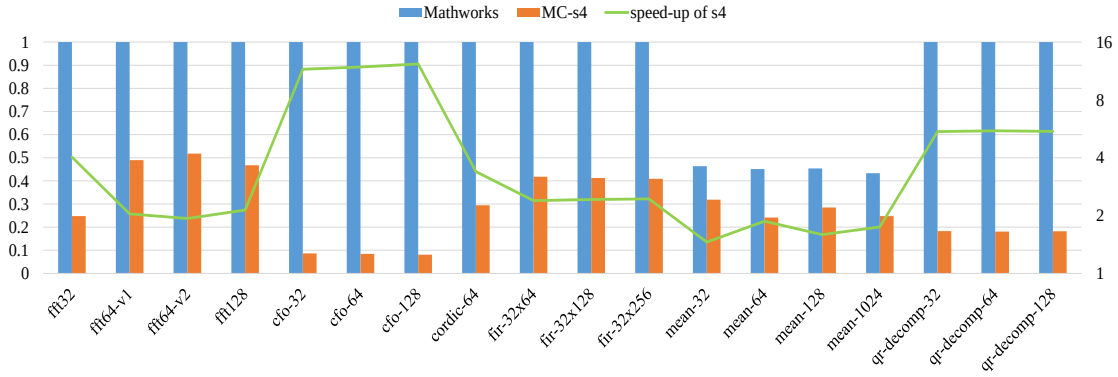

Figure 26: Speed-up comparing with MathWorks compiler on PI 2 using GCC with packed floating point data

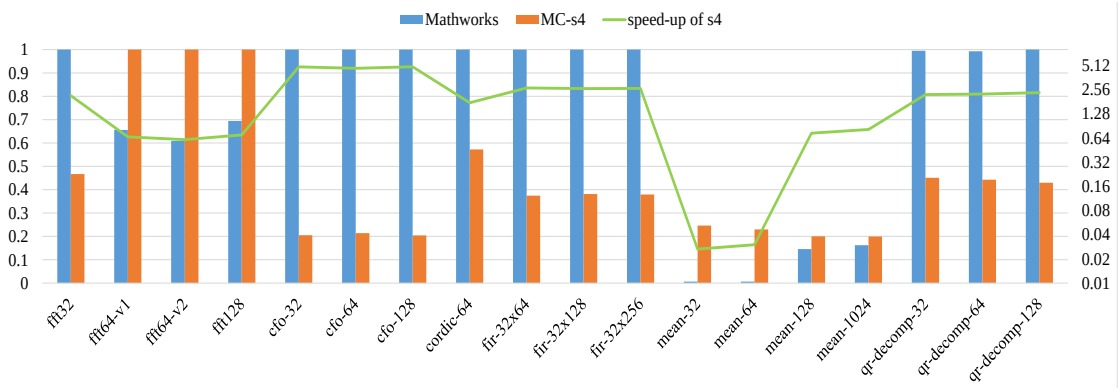

Figure 27: Speed-up comparing with MathWorks compiler on PI 2 using MSVC with packed floating point data

Performance using unpacked floating point types Figures 28, 29 and 30 present the normalized execution times (Table 20) of the vectorized generated code with unpacked of SIMD width 4 floating point types compared to MathWorks generated code. The vectorized generated code against MathWorks generated code achieves an average speed-up of 3.1x up to 9.7x compiling with Clang and executing the benchmark on Rasbian OS (Fig. 28), a maximum speed-up of 10.6x and 3.3x on average using GCC and Rasbian OS (Fig. 29), and an average speed-up of 1.9x up to 5.7x compiling with MSVC and executing the output code on Windows IoT OS (Fig. 30).

Discussion of performance using unpacked floating point types Similarly to the results with packed floating types, the highest speed-up is obtained at CFO application for all three C compilers. The Mean application achieves better performance compiled with MathWorks Coder than generating vectorized code. Finally, the vector-

ized code of FFT benchmarks cannot be efficiently compiled with MSVC obtaining same performance as that of MathWorks generated code.

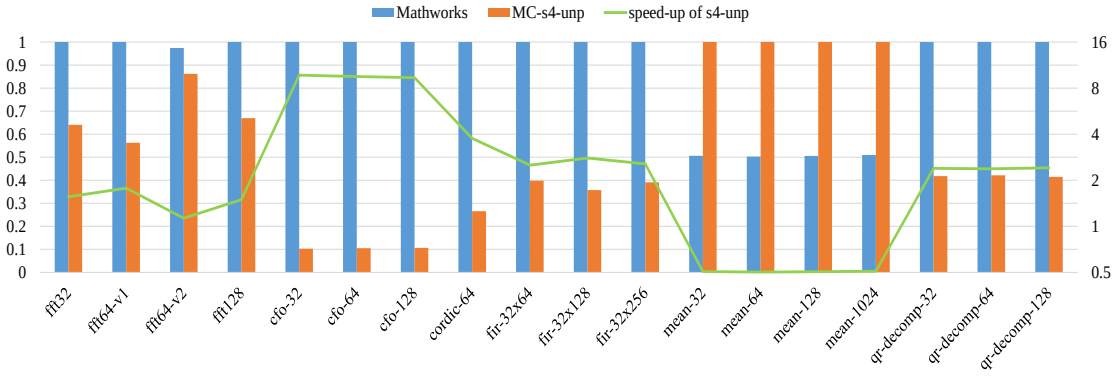

Figure 28: Speed-up comparing with MathWorks compiler on PI 2 using Clang/LLVM with unpacked floating point data

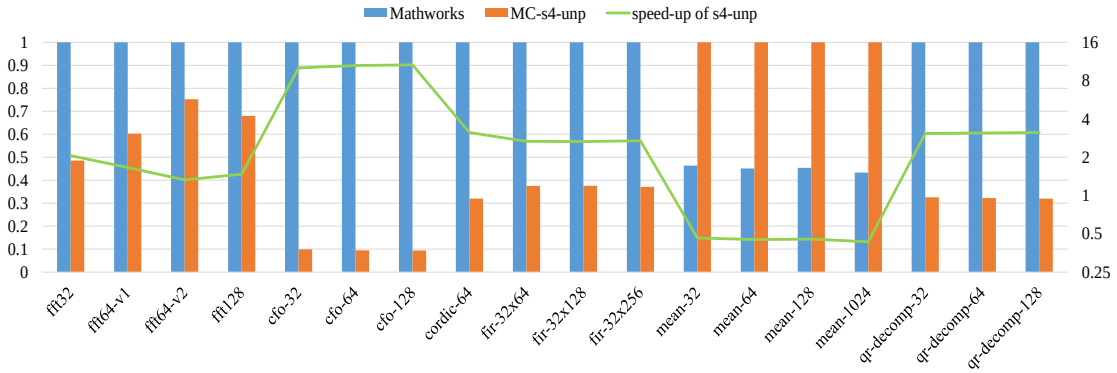

Figure 29: Speed-up comparing with MathWorks compiler on PI 2 using GCC with unpacked floating point data

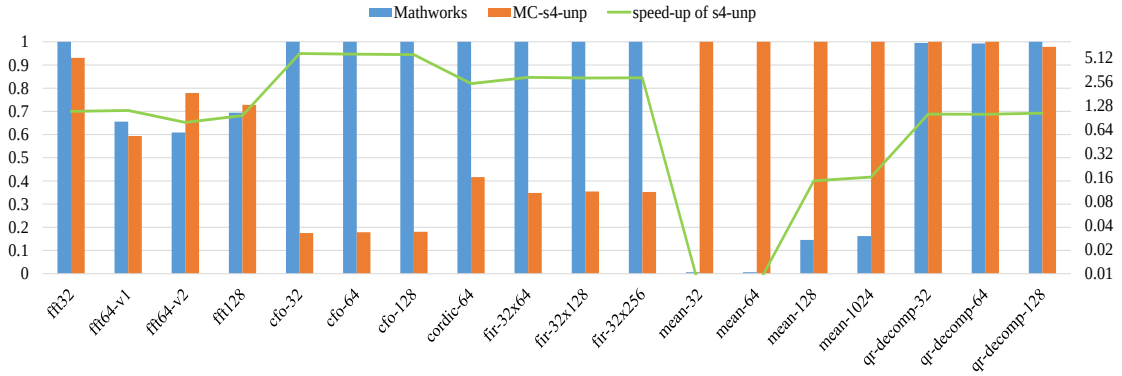

Figure 30: Speed-up comparing with MathWorks compiler on PI 2 using MSVC with unpacked floating point data

Performance among SIMD configurations using fixed point types Figures 31, 32 and 33 show the normalized execution times (Table 21) of the vectorized generated code among different SIMD configurations for the fixed point benchmark. The performance of vectorized code with packed data types compared to that of unpacked data types achieves speed-up 5.3x and 1.7x on average for SIMD width 4 and maximum speed-up of 9x with 2.7x on average for SIMD width 8 using clang and Rasbian OS (Fig. 31). The speed-up of the vectorized code with packed data types using GCC (Fig. 32) is up to 37.3x and 6.6x on average for SIMD width 4 and up to 86.7x with 10.8x on average for SIMD width 8. Finally, the speed-up of vectorized code with packed data types compared to that of unpacked data types using MSVC and Windows IoT OS (Fig. 33) is up to 4.8x and 9.9x with 1.9x and 2.8x on average for SIMD width 4 and 8 respectively.

Discussion of performance among SIMD configurations using fixed point types The vectorized code using packed data types and compiled with Clang achieves a better performance than the vectorized code using unpacked data types only for FFT-32, MEAN and QR-decomposition benchmarks while the performance of FIR with SIMD width 8 between packed/unpacked is same. The vectorized code with packed data types and compiled with GCC attains speed-up for almost all benchmarks except CORDIC application, leading to a worse performance compared to the unpacked vectorized code and MEAN application achieving same performance for packed and unpacked data types. Compiling with MSVC the performance of vectorized code with packed data types obtains speed-up only for MEAN and QR-decomposition (with any SIMD width) as well as for CFO application with SIMD width 4. The enhanced performance of vectorized code with unpacked data types is due to compilers' optimizations and code transformations allowing for the vectorized code to be processed more efficiently among the SIMD cores of the ARM processor.

	Fig. 31	Fig. 32	Fig. 33	Fig. 34	Fig. 35	Fig. 36
fft32	3.34	2.31	2.50	2.57	2.24	2.70
fft64-v1	15.40	10.62	14.50	12.66	9.55	15.60
fft64-v2	16.94	7.37	12.20	11.33	8.00	13.30
fft128	30.09	26.22	32.13	26.07	22.72	31.77
cfo-32	4.21	2.76	2.70	4.61	3.66	3.70
cfo-64	8.44	5.41	4.97	9.12	7.12	7.30
cfo-128	16.68	10.92	10.13	18.15	14.43	13.67
cordic-64	579.10	251.85	432.63	511.13	206.47	359.37
fir-32x64	170.32	185.54	174.57	196.06	168.65	148.43
fir-32x128	373.03	409.12	391.27	430.18	367.39	329.80
fir-32x256	785.11	863.30	811.63	896.55	768.87	682.67
mean-32	1.28	2.41	1.83	0.94	1.09	1.53
mean-64	2.36	4.62	3.30	1.82	2.07	2.90
mean-128	4.53	9.04	6.37	3.51	4.06	6.17
mean-1024	35.61	70.75	48.63	27.23	32.02	46.40
qr-decomp-32	175.46	229.31	243.20	148.02	118.49	168.17
qr-decomp-64	351.03	454.52	498.33	297.27	236.63	344.10
qr-decomp-128	700.66	912.01	1006.37	599.35	474.74	689.30

Table 21: Reference values (exec. time in µs) used for normalization of vectorized results on PI 2

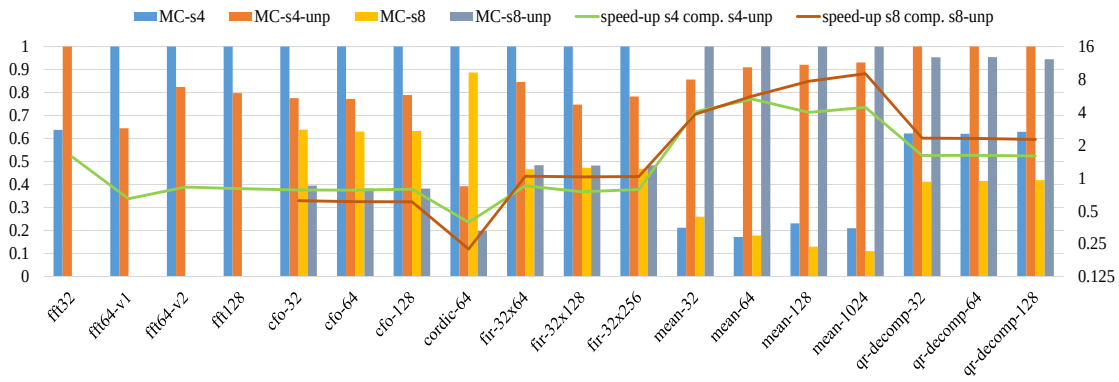

Figure 31: Performance of vectorized code with unpacked fixed point data types versus packed fixed point data types using Clang/LLVM on Raspberry PI 2

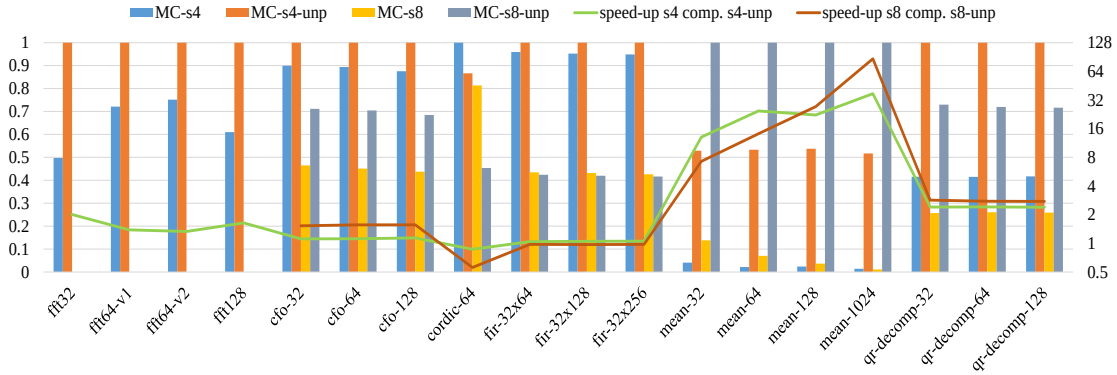

Figure 32: Performance of vectorized code with unpacked fixed point data types versus packed fixed point data types using GCC on Raspberry PI 2

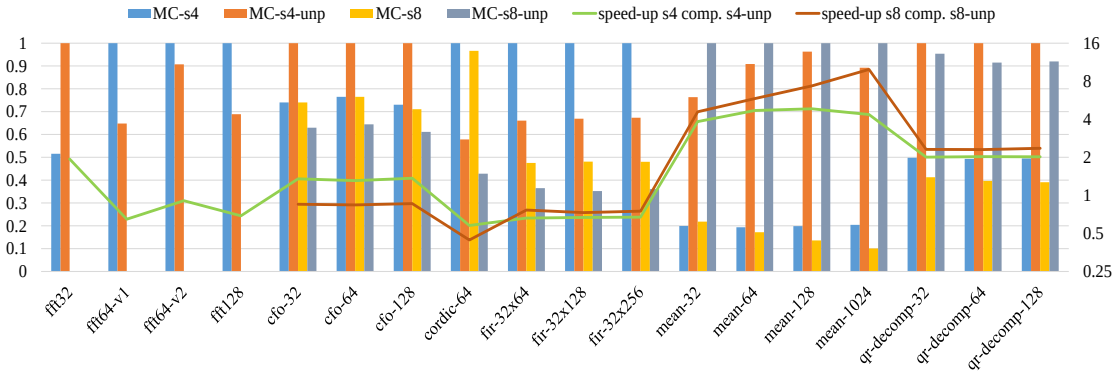

Figure 33: Performance of vectorized code with unpacked fixed point data types versus packed fixed point data types using MSVC on Raspberry PI 2

Performance among SIMD configurations using floating point types Figures 34, 35 and 36 show the normalized execution times (Table 21) of the vectorized generated code with different SIMD configurations for the floating point benchmark. The performance of the vectorized code with packed data types compared to that with unpacked data types achieves an average speed-up of 1.6x up to 3.7x using Clang and Rasbian OS (Fig. 34), a maximum speed-up of 4.1x and 1.9x on average using GCC and Rasbian OS (Fig. 35) and a maximum speed-up of 5x with 2x on average using the MSVC compiler and Windows IoT OS (Fig. 36).

Discussion of performance among SIMD configurations using floating point types The vectorized code with packed data types achieves better performance than the vectorized code with unpacked data types only for FFT-32, MEAN and QR-decomposition applications using Clang and MSVC. Moreover, the vectorized code of packed data types compiled with GCC attains a better performance than the unpacked vectorized code for all the benchmarks except MEAN application.

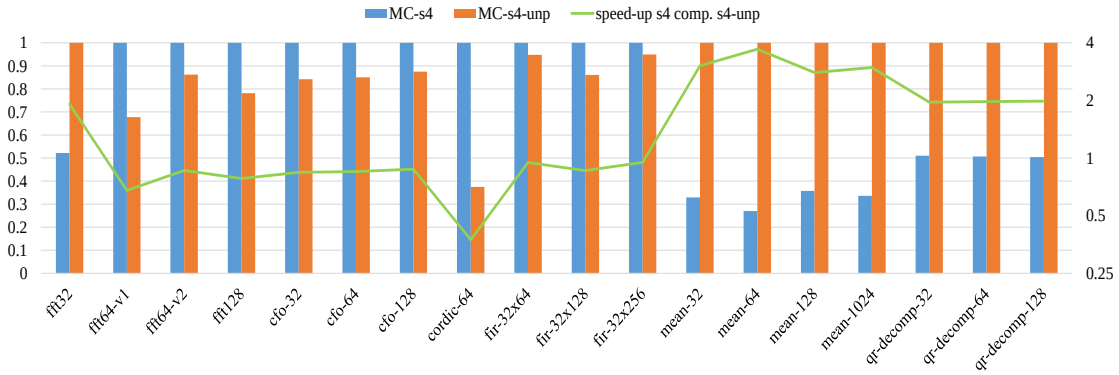

Figure 34: Performance of vectorized code with unpacked floating point data types versus packed floating point data types using Clang/LLVM on Raspberry PI 2

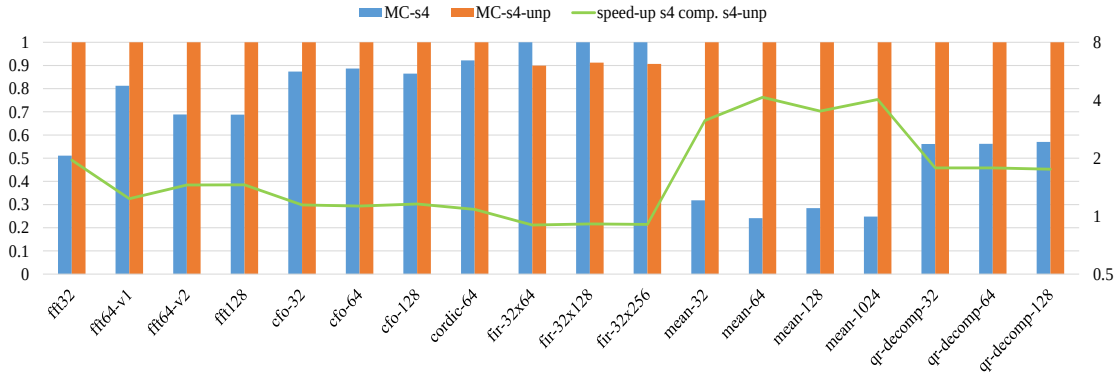

Figure 35: Performance of vectorized code with unpacked floating point data types versus packed floating point data types using GCC on Raspberry PI 2

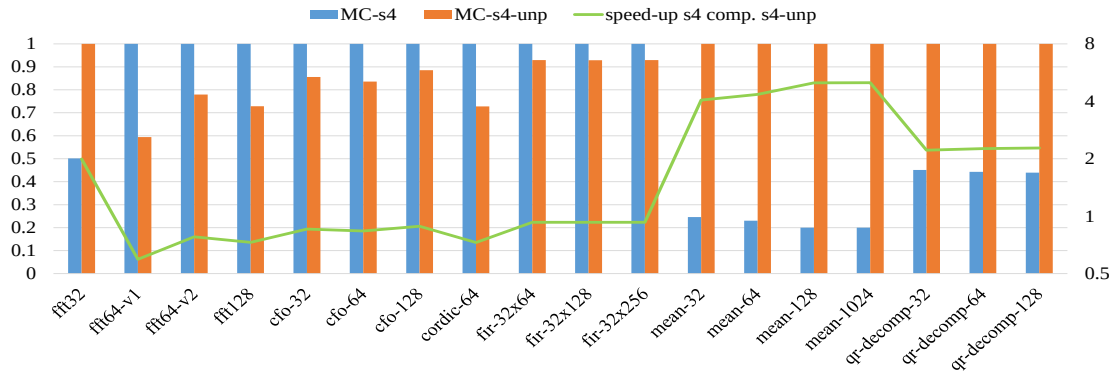

Figure 36: Performance of vectorized code with unpacked floating point data types versus packed floating point data types using MSVC on Raspberry PI 2

6.3.3 Performance of Generated Code on Raspberry PI 3

Performance using packed fixed point types This subsection presents the performance of the compiler's generated code compared to that of the code generated by MathWorks Coder using various data types and different C compilers/Operating systems on the Raspberry PI 3. Figures 37, 38 and 39 show the normalized execution times (Table 22) of the vectorized generated code with packed fixed point types compared to the MathWorks generated code using Clang, GCC and MSVC respectively. The speed-up achieved by the performance of vectorized code compared to that of MathWorks generated code compiling with Clang and executing on Rasbian OS is up to 16.6x and 12.8x on average for packed types of SIMD width 4 and between 1.1x-24.5x with average speed-up of 9x for packed types of SIMD width 8. By compiling with GCC and executing the output code on Rasbian OS, the speed-up achieved by the vectorized generated code compared to the generated code by MathWorks is from 1.8x up to 91.8x and from 2.6x up to 52x with average speed-up of 15.9x and 14.1x using SIMD width 4 and 8 respectively. Finally, the speed-up of vectorized generated code compared to the MathWorks generated code compiling with MSVC and executing the benchmark on Windows IoT OS is from 3.4x up to 46.7x with 10.5x on average for SIMD width 4 and between 4.1x-21.9x with 12.5x on average for SIMD width 8.

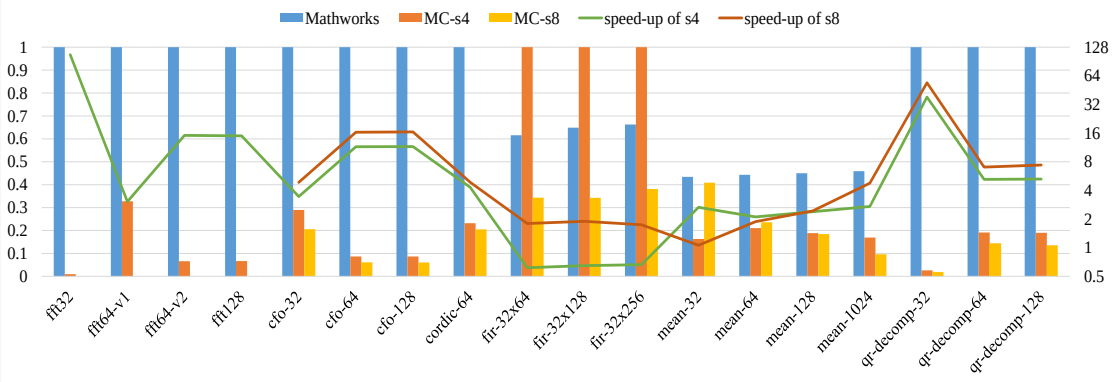

Figure 37: Speed-up comparing with MathWorks compiler on PI 3 using Clang/LLVM with packed fixed point data

	Fig. 37, Fig. 40	Fig. 38, Fig. 41	Fig. 39, Fig. 42	Fig. 43, Fig. 46	Fig. 44, Fig. 47	Fig. 45, Fig. 48
fft32	38.50	34.35	27.97	1.85	1.79	1.70
fft64-v1	14.18	9.85	33.73	5.93	6.02	8.10
fft64-v2	61.00	45.67	62.37	4.15	3.60	6.07
fft128	131.73	110.65	135.07	11.67	12.37	16.03
cfo-32	4.00	7.07	12.77	18.72	17.45	9.30
cfo-64	26.21	25.60	24.73	37.66	31.71	18.70
cfo-128	52.24	50.90	49.30	76.59	64.78	36.90
cordic-64	827.56	864.14	2378.07	313.81	257.29	328.37
fir-32x64	99.55	113.37	1087.40	206.20	219.40	225.53
fir-32x128	217.63	248.68	2402.27	450.65	480.07	496.00
fir-32x256	419.97	516.43	5010.60	935.71	996.66	1050.93
mean-32	0.62	0.90	1.50	0.49	0.40	0.90
mean-64	1.14	1.71	2.60	0.94	0.75	1.70
mean-128	2.20	3.40	5.00	1.83	1.45	3.50
mean-1024	16.85	26.66	36.70	14.80	12.88	26.97
qr-decomp-32	1249.65	1101.88	840.40	137.20	143.69	96.70
qr-decomp-64	343.48	343.97	464.33	272.60	286.99	194.83
qr-decomp-128	695.17	682.82	931.37	559.17	579.24	392.13

Table 22: Reference values (exec. time in µs) used for normalization of results on PI 3

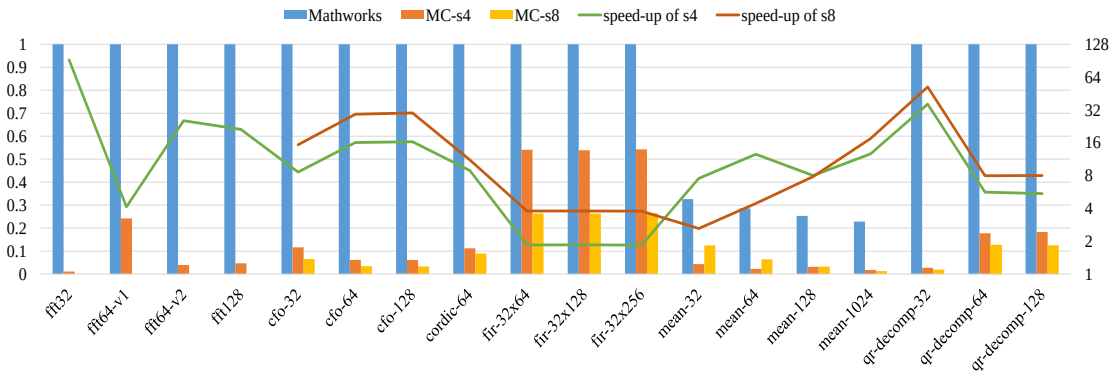

Figure 38: Speed-up comparing with MathWorks compiler on PI 3 using GCC with packed fixed point data

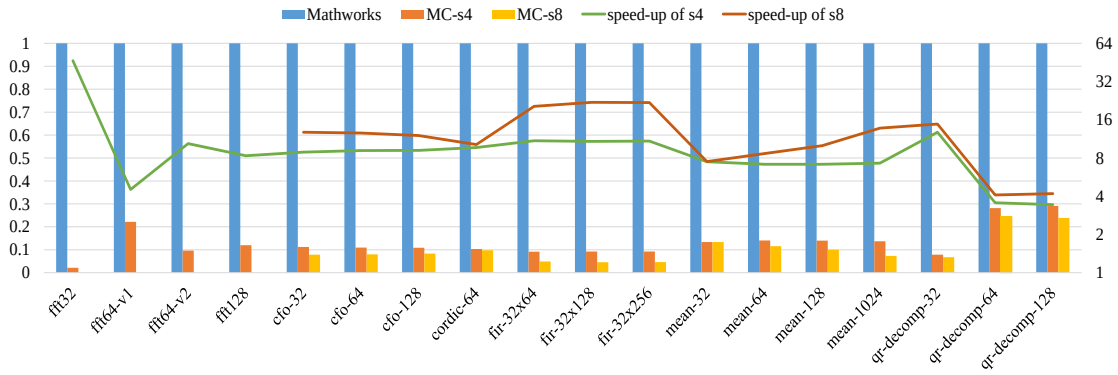

Figure 39: Speed-up comparing with MathWorks compiler on PI 3 using MSVC with packed fixed point data

Performance using unpacked fixed point types Figures 40, 41 and 42 present the normalized execution times (Table 22) of the compiler's vectorized generated code with unpacked fixed point data types compared to the generated code by MathWorks Coder. The speed-up achieved by compiling the generated code by both MATLAB compilers with Clang and executing the output code on Rasbian OS (Fig. 40) is up to 46.8x with average speed-up of 8.2x for SIMD width 4 and up to 24.5x with average speed-up of 7.3x for SIMD width 8. Compiling with GCC and executing the benchmark on Rasbian OS (Fig. 41) the speed-up is obtained by the performance of the vectorized code compared to that of MathWorks generated code is up to 43.2x and 23.6x with 8.2x and 7.2x on average for SIMD width 4 and 8 correspondingly. Finally, when using Windows IoT OS and MSVC compiler (Fig. 42) the speed-up achieved by the performance of the vectorized generated code compared to the performance of MathWorks generated code is between 1.5x-20x with 8.4 on average for SIMD width 4 and from 1.3x up to 25.9x with 12.5x on average for SIMD width of 8.

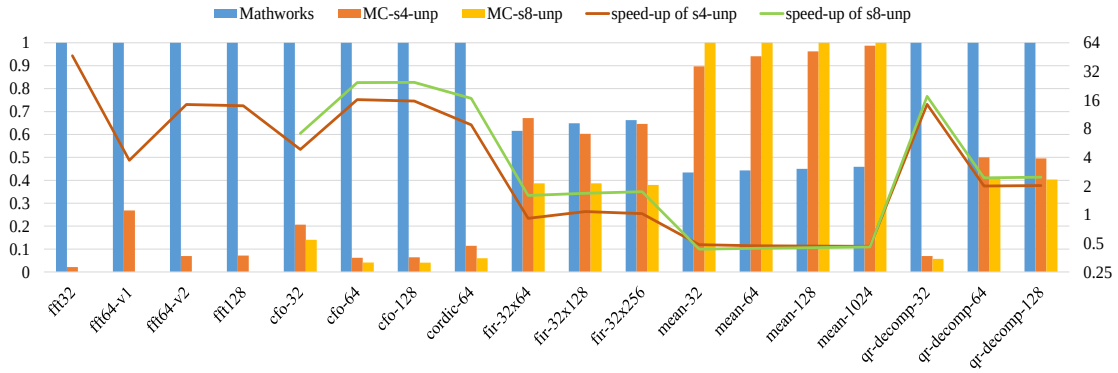

Figure 40: Speed-up comparing with MathWorks compiler on PI 3 using Clang/LLVM with unpacked fixed point data

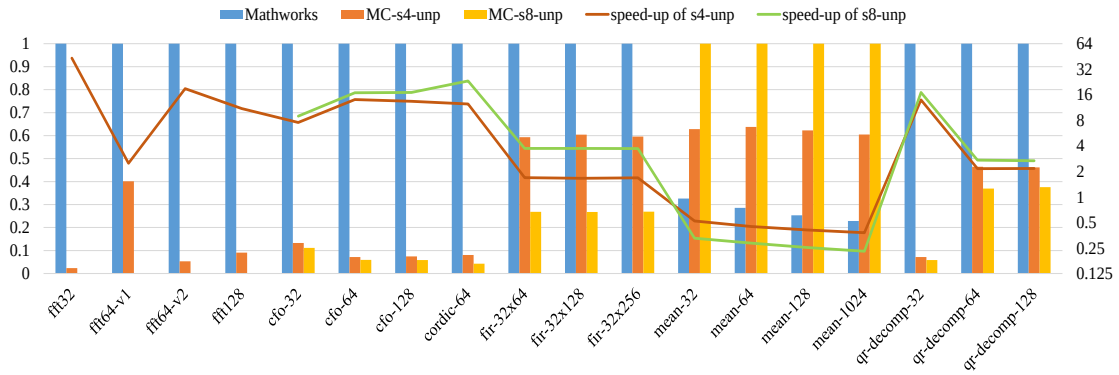

Figure 41: Speed-up comparing with MathWorks compiler on PI 3 using GCC with unpacked fixed point data

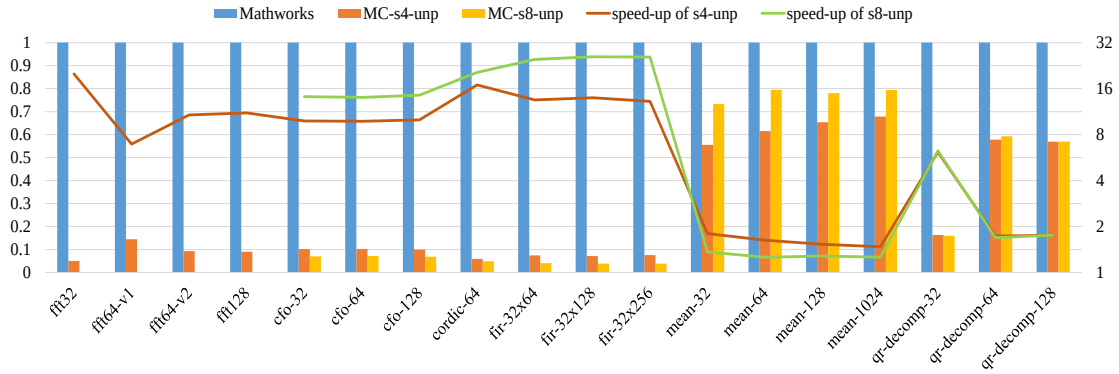

Figure 42: Speed-up comparing with MathWorks compiler on PI 3 using MSVC with unpacked fixed point data

Performance using packed floating point types Figures 43, 44 and 45 show the normalized execution times (Table 22) of the compiler's vectorized generated code with packed (SIMD width 4) floating point data types compared to the MathWorks generated code. The performance of the vectorized generated code compared to the performance of the generated code by MathWorks Coder achieves an average speed-up of 4.6x up to 13.4x compiling with Clang and executing the benchmark on Rasbian OS (Fig. 43). The performance's speed-up achieved by compiling with GCC (Fig. 44) is between 1.5x-18.4x and 6x on average. By using MSVC and executing the output code on Windows IoT OS (Fig. 45) the vectorized generated code obtains maximum speed-up of 5.3x with 2.2x on average.

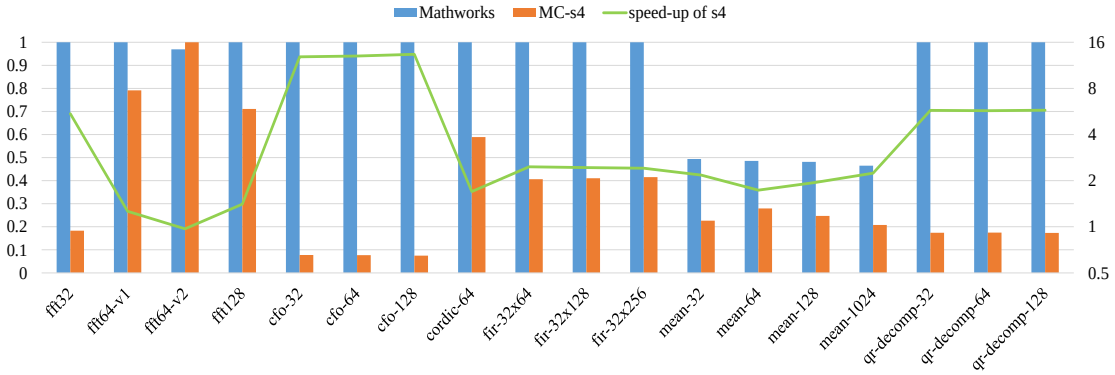

Figure 43: Speed-up comparing with MathWorks compiler on PI 3 using Clang/LLVM with packed floating point data

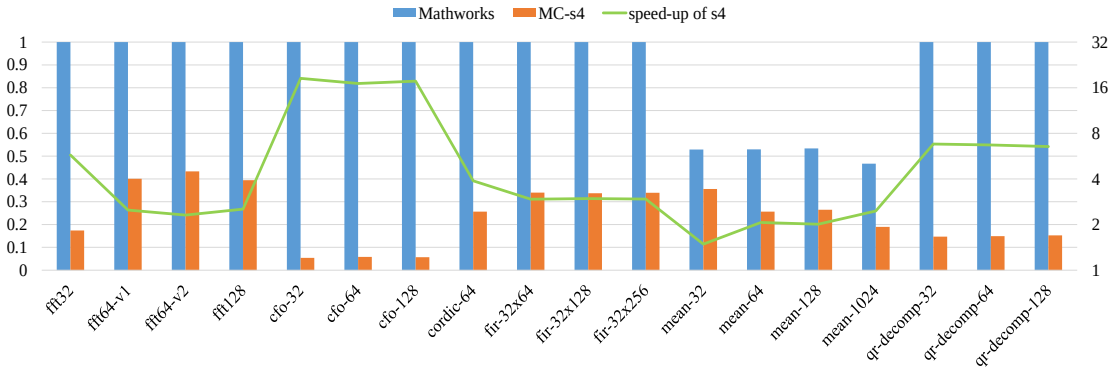

Figure 44: Speed-up comparing with MathWorks compiler on PI 3 using GCC with packed floating point data

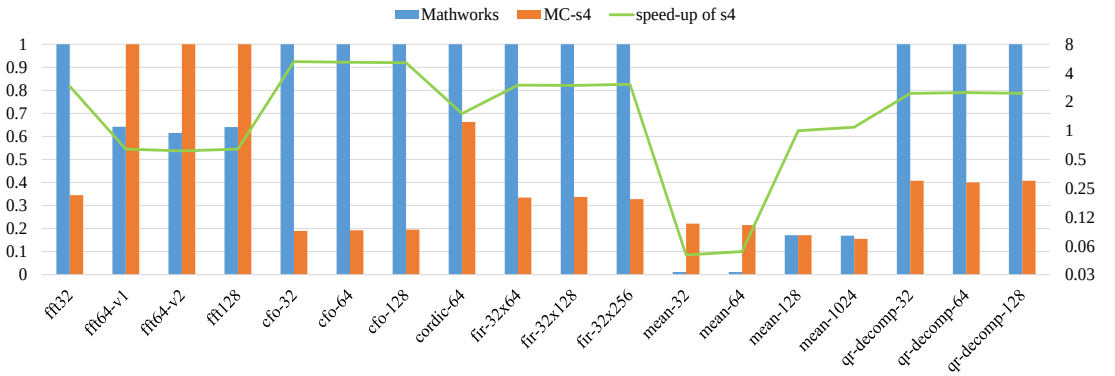

Figure 45: Speed-up comparing with MathWorks compiler on PI 3 using MSVC with packed floating point data

Performance using unpacked floating point types Figures 46, 47 and 48 show the normalized execution times (Table 22) of the vectorized generated code with unpacked of SIMD width 4 floating point types against the MathWorks generated code. The performance of the compiler's vectorized code compared to that of MathWorks generated code achieves maximum speed-up of 15.6x with 3.9x on average using Clang and Rasbian OS (Fig. 46), a maximum speed-up of 16.2x with 4.4x on average compiling with GCC and executing the benchmark on Rasbian OS (Fig. 47) and a maximum speed-up of 6.7x with 1.9x on average compiling with MSVC and executing the benchmark on Windows IoT OS (Fig. 48).

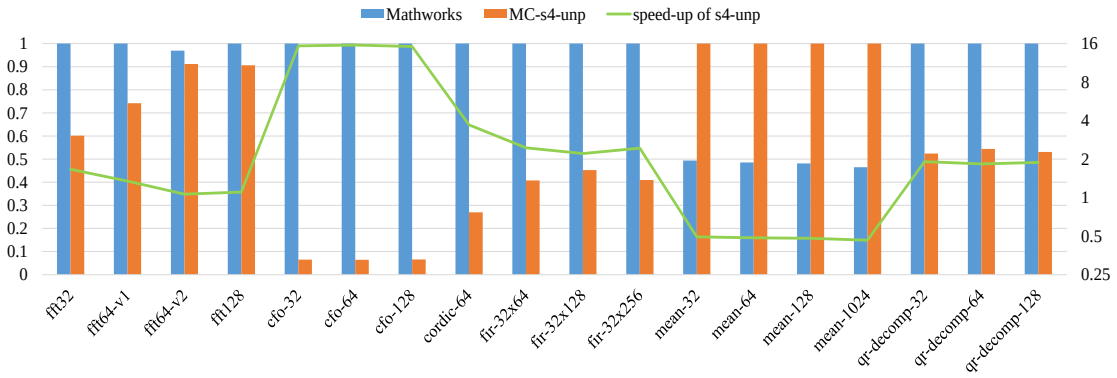

Figure 46: Speed-up comparing with MathWorks compiler on PI 3 using Clang/LLVM with unpacked floating point data

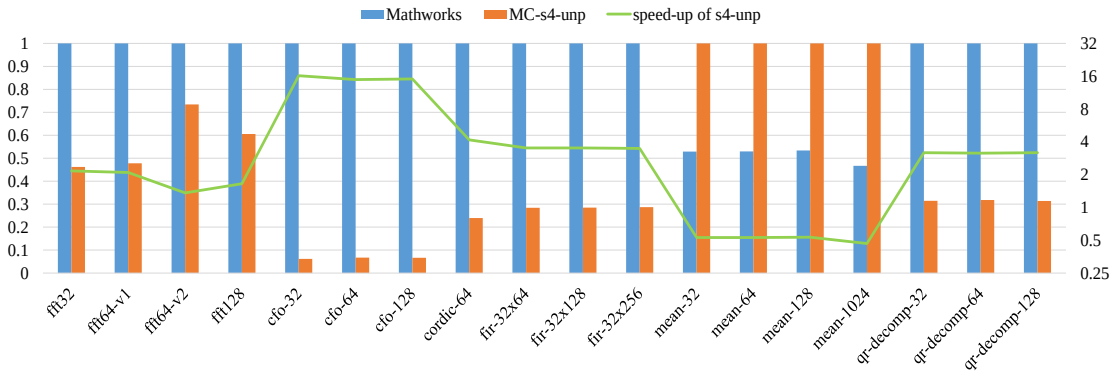

Figure 47: Speed-up comparing with MathWorks compiler on PI 3 using GCC with unpacked floating point data

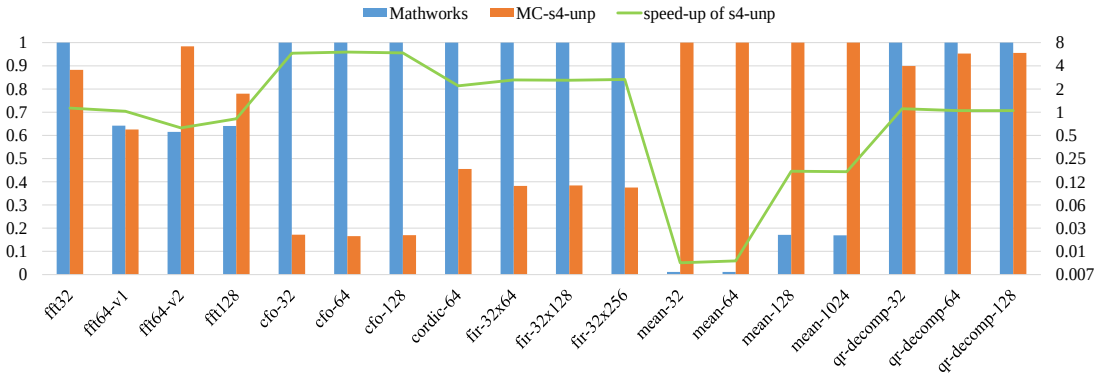

Figure 48: Speed-up comparing with MathWorks compiler on PI 3 using MSVC with unpacked floating point data

Performance among SIMD configurations using fixed point types Figures 49, 50 and 51 show the normalized execution times (Table 23) of the vectorized generated code using various SIMD configurations for the fixed point benchmark. The perfromance of the vectorized code with packed data types compared to that of unpacked data types achieves maximum speed-up of 5.8x with 2.1x on average for SIMD width 4 and maximum speed-up of 10.4x with 2.6x on average for SIMD width 8 using clang and Rasbian OS (Fig. 49). The speed-up of the vectorized code performance with packed data types using GCC on Rasbian OS (Fig. 50) is up to 33.5x with average speed-up of 6.6x for SIMD width 4 and up to 76.5x with average speed-up of 10.6x for SIMD width 8. Finally, the speed-up of the vectorized code performance with packed data types compiling with MSVC and executing the benchmark on Windows IoT OS (Fig. 51) is up to 4.9x and 10.9x with average speed-up of 1.9x and 3.1x for SIMD width 4 and 8 respectively.

	Fig. 49	Fig. 50	Fig. 51	Fig. 52	Fig. 53	Fig. 54
fft32	0.82	0.79	1.40	1.11	0.83	1.50
fft64-v1	4.64	3.95	7.47	4.69	2.88	8.10
fft64-v2	4.24	2.41	6.00	4.15	2.65	6.07
fft128	9.46	10.08	16.20	10.57	7.49	16.03
cfo-32	1.16	0.94	1.43	1.46	1.08	1.77
cfo-64	2.28	1.83	2.70	2.89	2.13	3.60
cfo-128	4.52	3.80	5.37	5.73	4.29	7.20
cordic-64	191.67	97.14	245.97	184.80	66.13	217.70
fir-32x64	99.55	67.22	99.37	84.10	74.62	86.20
fir-32x128	217.63	150.16	221.87	203.93	161.81	190.60
fir-32x256	419.97	307.68	461.27	388.39	338.12	394.70
mean-32	0.62	0.90	1.10	0.49	0.40	0.90
mean-64	1.14	1.71	2.07	0.94	0.75	1.70
mean-128	2.20	3.40	3.90	1.83	1.45	3.50
mean-1024	16.85	26.66	29.13	14.80	12.88	26.97
qr-decomp-32	86.54	79.06	136.87	71.89	45.18	86.93
qr-decomp-64	171.83	159.70	274.93	148.39	91.23	185.63
qr-decomp-128	344.19	315.42	530.70	296.95	181.99	374.77

Table 23: Reference values (exec. time in µs) used for normalization of vectorized results on PI 3

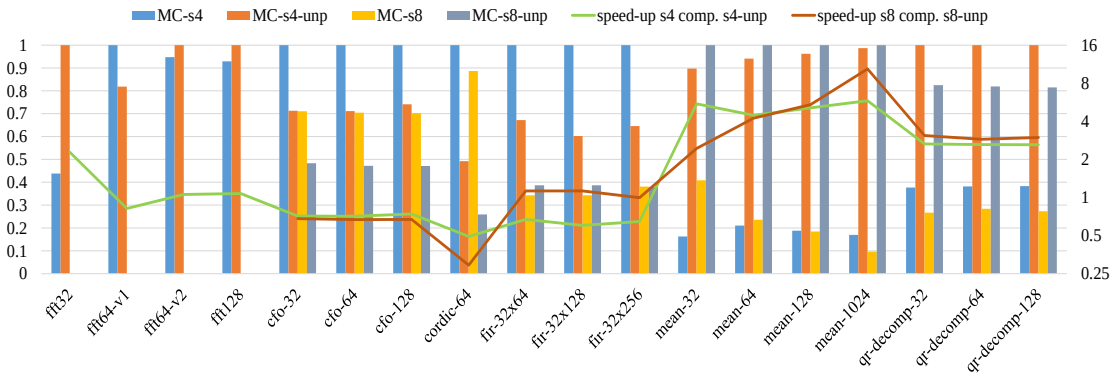

Figure 49: Performance of vectorized code with unpacked fixed point data types versus packed fixed point data types using Clang/LLVM on Raspberry PI 3

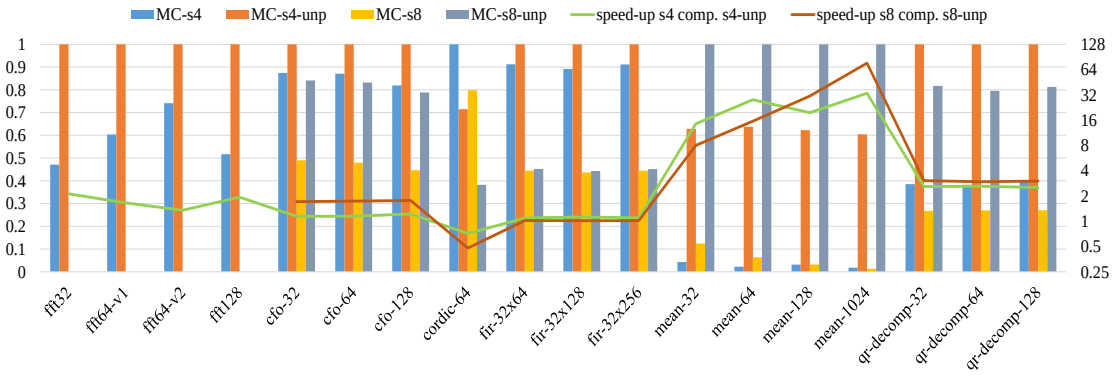

Figure 50: Performance of vectorized code with unpacked fixed point data types versus packed fixed point data types using GCC on Raspberry PI 3

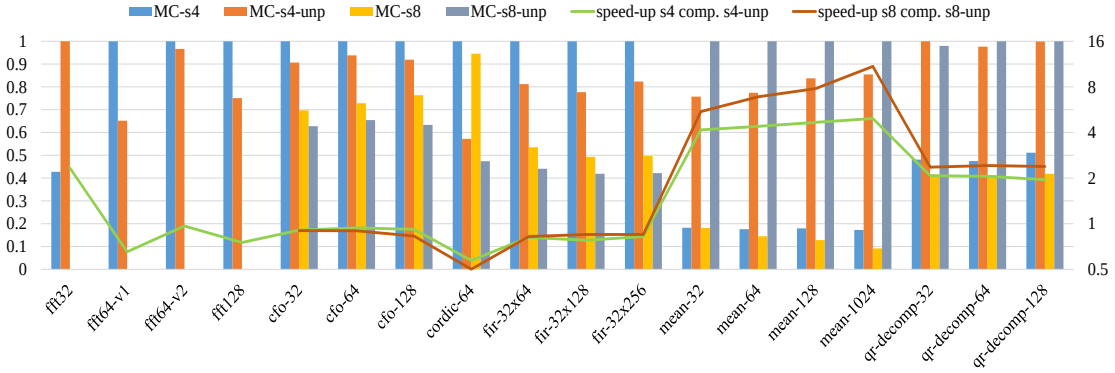

Figure 51: Performance of vectorized code with unpacked fixed point data types versus packed fixed point data types using MSVC on Raspberry PI 3

Performance among SIMD configurations using floating point types Figures 52, 53 and 54 show the normalized execution times (Table 23) of the compiler's vectorized generated code using different SIMD configurations for floating point data types. The performance of the vectorized code with packed data types compared to that with unpacked data types attains a maximum speed-up of 4.8x with 2.1x on average compiling with Clang and executing the benchmark on Rasbian OS (Fig. 52), a maximum speed-up of 5.3x with 2x on average using GCC and Rasbian OS (Fig. 53) and a an average speed-up of 2.2x up to 6.4x compiling with MSVC compiler and executing the output code on Windows IoT OS (Fig. 54).

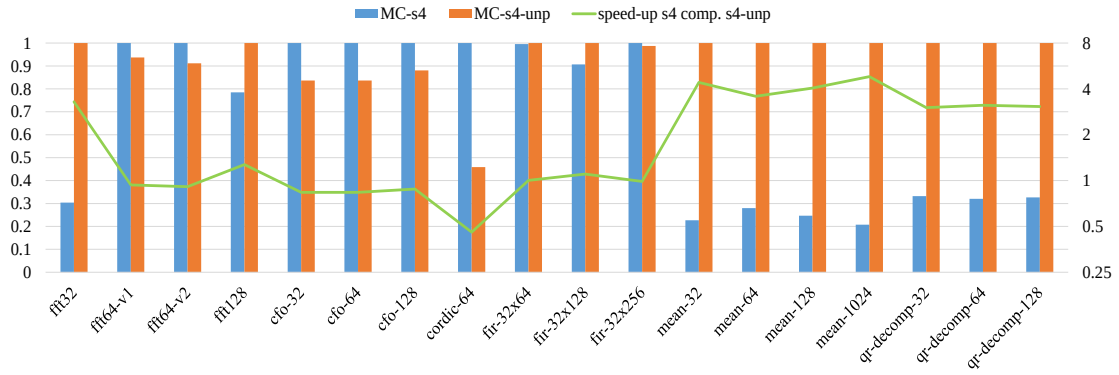

Figure 52: Performance of vectorized code with unpacked floating point data types versus packed floating point data types using Clang/LLVM on Raspberry PI 3

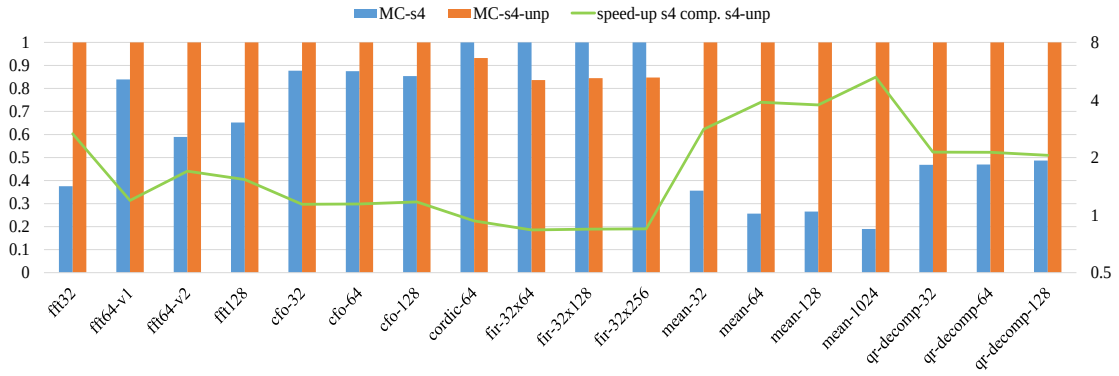

Figure 53: Performance of vectorized code with unpacked floating point data types versus packed floating point data types using GCC on Raspberry PI 3

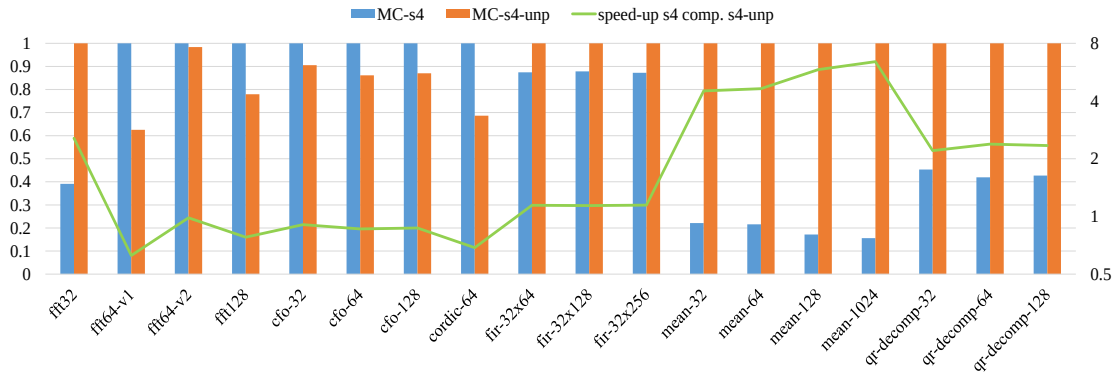

Figure 54: Performance of vectorized code with unpacked floating point data types versus packed floating point data types using MSVC on Raspberry PI 3

Discussion of performance among the different experiments The highest speed-up achieved across the different configurations and the test cases which have obtained slow up (instead of speed-up) are same for the experiments on PI 2 and PI 3. The observations based on the results on Raspberry PI 2 (diagrams of subsection 6.3.2) are similar for the results of Raspberry PI 3 (diagrams of the current subsection). However, there are some differences across the corresponding experiments between Raspberry PI 2 and PI 3 which are quoted below:

- MathWorks generated code versus vectorized code comparison of packed fixed point data types using Clang: The vectorized code using packed data types of SIMD width 4 attains better performance than the MathWorks generated code for FIR application only on PI 2.
- MathWorks generated code versus vectorized code comparison of packed fixed point data types using MSVC: The speed-up achieved for CFO and CORDIC applications by the vectorized code compared to MathWorks generated code, is same when using SIMD width 4 or 8 on PI 2. However, the speed-up of packed data types with SIMD width 8 is higher than the speed-up achieved with SIMD width 4 for these applications on PI 3.
- MathWorks generated code versus vectorized code comparison of unpacked fixed point data types using MSVC: The performance of vectorized code with unpacked data types with SIMD width 4 is better than the performance of MathWorks generated code for FIR application on PI 2. However, there isn't such a speed-up on PI 3.
- MathWorks generated code versus vectorized code comparison of unpacked fixed point data types using MSVC: The vectorized code achieves significant performance compared to the MathWorks generated code for the MEAN application only on PI 3.
- Packed data types versus unpacked data types comparison: The below benchmarks/applications obtain better performance using unpacked data types of SIMD width 4 on PI 2 but worse performance on PI 3, comparing with the performance of the vectorized code with packed data types of SIMD width 4.

    - Clang - FFT-128 application with floating point data types
    - MSVC - FIR application with floating point data types
    - Clang - FFT-64-v2 and FFT-128 benchmark with floating point data types
    - MSVC - CFO application with fixed point data types

6.4 Results from x86 Architectures

This section discusses the performance of the code generated by the compiler, compared to the performance of the code generated by the MathWorks Coder on two different x86 processors. For the compilation of generated code by both MATLAB compilers the Clang/LLVM, GCC and MSVC auto-vectorizing C compilers have been used. The benchmark has been executed on the x86-based desktops using Linux Ubuntu and Windows 10 operating systems.

6.4.1 Presentation of x86 Architectures

The x86 architecture [x86 reference manual, 2016] is a complex instruction set (CISC) architecture series for computer processors developed by INTEL [Intel, 2016] Corporation. The x86 architecture consists of variable length instructions and supports backward compatibility with previous generation x86 microarchitectures. Words are stored in memory in little-endian byte order and the largest native size of memory addresses depends on the architecture generation (16,32 or 64 bits), however memory segmentation is supported allowing programs to use larger (than native size) segments of memory. Most of the x86 instructions are 2 or 3 bytes, although some instructions are much longer, and some are single-byte. The instruction operands can be immediate values or references to memory or registers with the restriction that only one memory reference can be included in a instruction. The registers of the x86 vary depending on the generation architecture. Generally, all x86 processors include a set of registers for: general purpose, segment registers holding the segment address of various items, indexes and pointer registers with each of them having a specific function and the EFLAGS register which holds the state of the processor and miscellaneous/special purpose registers. Early generations of x86 architecture could optionally include floating-point hardware as external processor's component, but modern x86 processors integrate the floating point co-processor on chip. The FPU contains registers of 80 bits width and stores numbers in the IEEE floating-point standard double extended precision format. The FPU arithmetic, consists of numerical floating point operations, transcendental functions, including trigonometric and exponential functions, as well as instructions that load common constants such as the base of the natural logarithm.

The x86 architecture provides a 64-bit version of its instruction set, named x64, increasing the amount of virtual and physical memory. The x64 architecture provides 64-bit general purpose registers increasing their number as well. Specifically, the general purpose and XXM (SSE) registers are increased from 8 to 16 keeping more variables locally, rather than on the stack.

The x86 architecture provides advanced technologies to improve performance such as pipelining and superscalar execution, as well as SIMD extensions. For superscalar

execution, the x86 processors decodes most instructions to micro-operations (pipelining). Then, a control unit operates in compliance with x86-semantics so that they are partly executed in parallel (out-of order execution is also supported) by using the processor's specialized execution units. Regarding the SIMD extension, the x86 processors provide Streaming SIMD Extensions (SSE) and Advanced Vector Extensions (AVX) technologies which are discussed below.

Streaming SIMD Extensions (SSE) Streaming SIMD Extensions (SSE) [SSE, 2016] is an SIMD instruction set extension to the x86 architecture, designed by Intel Corporation and introduced in Pentium III series processors. The extension contains instructions of single precision floating point and sharing registers (MMX registers) with main CPU unit. The shared registers between CPU and SSE prevents the simultaneous processing of floating point data on FPU and SIMD data on SSE. Newer generation processors introduce the SSE2 extension which includes a new supplementary instruction set and replaces the MMX registers with new registers (XMM registers), dedicated to the SIMD unit. SSE3, SSSE3 and SSE4 subsequently introduced to enhance the SSE instruction set introducing double precision floating point instructions as well. SSE extension consists of 128-bit registers providing parallel processing of 4 32-bit single-precision floating point numbers. The SSE2, SSE3, SSE3 and SSE4 extensions provides operations on:

- 2 64-bit double precision floating point numbers
- 2 64-bit long integers
- 4 32-bit integers
- 8 16-bit short integers
- 16 8-bit bytes

AVX Advanced Vector Extensions (AVX) [AVX, 2016] is an extension of x86 architecture to provide SIMD processing of 256-bit length. The extension was developed by Intel and AMD Corporations and was firstly supported by Intel in the Sandy Bridge microarchitecture. Supplementary extensions of AVX are the AVX2 (firstly equipped in Haswell microarchitecture) which expands most integer commands to 256 bits and introduces Fused multiply–add (FMA) operation as well as the AVX-512 which extends the vector's width to 512-bit. The expansion of vector's width to the double size, allows for a maximum parallel processing of 4 64-bit double precision floating point numbers (or 8 single precision floating point) and a maximum parallel processing of 8 32-bit integers (16 16-bit short integers),

Programming with SSE/AVX and Visual Studio Environment The definitions of SIMD semantics such as the vector data types and the intrinsic functions are included in the *'xmmintrin.h'*, *'emmintrin.h'* and *'immintrin.h'* header files. The files

include the unions which represent the floating point and integers vectors of 128-bit and 256-bit length. In detail, the *'xmmintrin.h'* (SSE) contains the *'__m128'* union which has been used for the representation of single precision floating point with SIMD width 4 vectors. The *'immintrin.h'* header file (AVX) includes the *'__m256'* union, which has been used for the representation of single precision floating point with SIMD width 8 vectors. Finally, the *'emmintrin.h'* includes the *'__m128i'* union which has been deployed for the representation of the integer 128-bit vectors both for SIMD width 4 (32-bit short integers) and SIMD width 8 (16-bit short integers). The integer vectors are used for the representation of fixed point data types.

Experimental Setup of Visual Studio Environment Although the Intel processors that have been deployed for the experiments implement the x64 architecture, Visual Studio (MSVC compiler) has been configured to produce x86 code. The results of the below sections include benchmark's execution times of x86 code. Moreover, the C code has been compiled using *'arch:AVX'* setting to enable the production of AVX SIMD instructions (auto-vectorization is implemented using AVX instructions). However, results of x64 code and compiling with *'arch:SSE2'* are presented in the appendix.

INTEL Sandy Bridge Sandy Bridge is the Intel's second generation microarchitecture of Intel Core processor family. The processors are distinguished from previous-generation Intel processors by featuring all cores (including graphics) on a single chip. Sandy Bridge also introduces the Intel Turbo boost technology 2.0 which enables the processor to run above its base operating frequency via dynamic control of the processor's clock rate. The enhanced memory channel allows the execution of two load/store operations per CPU cycle. Furthermore, Sandy bridge processors may reach up to eight physical cores or 16 logical cores through Hyper-threading. The SIMD extensions supported by Sandy Bridge are: SSE, SSE2, SSE3, SSSE3, SSE4, SSE4.1, SSE4.2 and AVX.

INTEL Ivy Bridge Ivy Bridge is the third generation microarchitecture of processors developed by Intel. Ivy bridge mostly differentiate from Sandy Bridge as to the CMOS semiconductor device fabrication scaling from 32nm to 22nm. Ivy Bridge includes also improved graphic capabilities and processor's peripheral enhancements. The SIMD extensions supported by Ivy Bridge are: SSE, SSE2, SSE3, SSSE3, SSE4, SSE4.1, SSE4.2 and AVX as well.

x86-based desktops characteristics Two x86 based desktops have been used for experimentation consisting of the 64-bit (x64) version of x86 architecture. The first of them includes a i7-3820 [i7-3820 Processor, 2016] Sandy Bridge processor with 10Mb cashe and 8Gb of DDR3 RAM. The chip consists of 4 cores (8 threads) and operates on 3.6Ghz up to 3.8Ghz using the Turbo boost technology. The latter desktop which have

been used in the experiments is a i7-3770 [i7-3770 Processor, 2016] Ivy Bridge processor with 8Mb cashe and 16Gb of DDR3 RAM. The chip consists of 4 cores (8 threads) and operates on 3.4Ghz up to 3.9Ghz using the Turbo boost technology. Both desktops have been used for experiments operating on Linux Ubuntu 16.04 and Windows 10 OS.

6.4.2 Performance of Generated Code on Intel Sandy Bridge (i7-3820)

Performance using packed fixed point types This subsection presents the performance of the vectorized generated code compared to the performance of the generated code by MathWorks Coder on i7-3820 Sandy Bridge processor. Figures 55, 56 and 57 present the normalized execution times (Table 24) of the vectorized generated code with packed fixed point data types compared to the MathWorks generated code using Clang, GCC and MSVC respectively. The speed-up achieved by the performance of the vectorized generated code compiling with Clang and executing the benchmark on Linux Ubuntu OS is up to 154.6x with 13x on average using SIMD width 4 and up to 51.9x with 9.9x on average using SIMD width 8. Compiling with GCC, the speed-up achieved by the performance of the vectorized generated code compared to that of MathWorks generated code is from 1.5x up to 137.7x with 17x on average speed-up using SIMD width 4 and up to 39.6x with 12.2x on average using SIMD width 8. The speed-up achieved by the vectorized code compiling with MSVC and executing the benchmark on Windows 10 OS is from 3.3x up to 67.9x with 11x on average for SIMD width 4 and from 2x up to 17.3x with 10.5x on average for SIMD width 8.

Discussion of performance using packed fixed point types The average (and maximum) speed-up of benchmark is higher with SIMD width 4 than 8. This is due to the significant speed-up of FFT applications which take part only in experiments using SIMD width of 4. These applications affect the benchmark's average speed-up, by increasing the average value in relation with the value of average speed-up using SIMD width 8. The maximum speed-up achieved by the three compilers using SIMD width 4 is the FFT-32 application while CFO, QR-decomposition and FIR applications obtain the maximum speed-up with SIMD width 8 compiling with Clang, GCC and MSVC respectively. Compiling the generated code by both MATLAB compilers with Clang, the MathWorks generated code attains better performance than vectorized generated code with SIMD width 4 for FIR application. In addition, performance of the vectorized code for the QR-decomposition and MEAN of 32 and 64 input length obtain higher speed-up against the MathWorks generated code with SIMD width 4 than SIMD width 8. Using MSVC for the compilation of the generated code by both MATLAB compilers, the performance of the vectorized code achieves speed-up over the whole range of the benchmarks. However, the performance with SIMD width 8 for CFO-32, CORDIC,

QR-decomposition and MEAN (with input streams of 32,64,128) benchmarks is same or even worst compared to the performance using SIMD width 4. The performance of vectorized code compiling with GCC is better across the benchmark compared to that of MathWorks generated code except at MEAN-32. Additionally, for that benchmark and MEAN-64, the vectorized code with SIMD width 4 achieves better performance compared to the vectorized code with SIMD width 8.

	Fig. 55, Fig. 58	**Fig. 56, Fig. 59**	**Fig. 57, Fig. 60**	**Fig. 61, Fig. 64**	**Fig. 62, Fig. 65**	**Fig. 63, Fig. 66**
fft32	10.95	10.13	4.40	0.19	0.92	0.30
fft64-v1	6.13	4.32	3.80	1.36	2.06	196.92
fft64-v2	15.32	14.01	6.12	2.12	2.56	2.44
fft128	29.77	30.40	16.96	2.93	5.46	4.56
cfo-32	1.40	1.53	4.48	1.79	4.42	4.52
cfo-64	8.22	7.45	5.96	3.70	7.75	5.34
cfo-128	14.21	16.98	11.16	6.74	14.77	8.42
cordic-64	84.38	202.44	406.92	44.28	44.17	44.76
fir-32x64	15.74	25.16	104.30	15.66	20.97	14.62
fir-32x128	32.32	46.10	236.14	25.63	35.40	31.02
fir-32x256	59.88	83.03	492.72	49.34	70.25	64.72
mean-32	0.62	1.28	0.30	0.67	1.02	0.30
mean-64	0.86	2.26	0.50	0.86	1.49	0.50
mean-128	2.13	3.86	1.20	2.20	3.65	1.10
mean-1024	12.15	19.95	9.96	10.95	20.11	9.32
qr-decomp-32	209.50	221.95	151.62	35.65	40.74	37.96
qr-decomp-64	61.10	62.09	84.26	59.82	71.36	62.10
qr-decomp-128	112.70	117.99	168.40	115.95	134.57	123.56

Table 24: Reference values (exec. time in µs) used for normalization of results on i7-3820

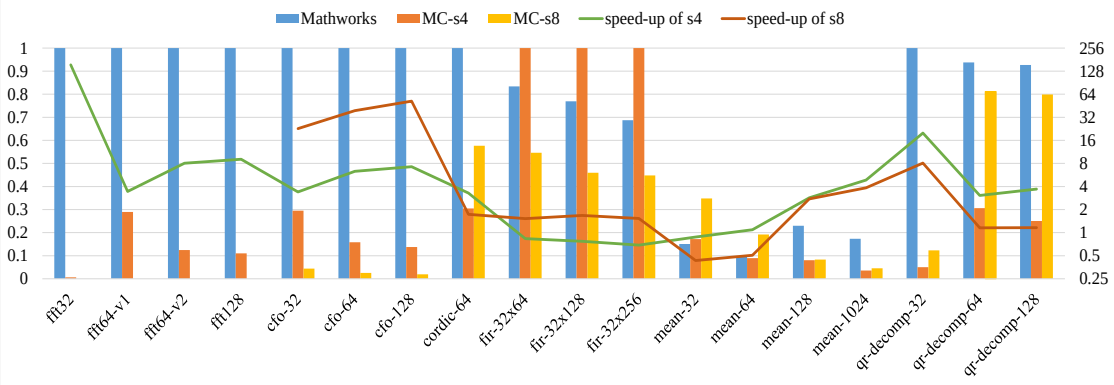

Figure 55: Speed-up comparing with MathWorks compiler on i7-3820 using Clang/LLVM with packed fixed point data

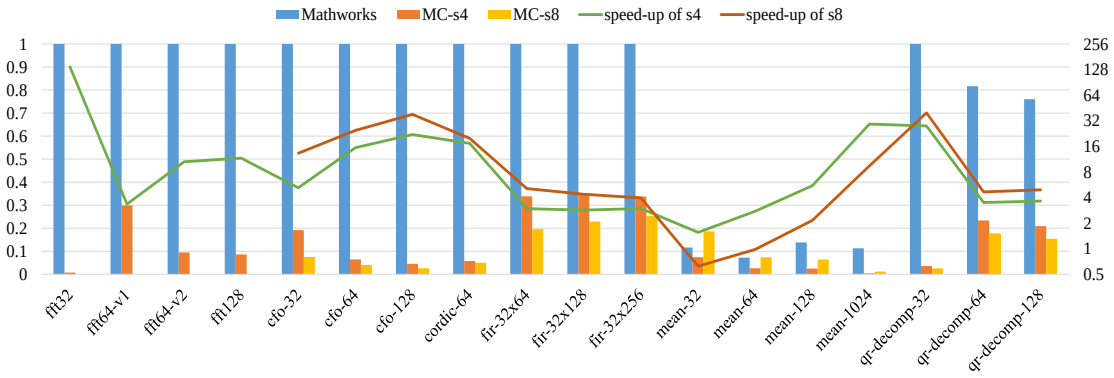

Figure 56: Speed-up comparing with MathWorks compiler on i7-3820 using GCC with packed fixed point data

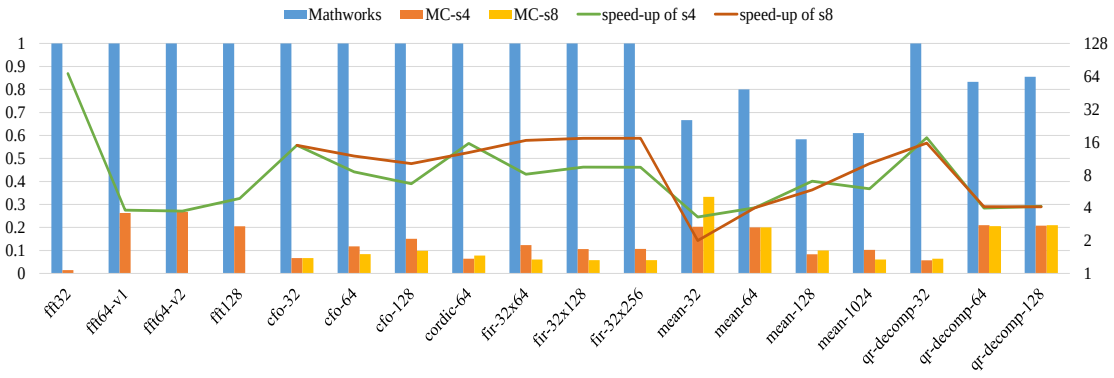

Figure 57: Speed-up comparing with MathWorks compiler on i7-3820 using MSVC with packed fixed point data

Performance using unpacked fixed point types Figures 58, 59 and 60 show the normalized execution times (Table 24) of the vectorized generated code with unpacked fixed point data types, compared to generated code by MathWorks Coder using Clang, GCC and MSVC, respectively. The performance of the vectorized generated code obtains maximum speed-up of 93.3x with 9.9x on average for SIMD width 4 and up to 36.4x with average speed-up of 7.3x for SIMD width 8 compiling the generated code by both MATLAB compilers with Clang and executing the output code on Linux Ubuntu OS. Using GCC, the performance of the vectorized code compared to that of MathWorks generated code achieves maximum speed-up of 24.8x and 24.9x with 6.6x and 7.3x on average for SIMD width 4 and 8 respectively. The speed-up of vectorized generated code against MathWorks generated code compiling with MSVC and executing the benchmark on Windows 10 OS is up to 28.9x with 8.4x on average using SIMD width 4 and up to 44.8x with 13.4x on average using SIMD width 8.

