Comparing Android Runtime with native: Fast Fourier Transform on Android

André Danielsson

March 6, 2017

KTH – Royal Institute of Technology

Master's Thesis in Computer Science

KTH Supervisor: Erik Isaksson

Bontouch Supervisor: Erik Westenius

Examiner: Olle Bälter

Abstract

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

Sammanfattning

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

Preface

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

André Danielsson

CONTENTS

GLOSSARY

Android Mobile operating system. 1

Clang Compiler used by the NDK. 12

CMake Build tool used by the NDK. 10

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

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

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

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

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

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

List of Figures

List of Tables

CHAPTER 1

Introduction

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

1.1 Background

Android is an operating system for smartphones and as of November 2016 it is the most used [**?**]. One reason for this is because it was designed to be run on multiple different architectures [**?**]. 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 [**?**]. The Android kernel is based on the Linux kernel although with some alterations to support the hardware of mobile devices.

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

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

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

1.2 Problem

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

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

1.3 Purpose

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

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

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

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

1.4 Goal

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

1.5 Procedure

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

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

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

Figure 1.1: Expression used to filter out relevant articles

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

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

1.6 Delimitations

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

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

1.7 Limitations

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

1.8 Ethics and Sustainability

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

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

1.9 Outline

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

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

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

- ***Chapter ?? - Experiments*** – The result of the experiments are presented here.

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

- ***Chapter ?? - 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 [**?**].

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 [**?**]. Android Runtime (ART) features Ahead-Of-Time (AOT) compilation and Just-In-Time (JIT) compilation, garbage collection and debugging support [**?**]. This is where the Java code is being run and because of the debugging and garbage collection support, it is also beneficial for the developer to write applications against this layer.

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

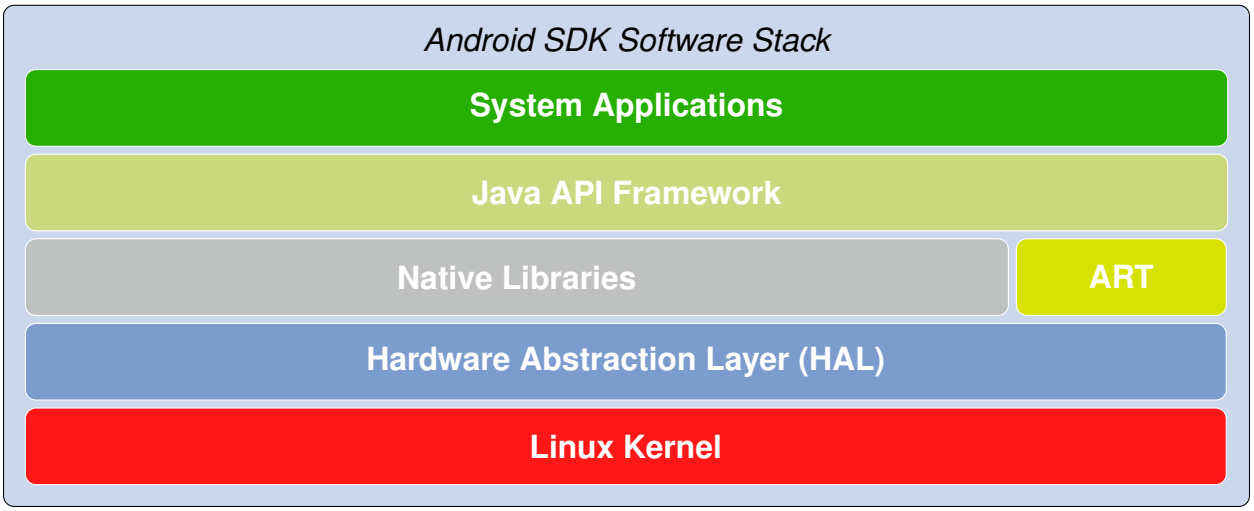


Figure 2.1: Android SDK Software Stack [?]

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 [?].

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

2.2 Dalvik Virtual Machine

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

The Dalvik Virtual Machine (DVM) is the VM initially used on Android. One difference between DVM and JVM is that the DVM uses a register-based architecture while the JVM uses a stack-based architecture. The most common virtual machine architecture is the stack-based [?, 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 [**?**, p. 159-160].

