KTH – Royal Institute of Technology

Master's Thesis in Computer Science

Comparing Android Runtime with native: Fast Fourier Transform on Android

André Danielsson

May 8, 2017

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

Abstract

This thesis investigates the performance differences between Java code compiled by Android Runtime and C++ code compiled by Clang on Android. For testing the differences, the Fast Fourier Transform (FFT) algorithm was chosen to demonstrate examples of when it is relevant to have high performance computing on a mobile device. Different aspects that could affect the execution time of a program were examined. One test measured the overhead related to the Java Native Interface (JNI). The results showed that the overhead was insignificant for FFT sizes larger than 64. Another test compared matching implementations between Java and native code. The conclusion drawn from this test was that, of the converted algorithms, Columbia Iterative FFT performed the best in both Java and C++. Vectorization proved to be an efficient option for native optimization. Finally, tests examining the effect of using single-point precision (`float`) versus double-point precision (`double`) data types were covered. Choosing `float` could improve performance using cache in an efficient manner.

Keywords: *Android, NDK, Dalvik, Java, C++, Native, Android Runtime, Clang, Java Native Interface, Digital Signal Processing, Fast Fourier Transform, Vectorization, NEON, Performance Evaluation*

Sammanfattning

I denna studie undersöktes prestandaskillnader mellan Java-kod kompilerad av Android Runtime och C++-kod kompilerad av Clang på Android. En Fast Fourier Transform (FFT) användes under experimenten för att visa vilka användningsområden som kräver hög prestanda på en mobil enhet. Olika påverkande aspekter vid användningen av en FFT undersöktes. Ett test undersökte hur mycket påverkan Java Native Interface (JNI) hade på ett program i helhet. Resultaten från dessa tester visade att påverkan inte var signifikant för FFT-storlekar större än 64. Ett annat test undersökte prestandaskillnader mellan FFT-algoritmer översatta från Java till C++. Slutsatsen kring dessa tester var att av de översatta algoritmerna var Columbia Iterative FFT den som presterade bäst, både i Java och i C++. Vektorisering visade sig vara en effektiv optimeringsteknik för arkitekturspecifik kod skriven i C++. Slutligen utfördes tester som undersökte prestandaskillnader mellan flyttalsprecision för datatyperna `float` och `double`. `float` kunde förbättra prestandan genom att på ett effektivt sätt utnyttja processorns cache.

Glossary

ABI	*Application Binary Interfaces*
AOT	*Ahead-Of-Time*
API	*Application Programming Interface*
APK	*Android Package*
ART	*Android Runtime*
Android	Mobile operating system
Apps	*Applications*
CMake	Build tool used by the NDK
Clang	Compiler used by the NDK
DEX	*Dalvik Executable*
DFT	*Discrete Fourier Transform* — Converts signal from time domain to frequency domain
DVM	*Dalvik Virtual Machine* — Virtual machine designed for Android
FFTW	*Fastest Fourier Transform in the West*
FFT	*Fast Fourier Transform* — Algorithm that implements the Discrete Fourier Transform
FPS	*Frames Per second*
HAL	*Hardware Abstraction Layer*
IOS	Mobile operating system
JIT	*Just-In-Time*
JNI	*Java Native Interface* — Framework that helps Java interact with native code
JVM	*Java Virtual Machine*
LLVM	*Low Level Virtual Machine* — collection of compilers
NDK	*Native Development Kit* — used to write android applications in C or C++
NEON	Tool that allows the use of vector instruction for the ARMv7 architecture
SDK	*Software Development Kit*
SIMD	*Single Instruction Multiple Data* — Operations that can be executed for multiple operands
SSL	*Secure Sockets Layer*
Static Library	Code compiled for a specific architecture

Contents

1 Introduction **1**
- 1.1 Background . . . 1
- 1.2 Problem . . . 2
- 1.3 Purpose . . . 2
- 1.4 Goal . . . 2
- 1.5 Procedure . . . 3
- 1.6 Delimitations . . . 3
- 1.7 Limitations . . . 4
- 1.8 Ethics and Sustainability . . . 4
- 1.9 Outline . . . 4

2 Background **5**
- 2.1 Android SDK . . . 5
- 2.2 Dalvik Virtual Machine . . . 6
- 2.3 Android Runtime . . . 7
- 2.4 Native Development Kit . . . 7
  - 2.4.1 Java Native Interface . . . 8
  - 2.4.2 LLVM and Clang . . . 9
- 2.5 Code Optimization . . . 9
  - 2.5.1 Java . . . 11
  - 2.5.2 C++ . . . 11
- 2.6 Discrete Fourier Transform . . . 12
- 2.7 Fast Fourier Transform . . . 13
- 2.8 Related work . . . 17

3 Method **18**
- 3.1 Experiment model . . . 18
  - 3.1.1 Hardware . . . 19
  - 3.1.2 Benchmark Environment . . . 19
  - 3.1.3 Time measurement . . . 20
- 3.2 Evaluation . . . 20
  - 3.2.1 Data representation . . . 21
  - 3.2.2 Sources of error . . . 21
  - 3.2.3 Statistical significance . . . 22
- 3.3 JNI Tests . . . 22
- 3.4 Fast Fourier Transform Algorithms . . . 24
  - 3.4.1 Java Libraries . . . 24
  - 3.4.2 C++ Libraries . . . 24

3.5 NEON Optimization 25

4 Results **26**
4.1 JNI 26
4.2 FFT Libraries 27
4.2.1 Small block sizes 27
4.2.2 Medium block sizes 29
4.2.3 Large block sizes 31
4.3 Optimizations 34
4.4 Garbage Collection 37

5 Discussion **38**
5.1 JNI Overhead 38
5.2 Simplicity and Efficiency 38
5.3 Vectorization as Optimization 40
5.4 Floats and Doubles 41

6 Conclusion **42**

A Source code **47**

B Results **48**
B.1 Data 48
B.1.1 Double Tables 52
B.1.2 Float Tables 53
B.2 Raw 54

List of Figures

1.1 Expression used to filter out relevant articles 3

2.1 Android SDK Software Stack 6
2.2 Native method declaration to implementation. 8
2.3 Loop unrolling in C 9
2.4 Loop unrolling in assembly 10
2.5 Optimized loop unrolling in assembly 10
2.6 Constant Propagation 11
2.7 Loop Tiling 11
2.8 Single Instruction Multiple Data 12
2.9 Time domain and frequency domain of a signal 13
2.10 Butterfly update for 8 values [1] 15
2.11 Butterfly update [1] 15

3.1 Timer placements for tests 20
3.2 JNI test function with no parameters and no return value 23
3.3 JNI test function with a double array as input parameter and return value 23
3.4 Get and release elements 23
3.5 JNI overhead for Columbia FFT 23

4.1 Line graph for all algorithms, *small* block sizes 28
4.2 Java line graph for *small* block sizes with standard deviation error bars . . 28
4.3 C++ line graph for *small* block sizes with standard deviation error bars . . 29
4.4 Line graph for all algorithms, *medium* block sizes 30
4.5 Java line graph for *medium* block sizes with standard deviation error bars . 30
4.6 C++ line graph for *medium* block sizes with standard deviation error bars 31
4.7 Line graph for all algorithms, *large* block sizes 32
4.8 Java line graph for *large* block sizes with standard deviation error bars . . 33
4.9 C++ line graph for *large* block sizes with standard deviation error bars . . 34
4.10 NEON results table for *extra large* block sizes, Time (ms) 36

B.1 Raw results from the Convert JNI test with block size 1024 54

List of Tables

2.1 Bit reversal conversion table for input size 8 15

3.1 Hardware used in the experiments 19
3.2 Software used in the experiments 19

4.1 Results from the JNI tests, Time (µs) 27
4.2 Java results table for *small* block sizes, Time (ms) 28
4.3 C++ results table for *small* block sizes, Time (ms) 29
4.4 Java results table for *medium* block sizes, Time (ms) 31
4.5 C++ results table for *medium* block sizes, Time (ms) 32
4.6 Java results table for *large* block sizes, Time (ms) 33
4.7 C++ results table for *large* block sizes, Time (ms) 33
4.8 NEON `float` results table for *extra large* block sizes, Time (ms) 34
4.9 Java `float` results table for *extra large* block sizes, Time (ms) 35
4.10 Java `double` results table for *extra large* block sizes, Time (ms) 35
4.11 C++ `float` results table for *extra large* block sizes, Time (ms) 35
4.12 C++ `double` results table for *extra large* block sizes, Time (ms) 36
4.13 Pauses due to garbage collection 37
4.14 Block size where each algorithm started to trigger garbage collection 37

B.1 Data for Java Princeton Iterative, Time (ms) 48
B.2 Data for Java Princeton Recursive, Time (ms) 48
B.3 Data for C++ Princeton Iterative, Time (ms) 48
B.4 Data for C++ Princeton Recursive, Time (ms) 48
B.5 Data for Java Columbia Iterative, Time (ms) 49
B.6 Data for C++ Columbia Iterative, Time (ms) 49
B.7 Data for C++ NEON Iterative, Time (ms) 49
B.8 Data for C++ NEON Recursive, Time (ms) 49
B.9 Data for C++ KISS, Time (ms) 50
B.10 Data for JNI No Params, Time (µs) 50
B.11 Data for JNI Vector, Time (µs) 50
B.12 Data for JNI Convert, Time (µs) 51
B.13 Data for JNI Columbia, Time (µs) 51
B.14 Common table for JNI tests, Time (µs) 51
B.15 Common table for `double` C++ FFT tests, Time (ms) 52
B.16 Common table for `double` Java tests, Time (ms) 52
B.17 Common table for `float` Java tests, Time (ms) 53
B.18 Common table for `float` C++ tests, Time (ms) 53
B.19 Common table for `float` NEON tests, Time (ms) 54

Chapter 1

Introduction

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

1.1 Background

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

Android applications are mainly written in Java to ensure portability in form of architecture independence. By using a virtual machine to run a Java app, you can use the same bytecode on multiple platforms. To ensure efficiency on low resources devices, a virtual machine called Dalvik was developed. Applications (apps) on Android have been run on the Dalvik Virtual Machine (DVM) up until Android version 5 [5] in November of 2014 [6]. Since then, Dalvik has been replaced by Android Runtime. Android Runtime, ART for short, differs from Dalvik in that it uses Ahead-Of-Time (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 DVM to work on devices running ART.

To allow developers to reuse libraries written in C or C++ or to write low level code, a tool called Native Development Kit (NDK) was released. It was first released in June 2009 [7] and has since gotten improvements such as new build tools, compiler versions and support for additional Application Binary Interfaces (ABI). ABIs are mechanisms that are used to allow binaries to communicate using specified rules. With the NDK, 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) using the NDK tools instead of using ART. Multiple implementations of FFTs were evaluated as well as the effects of the Java Native Interface (JNI), a framework for communicating between Java code and native static libraries. The following research question was formed on the basis of these topics:

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 will be a gain in writing a part of an Android application in C++. One purpose of this study is to educate the reader about the cost, in performance and effort, of porting parts of an app to native code using the Native Development Kit (NDK). Another is to explore the topic of performance differences between Android Runtime (ART) and native code compiled by Clang/LLVM. Because ART is relatively new (Nov 2014) [6], this study would contribute with more information about to the performance of ART and how it performs compared to native code compiled by the NDK. The results of the study can also be used to value the decision of implementing a given algorithm 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 a part of an application in native code is to potentially get better execution times on 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 app functionality to become platform independent. Component code can then be shared with a computer program and an iOS app.

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 a program has an impact on 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, additional literature could be found. Figure 1.1 contains an 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

This is a strictly quantitative study, meaning numerical data and its statistical significance was the basis for the discussion. 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.

