Comparing Android Runtime with native: Fast Fourier Transform on Android

André Danielsson

April 5, 2017

KTH – Royal Institute of Technology

Master's Thesis in Computer Science

KTH Supervisor: Erik Isaksson
Bontouch Supervisor: Erik Westenius
Examiner: Olle Bälter

Abstract

Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

Sammanfattning

Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

Preface

Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

André Danielsson

Contents

GLOSSARY

ABI *Application Binary Interfaces*. 2

Android Mobile operating system. 1

AOT *Ahead-Of-Time*. 7

API *Application Programming Interface*. 7

APK *Android Package*. 8, 10

Apps Applications. 2

ART *Android Runtime*. 2, 7

Clang Compiler used by the NDK. 12

CMake Build tool used by the NDK. 10

DEX *Dalvik Executable*. 9

DFT *Discrete Fourier Transform* – Converts signal from time domain to frequency domain. 3, 17

DVM *Dalvik Virtual Machine* – Virtual machine designed for Android. 1, 8, 9

FFT *Fast Fourier Transform* – Algorithm that implements the Discrete Fourier Transform. 2–5, 17, 22–24, 26, 27, 30–32

FFTW *Fastest Fourier Transform in the West*. 22

HAL *Hardware Abstraction Layer*. 7

JIT *Just-In-Time*. 1, 7

JNI *Java Native Interface* – Framework that helps Java interact with native code. 2–4, 10, 11, 22, 23, 27–30

JVM *Java Virtual Machine*. 8

LLVM *Low Level Virtual Machine* – collection of compilers. 2, 12

NDK *Native Development Kit* – used to write android applications in C or C++. 2, 3, 10, 12, 16, 22

NEON Tool that allows the use of vector instruction for the ARMv7 architecture. 16, 23

SDK *Software Development Kit*. 3, 7

SIMD *Single Instruction Multiple Data* – Operations that can be executed for multiple operands. 16

SSL *Secure Sockets Layer*. 7

List of Figures

List of Tables

CHAPTER 1

Introduction

This thesis explores differences in performance between bytecode and native libraries. The Fast Fourier Transform algorithm is the focus of this degree project. Experiments was be carried out to investigate how and when it is necessary to implement the Fast Fourier Transform in Java or C++ on Android.

1.1 Background

Android is an operating system for smartphones and as of November 2016 it is the most used [2]. One reason for this is because it was designed to be run on multiple different architectures [3]. Google states that they want to ensure that manufacturers and developers have an open platform to use and therefore releases Android as Open Source software [4]. The Android kernel is based on the Linux kernel although with some alterations to support the hardware of mobile devices.

Android applications are mainly written in Java to ensure portability in form of architecture independence. By using a virtual machine to run a Java app, you can use the same bytecode on multiple platforms. To ensure efficiency on low resources devices, a virtual machine called Dalvik was developed. Applications (apps) on Android have been using the Dalvik Virtual Machine (DVM) until Android version 5 [5] in November of 2014 [6]. Since then, Dalvik has been replaced by Android Runtime. Android Runtime, ART for short, differs from Dalvik in that it uses Ahead-Of-Time () compilation. This means that the bytecode is compiled during the installation of the app. Dalvik, however, exclusively uses a concept called Just-In-Time (JIT) compilation, meaning that code is compiled during runtime when needed. ART uses Dalvik bytecode to compile an appli-

cation, allowing most Apps that are aimed at Dalvik Virtual Machine to work on devices running ART.

To allow developers to reuse libraries written in C or C++ or just to write low level code, a tool called Native Development Kit (NDK) was released. It was first released in June 2009 [7] and has since gotten improvements such as new build tools, compiler versions and support for additional Application Binary Interfaces (ABI). ABIs are mechanisms that are used to allow binaries to communicate using specified rules. With the NDK, the developers can choose to write parts of an app in so called *native code*. This is used when wanting to do compression, graphics and other performance heavy tasks.

1.2 Problem

Nowadays, mobile phones are fast enough to handle heavy calculations on the devices themselves. To ensure that resources are spent in an efficient manner, this study has investigated how significant the performance boost is when compiling the Fast Fourier Transform (FFT) with the NDK tools instead of by ART. Multiple different implementations of FFTs was be evaluated as well as the effects of the Java Native Interface (JNI), a framework for communicating between Java code and native shared libraries. The following research question was formed on the basis of these requirements:

Is there a significant performance difference between implementations of a Fast Fourier Transform (FFT) in native code, compiled by Clang, and Dalvik bytecode, compiled by Android Runtime, on Android?

1.3 Purpose

This thesis is a study that evaluates when and where there is a gain in writing a part of an Android application in C++. One purpose of this study is to educate the reader about the process of porting parts of an app to native code using the Native Development Kit (NDK). Another is to explore the topic of performance differences between Android Runtime (ART) and native code compiled by Clang/LLVM. Because ART is relatively new (Nov 2014) [6], this study would contribute with more information about to the performance of ART and how it performs compared to native code compiled by the NDK. The results of the study can also be used to value the decision of implementing a given algorithm or other solutions in native code instead of Java. It is valuable to know

how efficient an implementation in native code is, depending on the size of the data.

The reason you would want to write part of an application in native code is to potentially get better execution times of computational heavy tasks such as the Fast Fourier Transform (FFT). The FFT is an algorithm that computes the Discrete Fourier Transform (DFT) of a signal. It is primarily used to analyze the components of a signal. This algorithm is used in signal processing and has multiple purposes such as image compression (taking photos), voice recognition (Siri, Google Assistant), fingerprint scanning (unlocking device) to name a few. Another reason you would want to write native libraries is to reuse already written code in C or C++ and incorporate it into your project. This allows the app to become more platform independent. Shared code can be used in a computer app, Apple iOS app and more.

Some of the findings in this thesis can help decide which method of programming for Android that should be used for a given problem. For some problems, it is necessary to choose the appropriate programming method to ensure that an application is smooth and responsive. It is therefore important to know when and where it is necessary to optimize code. Further, when developing for Android there are multiple types of problems that occur and it is relevant to know which problems are worth solving in NDK rather than the Software Development Kit (SDK).

1.4 Goal

The goal of this project was to examine the efficiency of ART and how it compares to natively written code using the NDK in combination with the Java Native Interface (JNI). This report presents a study that investigates the relevance of using the NDK to produce efficient code. Further, the cost to pass through the JNI is also a factor when analysing the code. A discussion about to what extent the efficiency of the program reduces the simplicity of the code is also present. For people who are interested to know about the impacts of implementing algorithms in C++ for Android, this study could be of some use.

1.5 Procedure

The method used to find the relevant literature and previous studies was to search through databases using boolean expressions. By specifying synonyms and required keywords,

more literature could be found. Figure 1.1 contains the expression that was used to narrow down the search results to relevant articles.

(NDK OR JNI) AND
Android AND
(benchmark* OR efficien*) AND
(Java OR C OR C++) AND
(Dalvik OR Runtime OR ART)

Figure 1.1: Expression used to filter out relevant articles

The execution time of the programs varied because of factors such as scheduling, CPU clock frequency scaling and other uncontrollable behaviour caused by the operating system. To get accurate measurements, a mean of a large numbers of runs were calculated for each program. Additionally, it was also necessary to calculate the standard error of each set of execution times. With the standard error we can determine if the difference in execution time between two programs are statistically significant or not.

Three different tests were carried out to gather enough data to be able to make reasonable statements about the results. The first one was to find out how significant the overhead of JNI is. This is important to know to be able to see exactly how large the cost of going between Java and native code is in relation to the actual work. The second test was a comparison between multiple well known libraries to find how much they differ in performance. In the third and final test, two comparable implementations of FFT were chosen, one in Java and one in C++. These two implementations were then optimized using different optimization techniques and later compared.

1.6 Delimitations

This thesis does only cover a performance evaluation of the FFT algorithm and does not go into detail on other related algorithms. The decision of choosing the FFT was due to it being a common algorithm to use in signal analysis. This thesis does not investigate

the performance differences for FFT in parallel due to the complexity of the Linux kernel used on Android. This would require more knowledge outside the scope of this project and would result in a too broad of a subject. The number of optimization methods covered in this thesis were also delimited to the scope of this degree project.

1.7 Limitations

The tests were carried out on the same phone under the same circumstances to reduce the number of affecting factors. By developing a benchmark program that run the tests during a single session, it was possible to reduce the varying factors that could affect the results. Because you cannot control the Garbage Collector in Java, it is important to have this in mind when constructing tests and analyzing the data.

1.8 Ethics and Sustainability

An ethical aspect of this thesis is that because there could be people making decisions based on this report, it is important that the conclusions are presented together with its conditions so that there are no misunderstandings. Another important thing is that every detail of each test is explicitly stated so that every test can be recreated by someone else. Finally, it is necessary to be critical of the results and find how reasonable the results are.

Environmental sustainability is kept in mind in this investigation because there is an aspect of battery usage in different implementations of algorithms. The less number of instructions an algorithm require, the faster will the CPU lower its frequency, saving power. This will also have an influence on the user experience and can therefore have an impact on the society aspect of sustainability. If this study is used as a basis on a decision that have an economical impact, this thesis would fulfil the economical sustainability goal.

1.9 Outline

- ***Chapter 1 - Introduction*** – Introduces the reader to the project. This chapter describes why this investigation is beneficial in its field and for whom it is useful.

- ***Chapter 2 - Background*** – Provides the reader with the necessary information to understand the content of the investigation.

- ***Chapter 3 - Method*** – Discusses the hardware, software and methods that are the basis of the experiment. Here, the different methods of measurement are compared and the most appropriate are chosen.

- ***Chapter 4 - Results*** – The results of the experiments are presented here.

- ***Chapter 5 - Discussion*** – Discussion regarding the results as well as the chosen method.

- ***Chapter 6 - Conclusion*** – Presents what the experiments showed and future work.

CHAPTER 2

Background

The process of developing for Android, how an app is installed and how it is being run is explained in this chapter. Additionally, common optimization techniques are described so that we can reason about the results. Lastly, some basic knowledge of the Discrete Fourier Transform is required when discussing differences in FFT implementations.

2.1 Android SDK

To allow developers to build Android apps, Google developed a Software Development Kit (SDK) to facilitate the process of writing Android applications. The Android SDK software stack is described in Figure 2.1. The Linux kernel is at the base of the stack, handling the core functionality of the device. Detecting hardware interaction, process scheduling and memory allocation are examples of services provided by the kernel. The Hardware Abstraction Layer (HAL) is an abstraction layer above the device drivers. This allows the developer to interact with hardware independent on the type of device [8].

The native libraries are low level libraries, written in C or C++, that handle functionality such as the Secure Sockets Layer (SSL) and Open GL [9]. Android Runtime (ART) features Ahead-Of-Time (AOT) compilation and Just-In-Time (JIT) compilation, garbage collection and debugging support [10]. This is where the Java code is being run and because of the debugging and garbage collection support, it is also beneficial for the developer to write applications against this layer.

The Java API Framework is the Java library you use when controlling the Android UI. It is the reusable code for managing activities, implementing data structures and designing

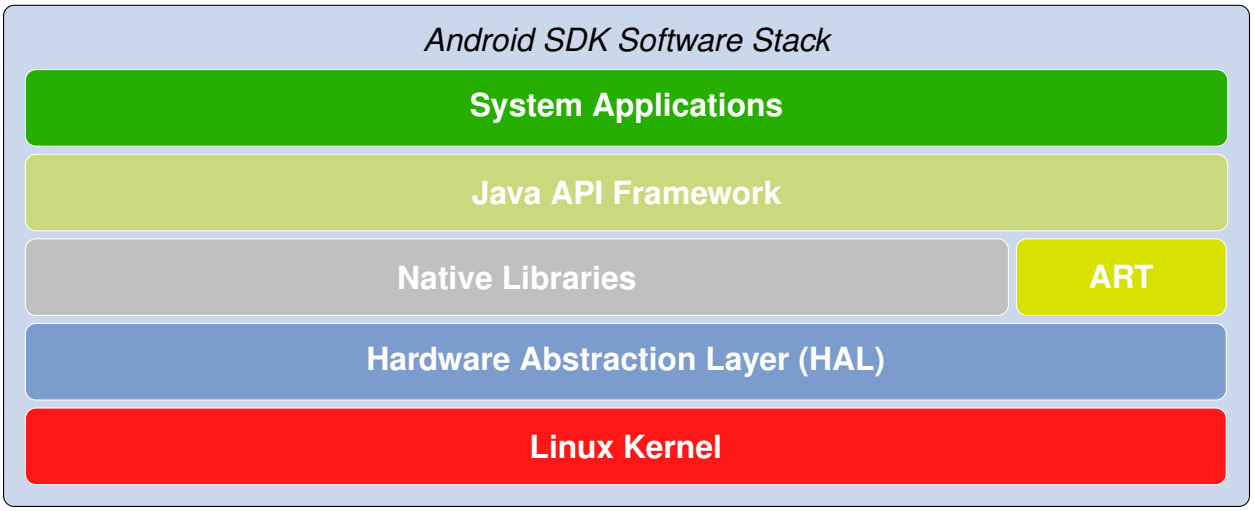


Figure 2.1: Android SDK Software Stack [10]

the application. The System Application layer represents the functionality that allows a third-party app to communicate with other apps. Example of usable applications are email, calendar and contacts [10].

All applications for Android are packaged in so called Android Packages (APK). These APKs are zipped archives that contain all the necessary resources required to run the app. Such resources are the AndroidManifest.xml file, Dalvik executables (.dex files), native libraries and other files the application depends on.

2.2 Dalvik Virtual Machine

Compiled Java code is executed on a virtual machine called the Java Virtual Machine (JVM). The reason for this is to allow compiled code to become portable. This way, every device, independent on architecture, with a JVM installed will be able to run the same code. The Android operating system is designed to be installed on many different devices [3]. Because of the many different devices, user applications would have to be compiled for all possible platforms it should work on. For this reason, Java bytecode is a sensible choice when wanting to distribute compiled applications.

The Dalvik Virtual Machine (DVM) is the VM initially used on Android. One difference between DVM and JVM is that the DVM uses a register-based architecture while the JVM uses a stack-based architecture. The most common virtual machine architecture is the stack-based [11, p. 158]. A stack-based architecture evaluates each expression directly

on the stack and always has the last evaluated value on top of the stack. Thus, only a stack pointer is needed to find the next instruction on the stack.

Contrary to this behaviour, a register-based virtual machine works more like a CPU. It uses a set of registers where it will place operands by fetching them from memory. One advantage of using a register-based architecture is that fetching data between registers is faster than fetching or storing data onto the hardware stack. The biggest disadvantage of using register-based architecture is that the compilers must be more complex than for stack-based architecture. This is because the code generators must take register management into consideration [11, p. 159-160].

The DVM is a virtual machine optimized for devices where resources are limited [12]. The main focus of the DVM is to lower memory consumption and lower the number of instructions needed to fulfil a task. Using register-based architecture, it is possible to execute more virtual machine instructions compared to a stack-based architecture [13].

Dalvik executables, or DEX files, are the files where Dalvik bytecode is stored. They are created by converting a Java class file to the DEX format. They are of a different structure than Java class files. Some differences are the header types that describes the data. One example of the differences is the string constant fields that are present in the DEX-file.

2.3 Android Runtime

Android Runtime is the new default runtime for Android as of version 5.0 [5]. The big improvement over Dalvik is the fact that applications are compiled to binary when they are installed on the device, rather than during runtime of the app. This results in faster start-up [14] and lets the compiler use more heavy optimization that is not otherwise possible during runtime. However, if the whole application is compiled ahead of time it is no longer possible to do any runtime optimizations. An example of a runtime optimization is to inline methods or functions that are called frequently.

When an app is installed on the device, a program called **dex2oat** converts a DEX-file to an executable file called an oat-file [15]. This oat-file is in the Executable and Linkable Format (ELF) and can be seen as a wrapper of multiple DEX-files [16]. An improvement made in Android Runtime is the optimized garbage collector. Changes include a decrease from two to one GC pause, reduced memory fragmentation (reduces calls to `GC_FOR_ALLOC`) and parallelization techniques to lower the time it takes to collect [15].

2.4 Native Development Kit

Native Development Kit (NDK) is a set of tools to help writing native apps for Android. It contains the necessary libraries, compilers, build tools and debugger for developing low level libraries. Google recommends using the NDK for two reasons: run computationally intensive tasks and usage of already written libraries [17]. Because Java is the supported language on Android, due to security and stability, native development is not recommended to use to build full apps, with an exception when developing games.

Historically, native libraries have been built using Make. Make is a tool used to coordinate compilation of source files. Android makefiles, `Android.mk` and `Application.mk`, are used to set compiler flags, choose which architectures that a project should be compiled for, location of source files and more. With Android Studio 2.2 CMake was introduced as the default build tool [18]. CMake is a more advanced tool for generating and running build scripts.

At each compilation, the architectures the source files will be built against must be specified. The source file(s) generated will be placed in a folder structure where the source file is located in a folder that determines the architecture. Each architecture-folder is located in a folder called `lib`. This folder will be placed at the root of the APK.

```
lib/
|--armeabi-v7a/
|  |--lib[libname].so
|--x86/
   |--lib[libname].so
```

2.4.1 Java Native Interface

To be able to call native libraries from Java code, a framework named Java Native Interface (JNI) is used. Using this interface, C/C++ functions are mapped as methods and primitive data types are converted between Java and C/C++. For this to work, special syntax is needed for JNI to recognize which method in which class a native function should correspond to.

To mark a function as native in Java, a special keyword called `native` is used to define a method. The library which implements this method must also be included in the same class. By using the `System.loadLibrary("mylib")` call, we can specify the name of the

shared object that should be loaded. Inside the native library we must follow a function naming convention to map a method to a function. The rules are that you must start the function name with `Java` followed by the package, class and method name. Figure 2.2 demonstrates how to map a method to a native function.

```
private native int myFun();
```

⇕

```
JNIEXPORT jint JNICALL
Java_com_example_MainActivity_myFun (JNIEnv *env, jobject thisObj)
```

Figure 2.2: Native method declaration to implementation.

The JNI also provides a library for C and C++ for handling the special JNI data types. They can be used to determine the size of a Java array, get position of elements of an array and handling Java objects. In C and C++ you are given a pointer to a list of JNI functions (`JNIEnv*`). With this pointer, you can communicate with the JVM [19, p. 22]. You typically use the JNI functions to fetch data from handled by the JVM, call methods and create objects.

The second parameter to a JNI function is of the `jobject` type. This is the current Java object that has called this specific JNI function. It can be seen as an equivalent to the `this` keyword in Java and C++ [19, p. 23]. There is a function-pair available in the `JNIEnv` pointer called `GetDoubleArrayElements()` and `ReleaseDoubleArrayElements()`. There are also functions for other primitive types such as `GetIntArrayElements()`, `GetShortArrayElements()` and others. `GetDoubleArrayElements()` is used to convert a Java array to a native memory buffer [19, p. 159]. This call also tries to "pin" the elements of the array.

Pinning allows JNI to provide the reference to an array directly instead of allocating new memory and copying the whole array. This is used to make the call more efficient although it is not always possible. Some implementations of the virtual machine does not allow this because it requires that the behaviour of the garbage collector must be changed to support this [19, p. 158]. There are two other functions, `GetPrimitiveArrayCritical()` and `ReleasePrimitiveArrayCritical()`, that can be used to avoid garbage collection in native code. Between these function calls, the native code should not run forever, no calls to any of the JNI functions are allowed and it is prohibited to block a thread that depends on a VM thread to continue.

2.4.2 LLVM and Clang

LLVM (Low Level Virtual Machine) is a suite that contains a set of compiler optimizers and backends. It is used as a foundation for compiler frontends and supports many architectures. An example of a frontend tool that uses LLVM is Clang. Clang is used compile C, C++ and Objective-C source code [20].

