Comparing Android Runtime with native: Fast Fourier Transform on Android

André Danielsson

March 16, 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

Android Mobile operating system. 1

Clang Compiler used by the NDK. 12

CMake Build tool used by the NDK. 10

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

FFT *Fast Fourier Transform* – Algorithm that implements the Discrete Fourier Transform. 2

JNI *Java Native Interface* – Framework that helps Java interact with native code. 3

NDK *Native Development Kit* – used to write android applications in C or C++. 10

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

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

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 [1]. One reason for this is because it was designed to be run on multiple different architectures [2]. 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 [3]. 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 [4] in November of 2014 [5]. Since then, Dalvik has been replaced by Android Runtime. Android Runtime, ART for short, differs from Dalvik in that it uses Ahead-Of-Time (AOT) 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

application, 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 [6] 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) [5], 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 [7].

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 [8]. Android Runtime (ART) features Ahead-Of-Time (AOT) compilation and Just-In-Time (JIT) compilation, garbage collection and debugging support [9]. 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

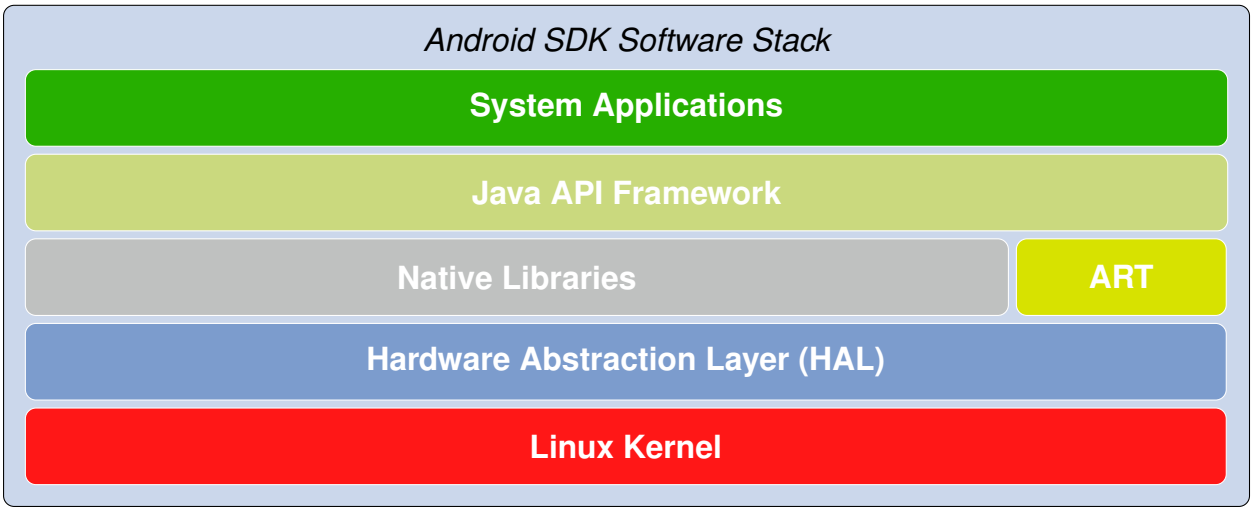


Figure 2.1: Android SDK Software Stack [9]

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 [9].

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 [2]. 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 [10, 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 [10, p. 159-160].

The DVM is a virtual machine optimized for devices where resources are limited [11]. 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 [12].

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 [4]. 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 [13] 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 [14]. This oat-file is in the Executable and Linkable Format (ELF) and can be seen as a wrapper of multiple dex-files [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 [16]. 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 [17]. 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 [18, 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++ [18, 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 [18, 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 [18, 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 [19].

Clang is as of March 2016 (NDK version 11) [20], 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 [21, 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

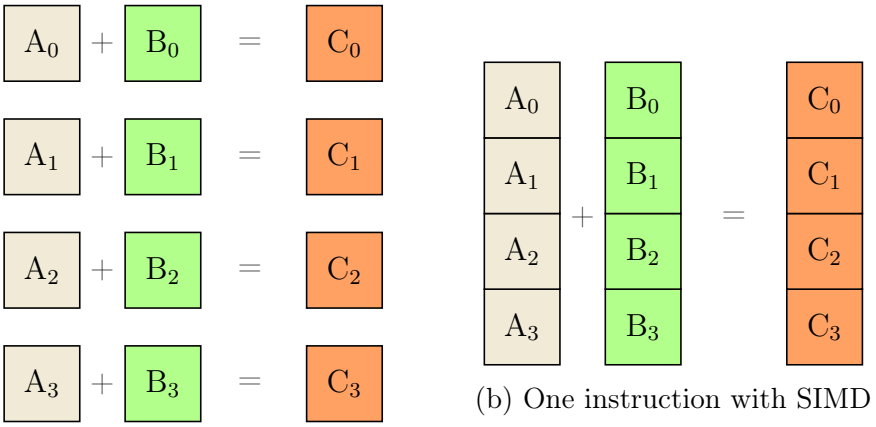
Figure 2.7: Loop Tiling

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 [23]

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 [22]. 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 [24]:

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 [25, 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 [26, 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 [27] called the Fast Fourier Transform (FFT).

$$X(k) = \sum_{n=0}^{N-1} x(n) \cdot e^{-j2\pi kn/N}, \quad k = 0, 1, 2, \ldots, N-1 \tag{2.1}$$

