OPTIMIZING CORE SIGNAL PROCESSING FUNCTIONS ON A SUPERSCALAR SIMD ARCHITECTURE

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES
OF
MIDDLE EAST TECHNICAL UNIVERSITY

BY

ÇAĞRI USLU

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR
THE DEGREE OF MASTER OF SCIENCE
IN
ELECTRICAL AND ELECTRONICS ENGINEERING

AUGUST 2019

Approval of the thesis:

OPTIMIZING CORE SIGNAL PROCESSING FUNCTIONS ON A SUPERSCALAR SIMD ARCHITECTURE

submitted by **ÇAĞRI USLU** in partial fulfillment of the requirements for the degree of **Master of Science in Electrical and Electronics Engineering Department, Middle East Technical University** by,

Prof. Dr. Halil Kalıpçılar
Dean, Graduate School of **Natural and Applied Sciences** _____

Prof. Dr. İlkay Ulusoy
Head of Department, **Electrical and Electronics Engineering** _____

Assoc. Prof. Dr. Cüneyt Fehmi Bazlamaçcı
Supervisor, **Electrical and Electronics Engineering, METU** _____

Examining Committee Members:

Prof. Dr. Gözde B. Akar
Electrical and Electronics Eng., METU _____

Assoc. Prof. Dr. Cüneyt F. Bazlamaçcı
Electrical and Electronics Eng., METU _____

Prof. Dr. İlkay Ulusoy
Electrical and Electronics Eng., METU _____

Assoc. Prof. Dr. Süleyman Tosun
Computer Eng., Hacettepe University _____

Assist. Prof. Dr. Gökhan K. Gültekin
Electrical and Electronics Eng., Ank. Yıldırım Beyazıt University _____

Date:

I hereby declare that all information in this document has been obtained and presented in accordance with academic rules and ethical conduct. I also declare that, as required by these rules and conduct, I have fully cited and referenced all material and results that are not original to this work.

Name, Surname: Çağrı Uslu

Signature :

ABSTRACT

OPTIMIZING CORE SIGNAL PROCESSING FUNCTIONS ON A SUPERSCALAR SIMD ARCHITECTURE

Uslu, Çağrı
M.S., Department of Electrical and Electronics Engineering
Supervisor: Assoc. Prof. Dr. Cüneyt Fehmi Bazlamaçcı

August 2019, 120 pages

Digital Signal Processing (DSP) is the basis of many technologies, such as Image Processing, Speech Recognition, Radars, etc. Use of electronic devices such as smartphones, smartwatches, self-driving cars and autonomous robots that take advantage of these technologies becomes widespread and hence it is more critical than ever for these technologies to be realized with high efficiency on cheaper and less power-hungry devices. Cortex-A15 processor architecture is one of the solutions from ARM to this requirement. Therefore, it is worth to optimize certain DSP functions on the Cortex-A15. In this thesis, four commonly used DSP operations are implemented on an ARM Cortex-A15 processor, heavily utilizing the vector co-processor NEON. The optimized operations are Matrix Addition, Matrix Multiplication, Convolution, and Fourier Transform. Although numerous DSP libraries implement these operations, they are not tailored to a specific processor. The functions implemented in this thesis aim to be most efficient on Cortex-A15, which is a superscalar, out-of-order executing processor. All types of processors may suffer from pipeline stalls. However, unlike scalar processors, superscalar processors may achieve a superscalar performance even

in the presence of pipeline stalls. This could be accomplished by utilizing the execution units of the processor better. One way of possibly increasing the utilization of the execution units is instruction reordering. To reorder instructions optimally, one must know certain specifications of the architecture. To discover one of those specifications, i.e. the cost of instructions in clock cycles, a method is developed for performing the appropriate time measurements. Additionally, a set of guidelines for instruction reordering is conceived. Using these guidelines, among other optimization techniques, the DSP functions mentioned earlier are manually optimized to achieve a high execution performance.

Keywords: ARM, NEON, SIMD, Optimization, Instruction Reordering, Digital Signal Processing

ÖZ

BÜYÜK ÖLÇEKLİ BİR SIMD MİMARİSİ ÜZERİNDE ÇEKİRDEK SİNYAL İŞLEME FONKSİYONLARININ PERFORMANSLARININ İYİLEŞTİRİLMESİ

Uslu, Çağrı
Yüksek Lisans, Elektrik ve Elektronik Mühendisliği Bölümü
Tez Yöneticisi: Doç. Dr. Cüneyt Fehmi Bazlamaçcı

Ağustos 2019 , 120 sayfa

Görüntü İşleme, Konuşma Tanıma, Radar gibi teknolojilerin temelinde Sayısal Sinyal İşleme (SSİ) bulunmaktadır. Bu teknolojileri kullanan elektronik cihazlar yaygınlaştıkça, bu teknolojilerin daha ucuz ve daha az enerji harcayan donanımlar tarafından gerçekleştirilmesi büyük önem kazanacaktır. Cortex-A15 işlemci mimarisi ise ARM'ın bu gereksinime yönelik geliştirdiği bir çözümdür. Bu sebeple, SSİ fonksiyonlarının bu mimari üzerinde olabilen en iyi şekilde çalışması faydalı olacaktır. Bu tezde, sık kullanılan 4 SSİ operasyonu ARM Cortex-A15 üzerinde, yardımcı paralel işlemci NEON olabildiğince etkin biçimde kullanılacak şekilde gerçeklenmiştir. Gerçeklenen operasyonlar, Matris Toplamı, Matris Çarpımı, Evrişim ve Fourier Dönüşümü'dür. Bu operasyonlar sayısız yazılım kütüphanesi tarafından gerçekleştirilmiş olsalar da bunların hiç biri belirli bir işlemciye yönelik geliştirilmemişlerdir. Bu tezde yazılan fonksiyonlar, büyük ölçekli, sırasız işleme yapabilen Cortex-A15 üzerinde en etkin çalışacak şekilde yazılmıştır. Her tipteki işlemci ardışık düzen oyalanmalarına maruz kalabilir. Fakat normal ölçekli işlemcilerden farklı olarak, büyük ölçekli işlem-

ciler ardışık düzen oyalanması durumlarında bile büyük ölçekli performans gösterebilirler. Bu, işlemcide bulunan işletme birimlerinden yüksek oranda faydalanılmasıyla sağlanabilir. Bu işletme birimlerinden alınan faydanın artılırmasının bir yolu komutların doğru bir şekilde sıralanması olabilir. Komutların en doğru şekilde sıralanabilmesi için mimari hakkında çeşitli özelliklerin bilinmesi gerekmektedir. Bu özellikler arasında yer alan, her komutun kaç saat döngüsü sürdüğü bilgisinin keşfi için bir metot geliştirilmiştir. Buna ek olarak, komut sıralaması sırasında yol gösterebilecek bazı yönergeler oluşturulmuştur. Başka yöntemlerle beraber bu yönergeler de kullanılarak yukarıda bahsedilen SSİ fonksiyonları geliştirilerek daha yüksek performans elde edilmeye çalışılmıştır.

Anahtar Kelimeler: ARM, NEON, SIMD, Optimizasyon, Komut Sıralama, Sayısal Sinyal İşleme

To my sister, a future scientist

ACKNOWLEDGMENTS

I want to express my earnest gratitude to my thesis supervisor, Cüneyt F. Bazlamaçcı, for his continual support and unbroken patience with me. I feel honored to have worked with him during my entire graduate studies, including this thesis.

TABLE OF CONTENTS

ABSTRACT v

ÖZ vii

ACKNOWLEDGMENTS x

TABLE OF CONTENTS xi

LIST OF TABLES xvi

LIST OF FIGURES xviii

LIST OF ABBREVIATIONS xxi

CHAPTERS

1 INTRODUCTION 1

1.1 Overview 1

1.2 Aim of the Thesis 1

1.3 Contributions of the Thesis 2

1.4 Outline of the Thesis 3

2 BACKGROUND AND LITERATURE OVERVIEW 5

2.1 Background Information 5

2.1.1 Computer Architecture Concepts 5

2.1.1.1 Superscalar Out-of-order Processors 5

2.1.1.2 ARM Cortex-A15 Processor 6

2.1.1.3 Vector Coprocessors 8
2.1.1.4 ARM NEON Extension 8
2.1.2 ARM and NEON assembly 11
2.1.3 Floating Point Numbers and Operations 13
2.1.3.1 Complex Floating Point Numbers and Operations in C Language 13
2.1.3.2 Fused Multiply-Accumulate Operations 14
2.1.4 Compiler Optimizations 14
2.1.4.1 Loop Unrolling and Register Renaming 14
2.1.4.2 Automatic Vectorization 17
2.1.4.3 Instruction Reordering 18
2.1.5 Digital Signal Processing Operations 21
2.1.5.1 Matrix Addition 21
2.1.5.2 Matrix Multiplication 22
2.1.5.3 Convolution 22
2.1.5.4 Fourier Transform 23
2.1.6 Ne10 Library 24
2.2 Literature Review 24
2.2.1 Vectorization 24
2.2.2 Instruction Scheduling 26
2.2.3 Fast Fourier Transform 27
3 DETERMINING THE COST OF ARM AND NEON INSTRUCTIONS IN CLOCK CYCLES 29
3.1 Test Setup 32

3.2 Measurement Method 32
3.2.1 PMU Cycle Counter 33
3.2.2 ISB Instruction 34
3.3 Measurement Results 35
4 OPTIMIZING CORE DSP FUNCTIONS 39
4.1 Matrix Addition 39
4.1.1 Canonical Implementation 39
4.1.2 Vectorization 40
4.1.3 Loop Unrolling 41
4.1.4 Instruction Reordering 41
4.1.5 Ne10 43
4.2 Matrix Multiplication 44
4.2.1 Four-by-four Real-Valued Matrix Multiplication 45
4.2.2 Real-Valued Matrix Multiplication 50
4.2.2.1 Canonical Implementation 50
4.2.2.2 Vectorization 52
4.2.2.3 Loop Unrolling 52
4.2.2.4 Instruction Reordering 53
4.2.3 Complex-Valued Matrix Multiplication 54
4.2.3.1 Canonical Implementation 54
4.2.3.2 Vectorization 54
4.2.3.3 Loop Unrolling 55
4.2.3.4 Instruction Reordering 56

4.3 Convolution 58
4.3.1 Canonical Implementation 58
4.3.2 Vectorization 59
4.3.3 Loop Unrolling 59
4.3.4 Instruction Reordering 59
4.4 Fourier Transform 60
5 PERFORMANCE EVALUATION 63
5.1 Functional Verification of the Operations 63
5.2 Test Setup 64
5.3 Performance Measurement Method 64
5.4 Using Speed-up as a Performance Metric 65
5.5 Per-cycle Performance Metrics 66
5.5.1 Using Additions Per Cycle as a Performance Metric 66
5.5.2 Using Multiply-Accumulate Operations Per Cycle as a Performance Metric 67
5.5.3 Using Butterfly Operations Per Cycle as a Performance Metric 67
5.6 Comparison Results 67
5.6.1 Matrix Addition 68
5.6.2 Matrix Multiplication 70
5.6.2.1 Four-by-Four Real-Valued Matrix Multiplication 70
5.6.2.2 Real-Valued Matrix Multiplication 71
5.6.2.3 Complex-Valued Matrix Multiplication 73
5.6.3 Convolution 75

5.6.4 Fourier Transform 77
5.7 Summary 78
5.8 Instruction Reordering Guidelines 79
6 CONCLUSION AND FUTURE WORK 81
6.1 Conclusion 81
6.2 Future Work 81
REFERENCES 83
A RESULTS 87
A.1 Matrix Addition 87
A.2 Four-by-Four Real-Valued Matrix Multiplication 90
A.3 Real-Valued Matrix Multiplication 91
A.4 Complex-Valued Matrix Multiplication 94
A.5 Convolution 97
A.6 Fourier Transform 99
B SOURCE CODE 103
B.1 Mat File Parser 103
B.2 Matrix Addition 106
B.3 Four-by-four Matrix Multiplication 107
B.4 Real-Valued Matrix Multiplication 111
B.5 Complex-Valued Matrix Multiplication 113
B.6 Convolution 115
B.7 Fast Fourier Transform 116

LIST OF TABLES

TABLES

Table 3.1 Cost of Integer Arithmetic Instructions 36

Table 3.2 Cost of Integer Multiply/Divide Instructions 36

Table 3.3 Cost of NEON Arithmetic Instructions 36

Table 3.4 Cost of NEON Multiplication Instructions 37

Table 3.5 Cost of NEON Load/Store Instructions 37

Table 4.1 Optimizations Performed on Matrix Addition Code 43

Table 4.2 Optimizations Performed on Real-Valued Matrix Multiplication Code 54

Table 4.3 Optimizations Performed on Complex-Valued Matrix Multiplication Code 58

Table 5.1 Results of 16-point DFT 77

Table 5.2 Summary of the Results, Minimum and Maximum Speed-up Observed 78

Table A.1 Results of Optimizations Performed on Matrix Addition (Part I) . . . 87

Table A.2 Results of Optimizations Performed on Matrix Addition (Part II) . . 88

Table A.3 Results of Optimizations Performed on Matrix Addition (Part III) . . 88

Table A.4 Results of Optimizations Performed on Matrix Addition (Part IV) . . 89

Table A.5 Results of Optimizations Performed on Matrix Addition (Part V) . . 89

Table A.6 Results of Optimizations Performed on Matrix Addition (Part VI) . . 90

Table A.7 Results of Optimizations Performed on Matrix Addition (Part VII) . 90

Table A.8 Results of Four-by-Four Real-Valued Matrix Multiplications 91

Table A.9 Results of Real-Valued Matrix Multiplications (Part I) 92

Table A.10 Results of Real-Valued Matrix Multiplications (Part II) 92

Table A.11 Results of Real-Valued Matrix Multiplications (Part III) 93

Table A.12 Results of Real-Valued Matrix Multiplications (Part IV) 93

Table A.13 Results of Real-Valued Matrix Multiplications (Part V) 94

Table A.14 Results of Real-Valued Matrix Multiplications (Part VI) 94

Table A.15 Results of Complex-Valued Matrix Multiplications (Part I) 95

Table A.16 Results of Complex-Valued Matrix Multiplications (Part II) 96

Table A.17 Results of Convolution (Part I) 97

Table A.18 Results of Convolution (Part II) 98

Table A.19 Results of Convolution (Part III) 99

Table A.20 Results of Different Fourier Transform Implementations (Part I) . . 100

Table A.21 Results of Different Fourier Transform Implementations (Part II) . . 101

LIST OF FIGURES

FIGURES

Figure 2.1 General Structure of an Out-of-order Superscalar Processor . . . 6

Figure 2.2 The General Structure of Cortex-A15 extracted from [1] 7

Figure 2.3 Execution Pipelines of Cortex-A15 extracted from [2] 8

Figure 2.4 NEON Registers 10

Figure 2.5 Addition of two registers with NEON 11

Figure 2.6 Matrix Addition 21

Figure 2.7 8-point Fast Fourier Transform expressed in terms of two 4-point transforms and a weighted sum 24

Figure 2.8 Hybrid compile-time run-time auto-vectorization proposal by [3] 25

Figure 2.9 Single-precision floating-point performance of FFTW against other FFT implementations 28

Figure 3.1 Timings of the instructions of the unscheduled code 30

Figure 3.2 Timings of the instructions of the scheduled code 31

Figure 5.1 Butterfly Operation 67

Figure 5.2 Addition per Cycle Results of Optimizations Performed on Matrix Addition (Part I) 68

Figure 5.3 Addition per Cycle Results of Optimizations Performed on Matrix Addition (Part II) 69

Figure 5.4 Speed-up Results of Optimizations Performed on Matrix Addition 69

Figure 5.5 Mul.-Acc. per Cycle Results of Optimizations Performed on Four-by-Four Real-Valued Matrix Multiplication 70

Figure 5.6 Speed-up Results of Optimizations Performed on Four-by-Four Real-Valued Matrix Multiplication 71

Figure 5.7 Mul.-Acc. per Cycle Results of Optimizations Performed on Real-Valued Matrix Multiplication (Part I) 72

Figure 5.8 Mul.-Acc. per Cycle Results of Optimizations Performed on Real-Valued Matrix Multiplication (Part II) 72

Figure 5.9 Speed-up Results of Optimizations Performed on Real-Valued Matrix Multiplication 73

Figure 5.10 Mul.-Acc. per Cycle Results of Optimizations Performed on Complex-Valued Matrix Multiplication (Part I) 74

Figure 5.11 Mul.-Acc. per Cycle Results of Optimizations Performed on Complex-Valued Matrix Multiplication (Part II) 74

Figure 5.12 Speed-up Results of Optimizations Performed on Complex-Valued Matrix Multiplication 75

Figure 5.13 Mul.-Acc. per Cycle Results of Optimizations Performed on Complex Convolution (Part I) 76

Figure 5.14 Mul.-Acc. per Cycle Results of Optimizations Performed on Complex Convolution (Part II) 76

Figure 5.15 Speed-up Results of Optimizations Performed on Complex Convolution 77

Figure 5.16 Butterflies per Cycle Results of Various Fourier Transform Implementations 78

LIST OF ABBREVIATIONS

D-Cache	Data Cache
DFT	Discrete Fourier Transform
DSP	Digital Signal Processing
FFT	Fast Fourier Transform
FIR	Finite Impulse Response
FP	Floating-point
GCC	GNU Compiler Collection
HEVC	High Efficiency Video Coding
I-Cache	Instruction Cache
IIR	Infinite Impulse Response
I/O	Input / Output
IUI	Instruction Under Investigation
L/S	Load / Store
PMU	Performance Monitor Unit
SIMD	Single Instruction Multiple Data
SSİ	Sayısal Sinyal İşleme
TLB	Translation Look-aside Buffer
VLIW	Very Large Instruction Word

CHAPTER 1

INTRODUCTION

1.1 Overview

Digital Signal Processing is used in many application areas such as Audio Signal Processing, Image Processing, Biosensors, Medical Imaging, Pattern Recognition, Signal Compression, Speech Recognition, Digital Communications, Radars, Sonars, Seismic Data Processing, etc. In mission-critical settings such as military or safety applications, keeping the execution time of digital signal processing operations as low as possible is essential. Moreover, these mission critical applications started to find their way into smaller and more mobile devices, which almost always operate on battery power. This brings additional requirements to the computer architecture and algorithm design. Since power consumption is a fairly recent requirement in computing devices, most compilers do not generate optimal code for less-power hungry architectures, or existing software libraries do not contain code that can run optimally on these devices. In this thesis, several optimization techniques, both architecture dependent and independent, are experimented on certain signal processing functions for a sample architecture, the ARM Cortex-A15. The Cortex-A15 aims to be power-efficient while delivering competitive performance in high-end consumer devices and military applications.

1.2 Aim of the Thesis

Learning from the experiments in this thesis, we aspire to obtain DSP subroutines that can achieve better performance than the code generated by the compiler from an

architecture-independent code, or the code readily available in architecture-dependent software libraries. It is expected that a performance gain will be achieved once certain optimization methods, such as vectorization, loop unrolling, and instruction reordering, are applied. In order to achieve an optimal instruction ordering, a deeper knowledge about the architecture is required, such as, the number of execution units, cost of each instruction in clock cycles, and the existing out-of-order execution capabilities of the processor. Since most of these topics are unknown to compilers, it is expected that manual optimization of the functions by the programmer will be beneficial.

It is not always possible to find all the necessary information needed to optimally reorder the instructions in the official documentation of the processor. Two such non-existent issues are the *exact cost of each instruction in clock cycles* and the *existing out-of-order execution capabilities of the processor* for Cortex-A15. A method is developed during this thesis to discover the cycle cost of each instruction, hoping that it will be beneficial during instruction reordering, however, details about the out-of-order capabilities of the processor still remained a mystery.

In addition, since there aren't any DSP libraries specifically optimized for the processor used in this thesis, a software library that is optimized to work on all ARM processors, namely the Ne10, is further optimized by us to run faster on Cortex-A15, whenever applicable.

1.3 Contributions of the Thesis

Our contributions can be summarized in three main points:

- A method is developed to determine the cost of each instruction in clock cycles. Although no occasion to use these discoveries is encountered while optimizing the four functions, the instruction timings are still provided in this thesis with the hope of helping other researchers.
- Certain optimization techniques are applied to decrease the execution time of four widely used DSP operations; namely, Matrix Addition, Matrix Multiplication, Convolution, and Fourier Transform. Although a considerable speed-up is

achieved in three of these operations, the performance gain has been marginal for the Fourier Transform case.

- During this thesis work, a set of guidelines for instruction reordering is conceived. These guidelines may be of use to fellow researchers in their struggles to achieve high performance systems.

1.4 Outline of the Thesis

In chapter 2, background information is provided. Some architectural topics, such as Superscalar Out-of-order Processors and Vector Co-processors, are briefly introduced. Software related topics about floating-point calculations are mentioned. The current optimization capabilities of compilers are summarized. The definition of the DSP operations examined in this thesis are given. Additionally, the current state of the art in auto-vectorization, instruction scheduling and Fourier Transform is summarized.

In chapter 3, the method used to determine the cost of instructions in clock cycles is discussed and the obtained results are provided.

In chapter 4, the optimizations applied to the DSP functions are presented in detail with accompanying code examples.

In chapter 5, the results of the applied optimizations, followed by the instruction reordering guidelines, are provided.

In chapter 6, concluding remarks and directions for future work are given.

CHAPTER 2

BACKGROUND AND LITERATURE OVERVIEW

2.1 Background Information

2.1.1 Computer Architecture Concepts

2.1.1.1 Superscalar Out-of-order Processors

Superscalar and out-of-order executing processors exploit a certain type of instruction-level parallelism by allowing the execution of more than one instructions per cycle. Together with the branch prediction methods, superscalar processors aim to achieve a full pipeline at all times.

The general structure of a superscalar out-of-order executing processor is given in figure 2.1. In these types of processors, the instructions are fetched into Instruction Buffer, in which they are buffered until the pipeline is ready for them to be decoded and dispatched. After buffering, the instructions are decoded and placed in reservation stations where they wait for their operands to be ready. The out-of-execution begins is reservation stations as the instructions whose operands are ready may be dispatched before other instructions which are still waiting for operands. Once the operands of an instruction are complete, the execution begins in the execution pipeline. This is called *being issued* to an execution unit. There may be more than one execution pipelines. This fact introduces the ability of executing more than one instructions per cycle. The difference in the duration of the waiting that happens in the reservation stations, and the difference in the lengths of the execution pipelines are the causes of the difference in the ordering of the instructions leaving the execution stage compared to the original ordering of the program. The execution results are gathered in the reorder buffer so

that the side-effects, such as setting or clearing of the condition codes, happen in the order intended by the program.

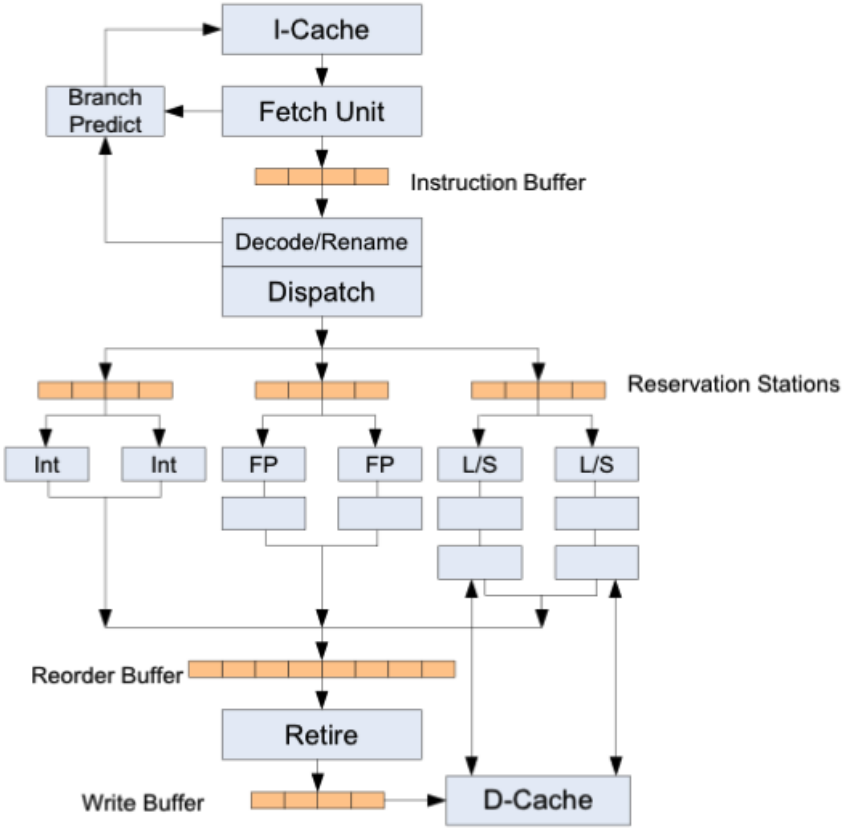

Figure 2.1: General Structure of an Out-of-order Superscalar Processor

Although the out-of-order processors allow reordering of instructions in the reservation stations to increase pipeline efficiency, it is aware only of a small window of instructions. As the results of this thesis shows, manual reordering of instructions may still be beneficial to a certain extent.