Clang is as of March 2016 (NDK version 11) [21], the only supported compiler in the NDK. Google has chosen to focus on supporting the Clang compiler instead of the GNU GCC compiler. This means that there is a bigger chance that a specific architecture used on an Android device is supported in the NDK. This also allows Google to focus on developing optimizations for these architectures with only one supported compiler.

2.5 Code Optimization

There are many ways your compiler can optimize your code during compilation. This chapter will first present some general optimization measures taken by the optimizer and will then describe some language specific methods for optimization.

Loop unrolling

Loop unrolling is a technique used to optimize loops. By explicitly coding multiple iterations in the body of the loop, it is possible to lower the amount of jump instructions in the produced code. Figure 2.3 demonstrates how unrolling works by decreasing the number of iterations but adding lines in the loop body. The loop unroll executes two iterations of the first code per iteration. It is therefore necessary to update the `i` variable accordingly. Figure 2.4 describes how the change could be represented in assembly language.

The gain in using loop unrolling is that you "save" the same amount of jump instructions as the amount of "hard coded" iterations you add. In theory, it is also possible to optimize even more by changing the offset of `LOAD WORD` instructions as shown in Figure 2.5. Then you would not need to update the iterator as often.

```
for (int i = 0; i < 6; ++i) {
    a[i] = a[i] + b[i];
}
```

(a) Normal

```
for (int i = 0; i < 6; i+=2) {
    a[i] = a[i] + b[i];
    a[i+1] = a[i+1] + b[i+1];
}
```

(b) One unroll

Figure 2.3: Loop unrolling in C

```
$s1 - a[] address | $s4 - value of a[x]
$s2 - b[] address | $s5 - value of b[x]
$s3 - i           | $s6 - value 6
```

```
1 loop: lw $s4, 0($s1) # Load a[i]
2       lw $s5, 0($s2) # Load b[i]
3       add $s4, $s4, $s5 # a[i] + b[i]
4       sw $s4, 0($s1)
5       addi $s1, $s1, 4 # next element
6       addi $s2, $s2, 4 # next element
7       addi $s3, $s3, 1 # i++
8       bge $s3, $s6, loop
```

(a) Normal

```
1  loop: lw $s4, 0($s1)
2        lw $s5, 0($s2)
3        add $s4, $s4, $s5
4        sw $s4, 0($s1)
5        addi $s1, $s1, 4
6        addi $s2, $s2, 4
7        addi $s3, $s3, 1
8        lw $s4, 0($s1)
9        lw $s5, 0($s2)
10       add $s4, $s4, $s5
11       sw $s4, 0($s1)
12       addi $s1, $s1, 4
13       addi $s2, $s2, 4
14       addi $s3, $s3, 1
15       bge $s3, $s6, loop
```

(b) One unroll

Figure 2.4: Loop unrolling in assembly

Inlining

Inlining allows the compiler to swap all the calls to an inline function with the content of the function. This removes the need to do all the preparations for a function call such as saving values in registers and preparing parameters and return values. This comes at a cost of a larger program if there are many calls to this function in the code and if the function is large. It is very useful to use inline functions in loops that are run many times. This is an optimization that can be used manually in C and C++ using the `inline` keyword and can also be optimized by the compiler.

```
$s1 - a[] address | $s4 - value of a[x]
$s2 - b[] address | $s5 - value of b[x]
$s3 - i           | $s6 - value 6
```

```
1  loop: lw $s4, 0($s1)
2        lw $s5, 0($s2)
3        add $s4, $s4, $s5
4        sw $s4, 0($s1)
5        addi $s1, $s1, 4
6        addi $s2, $s2, 4
7        addi $s3, $s3, 1
8        lw $s4, 0($s1)
9        lw $s5, 0($s2)
10       add $s4, $s4, $s5
11       sw $s4, 0($s1)
12       addi $s1, $s1, 4
13       addi $s2, $s2, 4
14       addi $s3, $s3, 1
15       bge $s3, $s6, loop
```

(a) One unroll

```
1  loop: lw $s4, 0($s1)
2        lw $s5, 0($s2)
3        add $s4, $s4, $s5
4        sw $s4, 0($s1)
5        lw $s4, 4($s1)
6        lw $s5, 4($s2)
7        add $s4, $s4, $s5
8        sw $s4, 4($s1)
9        addi $s1, $s1, 8
10       addi $s2, $s2, 8
11       addi $s3, $s3, 2
12       bge $s3, $s6, loop
```

(b) Optimized unroll

Figure 2.5: Optimized loop unrolling in assembly

Constant folding

Constant folding is a technique used to reduce the time it takes to evaluate an expression in runtime [22, p. 329]. By finding which variables that already have a value, the compiler can calculate and assign constants in compile time instead of during runtime. This method of analyzing the code to find expressions consisting of variables that are possible to calculate is called *Constant Propagation* as seen in Figure 2.6.

```
int x = 10;
int y = x * 5 + 3;
```

(a) Before optimization

```
int x = 10;
int y = 53;
```

(b) Constant propagation optimization

Figure 2.6: Constant Propagation

Loop Tiling

When processing elements in a large array multiple times it is beneficial to utilize as many reads from cache as possible. If the array is larger than the cache, it will kick out earlier elements for the next pass through the array. By processing partitions of the array multiple times before going on to next partition, temporal cache locality can help the program run faster. Temporal locality means that you can find a previously referenced value in the cache if you are trying to access it again. As Figure 2.7 shows, by introducing a new loop that operate over a small enough partition of the array such that every element is in cache, we will reduce the number of cache misses.

```
for (i = 0; i < NUM_REPS; ++i) {
    for (j = 0; j < ARR_SIZE; ++j) {
        a[j] = a[j] * 17;
    }
}
```

(a) Before loop tiling

```
for (j = 0; j < ARR_SIZE; j += 1024) {
    for (i = 0; i < NUM_REPS; ++i) {
        for (k = j; k < (j + 1024); ++k) {
            a[k] = a[k] * 17;
        }
    }
}
```

(b) After loop tiling

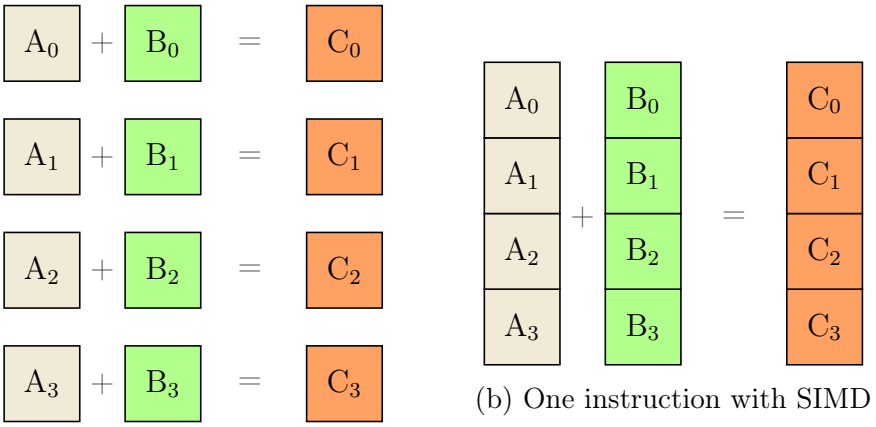
Figure 2.7: Loop Tiling

2.5.1 Java

In Java, an array is created during runtime and cannot change its size after it is created. This means that it will always be placed on the heap and the garbage collector will handle the memory it resides on when it is no longer needed. By keeping an array reference in scope and reusing the same array, we can circumvent this behaviour and save some instructions by not needing to ask for more memory from the heap.

2.5.2 C++

C and C++ arrays have predefined sizes and are located on the program stack. This makes the program run faster because it does not need to call malloc or new and ask for more memory on the heap. This require that the programmer knows the required size of the array in advance and is not always possible or memory efficient.



(a) Four separate instructions

(b) One instruction with SIMD

Figure 2.8: Single Instruction Multiple Data [24]

NEON

Android NDK includes a tool called NEON that contains functions which enables Single Instruction Multiple Data (SIMD). SIMD is an efficient way of executing the same type of operation on multiple operands at the same time. Figure 2.8 describes this concept where instead of operating on one piece of data at the time, a larger set of data that uses the same operation can be processed with one operation.

NEON provides a set of functions compatible with the ARM architecture. These functions can perform operations on double word and quad word registers. The reason you would want to use SIMD because you can have instructions that loads blocks of multiple values and operates on these blocks. The process starts by reading the data into larger vector registers, operate on these registers and storing the results as blocks [23]. This way you will have less instructions than if you loaded one element at a time and operated on only that value.

SIMD has some prerequisites on the data that is being processed. Firstly, the data blocks must line up meaning that you cannot operate between two operands that are not in the same area of the block. Secondly, all the operands of a block must be of the same type.

2.6 Discrete Fourier Transform

The Discrete Fourier Transform (DFT) is a method of converting a sampled signal from the time domain to the frequency domain. In other words, the DFT takes an observed signal and dissects each component that would form the observed signal. Every component of a signal can each be described as a sinusoidal wave with a frequency, amplitude and phase.

If we observe Figure 2.9, we can see how a signal in time domain looks like in frequency domain. The function displayed in the time domain consists of three sine components, each with its own amplitude and frequency. What the graph of the frequency domain shows, is the amplitude of each frequency. This can then be used to analyze the input signal.

One important thing to note is that you must sample at twice the frequency you want to analyze. The Nyquist sampling theorem states that [25]:

The sampling frequency should be at least twice the highest frequency contained in the signal.

In other words, you have to be able to reconstruct the signal given the samples [26, Ch 3]. If you are given a signal that is constructed of frequencies that are at most 500 Hz, your sample frequency must be at least 1000 samples per second to be able to find the amplitude for each frequency.

Equation 2.1 [27, p. 92] describes the mathematical process of converting a signal x to a spectrum X of x where N is the number of samples, n is the time step and k is the frequency sample. When calculating $X(k) \forall\ k \in [0, N-1]$ we clearly see that it will take N^2 multiplications. In 1965, Cooley and Tukey published a paper on an algorithm that could calculate the DFT in less than $2N\log(N)$ multiplications [28] called the Fast Fourier Transform (FFT).

$$X(k) = \sum_{n=0}^{N-1} x(n) \cdot e^{-j\frac{2\pi}{N}kn}, \quad k = 0, 1, 2, \ldots, N-1 \tag{2.1}$$

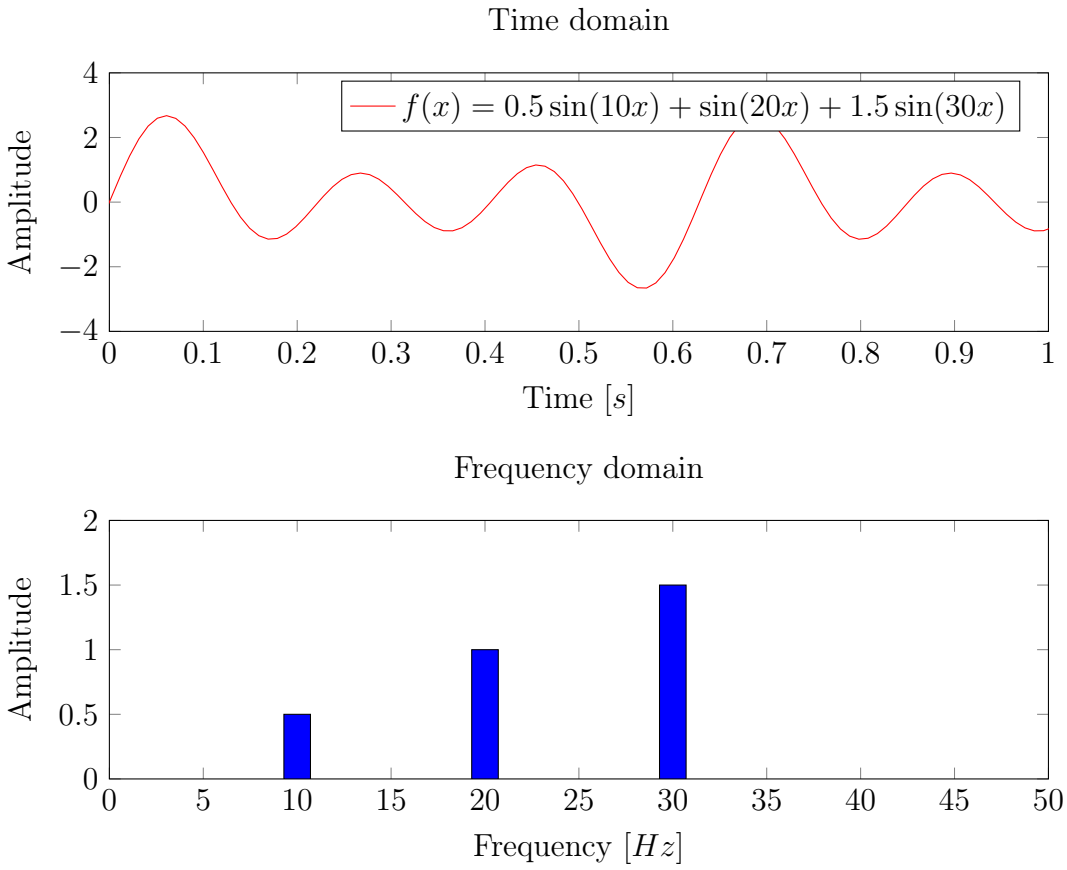


Figure 2.9: Time domain and frequency domain of a signal

2.7 Fast Fourier Transform

The Fast Fourier Transform algorithm composed by Cooley and Tukey is a recursive algorithm that runs in $O(N \log N)$ time. The following derivation is based on one found in this article from librow [1]. The notation for the imaginary number ($\sqrt{-1}$) was chosen to be j instead of i for consistency. If we expand the expression in Equation 2.1, presented in Chapter 2.6, for $N = 8$ we get:

$$X_k = x_0 + x_1 e^{-j\frac{2\pi}{8}k} + x_2 e^{-j\frac{2\pi}{8}2k} + x_3 e^{-j\frac{2\pi}{8}3k} + x_4 e^{-j\frac{2\pi}{8}4k} + x_5 e^{-j\frac{2\pi}{8}5k} + x_6 e^{-j\frac{2\pi}{8}6k} + x_7 e^{-j\frac{2\pi}{8}7k} \tag{2.2}$$

This expression can be factorized to use recurring factors of e to:

$$\begin{aligned} X_k = & \left[x_0 + x_2 e^{-j\frac{2\pi}{8}2k} + x_4 e^{-j\frac{2\pi}{8}4k} + x_6 e^{-j\frac{2\pi}{8}6k}\right] \\ & + e^{-j\frac{2\pi}{8}k}\left[x_1 + x_3 e^{-j\frac{2\pi}{8}2k} + x_5 e^{-j\frac{2\pi}{8}4k} + x_7 e^{-j\frac{2\pi}{8}6k}\right] \end{aligned} \tag{2.3}$$

In turn, each bracket can be factorized to:

$$\begin{aligned} X_k = & \left[\left(x_0 + x_4 e^{-j\frac{2\pi}{8}4k}\right) + e^{-j\frac{2\pi}{8}2k}\left(x_2 + x_6 e^{-j\frac{2\pi}{8}4k}\right)\right] \\ & + e^{-j\frac{2\pi}{8}k}\left[\left(x_1 + x_5 e^{-j\frac{2\pi}{8}4k}\right) + e^{-j\frac{2\pi}{8}2k}\left(x_3 + x_7 e^{-j\frac{2\pi}{8}4k}\right)\right] \end{aligned} \tag{2.4}$$

And finally simplified to:

$$\begin{aligned} X_k = & \left[\left(x_0 + x_4 e^{-j\pi k}\right) + e^{-j\frac{\pi}{2}k}\left(x_2 + x_6 e^{-j\pi k}\right)\right] \\ & + e^{-j\frac{\pi}{4}k}\left[\left(x_1 + x_5 e^{-j\pi k}\right) + e^{-j\frac{\pi}{2}k}\left(x_3 + x_7 e^{-j\pi k}\right)\right] \end{aligned} \tag{2.5}$$

Because of symmetry around the unit circle we have the following rules:

$$e^{j(\phi+2\pi)} = e^{j\phi}$$

$$e^{j(\phi+\pi)} = -e^{j\phi}$$

We can use these rules to prove that the factor multiplied with the second term in each parenthesis in Equation 2.5 will be 1 for $\{X_0, X_2, X_4, X_6\}$ and -1 for $\{X_1, X_3, X_5, X_7\}$. This means that each e-factor in front of the x_n will be the same for all values of k. For the third level of the recursion (Equation 2.5), we have four parentheses with two factors for a total of eight operands.

The second level (Equation 2.3) have the same sums for $\{X_0, X_4\}$, $\{X_2, X_6\}$, $\{X_1, X_5\}$ and $\{X_3, X_7\}$. They will have the factors 1, -1, $-i$ and i respectively. This level has two parentheses with four factors in each meaning that there is eight factors to sum here as for the third level. The first level (Equation 2.7) has eight unique factors to sum. In total, this recursion tree has $\log_2(8) = 3$ levels and each level has 8 factors to sum. Generally this can be described as $\log_2(N)$ levels and N factors at each level, giving a time complexity of $O(N \log N)$.

An iterative version of this algorithm would mimic the behaviour of the recursive version described previously. To demonstrate this process, the order of which the recursive

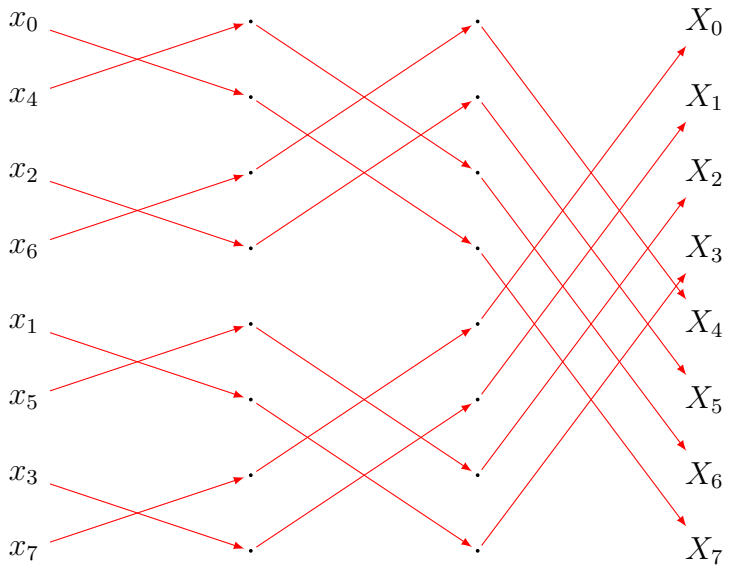


Figure 2.10: Butterfly update for 8 values [1]

Table 2.1: Bit reversal conversion table for input size 8

normal dec	**normal bin**	**reversed bin**	**reversed dec**
0	000	000	0
1	001	100	4
2	010	010	2
3	011	110	6
4	100	001	1
5	101	101	5
6	110	011	3
7	111	111	7

implementation operates is visualized in Figure 2.10. One butterfly operation is described in Figure 2.11. With mathematical notation, this relation is described as $x'_a = x_a + x_b\omega_N^k$ and $x'_b = x_a - x_b\omega_N^k$, where $\omega_N^k = e^{-j\frac{2\pi}{N}}$. The first step would be to arrange the order in which the sample array x's elements are in. One method for achieving this is to swap each element with the element at the bit-reverse of its index. Table 2.1 is a conversion table for an input array of size 8.

When we have achieved this, the operation order must be established. For the first iteration, the size of the gap between the operands is one. The next gap size is two and the third is four. It is now possible to construct an iterative algorithm. This process is shown in pseudocode in Algorithm 1. The fist part of the algorithm is the Bit reversal.