Discussion of performance using unpacked fixed point types The performance of the vectorized code using SIMD width 4 achieves the maximum speed-up compared to the performance of the MathWorks generated code for FFT-32 application by compiling with Clang and for CORDIC application by compiling with GCC and MSVC. The CFO application attains the maximum speed-up across the benchmark by the three compilers using SIMD width 8. The MathWorks generated code achieves a considerably better performance against the vectorized code for MEAN application by compiling the generated code with Clang and GCC. Furthermore, the performance of the MathWorks generated code for QR-decomposition with input streams of 2x2x64 and 2x2x128 achieves a lower speed-up compared to the performance of the vectorized generated code by compiling with GCC. Similar speed-up of MathWorks generated code for that application is obtained by compiling with Clang using only for SIMD width 4. Using for compilation the MSVC compiler, the generated code by MathWorks obtains substantially better performance against vectorized code for MEAN as well.

Explanation of performance using unpacked fixed point types The worse performance of vectorized code against the MathWorks generated code for MEAN application proves that packing and unpacking operations considerably costs in time for x86 architecture inserting overheads that significantly slow up the vectorized code performance and cannot be eliminated by the SIMD processing.

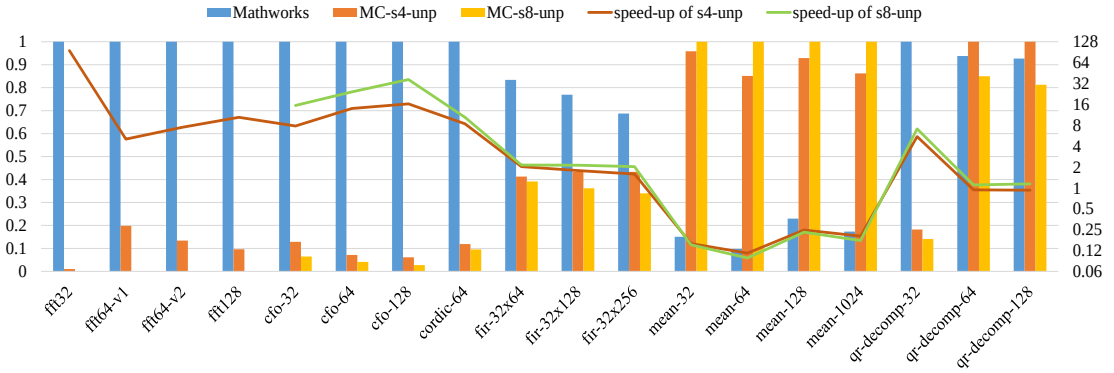

Figure 58: Speed-up comparing with MathWorks compiler on i7-3820 using Clang/LLVM with unpacked fixed point data

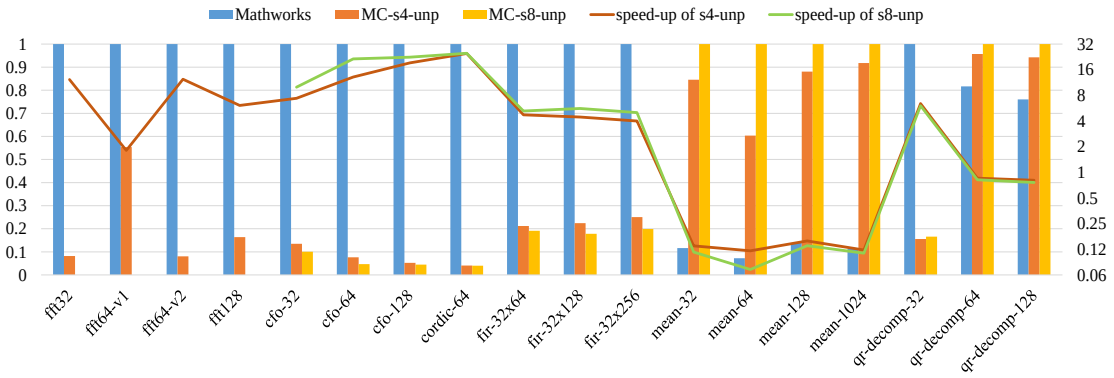

Figure 59: Speed-up comparing with MathWorks compiler on i7-3820 using GCC with unpacked fixed point data

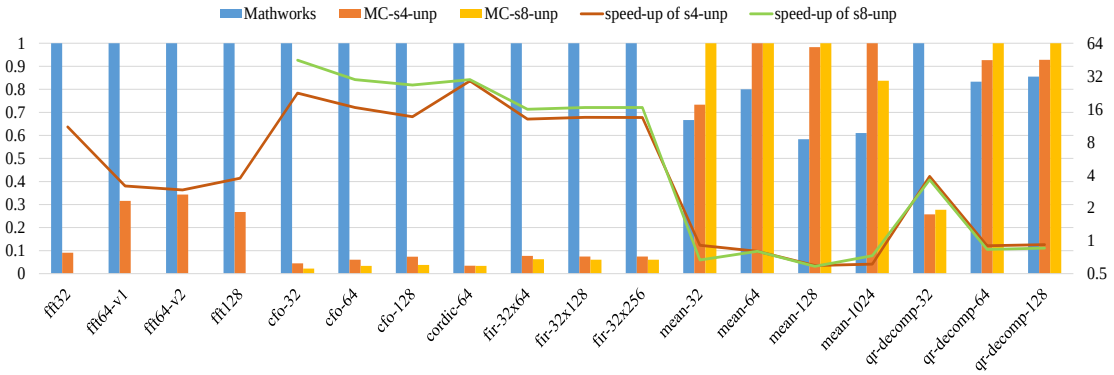

Figure 60: Speed-up comparing with MathWorks compiler on i7-3820 using MSVC with unpacked fixed point data

Performance using packed floating point types Figures 61, 62 and 63 present the normalized execution times (Table 24) of vectorized generated code with packed floating point data types compared to the generated code by MathWorks Coder. The performance's speed-up achieved by compiling the generated code by two MATLAB compilers with Clang and executing the output code on Linux Ubuntu OS is up to 4.8 with 1.7x on average using SIMD width 4 and up to 4.7x with 2.1x on average using SIMD width 8. The performance of the vectorized code by compiling with GCC obtains maximum speed-up of 6.5x with 2.1x on average using SIMD width 4 and 6.6x with 2.4x on average using SIMD width 8. Using MSVC for the compilation of the benchmark and executing on Windows 10 OS, the speed-up obtained is up to 164.1x with average speed-up of 11.9x using SIMD width 4 and up to 15.1x with average speed-up of 4.5x using SIMD width 8.

Discussion of performance using packed floating point types The maximum speed-up achieved for floating point data types using different SIMD width and C compilers vary. The performance of the vectorized generated code compiled with clang obtains a maximum speed-up for FFT-32 using SIMD width 4 and a maximum speed-up for MEAN-1024 using SIMD width 8. The maximum speed-up of vectorized code compiling with GCC achieved for CFO and the maximum speed-up of the vectorized code over the benchmark by compiling with MSVC achieved for FFT64-v1 using SIMD width 4 and for CFO using SIMD width 8. For several benchmarks the MathWorks generated code obtains better performance compared to vectorized generated code. For the majority of the configurations, CORDIC application achieves better performance compiling with MathWorks Coder than generating vectorized code. However, the vectorized code of that application using SIMD width 4 and compiling with GCC and MSVC attains significant speed-up against MathWorks generated code. MEAN application for small input streams using various configurations achieves better performance with MathWorks generated code than vectorized code as well. Finally, the vectorized code of FFT-64 and FFT-128 cannot be exploited efficiently, resulting on deterioration of the performance, comparing with the MathWorks generated code. Concerning the SIMD width, CORDIC application (and MEAN-32 using Clang and MSVC) achieves better performance by using SIMD width 4.

Explanation of performance using packed floating point types Considering that CORDIC application includes data and control flow dependencies, the compiler cannot vectorize that segments of code and it produces scalarized code composed by extracting and inserting data from/to vectors for each operation/condition. Therefore, the results regarding the performance of the vectorized code at CORDIC application show that packing and unpacking operations cause significant overhead for the x86 architecture and specifically the targeted processor.

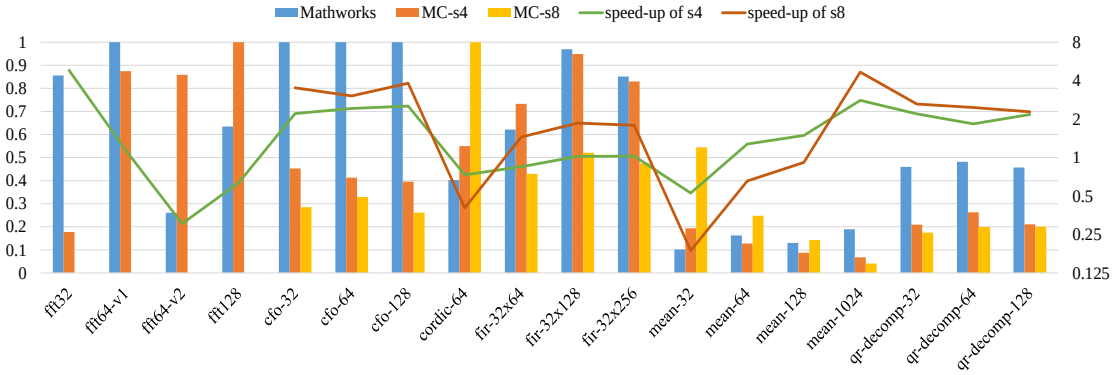

Figure 61: Speed-up comparing with MathWorks compiler on i7-3820 using Clang/LLVM with packed floating point data

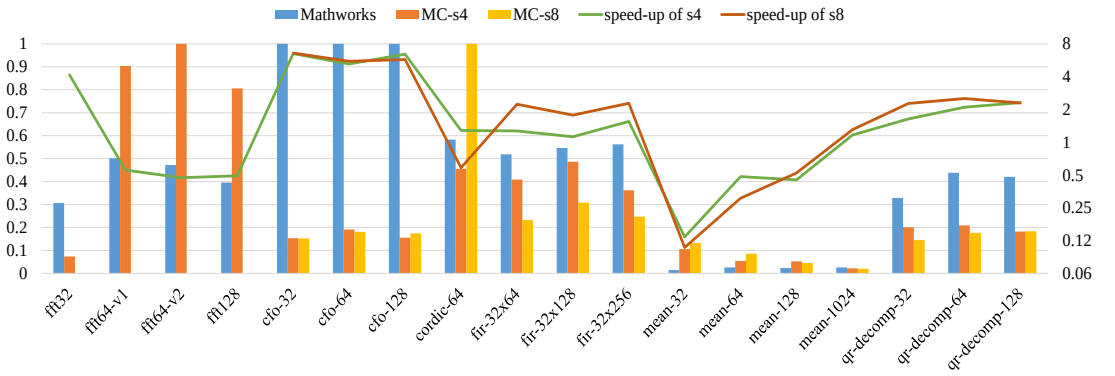

Figure 62: Speed-up comparing with MathWorks compiler on i7-3820 using GCC with packed floating point data

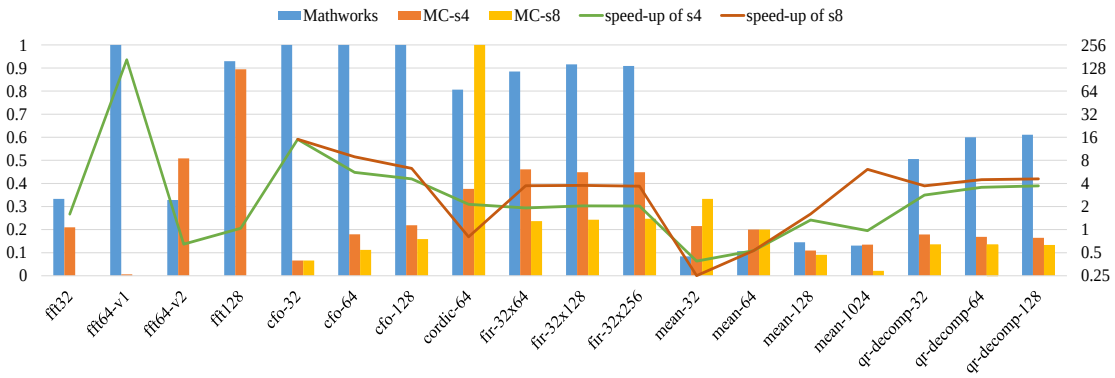

Figure 63: Speed-up comparing with MathWorks compiler on i7-3820 using MSVC with packed floating point data

Discussion of performance using unpacked floating point types Figures 64, 65 and 66 present the normalized execution times (Table 24) of the vectorized generated code with unpacked floating point data type compared to that of MathWorks code. The compiler's vectorized code compiling with Clang and executing the benchmark on Linux Ubuntu OS achieves a maximum speed-up of 8.2x with 1.7x on average using SIMD width 4 and up to 7.4x with 2x on average using SIMD width 8. The performance of the vectorized code against that of MathWorks generated code using the GCC compiler attains average speed-up of 2.6x up to 15.1x using SIMD width 4 and average speed-up of 3.3x up to 15.7x using SIMD width 8. Compiling with MSVC and executing on Windows 10 OS, the performance of the vectorized code achieves an average speed-up of 10.9x up to 142x using SIMD width 4 and average speed-up of 2.9x up to 15x using SIMD width 8.

Discussion of performance using unpacked floating point types The maximum performance's speed-up of the vectorized code against MathWorks generated code achieved for CFO application using Clang and GCC. When compiling the generated code by both MATLAB compilers with MSVC, the maximum speed-up of vectorized code obtained for FFT64-v1 application using SIMD width 4 and for CFO application using SIMD width 8. For the unpacked floating point data types on the current targeted processor, the majority of applications such as FFT-32, MEAN and QR-decomposition results to worse performance for the vectorized generated code compared to the MathWorks generated code. Furthermore, all benchmarks with SIMD width 4 achieve faster execution times compared to that of SIMD width 8 compiling the vectorized code with Clang and GCC. Finally, the vectorized code compiling with MSVC achieves same performance using any of the two SIMD widths.

Explanation of performance using unpacked floating point types Results indicate that packing and unpacking operations (especially involving floating point data types) cause considerable overhead, which deteriorates the overall performance of vectorized code. Thus, SIMD processing with unpacked data types for the current architecture isn't always efficient and in many cases the MathWorks scalarized code is exploited/executed more efficiently.

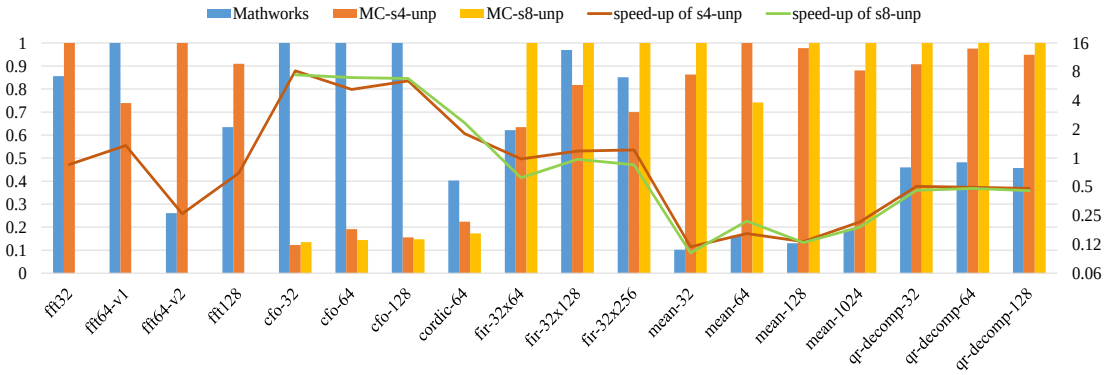

Figure 64: Speed-up comparing with MathWorks compiler on i7-3820 using Clang/LLVM with unpacked floating point data

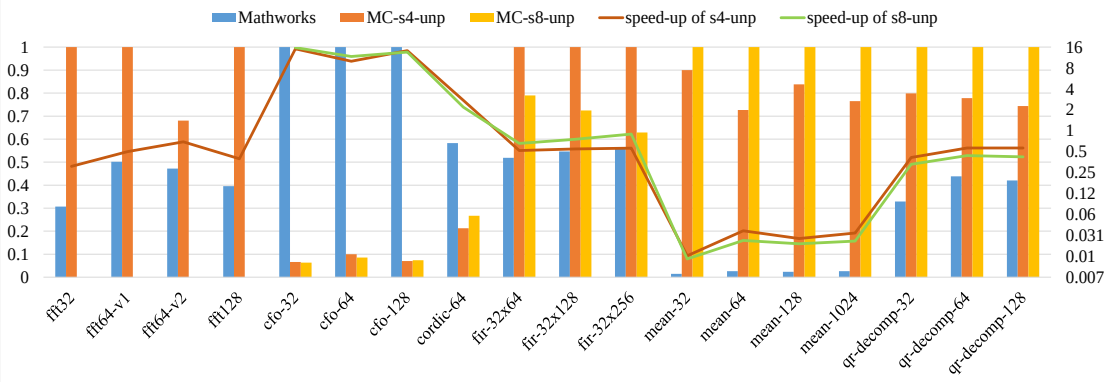

Figure 65: Speed-up comparing with MathWorks compiler on i7-3820 using GCC with unpacked floating point data

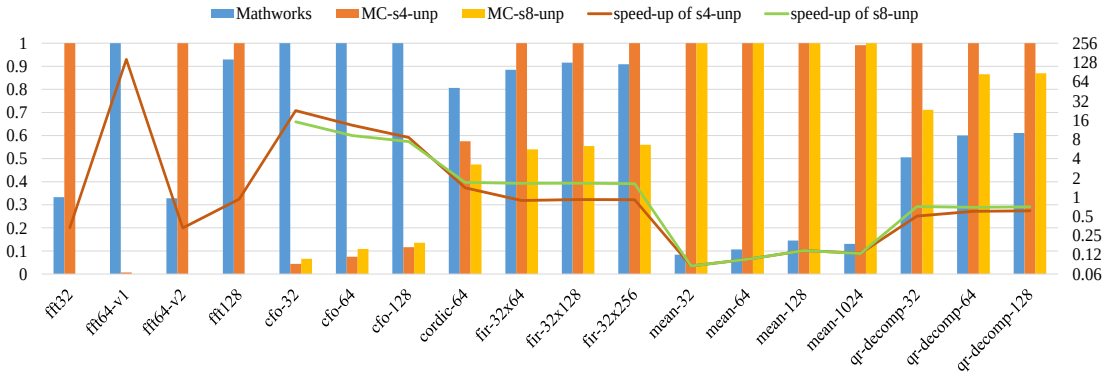

Figure 66: Speed-up comparing with MathWorks compiler on i7-3820 using MSVC with unpacked floating point data

Performance among SIMD configurations using fixed point types Figures 67, 68 and 69 show the normalized execution times (Table 25) of the vectorized generated code with different SIMD configurations for the benchmark with fixed point data types. The performance of vectorized code with packed data types compared to that with unpacked data types achieves a maximum speed-up of 24.1x with 3.8x on average using SIMD width 4, and maximum speed-up of 22.2x with 3.7x on average using SIMD width 8 by compiling with Clang and executing the benchmark on Linux Ubuntu OS (Fig. 67). The performance's speed-up of packed against unpacked data types compiling with GCC (Fig. 68) is up to 238.4x with 19x on average using SIMD width 4 and up to 83.5x with 10.3x on average using SIMD width 8. Compiling with MSVC and executing the output code on Windows 10 OS, the performance of the vectorized code (Fig. 69) with packed data types compared to that with unpacked data types achieves a maximum speed-up of 11.8x with 3.2x on average using SIMD width 4 and maximum speed-up of 13.9x with 3.6x on average using SIMD width 8.

Discussion of performance among SIMD configurations using fixed point types The CORDIC and FIR applications achieve faster execution times using unpacked data types with any of the available configurations except when using the MSVC compiler with SIMD width of 8. Moreover, FFT-64-v1, FFT-128 and CFO achieve better performance with unpacked data types compared to that of packed data types using SIMD width 4 and compiling with Clang. Compiling the vectorized code with GCC, FFT-64-v2 and CFO-32 benchmarks attains better performance with unpacked data types against packed data types using SIMD width 4. Finally, CFO application achieve faster execution times with unpacked data types compiling with MSVC.

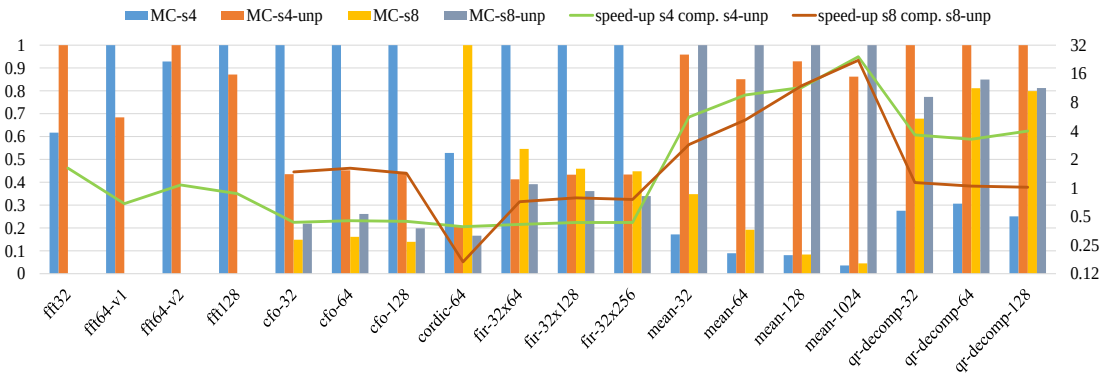

Figure 67: Performance of vectorized code with unpacked fixed point data types versus packed fixed point data types using Clang/LLVM on desktop with i7-3820

	Fig. 67	Fig. 68	Fig. 69	Fig. 70	Fig. 71	Fig. 72
fft32	0.11	0.83	0.40	0.19	0.92	0.30
fft64-v1	1.78	2.39	1.20	1.19	2.06	1.38
fft64-v2	2.05	1.33	2.10	2.12	2.56	2.44
fft128	3.28	4.98	4.54	2.93	5.46	4.56
cfo-32	0.41	0.29	0.30	0.81	0.68	0.30
cfo-64	1.30	0.57	0.70	1.53	1.48	0.96
cfo-128	1.96	0.88	1.68	2.67	2.57	1.84
cordic-64	48.64	11.70	31.76	44.28	44.17	44.76
fir-32x64	15.74	8.52	12.86	15.66	20.97	14.62
fir-32x128	32.32	16.18	25.08	25.63	35.40	31.02
fir-32x256	59.88	28.01	52.42	49.34	70.25	64.72
mean-32	0.62	1.28	0.30	0.67	1.02	0.30
mean-64	0.86	2.26	0.50	0.86	1.49	0.50
mean-128	2.13	3.86	1.20	2.20	3.65	1.10
mean-1024	12.15	19.95	9.96	10.95	20.11	9.32
qr-decomp-32	38.14	36.76	42.04	35.65	40.74	37.96
qr-decomp-64	61.10	62.09	84.26	59.82	71.36	62.10
qr-decomp-128	112.70	117.99	168.40	115.95	134.57	123.56

Table 25: Reference values (exec. time in µs) used for normalization of vectorized results on i7-3820

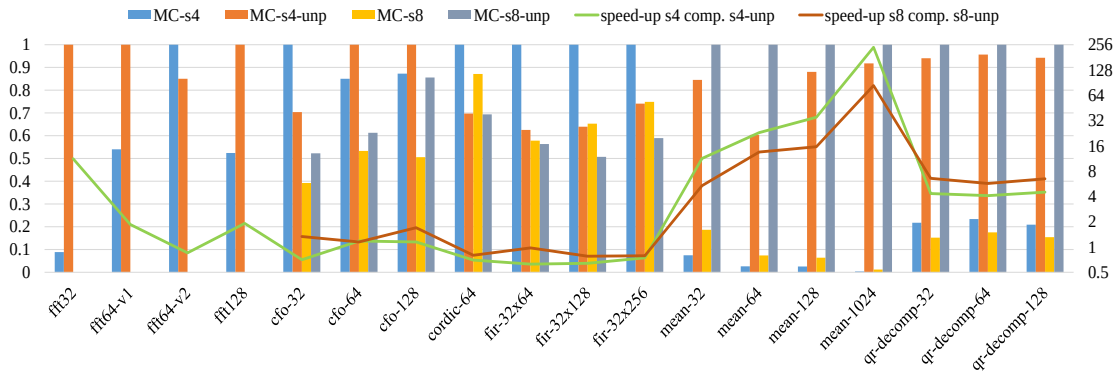

Figure 68: Performance of vectorized code with unpacked fixed point data types versus packed fixed point data types using GCC on desktop with i7-3820

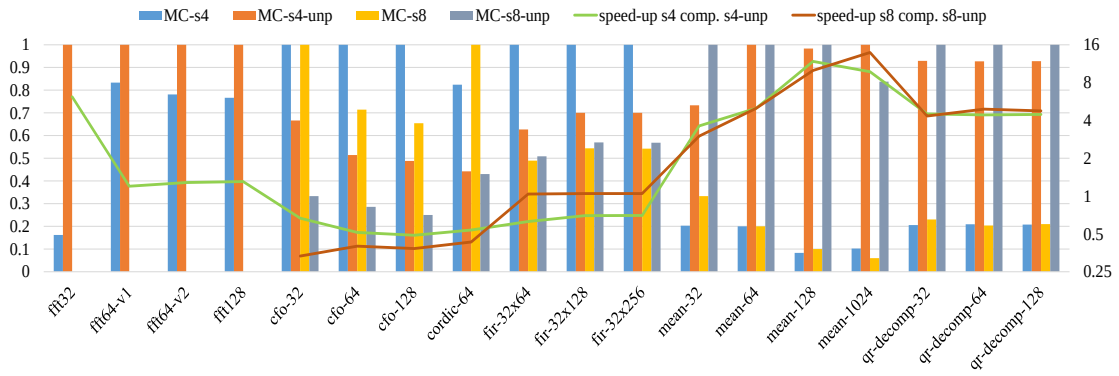

Figure 69: Performance of vectorized code with unpacked fixed point data types versus packed fixed point data types using MSVC on desktop with i7-3820

Performance among SIMD configurations using floating point types Figures 70, 71 and 72 show the normalized execution times (Table 25) of the vectorized generated code with different SIMD configurations of the floating point benchmark. The performance of vectorized code with packed data types against that of unpacked data types achieves maximum speed-up of 13x with 3.4x on average using SIMD width 4 and maximum speed-up of 24.6x with 4.3x on average using SIMD width 8 by compiling the vectorized code with Clang and executing the benchmark on Linux Ubuntu OS (Fig. 70). Compiling with GCC, the performance of vectorized code with packed data types compared to that with unpacked data types (Fig. 71) achieves a speed-up up to 34.3x with 6.1x on average using SIMD width 4 and up to 50.3x with 8.5x on average using SIMD width 8. The performance of the vectorized code using MSVC and executing the benchmark on Windows 10 OS (Fig. 72) with packed data types compared to that with unpacked data types achieves a speed-up up to 9.2x with 3.5x on average using SIMD width 4 and up to 46.6x with 6.7x on average using SIMD width 8.

Discussion of performance among SIMD configurations using floating point types The vectorized code using unpacked data types of CFO and CORDIC applications achieve faster execution times than the vectorized code using packed data types with any configuration, except when using the MSVC compiler with SIMD width 4 where CORDIC application obtains better performance with packed data types. Furthermore, FFT-64-v2 compiled with GCC and FFT-64-v1/FIR applications compiled with Clang achieve better execution times with unpacked data types compared to packed data types for SIMD width of 4.

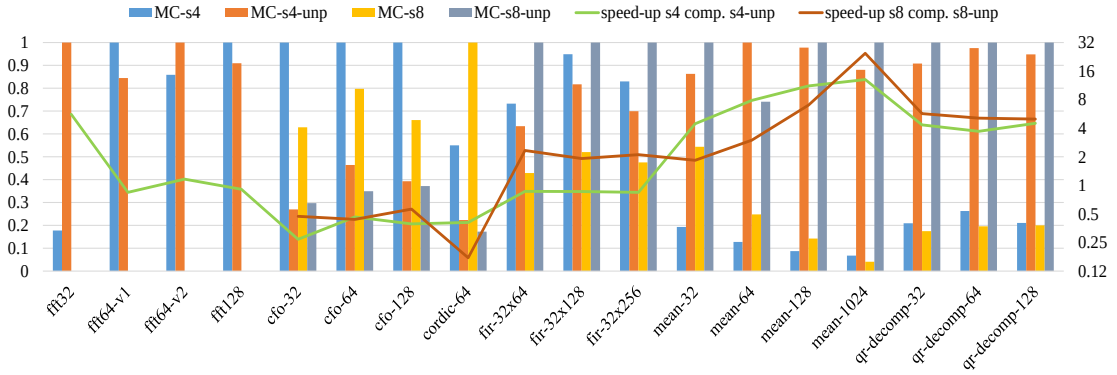

Figure 70: Performance of vectorized code with unpacked floating point data types versus packed floating point data types using Clang/LLVM on desktop with i7-3820

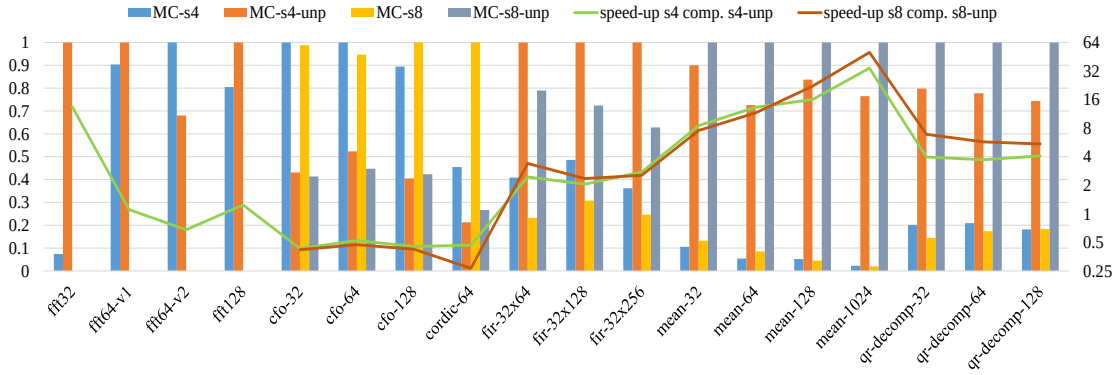

Figure 71: Performance of vectorized code with unpacked floating point data types versus packed floating point data types using GCC on desktop with i7-3820

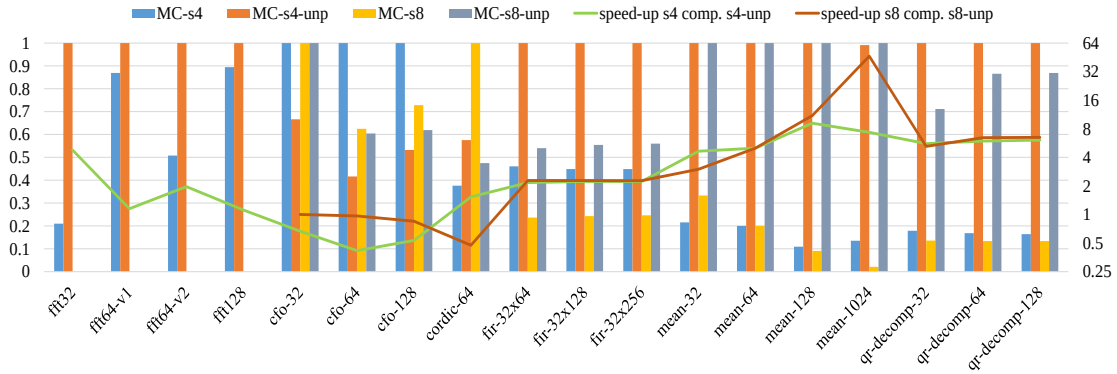

Figure 72: Performance of vectorized code with unpacked floating point data types versus packed floating point data types using MSVC on desktop with i7-3820

6.4.3 Performance of Generated Code on Intel Ivy Bridge (i7-3770)

Discussion of performance using packed fixed point types This section presents the performance of the compiler's generated vectorized code compared to the one generated code by MathWorks Coder on a desktop with Ivy Bridge i7-3820 processor. Figures 73, 74 and 75 show the normalized execution times (Table 26) of vectorized generated code with packed fixed point data types against the MathWorks generated code compiling with Clang, GCC and MSVC respectively. The performance of the vectorized code using the Clang compiler for the compilation of the generated code and executing the benchmark on Linux Ubuntu OS obtains a speed-up up to 270.7x with 19.3x on average using SIMD width 4 and a speed-up up to 7.5x with 2.9x on average using SIMD width 8. When compiling with GCC the generated code by two MATLAB compilers, the vectorized code performance achieves a speed-up between 2.5x-272.8x with 26x on average, using SIMD width 4 and up to 49.4x with 14.3x on average using SIMD width 8. Finally, the performance of vectorized code against MathWorks generated code using MSVC and Windows 10 OS is from 1.9x up to 65.6x with 10.9x on average, using SIMD width 4 and between 1x-41.4x with 9.1x on average using SIMD width 8.

Discussion of performance using packed fixed point types The vectorized code of FFT-32 application obtains the highest speed-up for SIMD width 4 while QR-decomposition and CFO applications achieve the highest speed-up for SIMD 8 by using GCC and Clang/MSVC correspondingly. Compiling with Clang, the FIR application with SIMD width of 4 and MEAN application for small input streams with SIMD width of 8 achieve better performance using MathWorks Coder than generating vectorized code. Moreover, MEAN application using small input streams results to better performance with SIMD width of 4 than 8. Similar results have been achieved for CORDIC and QR-decomposition applications compiling the generated code with Clang.

	Fig. 73, Fig. 76	Fig. 74, Fig. 77	Fig. 75, Fig. 78	Fig. 79, Fig. 82	Fig. 80, Fig. 83	Fig. 81, Fig. 84
fft32	7.86	8.68	3.50	0.15	0.76	0.30
fft64-v1	4.51	3.89	3.10	1.67	2.08	1.10
fft64-v2	13.94	12.69	4.80	1.84	1.61	1.30
fft128	26.25	27.91	13.70	3.22	3.94	2.90
cfo-32	1.07	1.21	4.14	1.37	4.10	1.46
cfo-64	6.70	8.68	4.00	2.80	8.23	1.96
cfo-128	13.44	15.36	8.00	6.31	15.32	2.44
cordic-64	69.40	158.71	326.98	40.91	39.75	37.72
fir-32x64	15.74	20.90	84.54	14.76	14.91	12.30
fir-32x128	29.64	39.10	189.76	26.43	35.18	26.90
fir-32x256	52.55	75.66	395.06	47.27	61.01	56.10
mean-32	0.42	0.90	0.20	0.45	0.77	0.20
mean-64	0.60	1.52	0.40	0.59	1.27	0.40
mean-128	1.62	3.12	0.90	1.73	2.99	0.90
mean-1024	9.40	18.19	7.74	9.61	15.54	7.72
qr-decomp-32	179.83	189.53	126.30	30.86	35.69	26.00
qr-decomp-64	53.96	54.81	113.46	51.87	60.07	52.50
qr-decomp-128	98.85	102.15	140.04	102.03	116.66	105.04

Table 26: Reference values (exec. time in µs) used for normalization of results on i7-3770

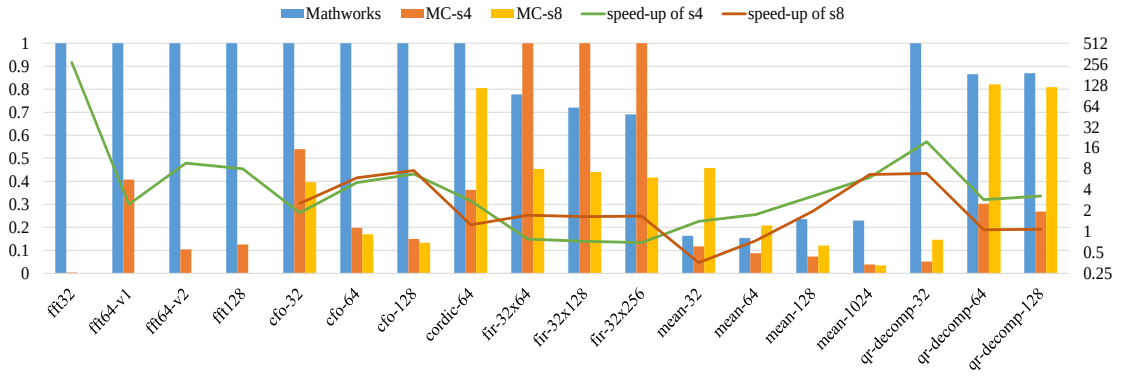

Figure 73: Speed-up comparing with MathWorks compiler on i7-3770 using Clang/LLVM with packed fixed point data

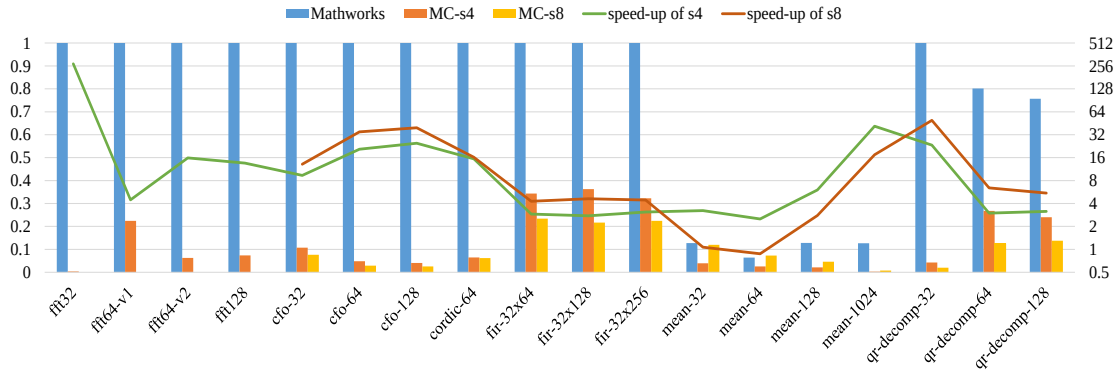

Figure 74: Speed-up comparing with MathWorks compiler on i7-3770 using GCC with packed fixed point data

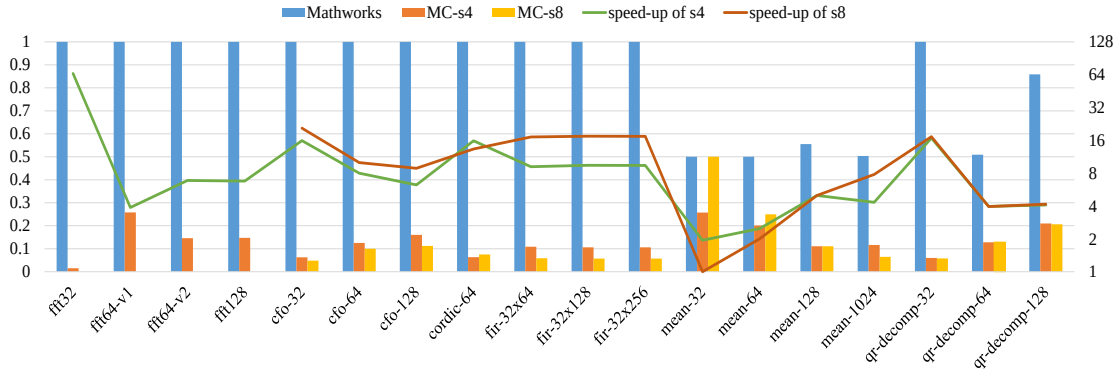

Figure 75: Speed-up comparing with MathWorks compiler on i7-3770 using MSVC with packed fixed point data

Discussion of performance using unpacked fixed point types Figures 76, 77 and 78 present the normalized execution times (Table 26) of the vectorized generated code with unpacked fixed point types compared to the generated code by MathWorks Coder. The speed-up obtained by compiling the generated code of two MATLAB compilers with Clang and executing the benchmark on Linux Ubuntu OS (Fig. 76) is up to 66.1x with average speed-up of 8.4x for SIMD width 4 and up to 43.2x with average speed-up of 8.3x for SIMD width 8. Using GCC for compilation (Fig. 77) the performance of vectorized code compared to that of MathWorks generated code attains a speed-up up to 17.4x with 6.3x on average for SIMD width 4 and up to 38.2x with 9.3x on average for SIMD width 8. The Windows 10 OS and MSVC configuration (Fig. 78) achieves a speed-up of vectorized code up to 41.4x and 9.1x on average for SIMD width 4 and an average speed-up of 11.7x up to 41.4x for SIMD width of 8.

Discussion of performance using unpacked fixed point types The vectorized code of CFO application achieves the highest speed-up except the one with Clang and SIMD width 4 configuration where FFT-32 application obtains the maximum speed-up. MathWorks generated code of MEAN application and QR-decomposition with input streams of 2x2x64 and 2x2x128 achieves faster execution times compared to that of the vectorized generated code by using any of C compilers. The specific benchmarks achieve a reduced performance with the vectorized code of SIMD width 8 compared to the performance of vectorized code of SIMD width 4 as well.

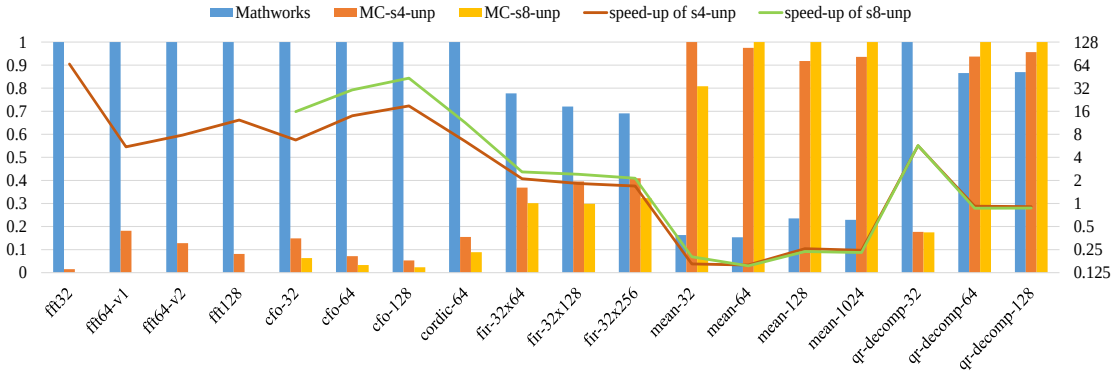

Figure 76: Speed-up comparing with MathWorks compiler on i7-3770 using Clang/LLVM with unpacked fixed point data

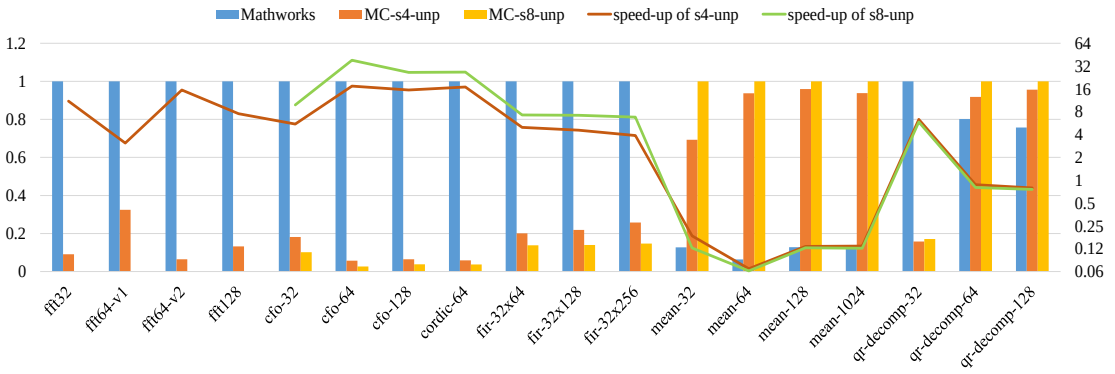

Figure 77: Speed-up comparing with MathWorks compiler on i7-3770 using GCC with unpacked fixed point data

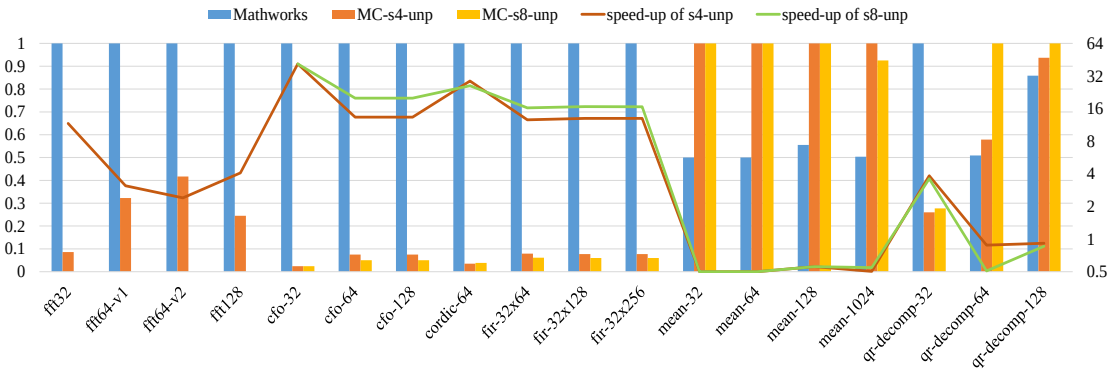

Figure 78: Speed-up comparing with MathWorks compiler on i7-3770 using MSVC with unpacked fixed point data

Discussion of performance using packed floating point types Figures 79, 80 and 81 show the normalized execution times (Table 26) of the vectorized generated code with packed floating point data types compared to generated code by MathWorks Coder using Clang, GCC and MSVC respectively. The performance of the vectorized generated code achieves maximum speed-up of 4.7x with 1.7x on average for SIMD width 4 and maximum speed-up of 3.9x with average 1.7x for SIMD width 8 compiling the generated code by two MATLAB compilers with Clang and executing the benchmark on Linux Ubuntu OS. By compiling with GCC, the performance of vectorized code compared to that of MathWorks generated code attains speed-up up to 9.2x with 2.9x on average for SIMD width 4 and up to 15.5x with average speed-up of 3.7x for SIMD width 8. The speed-up of the vectorized code performance against that of MathWorks generated code compiling with MSVC and executing the benchmark on Windows 10 OS is up to 4.9x with 1.8x on average using SIMD width 4 and up to 7.3x with 2.9x on average using SIMD width 8.

Discussion of performance using packed floating point types The maximum speed-up of vectorized code against MathWorks generated code achieved for FFT-32 application using Clang and GCC while compiling the generated code by both MATLAB compilers with MSVC, the maximum speed-up of vectorized code obtained for CFO application using SIMD width 4. Furthermore, performance of CFO application vectorized code also achieves the highest speed-up using any of the C compilers for SIMD width 8. FFT-64, FFT-128, CORDIC applications compiled with Clang and QR-decomposition compiled with MSVC obtain better performance with MathWorks generated code than executing vectorized code. Furthermore, CORDIC and MEAN applications as well as QR-decompotision compiled with Clang and GCC results to reduced performance with the vectorized code of SIMD width 8 compared to the performance of vectorized code of SIMD width 4.

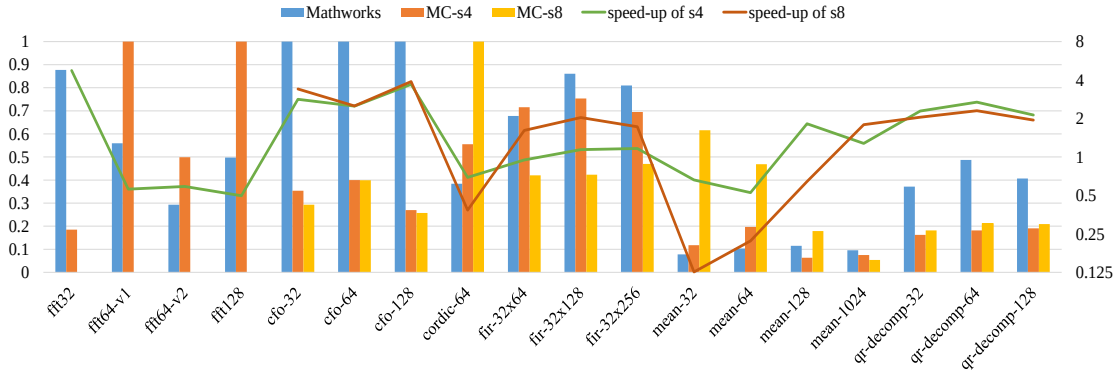

Figure 79: Speed-up comparing with MathWorks compiler on i7-3770 using Clang/LLVM with packed floating point data

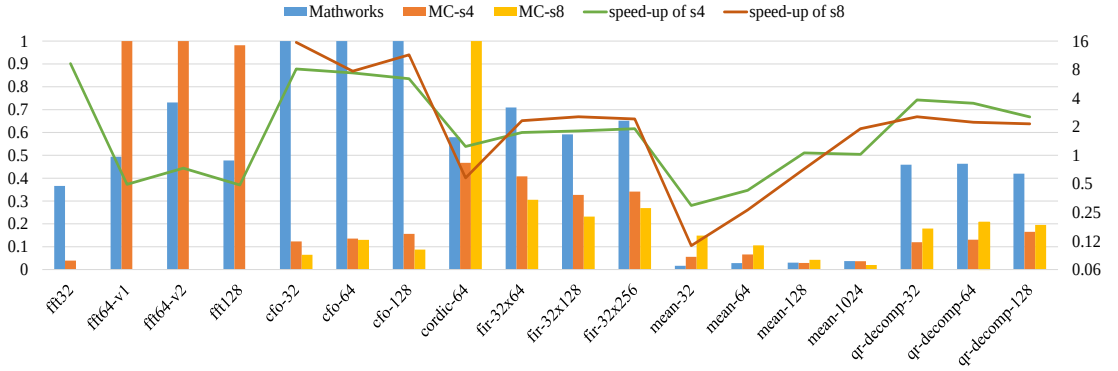

Figure 80: Speed-up comparing with MathWorks compiler on i7-3770 using GCC with packed floating point data

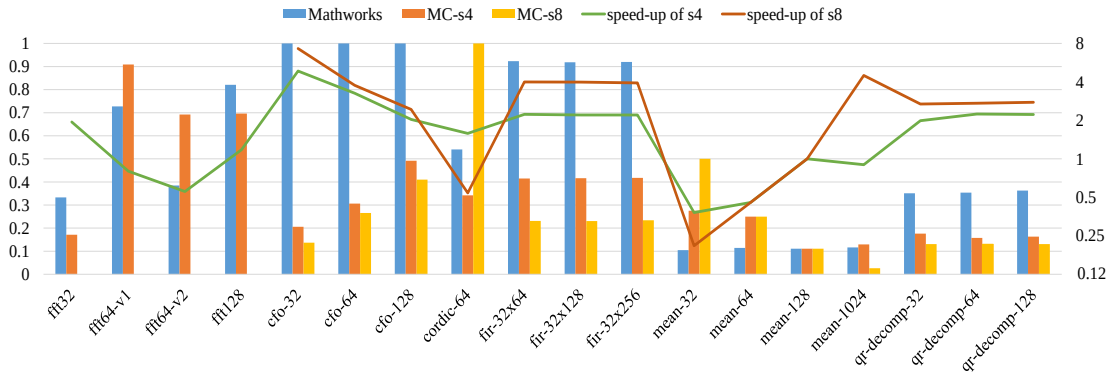

Figure 81: Speed-up comparing with MathWorks compiler on i7-3770 using MSVC with packed floating point data

Discussion of performance using unpacked floating point types Figures 82, 83 and 84 show the normalized execution times (Table 26) of the vectorized generated code with unpacked floating point data types compared to the MathWorks generated code using Clang, GCC and MSVC respectively. The performance of the vectorized code compared to that of MathWorks generated code compiling with Clang and executing the benchmark on Linux Ubuntu OS is up to 8.6x with 1.8x on average using SIMD width 4 and up to 8.2x with 1.9x on average using SIMD width 8. Compiling with GCC the speed-up achieved by the vectorized code performance compared to that of MathWorks generated code is up to 21.2x with 3.4x on average speed-up using SIMD width 4 and up to 24x with 5.2x on average using SIMD width 8. The performance of the vectorized code compiling with MSVC and executing the benchmark on Windows 10 OS obtains a maximum speed-up of 12.1x with 1.6x on average for SIMD width 4 and maximum speed-up of 14.6x with 2.3x on average for SIMD width 8.

Discussion of performance using unpacked floating point types The maximum speed-up of vectorized code compared to MathWorks generated code achieved for the CFO application. The vectorized code of the unpacked floating point data types achieves significant speed-up against the MathWorks generated code only for CFO and CORDIC applications. Moreover, most of the benchmarks using SIMD width 8 achieve incidental or reduced performance compared to that of vectorized code with SIMD width 4.

Explanation of performance of the i7-38770 processor The results of the targeted processor reveal similar performance as that of i7-3820 processor, proving that packing/unpacking operations insert significant overhead and reduce the overall performance of vectorized code. Concerning the floating point types, the MathWorks scalarized code of several benchmarks executed on x86 processors is more efficient than the corresponding vectorized code generated by the compiler. Thus, SIMD processing with floating point data types isn't always as efficient on x86 architecture as executing scalarized code.

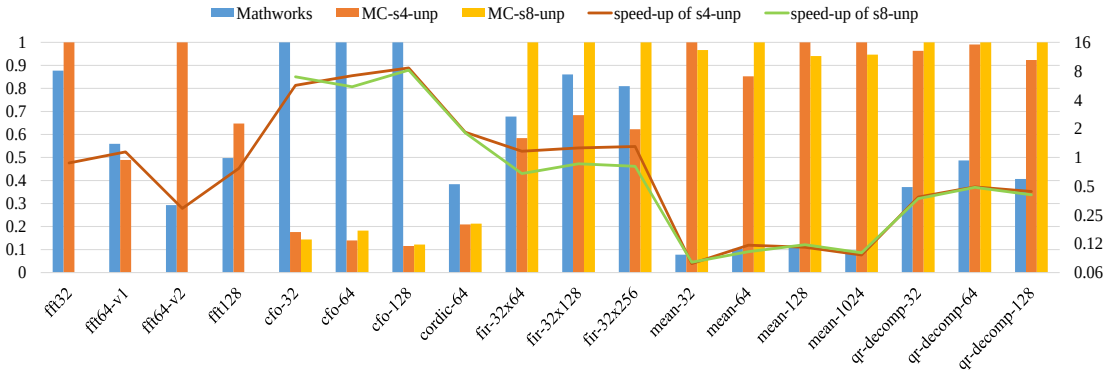

Figure 82: Speed-up comparing with MathWorks compiler on i7-3770 using Clang/LLVM with unpacked floating point data

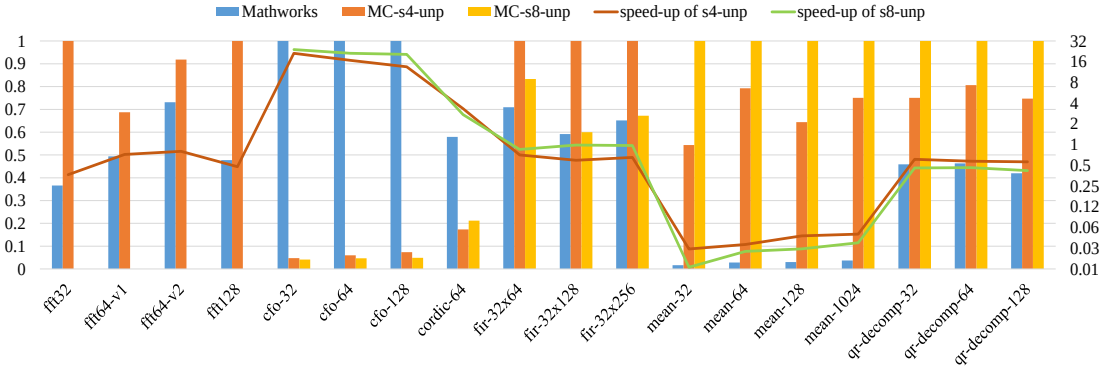

Figure 83: Speed-up comparing with MathWorks compiler on i7-3770 using GCC with unpacked floating point data

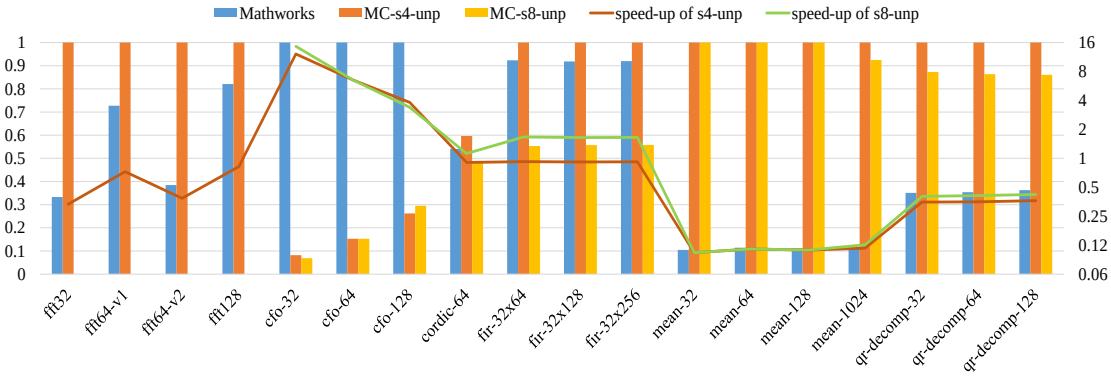

Figure 84: Speed-up comparing with MathWorks compiler on i7-3770 using MSVC with unpacked floating point data

Performance among SIMD configurations using fixed point types Figures 85, 86 and 87 show the normalized execution times (Table 27) of the vectorized generated code using different SIMD configurations for the benchmark with fixed point data types. The performance of vectorized code with packed data types compared to that of vectorized code with unpacked data types achieves speed-up up to 24.4x with 4.2x on average for SIMD width 4 and maximum speed-up of 28.8x with 3.6x on average for SIMD width 8 using Clang and Linux Ubuntu OS (Fig. 85). The speed-up of the vectorized code performance with packed data types compiling the generated code by MATLAB compilers with GCC (Fig. 86) is up to 306.2x with average speed-up of 25.2x for SIMD width 4 and up to 137.9x with average speed-up of 15.1x for SIMD width 8. Finally, the speed-up of vectorized code performance with packed data types compared to that of unpacked data types compiling with MSVC and executing the benchmark on Windows 10 OS (Fig. 87) is up to 9x with average speed-up of 3.1x for SIMD width 4 and up to 14.3x with 3.7x on average for SIMD width 8.

Discussion of performance using packed fixed point types The performance of the vectorized code with packed data types compared to that with unpacked data types achieves the highest speed-up for MEAN-1024 benchmark using any of the C compilers for the compilation of the benchmark. Furthermore, the performance for the most of the benchmarks using packed data types leads to significant speed-ups against the corresponding version with unpacked data types. However, CORDIC and FIR applications performs reduced performance when using packed data types compared to the performance using unpacked data types. Similar results achieved for CFO application compiling the generated code with Clang and MSVC C compilers.

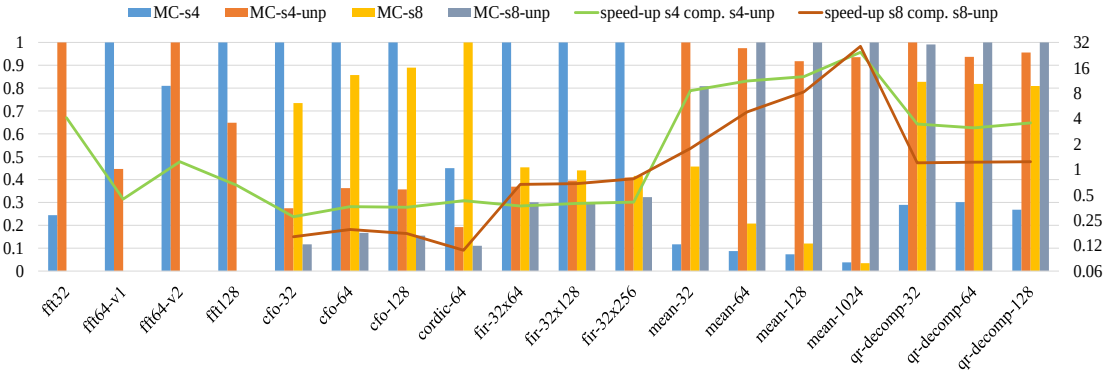

Figure 85: Performance of vectorized code with unpacked fixed point data types versus packed fixed point data types using Clang/LLVM on desktop with i7-3770

	Fig. 85	Fig. 86	Fig. 87	Fig. 88	Fig. 89	Fig. 90
fft32	0.12	0.79	0.30	0.15	0.76	0.30
fft64-v1	1.84	1.26	1.00	1.67	2.08	1.10
fft64-v2	1.79	0.82	2.00	1.84	1.61	1.30
fft128	3.29	3.70	3.36	3.22	3.94	2.90
cfo-32	0.58	0.22	0.26	0.49	0.50	1.46
cfo-64	1.32	0.50	0.50	1.12	1.12	1.96
cfo-128	2.00	0.99	1.28	1.70	2.39	2.44
cordic-64	55.86	10.31	24.52	40.91	39.75	37.72
fir-32x64	15.74	7.18	9.20	14.76	14.91	12.30
fir-32x128	29.64	14.17	20.10	26.43	35.18	26.90
fir-32x256	52.55	24.44	41.88	47.27	61.01	56.10
mean-32	0.42	0.90	0.20	0.45	0.77	0.20
mean-64	0.60	1.52	0.40	0.59	1.27	0.40
mean-128	1.62	3.12	0.90	1.73	2.99	0.90
mean-1024	9.40	18.19	7.74	9.61	15.54	7.72
qr-decomp-32	31.75	32.40	35.00	30.86	35.69	26.00
qr-decomp-64	53.96	54.81	113.46	51.87	60.07	52.50
qr-decomp-128	98.85	102.15	140.04	102.03	116.66	105.04

Table 27: Reference values (exec. time in µs) used for normalization of vectorized results on i7-3770

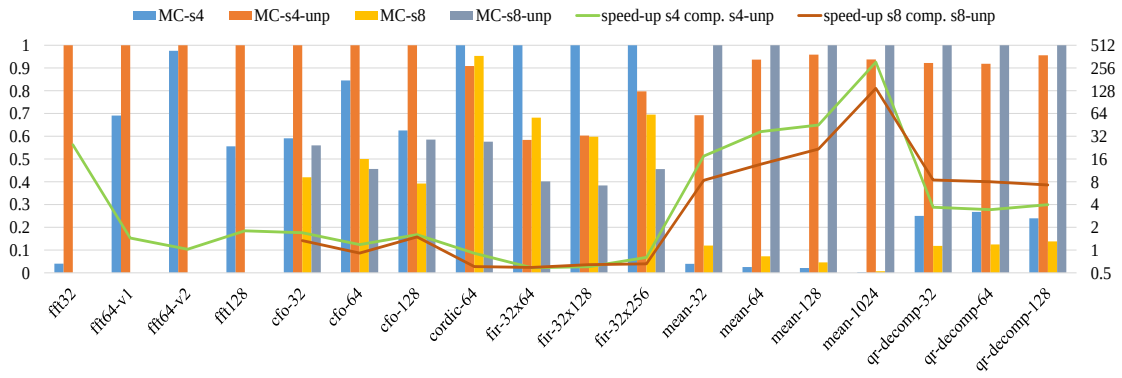

Figure 86: Performance of vectorized code with unpacked fixed point data types versus packed fixed point data types using GCC on desktop with i7-3770

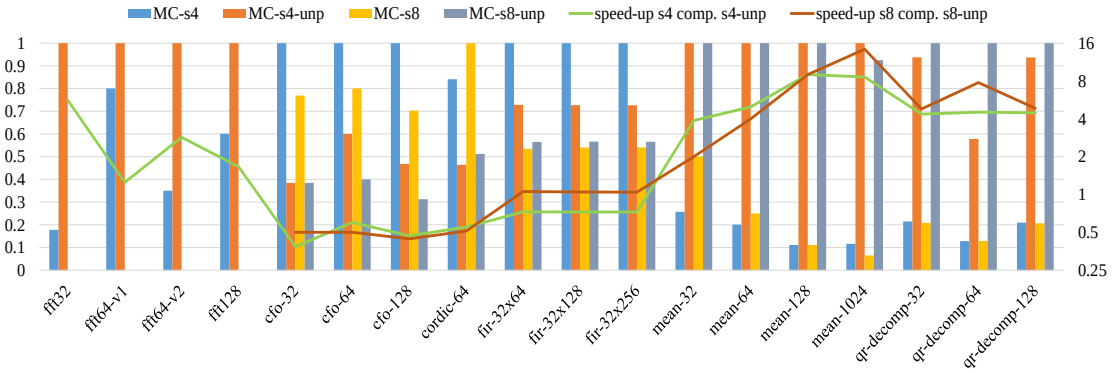

Figure 87: Performance of vectorized code with unpacked fixed point data types versus packed fixed point data types using MSVC on desktop with i7-3770

Performance among SIMD configurations using floating point types Figures 88, 89 and 90 show the normalized execution times (Table 27) of the vectorized generated code for different SIMD configurations for the floating point benchmark and compiling the generated code by MATLAB compilers with Clang, GCC and MSVC respectively. The performance of the vectorized code with packed data types compared to that of unpacked data types achieves speed-up up to 15.8x and 4x on average for SIMD width 4, and speed-up up to 17.7x and 3.6x on average for SIMD width 8, using clang and Linux Ubuntu OS. The speed-up of the vectorized code with packed data types using GCC on Linux Ubuntu OS is up to 25.3x and 6.6x on average for SIMD width 4 and up to 50.8x with 8.2x on average for SIMD width 8. Finally, the speed-up of vectorized code with packed data types compared to that of unpacked data types compiling with MSVC and executing the benchmark on Windows 10 OS is up to 9x and 3.4x on average for SIMD width 4 and up to 35.7x and 5.7x on average for SIMD width 8.

Discussion of performance using packed floating point types The performance of vectorized code with packed data types compared to that with unpacked data types achieves the highest speed-up for MEAN application besides GCC with SIMD width 4 configuration where FFT-32 application obtains the maximum speed-up. The majority of benchmarks achieve better performance using packed data types. However some applications such as CFO and CORDIC obtain reduced performance with packed data types.

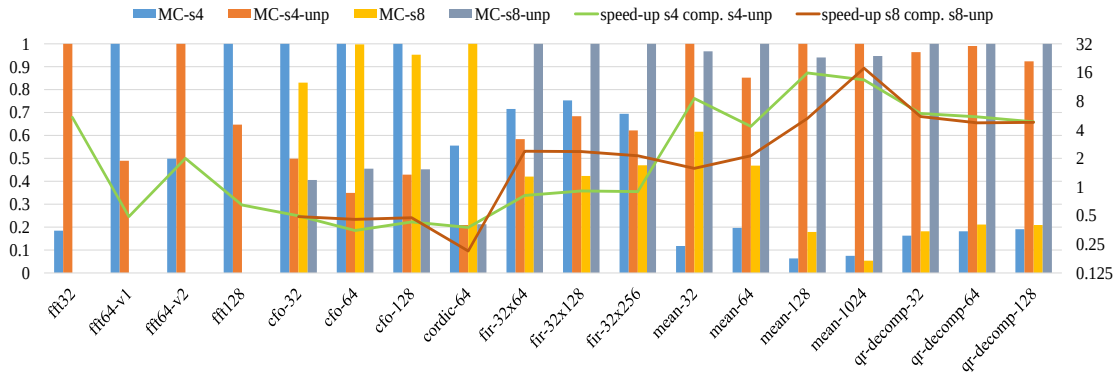

Figure 88: Performance of vectorized code with unpacked floating point data types versus packed floating point data types using Clang/LLVM on desktop with i7-3770

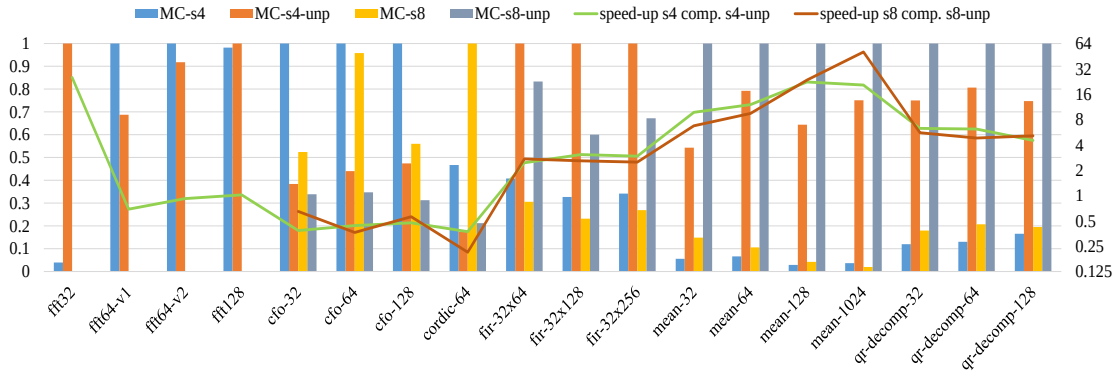

Figure 89: Performance of vectorized code with unpacked floating point data types versus packed floating point data types using GCC on desktop with i7-3770

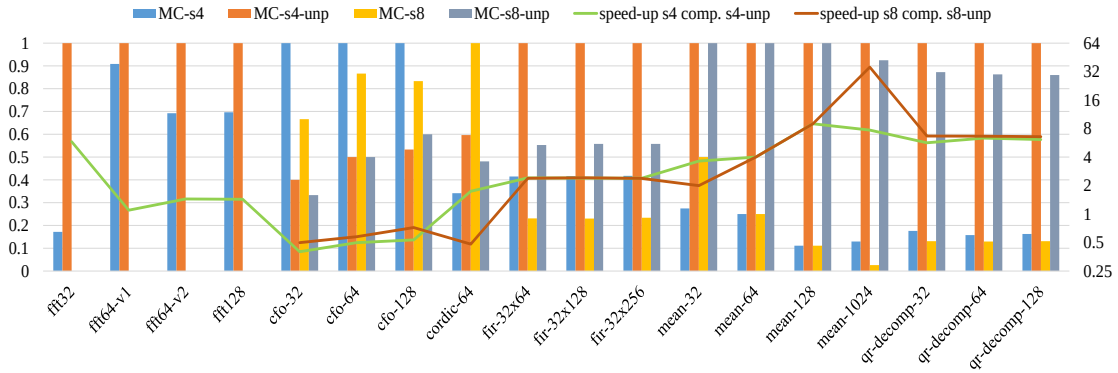

Figure 90: Performance of vectorized code with unpacked floating point data types versus packed floating point data types using MSVC on desktop with i7-3770

6.5 Results from BoT ASIP

This section describes the BoT architecture used in the experiments and discusses the performance (speed) of the generated C code by the compiler compared to that of the code generated by the MathWorks Coder. Further experiments present the execution times of the generated code by the compiler using different settings concerning the SIMD width of vectors and the form of data types (packed versus unpacked).

6.5.1 Presentation of BoT Architecture

The ASIP (application specific instruction processor) approach provides a trade-off between the less specialized general purpose processors on one hand and the rigidly defined and highly optimized ASIC platforms. ASIPs target application specific domains to sharpen the efficiency of implementation while keeping them programmable across various applications in the chosen domain. ADRES [Mei et al., 2003] architecture template which developed at research center IMEC targeting applications in the streaming domain suitable for many signal processing applications. This template has been used in the past for multi-media and wireless signal processing applications and very high performances in terms of power and throughput efficiencies have been reported. It has also been able to prove that in some cases [Fasthuber et al., 2013], that careful design choices while deriving ADRES can surpass the efficiency of ASIC and FPGA based designs in terms of performance achieved and more importantly, the development time. In this thesis, two instances of the ADRES template: BoT and TinyBoT (section 6.6) have been chosen to experiment with.

BoT Micro-architecture The BoT is a 10 way VLIW instruction set processor. It has been envisioned specially for implementing wireless PHY signal processing [BoT, 2014]. The BoT architecture consists of 4 essential parts as shown in Fig. 91. This architecture enables massive parallel computation by making use of the SIMD datapath in the vector units, while the scalar data path is used for managing the control flow. The shuffle unit consists of data shufflers that operate on vector data-types for ordering/ re-ordering of data in a vector register file. The scalar units are three in number and can uniformly accesses a 32 deep scalar register file each containing 32 bit words. The vector paths are also 3 in number with each of them capable of supporting 4 or 8 way SIMD complex data type with an aggregate width of 32 bits allocated as 16 bits each for imaginary and real parts. The choice of SIMD width between 4 and 8 is a design time choice that the template provides. The vector register file is also 32 deep and 256 bits (or 128 bits for 4 way SIMD) wide to hold vectored data serving any of the three vector units. The scalar unit supports the traditional Load/Store memory interface while the vector memory interaction is handled explicitly by the load-store

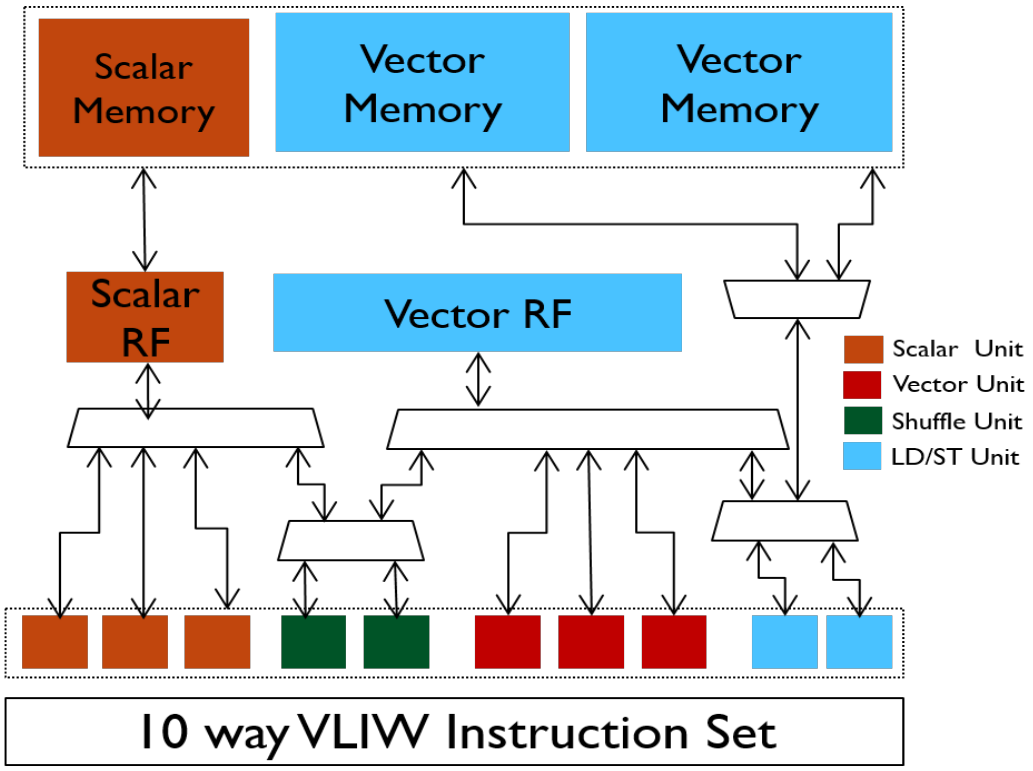

Figure 91: BoT architecture.