2.1.1.2 ARM Cortex-A15 Processor

ARM Cortex-A15 processor is a high-performance, low-power microprocessor that implements the ARMv7-A architecture specifications. A maximum of four cores can exist in a Cortex-A15 chip, with separate L1 caches and a shared L2 cache.

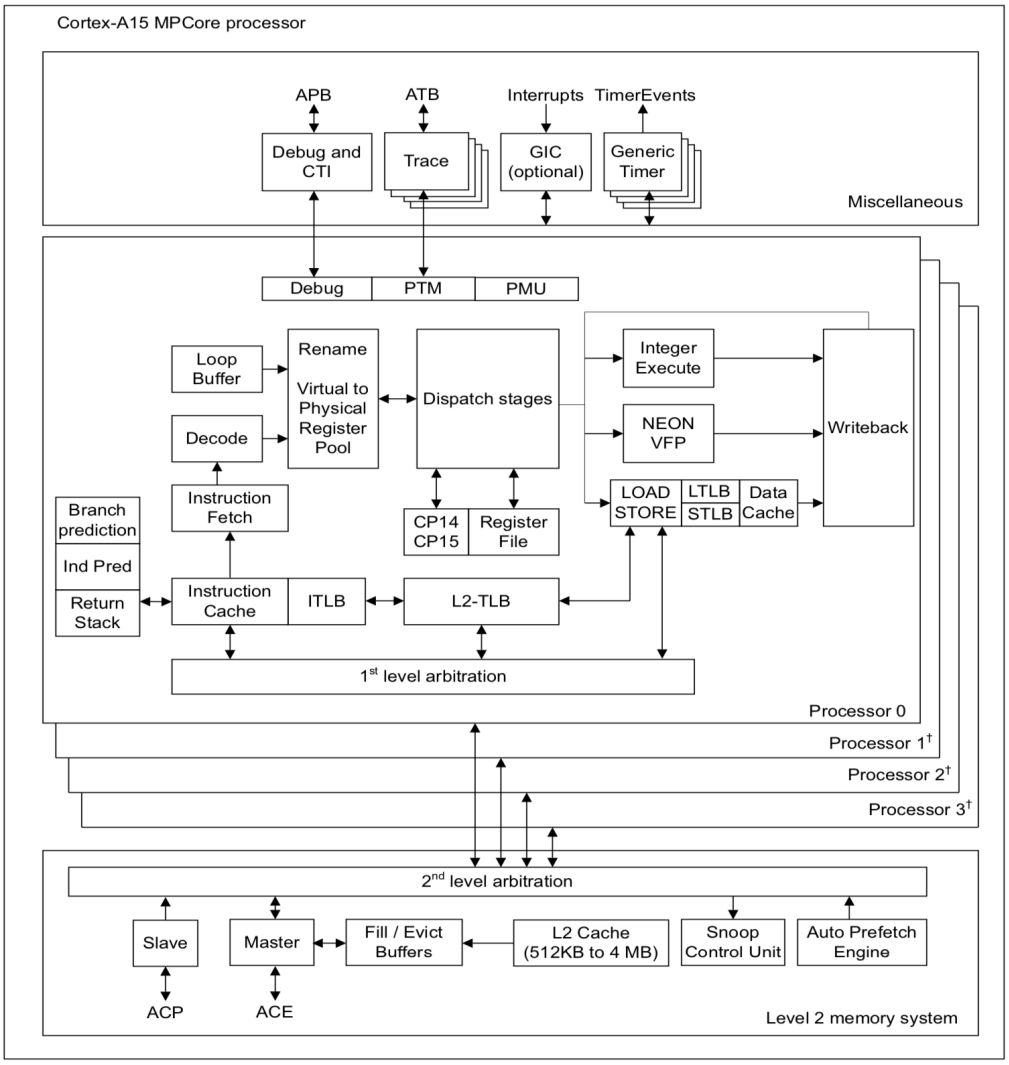

Figure 2.2: The General Structure of Cortex-A15 extracted from [1]

The general structure of Cortex-A15 is given in figure 2.2. This figure shows that Cortex-A15 has an instruction cache, branch prediction capability, loop buffer, register renaming, and multiple execution pipelines. In this figure, not all execution pipelines of the design is given. In figure 2.3, which is originally from [2], it is observed that the execution stage of Cortex-A15 contains additional units. It has two integer arithmetic pipelines, one integer multiply pipeline, one branch resolution pipeline, two NEON/VFP pipelines, one load pipeline, and one store pipeline. From [1], we learn that there is also an iterative integer divide pipeline, which is not shown in figures.

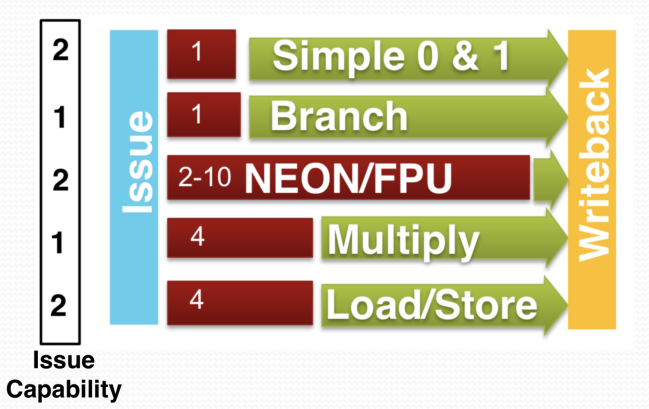

Figure 2.3: Execution Pipelines of Cortex-A15 extracted from [2]

The number of pipelines mirror the *issue capacity* of the execution units. For example, in the same cycle, two integer arithmetic instructions may be issued, since there are two parallel pipelines for it. Hence, this architecture has great potential for performance gain if the superscalar structure can be appropriately utilized.

2.1.1.3 Vector Coprocessors

After the disappearance of vector processors, vector *coprocessors* became the preferred and the cheapest way of acquiring high throughput in scientific computations and multimedia applications. Vector coprocessors, or commonly referred to as SIMD extensions, apply the same arithmetic or logical operations to vectorized data that is loaded from memory. All major chip manufacturers include SIMD extensions to their designs. IBM's AltiVec [4], Intel's AVX [5], and ARM's NEON [6] are examples of these extensions.

All widely used variations of SIMD extensions use their own set of registers that are distinct from the registers of the processor.

2.1.1.4 ARM NEON Extension

NEON is the name of the SIMD extensions in the ARM processors. In Cortex-A15, NEON has two separate execution pipelines to which two instructions may be issued

at the same time.

NEON operations use a register file that is separate from the ARM core. It has 16 128-bit wide registers where each of them can hold two 64-bit, or four 32-bit floating-point numbers. It can also hold many combinations of different width integer numbers, but in this thesis, only floating-point numbers are of interest. In [6], it is said that NEON is not fully compatible with IEEE Standard for Floating-Point Arithmetic (IEEE 754).

The registers in NEON can be accessed in a couple of different ways. First, there are 16 quad registers that are 128-bit wide and named from `q0` to `q15`. Second, there are 32 double registers that can hold 64-bits and named from `d0` to `31`. However, the registers `d(n)` and `d(n+1)` point to the lower and upper halves of the `q(n/2)` register, where `n` is an even integer in $[0, 30]$ interval. For example, the registers `d18` and `d19` point to the lower and upper halves of the `q9` register. Third, there are 32 single registers that can hold 32-bit values and named from `s0` to `s31`. Similarly, the registers `s(n)` and `s(n+1)` point to the lower and upper halves of the `d(n/2)` register, where `n` is an even integer in $[0, 30]$ interval. Hence, using single registers, it is not possible to access the `d16-31` registers. This register access pattern is visualized in figure 2.4.

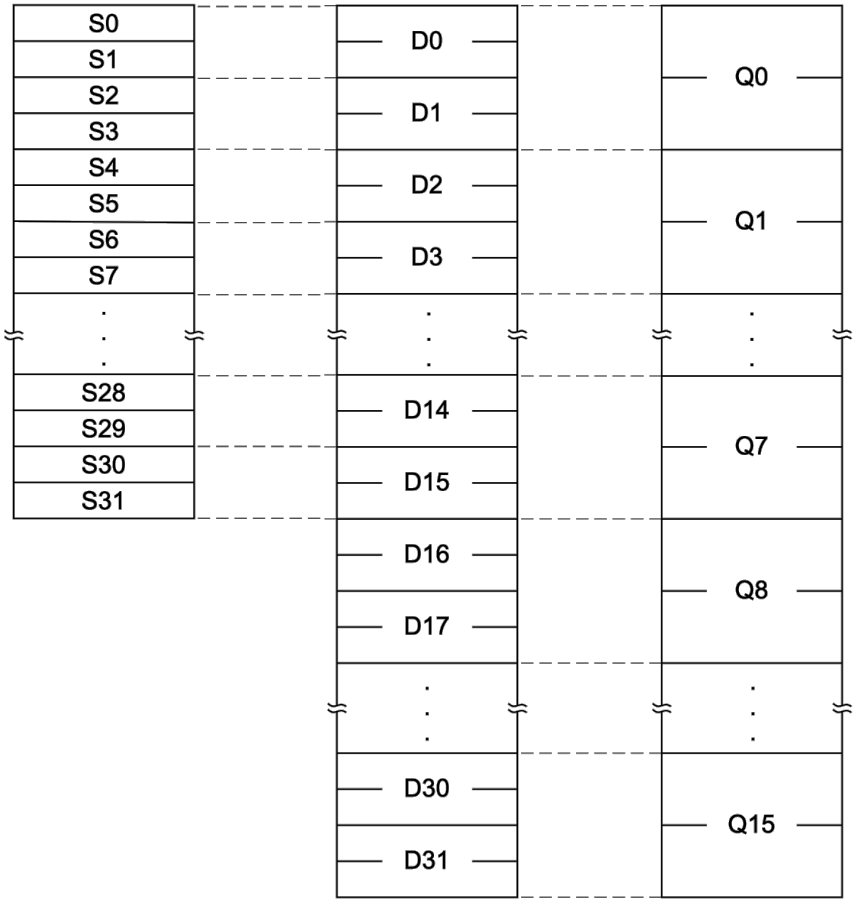

Figure 2.4: NEON Registers

When using 32-bit floating-point numbers, the NEON quad registers can hold four floating point numbers. Each number in a register is called a lane. The arithmetic operations between NEON registers are applied on respective lanes. The NEON processor, besides many types of data move instructions with different patterns, can add, subtract, accumulate, multiply, and multiply-accumulate floating-point numbers. Figure 2.5 shows the visualization of an addition operation with NEON.

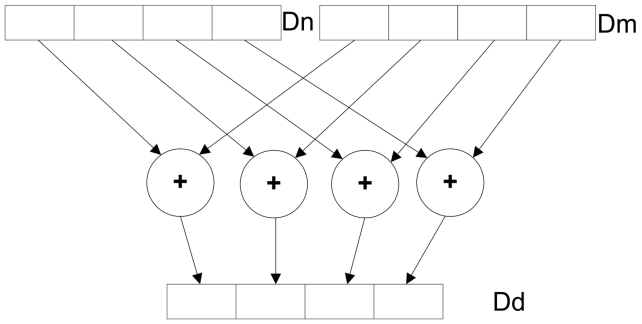

Figure 2.5: Addition of two registers with NEON

In Cortex-A15, the NEON execution unit includes the ARM's non-vectorized floating-point unit named VFP. This unit is also used, albeit rarely, in this thesis for non-parallel arithmetic operations.

2.1.2 ARM and NEON assembly

In this subsection, ARM and NEON assembly language is explained such that the codes given in this thesis are easier to understand.

In ARM assembly, the arithmetic and binary operation instructions use two or three operands:

```
add r0, r1      @ Two operands
add r0, r1, r2  @ Three operands
```

In two operand version of the arithmetic instructions, the first operand is both one of the source operands and the destination operand. For three operand version, the first operand is the destination, the second and third operands are the source. The respective RTL representations of above code is given below:

$$R_0 \leftarrow R_0 + R_1$$

$$R_0 \leftarrow R_1 + R_2$$

The instructions that begin with the letter `v` are NEON/VFP instructions. For example, the following code block performs the addition of four 32-bit floating point

numbers that are placed in NEON quad registers.

```
vadd.f32 q0, q1, q2
```

$$Q_0 \leftarrow Q_1 + Q_2$$

Similar to the non-NEON case, NEON instructions have also two operand versions:

```
vadd.f32 q0, q1
```

$$Q_0 \leftarrow Q_0 + Q_1$$

In this thesis, only the NEON load/store instructions are used. These instructions again begin with the letter v. The explanation of certain types of load/store instructions used in this thesis is given below.

```
@ Load 4 floating-point numbers from address r0 to q0
@ without deinterleaving
vld1.32 q0, [r0]

@ Load 8 floating-point numbers from address r0 to q0
@ and q1 without deinterleaving
vld1.32 {q0-q1}, [r0]

@ Load 8 floating-point numbers from address r0 to q0
@ and q1 without deinterleaving and increment r0 by
@ the number of bytes loaded
vld1.32 {q0-q1}, [r0]!

@ Load 8 floating-point numbers from address r0 to q0
@ and q1 with 2-way deinterleaving and increment r0
@ by the number of bytes loaded
vld2.32 {q0-q1}, [r0]!

@ Store 8 floating-point numbers from q0 and q1 to
@ address r0 with 2-way interleaving and increment
```

```
@ r0 by the number of bytes stored
vst2.32 {q0-q1}, [r0]!
```

2.1.3 Floating Point Numbers and Operations

2.1.3.1 Complex Floating Point Numbers and Operations in C Language

Complex numbers are added to the C language with the ISO/IEC 9899:1999 standard. This version brought floating-point complex number types with 3 different precisions; 32-bit, 64-bit, and 128-bit.

Complex floating numbers are kept in memory as two separate floating-point numbers where the imaginary part follows the real part of the number. As a result, complex number arrays are kept in memory in such a way that the real part of the number and the imaginary part are interleaved. Thankfully, NEON has a variant of the load instruction, VLD2, which can deinterleave the data while loading them to NEON registers:

```
vld2.32 {q0-q1}, [r1]
```

This instruction, while loading the data, places the even indexed numbers into `q0` register while placing the odd indexed numbers into `q1` register, thus separating the real part of the numbers from their imaginary parts.

Another frequently used variant of the load instruction is the post-increment load.

```
vld2.32 {q0-q1}, [r1]!    @ Notice the (!)
```

This instruction, after loading the data from the address pointed by `r1`, increments `r1` by the number of bytes loaded. As a result, the `r1` pointer points to the next number in a sequence of numbers. In the upcoming chapters of this thesis, we should keep in mind that the post-increment operation is executed *after* the loading of the data.

2.1.3.2 Fused Multiply-Accumulate Operations

In DSP applications, the multiply-accumulate operation is a frequently encountered step of many operations. For this reason, most floating-point computation units include specialized hardware that can handle this operation. Multiply-accumulate can either be done in two steps with rounding after each step, or in a single step with only one rounding. It is called a fused multiply-accumulate operation if the computation is done in a single step. If the hardware has native support for fused operations, the result may be more accurate.

Although the processor used in this thesis has support for fused floating point operations, the Cortex-A15 specifications do not mandate their existence. For this reason, fused floating-point operations are not used while implementing the functions to stay compatible with all Cortex-A15 implementations from all vendors.

2.1.4 Compiler Optimizations

Compilers usually employ many optimization techniques to generate better code. These optimizations either try to compensate for the overhead that comes from high-level programming languages or help the programmer by automatically detecting and applying certain optimizations opportunities.

One should note that, better code does not always mean faster code. For embedded systems, memory usage and binary size are also important factors during development. For the hardware setup used in this thesis, however, binary size and memory usage were not a limiting factor.

2.1.4.1 Loop Unrolling and Register Renaming

Loop unrolling, or sometimes referred to as loop unwinding, is the act of copying the instructions of a loop repeatedly to diminish the overhead of branching and comparison. Note that, the compiler cannot unroll all loops. Either the iteration count or one of its submultiples should be known during the compile time for the compiler to generate unrolled loops.

Decreasing the number of branch instructions may result in a noticeable performance gain due to increased pipeline utilization. Even though the branch prediction systems in modern processors reduce the cost of branches, the cost of fetching and decoding of a branch instruction is here to stay. Additionally, although it is not as costly as the branch operation, decreasing the number of comparison operations is also beneficial.

Register renaming, which itself is a separate optimization method, is often used hand-in-hand with loop unwinding. If register renaming is applied to the unwinded loop, and the calculations made in consecutive iterations are independent of each other, unrolled statements may be executed in parallel by a superscalar processor, improving the performance further.

There is usually a limit to how much performance can be gained by unrolling the loops. After a certain point of unrolling, the code size may become so large that the instruction cache misses may start to degrade the overall execution time of the loop.

In this thesis, loop unwinding is utilized both manually and automatically. For the compiler used during this thesis work, loop unrolling is enabled using *-funroll-loops* option.

An example for manual unrolling of a C loop is as follows:

```
@ Not unrolled
for (i = 0; i < 256; i++)
  y[i] = a[i] + b[i];

@ Unrolled with a factor of 2
for (i = 0; i < 256; i += 2)
{
  y[i] = a[i] + b[i];
  y[i+1] = a[i+1] + b[i+1];
}

@ Unrolled with a factor of 4
for (i = 0; i < 256; i += 4)
{
```

```
  y[i] = a[i] + b[i];
  y[i+1] = a[i+1] + b[i+1];
  y[i+2] = a[i+2] + b[i+2];
  y[i+3] = a[i+3] + b[i+3];
}
```

Unrolling of the same code using assembly is provided here:

```
@ Not unrolled
for (i = 0; i < 256; i++)
{
  asm (
    "vld1.32 s0, [%[a]]"
    "vld1.32 s1, [%[b]]"
    "vadd.f32 s0, s1"
    "vst1.32 s0, [%[y]]"
  );
}

@ Unrolled with a factor of 2
for (i = 0; i < 256; i += 2)
{
  asm (
    "vld1.32 s0, [%[a]]!"
    "vld1.32 s1, [%[b]]!"
    "vadd.f32 s0, s1"
    "vst1.32 s0, [%[y]]!"

    "vld1.32 s0, [%[a]]!"
    "vld1.32 s1, [%[b]]!"
    "vadd.f32 s0, s1"
    "vst1.32 s0, [%[y]]!"
  );
```

```
}

@ Unrolled with a factor of 4
for (i = 0; i < 256; i += 4)
{
  asm (
    "vld1.32 s0, [%[a]]!"
    "vld1.32 s1, [%[b]]!"
    "vadd.f32 s0, s1"
    "vst1.32 s0, [%[y]]!"

    "vld1.32 s2, [%[a]]!"
    "vld1.32 s3, [%[b]]!"
    "vadd.f32 s2, s3"
    "vst1.32 s2, [%[y]]!"

    "vld1.32 s4, [%[a]]!"
    "vld1.32 s5, [%[b]]!"
    "vadd.f32 s4, s5"
    "vst1.32 s4, [%[y]]!"

    "vld1.32 s6, [%[a]]!"
    "vld1.32 s7, [%[b]]!"
    "vadd.f32 s6, s7"
    "vst1.32 s6, [%[y]]!"
  );
}
```

2.1.4.2 Automatic Vectorization

Automatic vectorization is the act of converting sequential mathematical computations into vector operations, if possible. After the computation is converted to vector-

ized form, it is much easier to generate associated code to run on SIMD hardware.

The need for automatic vectorization stems from the fact that the most widely used programming languages, such as C and FORTRAN, do not have semantics to describe vector operations [7]. Programmers convert these vector operations to sequential loops because that is what the language supports. The compiler then tries to analyze the code to identify the vector operations and generate code that utilizes the SIMD hardware.

Below, a vectorization example is given.

```
int a[256], b[256], c[256];

@ Before vectorization
for (i = 0; i < 256; i++)
  a[i] = b[i] + c[i];

@ After vectorization
for (i = 0; i < 256; i += 8)
{
  asm (
    "vld1.32 {q0-q1}, [%[b]]!"
    "vld1.32 {q2-q3}, [%[c]]!"
    "vadd.f32 q0, q2"
    "vadd.f32 q1, q3"
    "vst1.32 {q0-q1}, [%[a]]!"
  );
}
```

2.1.4.3 Instruction Reordering

Instruction reordering is the act of reordering the instructions to maximize the processor's pipeline utilization. It can decrease or get rid of the cost associated with pipeline stalls. Although the out-of-order executing processors apply reordering internally, as

the results of this thesis shows, offline reordering of instructions can also lead to a performance gain.

Instruction reordering is usually utilized to decrease the data dependency of consecutive instructions. In the example given below, before reordering, almost all of the instructions have wait for the previous one to finish, since they require the result of the previous calculation. However, when the instructions are reordered as given, the second load operation need not wait for the first load operation to finish, thus, can start executing in parallel. Same reasoning also applies to the second addition instruction and the second store instruction.

```
@ Before reordering
vld1.32 {q0-q1}, [r0]
vadd.f32 q0, q1
vst1.32 {q0}, [r1]

vld1.32 {q2-q3}, [r2]
vadd.f32 q2, q3
vst1.32 {q2}, [r3]

@ After reordering
vld1.32 {q0-q1}, [r0]
vld1.32 {q2-q3}, [r2]
vadd.f32 q0, q1
vadd.f32 q2, q3
vst1.32 {q0}, [r1]
vst1.32 {q2}, [r3]
```

There are other reasons to apply reordering. The reordering of instructions may increase the utilization of the execution pipelines when the issue capability of the pipelines are maximized. An example is given below.

```
@ Before reordering
vmul.f32 s0, s1
vmul.f32 s2, s3
```

```
vmul.f32 s4, s5
add r0, #32
vmul.f32 s6, s7

@ After reordering
vmul.f32 s0, s1
vmul.f32 s2, s3
add r0, #32
vmul.f32 s4, s5
vmul.f32 s6, s7
```

For the above example, assume that a processor has two floating-point multiplication and a single integer arithmetic pipelines. Before reordering, the first two multiplications, the third multiplication, the addition, and the forth multiplication can be issue to their respective patterns at each clock. It takes four clock to issue all of the instructions. After the reordering of the instructions, the last two multiplications can be issued together, reducing the issue clock count to three.

Finally, one other reason for instruction reordering is to diminish the impact of the cache replacement policy. An example of this is given below.

```
@ Before reordering
vld1.32 {q0-q1}, [r0]
vmul.f32 s0, s1
vmul.f32 s2, s3
vmul.f32 s4, s5
vmul.f32 s6, s7
vld1.32 {q2-q3}, [r1]
vmul.f32 s8, s9
vmul.f32 s10, s11
vmul.f32 s12, s13
vmul.f32 s14, s15

@ After reordering
```

```
vld1.32 {q0-q1}, [r0]
vmul.f32 s0, s1
vmul.f32 s2, s3
vld1.32 {q2-q3}, [r1]
vmul.f32 s4, s5
vmul.f32 s6, s7
vmul.f32 s8, s9
vmul.f32 s10, s11
vmul.f32 s12, s13
vmul.f32 s14, s15
```

For the above example, the second load operation is moved above to start earlier. If the address pointed by `r1` is not in the cache, the processor has more time to bring this data into cache until the execution begins at the fifth multiplication operation.

2.1.5 Digital Signal Processing Operations

2.1.5.1 Matrix Addition

Matrix Addition is the entry-wise summation of two equally sized matrices. Its definition is given in 2.6.

$$\mathbf{A} + \mathbf{B} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix} + \begin{bmatrix} b_{11} & b_{12} & \cdots & b_{1n} \\ b_{21} & b_{22} & \cdots & b_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ b_{m1} & b_{m2} & \cdots & b_{mn} \end{bmatrix}$$
$$= \begin{bmatrix} a_{11} + b_{11} & a_{12} + b_{12} & \cdots & a_{1n} + b_{1n} \\ a_{21} + b_{21} & a_{22} + b_{22} & \cdots & a_{2n} + b_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} + b_{m1} & a_{m2} + b_{m2} & \cdots & a_{mn} + b_{mn} \end{bmatrix}$$

Figure 2.6: Matrix Addition

2.1.5.2 Matrix Multiplication

For the matrices A and B that are defined as follows:

$$\mathbf{A} = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1m} \\ a_{21} & a_{22} & \cdots & a_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nm} \end{pmatrix}, \quad \mathbf{B} = \begin{pmatrix} b_{11} & b_{12} & \cdots & b_{1p} \\ b_{21} & b_{22} & \cdots & b_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ b_{m1} & b_{m2} & \cdots & b_{mp} \end{pmatrix}$$

Matrix multiplication, $C = AB$ is defined to be the matrix:

$$\mathbf{C} = \begin{pmatrix} c_{11} & c_{12} & \cdots & c_{1p} \\ c_{21} & c_{22} & \cdots & c_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ c_{n1} & c_{n2} & \cdots & c_{np} \end{pmatrix}$$

such that,

$$c_{ij} = a_{i1}b_{1j} + \cdots + a_{im}b_{mj} = \sum_{k=1}^{m} a_{ik}b_{kj}.$$

Matrix multiplication appears in many field of science, such as applied mathematics, statistics, physics, economics, and engineering.