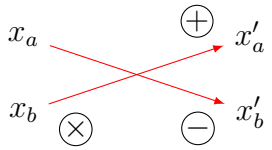


Figure 2.11: Butterfly update [1]

This has clearly $O(N)$ time complexity assuming the time complexity of bit_reverse is bounded by the size of an integer. For the butterfly updates, the outer while loop will run for $\log N$ iterations and the two inner loops will run a total of $\frac{step}{2}\frac{N}{step} = \frac{N}{2}$ times. It is now clear that the time complexity of this algorithm is $O(N \log N)$.

Algorithm 1: Iterative FFT

Data: Complex array $x = x_1, x_2, ..., x_N$ in time domain
Result: Complex array $X = X_1, X_2, ..., X_N$ in frequency domain

```
   /* Bit reversal                                                    */
1  for i ← 0 to N − 1 do
2  |  r ← bit_reverse(i)
3  |  if r > i then
4  |  |  temp ← x[i]
5  |  |  x[i] ← x[r]
6  |  |  x[r] ← temp
7  |  end
8  end
   /* Butterfly updates                                               */
9  step ← 2
10 while step ≤ N do
11 |  for k ← 0 to step/2 − 1 do
12 |  |  for p ← 0 to N/step − 1 do
13 |  |  |  curr ← p ∗ step + k
14 |  |  |  x[curr] = x[curr] + x[curr + step/2]∗ω^k_step
15 |  |  |  x[curr + step/2] = x[curr] − x[curr + step/2]∗ω^k_step
16 |  |  end
17 |  end
18 |  step ← 2 ∗ step
19 end
20 return x
```

2.8 Related work

A study called *FFT benchmark on Android devices: Java versus JNI* [29] was published in 2013 and investigated how two implementations of FFT performed on different Android devices. The main point of the study was to compare how a pure Java implementation would perform compared to a library written in C called FFTW. This library supports multi-threaded computation and this aspect is also covered in this study. Their benchmark application was run on 35 different devices with different versions to get a wide picture of how the algorithms ran on different phones.

Evaluating Performance of Android Platform Using Native C for Embedded Systems [30] explored how JNI overhead, arithmetic operations, memory access and heap allocation affected an application written in Java and native C. This study was written in 2010 when the Android NDK was relatively new. Since then, many patches has been released, improving performance of code written in native C/C++. In this study, Dalvik VM was the virtual machine that executed the Dalvik bytecode. This study found that the JNI overhead was insignificant and took 0.15 ms to run in their testing environment. Their test results indicated that C was faster than Java in every case. The performance difference was largest in the memory access test and smallest in floating point calculations.

Published in 2016, *Android App Energy Efficiency: The Impact of Language, Runtime, Compiler, and Implementation* [31] presented a performance comparison between ART and native on Android. The main focus of the report was to find how much more efficient one of them were in terms of energy consumption. Their tests consisted of measuring battery drainage in power as well as execution time of different algorithms. It also compares performance differences between ART and Dalvik. The conclusion states that native performs much better than code running on the Dalvik VM. However, code compiled by ART improves greatly from Dalvik and performs almost the same as code compiled by Android NDK.

CHAPTER 3

Method

To ensure that the experiments were carried out correctly, multiple tools for measurements were evaluated. Different implementations of the FFT are also compared to choose the ones that would typically be used in an Android project.

3.1 Experiment model

In this thesis, different aspects that can affect execution time for an FFT implementation on Android are tested. To get an overview of how much of an impact they have, the following subjects were investigated:

1. Cost of using the JNI
2. Compare well known libraries
3. Vectorization optimization with NEON, exclusive for native
4. Using `float` and `double` as primary data types

The reason it is relevant to know how significant the JNI overhead is, is because we want to see for what size of the data the transition time for going between Java and native is irrelevant compared to the total execution time of the JNI call. This would also show how much repeated calls to native code would affect the performance of a program. By minimizing the number of calls to the JNI, a program would get potentially faster.

There are many different implementations of the FFT publicly available that could be of interest for use in a project. This test demonstrates how different libraries compare. It is helpful to see how viable different implementations are on Android, both for C++ libraries and Java libraries. It can also be useful to know how small implementations can perform in terms of speed. The sample sizes used for the FFT can vary depending on the requirements for the implementation.

If the app needs to be efficient, it is common to lower the number of collected samples. This comes at a cost of accuracy. A fast FFT implementation allows for more data being passed to the FFT, improving frequency resolution. This is one of the reasons it is important to have a fast FFT.

Optimizations that are only possible in native code is a good demonstration of how a developer can improve performance even more and to perhaps achieve better execution times than what is possible in Java. Having one single source file is valuable, especially for native libraries. This facilitates the process of adding and editing libraries.

Finally, comparing how performance can change depending on which data types that are used is also interesting when choosing a given implementation. Using the `float` data type, you use less memory at the cost of precision. A `double` occupies double the amount of space compared to a `float`, although it allows higher precision numbers. Caching is one aspect that could be utilized by reducing the space required for the results array.

3.1.1 Hardware

In this thesis, the hardware was delimited to one device to prevent the experiment from being too extensive and allow a more narrow examination of the test performed. The setup used for performing the experiments were the following:

Table 3.1: Hardware used in the experiments

Phone model	Google Nexus 6P
CPU model	Qualcomm MSM8994 Snapdragon 810
Core frequency	4x2.0 GHz and 4x1.55 GHz
Total RAM	3 GB
Available RAM	1.5 GB

3.1.2 Benchmark Environment

During the tests, both cellular and Wi-Fi were switched off. There were no applications running in the background while performing the tests during the experiments. Additionally, there were no foreground services running. This was to prevent any external influences from affecting the results. The software versions, compiler versions and compiler flags are presented in Table 3.2. The `-O3` optimization was used because it resulted in a small performance improvements compared with no optimization. The app was signed and packaged with release as build type. It was then transferred and installed on the device.

Table 3.2: Software used in the experiments

Android version	7.1.1
Kernel version	3.10.73g7196b0d
Clang/LLVM version	3.8.256229
Java version	1.8.0_76
Java compiler flags	`FLAGS HERE`
C++ compiler flags	`-Wall -std=c++14 -llog -lm -O3`

3.1.3 Time measurement

There are multiple methods of measuring time in Java. It is possible to measure the wall-clock time using the `System.currentTimeMillis()` method. There are drawbacks of using wall-clock time for measuring time. Because it is possible to manipulate the wall-clock at any time, it could result in too small or too large runtime depending on seemingly random factors. A more preferable method is to measure elapsed CPU time. This does not depend on a changeable wall-clock but rather use hardware to measure time. It is possible to use both `System.nanoTime()` and `SystemClock.elapsedRealtimeNanos()` for this purpose and the latter was used for the tests covered in this thesis.

What is being measured is the time to execute the tests assuming we have the desired input data and will get the required output data where we do not convert the data. Different algorithms accepts different data types as input parameters. When using an algorithm, the easiest solution would be to design your application around the algorithm, its input parameters and its return type. When possible to calculate external dependencies such as lookup tables, this is done outside the timer as it is only done once and not

```
// Prepare formatted input
double[] z = combineComplex(re, im);

// Start timer
long start = SystemClock.elapsedRealtimeNanos();

// Native call
double[] nativeResult = fft_princeton_recursive(z);

// Stop timer
long stop = SystemClock.elapsedRealtimeNanos() - start;
```

Figure 3.1: Timer placements for tests

for each call to the FFT.

Some algorithms require a `Complex[]`, some require a `double[]` where the first half contains the real numbers and the second half contain the imaginary numbers and some require two double arrays, one for the real numbers and one for imaginary. Because of these different requirements, the timer encapsulates a function shown in Figure 3.1. The timer would not measure the conversion from the shared input to the input type required by the particular algorithm.

3.1.4 Memory measurement

TODO: *The profiling tool provided by Android Studio was used to measure the amount of memory that each test required. The method used to measure the memory was to attach the debugger to the app and measure using the profiler. To measure each test separately and equally, the app was launched freshly between tests and the garbage collector was forced before each test. After this, the memory allocation tracker was activated and then followed by starting a test. When the test had been done executing, the tracker was stopped and the results saved. The memory measure tests were executed at a different time than the execution time tests*

3.2 Evaluation

The unit of the resulting data was chosen to be in milliseconds. To be able to have 100 executions run in reasonable time, the maximum size of the input data was limited to $2^{18} = 262144$ for all the tests. The sampling rate is what determines the highest frequency that could be found in the result. The frequency range perceivable by the human ear ($\sim$ 20-22,000 Hz) is covered by the tests. According to the Nyquist theorem, the sampling rate must be at least twice the upper limit (44,000). Because the FFT is limited to sample sizes of powers of 2, the next power of 2 for a sampling rate of 44,000 is 2^{16}. This size was chosen as the upper limit for the library comparisons.

For the SIMD tests, even larger sizes were used. This was to demonstrate how the execution time grew when comparing Java with low level optimizations in C++. Here, sizes up to 2^{18} were used because the steps from $2^{16} - 2^{18}$ illustrated this point clearly. It is also with these sizes the garbage collection gets invoked many times due to large allocations.

3.2.1 Data representation

The block sizes chosen in the JNI and libraries tests are limited to every power of two from 2^4 to 2^{16}. For NEON tests, $2^{16} - 2^{18}$ will be used for the tests. One reason this interval was chosen was because it is relevant to have the largest data the largest number of blocks needed for sample rates of 44100 Hz. To get a resolution of at least one Hz for a frequency span of 0-22050 Hz, an FFT size of 2^{16} (next power of two for 44,100) is required. The lowest sample size was chosen to be 2^4 to get a variety of data points to test for to find the increase in execution time for larger data sizes.

Every test result were not presented in Chapter 4 - Results. In this chapter, only the results that were relevant to discuss about are included. The tests results not found in the results chapter is found in Appendix B. To visualize a result, tables and line graphs were used. FFT sizes were split into groups labeled *small* size ($2^4 - 2^7$) *medium* size ($2^8 - 2^{12}$), *large* size ($2^{13} - 2^{16}$) and *extra large* size ($2^{17} - 2^{18}$). This decision was made to allow the discussion to be divided into groups to see where the difference in performance between the algorithms is significant. An accelerometer samples at low frequencies, commonly at the ones grouped as *small*.

For the normal FFT tests, the data type `double` was used and when presenting the results for the optimization tests, `float` was used. This was to ensure that we could discuss the differences in efficiency for choosing a specific data type.

3.2.2 Sources of error

There are multiple factors that can skew the results when running the tests. Some are controllable and some are not. In these tests, allocation of objects were minimized as much as possible to prevent the overhead of allocating dynamic memory. Because the Java garbage collector is uncontrollable during runtime, this will depend on the sizes of the objects and other aspects dependent on a specific implementation. JNI allows native code to be run without interruption by the garbage collector by using the `GetPrimitiveArrayCritical` function call. Additionally, implementation details of the Java libraries were not altered to ensure that the exact library found was used.

3.2.3 Statistical significance

Because the execution times differ between runs, it is important to calculate the sample mean. This way we have an expected value to use in our results. To get an accurate sample mean, we must have a large sample size. This is the number of runs we choose to execute each test. The following formula calculates the sample mean [32, p.263]:

$$\bar{X} = \frac{1}{N}\sum_{k=1}^{N} X_k$$

We cannot say anything about how close to our mean the samples are with only the sample mean. Therefore, the standard deviation is needed to find the dispersion of the data for each test. The standard deviation for a set of random samples $X_1, \ldots, X_N$ is calculated using the following formula [32, p. 302]:

$$s = \sqrt{\frac{1}{N-1}\sum_{k=1}^{N}\left(X_k - \bar{X}\right)^2}$$

When comparing results, we need to find a confidence interval for a given test and choose a confidence level. For the data gathered in this study, a 95% two-sided confidence level was chosen when comparing the data. To find the confidence interval we must first find the standard error of the mean using the following formula [32, p. 304]:

$$SE_{\bar{X}} = \frac{s}{\sqrt{N}}$$

To find the confidence interval, we must calculate the margin of error by taking the appropriate z^*-value for a confidence level and multiplying it with the standard error. For a confidence level of 95%, we get a margin of error as follows:

$$ME_{\bar{X}} = SE_{\bar{X}} \cdot 1.96$$

Our confidence interval will then be:

$$\bar{X} \pm ME_{\bar{X}}$$

3.3 JNI Tests

For testing the JNI overhead, four different tests were constructed. The first test had no parameters, returned void and did no calculations. The purpose of this test was to see how long it would take to call the smallest function possible. The function shown in Figure 3.2 was used to test this.

```
void jniEmpty(JNIEnv*, jobject) {
    return;
}
```

Figure 3.2: JNI test function with no parameters and no return value

For the second test, a function was written (see Figure 3.3) that took a jdoubleArray as input and returned the same data type. The reason this test was made was to see if JNI introduced some extra overhead for passing an argument and having a return value.

```
jdoubleArray jniParams(JNIEnv*, jobject, jdoubleArray arr) {
    return arr;
}
```

Figure 3.3: JNI test function with a double array as input parameter and return value

```
jdoubleArray jniVectorConversion(JNIEnv* env, jobject, jdoubleArray arr) {
    jdouble* elements = (jdouble*)(*env).GetPrimitiveArrayCritical(arr, 0);
    (*env).ReleasePrimitiveArrayCritical(arr, elements, 0);
    return arr;
}
```

Figure 3.4: Get and release elements

```
jdoubleArray jniColumbia(JNIEnv* env,
                         jobject obj,
                         jdoubleArray arr,
                         jdoubleArray cos,
                         jdoubleArray sin) {
    jdouble* elements = (jdouble*)(*env).GetPrimitiveArrayCritical(arr, 0);
    jdouble* sin_v    = (jdouble*)(*env).GetPrimitiveArrayCritical(sin, 0);
    jdouble* cos_v    = (jdouble*)(*env).GetPrimitiveArrayCritical(cos, 0);
    (*env).ReleasePrimitiveArrayCritical(arr, elements, 0);
    (*env).ReleasePrimitiveArrayCritical(sin, sin_v, 0);
    (*env).ReleasePrimitiveArrayCritical(cos, cos_v, 0);
    return arr;
}
```

Figure 3.5: JNI overhead for Columbia FFT

In Figure 3.4, the third test started by calling the `GetPrimitiveArrayCritical` function to be able to access the elements stored in `arr`. When all the calculations are done, the function will return `arr`. To overwrite the changes made on `elements`, a function called `ReleasePrimitiveArrayCritical` has to be called.

The fourth and final test evaluated the performance of passing three arrays through JNI as well as the cost of getting and releasing the arrays. This test was included because the Columbia algorithm requires the precomputed trigonometric tables. This test is presented in Figure 3.5.

3.4 Fast Fourier Transform Algorithms

Different implementations of FFT were used in the libraries tests. Three of them were implemented in Java and one in C. The implementations chosen were all contained in one file. The following algorithms were used to compare and find a good estimate on the

performance of FFT implementations with varying complexity:

- Princeton Recursive [33]
- Princeton Iterative [34]
- Columbia Iterative [35]
- Kiss (*Keep It Simple, Stupid*) FFT [36]

3.4.1 Java Libraries

The Princeton Recursive FFT is a straightforward implementation of the FFT with no radical optimizations. It was implemented in Java by Robert Sedgewick and Kevin Wayne [33]. Twiddle factors are trigonometric constants used during the butterfly operations. They are not precomputed in this algorithm, leading to duplicate work when calling it multiple times.

Princeton Iterative, also written by Robert Sedgewick and Kevin Wayne [34], is an iterative version of the previous FFT (also written in Java). Bit reversal and butterfly operations are used to produce a faster algorithm.

Columbia Iterative [35] uses pre-computed trigonometric tables that are prepared in the class constructor. Because you commonly call FFT for the same sizes in your program, it is beneficial to have the trigonometric tables saved and use them in subsequent calls to the FFT.

3.4.2 C++ Libraries

Conversion to C++ was done manually for Princeton Iterative, Princeton Recursive and Columbia Iterative. Some changes were necessary to follow the C++ syntax. The Complex class used in Java was replaced by `std::complex` in all converted programs. Java dynamic arrays were replaced by `std::vector` for when they were created. This only occurred in the Princeton Recursive algorithm. In Princeton Iterative and Columbia Iterative, a Java array reference was sent to the function and there were no arrays created in the function. In C++, a pointer and a variable containing its size was used instead.

Kiss FFT is a small library that consists of one source file. It is available under the BSD

license. To use it, you first call the `kiss_fft_alloc` function which allocates memory for the twiddle factors as well as calculates them. This function returns a struct object that is used as a config. The FFT is executed when the `kiss_fft` function is called. The first parameter for this function is the config returned by the init function, followed by a pointer to the time domain input and a pointer to where the frequency output will be placed.

3.5 NEON Optimization

Two libraries were chosen to test how vectorization of the loops can improve performance. Both libraries were written in Intel SSE intrinsics and were converted to ARM NEON intrinsics. `float` was used so that the vector registers could hold 4 elements. It is possible to have the register hold two double precision variables although this would increase the number of instructions needed to calculate the FFT. For memory locality, this is also inefficient.

The first FFT algorithm was a recursive implementation written by Anthony Blake [37]. This algorithm has an initializer function that allocates space for the twiddle factors and calculates them. They are placed in a two dimensional array that utilizes memory locality to waste less memory bandwidth [38]. The converted program is listed in Appendix A.2. The second algorithm was an iterative implementation. This library is a straightforward implementation of FFT with SSE [39] and was written for a sound source localization system [40]. The code that was converted from SSE to NEON is presented in Appendix A.3.

CHAPTER 4

Results

Results from the JNI tests, FFT libraries and NEON optimizations are presented here.

4.1 JNI

The results from the tests that measure the JNI overhead can be found in Table 4.1. These tests are presented with block sizes defined in chapter 3 - Method. Execution time and confidence intervals are given in microseconds and are rounded to four decimal points. The number before the ± sign is the sample mean and the number after ± is a two sided confidence interval with a confidence level of 95%. Each test was executed 100 times to ensure that we get reliable sample means.

The test labeled **No params** is the test where a native void function with no parameters that returns immediately was called. **Vector** takes a `jdoubleArray` and returns a `jdoubleArray` immediately. **Convert** takes a `jdoubleArray`, converts it to a native array using `GetPrimitiveArrayCritical()`, converts it back to a `jdoubleArray` using `ReleasePrimitiveArrayCritical()` and returns a `jdoubleArray`. **Columbia** takes three `jdoubleArray`s, converts them the same way and returns the same way.

No surprising data regarding the first two tests were found. Neither the **No params** nor **Vector** tests had a clear increase in execution time for an increase in block size. **Vector** did have a higher mean for block size **65536**. On the other hand, we can see that the 95% confidence interval is very large (±3.1960 µs). This is likely due to its high standard deviation of 16.3058 found in Appendix B Table B.8. Likewise, there is a spike in execution time mean for a block size of **1024** in the **Convert** test.