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

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

2.3 Android Runtime

Android Runtime is the new default runtime for Android as of version 5.0 [**?**]. 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 [**?**] 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 [**?**]. This oat-file is in the Executable and Linkable Format (ELF) and can be seen as a wrapper of multiple dex-files [**?**].

2.4 Native Development Kit

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

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

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

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

2.4.1 Java Native Interface

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

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

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

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

⇕

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

Figure 2.2: Native method declaration to implementation.

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

The second parameter to a JNI function is of the `jobject` type. This is the current Java object that has called this specific JNI function. It can be seen as an equivalent to the `this` keyword in Java and C++ [**?**, 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 [**?**, 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 [**?**, p. 158]. There are two other functions, `GetPrimitiveArrayCritical()` and `ReleasePrimitiveArrayCritical()`, that can be used to avoid garbage collection in native code. Between these function calls, the native code should not run forever, no calls to any of the JNI functions are allowed and it is prohibited to block a thread that depends on a VM thread to continue.

2.4.2 LLVM and Clang

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

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

2.5 Code Optimization

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

Loop unrolling

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

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

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

(a) Normal

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

(b) One unroll

Figure 2.3: Loop unrolling in C

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

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

(a) Normal

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

(b) One unroll

Figure 2.4: Loop unrolling in assembly

Inlining

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

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

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

(a) One unroll

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

(b) Optimized unroll

Figure 2.5: Optimized loop unrolling in assembly

Constant folding

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

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

(a) Before optimization

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

(b) Constant propagation optimization

Figure 2.6: Constant Propagation

Loop Tiling

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

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

(a) Before loop tiling

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

(b) After loop tiling

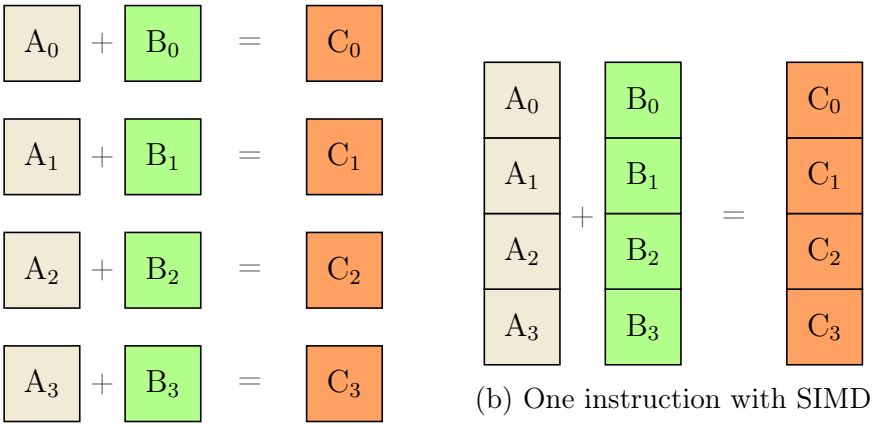
Figure 2.7: Loop Tiling

2.5.1 Java

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

2.5.2 C++

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



(a) Four separate instructions

(b) One instruction with SIMD

Figure 2.8: Single Instruction Multiple Data [?]

NEON

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

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

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

2.6 Discrete Fourier Transform

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

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

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

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

Equation 2.1 [**?**, 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 [**?**] called the Fast Fourier Transform (FFT).