2.1.5.3 Convolution

Convolution is a mathematical operation defined on complex valued functions. If f and g are such functions, which are defined only for integer valued indexes, and undefined anywhere else, their convolution is defined as such:

$$(f * g)[n] = \sum_{m=-\infty}^{\infty} f[m]g[n-m]$$

Convolution is used in very wide variety of applications. In image processing, convolution filter is frequently used in edge detection process, and for adding blur to images. In analytical chemistry convolutional filters are used to improve the signal-to-noise ratio of spectroscopic data. In acoustics, reverberation is the convolution of the original sound with echoes from surrounding objects. In probability theory,

the probability distribution of the sum of two independent random variables is the convolution of their individual distributions.

Convolution is encountered very frequently in electrical engineering also, where the convolution of a signal with the impulse response of a linear time-invariant filter gives the output of the system. In the context of this thesis, f is usually the input signal, g is usually the filter parameters.

2.1.5.4 Fourier Transform

Fourier Transform decomposes a signal into its constituent frequencies. The Fourier Transform of a real-valued signal is a complex-valued function whose magnitude represents the amount of that frequency in the signal. Discrete Fourier Transform is a version of the transform in which the input signal and the resulting transform consist of equally-spaced samples. For the signal $\{x_n\} := x_0, x_1, \ldots, x_{N-1}$, its transform $\{\mathbf{X}_k\} := X_0, X_1, \ldots, X_{N-1}$ is defined as:

$$X_k = \sum_{n=0}^{N-1} x_n \cdot e^{-\frac{i2\pi}{N}kn}$$

Computation of Discrete Fourier Transform consists of $O(N^2)$ multiplications. On the other hand, Fast Fourier Transform is an algorithm that computes the Discrete Fourier Transform of a signal with only $O(Nlog(N))$ multiplications. When the definition of Discrete Fourier Transform is expanded, it appears that the transform can be expressed as a weighted addition of two half-sized transforms. Fast Fourier Transform takes advantage of this fact and formulates the computation as a recursive computation of Fast Fourier Transforms. Computation of an 8-point FFT is visualized in figure 2.7.

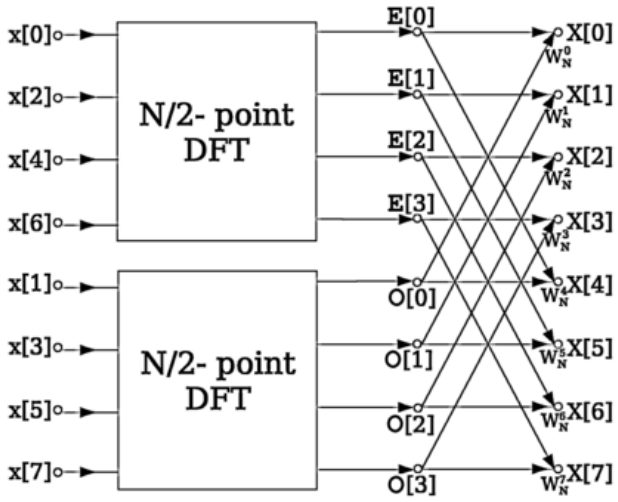

Figure 2.7: 8-point Fast Fourier Transform expressed in terms of two 4-point transforms and a weighted sum

2.1.6 Ne10 Library

Ne10 is a software library that contains a set of algebraic computations and DSP functions. Ne10 is the recommended library that is developed by ARM for its NEON-enabled architectures. It incorporates many vector/matrix operations such as addition, subtraction, multiplication, normalization, dot product, cross product, and determinant calculation. Besides, it has functions related to DSP operations for FFT, FIR filters, and IIR filters. This library is selected as the benchmark for our comparisons in our thesis, whenever possible.

2.2 Literature Review

2.2.1 Vectorization

Vectorization has been one of the most popular performance optimization techniques. In the 1970's, vector processors, which are specifically designed to take advantage of the vectorization techniques, began to appear. Around this time, auto-vectorization research has also began [8]. Later in 1990's, the vector processors could not keep

up with the conventional microprocessors and rapidly disappeared from the market. However, vectorization extensions to conventional instruction set architectures are adopted by major chip manufacturers. These instruction set extensions are often called as Single-Instruction-Multiple-Data (SIMD) instructions. IBM's AltiVec [4], Intel's AVX [5], and ARM's NEON [6] are examples of these extensions.

Even though auto-vectorization research has been going on for nearly 50 years, it is still not mature and new implementation techniques appear frequently, for example [9], [10], and [7]. The compilers adopt these techniques to keep up with performance requirements of new applications. However, not all effort in auto-vectorization is targeted towards compilers. Hybrid or hardware based solutions are also proposed as in [3] and [11] respectively. The hybrid auto-vectorization method proposed by [3] is visualized in 2.8. However, these solutions did not find their way into commercially available systems of devices. Another approach is binary translation, proposed in [12], [13], and [14]. These methods try to convert the vectorized code generated for one architecture to another without the source code. Again, these solutions are not encountered frequently in the industry.

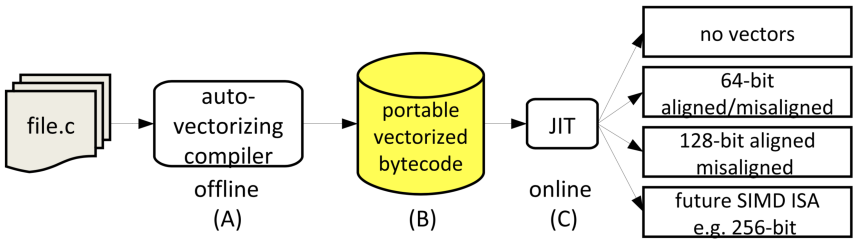

Figure 2.8: Hybrid compile-time run-time auto-vectorization proposal by [3]

Nevertheless, offline optimizations and manual vectorization is still considered to be superior to that of automatically generated ones [3]. There are many research papers focusing on manual optimization of widely used algorithms with vectorization. Such algorithms include H.264 decoding [15] [16] [17], HEVC decoding [18] [19], and FFT [20] [21] [22]. There are also SIMD libraries that provide a basic set of tools that programmers can use in their programs, such as [23]. Some SIMD libraries are specialized on certain topics such as image processing [24] [25].

Current state of the vectorization technology from programmer's perspective can be summarized as follows:

- The compilers can detect certain vector operations and generate SIMD-enabled code automatically.
- If the performance of the auto-vectorized code does not satisfy a programmer, the current state of the literature could be investigated. If the algorithm being implemented is found in one of the manual optimization research efforts, the techniques mentioned in the research can be applied.
- If a research effort concerning the algorithm under development cannot be found, the programmer can use architecture-dependent vector libraries that provide basic mathematical tools and implement the operation using the primitives found in those.
- If the algorithm cannot be expressed in terms of primitives found in vector libraries and the performance of the libraries do not satisfy the needs, the programmer must implement the algorithm with architecture-dependent statements using SIMD intrinsics or instructions.
- There is no guarantee that the vectorized implementation will always perform better than the sequential one. The programmer should perform performance tests.

2.2.2 Instruction Scheduling

The Instruction Scheduling research began with the appearance of out-of-order superscalar processors. At first, the out-of-order executing processors dynamically reordered the instructions to allow for the instructions that are not waiting for the result of previous calculations to start executing before others [26]. Multiple execution pipelines of superscalar processors even allowed for the parallel execution of independent instructions.

Historically, instruction scheduling is applied in two different ways. The first was dynamic scheduling of instructions in which the code is written sequentially and it is

reordered to execute in parallel by the processor at run-time. The second method was using an entirely different processor architecture called Very Large Instruction Word (VLIW) computers, which allowed the programmer, or compiler to decide which instructions should be issued to which pipeline in every cycle [27].

While the processors applying dynamic reordering, also known as out-of-order processors, did not break the compatibility with the existing binaries and compilers, VLIW processors required recompilation of the source along with a very complicated compiler [28]. Additionally, the out-of-order processors could, in time, expand their parallel execution capacity without breaking compatibility and providing more performance to already compiled programs. On the other hand, each iteration of a VLIW processor required a new revision of the compiler along with a recompilation.

For the given reasons, today, the VLIW processors can be encountered only in specialized applications, such as scientific computing. For desktop and mobile use, out-of-order superscalar processors dominate the market.

2.2.3 Fast Fourier Transform

Fast Fourier Transform is possibly the most frequently used algorithm in signal processing applications. It is discovered in 1965 by Cooley and Tukey [29]. Since then, it is adopted widely, modified to better fit into different architectures [30], variants have came up [31] [32], and many hardware & software implementations are developed. For low-cost architectures, sequential and vectorized implementations are also developed [21] [20].

One software library that tries to adopt the solutions of many researchers in one framework is the FFTW library [33]. When a dedicated hardware for FFT computation is not found in a system, FFTW is almost always the choice for the software alternative. This is due to the free availability of the library, and to its superiority against other free alternatives. A comparison of FFTW against other FFT libraries can be found in figure 2.9. For this comparison, a 2.4GHz Pentium 4 computer is used.

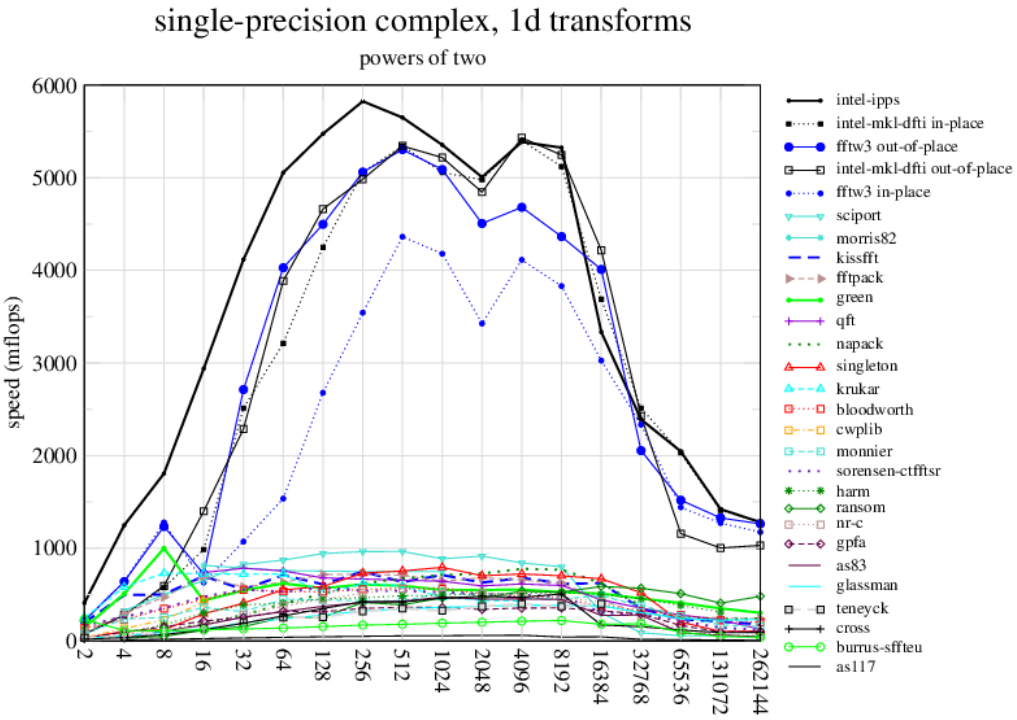

Figure 2.9: Single-precision floating-point performance of FFTW against other FFT implementations

Unfortunately, FFTW had an incompatibility with the compiler used in this thesis, which is GCC version 6.2.1, and it was not possible to use it for our comparisons. The FFTW maintainers are informed about this issue but no further comments have arrived yet.

CHAPTER 3

DETERMINING THE COST OF ARM AND NEON INSTRUCTIONS IN CLOCK CYCLES

We envisioned that knowing the cost of each instruction in clock cycles should be beneficial while reordering the instructions, either manually or automatically. However, this information is unfortunately not available in any official ARM documentation. Thus, we propose and implement an approach to determine the cost of each instruction in clock cycles.

One hypothetical case where knowing the cost of the instructions in clock cycles is essential while reordering is presented below. For this example, assume that the processor has two hypothetical instructions `add` and `mul`. In the given code, the `add` instructions are independent of each other but the `mul` instructions are not, and must wait for the other before starting executing.

```
add s0, s1
add s2, s3
add s4, s5
add s6, s7
add s8, s9
add s10, s11
add s12, s13
add s14, s15
mul r0, r1
mul r0, r2
mul r0, r3
```

The data dependency of `mul` instructions can be reduced by placing them as such:

```
add s0, s1
add s2, s3
add s4, s5
add s6, s7
add s8, s9
add s10, s11
mul r0, r1
add s12, s13
mul r0, r2
add s14, s15
mul r0, r3
```

However, we cannot know if this configuration is the optimal one. The previous `mul` instructions may still be executing while next one is issued.

If we assume that the `add` instruction executes in three clock cycles while the `mul` executes in five, the timings of the instructions of the unordered code will be as in figure 3.1.

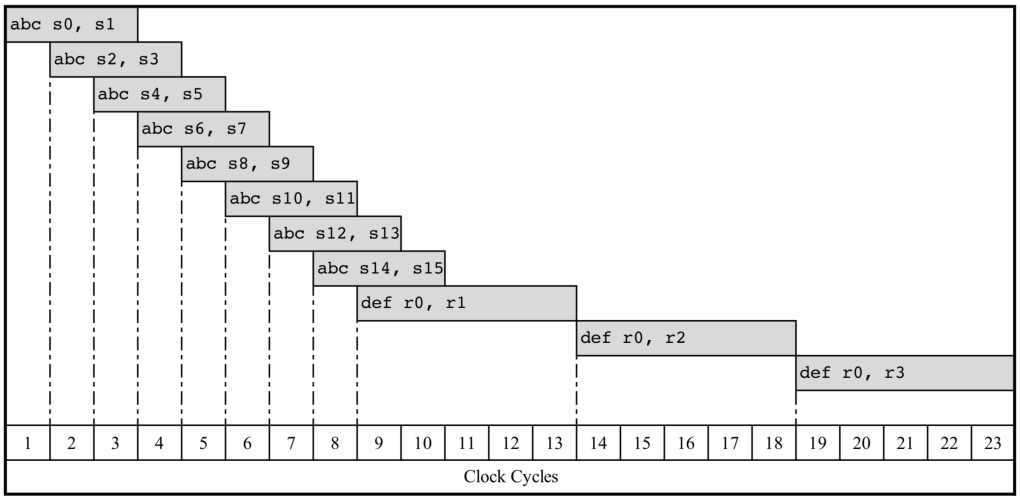

Figure 3.1: Timings of the instructions of the unscheduled code

Once the cost of the instructions in clock cycles are known, the code can be reordered

optimally as follows, which results in the timings given in 3.2.

```
mul r0, r1
add s0, s1
add s2, s3
add s4, s5
add s6, s7
mul r0, r2
add s8, s9
add s10, s11
add s12, s13
add s14, s15
mul r0, r3
```

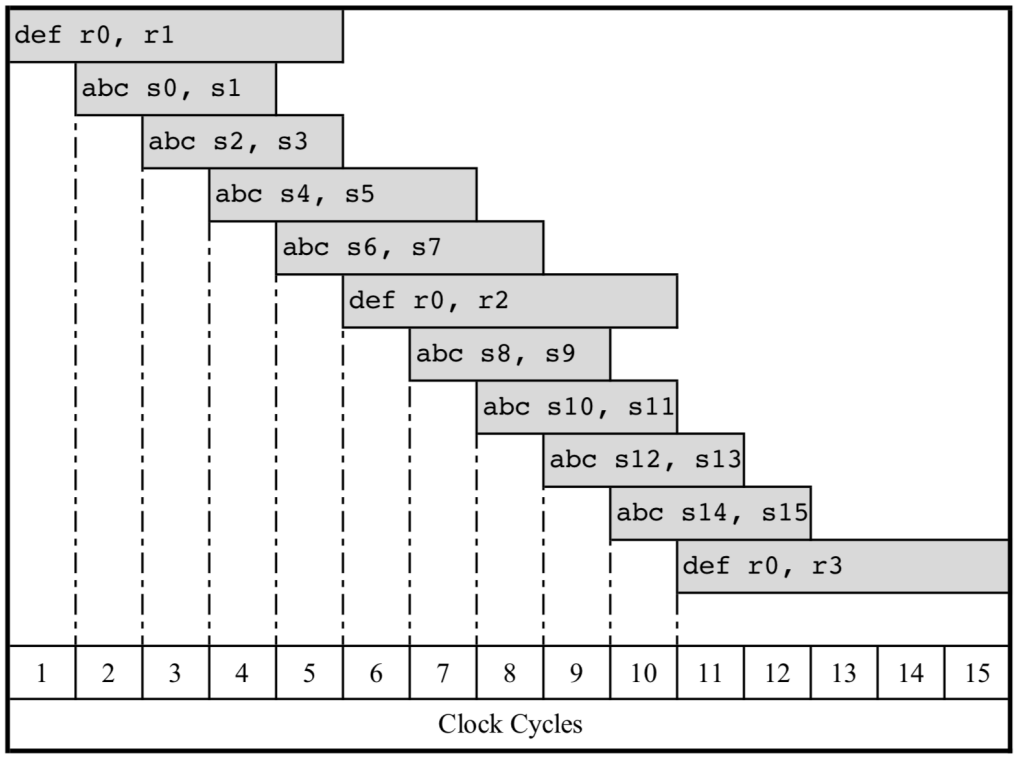

Figure 3.2: Timings of the instructions of the scheduled code

Consequently, there are cases where the cost of the instructions in clock cycle is essential if optimal reordering is intended.

3.1 Test Setup

The principal work conducted within the scope of this thesis required some architectural discoveries on an actual hardware. Therefore, we believe it is appropriate to explain our test setup early in this section of this chapter.

All of the code written for this thesis work is executed on the chip 66AK2H14 by Texas Instruments, which features four ARM Cortex-A15 cores running at 1.4GHz, 32KB of L1 instruction and data caches per core and 4MB of shared L2 cache.

The chip is found on an evaluation board EVMK2H designed by Advantech, which included a flash memory for the operating system and 4GB DDR3 memory. The evaluation board came preloaded with a GNU/Linux distribution with kernel version 4.9.41.

Texas Instruments provided the cross-compilation toolchain which included the GNU Compiler Collection (GCC) version 6.2.1. Most of the code in this thesis is written in C language with occasional assembly blocks for manual optimizations.

3.2 Measurement Method

To measure the number of clock cycles an instruction takes, should be able to get very accurate clock cycle measurements in the first place. However, the C language or the operating system does not provide such accurate timers. The C language standard library has the `time()` function which returns the number of seconds since Epoch (January 1, 1970 00:00), which is unusable for our case because of its one second granularity. The operating system has the `clock_gettime()` function which provides timing information with nanoseconds precision but its granularity is measured to be approximately 400 nanoseconds, which roughly corresponds to 560 clock cycles. Thankfully, the ARM chip used during this thesis included a cycle counter that is explained in detail in the following subsection.

There are other problems associated with cycle measurement when it comes to superscalar out-of-order processors with deep pipelines. Suppose that the Instruction

Under Investigation (IUI) is placed in between two measurements. First of all, in superscalar out-of-order executing processors, there is no guarantee that the IUI will be executed after the first and before the second measurement, especially if the instructions do not have data dependencies. Second, in pipelined processors, when the pipeline fills up with successive instructions, an instruction is retired in almost every cycle. Therefore, even if the IUI executes in between measurements, the measurement may be much smaller than the actual cost of that instruction. The ARM instruction `ISB` turned out to be the solution to the problems discussed in this paragraph. This instruction and the way we used it is discussed further in this chapter.

3.2.1 PMU Cycle Counter

The ARMv7-A Architecture adopted by the Cortex-A15 Processor allows the chip manufacturers to include a cycle counter named Performance Monitor Unit (PMU) in their designs. PMU is accessible by the user-space applications if configured as such. Requiring no operating system intervention further increases the granularity of the timing measurements.

After processor reset, the PMU is inaccessible from the user-space. The user-space access should be enabled by a privileged code, such as the kernel. Since the kernel does not incorporate such a functionality, a kernel module is written. This module, upon initialization, configures all cores to enable user-space access to PMU registers. With this configuration, the user-space programs gain the ability to read the current value of the counter or reset the counter to zero. The value of the counter is read with the *Move to ARM Core Register from Coprocessor (MRC)* instruction:

```
mrc p15, 0, r0, c9, c13, 0
```

This instruction reads the `c13` register in the `c9` bank of the coprocessor named as `p15` into the ARM register `r0`.

PMU can provide measurements with a single clock cycle granularity, however, the reading itself takes 8 clock cycles. This overhead must be subtracted from all further measurements.

Since the PMU cycle counter counts the number of clock cycles, its results are very stable for fine-grained measurements. However, for course-grained measurements, the measurements become less stable and can provide different results each time. Main reasons for this are the state of the instruction and data caches, the state of the pipeline, or the intervention of the operating system to the execution of the program for scheduling purposes. Handling of the results in course-grained measurements are described in 5.3.

The measurements done in this chapter is very fine-grained and showed very stable results. The intervention of the operating system between measurements is very rarely encountered. This is due to the huge difference between the measurement intervals and the intervention period of the operating system. The operating system intervenes with the processes once every millisecond. However, the measurements done in this chapter usually took at most 500 nanoseconds. This rough approximation alone shows that the probability of an operating system intervention between measurements in is 1 out of 2000 (0,05%).

3.2.2 ISB Instruction

ARMv7-A Instruction Set adopted by the Cortex-A15 Processor provides an instruction named Instruction Synchronization Buffer (ISB) which flushes the pipeline in the processor so that the instructions following ISB are fetched from the memory after all the existing pipelines are cleared.

This instruction is used to clear the pipeline before taking a measurement, which itself introduced an offset to the measurement because of the time it takes to fetch, decode and execute the ISB instruction. To diminish the effect of this error, instead of placing a single instance of the IUI between measurements, many copies of it are placed with an ISB instruction between each copy:

```
mrc p15, 0, r0, c9, c13, 0
...
add r2,r3
isb
```

```
        add r2,r3
        isb
        ...
        mrc p15, 0, r1, c9, c13, 0
```

The measurement result is divided by the number of repetitions to get the cost of each ISB-IUI pair. This cost includes the costs of fetch, decode and execute stages of the ISB and the IUI instructions. Since the fetch and decode costs are unknown, it was not possible to determine the execution time of an IUI with this measurement only.

In our work, the cycle cost measurement is performed for almost all data processing instructions. We observed the fastest and the slowest instructions to take 32 and 41 cycles, respectively. The official ARM documentation [2] states the longest execution pipeline to be 10 stages, and the shortest pipeline to be 1 stages. With this information, we can infer that the instructions measured as 32 cycles actually spend one clock in the execution stage, and the ones measured as 41 cycles actually spend ten clocks in the execution stage. This brings us to the conclusion that 31 clock cycles are spent on fetching and decoding of the instructions along with the execution of the ISB instruction. If we subtract 31 from all the measurements, we arrive at the actual execution duration of each instruction.

3.3 Measurement Results

The cost of select instructions are provided in tables 3.1, 3.2, 3.3, 3.4, and 3.5,

It is worth noting that the cost of an instruction may increase in the presence of a data dependency with nearby instructions. If an instruction requires the result of a previous instruction, it may cause a pipeline stall and wait until the previous result is ready.