(LD/ST) unit. The Load and Store operations can happen simultaneously (on different locations of memory) and can thus improve throughput by benefiting from using multi-port memory. The scalar and vector data buses are 32 bits and 256 (or 128 bits for 4 way SIMD) bits respectively. The shuffle unit operates on the scalar and vector register files to modify vector data layouts and there are two of them and they can operate on data independently.

Domain Specific Instructions in BoT The BoT processor supports up to 10 levels of pipelining of vector units. These vector instructions include Trigonometric functions like Sine, Cosine and Inverse-Tangent functions, complex number operations such as absolute, multiply accumulate (MAC), multiple shift, real operations such as inverse and inverse square on vector data types. The LD/ST unit can read/ write words of size 256 bits (or 128 bits for 4 way SIMD) from the vector Register file. The shuffle exposes intrinsics supporting packing and unpacking of scalar data into vectors, circular or linear shift of packed data between two vector data types etc. The Load-Store unit accesses vector data in packed format from one of the two vector memories. The choice of memories between the load and stores are independent of one another, may be operated simultaneously using a multi-port memory. In this way, it is possible to make a choice

in software to split the memory into two (say, for ping-pong data through iterations) or keep them as one as the specific application demands. The scalar unit performs classical control operations such as looping. Indeed, to alleviate the overhead of comparison in software, for loops have been accelerated with special intrinsic support for the control flow.

Programming BoT The BoT architecture is coded by using the Synopsys ASIP design tool. In this tool the processor is described using the nML language. The tool generates all required compilation, debug and simulation infrastructure for the defined architecture. Benefiting from these facilities, the BoT architecture exposes several domain specific functionalities as C language intrinsics. These intrinsics directly map to one or a more predefined group of instructions on the processor and can be called like a regular C-language function sub-routine.

6.5.2 Presentation and Discussion of Results on BoT Architecture

Figures 92 and 93 presents the normalized execution times per benchmark (Table 28) achieved by both MATLAB compilers on BoT using packed data types (fig 92) and unpacked data types (fig 93). The lines added to the figures depict the execution speed-up achieved with the current approach in respect to MathWorks MATLAB-to-C compiler. The comparison of the compiler to MathWorks Coder proves that the proposed approach achieves a substantial speed-up between 1.3x-41.3x on the with packed data of SIMD width 8 and a speed-up up to 97.1x the with unpacked data of SIMD width 8.

The code generated by the compiler includes SIMD instructions in contrast to the code generated by MathWorks compiler that does not exploit the target processor/ASIP. The largest performance difference between the compiler and MathWorks Coder is obtained for the CFO and QR-decomposition algorithms. In these benchmarks the compiler maps operations which are expensive for execution with software functions (like exponential, absolute value, Inverse-Tangent) to corresponding custom instructions of the processor while the MathWorks compiler generates C code that implements the calculation of these functions at fixed point.

The Mean application with unpacked data types attains worse performance than MathWorks generated code due to the overhead of packing and unpacking. The SIMD block of this example includes only one SIMD operation. The overhead caused by the packing and unpacking exceeds the acceleration is achieved by the SIMD processing leading to reduced performance.

Figure 94 presents the normalized execution times per benchmark (Table 28) on BoT with different configurations. Execution times achieved when packed data types are used are better (2.6x times faster for SIMD width 4 and 3.2x times faster for SIMD width

	Fig. 92 and Fig. 93	Fig. 94, Fig. 95 and Fig. 96
fft32	1443	612
fft64-v1	8032	2036
fft64-v2	4722	1334
fft128	17420	4846
cfo-32	11563	696
cfo-64	22483	1352
cfo-128	45668	2662
cordic-64	3975253	720155
fir-32x256	724074	719073
mean-32	288	288
mean-64	544	544
mean-128	1056	1056
mean-1024	8224	8224
qr-decomp-32	1726926	44222

Table 28: Reference values (instructions count) used for normalization of results at BoT

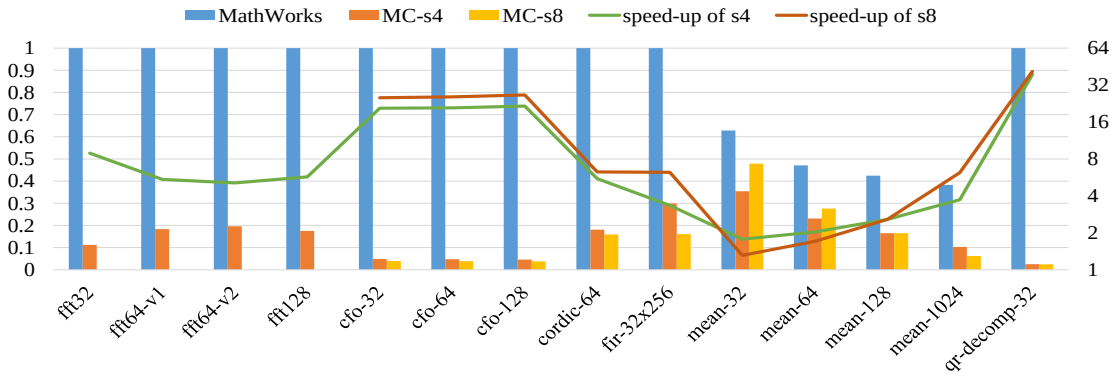

Figure 92: Speed up comparing with MathWorks compiler on BoT using packed data

8) than those achieved when unpacked data types are used. The generated code with packed data is suitable for execution of SIMD instructions in contrast to the generated code with unpacked data where packing and unpacking operations are required for SIMD processing. However, the CORDIC and QR-decomposition algorithms attain a better performance with unpacked data types. The CORDIC algorithm includes control-flow dependencies and the QR-decomposition algorithm contain code segments with scalar instructions. Thereby, the packed data in vector memory are unpacked and transfered to the scalar memory (and vice versa) for each scalar operation slowing up the execu-

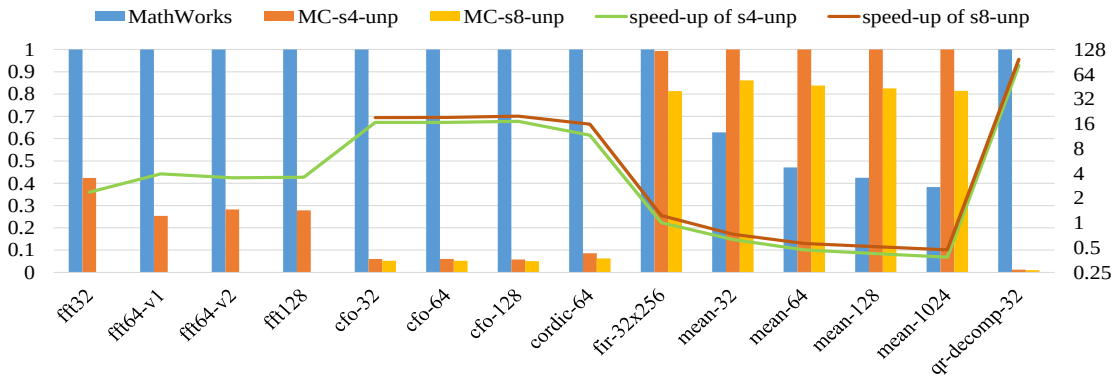

Figure 93: Speed up comparing with MathWorks compiler on BoT using unpacked data

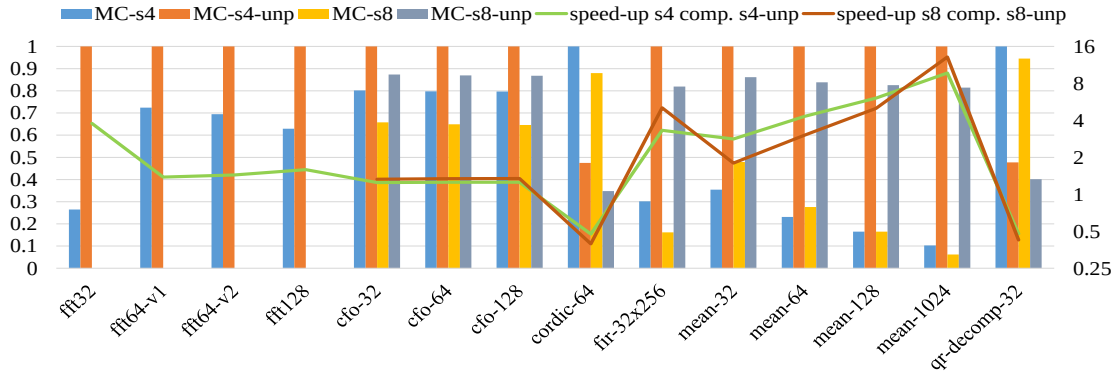

Figure 94: Performance of vectorized code with unpacked data types versus packed data types on BoT

tion performance of the generated code. On the other hand, the scalarized generated code with unpacked data types is executed without packing and unpacking operations. Additionally, the SIMD blocks of these algorithms include several SIMD operations eliminating the overhead is cause by packing and unpacking operations. Therefore, it is more beneficial to use unpacked data types for applications with scalar operations.

Figure 95 shows the speed-up achieved with SIMD width 8 comparing to SIMD width 4 using packed data types on BoT ASIP and figure 96 shows the same comparison using unpacked data types. The speed-up with SIMD width 8 is up to 1.8x and 1.2x times on average using packed data types and the speed-up using unpacked data types is up to 1.4x with 1.2x times on average.

For benchmarking FFT-64 and FFT-128 two scenarios have been used. The first concerns the typical execution of the third stage of the FFT using the MATLAB code of the corresponding user defined function. The second concerns the implementation of the third FFT stage using the custom instruction localR4 accelerating the data shuffling. Furthermore, two different implementations of FFT-64 have been evaluated - a standard one and an optimized which increasing the data locality. The optimized version includes

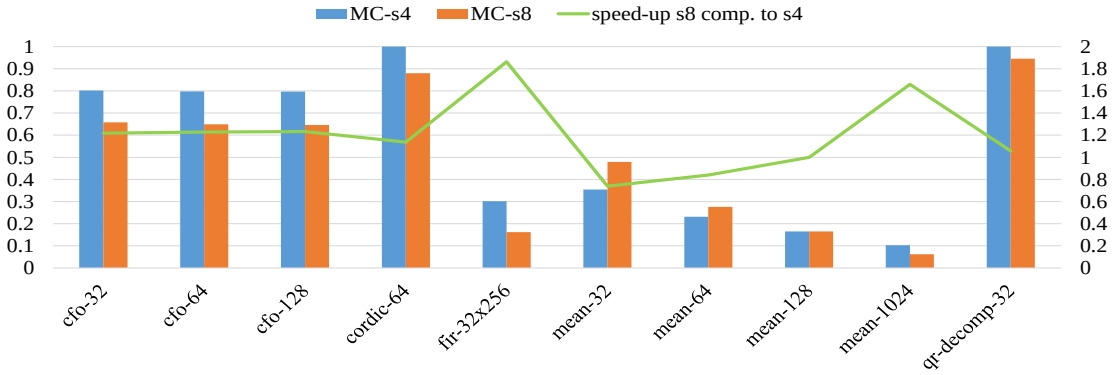

Figure 95: Executions times of vectorized code with SIMD width 8 comparing to SIMD width 4 on BoT using packed data

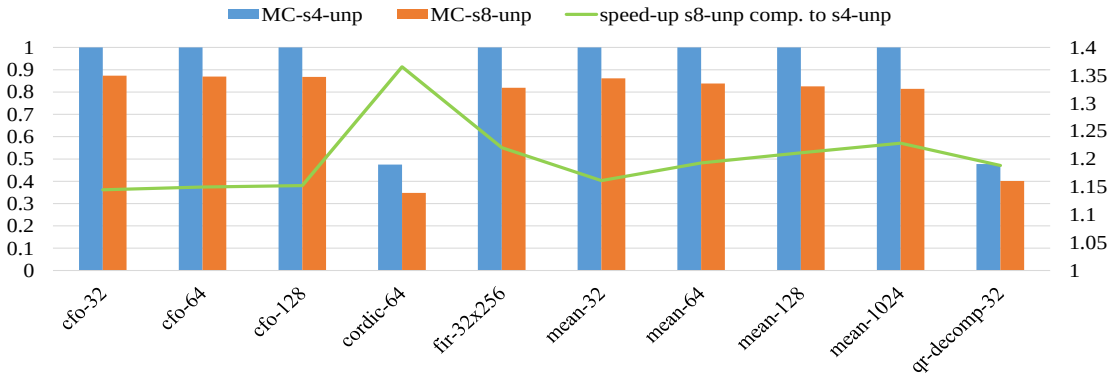

Figure 96: Executions times of vectorized code with SIMD width 8 comparing to SIMD width 4 on BoT using unpacked data

for-loops inside the function corresponding the FFT stages reusing the function's local variables.

Figure 97 presents the normalized execution times (fig. 29) of two optimized 64 point FFT versions and an optimized 128 point FFT version compared to their standard version on BoTs. The first optimized version (FFT64-v2) is obtained by performing loop transformations pushing loops across function boundaries and thus increasing the data locality. This approach is exploited by BoT architecture where vector register file is more than sufficient to store the data locally and therefore reduces the number of cycles.

The optimized versions FFT64-R4 and FFT128-R4 focus on the implementation of the third FFT's stage using custom instructions instead of software using the corresponding user defined function in MATLAB code that implements data shuffling. BoT processor provides the *localR4* instruction which implements data shuffling in hardware. This instruction can be exploited only by the proposed compiler, which maps the function call to the corresponding hardware intrinsic (*localR4* instruction). MathWorks Coder doesn't exploit custom instructions and generates generic software code for the given

tasks. The use of *localR4* instruction effects mostly the speed-up with packed data types where costly scalar operations are executed in hardware avoiding the execution of packing and unpacking operations. Finally, the optimized version (FFT64-v2-R4) combines optimizations of the two previous versions.

	Fig. 97
fft64-v1	8032
fft64-v1-R4	8032
fft64-v2	4722
fft64-v2-R4	4722
fft128	17420
fft128-R4	17420

Table 29: Reference values (instructions count) used for normalization at additional FFT results

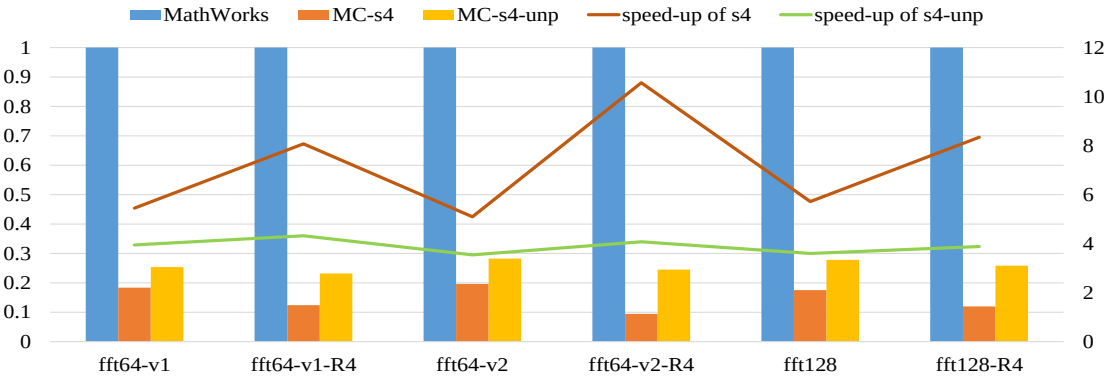

Figure 97: Execution times of FFT comparing with optimized versions on BoT

6.6 Results from tinyBoT ASIP

This section describes the tinyBoT architecture used in the experiments and presents the performance of the generated C code by the compiler compared to that of the code generated by the MathWorks Coder. Further experiments present the execution times of the generated code of an optimized version of FFT application compared to its basic version.

TinyBoT Architecture The tinyBoT ASIP is another instance of ADRES architecture template. More specifically, the TinyBoT architecture is derived from the BoT

architecture by removing everything but the Scalar Unit, Scalar RF and Scalar Memory. The TinyBoT architecture borrows essential complex arithmetic, trigonometric and inverse functionalities from the vector units of the BoT architecture and migrates it to the scalar units. The TinyBoT architecture is functionally compliant with the BoT architecture. That is, all special instructions are available on both architectures. The tinyBoT architecture is coded by using the Synopsys ASIP design tool [ASIP Designer, 2016].

6.6.1 Presentation and Discussion of Results on TinyBoT Architecture

Figure 98 presents the normalized execution times per benchmark (Table 30) achieved by both MATLAB compilers on tinyBoT ASIP. The line added to the figure shows the execution speed-up achieved with the current approach in respect to MathWorks Coder. The generated code by the compiler achieves a substantial speed-up between 1.3x-74x and 8.3x times on average. The code generated by the compiler includes scalarized custom instructions such as complex number operations and mathematical functions (mapped on specialized hardware) in contrast to the code generated by MathWorks compiler that does not include custom instructions. The highest speed-up of is obtained for the CFO, CORDIC and QR-decomposition algorithms where the compiler maps math tasks (like exponential, absolute value, Inverse-Tangent) to scalarized custom instructions instead of the MathWorks generated code which implement the calculation of these tasks without the use of hardware. The custom instructions are accelerated by hardware performing faster execution times than the corresponding software implementations generated by MathWorks coder.

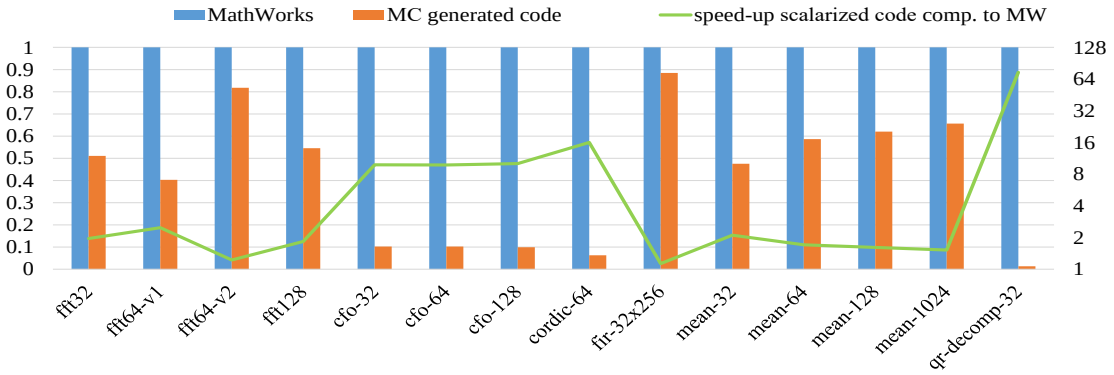

Figure 98: Speed up comparing with MathWorks compiler on TinyBoT.

Figure 99 shows the normalized execution times (fig. 31) of two optimized 64 point FFT versions and an optimized 128 point FFT version compared to their standard versions on tinyBoT. Although the optimized FFT-64 version (FFT64-v2) attain a better

	Fig. 98
fft32	1443
fft64-v1	8032
fft64-v2	4722
fft128	17420
cfo-32	11563
cfo-64	22483
cfo-128	45668
cordic-64	3975253
fir-32x256	724074
mean-32	181
mean-64	256
mean-128	448
mean-1024	3153
qr-decomp-32	1726926

Table 30: Reference values (instructions count) used for normalization of results at tinyBoT

performance comparing to the standard FFT-64 version (FFT64-v1) on BoT, the data locality doesn't benefit the generated code by the compiler on tinyBoT. This is due to the fact that some of scalar register files are used for other purposes (i.e. Stack Pointers). As a result, the local variables spill over into memory increasing load/stores at run time. In contrast, MathWorks compiler doesn't consider native complex data type of targeted architectures producing at least double the amount of C variables. This large number of variables causes register file spill overs on both FFT implementations. In the optimized version locality of variables is improved and this benefits MathWorks compiler.

Regarding the optimization of the data shuffling (third stage of FFT), the compiler maps the corresponding MATLAB function to the *localR4* custom instruction but only a small speed-up is achieved. The reason is that the biggest overhead of the scalarized generated code is included in the first and second FFT stages while the data shuffling of third stage doesn't impact (so much?, as much?) the total execution performance of the algorithm.

	Fig. 97
fft64-v1	8032
fft64-v1-R4	8032
fft64-v2	4722
fft64-v2-R4	4722
fft128	17420
fft128-R4	17420

Table 31: Reference values (instructions count) used for normalization at additional FFT results

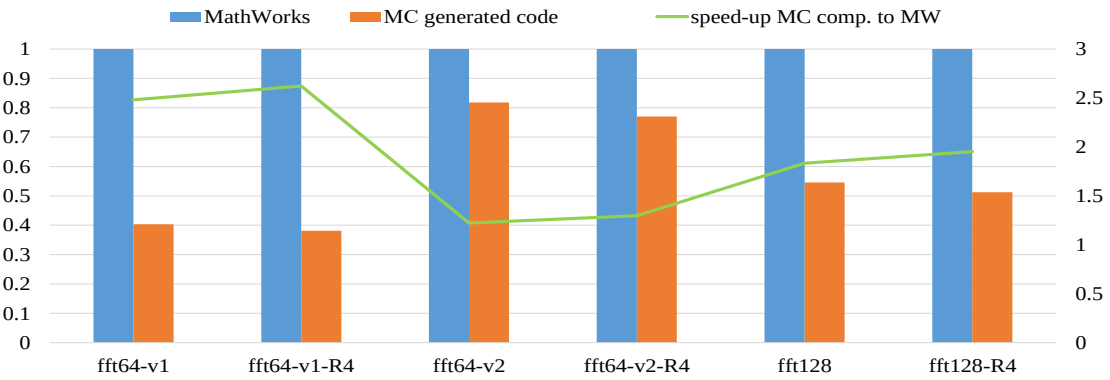

Figure 99: Execution times of FFT comparing with optimized versions on tinyBoT

6.7 Comparison of the Proposed Compiler at the Different Architectures

In this section the compiler's generated code is compared and discussed among the different targeted processors. Experiments include the comparison between the two ASIPs, as well as comparisons between the ARM and x86 targeted architectures using different configuration settings regarding the SIMD width of vectors and the form of the data types (packed versus unpacked).

6.7.1 Comparison of Generated Code by MathWorks and Proposed Compiler on ARM Processors

This section gives a summary regarding the speed-up of benchmark's execution times achieved on Raspberry PI 3 compared to that on Raspberry PI 2, for the generated code by both MATLAB compilers using various experimental settings. Figure 100 presents the minimum, maximum and average speed-up of MathWorks generated code and the scalarized (without use of intrinsics) generated code by compiler achieved on PI 3 compared to the corresponding generated code on PI 2. For the experiment, both the floating and fixed point benchmark versions have been evaluated and the generated code has been examined using Clang, GCC, and MSVC C compilers.

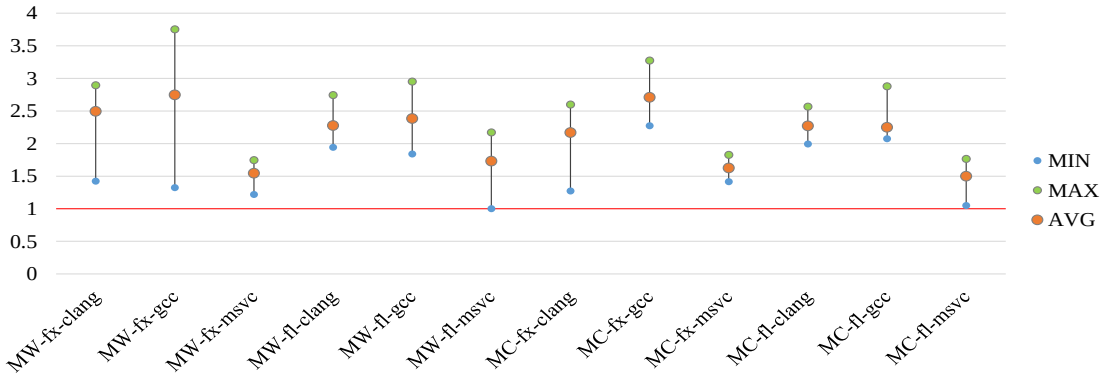

Figure 100: Speed-up of MathWorks and scalarized with no intrinsics generated code on PI 3 compared to PI 2

The execution times of the scalarized generated code by both MATLAB compilers on PI 3 are faster than the execution times on PI 2 using any of the aforementioned settings. The generated C code compiled with Clang or GCC executed on Rasbian OS obtains higher speed-up (comparing to the execution times on the two targeted processors) than the generated code compiled with MSVC and executed on Windows IoT OS. Furthermore, the benchmark with fixed point data types achieves higher speed-up compared to

floating point benchmark for both MathWorks generated code and compiler's scalarized generated code using any of the evaluated C compilers. Finally, the speed-up achieved for the performance of the MathWorks generated code among the various experimental options is quite similar to the speed-up achieved at the generated code by the compiler.

Figure 101 presents the speed-up of vectorized generated code achieved on PI 3 compared to that on PI 2 using various SIMD options with different data types and compiling the generated C code with Clang, GCC and MCVC C compilers. The execution times on PI 3 are faster than the execution times on PI 2 using any of the evaluated experimental options. The performance of the generated code compiled with MSVC and executed on Windows IoT OS achieves a lower speed-up (of the execution times between the two targeted processors) compared to that of generated code which have been compiled with Clang or GCC and have been executed on Rasbian OS. The performance of the generated code with fixed point data types achieves a higher speed-up compared to that of the generated code with floating point data types only by compiling with Clang while GCC and MSVC obtain a higher speed-up for the floating point data types benchmark. Finally, packed data types with SIMD width 4 configuration obtain highest speed-up using any of the three C compilers.

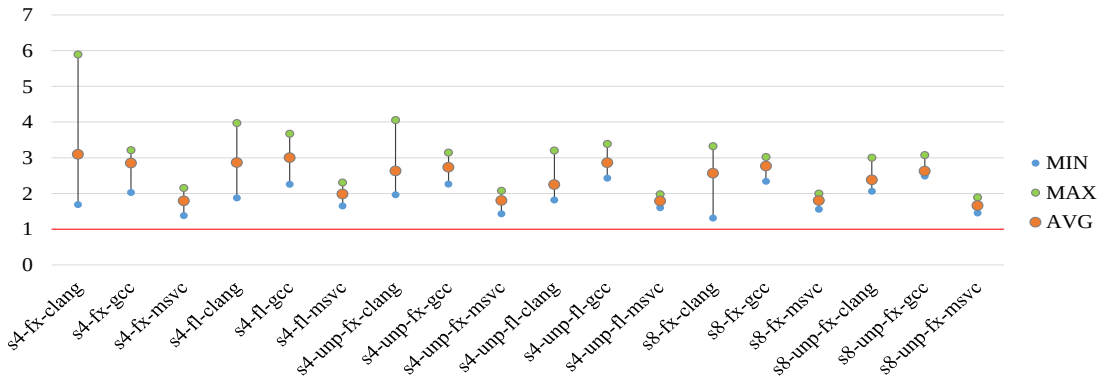

Figure 101: Speed-up of vectorized generated code on PI 3 compared to PI 2

Figure 102 shows the average speed-up achieved by the performance of the vectorized generated code compared to that of MathWorks generated code on PI 2 and PI 3. The vectorized code with fixed point data types (of any SIMD options) compiled with Clang and executed on Rasbian OS attain the biggest difference of speed-ups while for the floating point benchmark GCC obtains the biggest difference between the performance of the two targeted processors. Finally, The vectorized code compiled with MSVC and executed on Windows IoT OS achieves zero or marginal differentiation on the the speed-ups achieved on the two targeted ARM processors.

The execution times' performance of the generated C code depends on the processor/architecture capabilities and the efficiency of C compilers to produce high performance executable code for any target processor. According to the experiments, Clang

and GCC produce code (for Rasbian OS) that takes better advantage of the current processor efficiency and they obtain significant execution time speed-up in conjunction with the processor capabilities. However, the two compilers present a more unstable performance, due to big large span of the speed-up values, compared to the corresponding performance achieved by MSVC compiler where the range of minumum and maximum speed-up values is small, close to average value. Concerning the evaluation of SIMD code performance, the experiment shows that the vectorized generated code exploits efficiently the targeted processor resources and SIMD extension capabilities. The speed-ups obtained by the performance of the vectorized code comparing to the execution times on the two targeted ARM processors are similar or even better in several cases than the corresponding speed-ups of the performance of the scalarized code.

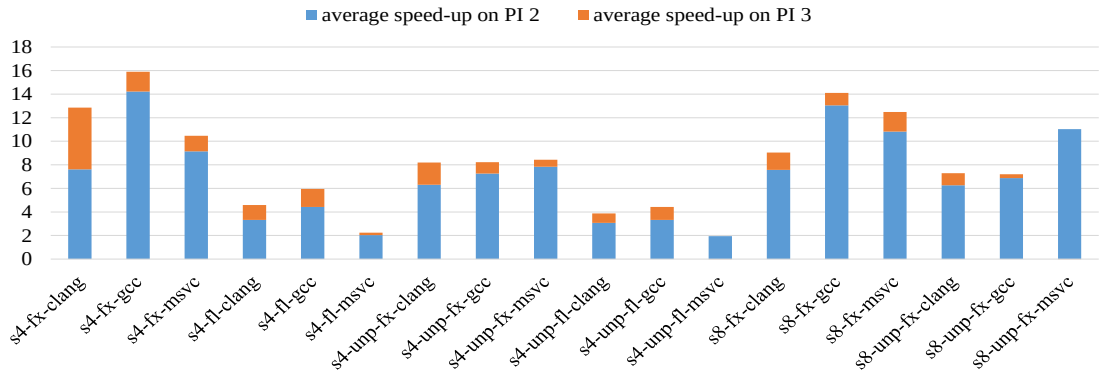

Figure 102: Difference between PI 2 and PI 3 of the speed-up achieved by the vectorized generated code against the MathWorks generated code

6.7.2 Comparison of Generated Code by MathWorks and Proposed Compiler on x86 Processors

This section discusses the summary speed-up of benchmark's execution times achieved on a desktop using the i7-3770 processor compared to that on a desktop using the i7-3820 processor for the generated code by both MATLAB compilers using different experimental settings. Figure 103 shows the minimum, maximum and average speed-up of MathWorks generated code and compiler's scalarized (without use of intrinsics) generated code performance on desktop with i7-3770, compared to that on desktop with i7-3820. For the experiment, both benchmarks using floating point and fixed point data types have been evaluated and the generated code has been examined using Clang, GCC, and MSVC C compilers.

All the evaluated configurations achieve minor speed-ups (average is lower than 2x) with the exception of the MathWorks generated code performance with floating point

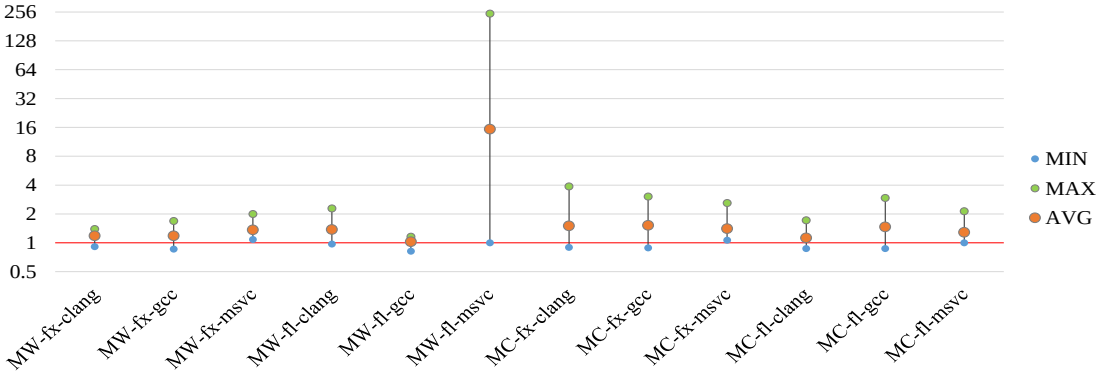

Figure 103: Speed-up of MathWorks and scalarized with no intrinsics generated code on i7-3770 compared to i7-3820

data types compiled by MSVC attaining an average speed-up of 16x. This high speed-up is due to the performance difference between the two targeted processors of FFT-64-v1 benchmark (speed-up of 246.1x). The performance of the scalarized generated code and the MathWorks generated code achieve similar average speed-ups across the different configurations. However, the scalarized generated code obtains higher maximum results compared to that achieved by the MathWorks generated code. Finally, when comparing the performance among C compilers, not significant difference is observed.

Figure 104 presents the speed-up of the vectorized generated code performance achieved on i7-3770 compared to that on i7-3820 using various SIMD options/data types and by compiling the generated code with Clang, GCC and MCVC C compilers. The performance of vectorized code on i7-3770 compared to the performance of the vectorized code on i7-3820 always achieves speed-up on average using SIMD width 4. Regarding the SIMD width 8, the vectorized generated code for some configurations and especially for floating point data types results to reduced performance (slower execution time on average) on i7-3770 against that on i7-3820. Speed-up comparison among C compilers using SIMD width 4, reveals not significant differences. Using SIMD width 8, the speed-up among C compilers vary, but without distinguishing any of them for a general superior performance.

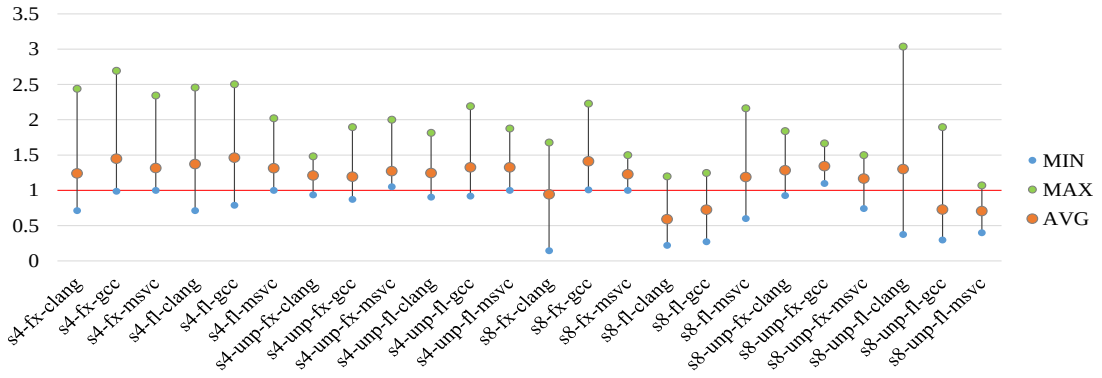

Figure 104: Speed-up of vectorized generated code on i7-3770 compared to i7-3820

Figure 105 shows the average speed-up achieved by the performance of the vectorized generated code compared to that of the MathWorks generated code on i7-3820 and i7-3770. The results show that none of the two targeted processor achieve clearly better speed-ups over the different configurations. Only the benchmark with fixed point data types and SIMD width 4 compiled by Clang and GCC achieves considerable speed-up executing on i7-3770. For the rest of the configurations the speed-up between two targeted processors is similar or even higher for the i7-3820 processor.

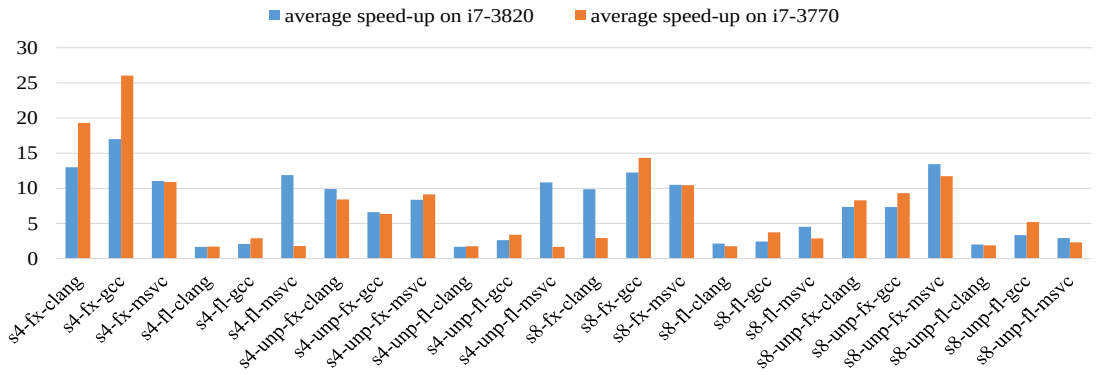

Figure 105: Difference between i7-3770 and i7-3820 of the speed-up achieved by the vectorized generated code against the MathWorks generated code

The ARM and x86 architectures achieve speed-up of performance comparing previous with next generation processors both for scalarized and vectorized generated code. Although, the speed-up achieved comparing the ARM processors is higher than the speed-up between x86 processors with the later comparison leading to slow up in many test cases using SIMD width 8. Comparing the C compilers, MSVC compiler shows a better acceleration between x86 architectures in contrary to the Clang and GCC compilers which exploit better the capabilities of ARM hardware obtaining higher speed-up comparing their performance between processors of that architecture.

6.7.3 Comparison of the Proposed Compiler on TinyBoT and BoT ASIPs

Figures 106 and 107 presents the normalized benchmark code execution times (Table 32) on tinyBoT and BoT processors with packed data types (fig. 106) and unpacked data types (fig. 107). Execution times are on average 2.6x and 1.3x times faster compared to tinyBot execution times with packed and unpacked types respectively using SIMD 4 for FFTs and SIMD 8 for the rest benchmark.

	Fig. 106 and Fig. 107
fft32	737
fft64-v1	3237
fft64-v2	3862
fft128	9503
cfo-32	1180
cfo-64	2300
cfo-128	4540
cordic-64	720155
fir-32x256	719073
mean-32	288
mean-64	544
mean-128	1056
mean-1024	8224
qr-decomp-32	44222

Table 32: Reference values (instructions count) used for normalization at ASIPs comparison

The faster execution times achieved on BoT are due to architectural features and capabilities. BoT processor provides SIMD custom instructions instead of tinyBoT where same scalar instructions are existed. The Mean application on BoT processor with unpacked data types attains worse performance than on tinyBoT processor due to the overhead of packing and unpacking. The SIMD block of this example includes only one SIMD operation which leads to reduced performance because the acceleration is outweighed by the overhead of operands' packing and unpacking. Similar results have been achieved for FIR when unpacked data types and an SIMD processing width 4 are used. Clearly, the execution of SIMD operations with unpacked data types is beneficial only when large SIMD blocks with multiple SIMD operations are used since in this case the intermediate packed results are reused. Furthermore, the mean application of 32

input elements in packed format achieves slower execution time compared to execution time in tinyBoT. The application firstly adds the input packed elements to a vector using SIMD addition, then the resulting vector is used to calculate the overall sum and finally it is divided by the number of input elements. During the calculation of the total vector's sum, operations for unpacking are required and additions for complex fixed point type are executed in software (BoT doesn't provide any scalar special instruction) which add overhead to the execution time and cannot be outweighed by the SIMD addition at the first stage of sum calculation (due to small input size).

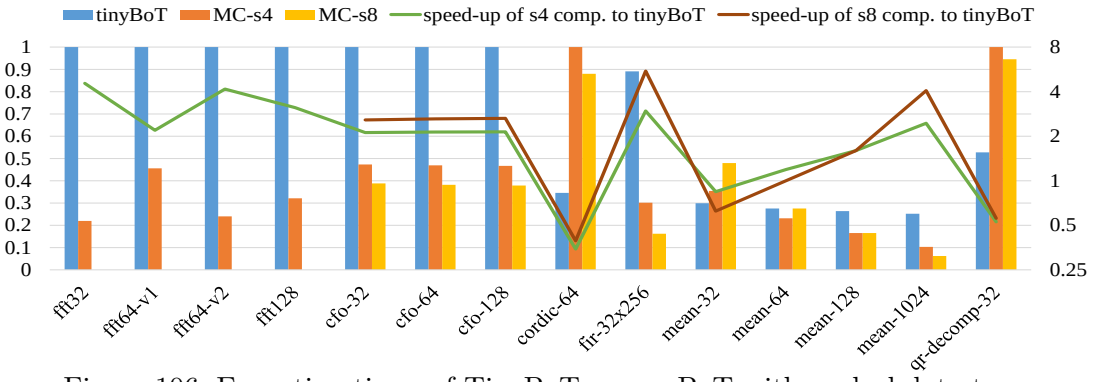

Figure 106: Execution times of TinyBoT versus BoT with packed data types

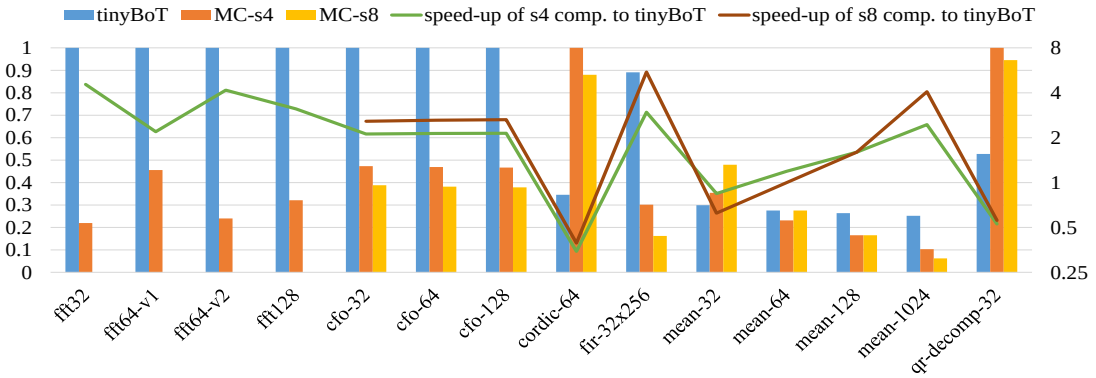

Figure 107: Execution times of TinyBoT versus BoT with unpacked data types

6.8 Comparison Against MathWorks Coder

This section presents the performance of the generated code without the use of the advanced capabilities of the target processor/ASIP compared to that of the code generated by the MathWorks Coder on the different targeted processors.

6.8.1 Comparison of Generated Code Without Intrinsics Against MathWorks Coder on ARM and x86 Processors

This section discusses the execution times of MathWorks generated code compared to the execution times of compiler's scalarized (without use of intrinsics) generated code. A summary of speed-ups on the targeted processors for floating and fixed point data types using the three C compilers is presented, as well as detailed speed-up results for specific targeted processors, C compilers and data types are presented.

Summary comparison of MathWorks generated code versus scalarized code Figure 108 presents the minimum, maximum and average speed-up of MathWorks generated code performance compared to that of compiler's scalarized generated code using different targeted processors, C compilers and benchmark's data types. The average execution time of MathWorks generated code for the most of evaluated configurations is slower than the scalarized generated code. Moreover, for the cases, almost the entire benchmark achieve a better performance. This is shown in the figure 108 where the maximum speed-up values are under the horizontal line of value 1 (some test cases are higher than 1 but for a small factor). Although, the MathWorks floating point generated code compiled with MSVC achieves faster execution times than the execution times of scalarized generated code for floating point data types. Finally, the minimum and maximum speed-ups achieved for the fixed point benchmark present a big difference in relation with the difference of maximum and minumum speed-up achieved on floating point benchmark.

Comparison of MathWorks generated code versus scalarized code for selected architectures Figures 109, 110 and 111 present the benchmark's normalized execution times (Table 33) of MathWorks generated code and compiler's scalarized generated code for selected targeted architectures, C compilers and data types. Figure 109 shows the comparison of the scalarized floating point code by the two MATLAB compilers on Raspberry PI 2 compiled by MSVC. The performance of the generated code by MathWorks Coder achieves a substantial speed-up of 47x for MEAN-32 and MEAN-64 benchmarks and a lower speed-up of 1.7x for MEAN-128 an MEAN-1024 benchmarks while for the rest applications speed-ups from 1.1x up to 1.8x are obtained by the scalarized code. Figure 110 shows the comparison of scalarized fixed point generated code

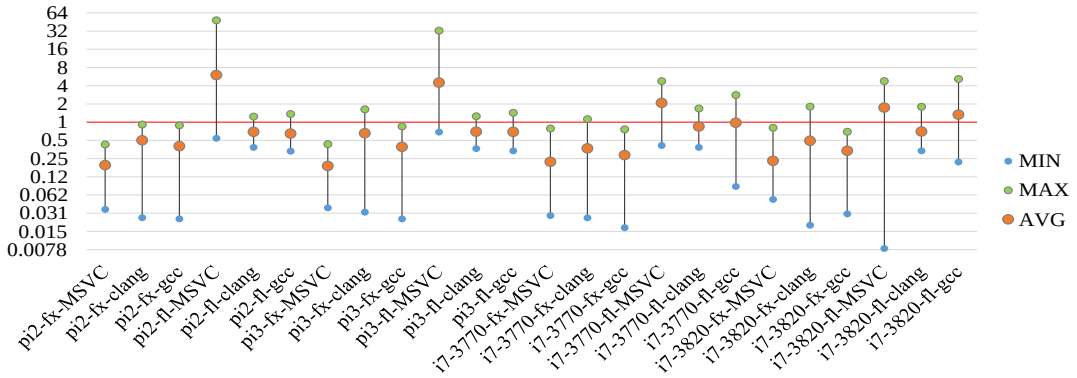

Figure 108: Speed-up of MathWorks generated code compared to scalarized generated code without intrinsics

by the MATLAB compilers on Raspberry PI 3 compiled with Clang. For the current configuration, the performance of the MathWorks generated code attains only a small speed-up of 1.5x for FIR application and the performance of the compiler's scalarized code achieves speed-up up to 33.3x. Figure 111 shows the execution times between MathWorks generated code and generated code by the compiler on the desktop with the i7-3820 processor for floating point data types compiled with GCC. The performance of MathWorks generated code achieves speed-up from 1.1x up to 5.1x for FFT-64, FFT-128 and MEAN benchmarks and the performance of the compiler's scalarized generated code achieves speed-up between 1.05x-4.5x for the rest benchmarks.

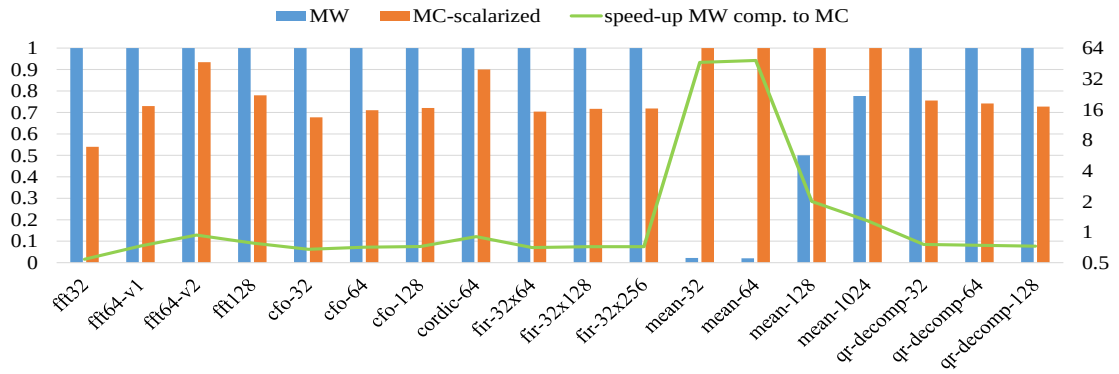

Figure 109: Speed-up of MathWorks floating point generated code compared to scalarized floating point generated code without intrinsics on PI 2 using MSVC

	Fig. 109	Fig. 110	Fig. 111
fft32	2.90	38.50	0.28
fft64-v1	10.23	14.18	1.27
fft64-v2	8.10	61.00	1.35
fft128	22.07	131.73	2.50
cfo-32	18.07	4.00	4.42
cfo-64	34.13	26.21	7.75
cfo-128	66.83	52.24	14.77
cordic-64	628.17	827.56	25.75
fir-32x64	396.20	90.90	10.88
fir-32x128	863.70	216.64	19.33
fir-32x256	1799.53	453.46	39.50
mean-32	0.46	0.27	0.08
mean-64	0.97	0.51	0.08
mean-128	1.80	0.99	0.25
mean-1024	9.70	7.73	1.71
qr-decomp-32	167.27	1249.65	13.40
qr-decomp-64	341.57	343.48	31.28
qr-decomp-128	704.80	695.17	56.60

Table 33: Reference values (exec. time in µs) used for normalization of MathWorks versus scalarized code comparison

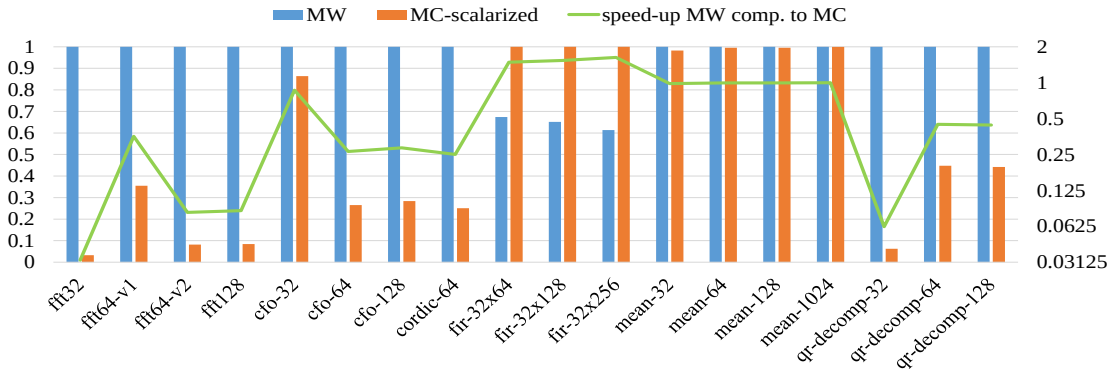

Figure 110: Speed-up of MathWorks fixed point generated code compared to scalarized fixed point generated code without intrinsics on PI 3 using Clang/LLVM

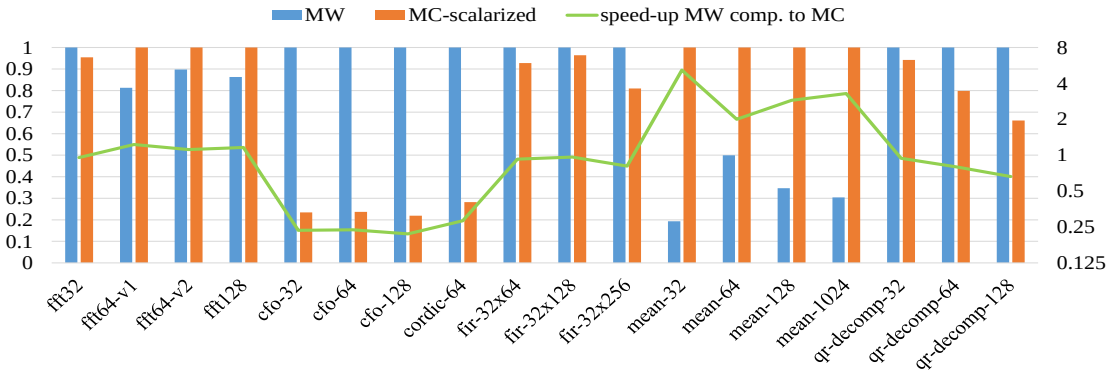

Figure 111: Speed-up of MathWorks floating generated code compared to scalarized floating point generated code without intrinsics on i7-3820 using GCC

Explanation of experimental results of the scalarized code The different execution times between the MathWorks generated code and the compiler's scalarized genarated code is primary due to the different style of generated C code. The MathWorks Coder generates code inlining the majority of MATLAB operations and functions in the main code. Furthermore, it flattens the multidimensional matrices to one-dimensional arrays and it performs internal transformations producing C loops of big scopes which include several operations and references to different arrays. On the other hand, the compiler performs a more direct translation of MATLAB to C code retaining the shape of MATLAB matrices without inlining segments of code in the main program's scope. Moreover, the compiler produces separate nested loops for each array MATLAB statement. The generated C statements include only one operation (or function) involving only the operands'/result's array references and for some cases indexing references are involved as well. Finally, the two MATLAB compilers generate different implementations of fixed point operations. The MathWorks Coder produces general and precision's accurate fixed point operations slowing up the generated code. The compiler produces faster fixed point operations, although for some input values the operation's result isn't accurate. The high maximum speed-up of the scalarized generated code performance compared to that of generated code of MathWorks Coder on fixed point benchmark is due to the fixed point implementations of MATLAB operations and functions.

Conclusion of the scalarized generated code comparison Regardless the generated style of MATLAB code, the execution time is depended on the C compiler and the targeted processor as well. This is proven by the discussed experiment where the same generated code achieve different performance across various processors and C compilers. However, execution time results between processors of same architecture (ex. PI 2 compared to PI 3) for the same generated code and compiled by the same C compiler present similar summary speed-ups (and speed-ups per benchmark). According to the

experiment of this section, the compiler generates quality and simple C code without using any available custom instruction. The scalarized code can be efficiently compiled by the C compilers to produce executable code which in several cases is faster than the code derived by MathWorks Coder for floating and fixed point DSP applications. However, MathWorks Coder is a state of the art MATLAB compiler producing high performance C code and potentially would be faster than the compiler's scalarized generated code for other application domains or applications with other characteristics.

6.8.2 Comparison of Generated Code Without Intrinsics Against MathWorks Coder on TinyBoT ASIP

Figure 112 shows the normalized execution times per benchmark (Table 34) of the code generated by MathWorks and the code generated by the compiler without using custom instructions. In both cases, benchmarks implemented fixed point precision and the generated codes have been mapped on tinyBoT. During compilation with the compiler, MATLAB functions and MATLAB operations with complex numbers were mapped to custom instructions which correspond to C functions. The compiler's generated code achieves better performance for the CFO, FIR and QR-decomposition benchmarks. The differences in the performance of the code generated by the two compilers is mostly due to the following reasons:

- Different C implementations of fixed point operations.
- Different MATLAB indexing. The MathWorks Coder produces flattened arrays instead of the compiler, which retains the shape of the MATLAB arrays when vectorization is not applied.
- Different style of generated C code. For example, MathWorks coder inlines the majority of MATLAB operations and functions in the main code and fuses the scalarized code generated by MATLAB array statements in same for-loops. Such transformations are not applied by the compiler.

	Fig. 112
fft32	4667
fft64-v1	19463
fft64-v2	11442
fft128	46124
cfo-32	11563
cfo-64	22483
cfo-128	45668
cordic-64	3975253
fir-32x256	724074
mean-32	992
mean-64	1952
mean-128	3872
mean-1024	30752
qr-decomp-32	1726926

Table 34: Reference values (instructions count) used for normalization of baseline experiment at tinyBoT

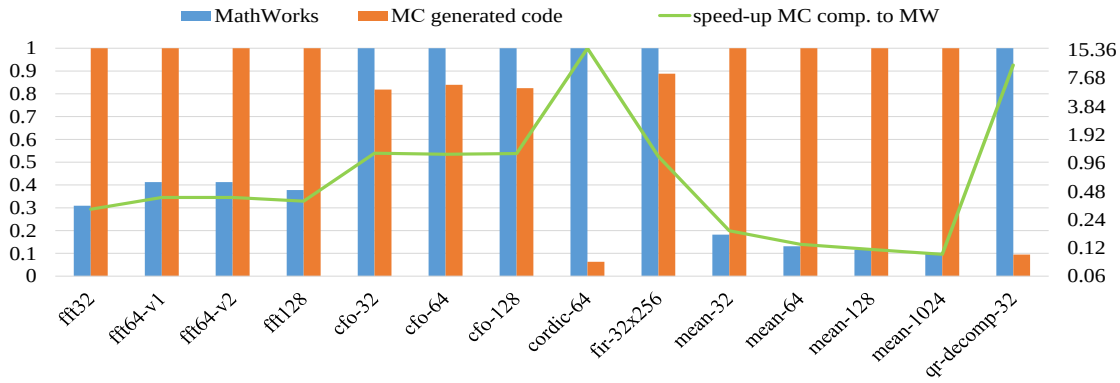

Figure 112: Performance of MathWorks code and generated code without intrinsics on tinyBoT.

6.9 Examination of Clang/LLVM Aggressive Auto-vectorization Options

This section discusses the performance of MathWorks generated code and scalarized generated code by the compiler examining the aggressive Clang optimization options. Figures 113, 114, 115 and 116 present the normalized execution times (Table 35) of MathWorks fixed point generated code on the Raspberry PI 2 (Fig. 113 and 114) and scalarized floating point generated code by the compiler on the desktop with i7-3770 (Fig. 115 and 116) compiling with aggressive Clang optimization options. The *'force SIMD=4'* column in diagrams refers to the *'-force-vector-width=4'* Clang options. The option forces the auto-vectorizer to vectorize the C loops using SIMD instructions of width 4 regardless the decision of auto-vectorization cost model as to whether or not vectorization is beneficial. Moreover, *'force SIMD=8'* column refers to *'-force-vector-width=8'* option which is used to force auto-vectorization with SIMD instructions of width 8. The 'slp-aggressive' column corresponds to the '-fslp-vectorize-aggressive' Clang option enabling the compiler to perform aggressive superword-level parallelism (SLP) of basic-blocks - an optimization of combining similar independent instructions into vector instructions [Larsen and Amarasinghe, 2000]. Finally, the *'slp no'* column corresponds to compilation options that disables the normal SLP vectorization (*'-fno-slp-vectorize'* option) and column *'slp aggr + SIMD=8'* is a test case combining aggressive SLP vectorization and the indication for generation of SIMD instructions with width 8.

Our experiments show that the evaluated compiler's optimizations doesn't always achieve speed-up. On the contrary, the specific optimizations significantly decrease the performance of the executable code in most cases. More specifically, the indication for auto-vectorization of any C loop (*'force SIMD=4'* and *'force SIMD=8'*) obtains speed-up only by using SIMD width of 8 for MathWorks generated code of FIR application on Raspberry PI 2. The rest benchmarks obtains worse performance for that test case compared to the case where the auto-vectorization cost-model decides the loop are beneficial to be vectorized. The aggressive SLP vectorization achieves only a small speed-up on desktop using i7-3770 processor but not for all the benchmarks while disabling the normal SLP vectorization, the performance is increased for some benchmarks on the same processor. Finally, the combination of *'slp aggr + SIMD=8'* options obtain a significant speed-up at FFT64, FFT128, CFO, CORDIC and FIR-32x64 benchmarks on the i7-3770 processor.

In conclusion, the examined Clang optimizations cannot always accelerate the execution times of the MathWorks generated code and compiler's scalarized generated code. Additionally, the specific optimizations don't ensure that the output code is always optimized. On the contrary, the execution time usually attains reduced performance by enabling these optimizations. The worse performance by enabling the *force* options indicates that the cost model for most of the cases vectorizes only parts of code which are beneficial to be vectorized. Although, the test cases which achieve acceleration of

performance applying the specific options shows that there is room for improvement of the cost model to make correct decisions regarding the target processor. Furthermore, the enhancement of performance deactivating the SLP optimization proves that the cost model must perform a more further and accurate analysis to determine the optimal transformations of code preventing the vectorization of basic blocks that is not profitable to be paralleled.

	Fig. 113, Fig. 114	**Fig. 115, Fig. 116**
fft32	97138.83	180.96
fft64-v1	45648.50	893.18
fft64-v2	214133.03	1095.54
fft128	434562.00	2350.02
cfo-32	9264.93	698.78
cfo-64	66805.30	6784.72
cfo-128	131323.13	12416.92
cordic-64	1207044.07	8281.10
fir-32x64	178774.67	7996.44
fir-32x128	402998.60	16334.70
fir-32x256	810921.97	28365.70
mean-32	698.13	25.46
mean-64	1364.57	87.78
mean-128	2659.83	169.96
mean-1024	21071.00	1795.46
qr-decomp-32	3192630.97	31638.96
qr-decomp-64	1433642.37	61933.68
qr-decomp-128	2830109.40	111648.84

Table 35: Reference values (exec. time in µs) used for normalization of aggressive Clang options examination

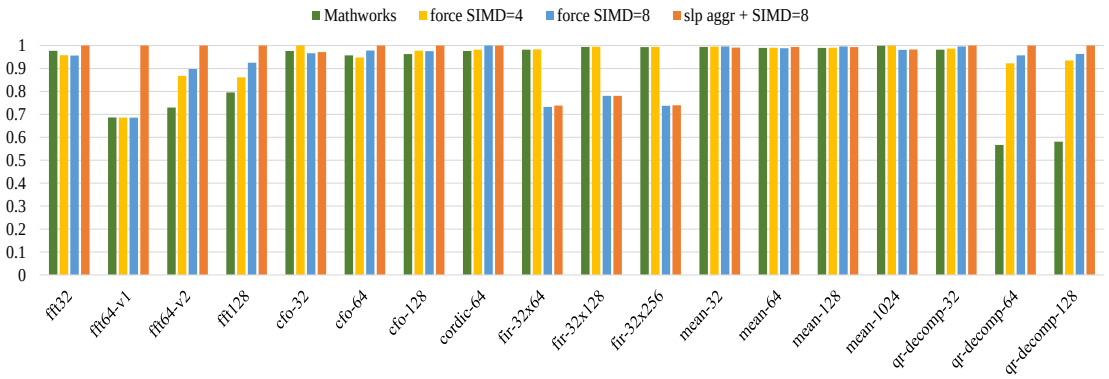

Figure 113: Normalized execution times of MathWorks fixed point generated code, compiled with aggressive Clang auto-vectorization options

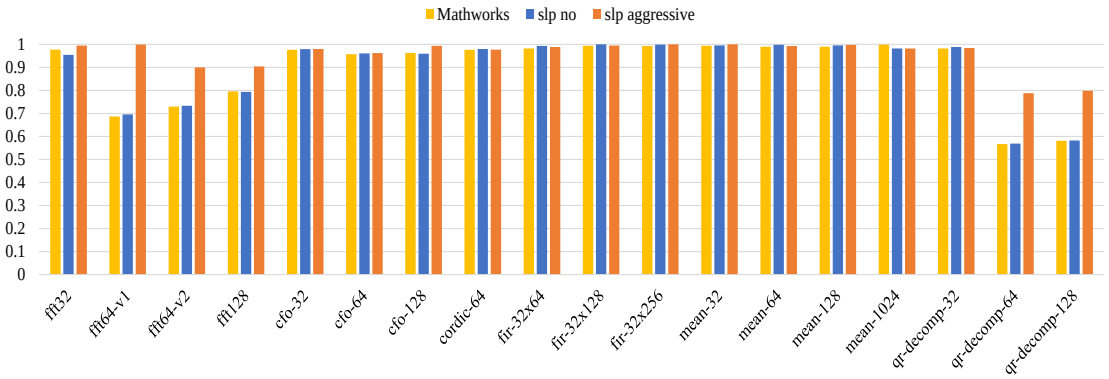

Figure 114: Normalized execution times of MathWorks fixed point generated code, compiled with aggressive superword-level parallelism Clang options

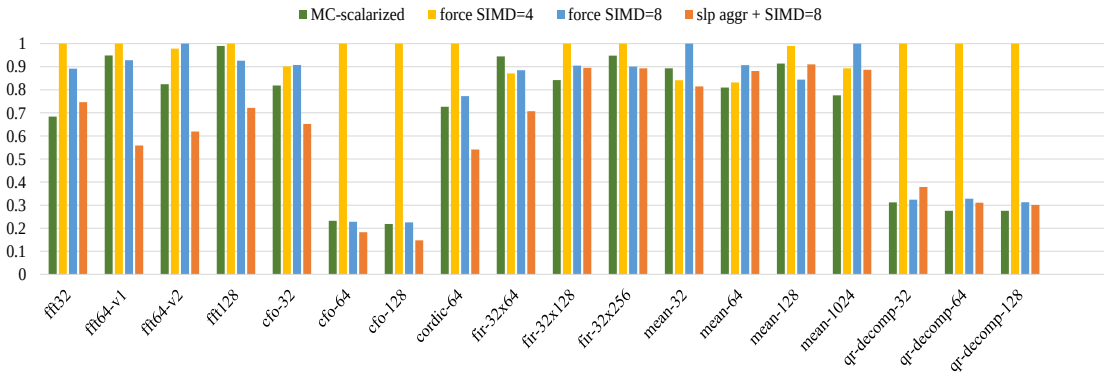

Figure 115: Normalized execution times of scalarized floating point generated code, compiled with aggressive Clang auto-vectorization options

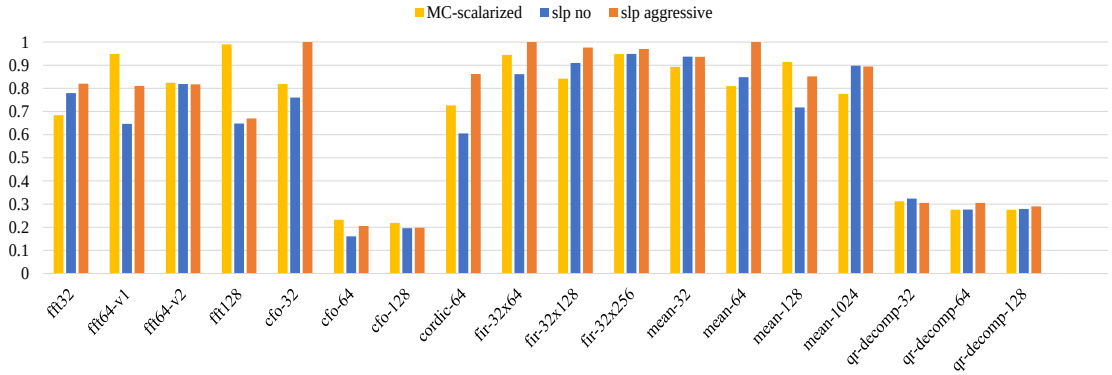

Figure 116: Normalized execution times of scalarized floating point generated code, compiled with aggressive superword-level parallelism Clang options

6.10 Auto-vectorization Evaluation for C compilers

This section discusses the performance of auto-vectorization applied by C compilers. A report of the benchmark's successfully auto-vectorized loops is presented aw well as auto-vectorization speed-up results are presented. Experiments concern the performance of Clang/LLVM, GCC and MSVC auto-vectorizing compilers on the ARM and x86 targeted processors for the benchmark's scalarized generated code by the MathWorks Coder and compiler's scalarized generated code.

6.10.1 Report of Successfully Auto-vectorized Loops by C Compilers

Auto-vectorizing C compilers provide compilation options to print diagnosing messages about the auto-vectorization activity. The specific compiler's option, reports the successfully/unsuccessfully auto-vectorized loops and it prints messages indicating the reason preventing compiler from auto-vectorizing the current loop. The auto-vectorization report options are shown in Tables 42, 43 and 44 for Clang/LLVM, GCC and MSVC respectively.

Tables 36, 37, 38 and 39 present the successfully auto-vectorized loops for a x86 machine supporting *AVX* extension (and consequently SSE). More specifically, the experiment concerns the auto-vectorization of benchmark's generated code by MathWorks Coder for fixed point data types (Table 36) and floating point data types (Table 37), and the compiler's scalarized generated code for fixed point data types (Table 38) and floating point data types (Table 39). Tables show the successfully auto-vectorized loops by Clang/LLVM, GCC and MSVC (columns 2,3 and 4) against the total application's loops (column 5). The last row in tables show the sum of benchmark's loops and the overall auto-vectorized loops.

For the fixed point generated code by MathWorks Coder (Table 36) only the 7%, 14% and 1% of the benchmark loops are successfully vectorized by Clang/LLVM, GCC and MSVC respectively. For the floating point MathWorks generated code (Table 37) Clang/LLVM doesn't vectorize any loop while GCC and MSVC vectorize 10% of benchmark loops. The scalarized generated code with fixed point data types (Table 38) is vectorized by a percentage of 13%, 21% and 6% using Clang/LLVM, GCC and MSVC correspondingly. Finally, Clang/LLVM, GCC and MSVC vectorize the 3%, 16% and 2% of floating point scalarized code. The GCC compiler vectorizes more loops compared to Clang/LLVM and MSVC for both MathWorks generated code and scalarized generated code by the compiler. Furthermore, the three auto-vectorizing C compilers achieve a higher percentage of successfully vectorized loops for the fixed point data types benchmark except the MSVC which vectorizes a higher loops percentage of the scalarized generated code by the compiler with floating point data types. The mostly vectorizable

application is the QR-decomposition where a significant amount of loops are vectorized. CFO and CORDIC applications are also sufficiently vectorized but only for the scalarized generated code by the compiler. The compiler's scalarized code consists of simple loops including only one operation each of them and allowing the C compilers to vectorize higher amount of loops compared to the percentage of vectorized loops of MathWorks generated code. The auto-vectorizing C compilers achieve better applicability at scalarized generated code by the compiler and especially for CFO and CORDIC applications concluding that auto-vectorization is applied more efficiently at code with single operation loops with one-dimensional arrays.

The three examined C compilers don't vectorize a significant amount of loops (only the 10% on average) across the benchmark. Furthermore, the different number of vectorized loops among the C compilers proves that there is a significant room for improvement of auto-vectorization techniques which are applied by the auto-vectorizing C compilers. However, some of the generated loops cannot be vectorized (even with manual code transformations) due to code restrictions, dependencies and hardware limitations. According to the reporting messages some loops are not vectorized because of:

- Outer loop.
- Call instruction cannot be vectorized.
- Loop contains loop-carried data dependences that prevent vectorization.
- Unsafe dependent memory optimizations in loop.

The above cases prevent vectorization of loops and such loops cannot be vectorized even in theory.

	vectorized by Clang/LLVM	**vectorized by GCC**	**vectorized by MSVC**	**benchmark's loops**
cfo	2	3	0	26
cordic	1	1	0	11
fft32	1	2	0	19
fft64-v1	0	0	0	23
fft64-v2	2	7	0	24
fft128	2	7	0	31
mean	0	1	0	27
fir	1	1	0	8
qr-decomp	13	19	3	108
OVERALL	**22**	**41**	**3**	**277**

Table 36: Successfully auto-vectorized loops of MathWorks fixed point generated code

	vectorized by Clang/LLVM	vectorized by GCC	vectorized by MSVC	benchmark's loops
cfo	0	0	1	1
cordic	0	2	0	4
fft32	0	0	0	6
fft64-v1	0	2	0	9
fft64-v2	0	0	0	7
fft128	0	0	0	14
mean	0	1	0	1
fir	0	1	0	3
qr-decomp	0	3	8	39
OVERALL	**0**	**9**	**9**	**84**

Table 37: Successfully auto-vectorized loops of MathWorks floating point generated code

	vectorized by Clang/LLVM	vectorized by GCC	vectorized by MSVC	benchmark's loops
cfo	7	8	7	9
cordic	10	10	5	16
fft32	0	1	0	15
fft64-v1	0	2	0	14
fft64-v2	0	1	0	28
fft128	0	2	0	32
mean	0	0	0	2
fir	2	2	0	5
qr-decomp	12	24	3	108
OVERALL	**31**	**50**	**15**	**229**

Table 38: Successfully auto-vectorized loops of scalarized fixed point generated code

	vectorized by Clang/LLVM	**vectorized by GCC**	**vectorized by MSVC**	**benchmark's loops**
cfo	2	3	2	4
cordic	5	5	0	11
fft32	0	1	0	15
fft64-v1	0	1	0	14
fft64-v2	0	1	0	27
fft128	0	2	0	32
mean	0	0	0	1
fir	0	1	0	4
qr-decomp	0	18	2	91
OVERALL	**7**	**32**	**4**	**199**

Table 39: Successfully auto-vectorized loops of scalarized floating point generated code

6.10.2 Comparison of Auto-vectorizing C Compilers

This subsection presents the performance of execution times achieved by auto-vectorization. Figures 117 and 118 present the minimum, maximum and average speed-up achieved by C compilers auto-vectorization across the benchmark for the MathWorks generated code (Fig. 117) and compiler's scalarized generated code (Fig. 118). The auto-vectorizers achieve higher maximum and average speed-up targeting x86 architecture. However, for that architecture the performance of vectorized code may be worse than disabling auto-vectorization (minimum values below of 1x). Furthermore, the auto-vectorization of Clang/LLVM and MSVC don't affect the performance of generated code for ARM processors. Instead, GCC achieves an average speed-up of 1.8x for the compiler's scalarized generated code targeting ARM architecture. Moreover, GCC obtains in general, better speed-ups compared to Clang/LLVM and MSVC auto-vectorization performance besides the floating point scalarized generated code by the compiler on x86 targeted architecture where MSVC achieves the higher maximum and average speed-up. Finally, there isn't any noticeable difference of auto-vectorization performance between the fixed point and floating point generated code.

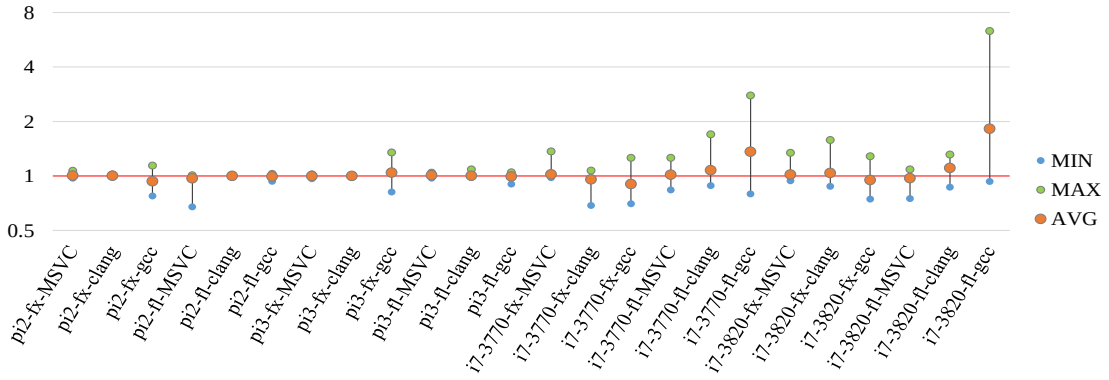

Figure 117: Auto-vectorization speed-up of MathWorks generated code

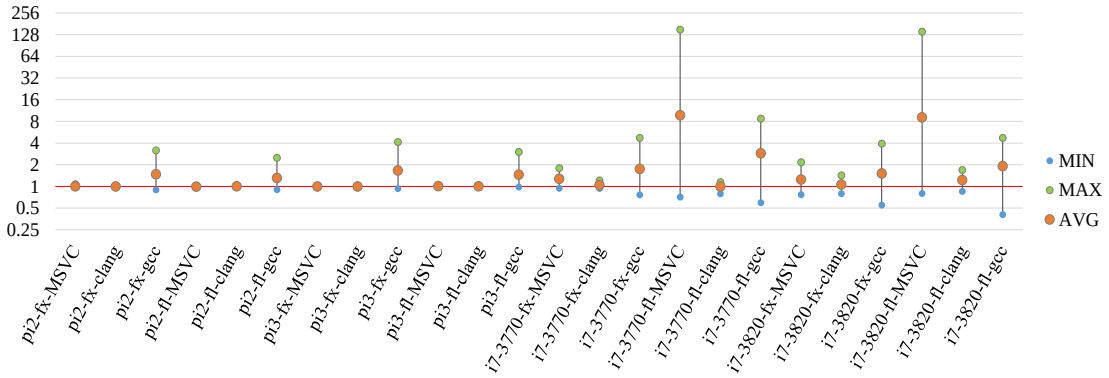

Figure 118: Auto-vectorization speed-up of scalarized without intrinsics generated code

Figures 119, 120 and 121 present the normalized execution times (Table 40) of vectorized compared to non-vectorized generated code for the: MathWorks floating point generated code on desktop with i7-3820 using GCC (Fig. 119), scalarized floating point generated code by the compiler on i7-3820 using MSVC (Fig. 120) and scalarized fixed point generated code by the compiler on Raspberry PI 3 using GCC (Fig. 121). GCC achieves an average speed-up of 1.8x and up to 6.3x for the MathWorks generated code with floating point data types on i7-3820 (Fig. 119). MSVC doesn't always achieve speed-up vectorizing the compiler's scalarized generated code on i7-3820 (Fig. 120). Only the FFT-32, FFT-64-v1, FFT-128, CFO and QR-decomposition perform a better execution time when they are vectorized for the specific target processor. Futhermore, MSVC achieves a remarkable speed-up of 128x for the FFT-64-v1 benchmark on the desktop with the i7-3820 processor. Finally, GCC achieves an average speed-up of 1.6x and up to 4.1x vectorizing only the CFO, CORDIC and FIR applications of the scalarized fixed point generated code by the compiler on Raspberry PI 3 (Fig. 121).