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

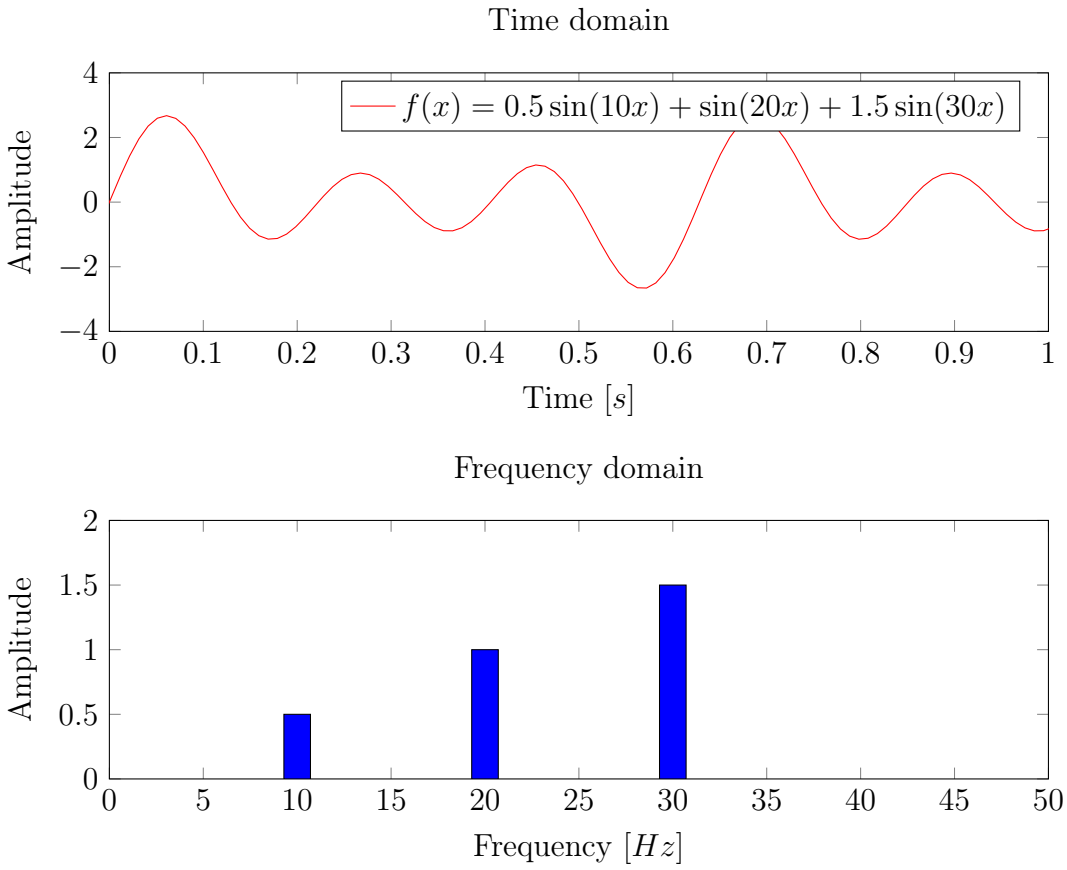


Figure 2.9: Time domain and frequency domain of a signal

2.7 Related work

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

Evaluating Performance of Android Platform Using Native C for Embedded Systems [?] 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* [**?**] presented a performance comparison between ART and native on Android. The main focus of the report was to find how much more efficient one of them were in terms of energy consumption. Their tests consisted of measuring battery drainage in power as well as execution time of different algorithms. It also compares performance differences between ART and Dalvik. The conclusion states that native performs much better than code running on the Dalvik VM. However, code compiled by ART improves greatly from Dalvik and performs almost the same as code compiled by Android NDK.

CHAPTER 3

Method

To ensure that the experiment is carried out correctly, many different tools for measurements was evaluated. Different implementations of the FFT are also compared to choose the ones that would typically be used in an Android project.

3.1 Experiment model

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

1. Cost of using the JNI

2. Compare well known libraries

3. Compare two optimized code samples in Java and C++

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

There are many different implementations of the FFT publicly available that could be of interest for use in ones project. This test demonstrates how some different libraries

compare. It is helpful to see how viable different implementations are on Android, for both C++ libraries and Java libraries. It can also be useful to know how a small implementation compares to a large and complex library.

Finally, comparing optimization techniques for small libraries is a good way of demonstrating how a developer can improve performance to fit the requirements while still retaining manageable source code. Having one single source file is valuable, especially for native libraries.