Table 3.1: Cost of Integer Arithmetic Instructions

Instruction	Cost (in clock cycles)
`add Rd, Rs`	1
`add Rd, #IMM`	1
`add Rd, Rs1, Rs2`	1
`sub Rd, Rs`	1
`sub Rd, #IMM`	1
`sub Rd, Rs1, Rs2`	1
`eor Rd, Rs`	1
`eor Rd, Rs1, Rs2`	1
`mov Rd, Rs`	1
`mov Rd, #IMM`	1

Table 3.2: Cost of Integer Multiply/Divide Instructions

Instruction	Cost (in clock cycles)
`mul Rd, Rs`	2
`mul Rd, Rs1, Rs2`	3
`mla Rd1, Rs1, Rs1, Ra`	3
`umull Rd1, Rd2, Rs1, Rs2`	4
`umlal Rd1, Rd2, Rs1, Rs2`	4
`udiv Rd, Rs`	5
`udiv Rd, Rs1, Rs2`	5

Table 3.3: Cost of NEON Arithmetic Instructions

Instruction	Cost (in clock cycles)
`vadd.f32 Qd, Qs`	5
`vadd.f32 Qd, Qs1, Qs2`	5
`vsub.f32 Qd, Qs`	5
`vsub.f32 Qd, Qs1, Qs2`	5
`vand.f32 Qd, Qs`	3
`vand.f32 Qd, Qs1, Qs2`	4

Table 3.4: Cost of NEON Multiplication Instructions

Instruction	Cost (in clock cycles)
`vmul.f32 Qd, Qs`	4
`vmul.f32 Qd, Qs1, Qs2`	4
`vmla.f32 Qd, Qs`	7
`vmla.f32 Qd, Qs1, Qs2`	8

Table 3.5: Cost of NEON Load/Store Instructions

Instruction	Cost (in clock cycles)
`vld1.32 {Dn,Dn+1}, [Rd]`	7
`vld1.32 {Dn,Dn+1}, [Rd]!`	7
`vld2.32 {Dn,Dn+1,Dn+2,Dn+3}, [Rd]`	9
`vld2.32 {Dn,Dn+1,Dn+2,Dn+3}, [Rd]!`	10
`vst1.32 {Dn,Dn+1}, [Rd]`	4
`vst1.32 {Dn,Dn+1}, [Rd]!`	6
`vst2.32 {Dn,Dn+1,Dn+2,Dn+3}, [Rd]`	10
`vst2.32 {Dn,Dn+1,Dn+2,Dn+3}, [Rd]!`	9

CHAPTER 4

OPTIMIZING CORE DSP FUNCTIONS

4.1 Matrix Addition

Matrix Addition is the simplest operation examined in this thesis. During the implementation of this operation, the development cycle is established, compile and deploy scripts are written, measurement methods are decided upon, and the functional verification tactics are developed.

4.1.1 Canonical Implementation

First of all, matrix addition is implemented using only architecture-independent statements in C language. During the implementation, the auto-vectorizable loop examples in [34] are taken into consideration, and the implementation is made to resemble the examples given there.

```
for (int y = 0; y < Y_count; ++y)
  Y[y] = A[y] + B[y];
```

It is worth mentioning that the matrices are accessed as if they were single-dimensional arrays. For matrix addition, this does not impose any difference, and the generated code is the same as if they were accessed in two dimensions.

This first implementation can be considered as the canonical implementation because it is simple to read and understand. It is possibly the easiest implementation because of its straightforwardness. One might say that it is the cheapest piece of matrix addition code a programmer might write in C, in terms of development cost.

In the dictionary [35], it is stated that if something has a canonical status, it is accepted as having all the qualities that a thing of its kind should have. In the context of this thesis, the implementations denoted as canonical have all the qualities that a thing of its kind should have, *and nothing more*. This small addition to the meaning signifies that, in this thesis, the canonical implementations are the cheapest implementations of an algorithm that still works correctly. The cheapness of a code does not have a widely recognized meaning. Here, it is used for the code that is either implemented *quickly* by an experienced programmer or *correctly* by an inexperienced programmer.

4.1.2 Vectorization

After the canonical implementation, the architecture-independence of the code is broken with manual vectorization operation. The way it is done is by inserting inline assembly statements. Since the operation in hand can easily be represented in one-dimensional vector operations, the first optimized implementation used the vector coprocessor NEON. Here is the SIMD segment of the code:

```
vld1.32 {d0,d1,d2,d3}, [r1]!
vld1.32 {d4,d5,d6,d7}, [r2]!
vadd.f32 q0, q2
vadd.f32 q1, q3
vst1.32 {d0,d1,d2,d3}, [r0]!
```

Four computations are made in every iteration of the loop to utilize NEON efficiently. In order not to deal with edge cases, the measurements are done only with data sizes that have a multiple of four elements. After this restriction is established, it seems fair that the compiler should be given another chance because this restriction puts an alignment requirement on the data. If the compiler knows that the data will be aligned to a certain number of elements, it does not need to generate the code that handles the edge cases of vectorization. The restriction on the number of elements is delivered to the compiler by unrolling the loop in the canonical implementation by a factor of four. After a performance gain is observed in the code automatically vectorized by the compiler, we decided that the unrolling of the canonical implementation should be repeated after every unrolling performed on the manually optimized code.

4.1.3 Loop Unrolling

Next, loop unrolling is performed on the vectorized code. The loop is unrolled until there are no NEON registers left to use, which resulted in a loop unrolling factor of 32. The same number of unrolling is applied to architecture-independent code, too.

While unrolling the loop, the factor of unrolling is an important parameter. As this factor increases, up to a certain point, the code gets faster because of the decreased number of condition and branching instructions. After a certain point, the code becomes so large that instruction cache misses start to occur, causing a drop in the performance. Furthermore, as the unrolling factor increases, the restriction on the size of the data also increases. Meaning that, for an unrolled loop of 1024 repetitions, the data to be computed must have a multiple of 1024 items, which is not as flexible.

Another point of consideration is the number of registers reused while unrolling. While doing matrix addition, four numbers from the first matrix are loaded into a NEON quad register, and four numbers from the second matrix are loaded into another NEON quad register. Then, the first register is accumulated with the contents of the second register, meaning that the result lay in the first register. For the addition of 4 numbers, only two quad registers are used. With 16 quad registers in hand, 32 numbers can be added without any register reuse.

The loop is rolled four more times to introduce register reuse, but this showed no degradation in performance. For our case, register reuse is not an issue because there is plenty of code between the reusing of the registers. Register reuse becomes an issue for calculations that use more registers, or for calculations that take many numbers of cycles that the previous calculation on the same register might not be finished. This claim can be proved very easily by using the same set of registers while unrolling the loop, instead of utilizing the free registers.

4.1.4 Instruction Reordering

Following the above mentioned optimizations, instruction reordering techniques are given a try. The analysis of the code revealed some opportunities for reordering, how-

ever, no net performance gain is observed when it is applied. Hence matrix addition is considered to be very simple to provide any gain using instruction reordering.

The final state of the manually optimized code is as follows:

```
@ Data dependency of load
@ instructions is reduced
vld1.32 {d0,d1,d2,d3}, [r1]!
vld1.32 {d4,d5,d6,d7}, [r2]!
vadd.f32 q0, q2
vadd.f32 q1, q3
@ Store operation can safely began here
vst1.32 {d0,d1,d2,d3}, [r0]!
vld1.32 {d8,d9,d10,d11}, [r1]!
vld1.32 {d12,d13,d14,d15}, [r2]!
vadd.f32 q4, q6
vadd.f32 q5, q7
vst1.32 {d8,d9,d10,d11}, [r0]!
```

In table 4.1, the optimizations applied on the Matrix Addition code is shown. All these implementations are compared against each other and results of only the most performant one in each category is provided in Chapter 5.

Table 4.1: Optimizations Performed on Matrix Addition Code

Architecture-independent
Canonical (no optimization)
Canonical + Loop Unrolling (2 times)
Canonical + Loop Unrolling (4 times)
Canonical + Loop Unrolling (8 times)
Canonical + Loop Unrolling (16 times)
Canonical + Loop Unrolling (32 times)
Architecture-dependent
Vectorization (8-way)
Vectorization + Loop Unrolling (16 times)
Vectorization + Loop Unrolling (32 times)

4.1.5 Ne10

Ne10 library is compiled for the target board used in this thesis. Matrix addition is implemented in Ne10 as a combination of 4-by-4 matrixes. Thus, the matrix size in hand should be a multiple of 16.

As a result of our performance evaluation study, whose results are provided formally in Chapter 5, we observed that Ne10 implementation is 1.03 to 2.5 times slower than that of our manually optimized code. Ne10 implementation is then examined to find a possible cause for this. We discovered that Ne10 implementation used a different ordering of instructions than that of our implementation. In Ne10's implementation, loading of registers is done consecutively. Although one might believe that this ordering fills the pipeline of the processor optimally, one thing to note is the usage of post-increment variant of the load instructions. Ne10 used these load instructions with a post-increment flag, meaning that the pointer holding the location of the data is to be incremented *after* the data is loaded. Thus, every load instruction waits for the previous one to finish because the address register is incremented at the *end* of the execution cycle, in the write-back stage. We believe the authors of Ne10 were aware of this aspect of the load instructions, but readability is preferred above performance

in this case. We will see similar examples of the same preference in the following sections for other functions also.

Ne10 implementation of four-by-four matrix addition is provided below:

```
@ Load matrices
@ r1 points to first matrix array
@ r2 points to second matrix array
vld4.32 {d0, d2, d4, d6}, [r1]!
vld4.32 {d1, d3, d5, d7}, [r1]!
vld4.32 {d16, d18, d20, d22}, [r2]!
vld4.32 {d17, d19, d21, d23}, [r2]!
@ Calculate values
vadd.f32 q12, q0, q8
vadd.f32 q13, q1, q9
vadd.f32 q14, q2, q10
vadd.f32 q15, q3, q11
@ Store the results
@ r0 points to destination array
vst4.32 {d24, d26, d28, d30}, [r0]!
vst4.32 {d25, d27, d29, d31}, [r0]!
```

The solution is to *reordering* the instructions to reduce data dependency. If the calculation starts as soon as the necessary data is loaded, the forthcoming load instructions are not blocked by the previous ones. Also, as explained above, it is beneficial to break up the load instructions that use the same register as the address pointer, if the post-increment variant is used. This statement is valid for store operations as well. It is observed that our implementation included a simplest form of unintentional instruction reordering anyway.

4.2 Matrix Multiplication

This section consists of three subsections; Four-by-four Real-Valued Matrix Multiplication, Real-Valued Matrix Multiplication, and Complex-Valued Matrix Multipli-

cation. The reason behind the addition of the "Four-by-four Real-Valued Matrix Multiplication" category is that it is the only matrix multiplication method supported by the Ne10 library. For that category, the comparison was made between the Ne10 implementation and the manually optimized code. For generic sized implementations, the comparison was made between the compiler generated code and the manually optimized code, where the comparison results can be found in Chapter 5 that presents the evaluation study.

4.2.1 Four-by-four Real-Valued Matrix Multiplication

Ne10 library has functions that can only multiply two-by-two, three-by-three, or four-by-four real-valued matrices. These functions take two arrays of four-by-four matrices as inputs. It multiplies the matrices which correspond to the same index in those two arrays with a simple for-loop. For the rest of the discussions in this subsection, the four-by-four version of these functions is selected. It is left to the programmer to express a generic matrix multiplication in terms of four-by-four multiplications.

Four-by-four matrix multiplication can be implemented on NEON very efficiently because the row and the column size of the matrix fit perfectly with the size of the NEON quad registers. A row-to-column multiplication can be made with a single instruction, without any iteration logic, for-loops, and condition checking.

The optimizations for this operation took the implementation of Ne10 as a basis. The implementation of Ne10 used macros to increase readability but showed high improvement potential at first glance. If we look at the load instructions in the beginning of the implementation, it can be observed that even a simple reordering reduces the data dependency between instructions.

```
@ Before
vld1.32 {q8-q9}, [r1]!     (1st load)
vld1.32 {q10-q11}, [r1]!   (2nd load)
vld1.32 {q0-q1}, [r2]!     (3rd load)
vld1.32 {q2-q3}, [r2]!     (4th load)
...
```

```
@ After
vld1.32 {q8-q9}, [r1]!
vld1.32 {q0-q1}, [r2]!
vld1.32 {q10-q11}, [r1]!
vld1.32 {q2-q3}, [r2]!
...
```

With this simple swap operation, we have enabled the third load instruction to get in front of the second load instruction and start executing right away. Otherwise, the second load instruction would have waited for the post-increment operation in the first load instruction to finish and stall the pipeline until then. In a way, we have mitigated the Read-after-write hazard, which might happen if a value is read just after it is written, up to a certain point. Even in this configuration, there is a pipeline stall since the load instruction takes 10 clock cycles to execute as measured in Chapter 3, but we have allowed the third instruction to begin execution earlier. For this piece of code, no further optimization can be applied to reduce the data dependency.

After the loading of the data, successive multiplication operations begin, followed by the loading of the second set of registers. The successive multiplication operations are already placed optimally, and no further optimization could be done.

```
...
@ Before
@ First multiplication group
vmul.f32 q12, q8, d0[0]
vmul.f32 q13, q8, d2[0]
vmul.f32 q14, q8, d4[0]
vmul.f32 q15, q8, d6[0]

vmla.f32 q12, q9, d0[1]
vmla.f32 q13, q9, d2[1]
vmla.f32 q14, q9, d4[1]
vmla.f32 q15, q9, d6[1]
```

```
vmla.f32 q12, q10, d1[0]
vmla.f32 q13, q10, d3[0]
vmla.f32 q14, q10, d5[0]
vmla.f32 q15, q10, d7[0]

vmla.f32 q12, q11, d1[1]
vmla.f32 q13, q11, d3[1]
vmla.f32 q14, q11, d5[1]
vmla.f32 q15, q11, d7[1]

@ Second load group
vld1.32 {q8-q9}, [r1]!
vld1.32 {q10-q11}, [r1]!
vld1.32 {q0-q1}, [r2]!
vld1.32 {q2-q3}, [r2]!

@ First store group
vst1.32 {q12-q13}, [r0]!
vst1.32 {q14-q15}, [r0]!
...
```

It is worth noting that, instead of storing the multiplication results to the memory right away, Ne10 authors chose to load the second set of data first. In addition, the second group of load instructions bear the same problem with the first group discussed above. However, in the case of the second load group, the instructions having data dependency can be placed further apart, in between the multiplication operations above them. Early submission of load instructions should help dealing with the cache replacement latency that occurs occasionally, as the memory loading begins earlier than required.

```
...
@ After
```

```
vmul.f32 q12, q8, d0[0]
vmul.f32 q13, q8, d2[0]
vmul.f32 q14, q8, d4[0]
vmul.f32 q15, q8, d6[0]
vld1.32 {q4-q5}, [r1]!
vmla.f32 q12, q9, d0[1]
vmla.f32 q13, q9, d2[1]
vmla.f32 q14, q9, d4[1]
vmla.f32 q15, q9, d6[1]
vld1.32 {q6-q7}, [r1]!
vmla.f32 q12, q10, d1[0]
vmla.f32 q13, q10, d3[0]
vmla.f32 q14, q10, d5[0]
vmla.f32 q15, q10, d7[0]
vmla.f32 q12, q11, d1[1]
vmla.f32 q13, q11, d3[1]
vld1.32 {q0-q1}, [r2]!
vmla.f32 q14, q11, d5[1]
vmla.f32 q15, q11, d7[1]
vld1.32 {q2-q3}, [r2]!
...
```

Notice that the data loaded from the address pointed by `r1` is loaded into another group of registers, thus they could have been placed anywhere in the above code. However, the same is not true for the instructions loading data from the address pointed by `r2`. Since they had to use the same registers as the destination, their placement is not completely independent of the multiply instructions placed around them.

After the multiplications, the first group of data is ready to be written back to memory. In Ne10's implementation, this is done sequentially again, with a data dependency between instructions. The store operations are followed by the second multiplication group.

```
...
@ Before
@ First store group
vst1.32 {q12-q13}, [r0]!
vst1.32 {q14-q15}, [r0]!

@ Second multiplication group
vmul.f32 q12, q4, d0[0]
vmul.f32 q13, q4, d2[0]
vmul.f32 q14, q4, d4[0]
vmul.f32 q15, q4, d6[0]
vmla.f32 q12, q5, d0[1]
vmla.f32 q13, q5, d2[1]
vmla.f32 q14, q5, d4[1]
vmla.f32 q15, q5, d6[1]
vmla.f32 q12, q6, d1[0]
vmla.f32 q13, q6, d3[0]
vmla.f32 q14, q6, d5[0]
vmla.f32 q15, q6, d7[0]
vmla.f32 q12, q7, d1[1]
vmla.f32 q13, q7, d3[1]
vmla.f32 q14, q7, d5[1]
vmla.f32 q15, q7, d7[1]
...
```

The store operations can be placed further apart as shown below. This placement increases the distance between store instructions having a data dependency. Otherwise, the second store instruction would block the execution of the multiplication operation below.

```
...
@ After
@ First store group
```

```
vst1.32 {q12-q13}, [r0]!

@ Second multiplication group
vmul.f32 q12, q4, d0[0]
vmul.f32 q13, q4, d2[0]
vst1.32 {q14-q15}, [r0]!
vmul.f32 q14, q4, d4[0]
vmul.f32 q15, q4, d6[0]
...
```

After these statements, one might point out that Cortex-A15 is already an out-of-order architecture with an internal instruction scheduling mechanism. One might ask why reordering operations such as the ones we carried out above should lead to performance improvement after all. We argue that the processor has a fixed window in which reordering could take place, however, we - as the programmers - can do reordering over the whole function. The performance evaluation performed on the above code demonstrates that 5 to 20 percent speed-up is observer after all. The results are given in detail in A.8.

One might ask why these reordering operations lead to performance improvement at all. We argue that the processor has a fixed window in which reordering could take place. We, as the programmer, can do reordering over the whole function.

4.2.2 Real-Valued Matrix Multiplication

Similar to Matrix Addition, Real-Valued Matrix Multiplication began with the canonical implementation.

4.2.2.1 Canonical Implementation

The first canonical implementation is given below.

```
for (int y = 0; y < Y_row_count * Y_col_count; ++y)
```

```
  Y[y] = 0;

for (int y_row = 0; y_row < A_row_count; ++y_row) {
  for (int y_col = 0; y_col < B_col_count; ++y_col) {
    for (int a_col = 0; a_col < A_col_count; ++a_col) {
      Y[y_row * B_col_count + y_col] +=
        A[y_row * A_col_count + a_col] *
        B[a_col * B_col_count + y_col];
    }
  }
}
```

When the memory access pattern of this code is analyzed, one can notice that the following term is introducing a huge performance hit.

```
B[a_col * B_col_count + y_col];
```

At every iteration of the innermost loop, the given portion of the code accesses addresses sequentially that can be very far away from each other when column count of B is high. This access pattern could cause a cache miss at *every* iteration. The loop can be rearranged to decrease the number of cache misses as follows:

```
for (int y = 0; y < Y_row_count * Y_col_count; ++y)
  Y[y] = 0;

for (int a_row = 0; a_row < A_row_count; ++a_row) {
  for (int a_col = 0; a_col < A_col_count; ++a_col) {
    for (int b_col = 0; b_col < B_col_count; ++b_col) {
      Y[a_row * B_col_count + b_col] +=
        A[a_row * A_col_count + a_col] *
        B[a_col * B_col_count + b_col];
    }
  }
}
```

With this configuration, at every iteration, Y and B arrays are accessed sequentially, resulting in a much smaller number of cache misses. Although in the above code it seems like array A is accessed at every iteration of the loop, when the compiler optimizations are enabled, even at its lowest level, the generated code does not access A each time, as its value does not change at every iteration of the innermost loop. Similarly, for the first for-loop, the compiler is smart enough not to compute `Y_row_count * Y_col_count` at every iteration.

4.2.2.2 Vectorization

As the first step of manual optimization, the operation is vectorized using NEON capabilities. This caused the code to become architecture independent. This step alone proved to be much faster than the auto-vectorization capabilities of the compiler. As before, in order not to deal with edge cases, a restriction is made on the size of the matrices such that their row or column count should be a multiple of four. The restriction is again communicated to the compiler by manual loop unrolling done on the architecture-independent code.

Below, the multiplication section of the code is given.

```
@ q0 holds the values loaded from the A array
@ r1 points to the array B
@ r2 points to the array Y

vld1.32 {d2,d3}, [r1]!
vld1.32 {d4,d5}, [r2]
vmla.f32 q2, q0, q1
vst1.32 {d4,d5}, [r2]!
```

4.2.2.3 Loop Unrolling

Loop unrolling is performed in two steps. After 16 factor unrolling, the performance did not improve, and no further unrolling is performed.

Below is the multiplication section of the final version of the code.

```
@ q0 holds the values loaded from the A array
@ r1 points to the array B
@ r2 points to the array Y

vld1.32 {d2,d3}, [r1]!
vld1.32 {d4,d5}, [r2]
vmla.f32 q2, q0, q1
vst1.32 {d4,d5}, [r2]!

vld1.32 {d6,d7}, [r1]!
vld1.32 {d8,d9}, [r2]
vmla.f32 q4, q0, q3
vst1.32 {d8,d9}, [r2]!

vld1.32 {d10,d11}, [r1]!
vld1.32 {d12,d13}, [r2]
vmla.f32 q6, q0, q5
vst1.32 {d12,d13}, [r2]!

vld1.32 {d14,d15}, [r1]!
vld1.32 {d16,d17}, [r2]
vmla.f32 q8, q0, q7
vst1.32 {d16,d17}, [r2]!
```

4.2.2.4 Instruction Reordering

No opportunities for instruction reordering is discovered for this function. A number of experiments without any structured approach provided no positive result.

In table 4.2, the optimizations applied on the Real-Valued Matrix Multiplication code is shown. All these implementations are compared against each other and results of only the most performant one in each category is provided in Chapter 5.

Table 4.2: Optimizations Performed on Real-Valued Matrix Multiplication Code

Architecture-independent
Canonical (no optimization)
Canonical + Loop Unrolling (4 times)
Canonical + Loop Unrolling (8 times)
Canonical + Loop Unrolling (16 times)
Architecture-dependent
Vectorization (4-way)
Vectorization + Loop Unrolling (8 times)
Vectorization + Loop Unrolling (16 times)

4.2.3 Complex-Valued Matrix Multiplication

Optimization of Complex-Valued Matrix Multiplication begins with the canonical form.

4.2.3.1 Canonical Implementation

The canonical implementation of Complex-Valued Matrix Multiplication is almost the same as Real-Valued Matrix Multiplication. The only difference is in the declaration part of the types of `A`, `B`, and `Y` pointers.

```
@ Real-Valued Matrix Multiplication
float *A, *B, *Y;

@ Complex-Valued Matrix Multiplication
float complex *A, *B, *Y;
```

4.2.3.2 Vectorization

As the first step of manual optimizations, the operation is vectorized using NEON capabilities. This caused the code to become architecture dependent. Although the

vectorization was as simple as the previous function, the vectorized version required twice the number of registers. Additionally, due to complex multiplication, four times the number of floating-point multiplications were needed. As before, in order not to deal with edge cases, a restriction is made on the size of the matrices such that their row or column count should be a multiple of four. The restriction is again communicated to the compiler by manual loop unrolling of the canonical implementation of the operation. However, the auto-vectorized code generated by the compiler did not benefit from this restriction at all, and thus those results are taken out of consideration.

```
vld2.32 {d4,d5,d6,d7}, [r1]!
vld2.32 {d8,d9,d10,d11}, [r2]
vmla.f32 q4, q0, q2
vmls.f32 q4, q1, q3
vmla.f32 q5, q0, q3
vmla.f32 q5, q1, q2
vst2.32 {d8,d9,d10,d11}, [r2]!
```

4.2.3.3 Loop Unrolling

Loop unrolling is performed with a factor of 8. Any other unrolling factor did not bring further performance gain. The state of the code at this stage is given below.

```
@ q0 holds the real values loaded from A
@ q1 holds the complex values loaded from A
@ r1 points to the array B
@ r2 points to the array Y

vld2.32 {d4,d5,d6,d7}, [r1]!
vld2.32 {d8,d9,d10,d11}, [r2]
vmla.f32 q4, q0, q2
vmls.f32 q4, q1, q3
vmla.f32 q5, q0, q3
vmla.f32 q5, q1, q2
vst2.32 {d8,d9,d10,d11}, [r2]!
```

```
vld2.32 {d12,d13,d14,d15}, [r1]! @ Mark
vld2.32 {d16,d17,d18,d19}, [r2]
vmla.f32 q8, q0, q6
vmls.f32 q8, q1, q7
vmla.f32 q9, q0, q7
vmla.f32 q9, q1, q6
vst2.32 {d16,d17,d18,d19}, [r2]!
```

4.2.3.4 Instruction Reordering

This function was one of the functions examined in this thesis showing that manual instruction reordering can lead to a net performance gain. When the marked instruction in the above code segment is examined, one might notice that it does not have any data dependencies with the instructions above or below it. However, when the marked instruction is moved above, as shown in the below code segment, a net performance gain is achieved. The reason for this is believed to be the cache replacement latency occurring from time to time. If the load instruction is issued earlier than required, the memory system has more time to bring the data that will be used in the calculation.

```
@ q0 holds the real values loaded from A
@ q1 holds the complex values loaded from A
@ r1 points to the array B
@ r2 points to the array Y

vld2.32 {q2-q3}, [r1]!
vld2.32 {q4-q5}, [r2]
vmla.f32 q4, q0, q2             @ (1st multiplication)
vmla.f32 q5, q0, q3             @ (2nd multiplication)
vld2.32 {q6-q7}, [r1]! @ Mark
vmls.f32 q4, q1, q3             @ (3rd multiplication)
vmla.f32 q5, q1, q2             @ (4th multiplication)
vst2.32 {q4-q5}, [r2]!
```

```
vld2.32 {q8-q9}, [r2]
vmla.f32 q8, q0, q6
vmla.f32 q9, q0, q7
vmls.f32 q8, q1, q7
vmla.f32 q9, q1, q6
vst2.32 {q8-q9}, [r2]!
```

One might wonder why the out-of-order executing processor such as Cortex-A15 would not reorder this instruction to start earlier. For architectures that use memory-mapped hardware registers, load/store instructions cannot be reordered by the processor automatically. The hardware registers need to be accessed in the way the programmer wants to. Otherwise, the associated hardware might behave unexpectedly. For this piece of program load/store instructions are used with addresses that are dynamically computed. This address might not be available during the reordering period. Thus the processor cannot know whether a given load/store instruction accesses a hardware register or the memory. It is believed that instead of finding a way around these complications associated with the rearrangement of load/store instructions, they are not reordered. Experimental results point to the same conclusion. On the other hand, we, as the programmer, are free to do such reordering since we are sure the accessed address does not belong to a hardware register.