Four 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 test, two comparable optimized implementations of FFTs were chosen, one recursive and one iterative in C++. These implementations were optimized using NEON, a vectorization library for the ARM architecture. In the fourth and final test, the `float` and `double` data types were 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 digital signal analysis which is useful in many mobile applications. 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 this thesis being too broad. 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 presented and chosen.
- ***Chapter 4 - Results*** – The results of the experiments are presented here.
- ***Chapter 5 - Discussion*** – Discussion regarding the results as well as the chosen method.
- ***Chapter 6 - Conclusion*** – Presents what the experiments showed and future work.

Chapter 2

Background

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

2.1 Android SDK

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

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

The Java API Framework is the Java library you use when controlling the Android UI. It is the reusable code for managing activities, implementing data structures and designing the application. The System Application layer represents the functionality that allows a third-party app to communicate with other apps. Example of usable applications are email, calendar and contacts [10].

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

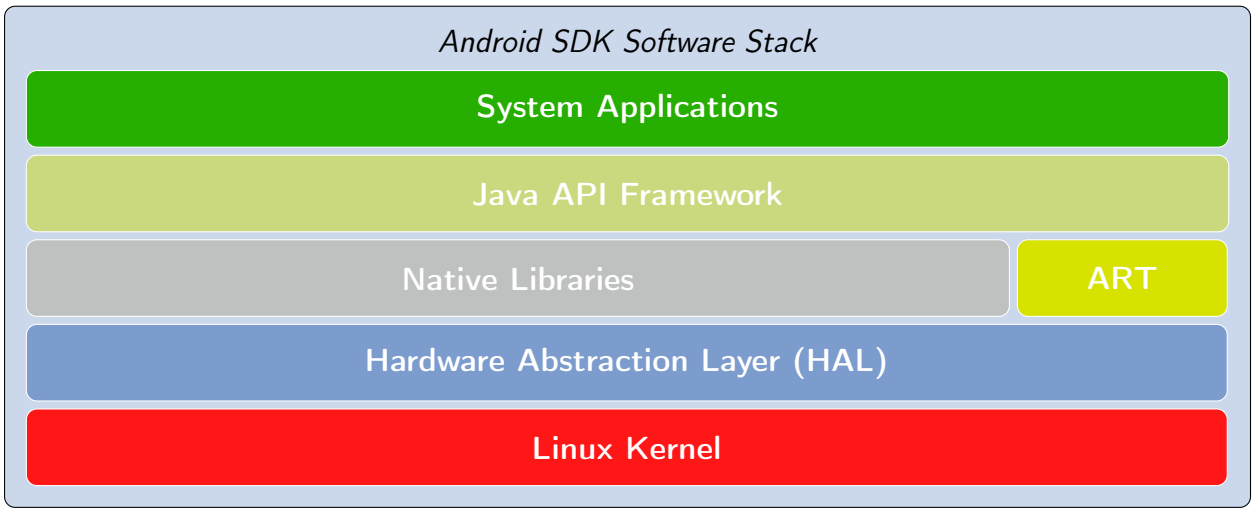


Figure 2.1: Android SDK Software Stack [10]

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 portable compiled code. This way, every device, independent on architecture, with a JVM installed will be able to run the same code. The Android operating system is designed to be installed on many different devices [3]. Compiling to machine code for the targeted devices could become impractical because a program must be compiled against all possible platforms it should work on. For this reason, Java bytecode is a sensible choice when wanting to distribute compiled applications.

The Dalvik Virtual Machine (DVM) is the VM initially used on Android. One difference between DVM and JVM is that the DVM uses a register-based architecture while the JVM uses a stack-based architecture. The most common virtual machine architecture is the stack-based [11, p. 158]. A stack-based architecture evaluates each expression directly on the stack and always has the last evaluated value on top of the stack. Thus, only a stack pointer is needed to find the next instruction on the stack.

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

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

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

2.3 Android Runtime

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

When an app is installed on the device, a program called **dex2oat** converts a DEX-file to an executable file called an oat-file [15]. This oat-file is in the Executable and Linkable Format (ELF) and can be seen as a wrapper of multiple DEX-files [16]. An improvement made in Android Runtime is the optimized garbage collector. Changes include a decrease from two to one GC pause, reduced memory fragmentation (reduces calls to `GC_FOR_ALLOC`) and parallelization techniques to lower the time it takes to collect garbage [15]. There are two common garbage collects plans, Sticky Concurrent Mark Sweep (Sticky CMS) and Partial Concurrent Mark Sweep (Partial CMS). Sticky CMS does not move data and does only reclaim data that has been allocated since the last garbage collect [17]. Partial CMS frees from the active heap of the process [18, p. 122].

2.4 Native Development Kit

Native Development Kit (NDK) is a set of tools to help write 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 [19]. Because Java is the supported language on Android, due to security and stability, native development is not recommended to use when building full apps, with an exception of 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 [20]. 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 (shown below) where the compiled 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 static library 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 [21, p. 22]. You typically use the JNI functions to fetch data 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++ [21, 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 [21, 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 [21, 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.

```
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

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 [22].

Clang is as of March 2016 (NDK version 11) [23], 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 by 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 having 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 while 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.

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

```
$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

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 requested in C and C++ by using the `inline` keyword and can also be optimized by the compiler automatically.

```
$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 during runtime [24, 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

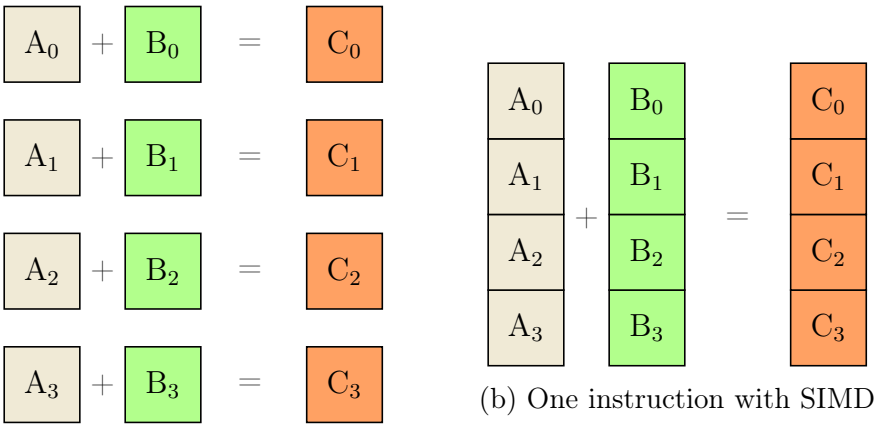
In Java, an array is created during runtime and cannot change its size after it is created. This means that it will always be placed on the heap and the garbage collector will handle the memory it resides on when it is no longer needed. By keeping an array reference in scope and reusing the same array, 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 although this is not always possible or memory efficient.

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



(a) Four separate instructions

(b) One instruction with SIMD

Figure 2.8: Single Instruction Multiple Data [26]

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 a 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 is because you can have instructions that load blocks of multiple values and operates on these blocks. SIMD starts by reading the data into larger vector registers, operate on these registers and storing the results as blocks [25]. 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. First, 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 be described as a sinusoidal wave with a frequency, amplitude and phase.

If we observe Figure 2.9, we can see how the same signal looks in time domain and 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 [27]:

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 [28, 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.

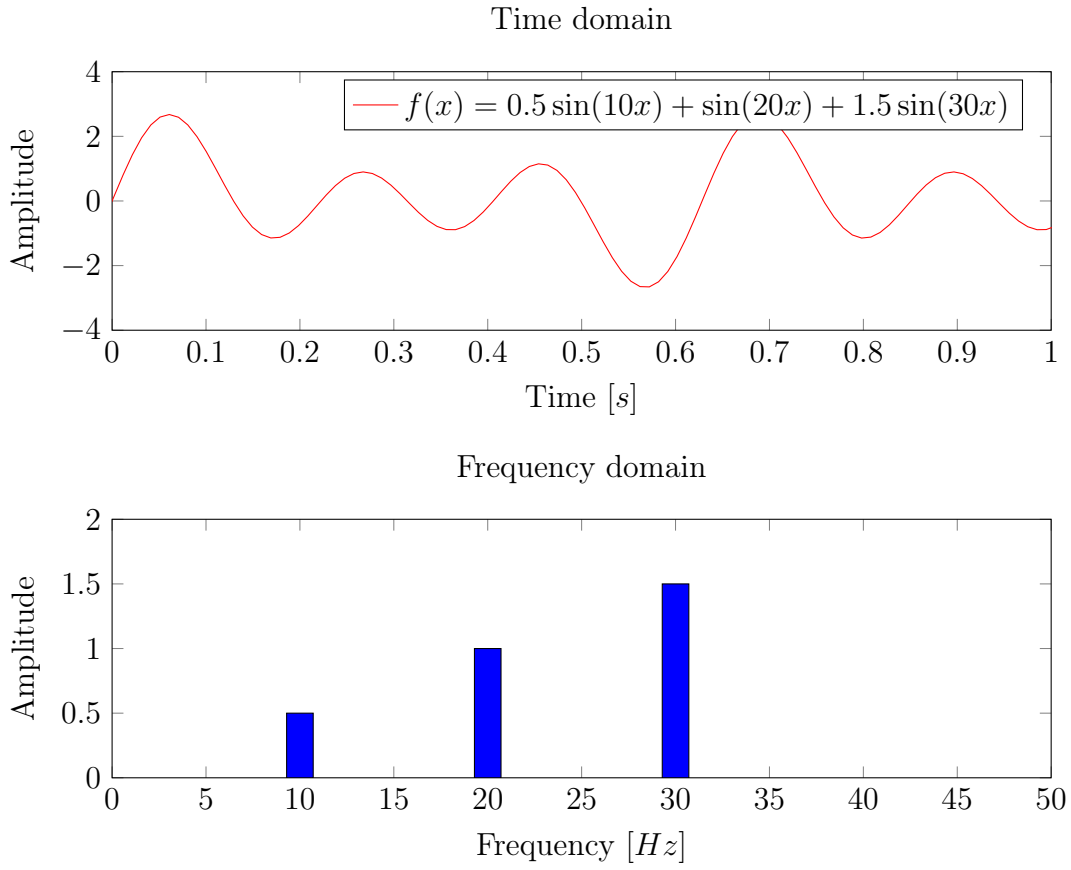


Figure 2.9: Time domain and frequency domain of a signal

Equation 2.1 [29, 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 [30] called the Fast Fourier Transform (FFT).

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

2.7 Fast Fourier Transform

The Fast Fourier Transform algorithm composed by Cooley and Tukey is a recursive algorithm that runs in $O(N \log N)$ time. The following derivation is based on one found in this article from librow [1]. The notation for the imaginary number ($\sqrt{-1}$) was chosen

to be j instead of i for consistency. If we expand the expression in Equation 2.1, presented in Chapter 2.6, for $N = 8$ we get:

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

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

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

In turn, each bracket can be factorized to:

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

And finally simplified to:

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

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

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

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

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

An iterative version of this algorithm would mimic the behaviour of the recursive version described previously. To demonstrate this process, the order of which the recursive implementation operates is visualized in Figure 2.10. One butterfly operation is described in Figure 2.11. With mathematical notation, this relation is described as $x'_a = x_a + x_b\omega_N^k$

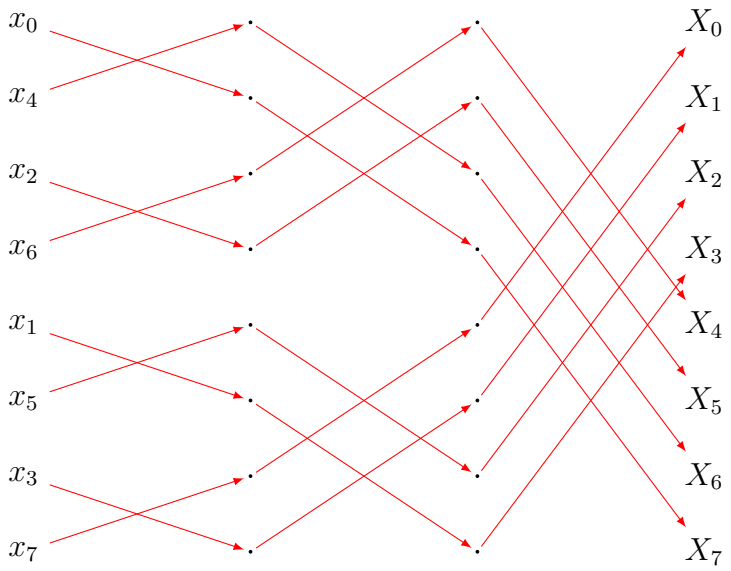


Figure 2.10: Butterfly update for 8 values [1]

Table 2.1: Bit reversal conversion table for input size 8

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

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

When we have achieved this, the operation order must be established. For the first iteration, the size of the gap between the operands is one. The next gap size is two and the third is four. It is now possible to construct an iterative algorithm. This process is shown in pseudocode in Algorithm 1. The fist part of the algorithm is the Bit reversal. This has clearly $O(N)$ time complexity assuming the time complexity of bit_reverse is bounded by the number of bits in an integer. For the butterfly updates, the outer while loop will run for $\log N$ iterations and the two inner loops will run a total of $\frac{step}{2}\frac{N}{step} = \frac{N}{2}$

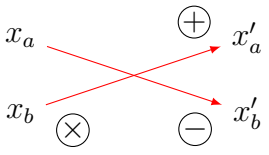


Figure 2.11: Butterfly update [1]

times. It is now clear that the time complexity of this algorithm is $O(N \log N)$.

Algorithm 1: Iterative FFT

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

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

2.8 Related work

A study called *FFT benchmark on Android devices: Java versus JNI* [31] 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. The FFTW library supports multi-threaded computation and this aspect is also covered in the study. Their benchmark application was run on 35 different devices with different Android 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 [32] 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* [33] 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. Their conclusion was that native performed much better than code running on the Dalvik VM. However, code compiled by ART improves greatly from Dalvik and performs almost the same as code compiled by Android NDK.

Chapter 3

Method

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

3.1 Experiment model

In this thesis, different aspects that can affect execution time for an FFT implementation on Android were tested. A link to a repository where the benchmark program, data and algorithms can be found in Appendix A. To get an overview of how much of an impact they have, the following subjects were investigated:

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

The reason it is relevant to know how significant the JNI overhead is, is because we want to see for what data size 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 potentially get faster.

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

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

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

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

3.1.1 Hardware

In this thesis, the hardware was delimited to one device to prevent the experiment from being too extensive and allow a more narrow examination of the test performed. The setup used for performing the experiments is described in Table 3.1.

Table 3.1: Hardware used in the experiments

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

3.1.2 Benchmark Environment

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

Table 3.2: Software used in the experiments

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