	Fig. 119	**Fig. 120**	**Fig. 121**
fft32	0.33	0.56	0.87
fft64-v1	1.32	224.60	3.90
fft64-v2	1.21	1.80	4.20
fft128	2.16	5.50	8.81
cfo-32	6.29	2.10	4.62
cfo-64	10.28	5.70	9.18
cfo-128	17.86	8.68	18.20
cordic-64	32.79	40.36	311.96
fir-32x64	18.74	15.64	208.31
fir-32x128	39.12	34.30	456.20
fir-32x256	71.85	71.84	948.89
mean-32	0.09	0.13	0.19
mean-64	0.05	0.20	0.35
mean-128	0.30	0.50	0.67
mean-1024	0.96	4.98	5.17
qr-decomp-32	22.57	21.58	82.02
qr-decomp-64	45.93	43.08	157.88
qr-decomp-128	88.49	85.96	321.35

Table 40: Reference values (exec. time in µs) used for normalization of auto-vectorization results

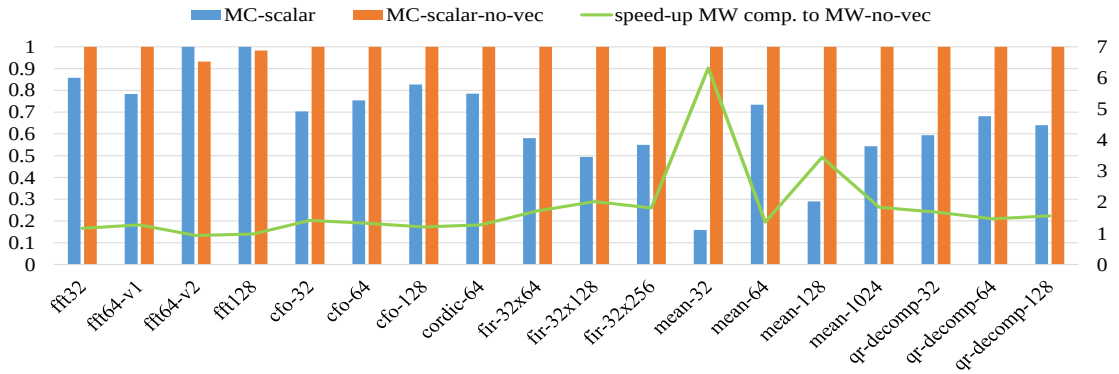

Figure 119: GCC auto-vectorization speed-up of MathWorks floating point generated code on i7-3820

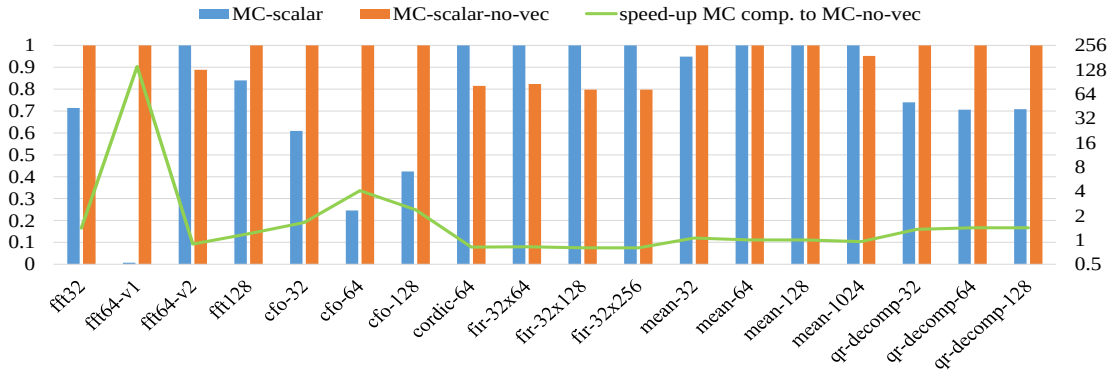

Figure 120: MSVC auto-vectorization speed-up of scalarized floating point generated code on i7-3820

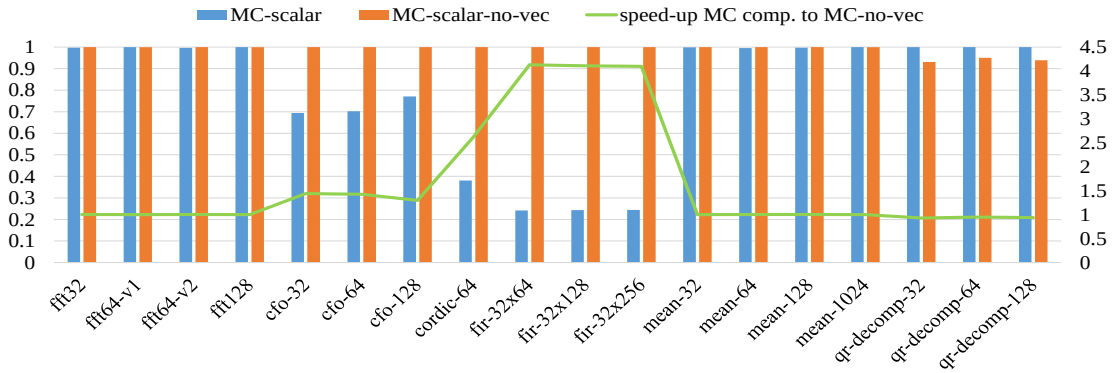

Figure 121: GCC auto-vectorization speed-up of scalarized fixed point generated code on PI 3

Figures 122, 123, 124 and 125 present the minimum, maximum and average speed-up achieved by the three C compilers per benchmark. The four different figures concern the fixed and floating data types and the different generated code derived from MathWorks Coder and compiler. The FIR and MEAN fixed point applications of MathWorks generated code (Fig. 122) can be vectorized attaining a speed-up of 1.6x. However, the vectorization of MathWorks fixed point generated code across all benchmarks performs worse than when disabling auto-vectorization among the different C compilers and targeted architectures. Auto-vectorization on MathWorks floating point generated code (Fig. 123) doesn't obtain any significant speed-up across the benchmarks. However, FIR MEAN and QR-decomposition can be vectorized achieving a maximum speed-up of 2x, 6.3x and 1.7x respectively. The scalarized fixed point generated code by the compiler (Fig. 124) achieve a speed-up up to 4x with 2x on average by the three compilers for FIR application. Rest of the benchmarks attain a maximum speed-up close to 2x and a minimum speed-up which is negative for the majority of the benchmarks. The auto-vectorizing C compilers achieve substantial speed-ups on the scalarized floating point generated code by the compiler (Fig. 125). Auto-vectorization speed-up is achieved for FFT-64-v1, CFO, CORDIC, FIR and QR-decomposition applications with an average speed-up of 32x, 2x, 2x, 1.8x and 1.6x (speed-up up to 128x, 8x, 6x, 3.5x and 4x) respectively. The high average and maximum speed-up of FFT-64-v1 is due to the vectorization speed-up achieved by MSVC on x86 architecture.

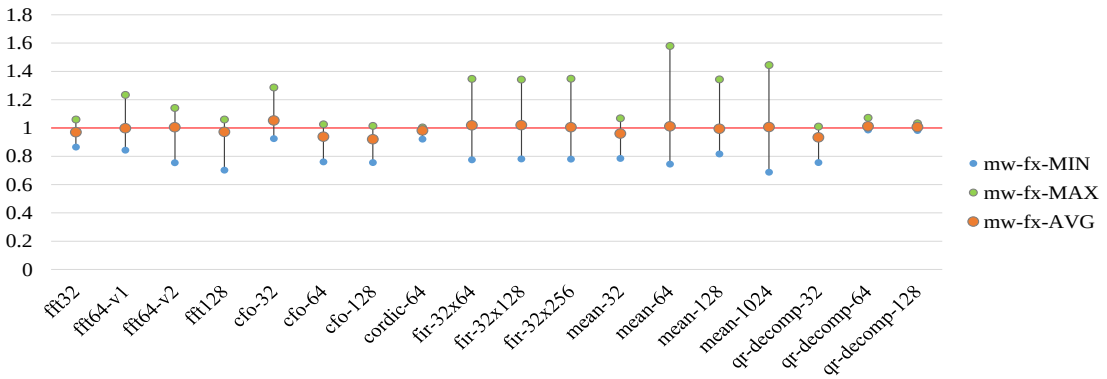

Figure 122: Auto-vectorization speed-up of MathWorks fixed point generated code

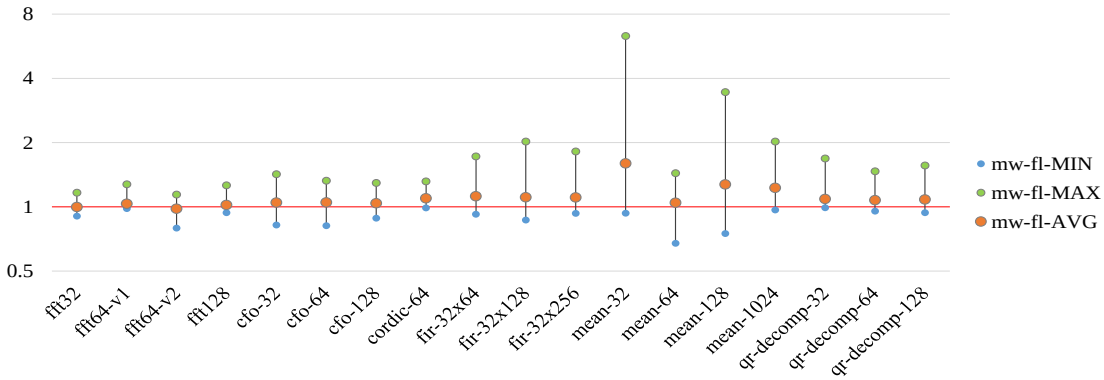

Figure 123: Auto-vectorization speed-up of MathWorks floating point generated code

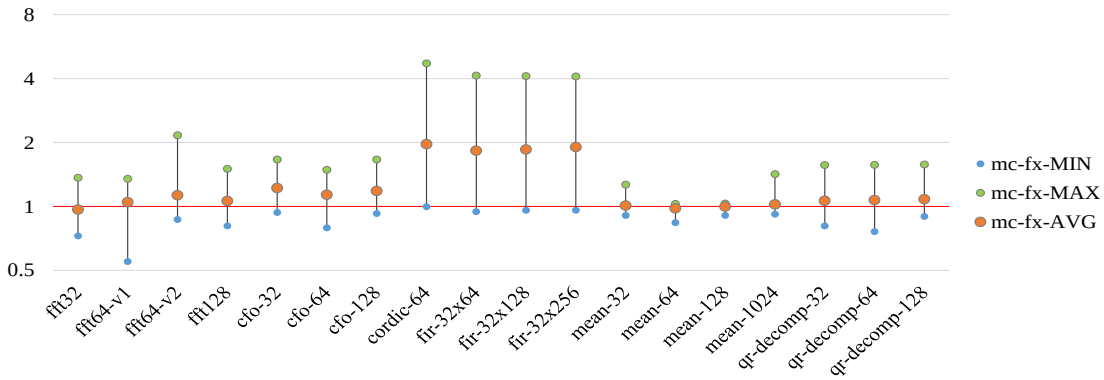

Figure 124: Auto-vectorization speed-up of scalarized fixed point generated code

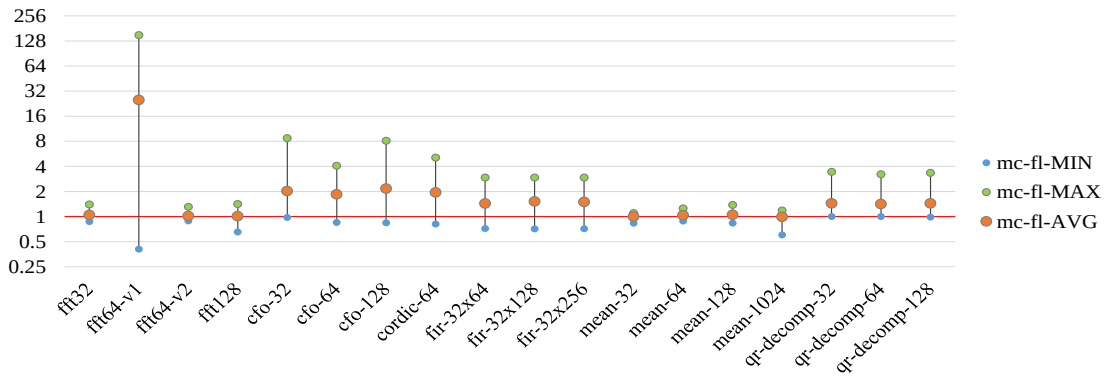

Figure 125: Auto-vectorization speed-up of scalarized floating point generated code

6.11 Comparison of C Compilers on the Generated Code

This section presents a comparison of C compilers (using also different operating systems) performance regarding the benchmark's execution times over the different targeted processors. The various diagrams in the section concern the different styles of generated code by compiler and MathWorks Coder. The experiments/diagrams present the average speed-up of benchmark among the C three compilers for floating point and fixed point data types on the various targeted processors. For the conduction of the experiment, the speed-up per benchmark has been calculated by the comparison of executable code derived from each C compiler against the worst executable code among the three C compilers. Finally, the generated code compiled with Clang and GCC has been executed on Rasbian OS for ARM processors and on Linux Ubuntu OS for x86 processors. The generated code compiled by MSVC has been executed on Windows IoT OS targeting ARM processors and Windows 10 OS targeting x86 processors.

Figure 126 presents the performance of MathWorks generated code using different C compilers. Clang and GCC achieve similar performance across the different targeted processors. MSVC shows worse performance when using fixed point data types for all the targeted processors as well as using floating point data types on i7-3820 processor.

Figure 127 shows the performance of compiler's scalarized generated code using the three C compilers. For scalarized code, GCC achieves the faster execution times across benchmark except targeting i7-3820 and using floating point data types. For that configuration Clang achieves the best performance instead. Furthermore, Clang and GCC obtain substantial speed-up compared to the performance of MSVC for Raspberry PI 3 and x86 processors using floating point data types.

Figure 128 presents the performance of vectorized generated code with packed data types of SIMD width 4 compiled with Clang, GCC and MSVC. For vectorized code, GCC achieves better performance than Clang across all the targeted processors and by using fixed or floating point data types. Comparing GCC against MSVC, GCC achieves a significant speed-up across the targeted processors except the i7-3770 and i7-3820 using floating point data types where MSVC attains a minor speed-up against GCC and Clang.

Figure 129 shows the performance of vectorized generated code with unpacked data types of SIMD width 4 compiled with Clang, GCC and MSVC. For that generated code, significant performance speed-up is obtained by Clang and GCC compared to MSVC performance on Raspberry PI 3 using fixed point data types and on Raspberry PI 3 using floating point data types. Moreover, GCC attains worse performance than Clang and MSVC for x86 processors.

Figure 130 presents the performance of vectorized generated code with packed data types of SIMD width 8 using the three C compilers. GCC achieves better performance compared to Clang for any targeted processors. GCC also obtain enhanced performance compared to MSVC except the x86 processors using floating point data types.

Figure 131 presents the performance of vectorized generated code with unpacked data types of SIMD width 8 using the three C compilers. For that style of generated code Clang achieves better performance than GCC on the different processors. The Clang and MSVC performing in a similar way across the targeted processors except of the case of the Raspberry PI 3 using fixed point data types, where Clang achieves better performance.

In conclusion, GCC achieves better performance compared to Clang among the different styles of generated code. However, using unpacked data types Clang is most appropriate compiler to be used since it achieves a minor speed-up on benchmark's execution compared to GCC for any targeted processor. Regarding MSVC, the compiler attains a substantial speed-up compared to GCC and Clang for vectorized generated code using floating point data types on x86 targeted processors.

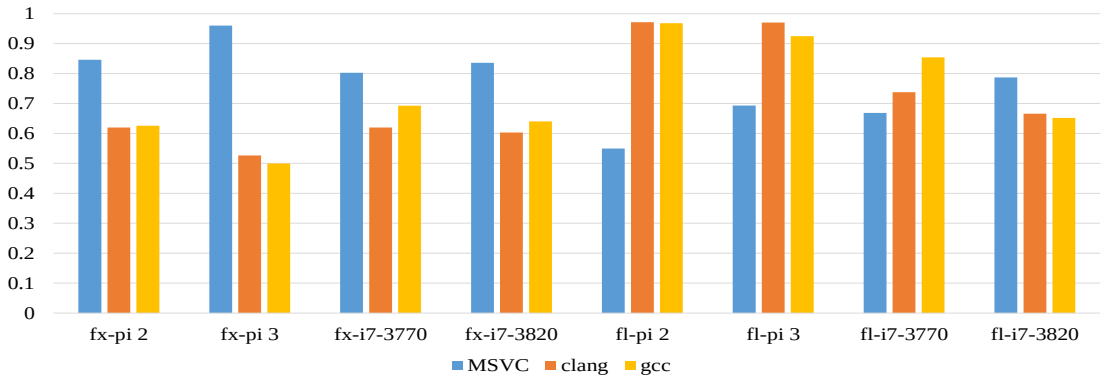

Figure 126: Performance of the MathWorks generated code by C compilers

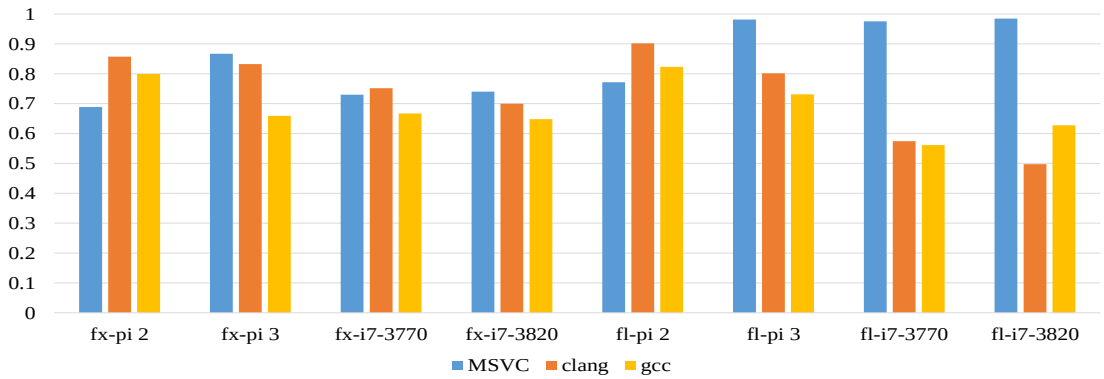

Figure 127: Performance of the scalarized generated code by C compilers

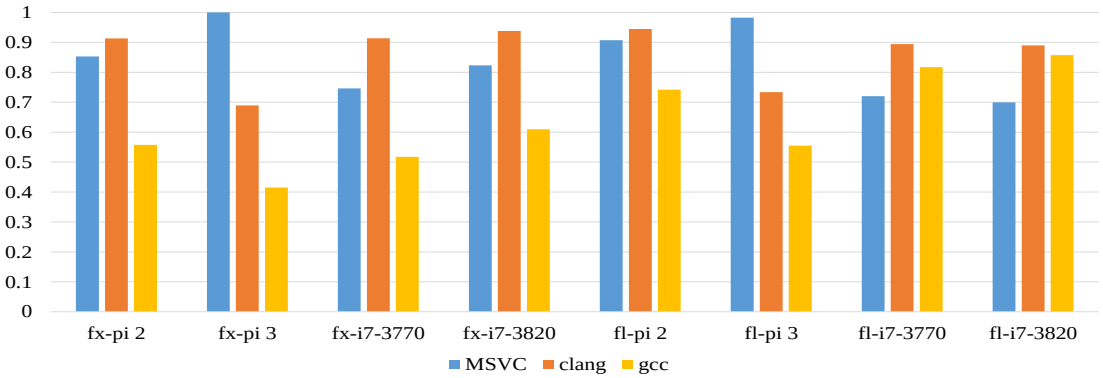

Figure 128: Performance of the vectorized (SIMD width 4) generated code by C compilers

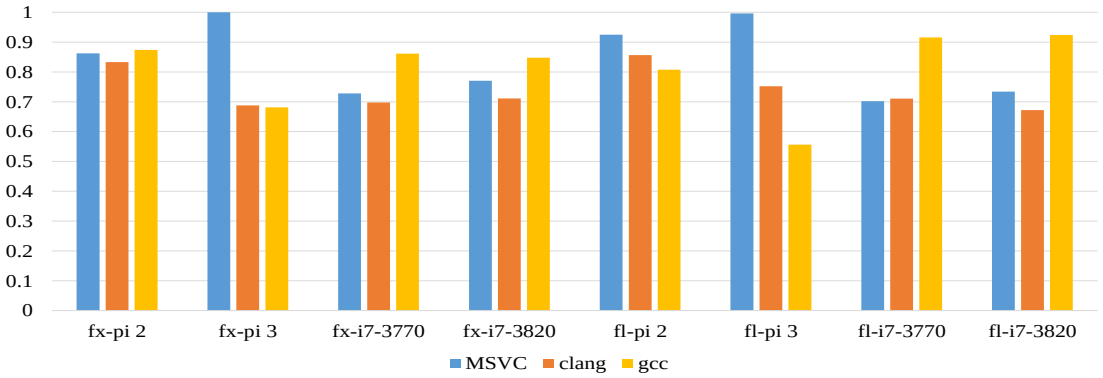

Figure 129: Performance of the vectorized (unpacked data with SIMD width 4) generated code by C compilers

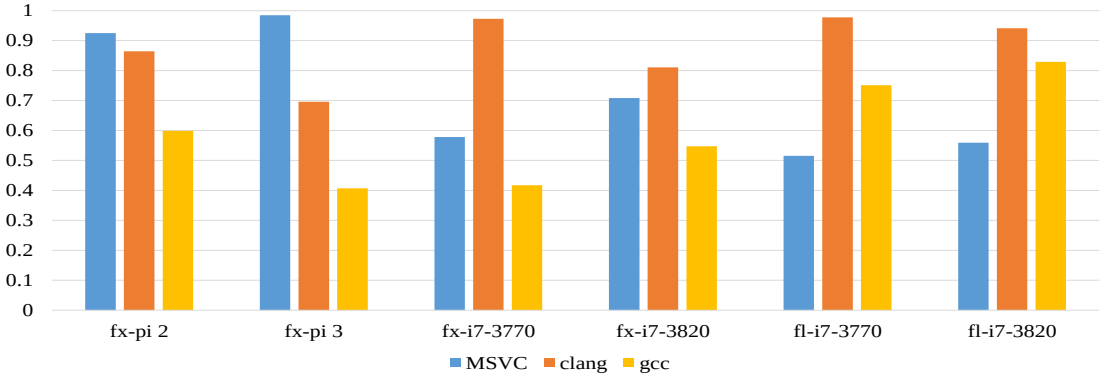

Figure 130: Performance of the vectorized (SIMD width 8) generated code by C compilers

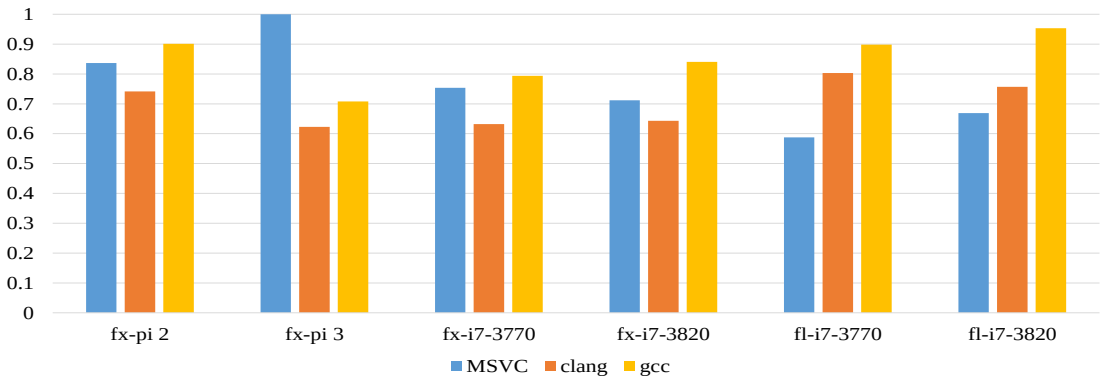

Figure 131: Performance of the vectorized (unpacked data with SIMD width 8) generated code by C compilers

6.12 Compilation Times

The compilation times of the compiler are on average 13.1 times faster than the MathWorks compiler for compiling the selected set of benchmark's. Table 132 shows the compilation times achieved for each benchmark by the MathWorks Coder and the MATLAB compiler using for compilation unpacked data types of SIMD width 4. This is an impressive additional advantage of the compiler. It is difficult to fully explain the compilation times of the MathWorks compiler because its implementation details are not available. The speed up achieved by the compiler is possibly due to the fact that the MathWorks compiler uses generic optimization strategy and extensively applies scalarization without considering special features/custom instructions of the target processors. The compiler generates high quality code by focusing on the exploitation of special features/custom instructions of the targeted processor.

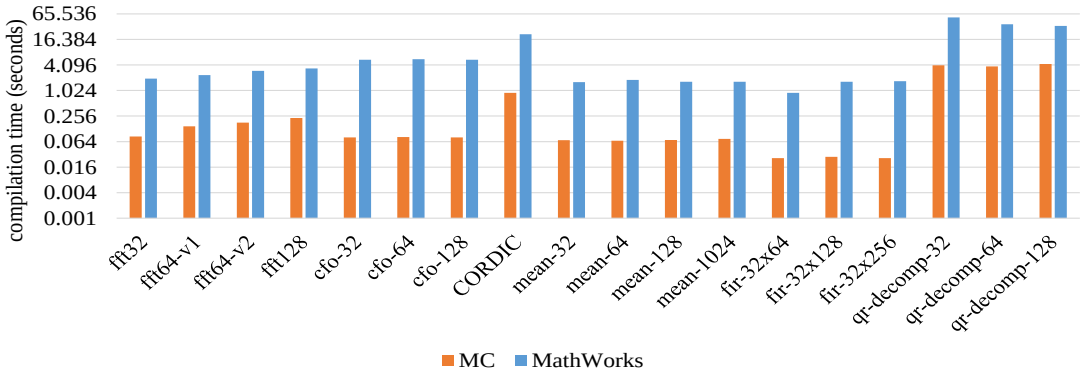

Figure 132: Compilations times.

6.13 Conclusion

In this chapter the performance of the generated code by the MATLAB compiler has been evaluated. The experimental evaluation was conducted comparing the performance of the generated code against the performance of the generated code by the MathWorks coder. The experimental environment composed by a set of eight benchmarks from signal processing domain in floating and fixed point. The target architectures where the generated code was evaluated are two ASIPs, one of them supporting SIMD instructions, and four general purpose processors of ARM and x86 architectures. The experimental settings include different configurations of SIMD widths and data types between packed or unpacked. Finally, various C compilers (Clang, GCC, MSVC) were used for the compilation of the C generated code.

The experimental results show that the performance of the vectorized code using packed data types surpass the performance of the code that is generated by MathWorks coder. The compiler is able to produce vectorized code that is executed faster than the MathWorks Coder generated code on various target ASIPs/processor by using different settings of C compilers and data types. However, by using unpacked data types the performance of the vectorized code for some benchmarks isn't always higher comparing to the performance of the MathWorks generated code. This is due to the fact that for some applications the overhead of packing/unpacking operations is higher than the acceleration is achieved by SIMD processing. Additional experiments were conducted concerning the performance of the scalarized code that is generated by the MATLAB compiler as well as the performance of the auto-vectorization by C compilers. The performance of the scalarized code by the MATLAB compiler against the MathWorks generated code vary among the different target architectures and C compilers. The varying results shows that the execution performance of the generated C code is strongly depended on the target architecture and the underlying C compiler. Regarding the evaluation of auto-vectorizing C compilers, the experiments prove that only a small percentage of benchmark loops can be vectorized by the C compilers. Additionally, vectorizing the scalarized code that is generated by the two MATLAB compiler, only minor speed-ups are achieved.

Chapter 7

Conclusion and Future Work

MATLAB is a popular language for rapid prototyping of algorithms in several domains including engineering, science and embedded computing. In the context of embedded systems and Systems-on-Chip MATLAB is used for the development of executable specification. Development of applications with the MATLAB language avoids the detailed specification of the algorithm where strong programming skills and a lot of effort/time are required. Consequently, MATLAB compilation to code that can be implemented in an automated way to software or hardware has been made essential in the domain of embedded systems.

Several approaches have been presented for the compilation of MATLAB language to implementation code providing a more efficient execution environment. However, existing compilation frameworks don't leverage the capabilities of the target architecture and especially the processor's vector units. Previous related studies focus on the generation of code which is applicable on parallel distributed architectures and GPUs. Moreover, by using the MathWorks compilation tools with an auto-vectorizing C compiler lead to sub-optimal results due to deficiencies presented in C auto-vectorization process using as input the scalarized generated code by MathWorks tools.

In this thesis a MATLAB to C compiler is presented targeting embedded systems domain and especially Application Specific Instruction Set Processors. The compiler matches the MATLAB expressions with the available hardware modules and the custom instructions of the targeted architecture (such as instructions for SIMD processing and for complex arithmetic). This is achieved through the use of a parametrized processor model where the custom instruction set is described. This approach allows the compiler to efficiently support any target processor. Additionally, the information contained in the target processor description is used for the type inference of functions mapped with specialized intrinsics to determine the types of function call. The generated code is ANSI C in which the custom instructions are represented as intrinsic functions that can be still exploited by conventional C compilers at a later stage. Moreover, the developer can configure the compiler to generate either SIMD or scalarized C code at the output and is able to select appropriately the derived data types among floating point, fixed point or integer.

The compiler has been benchmarked on a set of eight DSP benchmarks using a variety of ASIP and general purpose processors against the generated code by MathWorks

Coder. More specifically, two different ASIP processors (BoT ASIP supports SIMD), two processors of ARM architecture and two processors of x86 architecture have been employed. The Synopsys ASIP Designer [ASIP Designer, 2016] have been used for the compilation of the benchmark and the simulation of ASIPs. For the compilation of the generated code by the MATLAB compilers, the GCC, Clang/LLVM and MSVC C compilers have been used. The code generated by the approach achieves a speed-up of 2x-74x on tinyBot and a speed-up of 2x-97x on BoT compared to the code generated by MathWorks MATLAB-to-C compiler. Moreover, the performance of the vectorized code using floating point data types compared to that of MathWorks achieves an average speed-up of 12.3x (using SIMD width 4) and 8.9x on the ARMv8 (Raspberry PI 3) and i7-3770 x86 processors respectively. Finally, experimental results have been conducted to evaluate the performance of the auto-vectorization. The experiments prove that only a small percentage of benchmark loops (compiling with MATLAB compiler or MathWorks Coder) can be vectorized by the evaluated C compilers.

7.1 Contribution of Dissertation

The compilation framework that has been presented in the current thesis allows the optimized C code generation of MATLAB language targeting embedded systems and advanced architectures with instruction set extensions. The contributions of the thesis are:

- A parametrized processor model that can be used for the specification of the target processor architecture. The model provides the description of the necessary information for the production of optimized C code for a target architecture including its custom instructions, native data types and operations for packing (or unpacking) data to vectors. The processor model enables a multi-target compilation framework leading to the optimized code generation of the target architecture.
- An instruction selection algorithm for the mapping of MATLAB code with the available hardware modules according to the specification of the target processor. The algorithm matches the MATLAB operations/functions with custom instructions of the target architecture depending on their type of operands/parameters and the style of the generated code (scalarized or vectorized). The algorithm allows broadness of application matching function calls and operations of any type with custom instructions of diverse architectures.
- A detailed discussion of the compiler's infrastructure about the support for parallel processing. The compilation framework includes an algorithm for the introduction of instructions in the AST for the packing/unpacking of data which are processed in parallel. A procedure for the elimination of the redundant packing/unpacking instructions is also presented leading to the optimized application of the packing/unpacking introduction algorithm. The methodology enables the preparation

of AST for SIMD code generation with unpacked data types. The combination of two processes leads to the introduction of packing/unpacking statements that are required for the valid code generation.

- A detailed description of the code generator regarding the vectorized code generation as well as the code which is related to the target architecture. The innovative parts of the code generation procedure which are discussed in the thesis are:

  A) The for-loops that are generated for the implementation of the SIMD blocks.
  B) The code generation of packing/unpacking instructions and their corresponding for-loops.
  C) The printing of the vector C variables focusing to the generation of the vector indexing.

- A type inference approach for type checking and inference of function calls which are matched to custom instructions using the parametrized processor model. It allows the type inference of function calls that are subsequently matched to customized instructions according to their description in the processor specification. The approach provides a sophisticated method for functions type resolution combining the description of the type result in the processor model with the type of function parameters. The type inference approach discussed in the thesis introduces a methodology to infer function calls according to the architecture's implementation with respect to the input parameter types.
- The evaluation of the compiler's performance regarding the speed of the generated code and its comparison against the generated code by MathWorks Coder. The thesis presents an extensive benchmarking of the MATLAB compiler in an experimental environment of various options including different processors, C compilers and operating systems, and diverse data types. An examination of the C compilers auto-vectorization is presented in the current thesis as well.

7.2 Future Work

The current thesis presents an integrated compilation framework for the generation of C code from MATLAB source code. However, further extensions may be implemented to improve the derived code and expand compiler's applicability. Topics for further research and development are reported below.

- Extensions of the core functionality to make compiler even broader applicable:
  - The code generator can be expanded to generate vectorized code for control flow statements with array dimension conditions. Sophisticated methods as these are described in [Kennedy and McKinley, 1990], [Shin et al., 2005] and [Shin, 2007] may be implemented in the current compilation framework

to support the vectorized code generation of SIMD blocks including array conditional control flow statements.
  - The compiler's code generator can be extended to generate OpenCL code. OpenCL is a framework for a development of programs that are executed across heterogeneous platforms improving their performance. OpenCL supports a subset of C++ language allowing parallel programming of applications. The code generator can be expanded to generated parallel code assigning the array functions/operations to the compute units of the target architecture. The automatic SIMD block extension which is discussed above can be used to identify the segments of code that are translated to OpenCL code (*kernels*).

- Extensions of methodologies to improve the user support:
  - The parametrized processor model and the instruction selection algorithm can be extended in order to map customized instructions with complicated expressions or multiple MATLAB statements instead of matching only single operations/functions. Primary example of the usage of this extension is the mapping of MATLAB expressions such as $a*x+b$ with the cumulative operations of hardware which are commonly provided by DSP processors. The extension can be implemented by using patterns for the description of the MATLAB code represented by customized instructions. A sophisticated instruction selection algorithm is also required to map the MATLAB code with customized instructions. The basic principle of the algorithm's extension is to iteratively map MATLAB operations/ functions which are subsequently used as operands for further mapping. MATLAB code reordering could enhance the results of mapping process considering that compilation time should be preserved low.
  - An automatic SIMD block identification can be implemented allowing to the compiler the selection of the parts of source code that are beneficial to be vectorized. The extension require analysis of the input code and a cost model to determine the segments of code that will be translated to SIMD code.

- Improvement of the implemented methodologies to better support the compiler's functionality:
  - Several optimizations can be implemented to improve the performance of the generated code. MATLAB is a programming language commonly used for its array syntax. Therefore, reuse distance analysis and code transformations [Lezos et al., 2015] and [Lezos et al., 2016] could lead to considerable performance acceleration. Furthermore, loop fusion of the scalarized generated for-loops can be applied to increase data locality. Finally, additional experiments may be conducted to investigate the capabilities of auto-vectorizing

C compilers on the different styles of the generated code by enabling and disabling the MATLAB compiler's implemented optimizations.

- The type inference mechanism is possible to be expanded supporting variables with unknown dimensions in compile time. This can be implemented representing the unknown dimensions with symbolic values instead of using constants inferred dimensions. However, runtime dimensions detection or variables with variant types wouldn't be a priority for compiler's extension since these characteristics would reduce the performance of generated code and are rarely used in embedded systems domain applications.

- Additional evaluation of compiler to examine thoroughly the performance of generated code:

  - Finally, the compiler can be evaluated on applications from other domains as well as on additional architectures to explore its performance in order to potentially optimize it. Especially, the examination of the compiler in a broad range of applications would improve the compilation framework refining the methodologies which are discussed in the thesis and specifically the optimizations and the code generation process.

Chapter 8

Appendix

The appendix includes technical content regarding the MATLAB compilation framework and further benchmarking results of the compiler. Section 8.1 presents the MATLAB compiler options, section 8.2 presents the compilation options that are used in the experiments by the C compilers and section 8.3 presents examples of the description of the parametrized processor model for the target architectures. The rest chapter shows further experimental results. The appendix includes a large set of detailed results related to that are presented in the chapter 6. The figure 133 shows the association of the experiments between chapter 6 and the appendix.

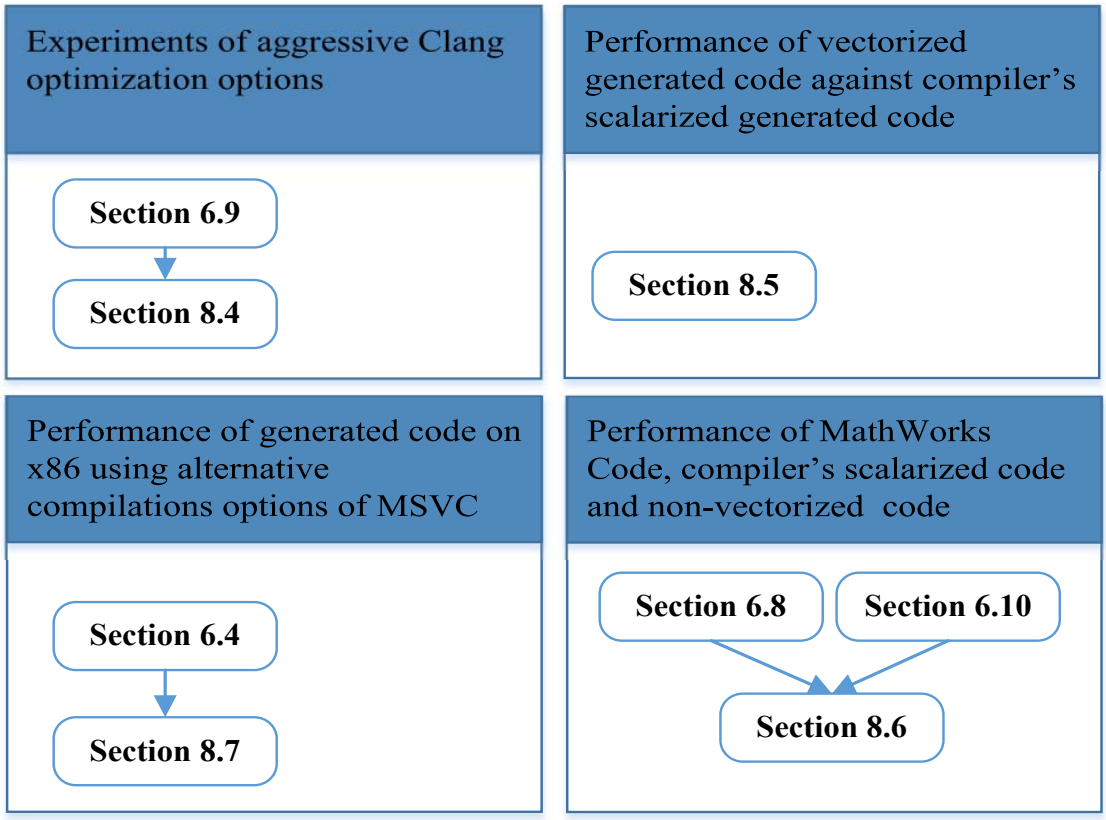

Figure 133: Dependence graph of experiments.

8.1 Compiler Options

In this section the compiler's options are discussed. The compiler's program accepts options and filenames as operands regarding the input, output and configuration files. Compilation options are also provided for the specification of the target architecture and inclusion of additional directories as well as supplementary options are available for debugging purposes. Table 41 shows the complete list of compiler's options. A description of the compiler options is listed below:

- *-help*: Print the options usage.
- *<MATLAB_file.m>*: The filename of input file.
- *-o <output_file.c*: The filename of output file.
- *-dir=<dir>*: It specifies additional directories (it may be used multi times in the command line). The MATLAB files included in additional directories are also compiled when a primary function is called by the compiling MATLAB code.
- *-arch <target>*: It specifies the target architecture. Available values are *ARM*, *x86*, *tinyBot*, *BoT* and *other_target*. The compiler uses the option to handle specifically the MATLAB code for some special cases. For instance, current ASIPs support only fixed point arithmetic and complex vector data types. For a MATLAB assignment of a floating point constant to an unknown type variable, the compiler will generate a transformed fixed point constant and the statement's right-side variable will be inferred as fixed point type. For a MATLAB statement with non-complex variables generated in SIMD-style, the compiler will generate vectorized code using any available complex vectors (instead of non-complex).
- *-noMain*: Using this option the compiler doesn't generate a *main* C function. Otherwise the function is generated with code that it calls the primary function of MATLAB input file.
- *-noConstProp*: During type inference stage, constant propagation is applied to improve the deduction of variables' types. Constant propagation is disabled including the option in command line.
- *-conf=<conf_file>*: The filename of XML file with the description of parametrized processor model.
- *-parseTree*: Using this option, a graph representing the parse tree is produced.
- *-ast*: Using this option, a graph representing the AST is produced.
- *-matlab*: It commands compiler to generate the MATLAB code from AST.
- *-analyze*: It enables the *-parseTree*, *-ast* and *-analyze* options. The options are used for debugging purposes.

The following line is an example of the MATLAB compiler usage:
./matcom -o fft32.m fft32.c -conf ARM_instr.config -arch ARM -noMain -dir=./myScripts

Compiler options	description
- -help	Prints the description of the command line options
<MATLAB_file.m>	The input file
-o <output_file.c>	The output file
-dir=<dir>	Additional include directory
-arch=<target>	Target architecture
-noMain	Disable the generation of C *'main'* function
-noConstProp	Disable constant propagation in type inference stage
-conf=<conf_file>	The XML file of parametrized processor model
-noC	Disable the generation of C code
-parseTree	Generate parse tree graph
-ast	Generate AST graph
-matlab	Print MATLAB input code
-analyze	Enable -parseTree, -ast and -matlab options

Table 41: Compilation options

The command compiles the *fft32.m* MATLAB file generating the output at the *fft32.c*. The target processor is ARM architecture and the parametrized processor model is described in the *ARM_instr.config* file. The compiler will not generate a *main* C function and the *./myScripts* folder includes any MATLAB file may be used during compilation.

8.2 Compilation Options Used in The Experiments

This section discusses the compilation options that have been used by the C compilers at various experiments in chapters 6 and 8. Tables 42, 43 and 44 present the compilation options used by the Clang/LLVM, GCC and the compiler of Visual studio (named MSVC) respectively.

The first lines (rows 1-4) in tables 42 and 43 show the options related to all experiments with Clang/LLVM and GCC for the generation of optimized executable code. The option of *'x86 target'* category concerns the compilation of C code on x86 target architectures while the options in *'ARM target'* category have been used for compilation on ARM architectures (different option values have been used for Raspberry PI 2 and 3). The *fno-vectorize'* and *'-fno-tree-vectorize'* have been used in the experiment of auto-vectorizing C compilers evaluation in subsection 6.10 in order to disable the auto-vectorization applied by the C compilers. The *'-Rpass=loop-vectorize'* and *'-fopt-info-vec'* options have been also used by the Clang/LLVM and GCC in the experiment of subsection 6.10 to report the successfully auto-vectorized loops by the specific C compilers. Finally, the rest of the options of the *'Extra experiment options'* category in table 42 are Clang/LLVM options and have been used in the experiment concerning the Clang/LLVM aggressive auto-vectorization options in subsection 6.9.

	Option	description
Optimization options	-O3	optimization level
	-ffast-math	aggressive floating-point math optimization
Other options	-lm	linking with math library
	-lstdc++	supporting C++
x86 target	-march=corei7-avx	target architecture
ARM target {PI2,PI3}	-target {armv7–linux-gnueabihf, armv8–linux-gnueabihf}	target architecture
	-mcpu={cortex-a7, cortex-a53}	target processor
	-mfpu={neon-vfpv4, neon-fp-armv8}	controlling the FP unit available
	-mfloat-abi=hard	controlling which registers to use for floating-point
Extra experiment options	-fno-vectorize	disable auto-vectorization
	-mllvm -force-vector-width=4/8	force auto-vectorization with SIMD 4/8
	-fslp-vectorize-aggressive	enable aggresive superword-level parallelism
	-fno-slp-vectorize	disable superword-level parallelism
	-Rpass=loop-vectorize	reports the auto-vectorized loops

Table 42: Compilation options of Clang/LLVM

Table 44 shows the compilation option that has been used by the MSVC compiler for the generation of executable code on the various architectures. The specification of the target architecture is configured in Visual Studio and no compilation options have been used. The *'/arch'* option with *AVX* (or *SSE*) value is used for the auto-vectorization of the C code. The *'No Set'* value of *'/arch'* option disables the auto-vectorization and has been used in the experiment of auto-vectorizing C compilers evaluation in subsection 6.10. Finally, the *'/Qvec-report:1'* reports the successfully auto-vectorized loops and has been also used in the examination of MSVC at auto-vectorization in subsection 6.10.

	Option	description
Optimization options	-O3	optimization level
	-ffast-math	aggressive floating-point math optimization
Other options	-lm	linking with math library
	-lstdc++	supporting C++
x86 target	-march=corei7-avx	target architecture
ARM target {PI2,PI3}	-march {armv7ve, armv8-a+crc}	target architecture
	-mcpu={cortex-a7, cortex-a53}	target processor
	-mfpu={neon-vfpv4, neon-fp-armv8}	controlling the FP unit available
	-mfloat-abi=hard	controlling which registers to use for floating-point
Extra experiment options	-fno-tree-vectorize	disable auto-vectorization
	-fopt-info-vec	reports the auto-vectorized loops

Table 43: Compilation options of GCC

	Option	description
General options	/sdl-	disable the Security Development Lifecycle checks
Optimization options	/O2	Maximize speed
	/Ob2	enable inline functions expansion
	/Oi	enable intrinsic functions generation
Code Generation	/GS-	disable Security checks
	/Gy	enable function-level linking
	/Qpar	enable Parallel code generation
	/arch:{AVX,SSE,No Set}	enable/disable auto-vectorization
	/fp:fast	set fast floating point model
Extra experiment options	/Qvec-report:1	reports the auto-vectorized loops

Table 44: Compilation options of MSVC

8.3 XML description examples of the target architectures

The section present the XML specification for the description of the parametrized processor model for the target architectures. The examples below show only a code snippet of the XML files.

8.3.1 XML description of Bot Processor

The BoT processor provides a variety of SIMD instructions such as trigonometric functions and complex number operations which operate on vectors of complex data type with SIMD width of 4 or 8. Most of the available instructions accept fixed point numbers with any fraction length while some trigonometric functions require (and produce) fixed point numbers with specific fraction length.

Listing 8.1 shows a snippet of the parametrized processor model which has been used for the description of the BoT architecture. Firstly, the *common.h* which includes the semantics of BoT processor is declared as additional header C file. Subsequently, various types are specified (lines 4-7) which are used for the description of the customized instruction and other special operations. The XML tag in line 9 defines the native vector data type of BoT architecture which is used by the compiler's code generator and XML tags of lines 11 and 13 specify shift and reciprocal operations which are used in the code generation of fixed point operations. In the main part of parametrized processor model vector instructions of addition, multiplication and division are described. The BoT processor doesn't provide any instruction for the division of vectors. The operation is performed multiplying the divider with the reciprocal of the divisor, therefore *VMUL* instruction (implements vector multiplication) is specified for the './' operator. The description of customized instructions in lines 22 and 24 concern the specification of the *cos* and *abs* MATLAB functions. BoT implements that functions for specific fraction lengths, thus the operand types of the described instructions include specific fraction lengths (15 and 12). The result type of *cos* function is fully inherited by the operand's type and the *res_type* attribute can be skipped. However, the result type of a *abs* function doesn't completely related with the type of operand. The absolute value of a complex value is a non-complex number. Consequently, the result type attribute of *CABS* instruction is set with the *fxpR* type denoting that the result will inherit partly the type of operand (dimensions, word and fraction length) but the result type is always set statically to non-complex fixed point type.

```
<!--BoT processor description-->
<header_files common="common.h"></header_files>
<!--types-->
<type id="fxp" dt="fixp(any,any)"></type>
<type id="fxpR" dt="fixp(any,any)" complex="false"></type>
<type id="fxp_15" dt="fixp(any,15)" ></type>
<type id="fxpC_12" dt="fixp(any,12)" complex="true"></type>
<!--Derived types for code generation-->
<derived_type name="vcomplex_t" type="fxp" SIMD_width="4" ></derived_type>
<!--Shift operations for fixed point arithmetic-->
<shift leftShift="LSL" rightShift="ASR" type="SIMD" SIMD_width ="true" op_types="fxp"></shift>
<!--Reciprocal intrinsic.  Used in fixed point division-->
<reciprocal recip="V_RECIP" type="SIMD" SIMD_width ="true" op_types="fxp"></reciprocal>
<!--Customized operations-->
<instruction name="ADD" type="SIMD" op="+" pack="p,p" SIMD_width ="true" op_types="fxp,fxp">
</instruction>
<instruction name="VMUL" type="SIMD" op="*" pack="p,p" SIMD_width ="true" op_types="fxp,fxp">
</instruction>
<instruction name="VMUL" type="SIMD" op="./" pack="p,p" SIMD_width ="true" op_types="fxp,fxp">
</instruction>
<!--Customized operations requiring specific fraction length-->
<instruction name="COS" type="SIMD" func="cos" pack="p,p" SIMD_width ="true" op_types="fxp_15">
</instruction>
<instruction name="CABS" type="SIMD" func="abs" res_type="fxpR" pack="p,p" SIMD_width="true"
     op_types="fxpC_12">
</instruction>
```

Listing 8.1: Snippet of parametrized processor model for the BoT architecture

8.3.2 XML description of tinyBot Processor

The tinyBoT processor is derived from the BoT architecture including scalar instructions of complex arithmetic and trigonometric operations. Listing 8.2 shows a snippet of the parametrized processor model for the tinyBoT processor. Similarly to the BoT description of listing 8.1, in the parametrized processor model of tinyBoT the *common.h* is declared as additional header file and complex and non-complex fixed point types are defined as well. Then, specification of derived types as well as scalar shift and reciprocal operations are following. Finally, multiplication and conjugation (transpose operator) scalar instructions are described. The *i_CMUL* instruction performs multiplication of complex numbers. Thus, in the description of the instruction is required at least one operand to be complex. The instruction could be used for multiplication of non-complex numbers as well. However, in that case the use of the instruction can be avoided and a simple multiplication statement of two integers (and shifting operations) may be produced.

```
<!--TinyBot processor description-->
<header_files common="common.h"></header_files>
<!--types-->
<type id="fxp" dt="fixp"></type>
<type id="fxpC" dt="fixp" complex="true"></type>
<!--Derived types for code generation-->
<derived_type name="ntype_t" type="fxpC"></derived_type>
<!--Shift operations for fixed point arithmetic-->
<shift leftShift="i_CLSL" rightShift="i_CASR" type="SCALAR" op_types="fxpC"></shift>
<!--Reciprocal intrinsic.  Used in fixed point division-->
<reciprocal recip="RECIP" type="SCALAR" op_types="fxpC"></reciprocal>
<!--Customized operations-->
<instruction name="i_CMUL" type="SCALAR" op="*" op_types="(fxp|fxpC),fxpC"></instruction>
<instruction name="i_CMUL" type="SCALAR" op="*" op_types="fxpC,(fxp|fxpC)"></instruction>
<instruction name="i_CONJ" type="SCALAR" op="'"  op_types="fxpC"></instruction>
```

Listing 8.2: Snippet of parametrized processor model for the tinyBoT architecture

8.3.3 XML description of ARM and x86 Architectures

ARM and x86 architectures support SIMD processing through the use of SSE (or AVX) and NEON SIMD extensions. The extensions provide a conventional SIMD instruction set with no support of complex arithmetic operations or trigonometric functionalities. Thus, it can't always be a direct MATLAB to C translation exploiting the available architectures capabilities. To surpass that, several MATLAB functions and MATLAB operations of complex arithmetic has been developed in C using the SIMD extensions of ARM and x86 architectures. The implemented functions are inserted in the parametrized processor model and they are handled by the compiler as hardware intrinsics. Furthermore, wrapping functions have been implemented in C (and have been defined in the processor model) including non-complex SIMD operations and scalar operations of complex arithmetic. Although, the SIMD instructions could be directly described in the parametrized processor model and the scalar operations of complex numbers could be generated without the generation of any customized instruction, the wrapping in functions have been used for two reasons: a) the generated code can be applied both on ARM and x86 architecture developing different implementations of wrapping functions, b) instruction selection mechanism is extensively evaluated and the applicability of the tool is widely demonstrated.

Listing 8.3 presents a snippet of the parametrized processor model which have been used for the code generation on ARM and x86 architectures. The two architectures share the same parametrized processor model. Their vector semantics and SIMD intrinsics have been encapsulated in the developed C code which is included in the *common.h* and *types.h* header files (different version for each architecture). Thus, the generated

code can be executed at any ARM or x86 processor including in the compilation the appropriate header files.

At the snippet of parametrized processor model for the description of ARM and x86 architectures shown at listing 8.3, the complex floating point and complex fixed point types are specified. Then, derived C data types are defined regarding unpacked variables and packed variables of different SIMD widths. Finally, the XML tags of lines 14-24 describe instructions of the MATLAB add operation with complex numbers concerning scalar and SIMD processing of floating and fixed point types.

```
1  <!--Description of general purpose architecture-->
2  <header_files common="common.h" types="types.h"></header_files>
3  <!--types-->
4  <type id="fc_t" dt="double" complex="true"></type>
5  <type id="ic_t" dt="fixp" complex="true"></type>
6  <!--Derived types for code generation-->
7  <derived_type name="cfloat_t" type="fc_t" ></derived_type>
8  <derived_type name="cfloat32x4_t" type="fc_t" SIMD_width="4"></derived_type>
9  <derived_type name="cfloat32x8_t" type="fc_t" SIMD_width="8"></derived_type>
10 <derived_type name="cfixp_t" type="ic_t" ></derived_type>
11 <derived_type name="cfixp32x4_t" type="ic_t" SIMD_width="4"></derived_type>
12 <derived_type name="cfixp32x8_t" type="ic_t" SIMD_width="8"></derived_type>
13 <!--Customized floating point operations-->
14 <instruction name="cadd_f" type="SCALAR" op="+" op_types="fc_t,fc_t"></instruction>
15 <instruction name="v4_add_f" type="SIMD" op="+" pack="p,p" SIMD_width ="4" op_types="fc_t,fc_t">
16 </instruction>
17 <instruction name="vadd_f" type="SIMD" op="+" pack="p,p" SIMD_width ="8" op_types="fc_t,fc_t">
18 </instruction>
19 <!--Customized fixed point operations-->
20 <instruction name="cadd_i" type="SCALAR" op="+" op_types="ic_t,ic_t"></instruction>
21 <instruction name="v4_add_i" type="SIMD" op="+" pack="p,p" SIMD_width ="4" op_types="ic_t,ic_t">
22 </instruction>
23 <instruction name="vadd_i" type="SIMD" op="+" pack="p,p" SIMD_width ="8" op_types="ic_t,ic_t">
24 </instruction>
```

Listing 8.3: Snippet of parametrized processor model for the ARM and x86 architectures

8.4 Comprehensive Results Compiling with Aggressive Clang Optimization Options

This section presents the performance results of MathWorks generated code and compiler's scalarized generated code using aggressive optimization options of Clang. The optimizations concern the auto-vectorization and superword-level parallelism optimization on scalarized code with fixed point and floating point data types. For the experiments, the Raspberry PI 2 board and the desktop with i7-3770 processor has been deployed. The results complement those presented at section 6.9.

	Fig. 134, Fig. 135	**Fig. 136, Fig. 137**	**Fig. 138, Fig. 139**	**Fig. 140, Fig. 141**	**Fig. 142, Fig. 143**	**Fig. 144, Fig. 145**	**Fig. 146, Fig. 147**	**Fig. 148, Fig. 149**
fft32	15511.10	206.06	158.24	180.96	97138.83	2856.53	4097.93	2093.67
fft64-v1	4509.88	1011.16	1311.64	893.18	45648.50	14536.60	15641.83	11083.93
fft64-v2	14964.04	1029.02	643.36	1095.54	214133.03	13428.73	11191.77	11673.87
fft128	28139.46	2425.48	1995.08	2350.02	434562.00	33774.37	30419.47	25206.10
cfo-32	1073.54	766.08	1697.68	698.78	9264.93	8708.50	38467.57	18837.07
cfo-64	7782.40	1606.52	3686.50	6784.72	66805.30	17760.83	77349.33	38414.20
cfo-128	13726.14	2811.22	6604.64	12416.92	131323.13	36197.80	152719.80	82792.90
cordic-64	69999.22	10658.70	22038.66	8281.10	1207044.07	338458.00	719763.13	274729.33
fir-32x64	12952.80	5610.12	10403.78	7996.44	178774.67	123754.67	468978.93	207613.43
fir-32x128	23012.98	11356.58	22995.04	16334.70	402998.60	307584.17	1035533.17	421980.23
fir-32x256	41247.96	21871.46	38984.64	28365.70	810921.97	647600.73	2176108.23	964808.63
mean-32	67.88	37.95	36.19	25.46	698.13	622.70	563.50	592.43
mean-64	127.72	67.98	62.34	87.78	1364.57	1282.13	1041.13	1168.00
mean-128	381.40	130.35	256.68	169.96	2659.83	2475.37	2006.77	2281.20
mean-1024	2492.76	2436.86	1558.20	1795.46	21071.00	19405.57	16208.10	17735.03
qr-dec-32	181549.56	15240.52	13010.62	31638.96	3192630.97	272726.93	395732.97	188503.20
qr-dec-64	56056.00	28860.94	27635.04	61933.68	1433642.37	533290.40	792714.20	372267.57
qr-dec-128	102229.48	53411.56	47562.36	111648.84	2830109.40	1053708.07	1601364.40	744831.07

Table 45: Reference values (exec. time in µs) used for normalization of aggressive Clang options examination

8.4.1 Results Compiling with Aggressive Clang Optimization Options on Raspberry PI 2

8.4.2 Results Compiling with Aggressive Clang Optimization Options on i7-3770

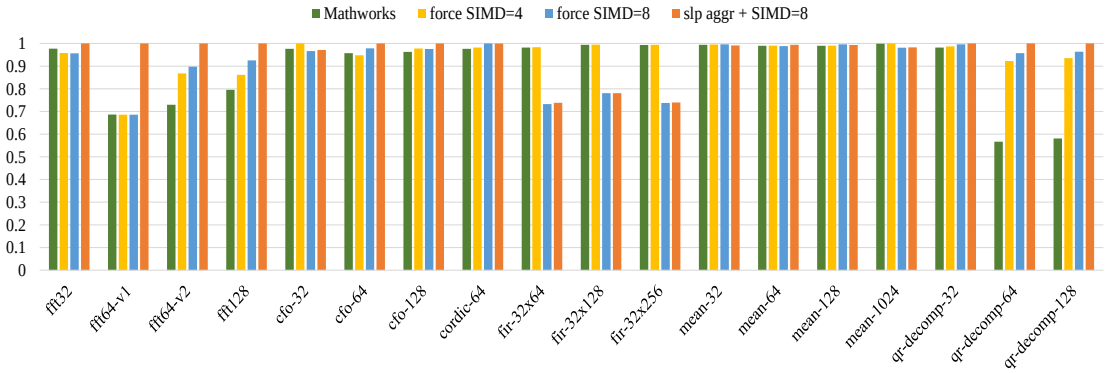

Figure 134: Normalized execution times of MathWorks fixed point generated code on PI 2, compiled with aggressive Clang auto-vectorization options

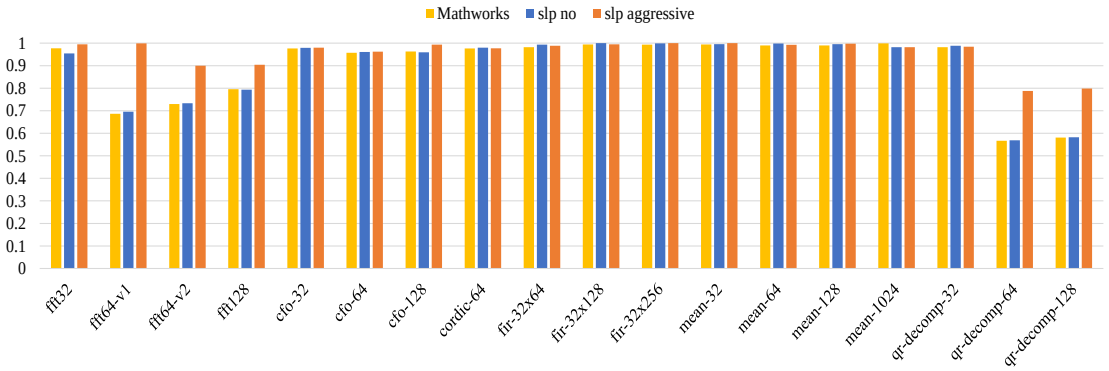

Figure 135: Normalized execution times of MathWorks fixed point generated code on PI 2, compiled with aggressive superword-level parallelism Clang options

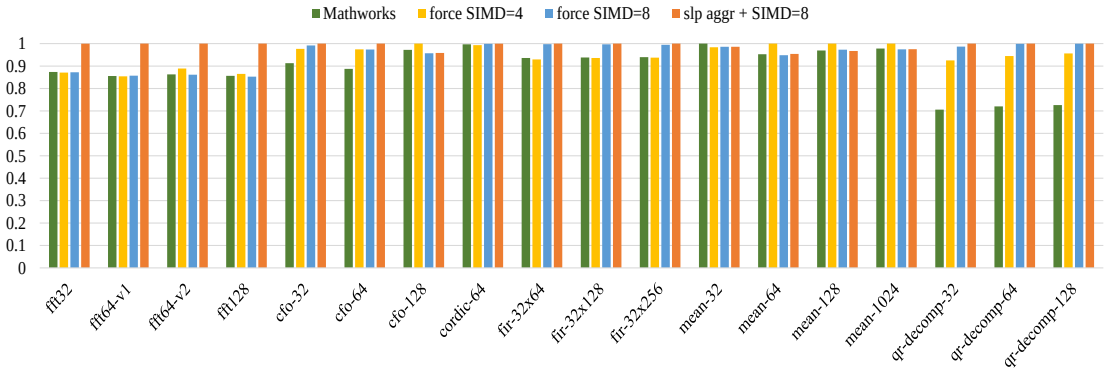

Figure 136: Normalized execution times of compiler's scalarized fixed point generated code on PI 2, compiled with aggressive Clang auto-vectorization options

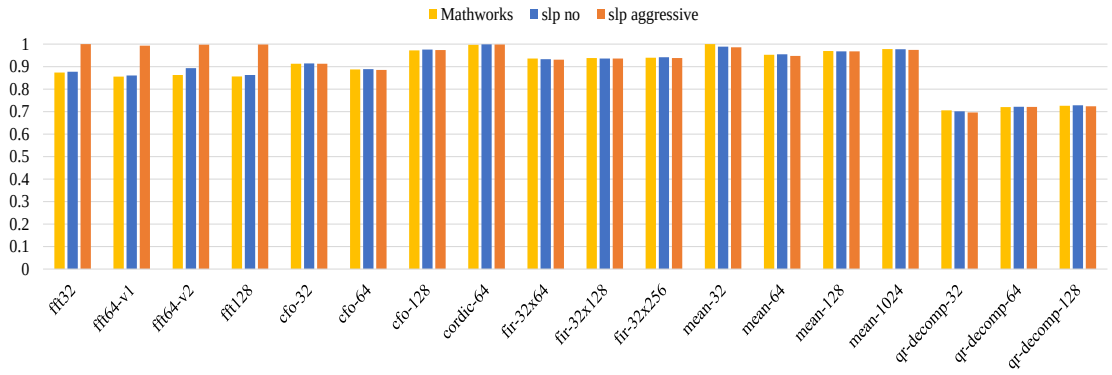

Figure 137: Normalized execution times of compiler's scalarized fixed point generated code on PI 2, compiled with aggressive superword-level parallelism Clang options

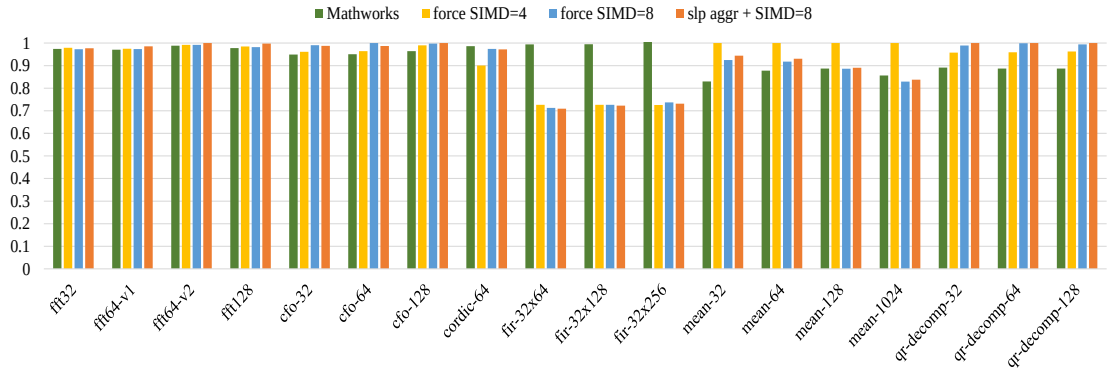

Figure 138: Normalized execution times of MathWorks floating point generated code on PI 2, compiled with aggressive Clang auto-vectorization options

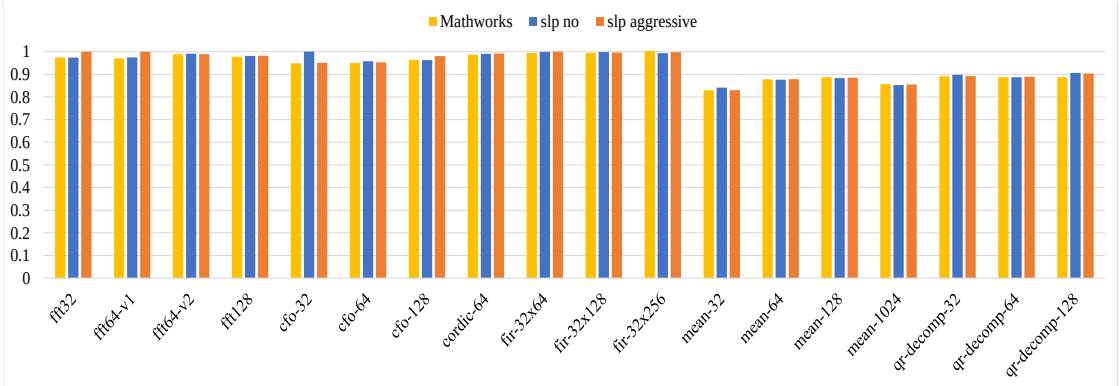

Figure 139: Normalized execution times of MathWorks floating point generated code on PI 2, compiled with aggressive superword-level parallelism Clang options

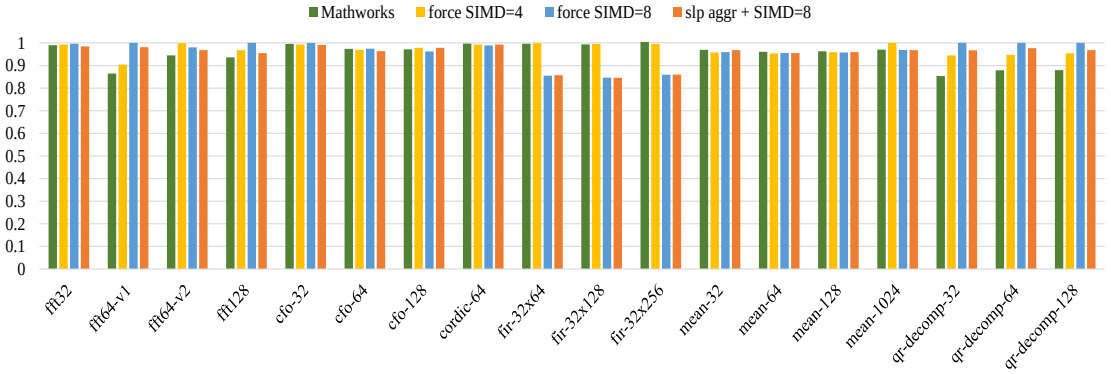

Figure 140: Normalized execution times of compiler's scalarized floating point generated code on PI 2, compiled with aggressive Clang auto-vectorization options

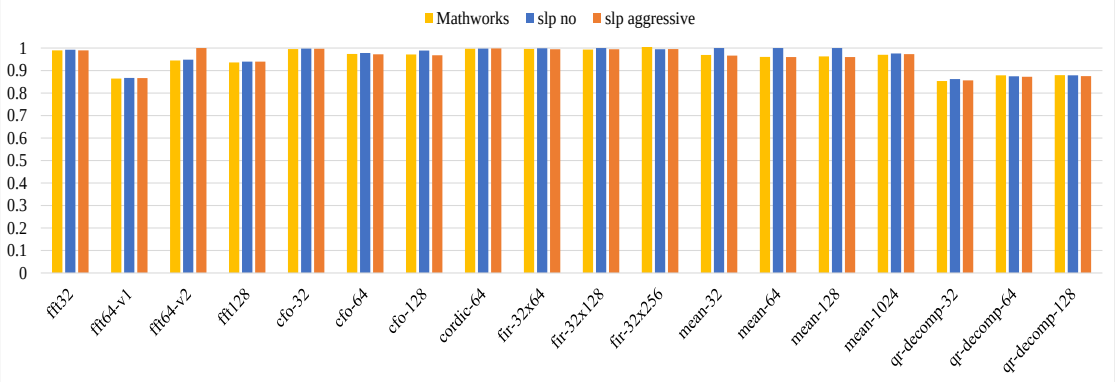

Figure 141: Normalized execution times of compiler's scalarized floating point generated code on PI 2, compiled with aggressive superword-level parallelism Clang options

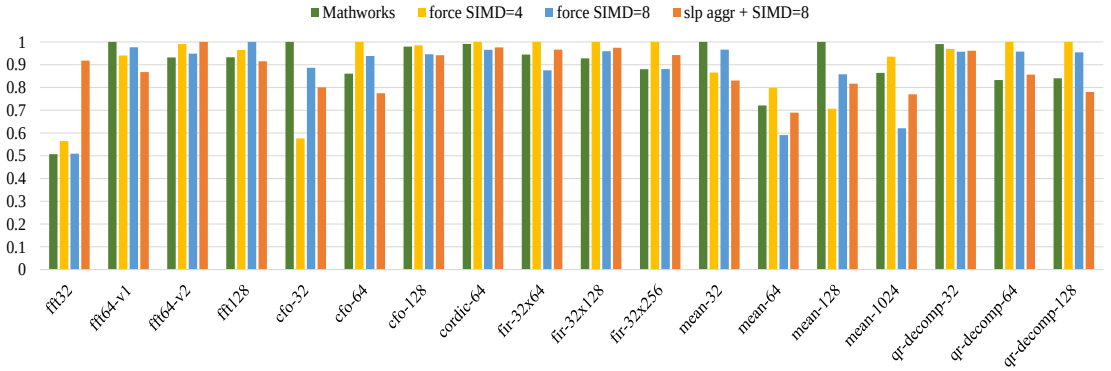

Figure 142: Normalized execution times of MathWorks fixed point generated code on i7-3770, compiled with aggressive Clang auto-vectorization options

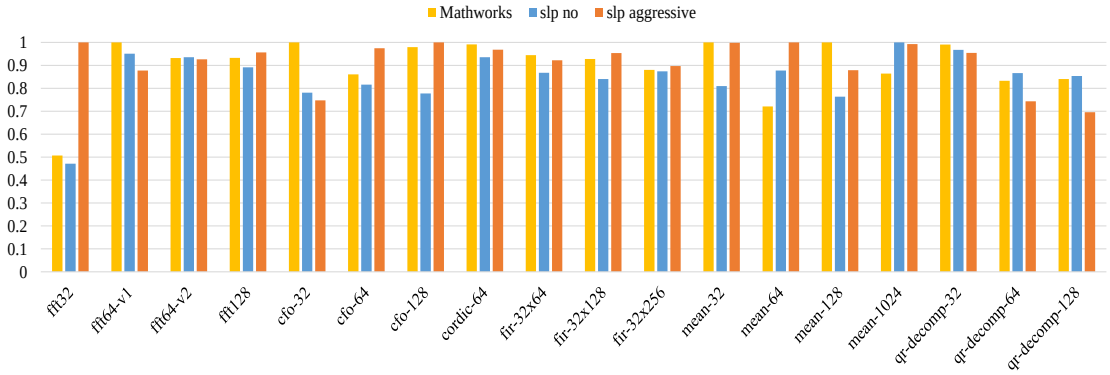

Figure 143: Normalized execution times of MathWorks fixed point generated code on i7-3770, compiled with aggressive superword-level parallelism Clang options

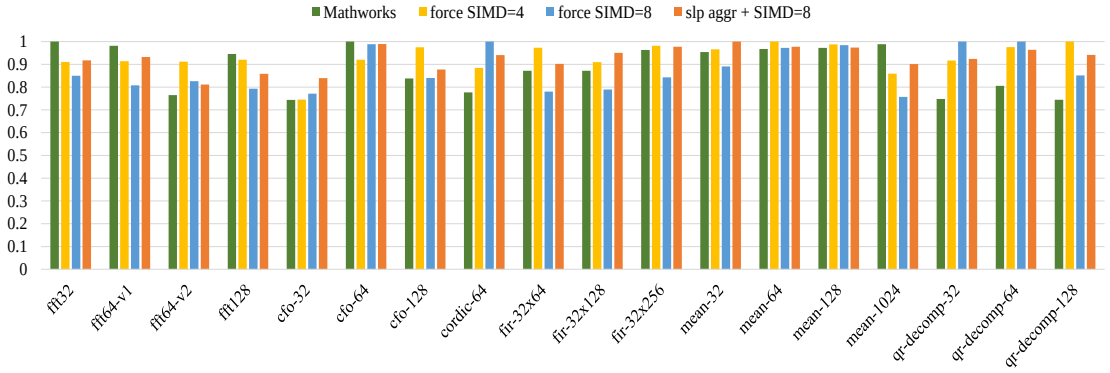

Figure 144: Normalized execution times of compiler's scalarized fixed point generated code on i7-3770, compiled with aggressive Clang auto-vectorization options

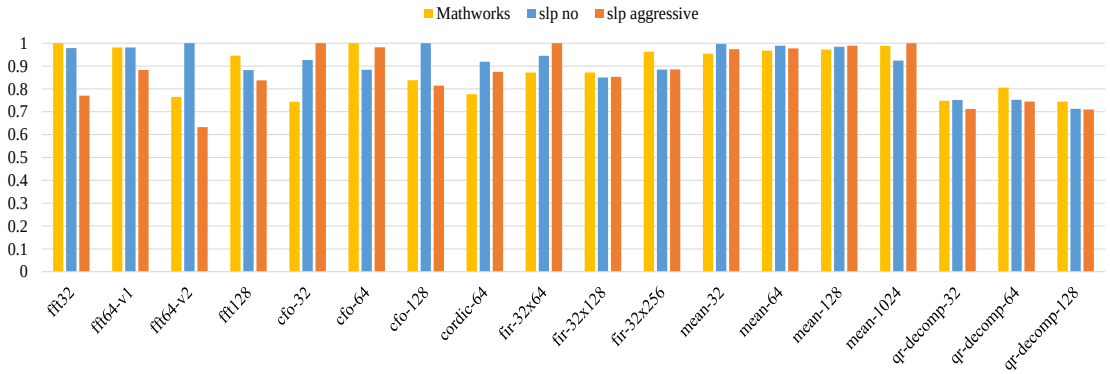

Figure 145: Normalized execution times of compiler's scalarized fixed point generated code on i7-3770, compiled with aggressive superword-level parallelism Clang options

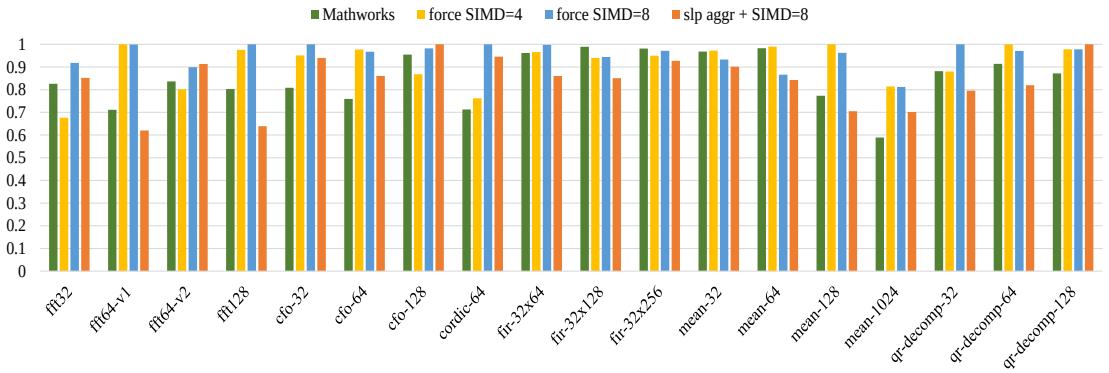

Figure 146: Normalized execution times of MathWorks floating point generated code on i7-3770, compiled with aggressive Clang auto-vectorization options

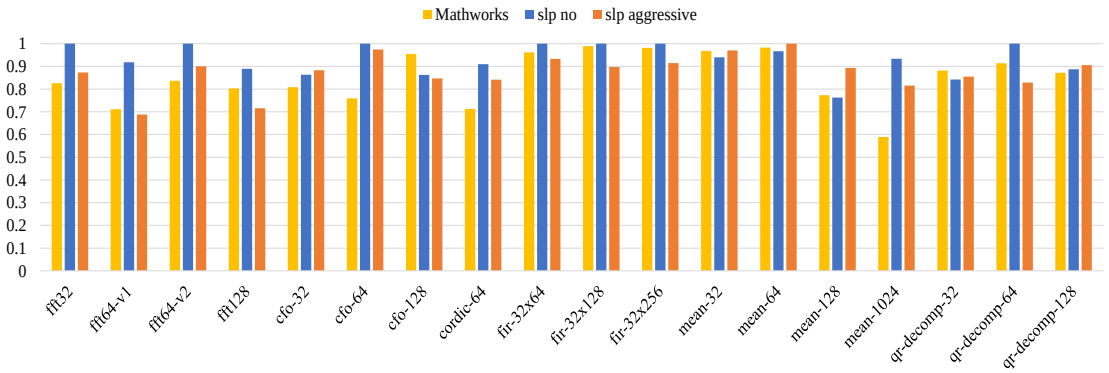

Figure 147: Normalized execution times of MathWorks floating point generated code on i7-3770, compiled with aggressive superword-level parallelism Clang options

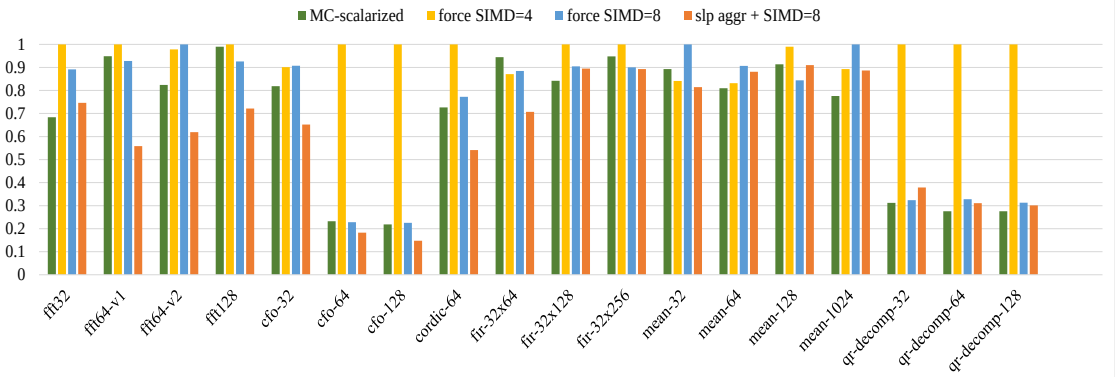

Figure 148: Normalized execution times of compiler's scalarized floating point generated code on i7-3770, compiled with aggressive Clang auto-vectorization options

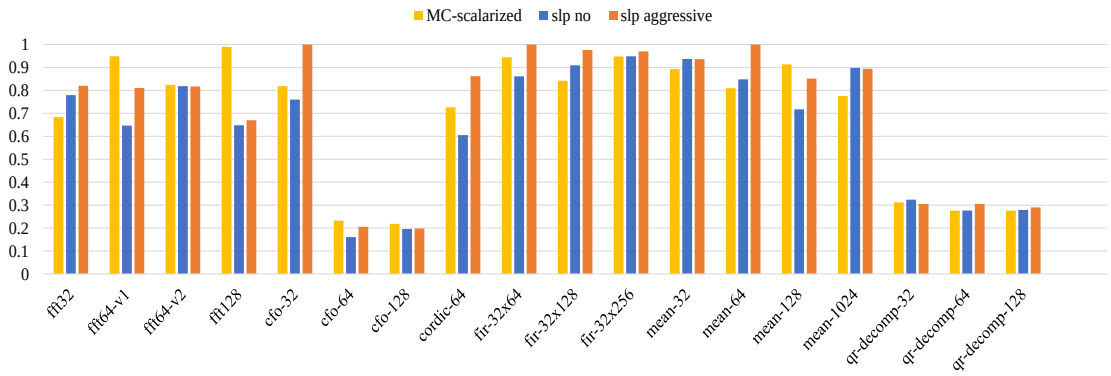

Figure 149: Normalized execution times of compiler's scalarized floating point generated code on i7-3770, compiled with aggressive superword-level parallelism Clang options

8.5 Performance of Vectorized Generated Code Against Compiler's Scalarized Generated Code

This section presents the performance of vectorized generated code compared to that of compiler's scalarized generated code. The various results of the section concern the different data types, C compilers and targeted architectures that have been used in the experiments. The Tables following include the reference values used for the normalization of the results.