3.1.1 Hardware

The setup used for performing the experiments were the following:

Table 3.1: Hardware used in the experiments

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

3.1.2 Benchmark Environment

During the tests, cellular was switched off and wifi was enabled and connected. There were no applications running in the background while performing the tests during the experiments. The tests were executed and compiled with the following versions:

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

3.1.3 Time measurement

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

To get comparable results, matching work between algorithms were included in the time measurements. Each test was given the formatted data it needed because . Although every test had to return a complex representation of the results. They all described this using the same class called *Complex* found in Appendix **??** Listing **??**. This class was written by Robert Sedgewick and Kevin Wayne [**?**].

3.1.4 Memory measurement

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

3.2 Evaluation

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

3.2.1 Data representation

3.2.2 Data interpretation

3.2.3 Sources of error

3.2.4 Statistical significance

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

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

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

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

When comparing results, we need to find a confidence interval for a given test and choose a confidence level. For the data gathered in this study, a 95% two-sided confidence level was chosen when comparing the data. The confidence interval is calculated by taking the standard error of the mean which is found by using the following formula [**?**, 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:

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

Figure 3.1: JNI test function with no parameters and no return value

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

Figure 3.2: JNI test function with a double array as input parameter and return value

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

Our confidence interval will then be:

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

3.3 JNI Tests

For testing the JNI overhead, three different tests were constructed. The first test had no parameters, returned void and did no calculations. The purpose of this test was to see how long it would take to call the smallest function possible. The function shown in Figure 3.1 was used to test this.

For the second test, a function was written (see Figure 3.2) that took a jdoubleArray as input and output. The reason this test was made was to see if JNI introduced some extra overhead for passing an argument and having a return value.

In Figure 3.3, the third test started by calling the `GetDoubleArrayElements` function to be able to access the elements stored in `arr`. When all the calculations are done, the function will return `arr`. To overwrite the changes made on `elements`, a function called `ReleaseDoubleArrayElements` must be called.

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

Figure 3.3: Get and release elements

3.4 Fast Fourier Transform Algorithms

3.4.1 Java libraries

3.4.2 C++ libraries

CHAPTER 4

Experiments

Summarizing the chapter

Table 4.1: Common table for C++ tests

Block size	Columbia converted Iterative	Columbia optimized Iterative	KISS	Princeton converted Iterative	Princeton converted Recursive
16	0.0225 ± 0.0033	0.0198 ± 0.0025	0.0195 ± 0.0067	0.0342 ± 0.0053	0.0612 ± 0.0065
32	0.0322 ± 0.0025	0.0322 ± 0.0031	0.0239 ± 0.0031	0.0545 ± 0.0043	0.1085 ± 0.0020
64	0.0525 ± 0.0014	0.0524 ± 0.0012	0.0338 ± 0.0020	0.0847 ± 0.0059	0.2148 ± 0.0024
128	0.1025 ± 0.0033	0.0814 ± 0.0127	0.0629 ± 0.0084	0.1328 ± 0.0029	0.4517 ± 0.0057
256	0.0925 ± 0.0178	0.0822 ± 0.0039	0.1158 ± 0.0035	0.2807 ± 0.0073	0.9139 ± 0.0067
512	0.1709 ± 0.0267	0.1744 ± 0.0308	0.2109 ± 0.0049	0.5486 ± 0.0253	1.9142 ± 0.0102
1024	0.3656 ± 0.0284	0.3397 ± 0.0108	0.4072 ± 0.0086	1.1691 ± 0.0172	4.0665 ± 0.0127
2048	0.9177 ± 0.0541	0.7402 ± 0.0190	0.8635 ± 0.0243	2.4714 ± 0.0188	8.7235 ± 0.0725
4096	1.6737 ± 0.0461	1.9889 ± 0.0982	1.9558 ± 0.1347	5.3867 ± 0.1000	18.3487 ± 0.1235
8192	3.7768 ± 0.1838	3.8584 ± 0.2236	3.8499 ± 0.1603	11.7050 ± 0.5076	38.4780 ± 0.6441
16384	8.2947 ± 0.3759	8.5556 ± 0.5672	7.8854 ± 0.2775	24.3807 ± 0.5902	80.4920 ± 0.8228
32768	19.1886 ± 1.1809	18.5907 ± 0.9959	17.6197 ± 0.5490	52.3713 ± 1.1313	167.3867 ± 1.5300
65536	42.8520 ± 1.4120	44.2337 ± 2.4361	38.3601 ± 0.7332	112.4273 ± 1.1197	346.7600 ± 1.9190