```
// 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

3.1.3 Time measurement

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

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

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

3.2 Evaluation

The unit of the resulting data was chosen to be in micro- and milliseconds. Microseconds was in the JNI tests while milliseconds was used for the library and optimization tests. To be able to have 100 executions run in reasonable time, the maximum size of the input data was limited to $2^{18} = 262144$ for all the tests. We need this many executions of the same test to get statistically significant results. The sampling rate is what determines the highest frequency that could be found in the result. The frequency range perceivable by the human ear ($\sim$ 20-22,000 Hz) is covered by the tests. According to the Nyquist

theorem, the sampling rate must be at least twice the upper limit (44,000). Because the FFT is limited to sample sizes of powers of 2, the next power of 2 for a sampling rate of 44,000 is 2^{16}. This size was chosen as the upper limit for the library comparisons.

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

3.2.1 Data representation

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

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

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

3.2.2 Sources of error

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

3.2.3 Statistical significance

Because the execution times differ between runs, it is important to calculate the sample mean and a confidence interval. This way we have an expected value to use in our results as well as being able to say with a chosen certainty that one mean is larger than the other. To get an accurate sample mean, we must have a large sample size. The sample size chosen for the tests in this thesis was 100. The following formula calculates the sample mean [34, p.263]:

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

Now, 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 [34, p. 302]:

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

When comparing results, we need to find a confidence interval for a given test and choose a confidence level. For the data gathered in this study, a 95% two-sided confidence level was chosen when comparing the data. To find the confidence interval we must first find the standard error of the mean using the following formula [34, p. 304]:

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

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

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

Our confidence interval will then be:

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

3.3 JNI Tests

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

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

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

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

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

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

In the third test seen in Figure 3.4, the `GetPrimitiveArrayCritical` function was called to be able to access the elements stored in `arr`. When all the calculations were done, the function would return `arr`. To overwrite the changes made on `elements`, a function called `ReleasePrimitiveArrayCritical` had to be called.

```
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

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

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

Figure 3.5: JNI overhead for Columbia FFT

3.4 Fast Fourier Transform Algorithms

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

- Princeton Recursive [35]
- Princeton Iterative [36]
- Columbia Iterative [37]
- Kiss (*Keep It Simple, Stupid*) FFT [38]

3.4.1 Java Libraries

The Princeton Recursive FFT is a straightforward implementation of the FFT with no radical optimizations. It was implemented in Java by Robert Sedgewick and Kevin Wayne [35]. 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 [36], is an iterative version of the previous FFT (also written in Java). Iterative bit reversal and butterfly operations are used to produce a faster algorithm.

Columbia Iterative [37] uses pre-computed trigonometric tables that are prepared in the class constructor. Because you commonly call FFT for the same sizes in your program, it is beneficial to have the trigonometric tables saved and use them in subsequent calls to the FFT.

3.4.2 C++ Libraries

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

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

3.5 NEON Optimization

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

The first FFT algorithm was a recursive implementation written by Anthony Blake [39]. This algorithm has an initializer function that allocates space for the twiddle factors and calculates them. They are placed in a two dimensional array that utilizes memory locality to waste less memory bandwidth [40]. The converted program is located in the project repo with the following link[1]. The second algorithm was an iterative implementation. This library is a straightforward implementation of FFT with SSE [41] and was written for a sound source localization system [42]. The code that was converted from SSE to NEON is located at this URL[2].

[1] `https://github.com/anddani/thesis/blob/master/BenchmarkApp/app/src/main/cpp/FFTRecursiveNeon.cpp`

[2] `https://github.com/anddani/thesis/blob/master/BenchmarkApp/app/src/main/cpp/FFTIterativeNeon.cpp`

Chapter 4

Results

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

4.1 JNI

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

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

No surprising data regarding the first two tests were found. Neither the **No params** nor **Vector** tests had a clear increase in execution time for an increase in block size. **Vector** did have a higher mean for block size **65536**. On the other hand, we can see that the 95% confidence interval is very large (±3.1960 µs). This is likely due to its high standard deviation of 16.3058 found in Appendix B Table B.11. Likewise, there is a spike in execution time mean for block size **1024** in the **Convert** test.

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

Block size	No params	Vector	Convert	Columbia
16	1.7922 ± 0.1392	1.9333 ± 0.1223	2.6052 ± 0.1004	4.1058 ± 0.3042
32	1.6983 ± 0.0220	2.8130 ± 1.7924	2.6006 ± 0.0370	3.9109 ± 0.0535
64	1.6755 ± 0.0149	1.6344 ± 0.1809	2.6630 ± 0.0425	3.9296 ± 0.0566
128	1.9604 ± 0.4978	1.2349 ± 0.1262	1.9375 ± 0.0843	3.0823 ± 0.0892
256	1.7292 ± 0.0694	1.3276 ± 0.2589	1.8141 ± 0.0276	3.0958 ± 0.0441
512	1.6916 ± 0.0110	1.2567 ± 0.1227	2.2818 ± 0.7011	3.1656 ± 0.0457
1024	2.0228 ± 0.5684	1.3167 ± 0.1341	6.3756 ± 8.4676	3.2896 ± 0.1396
2048	1.7218 ± 0.0288	1.5416 ± 0.1405	1.9099 ± 0.0898	3.4844 ± 0.1113
4096	1.1411 ± 0.0404	1.4010 ± 0.0788	2.0062 ± 0.1562	3.8562 ± 0.3197
8192	1.1105 ± 0.0078	1.4818 ± 0.0759	2.3671 ± 0.1897	3.8474 ± 0.4784
16384	1.1183 ± 0.0280	1.7308 ± 0.1043	2.5833 ± 0.1737	4.9724 ± 0.8955
32768	1.1162 ± 0.0084	2.2099 ± 0.1880	3.2062 ± 0.2029	5.3719 ± 0.2875
65536	1.7463 ± 1.2217	4.7474 ± 3.1960	4.3198 ± 0.2926	6.8136 ± 0.2499
131072	1.1027 ± 0.0141	2.6375 ± 0.1531	5.7004 ± 0.2681	9.6912 ± 1.4337
262144	1.1006 ± 0.0118	3.3172 ± 0.1164	7.4630 ± 0.2309	10.2781 ± 0.2278

4.2 FFT Libraries

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

4.2.1 Small block sizes

Results from the small blocks tests shows a clear difference between the different algorithms. In Figure 4.1, Princeton Recursive in Java perform the worst. Princeton Recursive in C++ and Princeton Iterative in Java perform better than Princeton Recursive Java although worse than the rest of the algorithms. The rest of the algorithms perform similarly and does not seem to grow with larger block sizes.

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

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

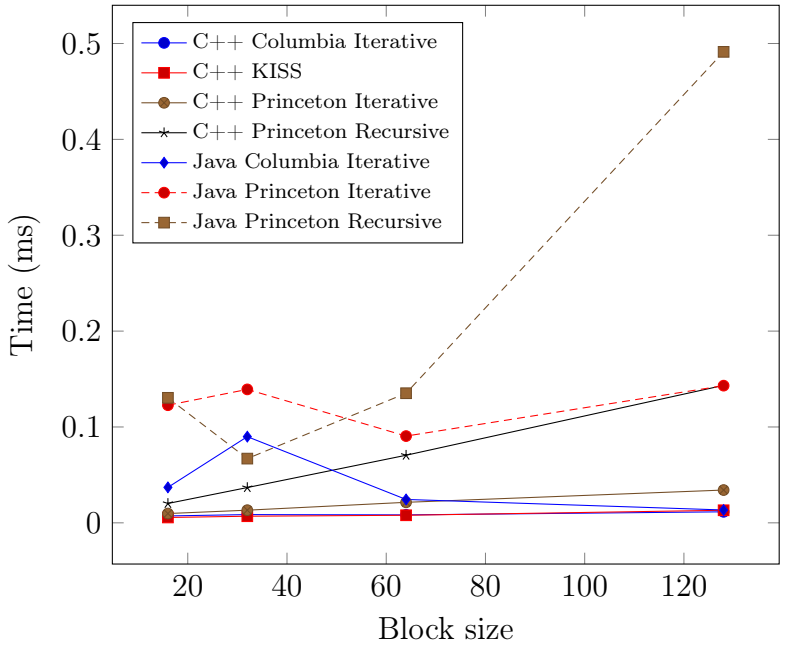


Figure 4.1: Line graph for all algorithms, *small* block sizes

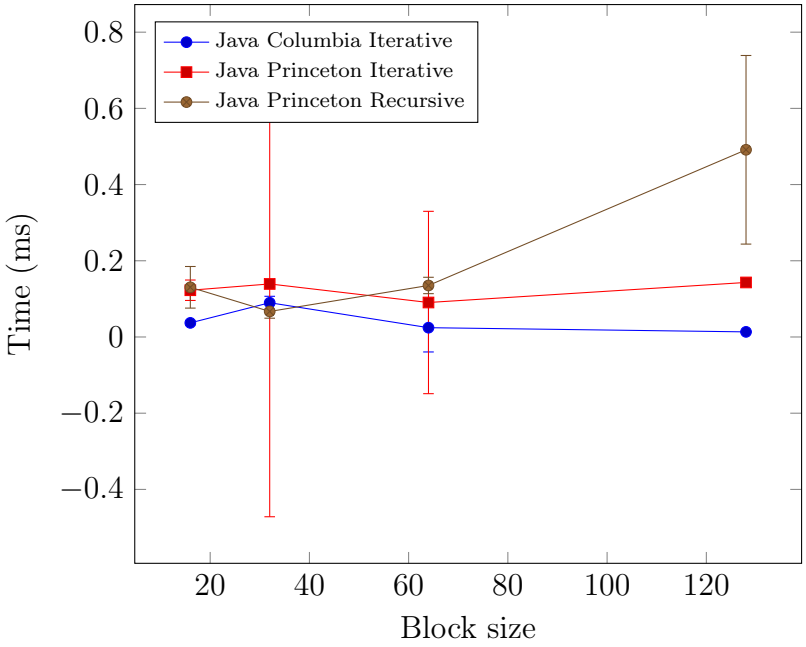


Figure 4.2: Java line graph for *small* block sizes with standard deviation error bars

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

Block size	Columbia Iterative	Princeton Iterative	Princeton Recursive
16	0.0370 ± 0.0010	0.1227 ± 0.0053	0.1304 ± 0.0108
32	0.0899 ± 0.0033	0.1392 ± 0.1198	0.0670 ± 0.0035
64	0.0244 ± 0.0125	0.0905 ± 0.0468	0.1352 ± 0.0043
128	0.0134 ± 0.0002	0.1431 ± 0.0020	0.4913 ± 0.0486

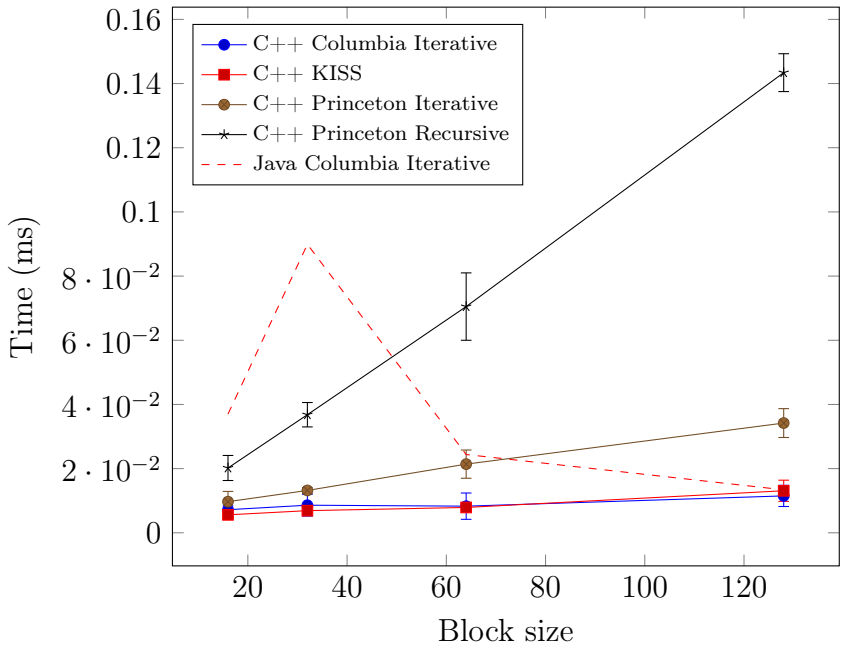


Figure 4.3: C++ line graph for *small* block sizes with standard deviation error bars

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

Block size	Columbia Iterative	KISS	Princeton Iterative	Princeton Recursive
16	0.0072 ± 0.0002	0.0056 ± 0.0002	0.0097 ± 0.0006	0.0202 ± 0.0008
32	0.0086 ± 0.0006	0.0069 ± 0.0002	0.0132 ± 0.0002	0.0368 ± 0.0008
64	0.0083 ± 0.0008	0.0079 ± 0.0002	0.0214 ± 0.0008	0.0705 ± 0.0022
128	0.0115 ± 0.0006	0.0131 ± 0.0006	0.0342 ± 0.0008	0.1434 ± 0.0012

Recursive. If we look at the Java implementation, it has a general decrease in execution time for larger block sizes although it is faster than Princeton Iterative and Recursive.

4.2.2 Medium block sizes

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

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

For the C++ algorithms, it is now apparent which order the algorithms rank regarding performance found in Figure 4.6. Princeton Recursive perform the worst while the rest has similar execution times. It is now clear that KISS performs best, followed by Columbia

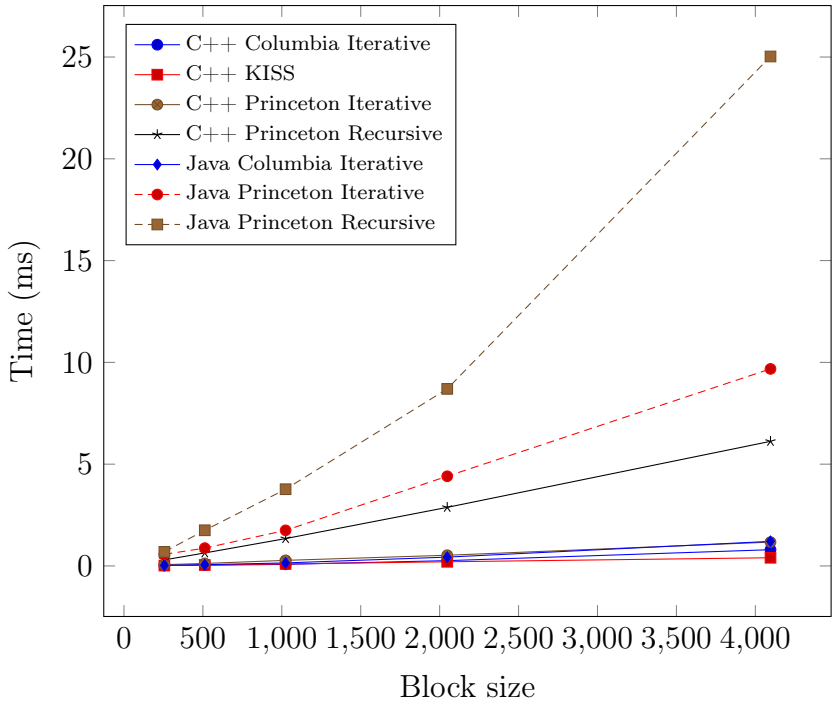


Figure 4.4: Line graph for all algorithms, *medium* block sizes

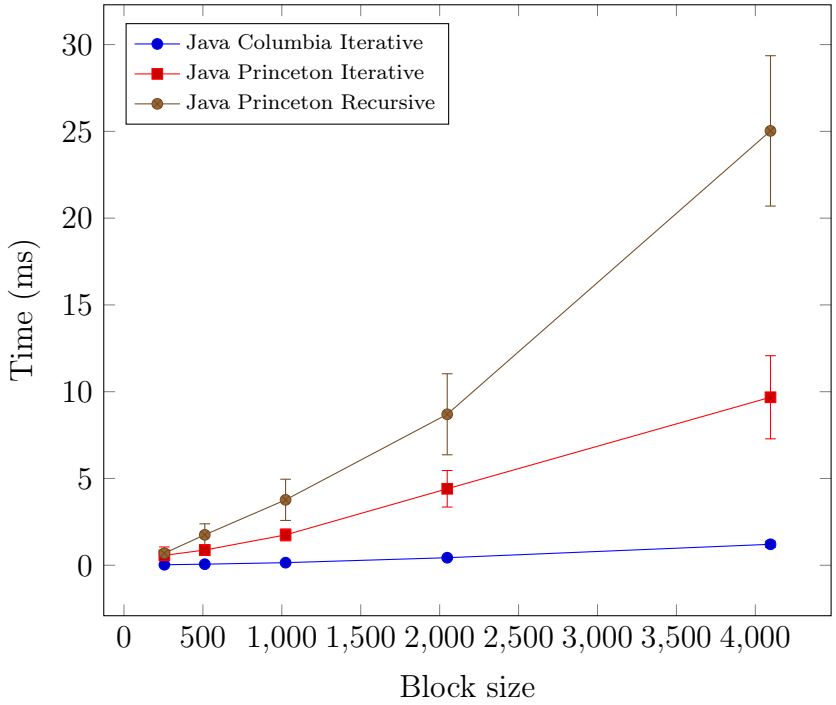


Figure 4.5: Java line graph for *medium* block sizes with standard deviation error bars

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

Block size	Columbia Iterative	Princeton Iterative	Princeton Recursive
256	0.0293 ± 0.0008	0.5698 ± 0.0947	0.6929 ± 0.0325
512	0.0615 ± 0.0010	0.8758 ± 0.0615	1.7515 ± 0.1245
1024	0.1484 ± 0.0061	1.7488 ± 0.0676	3.7688 ± 0.2328
2048	0.4338 ± 0.0159	4.4055 ± 0.2064	8.6983 ± 0.4575
4096	1.2071 ± 0.0355	9.6792 ± 0.4694	25.0276 ± 0.8491

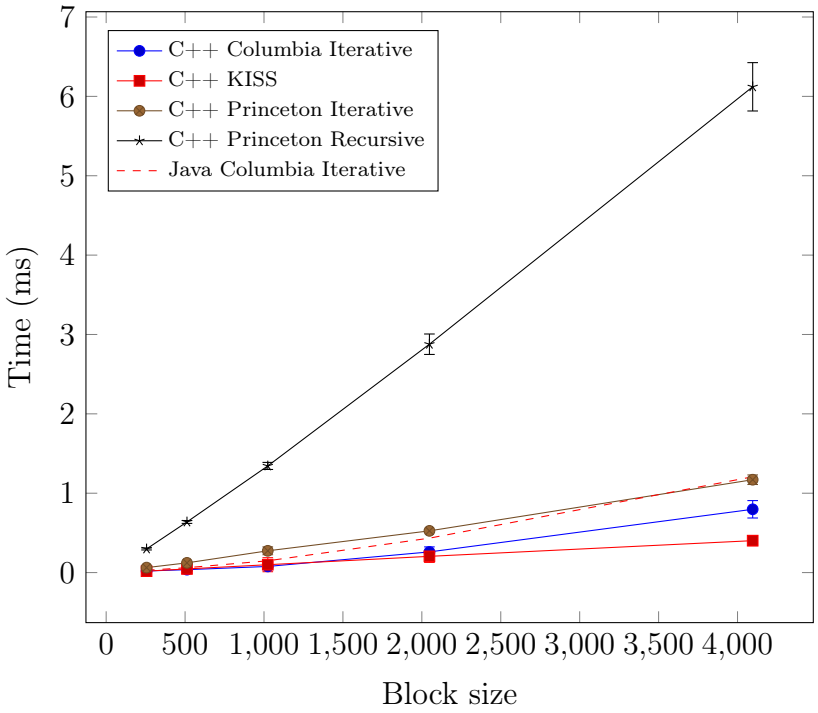


Figure 4.6: C++ line graph for *medium* block sizes with standard deviation error bars

Iterative and then Princeton Iterative. The Java Columbia Iterative implementation proves to be faster than Princeton for block sizes smaller than 4096.

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

4.2.3 Large block sizes

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

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

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

Block size	Columbia Iterative	KISS	Princeton Iterative	Princeton Recursive
256	0.0200 ± 0.0008	0.0182 ± 0.0008	0.0627 ± 0.0008	0.2978 ± 0.0025
512	0.0366 ± 0.0006	0.0461 ± 0.0022	0.1225 ± 0.0006	0.6379 ± 0.0035
1024	0.0773 ± 0.0020	0.0992 ± 0.0172	0.2744 ± 0.0098	1.3437 ± 0.0086
2048	0.2604 ± 0.0123	0.2047 ± 0.0149	0.5257 ± 0.0035	2.8773 ± 0.0253
4096	0.7974 ± 0.0216	0.4022 ± 0.0123	1.1701 ± 0.0118	6.1206 ± 0.0598

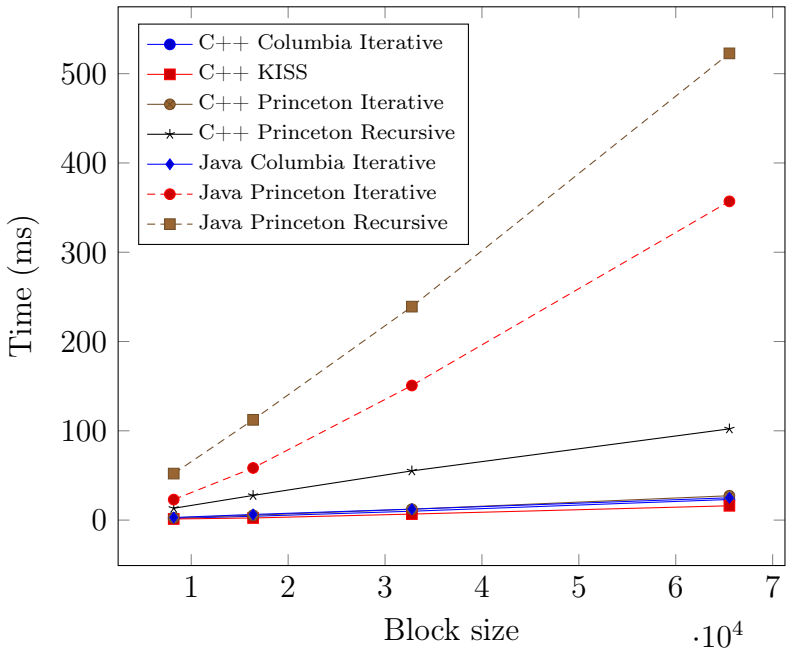


Figure 4.7: Line graph for all algorithms, *large* block sizes

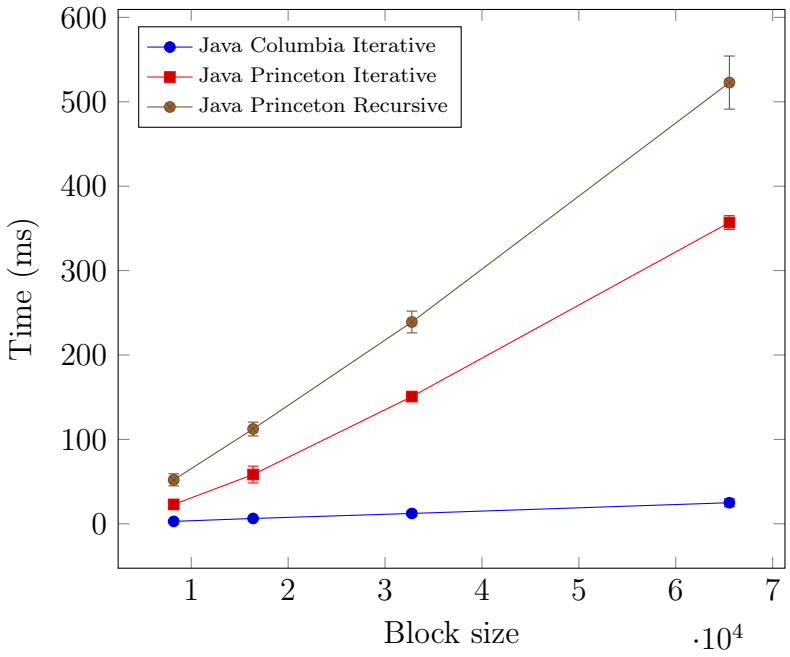


Figure 4.8: Java line graph for *large* block sizes with standard deviation error bars