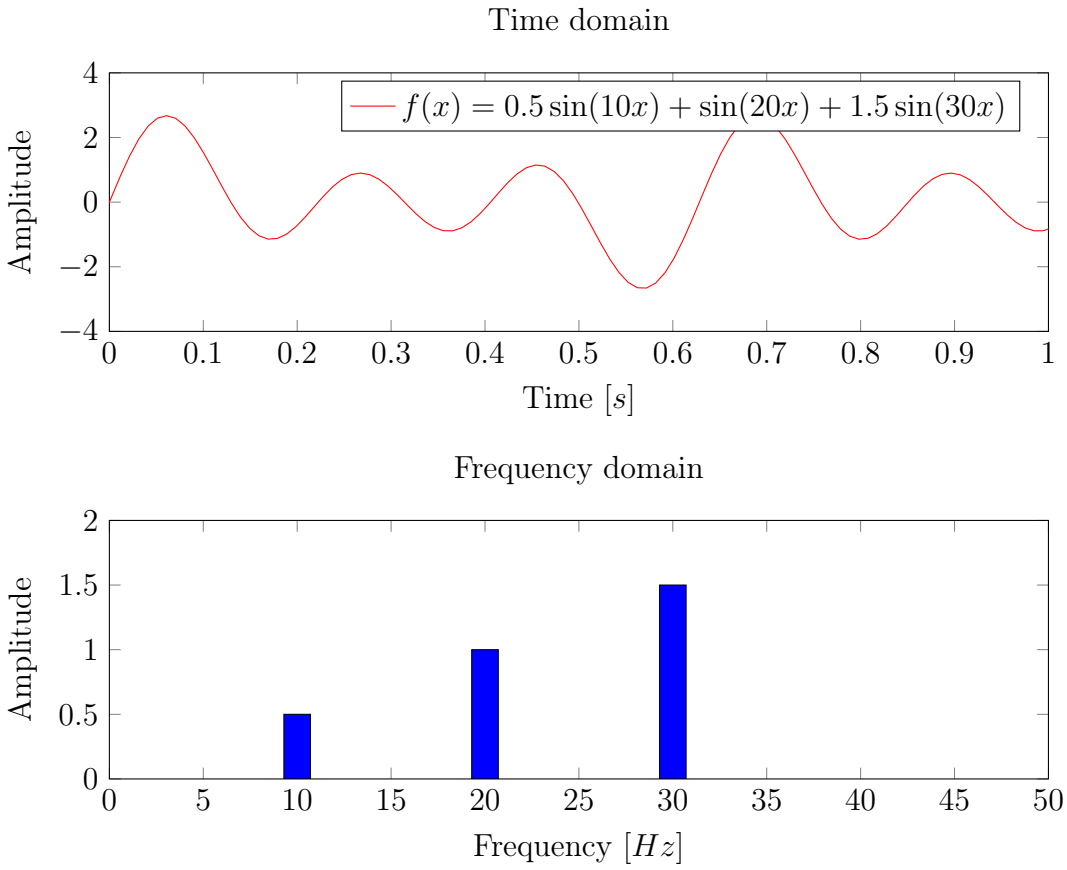


Figure 2.9: Time domain and frequency domain of a signal

2.7 Related work

A study called *FFT benchmark on Android devices: Java versus JNI* [28] 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 [29] 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* [30] 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 experiment is carried out correctly, many different tools for measurements was 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

This experiment consisted of tests for three different aspects of implementing algorithms in Java and in native code. To get an overview of how much of an impact different parts of an implementation have, the following subjects were investigated:

1. Cost of using the JNI

2. Compare well known libraries

3. Native exclusive optimization with NEON

The reason why it is relevant to know how significant the JNI 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 ones project. This test demonstrates how some different libraries

compare. It is helpful to see how viable different implementations are on Android, for both C++ libraries and Java libraries. It can also be useful to know how small implementations 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 the frequency resolution. This is the reason it is important to have a fast FFT.

Finally, optimizations only possible with native code is a good demonstration of how a developer can improve performance even more to fit the requirements while still retaining manageable source code. Having one single source file is valuable, especially for native libraries. This facilitates the process of adding libraries and based on the results found in this report, they can be sufficiently fast.

3.1.1 Hardware

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

During the tests, the screen brightness was set to as low as possible to minimize the risk for CPU throttling.

3.1.2 Benchmark Environment

During the tests, both cellular and wifi was 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 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 improvement. 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 drawback of using wall-clock time for measuring time. Because it can be changed at any time, it could result in too small or too large runtime depending on seemingly random factors. What is more preferable is to measure elapsed cpu time. This do not depend on a changeable wall clock but rather use hardware to interpret time. It is possible to use both `System.nanoTime()` and `SystemClock.elapsedRealtimeNanos()` for this purpose.

What is measured is the time to execute the tests assuming we have the desired input data and will get the required output data with no conversions. Different algorithms accepts different data types as input parameters. When using an algorithm, you would design your application such that it represents the data in the way it needs it. 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.

3.1.4 Memory measurement

The profiling tool provided by Android Studio was used to measure the amount of memory each test required. The method used was to attach the debugger to the app and measure

```
// 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

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.

3.2 Evaluation

The unit of the resulting data will be in milliseconds. To be able to have 100 executions run in reasonable time, the maximum size of the input data was limited to $2^{16} = 65536$. The sampling rate is what determines the highest frequency that could be found in the result. In this thesis, only the frequency range perceivable by the human ear ($\sim$ 20-22,000 Hz) is covered by the tests. Because the FFT is limited to sample sizes of powers of 2, the next power of 2 for a sampling rate of 44,100 is 2^{16}.

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 44100 (2^{16}) 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.

All test results are not presented in Chapter 4 - Results. In this chapter, only the results that were relevant to discuss about are brought up. All the test results are found in Appendix B. To visualize a result, bar charts and line graphs were used. To solve the problem that the FFT sizes were split into groups labeled *small size* ($2^4 - 2^7$) *medium size* ($2^8 - 2^{12}$) and *large size* ($2^{13} - 2^{16}$).

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.

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 which is the number of runs we execute for each test. The following formula calculates the sample mean [31, 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 [31, 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. The confidence interval is calculated by taking the standard error of the mean which is found by using the following formula [31, 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, three 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.