8.5.1 Performance of Vectorized Code Against Compiler's Scalarized Code on Raspberry PI 2

	Fig. 150, Fig. 151	**Fig. 152, Fig. 153**	**Fig. 154, Fig. 155**	**Fig. 156, Fig. 157**	**Fig. 158, Fig. 159**	**Fig. 160, Fig. 161**
fft32	3.34	2.57	2.31	2.24	2.50	2.70
fft64-v1	15.40	12.66	10.62	9.55	14.50	15.60
fft64-v2	16.94	11.33	12.78	8.10	12.20	13.30
fft128	30.09	26.07	26.22	22.72	32.13	31.77
cfo-32	7.95	18.79	7.41	14.64	6.10	12.23
cfo-64	15.76	38.41	15.00	29.34	12.03	24.23
cfo-128	35.18	80.78	31.90	62.09	24.77	48.17
cordic-64	579.10	511.13	274.82	213.83	486.70	565.30
fir-32x64	170.32	207.25	185.54	168.65	406.73	278.77
fir-32x128	373.03	430.18	409.12	367.39	889.23	619.00
fir-32x256	785.11	964.81	863.30	768.87	1855.93	1293.43
mean-32	1.28	0.94	2.41	1.09	1.83	1.53
mean-64	2.36	1.82	4.62	2.07	3.30	2.90
mean-128	4.53	3.51	9.04	4.06	6.37	6.17
mean-1024	35.61	27.23	70.75	32.02	48.63	46.40
qr-dec-32	192.52	161.09	229.31	164.60	243.20	168.17
qr-dec-64	384.00	325.87	454.52	325.12	498.33	344.10
qr-dec-128	764.73	653.59	912.01	650.94	1006.37	689.30

Table 46: Reference values (exec. time in µs) used for normalization of results on PI 2

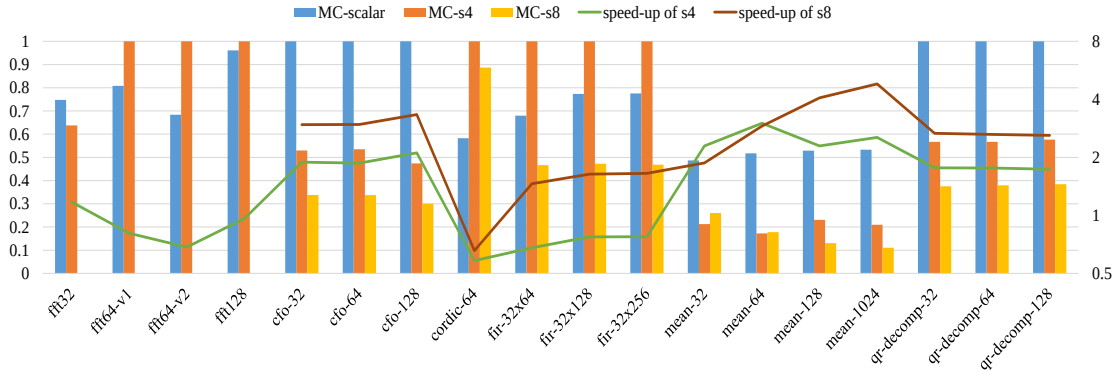

Figure 150: Performance of vectorized code compared to scalarized code using packed fixed point data types, compiled with Clang on Raspberry PI 2

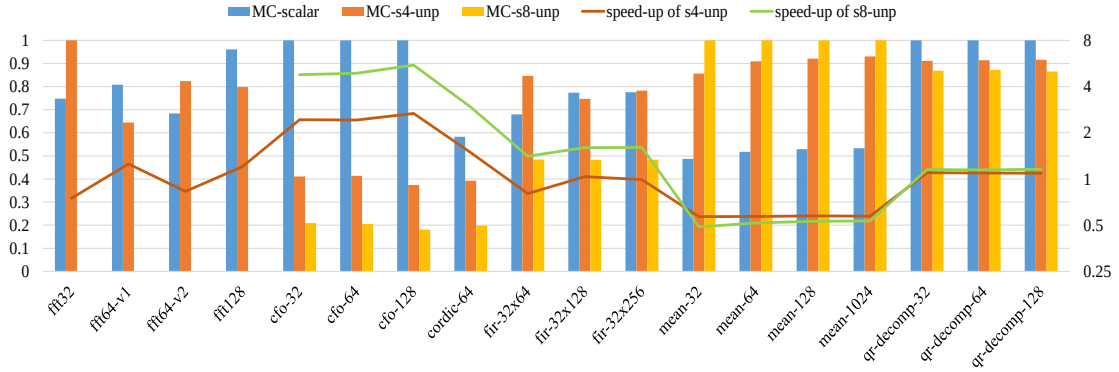

Figure 151: Performance of vectorized code compared to scalarized code using unpacked fixed point data types, compiled with Clang on Raspberry PI 2

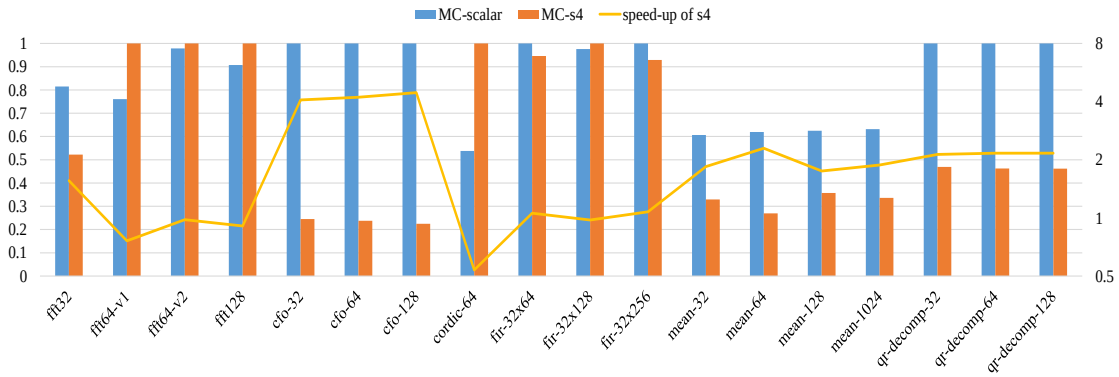

Figure 152: Performance of vectorized code compared to scalarized code using packed floating point data types, compiled with Clang on Raspberry PI 2

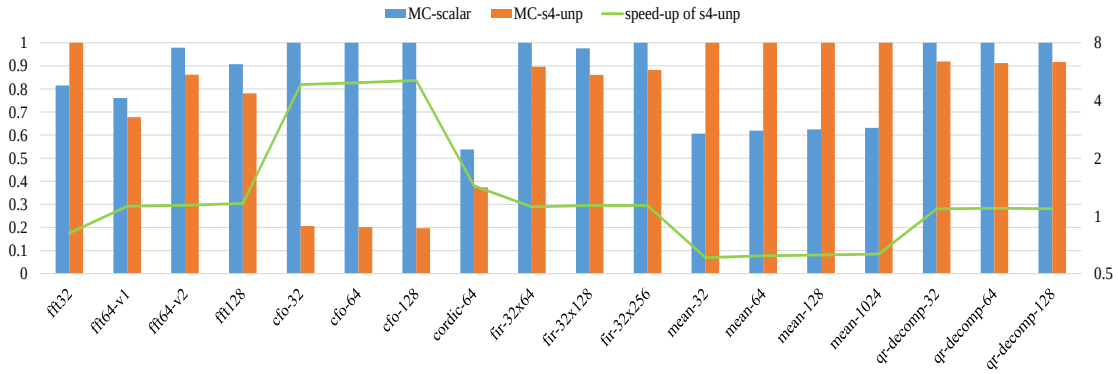

Figure 153: Performance of vectorized code compared to scalarized code using unpacked floating point data types, compiled with Clang on Raspberry PI 2

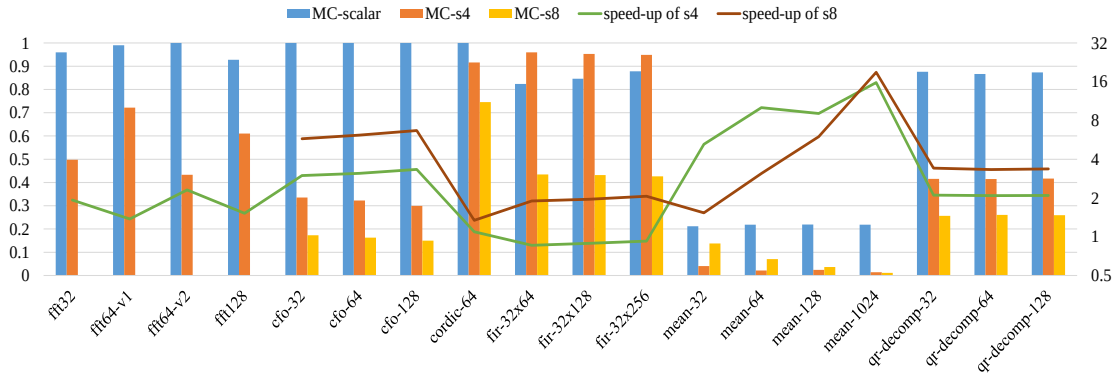

Figure 154: Performance of vectorized code compared to scalarized code using packed fixed point data types, compiled with GCC on Raspberry PI 2

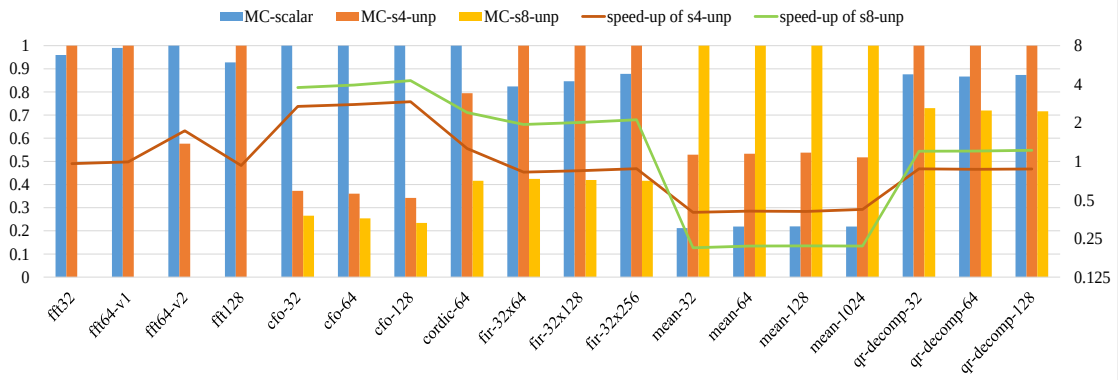

Figure 155: Performance of vectorized code compared to scalarized code using unpacked fixed point data types, compiled with GCC on Raspberry PI 2

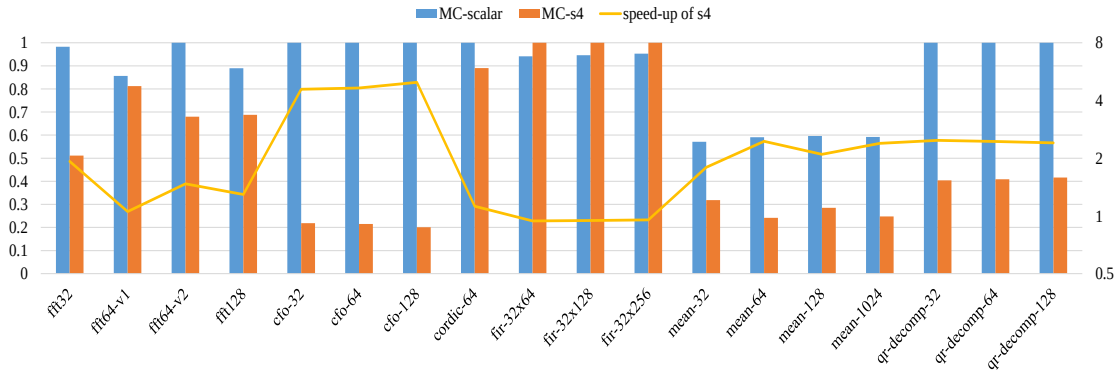

Figure 156: Performance of vectorized code compared to scalarized code using packed floating point data types, compiled with GCC on Raspberry PI 2

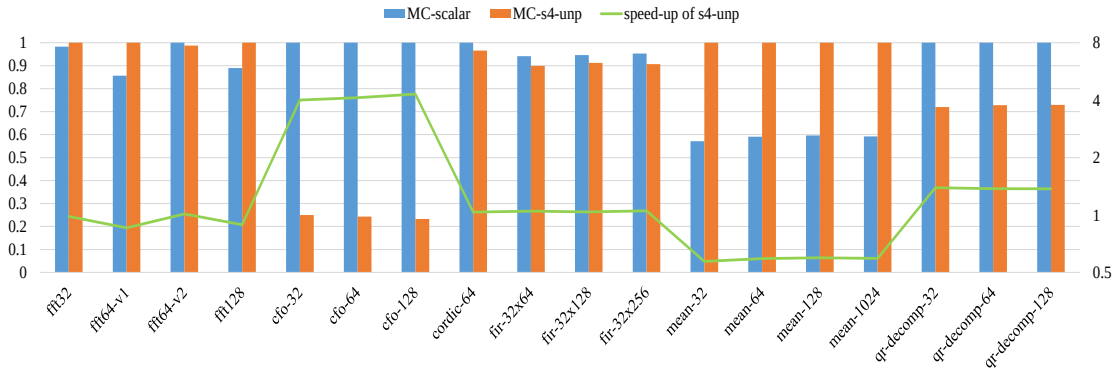

Figure 157: Performance of vectorized code compared to scalarized code using unpacked floating point data types, compiled with GCC on Raspberry PI 2

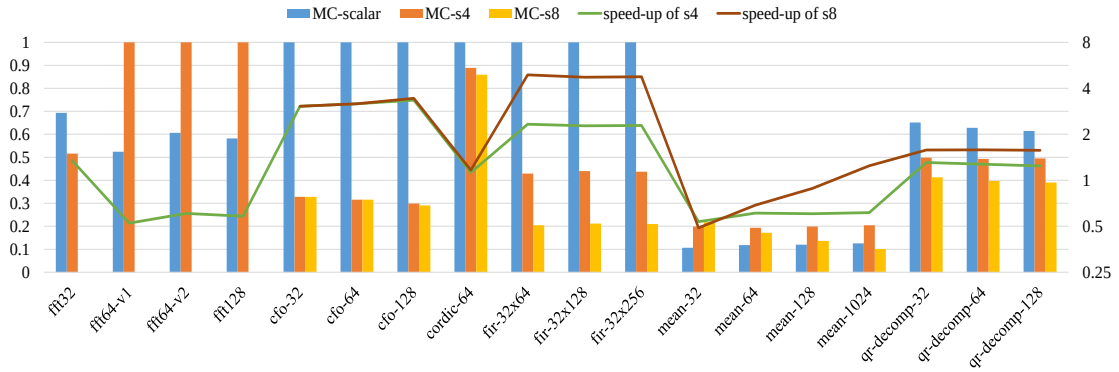

Figure 158: Performance of vectorized code compared to scalarized code using packed fixed point data types, compiled with MSVC on Raspberry PI 2

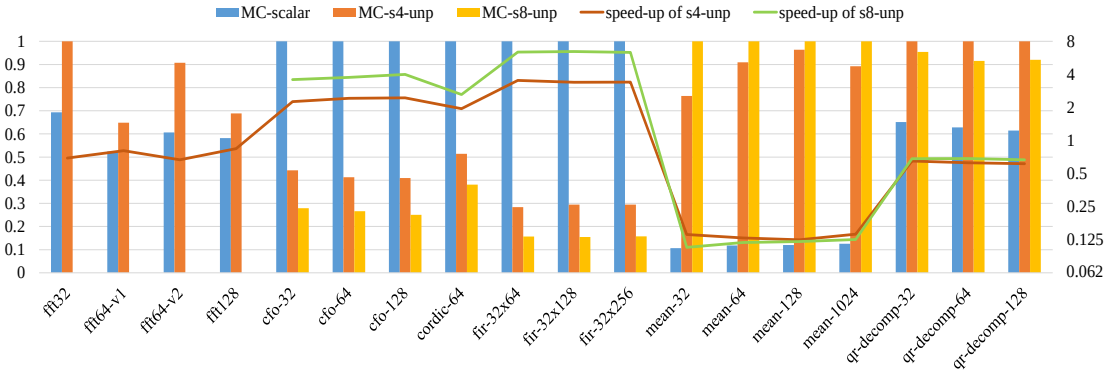

Figure 159: Performance of vectorized code compared to scalarized code using unpacked fixed point data types, compiled with MSVC on Raspberry PI 2

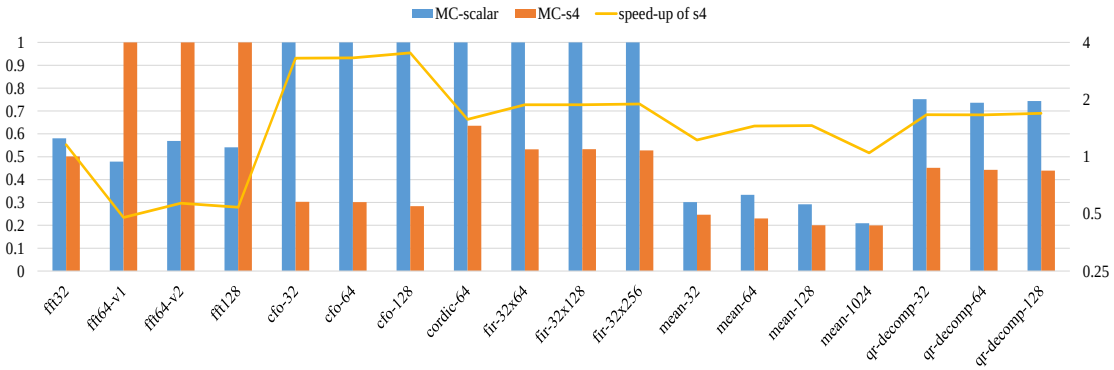

Figure 160: Performance of vectorized code compared to scalarized code using packed floating point data types, compiled with MSVC on Raspberry PI 2

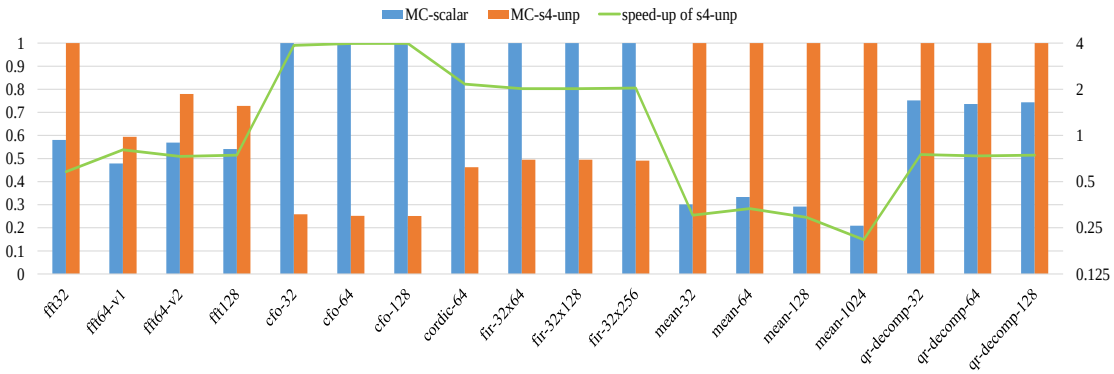

Figure 161: Performance of vectorized code compared to scalarized code using unpacked floating point data types, compiled with MSVC on Raspberry PI 2

8.5.2 Performance of Vectorized Code Against Compiler's Scalarized Code on Raspberry PI 3

	Fig. 162, Fig. 163	Fig. 164, Fig. 165	Fig. 166, Fig. 167	Fig. 168, Fig. 169	Fig. 170, Fig. 171	Fig. 172, Fig. 173
fft32	1.25	1.11	0.86	1.06	1.40	1.50
fft64-v1	5.03	4.69	3.95	3.61	7.47	8.10
fft64-v2	4.98	4.32	4.18	2.81	6.00	6.07
fft128	11.13	10.57	10.08	8.61	16.20	16.03
cfo-32	3.45	7.99	3.21	6.63	3.80	7.43
cfo-64	6.95	15.95	6.44	13.29	7.47	14.57
cfo-128	14.83	34.99	14.02	28.16	14.63	29.47
cordic-64	207.38	184.80	118.74	98.72	266.43	384.00
fir-32x64	99.55	84.97	67.22	74.62	245.53	208.50
fir-32x128	217.63	203.93	150.16	161.81	536.40	457.90
fir-32x256	453.46	388.39	307.68	338.12	1127.87	951.63
mean-32	0.62	0.49	0.90	0.40	1.10	0.90
mean-64	1.14	0.94	1.71	0.75	2.07	1.70
mean-128	2.20	1.83	3.40	1.45	3.90	3.50
mean-1024	16.85	14.80	26.66	12.88	29.13	26.97
qr-decomp-32	86.54	71.89	82.02	71.20	136.87	86.93
qr-decomp-64	171.83	148.39	159.70	141.07	274.93	185.63
qr-decomp-128	344.19	296.95	321.35	284.27	530.70	374.77

Table 47: Reference values (exec. time in µs) used for normalization of results on PI 3

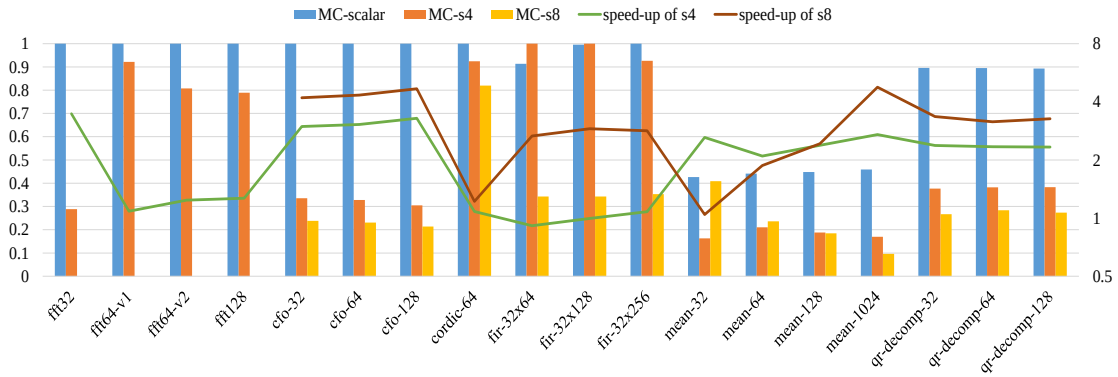

Figure 162: Performance of vectorized code compared to scalarized code using packed fixed point data types, compiled with Clang on Raspberry PI 3

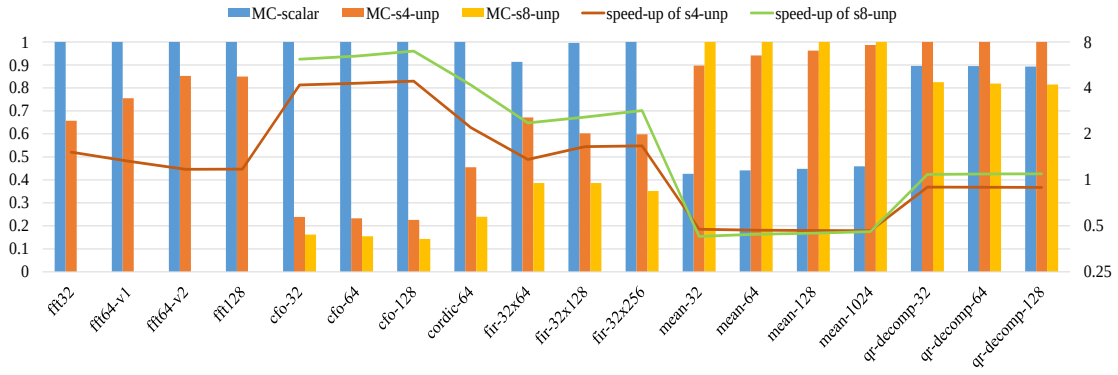

Figure 163: Performance of vectorized code compared to scalarized code using unpacked fixed point data types, compiled with Clang on Raspberry PI 3

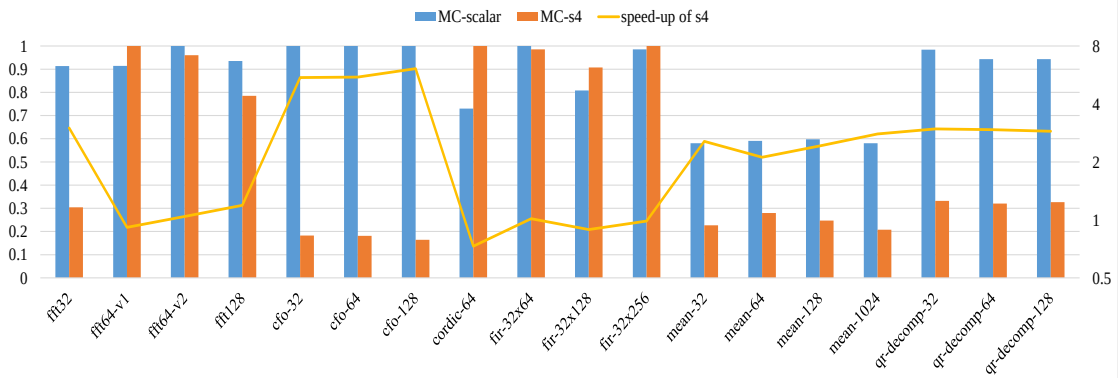

Figure 164: Performance of vectorized code compared to scalarized code using packed floating point data types, compiled with Clang on Raspberry PI 3

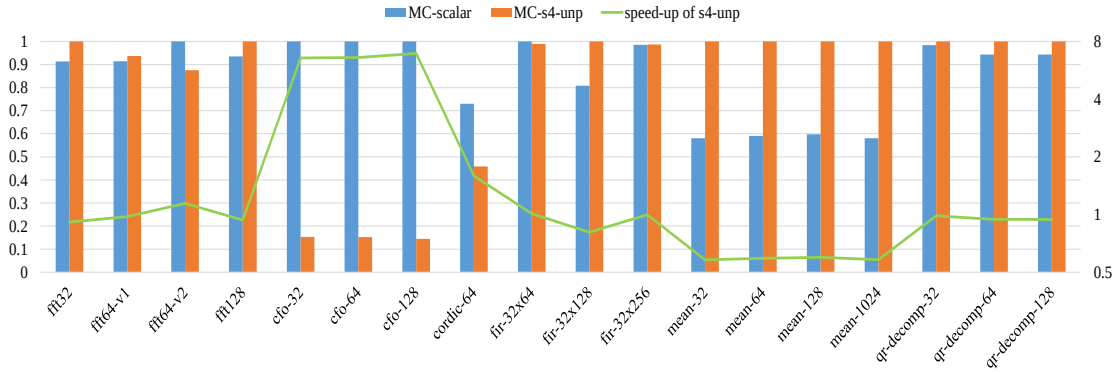

Figure 165: Performance of vectorized code compared to scalarized code using unpacked floating point data types, compiled with Clang on Raspberry PI 3

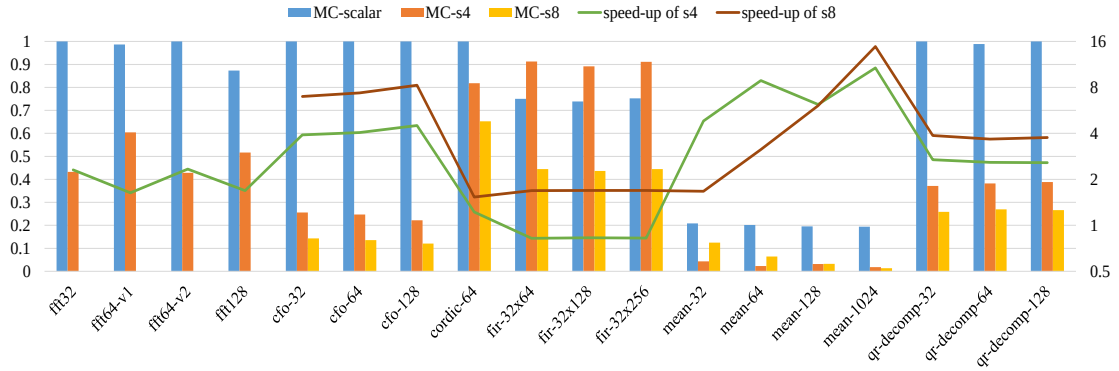

Figure 166: Performance of vectorized code compared to scalarized code using packed fixed point data types, compiled with GCC on Raspberry PI 3

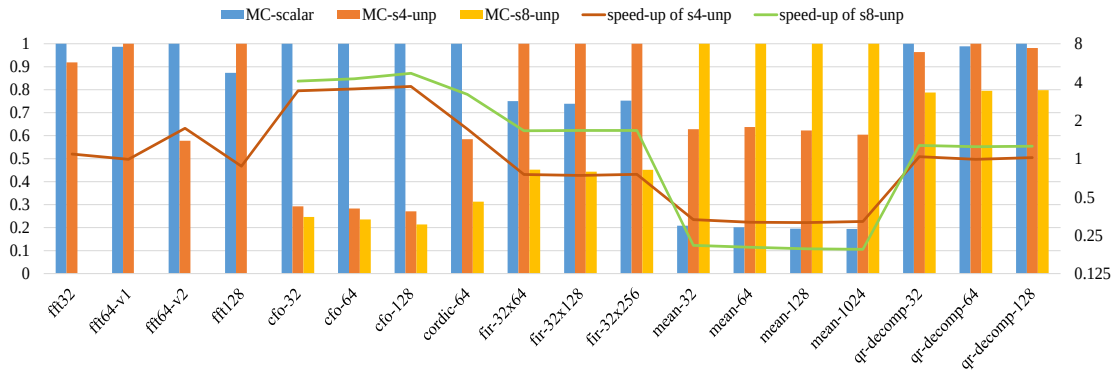

Figure 167: Performance of vectorized code compared to scalarized code using unpacked fixed point data types, compiled with GCC on Raspberry PI 3

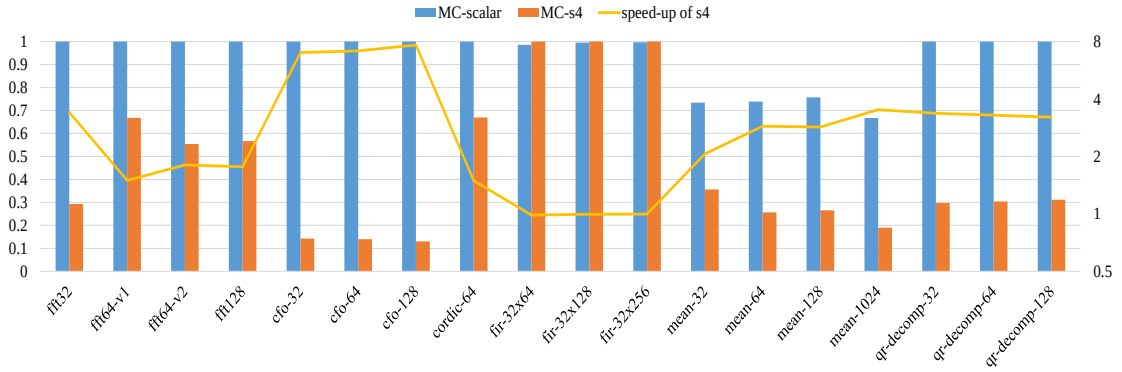

Figure 168: Performance of vectorized code compared to scalarized code using packed floating point data types, compiled with GCC on Raspberry PI 3

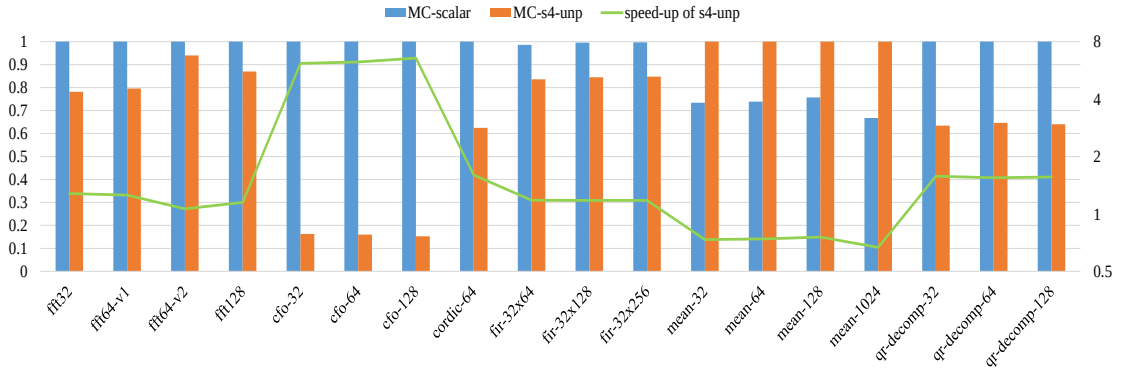

Figure 169: Performance of vectorized code compared to scalarized code using unpacked floating point data types, compiled with GCC on Raspberry PI 3

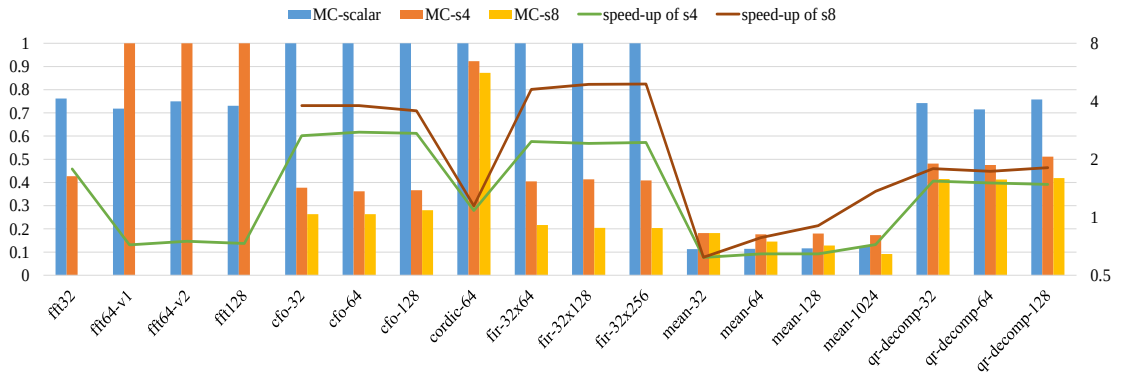

Figure 170: Performance of vectorized code compared to scalarized code using packed fixed point data types, compiled with MSVC on Raspberry PI 3

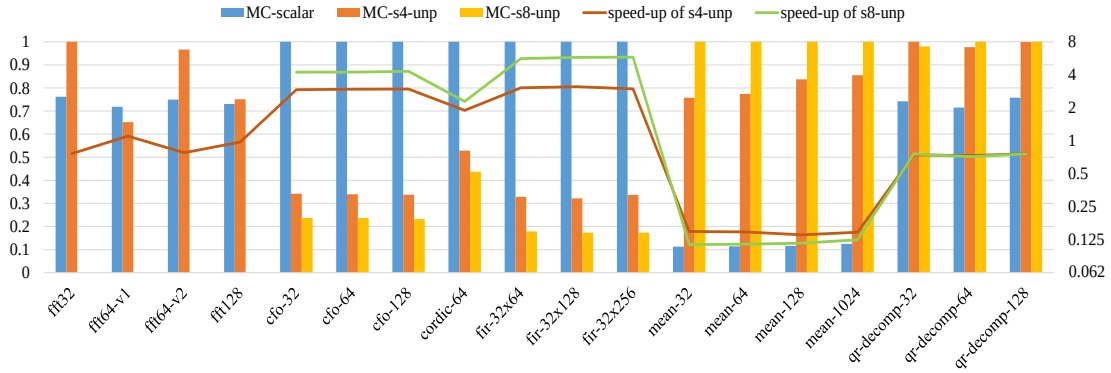

Figure 171: Performance of vectorized code compared to scalarized code using unpacked fixed point data types, compiled with MSVC on Raspberry PI 3

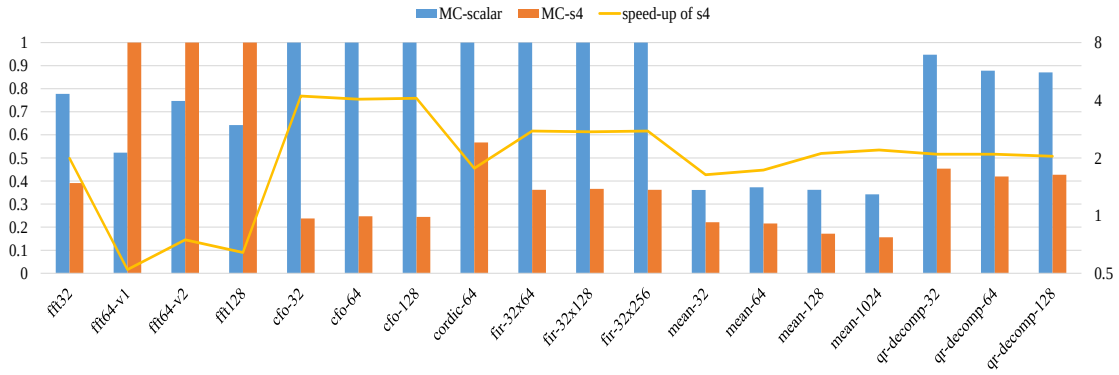

Figure 172: Performance of vectorized code compared to scalarized code using packed floating point data types, compiled with MSVC on Raspberry PI 3

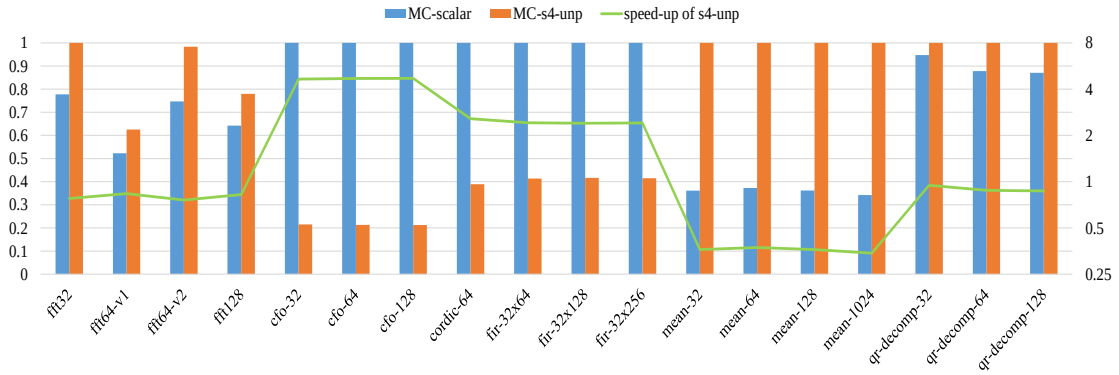

Figure 173: Performance of vectorized code compared to scalarized code using unpacked floating point data types, compiled with MSVC on Raspberry PI 3

8.5.3 Performance of Vectorized Code Against Compiler's Scalarized Code on x86 Desktop with i7-3820 Processor

	Fig. 174, Fig. 175	Fig. 176, Fig. 177	Fig. 178, Fig. 179	Fig. 180, Fig. 181	Fig. 182, Fig. 183	Fig. 184, Fig. 185
fft32	0.22	0.19	0.83	0.92	0.40	0.40
fft64-v1	1.78	1.19	2.39	2.06	1.32	1.60
fft64-v2	2.05	2.12	1.33	2.56	2.10	2.44
fft128	3.28	2.93	4.98	5.46	4.54	4.62
cfo-32	0.51	0.87	0.52	1.04	0.84	1.28
cfo-64	2.41	1.94	1.88	1.84	1.30	1.40
cfo-128	3.28	2.91	2.76	3.23	2.52	3.68
cordic-64	48.64	44.28	12.13	44.17	31.76	44.76
fir-32x64	15.74	15.66	8.93	20.97	22.96	15.64
fir-32x128	32.32	25.63	17.64	35.40	48.94	34.30
fir-32x256	59.88	49.34	28.01	70.25	102.38	71.84
mean-32	0.62	0.67	1.28	1.02	0.30	0.30
mean-64	0.86	0.86	2.26	1.49	0.50	0.50
mean-128	2.13	2.20	3.86	3.65	1.20	1.10
mean-1024	12.15	10.95	19.95	20.11	9.96	9.32
qr-dec-32	38.14	35.65	36.76	40.74	42.04	37.96
qr-dec-64	61.10	59.82	62.09	71.36	84.26	62.10
qr-dec-128	112.70	115.95	117.99	134.57	168.40	123.56

Table 48: Reference values (exec. time in µs) used for normalization of results on desktop with i7-3820 processor

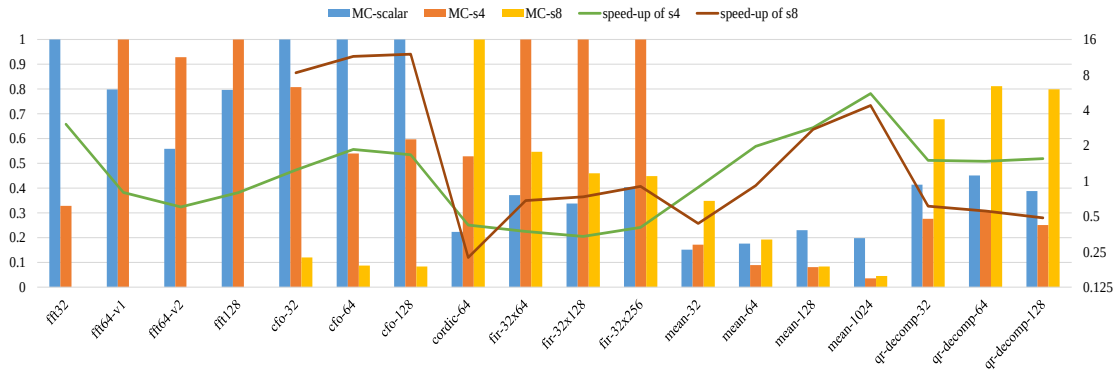

Figure 174: Performance of vectorized code compared to scalarized code using packed fixed point data types, compiled with Clang on desktop with i7-3820 processor

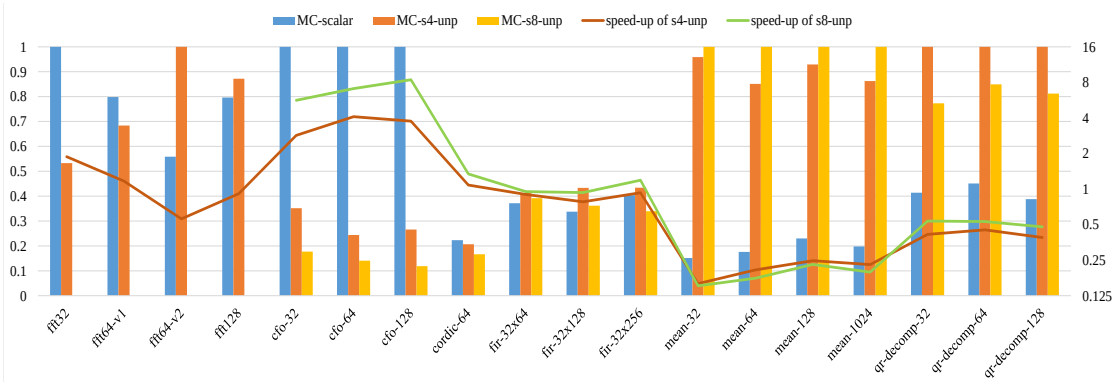

Figure 175: Performance of vectorized code compared to scalarized code using unpacked fixed point data types, compiled with Clang on desktop with i7-3820 processor

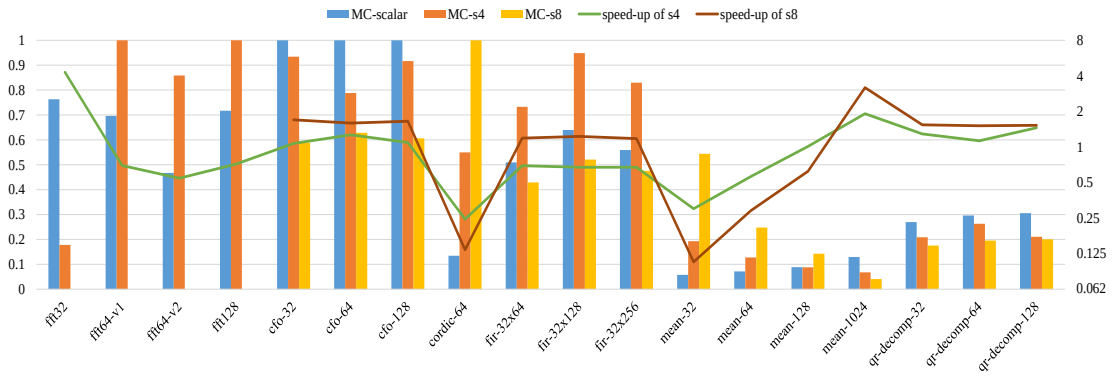

Figure 176: Performance of vectorized code compared to scalarized code using packed floating point data types, compiled with Clang on desktop with i7-3820 processor

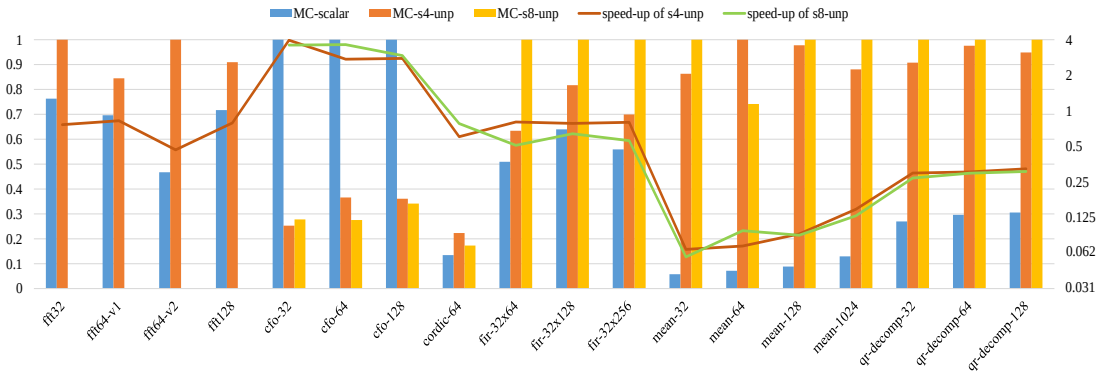

Figure 177: Performance of vectorized code compared to scalarized code using unpacked floating point data types, compiled with Clang on on desktop with i7-3820 processor

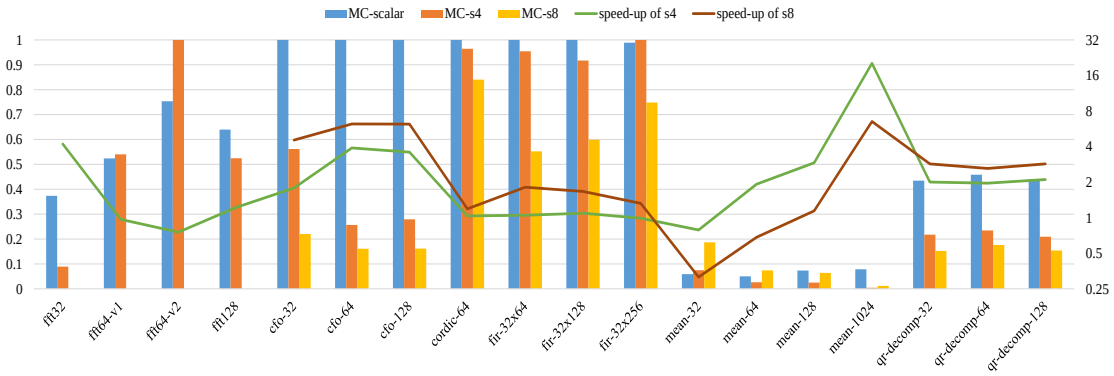

Figure 178: Performance of vectorized code compared to scalarized code using packed fixed point data types, compiled with GCC on on desktop with i7-3820 processor

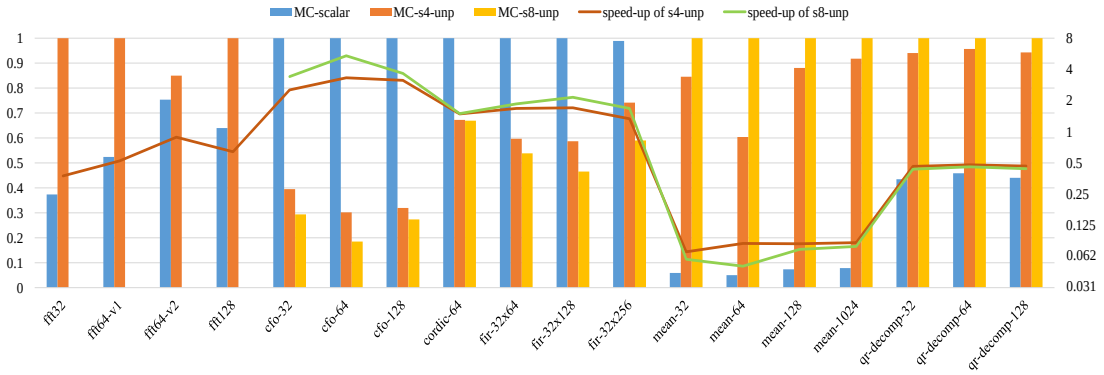

Figure 179: Performance of vectorized code compared to scalarized code using unpacked fixed point data types, compiled with GCC on on desktop with i7-3820 processor

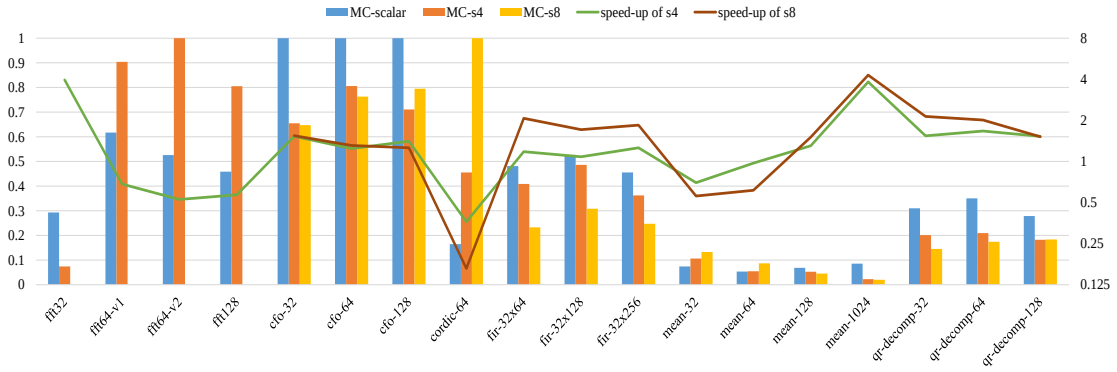

Figure 180: Performance of vectorized code compared to scalarized code using packed floating point data types, compiled with GCC on on desktop with i7-3820 processor

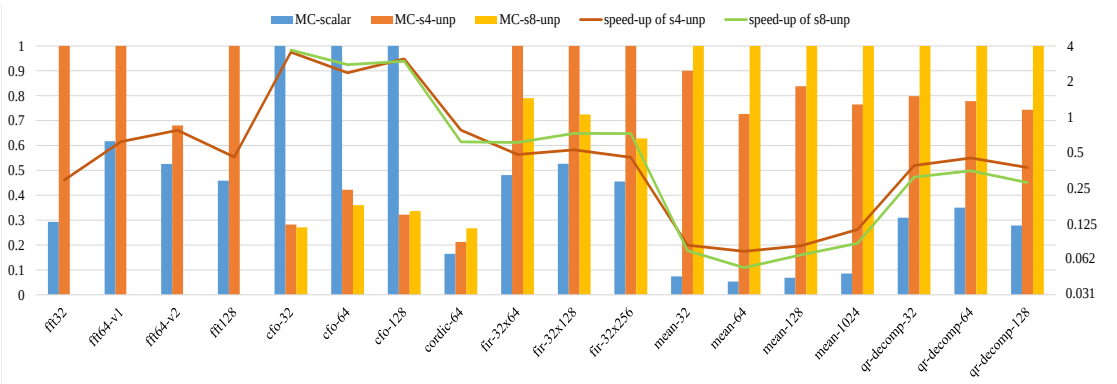

Figure 181: Performance of vectorized code compared to scalarized code using unpacked floating point data types, compiled with GCC on on desktop with i7-3820 processor

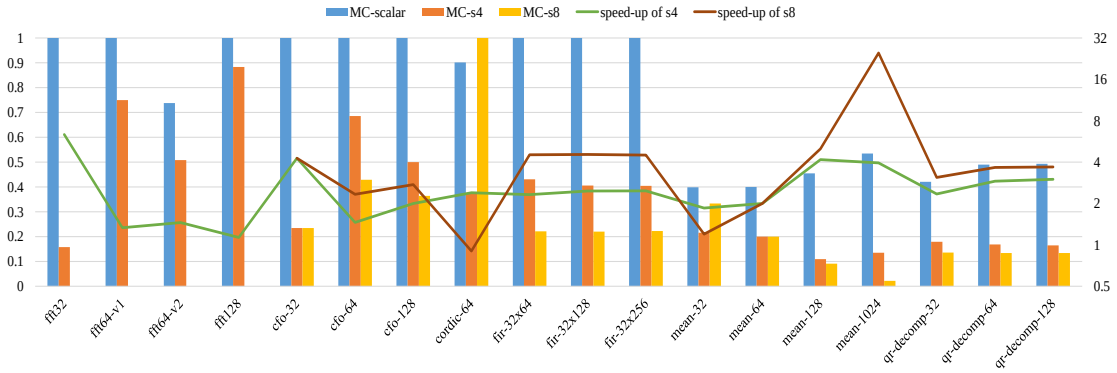

Figure 182: Performance of vectorized code compared to scalarized code using packed fixed point data types, compiled with MSVC on on desktop with i7-3820 processor

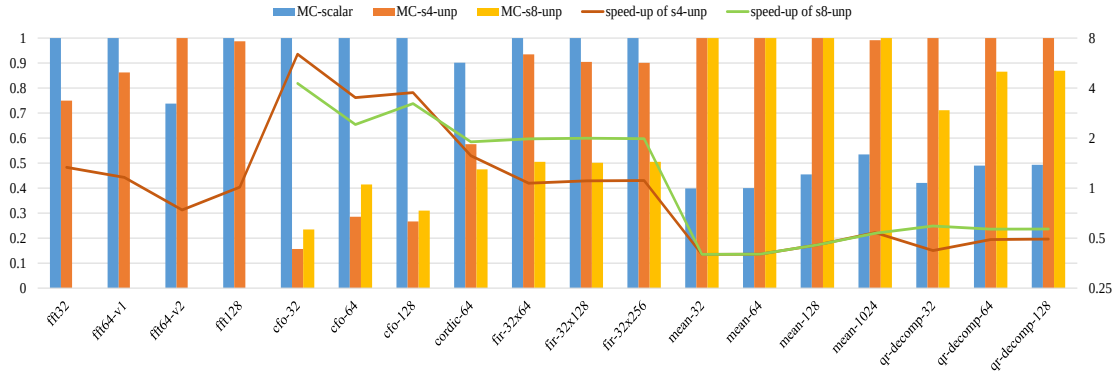

Figure 183: Performance of vectorized code compared to scalarized code using unpacked fixed point data types, compiled with MSVC on on desktop with i7-3820 processor

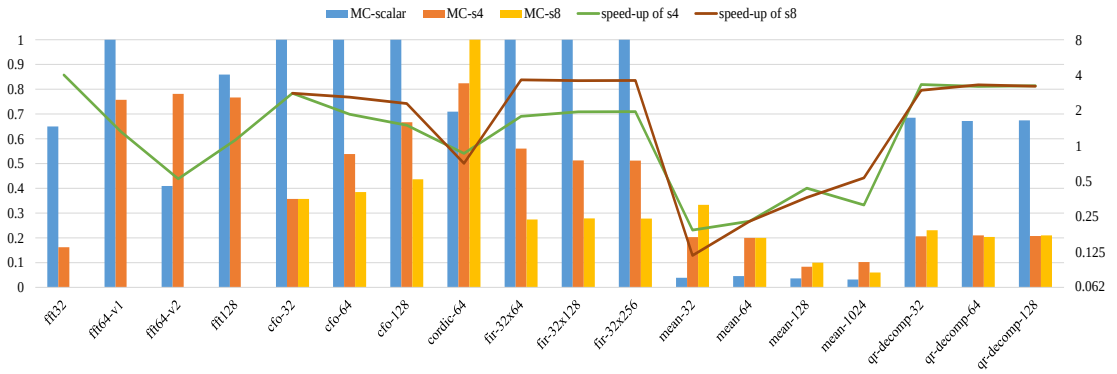

Figure 184: Performance of vectorized code compared to scalarized code using packed floating point data types, compiled with MSVC on on desktop with i7-3820 processor

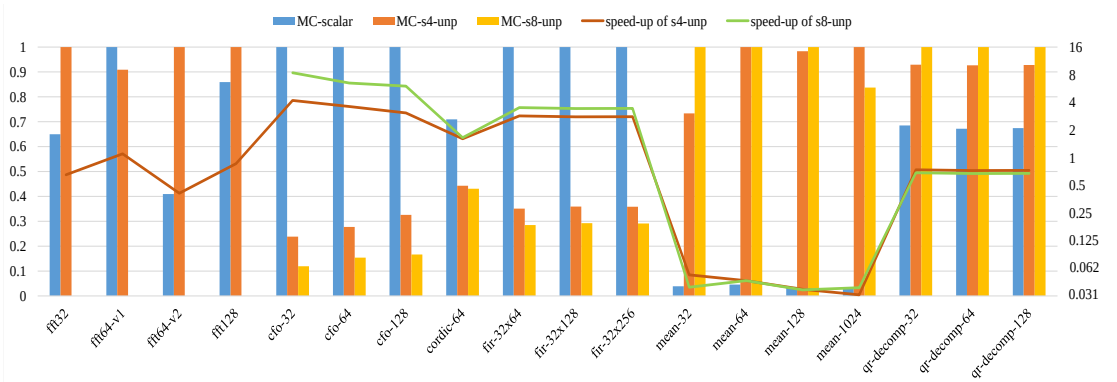

Figure 185: Performance of vectorized code compared to scalarized code using unpacked floating point data types, compiled with MSVC on on desktop with i7-3820 processor

8.5.4 Performance of Vectorized Code Against Compiler's Scalarized Code on x86 Desktop with i7-3770 Processor

	Fig. 186, Fig. 187	Fig. 188, Fig. 189	Fig. 190, Fig. 191	Fig. 192, Fig. 193	Fig. 194, Fig. 195	Fig. 196, Fig. 197
fft32	0.21	0.15	0.79	0.76	0.30	0.30
fft64-v1	1.84	1.67	1.26	2.08	1.00	1.30
fft64-v2	1.79	1.84	0.87	1.61	2.00	1.30
fft128	3.29	3.22	3.70	3.94	3.36	3.60
cfo-32	0.58	0.57	0.46	0.50	0.40	0.60
cfo-64	1.61	1.57	1.30	1.58	0.90	1.12
cfo-128	2.36	2.71	1.78	2.39	1.90	2.30
cordic-64	55.86	40.91	10.31	39.75	24.52	37.72
fir-32x64	15.74	14.76	7.18	14.91	19.66	15.22
fir-32x128	29.64	26.43	14.17	35.18	42.20	33.34
fir-32x256	52.55	47.27	26.62	61.01	88.04	69.50
mean-32	0.42	0.45	0.90	0.77	0.20	0.20
mean-64	0.60	0.59	1.52	1.27	0.40	0.40
mean-128	1.62	1.73	3.12	2.99	0.90	0.90
mean-1024	9.40	9.61	18.19	15.54	7.74	7.72
qr-dec-32	31.75	30.86	32.40	35.69	35.00	26.00
qr-dec-64	53.96	51.87	54.81	60.07	113.46	52.50
qr-dec-128	98.85	102.03	102.15	116.66	140.04	105.04

Table 49: Reference values (exec. time in µs) used for normalization of results on desktop with i7-3770 processor

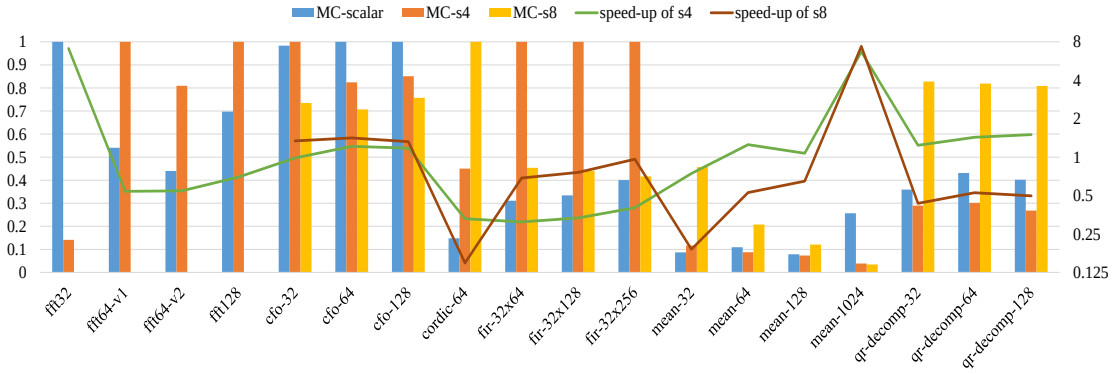

Figure 186: Performance of vectorized code compared to scalarized code using packed fixed point data types, compiled with Clang on desktop with i7-3770 processor

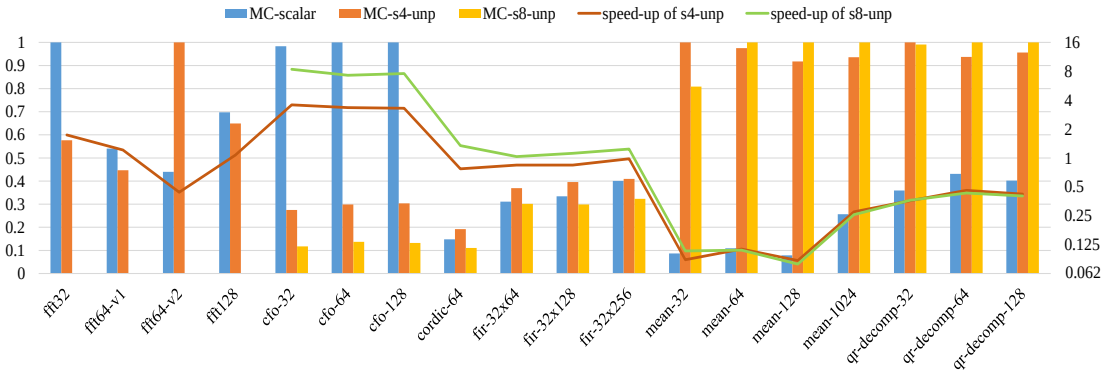

Figure 187: Performance of vectorized code compared to scalarized code using unpacked fixed point data types, compiled with Clang on desktop with i7-3770 processor

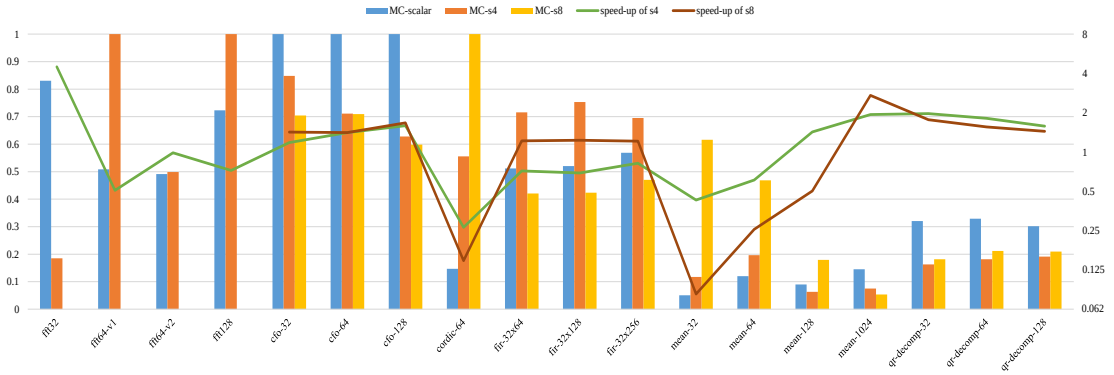

Figure 188: Performance of vectorized code compared to scalarized code using packed floating point data types, compiled with Clang on desktop with i7-3770 processor

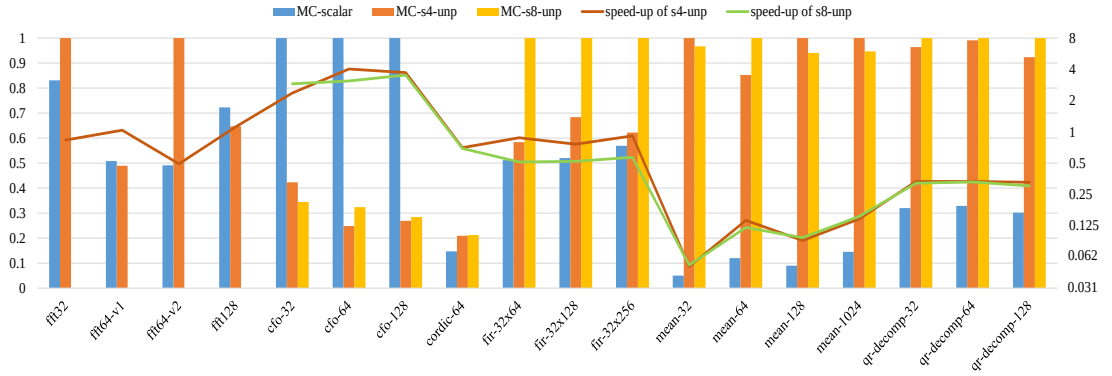

Figure 189: Performance of vectorized code compared to scalarized code using unpacked floating point data types, compiled with Clang on on desktop with i7-3770 processor