Table 4.1: Results from the JNI tests, Time (µs)

Block size	No params	Vector	Convert	Columbia
16	1.4510 ± 0.0664	1.5287 ± 0.0116	1.7662 ± 0.0655	2.6886 ± 0.0670
32	1.4303 ± 0.0096	1.5296 ± 0.0302	1.7688 ± 0.0214	2.7277 ± 0.1705
64	1.4307 ± 0.0092	1.5208 ± 0.0118	1.7427 ± 0.0155	2.9297 ± 0.5374
128	1.4541 ± 0.0161	1.5489 ± 0.0221	1.7880 ± 0.0455	2.7176 ± 0.0739
256	1.4349 ± 0.0084	1.5052 ± 0.0116	1.8713 ± 0.1376	2.7254 ± 0.0529
512	1.5359 ± 0.1423	1.5333 ± 0.0082	1.7869 ± 0.0149	2.7130 ± 0.0214
1024	1.4427 ± 0.0151	1.6620 ± 0.1152	1.7818 ± 0.0449	2.7870 ± 0.0394
2048	1.4416 ± 0.0153	2.0567 ± 0.9071	1.7636 ± 0.0347	2.7469 ± 0.0304
4096	1.4375 ± 0.0086	1.6318 ± 0.0290	1.9333 ± 0.1082	2.5359 ± 0.0394
8192	1.4287 ± 0.0082	2.5256 ± 2.0384	2.7318 ± 1.3338	2.5776 ± 0.0408
16384	1.4374 ± 0.0074	1.3823 ± 0.0347	2.2114 ± 0.2242	2.6386 ± 0.1345
32768	1.5422 ± 0.0531	1.2437 ± 0.0374	2.2224 ± 0.1211	2.8062 ± 0.0982
65536	1.5556 ± 0.0517	1.4547 ± 0.1378	2.6542 ± 0.3889	3.5229 ± 0.7291
131072	1.5282 ± 0.0116	1.4870 ± 0.0684	3.5047 ± 0.4238	3.8781 ± 0.3591
262144	1.5438 ± 0.0125	2.2171 ± 0.1580	5.8172 ± 0.9490	5.8157 ± 0.6435

4.2 FFT Libraries

The results from the FFT Libraries are presented in line graphs, both language specific and graphs with both Java and C++ are given to illustrate the differences between languages and also provide differences for specific languages clearly. The time unit for these tests are presented in milliseconds. This was because the FFT ran in ranges below one millisecond and above one second among different algorithms and different block sizes. The means were calculated from the results of 100 test runs. In each C++ line graph, the fastest Java test was added to make it easier to get a reference to compare the languages.

4.2.1 Small block sizes

Results from the small blocks tests shows a clear difference between the different algorithms. In Figure 4.1, Princeton Recursive in Java perform the worst. Princeton Recursive in C++ and Princeton Iterative in Java perform better than Princeton Re-

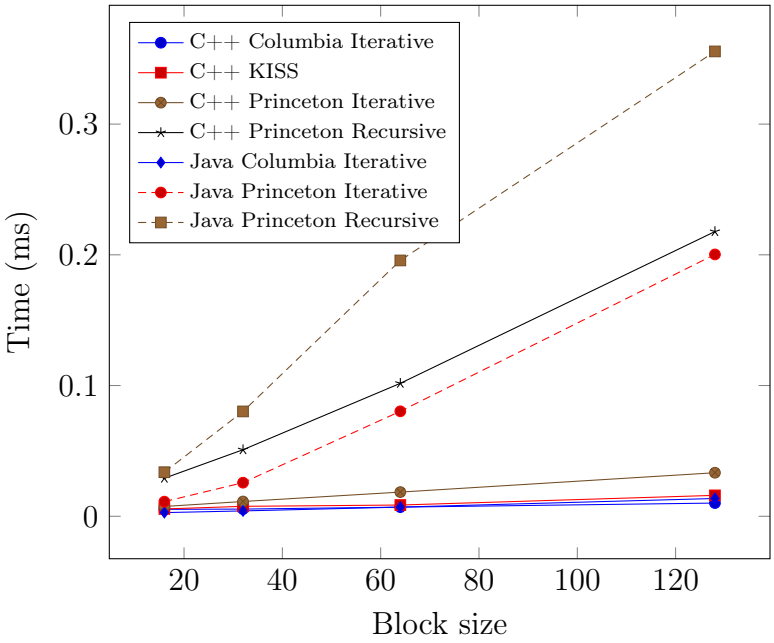


Figure 4.1: Line graph for all algorithms, *small* block sizes

Table 4.2: Java results table for *small* block sizes, Time (ms)

Block size	Columbia Iterative	Princeton Iterative	Princeton Recursive
16	0.0028 ± 0.0001	0.0112 ± 0.0004	0.0337 ± 0.0010
32	0.0041 ± 0.0001	0.0257 ± 0.0010	0.0802 ± 0.0031
64	0.0070 ± 0.0002	0.0803 ± 0.0041	0.1957 ± 0.0043
128	0.0136 ± 0.0008	0.2003 ± 0.0147	0.3556 ± 0.0123

cursive Java although worse than the rest of the algorithms. The rest of the algorithms perform similarly and does not seem to grow given the small block sizes.

As we can see in Figure 4.2, the standard deviation of Princeton Recursive and Princeton Iterative are very large. This means that the samples were sparse and as a result of this, not a reliable mean. We can see in Table 4.2 that the confidence interval is generally larger for Princeton Iterative and Princeton Recursive compared to Columbia Iterative.

As for the C++ tests, the results were less scattered and had a more apparent increase in time with increasing block sizes. We can also see that the slowest algorithm, Princeton Recursive, has the largest standard deviation. It is clear by looking at Table 4.3 that KISS performs the best, followed by Columbia Iterative, Princeton Iterative and then Princeton Recursive. If we look at the Java implementation, it has a general decrease

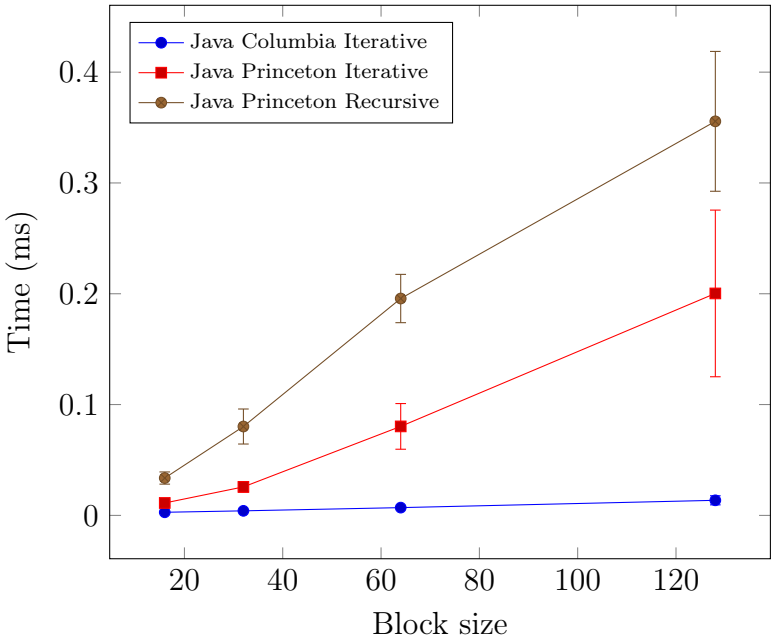


Figure 4.2: Java line graph for *small* block sizes with standard deviation error bars

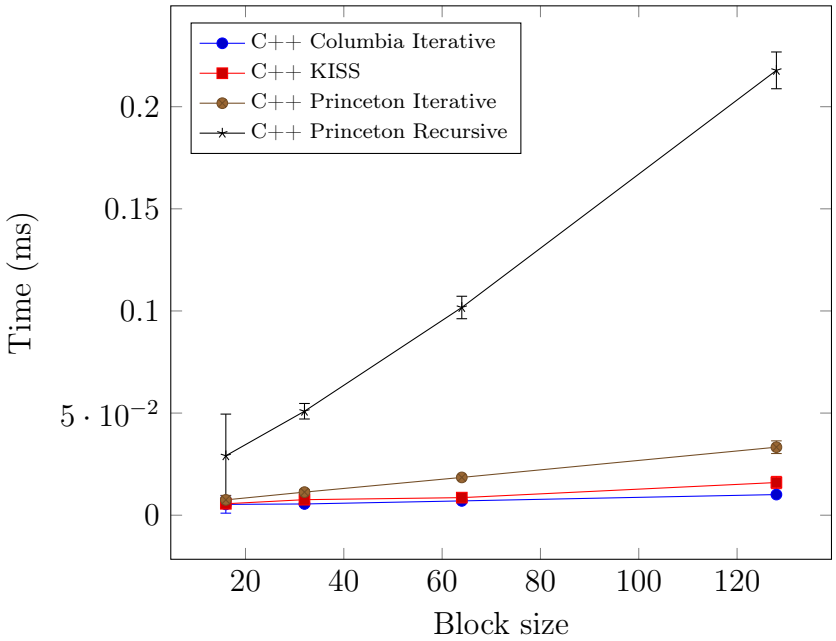


Figure 4.3: C++ line graph for *small* block sizes with standard deviation error bars

in execution time for larger block sizes although it is faster than Princeton Iterative and Recursive.

Table 4.3: C++ results table for *small* block sizes, Time (ms)

Block size	Columbia Iterative	KISS	Princeton Iterative	Princeton Recursive
16	0.0053 ± 0.0008	0.0056 ± 0.0002	0.0075 ± 0.0004	0.0291 ± 0.0039
32	0.0055 ± 0.0002	0.0076 ± 0.0004	0.0113 ± 0.0002	0.0509 ± 0.0008
64	0.0070 ± 0.0002	0.0086 ± 0.0002	0.0185 ± 0.0002	0.1017 ± 0.0010
128	0.0101 ± 0.0001	0.0160 ± 0.0006	0.0333 ± 0.0006	0.2178 ± 0.0018

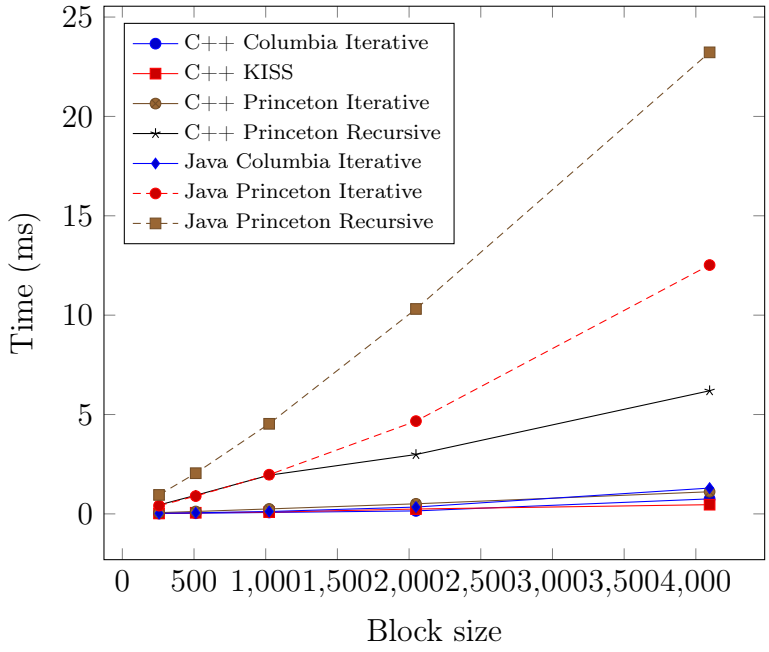


Figure 4.4: Line graph for all algorithms, *medium* block sizes

4.2.2 Medium block sizes

The medium block sizes continues the trend where Java Princeton Recursive performs the worst followed by Java Princeton Iterative and C++ Princeton Recursive. As for the small block sizes, Java Columbia Iterative, C++ Princeton Iterative, C++ Columbia Iterative and KISS have the smallest execution time and perform similarly.

The results found in Figure 4.5 are somewhat different than for the small block sizes. We can still see that Java Princeton Recursive diverges from the other algorithms. What is interesting is that it is now clearer which of Princeton Iterative and Columbia Iterative is the fastest. Columbia Iterative is clearly faster than Princeton Iterative as shown by the confidence intervals given in Table 4.4. The standard deviations for the samples in the Princeton Recursive and Princeton Iterative are still relatively large compared to

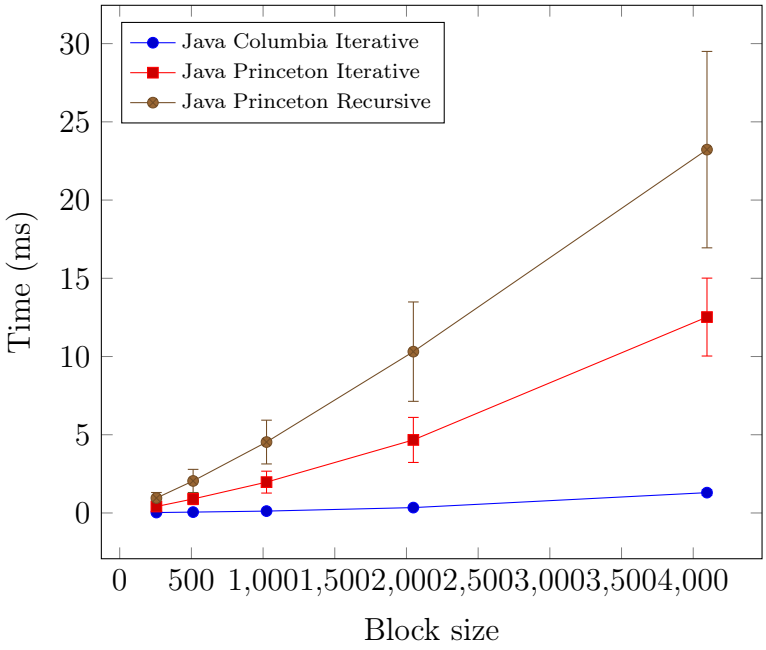


Figure 4.5: Java line graph for *medium* block sizes with standard deviation error bars

Table 4.4: Java results table for *medium* block sizes, Time (ms)

Block size	Columbia Iterative	Princeton Iterative	Princeton Recursive
256	0.0266 ± 0.0008	0.4139 ± 0.0206	0.9540 ± 0.0694
512	0.0557 ± 0.0010	0.8952 ± 0.0708	2.0471 ± 0.1450
1024	0.1197 ± 0.0012	1.9717 ± 0.1370	4.5326 ± 0.2742
2048	0.3434 ± 0.0053	4.6686 ± 0.2820	10.3113 ± 0.6225
4096	1.2992 ± 0.0335	12.5215 ± 0.4880	23.2240 ± 1.2303

Columbia Iterative.

For the C++ algorithms, it is now clearer to see in which order the algorithms rank regarding performance in Figure 4.6. Princeton Recursive perform the worst while the rest has similar execution times. It is now clear that KISS performs best, followed by Columbia Iterative and then Princeton Iterative. The Java Columbia Iterative implementation proves to be faster than Princeton for block sizes smaller than 4096.

In Table 4.5 we can see that the confidence intervals are relatively small, meaning these results have higher precision than for the same test with smaller block sizes. We can also see that there is no overlap between any confidence intervals thereby giving us a strong

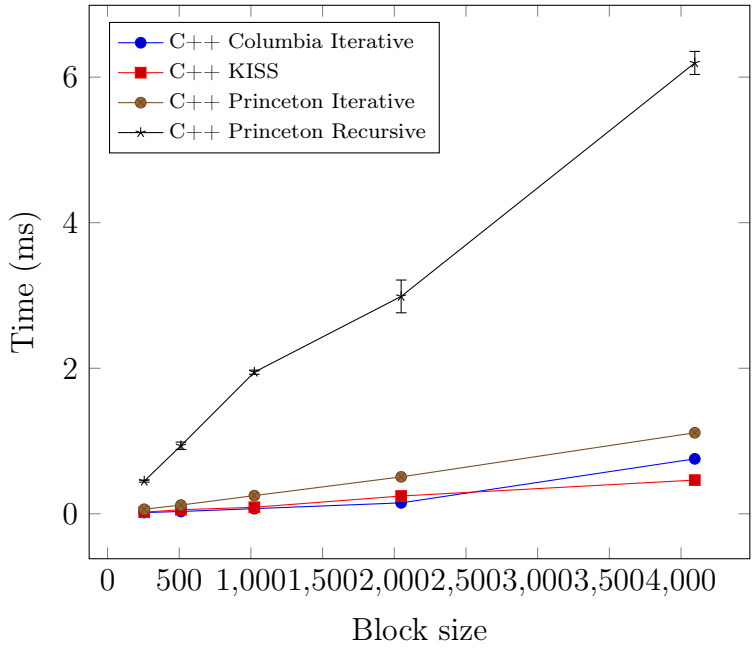


Figure 4.6: C++ line graph for *medium* block sizes with standard deviation error bars

Table 4.5: C++ results table for *medium* block sizes, Time (ms)

Block size	Columbia Iterative	KISS	Princeton Iterative	Princeton Recursive
256	0.0169 ± 0.0001	0.0225 ± 0.0002	0.0628 ± 0.0016	0.4492 ± 0.0027
512	0.0319 ± 0.0006	0.0559 ± 0.0010	0.1206 ± 0.0010	0.9353 ± 0.0100
1024	0.0709 ± 0.0010	0.0898 ± 0.0024	0.2500 ± 0.0016	1.9461 ± 0.0057
2048	0.1503 ± 0.0010	0.2446 ± 0.0025	0.5075 ± 0.0029	2.9871 ± 0.0441
4096	0.7553 ± 0.0033	0.4632 ± 0.0078	1.1136 ± 0.0043	6.1952 ± 0.0310

indication that the order in performance given in the previous paragraph is true.

4.2.3 Large block sizes

Figure 4.7 shows the growth in execution time for increasing block sizes of type *large*. It continues the trend set by the tests with a block size of *medium*. It is easy to see which algorithms that perform worse than the others. As previous tests shows, Java Princeton Recursive, Java Princeton Iterative and C++ Princeton Recursive are still the slowest.

The results for the Java tests with *large* block sizes are presented in Figure 4.8. The results from *large* verifies the order in which the algorithms performs. The order in performance

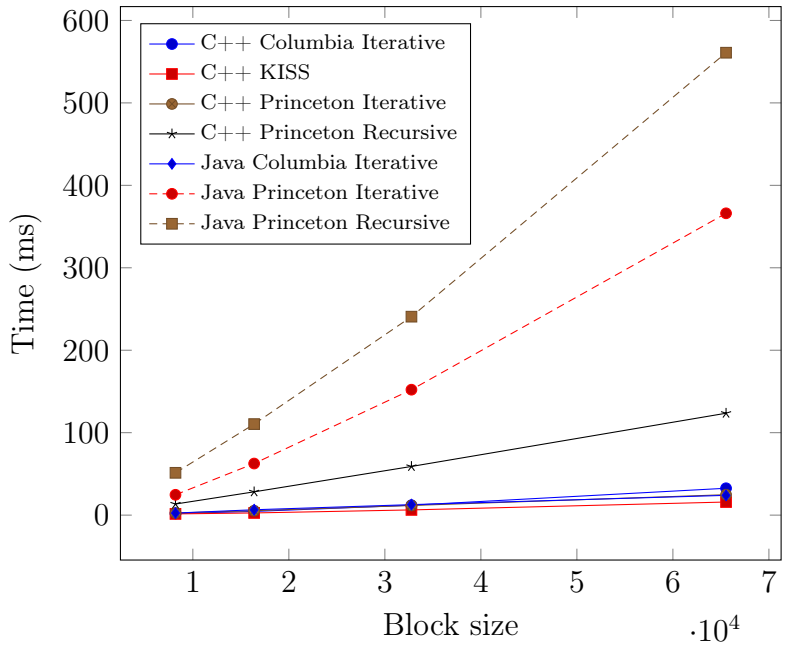


Figure 4.7: Line graph for all algorithms, *large* block sizes

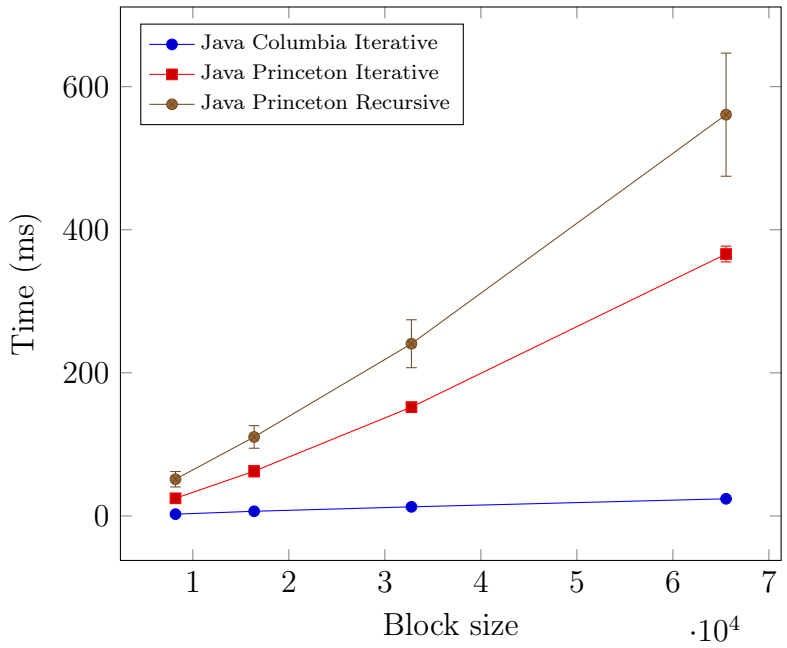


Figure 4.8: Java line graph for *large* block sizes with standard deviation error bars

for Java is still Columbia Iterative, Princeton Iterative and Princeton Recursive (where Columbia Iterative is the fastest).