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

Block size	Columbia Iterative	Princeton Iterative	Princeton Recursive
8192	2.8726 ± 0.0637	22.9609 ± 1.1025	52.0853 ± 1.3973
16384	6.3214 ± 0.2066	58.3825 ± 1.9484	112.3024 ± 1.6050
32768	12.2634 ± 0.5700	150.7299 ± 1.0864	239.0777 ± 2.5276
65536	24.9874 ± 0.9069	356.9871 ± 1.5864	522.7409 ± 6.1660

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

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

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

Block size	Columbia Iterative	KISS	Princeton Iterative	Princeton Recursive
8192	1.9326 ± 0.0519	1.2470 ± 0.0451	2.5845 ± 0.0410	13.2345 ± 0.1433
16384	4.2789 ± 0.1254	2.3713 ± 0.0962	5.4518 ± 0.1149	27.6080 ± 0.1784
32768	9.9388 ± 0.3203	6.7420 ± 0.2534	12.2266 ± 0.4831	55.1227 ± 1.2277
65536	23.1031 ± 0.7648	16.0281 ± 0.3542	27.2805 ± 0.9530	102.1585 ± 3.5262

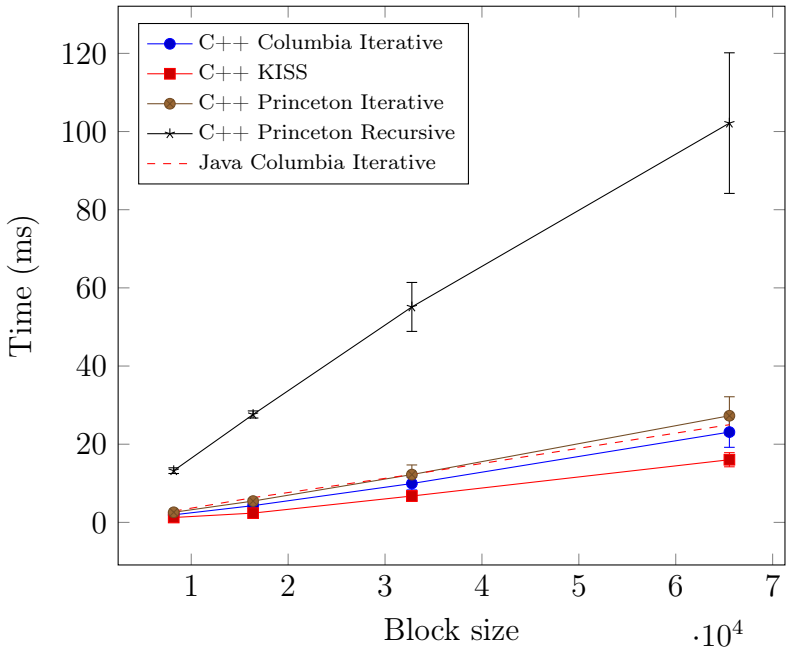


Figure 4.9: C++ line graph for *large* block sizes with standard deviation error bars

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

Block size	Iterative	Recursive
8192	1.0051 ± 0.0129	1.6133 ± 0.0278
16384	2.1559 ± 0.0502	3.5690 ± 0.0857
32768	4.7898 ± 0.1543	7.6009 ± 0.1878
65536	9.8807 ± 0.2452	16.1129 ± 0.3712
131072	27.8158 ± 0.7146	34.1650 ± 0.4722
262144	63.1858 ± 1.2662	74.0746 ± 1.0666

4.3 Optimizations

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

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

When comparing the NEON results from Table 4.8 with the results from the C++ tests found in Table 4.11, it is clear that vectorization as an optimization is beneficial. NEON intrinsics resulted in almost twice as fast execution time in comparison with the normal

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

Block size	Columbia Iterative	Princeton Iterative	Princeton Recursive
8192	2.1766 ± 0.0639	21.1757 ± 0.6689	50.0776 ± 1.4876
16384	3.9995 ± 0.0902	46.8150 ± 1.0327	103.9682 ± 3.0954
32768	8.9846 ± 0.2566	113.1953 ± 3.9572	219.0208 ± 5.5642
65536	20.2833 ± 0.4786	261.9954 ± 9.1987	485.1020 ± 13.3737
131072	47.2950 ± 1.3073	622.4328 ± 24.9022	1039.4937 ± 26.9767
262144	156.7135 ± 1.1934	1728.4640 ± 53.8042	2297.8011 ± 58.2951

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

Block size	Columbia Iterative	Princeton Iterative	Princeton Recursive
8192	2.8726 ± 0.0637	22.9609 ± 1.1025	52.0853 ± 1.3973
16384	6.3214 ± 0.2066	58.3825 ± 1.9484	112.3024 ± 1.6050
32768	12.2634 ± 0.5700	150.7299 ± 1.0864	239.0777 ± 2.5276
65536	24.9874 ± 0.9069	356.9871 ± 1.5864	522.7409 ± 6.1660
131072	85.9483 ± 1.6097	815.8607 ± 3.4304	1144.8802 ± 17.8736
262144	274.5134 ± 5.1129	2108.0771 ± 27.5366	2638.0547 ± 40.5424

C++ tests. It was also much faster than Java, especially compared to Java Princeton Iterative/Recursive.

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

As we can see in Table 4.10, the results show that when using `doubles`, the performance will be worse for Columbia Iterative and Princeton Recursive. We can see that the `float` tests were always running faster than the corresponding `double` test.

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

Block size	Columbia Iterative	Princeton Iterative	Princeton Recursive
8192	1.8030 ± 0.1929	2.0605 ± 0.0110	10.7496 ± 0.0512
16384	2.9629 ± 0.1311	4.5027 ± 0.0341	22.8640 ± 0.1748
32768	8.2847 ± 0.2364	9.6797 ± 0.1288	48.0022 ± 0.2458
65536	18.8158 ± 0.5494	20.3049 ± 0.3467	96.9572 ± 0.2658
131072	43.7807 ± 1.2632	44.5770 ± 0.9577	192.4607 ± 5.9702
262144	134.8093 ± 2.0115	129.5150 ± 1.8201	398.9698 ± 12.9207

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

Block size	Columbia Iterative	KISS	Princeton Iterative	Princeton Recursive
8192	1.9326 ± 0.0519	1.2470 ± 0.0451	2.5845 ± 0.0410	13.2345 ± 0.1433
16384	4.2789 ± 0.1254	2.3713 ± 0.0962	5.4518 ± 0.1149	27.6080 ± 0.1784
32768	9.9388 ± 0.3203	6.7420 ± 0.2534	12.2266 ± 0.4831	55.1227 ± 1.2277
65536	23.1031 ± 0.7648	16.0281 ± 0.3542	27.2805 ± 0.9530	102.1585 ± 3.5262
131072	75.4942 ± 1.6429	41.7620 ± 0.5968	78.9501 ± 1.9300	231.2663 ± 6.7357
262144	243.8496 ± 1.0072	102.1196 ± 1.0817	250.4870 ± 5.2771	494.7038 ± 13.6690

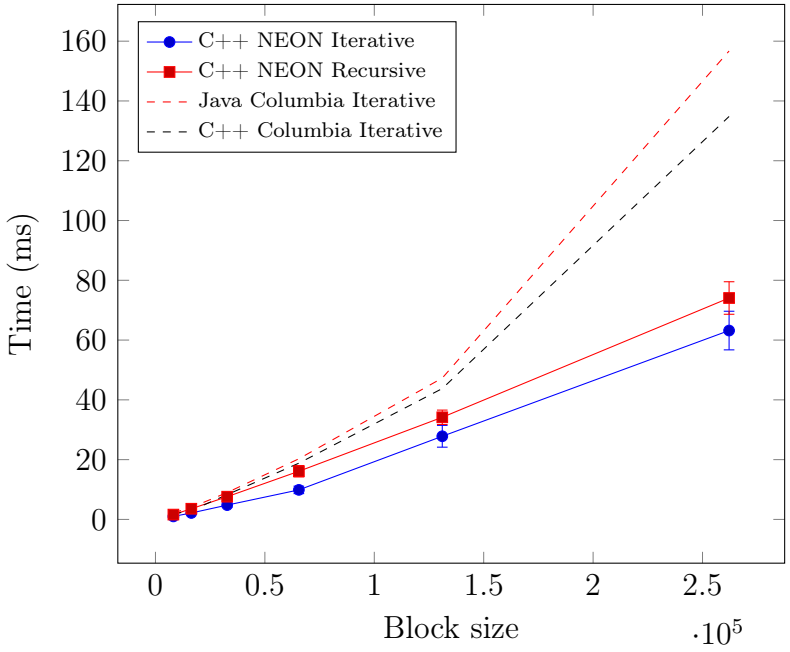


Figure 4.10: NEON results table for *extra large* block sizes, Time (ms)

4.4 Garbage Collection

Table 4.13 shows how many garbage collections that were triggered during the tests. Each row includes all block sizes and 100 iterations for one algorithm. This figure lists the number of partial Concurrent Mark Sweeps, the sum of its garbage collection pauses, the number of sticky Concurrent Mark Sweeps and the sum of its garbage collection pauses.

Table 4.13: Pauses due to garbage collection

Algorithm	# partial	tot. time (ms)	# sticky	tot. time (ms)
Java Columbia Iterative	0	0	0	0
Java Princeton Iterative	477	2,825.52	406	2,959.24
Java Princeton Recursive	240	602.10	397	887.39
Java Float Columbia Iterative	0	0	0	0
Java Float Princeton Iterative	269	1,541.97	334	2,316.53
Java Float Princeton Recursive	167	313.05	27	71.39
C++ Columbia Iterative	0	0	0	0
C++ Princeton Iterative	0	0	0	0
C++ Princeton Recursive	0	0	0	0
C++ Float Columbia Iterative	0	0	0	0
C++ Float Princeton Iterative	0	0	0	0
C++ Float Princeton Recursive	0	0	0	0
KISS	0	0	0	0
NEON Iterative	0	0	0	0
NEON Recursive	0	0	0	0

Table 4.14 shows the block size at which each algorithm first triggered the garbage collector. This data is relevant because we can see if there is a possibility to reduce the risk of triggering the garbage collector for different block sizes. This can also be significant in the discussion about how `float`s and `double`s perform. When the garbage collector is running, we can see that the pauses can be between 0.468-23.760 ms as seen in the raw data[1].

Table 4.14: Block size where each algorithm started to trigger garbage collection

Algorithm	Block Size
Princeton Iterative	8192
Princeton Recursive	4096
Float Princeton Iterative	16384
Float Princeton Recursive	8192

[1] https://github.com/anddani/thesis/tree/master/gc_results

Chapter 5

Discussion

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

5.1 JNI Overhead

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

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

Regarding the spike in mean for some JNI results, this can be a cause of one large execution time skewing the results. This is actually the case for the **Convert** tests with block size 1024 as seen in Figure B.1 found in Appendix B. A reason for this could be that the garbage collector began executing during the timing of the test.

5.2 Simplicity and Efficiency

The slowest algorithm was the Java Princeton Recursive. The reason for this is because it executes many method calls. For each method call, registers must be saved by pushing them onto a stack, function arguments must them be moved to appropriate registers and a jump instruction to the new code must be executed. When the function has finished executing, the registers are popped from the stack. This causes a lot of overhead.

Additionally, each call creates new `Complex` arrays when splitting up the array in odd and even indices. Lastly, when combining the arrays, new `Complex` objects are created each time an operation is done between two complex numbers. The reason for the is because the `Complex` class creates immutable objects. This slows down the process and increases memory consumption, increasing work for the garbage collector.

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

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

Of the algorithms tested in the FFT library tests, KISS FFT was the fastest. It is more optimized than the "basic" implementations found in the Columbia and Princeton algorithms. In C++, Princeton Iterative and Columbia Iterative were the fastest and in Java, Columbia Iterative was the fastest. The reason for Princeton Iterative being faster in C++ is, as for Java Princeton Recursive, because it uses the `Complex` class to represent complex numbers. Because Java Columbia Iterative used double arrays to represent real and imaginary numbers, no new objects needed to be created.

The results from Chapter 4.4 - Garbage Collection show that the Java Princeton algorithms caused the GC to run. This behaviour was as expected because they called new during their execution. Java Columbia did not cause any garbage collection and neither did any of the C++ algorithms.

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

As we have seen in the tests, choosing an iterative implementation is preferable and choosing the correctly implemented iterative implementation is also important. It is possible to compare implementations by setting up small tests. This is a small time investment that ensures that you get the performance you need in a program. It is also possible to read through the implementation, looking for places where it allocates memory and edit it so that it uses an already allocated array. This reduces the number of calls for more memory which lessens the chance of a garbage collect.