```
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

```
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 output. 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

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` must be called.

3.4 Fast Fourier Transform Algorithms

Different implementations of FFT was used in the libraries tests. Three of them were implemented in Java and one in C. All of them were 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 [32]
- Princeton Iterative [33]
- Columbia Iterative [34]
- Kiss (*Keep It Simple, Stupid*) FFT [35]

3.4.1 Java Libraries

The Princeton Recursive FFT is a straightforward implementation of the FFT with no radical optimizations written by Robert Sedgewick and Kevin Wayne [32]. This algorithm is implemented in Java. 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 [33], 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 [34] 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 used in future 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 which is used as a config. The FFT is executed when the `kiss_fft` function is called. The first parameter 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 it will 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 [36]. This algorithm has a 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 [37]. The converted program is listed in Appendix A.2. The second algorithm was a iterative implementation. This library is a straightforward implementation of FFT with SSE [38] and was written for a sound source localization system [39]. The code that was converted from SSE to NEON is presented in Appendix A.3.

CHAPTER 4

Results

Results from the experiments that were considered relevant are presented here.

4.1 JNI

4.2 FFT Libraries

4.2.1 Small block sizes

4.2.2 Medium block sizes

4.2.3 Large block sizes

4.3 Optimizations

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

Block size	No params	Vector	Convert	Columbia
500	1.2621 ± 0.2899	1.4861 ± 0.3212	3.9913 ± 4.0045	4.1858 ± 0.5788
1000	1.1597 ± 0.1160	1.5833 ± 0.2732	2.1632 ± 0.3910	4.3697 ± 0.5949
1500	1.1111 ± 0.0486	1.6789 ± 0.2603	2.1718 ± 0.2434	3.9478 ± 0.4098
2000	1.0504 ± 0.0163	2.2118 ± 0.3365	3.2917 ± 0.5392	8.2707 ± 2.2060
2500	1.0538 ± 0.0239	2.1580 ± 0.2989	2.8298 ± 0.8597	5.6526 ± 0.9798
3000	1.0382 ± 0.0147	2.8107 ± 0.5664	3.8542 ± 0.9794	6.4585 ± 1.6278
3500	1.0780 ± 0.0184	1.9844 ± 0.4092	3.5192 ± 0.7070	6.4132 ± 1.1072
4000	1.0886 ± 0.0192	1.3749 ± 0.1060	2.7327 ± 0.5055	5.7293 ± 1.0118
4500	1.0434 ± 0.0133	1.3282 ± 0.0843	3.5868 ± 1.0221	4.9514 ± 0.4904
5000	1.0798 ± 0.0372	1.5452 ± 0.2321	2.6892 ± 0.8099	4.7952 ± 0.4861

Figure 4.1: Line graph for all algorithms, Small block sizes, Time (ms)

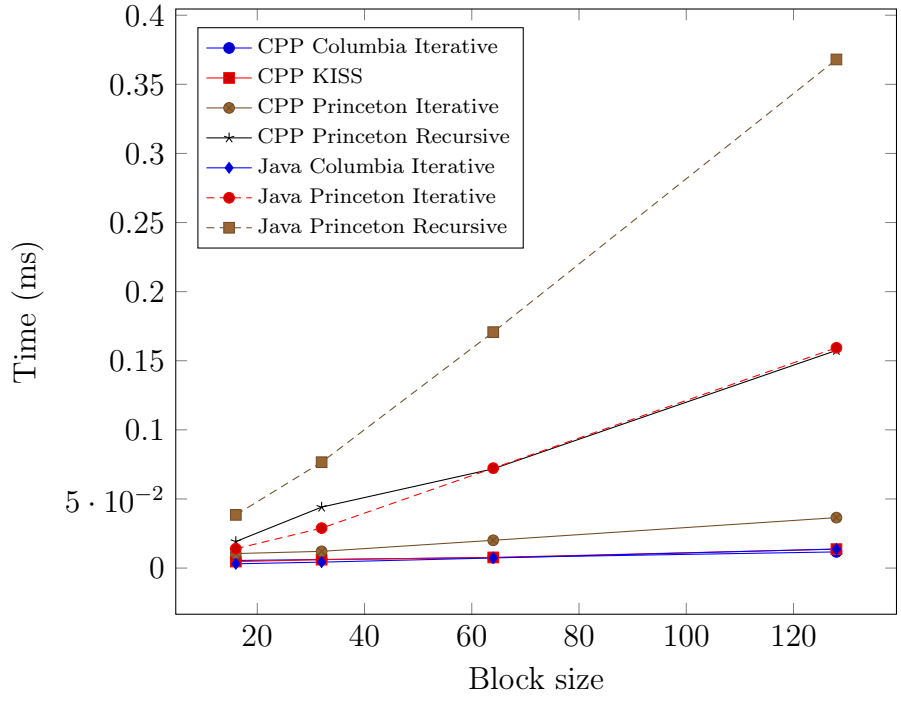


Figure 4.2: CPP Line graph SMALL

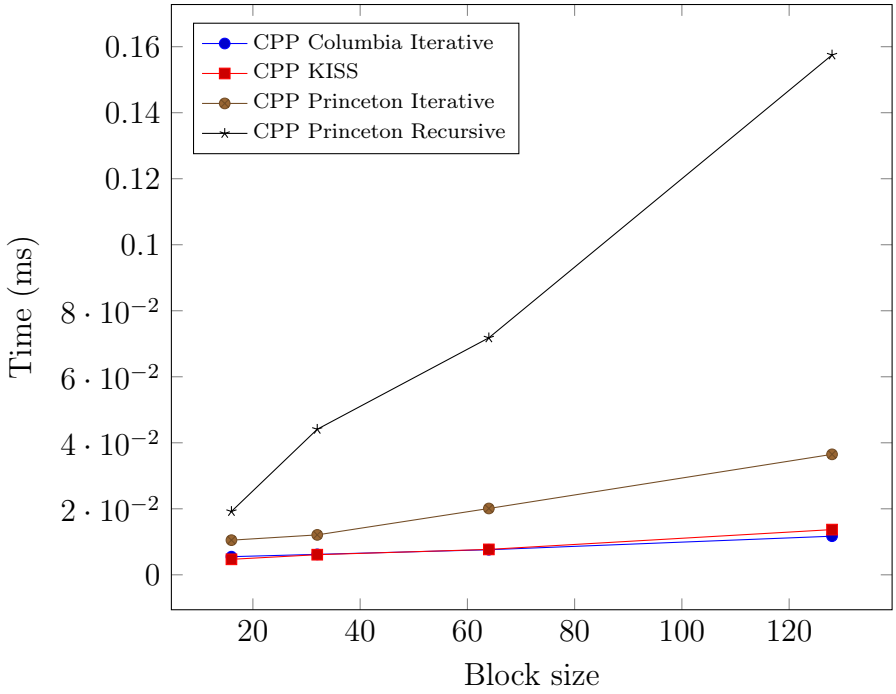


Table 4.2: Small block sizes C++ execution times, Time (ms)

Block size	Columbia Iterative	KISS	Princeton Iterative	Princeton Recursive
16	0.0055 ± 0.0004	0.0047 ± 0.0008	0.0105 ± 0.0035	0.0192 ± 0.0020
32	0.0062 ± 0.0004	0.0061 ± 0.0004	0.0121 ± 0.0004	0.0441 ± 0.0161
64	0.0076 ± 0.0002	0.0077 ± 0.0010	0.0201 ± 0.0010	0.0718 ± 0.0025
128	0.0117 ± 0.0004	0.0137 ± 0.0010	0.0365 ± 0.0027	0.1575 ± 0.0092

Table 4.3: Small block sizes Java execution times, Time (ms)

Block size	Columbia Iterative	KISS	Princeton Iterative	Princeton Recursive