Table 4.2: Common table for Java tests

Block size	**Columbia Iterative**	**Princeton Iterative**	**Princeton Recursive**
16	0.0210 ± 0.0008	0.1738 ± 0.0368	0.2730 ± 0.0708
32	0.0429 ± 0.0018	0.0571 ± 0.0071	0.3983 ± 0.0568
64	0.0906 ± 0.0010	0.0916 ± 0.0073	0.2104 ± 0.0402
128	0.2233 ± 0.0382	0.2353 ± 0.0339	0.3204 ± 0.0425
256	0.0372 ± 0.0022	0.4380 ± 0.0316	0.7415 ± 0.0431
512	0.0754 ± 0.0029	0.9865 ± 0.0672	1.7743 ± 0.1944
1024	0.1507 ± 0.0059	2.0255 ± 0.0933	3.5339 ± 0.2466
2048	0.4299 ± 0.0312	4.5366 ± 0.3038	8.2740 ± 0.6905
4096	0.9984 ± 0.0915	10.6191 ± 0.8679	17.9445 ± 0.9022
8192	2.3125 ± 0.2709	27.5617 ± 1.9755	37.6118 ± 1.5008
16384	4.8597 ± 0.4114	62.6869 ± 1.7191	80.1848 ± 1.9414
32768	11.2535 ± 0.9718	155.5247 ± 2.7411	172.5062 ± 2.2489
65536	26.3906 ± 2.1662	366.0557 ± 2.8910	366.6833 ± 3.3610

Table 4.3: Java Princeton Iterative, Time (ms)

FFT Size	$\bar{X}$	s	$SE_{\bar{X}}$	$ME_{\bar{X}}$
16	0.1738	0.1030	0.0188	± 0.0368
32	0.0571	0.0199	0.0036	± 0.0071
64	0.0916	0.0201	0.0037	± 0.0073
128	0.2353	0.0948	0.0173	± 0.0339
256	0.4380	0.0880	0.0161	± 0.0316
512	0.9865	0.1878	0.0343	± 0.0672
1024	2.0255	0.2609	0.0476	± 0.0933
2048	4.5366	0.8488	0.1550	± 0.3038
4096	10.6191	2.4252	0.4428	± 0.8679
8192	27.5617	5.5205	1.0079	± 1.9755
16384	62.6869	4.8042	0.8771	± 1.7191
32768	155.5247	7.6599	1.3985	± 2.7411
65536	366.0557	8.0791	1.4750	± 2.8910

Table 4.4: Java Princeton Recursive, Time (ms)

FFT Size	$\bar{X}$	s	$SE_{\bar{X}}$	$ME_{\bar{X}}$
16	0.2730	0.1976	0.0361	± 0.0708
32	0.3983	0.1589	0.0290	± 0.0568
64	0.2104	0.1121	0.0205	± 0.0402
128	0.3204	0.1191	0.0217	± 0.0425
256	0.7415	0.1205	0.0220	± 0.0431
512	1.7743	0.5432	0.0992	± 0.1944
1024	3.5339	0.6888	0.1258	± 0.2466
2048	8.2740	1.9298	0.3523	± 0.6905
4096	17.9445	2.5209	0.4603	± 0.9022
8192	37.6118	4.1940	0.7657	± 1.5008
16384	80.1848	5.4254	0.9905	± 1.9414
32768	172.5062	6.2846	1.1474	± 2.2489
65536	366.6833	9.3924	1.7148	± 3.3610

Table 4.5: Java Columbia Iterative, Time (ms)