The marked instruction showed a performance gain when moved in between any of the multiplication operations. However, the highest speed-up is observed when it is placed below the 2nd multiplication. This has to do with the instruction issue capability of the Cortex-A15. The NEON co-processor has two parallel pipelines. Thus two instructions can be issued to it in each cycle. To take advantage of this, the ordering of the multiplications is also altered to reduce the data dependency. With this configuration, the 1st and 2nd multiplications can be issued to NEON in the same cycle, similar to 3rd and 4th multiplication instructions. The marked instruction is placed in between these groups in order not to prevent this from happening.

In table 4.3, the optimizations applied on the Complex-Valued Matrix Multiplication code is shown. All these implementations are compared against each other and results

of only the most performant one in each category is provided in Chapter 5.

Table 4.3: Optimizations Performed on Complex-Valued Matrix Multiplication Code

Architecture-independent
Canonical (no optimization)
Architecture-dependent
Vectorization (4-way)
Vectorization + Loop Unrolling (8 times)
Vectorization + Loop Unrolling (8 times) + Instruction Reordering

4.3 Convolution

Convolution is simpler than the operations discussed above and below. This is due to the one-dimensionality of the chosen operation. To further increase the simplicity of the implementation, it is assumed that the Convolution will be used in the context described in 2.1.5.3, where the first operand is assumed to be the input signal and the second operand is assumed to be the impulse response of a linear time-invariant filter. Since the filter is time-invariant, its impulse response does not change as time advances. The implementations given below further assumes that the array which holds the impulse response of the filter is kept in memory in reverse order because such a configuration would increase the performance significantly.

4.3.1 Canonical Implementation

Canonical implementation of Convolution is given below:

```
for (int y = 0; y < X_SIZE - H_SIZE + 1; ++y) {
  float complex sum = 0;
  for (int i = 0; i < H_SIZE; ++i)
    sum += X[y + i] * H[i];
  Y[y] = sum;
}
```

In the given code segment, `X` is the input signal, `H` holds the impulse response of the filter in reverse order, and `Y` is the output signal. `X_SIZE` and `H_SIZE` are lengths of `X` and `H` arrays, respectively.

Manual unrolling of the canonical implementation did not bring any performance improvements.

4.3.2 Vectorization

The Convolution operation is implemented utilizing the NEON co-processor. This operation alone provided more speed-up than that of auto-vectorized code generated by the compiler. The innermost loop of the canonical implementation appears in the vectorized version as such:

```
vld2.32 {d4,d5,d6,d7}, [r1]!
vld2.32 {d8,d9,d10,d11}, [r2]!
vmla.f32 q0, q2, q4
vmls.f32 q0, q3, q5
vmla.f32 q1, q2, q5
vmla.f32 q1, q3, q4
```

In this code segment, `r1` corresponds to `X[y + i]`, `r2` corresponds to `H[i]`, and `q0-q1` hold the real and imaginary parts of `sum`.

4.3.3 Loop Unrolling

The vectorized code is unrolled four times to achieve 16 complex multiplications per iteration, since this much unrolling provided the maximum performance. Again, the edge cases are ignored, restricting the length of `H` to multiples of 16.

4.3.4 Instruction Reordering

A similar approach to instruction Reordering applied in Complex Matrix Multiplication is applied to the unrolled code. To take advantage of the two parallel NEON

pipelines in Cortex-A15, the instructions having data dependency are reordered as given:

```
vmla.f32 q0, q2, q4
vmla.f32 q1, q2, q5
vmls.f32 q0, q3, q5
vmla.f32 q1, q3, q4
```

Only the first block of multiplications is provided above. The same reordering is repeated for the rest of the unrolled code.

4.4 Fourier Transform

We decided that the simple optimization techniques employed during this thesis should be applied upon Discrete Fourier Transform, as it is one of the most frequently used, most studied, and most complex algorithms used in Digital Signal Processing applications. The DFT implementation of the Ne10 library is chosen as the competitor for this endeavor since it is one of the most recognized DSP library optimized for ARM processors.

First, the approach used in previous DSP functions studied in this thesis is applied. The canonical implementation is followed by manual vectorization, loop unrolling, and instructions reordering. For these trials, 16-point DFT was chosen with the hope of using it as a building block for higher-point FFT operations. Additionally, 16-point DFT is the smallest DFT calculation that can fully utilize the NEON co-processor.

When the results are obtained, it is observed that the 16-point DFT implementation of Ne10 was at least four times faster. When Ne10's implementation is analyzed, it is realized that our DFT implementation was algorithmically too weak to show the performance gained by the manual optimizations.

On the other hand, the existing literature shows that most of the research focuses on efficient partitioning of high-point FFT calculations. Almost all of the FFT implementations used building blocks that are implemented as DFTs, which are usually found in sizes of 8, 16, 32, or 64-points. Thus, instead of reinventing the wheel, we

decided to use Ne10's implementation of 16-point DFT as a starting point. For this function, the authors of Ne10 chose to go with the C language instead of the ARM assembly. The NEON co-processor is utilized by the help of NEON intrinsics. Vectorization and loop unrolling were already applied. When the output of the compiler is examined, we discovered that the compiler has also applied reordering to decrease data dependance between instruction. However, when the code is analyzed with the cache usage in mind, it is observed that the store instructions found at the end of the operation could be reordered to start earlier as to reduce the cache replacement latency.

```
@ Before
vadd.f32 q3, q1, q4
vsub.f32 q1, q1, q4
vadd.f32 q12, q10, q9
vadd.f32 q15, q14, q7
vsub.f32 q11, q10, q9
vsub.f32 q14, q14, q7
vadd.f32 q10, q13, q8
vsub.f32 q0, q13, q8
vadd.f32 q6, q15, q3
vsub.f32 q2, q15, q3
vadd.f32 q4, q14, q11
vadd.f32 q7, q10, q12
vsub.f32 q3, q10, q12
vsub.f32 q5, q0, q1
vsub.f32 q12, q14, q11
vadd.f32 q13, q0, q1
vst2.32 {d12-d15}, [r4]
vst2.32 {d8-d11}, [r0]
vst2.32 {d4-d7}, [r1]
vst2.32 {d24-d27}, [r2]

@ After
vadd.f32 q3, q1, q4
```

```
vadd.f32 q12, q10, q9
vadd.f32 q15, q14, q7
vsub.f32 q11, q10, q9
vsub.f32 q14, q14, q7
vadd.f32 q10, q13, q8
vadd.f32 q6, q15, q3
vadd.f32 q7, q10, q12
vst2.32 {d12-d15}, [r4]
vsub.f32 q1, q1, q4
vsub.f32 q0, q13, q8
vadd.f32 q4, q14, q11
vsub.f32 q5, q0, q1
vst2.32 {d8-d11}, [r0]
vsub.f32 q2, q15, q3
vsub.f32 q3, q10, q12
vst2.32 {d4-d7}, [r1]
vsub.f32 q12, q14, q11
vadd.f32 q13, q0, q1
vst2.32 {d24-d27}, [r2]
```

This modification gained us 5 clock cycles, which corresponds to 1.05 speed-up.

For 32 and higher-point FFT calculations, Ne10 provides one function that uses a different implementation of 16-point DFT as the building block. This implementation uses assembler macros to organize the code in such a way that enables mixed-radix FFT computation, as well as improve code readability. Even though Ne10 only supports FFT calculations of sizes in powers of 2, the authors chose not to generate separate code for higher-point DFTs, but use a generic one. This version of the code is not suitable for applying instruction reordering methods since the code blocks have branching instructions in between. This prevents reordering of instructions that can jump into neighboring blocks. No other type of reordering opportunities is discovered that can potentially lead to a performance gain.

CHAPTER 5

PERFORMANCE EVALUATION

Before giving the results of our comparative evaluations, we provide below the verification methods, the test setup, the measurement approach, and the performance metrics used.

5.1 Functional Verification of the Operations

During the development period, it is possible to make a mistake while optimizing the implementations which may result in an increased performance measurement but with incorrect computations. To make sure that every performance measurement is performed under correct working conditions, a functional verification method needs to be established. For the rest of the thesis work, we first performed the functional verification of the functions and after a pass, we carried out successive measurements on each to find out the speed-up or other metrics under investigation.

The following custom functional verification procedure is established. The test data is generated in GNU Octave and saved in Mat file format, since this format is relatively easy to parse. A function that can load this file into memory is written. The operation is carried out on the input data, and the output data is compared with the results. The verification is used every time the implementations are modified. Once the correctness of the functions are established, the verification is disabled, and timing measurements are then taken repeatedly.

5.2 Test Setup

The test setup described in Section 3.1 is used exactly as described there.

5.3 Performance Measurement Method

The execution time of the codes written during the course of this thesis does not depend on the input data, does not use any I/O devices or operating system calls. For this reason, the execution time of the code can be used as a performance measurement. If I/O devices were used, the performance of the I/O device might be a factor in the measurements. If operating system calls were used, the performance of the operating system or the context switching speed might be a factor in the measurements. If the execution time of the functions were dependent on the data, the distribution of it might be a factor in the measurements. Without any of these factors intervening with the results, only the execution time of the program is measured.

Even though the DSP functions implemented in this thesis do not have any operating system calls, they are still operating in the supervision of one. This means that the operating system will intervene with the process to do scheduling, other processes may be scheduled during the execution of the program, and these processes may bring in their own instruction and data to the processor caches, which in turn means that the execution time measurements will not necessarily be the same at every run. To provide meaningful and accurate performance metrics, the measurements should be interpreted reasonably.

A reasonable expectation from a performance measurement is that the improvement, if exists, should be observable by future researchers with noticeably high occurrence rate. The best performance, if observed very rarely, should not be reported. Similarly worst performance might also not be a correct representation of the gain a piece of work may bring.

The median of measurements is a commonly used benchmark since it is not sensitive to outliers. The intervention of the operating system should not count as an outlier since the operating system will always be there. However, network activity, disk ac-

tivity, and maintenance operations running in the background can count as outliers and be safely ignored. Luckily, for our system, we had complete control over the system and was able to stop all other non-essential processes that may generate network or disk activity. To include the operating system intervention on the program's performance, the measurements are taken over a longer time than a millisecond.

Touati et. al. [36] argues that when performance benchmarking tests are done using the same data, it is crucial to justify the measurements using statistical tests. Meaning, if system A is claimed to be more performant than system B, the probability at which such a result may not be encountered should also be provided. In [36] this probability is called as α. For this thesis, we chose α to be *at most* 0.05, i.e. if system A is claimed to be faster than system B, enough measurements will be made to ensure that system A performs better than system B 95% of the time and median of those many measurements is provided.

Ideally, α should be calculated after every measurement. This is a very time-consuming process. Instead of calculating the exact value of α after every measurement, it is calculated after many measurements. If the calculated value turns out to be smaller than 0.05, the median of the measurements is recorded. If not, the number of trials is increased, and measurement is repeated. This measure-calculate-increase-repeat loop is utilized until α drops below 0.05.

5.4 Using Speed-up as a Performance Metric

The speed-up is the ratio of the execution times of two implementations.

$$S = \frac{T_{impl1}}{T_{impl2}}$$

In this thesis, whenever two implementations of the same operation are compared, speed-up is used as the performance metric.

5.5 Per-cycle Performance Metrics

While designing a product, if the hardware is already fixed, it is meaningful to compare the execution times of two programs to decide on which one should be used. Alternatively, if the software is final and cannot be modified, it is meaningful to compare the execution times of the same software on different hardware to decide on which one should be used. However, it is meaningless to compare the execution times of two different implementations of a program running on two different hardware. To get around this problem, per-cycle metrics are generally used to compare the efficiency of implementations.

For example, let us say there is a program A1 that performs an operation. This program is optimized for processor P1. It takes t_1 seconds for this program to run on P1. Additionally, there is a program A2 that performs the same operation, which is optimized for the processor P2, and it takes t_2 seconds to run. It is meaningless to compare t_1 and t_2 if the efficiency of the programs are intended to be compared. If we could find an operation that dominates all other operations during the execution of the program, we could compare how many of those operations can be done in unit time. Furthermore, the processors can usually run at different clock speeds and the *unit time* concept might be different from system to system. To compensate for the effect of clock speed, we can use the number of dominant operations done in a single clock cycle. With this value in hand, we can estimate the approximate performance of a processor running on different speeds, even further, we can look at the possible hardware choices and their clock specifications, and estimate which hardware will result in a better performance.

5.5.1 Using Additions Per Cycle as a Performance Metric

The number of floating-point additions per clock cycles is given for those operations where addition is the dominant operation in a function, such as matrix summation.

5.5.2 Using Multiply-Accumulate Operations Per Cycle as a Performance Metric

Floating-point multiplications followed by accumulation operation is frequently encountered in matrix operations. In this thesis, the number of multiply-accumulate operations is provided whenever meaningful.

5.5.3 Using Butterfly Operations Per Cycle as a Performance Metric

The dominant operation in Fast Fourier Transform is the butterfly operation visualized in 5.1. It includes one complex multiply-accumulate and one complex multiply-subtract operations. For DFT calculations, the number of butterfly operations is also provided.

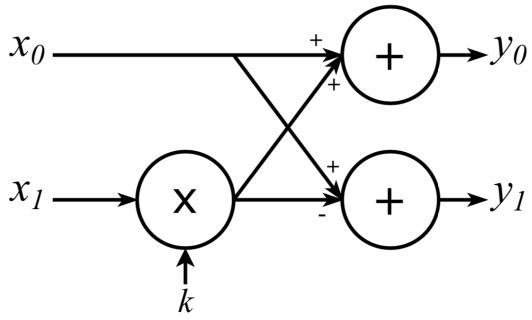

Figure 5.1: Butterfly Operation

5.6 Comparison Results

In this section the comparison of results are provided. Exact measurement results are provided in Appendix A. The results are categorized according to the implemented operations. The results of the Ne10 library are also provided whenever applicable.

5.6.1 Matrix Addition

The speed-up and addition per cycle results of the optimizations performed on Matrix Addition implementations are provided in figures 5.2, 5.3, and 5.4, along with the results of Ne10's implementation. The results that are obtained for canonical implementations are for architecture-independent C code that is compiled with auto-vectorization, loop unrolling, and instruction scheduling optimizations enabled. The speed-up is computed against *Canonical + Unroll (x32)* version.

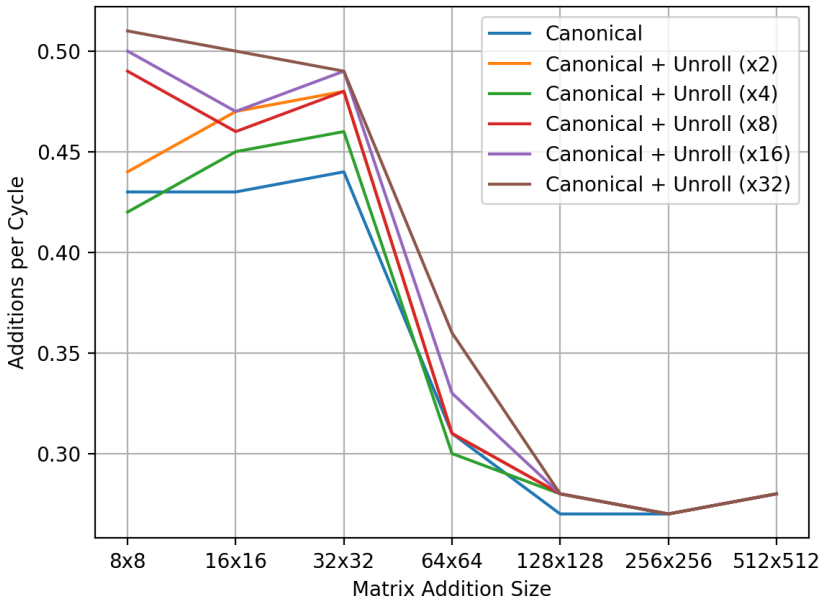

Figure 5.2: Addition per Cycle Results of Optimizations Performed on Matrix Addition (Part I)

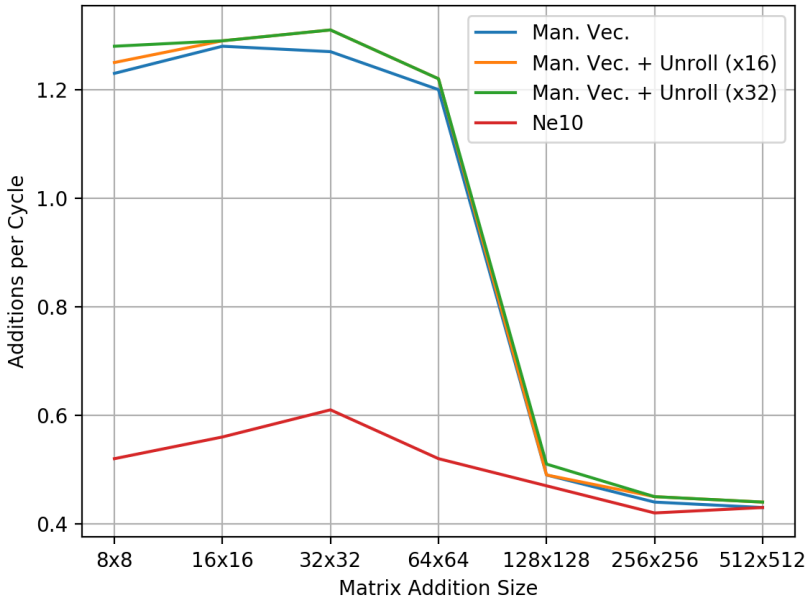

Figure 5.3: Addition per Cycle Results of Optimizations Performed on Matrix Addition (Part II)

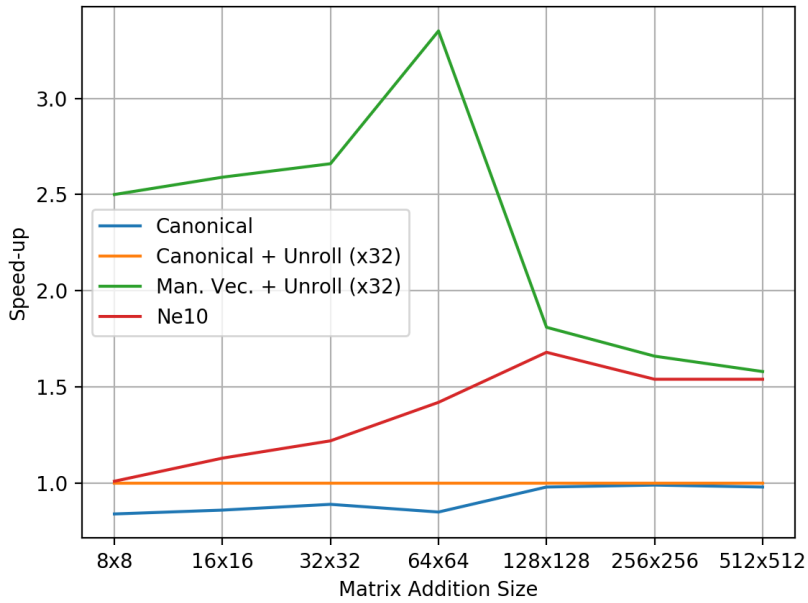

Figure 5.4: Speed-up Results of Optimizations Performed on Matrix Addition

5.6.2 Matrix Multiplication

5.6.2.1 Four-by-Four Real-Valued Matrix Multiplication

The speed-up and addition per cycle results of the optimizations performed on Four-by-Four Real-Valued Matrix Multiplication implementations are provided in figures 5.5, and 5.6, along with the results of Ne10's implementation. The speed-up is computed against *Ne10* version.

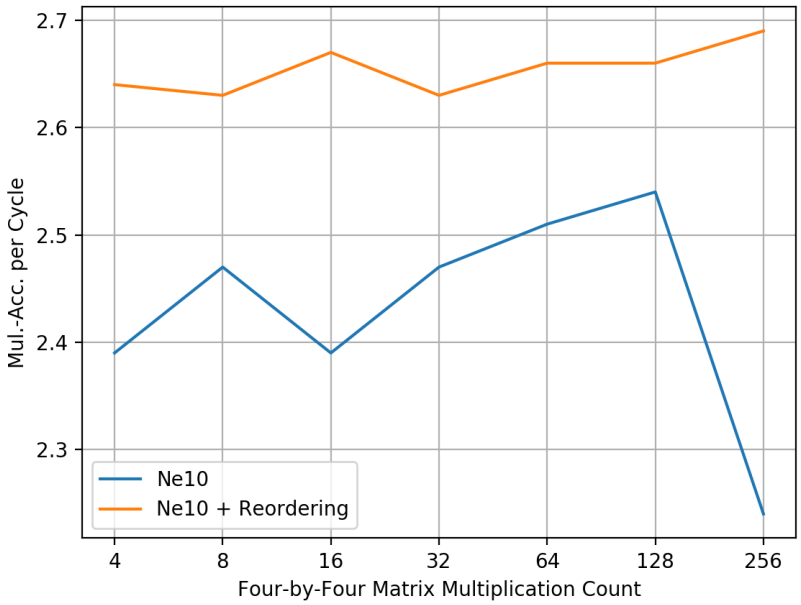

Figure 5.5: Mul.-Acc. per Cycle Results of Optimizations Performed on Four-by-Four Real-Valued Matrix Multiplication

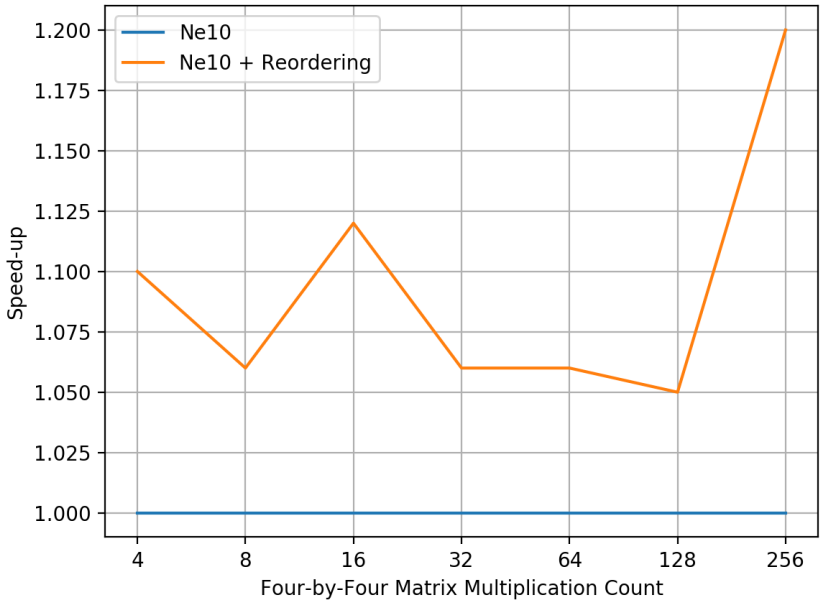

Figure 5.6: Speed-up Results of Optimizations Performed on Four-by-Four Real-Valued Matrix Multiplication

5.6.2.2 Real-Valued Matrix Multiplication

The speed-up and addition per cycle results of the optimizations performed on Real-Valued Matrix Multiplication implementations are provided in figures 5.7, 5.8, and 5.9. The results that are obtained for canonical implementations are for architecture-independent C code that is compiled with auto-vectorization, loop unrolling, and instruction scheduling optimizations enabled. The speed-up is computed against *Canonical + Unroll (x4)* version. Ne10 does not appear in these figures since it does not support generic sized matrix multiplication.