Comparing the Java and C++ implementations of the Columbia Iterative FFT, for *small-large* block sizes, the Java version is almost as fast as the C++ version. For the block sizes added in the *extra large* group, the C++ version perform better. If you were to choose between the C++ and the Java variant of Columbia, it is better to choose the Java version if the performance requirement allows it. Avoiding having to implement a JNI bridge between Java and native is more preferable than the small increase in performance.

Another argument why it is sensible to choose the Java version is that is has enough performance for processing to allow graphical rendering in 60 FPS. Let's say we have an application that visualizes sound with a sampling frequency of 44100 Hz and updates the screen at 60 frames per second. It has 16.66 ms to do all the processing needed between the screen updates. If we want to have double precision numbers and use Java it is possible to do an FFT with a size of **32768**.

This size is more than needed and would require $32768/44100 \approx 743$ ms to sample. The fastest Java algorithm for double point precision ran in 12 ms for this block size. As we can see, in relation, the time it takes to execute an FFT is only a small fraction of the time it takes to gather the same number of samples. This is important because it allows larger FFT sizes from the sampled audio, increasing the frequency resolution in the results, while still being sufficiently fast.

For smaller block sizes (*medium* and *large*), common when sampling low frequencies such as speech and acceleration data, it is relevant to have fast transforming for these sizes. Comparing C++ and Java we see that Java is not that much slower, at best, than C++. This gives an incentive to use Java, allowing more consistent code and does not add the complexity of JNI.

When the block sizes are large, a bigger difference is found between the algorithms. One reason for this is the impact the garbage collection has when triggered during the timing of an algorithm. The pauses can range between, depending on the work done in the garbage collect, 0.468-23.760 ms.

KISS FFT was the fastest of all the native implementations. This was because of the optimizations done to improve its performance. Using multiple radix implementations of the FFT, it is possible to get more performance at the cost of higher code complexity. It shows the potential in doing small optimizations that does not rely on multi-core or other architecture dependent optimization techniques.

How memory is used differs between algorithms. As seen in Chapter 4.4, only four tests caused the garbage collector to be run. Java Princeton Iterative and Recursive with both `float` and `double` as primary data types were the algorithms causing this. This is because they create new arrays each time they are called. The garbage collector needed to remove each array that was allocated each call, thus increasing the total work the garbage collector must achieve. It is also possible to see that for the `double` data type, garbage collection are caused earlier than for `float` during the tests.

5.3 Vectorization as Optimization

As the results show, vectorization was an improvement in regards to efficiency. This was as expected because it is possible to process more elements for each instruction. The optimizations implicitly implemented loop unrolling because more operations were covered in the body of the loop, resulting in less jump instructions, leading to less instructions in total.

The Recursive NEON implementation did not perform much worse than the Iterative version. The difference between the implementations were of a smaller factor than between Princeton Iterative and Recursive. One reason for this could be that because the twiddle

factors are moved so that they are closer in cache in a two dimensional array for the recursive version. They are placed in order of access to reduce the number of cache misses. This could be the reason for the performance increase.

One disadvantage of using NEON intrinsics is the fact that you cannot run it on all CPU architectures. Not all Android devices runs on ARM, and not all ARMv7 supports NEON [43]. This makes it less compatible if the program depends on its optimizations. One example of an optimization that is architecture independent and works on all multi-core CPUs is parallelization. Another disadvantage of NEON is that it makes a program harder to maintain. It introduces complexity to the program in addition to JNI.

A possible implementation could be optimized by fitting all the components in cache. This reduces the overhead of fetching data from memory which would slow down the overall performance. If it is not possible for the data to fit in cache, you can rewrite the algorithm such that the elements are stored close by in order of access. This will result in less cache invalidation and less direct memory reads.

When needing to do a lot of processing with the data in frequency domain, a fast FFT will help in lowering the total execution time. If the data is also transformed back into time domain, for example when you want to filter out noise in an audio signal and play it back. Another reason you would want a fast FFT is when the computation time for some validation need to be as fast as possible. An example of this is voice recognition where we want answers in reasonable time. Another example could be to do image recognition on fingerprints for fast biometric authentication.

5.4 Floats and Doubles

One reason for using the `float` data type is that they are stored as 32-bit values instead of 64-bit values as `double`s are stored. This means that the program requires less memory and reduces the risk of triggering the `GC_FOR_ALLOC` due to the lack of free memory distributed to the process. Most modern devices has pipelined floating point operations to optimize calculations [44]. In the case of the experiments performed in this thesis, `float` performed better than `double` for the device used in the tests.

The reason for this is that because `float`s are half the size of `double`s, they have a large chance of fitting in cache. Depending on the cache size, it could be beneficial to use the `float` data type. This is an important point to keep in mind. A reason why you would not use `float`s is when you need the precision of `double`s or when the performance difference is not an issue or non-existent.

The test results showed that there is a big difference when computing floats and doubles for in both Java and C++. It is also the case that some `float` tests in Java execute at half the execution time of the corresponding algorithm for `double` in C++. This shows the importance, on some architectures with different cache sizes, to choose the appropriate data structures as they can impact the performance greatly.

Chapter 6

Conclusion

The conclusions drawn from the discussion will be presented in this chapter as well as an answer to the scientific question of this thesis. Propositions of future work will also be included.

JNI overhead is very small, around 1-10 µs for a Qualcomm MSM8994 Snapdragon 810 chip. If a program were to call native code in a loop, it could add up to some overhead. It is only for very small block sizes around $2^4 - 2^6$ where this overhead becomes a substantial part of the total execution time. For larger and more common block sizes, the overhead from JNI is not significant.

One of the conclusions to draw from the results is that when choosing Java, the best algorithm was the Columbia Iterative. It was the easiest to convert to C++ because it did not include any Java specific constructs (except for method definitions). It was also the fastest between it, Princeton Iterative and Princeton Recursive for both Java and C++. In Columbia Iterative, after the initial setup of calculating the trigonometric tables and allocating space for the input/output array, no memory allocation is done during the actual transform.

To prevent garbage collection pauses throughout the program, it is necessary to pre-allocate data structures that should be used in a function that is called multiple times instead of allocating new memory for each call. By doing this, it reduces the work for the garbage collector. Less allocated memory will go out of scope, making garbage collection pauses shorter and less frequent.