16	0.0055 ± 0.0004	0.0047 ± 0.0008	0.0105 ± 0.0035	0.0192 ± 0.0020
32	0.0062 ± 0.0004	0.0061 ± 0.0004	0.0121 ± 0.0004	0.0441 ± 0.0161
64	0.0076 ± 0.0002	0.0077 ± 0.0010	0.0201 ± 0.0010	0.0718 ± 0.0025
128	0.0117 ± 0.0004	0.0137 ± 0.0010	0.0365 ± 0.0027	0.1575 ± 0.0092

Figure 4.3: Java Line graph SMALL

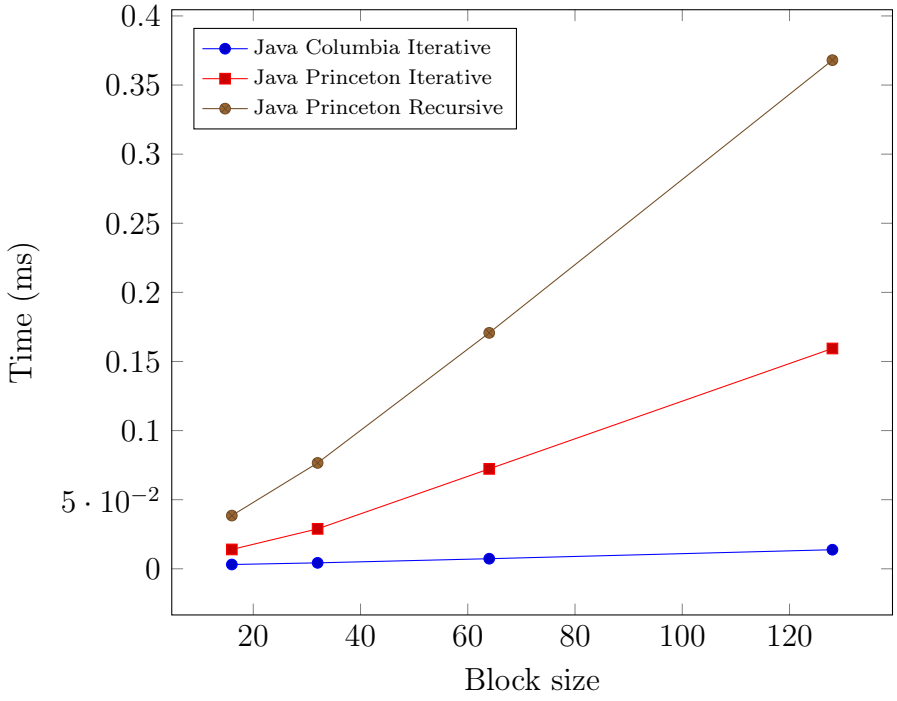


Figure 4.4: Line graph for all algorithms, Medium block sizes, Time (ms)

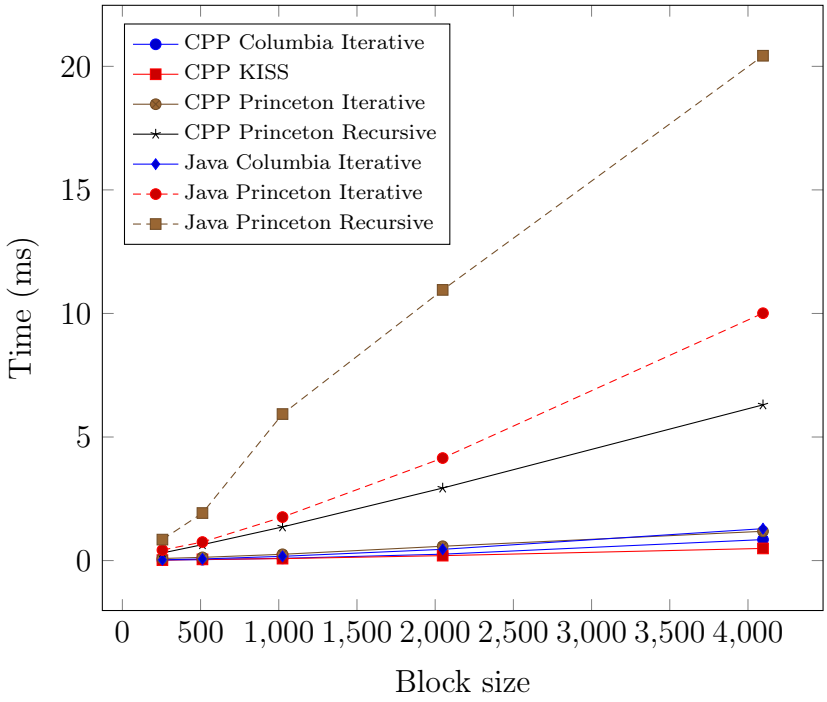


Figure 4.5: CPP Line graph MEDIUM

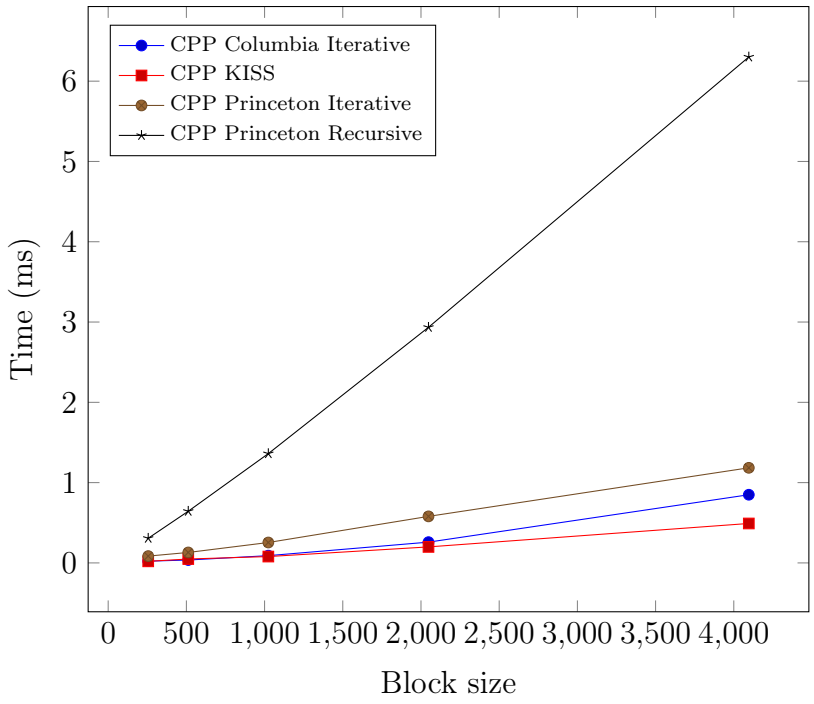


Table 4.4: Medium block sizes C++ execution times, Time (ms)

Block size	Columbia Iterative	KISS	Princeton Iterative	Princeton Recursive
256	0.0219 ± 0.0022	0.0197 ± 0.0016	0.0849 ± 0.0339	0.3071 ± 0.0051
512	0.0360 ± 0.0006	0.0487 ± 0.0029	0.1302 ± 0.0037	0.6434 ± 0.0076
1024	0.0905 ± 0.0076	0.0792 ± 0.0027	0.2542 ± 0.0051	1.3612 ± 0.0108
2048	0.2587 ± 0.0237	0.1994 ± 0.0053	0.5796 ± 0.0100	2.9344 ± 0.0178
4096	0.8491 ± 0.0335	0.4912 ± 0.0327	1.1845 ± 0.0241	6.3011 ± 0.1392

Table 4.5: Medium block sizes Java execution times, Time (ms)

Block size	Columbia Iterative	KISS	Princeton Iterative	Princeton Recursive
256	0.0219 ± 0.0022	0.0197 ± 0.0016	0.0849 ± 0.0339	0.3071 ± 0.0051
512	0.0360 ± 0.0006	0.0487 ± 0.0029	0.1302 ± 0.0037	0.6434 ± 0.0076
1024	0.0905 ± 0.0076	0.0792 ± 0.0027	0.2542 ± 0.0051	1.3612 ± 0.0108
2048	0.2587 ± 0.0237	0.1994 ± 0.0053	0.5796 ± 0.0100	2.9344 ± 0.0178
4096	0.8491 ± 0.0335	0.4912 ± 0.0327	1.1845 ± 0.0241	6.3011 ± 0.1392

Figure 4.6: Java Line graph MEDIUM

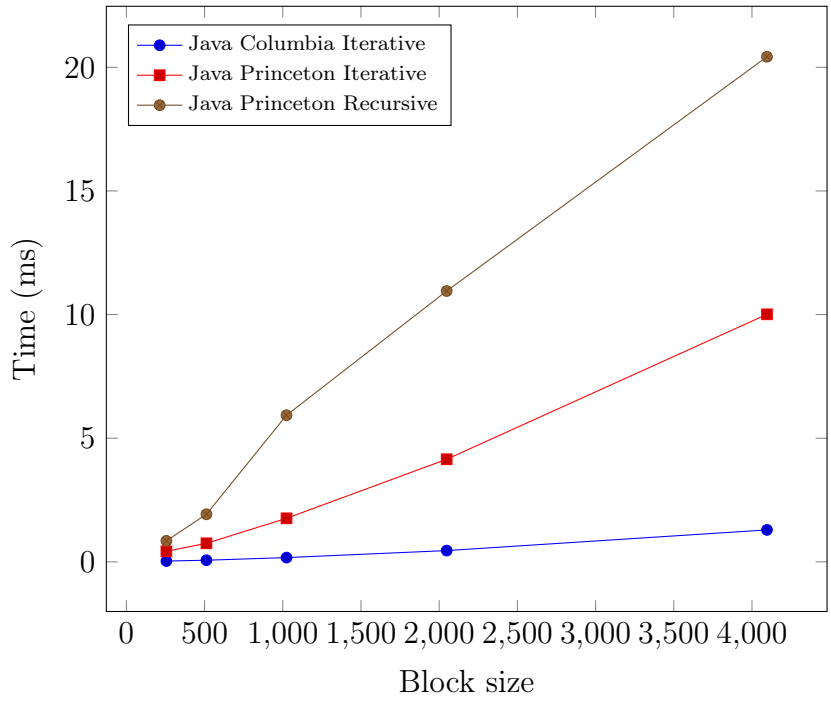


Figure 4.7: Line graph for all algorithms, Large block sizes, Time (ms)

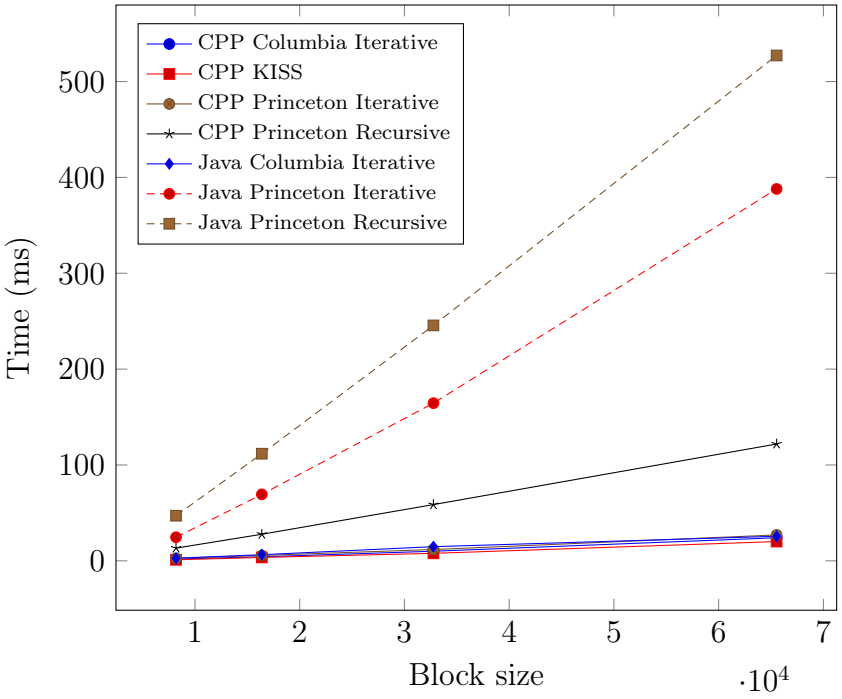


Figure 4.8: CPP Line graph LARGE