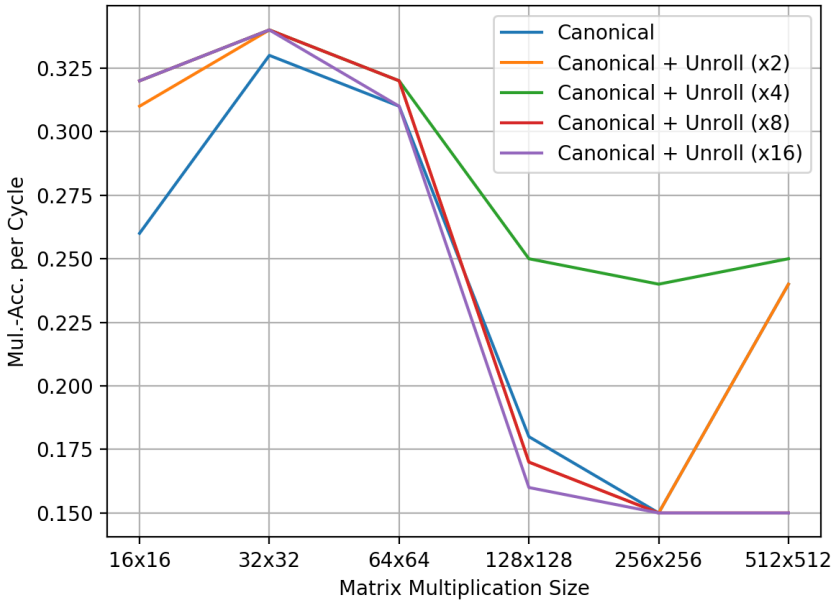

Figure 5.7: Mul.-Acc. per Cycle Results of Optimizations Performed on Real-Valued Matrix Multiplication (Part I)

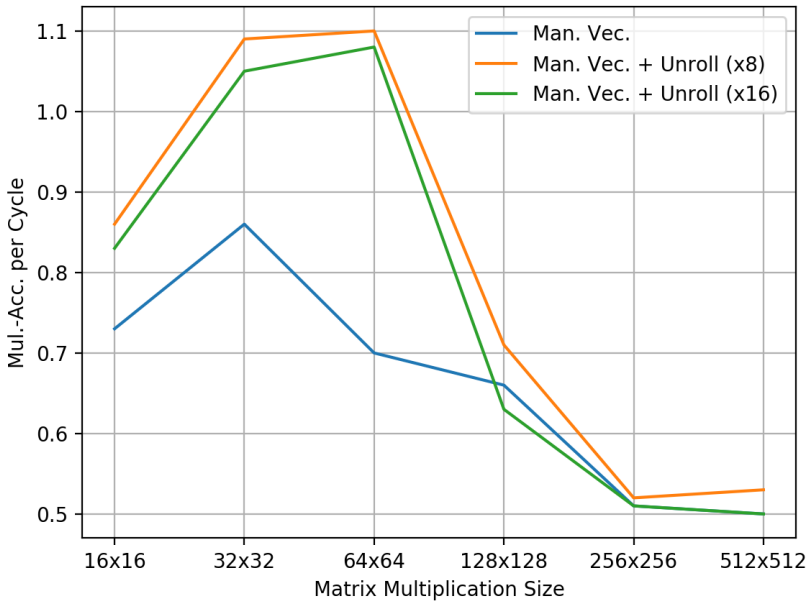

Figure 5.8: Mul.-Acc. per Cycle Results of Optimizations Performed on Real-Valued Matrix Multiplication (Part II)

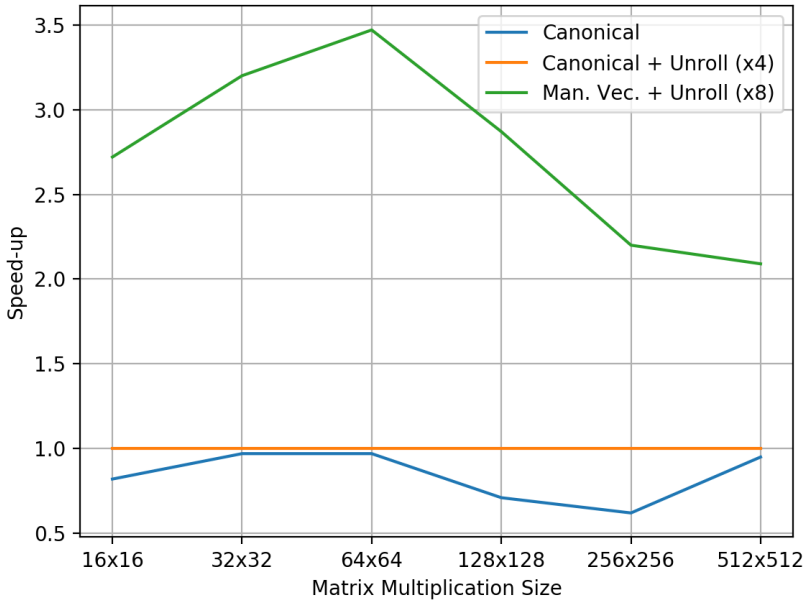

Figure 5.9: Speed-up Results of Optimizations Performed on Real-Valued Matrix Multiplication

5.6.2.3 Complex-Valued Matrix Multiplication

The speed-up and addition per cycle results of the optimizations performed on Complex-Valued Matrix Multiplication implementations are provided in figures 5.10, 5.11, and 5.12. The results that are obtained for canonical implementation is for architecture-independent C code that is compiled with auto-vectorization, loop unrolling, and instruction scheduling optimizations enabled. The speed-up is computed against *Canonical* version. Ne10 does not appear in these figures since it does not support generic sized matrix multiplication.

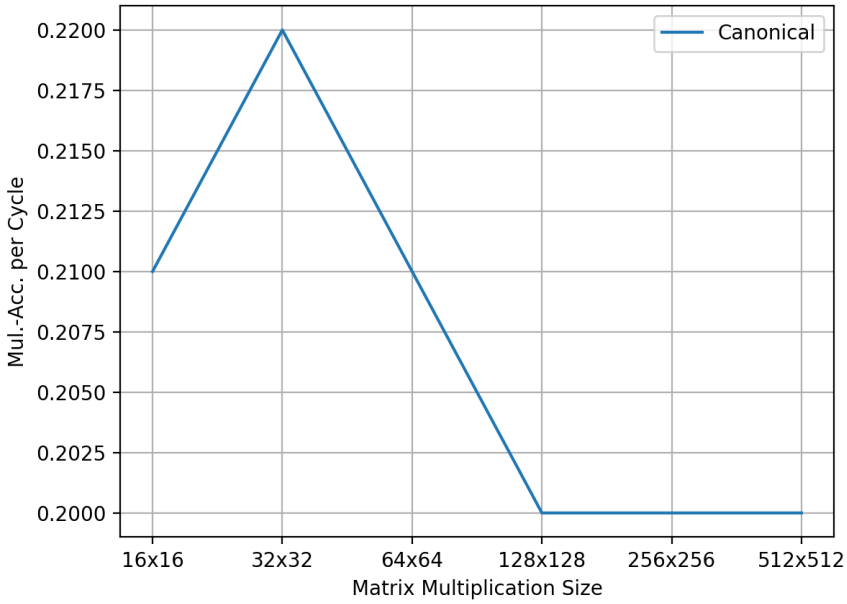

Figure 5.10: Mul.-Acc. per Cycle Results of Optimizations Performed on Complex-Valued Matrix Multiplication (Part I)

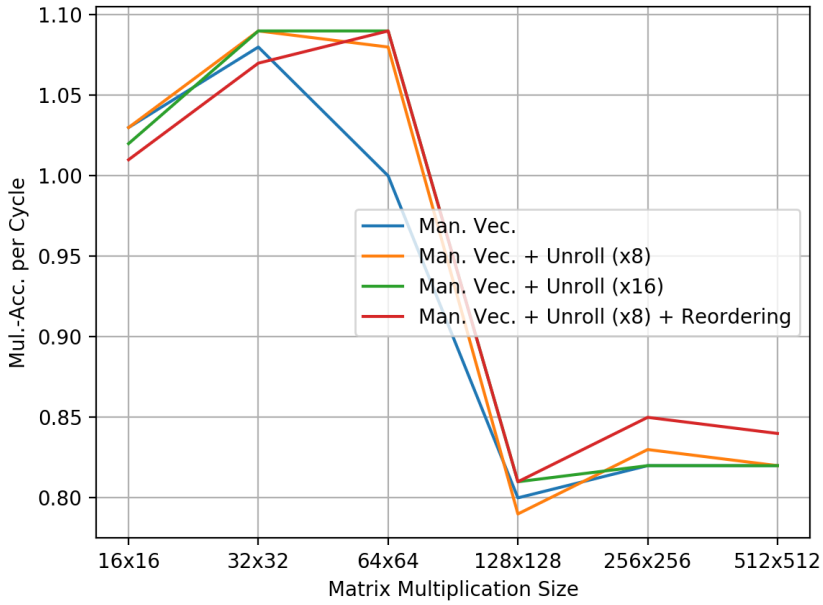

Figure 5.11: Mul.-Acc. per Cycle Results of Optimizations Performed on Complex-Valued Matrix Multiplication (Part II)

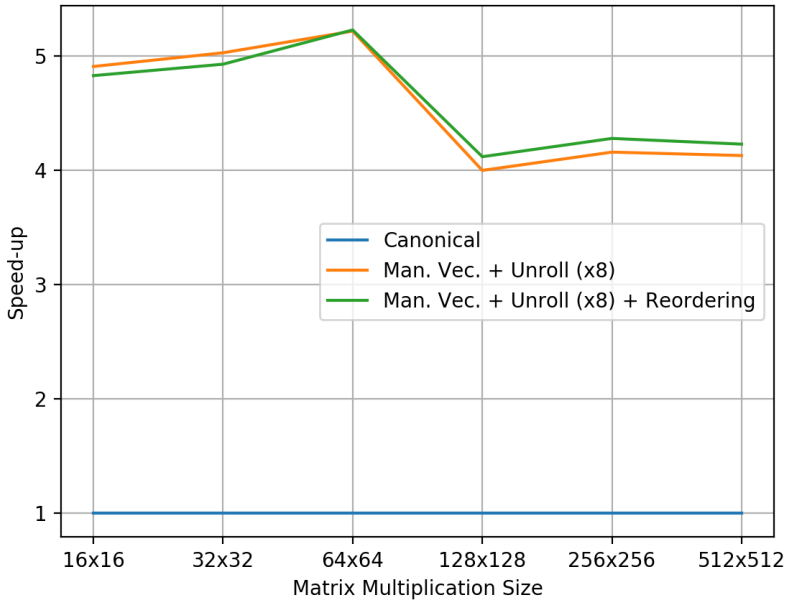

Figure 5.12: Speed-up Results of Optimizations Performed on Complex-Valued Matrix Multiplication

5.6.3 Convolution

The results of the optimizations performed on Convolution implementations are provided in figures 5.13, 5.14, and 5.15. The results that are obtained for canonical implementations, i.e. *Canonical*, are for architecture-independent C code that is compiled with auto-vectorization, loop unrolling, and instruction scheduling optimizations enabled. The speed-up is computed against *Canonical* version. Ne10 does not appear in these figures since it does not support Convolution.

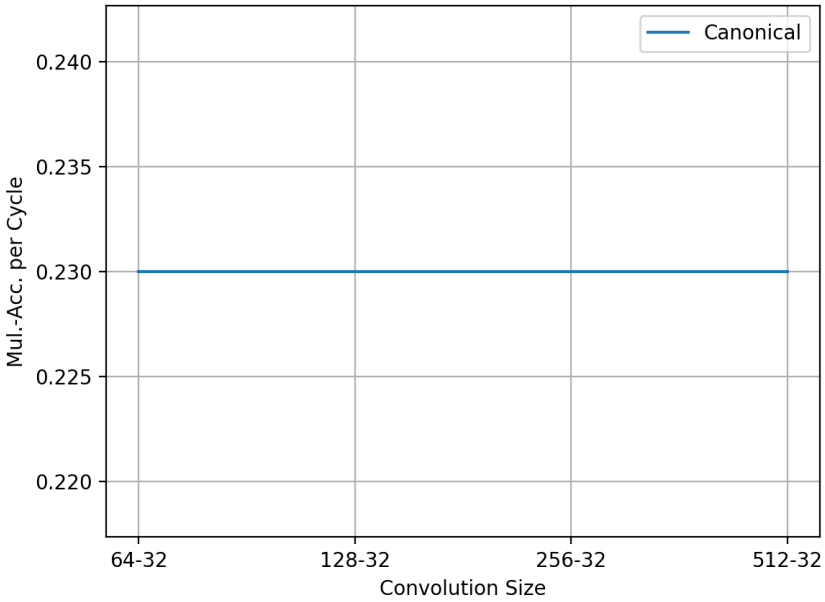

Figure 5.13: Mul.-Acc. per Cycle Results of Optimizations Performed on Complex Convolution (Part I)

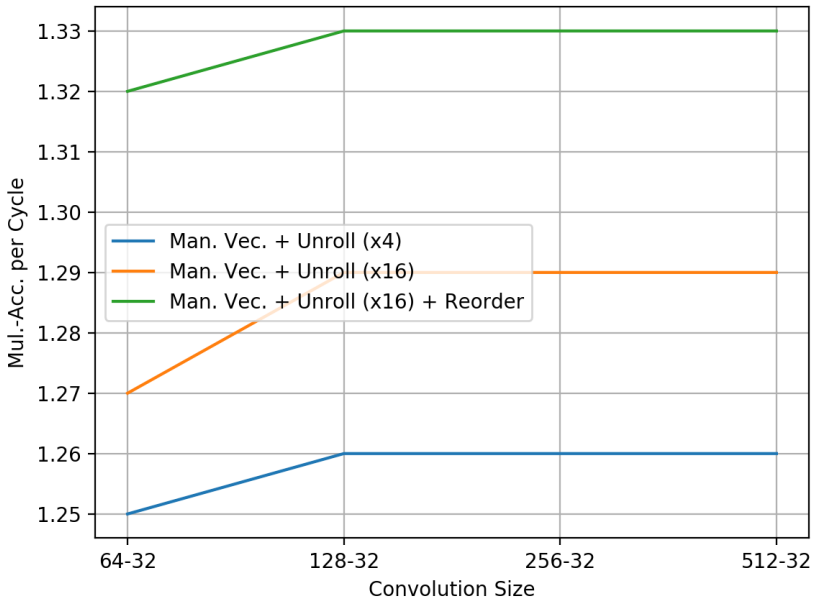

Figure 5.14: Mul.-Acc. per Cycle Results of Optimizations Performed on Complex Convolution (Part II)

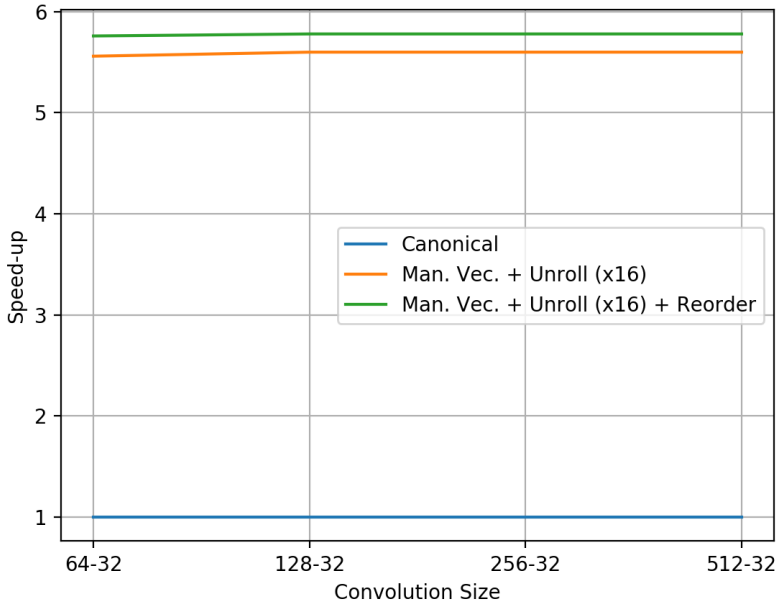

Figure 5.15: Speed-up Results of Optimizations Performed on Complex Convolution

5.6.4 Fourier Transform

The results of the optimizations performed on 16-point DFT implementation of NEON is provided in table 5.1.

Table 5.1: Results of 16-point DFT

16-point DFT	Exec. time (cycles)	Speed-up	Butterflies per cycle
Ne10	103	1.00	0.31
Ne10 + Reordering	98	1.05	0.33

The results of various Fourier Transform implementations are given in figure 5.16. The first implementation in the figure belongs to Radix-2 Fast Fourier Transform. The second implementation belongs to Radix-2 FFT which uses 16-point DFT, which is the reordered version of Ne10's implementation, as its basis. The third result belongs to Ne10's Fourier Transform implementation.

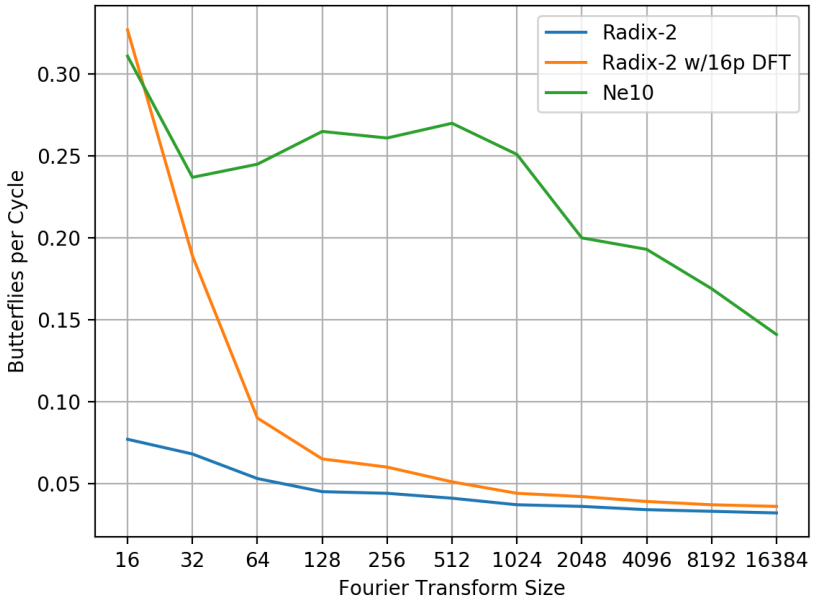

Figure 5.16: Butterflies per Cycle Results of Various Fourier Transform Implementations

5.7 Summary

In table 5.2, a summary of the results of this thesis is provided. Our results show that manual optimization of certain DSP functions is still beneficial for new architectures, such as Cortex-A15.

Table 5.2: Summary of the Results, Minimum and Maximum Speed-up Observed

Function	Speed-up vs. compiler	Speed-up vs. Ne10
Matrix Addition	1.58 - 3.35	1.03 - 2.48
Four-by-Four Matrix Multiplication	-	1.05 - 1.20
Real-Valued Matrix Multiplication	2.09 - 3.47	-
Complex-Valued Matrix Multiplication	4.00 - 5.23	-
Convolution	5.09 - 5.78	-
Fourier Transform	-	1.05

5.8 Instruction Reordering Guidelines

During this thesis study, we found the following instruction reordering techniques to prove beneficial. Hence we believe that they could be stated as general purpose guidelines while performing instruction re-ordering:

- The instructions having data dependency in between, should be placed as away from each other as possible to allow for the former one to finish executing before the latter one starts. Performance gain is possible by placing other instructions in between the dependent ones.
- Instructions loading data from memory should be executed as early as possible. Loading the data just before the calculations may cause the program to block and wait for the data to be brought to the cache.
- Even though out-of-order executing processors can reorder the instructions that have a data dependency, the processor has a fixed number of instructions it can monitor and apply reordering. The reordering should not be left to the processor entirely but applied when a potential gain is detected.
- For architectures that use memory-mapped hardware registers, load/store instructions cannot be reordered by the processor automatically. Reordering of the load/store instructions that do not have data dependency might still be beneficial. These instructions can be placed away from each other to decrease the impact of cache replacement latency.
- If the instruction issue capacity of execution units is known, it may be beneficial to keep instructions of such units in groups. For example, if the architecture has two floating-point multiplication units, it may be beneficial to keep floating-point multiplications in groups of two. If the group is broken into pieces, their chance of being issued together decreases.

CHAPTER 6

CONCLUSION AND FUTURE WORK

6.1 Conclusion

In this thesis, we have tried to show that certain frequently used functions may still benefit from manual optimizations, such as vectorization, loop unrolling, and instruction reordering. The methods and guidelines that are employed and conceived in this thesis proved their usefulness. A performance gain is encountered in almost all categories tackled in this study. However, per-cycle metrics show that the efficiency of the implementations is still far from the theoretical limit, which is 4 additions or multiplications per cycle for NEON when single-precision floating-point numbers are used.

The optimization of algorithms for newer architectures is a never-ending study. With ever increasing high performance and power-efficiency demands of modern applications, there are endless research opportunities in the low-level optimization field. We hope that the methods and the set of guidelines conceived in this study will help future researchers in their similar efforts.

6.2 Future Work

Although we have showed that manual optimization of the code is beneficial, we do not say that it is necessary. We hope that the compilers in the future would adopt the techniques applied here and remove the burden of manual optimization altogether.

Additionally, as new architectures emerge, the optimization of algorithms on them

will continue to be a hot topic of investigation. However, there are some exciting developments in processor technology that might require a different approach to optimization. ARM SVE (Scalable Vector Extension) is one such development. This architecture allows the vector coprocessor to have different width SIMD registers across different implementations but still use the same instruction set, allowing better performance without recompilation of the program. The dynamism of this architecture may require different optimization approaches.

REFERENCES

[1] Arm, "Arm cortex-a15 mpcore processor technical reference manual." Available at https://static.docs.arm.com/ddi0438/i/DDI0438I_cortex_a15_r4p0_trm.pdf.

[2] Arm, "Exploring the design of the cortex-a15 processor." Available at https://www.arm.com/files/pdf/AT-Exploring_the_Design_of_the_Cortex-A15.pdf.

[3] D. Nuzman, S. Dyshel, E. Rohou, I. Rosen, K. Williams, D. Yuste, A. Cohen, and A. Zaks, "Vapor simd: Auto-vectorize once, run everywhere," *International Symposium on Code Generation and Optimization (CGO 2011)*, 2011.

[4] IBM, "Vector/simd multimedia extension technology programming environments manual," 2005.

[5] N. Firasta, M. Buxton, P. Jinbo, K. Nasri, and S. Kuo, "Intel avx: New frontiers in performance improvements and energy efficiency," *Intel white paper*, vol. 19, no. 20, 2008.

[6] Arm, "Simd isas: Neon." Available at https://developer.arm.com/architectures/instruction-sets/simd-isas/neon.

[7] A. H. Hormati, Y. Choi, M. Woh, M. Kudlur, R. Rabbah, T. Mudge, and S. Mahlke, "Macross: Macro-simdization of streaming applications," *ACM SIGPLAN Notices*, vol. 45, no. 3, p. 285, 2010.

[8] R. Allen and K. Kennedy, "Automatic translation of fortran programs to vector form," *ACM Transactions on Programming Languages and Systems*, vol. 9, no. 4, p. 491–542, 1987.

[9] P. R. W. Francesco Petrogalli, "Llvm and the automatic vectorization of loops invoking math routines: -fsimdmath," *2018 IEEE/ACM 5th Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC)*, pp. 30–38, 2018.

[10] F. Aleen, V. P. Zakharin, R. Krishaniyer, G. Gupta, D. Kreitzer, and C.-S. Lin, "Automated compiler optimization of multiple vector loads/stores," *Proceedings of the ACM International Conference on Computing Frontiers - CF 16*, 2016.

[11] A. Pajuelo, A. Gonzalez, and M. Valero, "Speculative dynamic vectorization," *Proceedings 29th Annual International Symposium on Computer Architecture*, 2002.