Table 4.6: Java results table for *large* block sizes, Time (ms)

Block size	Columbia Iterative	Princeton Iterative	Princeton Recursive
8192	2.5070 ± 0.0443	24.6632 ± 1.2524	51.3207 ± 2.1101
16384	6.5204 ± 0.0316	62.5681 ± 1.5639	110.4185 ± 3.0921
32768	12.7414 ± 0.3812	152.1322 ± 1.3281	240.7083 ± 6.5646
65536	23.9784 ± 0.0788	366.0864 ± 2.1605	560.8628 ± 16.8682

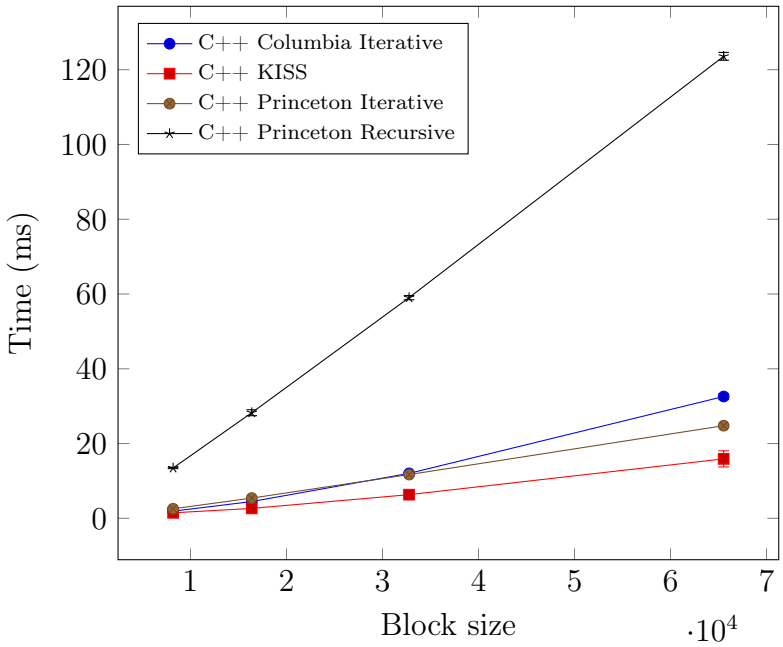


Figure 4.9: C++ line graph for *large* block sizes with standard deviation error bars

Results for tests with *large* block sizes in C++ produced some interesting results. In Figure 4.9 Princeton Recursive has a much larger standard deviation than the other algorithms. It is also the slowest (see Table 4.7). We can also see in the same table that all algorithms are faster in C++ than their equivalent in Java and that there are no overlapping confidence intervals for a given algorithm or block size. Another thing to note is that the Columbia Iterative algorithm performs almost the same between Java and C++. This is not the case for Princeton Iterative or Recursive.

Table 4.7: C++ results table for *large* block sizes, Time (ms)

Block size	Columbia Iterative	KISS	Princeton Iterative	Princeton Recursive
8192	1.9094 ± 0.0094	1.4667 ± 0.0410	2.5455 ± 0.0561	13.4611 ± 0.0349
16384	4.4813 ± 0.0318	2.6454 ± 0.1143	5.4095 ± 0.0476	28.2437 ± 0.1556
32768	12.0190 ± 0.0472	6.3225 ± 0.0429	11.6946 ± 0.0331	59.0249 ± 0.1113
65536	32.5804 ± 0.1615	15.8927 ± 0.4234	24.7761 ± 0.0719	123.5847 ± 0.2058

Table 4.8: NEON `float` results table for *extra large* block sizes, Time (ms)

Block size	Iterative	Recursive
8192	1.3282 ± 0.0512	2.0665 ± 0.0098
16384	2.1058 ± 0.0563	3.4816 ± 0.1082
32768	3.9077 ± 0.0365	6.7110 ± 0.0468
65536	11.5019 ± 0.1031	16.7593 ± 0.5551
131072	31.5155 ± 0.7944	38.4122 ± 1.2140
262144	70.6807 ± 1.1096	79.6650 ± 2.4153

4.3 Optimizations

The NEON optimizations proved to be very efficient for *extra large* block sizes. These results can be found in Table 4.8. Comparing these figures with the results, for the same block sizes in Java (Table 4.9), we can see that the results from the NEON tests are more than double the speed of the fastest Java implementation. Note that the data type for all of the optimization tests (as well as the C++ and Java tests) were `float`. Because we get faster execution time by lining up more elements, the `float` data type was used for the NEON intrinsics. When comparing this with non-NEON tests, `float`s were used in Java and C++ to make the results more comparable.

Table 4.9 also show that for very large block sizes, Princeton Iterative and Princeton Recursive are very inefficient compared to Columbia Iterative. This also holds true for the C++ implementation. For these block sizes, however, the differences in execution time is smaller than for Java (as seen in Table 4.11).

When comparing the NEON results from Table 4.8 with the results from the C++ tests found in Table 4.11, it is clear that vectorization as an optimization is beneficial. NEON intrinsics resulted in almost twice as fast execution time in comparison with the normal C++ tests. It was also much faster than Java, especially compared to Java Princeton Iterative/Recursive.

Table 4.9: Java `float` results table for *extra large* block sizes, Time (ms)

Block size	Columbia Iterative	Princeton Iterative	Princeton Recursive
8192	2.1624 ± 0.0053	32.7020 ± 1.0504	57.2447 ± 0.7748
16384	5.4469 ± 0.0435	75.3437 ± 2.2330	121.2444 ± 1.1907
32768	11.9213 ± 0.0994	174.8462 ± 4.9882	265.7005 ± 2.1829
65536	26.5285 ± 0.1713	404.4631 ± 11.1087	565.5092 ± 4.2597
131072	62.9644 ± 0.2256	898.7526 ± 19.9597	1218.9377 ± 7.5046
262144	195.5186 ± 0.5860	1993.8636 ± 35.9358	2536.9244 ± 5.0801

Table 4.10: Java `double` results table for *extra large* block sizes, Time (ms)

Block size	Columbia Iterative	Princeton Iterative	Princeton Recursive
8192	2.5070 ± 0.0443	24.6632 ± 1.2524	51.3207 ± 2.1101
16384	6.5204 ± 0.0316	62.5681 ± 1.5639	110.4185 ± 3.0921
32768	12.7414 ± 0.3812	152.1322 ± 1.3281	240.7083 ± 6.5646
65536	23.9784 ± 0.0788	366.0864 ± 2.1605	560.8628 ± 16.8682
131072	95.1627 ± 1.7969	791.0354 ± 3.1033	1254.8310 ± 28.6315
262144	290.7303 ± 4.0778	1875.8632 ± 23.0398	2919.3773 ± 24.0825

In Figure 4.10 both the Iterative NEON and the Recursive NEON seems to have the same growth in execution time with increasing block size. Of these two algorithms, Iterative is the fastest. If we compare the results found for Java in Table 4.9 with the C++ results in Table 4.11 there is a big difference between the Princeton algorithms. The execution times are much larger in Java than in C++. The Java Columbia Iterative is faster than C++ Princeton Iterative for block sizes of 65536.

As we can see in Table 4.10, the results show that when using `double`s, the performance will be worse for Columbia Iterative and Princeton Recursive. Interestingly, the Princeton Iterative was faster when using `double`s than with `float`s. This characteristic can not be found in the C++ `float` vs `double` tests. We can see that the `float` tests were always running faster than the corresponding `double` test.

Table 4.11: C++ `float` results table for *extra large* block sizes, Time (ms)

Block size	Columbia Iterative	Princeton Iterative	Princeton Recursive
8192	1.5140 ± 0.0178	2.7634 ± 0.0188	10.5867 ± 0.0666
16384	3.8153 ± 0.0592	6.2555 ± 0.0327	22.6880 ± 0.0529
32768	9.2565 ± 0.3210	13.1659 ± 0.0670	46.3923 ± 0.1366
65536	25.0067 ± 0.5223	27.6772 ± 0.1523	101.9946 ± 2.5992
131072	57.1731 ± 2.2834	51.3645 ± 1.7228	207.7519 ± 3.9006
262144	211.9326 ± 3.1952	152.0646 ± 3.0811	436.9925 ± 5.0080

Table 4.12: C++ `double` results table for *extra large* block sizes, Time (ms)

Block size	Columbia Iterative	KISS	Princeton Iterative	Princeton Recursive
8192	1.9094 ± 0.0094	1.4667 ± 0.0410	2.5455 ± 0.0561	13.4611 ± 0.0349
16384	4.4813 ± 0.0318	2.6454 ± 0.1143	5.4095 ± 0.0476	28.2437 ± 0.1556
32768	12.0190 ± 0.0472	6.3225 ± 0.0429	11.6946 ± 0.0331	59.0249 ± 0.1113
65536	32.5804 ± 0.1615	15.8927 ± 0.4234	24.7761 ± 0.0719	123.5847 ± 0.2058
131072	131.0146 ± 0.4794	39.9681 ± 0.7991	72.5146 ± 0.4541	257.7359 ± 0.7842
262144	303.4669 ± 3.8967	92.1686 ± 1.8916	229.1227 ± 4.1027	551.2218 ± 1.8673

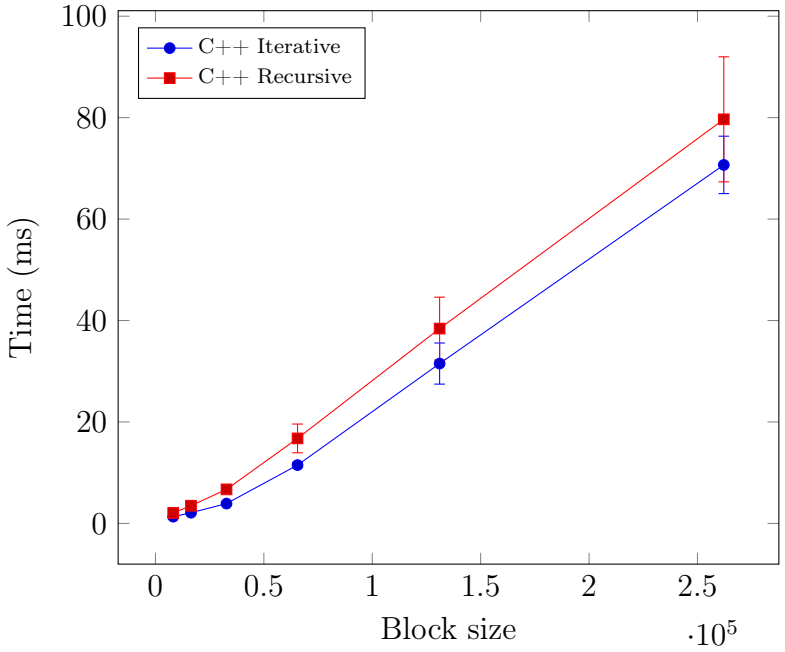


Figure 4.10: NEON results table for *extra large* block sizes, Time (ms)

CHAPTER 5

Discussion

The discussion chapter covers how the JNI affects performance, how efficient smaller FFT libraries are, why the optimization gave the results found in Chapter 4 and the efficiency difference between floats and doubles.

5.1 JNI Overhead

The test results from the JNI tests showed that the overhead is small relative to the computation of the FFT algorithm. As long as it is being run once per calculation, it will not affect the performance significantly. If the JNI is called in a loop when it might not be necessary, the overhead add up and becomes a larger part of the total execution time. Another thing to note is that the execution time stay within about 10 µs. They are also not growing with larger input.

The confidence intervals overlap for many of the values, meaning we cannot say whether or not one input yields a faster execution time than the other. Some larger block sizes has lower execution time than smaller block sizes and some grow for larger input. Because of this, it is reasonable to assume that nothing is done to the arrays when they are passed to the JNI, only pointers are copied. The `GetPrimitiveArrayCritical()` and `ReleasePrimitiveArrayCritical()` seem to introduce overhead when used on larger arrays.

Regarding the sudden spike in mean in the JNI tests, this can be a cause of one large execution time, skewing the results. This is actually the case for the **Convert** tests with block size 1024 as seen in figure B.1 found in Appendix B. A reason for this could be

that the garbage collector began executing during the timing of the test.

5.2 Simplicity and Efficiency

The slowest algorithm was the Java Princeton Recursive. The reason for this was that it does a lot of method calls. Additionally, each call creates new `Complex` arrays when splitting up the array in odd and even indices. Lastly, when combining the arrays, new `Complex` objects are created each time an operation is done between two complex numbers. The `Complex` class creates immutable objects.

The C++ version of the Princeton Recursive algorithm is faster than the Java version. One big difference is that the `std::complex` type used in C++ does not create new instances each time is is being operated with. This lowers the number of calls requesting more memory from the heap. Depending on the situation for the program, there is a risk that the program must ask the system for more memory, slowing down the allocation process.

Additionally, this will increase the work for the garbage collector, increasing the risk of it being triggered during the timing. To prevent the number of allocations you are doing inside a repeated process is to reuse allocated memory. This is done by pre-allocating the necessary arrays or other data structures and overwrite the results for each call. Avoiding calls to new in a method that is called multiple times can increase time and memory efficiency.

Which is fastest and why??

Which tests triggered the GC??

Precision (smaller intervals) for larger block sizes C++.
Because larger execution times result in more consistent execution times??

One thing that is clear is that the Columbia Iterative algorithm is the best one to choose from of the Java versions. It performs better than both Princeton Iterative and Princeton Recursive. It also allows simple modifications such as changing between using `floats` or `doubles` to represent the data.

The reason Princeton Iterative and Princeton Recursive is slower in Java than in C++ is because they operate with `Complex` elements. Each time two `Complex` numbers are

added, multiplied or subtracted, a new `Complex` object is created. This slows down the process and increases memory consumption, increasing work for the garbage collector.

Although you can have two equally long implementations,
one could be faster than the other.
It is important to do small tests with different sized data.

We can see in the tests that the best option is to choose KISS fft as it is the fastest of the options.

NEON iterative is faster than the recursive because...

For smaller Block sizes, the Java version is almost as fast as the C++ one.
It is easier to implement and more flexible to edit. No need for complicated JNI integration.

It is with very large block sizes the difference is clear.

C++ Tests got more clear growth because of GetPrimitiveArrayCritical?

Speed required when rendering to screen at a constant refresh rate

5.3 Vectorization as Optimization

Implementation effort

Harder to maintain

Pros/Cons with NEON (Vectorization)

As the results show, vectorization was an improvement in regards to efficiency. This was as expected because it is possible to process more elements for each instruction.

The benefit of fitting it all in cache.

Preparations by lining up data correctly.
Minimize RAM access.

When is the fastest required performance-wise??

Limitations of the NEON intrinsics. Only works for ARM

Compare percentage speedup for iterative compared with recursive.

Discuss differences in performance large java times, small java times WHY?

Accelerometer?

5.4 Floats and Doubles

Caching

Array would be twice as large if doubles were used.
Greater chance that the whole array can fit when using float elements.
There is a really big difference in both Java and C++.

Chapter 6

Conclusion

Describing text

When choosing a java version, choose Columbia Iterative (Conclusion)

Bibliography

[1] Sergey Chernenko, "Fast Fourier transform — FFT." http://www.librow.com/articles/article-10. [Accessed: 31 March 2017].

[2] International Data Corporation, "IDC: Smartphone OS Market Share 2016, 2015." http://www.idc.com/promo/smartphone-market-share/os. [Accessed: 2 February 2017].

[3] Android, "The Android Source Code." https://source.android.com/source/index.html. [Accessed: 1 February 2017].

[4] Android, "Why did we open the Android source code?." https://source.android.com/source/faqs.html. [Accessed: 2 February 2017].

[5] Google, "Android 5.0 Behavior Changes – Android Runtime (ART)." https://developer.android.com/about/versions/android-5.0-changes.html. [Accessed: 24 January 2017].

[6] Google, "android-5.0.0_r1 - platform/build - Git at Google." https://android.googlesource.com/platform/build/+/android-5.0.0_r2. [Accessed: 24 January 2017].

[7] C. M. Lin, J. H. Lin, C. R. Dow, and C. M. Wen, "Benchmark Dalvik and native code for Android system," *Proceedings - 2011 2nd International Conference on Innovations in Bio-Inspired Computing and Applications, IBICA 2011*, pp. 320–323, 2011.

[8] Google, "Android Interfaces and Architecture - Hardware Abstraction Layer (HAL)." https://source.android.com/devices/index.html. [Accessed: 30 January 2017].

[9] S. Komatineni and D. MacLean, *Pro Android 4*. Apressus Series, Apress, 2012.

[10] Google, "Platform Architecture." `https://developer.android.com/guide/platform/index.html`. [Accessed: 30 January 2017].

[11] I. Craig, *Virtual Machines*. Springer London, 2010.

[12] D. Bornstein, "Dalvik VM internals." *Google I/O*. 2008.

[13] Y. Shi, K. Casey, M. A. Ertl, and D. Gregg, "Virtual machine showdown: Stack versus registers," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 4, no. 4, p. 2, 2008.

[14] X. Li, *Advanced Design and Implementation of Virtual Machines*. CRC Press, 2016.

[15] Android, "ART and Dalvik." `http://source.android.com/devices/tech/dalvik/index.html`. [Accessed: 3 February 2017].

[16] L. Dresel, M. Protsenko, and T. Muller, "ARTIST: The Android Runtime Instrumentation Toolkit," *2016 11th International Conference on Availability, Reliability and Security (ARES)*, pp. 107–116, 2016.

[17] Android Developers, "Getting Started with the NDK." `https://developer.android.com/ndk/guides/index.html`. [Accessed: 6 February 2017].

[18] Android Developers, "CMake." `https://developer.android.com/ndk/guides/cmake.html#variables`. [Accessed: 6 February 2017].

[19] S. Liang, *The Java Native Interface: Programmer's Guide and Specification*. Addison-Wesley Java series, Addison-Wesley, 1999.

[20] UIUC, "Language Compatibility." `https://clang.llvm.org/compatibility.html`. [Accessed: 8 February 2017].

[21] Android Developers, "NDK Revision History." `https://developer.android.com/ndk/downloads/revision_history.html`. [Accessed: 6 February 2017].

[22] S. Muchnick, *Advanced Compiler Design Implementation*. Morgan Kaufmann Publishers, 1997.

[23] Piotr Luszczek, "Data-Level Parallelism in Vector, SIMD, and GPU Architectures."

http://www.icl.utk.edu/~luszczek/teaching/courses/fall2013/cosc530/cosc530_ch4all6up.pdf. University of Tennessee, [Accessed: 15 February 2017].

[24] Kernel.org, "How SIMD Operates." https://www.kernel.org/pub/linux/kernel/people/geoff/cell/ps3-linux-docs/CellProgrammingTutorial/BasicsOfSIMDProgramming.html. [Accessed: 14 February 2017].

[25] Bruno A. Olshausen, "Aliasing." http://redwood.berkeley.edu/bruno/npb261/aliasing.pdf. [Accessed: 9 February 2017].

[26] S. Smith, *The Scientist and Engineer's Guide to Digital Signal Processing*. California Technical Pub., 1997.