Vectorization was a very efficient optimization which performed much better than their corresponding algorithms in C++. Native optimization is architecture dependent. This means that an application will only work on a subset of Android devices running some architectures the program is compiled for. Further, some optimizations are only possible on a few architectures. This makes some optimizations better than other. An example of a more portable optimization is parallelization.

Operations between elements of the `float` data structure resulted in faster execution time than between `double`s on the Nexus 6P. It is therefore important to know if `float` gives sufficient precision for the given application. If high precision is important, you should use a `double` as primary data type. `float` is useful for architectures where it is faster to compute between floats or when memory efficiency is important.

To answer the research question, based on the results from the experiments, we can see that, of the chosen algorithms, there is not a significant performance difference between the fastest FFT library for Java with its corresponding implementation in native code. Optimization in native code does make native code significantly faster than the fastest Java implementation. This does increase the complexity of the code and decrease the compatibility between devices.

One interesting aspect that could be examined for future work is parallelization. This optimization has more compatibility than the optimization in NEON because it can be used in Java. There are different libraries to compare a possible implementation with such as JTransforms [45].

Bibliography

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[17] Android, "ART GC overview." `https://source.android.com/devices/tech/dalvik/gc-debug`. [Accessed: 11 April 2017].

[18] D. Sillars, *High Performance Android Apps: Improve Ratings with Speed, Optimizations, and Testing*. O'Reilly Media, 2015.

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

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

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

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

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

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

[25] 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].

[26] 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].

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

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

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

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

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

[32] 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.

[33] 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.

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

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

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

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

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

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

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

[41] 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].

[42] 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].

[43] Android, "NEON Support." `https://developer.android.com/ndk/guides/cpu-arm-neon.html`. [Accessed: 12 April 2017].

[44] Android, "Avoid Using Floating-Point." `https://developer.android.com/training/articles/perf-tips.html`. [Accessed: 13 April 2017].

[45] P. Wendykier, "JTransforms - Benchmark." `https://sites.google.com/site/piotrwendykier/software/jtransforms`. [Accessed: 30 January 2017].

Appendix A

Source code

Converted code, raw data and the benchmark program used for the tests can be found in the following repository:

`https://github.com/anddani/thesis`

Appendix B

Results

B.1 Data

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

Block size	$\bar{X}$	s	$SE_{\bar{X}}$	$ME_{\bar{X}}$
16	0.1227	0.0268	0.0027	0.0053
32	0.1392	0.6110	0.0611	0.1198
64	0.0905	0.2393	0.0239	0.0468
128	0.1431	0.0097	0.0010	0.0020
256	0.5698	0.4834	0.0483	0.0947
512	0.8758	0.3145	0.0314	0.0615
1024	1.7488	0.3452	0.0345	0.0676
2048	4.4055	1.0527	0.1053	0.2064
4096	9.6792	2.3947	0.2395	0.4694
8192	22.9609	5.6248	0.5625	1.1025
16384	58.3825	9.9410	0.9941	1.9484
32768	150.7299	5.5432	0.5543	1.0864
65536	356.9871	8.0942	0.8094	1.5864
131072	815.8607	17.5017	1.7502	3.4304
262144	2108.0771	140.4932	14.0493	27.5366

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

Block size	$\bar{X}$	s	$SE_{\bar{X}}$	$ME_{\bar{X}}$
16	0.1304	0.0546	0.0055	0.0108
32	0.0670	0.0177	0.0018	0.0035
64	0.1352	0.0215	0.0022	0.0043
128	0.4913	0.2475	0.0248	0.0486
256	0.6929	0.1662	0.0166	0.0325
512	1.7515	0.6352	0.0635	0.1245
1024	3.7688	1.1879	0.1188	0.2328
2048	8.6983	2.3341	0.2334	0.4575
4096	25.0276	4.3322	0.4332	0.8491
8192	52.0853	7.1292	0.7129	1.3973
16384	112.3024	8.1890	0.8189	1.6050
32768	239.0777	12.8962	1.2896	2.5276
65536	522.7409	31.4586	3.1459	6.1660
131072	1144.8802	91.1919	9.1192	17.8736
262144	2638.0547	206.8494	20.6849	40.5424

Table B.3: Data for C++ Princeton Iterative, Time (ms)

Block size	$\bar{X}$	s	$SE_{\bar{X}}$	$ME_{\bar{X}}$
16	0.0097	0.0032	0.0003	0.0006
32	0.0132	0.0009	0.0001	0.0002
64	0.0214	0.0044	0.0004	0.0008
128	0.0342	0.0045	0.0004	0.0008
256	0.0627	0.0040	0.0004	0.0008
512	0.1225	0.0032	0.0003	0.0006
1024	0.2744	0.0495	0.0050	0.0098
2048	0.5257	0.0185	0.0018	0.0035
4096	1.1701	0.0602	0.0060	0.0118
8192	2.5845	0.2085	0.0209	0.0410
16384	5.4518	0.5858	0.0586	0.1149
32768	12.2266	2.4651	0.2465	0.4831
65536	27.2805	4.8616	0.4862	0.9530
131072	78.9501	9.8472	0.9847	1.9300
262144	250.4870	26.9237	2.6924	5.2771

Table B.4: Data for C++ Princeton Recursive, Time (ms)

Block size	$\bar{X}$	s	$SE_{\bar{X}}$	$ME_{\bar{X}}$
16	0.0202	0.0039	0.0004	0.0008
32	0.0368	0.0038	0.0004	0.0008
64	0.0705	0.0105	0.0011	0.0022
128	0.1434	0.0059	0.0006	0.0012
256	0.2978	0.0127	0.0013	0.0025
512	0.6379	0.0181	0.0018	0.0035
1024	1.3437	0.0445	0.0044	0.0086
2048	2.8773	0.1288	0.0129	0.0253
4096	6.1206	0.3046	0.0305	0.0598
8192	13.2345	0.7311	0.0731	0.1433
16384	27.6080	0.9098	0.0910	0.1784
32768	55.1227	6.2636	0.6264	1.2277
65536	102.1585	17.9911	1.7991	3.5262
131072	231.2663	34.3656	3.4366	6.7357
262144	494.7038	69.7402	6.9740	13.6690

Table B.5: Data for Java Columbia Iterative, Time (ms)

Block size	$\bar{X}$	s	$SE_{\bar{X}}$	$ME_{\bar{X}}$
16	0.0370	0.0055	0.0005	0.0010
32	0.0899	0.0169	0.0017	0.0033
64	0.0244	0.0637	0.0064	0.0125
128	0.0134	0.0005	0.0001	0.0002
256	0.0293	0.0042	0.0004	0.0008
512	0.0615	0.0047	0.0005	0.0010
1024	0.1484	0.0308	0.0031	0.0061
2048	0.4338	0.0806	0.0081	0.0159
4096	1.2071	0.1807	0.0181	0.0355
8192	2.8726	0.3247	0.0325	0.0637
16384	6.3214	1.0542	0.1054	0.2066
32768	12.2634	2.9076	0.2908	0.5700
65536	24.9874	4.6266	0.4627	0.9069
131072	85.9483	8.2128	0.8213	1.6097
262144	274.5134	26.0859	2.6086	5.1129

Table B.6: Data for C++ Columbia Iterative, Time (ms)

Block size	$\bar{X}$	s	$SE_{\bar{X}}$	$ME_{\bar{X}}$
16	0.0072	0.0012	0.0001	0.0002
32	0.0086	0.0034	0.0003	0.0006
64	0.0083	0.0041	0.0004	0.0008
128	0.0115	0.0033	0.0003	0.0006
256	0.0200	0.0043	0.0004	0.0008
512	0.0366	0.0033	0.0003	0.0006
1024	0.0773	0.0097	0.0010	0.0020
2048	0.2604	0.0634	0.0063	0.0123
4096	0.7974	0.1097	0.0110	0.0216
8192	1.9326	0.2654	0.0265	0.0519
16384	4.2789	0.6399	0.0640	0.1254
32768	9.9388	1.6341	0.1634	0.3203
65536	23.1031	3.9019	0.3902	0.7648
131072	75.4942	8.3819	0.8382	1.6429
262144	243.8496	5.1390	0.5139	1.0072

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

Block size	$\bar{X}$	s	$SE_{\bar{X}}$	$ME_{\bar{X}}$
16	0.0057	0.0015	0.0002	0.0004
32	0.0050	0.0010	0.0001	0.0002
64	0.0056	0.0006	0.0001	0.0002
128	0.0089	0.0012	0.0001	0.0002
256	0.0134	0.0045	0.0004	0.0008
512	0.0348	0.0034	0.0003	0.0006
1024	0.0608	0.0096	0.0010	0.0020
2048	0.1925	0.0901	0.0090	0.0176
4096	0.3656	0.0817	0.0082	0.0161
8192	1.0051	0.0657	0.0066	0.0129
16384	2.1559	0.2562	0.0256	0.0502
32768	4.7898	0.7872	0.0787	0.1543
65536	9.8807	1.2509	0.1251	0.2452
131072	27.8158	3.6457	0.3646	0.7146
262144	63.1858	6.4595	0.6460	1.2662

Table B.8: Data for C++ NEON Recursive, Time (ms)

Block size	$\bar{X}$	s	$SE_{\bar{X}}$	$ME_{\bar{X}}$
16	0.0108	0.0035	0.0003	0.0006
32	0.0150	0.0051	0.0005	0.0010
64	0.0187	0.0069	0.0007	0.0014
128	0.0286	0.0077	0.0008	0.0016
256	0.0544	0.0392	0.0039	0.0076
512	0.0961	0.0071	0.0007	0.0014
1024	0.1868	0.0148	0.0015	0.0029
2048	0.3659	0.0419	0.0042	0.0082
4096	0.8086	0.1021	0.0102	0.0200
8192	1.6133	0.1416	0.0142	0.0278
16384	3.5690	0.4367	0.0437	0.0857
32768	7.6009	0.9583	0.0958	0.1878
65536	16.1129	1.8936	0.1894	0.3712
131072	34.1650	2.4085	0.2409	0.4722
262144	74.0746	5.4424	0.5442	1.0666

Table B.9: Data for C++ KISS, Time (ms)

Block size	$\bar{X}$	s	$SE_{\bar{X}}$	$ME_{\bar{X}}$
16	0.0056	0.0013	0.0001	0.0002