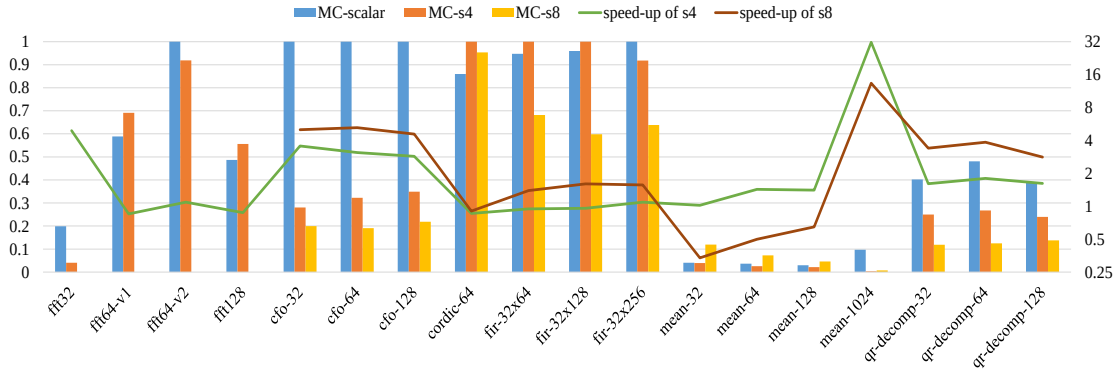

Figure 190: Performance of vectorized code compared to scalarized code using packed fixed point data types, compiled with GCC on on desktop with i7-3770 processor

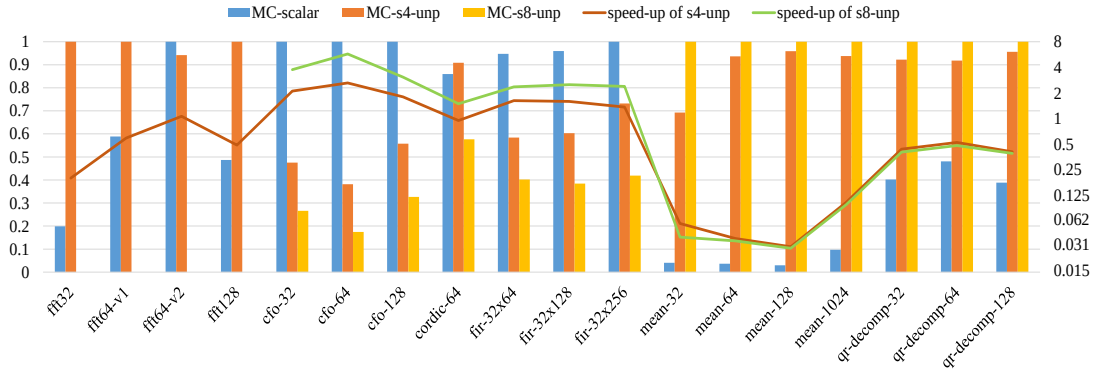

Figure 191: Performance of vectorized code compared to scalarized code using unpacked fixed point data types, compiled with GCC on on desktop with i7-3770 processor

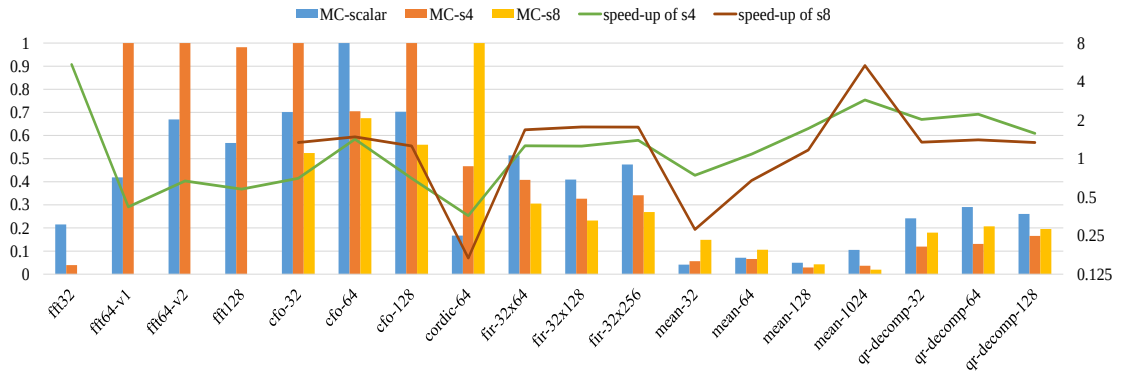

Figure 192: Performance of vectorized code compared to scalarized code using packed floating point data types, compiled with GCC on on desktop with i7-3770 processor

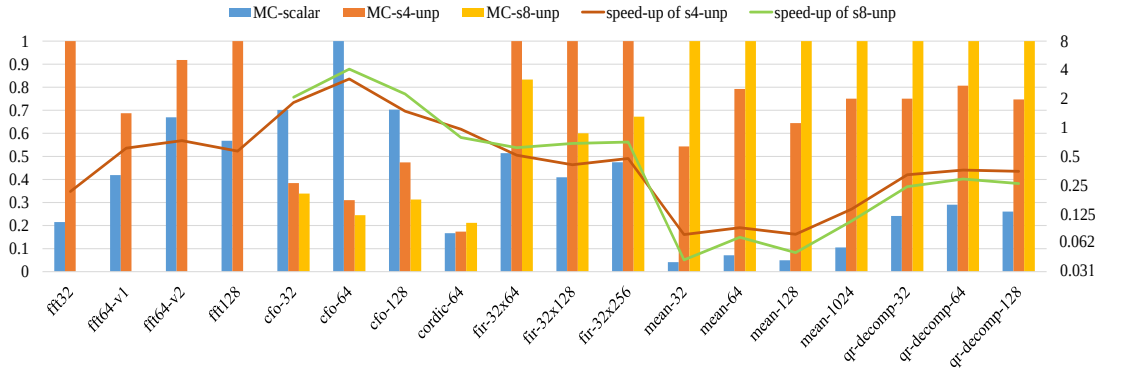

Figure 193: Performance of vectorized code compared to scalarized code using unpacked floating point data types, compiled with GCC on on desktop with i7-3770 processor

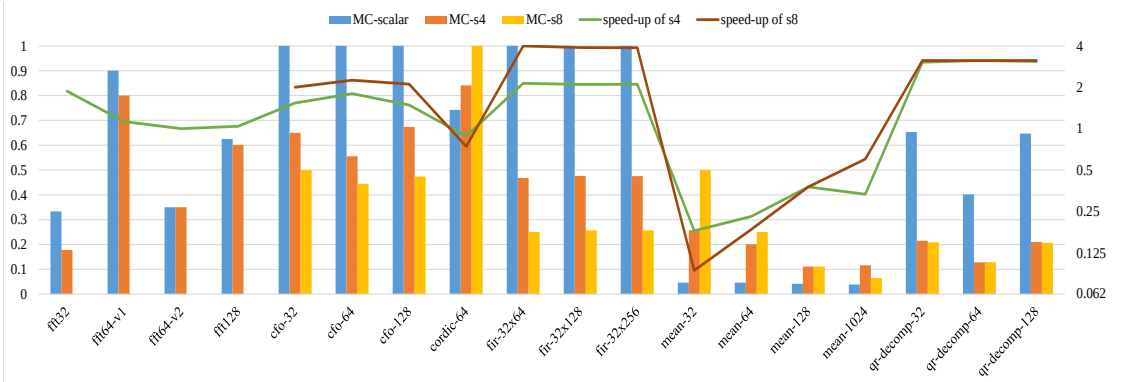

Figure 194: Performance of vectorized code compared to scalarized code using packed fixed point data types, compiled with MSVC on on desktop with i7-3770 processor

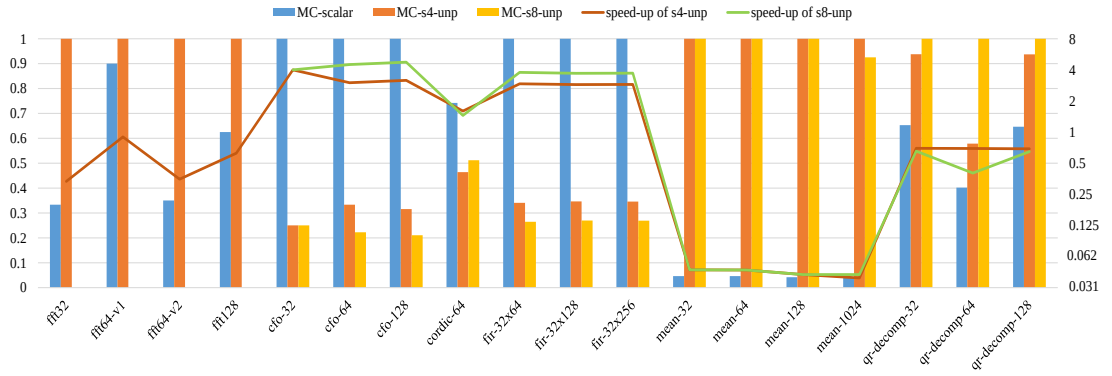

Figure 195: Performance of vectorized code compared to scalarized code using unpacked fixed point data types, compiled with MSVC on on desktop with i7-3770 processor

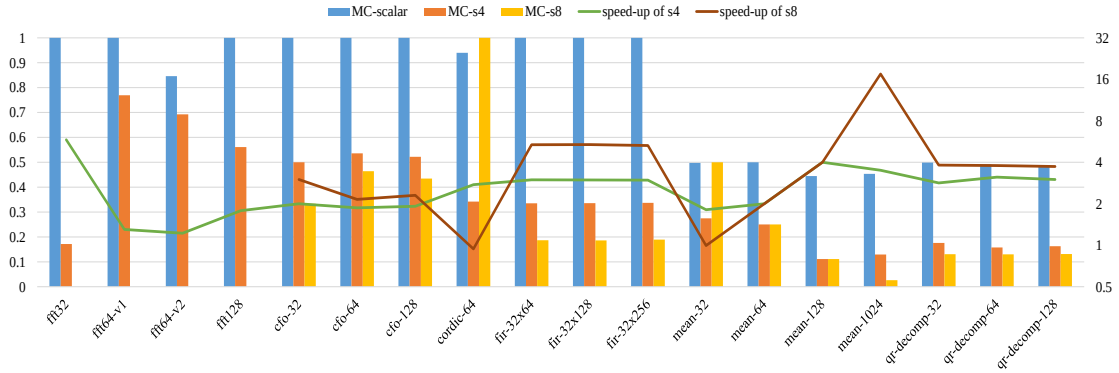

Figure 196: Performance of vectorized code compared to scalarized code using packed floating point data types, compiled with MSVC on on desktop with i7-3770 processor

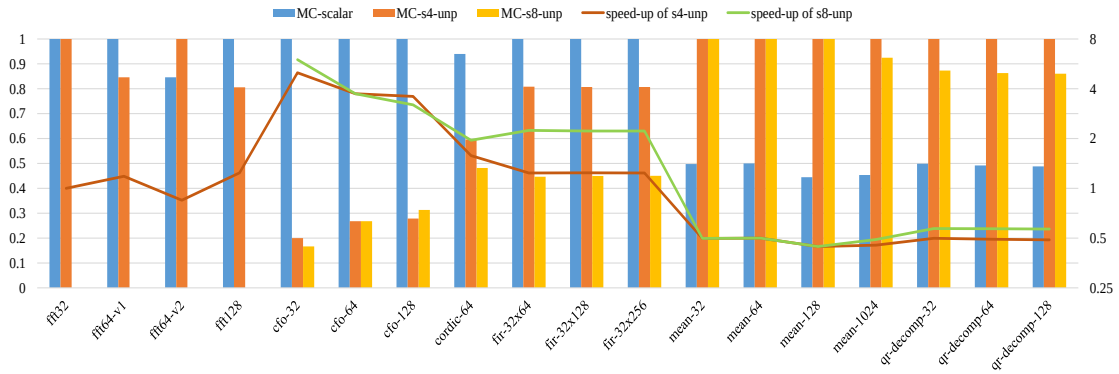

Figure 197: Performance of vectorized code compared to scalarized code using unpacked floating point data types, compiled with MSVC on on desktop with i7-3770 processor

8.6 Performance of MathWorks Generated Code, Compiler's Scalarized Generated Code and Non-vectorized Generated Code

This section present the performance results of MathWorks generated code against compiler's scalarized generated (without intrinsics) code. The performance results of auto-vectorized scalarized code compared to that disabling auto-vectorization are presented as well. The various experimental results concern the different data types, C compilers and targeted architectures. The results complement those presented at section (6.8) comparing the performance of MathWorks generated code with that of compiler's scalarized generated code and section (6.10) regarding the evaluation of auto-vectorization.

8.6.1 Performance of MathWorks Generated Code, Compiler's Scalarized Generated Code and Non-vectorized Generated Code on Raspberry PI 2

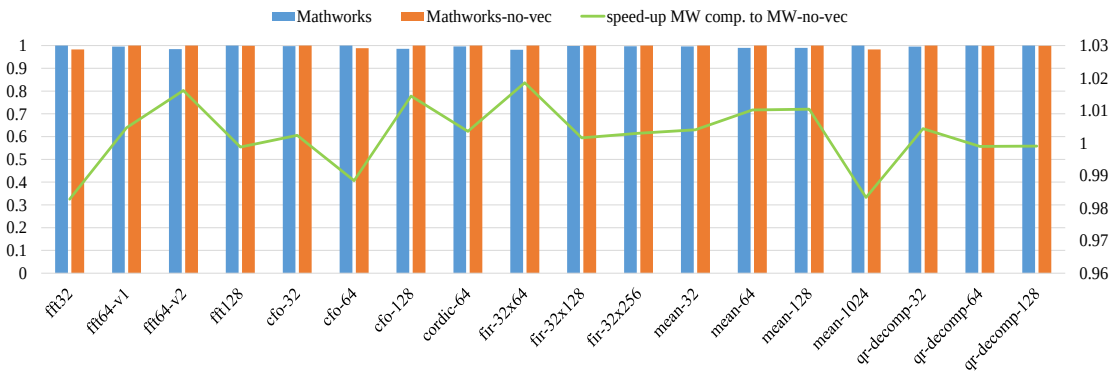

Figure 198: Performance of MathWorks fixed point generated code compared to MathWorks non-vectorized code compiling with Clang on Raspberry PI 2

	Fig. 198, Fig. 199, Fig. 200	Fig. 201, Fig. 202, Fig. 203	Fig. 204, Fig. 205, Fig. 206	Fig. 207, Fig. 208, Fig. 209	Fig. 210, Fig. 211, Fig. 212	Fig. 213, Fig. 214, Fig. 215
fft32	94.92	4.02	87.75	4.61	48.07	2.90
fft64-v1	31.48	15.25	27.06	15.84	52.33	10.23
fft64-v2	158.72	11.11	141.34	10.63	100.20	8.10
fft128	345.61	30.42	311.50	33.43	226.77	22.07
cfo-32	9.07	37.69	10.31	37.91	22.30	18.07
cfo-64	63.95	74.88	64.83	75.86	42.67	34.13
cfo-128	128.30	148.68	132.69	153.63	83.03	66.83
cordic-64	1182.32	719.76	2081.48	658.68	4106.57	632.87
fir-32x64	178.77	468.58	484.10	404.60	1695.80	396.20
fir-32x128	401.27	1035.53	1062.35	893.02	3694.17	863.70
fir-32x256	807.75	2176.11	2231.41	1879.36	7743.83	1806.10
mean-32	0.70	0.57	0.81	0.65	2.10	0.46
mean-64	1.36	1.13	1.37	1.28	3.40	0.97
mean-128	2.66	2.19	2.44	2.46	6.10	1.83
mean-1024	21.05	17.20	17.48	19.22	46.60	9.70
qr-dec-32	3148.17	353.98	2690.74	363.93	1388.73	167.27
qr-dec-64	812.29	708.34	832.95	734.44	730.57	343.57
qr-dec-128	1644.79	1459.01	1666.91	1490.86	1445.83	704.80

Table 50: Reference values (exec. time in µs) used for normalization of results on Raspberry PI 2

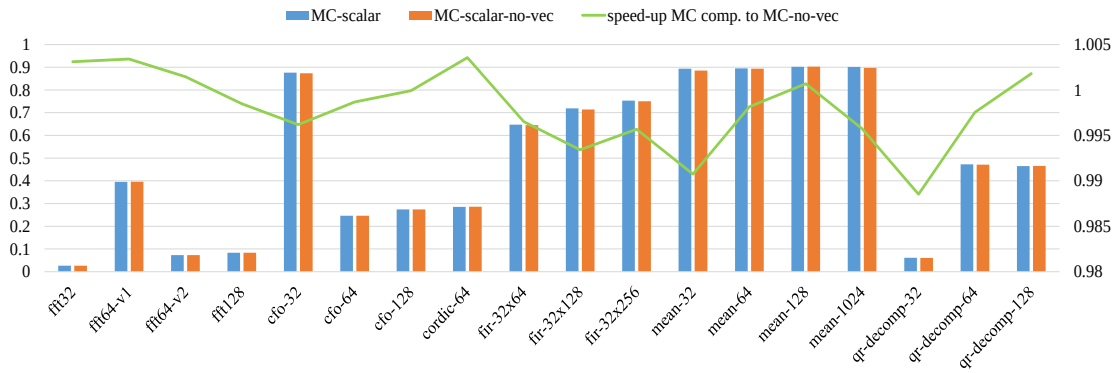

Figure 199: Performance of compiler's scalarized fixed point code compared to compiler's non-vectorized code compiling with Clang on Raspberry PI 2

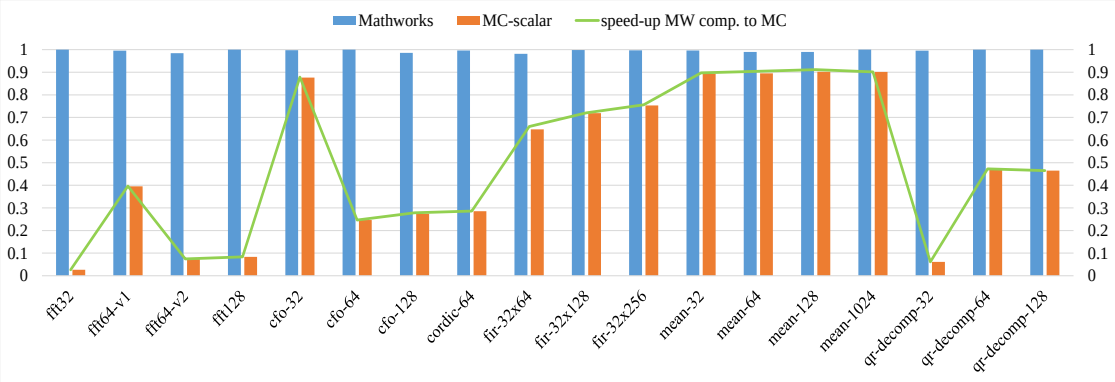

Figure 200: Performance of compiler's scalarized fixed point code compared to MathWorks fixed point generated code compiling with Clang on Raspberry PI 2

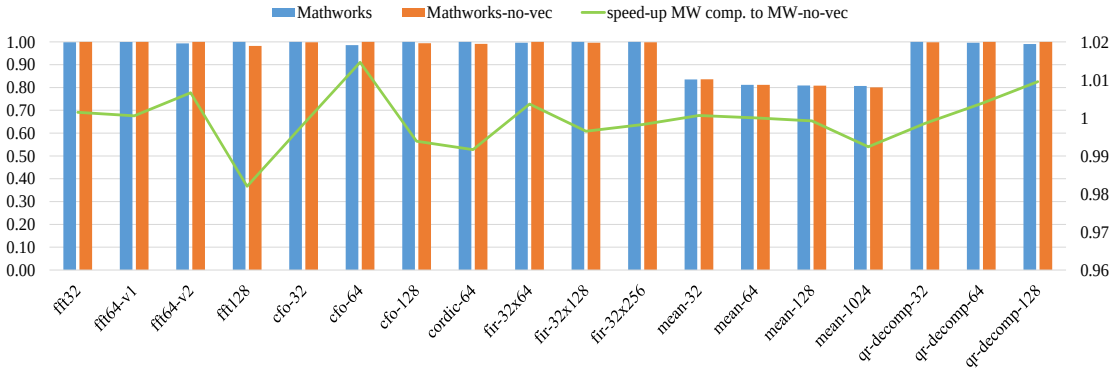

Figure 201: Performance of MathWorks floating point generated code compared to MathWorks non-vectorized code compiling with Clang on Raspberry PI 2

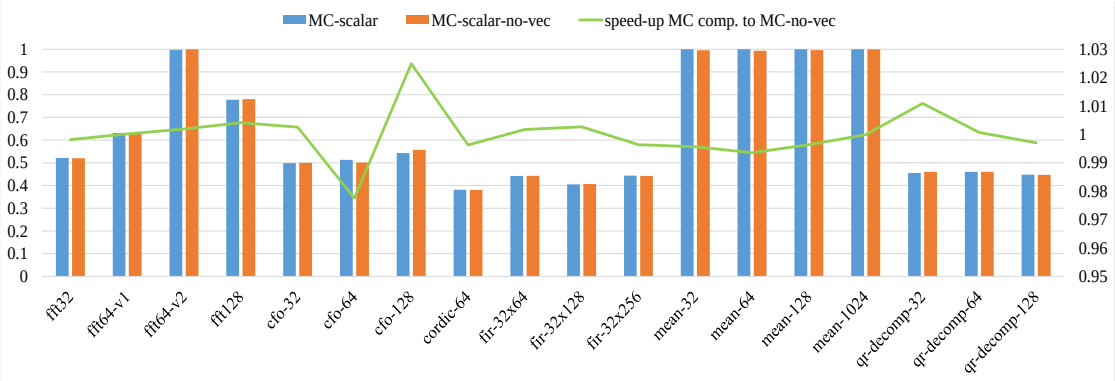

Figure 202: Performance of compiler's scalarized floating point code compared to compiler's non-vectorized code compiling with Clang on Raspberry PI 2

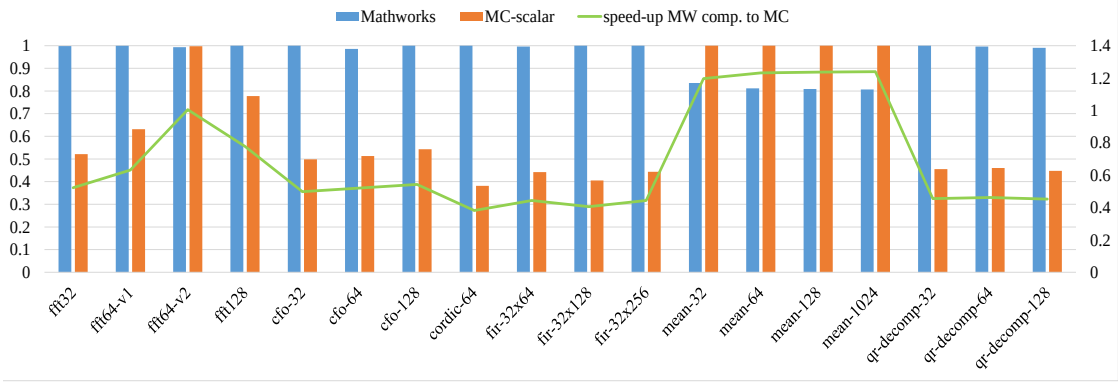

Figure 203: Performance of compiler's scalarized floating point code compared to MathWorks floating point generated code compiling with Clang on Raspberry PI 2

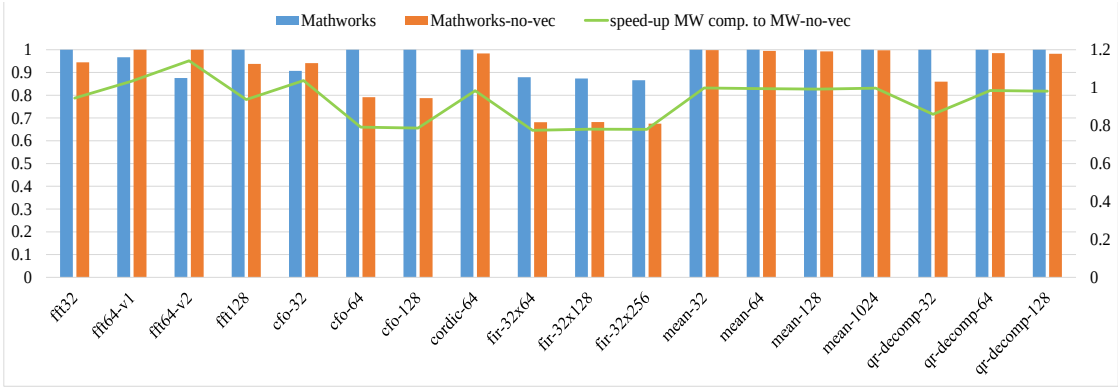

Figure 204: Performance of MathWorks fixed point generated code compared to MathWorks non-vectorized code compiling with GCC on Raspberry PI 2

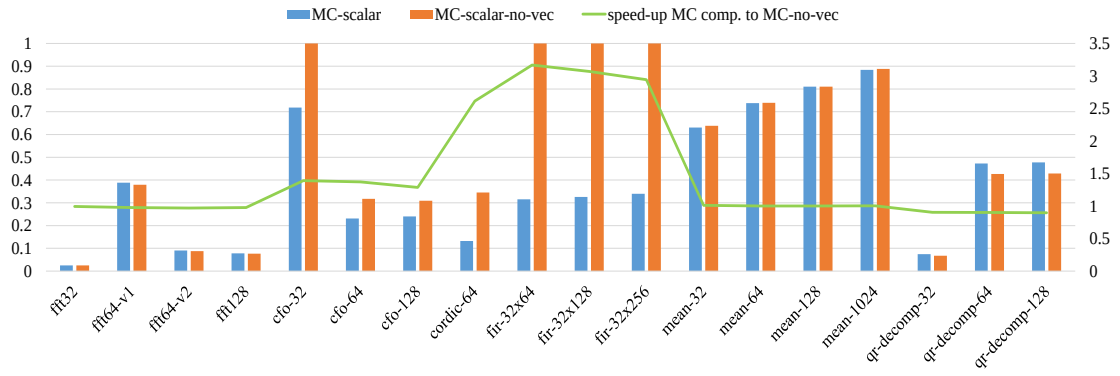

Figure 205: Performance of compiler's scalarized fixed point code compared to compiler's non-vectorized code compiling with GCC on Raspberry PI 2

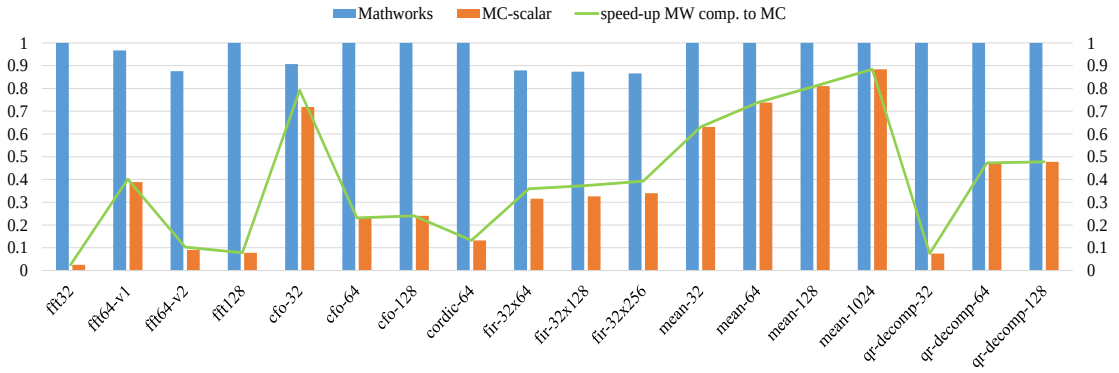

Figure 206: Performance of compiler's scalarized fixed point code compared to MathWorks fixed point generated code compiling with GCC on Raspberry PI 2

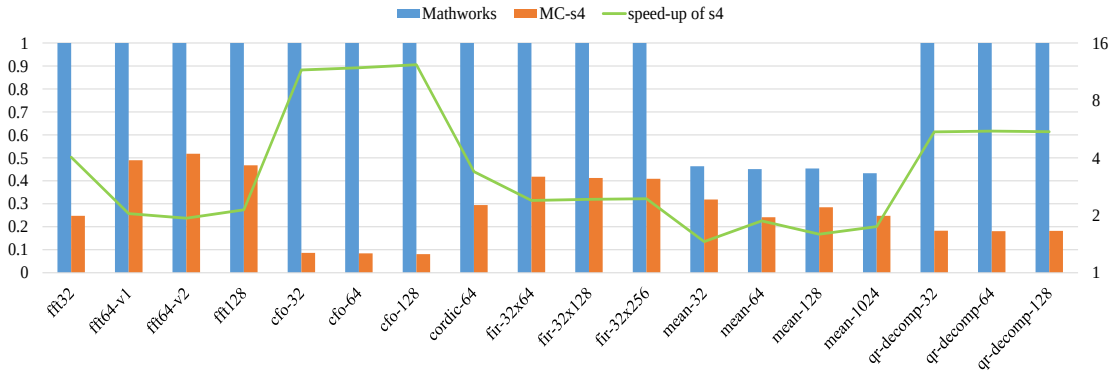

Figure 207: Performance of MathWorks floating point generated code compared to MathWorks non-vectorized code compiling with GCC on Raspberry PI 2

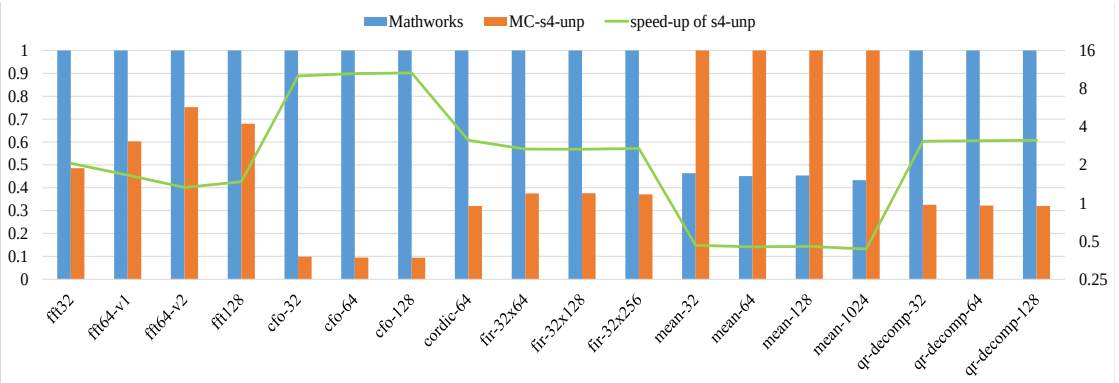

Figure 208: Performance of compiler's scalarized floating point code compared to compiler's non-vectorized code compiling with GCC on Raspberry PI 2

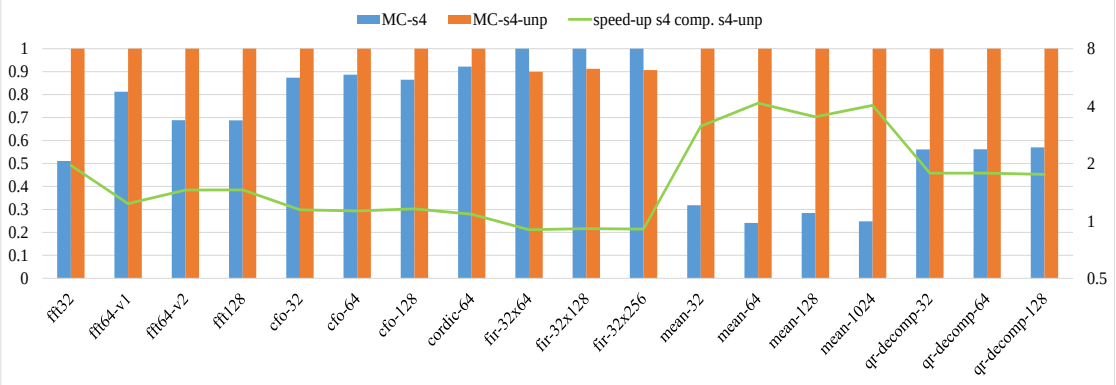

Figure 209: Performance of compiler's scalarized floating point code compared to MathWorks floating point generated code compiling with GCC on Raspberry PI 2

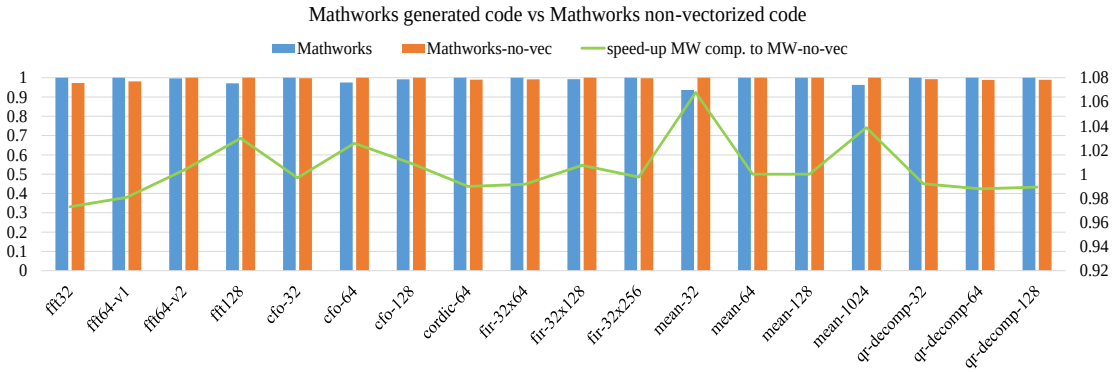

Figure 210: Performance of MathWorks fixed point generated code compared to MathWorks non-vectorized code compiling with MSVC on Raspberry PI 2

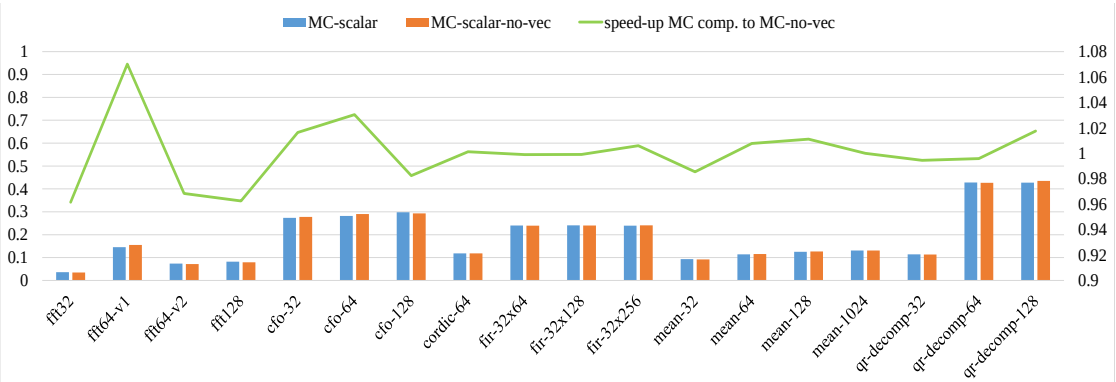

Figure 211: Performance of compiler's scalarized fixed point code compared to compiler's non-vectorized code compiling with MSVC on Raspberry PI 2

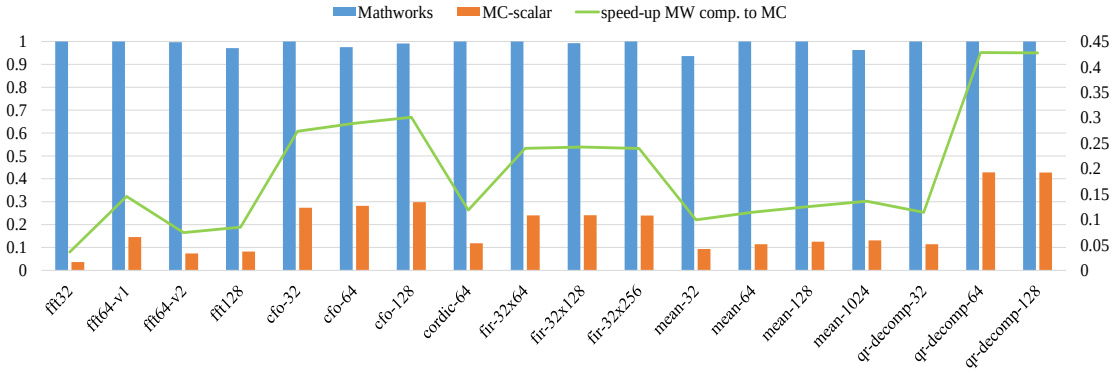

Figure 212: Performance of compiler's scalarized fixed point code compared to MathWorks fixed point generated code compiling with MSVC on Raspberry PI 2

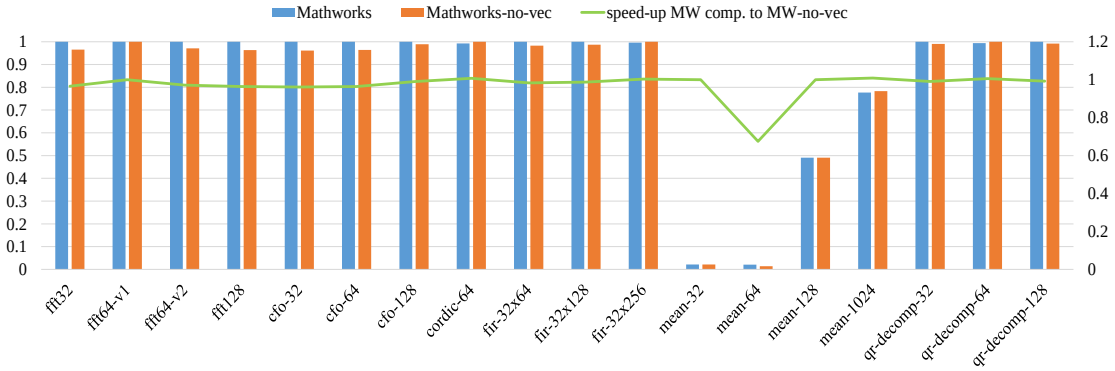

Figure 213: Performance of MathWorks floating point generated code compared to MathWorks non-vectorized code compiling with MSVC on Raspberry PI 2

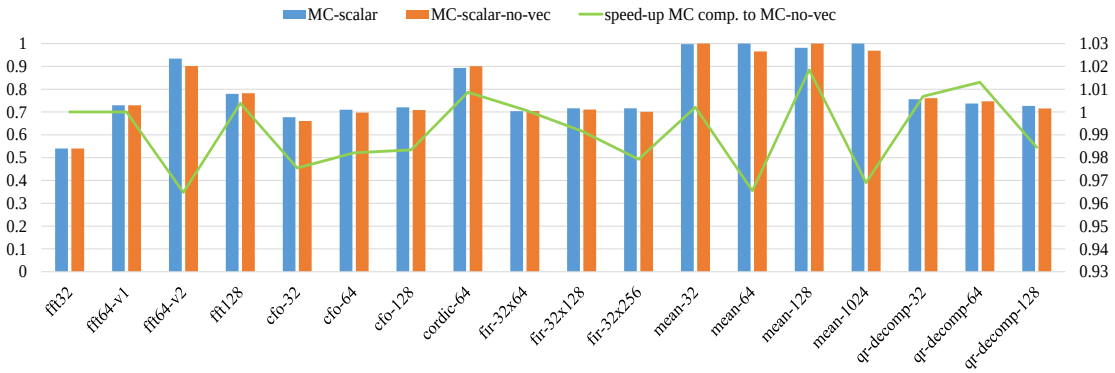

Figure 214: Performance of compiler's scalarized floating point code compared to compiler's non-vectorized code compiling with MSVC on Raspberry PI 2

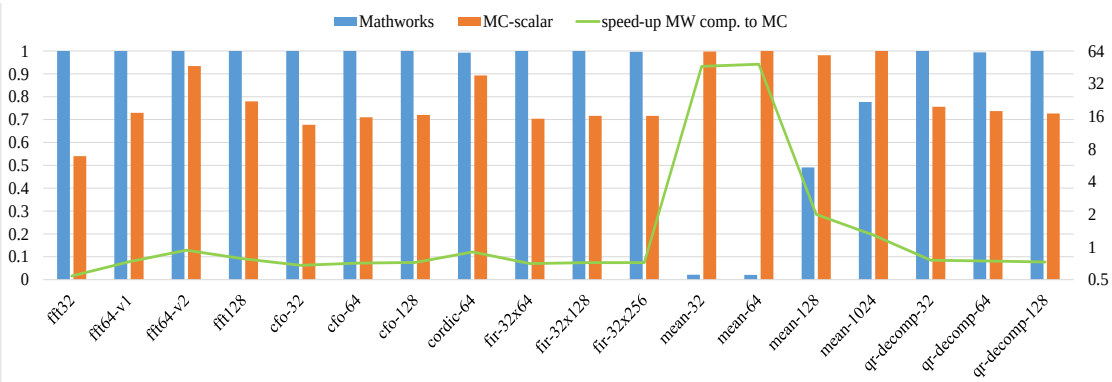

Figure 215: Performance of compiler's scalarized floating point code compared to MathWorks floating point generated code compiling with MSVC on Raspberry PI 2

8.6.2 Performance of MathWorks Generated Code, Compiler's Scalarized Generated Code and Non-vectorized Generated Code on Raspberry PI 3

	Fig. 216, Fig. 217, Fig. 218	**Fig. 219, Fig. 220, Fig. 221**	**Fig. 222, Fig. 223, Fig. 224**	**Fig. 225, Fig. 226, Fig. 227**	**Fig. 228, Fig. 229, Fig. 230**	**Fig. 231, Fig. 232, Fig. 233**
fft32	38.50	1.85	34.35	1.79	27.97	1.77
fft64-v1	14.20	5.93	10.86	6.02	33.73	5.20
fft64-v2	61.00	4.32	50.62	3.60	62.37	4.57
fft128	132.02	11.72	110.65	12.37	137.77	10.70
cfo-32	4.00	20.38	8.01	17.45	13.00	9.57
cfo-64	26.21	37.66	25.60	31.73	25.07	18.97
cfo-128	52.25	76.59	50.90	64.78	49.90	38.27
cordic-64	827.56	313.81	864.14	296.64	2378.07	388.80
fir-32x64	90.90	206.20	208.31	219.40	1093.00	230.23
fir-32x128	216.64	450.65	456.20	480.07	2402.27	499.03
fir-32x256	454.00	935.71	948.89	998.51	5010.60	1050.93
mean-32	0.27	0.29	0.29	0.29	1.50	0.33
mean-64	0.51	0.56	0.49	0.56	2.60	0.63
mean-128	0.99	1.09	0.86	1.10	5.00	1.30
mean-1024	7.73	8.59	6.10	8.60	36.70	9.23
qr-dec-32	1250.31	137.20	1101.88	143.69	840.40	96.73
qr-dec-64	343.48	272.60	343.97	286.99	468.63	196.43
qr-dec-128	695.87	559.17	685.38	579.24	931.37	393.00

Table 51: Reference values (exec. time in µs) used for normalization of results on Raspberry PI 3

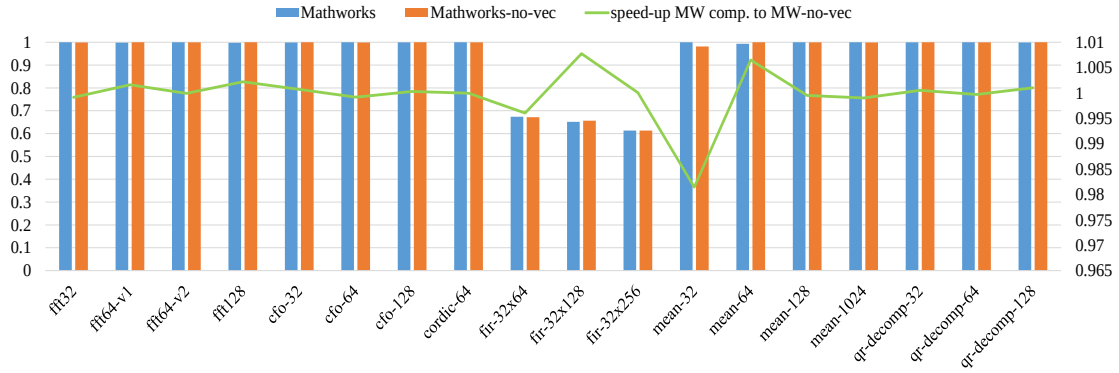

Figure 216: Performance of MathWorks fixed point generated code compared to MathWorks non-vectorized code compiling with Clang on Raspberry PI 3

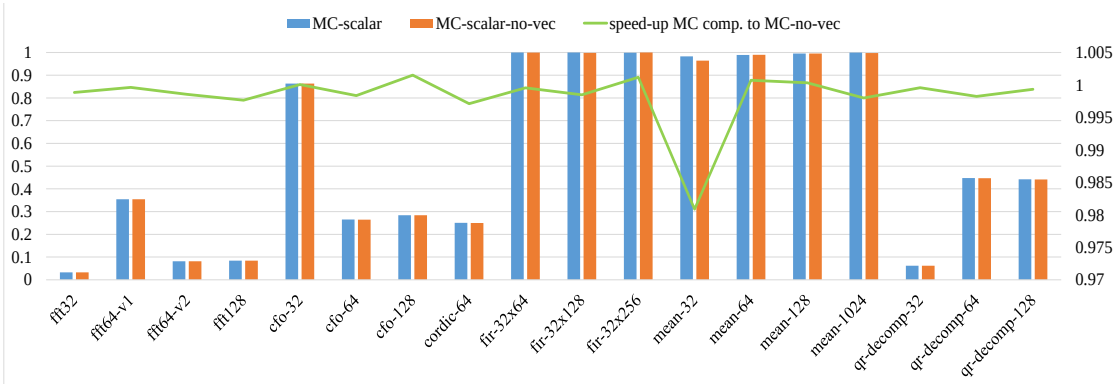

Figure 217: Performance of compiler's scalarized fixed point code compared to compiler's non-vectorized code compiling with Clang on Raspberry PI 3

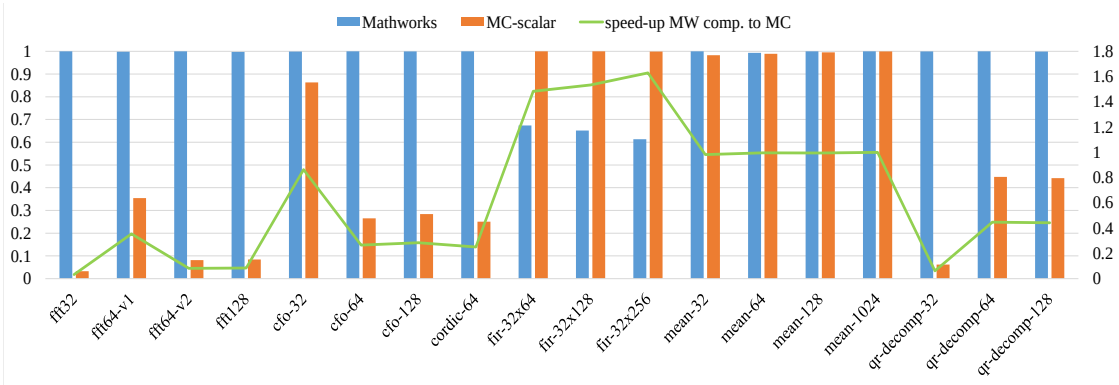

Figure 218: Performance of compiler's scalarized fixed point code compared to MathWorks fixed point generated code compiling with Clang on Raspberry PI 3

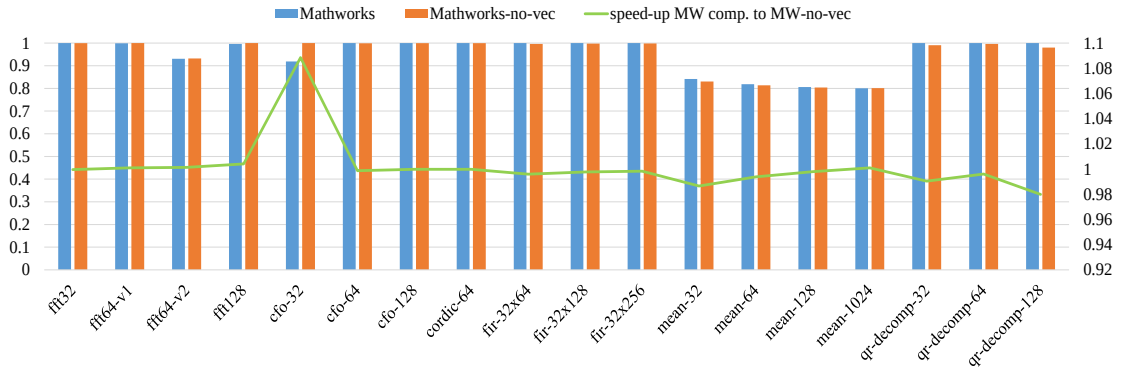

Figure 219: Performance of MathWorks floating point generated code compared to MathWorks non-vectorized code compiling with Clang on Raspberry PI 3

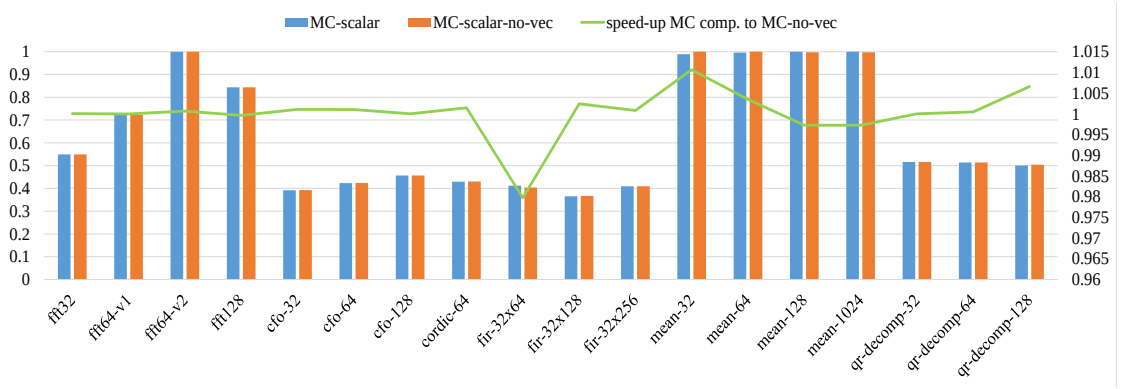

Figure 220: Performance of compiler's scalarized floating point code compared to compiler's non-vectorized code compiling with Clang on Raspberry PI 3

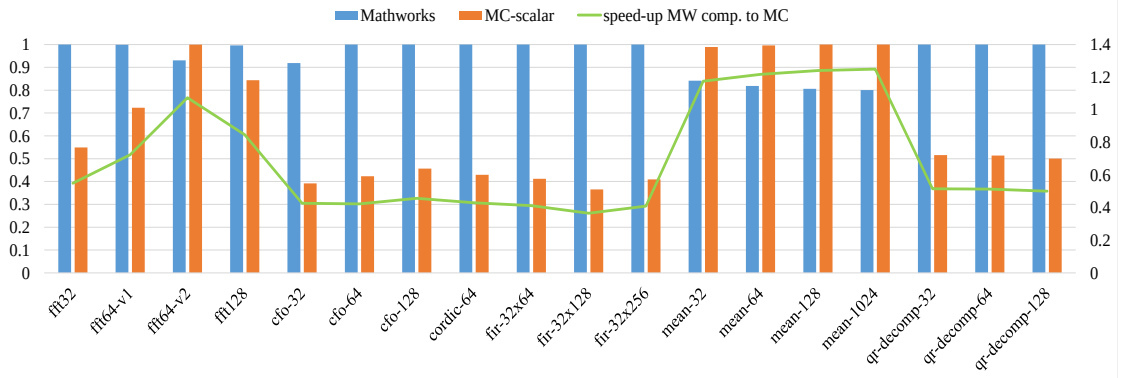

Figure 221: Performance of compiler's scalarized floating point code compared to MathWorks floating point generated code compiling with Clang on Raspberry PI 3

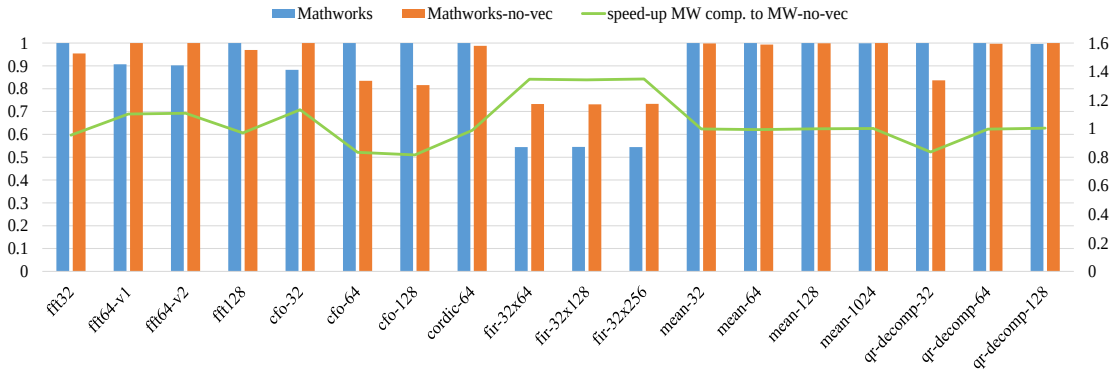

Figure 222: Performance of MathWorks fixed point generated code compared to MathWorks non-vectorized code compiling with GCC on Raspberry PI 3

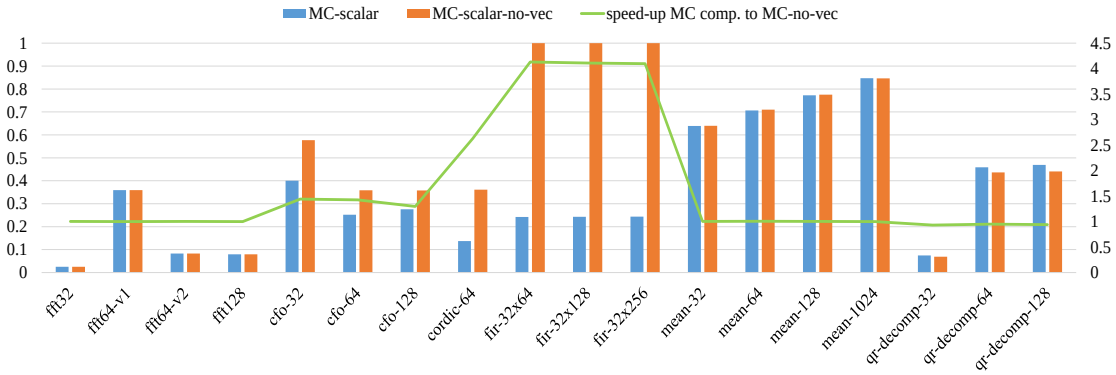

Figure 223: Performance of compiler's scalarized fixed point code compared to compiler's non-vectorized code compiling with GCC on Raspberry PI 3

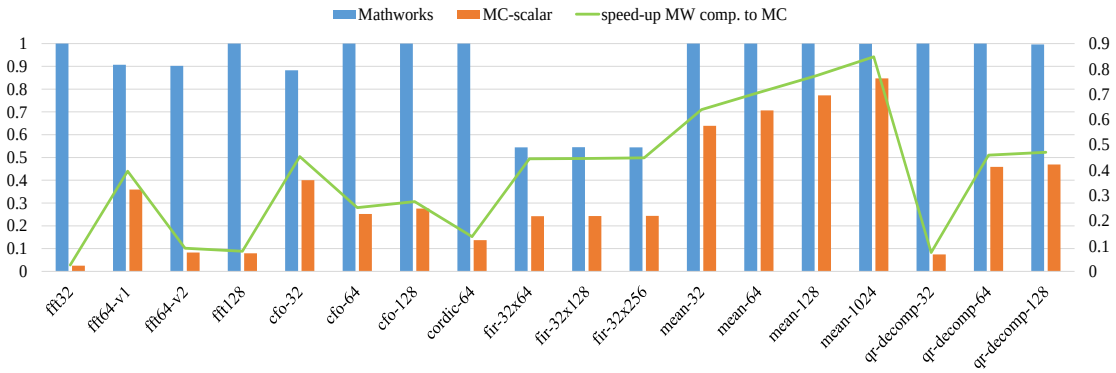

Figure 224: Performance of compiler's scalarized fixed point code compared to MathWorks fixed point generated code compiling with GCC on Raspberry PI 3

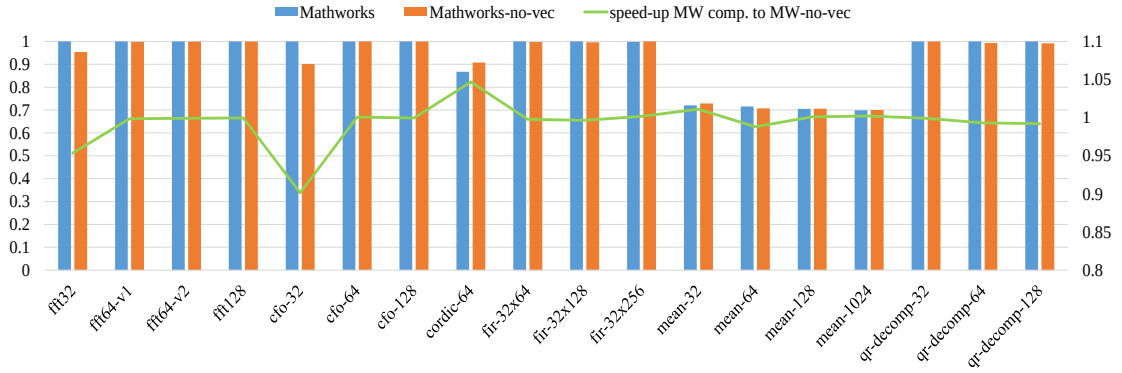

Figure 225: Performance of MathWorks floating point generated code compared to MathWorks non-vectorized code compiling with GCC on Raspberry PI 3

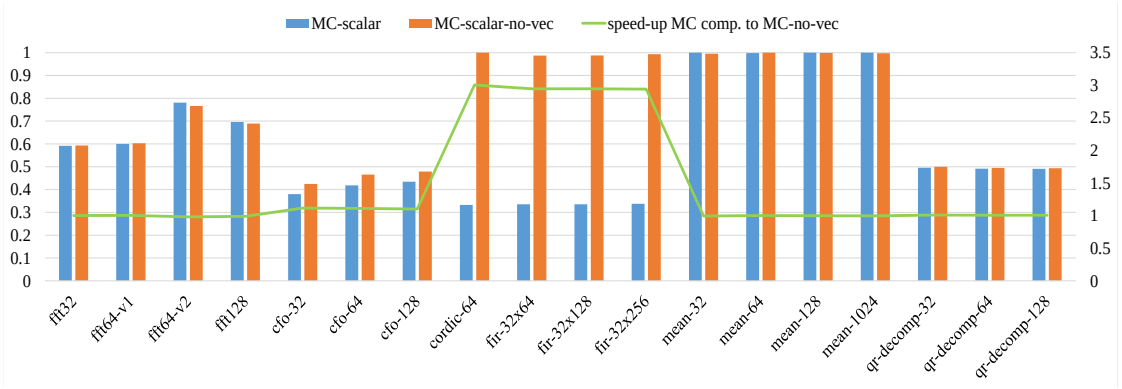

Figure 226: Performance of compiler's scalarized floating point code compared to compiler's non-vectorized code compiling with GCC on Raspberry PI 3

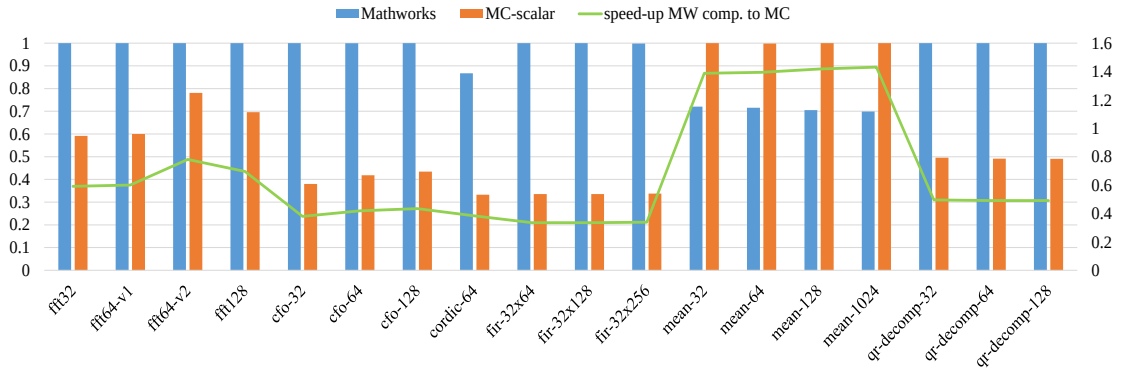

Figure 227: Performance of compiler's scalarized floating point code compared to MathWorks floating point generated code compiling with GCC on Raspberry PI 3

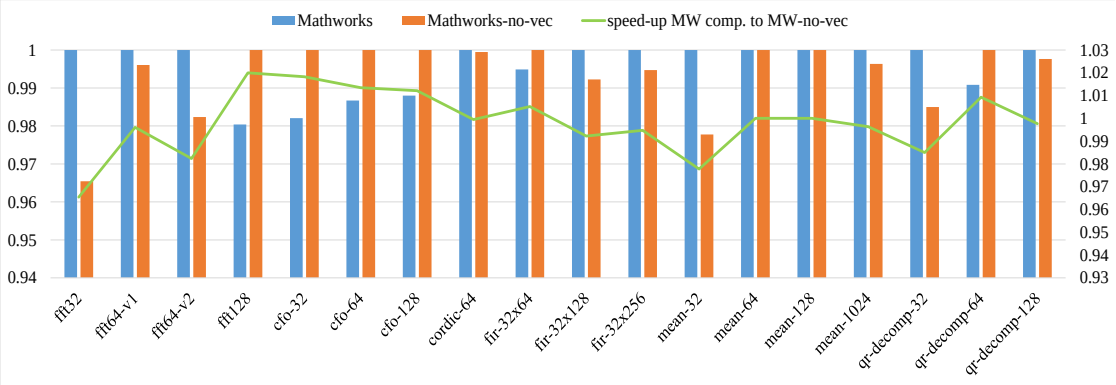

Figure 228: Performance of MathWorks fixed point generated code compared to MathWorks non-vectorized code compiling with MSVC on Raspberry PI 3

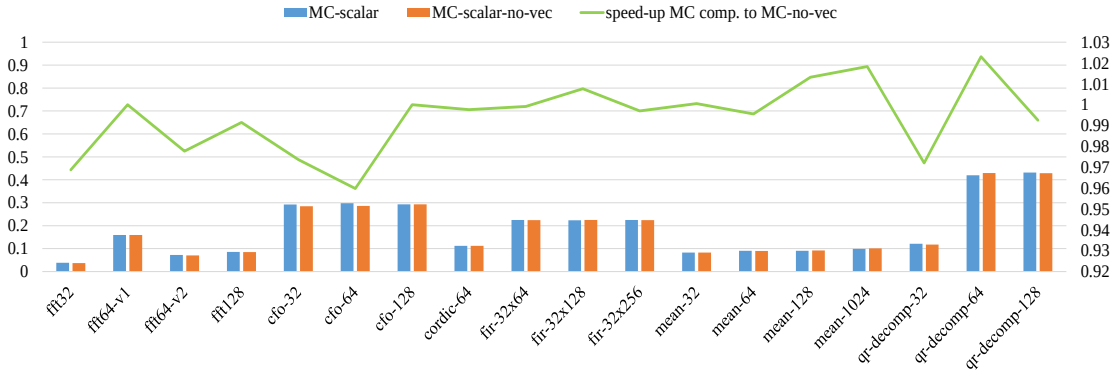

Figure 229: Performance of compiler's scalarized fixed point code compared to compiler's non-vectorized code compiling with MSVC on Raspberry PI 3

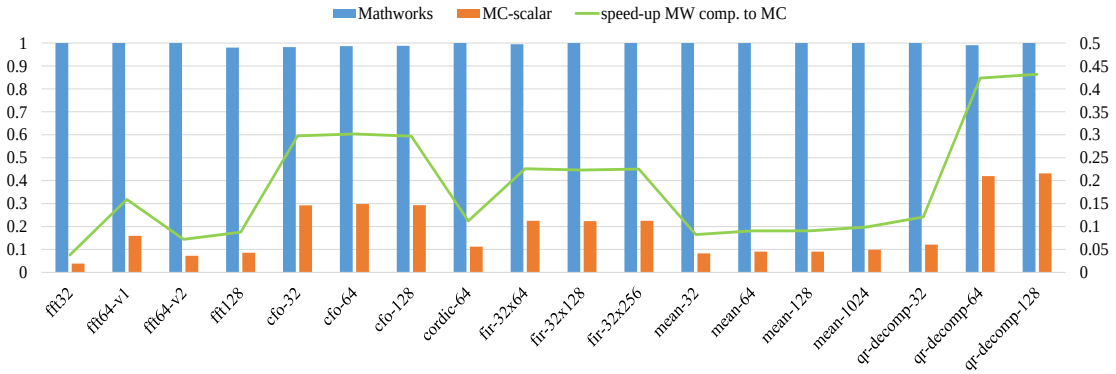

Figure 230: Performance of compiler's scalarized fixed point code compared to MathWorks fixed point generated code compiling with MSVC on Raspberry PI 3

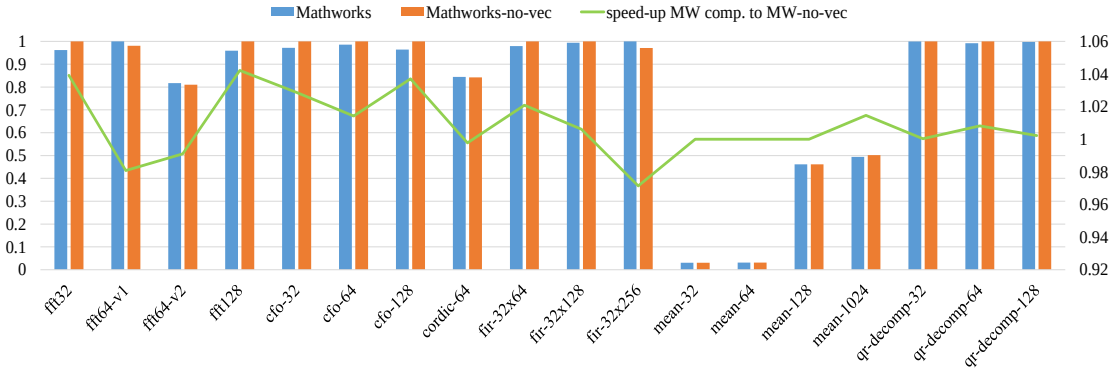

Figure 231: Performance of MathWorks floating point generated code compared to MathWorks non-vectorized code compiling with MSVC on Raspberry PI 3

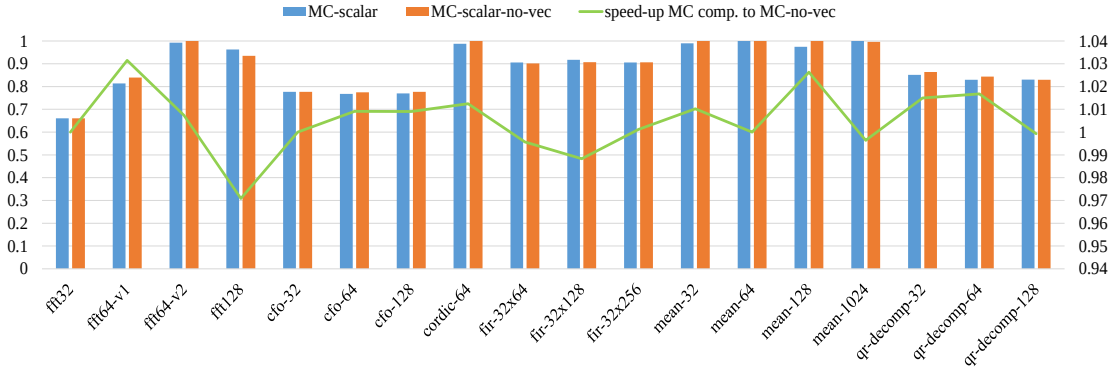

Figure 232: Performance of compiler's scalarized floating point code compared to compiler's non-vectorized code compiling with MSVC on Raspberry PI 3

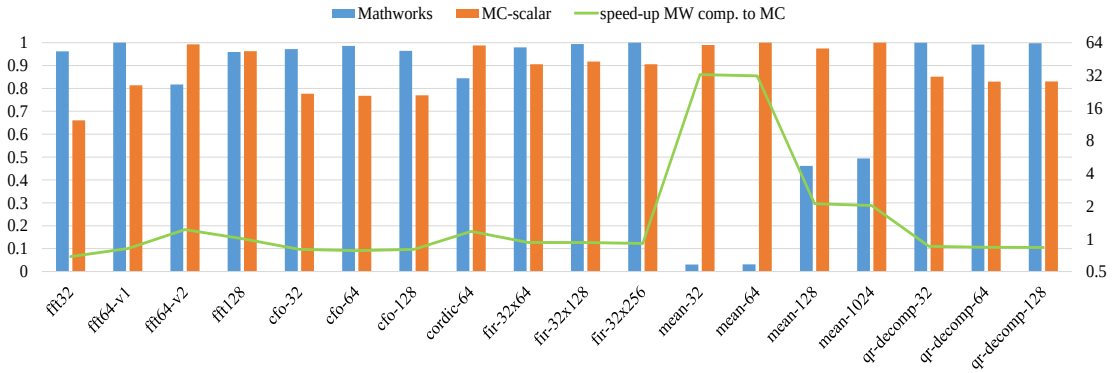

Figure 233: Performance of compiler's scalarized floating point code compared to MathWorks floating point generated code compiling with MSVC on Raspberry PI 3

8.6.3 Performance of MathWorks Generated Code, Compiler's Scalarized Generated Code and Non-vectorized Generated Code on Desktop with i7-3820 Processor

	Fig. 234, Fig. 235, Fig. 236	**Fig. 237, Fig. 238, Fig. 239**	**Fig. 240, Fig. 241, Fig. 242**	**Fig. 243, Fig. 244, Fig. 245**	**Fig. 246, Fig. 247, Fig. 248**	**Fig. 249, Fig. 250, Fig. 251**
fft32	10.95	0.18	10.13	0.18	4.40	0.56
fft64-v1	6.13	1.40	5.33	1.40	3.80	224.60
fft64-v2	16.64	1.29	14.01	1.29	6.12	1.80
fft128	30.06	2.98	30.40	2.98	17.42	5.50
cfo-32	1.40	2.29	1.97	2.29	4.68	4.52
cfo-64	8.22	4.67	7.45	4.67	6.06	5.70
cfo-128	14.21	8.72	16.98	8.72	11.16	8.68
cordic-64	84.38	23.44	202.44	23.44	406.94	40.36
fir-32x64	13.70	11.23	26.23	11.23	104.44	15.64
fir-32x128	24.86	24.85	54.53	24.85	236.48	34.30
fir-32x256	41.15	41.98	108.55	41.98	492.94	71.84
mean-32	0.09	0.08	0.15	0.08	0.20	0.13
mean-64	0.15	0.15	0.16	0.15	0.40	0.20
mean-128	0.50	0.36	0.53	0.36	0.94	0.50
mean-1024	3.41	2.47	2.25	2.47	6.08	4.98
qr-dec-32	209.50	17.01	221.95	17.01	152.02	21.58
qr-dec-64	60.49	30.75	51.77	30.75	86.48	43.08
qr-dec-128	107.87	52.97	91.97	52.97	172.36	85.96

Table 52: Reference values (exec. time in µs) used for normalization of results on desktop with i7-3820 processor

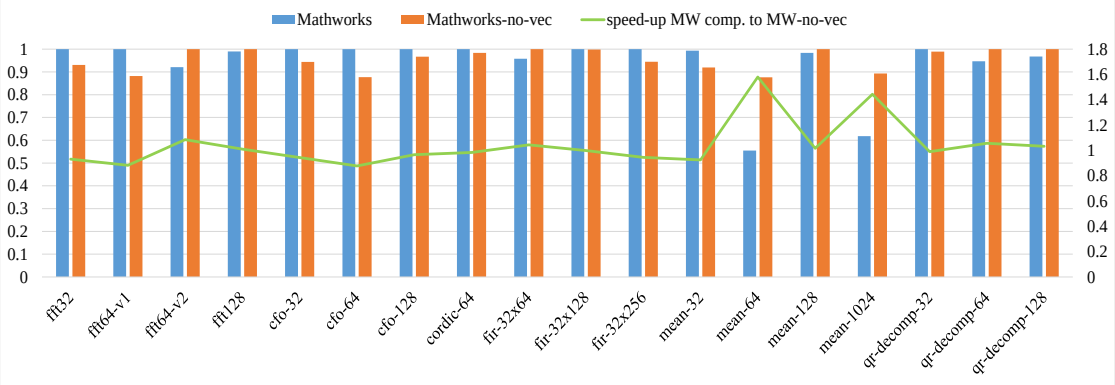

Figure 234: Performance of MathWorks fixed point generated code compared to MathWorks non-vectorized code compiling with Clang on desktop with i7-3820

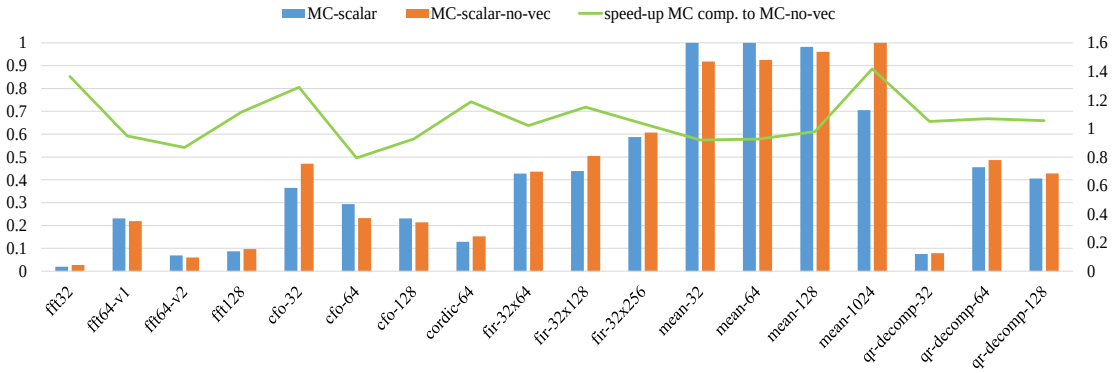

Figure 235: Performance of compiler's scalarized fixed point code compared to compiler's non-vectorized code compiling with Clang on desktop with i7-3820