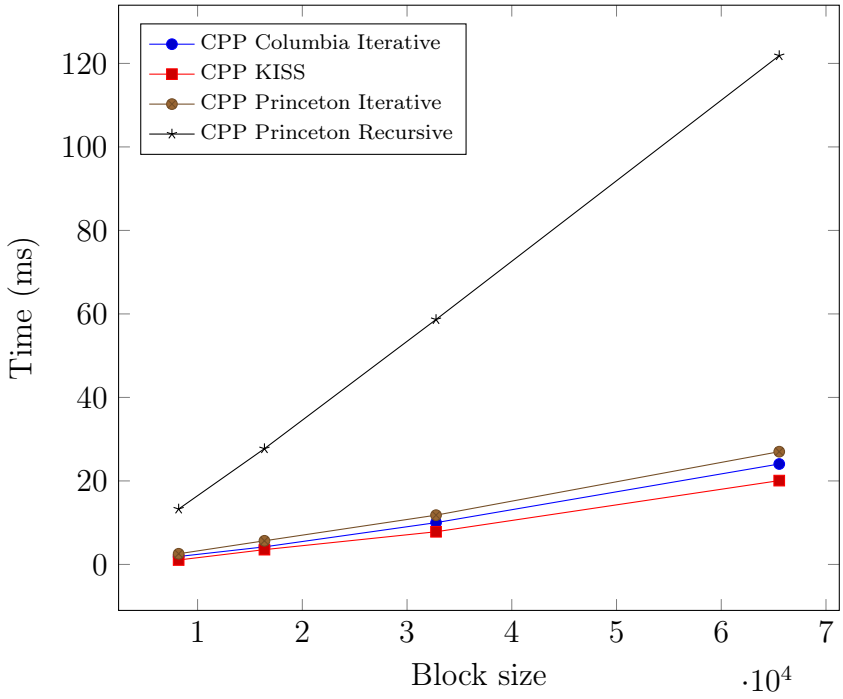


Table 4.6: Large block sizes C++ execution times, Time (ms)

Block size	Columbia Iterative	KISS	Princeton Iterative	Princeton Recursive
8192	1.8974 ± 0.0792	1.0646 ± 0.0319	2.5542 ± 0.0216	13.2709 ± 0.0907
16384	4.2062 ± 0.1739	3.5426 ± 0.2918	5.6595 ± 0.2360	27.7240 ± 0.2127
32768	10.0062 ± 0.6399	7.8248 ± 0.5188	11.8010 ± 0.6380	58.6556 ± 0.2875
65536	24.0547 ± 1.9218	20.0757 ± 1.2195	27.0050 ± 1.6244	121.8623 ± 0.3583

Table 4.7: Large block sizes Java execution times, Time (ms)

Block size	Columbia Iterative	KISS	Princeton Iterative	Princeton Recursive
8192	1.8974 ± 0.0792	1.0646 ± 0.0319	2.5542 ± 0.0216	13.2709 ± 0.0907
16384	4.2062 ± 0.1739	3.5426 ± 0.2918	5.6595 ± 0.2360	27.7240 ± 0.2127
32768	10.0062 ± 0.6399	7.8248 ± 0.5188	11.8010 ± 0.6380	58.6556 ± 0.2875
65536	24.0547 ± 1.9218	20.0757 ± 1.2195	27.0050 ± 1.6244	121.8623 ± 0.3583

Figure 4.9: Java Line graph LARGE

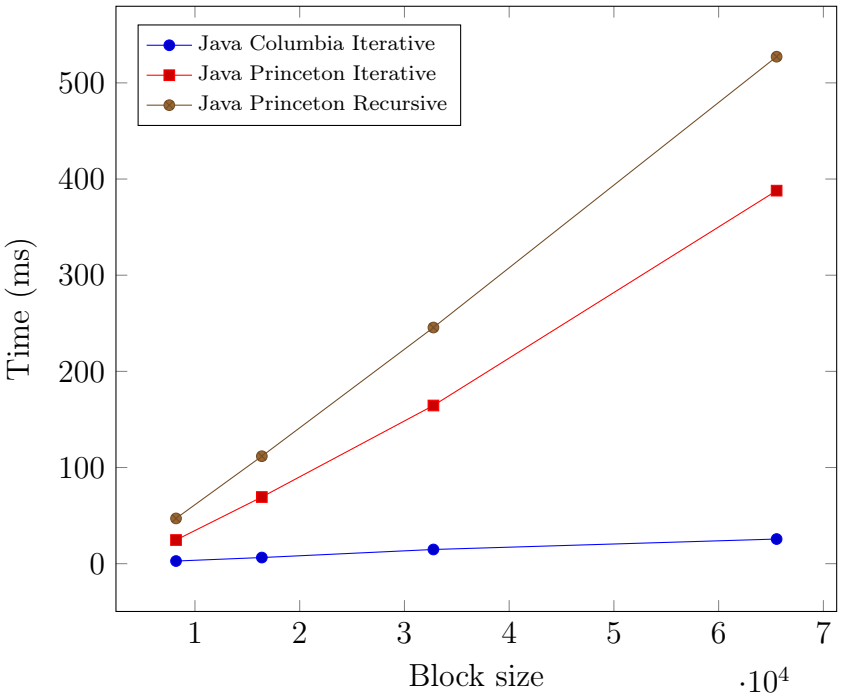


Table 4.8: Results from the Java FFT tests, Time (ns)

Block size	Columbia Iterative	Princeton Iterative	Princeton Recursive
8192	2.7505 ± 0.1029	24.6023 ± 2.4214	47.0631 ± 3.9029
16384	6.3863 ± 0.3338	69.3125 ± 3.6074	111.7487 ± 3.1591
32768	14.7634 ± 1.1282	164.4980 ± 4.4370	245.5791 ± 8.1252
65536	25.6225 ± 1.3401	387.9258 ± 4.8208	527.2174 ± 12.2978
131072	83.8468 ± 2.0645	886.4821 ± 9.4995	1135.4969 ± 25.3095
262144	298.4591 ± 16.3017	2141.9794 ± 60.1126	2775.4740 ± 87.5742

Table 4.9: Results from the CPP FFT tests, Time (ns)

Block size	Columbia Iterative	KISS	Princeton Iterative	Princeton Recursive
8192	1.8974 ± 0.0792	1.0646 ± 0.0319	2.5542 ± 0.0216	13.2709 ± 0.0907
16384	4.2062 ± 0.1739	3.5426 ± 0.2918	5.6595 ± 0.2360	27.7240 ± 0.2127