FFT Size	$\bar{X}$	s	$SE_{\bar{X}}$	$ME_{\bar{X}}$
16	0.0210	0.0024	0.0004	± 0.0008
32	0.0429	0.0052	0.0009	± 0.0018
64	0.0906	0.0027	0.0005	± 0.0010
128	0.2233	0.1069	0.0195	± 0.0382
256	0.0372	0.0058	0.0011	± 0.0022
512	0.0754	0.0083	0.0015	± 0.0029
1024	0.1507	0.0164	0.0030	± 0.0059
2048	0.4299	0.0870	0.0159	± 0.0312
4096	0.9984	0.2558	0.0467	± 0.0915
8192	2.3125	0.7572	0.1382	± 0.2709
16384	4.8597	1.1497	0.2099	± 0.4114
32768	11.2535	2.7156	0.4958	± 0.9718
65536	26.3906	6.0532	1.1052	± 2.1662

Table 4.6: C++ Princeton Iterative, Time (ms)

FFT Size	$\bar{X}$	s	$SE_{\bar{X}}$	$ME_{\bar{X}}$
16	0.0342	0.0149	0.0027	± 0.0053
32	0.0545	0.0121	0.0022	± 0.0043
64	0.0847	0.0162	0.0030	± 0.0059
128	0.1328	0.0083	0.0015	± 0.0029
256	0.2807	0.0201	0.0037	± 0.0073
512	0.5486	0.0704	0.0129	± 0.0253
1024	1.1691	0.0484	0.0088	± 0.0172
2048	2.4714	0.0524	0.0096	± 0.0188
4096	5.3867	0.2794	0.0510	± 0.1000
8192	11.7050	1.4185	0.2590	± 0.5076
16384	24.3807	1.6491	0.3011	± 0.5902
32768	52.3713	3.1612	0.5772	± 1.1313
65536	112.4273	3.1292	0.5713	± 1.1197

Table 4.7: C++ Princeton Recursive, Time (ms)

FFT Size	$\bar{X}$	s	$SE_{\bar{X}}$	$ME_{\bar{X}}$
16	0.0612	0.0182	0.0033	± 0.0065
32	0.1085	0.0055	0.0010	± 0.0020
64	0.2148	0.0065	0.0012	± 0.0024
128	0.4517	0.0159	0.0029	± 0.0057
256	0.9139	0.0184	0.0034	± 0.0067
512	1.9142	0.0287	0.0052	± 0.0102
1024	4.0665	0.0354	0.0065	± 0.0127
2048	8.7235	0.2027	0.0370	± 0.0725
4096	18.3487	0.3453	0.0630	± 0.1235
8192	38.4780	1.7997	0.3286	± 0.6441
16384	80.4920	2.2996	0.4198	± 0.8228
32768	167.3867	4.2754	0.7806	± 1.5300
65536	346.7600	5.3628	0.9791	± 1.9190

Table 4.8: C++ Columbia Iterative, Time (ms)

FFT Size	$\bar{X}$	s	$SE_{\bar{X}}$	$ME_{\bar{X}}$
16	0.0225	0.0092	0.0017	± 0.0033
32	0.0322	0.0073	0.0013	± 0.0025
64	0.0525	0.0036	0.0007	± 0.0014
128	0.1025	0.0092	0.0017	± 0.0033
256	0.0925	0.0500	0.0091	± 0.0178
512	0.1709	0.0747	0.0136	± 0.0267
1024	0.3656	0.0792	0.0145	± 0.0284
2048	0.9177	0.1510	0.0276	± 0.0541
4096	1.6737	0.1286	0.0235	± 0.0461
8192	3.7768	0.5139	0.0938	± 0.1838
16384	8.2947	1.0503	0.1918	± 0.3759
32768	19.1886	3.3001	0.6025	± 1.1809
65536	42.8520	3.9460	0.7204	± 1.4120

Table 4.9: C++ KISS, Time (ms)

FFT Size	$\bar{X}$	s	$SE_{\bar{X}}$	$ME_{\bar{X}}$
16	0.0195	0.0186	0.0034	± 0.0067
32	0.0239	0.0089	0.0016	± 0.0031
64	0.0338	0.0055	0.0010	± 0.0020
128	0.0629	0.0237	0.0043	± 0.0084
256	0.1158	0.0096	0.0018	± 0.0035
512	0.2109	0.0136	0.0025	± 0.0049
1024	0.4072	0.0242	0.0044	± 0.0086
2048	0.8635	0.0681	0.0124	± 0.0243
4096	1.9558	0.3763	0.0687	± 0.1347
8192	3.8499	0.4482	0.0818	± 0.1603
16384	7.8854	0.7758	0.1416	± 0.2775
32768	17.6197	1.5340	0.2801	± 0.5490
65536	38.3601	2.0493	0.3741	± 0.7332