[12] D.-Y. Hong, Y.-P. Liu, S.-Y. Fu, J.-J. Wu, and W.-C. Hsu, "Improving simd parallelism via dynamic binary translation," *ACM Transactions on Embedded Computing Systems*, vol. 17, no. 3, p. 1–27, 2018.

[13] J. Li, Q. Zhang, S. Xu, and B. Huang, "Optimizing dynamic binary translation for simd instructions," *International Symposium on Code Generation and Optimization (CGO06)*, 2006.

[14] Y.-P. Liu, D.-Y. Hong, J.-J. Wu, S.-Y. Fu, and W.-C. Hsu, "Exploiting asymmetric simd register configurations in arm-to-x86 dynamic binary translation," *2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2017.

[15] C. Pujara, A. Modi, G. Sandeep, S. Inamdar, D. Kolavil, and V. Tholath, "H.264 video decoder optimization on arm cortex-a8 with neon," *2009 Annual IEEE India Conference*, 2009.

[16] J. Lee, G. Jeon, S. Park, T. Jung, and J. Jeong, "Simd optimization of the h.264/svc decoder with efficient data structure," *2008 IEEE International Conference on Multimedia and Expo*, 2008.

[17] Y.-K. Chen, E. Q. Li, X. Zhou, and S. Ge, "Implementation of h.264 encoder and decoder on personal computers," *Journal of Visual Communication and Image Representation*, vol. 17, no. 2, p. 509–532, 2006.

[18] A. E. Ansari, A. Mansouri, and A. Ahaitouf, "Comparative analysis of the hevc decoder on two embedded processors using neon technology," *2016 IEEE/ACS 13th International Conference of Computer Systems and Applications (AICCSA)*, 2016.

[19] B. Bross, M. Alvarez-Mesa, V. George, C. C. Chi, T. Mayer, B. Juurlink, and T. Schierl, "Hevc real-time decoding," *Applications of Digital Image Processing XXXVI*, 2013.

[20] M. Bahtat, S. Belkouch, P. Elleaume, and P. L. Gall, "Instruction scheduling heuristic for an efficient fft in vliw processors with balanced resource usage," *EURASIP Journal on Advances in Signal Processing*, vol. 2016, no. 1, 2016.

[21] W. Xu, Z. Yan, and D. Shunying, "A high performance fft library with single instruction multiple data (simd) architecture," *2011 International Conference on Electronics, Communications and Control (ICECC)*, 2011.

[22] S. Kral, F. Franchetti, J. Lorenz, and C. W. Ueberhuber, "Simd vectorization of straight line fft code," *Euro-Par 2003 Parallel Processing Lecture Notes in Computer Science*, p. 251–260, 2003.

[23] P. Kůs, A. Marek, S. Köcher, H.-H. Kowalski, C. Carbogno, C. Scheurer, K. Reuter, M. Scheffler, and H. Lederer, "Optimizations of the eigensolvers in the elpa library," *Parallel Computing*, vol. 85, p. 167–177, 2019.

[24] K. Zhang, L. Ding, Y. Cai, W. Yin, F. Yang, J. Tao, and L. Wang, "A high performance real-time edge detection system with neon," *2017 IEEE 12th International Conference on ASIC (ASICON)*, 2017.

[25] D. Demirovic, A. Serifovic-Trbalic, N. Prljaca, and P. C. Cattin, "Evaluation of image processing algorithms on arm powered mobile devices," *2014 37th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*, 2014.

[26] B. R. Rau and J. A. Fisher, "Instruction-level parallel processing: History, overview, and perspective," *Instruction-Level Parallelism*, p. 9–50, 1993.

[27] K. Ebcioglu, E. Altman, M. Gschwind, and S. Sathaye, "Dynamic binary translation and optimization," *IEEE Transactions on Computers*, vol. 50, no. 6, p. 529–548, 2001.

[28] P. Faraboschi, J. Fisher, and C. Young, "Instruction scheduling for instruction level parallel processors," *Proceedings of the IEEE*, vol. 89, no. 11, p. 1638–1659, 2001.

[29] J. W. Cooley and J. W. Tukey, "An algorithm for the machine calculation of complex fourier series," *Mathematics of Computation*, vol. 19, no. 90, p. 297–297, 1965.

[30] R. Weber, A. Gothandaraman, R. J. Hinde, and G. D. Peterson, "Comparing hardware accelerators in scientific applications: A case study," *IEEE Transactions on Parallel and Distributed Systems*, vol. 22, no. 1, p. 58–68, 2011.

[31] G. Bergland, "A radix-eight fast fourier transform subroutine for real-valued series," *IEEE Transactions on Audio and Electroacoustics*, vol. 17, no. 2, p. 138–144, 1969.

[32] P. Duhamel and H. Hollmann, "'split radix' fft algorithm," *Electronics Letters*, vol. 20, no. 1, p. 14, 1984.

[33] M. Frigo and S. Johnson, "The design and implementation of fftw3," *Proceedings of the IEEE*, vol. 93, no. 2, p. 216–231, 2005.

[34] "Auto-vectorization in gcc." Available at `https://www.gnu.org/software/gcc/projects/tree-ssa/vectorization.html`.

[35] *Collins English Dictionary*. HarperCollins Publishers, 2018.

[36] S.-A.-A. Touati, J. Worms, and S. Briais, "The speedup-test: a statistical methodology for programme speedup analysis and computation," *Concurrency and Computation: Practice and Experience*, vol. 25, no. 10, p. 1410–1426, 2012.

Appendix A

RESULTS

The exact results are provided in this chapter in table form.

A.1 Matrix Addition

The results of the optimizations performed on Matrix Addition implementations are provided in tables A.1, A.2, A.3, A.4, A.5, A.6, and A.7, along with the performance of Ne10's implementation.

Table A.1: Results of Optimizations Performed on Matrix Addition (Part I)

8x8-8x8 Matrix Addition	Exec. time (cycles)	Speed-up	Additions per Cycle
Canonical	149	0.84	0.43
Canonical + Unroll (x2)	147	0.85	0.44
Canonical + Unroll (x4)	152	0.82	0.42
Canonical + Unroll (x8)	131	0.95	0.49
Canonical + Unroll (x16)	127	0.98	0.50
Canonical + Unroll (x32)	125	1.00	0.51
Manual Vectorization	52	2.40	1.23
Man. Vec. + Unroll (x16)	51	2.45	1.25
Man. Vec. + Unroll (x32)	50	2.50	1.28
Ne10	124	1.01	0.52

Table A.2: Results of Optimizations Performed on Matrix Addition (Part II)

16x16-16x16 Matrix Addition	Exec. time (cycles)	Speed-up	Additions per Cycle
Canonical	601	0.86	0.43
Canonical + Unroll (x2)	549	0.94	0.47
Canonical + Unroll (x4)	570	0.90	0.45
Canonical + Unroll (x8)	554	0.93	0.46
Canonical + Unroll (x16)	539	0.96	0.47
Canonical + Unroll (x32)	515	1.00	0.50
Manual Vectorization	200	2.58	1.28
Man. Vec. + Unroll (x16)	199	2.59	1.29
Man. Vec. + Unroll (x32)	199	2.59	1.29
Ne10	455	1.13	0.56

Table A.3: Results of Optimizations Performed on Matrix Addition (Part III)

32x32-32x32 Matrix Addition	Exec. time (cycles)	Speed-up	Additions per Cycle
Canonical	2329	0.89	0.44
Canonical + Unroll (x2)	2132	0.97	0.48
Canonical + Unroll (x4)	2215	0.94	0.46
Canonical + Unroll (x8)	2142	0.97	0.48
Canonical + Unroll (x16)	2095	0.99	0.49
Canonical + Unroll (x32)	2073	1.00	0.49
Manual Vectorization	805	2.58	1.27
Man. Vec. + Unroll (x16)	782	2.65	1.31
Man. Vec. + Unroll (x32)	779	2.66	1.31
Ne10	1704	1.22	0.61

Table A.4: Results of Optimizations Performed on Matrix Addition (Part IV)

64x64-64x64 Matrix Addition	Exec. time (cycles)	Speed-up	Additions per Cycle
Canonical	13164	0.85	0.31
Canonical + Unroll (x2)	13367	0.84	0.31
Canonical + Unroll (x4)	13434	0.84	0.30
Canonical + Unroll (x8)	13200	0.85	0.31
Canonical + Unroll (x16)	12351	0.91	0.33
Canonical + Unroll (x32)	11241	1.00	0.36
Manual Vectorization	3410	3.30	1.20
Man. Vec. + Unroll (x16)	3364	3.34	1.22
Man. Vec. + Unroll (x32)	3360	3.35	1.22
Ne10	7933	1.42	0.52

Table A.5: Results of Optimizations Performed on Matrix Addition (Part V)

128x128-128x128 Matrix Addition	Exec. time (cycles)	Speed-up	Additions per Cycle
Canonical	59556	0.98	0.27
Canonical + Unroll (x2)	59576	0.98	0.28
Canonical + Unroll (x4)	58330	1.00	0.28
Canonical + Unroll (x8)	58719	0.99	0.28
Canonical + Unroll (x16)	58309	1.00	0.28
Canonical + Unroll (x32)	58303	1.00	0.28
Manual Vectorization	33540	1.74	0.49
Man. Vec. + Unroll (x16)	33168	1.76	0.49
Man. Vec. + Unroll (x32)	32252	1.81	0.51
Ne10	34680	1.68	0.47

Table A.6: Results of Optimizations Performed on Matrix Addition (Part VI)

256x256-256x256 Matrix Addition	Exec. time (cycles)	Speed-up	Additions per Cycle
Canonical	245370	0.99	0.27
Canonical + Unroll (x2)	242132	1.00	0.27
Canonical + Unroll (x4)	242680	1.00	0.27
Canonical + Unroll (x8)	245670	0.99	0.27
Canonical + Unroll (x16)	242126	1.00	0.27
Canonical + Unroll (x32)	242083	1.00	0.27
Manual Vectorization	147599	1.64	0.44
Man. Vec. + Unroll (x16)	146827	1.65	0.45
Man. Vec. + Unroll (x32)	146233	1.66	0.45
Ne10	157201	1.54	0.42

Table A.7: Results of Optimizations Performed on Matrix Addition (Part VII)

512x512-512x512 Matrix Addition	Exec. time (cycles)	Speed-up	Additions per Cycle
Canonical	952881	0.98	0.28
Canonical + Unroll (x2)	937222	1.00	0.28
Canonical + Unroll (x4)	947984	0.99	0.28
Canonical + Unroll (x8)	951625	0.98	0.28
Canonical + Unroll (x16)	936927	1.00	0.28
Canonical + Unroll (x32)	936563	1.00	0.28
Manual Vectorization	599872	1.56	0.43
Man. Vec. + Unroll (x16)	597439	1.57	0.44
Man. Vec. + Unroll (x32)	593935	1.58	0.44
Ne10	609346	1.54	0.43

A.2 Four-by-Four Real-Valued Matrix Multiplication

The results of the optimizations performed on Four-by-Four matrix multiplication implementations are provided in table A.8, along with the performance of Ne10's implementation.

Table A.8: Results of Four-by-Four Real-Valued Matrix Multiplications

4 4x4 Matrix Mul.	Exec. time (cycles)	Speed-up	Mul.-Acc. per Cycle
Ne10	107	1.00	2.39
Ne10 + Reordering	97	1.10	2.64
8 4x4 Matrix Mul.			
Ne10	207	1.00	2.47
Ne10 + Reordering	195	1.06	2.63
16 4x4 Matrix Mul.			
Ne10	429	1.00	2.39
Ne10 + Reordering	383	1.12	2.67
32 4x4 Matrix Mul.			
Ne10	830	1.00	2.47
Ne10 + Reordering	780	1.06	2.63
64 4x4 Matrix Mul.			
Ne10	1630	1.00	2.51
Ne10 + Reordering	1542	1.06	2.66
128 4x4 Matrix Mul.			
Ne10	3229	1.00	2.54
Ne10 + Reordering	3062	1.05	2.66
256 4x4 Matrix Mul.			
Ne10	7308	1.00	2.24
Ne10 + Reordering	6102	1.20	2.69

A.3 Real-Valued Matrix Multiplication

The results of the optimizations performed on Real-Valued Matrix Multiplication implementations are provided in tables A.9, A.10, A.11, A.12, A.13, and A.14.

Table A.9: Results of Real-Valued Matrix Multiplications (Part I)

16x16-16x16 Matrix Mul.	Exec. time (cycles)	Speed-up	Mul.-Acc. per Cycle
Canonical	15744	0.82	0.26
Canonical + Unroll (x2)	13120	0.98	0.31
Canonical + Unroll (x4)	12864	1.00	0.32
Canonical + Unroll (x8)	12864	1.00	0.32
Canonical + Unroll (x16)	12864	1.00	0.32
Manual Vectorization	5632	2.28	0.73
Man. Vec. + Unroll (x8)	4736	2.72	0.86
Man. Vec. + Unroll (x16)	4928	2.61	0.83

Table A.10: Results of Real-Valued Matrix Multiplications (Part II)

32x32-32x32 Matrix Mul.	Exec. time (cycles)	Speed-up	Mul.-Acc. per Cycle
Canonical	99646	0.97	0.33
Canonical + Unroll (x2)	96401	1.00	0.34
Canonical + Unroll (x4)	96379	1.00	0.34
Canonical + Unroll (x8)	96431	1.00	0.34
Canonical + Unroll (x16)	97287	0.99	0.34
Manual Vectorization	37888	2.54	0.86
Man. Vec. + Unroll (x8)	30144	3.20	1.09
Man. Vec. + Unroll (x16)	31296	3.07	1.05

Table A.11: Results of Real-Valued Matrix Multiplications (Part III)

64x64-64x64 Matrix Mul.	Exec. time (cycles)	Speed-up	Mul.-Acc. per Cycle
Canonical	856960	0.97	0.31
Canonical + Unroll (x2)	829056	1.00	0.32
Canonical + Unroll (x4)	828864	1.00	0.32
Canonical + Unroll (x8)	829312	1.00	0.32
Canonical + Unroll (x16)	836672	0.99	0.31
Manual Vectorization	379776	2.18	0.70
Man. Vec. + Unroll (x8)	238528	3.47	1.10
Man. Vec. + Unroll (x16)	242752	3.41	1.08

Table A.12: Results of Real-Valued Matrix Multiplications (Part IV)

128x128-128x128 Matrix Mul.	Exec. time (cycles)	Speed-up	Mul.-Acc. per Cycle
Canonical	11833152	0.71	0.18
Canonical + Unroll (x2)	12499200	0.67	0.17
Canonical + Unroll (x4)	8422976	1.00	0.25
Canonical + Unroll (x8)	12576768	0.67	0.17
Canonical + Unroll (x16)	13010816	0.65	0.16
Manual Vectorization	3185408	2.64	0.66
Man. Vec. + Unroll (x8)	2937792	2.87	0.71
Man. Vec. + Unroll (x16)	3323584	2.53	0.63

Table A.13: Results of Real-Valued Matrix Multiplications (Part V)

256x256-256x256 Matrix Mul.	Exec. time (cycles)	Speed-up	Mul.-Acc. per Cycle
Canonical	113966592	0.62	0.15
Canonical + Unroll (x2)	109710208	0.64	0.15
Canonical + Unroll (x4)	70723648	1.00	0.24
Canonical + Unroll (x8)	111758144	0.63	0.15
Canonical + Unroll (x16)	111737728	0.63	0.15
Manual Vectorization	33350400	2.12	0.51
Man. Vec. + Unroll (x8)	32207168	2.20	0.52
Man. Vec. + Unroll (x16)	32809344	2.16	0.51

Table A.14: Results of Real-Valued Matrix Multiplications (Part VI)

512x512-512x512 Matrix Mul.	Exec. time (cycles)	Speed-up	Mul.-Acc. per Cycle
Canonical	562383296	0.95	0.24
Canonical + Unroll (x2)	557327040	0.96	0.24
Canonical + Unroll (x4)	533936576	1.00	0.25
Canonical + Unroll (x8)	905502272	0.59	0.15
Canonical + Unroll (x16)	897988288	0.59	0.15
Manual Vectorization	268527744	1.99	0.50
Man. Vec. + Unroll (x8)	255559040	2.09	0.53
Man. Vec. + Unroll (x16)	268646336	1.99	0.50

A.4 Complex-Valued Matrix Multiplication

The results of the optimizations performed on Complex-Valued Matrix Multiplication implementations are provided in table A.15 and A.16.

Table A.15: Results of Complex-Valued Matrix Multiplications (Part I)

16x16-16x16 Comp. Mat. Mul.	Exec. time (cycles)	Speed-up	Mul.-Acc. per Cycle
Canonical	78208	1.00	0.21
Manual Vectorization	15936	4.91	1.03
Man. Vec. + Unroll (x8)	15936	4.91	1.03
Man. Vec. + Unroll (x16)	16000	4.89	1.02
Man. Vec. + Unroll (x8) + Reordering	16192	4.83	1.01
32x32-32x32 Comp. Mat. Mul.			
Canonical	606784	1.00	0.22
Manual Vectorization	121088	5.01	1.08
Man. Vec. + Unroll (x8)	120640	5.03	1.09
Man. Vec. + Unroll (x16)	120768	5.02	1.09
Man. Vec. + Unroll (x8) + Reorder	123072	4.93	1.07
64x64-64x64 Comp. Mat. Mul.			
Canonical	5047296	1.00	0.21
Manual Vectorization	1048960	4.81	1.00
Man. Vec. + Unroll (x8)	967424	5.22	1.08
Man. Vec. + Unroll (x16)	966208	5.22	1.09
Man. Vec. + Unroll (x8) + Reorder	964544	5.23	1.09
128x128-128x128 Comp. Mat. Mul.			
Canonical	42471808	1.00	0.20
Manual Vectorization	10501440	4.04	0.80
Man. Vec. + Unroll (x8)	10609024	4.00	0.79
Man. Vec. + Unroll (x16)	10327360	4.11	0.81
Man. Vec. + Unroll (x8) + Reorder	10314944	4.12	0.81
256x256-256x256 Comp. Mat. Mul.			
Canonical	338038528	1.00	0.20
Manual Vectorization	81839296	4.13	0.82
Man. Vec. + Unroll (x8)	81306368	4.16	0.83
Man. Vec. + Unroll (x16)	81680448	4.14	0.82
Man. Vec. + Unroll (x8) + Reorder	78999552	4.28	0.85
512x512-512x512 Comp. Mat. Mul.			
Canonical	2706064832	1.00	0.20
Manual Vectorization	658233792	4.11	0.82
Man. Vec. + Unroll (x8)	655985600	4.13	0.82
Man. Vec. + Unroll (x16)	656628928	4.12	0.82
Man. Vec. + Unroll (x8) + Reorder	639349376	4.23	0.84

Table A.16: Results of Complex-Valued Matrix Multiplications (Part II)

16x16-16x256 Comp. Mat. Mul.	Exec. time (cycles)	Speed-up	Mul.-Acc. per Cycle
Canonical	1296576	1.00	0.20
Manual Vectorization	297088	4.36	0.88
Man. Vec. + Unroll (x8)	294912	4.40	0.89
Man. Vec. + Unroll (x8) + Reordering	282560	4.59	0.93
24x24-24x256 Comp. Mat. Mul.			
Canonical	2871808	1.00	0.21
Manual Vectorization	673280	4.27	0.88
Man. Vec. + Unroll (x8)	670144	4.29	0.88
Man. Vec. + Unroll (x8) + Reorder	639616	4.49	0.92
16x16-16x1000 Comp. Mat. Mul.			
Canonical	5130112	1.00	0.20
Manual Vectorization	1276672	4.02	0.80
Man. Vec. + Unroll (x8)	1273984	4.03	0.80
Man. Vec. + Unroll (x8) + Reorder	1198784	4.28	0.85
32x32-32x1000 Comp. Mat. Mul.			
Canonical	20592000	1.00	0.20
Manual Vectorization	5115968	4.03	0.80
Man. Vec. + Unroll (x8)	5142528	4.00	0.80
Man. Vec. + Unroll (x8) + Reorder	4959168	4.15	0.83
64x64-64x3000 Comp. Mat. Mul.			
Canonical	247296064	1.00	0.20
Manual Vectorization	66112512	3.74	0.74
Man. Vec. + Unroll (x8)	67234112	3.68	0.73
Man. Vec. + Unroll (x8) + Reorder	66601408	3.71	0.74
128x128-128x3000 Comp. Mat. Mul.			
Canonical	1015243136	1.00	0.19
Manual Vectorization	289148608	3.51	0.68
Man. Vec. + Unroll (x8)	285672832	3.55	0.69
Man. Vec. + Unroll (x8) + Reorder	290692544	3.49	0.68

A.5 Convolution

The results of the optimizations performed on Complex Convolution implementations are provided in table A.17, A.18, and A.19.

Table A.17: Results of Convolution (Part I)

32-to-16 Convolution	Exec. time (cycles)	Speed-up	Mul.-Acc. per Cycle
Canonical	4823	1.00	0.23
Man. Vec. + Unroll (x4)	977	4.94	1.11
Man. Vec. + Unroll (x16)	947	5.09	1.14
Man. Vec. + Unroll (x16) + Reorder	935	5.16	1.16
64-to-16 Convolution			
Canonical	13853	1.00	0.23
Man. Vec. + Unroll (x4)	2749	5.04	1.14
Man. Vec. + Unroll (x16)	2705	5.12	1.16
Man. Vec. + Unroll (x16) + Reorder	2652	5.22	1.18
128-to-16 Convolution			
Canonical	31878	1.00	0.23
Man. Vec. + Unroll (x4)	6279	5.08	1.15
Man. Vec. + Unroll (x16)	6221	5.12	1.17
Man. Vec. + Unroll (x16) + Reorder	6044	5.23	1.20
256-to-16 Convolution			
Canonical	68001	1.00	0.23
Man. Vec. + Unroll (x4)	13379	5.08	1.15
Man. Vec. + Unroll (x16)	13262	5.13	1.16
Man. Vec. + Unroll (x16) + Reorder	13031	5.21	1.18
512-to-16 Convolution	Exec. time (cycles)	Speed-up	Mul.-Acc. per Cycle
Canonical	140183	1.00	0.23
Man. Vec. + Unroll (x4)	27557	5.09	1.15
Man. Vec. + Unroll (x16)	27345	5.13	1.17
Man. Vec. + Unroll (x16) + Reorder	26751	5.24	1.19

Table A.18: Results of Convolution (Part II)

64-to-32 Convolution			
Canonical	18443	1.00	0.23
Man. Vec. + Unroll (x4)	3376	5.46	1.25
Man. Vec. + Unroll (x16)	3317	5.56	1.27
Man. Vec. + Unroll (x16) + Reorder	3203	5.76	1.32
128-to-32 Convolution			
Canonical	54155	1.00	0.23
Man. Vec. + Unroll (x4)	9874	5.48	1.26
Man. Vec. + Unroll (x16)	9675	5.60	1.29
Man. Vec. + Unroll (x16) + Reorder	9364	5.78	1.33
256-to-32 Convolution			
Canonical	125579	1.00	0.23
Man. Vec. + Unroll (x4)	22866	5.49	1.26
Man. Vec. + Unroll (x16)	22414	5.60	1.29
Man. Vec. + Unroll (x16) + Reorder	21714	5.78	1.33
512-to-32 Convolution			
Canonical	268427	1.00	0.23
Man. Vec. + Unroll (x4)	48985	5.48	1.26
Man. Vec. + Unroll (x16)	47975	5.60	1.29
Man. Vec. + Unroll (x16) + Reorder	46442	5.78	1.33

Table A.19: Results of Convolution (Part III)

1000-to-512 Convolution	Exec. time (cycles)	Speed-up	Mul.-Acc. per Cycle
Canonical	4330189	1.00	0.23
Man. Vec. + Unroll (x4)	705543	6.14	1.42
Man. Vec. + Unroll (x16)	692121	6.26	1.45
Man. Vec. + Unroll (x16) + Reorder	659565	6.57	1.52
1000-to-32 Convolution			
Canonical	540731	1.00	0.23
Man. Vec. + Unroll (x4)	98380	5.50	1.26
Man. Vec. + Unroll (x16)	95883	5.64	1.29
Man. Vec. + Unroll (x16) + Reorder	93004	5.81	1.33
10000-to-512 Convolution			
Canonical	84399160	1.00	0.23
Man. Vec. + Unroll (x4)	13962768	6.04	1.39
Man. Vec. + Unroll (x16)	13742893	6.14	1.41
Man. Vec. + Unroll (x16) + Reorder	13096150	6.44	1.48
10000-to-32 Convolution			
Canonical	5591105	1.00	0.23
Man. Vec. + Unroll (x4)	1035676	5.40	1.23
Man. Vec. + Unroll (x16)	1003375	5.57	1.27
Man. Vec. + Unroll (x16) + Reorder	978549	5.71	1.30

A.6 Fourier Transform

The results of different Fourier Transform implementations are provided in table A.20, and A.21. The first implementation belongs to Radix-2 Fast Fourier Transform. The second implementation belongs to Radix-2 FFT which uses 16-point DFT, which is the reordered version of Ne10's implementation, as its basis. The third result belongs to Ne10's Fourier Transform implementation.