32	0.0069	0.0005	0.0001	0.0002
64	0.0079	0.0009	0.0001	0.0002
128	0.0131	0.0033	0.0003	0.0006
256	0.0182	0.0037	0.0004	0.0008
512	0.0461	0.0110	0.0011	0.0022
1024	0.0992	0.0883	0.0088	0.0172
2048	0.2047	0.0765	0.0076	0.0149
4096	0.4022	0.0625	0.0063	0.0123
8192	1.2470	0.2301	0.0230	0.0451
16384	2.3713	0.4915	0.0491	0.0962
32768	6.7420	1.2931	0.1293	0.2534
65536	16.0281	1.8073	0.1807	0.3542
131072	41.7620	3.0449	0.3045	0.5968
262144	102.1196	5.5195	0.5519	1.0817

Table B.10: Data for JNI No Params, Time (µs)

Block size	$\bar{X}$	s	$SE_{\bar{X}}$	$ME_{\bar{X}}$
16	1.7922	0.7104	0.0710	0.1392
32	1.6983	0.1117	0.0112	0.0220
64	1.6755	0.0763	0.0076	0.0149
128	1.9604	2.5399	0.2540	0.4978
256	1.7292	0.3541	0.0354	0.0694
512	1.6916	0.0556	0.0056	0.0110
1024	2.0228	2.8995	0.2900	0.5684
2048	1.7218	0.1475	0.0147	0.0288
4096	1.1411	0.2056	0.0206	0.0404
8192	1.1105	0.0404	0.0040	0.0078
16384	1.1183	0.1434	0.0143	0.0280
32768	1.1162	0.0427	0.0043	0.0084
65536	1.7463	6.2330	0.6233	1.2217
131072	1.1027	0.0722	0.0072	0.0141
262144	1.1006	0.0605	0.0060	0.0118

Table B.11: Data for JNI Vector, Time (µs)

Block size	$\bar{X}$	s	$SE_{\bar{X}}$	$ME_{\bar{X}}$
16	1.9333	0.6239	0.0624	0.1223
32	2.8130	9.1452	0.9145	1.7924
64	1.6344	0.9228	0.0923	0.1809
128	1.2349	0.6441	0.0644	0.1262
256	1.3276	1.3207	0.1321	0.2589
512	1.2567	0.6257	0.0626	0.1227
1024	1.3167	0.6842	0.0684	0.1341
2048	1.5416	0.7166	0.0717	0.1405
4096	1.4010	0.4024	0.0402	0.0788
8192	1.4818	0.3871	0.0387	0.0759
16384	1.7308	0.5318	0.0532	0.1043
32768	2.2099	0.9591	0.0959	0.1880
65536	4.7474	16.3058	1.6306	3.1960
131072	2.6375	0.7815	0.0781	0.1531
262144	3.3172	0.5938	0.0594	0.1164

Table B.12: Data for JNI Convert, Time (µs)

Block size	$\bar{X}$	s	$SE_{\bar{X}}$	$ME_{\bar{X}}$
16	2.6052	0.5122	0.0512	0.1004
32	2.6006	0.1886	0.0189	0.0370
64	2.6630	0.2169	0.0217	0.0425
128	1.9375	0.4305	0.0430	0.0843
256	1.8141	0.1415	0.0141	0.0276
512	2.2818	3.5774	0.3577	0.7011
1024	6.3756	43.2018	4.3202	8.4676
2048	1.9099	0.4584	0.0458	0.0898
4096	2.0062	0.7973	0.0797	0.1562
8192	2.3671	0.9683	0.0968	0.1897
16384	2.5833	0.8863	0.0886	0.1737
32768	3.2062	1.0345	0.1035	0.2029
65536	4.3198	1.4928	0.1493	0.2926
131072	5.7004	1.3676	0.1368	0.2681
262144	7.4630	1.1779	0.1178	0.2309

Table B.13: Data for JNI Columbia, Time (µs)

Block size	$\bar{X}$	s	$SE_{\bar{X}}$	$ME_{\bar{X}}$
16	4.1058	1.5515	0.1552	0.3042
32	3.9109	0.2731	0.0273	0.0535
64	3.9296	0.2887	0.0289	0.0566
128	3.0823	0.4548	0.0455	0.0892
256	3.0958	0.2248	0.0225	0.0441
512	3.1656	0.2326	0.0233	0.0457
1024	3.2896	0.7122	0.0712	0.1396
2048	3.4844	0.5676	0.0568	0.1113
4096	3.8562	1.6312	0.1631	0.3197
8192	3.8474	2.4407	0.2441	0.4784
16384	4.9724	4.5685	0.4569	0.8955
32768	5.3719	1.4667	0.1467	0.2875
65536	6.8136	1.2745	0.1275	0.2499
131072	9.6912	7.3151	0.7315	1.4337
262144	10.2781	1.1616	0.1162	0.2278

Table B.14: Common table for JNI tests, Time (µs)

Block size	**No params**	**Vector**	**Convert**	**Columbia**
16	1.7922 ± 0.1392	1.9333 ± 0.1223	2.6052 ± 0.1004	4.1058 ± 0.3042
32	1.6983 ± 0.0220	2.8130 ± 1.7924	2.6006 ± 0.0370	3.9109 ± 0.0535
64	1.6755 ± 0.0149	1.6344 ± 0.1809	2.6630 ± 0.0425	3.9296 ± 0.0566
128	1.9604 ± 0.4978	1.2349 ± 0.1262	1.9375 ± 0.0843	3.0823 ± 0.0892
256	1.7292 ± 0.0694	1.3276 ± 0.2589	1.8141 ± 0.0276	3.0958 ± 0.0441
512	1.6916 ± 0.0110	1.2567 ± 0.1227	2.2818 ± 0.7011	3.1656 ± 0.0457
1024	2.0228 ± 0.5684	1.3167 ± 0.1341	6.3756 ± 8.4676	3.2896 ± 0.1396
2048	1.7218 ± 0.0288	1.5416 ± 0.1405	1.9099 ± 0.0898	3.4844 ± 0.1113
4096	1.1411 ± 0.0404	1.4010 ± 0.0788	2.0062 ± 0.1562	3.8562 ± 0.3197
8192	1.1105 ± 0.0078	1.4818 ± 0.0759	2.3671 ± 0.1897	3.8474 ± 0.4784
16384	1.1183 ± 0.0280	1.7308 ± 0.1043	2.5833 ± 0.1737	4.9724 ± 0.8955
32768	1.1162 ± 0.0084	2.2099 ± 0.1880	3.2062 ± 0.2029	5.3719 ± 0.2875
65536	1.7463 ± 1.2217	4.7474 ± 3.1960	4.3198 ± 0.2926	6.8136 ± 0.2499
131072	1.1027 ± 0.0141	2.6375 ± 0.1531	5.7004 ± 0.2681	9.6912 ± 1.4337
262144	1.1006 ± 0.0118	3.3172 ± 0.1164	7.4630 ± 0.2309	10.2781 ± 0.2278

B.1.1 Double Tables

Table B.15: Common table for `double` C++ FFT tests, Time (ms)

Block size	Columbia Iterative	KISS	Princeton Iterative	Princeton Recursive
16	0.0072 ± 0.0002	0.0056 ± 0.0002	0.0097 ± 0.0006	0.0202 ± 0.0008
32	0.0086 ± 0.0006	0.0069 ± 0.0002	0.0132 ± 0.0002	0.0368 ± 0.0008
64	0.0083 ± 0.0008	0.0079 ± 0.0002	0.0214 ± 0.0008	0.0705 ± 0.0022
128	0.0115 ± 0.0006	0.0131 ± 0.0006	0.0342 ± 0.0008	0.1434 ± 0.0012
256	0.0200 ± 0.0008	0.0182 ± 0.0008	0.0627 ± 0.0008	0.2978 ± 0.0025
512	0.0366 ± 0.0006	0.0461 ± 0.0022	0.1225 ± 0.0006	0.6379 ± 0.0035
1024	0.0773 ± 0.0020	0.0992 ± 0.0172	0.2744 ± 0.0098	1.3437 ± 0.0086
2048	0.2604 ± 0.0123	0.2047 ± 0.0149	0.5257 ± 0.0035	2.8773 ± 0.0253
4096	0.7974 ± 0.0216	0.4022 ± 0.0123	1.1701 ± 0.0118	6.1206 ± 0.0598
8192	1.9326 ± 0.0519	1.2470 ± 0.0451	2.5845 ± 0.0410	13.2345 ± 0.1433
16384	4.2789 ± 0.1254	2.3713 ± 0.0962	5.4518 ± 0.1149	27.6080 ± 0.1784
32768	9.9388 ± 0.3203	6.7420 ± 0.2534	12.2266 ± 0.4831	55.1227 ± 1.2277
65536	23.1031 ± 0.7648	16.0281 ± 0.3542	27.2805 ± 0.9530	102.1585 ± 3.5262
131072	75.4942 ± 1.6429	41.7620 ± 0.5968	78.9501 ± 1.9300	231.2663 ± 6.7357
262144	243.8496 ± 1.0072	102.1196 ± 1.0817	250.4870 ± 5.2771	494.7038 ± 13.6690

Table B.16: Common table for `double` Java tests, Time (ms)

Block size	Columbia Iterative	Princeton Iterative	Princeton Recursive
16	0.0370 ± 0.0010	0.1227 ± 0.0053	0.1304 ± 0.0108
32	0.0899 ± 0.0033	0.1392 ± 0.1198	0.0670 ± 0.0035
64	0.0244 ± 0.0125	0.0905 ± 0.0468	0.1352 ± 0.0043
128	0.0134 ± 0.0002	0.1431 ± 0.0020	0.4913 ± 0.0486
256	0.0293 ± 0.0008	0.5698 ± 0.0947	0.6929 ± 0.0325
512	0.0615 ± 0.0010	0.8758 ± 0.0615	1.7515 ± 0.1245
1024	0.1484 ± 0.0061	1.7488 ± 0.0676	3.7688 ± 0.2328
2048	0.4338 ± 0.0159	4.4055 ± 0.2064	8.6983 ± 0.4575
4096	1.2071 ± 0.0355	9.6792 ± 0.4694	25.0276 ± 0.8491
8192	2.8726 ± 0.0637	22.9609 ± 1.1025	52.0853 ± 1.3973
16384	6.3214 ± 0.2066	58.3825 ± 1.9484	112.3024 ± 1.6050
32768	12.2634 ± 0.5700	150.7299 ± 1.0864	239.0777 ± 2.5276
65536	24.9874 ± 0.9069	356.9871 ± 1.5864	522.7409 ± 6.1660
131072	85.9483 ± 1.6097	815.8607 ± 3.4304	1144.8802 ± 17.8736
262144	274.5134 ± 5.1129	2108.0771 ± 27.5366	2638.0547 ± 40.5424

B.1.2 Float Tables

Table B.17: Common table for `float` Java tests, Time (ms)

Block size	Columbia Iterative	Princeton Iterative	Princeton Recursive
16	0.0265 ± 0.0002	0.1398 ± 0.0906	0.0387 ± 0.0041
32	0.0658 ± 0.0029	0.0492 ± 0.0025	0.0850 ± 0.0078
64	0.0179 ± 0.0096	0.1078 ± 0.0067	0.1761 ± 0.0027
128	0.0158 ± 0.0022	0.1991 ± 0.0378	0.4199 ± 0.0237
256	0.0290 ± 0.0016	0.3547 ± 0.0104	0.8938 ± 0.0292
512	0.0603 ± 0.0014	0.7740 ± 0.1313	1.9155 ± 0.0437
1024	0.1289 ± 0.0027	1.7228 ± 0.1009	4.1350 ± 0.0866
2048	0.3302 ± 0.0182	3.8546 ± 0.2185	9.2740 ± 0.1674
4096	0.8200 ± 0.0272	8.8053 ± 0.4232	26.0912 ± 0.7752
8192	2.1766 ± 0.0639	21.1757 ± 0.6689	50.0776 ± 1.4876
16384	3.9995 ± 0.0902	46.8150 ± 1.0327	103.9682 ± 3.0954
32768	8.9846 ± 0.2566	113.1953 ± 3.9572	219.0208 ± 5.5642
65536	20.2833 ± 0.4786	261.9954 ± 9.1987	485.1020 ± 13.3737
131072	47.2950 ± 1.3073	622.4328 ± 24.9022	1039.4937 ± 26.9767
262144	156.7135 ± 1.1934	1728.4640 ± 53.8042	2297.8011 ± 58.2951

Table B.18: Common table for `float` C++ tests, Time (ms)

Block size	Columbia Iterative	Princeton Iterative	Princeton Recursive
16	0.0049 ± 0.0004	0.0090 ± 0.0004	0.0215 ± 0.0043
32	0.0051 ± 0.0001	0.0128 ± 0.0010	0.0328 ± 0.0006
64	0.0052 ± 0.0002	0.0188 ± 0.0006	0.0617 ± 0.0010
128	0.0071 ± 0.0001	0.0292 ± 0.0006	0.1252 ± 0.0018
256	0.0160 ± 0.0080	0.0541 ± 0.0010	0.2642 ± 0.0078
512	0.0223 ± 0.0006	0.1038 ± 0.0012	0.5573 ± 0.0155
1024	0.0516 ± 0.0108	0.2082 ± 0.0025	1.1380 ± 0.0047
2048	0.1206 ± 0.0155	0.4536 ± 0.0106	2.4265 ± 0.0094
4096	0.8794 ± 0.0857	0.9690 ± 0.0061	5.1699 ± 0.0580
8192	1.8030 ± 0.1929	2.0605 ± 0.0110	10.7496 ± 0.0512
16384	2.9629 ± 0.1311	4.5027 ± 0.0341	22.8640 ± 0.1748
32768	8.2847 ± 0.2364	9.6797 ± 0.1288	48.0022 ± 0.2458
65536	18.8158 ± 0.5494	20.3049 ± 0.3467	96.9572 ± 0.2658
131072	43.7807 ± 1.2632	44.5770 ± 0.9577	192.4607 ± 5.9702
262144	134.8093 ± 2.0115	129.5150 ± 1.8201	398.9698 ± 12.9207

Table B.19: Common table for `float` NEON tests, Time (ms)

Block size	Iterative	Recursive
16	0.0057 ± 0.0004	0.0108 ± 0.0006
32	0.0050 ± 0.0002	0.0150 ± 0.0010
64	0.0056 ± 0.0002	0.0187 ± 0.0014
128	0.0089 ± 0.0002	0.0286 ± 0.0016
256	0.0134 ± 0.0008	0.0544 ± 0.0076
512	0.0348 ± 0.0006	0.0961 ± 0.0014
1024	0.0608 ± 0.0020	0.1868 ± 0.0029
2048	0.1925 ± 0.0176	0.3659 ± 0.0082
4096	0.3656 ± 0.0161	0.8086 ± 0.0200
8192	1.0051 ± 0.0129	1.6133 ± 0.0278
16384	2.1559 ± 0.0502	3.5690 ± 0.0857
32768	4.7898 ± 0.1543	7.6009 ± 0.1878
65536	9.8807 ± 0.2452	16.1129 ± 0.3712
131072	27.8158 ± 0.7146	34.1650 ± 0.4722
262144	63.1858 ± 1.2662	74.0746 ± 1.0666

B.2 Raw

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

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