Table 4.10: C++ Columbia Iterative Optimized, Time (ms)

FFT Size	$\bar{X}$	s	$SE_{\bar{X}}$	$ME_{\bar{X}}$
16	0.0198	0.0070	0.0013	± 0.0025
32	0.0322	0.0085	0.0016	± 0.0031
64	0.0524	0.0032	0.0006	± 0.0012
128	0.0814	0.0357	0.0065	± 0.0127
256	0.0822	0.0107	0.0020	± 0.0039
512	0.1744	0.0858	0.0157	± 0.0308
1024	0.3397	0.0302	0.0055	± 0.0108
2048	0.7402	0.0532	0.0097	± 0.0190
4096	1.9889	0.2742	0.0501	± 0.0982
8192	3.8584	0.6249	0.1141	± 0.2236
16384	8.5556	1.5850	0.2894	± 0.5672
32768	18.5907	2.7830	0.5081	± 0.9959
65536	44.2337	6.8074	1.2429	± 2.4361

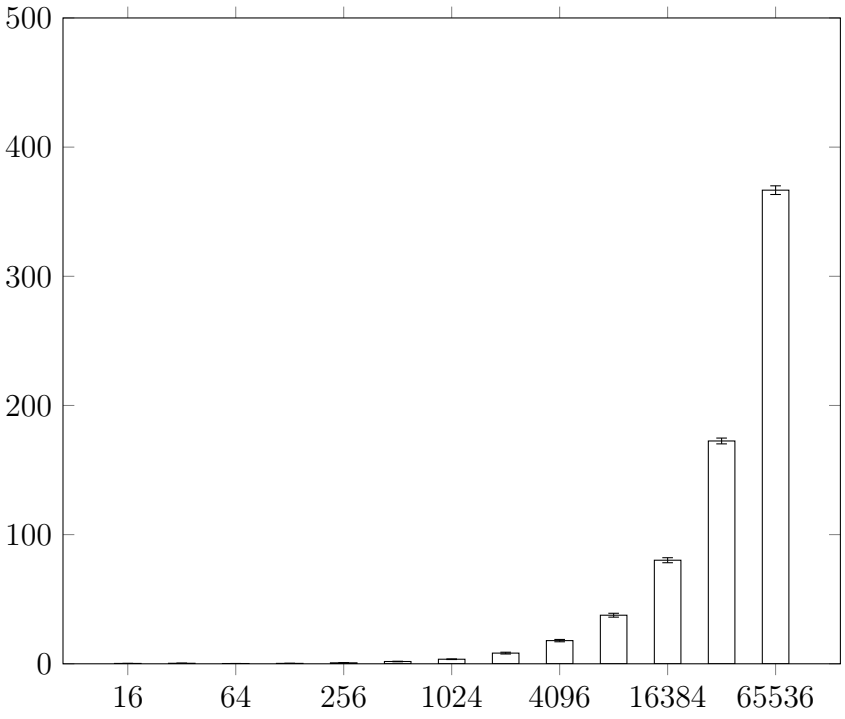


Figure 4.1: C++ Columbia Iterative Optimized bar plot

CHAPTER 5

Discussion

CHAPTER 6

Conclusion

Appendix A

Source code

Listing A.1: Complex.java [?]

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

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

import java.util.Objects;

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

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

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

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

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

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

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

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

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

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

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

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

    // return the real or imaginary part
```

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

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

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

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

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

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



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

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

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

}
```

Bibliography

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[22] Piotr Luszczek, "Data-Level Parallelism in Vector, SIMD, and GPU Architectures." http://www.icl.utk.edu/~luszczek/teaching/courses/fall2013/cosc530/cosc530_ch4all6up.pdf. University of Tennessee, [Accessed: 15 February 2017].

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

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

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

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

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

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

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

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

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

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