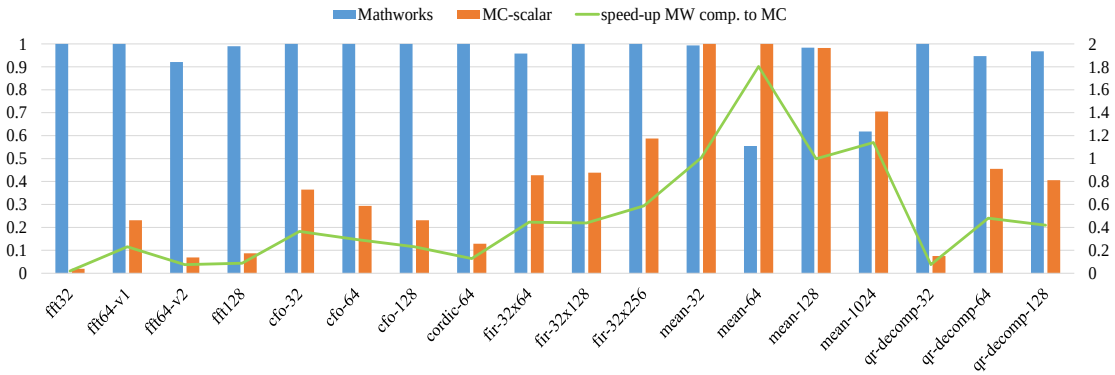

Figure 236: Performance of compiler's scalarized fixed point code compared to MathWorks fixed point generated code compiling with Clang on desktop with i7-3820

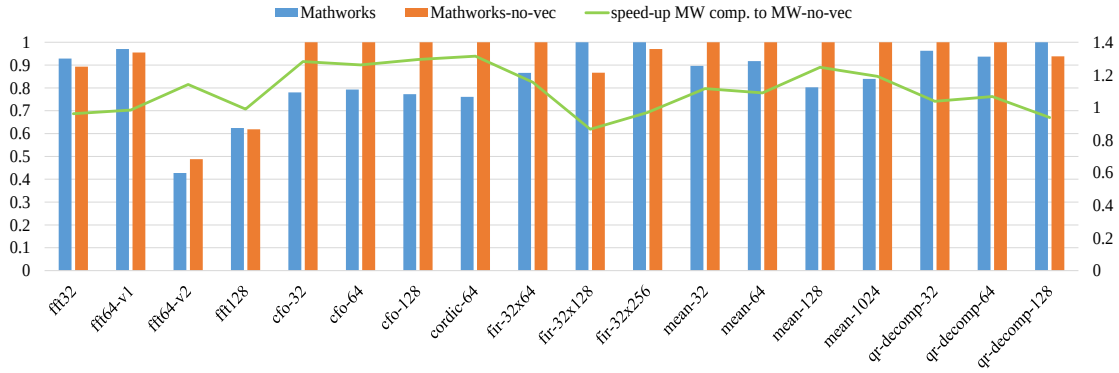

Figure 237: Performance of MathWorks floating point generated code compared to MathWorks non-vectorized code compiling with Clang on desktop with i7-3820

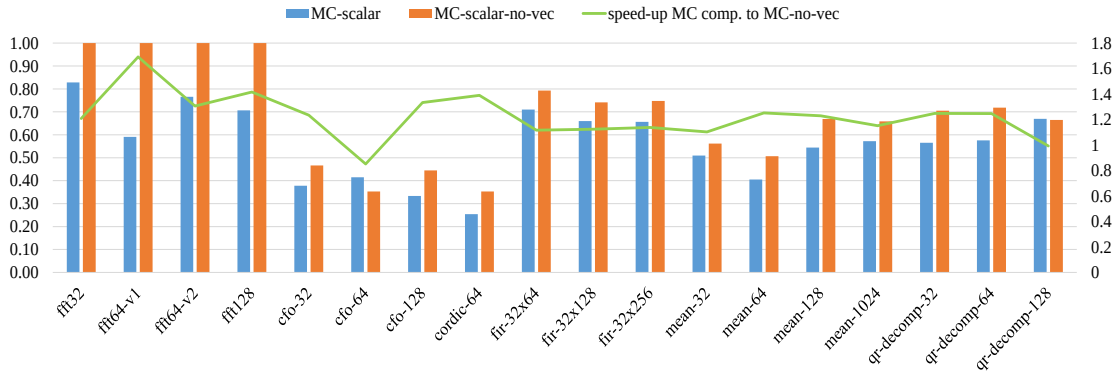

Figure 238: Performance of compiler's scalarized floating point code compared to compiler's non-vectorized code compiling with Clang on desktop with i7-3820

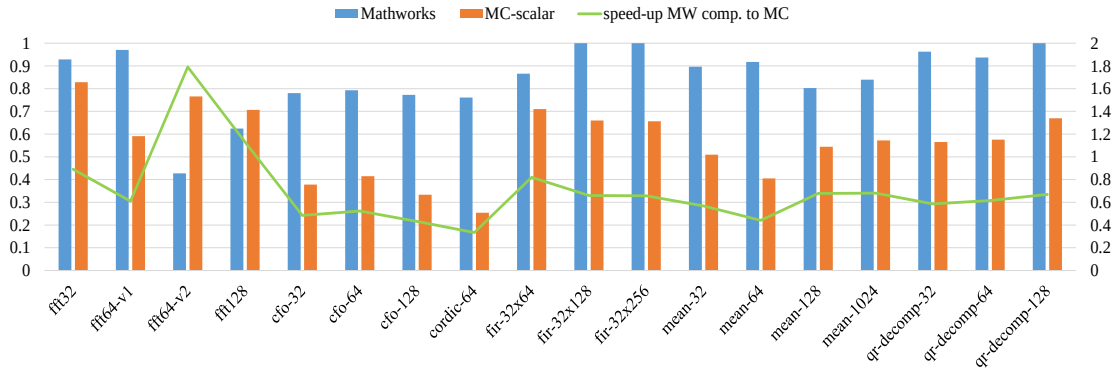

Figure 239: Performance of compiler's scalarized floating point code compared to MathWorks floating point generated code compiling with Clang on desktop with i7-3820

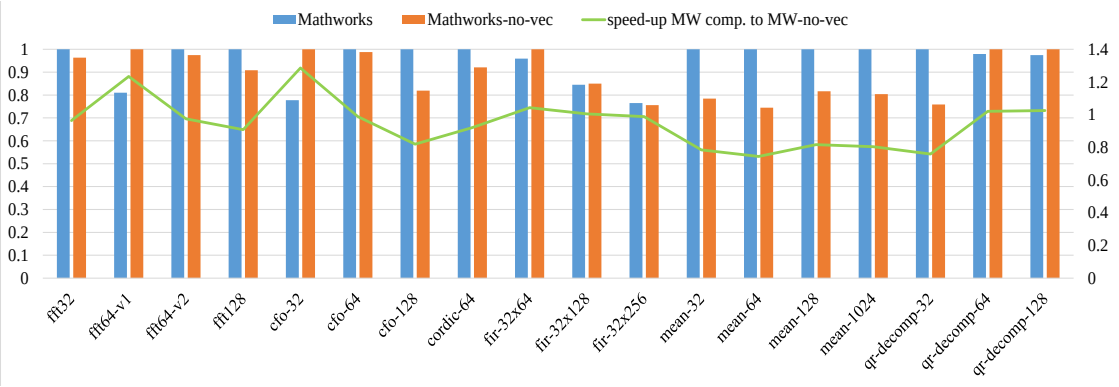

Figure 240: Performance of MathWorks fixed point generated code compared to MathWorks non-vectorized code compiling with GCC on desktop with i7-3820

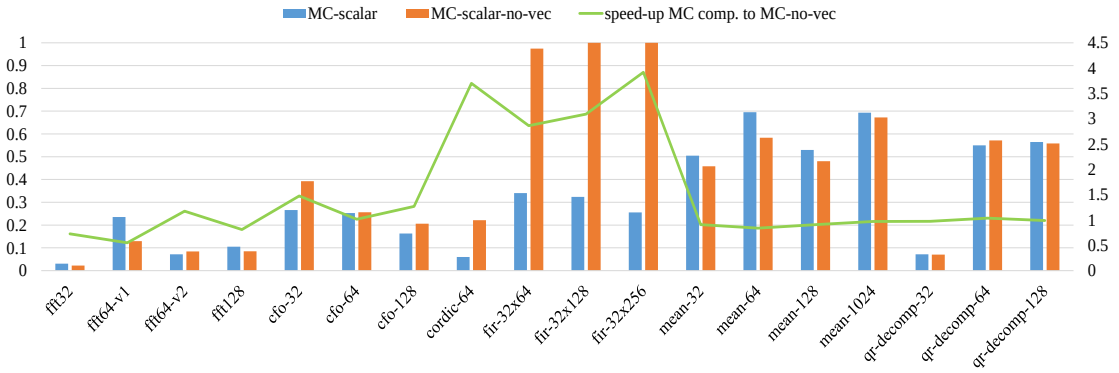

Figure 241: Performance of compiler's scalarized fixed point code compared to compiler's non-vectorized code compiling with GCC on desktop with i7-3820

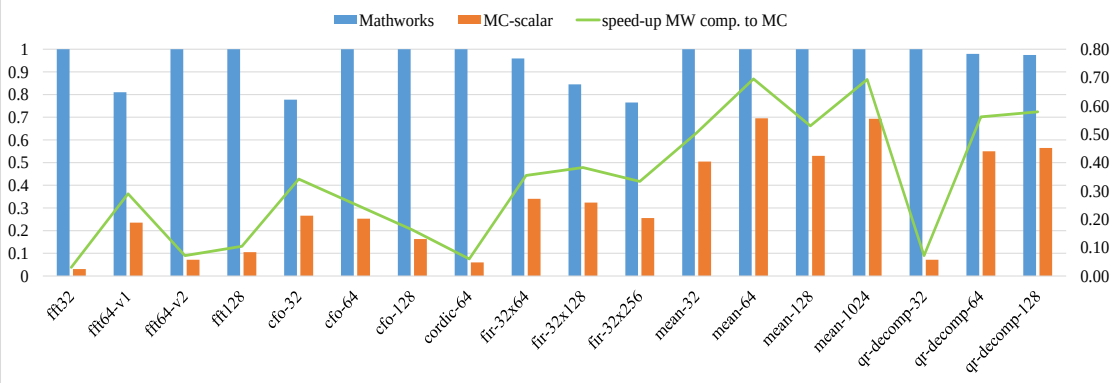

Figure 242: Performance of compiler's scalarized fixed point code compared to MathWorks fixed point generated code compiling with GCC on desktop with i7-3820

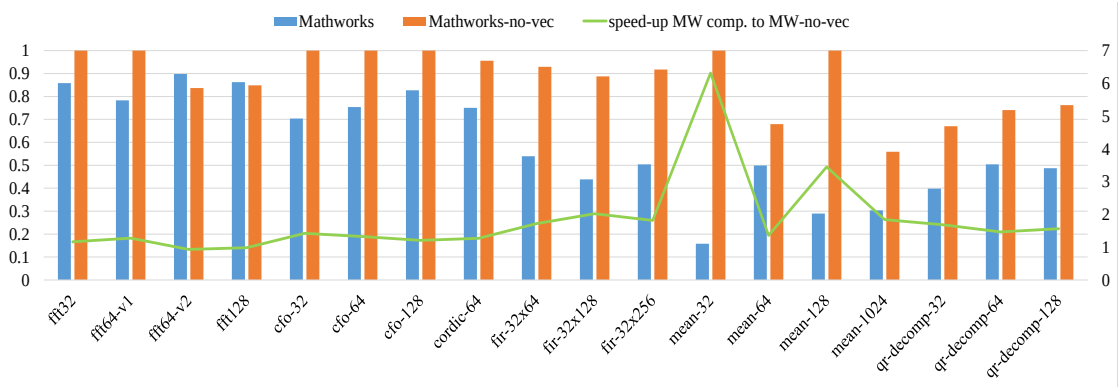

Figure 243: Performance of MathWorks floating point generated code compared to MathWorks non-vectorized code compiling with GCC on desktop with i7-3820

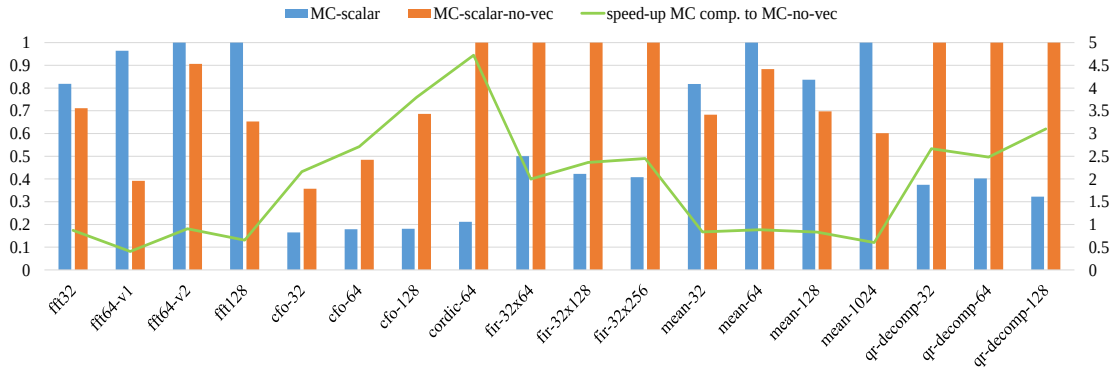

Figure 244: Performance of compiler's scalarized floating point code compared to compiler's non-vectorized code compiling with GCC on desktop with i7-3820

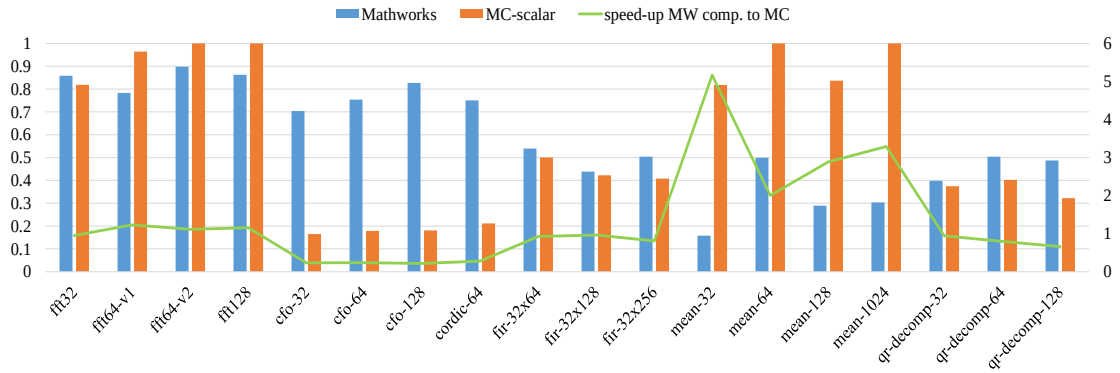

Figure 245: Performance of compiler's scalarized floating point code compared to MathWorks floating point generated code compiling with GCC on desktop with i7-3820

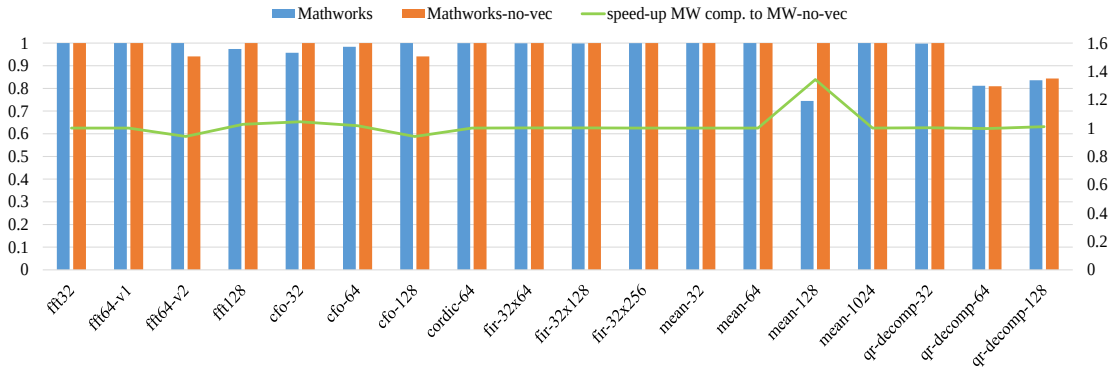

Figure 246: Performance of MathWorks fixed point generated code compared to MathWorks non-vectorized code compiling with MSVC on desktop with i7-3820

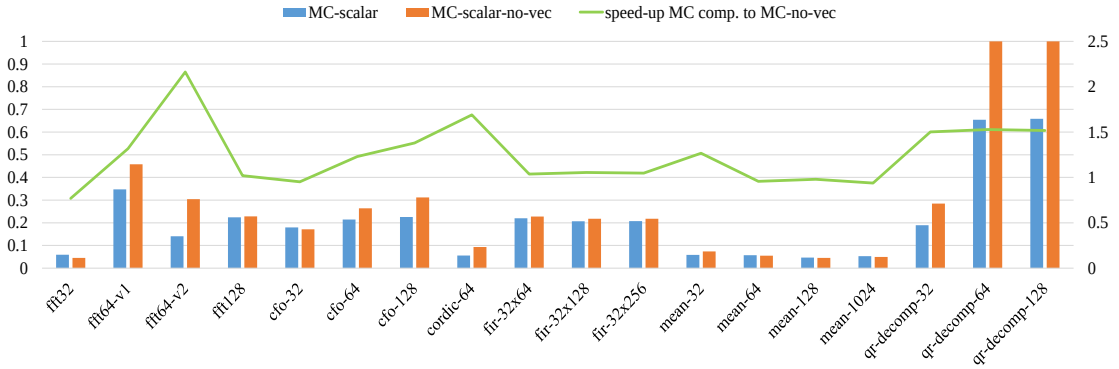

Figure 247: Performance of compiler's scalarized fixed point code compared to compiler's non-vectorized code compiling with MSVC on desktop with i7-3820

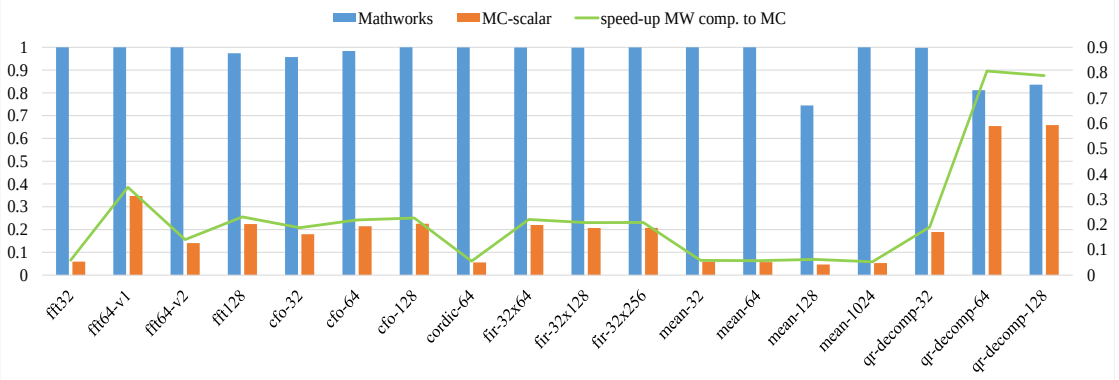

Figure 248: Performance of compiler's scalarized fixed point code compared to MathWorks fixed point generated code compiling with MSVC on desktop with i7-3820

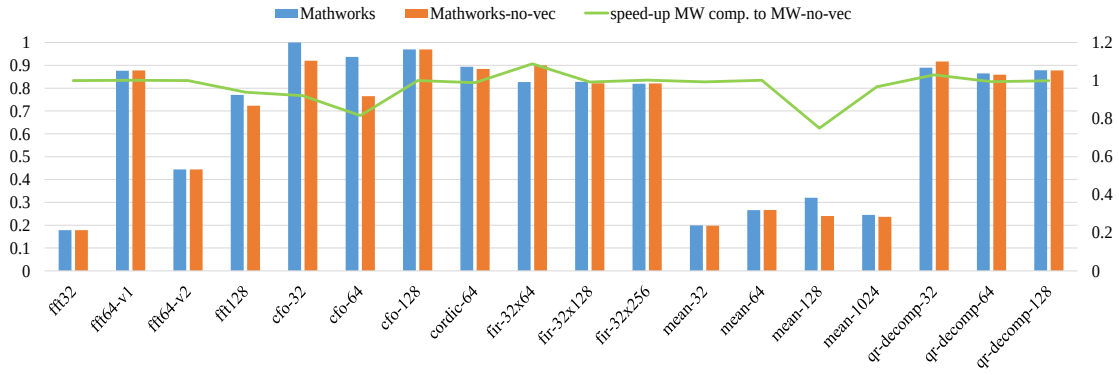

Figure 249: Performance of MathWorks floating point generated code compared to MathWorks non-vectorized code compiling with MSVC on desktop with i7-3820

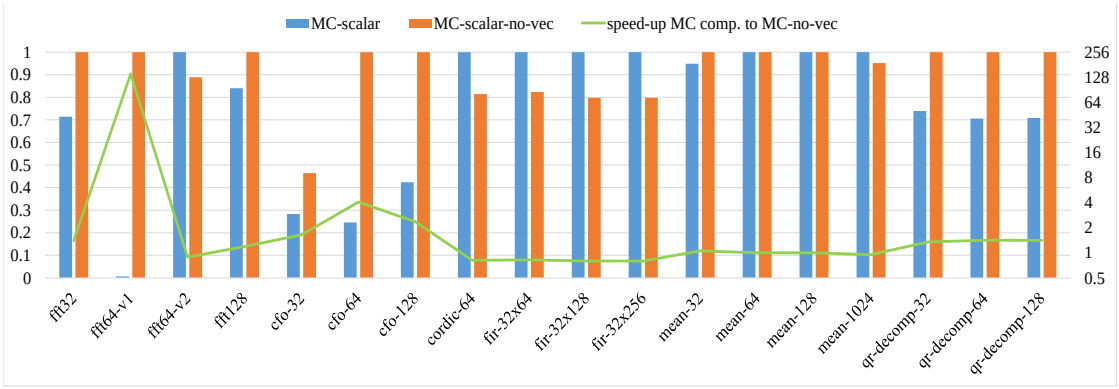

Figure 250: Performance of compiler's scalarized floating point code compared to compiler's non-vectorized code compiling with MSVC on desktop with i7-3820

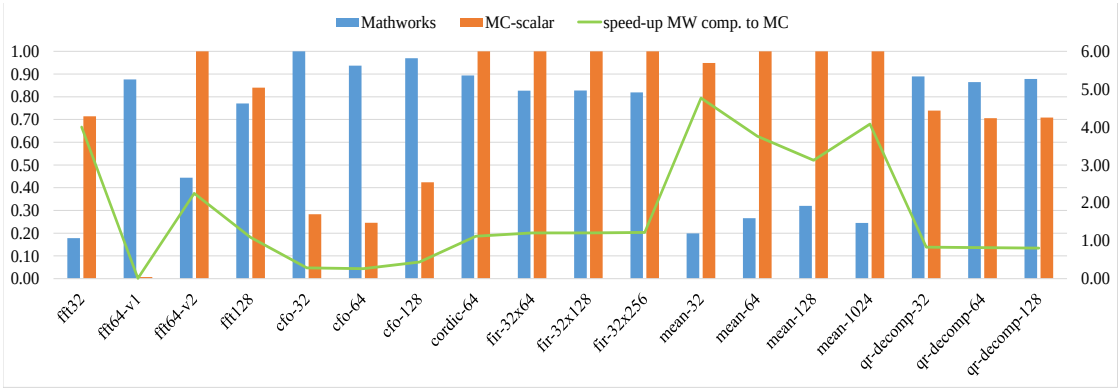

Figure 251: Performance of compiler's scalarized floating point code compared to MathWorks floating point generated code compiling with MSVC on desktop with i7-3820

8.6.4 Performance of MathWorks Generated Code, Compiler's Scalarized Generated Code and Non-vectorized Generated Code on Desktop with i7-3770 Processor

	Fig. 252, Fig. 253, Fig. 254	**Fig. 255, Fig. 256, Fig. 257**	**Fig. 258, Fig. 259, Fig. 260**	**Fig. 261, Fig. 262, Fig. 263**	**Fig. 264, Fig. 265, Fig. 266**	**Fig. 267, Fig. 268, Fig. 269**
fft32	8.33	0.13	8.68	0.13	3.50	0.30
fft64-v1	4.51	1.09	3.89	1.09	3.10	195.54
fft64-v2	14.59	0.90	12.69	0.90	4.80	1.40
fft128	27.82	2.33	27.91	2.33	13.70	4.70
cfo-32	1.07	1.51	1.53	1.51	4.14	1.90
cfo-64	6.70	3.45	8.68	3.45	4.03	3.80
cfo-128	13.44	6.31	15.36	6.31	8.00	7.52
cordic-64	69.40	19.55	158.71	19.55	326.98	35.46
fir-32x64	12.23	10.01	24.87	10.01	85.03	15.22
fir-32x128	22.11	22.75	49.88	22.75	190.35	33.34
fir-32x256	37.27	38.26	100.46	38.26	395.15	69.50
mean-32	0.07	0.04	0.11	0.04	0.10	0.11
mean-64	0.09	0.08	0.10	0.08	0.20	0.20
mean-128	0.38	0.21	0.40	0.21	0.50	0.40
mean-1024	2.42	1.59	2.31	1.59	5.33	4.14
qr-dec-32	181.55	12.47	189.53	12.47	126.48	18.34
qr-dec-64	50.05	25.26	44.66	25.26	71.50	36.62
qr-dec-128	87.38	42.23	78.12	42.23	142.78	73.20

Table 53: Reference values (exec. time in µs) used for normalization of results on desktop with i7-3770 processor

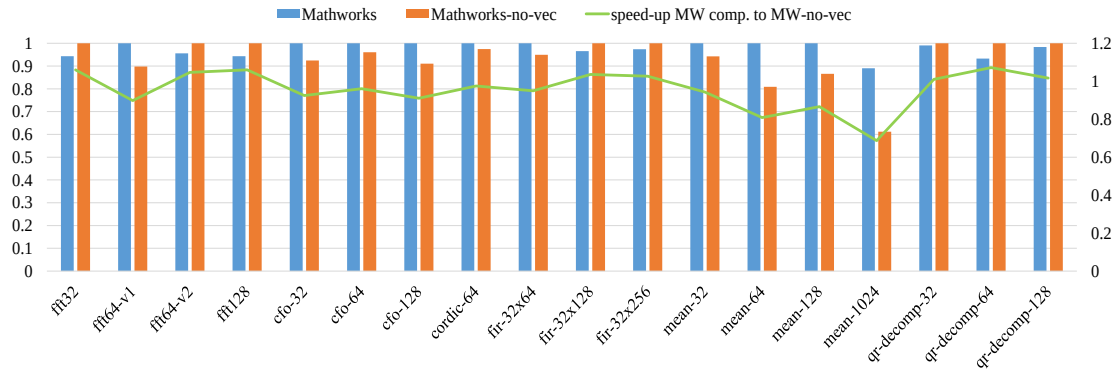

Figure 252: Performance of MathWorks fixed point generated code compared to MathWorks non-vectorized code compiling with Clang on desktop with i7-3770

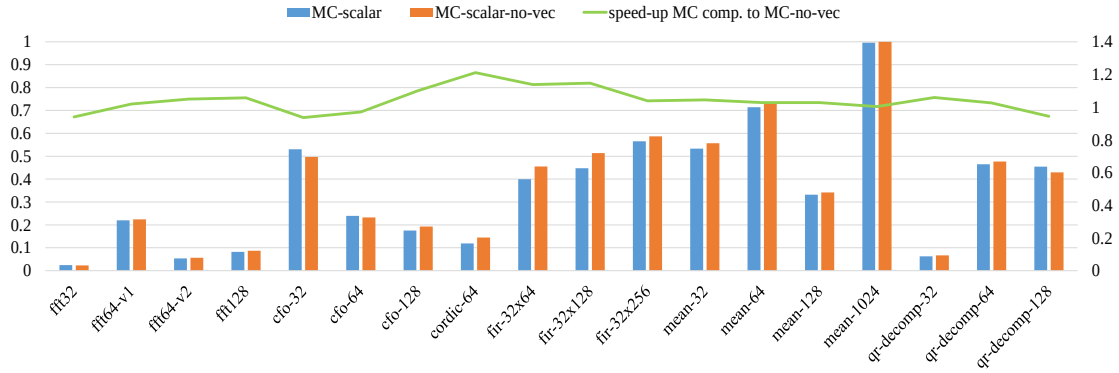

Figure 253: Performance of compiler's scalarized fixed point code compared to compiler's non-vectorized code compiling with Clang on desktop with i7-3770

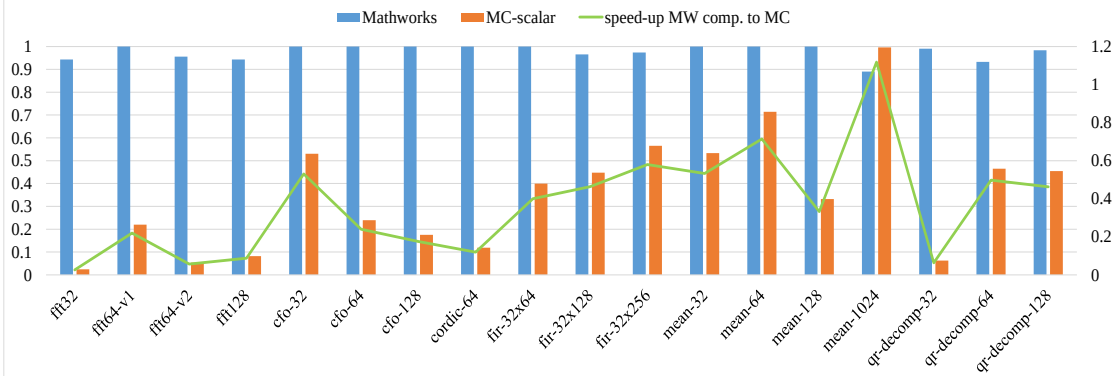

Figure 254: Performance of compiler's scalarized fixed point code compared to MathWorks fixed point generated code compiling with Clang on desktop with i7-3770

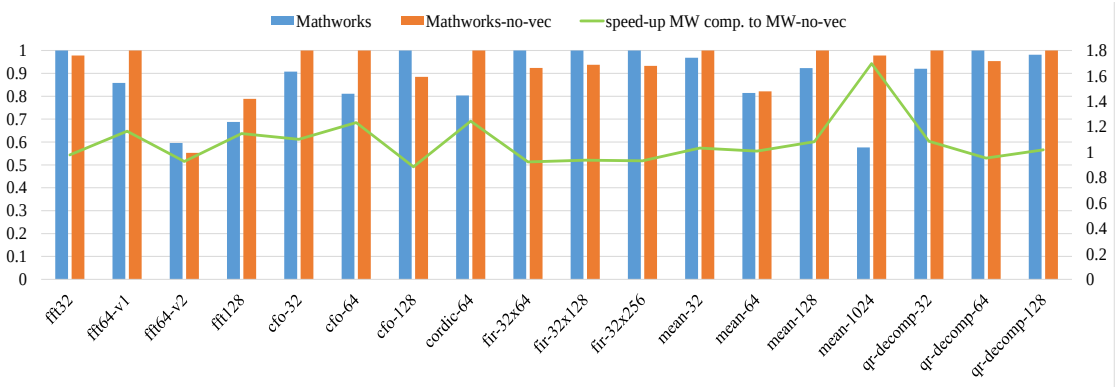

Figure 255: Performance of MathWorks floating point generated code compared to MathWorks non-vectorized code compiling with Clang on desktop with i7-3770

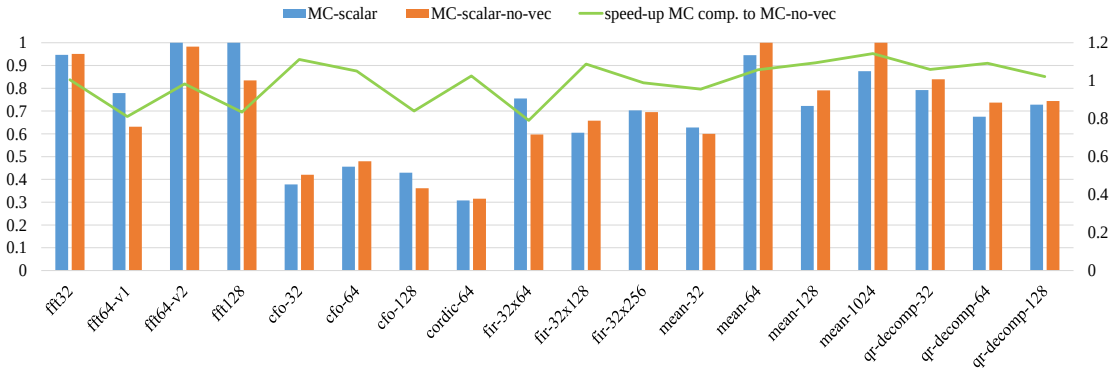

Figure 256: Performance of compiler's scalarized floating point code compared to compiler's non-vectorized code compiling with Clang on desktop with i7-3770

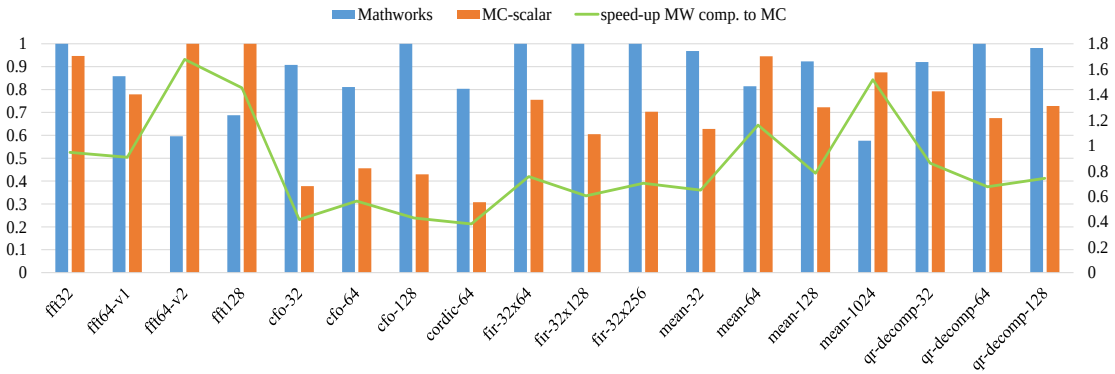

Figure 257: Performance of compiler's scalarized floating point code compared to MathWorks floating point generated code compiling with Clang on desktop with i7-3770

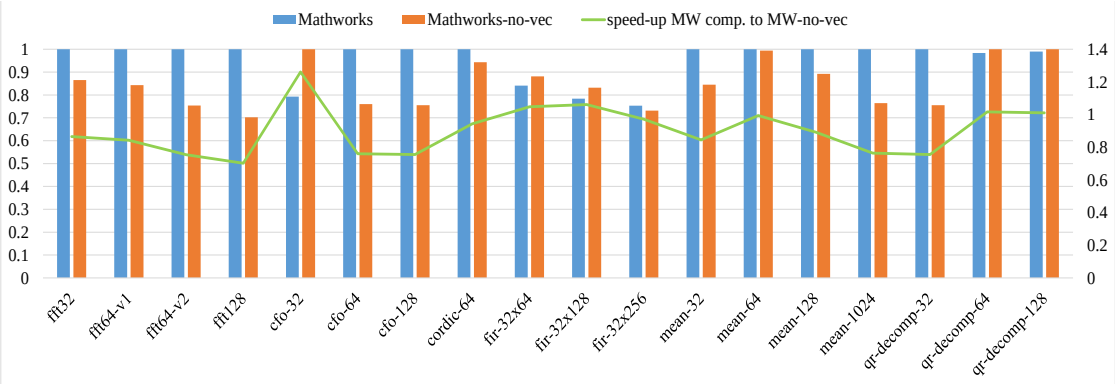

Figure 258: Performance of MathWorks fixed point generated code compared to MathWorks non-vectorized code compiling with GCC on desktop with i7-3770

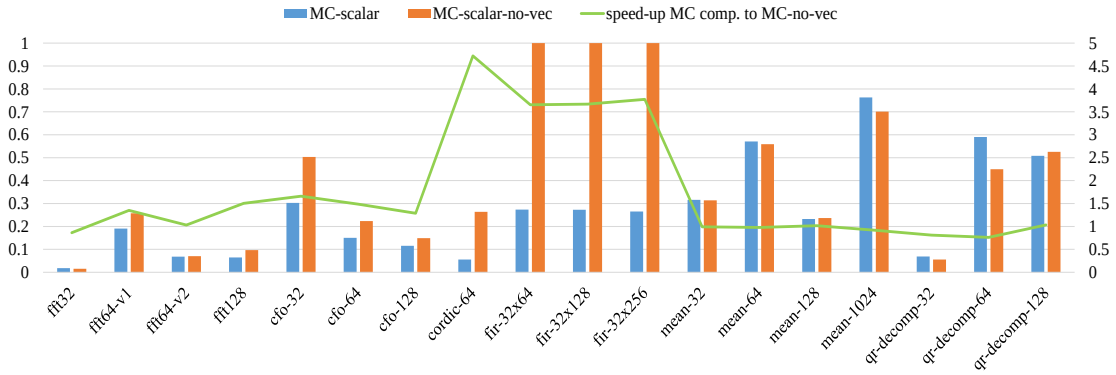

Figure 259: Performance of compiler's scalarized fixed point code compared to compiler's non-vectorized code compiling with GCC on desktop with i7-3770

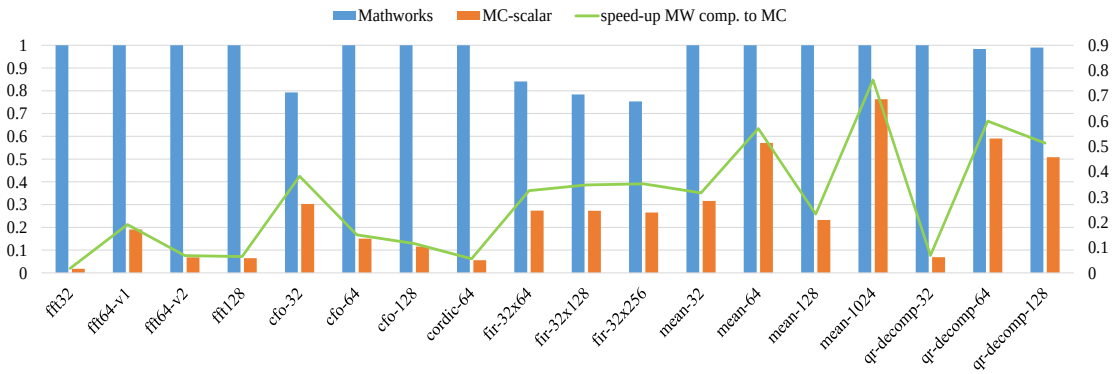

Figure 260: Performance of compiler's scalarized fixed point code compared to MathWorks fixed point generated code compiling with GCC on desktop with i7-3770

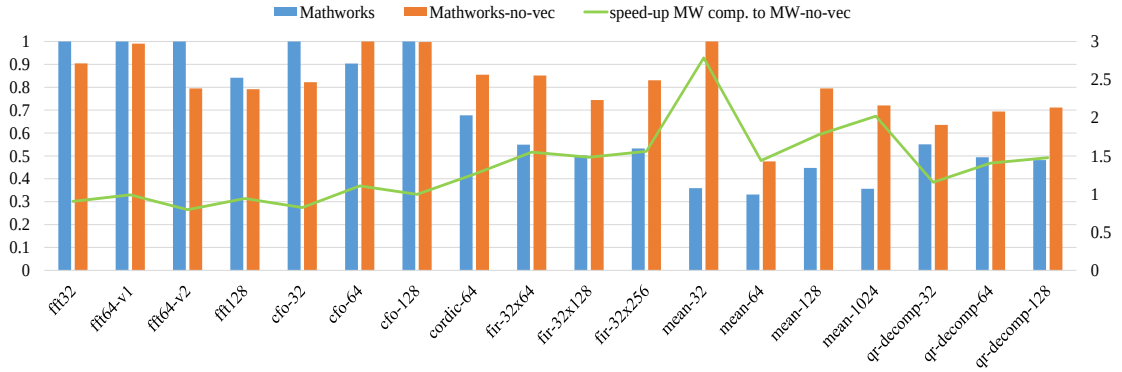

Figure 261: Performance of MathWorks floating point generated code compared to MathWorks non-vectorized code compiling with GCC on desktop with i7-3770

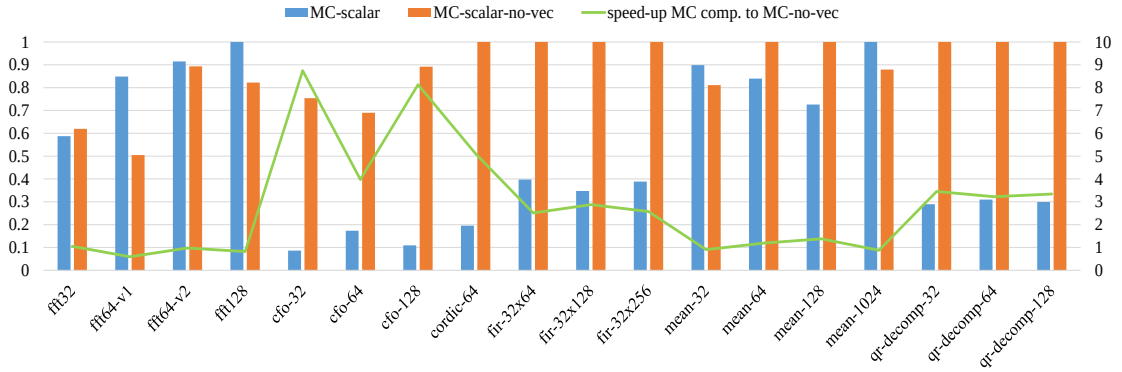

Figure 262: Performance of compiler's scalarized floating point code compared to compiler's non-vectorized code compiling with GCC on desktop with i7-3770

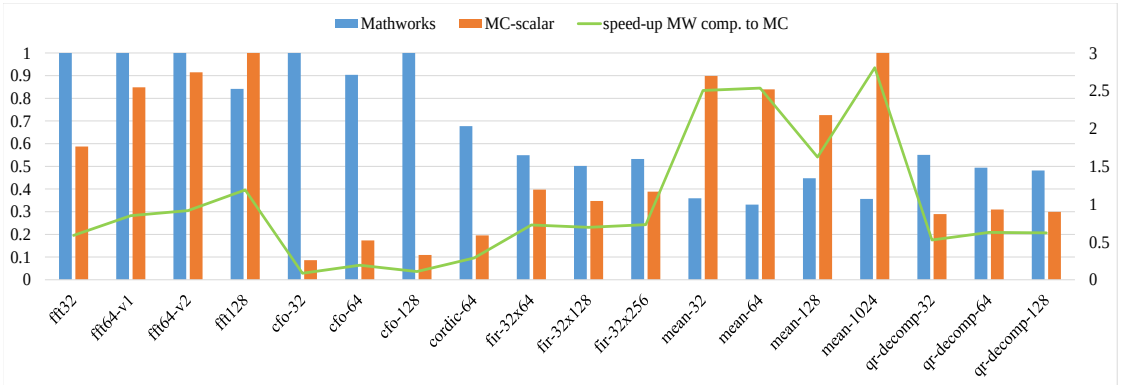

Figure 263: Performance of compiler's scalarized floating point code compared to MathWorks floating point generated code compiling with GCC on desktop with i7-3770

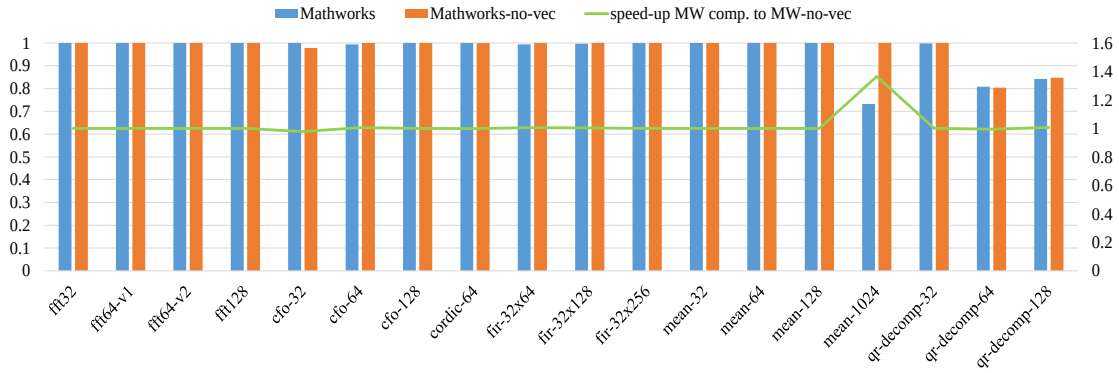

Figure 264: Performance of MathWorks fixed point generated code compared to MathWorks non-vectorized code compiling with MSVC on desktop with i7-3770

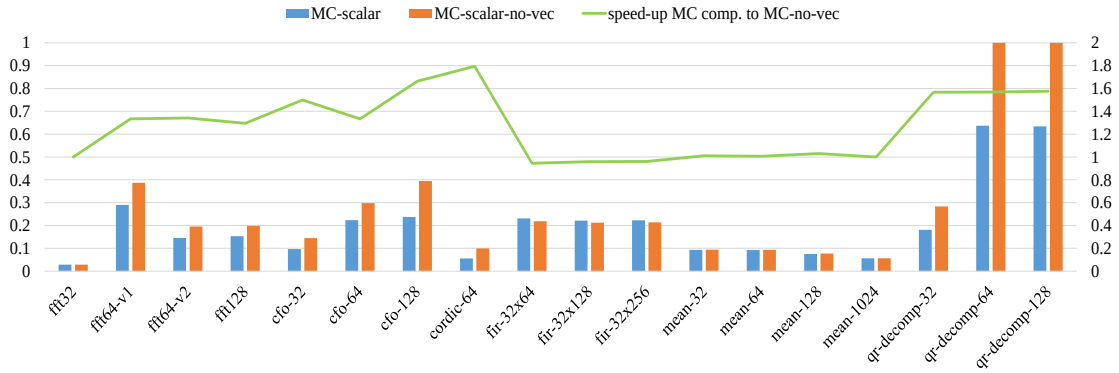

Figure 265: Performance of compiler's scalarized fixed point code compared to compiler's non-vectorized code compiling with MSVC on desktop with i7-3770

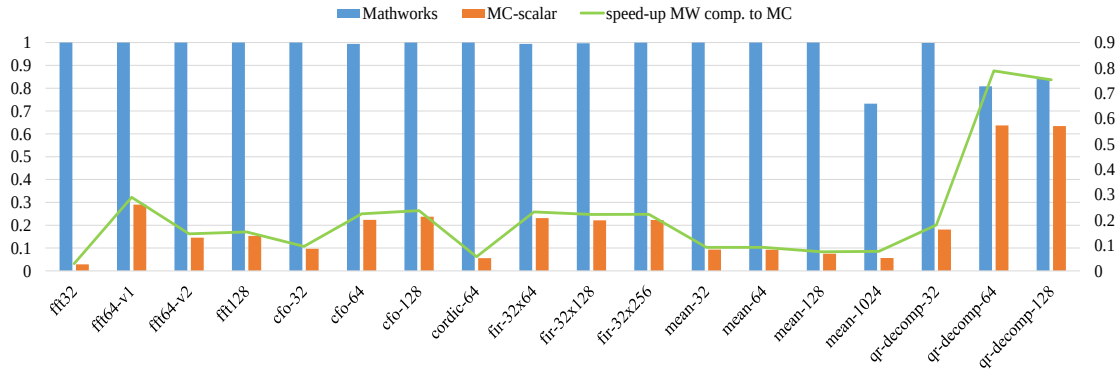

Figure 266: Performance of compiler's scalarized fixed point code compared to MathWorks fixed point generated code compiling with MSVC on desktop with i7-3770

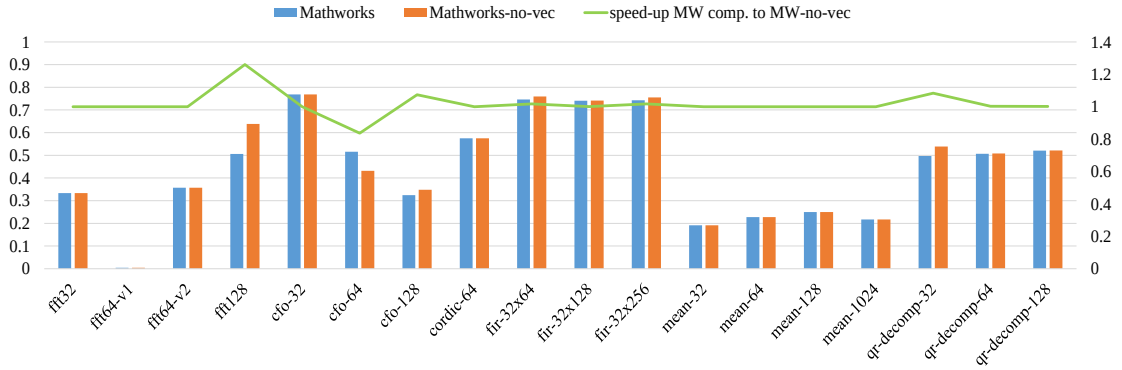

Figure 267: Performance of MathWorks floating point generated code compared to MathWorks non-vectorized code compiling with MSVC on desktop with i7-3770

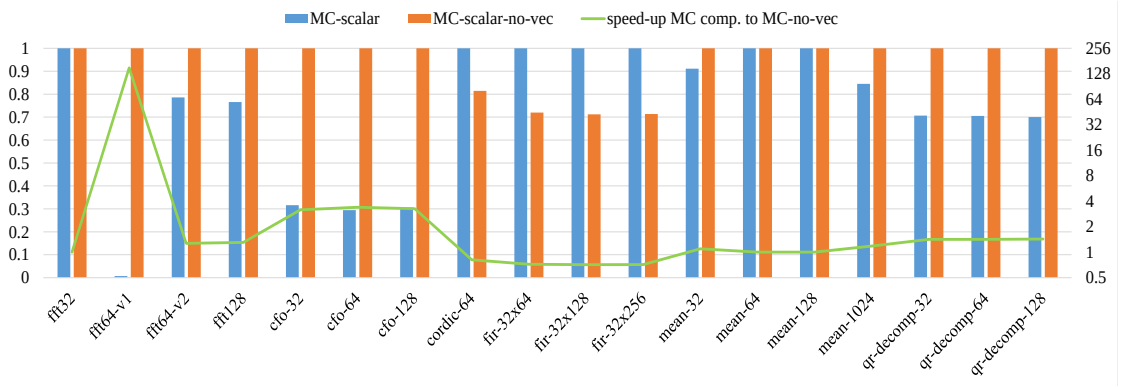

Figure 268: Performance of compiler's scalarized floating point code compared to compiler's non-vectorized code compiling with MSVC on desktop with i7-3770

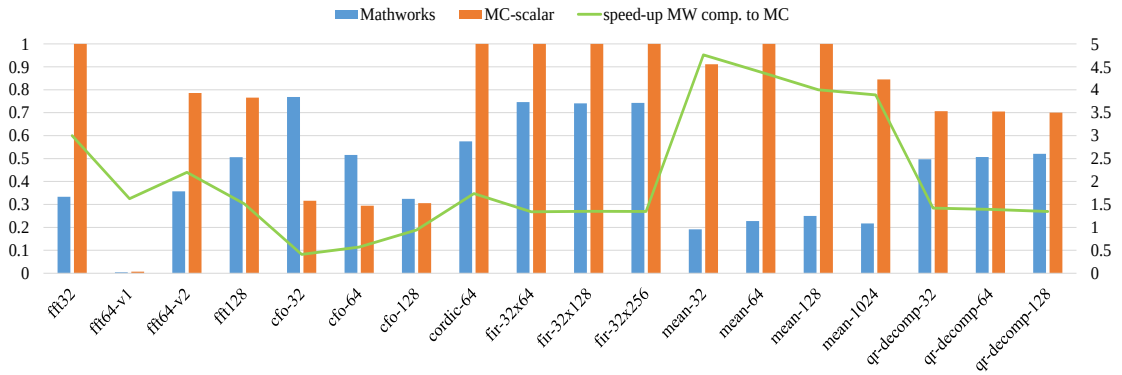

Figure 269: Performance of compiler's scalarized floating point code compared to MathWorks floating point generated code compiling with MSVC on desktop with i7-3770

8.7 Performance of Generated Code on x86 Architectures Using Alternative Compilations Options of MSVC

The section shows the performance of MathWorks generated code and compiler's vectorized generated code for x86 architectures using alternative compilation options of MSVC compiler. Subsection 8.7.1 shows results of the generated code performance producing x64 executable code, subsection 8.7.2 show results of x86 output code which have been compiled with SSE options (auto-vectorization using only SSE extension) and subsection 8.7.3 shows performance results compiling with x64 and SSE options.

8.7.1 Performance of Generated Code on x86 Architectures Producing x64 Code

	Fig. 270, Fig. 271, Fig. 272	**Fig. 273, Fig. 274, Fig. 275**	**Fig. 276, Fig. 277, Fig. 278**	**Fig. 279, Fig. 280, Fig. 281**
fft32	3.30	0.72	2.70	0.30
fft64-v1	4.80	1.98	3.42	1.10
fft64-v2	6.10	1.94	5.28	1.10
fft128	17.80	4.10	14.42	3.16
cfo-32	4.66	1.72	3.46	1.36
cfo-64	5.24	2.74	3.86	2.00
cfo-128	9.72	3.32	7.60	2.62
cordic-64	313.24	75.50	253.12	37.70
fir-32x64	107.78	15.06	90.24	12.70
fir-32x128	235.30	32.08	197.70	27.80
fir-32x256	471.46	66.78	392.42	57.90
mean-32	0.30	0.30	0.20	0.20
mean-64	0.50	0.50	0.40	0.40
mean-128	1.24	1.18	0.80	0.90
mean-1024	8.16	9.30	6.82	7.00
qr-decomp-32	139.96	33.54	117.76	23.00
qr-decomp-64	86.48	67.22	72.52	46.46
qr-decomp-128	173.66	134.62	143.74	92.88

Table 54: Reference values (exec. time in µs) used for normalization of aggressive Clang options examination

8.7.1.1 Performance of Generated Code on Intel i7-3820 Producing x64 Code

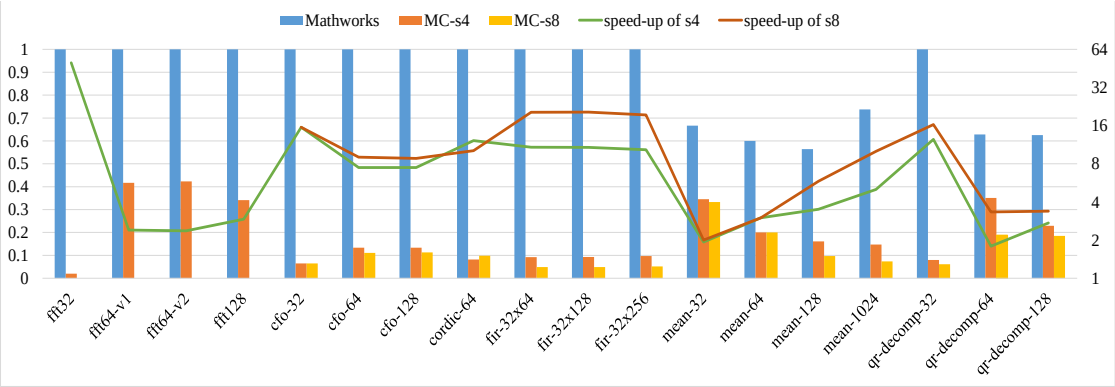

Figure 270: Performance of vectorized generated code (x64) with packed fixed point data types compared to MathWorks generated code on i7-3820

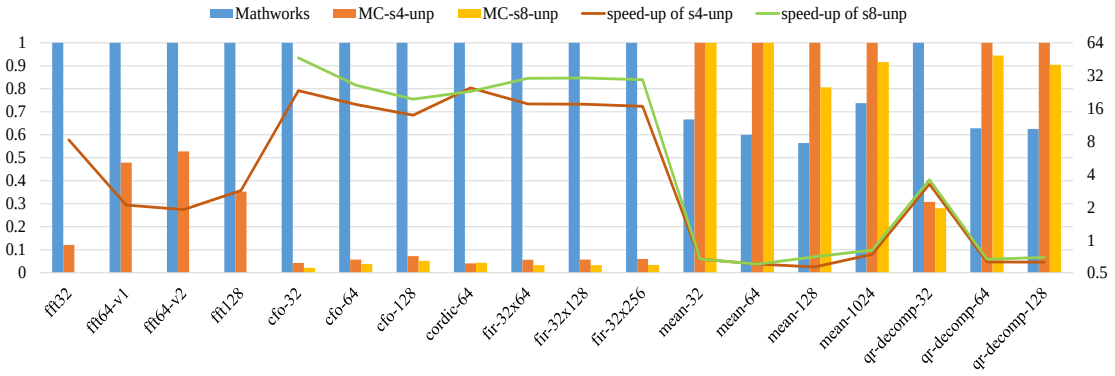

Figure 271: Performance of vectorized generated code (x64) with unpacked fixed point data types compared to MathWorks generated code on i7-3820

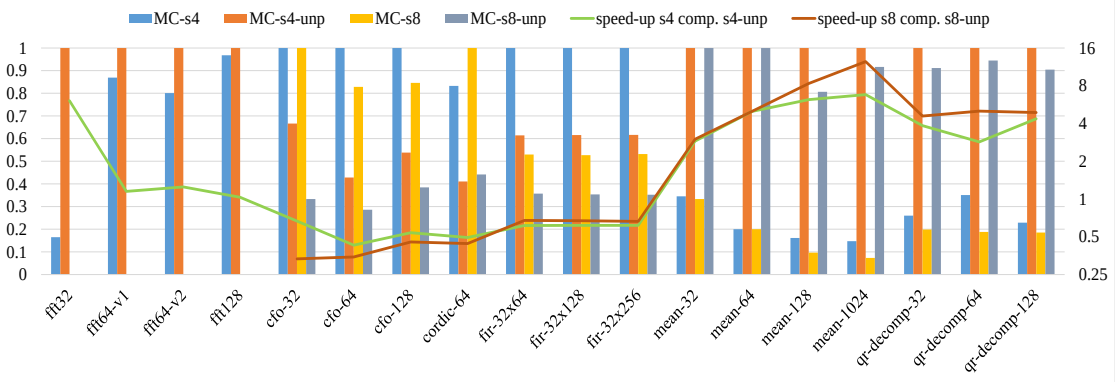

Figure 272: Performance of vectorized code (x64) with unpacked fixed point data types versus packed fixed point data types on desktop with i7-3820

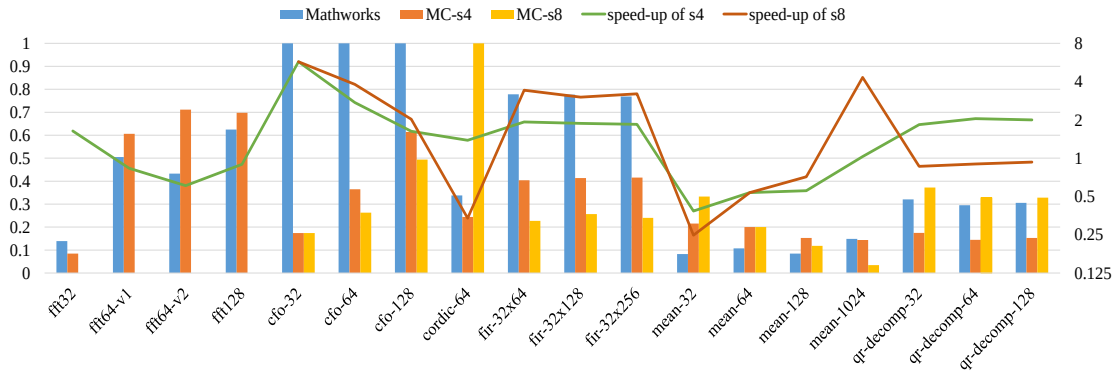

Figure 273: Performance of vectorized generated code (x64) with packed floating point data types compared to MathWorks generated code on i7-3820

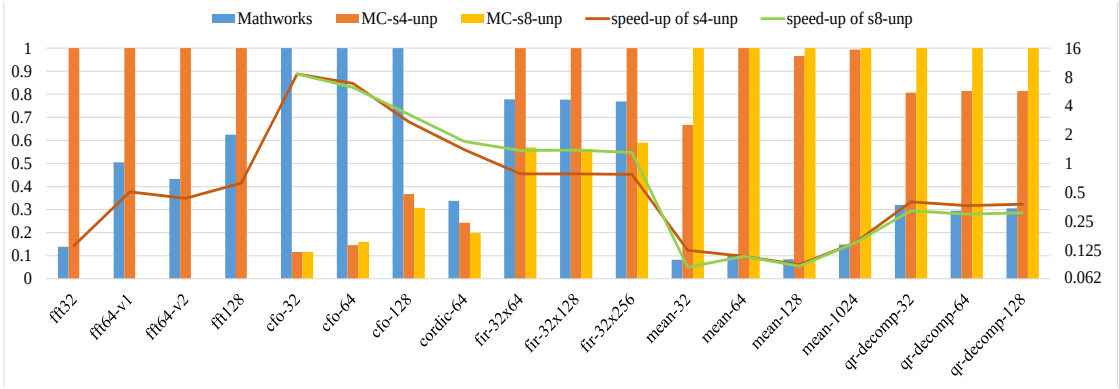

Figure 274: Performance of vectorized generated code (x64) with unpacked floating point data types compared to MathWorks generated code on i7-3820

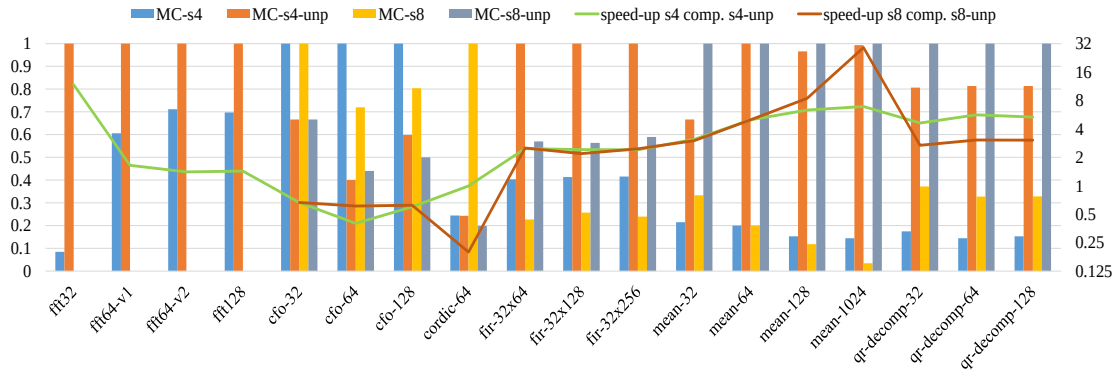

Figure 275: Performance of vectorized code (x64) with unpacked floating point data types versus packed floating point data types on desktop with i7-3820

8.7.1.2 Performance of Generated Code on Intel i7-3770 Producing x64 Code

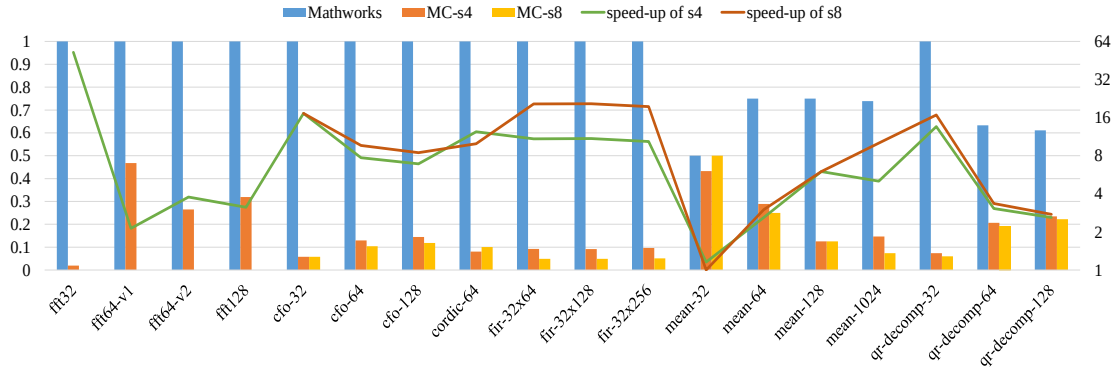

Figure 276: Performance of vectorized generated code (x64) with packed fixed point data types compared to MathWorks generated code on i7-3770

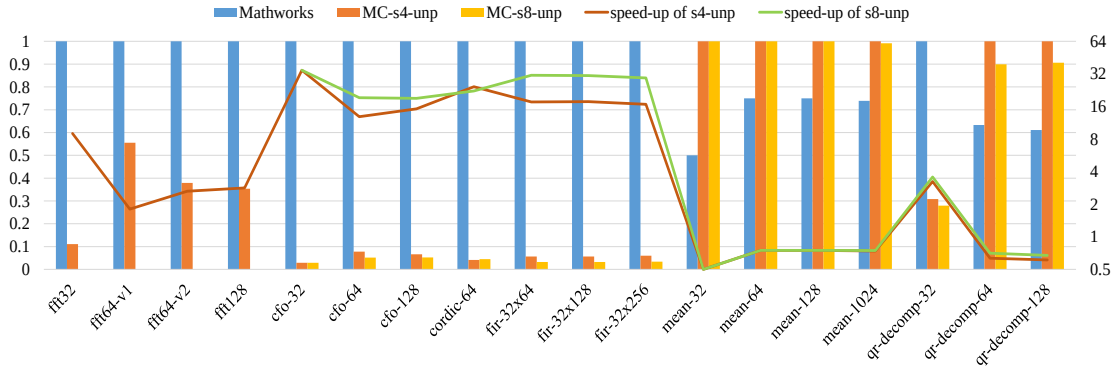

Figure 277: Performance of vectorized generated code (x64) with unpacked fixed point data types compared to MathWorks generated code on i7-3770

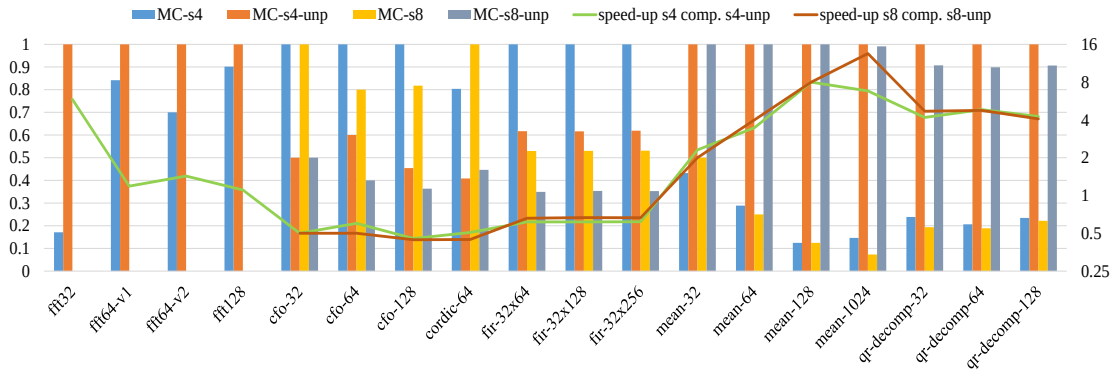

Figure 278: Performance of vectorized code (x64) with unpacked fixed point data types versus packed fixed point data types on desktop with i7-3770

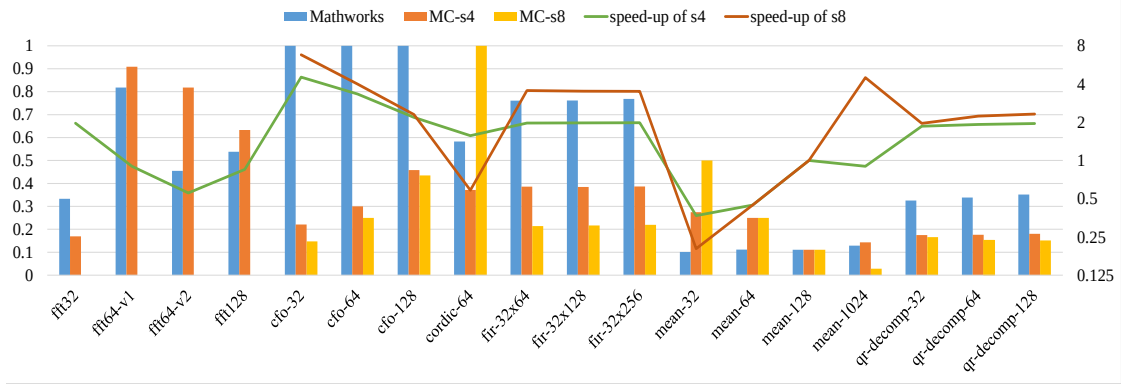

Figure 279: Performance of vectorized generated code (x64) with packed floating point data types compared to MathWorks generated code on i7-3770

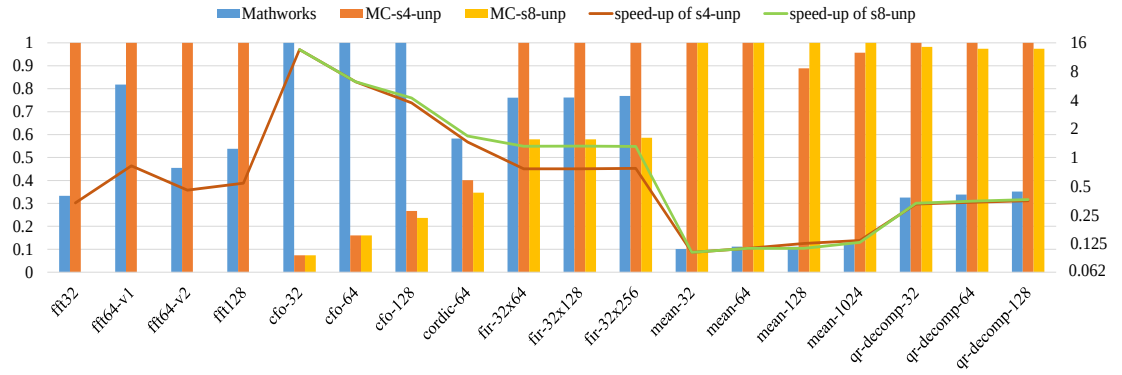

Figure 280: Performance of vectorized generated code (x64) with unpacked floating point data types compared to MathWorks generated code on i7-3770

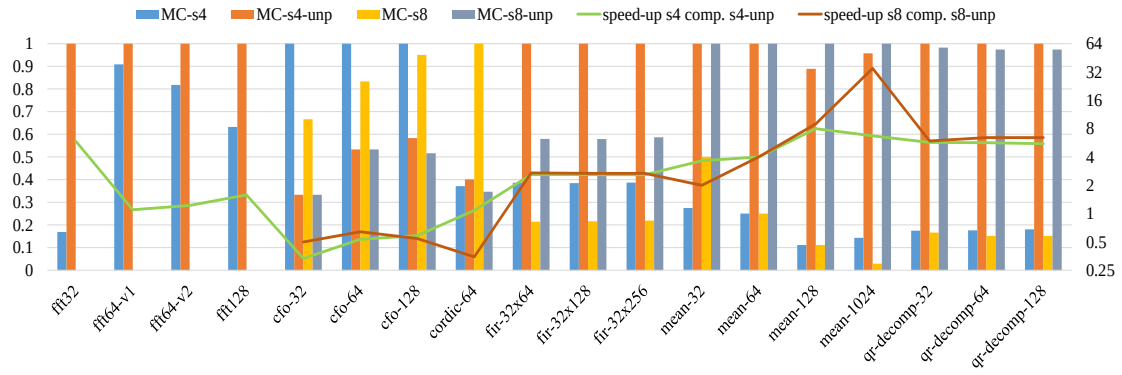

Figure 281: Performance of vectorized code (x64) with unpacked floating point data types versus packed floating point data types on desktop with i7-3770

8.7.2 Performance of Generated Code on x86 Architectures Producing x86 Code with SSE

	Fig. 282, Fig. 283, Fig. 284	**Fig. 285, Fig. 286, Fig. 287**	**Fig. 288, Fig. 289, Fig. 290**	**Fig. 291, Fig. 292, Fig. 293**
fft32	4.40	0.40	3.50	0.30
fft64-v1	3.84	1.20	3.10	1.10
fft64-v2	5.76	2.00	4.80	1.30
fft128	17.46	4.18	13.70	2.90
cfo-32	4.80	1.66	3.78	1.42
cfo-64	5.12	2.70	4.06	1.80
cfo-128	10.26	3.70	8.00	2.32
cordic-64	406.82	45.40	326.68	37.38
fir-32x64	104.34	14.96	84.52	12.30
fir-32x128	236.12	31.14	189.78	26.90
fir-32x256	492.16	64.90	395.08	56.10
mean-32	0.20	0.30	0.20	0.20
mean-64	0.50	0.60	0.40	0.40
mean-128	1.02	1.10	0.90	0.90
mean-1024	8.10	9.26	7.34	7.40
qr-decomp-32	151.34	31.78	126.60	26.04
qr-decomp-64	78.56	63.34	70.00	52.52
qr-decomp-128	157.34	126.84	139.88	104.96

Table 55: Reference values (exec. time in µs) used for normalization of aggressive Clang options examination

8.7.2.1 Performance of Generated Code on Intel i7-3820 Producing x86 with SSE Extension

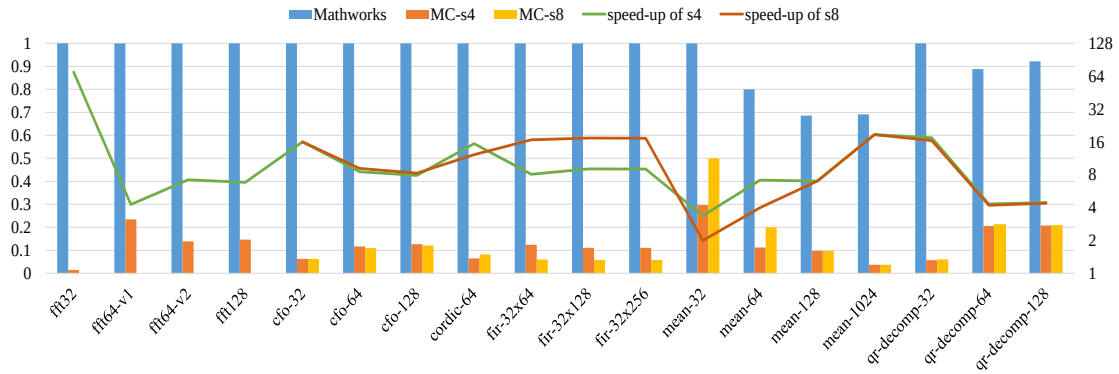

Figure 282: Performance of vectorized generated code (x86-SSE) with packed fixed point data types compared to MathWorks generated code on i7-3820