[27] L. Tan and J. Jiang, *Digital Signal Processing: Fundamentals and Applications*. Elsevier Science, 2013.

[28] B. J. W. Cooley and J. W. Tukey, "An Algorithm for the Machine Calculation Complex Fourier Series," pp. 297–301, 1964.

[29] A. D. D. C. Jr, M. Rosan, and M. Queiroz, "FFT benchmark on Android devices : Java versus JNI," pp. 4–7, 2013.

[30] S. Lee and J. W. Jeon, "Evaluating Performance of Android Platform Using Native C for Embedded Systems," *International Conference on Control, Automation and Systems*, pp. 1160–1163, 2010.

[31] X. Chen and Z. Zong, "Android App Energy Efficiency: The Impact of Language, Runtime, Compiler, and Implementation," *2016 IEEE International Conferences on Big Data and Cloud Computing (BDCloud), Social Computing and Networking (SocialCom), Sustainable Computing and Communications (SustainCom) (BDCloud-SocialCom-SustainCom)*, pp. 485–492, 2016.

[32] P. Olofsson and M. Andersson, *Probability, Statistics, and Stochastic Processes*. Wiley, 2012.

[33] Robert Sedgewick and Kevin Wayne, "FFT.java." http://introcs.cs.princeton.edu/java/97data/FFT.java.html. [Accessed: 9 March 2017].

[34] Robert Sedgewick and Kevin Wayne, "InplaceFFT.java." http://introcs.cs.princeton.edu/java/97data/InplaceFFT.java.html. [Accessed: 9 March 2017].

[35] Columbia University, "FFT.java." https://www.ee.columbia.edu/~ronw/code/MEAPsoft/doc/html/FFT_8java-source.html. [Accessed: 10 March 2017].

[36] Mark Borgerding, "Kiss FFT." https://sourceforge.net/projects/kissfft/. [Accessed: 10 March 2017].

[37] Anthony Blake, "Appendix 3 - FFTs with vectorized loops." http://cnx.org/contents/918459f2-a528-4fd1-a0ef-e48f4c5b6b5d@1. OpenStax CNX, [Accessed: 10 March 2017].

[38] Anthony Blake, "Implementation Details." http://cnx.org/contents/2b826002-1ba5-45da-a100-ffdfdbfc3159@4. OpenStax CNX, [Accessed: 10 March 2017].

[39] François Grondin, Jean-Marc Valin, Simon Brière, Dominic Létourneau, "ManyEars Microphone Array-Based Audition for Mobile Robots." https://github.com/introlab/manyears. [Accessed: 10 March 2017].

[40] François Grondin, Jean-Marc Valin, Simon Brière, Dominic Létourneau, "ManyEars Sound Source Localization, Tracking and Separation." http://introlab.github.io/manyears/. [Accessed: 10 March 2017].

[41] Robert Sedgewick and Kevin Wayne, "Complex.java." http://introcs.cs.princeton.edu/java/97data/Complex.java.html. [Accessed: 24 February 2017].

[42] François Grondin, Jean-Marc Valin, Simon Brière, Dominic Létourneau, "fft.c." https://github.com/introlab/manyears/blob/master/manyears-C/dsplib/Utilities/fft.c. [Accessed: 10 March 2017].

APPENDIX A

Source code

Listing A.1: Complex.java [41]

```
package com.example.algo.benchmarkapp.algorithms;

/*************************************************************************
 *  Compilation:  javac Complex.java
 *  Execution:    java Complex
 *
 *  Data type for complex numbers.
 *
 *  The data type is "immutable" so once you create and initialize
 *  a Complex object, you cannot change it. The "final" keyword
 *  when declaring re and im enforces this rule, making it a
 *  compile-time error to change the .re or .im instance variables after
 *  they've been initialized.
 *
 *  % java Complex
 *  a            = 5.0 + 6.0i
 *  b            = -3.0 + 4.0i
 *  Re(a)        = 5.0
 *  Im(a)        = 6.0
 *  b + a        = 2.0 + 10.0i
 *  a - b        = 8.0 + 2.0i
 *  a * b        = -39.0 + 2.0i
 *  b * a        = -39.0 + 2.0i
 *  a / b        = 0.36 - 1.52i
 *  (a / b) * b  = 5.0 + 6.0i
 *  conj(a)      = 5.0 - 6.0i
 *  |a|          = 7.810249675906654
 *  tan(a)       = -6.685231390246571E-6 + 1.0000103108981198i
 *
 *************************************************************************/

import java.util.Objects;

public class Complex {
    public double re;   // the real part
    public double im;   // the imaginary part

    // create a new object with the given real and imaginary parts
    public Complex(double real, double imag) {
```

```
        re = real;
        im = imag;
    }

    // return a string representation of the invoking Complex object
    public String toString() {
        if (im == 0) return re + "";
        if (re == 0) return im + "i";
        if (im <  0) return re + " - " + (-im) + "i";
        return re + " + " + im + "i";
    }

    // return abs/modulus/magnitude
    public double abs() {
        return Math.hypot(re, im);
    }

    // return angle/phase/argument, normalized to be between -pi and pi
    public double phase() {
        return Math.atan2(im, re);
    }

    // return a new Complex object whose value is (this + b)
    public Complex plus(Complex b) {
        Complex a = this;             // invoking object
        double real = a.re + b.re;
        double imag = a.im + b.im;
        return new Complex(real, imag);
    }

    // return a new Complex object whose value is (this - b)
    public Complex minus(Complex b) {
        Complex a = this;
        double real = a.re - b.re;
        double imag = a.im - b.im;
        return new Complex(real, imag);
    }

    // return a new Complex object whose value is (this * b)
    public Complex times(Complex b) {
        Complex a = this;
        double real = a.re * b.re - a.im * b.im;
        double imag = a.re * b.im + a.im * b.re;
        return new Complex(real, imag);
    }

    // return a new object whose value is (this * alpha)
    public Complex scale(double alpha) {
        return new Complex(alpha * re, alpha * im);
    }

    // return a new Complex object whose value is the conjugate of this
    public Complex conjugate() {
        return new Complex(re, -im);
    }

    // return a new Complex object whose value is the reciprocal of this
    public Complex reciprocal() {
        double scale = re*re + im*im;
        return new Complex(re / scale, -im / scale);
    }

    // return the real or imaginary part
```

```
    public double re() { return re; }
    public double im() { return im; }

    // return a / b
    public Complex divides(Complex b) {
        Complex a = this;
        return a.times(b.reciprocal());
    }

    // return a new Complex object whose value is the complex exponential of this
    public Complex exp() {
        return new Complex(Math.exp(re) * Math.cos(im), Math.exp(re) * Math.sin(im));
    }

    // return a new Complex object whose value is the complex sine of this
    public Complex sin() {
        return new Complex(Math.sin(re) * Math.cosh(im), Math.cos(re) * Math.sinh(im));
    }

    // return a new Complex object whose value is the complex cosine of this
    public Complex cos() {
        return new Complex(Math.cos(re) * Math.cosh(im), -Math.sin(re) * Math.sinh(im));
    }

    // return a new Complex object whose value is the complex tangent of this
    public Complex tan() {
        return sin().divides(cos());
    }



    // a static version of plus
    public static Complex plus(Complex a, Complex b) {
        double real = a.re + b.re;
        double imag = a.im + b.im;
        Complex sum = new Complex(real, imag);
        return sum;
    }

    // See Section 3.3.
    public boolean equals(Object x) {
        if (x == null) return false;
        if (this.getClass() != x.getClass()) return false;
        Complex that = (Complex) x;
        return (this.re == that.re) && (this.im == that.im);
    }

    // See Section 3.3.
    public int hashCode() {
        return Objects.hash(re, im);
    }

}
```

Listing A.2: Conversion of a recursive SSE FFT [38]

```
#include "FFTRecursiveNeon.h"
#include <arm_neon.h>
#define LOGTAG "FFTLIB"

cd **LUT;
cd I(0.0, 1.0);
```

```
void fftRecursiveNeonInit(int N) {
    int i;
    int n_luts = (int)(log(N)/log(2)) - 2;
    LUT = (cd**)malloc(n_luts * sizeof(cd*));
    for(i = 0; i < n_luts; i++) {
        int n = N / pow(2, i);
        LUT[i] = (cd*)memalign(16, n/2 * sizeof(cd));

        int j;
        for(j = 0; j < n/2; j+=4) {
            cd w[4];
            int k;
            for(k = 0; k < 4; k++) {
                double kth = -2 * (j+k) * M_PI / n;
                w[k] = cd(cos(kth), sin(kth));
            }
            LUT[i][j] = cd(w[0].real(), w[1].real());
            LUT[i][j+1] = cd(w[2].real(), w[3].real());
            LUT[i][j+2] = cd(w[0].imag(), w[1].imag());
            LUT[i][j+3] = cd(w[2].imag(), w[3].imag());
        }
    }

}

void fftRecursiveNeon(cd *in, cd* out, int log2stride, int stride, int N) {
    if(N == 2) {
        out[0]   = in[0] + in[stride];
        out[N/2] = in[0] - in[stride];
    }else if(N == 4){
        fftRecursiveNeon(in, out, log2stride+1, stride << 1, N >> 1);
        fftRecursiveNeon(in+stride, out+N/2, log2stride+1, stride << 1, N >> 1);


        cd temp0 = out[0] + out[2];
        cd temp1 = out[0] - out[2];
        cd temp2 = out[1] - I*out[3];
        cd temp3 = out[1] + I*out[3];
        if(log2stride) {
            out[0] = temp0.real() + temp2.real()*I;
            out[1] = temp1.real() + temp3.real()*I;
            out[2] = temp0.imag() + temp2.imag()*I;
            out[3] = temp1.imag() + temp3.imag()*I;
        }else{
            out[0] = temp0;
            out[2] = temp1;
            out[1] = temp2;
            out[3] = temp3;
        }
    }else if(!log2stride){
        fftRecursiveNeon(in, out, log2stride+1, stride << 1, N >> 1);
        fftRecursiveNeon(in+stride, out+N/2, log2stride+1, stride << 1, N >> 1);

        int k;
        for(k=0;k<N/2;k+=4) {
            float32x4_t Ok_re = vld1q_f32((float *)&out[k+N/2]);
            float32x4_t Ok_im = vld1q_f32((float *)&out[k+N/2+2]);
            float32x4_t w_re = vld1q_f32((float *)&LUT[log2stride][k]);
            float32x4_t w_im = vld1q_f32((float *)&LUT[log2stride][k+2]);
            float32x4_t Ek_re = vld1q_f32((float *)&out[k]);
            float32x4_t Ek_im = vld1q_f32((float *)&out[k+2]);
            float32x4_t wOk_re = vsubq_f32(vmulq_f32(Ok_re,w_re),vmulq_f32(Ok_im,w_im));
            float32x4_t wOk_im = vaddq_f32(vmulq_f32(Ok_re,w_im),vmulq_f32(Ok_im,w_re));
```

```
            float32x4_t out0_re = vaddq_f32(Ek_re, wOk_re);
            float32x4_t out0_im = vaddq_f32(Ek_im, wOk_im);
            float32x4_t out1_re = vsubq_f32(Ek_re, wOk_re);
            float32x4_t out1_im = vsubq_f32(Ek_im, wOk_im);
            float32x4_t out_0_low = vcombine_f32(vget_low_f32(out0_re), vget_low_f32(out0_im));
            float32x4_t out_0_high = vcombine_f32(vget_high_f32(out0_re), vget_high_f32(out0_im));
            float32x4_t out_1_low = vcombine_f32(vget_low_f32(out1_re), vget_low_f32(out1_im));
            float32x4_t out_1_high = vcombine_f32(vget_high_f32(out1_re), vget_high_f32(out1_im));
            vst1q_f32((float *)(out+k)      , out_0_low);
            vst1q_f32((float *)(out+k+2)    , out_0_high);
            vst1q_f32((float *)(out+k+N/2)  , out_1_low);
            vst1q_f32((float *)(out+k+N/2+2), out_1_high);
        }
    }else{
        fftRecursiveNeon(in, out, log2stride+1, stride << 1, N >> 1);
        fftRecursiveNeon(in+stride, out+N/2, log2stride+1, stride << 1, N >> 1);

        int k;
        for(k=0;k<N/2;k+=4) {
            float32x4_t Ok_re = vld1q_f32((float *)&out[k+N/2]);
            float32x4_t Ok_im = vld1q_f32((float *)&out[k+N/2+2]);
            float32x4_t w_re = vld1q_f32((float *)&LUT[log2stride][k]);
            float32x4_t w_im = vld1q_f32((float *)&LUT[log2stride][k+2]);
            float32x4_t Ek_re = vld1q_f32((float *)&out[k]);
            float32x4_t Ek_im = vld1q_f32((float *)&out[k+2]);
            float32x4_t wOk_re = vsubq_f32(vmulq_f32(Ok_re,w_re),vmulq_f32(Ok_im,w_im));
            float32x4_t wOk_im = vaddq_f32(vmulq_f32(Ok_re,w_im),vmulq_f32(Ok_im,w_re));
            vst1q_f32((float *)(out+k)      , vaddq_f32(Ek_re, wOk_re));
            vst1q_f32((float *)(out+k+2)    , vaddq_f32(Ek_im, wOk_im));
            vst1q_f32((float *)(out+k+N/2)  , vsubq_f32(Ek_re, wOk_re));
            vst1q_f32((float *)(out+k+N/2+2), vsubq_f32(Ek_im, wOk_im));
        }
    }
}
```