32768	10.0062 ± 0.6399	7.8248 ± 0.5188	11.8010 ± 0.6380	58.6556 ± 0.2875
65536	24.0547 ± 1.9218	20.0757 ± 1.2195	27.0050 ± 1.6244	121.8623 ± 0.3583
131072	77.2874 ± 2.2952	50.9192 ± 2.3253	77.2266 ± 2.6427	255.9546 ± 1.4665
262144	248.3112 ± 6.5313	96.8293 ± 4.5452	240.8399 ± 6.6391	544.0358 ± 2.3896

Figure 4.10: NEON line graph EXTRA

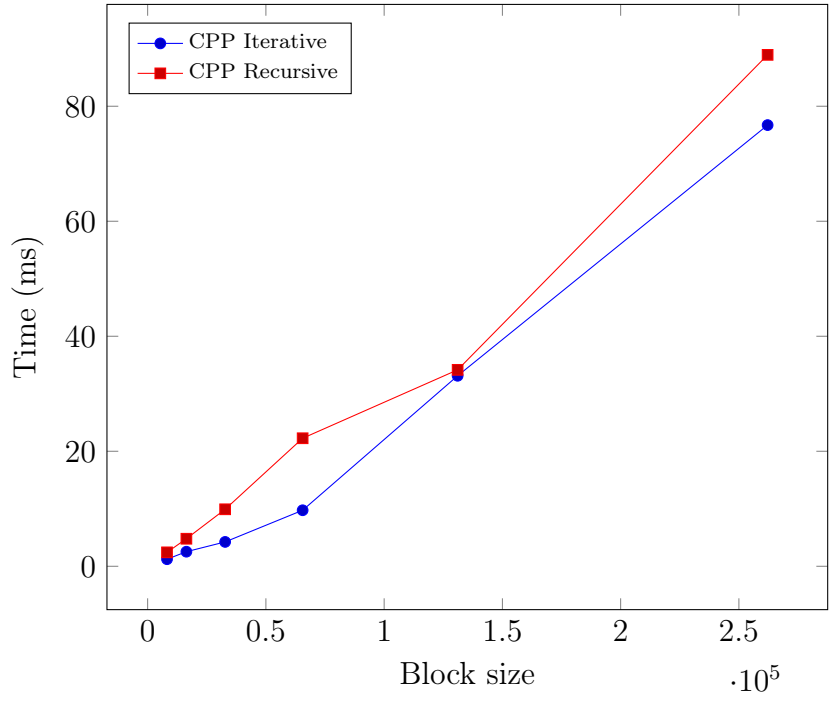
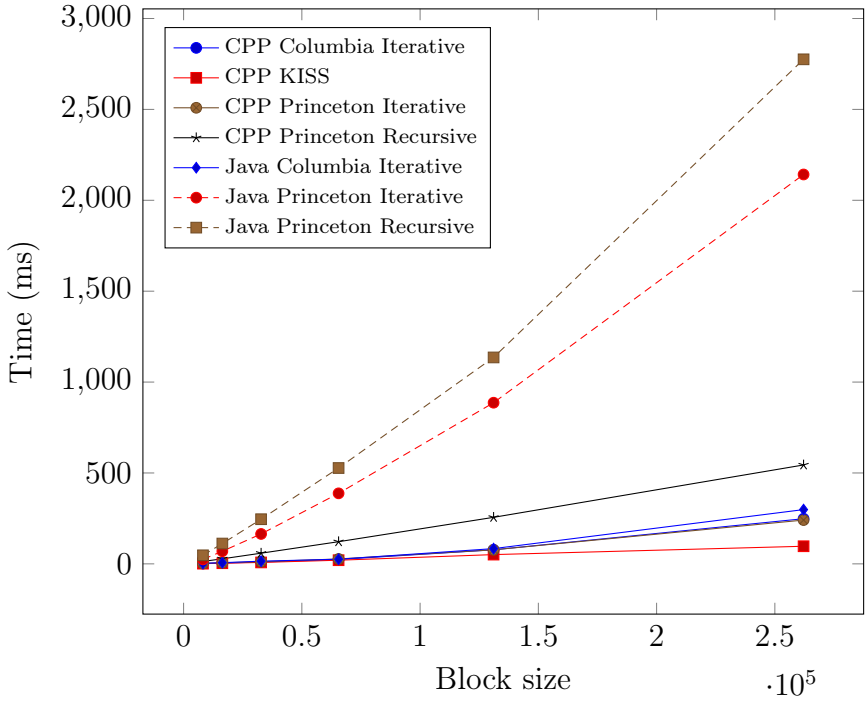


Figure 4.11: Line graph EXTRA



CHAPTER 5

Discussion

Describing text

5.1 JNI Overhead

5.2 Simplicity vs Efficiency

5.3 Vectorization as Optimization

CHAPTER 6

Conclusion

Describing

Appendix A

Source code

Listing A.1: Complex.java [40]

```
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 {
    private final double re;   // the real part
    private final 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 [37]

```
#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 [41]

```
#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 Raw Data

Bibliography

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

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

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

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

[5] 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].

[6] 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.

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

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

[9] Google, "Platform Architecture." https://developer.android.com/guide/

platform/index.html. [Accessed: 30 January 2017].

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

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

[12] 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.

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

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

[15] 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.

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

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

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

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

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

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

[22] 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].

[23] 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].

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

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

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

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

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

[29] 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.

[30] 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.

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

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

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

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

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

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

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

[38] 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].

[39] 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].

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

[41] 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].