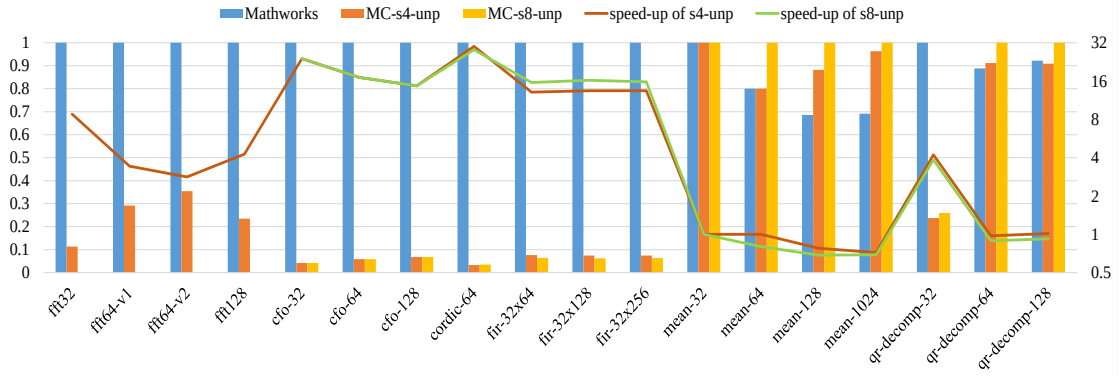

Figure 283: Performance of vectorized generated code (x86-SSE) with unpacked fixed point data types compared to MathWorks generated code on i7-3820

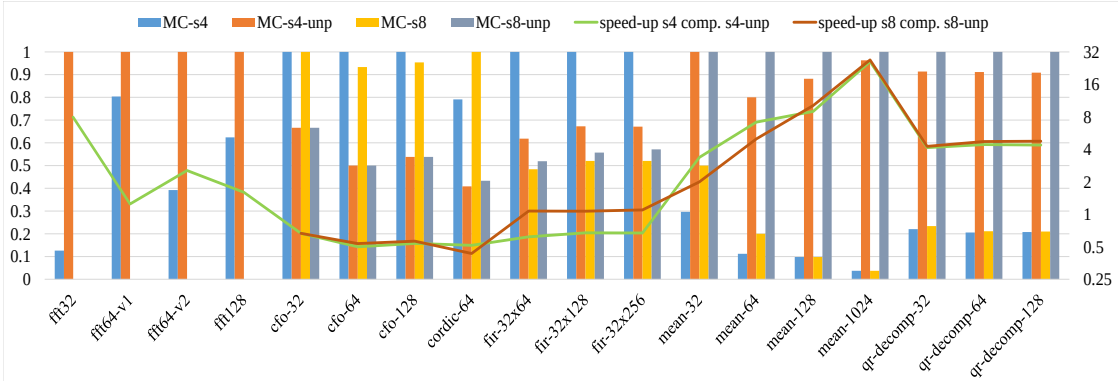

Figure 284: Performance of vectorized code (x86-SSE) with unpacked fixed point data types versus packed fixed point data types on desktop with i7-3820

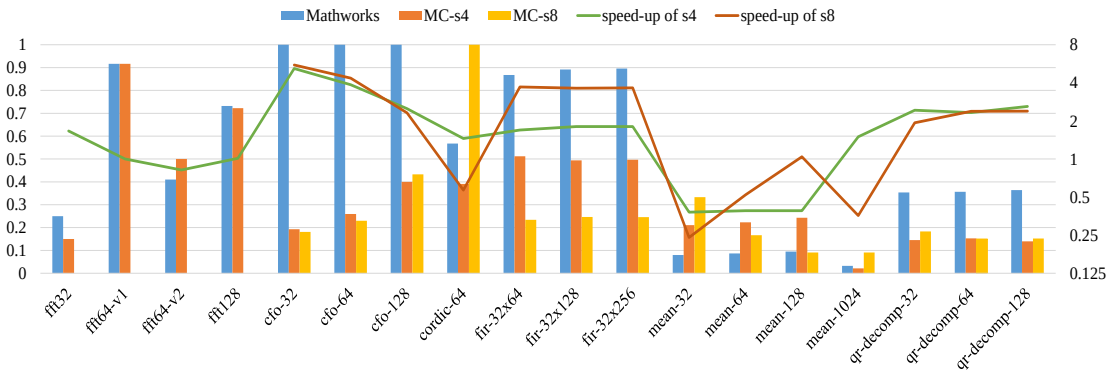

Figure 285: Performance of vectorized generated code (x86-SSE) with packed floating point data types compared to MathWorks generated code on i7-3820

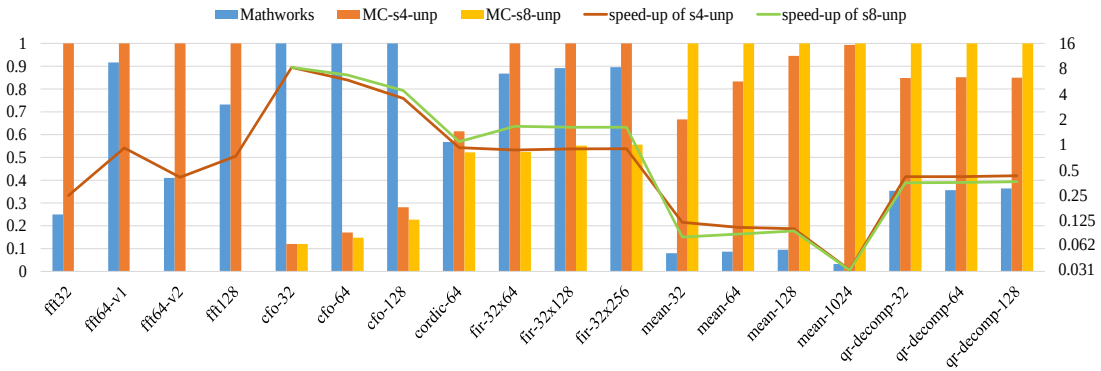

Figure 286: Performance of vectorized generated code (x86-SSE) with unpacked floating point data types compared to MathWorks generated code on i7-3820

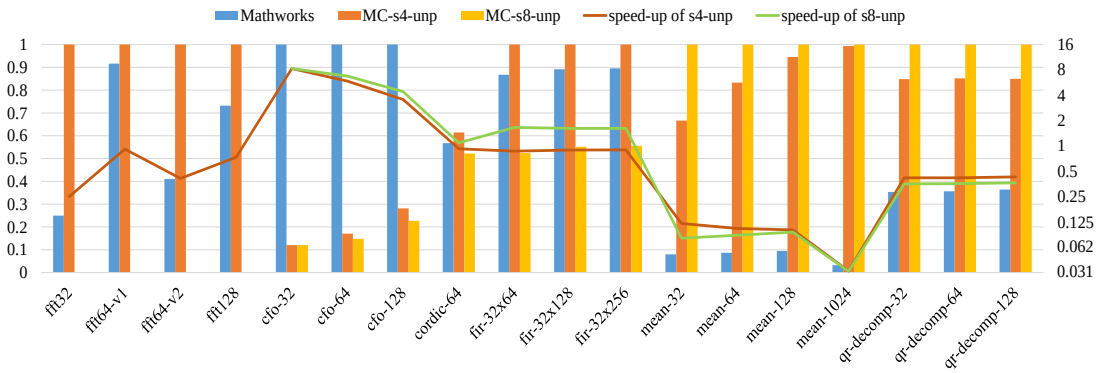

Figure 287: Performance of vectorized code (x86-SSE) with unpacked floating point data types versus packed floating point data types on desktop with i7-3820

8.7.2.2 Performance of Generated Code on Intel i7-3770 Producing x86 with SSE Extension

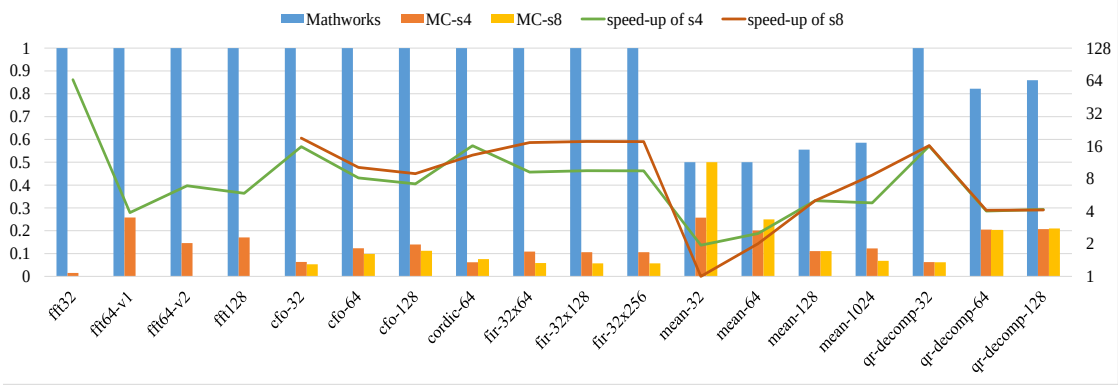

Figure 288: Performance of vectorized generated code (x86-SSE) with packed fixed point data types compared to MathWorks generated code on i7-3770

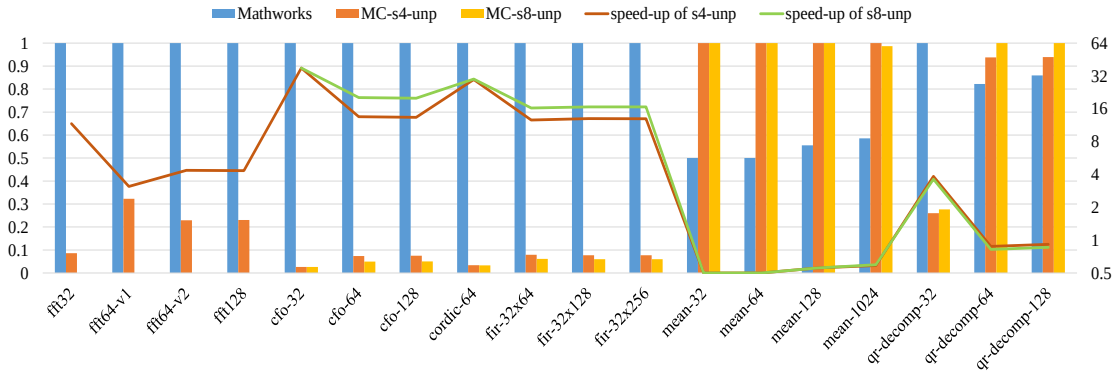

Figure 289: Performance of vectorized generated code (x86-SSE) with unpacked fixed point data types compared to MathWorks generated code on i7-3770

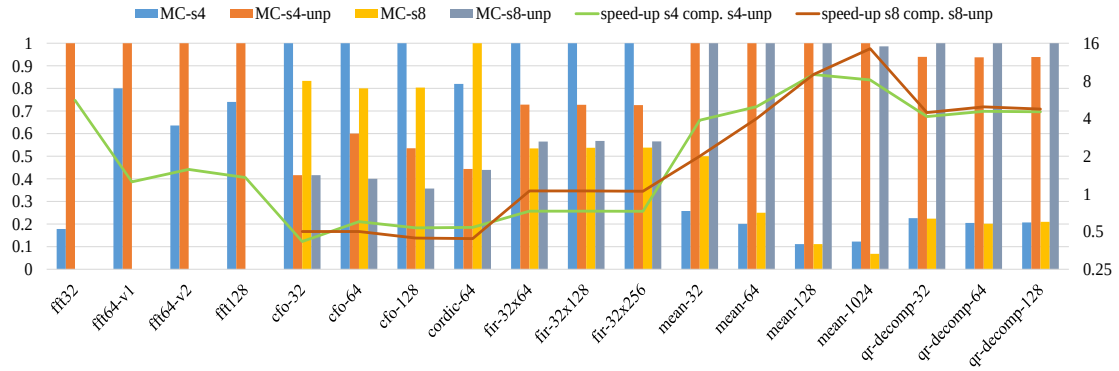

Figure 290: Performance of vectorized code (x86-SSE) with unpacked fixed point data types versus packed fixed point data types on desktop with i7-3770

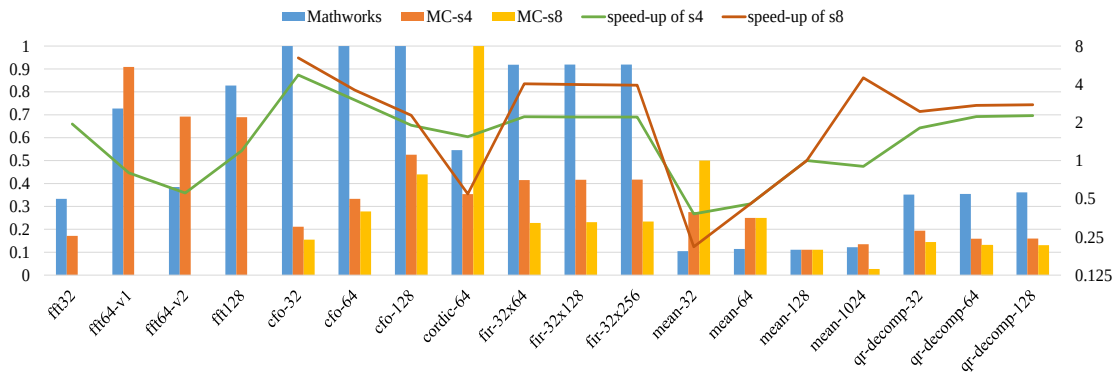

Figure 291: Performance of vectorized generated code (x86-SSE) with packed floating point data types compared to MathWorks generated code on i7-3770

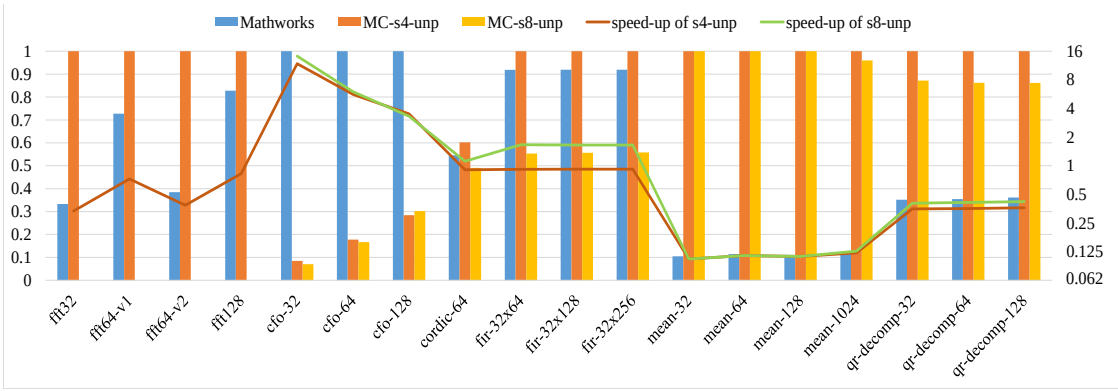

Figure 292: Performance of vectorized generated code (x86-SSE) with unpacked floating point data types compared to MathWorks generated code on i7-3770

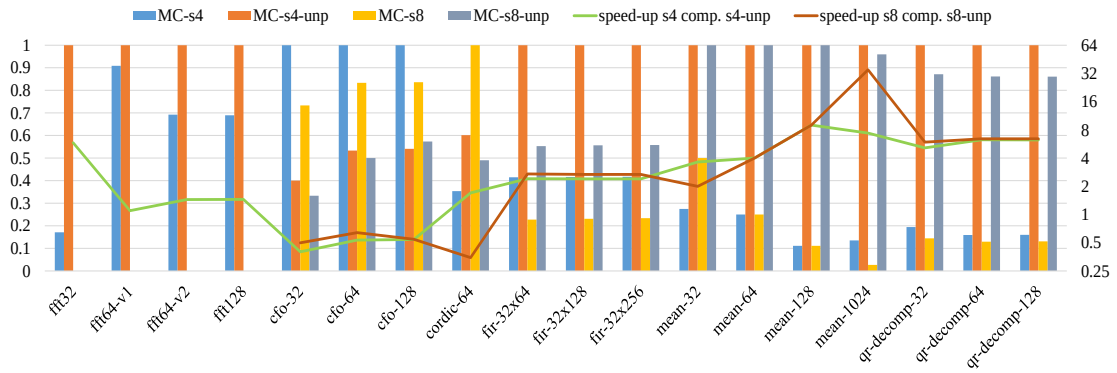

Figure 293: Performance of vectorized code (x86-SSE) with unpacked floating point data types versus packed floating point data types on desktop with i7-3770

8.7.3 Performance of Generated Code on x86 architectures Producing x64 Code with SSE

	Fig. 294, Fig. 295, Fig. 296	**Fig. 297, Fig. 298, Fig. 299**	**Fig. 300, Fig. 301, Fig. 302**	**Fig. 303, Fig. 304, Fig. 305**
fft32	4.40	0.40	3.50	0.30
fft64-v1	3.84	1.20	3.10	1.10
fft64-v2	5.76	2.00	4.80	1.30
fft128	17.46	4.18	13.70	2.90
cfo-32	4.80	1.66	3.78	1.42
cfo-64	5.12	2.70	4.06	1.80
cfo-128	10.26	3.70	8.00	2.32
cordic-64	406.82	45.40	326.68	37.38
fir-32x64	104.34	14.96	84.52	12.30
fir-32x128	236.12	31.14	189.78	26.90
fir-32x256	492.16	64.90	395.08	56.10
mean-32	0.20	0.30	0.20	0.20
mean-64	0.50	0.60	0.40	0.40
mean-128	1.02	1.10	0.90	0.90
mean-1024	8.10	9.26	7.34	7.40
qr-decomp-32	151.34	31.78	126.60	26.04
qr-decomp-64	78.56	63.34	70.00	52.52
qr-decomp-128	157.34	126.84	139.88	104.96

Table 56: Reference values (exec. time in µs) used for normalization of aggressive Clang options examination

8.7.3.1 Performance of Generated Code on Intel i7-3820 Producing x64 with SSE Extension

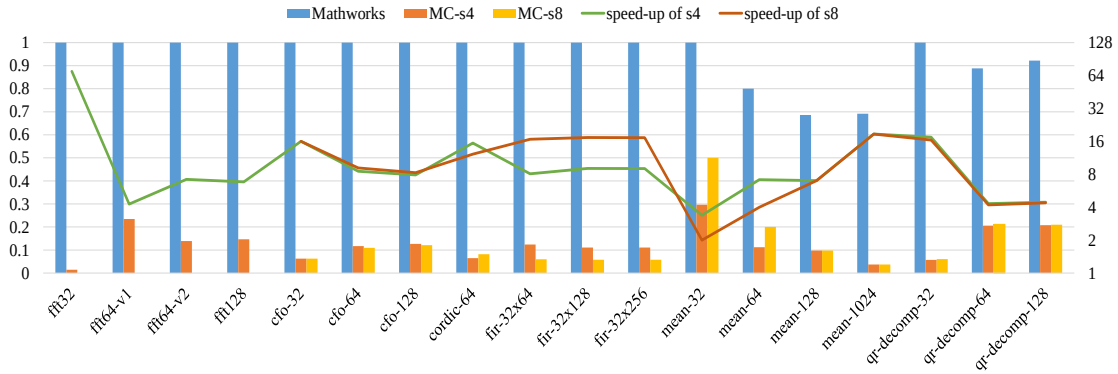

Figure 294: Performance of vectorized generated code (x64-SSE) with packed fixed point data types compared to MathWorks generated code on i7-3820

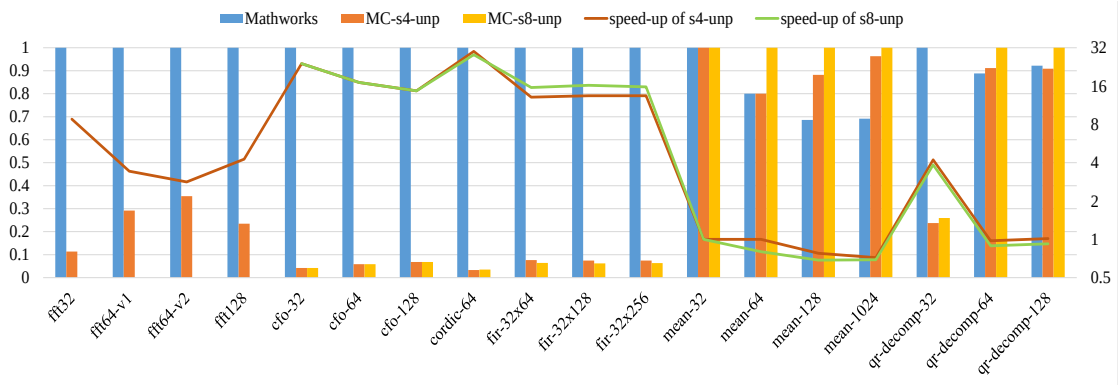

Figure 295: Performance of vectorized generated code (x64-SSE) with unpacked fixed point data types compared to MathWorks generated code on i7-3820

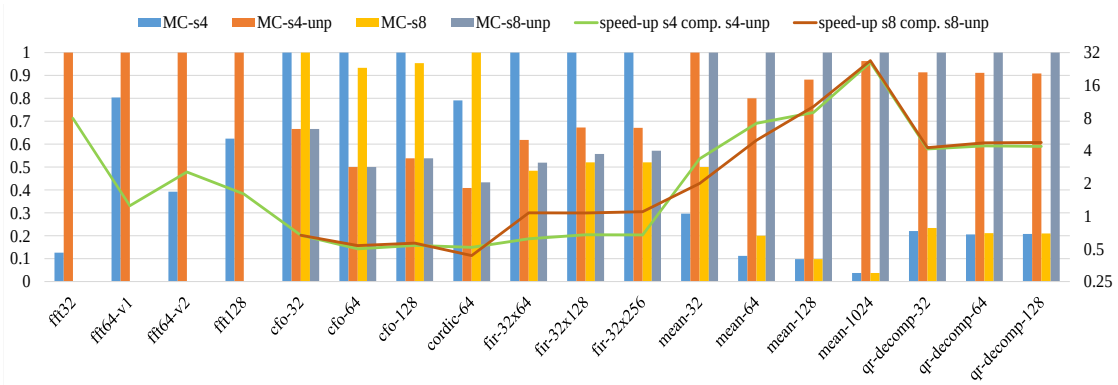

Figure 296: Performance of vectorized code (x64-SSE) with unpacked fixed point data types versus packed fixed point data types on desktop with i7-3820

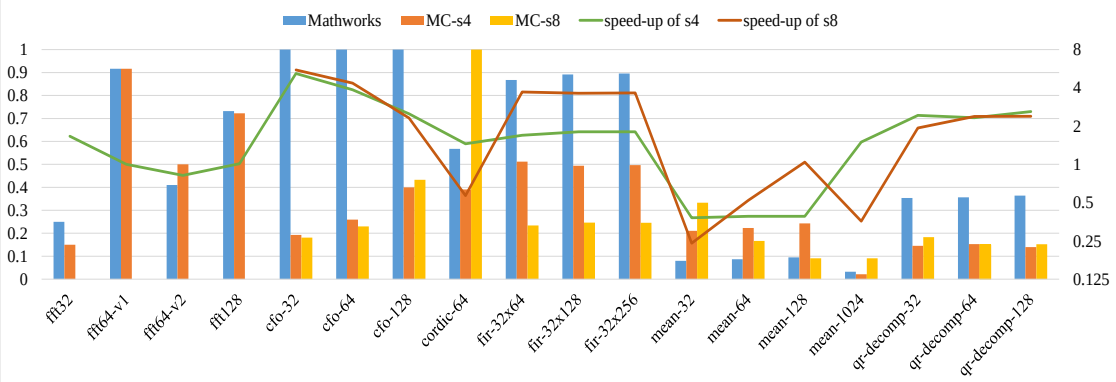

Figure 297: Performance of vectorized generated code (x64-SSE) with packed floating point data types compared to MathWorks generated code on i7-3820

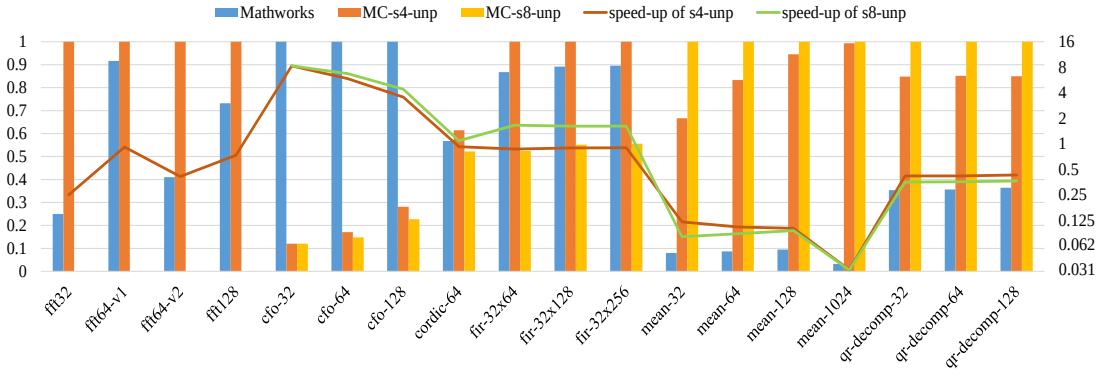

Figure 298: Performance of vectorized generated code (x64-SSE) with unpacked floating point data types compared to MathWorks generated code on i7-3820

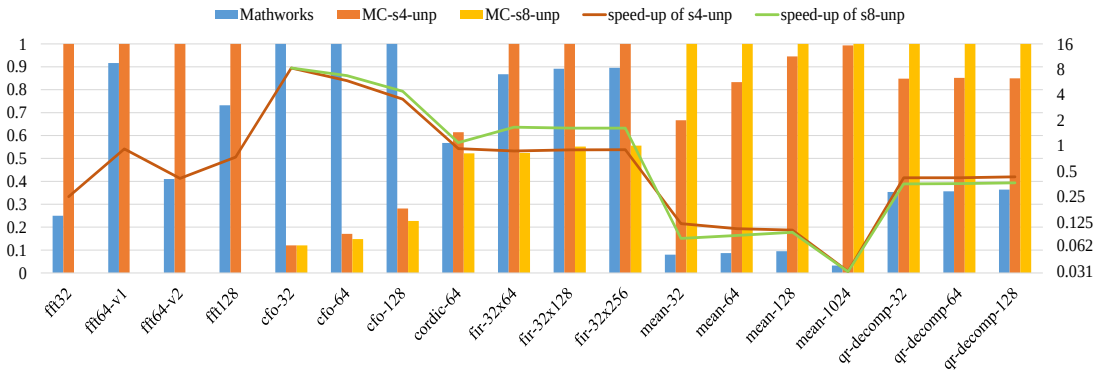

Figure 299: Performance of vectorized code (x64-SSE) with unpacked floating point data types versus packed floating point data types on desktop with i7-3820

8.7.3.2 Performance of Generated Code on Intel i7-3770 Producing x64 with SSE Extension

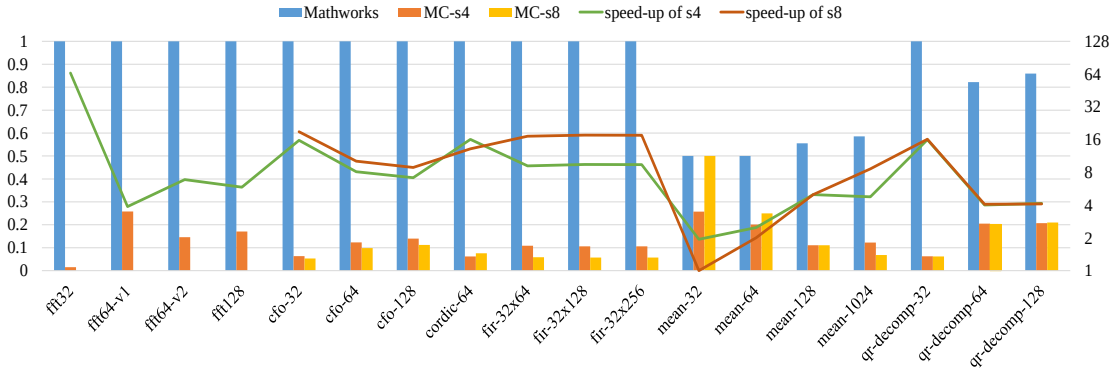

Figure 300: Performance of vectorized generated code (x64-SSE) with packed fixed point data types compared to MathWorks generated code on i7-3770

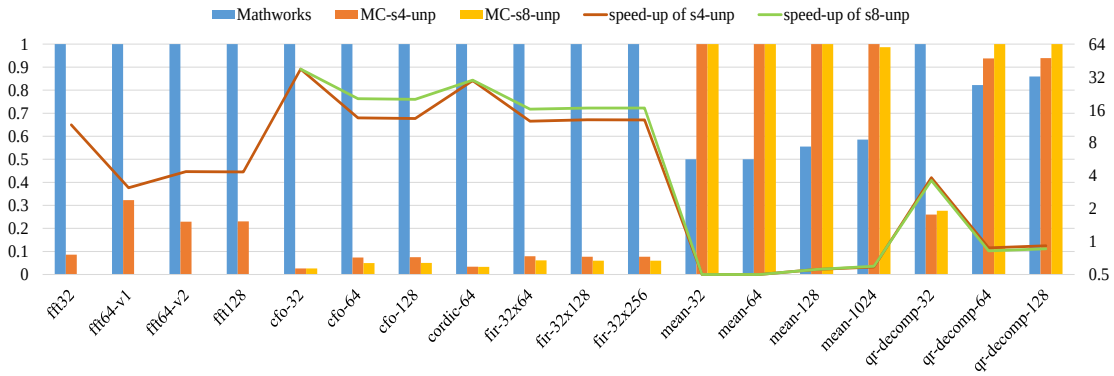

Figure 301: Performance of vectorized generated code (x64-SSE) with unpacked fixed point data types compared to MathWorks generated code on i7-3770

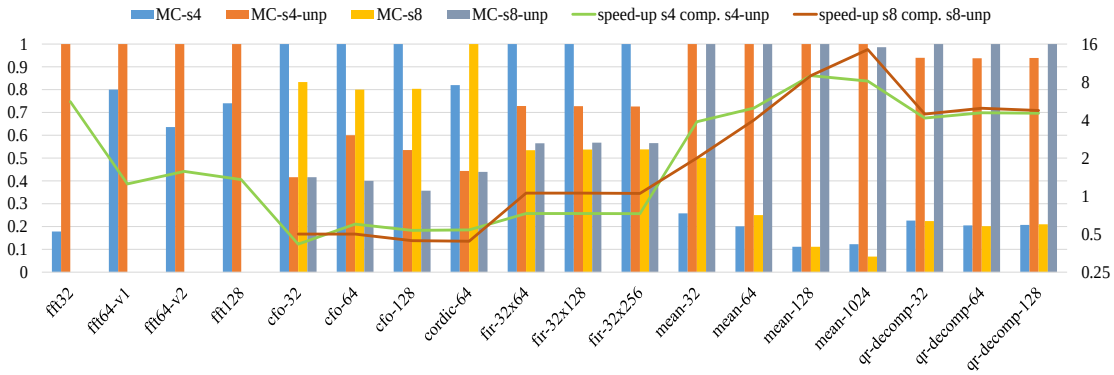

Figure 302: Performance of vectorized code (x64-SSE) with unpacked fixed point data types versus packed fixed point data types on desktop with i7-3770

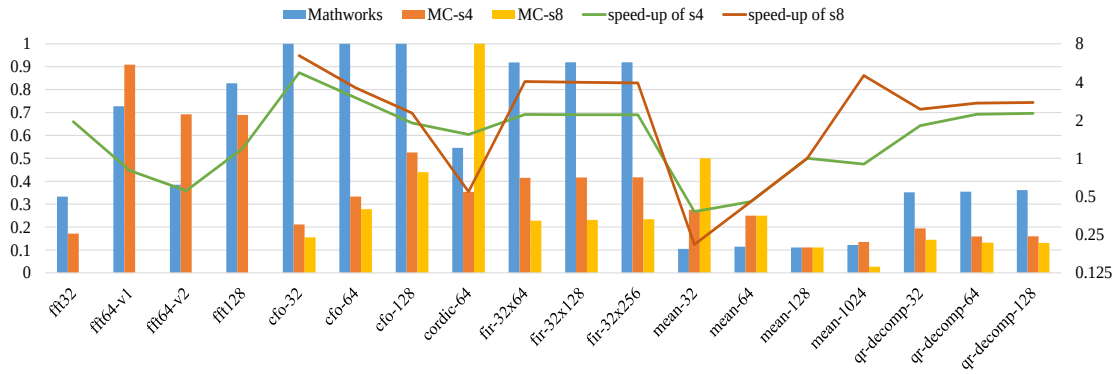

Figure 303: Performance of vectorized generated code (x64-SSE) with packed floating point data types compared to MathWorks generated code on i7-3770

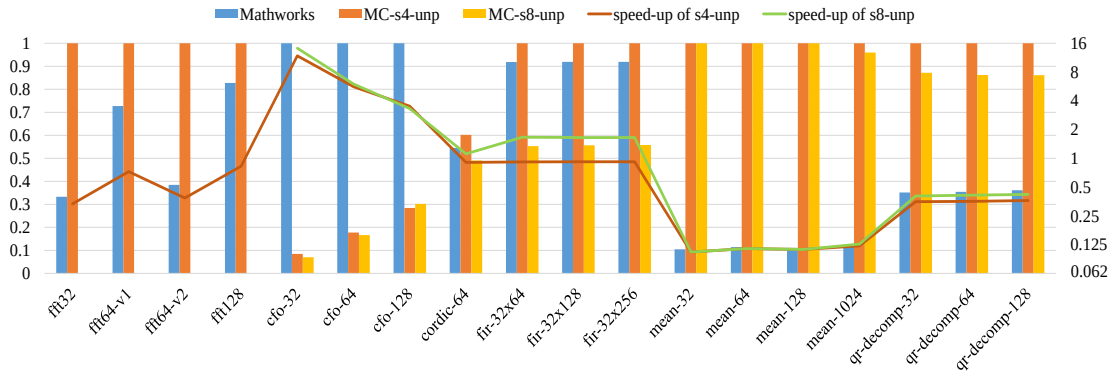

Figure 304: Performance of vectorized generated code (x64-SSE) with unpacked floating point data types compared to MathWorks generated code on i7-3770

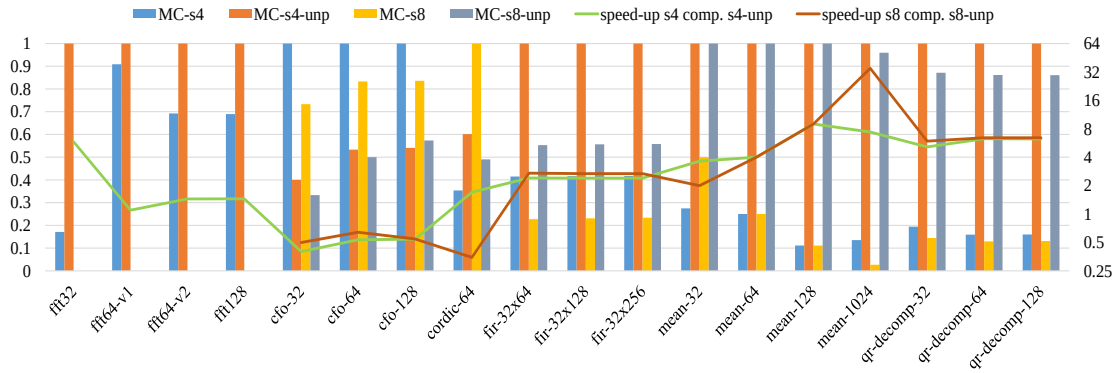

Figure 305: Performance of vectorized code (x64-SSE) with unpacked floating point data types versus packed floating point data types on desktop with i7-3770

Publications

[Latifis et al., 2017] Latifis, I., Parashar, K., Dimitroulakos, G., Cappelle, H., Lezos, C., Masselos, K., and Catthoor, F. (2017). A MATLAB Vectorizing Compiler Targeting Application-Specific Instruction Set Processors, ACM Trans. Des. Autom. Electron. Syst. (TODAES), vol. 22, no. 2, p. 32:1–32:28, Jan. 2017.

[Latifis et al., 2016] Latifis, I., Parashar, K., Dimitroulakos, G., Cappelle, H., Lezos, C., Masselos, K., and Catthoor, F. (2016). MATLAB-to-C compilation targeting Application Specific Instruction Set Processors. In Proceedings of the 2016 Design, Automation & Test in Europe Conference & Exhibition (DATE), pages 1453–1456, Dresden, Germany.

[Lezos et al., 2016a] Lezos, C., Latifis, I., Dimitroulakos, G., and Masselos, K. (2016b). Compiler-directed data locality optimization in MATLAB. In Proceedings of the 19th International Workshop on Software and Compilers for Embedded Systems (SCOPES), pages 6–9, Sankt Goar, Germany.

[Lezos et al., 2016b] Lezos, C., Dimitroulakos, G., Latifis, I., and Masselos, K. (2016a). Automatic generation of code analysis tools: The CastQL approach. In Proceedings of the 1st International Workshop on Real World Domain Specific Languages (RWDSL), pages 3.1–3.10, Barcelona, Spain.

[Lezos et al., 2016c] Christakis Lezos, Grigoris Dimitroulakos, Ioannis Latifis, Konstantinos Masselos, “MAFE: An Environment for MATLAB-to-C Compilation Supporting Static and Dynamic Memory Allocation and Multi-Level User Interactive Code Optimization”, International Symposium on Code Generation and Optimization (CGO), Barcelona, Spain, March 12-18, 2016, [Poster session].

Bibliography

[3GPP-LTE, 2016] 3GPP-LTE (2016). Ieee standard association - ieee get program. http://www.3gpp.org/specifications/releases.

[Allen, 2005] Allen, R. (2005). Compiling high-level languages to DSPs: automating the implementation path. *IEEE Signal Processing Magazine*, 22(3):47–56.

[Allen and Johnson, 1988] Allen, R. and Johnson, S. (1988). Compiling C for Vectorization, Parallelization, and Inline Expansion. In *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*, PLDI '88, pages 241–249, New York, NY, USA. ACM.

[Allen and Kennedy, 2001] Allen, R. and Kennedy, K. (2001). *Optimizing Compilers for Modern Architectures: A Dependence-based Approach*. Morgan Kaufmann, San Francisco, 1 edition edition.

[Almer et al., 2012] Almer, O., Bennett, R., Böhm, I., Murray, A., Qu, X., Zuluaga, M., Franke, B., and Topham, N. (2012). *An End-to-End Design Flow for Automated Instruction Set Extension and Complex Instruction Selection based on GCC*.

[Almási and Padua, 2002] Almási, G. and Padua, D. (2002). MaJIC: Compiling MATLAB for Speed and Responsiveness. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, PLDI '02, pages 294–303, New York, NY, USA. ACM.

[Altera, 2016] Altera (2016). Overview.

[Anderson et al., 1992] Anderson, E., Bai, Z., and Dongarra, J. (1992). Generalized QR factorization and its applications. *Linear Algebra and its Applications*, 162:243–271.

[Andraka, 1998] Andraka, R. (1998). A Survey of CORDIC Algorithms for FPGA Based Computers. In *Proceedings of the 1998 ACM/SIGDA Sixth International Symposium on Field Programmable Gate Arrays*, FPGA '98, pages 191–200, New York, NY, USA. ACM.

[ARM, 2016] ARM (2016). Arm processor architecture - arm. http://www.arm.com/products/processors/instruction-set-architectures/index.php.

[ARM markets, 2016] ARM markets (2016). Markets – arm. http://www.arm.com/markets/.

[ARM reference manual, 2016] ARM reference manual (2016). Arm information center. http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0406c/index.html.

[ARMv8 reference manual, 2016] ARMv8 reference manual (2016). Arm information center. http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.set.architecture/index.html.

[Arnold and Corporaal, 2001] Arnold, M. and Corporaal, H. (2001). Designing Domain-specific Processors. In *Proceedings of the Ninth International Symposium on Hardware/Software Codesign*, CODES '01, pages 61–66, New York, NY, USA. ACM.

[ASIP Designer, 2016] ASIP Designer (2016). Synopsys - asip designer. http://www.synopsys.com/dw/ipdir.php?ds=asip-designer.

[Aslam and Hendren, 2010] Aslam, A. and Hendren, L. (2010). McFLAT: A Profile-Based Framework for MATLAB Loop Analysis and Transformations. In Cooper, K., Mellor-Crummey, J., and Sarkar, V., editors, *Languages and Compilers for Parallel Computing*, number 6548 in Lecture Notes in Computer Science, pages 1–15. Springer Berlin Heidelberg. DOI: 10.1007/978-3-642-19595-2_1.

[Aslam et al., 2010] Aslam, T., Doherty, J., Dubrau, A., and Hendren, L. (2010). AspectMatlab: An Aspect-oriented Scientific Programming Language. In *Proceedings of the 9th International Conference on Aspect-Oriented Software Development*, AOSD '10, pages 181–192, New York, NY, USA. ACM.

[AVX, 2016] AVX (2016). Introduction to Intel® Advanced Vector Extensions | Intel® Software. https://software.intel.com/en-us/articles/introduction-to-intel-advanced-vector-extensions.

[Banerjee, 2003] Banerjee, P. (2003). An overview of a compiler for mapping MATLAB programs onto FPGAs. In *Design Automation Conference, 2003. Proceedings of the ASP-DAC 2003. Asia and South Pacific*, pages 477–482.

[Banerjee et al., 2003] Banerjee, P., Bagchi, D., Haldar, M., Nayak, A., Kim, V., and Uribe, R. (2003). Automatic Conversion of Floating Point MATLAB Programs into Fixed Point FPGA Based Hardware Design. In *Proceedings of the 11th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, FCCM '03, pages 263–, Washington, DC, USA. IEEE Computer Society.

[Banerjee et al., 2004] Banerjee, P., Haldar, M., Nayak, A., Kim, V., Saxena, V., Parkes, S., Bagchi, D., Pal, S., Tripathi, N., Zaretsky, D., Anderson, R., and Uribe, J. (2004). Overview of a compiler for synthesizing MATLAB programs onto FPGAs. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 12(3):312–324.

[Banerjee et al., 1999] Banerjee, P., Shenoy, N., Choudhary, A., Hauck, S., Bachmann, C., Chang, M., Haldar, M., Joisha, P., Jones, A., Kanhare, A., Nayak, A., Periyacheri, S., and Walkden, M. (1999). MATCH: A MATLAB Compiler for Configurable Computing Systems. Technical report.

[Banerjee et al.,] Banerjee, P., Shenoy, N., Choudhary, A., Hauck, S., Bachmann, C., Haldar, M., Joisha, P., Jones, A., Kanhare, A., Nayak, A., Periyacheri, S., Walkden, M., and Zaretsky, D. A MATLAB compiler for distributed, heterogeneous, reconfigurable computing systems. In *FCCM '00*.

[Becker et al., 2012] Becker, J., Stripf, T., Oey, O., Huebner, M., Derrien, S., Menard, D., Sentieys, O., Rauwerda, G., Sunesen, K., Kavvadias, N., Masselos, K., Goulas, G., Alefragis, P., Voros, N. S., Kritharidis, D., Mitas, N., and Goehringer, D. (2012). From Scilab to High Performance Embedded Multicore Systems: The ALMA Approach. In *2012 15th Euromicro Conference on Digital System Design (DSD)*, pages 114–121.

[Benincasa et al., 1998] Benincasa, M., Besler, R., Brassaw, D., and Kohler, R. L. (1998). Rapid development of real-time systems using RTExpressTM. In *Parallel Processing Symposium, 1998. IPPS/SPDP 1998. Proceedings of the First Merged International Parallel Processing Symposium and Symposium on Parallel and Distributed Processing 1998*, pages 594–599.

[Bhatt and McCain, 2005] Bhatt, T. M. and McCain, D. (2005). Matlab As a Development Environment for FPGA Design. In *Proceedings of the 42Nd Annual Design Automation Conference*, DAC '05, pages 607–610, New York, NY, USA. ACM.

[Bik, 2004] Bik, A. J. C. (2004). *Software Vectorization Handbook, The: Applying Intel Multimedia Extensions for Maximum Performance*. Intel Press, Hillsboro, Or.

[Birkbeck et al., 2007] Birkbeck, N., Levesque, J., and Amaral, J. N. (2007). A Dimension Abstraction Approach to Vectorization in Matlab. In *International Symposium on Code Generation and Optimization (CGO'07)*, pages 115–130.

[Bispo et al., 2014] Bispo, J., Reis, L., and Cardoso, J. M. P. (2014). Multi-Target C Code Generation from MATLAB. In *Proceedings of ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming*, ARRAY'14, pages 95:95–95:100, New York, NY, USA. ACM.

[Bispo et al., 2015] Bispo, J., Reis, L., and Cardoso, J. M. P. (2015). Techniques for Efficient MATLAB-to-C Compilation. In *Proceedings of the 2Nd ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming*, ARRAY 2015, pages 7–12, New York, NY, USA. ACM.

[BoT, 2014] BoT (2014). Bot - a low power processor for wireless baseband. http://tinyurl.com/olasdj6.

[Burrus and Parks, 1991] Burrus, C. S. and Parks, T. W. (1991). *DFT/FFT and Convolution Algorithms: Theory and Implementation*. John Wiley & Sons, Inc., New York, NY, USA, 1st edition.

[Cardoso et al., 2012] Cardoso, J. M., Carvalho, T., Coutinho, J. G., Luk, W., Nobre, R., Diniz, P., and Petrov, Z. (2012). LARA: An Aspect-oriented Programming Language for Embedded Systems. In *Proceedings of the 11th Annual International Conference on Aspect-oriented Software Development*, AOSD '12, pages 179–190, New York, NY, USA. ACM.

[Cardoso et al., 2010] Cardoso, J. M. P., Diniz, P. C., Monteiro, M. P., Fernandes, J. M., and Saraiva, J. (2010). A Domain-Specific Aspect Language for Transforming MATLAB Programs. In *Domain-Specific Aspect Language Workshop (DSAL'2010), part of AOSD'10, March 2010*.

[Cardoso et al., 2006] Cardoso, J. M. P., Fernandes, J. M., and Monteiro, M. P. (2006). Adding Aspect-Oriented Features to MATLAB. In *Software Engineering Properties of Languages and Aspect Technologies, Workshop affiliated with AOSD March 2006*, Germany.

[Cardoso et al., 2013] Cardoso, J. M. P., Fernandes, J. M., Monteiro, M. P., Carvalho, T., and Nobre, R. (2013). Enriching MATLAB with Aspect-oriented Features for Developing Embedded Systems. *J. Syst. Archit.*, 59(7):412–428.

[Casey et al., 2010] Casey, A., Li, J., Doherty, J., Chevalier-Boisvert, M., Aslam, T., Dubrau, A., Lameed, N., Aslam, A., Garg, R., Radpour, S., Belanger, O. S., Hendren, L., and Verbrugge, C. (2010). McLab: An Extensible Compiler Toolkit for MATLAB and Related Languages. In *Proceedings of the Third C* Conference on Computer Science and Software Engineering*, C3S2E '10, pages 114–117, New York, NY, USA. ACM.

[Chauhan and Kennedy, 2003] Chauhan, A. and Kennedy, K. (2003). Slice-Hoisting for Array-Size Inference in MATLAB. In Rauchwerger, L., editor, *Languages and Compilers for Parallel Computing*, number 2958 in Lecture Notes in Computer Science, pages 495–508. Springer Berlin Heidelberg. DOI: 10.1007/978-3-540-24644-2_32.

[Chauhan et al., 2003] Chauhan, A., McCosh, C., Kennedy, K., and Hanson, R. (2003). Automatic Type-Driven Library Generation for Telescoping Languages. In *Proceedings of the 2003 ACM/IEEE Conference on Supercomputing*, SC '03, pages 51–, New York, NY, USA. ACM.

[Chauveau and Bodin, 1999] Chauveau, S. and Bodin, F. (1999). Menhir: An Environment for High Performance MATLAB. *Sci. Program.*, 7(3-4):303–312.

[Chevalier-Boisvert et al., 2010] Chevalier-Boisvert, M., Hendren, L., and Verbrugge, C. (2010). Optimizing MATLAB Through Just-in-time Specialization. In *Proceedings of the 19th Joint European Conference on Theory and Practice of Software, International Conference on Compiler Construction*, CC'10/ETAPS'10, pages 46–65, Berlin, Heidelberg. Springer-Verlag.

[Clark et al., 2006] Clark, N., Hormati, A., Mahlke, S., and Yehia, S. (2006). Scalable Subgraph Mapping for Acyclic Computation Accelerators. In *Proceedings of the 2006 International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, CASES '06, pages 147–157, New York, NY, USA. ACM.

[Cooper and Torczon, 2012] Cooper, K. and Torczon, L. (2012). *Engineering a Compiler (Second Edition)*. Morgan Kaufmann, Boston.

[De Rose and Padua, 1996] De Rose, L. and Padua, D. (1996). A MATLAB to Fortran 90 Translator and Its Effectiveness. In *Proceedings of the 10th International Conference on Supercomputing*, ICS '96, pages 309–316, New York, NY, USA. ACM.

[De Rose and Padua, 1999] De Rose, L. and Padua, D. (1999). Techniques for the Translation of MATLAB Programs into Fortran 90. *ACM Trans. Program. Lang. Syst.*, 21(2):286–323.

[De Rose and Padua, 2003] De Rose, L. and Padua, D. (2003). Benchmarking FALCON's MATLAB-to-Fortran 90.

[D'Elia and Demetrescu, 2016] D'Elia, D. C. and Demetrescu, C. (2016). Flexible On-stack Replacement in LLVM. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization*, CGO 2016, pages 250–260, New York, NY, USA. ACM.

[DeRose et al., 1995] DeRose, L., Gallivan, K., Gallopoulos, E., Marsolf, B., and Padua, D. (1995). *A MATLAB Compiler and Restructurer for the Development of Scientific Libraries and Applications.*

[DeRose, L. A., 1996] DeRose, L. A. (1996). *Compiler techniques for MATLAB programs.* Ph.D dissertion, University of Illinois at Urbana-Champaign.

[Eichenberger et al., 2004] Eichenberger, A. E., Wu, P., and O'Brien, K. (2004). Vectorization for SIMD Architectures with Alignment Constraints. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, PLDI '04, pages 82–93, New York, NY, USA. ACM.

[Fasthuber et al., 2013] Fasthuber, R., Catthoor, F., Raghavan, P., and Naessens, F. (2013). *Energy-Efficient Communication Processors.* Springer New York, New York, NY.

[Fixed-Point Designer, 2016] Fixed-Point Designer (2016). Fixed-Point Designer - MATLAB. http://www.mathworks.com/products/fixed-point-designer/.

[Floc'h et al., 2013] Floc'h, A., Yuki, T., El-Moussawi, A., Morvan, A., Martin, K., Naullet, M., Alle, M., L'Hours, L., Simon, N., Derrien, S., Charot, F., Wolinski, C., and Sentieys, O. (2013). Gecos: A framework for prototyping custom hardware design flows. In *Source Code Analysis and Manipulation (SCAM), 2013 IEEE 13th International Working Conference on*, pages 100–105.

[Furber, 2000] Furber, S. (2000). *ARM System-on-Chip Architecture*. Addison-Wesley Professional, Harlow, England; New York, 2 edition edition.

[Garg and Hendren, 2014] Garg, R. and Hendren, L. (2014). Velociraptor: An Embedded Compiler Toolkit for Numerical Programs Targeting CPUs and GPUs. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, PACT '14, pages 317–330, New York, NY, USA. ACM.

[GCC, 2016] GCC (2016). GCC, the GNU Compiler Collection. https://gcc.gnu.org/.

[Guelton et al., 2014] Guelton, S., Falcou, J., and Brunet, P. (2014). Exploring the Vectorization of Python Constructs Using Pythran and Boost SIMD. In *Proceedings of the 2014 Workshop on Programming Models for SIMD/Vector Processing*, WPMVP '14, pages 79–86, New York, NY, USA. ACM.

[Haldar et al., 2001a] Haldar, M., Nayak, A., Choudhary, A., and Banerjee, P. (2001a). A System for Synthesizing Optimized FPGA Hardware from MATLAB. In *Proceedings of the 2001 IEEE/ACM International Conference on Computer-aided Design*, ICCAD '01, pages 314–319, Piscataway, NJ, USA. IEEE Press.

[Haldar et al., 2001b] Haldar, M., Nayak, A., Shenoy, N., Choudhary, A., and Banerjee, P. (2001b). FPGA hardware synthesis from MATLAB. In *Fourteenth International Conference on VLSI Design, 2001*, pages 299–304.

[HDL Coder,] HDL Coder.

[Hendren, 2011] Hendren, L. (2011). Typing Aspects for MATLAB. In *Proceedings of the Sixth Annual Workshop on Domain-specific Aspect Languages*, DSAL '11, pages 13–18, New York, NY, USA. ACM.

[Hendren et al., 2011] Hendren, L., Doherty, J., Dubrau, A., Garg, R., Lameed, N., Radpour, S., Aslam, A., Aslam, T., Casey, A., Chevalier Boisvert, M., Li, J., Verbrugge, C., and Savary Belanger, O. (2011). McLAB: Enabling Programming Language, Compiler and Software Engineering Research for Matlab. In *Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion*, OOPSLA '11, pages 195–196, New York, NY, USA. ACM.

[i7-3770 Processor, 2016] i7-3770 Processor (2016). Intel® Core™ i7-3770 Processor (8M Cache, up to 3.90 GHz) Specifications. http://ark.intel.com/products/65719/Intel-Core-i7-3770-Processor-8M-Cache-up-to-3_90-GHz.

[i7-3820 Processor, 2016] i7-3820 Processor (2016). Intel® Core™ i7-3820 Processor (10M Cache, up to 3.80 GHz) Specifications. http://ark.intel.com/products/63698/Intel-Core-i7-3820-Processor-10M-Cache-up-to-3_80-GHz.

[IEEE-802.11, 2009] IEEE-802.11 (2009). Ieee standard association - ieee get program. http://standards.ieee.org/getieee802/download/802.11n-2009.pdf.

[Intel, 2016] Intel (2016). Intel | data center solutions, iot, and pc innovation. http://www.intel.com/content/www/us/en/homepage.html.

[Jahanzeb et al., 2014] Jahanzeb, M., Palanisamy, A., Sjölund, M., and Fritzson, P. (2014). A MATLAB to Modelica Translator. pages 1285–1294.

[Joisha and Banerjee, 2003a] Joisha, P. G. and Banerjee, P. (2003a). The MAGICA Type Inference Engine for MATLAB ®. In *Proceedings of the 12th International Conference on Compiler Construction*, CC'03, pages 121–125, Berlin, Heidelberg. Springer-Verlag.

[Joisha and Banerjee, 2003b] Joisha, P. G. and Banerjee, P. (2003b). Static Array Storage Optimization in MATLAB. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, PLDI '03, pages 258–268, New York, NY, USA. ACM.

[Joisha and Banerjee, 2006] Joisha, P. G. and Banerjee, P. (2006). An Algebraic Array Shape Inference System for MATLAB®. *ACM Trans. Program. Lang. Syst.*, 28(5):848–907.

[Joisha and Banerjee, 2007] Joisha, P. G. and Banerjee, P. (2007). A Translator System for the MATLAB Language: Research Articles. *Softw. Pract. Exper.*, 37(5):535–578.

[Joisha et al., 2001] Joisha, P. G., Shenoy, U. N., and Banerjee, P. (2001). Computing Array Shapes in MATLAB. In Dietz, H. G., editor, *Languages and Compilers for Parallel Computing*, number 2624 in Lecture Notes in Computer Science, pages 395–410. Springer Berlin Heidelberg. DOI: 10.1007/3-540-35767-X_26.

[Jones et al., 1993] Jones, N. D., Gomard, C. K., and Sestoft, P. (1993). *Partial Evaluation and Automatic Program Generation*. Peter Sestoft. Google-Books-ID: 7rPP-ScYo8w8C.

[Kawabata et al., 2004] Kawabata, H., Suzuki, M., and Kitamura, T. (2004). A MATLAB-Based Code Generator for Sparse Matrix Computations. In Chin, W.-N., editor, *Programming Languages and Systems*, number 3302 in Lecture Notes in Computer Science, pages 280–295. Springer Berlin Heidelberg. DOI: 10.1007/978-3-540-30477-7_19.

[Kennedy and Allen, 2002] Kennedy, K. and Allen, J. R. (2002). *Optimizing Compilers for Modern Architectures: A Dependence-based Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[Kennedy and McKinley, 1990] Kennedy, K. and McKinley, K. S. (1990). Loop Distribution with Arbitrary Control Flow. In *Proceedings of the 1990 ACM/IEEE Conference on Supercomputing*, Supercomputing '90, pages 407–416, Los Alamitos, CA, USA. IEEE Computer Society Press.

[Khoury et al., 2011] Khoury, R., Burgstaller, B., and Scholz, B. (2011). Accelerating the Execution of Matrix Languages on the Cell Broadband Engine Architecture. *IEEE Transactions on Parallel and Distributed Systems*, 22(1):7–21.

[Krukowski and Kale, 1999] Krukowski, A. and Kale, I. (1999). Simulink/matlab-tovhdl route for full-custom/fpga rapid prototyping. In *of DSP Algorithms, Matlab DSP Conference (DSP'99*, pages 16–17.

[Kumar and Hendren, 2014] Kumar, V. and Hendren, L. (2014). MIX10: Compiling MATLAB to X10 for High Performance. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '14, pages 617–636, New York, NY, USA. ACM.

[Lameed and Hendren, 2011] Lameed, N. and Hendren, L. (2011). Staged Static Techniques to Efficiently Implement Array Copy Semantics in a MATLAB JIT Compiler. In Knoop, J., editor, *Compiler Construction*, number 6601 in Lecture Notes in Computer Science, pages 22–41. Springer Berlin Heidelberg. DOI: 10.1007/978-3-642-19861-8_3.

[Lameed and Hendren, 2013a] Lameed, N. A. and Hendren, L. J. (2013a). A Modular Approach to On-stack Replacement in LLVM. In *Proceedings of the 9th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '13, pages 143–154, New York, NY, USA. ACM.

[Lameed and Hendren, 2013b] Lameed, N. A. and Hendren, L. J. (2013b). Optimizing MATLAB Feval with Dynamic Techniques. In *Proceedings of the 9th Symposium on Dynamic Languages*, DLS '13, pages 85–96, New York, NY, USA. ACM.

[Larsen and Amarasinghe, 2000] Larsen, S. and Amarasinghe, S. (2000). Exploiting Superword Level Parallelism with Multimedia Instruction Sets. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, PLDI '00, pages 145–156, New York, NY, USA. ACM.

[Leupers and Marwedel, 1996] Leupers, R. and Marwedel, P. (1996). Instruction Selection for Embedded DSPs with Complex Instructions. In *Proceedings of the Conference on European Design Automation*, EURO-DAC '96/EURO-VHDL '96, pages 200–205, Los Alamitos, CA, USA. IEEE Computer Society Press.

[Leupers and Bashford, 2000] Leupers, R. L. and Bashford, S. (2000). Graph-based Code Selection Techniques for Embedded Processors. *ACM Trans. Des. Autom. Electron. Syst.*, 5(4):794–814.

[Levine, 2009] Levine, J. (2009). *flex & bison*. O'Reilly Media, Sebastopol, CA, 1 edition edition.

[Lezos et al., 2015] Lezos, C., Dimitroulakos, G., and Masselos, K. (2015). Reuse Distance Analysis for Locality Optimization in Loop-dominated Applications. In *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition*, DATE '15, pages 1237–1240, San Jose, CA, USA. EDA Consortium.

[Lezos et al., 2016] Lezos, C., Latifis, I., Dimitroulakos, G., and Masselos, K. (2016). Compiler-Directed Data Locality Optimization in MATLAB. In *Proceedings of the 19th International Workshop on Software and Compilers for Embedded Systems*, SCOPES '16, pages 6–9, New York, NY, USA. ACM.

[Li et al., 2006] Li, J., Zhang, Q., Xu, S., and Huang, B. (2006). Optimizing Dynamic Binary Translation for SIMD Instructions. In *Proceedings of the International Symposium on Code Generation and Optimization*, CGO '06, pages 269–280, Washington, DC, USA. IEEE Computer Society.

[Li et al., 2009] Li, T., Jigang, W., Lam, S. K., Srikanthan, T., and Lu, X. (2009). Efficient Heuristic Algorithm for Rapid Custom-Instruction Selection. In *Eighth IEEE/ACIS International Conference on Computer and Information Science, 2009. ICIS 2009*, pages 266–270.

[Li and Hendren, 2014] Li, X. and Hendren, L. (2014). Mc2for: A tool for automatically translating MATLAB to FORTRAN 95. In *2014 Software Evolution Week - IEEE Conference on Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE)*, pages 234–243.

[Linux, 2016] Linux (2016). The leading os for pc, tablet, phone and cloud | ubuntu. http://www.ubuntu.com/.

[Lopes and Auler, 2014] Lopes, B. C. and Auler, R. (2014). *Getting Started with LLVM Core Libraries*. Packt Publishing.

[Maleki et al., 2011] Maleki, S., Gao, Y., Garzarán, M. J., Wong, T., and Padua, D. A. (2011). An Evaluation of Vectorizing Compilers. In *Proceedings of the 2011 Interna-*

tional Conference on Parallel Architectures and Compilation Techniques, PACT '11, pages 372–382, Washington, DC, USA. IEEE Computer Society.

[Manilov et al., 2015] Manilov, S., Franke, B., Magrath, A., and Andrieu, C. (2015). Free Rider: A Tool for Retargeting Platform-Specific Intrinsic Functions. In *Proceedings of the 16th ACM SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems 2015 CD-ROM*, LCTES'15, pages 5:1–5:10, New York, NY, USA. ACM.

[Mathematica, 2014] Mathematica (2014). Wolfram Mathematica: Modern Technical Computing. https://www.wolfram.com/mathematica/.

[MathWorks Coder, 2016] MathWorks Coder (2016). MATLAB Coder. http://www.mathworks.com/products/matlab-coder/.

[MathWorks Embedded Coder, 2016] MathWorks Embedded Coder (2016). Code Generation - Embedded Coder - Simulink.

[MATLAB, 2016] MATLAB (2016). MATLAB - the Language of Technical Computing. http://www.mathworks.com/products/matlab.

[MATLAB compiler, 2016] MATLAB compiler (2016). Matlab compiler - build standalone applications from matlab programs. http://www.mathworks.com/products/compiler/index.html.

[MATLAB fi, 2016] MATLAB fi (2016). Construct fixed-point numeric object - MATLAB fi. http://www.mathworks.com/help/fixedpoint/ref/fi.html.

[McFarlin and Chauhan, 2007] McFarlin, D. and Chauhan, A. (2007). Library Function Selection in Compiling Octave. In *2007 IEEE International Parallel and Distributed Processing Symposium*, pages 1–8.

[Mei et al., 2003] Mei, B., Vernalde, S., Verkest, D., Man, H. D., and Lauwereins, R. (2003). ADRES: An Architecture with Tightly Coupled VLIW Processor and Coarse-Grained Reconfigurable Matrix. In Cheung, P. Y. K. and Constantinides, G. A., editors, *Field Programmable Logic and Application*, number 2778 in Lecture Notes in Computer Science, pages 61–70. Springer Berlin Heidelberg. DOI: 10.1007/978-3-540-45234-8_7.

[Menon and Pingali, 1999] Menon, V. and Pingali, K. (1999). A Case for Source-level Transformations in MATLAB. In *Proceedings of the 2Nd Conference on Domain-specific Languages*, DSL '99, pages 53–65, New York, NY, USA. ACM.

[Micheli, 1994] Micheli, G. D. (1994). *Synthesis and Optimization of Digital Circuits*. McGraw-Hill Higher Education, 1st edition.

[MSVC, 2016] MSVC (2016). Visual C++ in Visual Studio 2015. https://msdn.microsoft.com/en-us/library/60k1461a.aspx.

[Murray and Franke, 2012] Murray, A. and Franke, B. (2012). Compiling for Automatically Generated Instruction Set Extensions. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, CGO '12, pages 13–22, New York, NY, USA. ACM.

[Naishlos, 2004] Naishlos, D. (2004). Autovectorization in GCC. In *Proceedings of the GCC Developers' Summit*, pages 105–117.

[NEON, 2016] NEON (2016). NEON - ARM. http://www.arm.com/products/processors/technologies/neon.php.

[NEON reference manual, 2016] NEON reference manual (2016). Arm information center. http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.den0018a/index.html.

[Nuzman et al., 2011] Nuzman, D., Dyshel, S., Rohou, E., Rosen, I., Williams, K., Yuste, D., Cohen, A., and Zaks, A. (2011). Vapor SIMD: Auto-vectorize Once, Run Everywhere. In *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '11, pages 151–160, Washington, DC, USA. IEEE Computer Society.

[Nuzman and Henderson, 2006] Nuzman, D. and Henderson, R. (2006). Multi-platform Auto-vectorization. In *Proceedings of the International Symposium on Code Generation and Optimization*, CGO '06, pages 281–294, Washington, DC, USA. IEEE Computer Society.

[Nuzman et al., 2006] Nuzman, D., Rosen, I., and Zaks, A. (2006). Auto-vectorization of Interleaved Data for SIMD. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '06, pages 132–143, New York, NY, USA. ACM.

[Nuzman and Zaks, 2008] Nuzman, D. and Zaks, A. (2008). Outer-loop Vectorization: Revisited for Short SIMD Architectures. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, PACT '08, pages 2–11, New York, NY, USA. ACM.

[Prasad et al., 2011] Prasad, A., Anantpur, J., and Govindarajan, R. (2011). Automatic Compilation of MATLAB Programs for Synergistic Execution on Heterogeneous Processors. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 152–163, New York, NY, USA. ACM.

[Prieto et al., 2005] Prieto, M., Pinuel, L., Catthoor, F., Tirado, F., and Tenllado, C. (2005). Improving superword level parallelism support in modern compilers. In *2005 Third IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS'05)*, pages 303–308.

[Quinn et al., 1998a] Quinn, M., Malishevsky, A., and Seelam, N. (1998a). Otter: bridging the gap between MATLAB and ScaLAPACK. In *The Seventh International Symposium on High Performance Distributed Computing, 1998. Proceedings*, pages 114–121.

[Quinn et al., 1998b] Quinn, M., Malishevsky, A., Seelam, N., and Zhao, Y. (1998b). Preliminary results from a parallel MATLAB compiler. In *Parallel Processing Symposium, 1998. IPPS/SPDP 1998. Proceedings of the First Merged International Parallel Processing Symposium and Symposium on Parallel and Distributed Processing 1998*, pages 81–87.

[Rasbian, 2016] Rasbian (2016). Frontpage - raspbian. https://www.raspbian.org/.

[Raspberry Pi, 2016] Raspberry Pi (2016). Raspberry pi products. https://www.raspberrypi.org/products/.

[Raspberry Pi 2, 2016] Raspberry Pi 2 (2016). Raspberry pi 2 model b. https://www.raspberrypi.org/products/raspberry-pi-2-model-b/.

[Raspberry Pi 3, 2016] Raspberry Pi 3 (2016). Raspberry pi 3 model b. https://www.raspberrypi.org/products/raspberry-pi-3-model-b/.

[Ren et al., 2003] Ren, G., Wu, P., and Padua, D. (2003). A Preliminary Study on the Vectorization of Multimedia Applications for Multimedia Extensions. In *In 16th International Workshop of Languages and Compilers for Parallel Computing*, pages 420–435.

[Ren et al., 2005] Ren, G., Wu, P., and Padua, D. (2005). An Empirical Study On the Vectorization of Multimedia Applications for Multimedia Extensions. In *19th IEEE International Parallel and Distributed Processing Symposium*, pages 89b–89b.

[Ren et al., 2006] Ren, G., Wu, P., and Padua, D. (2006). Optimizing Data Permutations for SIMD Devices. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '06, pages 118–131, New York, NY, USA. ACM.

[Rose et al., 1995] Rose, L. D., Gallivan, K., Gallopoulos, E., Marsolf, B., and Padua, D. (1995). FALCON: A MATLAB interactive restructuring compiler. In Huang, C.-H., Sadayappan, P., Banerjee, U., Gelernter, D., Nicolau, A., and Padua, D., editors, *Languages and Compilers for Parallel Computing*, number 1033 in Lecture Notes in Computer Science, pages 269–288. Springer Berlin Heidelberg. DOI: 10.1007/BFb0014205.

[Roy and Banerjee, 2004] Roy, S. and Banerjee, P. (2004). An algorithm for converting floating-point computations to fixed-point in MATLAB based FPGA design. In *Design Automation Conference, 2004. Proceedings. 41st*, pages 484–487.

[Scharwaechter et al., 2007] Scharwaechter, H., Leupers, R., Ascheid, G., Meyr, H., Youn, J. M., and Paek, Y. (2007). A code-generator generator for Multi-Output Instructions. In *2007 5th IEEE/ACM/IFIP International Conference on Hardware/-Software Codesign and System Synthesis (CODES+ISSS)*, pages 131–136.

[Shei et al., 2009] Shei, C. Y., Chauhan, A., and Shaw, S. (2009). Compile-time disambiguation of MATLAB types through concrete interpretation with automatic run-time fallback. In *2009 International Conference on High Performance Computing (HiPC)*, pages 264–273.

[Shei et al., 2011] Shei, C.-Y., Yoga, A., Ramesh, M., and Chauhan, A. (2011). MATLAB Parallelization through Scalarization. In *2011 15th Workshop on Interaction between Compilers and Computer Architectures (INTERACT)*, pages 44–53.

[Shin, 2007] Shin, J. (2007). Introducing Control Flow into Vectorized Code. In *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*, PACT '07, pages 280–291, Washington, DC, USA. IEEE Computer Society.

[Shin et al., 2005] Shin, J., Hall, M., and Chame, J. (2005). Superword-level parallelism in the presence of control flow. In *International Symposium on Code Generation and Optimization*, pages 165–175.

[Simulink Coder, 2016] Simulink Coder (2016). Automatic Code Generation - Simulink Coder.

[Smith, 1991] Smith, L. L. (1991). Vectorizing C compilers: how good are they? In *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing, 1991. Supercomputing '91*, pages 544–553.

[SoC Embedded Design Suite, 2016] SoC Embedded Design Suite (2016). SoC EDS - Overview.

[SSE, 2016] SSE (2016). SSE - ISA Extensions | Intel® Software. https://software.intel.com/en-us/isa-extensions.

[Stripf et al., 2013] Stripf, T., Oey, O., Bruckschloegl, T., Becker, J., Rauwerda, G., Sunesen, K., Goulas, G., Alefragis, P., Voros, N. S., Derrien, S., Sentieys, O., Kavvadias, N., Dimitroulakos, G., Masselos, K., Kritharidis, D., Mitas, N., and Perschke, T. (2013). Compiling Scilab to high performance embedded multicore systems. *Microprocessors and Microsystems*, 37(8, Part C):1033–1049.

[Stripf et al., 2012] Stripf, T., Oey, O., Bruckschloegl, T., Koenig, R., Huebner, M., Becker, J., Rauwerda, G., Sunesen, K., Kavvadias, N., Dimitroulakos, G., Masselos, K., Kritharidis, D., Mitas, N., Goulas, G., Alefragis, P., Voros, N. S., Derrien, S., Menard, D., Sentieys, O., Goehringer, D., and Perschke, T. (2012). A flexible approach for compiling scilab to reconfigurable multi-core embedded systems. In *2012 7th International Workshop on Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC)*, pages 1–8.

[Stuber et al., 2004] Stuber, G. L., Barry, J. R., McLaughlin, S. W., Li, Y., Ingram, M. A., and Pratt, T. G. (2004). *Broadband MIMO-OFDM wireless communications*, volume 92.

[Sui et al., 2016] Sui, Y., Fan, X., Zhou, H., and Xue, J. (2016). Loop-oriented Array- and Field-sensitive Pointer Analysis for Automatic SIMD Vectorization. In *Proceedings of the 17th ACM SIGPLAN/SIGBED Conference on Languages, Compilers, Tools, and Theory for Embedded Systems*, LCTES 2016, pages 41–51, New York, NY, USA. ACM.

[Thoma et al., 2012] Thoma, Y., Messerli, E., Starkier, M., Molla, D., Masle, S., Bianchi, C., Gubler, O., Magliocco, C., Crausaz, P., Tâche, S., Prêtre, D., and Trolliet, G. (2012). Math2mat: From Octave/Matlab to VHDL. In *2012 NASA/ESA Conference on Adaptive Hardware and Systems (AHS)*, pages 264–271.

[Trifunovic et al., 2009] Trifunovic, K., Nuzman, D., Cohen, A., Zaks, A., and Rosen, I. (2009). Polyhedral-Model Guided Loop-Nest Auto-Vectorization. In *Proceedings of the 2009 18th International Conference on Parallel Architectures and Compilation Techniques*, PACT '09, pages 327–337, Washington, DC, USA. IEEE Computer Society.

[Weijers et al., 2006] Weijers, J.-W., Derudder, V., Janssens, S., Petré, F., and Bourdoux, A. (2006). From MIMO-OFDM Algorithms to a Real-Time Wireless Prototype: A Systematic Matlab-to-Hardware Design Flow. *EURASIP Journal on Advances in Signal Processing*, 2006(1):039297.

[Windows IoT, 2016] Windows IoT (2016). Develop windows 10 iot apps on raspberry pi 3 and arduino - windows iot. https://developer.microsoft.com/en-us/windows/iot.

[Wu et al., 2005] Wu, P., Eichenberger, A. E., and Wang, A. (2005). Efficient SIMD Code Generation for Runtime Alignment and Length Conversion. In *Proceedings of the International Symposium on Code Generation and Optimization*, CGO '05, pages 153–164, Washington, DC, USA. IEEE Computer Society.

[x86 reference manual, 2016] x86 reference manual (2016). Intel® 64 and IA-32 Architectures Optimization Reference Manual.

http://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-optimization-manual.html.

[XILINX, 2016] XILINX (2016). Programmable Devices.

[Zhang et al., 2013] Zhang, J., Xiang, D., Li, T., and Pan, Y. (2013). M2m: A simple Matlab-to-MapReduce translator for cloud computing. *Tsinghua Science and Technology*, 18(1):1–9.