Listing A.3: Conversion of an iterative SSE FFT [42]

```
#include "FFTIterativeNeon.h"
#define LOGTAG "FFTLIB"

unsigned int reverse(int x)
{
    x = ((x >> 1) & 0x55555555u) | ((x & 0x55555555u) << 1);
    x = ((x >> 2) & 0x33333333u) | ((x & 0x33333333u) << 2);
    x = ((x >> 4) & 0x0f0f0f0fu) | ((x & 0x0f0f0f0fu) << 4);
    x = ((x >> 8) & 0x00ff00ffu) | ((x & 0x00ff00ffu) << 8);
    x = ((x >> 16) & 0xffffu) | ((x & 0xffffu) << 16);
    return x;
}

void* newTable1D(int len, int sizeOneElement)
{

    // Declare the table pointer
    char* tablePtr = NULL;

    // Declare the memory size
    int sizeTable = 0;

    // Padding
    char padding = 0;

    // Round up the size of the table such
```

```
    // that it can fit an alignment to 16 bytes
    sizeTable = sizeOneElement * len + 16;

    // Allocate memory
    tablePtr = (char *) malloc(sizeTable);

    // Compute the padding required
    padding = (char) (16 - (((size_t) tablePtr) & 0x0000000F));

    *((char*) (tablePtr + padding - 1)) = padding;

    // Return the pointer to the beginning of the table
    return ((void*) (tablePtr + padding));

}

void deleteTable1D(void* tablePtr)
{

    // Padding
    char padding;

    // Beginning of the allocated memory
    void* allocatedMemory;

    // Get the padding
    padding = *(((char*) tablePtr) - 1);

    // Get the pointer
    allocatedMemory = (void*) (((char*) tablePtr) - padding);

    // Free
    free(allocatedMemory);

}

void fftTerminate(struct objFFT* myFFT)
{

    // Free memory
    deleteTable1D((void*) myFFT->WnReal);
    deleteTable1D((void*) myFFT->WnImag);
    deleteTable1D((void*) myFFT->simdWnReal);
    deleteTable1D((void*) myFFT->simdWnImag);
    deleteTable1D((void*) myFFT->workingArrayReal);
    deleteTable1D((void*) myFFT->workingArrayImag);
    deleteTable1D((void*) myFFT->fftTwiceReal);
    deleteTable1D((void*) myFFT->fftTwiceRealFlipped);
    deleteTable1D((void*) myFFT->fftTwiceImag);
    deleteTable1D((void*) myFFT->fftTwiceImagFlipped);
    deleteTable1D((void*) myFFT->emptyArray);
    deleteTable1D((void*) myFFT->trashArray);
    deleteTable1D((void*) myFFT->revBitOrderArray);
    deleteTable1D((void*) myFFT->simdARealGroups);
    deleteTable1D((void*) myFFT->simdAImagGroups);
    deleteTable1D((void*) myFFT->simdBRealGroups);
    deleteTable1D((void*) myFFT->simdBImagGroups);
    deleteTable1D((void*) myFFT->simdRRealGroups);
    deleteTable1D((void*) myFFT->simdRImagGroups);
    deleteTable1D((void*) myFFT->simdAIndividual);
    deleteTable1D((void*) myFFT->simdBIndividual);

}
```

```
void fftIterativeNeonInit(struct objFFT* myFFT,
                          struct ParametersStruct* myParameters,
                          unsigned int size) {
    // Temporary variable
    unsigned int tmpFrameSize;

    // Temporary variable
    unsigned int tmpNumberLevels;

    // Define the index to generate Wn(r)
    unsigned int r;

    // Define the index to generate the reverse bit order array
    unsigned int indexRevBitOrder;

    // Define the index to generate the empty array
    unsigned int emptyIndex;

    // Define the INDEX of the input parameter a
    unsigned int a;
    // Define the INDEX of the input parameter b
    unsigned int b;

    // Define accumulator to compute the index of parameters a and b
    unsigned int accumulatorA;
    // Define accumulator to compute the index of parameter r
    unsigned int accumulatorR;

    // Define the nubmer of groups in the current level
    unsigned int numberGroups;
    // Define the number of points per group
    unsigned int numberSubGroups;

    // Define the index of the level
    unsigned int indexLevel;
    // Define the index of the group
    unsigned int indexGroup;
    // Define the index of the point inside the group
    unsigned int indexSubGroup;

    // Define the index of the twiddle-factor in memory
    unsigned int indexTwiddle;

    // Define the index of the simd array for a with groups
    unsigned int simdAIndexGroup;
    // Define the index of the simd array for b with groups
    unsigned int simdBIndexGroup;
    // Define the index of the simd array for r with groups
    unsigned int simdRIndexGroup;
    // Define the index of the simd array for a with individual elements
    unsigned int simdAIndexIndividual;
    // Define the index of the simd array for b with individual elements
    unsigned int simdBIndexIndividual;

    // ***************************************************************
    // * STEP 1: Load parameters                                     *
    // ***************************************************************

    myFFT->FFT_SIZE = size;

    tmpFrameSize = myFFT->FFT_SIZE;
```

```
tmpNumberLevels = 0;

while(tmpFrameSize > 1)
{
    tmpNumberLevels++;
    tmpFrameSize /= 2;
}

myFFT->FFT_NBLEVELS = tmpNumberLevels;
myFFT->FFT_HALFSIZE = myFFT->FFT_SIZE / 2;
myFFT->FFT_SIZE_INV = (1.0f / myFFT->FFT_SIZE);
myFFT->FFT_SIMD_GROUP = ((myFFT->FFT_SIZE/2) * (myFFT->FFT_NBLEVELS-2) / 4);
myFFT->FFT_SIMD_INDIVIDUAL = ((myFFT->FFT_SIZE/2) * 2);

// ************************************************************************
// * STEP 2: Initialize context                                           *
// ************************************************************************

// +--------------------------------------------------------------------+
// | Step A: Create arrays                                              |
// +--------------------------------------------------------------------+

myFFT->WnReal = (float*) newTable1D(myFFT->FFT_HALFSIZE, sizeof(float));
myFFT->WnImag = (float*) newTable1D(myFFT->FFT_HALFSIZE, sizeof(float));
myFFT->simdWnReal = (float*) newTable1D(myFFT->FFT_SIMD_GROUP*4, sizeof(float));
myFFT->simdWnImag = (float*) newTable1D(myFFT->FFT_SIMD_GROUP*4, sizeof(float));
myFFT->workingArrayReal = (float*) newTable1D(myFFT->FFT_SIZE, sizeof(float));
myFFT->workingArrayImag = (float*) newTable1D(myFFT->FFT_SIZE, sizeof(float));
myFFT->fftTwiceReal = (float*) newTable1D(myFFT->FFT_SIZE, sizeof(float));
myFFT->fftTwiceRealFlipped = (float*) newTable1D(myFFT->FFT_SIZE, sizeof(float));
myFFT->fftTwiceImag = (float*) newTable1D(myFFT->FFT_SIZE, sizeof(float));
myFFT->fftTwiceImagFlipped = (float*) newTable1D(myFFT->FFT_SIZE, sizeof(float));
myFFT->emptyArray = (float*) newTable1D(myFFT->FFT_SIZE, sizeof(float));
myFFT->trashArray = (float*) newTable1D(myFFT->FFT_SIZE, sizeof(float));
myFFT->revBitOrderArray = (unsigned int*) newTable1D(myFFT->FFT_SIZE, sizeof(unsigned int));
myFFT->simdARealGroups = (float**) newTable1D(myFFT->FFT_SIMD_GROUP, sizeof(float*));
myFFT->simdAImagGroups = (float**) newTable1D(myFFT->FFT_SIMD_GROUP, sizeof(float*));
myFFT->simdBRealGroups = (float**) newTable1D(myFFT->FFT_SIMD_GROUP, sizeof(float*));
myFFT->simdBImagGroups = (float**) newTable1D(myFFT->FFT_SIMD_GROUP, sizeof(float*));
myFFT->simdRRealGroups = (float**) newTable1D(myFFT->FFT_SIMD_GROUP, sizeof(float*));
myFFT->simdRImagGroups = (float**) newTable1D(myFFT->FFT_SIMD_GROUP, sizeof(float*));
myFFT->simdAIndividual = (float**) newTable1D(myFFT->FFT_SIMD_INDIVIDUAL, sizeof(float*));
myFFT->simdBIndividual = (float**) newTable1D(myFFT->FFT_SIMD_INDIVIDUAL, sizeof(float*));

// +--------------------------------------------------------------------+
// | Step B: Generate the FFT factors Wn(r)                             |
// +--------------------------------------------------------------------+

// Generate Wn(r) = exp(-j*2*pi*r/N) for r = 0 ... (N/2 - 1)
for (r = 0; r < myFFT->FFT_HALFSIZE; r++)
{

    myFFT->WnReal[r] = cosf(2.0f * M_PI * r / myFFT->FFT_SIZE);
    myFFT->WnImag[r] = -1.0f * sinf(2.0f * M_PI * r / myFFT->FFT_SIZE);

}

// +--------------------------------------------------------------------+
// | Step C: Generate an array with reverse-bit indexes                 |
// +--------------------------------------------------------------------+

int shift = 1 + __builtin_clz(myFFT->FFT_SIZE);
// Generate an array of reverse bit order
```

```
for (indexRevBitOrder = 0; indexRevBitOrder < myFFT->FFT_SIZE; indexRevBitOrder++)
{
    myFFT->revBitOrderArray[indexRevBitOrder] = reverse(indexRevBitOrder) >> shift;
}

// +-----------------------------------------------------------------+
// | Step D: Generate an empty array                                 |
// +-----------------------------------------------------------------+

// Generate an empty array (will be used as a dummy array for the imaginary
// part when the FFT of a single real signal is computed)
for (emptyIndex = 0; emptyIndex < myFFT->FFT_SIZE; emptyIndex++)
{
    myFFT->emptyArray[emptyIndex] = 0;
}

// +-----------------------------------------------------------------+
// | Step E: SIMD: Compute the indexes used for memory accesses      |
// +-----------------------------------------------------------------+

// Load parameters
numberGroups = 1;
numberSubGroups = myFFT->FFT_HALFSIZE;

// Initialize pointers
simdAIndexGroup = 0;
simdBIndexGroup = 0;
simdRIndexGroup = 0;
simdAIndexIndividual = 0;
simdBIndexIndividual = 0;
indexTwiddle = 0;

// Loop for each level
for (indexLevel = 0; indexLevel < myFFT->FFT_NBLEVELS; indexLevel++)
{

    accumulatorA = 0;
    accumulatorR = 0;

    // Loop for each group in the current level
    for (indexGroup = 0; indexGroup < numberGroups; indexGroup++)
    {

        // Loop for each element of the group
        for (indexSubGroup = 0; indexSubGroup < numberSubGroups; indexSubGroup++)
        {

            // Calculate the indexes
            a = accumulatorA;
            b = accumulatorA + numberSubGroups;
            r = accumulatorR;
            accumulatorA++;
            accumulatorR += numberGroups;

            // Check if there are groups of at least 4 elements
            if (numberSubGroups >= 4)
            {

                // Copy corresponding twiddle factor
                myFFT->simdWnReal[indexTwiddle] = myFFT->WnReal[r];
                myFFT->simdWnImag[indexTwiddle] = myFFT->WnImag[r];
                indexTwiddle++;
```

```
                    // Check if a is a multiple of 4
                    if ((a / 4.0f) == floorf(a/4.0f))
                    {
                        myFFT->simdARealGroups[simdAIndexGroup] = &myFFT->workingArrayReal[a];
                        myFFT->simdAImagGroups[simdAIndexGroup] = &myFFT->workingArrayImag[a];
                        myFFT->simdBRealGroups[simdBIndexGroup] = &myFFT->workingArrayReal[b];
                        myFFT->simdBImagGroups[simdBIndexGroup] = &myFFT->workingArrayImag[b];
                        myFFT->simdRRealGroups[simdRIndexGroup] = &myFFT->simdWnReal[indexTwiddle - 1];
                        myFFT->simdRImagGroups[simdRIndexGroup] = &myFFT->simdWnImag[indexTwiddle - 1];

                        simdAIndexGroup++;
                        simdBIndexGroup++;
                        simdRIndexGroup++;
                    }

                }
                else
                {

                    myFFT->simdAIndividual[simdAIndexIndividual++] = &myFFT->workingArrayReal[a];
                    myFFT->simdBIndividual[simdBIndexIndividual++] = &myFFT->workingArrayReal[b];

                }

            }

            // Update accumulators
            accumulatorA += numberSubGroups;
            accumulatorR = 0;

        }

        // Divide the number of points by group by 2 for the next level
        numberSubGroups >>= 1;
        // Multiply the number of groups by 2 for the next level
        numberGroups <<= 1;

    }

}

void fftIterativeNeon(struct objFFT* myFFT,
                      float* sourceArrayReal,
                      float* sourceArrayImag,
                      float* destArrayReal,
                      float* destArrayImag) {

    // Array index
    unsigned int indexGroup;
    unsigned int indexLevel;
    unsigned int indexArray;

    // Define variables for the last two levels
    float valueAReal;
    float valueAImag;
    float valueBReal;
    float valueBImag;
    float newValueAReal;
    float newValueAImag;
    float newValueBReal;
    float newValueBImag;
    unsigned int a;
    unsigned int b;
```

```
unsigned int accumulatorA;

// Define the index to generate the reverse bit order array
unsigned int indexRevBitOrder;

// SIMD registers
__m128_mod regA, regB, regC, regD, regE, regF, regG;

// ************************************************************************
// * STEP 0: Copy source                                                  *
// ************************************************************************

// Copy all elements from the source array in the working array
for (indexArray = 0; indexArray < myFFT->FFT_SIZE; indexArray+=4)
{

    // Load sourceArrayReal[k] in regA
    regA.m128 = vld1q_f32(&sourceArrayReal[indexArray]);

    // Load sourceArrayImag[k] in regB
    regB.m128 = vld1q_f32(&sourceArrayImag[indexArray]);

    // Copy regA in workingArrayReal[k]
    vst1q_f32(&myFFT->workingArrayReal[indexArray], regA.m128);

    // Copy regB in workingArrayImag[k]
    vst1q_f32(&myFFT->workingArrayImag[indexArray], regB.m128);

}

// ************************************************************************
// * STEP 1: Perform computations for all levels except two last one      *
// ************************************************************************

// Loop for the groups
indexGroup = 0;

for (indexLevel = 0; indexLevel < (myFFT->FFT_NBLEVELS - 2); indexLevel++)
{

    for (indexArray = 0; indexArray < (myFFT->FFT_SIZE/8); indexArray++)
    {

        // Load arguments aReal, aImag, bReal and bImag
        regA.m128 = vld1q_f32(myFFT->simdARealGroups[indexGroup]);
        regB.m128 = vld1q_f32(myFFT->simdAImagGroups[indexGroup]);
        regC.m128 = vld1q_f32(myFFT->simdBRealGroups[indexGroup]);
        regD.m128 = vld1q_f32(myFFT->simdBImagGroups[indexGroup]);

        // First addition: (aReal + bReal), (aImag + bImag)
        regE.m128 = vaddq_f32(regA.m128,regC.m128);
        regF.m128 = vaddq_f32(regB.m128,regD.m128);

        // Store A = (aReal + bReal) + j(aImag + bImag)
        vst1q_f32(myFFT->simdARealGroups[indexGroup], regE.m128);
        vst1q_f32(myFFT->simdAImagGroups[indexGroup], regF.m128);

        // Second addition: B = (aReal - bReal), (aImag - bImag)
        regE.m128 = vsubq_f32(regA.m128,regC.m128);
        regF.m128 = vsubq_f32(regB.m128,regD.m128);

        // Load twiddle factor WnReal and WnImag
        regA.m128 = vld1q_f32(myFFT->simdRRealGroups[indexGroup]);
```

```
            regB.m128 = vld1q_f32(myFFT->simdRImagGroups[indexGroup]);

            // Multiplications

            // (E*A - F*B)
            regC.m128 = vmulq_f32(regE.m128,regA.m128);
            regD.m128 = vmulq_f32(regF.m128,regB.m128);
            regG.m128 = vsubq_f32(regC.m128,regD.m128);

            // (F*A + E*B)
            regC.m128 = vmulq_f32(regF.m128,regA.m128);
            regD.m128 = vmulq_f32(regE.m128,regB.m128);
            regA.m128 = vaddq_f32(regC.m128,regD.m128);

            // Store B = (aReal - bReal) * WnReal - (aImag - bImag) * WnImag
            //         + j[ (aImag - bImag) * WnReal + (aReal - bReal) * WnImag ]

            vst1q_f32(myFFT->simdBRealGroups[indexGroup], regG.m128);
            vst1q_f32(myFFT->simdBImagGroups[indexGroup], regA.m128);

            // Increment the counter
            indexGroup++;

        }

    }

    // ************************************************************************
    // * STEP 2: Perform computations for level 1                             *
    // ************************************************************************

    accumulatorA = 0;

    // Loop for each group in the current level
    for (indexGroup = 0; indexGroup < myFFT->FFT_SIZE/4; indexGroup++)
    {

        // Calculate the indexes
        a = accumulatorA;
        b = accumulatorA + 2;
        accumulatorA++;

        // Load the values a and b (these are complex values)
        valueAReal = myFFT->workingArrayReal[a];
        valueAImag = myFFT->workingArrayImag[a];
        valueBReal = myFFT->workingArrayReal[b];
        valueBImag = myFFT->workingArrayImag[b];

        // Apply A = a + b
        newValueAReal = valueAReal + valueBReal;
        newValueAImag = valueAImag + valueBImag;

        // Apply B = a - b
        newValueBReal = valueAReal - valueBReal;
        newValueBImag = valueAImag - valueBImag;

        // Save results at the same place as the initial values
        myFFT->workingArrayReal[a] = newValueAReal;
        myFFT->workingArrayImag[a] = newValueAImag;
        myFFT->workingArrayReal[b] = newValueBReal;
        myFFT->workingArrayImag[b] = newValueBImag;

        // Calculate the indexes
```

```
        a = accumulatorA;
        b = accumulatorA + 2;
        accumulatorA+=3;

        // Load the values a and b (these are complex values)
        valueAReal = myFFT->workingArrayReal[a];
        valueAImag = myFFT->workingArrayImag[a];
        valueBReal = myFFT->workingArrayReal[b];
        valueBImag = myFFT->workingArrayImag[b];

        // Apply A = a + b
        newValueAReal = valueAReal + valueBReal;
        newValueAImag = valueAImag + valueBImag;

        // Apply B = (a - b) * -j = [(aReal - bReal) + j * (aImag - bImag)] * -j =
        //                          (aImag - bImag) + j * (bReal - aReal)
        newValueBReal = valueAImag - valueBImag;
        newValueBImag = valueBReal - valueAReal;

        // Save results at the same place as the initial values
        myFFT->workingArrayReal[a] = newValueAReal;
        myFFT->workingArrayImag[a] = newValueAImag;
        myFFT->workingArrayReal[b] = newValueBReal;
        myFFT->workingArrayImag[b] = newValueBImag;

    }

    // ************************************************************************
    // * STEP 3: Perform computations for level 0                             *
    // ************************************************************************

    accumulatorA = 0;

    // Loop for each group in the current level
    for (indexGroup = 0; indexGroup < myFFT->FFT_SIZE/2; indexGroup++)
    {

        // Calculate the indexes
        a = accumulatorA;
        b = accumulatorA + 1;
        accumulatorA+=2;

        // Load the values a and b (these are complex values)
        valueAReal = myFFT->workingArrayReal[a];
        valueAImag = myFFT->workingArrayImag[a];
        valueBReal = myFFT->workingArrayReal[b];
        valueBImag = myFFT->workingArrayImag[b];

        // Apply A = a + b
        newValueAReal = valueAReal + valueBReal;
        newValueAImag = valueAImag + valueBImag;

        // Apply B = a - b
        newValueBReal = valueAReal - valueBReal;
        newValueBImag = valueAImag - valueBImag;

        // Save results at the same place as the initial values
        myFFT->workingArrayReal[a] = newValueAReal;
        myFFT->workingArrayImag[a] = newValueAImag;
        myFFT->workingArrayReal[b] = newValueBReal;
        myFFT->workingArrayImag[b] = newValueBImag;

    }
```

```
    // ************************************************************************
    // * STEP 4: Copy result                                                  *
    // ************************************************************************

    // Reorder result (it is actually in reverse bit order) and copy to destination array
    for (indexRevBitOrder = 0; indexRevBitOrder < myFFT->FFT_SIZE; indexRevBitOrder++)
    {
        destArrayReal[indexRevBitOrder] = myFFT->workingArrayReal[myFFT->revBitOrderArray[indexRevBitOrder
        destArrayImag[indexRevBitOrder] = myFFT->workingArrayImag[myFFT->revBitOrderArray[indexRevBitOrder

    }
}
```

Appendix B

Results

B.1 Data

Table B.1: Data for Java Columbia Iterative, Time (ms)

Block size	$\bar{X}$	s	$SE_{\bar{X}}$	$ME_{\bar{X}}$
16	0.0028	0.0003	0.0000	0.0001
32	0.0041	0.0002	0.0000	0.0001
64	0.0070	0.0006	0.0001	0.0002
128	0.0136	0.0041	0.0004	0.0008
256	0.0266	0.0038	0.0004	0.0008
512	0.0557	0.0054	0.0005	0.0010
1024	0.1197	0.0061	0.0006	0.0012
2048	0.3434	0.0267	0.0027	0.0053
4096	1.2992	0.1707	0.0171	0.0335
8192	2.5070	0.2263	0.0226	0.0443
16384	6.5204	0.1613	0.0161	0.0316
32768	12.7414	1.9447	0.1945	0.3812
65536	23.9784	0.4023	0.0402	0.0788
131072	95.1627	9.1676	0.9168	1.7969
262144	290.7303	20.8047	2.0805	4.0778

Table B.2: Data for Java Princeton Iterative, Time (ms)

Block size	$\bar{X}$	s	$SE_{\bar{X}}$	$ME_{\bar{X}}$
16	0.0112	0.0023	0.0002	0.0004
32	0.0257	0.0047	0.0005	0.0010
64	0.0803	0.0206	0.0021	0.0041
128	0.2003	0.0752	0.0075	0.0147
256	0.4139	0.1053	0.0105	0.0206
512	0.8952	0.3609	0.0361	0.0708
1024	1.9717	0.6992	0.0699	0.1370
2048	4.6686	1.4393	0.1439	0.2820
4096	12.5215	2.4898	0.2490	0.4880
8192	24.6632	6.3895	0.6390	1.2524
16384	62.5681	7.9793	0.7979	1.5639
32768	152.1322	6.7763	0.6776	1.3281
65536	366.0864	11.0232	1.1023	2.1605
131072	791.0354	15.8332	1.5833	3.1033
262144	1875.8632	117.5501	11.7550	23.0398

Table B.3: Data for Java Princeton Recursive, Time (ms)

Block size	$\bar{X}$	s	$SE_{\bar{X}}$	$ME_{\bar{X}}$
16	0.0337	0.0055	0.0005	0.0010
32	0.0802	0.0158	0.0016	0.0031
64	0.1957	0.0218	0.0022	0.0043
128	0.3556	0.0631	0.0063	0.0123
256	0.9540	0.3545	0.0354	0.0694
512	2.0471	0.7398	0.0740	0.1450
1024	4.5326	1.3985	0.1399	0.2742
2048	10.3113	3.1760	0.3176	0.6225
4096	23.2240	6.2771	0.6277	1.2303
8192	51.3207	10.7660	1.0766	2.1101
16384	110.4185	15.7758	1.5776	3.0921
32768	240.7083	33.4933	3.3493	6.5646
65536	560.8628	86.0623	8.6062	16.8682
131072	1254.8310	146.0792	14.6079	28.6315
262144	2919.3773	122.8702	12.2870	24.0825

Table B.4: Data for C++ Columbia Iterative, Time (ms)

Block size	$\bar{X}$	s	$SE_{\bar{X}}$	$ME_{\bar{X}}$
16	0.0053	0.0043	0.0004	0.0008
32	0.0055	0.0011	0.0001	0.0002
64	0.0070	0.0005	0.0001	0.0002
128	0.0101	0.0004	0.0000	0.0001
256	0.0169	0.0004	0.0000	0.0001
512	0.0319	0.0028	0.0003	0.0006
1024	0.0709	0.0049	0.0005	0.0010
2048	0.1503	0.0055	0.0005	0.0010
4096	0.7553	0.0167	0.0017	0.0033
8192	1.9094	0.0477	0.0048	0.0094
16384	4.4813	0.1622	0.0162	0.0318
32768	12.0190	0.2408	0.0241	0.0472
65536	32.5804	0.8242	0.0824	0.1615
131072	131.0146	2.4456	0.2446	0.4794
262144	303.4669	19.8806	1.9881	3.8967

Table B.5: Data for C++ Princeton Iterative, Time (ms)

Block size	$\bar{X}$	s	$SE_{\bar{X}}$	$ME_{\bar{X}}$
16	0.0075	0.0022	0.0002	0.0004
32	0.0113	0.0013	0.0001	0.0002
64	0.0185	0.0007	0.0001	0.0002
128	0.0333	0.0031	0.0003	0.0006
256	0.0628	0.0078	0.0008	0.0016
512	0.1206	0.0052	0.0005	0.0010
1024	0.2500	0.0077	0.0008	0.0016
2048	0.5075	0.0154	0.0015	0.0029
4096	1.1136	0.0223	0.0022	0.0043
8192	2.5455	0.2865	0.0286	0.0561
16384	5.4095	0.2428	0.0243	0.0476
32768	11.6946	0.1686	0.0169	0.0331
65536	24.7761	0.3669	0.0367	0.0719
131072	72.5146	2.3169	0.2317	0.4541
262144	229.1227	20.9318	2.0932	4.1027

Table B.6: Data for C++ Princeton Recursive, Time (ms)