Table A.20: Results of Different Fourier Transform Implementations (Part I)

16-point Fourier Transform	Exec. time (cycles)	Butterflies per Cycle
Radix-2	414	0.077
Radix-2 w/reordered Ne10 16-point impl.	98	0.327
Ne10	103	0.311
32-point Fourier Transform		
Radix-2	1182	0.068
Radix-2 w/reordered Ne10 16-point impl.	423	0.189
Ne10	337	0.237
64-point Fourier Transform		
Radix-2	3615	0.053
Radix-2 w/reordered Ne10 16-point impl.	2139	0.090
Ne10	785	0.245
128-point Fourier Transform		
Radix-2	9880	0.045
Radix-2 w/reordered Ne10 16-point impl.	6917	0.065
Ne10	1692	0.265
256-point Fourier Transform		
Radix-2	23282	0.044
Radix-2 w/reordered Ne10 16-point impl.	17116	0.060
Ne10	3916	0.261
512-point Fourier Transform		
Radix-2	55896	0.041
Radix-2 w/reordered Ne10 16-point impl.	45282	0.051
Ne10	8538	0.270

Table A.21: Results of Different Fourier Transform Implementations (Part II)

1024-point Fourier Transform	Exec. time (cycles)	Butterflies per Cycle
Radix-2	136849	0.037
Radix-2 w/reordered Ne10 16-point impl.	117094	0.044
Ne10	20413	0.251
2048-point Fourier Transform		
Radix-2	313950	0.036
Radix-2 w/reordered Ne10 16-point impl.	265940	0.042
Ne10	56224	0.200
4096-point Fourier Transform		
Radix-2	715767	0.034
Radix-2 w/reordered Ne10 16-point impl.	627959	0.039
Ne10	127304	0.193
8192-point Fourier Transform		
Radix-2	1615139	0.033
Radix-2 w/reordered Ne10 16-point impl.	1424822	0.037
Ne10	314995	0.169
16384-point Fourier Transform		
Radix-2	3533664	0.032
Radix-2 w/reordered Ne10 16-point impl.	3197615	0.036
Ne10	814480	0.141

Appendix B

SOURCE CODE

The source code for the functions that prove to be most performant are provided in this chapter.

B.1 Mat File Parser

```
unsigned mat(const char *fpath, float **out_buffer,
  unsigned *out_row_count, unsigned *out_col_count,
  int *out_is_complex)
{
  // open file as text file
  FILE *f = fopen(fpath, "r");
  if (!f) {
  fprintf(stderr, "Unable to open file: %s\n", fpath);
  return 0;
  }

  // consume first two lines
  fscanf(f, "%*[^\n]\n");
  fscanf(f, "%*[^\n]\n");

  // consume the line describing the type of the variable
  // supported types: matrix, complex matrix
  char linebuf[32];
  int is_complex;
```

```
fscanf(f, "%[^\n]\n", linebuf);
if (!strcmp(linebuf, "# type: matrix"))
  is_complex = 0;
else if (!strcmp(linebuf, "# type: complex matrix"))
  is_complex = 1;
else {
  fprintf(stderr, "Unsupported variable type\n");
  return 0;
}

// read row and column count
unsigned row_cnt, col_cnt;
fscanf(f, "# rows: %u\n", &row_cnt);
fscanf(f, "# columns: %u\n", &col_cnt);
if (row_cnt == 0 || col_cnt == 0) {
  fprintf(stderr, "Invalid matrix size\n");
  return 0;
}

// allocate buffer
float *buf = malloc(
  row_cnt *
  col_cnt *
  sizeof(float) *
  (is_complex + 1));
if (!buf) {
  fprintf(stderr, "Unable to allocate buffer\n");
  return 0;
}

// read matrix
float *next = buf;
```

```
for (unsigned row = 0; row < row_cnt; ++row) {
  for (unsigned col = 0; col < col_cnt; ++col) {
    if (is_complex == 0) {
      float tmp;
      fscanf(f, "%f", &tmp);
      *next++ = tmp;
    }
    else {
      float tmp, tmp2;
      fscanf(f, " (%f,%f) ", &tmp, &tmp2);
      *next++ = tmp;
      *next++ = tmp2;
    }
  }
}

// close file
fclose(f);

if (out_buffer)
  *out_buffer = buf;
else
  free(buf);
if (out_row_count)
  *out_row_count = row_cnt;
if (out_col_count)
  *out_col_count = col_cnt;
if (out_is_complex)
  *out_is_complex = is_complex;

return row_cnt *
  col_cnt *
```

```
    sizeof(float) *
    (is_complex + 1);
}
```

B.2 Matrix Addition

```
void matadd(struct params *params)
{
  register float *A asm("r4") = params->in1;
  register float *B asm("r5") = params->in2;
  register float *Y asm("r6") = params->out;

  unsigned A_row_cnt = params->in1_row_cnt;
  unsigned A_col_cnt = params->in1_col_cnt;
  unsigned B_row_cnt = params->in2_row_cnt;
  unsigned B_col_cnt = params->in2_col_cnt;
  unsigned Y_row_cnt = A_row_cnt, Y_col_cnt = B_col_cnt;
  register unsigned Y_cnt = Y_row_cnt * Y_col_cnt;

  for (unsigned y = 0; y < Y_cnt; y += 32) {
    asm volatile (
      "vld1.32 {d0,d1,d2,d3}, [%[a]]! \n\t"
      "vld1.32 {d4,d5,d6,d7}, [%[b]]! \n\t"
      "vadd.f32 q0, q2 \n\t"
      "vadd.f32 q1, q3 \n\t"
      "vst1.32 {d0,d1,d2,d3}, [%[y]]! \n\t"

      "vld1.32 {d8,d9,d10,d11}, [%[a]]! \n\t"
      "vld1.32 {d12,d13,d14,d15}, [%[b]]! \n\t"
      "vadd.f32 q4, q6 \n\t"
      "vadd.f32 q5, q7 \n\t"
      "vst1.32 {d8,d9,d10,d11}, [%[y]]! \n\t"
```

```
      "vld1.32 {d16,d17,d18,d19}, [%[a]]! \n\t"
      "vld1.32 {d20,d21,d22,d23}, [%[b]]! \n\t"
      "vadd.f32 q8, q10 \n\t"
      "vadd.f32 q9, q11 \n\t"
      "vst1.32 {d16,d17,d18,d19}, [%[y]]! \n\t"

      "vld1.32 {d24,d25,d26,d27}, [%[a]]! \n\t"
      "vld1.32 {d28,d29,d30,d31}, [%[b]]! \n\t"
      "vadd.f32 q12, q14 \n\t"
      "vadd.f32 q13, q15 \n\t"
      "vst1.32 {d24,d25,d26,d27}, [%[y]]! \n\t"
      : [a] "+r" (A), [b] "+r" (B), [y] "+r" (Y)
    );
  }
}
```

B.3 Four-by-four Matrix Multiplication

```
void matmul_4by4(struct params *params)
{
  float *A = params->in1;
  float *B = params->in2;
  float *Y = params->out;
  register float *A_p asm("r4") = A;
  register float *B_p asm("r5") = B;
  register float *Y_p asm("r6") = Y;
  const unsigned A_row_cnt = params->in1_row_cnt;
  const unsigned A_col_cnt = params->in1_col_cnt;
  const unsigned B_row_cnt = params->in2_row_cnt;
  register const unsigned B_col_cnt =
    params->in2_col_cnt;
```

```
const unsigned Y_row_cnt = A_row_cnt;
const unsigned Y_col_cnt = B_col_cnt;
const unsigned Y_dim = Y_row_cnt * Y_col_cnt;
register float32x4_t aaaa asm("q0");
register float32x4_t bbbb asm("q1");
register float32x4_t yyyy asm("q2");

for (int i = 0; i < N; ++i)
  asm volatile (
    "vld1.32 {q8-q9}, [%[a]]! \n\t"
    "vld1.32 {q0-q1}, [%[b]]! \n\t"
    "vld1.32 {q10-q11}, [%[a]]! \n\t"
    "vld1.32 {q2-q3}, [%[b]]! \n\t"

    "vmul.f32 q12, q8, d0[0] \n\t"
    "vmul.f32 q13, q8, d2[0] \n\t"
    "vmul.f32 q14, q8, d4[0] \n\t"
    "vmul.f32 q15, q8, d6[0] \n\t"
    "vld1.32 { q4-q5 }, [%[a]]! \n\t"
    "vmla.f32 q12, q9, d0[1] \n\t"
    "vmla.f32 q13, q9, d2[1] \n\t"
    "vmla.f32 q14, q9, d4[1] \n\t"
    "vmla.f32 q15, q9, d6[1] \n\t"
    "vld1.32 {q6-q7}, [%[a]]! \n\t"
    "vmla.f32 q12, q10, d1[0] \n\t"
    "vmla.f32 q13, q10, d3[0] \n\t"
    "vmla.f32 q14, q10, d5[0] \n\t"
    "vmla.f32 q15, q10, d7[0] \n\t"
    "vmla.f32 q12, q11, d1[1] \n\t"
    "vmla.f32 q13, q11, d3[1] \n\t"
    "vld1.32 {q0-q1}, [%[b]]! \n\t"
    "vmla.f32 q14, q11, d5[1] \n\t"
```

```
"vmla.f32 q15, q11, d7[1] \n\t"

"vst1.32 {q12-q13}, [%[y]]! \n\t"
"vld1.32 {q2-q3}, [%[b]]! \n\t"

"vmul.f32 q12, q4, d0[0] \n\t"
"vmul.f32 q13, q4, d2[0] \n\t"
"vst1.32 {q14-q15}, [%[y]]! \n\t"
"vmul.f32 q14, q4, d4[0] \n\t"
"vmul.f32 q15, q4, d6[0] \n\t"
"vld1.32 { q8-q9 }, [%[a]]! \n\t"
"vmla.f32 q12, q5, d0[1] \n\t"
"vmla.f32 q13, q5, d2[1] \n\t"
"vmla.f32 q14, q5, d4[1] \n\t"
"vmla.f32 q15, q5, d6[1] \n\t"
"vld1.32 {q10-q11}, [%[a]]! \n\t"
"vmla.f32 q12, q6, d1[0] \n\t"
"vmla.f32 q13, q6, d3[0] \n\t"
"vmla.f32 q14, q6, d5[0] \n\t"
"vmla.f32 q15, q6, d7[0] \n\t"
"vmla.f32 q12, q7, d1[1] \n\t"
"vmla.f32 q13, q7, d3[1] \n\t"
"vld1.32 {q0-q1}, [%[b]]! \n\t"
"vmla.f32 q14, q7, d5[1] \n\t"
"vmla.f32 q15, q7, d7[1] \n\t"

"vst1.32 {q12-q13}, [%[y]]! \n\t"
"vld1.32 {q2-q3}, [%[b]]! \n\t"

"vmul.f32 q12, q8, d0[0] \n\t"
"vmul.f32 q13, q8, d2[0] \n\t"
"vst1.32 {q14-q15}, [%[y]]! \n\t"
```

```
"vmul.f32 q14, q8, d4[0] \n\t"
"vmul.f32 q15, q8, d6[0] \n\t"
"vld1.32 {q4-q5}, [%[a]]! \n\t"
"vmla.f32 q12, q9, d0[1] \n\t"
"vmla.f32 q13, q9, d2[1] \n\t"
"vmla.f32 q14, q9, d4[1] \n\t"
"vmla.f32 q15, q9, d6[1] \n\t"
"vld1.32 {q6-q7}, [%[a]]! \n\t"
"vmla.f32 q12, q10, d1[0] \n\t"
"vmla.f32 q13, q10, d3[0] \n\t"
"vmla.f32 q14, q10, d5[0] \n\t"
"vmla.f32 q15, q10, d7[0] \n\t"
"vmla.f32 q12, q11, d1[1] \n\t"
"vmla.f32 q13, q11, d3[1] \n\t"
"vld1.32 {q0-q1}, [%[b]]! \n\t"
"vmla.f32 q14, q11, d5[1] \n\t"
"vmla.f32 q15, q11, d7[1] \n\t"

"vst1.32 {q12-q13}, [%[y]]! \n\t"
"vld1.32 {q2-q3}, [%[b]]! \n\t"

"vmul.f32 q12, q4, d0[0] \n\t"
"vmul.f32 q13, q4, d2[0] \n\t"
"vst1.32 {q14-q15}, [%[y]]! \n\t"
"vmul.f32 q14, q4, d4[0] \n\t"
"vmul.f32 q15, q4, d6[0] \n\t"
"vmla.f32 q12, q5, d0[1] \n\t"
"vmla.f32 q13, q5, d2[1] \n\t"
"vmla.f32 q14, q5, d4[1] \n\t"
"vmla.f32 q15, q5, d6[1] \n\t"
"vmla.f32 q12, q6, d1[0] \n\t"
"vmla.f32 q13, q6, d3[0] \n\t"
```

```
      "vmla.f32 q14, q6, d5[0] \n\t"
      "vmla.f32 q15, q6, d7[0] \n\t"
      "vmla.f32 q12, q7, d1[1] \n\t"
      "vmla.f32 q13, q7, d3[1] \n\t"
      "vmla.f32 q14, q7, d5[1] \n\t"
      "vmla.f32 q15, q7, d7[1] \n\t"

      "vst1.32 {q12-q13}, [%[y]]! \n\t"
      "vst1.32 {q14-q15}, [%[y]]! \n\t"

      : [a] "+r" (A_p), [b] "+r" (B_p), [y] "+r" (Y_p)
    );
}
```

B.4 Real-Valued Matrix Multiplication

```
void matmul(struct params *params)
{
  float *A = params->in1;
  float *B = params->in2;
  float *Y = params->out;
  register float *A_p asm("r4") = A;
  register float *B_p asm("r5") = B;
  register float *Y_p asm("r6") = Y;
  const unsigned A_row_cnt = params->in1_row_cnt;
  const unsigned A_col_cnt = params->in1_col_cnt;
  const unsigned B_row_cnt = params->in2_row_cnt;
  register const unsigned B_col_cnt =
    params->in2_col_cnt;
  const unsigned Y_row_cnt = A_row_cnt;
  const unsigned Y_col_cnt = B_col_cnt;
  const unsigned Y_dim = Y_row_cnt * Y_col_cnt;
```

```
register float32x4_t aaaa asm("q0");
register float32x4_t bbbb asm("q1");
register float32x4_t yyyy asm("q2");

memset(Y, 0, Y_dim * sizeof(float));

for (unsigned a_row = 0; a_row < A_row_cnt; ++a_row) {
  B_p = B;
  unsigned y_row_wo_col = a_row * B_col_cnt;
  for (unsigned a_col = 0;
    a_col < A_col_cnt;
    ++a_col) {
    Y_p = Y + y_row_wo_col;
    asm volatile (
      "vld1.32 {d0[],d1[]}, [%[a]]! \n\t"
      : [a] "+r" (A_p)
    );
    for (register unsigned b_col = 0;
      b_col < B_col_cnt;
      b_col += 8) {
      asm volatile (
        "vld1.32 {d2,d3}, [%[b]]! \n\t"
        "vld1.32 {d4,d5}, [%[y]]  \n\t"
        "vmla.f32 q2, q0, q1    \n\t"
        "vst1.32 {d4,d5}, [%[y]]! \n\t"

        "vld1.32 {d6,d7}, [%[b]]! \n\t"
        "vld1.32 {d8,d9}, [%[y]]  \n\t"
        "vmla.f32 q4, q0, q3    \n\t"
        "vst1.32 {d8,d9}, [%[y]]! \n\t"
        : [b] "+r" (B_p), [y] "+r" (Y_p)
      );
```

```
      }
    }
  }
}
```

B.5 Complex-Valued Matrix Multiplication

```
void matmulc(struct params *params)
{
  float complex *A = params->in1;
  float complex *B = params->in2;
  float complex *Y = params->out;
  register float complex *A_p asm("r4") = A;
  register float complex *B_p asm("r5") = B;
  register float complex *Y_p asm("r6") = Y;
  const unsigned A_row_cnt = params->in1_row_cnt;
  const unsigned A_col_cnt = params->in1_col_cnt;
  const unsigned B_row_cnt = params->in2_row_cnt;
  register const unsigned B_col_cnt =
    params->in2_col_cnt;
  const unsigned Y_row_cnt = A_row_cnt;
  const unsigned Y_col_cnt = B_col_cnt;
  const unsigned Y_dim = Y_row_cnt * Y_col_cnt;

  memset(Y, 0, Y_dim * sizeof(float complex));

  for (unsigned a_row = 0; a_row < A_row_cnt; ++a_row) {
    B_p = B;
    unsigned y_row_wo_col = a_row * B_col_cnt;
    for (unsigned a_col = 0;
      a_col < A_col_cnt;
      ++a_col) {
```

```
      Y_p = Y + y_row_wo_col;
      asm volatile (
        "vld1.32 {d0[],d1[]}, [%[a]]! \n\t"
        "vld1.32 {d2[],d3[]}, [%[a]]! \n\t"
        : [a] "+r" (A_p)
      );
      for (register unsigned b_col = 0;
        b_col < B_col_cnt;
        b_col += 8) {
        asm volatile (
          "vld2.32 {d4,d5,d6,d7}, [%[b]]! \n\t"
          "vld2.32 {d8,d9,d10,d11}, [%[y]] \n\t"
          "vmla.f32 q4, q0, q2 \n\t"
          "vld2.32 {d12,d13,d14,d15}, [%[b]]! \n\t"
          "vmls.f32 q4, q1, q3 \n\t"
          "vmla.f32 q5, q0, q3 \n\t"
          "vmla.f32 q5, q1, q2 \n\t"
          "vst2.32 {d8,d9,d10,d11}, [%[y]]! \n\t"
          "vld2.32 {d16,d17,d18,d19}, [%[y]] \n\t"
          "vmla.f32 q8, q0, q6 \n\t"
          "vmls.f32 q8, q1, q7 \n\t"
          "vmla.f32 q9, q0, q7 \n\t"
          "vmla.f32 q9, q1, q6 \n\t"
          "vst2.32 {d16,d17,d18,d19}, [%[y]]! \n\t"
          : [b] "+r" (B_p), [y] "+r" (Y_p)
        );
      }
    }
  }
}
```

B.6 Convolution

```
void convc(struct params *params)
{
  float complex *X = params->in1;
  float complex *H = params->in2;
  float complex *Y = params->out;
  unsigned X_SIZE = params->in1_col_cnt;
  unsigned H_SIZE = params->in2_col_cnt;

  register float complex *X_p;
  register float complex *H_p = H;

  for (unsigned y = 0; y < X_SIZE - H_SIZE + 1; ++y) {
    X_p = X + y;
    asm volatile (
      "veor q0, q0\n\t"
      "veor q1, q1\n\t"
    );
    for (unsigned i = 0; i < H_SIZE; i += 16) {
    asm volatile (
      "vld2.32 {q2-q3}, [%[x]]!\n\t"
      "vld2.32 {q4-q5}, [%[h]]!\n\t"
      "vmla.f32 q0, q2, q4\n\t" // real * real
      "vmla.f32 q1, q2, q5\n\t" // real * imag
      "vld2.32 {q6-q7}, [%[x]]!\n\t"
      "vmls.f32 q0, q3, q5\n\t" // imag * imag
      "vmla.f32 q1, q3, q4\n\t" // imag * real

      "vld2.32 {q8-q9}, [%[h]]!\n\t"
      "vmla.f32 q0, q6, q8\n\t"
      "vmla.f32 q1, q6, q9\n\t"
      "vld2.32 {q10-q11}, [%[x]]!\n\t"
```

```
      "vmls.f32 q0, q7, q9\n\t"
      "vmla.f32 q1, q7, q8\n\t"

      "vld2.32 {q12-q13}, [%[h]]!\n\t"
      "vmla.f32 q0, q10, q12\n\t"
      "vmla.f32 q1, q10, q13\n\t"
      "vld2.32 {q2-q3}, [%[x]]!\n\t"
      "vmls.f32 q0, q11, q13\n\t"
      "vmla.f32 q1, q11, q12\n\t"

      "vld2.32 {q4-q5}, [%[h]]!\n\t"
      "vmla.f32 q0, q2, q4\n\t"
      "vmla.f32 q1, q2, q5\n\t"
      "vmls.f32 q0, q3, q5\n\t"
      "vmla.f32 q1, q3, q4\n\t"

      : [x] "+r" (X_p), [h] "+r" (H_p)
    );
    }
    asm volatile (
      "vpadd.f32 d0, d0, d1\n\t"
      "vadd.f32 s30, s0, s1\n\t"
      "vpadd.f32 d2, d2, d3\n\t"
      "vadd.f32 s31, s4, s5\n\t"
      "vstmia %[y]!, {s30,s31}\n\t"
      : [y] "+r" (Y)
    );
  }
}
```

B.7 Fast Fourier Transform

```
fft16:
  ...
  @ r5 contains the address of the input array
  @ r4 contains the address of the output array
  @ :GLOBAL is the base address of rodata section
  @ :GLOBAL + 2048 contains the twiddle factor array
  strd r0, [sp]
  add r1, r5, #32
  add r2, r5, #96
  add r7, r5, #64
  vld2.32 {d20-d23}, [r5]
  movw r3, #:GLOBAL
  add r0, r4, #32
  vld2.32 {d8-d11}, [r2]
  movt r3, #:GLOBAL
  add r6, r3, #2048
  add ip, r3, #2112
  vld2.32 {d28-d31}, [r1]
  add r5, r3, #2080
  add r1, r4, #64
  add r2, r4, #96
  vld2.32 {d4-d7}, [r7]
  vld2.32 {d0-d3}, [r6]
  vmov q8, q10
  vmov q10, q5
  vmov q12, q14
  vmov q13, q15
  vmov q15, q4
  vmov q4, q2
  vmov q14, q3
  vsub.f32 q2, q13, q5
  vadd.f32 q3, q12, q15
```

```
vadd.f32 q9, q8, q4
vsub.f32 q5, q8, q4
vsub.f32 q6, q12, q15
vadd.f32 q7, q13, q10
vadd.f32 q12, q11, q14
vsub.f32 q11, q11, q14
vsub.f32 q14, q9, q3
vsub.f32 q8, q5, q2
vadd.f32 q9, q9, q3
vadd.f32 q3, q5, q2
vzip.32 q14, q8
vsub.f32 q10, q12, q7
vzip.32 q9, q3
vadd.f32 q13, q12, q7
vsub.f32 q15, q11, q6
vadd.f32 q11, q11, q6
vld2.32 {d8-d11}, [ip]
vmov q2, q0
vzip.32 q13, q15
vzip.32 q10, q11
vld2.32 {d12-d15}, [r5]
vmov q12, q1
vmov d1, d29
vmov d0, d19
vmov d29, d28
vmov d28, d18
vmul.f32 q1, q0, q2
vmul.f32 q9, q0, q12
vmov d0, d27
vmov d1, d21
vmov d27, d20
vmla.f32 q9, q0, q2
```

```
vmov q2, q4
vmls.f32 q1, q0, q12
vmov q0, q5
vmov q10, q9
vmov d18, d7
vmov d19, d17
vmov d17, d16
vmov d16, d6
vmov d7, d23
vmov d6, d31
vmov d31, d22
vmul.f32 q4, q9, q4
vmul.f32 q9, q9, q5
vmov q5, q7
vmul.f32 q7, q8, q6
vmls.f32 q4, q3, q0
vmul.f32 q8, q8, q5
vmla.f32 q9, q3, q2
vmls.f32 q7, q15, q5
vmla.f32 q8, q15, q6
vadd.f32 q3, q1, q4
vadd.f32 q12, q10, q9
vadd.f32 q15, q14, q7
vsub.f32 q11, q10, q9
vsub.f32 q14, q14, q7
vadd.f32 q10, q13, q8
vadd.f32 q6, q15, q3
vadd.f32 q7, q10, q12
vst2.32 {d12-d15}, [r4]
vsub.f32 q1, q1, q4
vsub.f32 q0, q13, q8
vadd.f32 q4, q14, q11
```

```
vsub.f32 q5, q0, q1
vst2.32 {d8-d11}, [r0]
vsub.f32 q2, q15, q3
vsub.f32 q3, q10, q12
vst2.32 {d4-d7}, [r1]
vsub.f32 q12, q14, q11
vadd.f32 q13, q0, q1
vst2.32 {d24-d27}, [r2]
...
```