Block size	$\bar{X}$	s	$SE_{\bar{X}}$	$ME_{\bar{X}}$
16	0.0291	0.0204	0.0020	0.0039
32	0.0509	0.0038	0.0004	0.0008
64	0.1017	0.0055	0.0005	0.0010
128	0.2178	0.0090	0.0009	0.0018
256	0.4492	0.0141	0.0014	0.0027
512	0.9353	0.0506	0.0051	0.0100
1024	1.9461	0.0288	0.0029	0.0057
2048	2.9871	0.2251	0.0225	0.0441
4096	6.1952	0.1583	0.0158	0.0310
8192	13.4611	0.1781	0.0178	0.0349
16384	28.2437	0.7945	0.0794	0.1556
32768	59.0249	0.5676	0.0568	0.1113
65536	123.5847	1.0501	0.1050	0.2058
131072	257.7359	4.0012	0.4001	0.7842
262144	551.2218	9.5268	0.9527	1.8673

Table B.7: Data for C++ KISS, Time (ms)

Block size	$\bar{X}$	s	$SE_{\bar{X}}$	$ME_{\bar{X}}$
16	0.0056	0.0012	0.0001	0.0002
32	0.0076	0.0015	0.0002	0.0004
64	0.0086	0.0005	0.0001	0.0002
128	0.0160	0.0030	0.0003	0.0006
256	0.0225	0.0011	0.0001	0.0002
512	0.0559	0.0050	0.0005	0.0010
1024	0.0898	0.0116	0.0012	0.0024
2048	0.2446	0.0132	0.0013	0.0025
4096	0.4632	0.0397	0.0040	0.0078
8192	1.4667	0.2088	0.0209	0.0410
16384	2.6454	0.5833	0.0583	0.1143
32768	6.3225	0.2190	0.0219	0.0429
65536	15.8927	2.1600	0.2160	0.4234
131072	39.9681	4.0773	0.4077	0.7991
262144	92.1686	9.6509	0.9651	1.8916

2.1.1 JNI

Table B.8: Data for JNI Vector, Time (ms)

Block size	$\bar{X}$	s	$SE_{\bar{X}}$	$ME_{\bar{X}}$
16	1.5287	0.0589	0.0059	0.0116
32	1.5296	0.1543	0.0154	0.0302
64	1.5208	0.0597	0.0060	0.0118
128	1.5489	0.1127	0.0113	0.0221
256	1.5052	0.0590	0.0059	0.0116
512	1.5333	0.0420	0.0042	0.0082
1024	1.6620	0.5883	0.0588	0.1152
2048	2.0567	4.6277	0.4628	0.9071
4096	1.6318	0.1484	0.0148	0.0290
8192	2.5256	10.4005	1.0400	2.0384
16384	1.3823	0.1767	0.0177	0.0347
32768	1.2437	0.1910	0.0191	0.0374
65536	1.4547	0.7032	0.0703	0.1378
131072	1.4870	0.3491	0.0349	0.0684
262144	2.2171	0.8063	0.0806	0.1580

Table B.9: Common table for JNI tests, Time (µs)

Block size	No params	Vector	Convert	Columbia
16	1.4510 ± 0.0664	1.5287 ± 0.0116	1.7662 ± 0.0655	2.6886 ± 0.0670
32	1.4303 ± 0.0096	1.5296 ± 0.0302	1.7688 ± 0.0214	2.7277 ± 0.1705
64	1.4307 ± 0.0092	1.5208 ± 0.0118	1.7427 ± 0.0155	2.9297 ± 0.5374
128	1.4541 ± 0.0161	1.5489 ± 0.0221	1.7880 ± 0.0455	2.7176 ± 0.0739
256	1.4349 ± 0.0084	1.5052 ± 0.0116	1.8713 ± 0.1376	2.7254 ± 0.0529
512	1.5359 ± 0.1423	1.5333 ± 0.0082	1.7869 ± 0.0149	2.7130 ± 0.0214
1024	1.4427 ± 0.0151	1.6620 ± 0.1152	1.7818 ± 0.0449	2.7870 ± 0.0394
2048	1.4416 ± 0.0153	2.0567 ± 0.9071	1.7636 ± 0.0347	2.7469 ± 0.0304
4096	1.4375 ± 0.0086	1.6318 ± 0.0290	1.9333 ± 0.1082	2.5359 ± 0.0394
8192	1.4287 ± 0.0082	2.5256 ± 2.0384	2.7318 ± 1.3338	2.5776 ± 0.0408
16384	1.4374 ± 0.0074	1.3823 ± 0.0347	2.2114 ± 0.2242	2.6386 ± 0.1345
32768	1.5422 ± 0.0531	1.2437 ± 0.0374	2.2224 ± 0.1211	2.8062 ± 0.0982
65536	1.5556 ± 0.0517	1.4547 ± 0.1378	2.6542 ± 0.3889	3.5229 ± 0.7291
131072	1.5282 ± 0.0116	1.4870 ± 0.0684	3.5047 ± 0.4238	3.8781 ± 0.3591
262144	1.5438 ± 0.0125	2.2171 ± 0.1580	5.8172 ± 0.9490	5.8157 ± 0.6435

2.1.2 Double Tables

Table B.10: Common table for `double` C++ FFT tests, Time (ms)

Block size	Columbia Iterative	KISS	Princeton Iterative	Princeton Recursive
16	0.0053 ± 0.0008	0.0056 ± 0.0002	0.0075 ± 0.0004	0.0291 ± 0.0039
32	0.0055 ± 0.0002	0.0076 ± 0.0004	0.0113 ± 0.0002	0.0509 ± 0.0008
64	0.0070 ± 0.0002	0.0086 ± 0.0002	0.0185 ± 0.0002	0.1017 ± 0.0010
128	0.0101 ± 0.0001	0.0160 ± 0.0006	0.0333 ± 0.0006	0.2178 ± 0.0018
256	0.0169 ± 0.0001	0.0225 ± 0.0002	0.0628 ± 0.0016	0.4492 ± 0.0027
512	0.0319 ± 0.0006	0.0559 ± 0.0010	0.1206 ± 0.0010	0.9353 ± 0.0100
1024	0.0709 ± 0.0010	0.0898 ± 0.0024	0.2500 ± 0.0016	1.9461 ± 0.0057
2048	0.1503 ± 0.0010	0.2446 ± 0.0025	0.5075 ± 0.0029	2.9871 ± 0.0441
4096	0.7553 ± 0.0033	0.4632 ± 0.0078	1.1136 ± 0.0043	6.1952 ± 0.0310
8192	1.9094 ± 0.0094	1.4667 ± 0.0410	2.5455 ± 0.0561	13.4611 ± 0.0349
16384	4.4813 ± 0.0318	2.6454 ± 0.1143	5.4095 ± 0.0476	28.2437 ± 0.1556
32768	12.0190 ± 0.0472	6.3225 ± 0.0429	11.6946 ± 0.0331	59.0249 ± 0.1113
65536	32.5804 ± 0.1615	15.8927 ± 0.4234	24.7761 ± 0.0719	123.5847 ± 0.2058
131072	131.0146 ± 0.4794	39.9681 ± 0.7991	72.5146 ± 0.4541	257.7359 ± 0.7842
262144	303.4669 ± 3.8967	92.1686 ± 1.8916	229.1227 ± 4.1027	551.2218 ± 1.8673

Table B.11: Common table for `double` Java tests, Time (ms)

Block size	Columbia Iterative	Princeton Iterative	Princeton Recursive
16	0.0028 ± 0.0001	0.0112 ± 0.0004	0.0337 ± 0.0010
32	0.0041 ± 0.0001	0.0257 ± 0.0010	0.0802 ± 0.0031
64	0.0070 ± 0.0002	0.0803 ± 0.0041	0.1957 ± 0.0043
128	0.0136 ± 0.0008	0.2003 ± 0.0147	0.3556 ± 0.0123
256	0.0266 ± 0.0008	0.4139 ± 0.0206	0.9540 ± 0.0694
512	0.0557 ± 0.0010	0.8952 ± 0.0708	2.0471 ± 0.1450
1024	0.1197 ± 0.0012	1.9717 ± 0.1370	4.5326 ± 0.2742
2048	0.3434 ± 0.0053	4.6686 ± 0.2820	10.3113 ± 0.6225
4096	1.2992 ± 0.0335	12.5215 ± 0.4880	23.2240 ± 1.2303
8192	2.5070 ± 0.0443	24.6632 ± 1.2524	51.3207 ± 2.1101
16384	6.5204 ± 0.0316	62.5681 ± 1.5639	110.4185 ± 3.0921
32768	12.7414 ± 0.3812	152.1322 ± 1.3281	240.7083 ± 6.5646
65536	23.9784 ± 0.0788	366.0864 ± 2.1605	560.8628 ± 16.8682
131072	95.1627 ± 1.7969	791.0354 ± 3.1033	1254.8310 ± 28.6315
262144	290.7303 ± 4.0778	1875.8632 ± 23.0398	2919.3773 ± 24.0825

Table B.12: Common table for `double` NEON tests, Time (ms)

Block size	Iterative	Recursive
16	0.0055 ± 0.0004	0.0102 ± 0.0006
32	0.0058 ± 0.0002	0.0132 ± 0.0004
64	0.0072 ± 0.0002	0.0210 ± 0.0008
128	0.0100 ± 0.0002	0.0359 ± 0.0024
256	0.0163 ± 0.0002	0.0657 ± 0.0012
512	0.0321 ± 0.0012	0.1242 ± 0.0022
1024	0.0673 ± 0.0014	0.2413 ± 0.0029
2048	0.1347 ± 0.0020	0.4821 ± 0.0041
4096	0.3530 ± 0.0123	0.9964 ± 0.0045
8192	1.3282 ± 0.0512	2.0665 ± 0.0098
16384	2.1058 ± 0.0563	3.4816 ± 0.1082
32768	3.9077 ± 0.0365	6.7110 ± 0.0468
65536	11.5019 ± 0.1031	16.7593 ± 0.5551
131072	31.5155 ± 0.7944	38.4122 ± 1.2140
262144	70.6807 ± 1.1096	79.6650 ± 2.4153

2.1.3 Float Tables

Table B.13: Common table for `float` Java tests, Time (ms)

Block size	Columbia Iterative	Princeton Iterative	Princeton Recursive
16	0.0034 ± 0.0010	0.0213 ± 0.0014	0.0519 ± 0.0043
32	0.0041 ± 0.0001	0.0526 ± 0.0145	0.1215 ± 0.0129
64	0.0070 ± 0.0001	0.0981 ± 0.0020	0.2413 ± 0.0108
128	0.0134 ± 0.0006	0.2275 ± 0.0208	0.4816 ± 0.0067
256	0.0269 ± 0.0020	0.5715 ± 0.0702	1.0557 ± 0.0143
512	0.0543 ± 0.0006	1.0707 ± 0.0400	1.9630 ± 0.0700
1024	0.1185 ± 0.0012	2.6187 ± 0.0913	4.0399 ± 0.0906
2048	0.2470 ± 0.0018	6.7912 ± 0.4751	10.3206 ± 0.3587
4096	0.8706 ± 0.0043	13.1420 ± 0.4057	26.1748 ± 0.4700
8192	2.1624 ± 0.0053	32.7020 ± 1.0504	57.2447 ± 0.7748
16384	5.4469 ± 0.0435	75.3437 ± 2.2330	121.2444 ± 1.1907
32768	11.9213 ± 0.0994	174.8462 ± 4.9882	265.7005 ± 2.1829
65536	26.5285 ± 0.1713	404.4631 ± 11.1087	565.5092 ± 4.2597
131072	62.9644 ± 0.2256	898.7526 ± 19.9597	1218.9377 ± 7.5046
262144	195.5186 ± 0.5860	1993.8636 ± 35.9358	2536.9244 ± 5.0801

Table B.14: Common table for `float` C++ tests, Time (ms)

Block size	Columbia Iterative	Princeton Iterative	Princeton Recursive
16	0.0050 ± 0.0002	0.0092 ± 0.0006	0.0235 ± 0.0012
32	0.0058 ± 0.0001	0.0140 ± 0.0010	0.0435 ± 0.0022
64	0.0071 ± 0.0002	0.0226 ± 0.0008	0.0844 ± 0.0012
128	0.0108 ± 0.0008	0.0395 ± 0.0010	0.1804 ± 0.0016
256	0.0169 ± 0.0002	0.0741 ± 0.0010	0.3778 ± 0.0027
512	0.0319 ± 0.0008	0.1455 ± 0.0018	0.7779 ± 0.0035
1024	0.0707 ± 0.0012	0.2929 ± 0.0063	1.6040 ± 0.0047
2048	0.1443 ± 0.0016	0.6039 ± 0.0049	3.3529 ± 0.0155
4096	0.5990 ± 0.0098	1.2772 ± 0.0063	6.3386 ± 0.1870
8192	1.5140 ± 0.0178	2.7634 ± 0.0188	10.5867 ± 0.0666
16384	3.8153 ± 0.0592	6.2555 ± 0.0327	22.6880 ± 0.0529
32768	9.2565 ± 0.3210	13.1659 ± 0.0670	46.3923 ± 0.1366
65536	25.0067 ± 0.5223	27.6772 ± 0.1523	101.9946 ± 2.5992
131072	57.1731 ± 2.2834	51.3645 ± 1.7228	207.7519 ± 3.9006
262144	211.9326 ± 3.1952	152.0646 ± 3.0811	436.9925 ± 5.0080

Table B.15: Common table for `float` NEON tests, Time (ms)

Block size	Iterative	Recursive
16	0.0055 ± 0.0004	0.0102 ± 0.0006
32	0.0058 ± 0.0002	0.0132 ± 0.0004
64	0.0072 ± 0.0002	0.0210 ± 0.0008
128	0.0100 ± 0.0002	0.0359 ± 0.0024
256	0.0163 ± 0.0002	0.0657 ± 0.0012
512	0.0321 ± 0.0012	0.1242 ± 0.0022
1024	0.0673 ± 0.0014	0.2413 ± 0.0029
2048	0.1347 ± 0.0020	0.4821 ± 0.0041
4096	0.3530 ± 0.0123	0.9964 ± 0.0045
8192	1.3282 ± 0.0512	2.0665 ± 0.0098
16384	2.1058 ± 0.0563	3.4816 ± 0.1082
32768	3.9077 ± 0.0365	6.7110 ± 0.0468
65536	11.5019 ± 0.1031	16.7593 ± 0.5551
131072	31.5155 ± 0.7944	38.4122 ± 1.2140
262144	70.6807 ± 1.1096	79.6650 ± 2.4153

2.1.4 C++ Float graphs

2.1.5 Java Float graphs

B.2 ARR

Table B.16: Common table for ARR C++ tests, Time (ms)

Block size	Columbia Iterative	Float Columbia Iterative	Float Princeton Iterative	Float Princeton Recursive	KISS	NEON Iterative	NEON Recursive	Princeton Iterative	Princeton Recursive
16	0.0061 ± 0.0018	0.0053 ± 0.0002	0.0094 ± 0.0008	0.0163 ± 0.0008	0.0064 ± 0.0012	0.0040 ± 0.0002	0.0073 ± 0.0006	0.0111 ± 0.0008	0.0277 ± 0.0024
32	0.0059 ± 0.0002	0.0059 ± 0.0002	0.0138 ± 0.0006	0.0313 ± 0.0014	0.0076 ± 0.0002	0.0043 ± 0.0002	0.0095 ± 0.0002	0.0162 ± 0.0002	0.0459 ± 0.0008
64	0.0075 ± 0.0002	0.0076 ± 0.0002	0.0235 ± 0.0024	0.0604 ± 0.0008	0.0091 ± 0.0002	0.0051 ± 0.0001	0.0153 ± 0.0006	0.0270 ± 0.0008	0.0996 ± 0.0012
128	0.0115 ± 0.0010	0.0109 ± 0.0002	0.0392 ± 0.0002	0.1288 ± 0.0012	0.0169 ± 0.0002	0.0081 ± 0.0006	0.0258 ± 0.0006	0.0497 ± 0.0037	0.2138 ± 0.0025
256	0.0193 ± 0.0006	0.0182 ± 0.0002	0.0751 ± 0.0010	0.2707 ± 0.0014	0.0249 ± 0.0008	0.0127 ± 0.0006	0.0473 ± 0.0010	0.0940 ± 0.0069	0.4424 ± 0.0029
512	0.0361 ± 0.0008	0.0336 ± 0.0002	0.1459 ± 0.0014	0.5583 ± 0.0047	0.0601 ± 0.0012	0.0260 ± 0.0016	0.0894 ± 0.0012	0.1777 ± 0.0057	0.9194 ± 0.0102
1024	0.0654 ± 0.0006	0.0821 ± 0.0078	0.2910 ± 0.0027	1.1490 ± 0.0037	0.0979 ± 0.0024	0.0485 ± 0.0012	0.1779 ± 0.0037	0.3384 ± 0.0022	1.9181 ± 0.0092
2048	0.1543 ± 0.0014	0.1429 ± 0.0016	0.6292 ± 0.0033	2.3902 ± 0.0123	0.2656 ± 0.0037	0.0949 ± 0.0014	0.3460 ± 0.0024	0.7161 ± 0.0035	4.0765 ± 0.0584
4096	0.7543 ± 0.0031	0.5091 ± 0.0104	1.2857 ± 0.0043	5.0112 ± 0.0155	0.4868 ± 0.0110	0.2473 ± 0.0069	0.7212 ± 0.0047	1.5521 ± 0.0059	6.5505 ± 0.1952
8192	1.9032 ± 0.0086	1.5087 ± 0.0186	2.7573 ± 0.0114	10.6528 ± 0.0696	1.4676 ± 0.0447	1.0102 ± 0.0259	1.5002 ± 0.0100	3.3914 ± 0.0417	13.3271 ± 0.0431
16384	4.4827 ± 0.0365	3.8806 ± 0.0729	6.2855 ± 0.0378	22.7562 ± 0.0700	3.1019 ± 0.1149	1.8799 ± 0.0220	3.1963 ± 0.0343	7.7722 ± 0.4914	27.7144 ± 0.0676
32768	12.0508 ± 0.0655	8.6418 ± 0.1760	13.1913 ± 0.0617	46.4408 ± 0.1043	9.4939 ± 0.0790	4.4666 ± 0.1247	9.0934 ± 0.1490	15.6001 ± 0.2354	57.6413 ± 0.1311
65536	32.8225 ± 0.1468	17.8318 ± 0.4596	19.9860 ± 0.2369	97.1953 ± 0.2493	14.3611 ± 0.3932	11.8452 ± 0.1245	20.7636 ± 0.2417	24.4857 ± 0.0737	121.5388 ± 0.2072
131072	131.1394 ± 0.4422	58.2148 ± 2.1795	49.5132 ± 1.5958	202.7283 ± 0.5855	43.7252 ± 1.2334	34.6190 ± 0.3093	37.1436 ± 1.0804	81.1304 ± 2.0184	254.6866 ± 0.7044
262144	315.5464 ± 4.5006	210.2958 ± 4.0184	125.8880 ± 2.9171	437.5968 ± 2.7505	105.4245 ± 2.5970	75.2245 ± 1.1039	78.6418 ± 2.3869	229.3064 ± 4.0156	540.1025 ± 1.5445

```
2.448 1.875 2.343 1.927 2.343 2.188 1.875 4.114 1.927 1.927 1.927 2.24
1.927 2.187 2.032 1.927 1.875 2.344 1.928 1.927 1.979 2.032 1.979 2.188
1.823 1.927 1.875 1.927 1.927 2.032 3.437 1.979 1.875 1.979 1.875
1.927 1.927 1.875 1.927 1.979 2.032 2.031 1.927 1.979 1.979 1.875
1.875 2.344 1.927 2.032 1.927 2.031 1.927 1.875 1.927 2.344 1.927
1.979 1.927 1.979 2.188 1.875 1.875 1.927 1.875 1.875 434.063 2.292
2.032 1.927 1.927 1.927 2.552 2.5 2.188 1.979 1.927 1.979 1.927 1.979
2.5 2.136 1.927 1.875 1.875 1.927 2.031 1.927 2.187 2.032 1.979 2.24
1.875 1.927 1.927 2.709 2.031 1.979 2.136 1.928
```

Figure B.1: Raw results from the Convert JNI test with block size 1024