1-28-2015

Improving Mobile SOC's Performance as an Energy Efficient DSP Platform with Heterogeneous Computing

Matthew Briggs

Matthew Briggs

Candidate

Electrical and Computer Engineering

Department

This thesis is approved, and it is acceptable in quality and form for publication:

Approved by the Thesis Committee:

Payman Zarkesh-Ha , Chairperson

James Plusquellic

L. Howard Pollard

Improving Mobile SOC's Performance as an Energy Efficient DSP Platform with Heterogeneous Computing

by

Matt Briggs

B.S., New Mexico Institute of Mining and Technology

THESIS

Submitted in Partial Fulfillment of the
Requirements for the Degree of

Master of Science
Computer Engineering

The University of New Mexico

Albuquerque, New Mexico

December, 2014

Acknowledgments

I would first like to thank my graduate advisor Dr. Payman Zarkesh-Ha who has kindly supported me through not just my thesis but my entire graduate career. He has always been able to ask the right questions and provide feedback when it was needed most. I would also like to thank, Dr. Plusquellic and Dr. Pollard for their willingness to serve on my committee and share their knowledge and expertise.

I would also like to thank the following individuals who trusted me with their personal electronics which were tested as part of this research: Laxmi Botla, Pat Briggs, Robert Engelhardt, and Rosalie McCreary.

In addition, this body of work would not be anywhere as clear or understandable without the editing skills and perseverance of: Robert Engelhardt and Lauren Ratliff.

I would like to especially thank my wife, Tova, who has been willing to put up with me working many long hours, taking over her office, and believing in me even when I lost faith in myself.

Improving Mobile SOC's Performance as an Energy Efficient DSP Platform with Heterogeneous Computing

by

Matt Briggs

B.S., New Mexico Institute of Mining and Technology

M.S., Computer Engineering, University of New Mexico, 2014

Abstract

Mobile system-on-chip (SOC) technology is improving at a staggering rate spurred primarily by the adoption of smartphones and tablets. This rapid innovation has allowed the mobile SOC to be considered in everything from high performance computing to embedded applications. In this work, modern SOC's heterogeneous computing capabilities are evaluated with a focus toward digital signal processing (DSP). Evaluation is conducted on modern consumer devices running Android operating system and leveraging the relatively new RenderScript Compute to utilize CPU resources alongside other compute resources such as graphics processing units (GPUs) and digital signal processors. In order to benchmark these concepts, several implementations of both the discrete Fourier transform (DFT) and the fast Fourier transform (FFT) are tested across devices. The results show both improvement in performance and energy efficiency on many devices compared to traditional Java implementations and indicate that the mobile SOC is a relevant platform for DSP applications.

Contents

List of Figures

List of Tables

Glossary

ACE	AXI Coherency Extensions. This extension to the AXI bus can provide the following support for hardware coherent caches, barrier transactions for guaranteeing ordering, and distributed virtual memory messaging/management.
ACP	AXI Accelerator Coherency Port
ADC	Analog to digital converter
ALU	Arithmetic Logic Unit
API	Application Programming Interface
APK	Android Package, This package contains byte code, Renderscipt files, XML GUI configuration files needed for an Android application
APP	Application
AXI	An open standard microcontroller bus used by ARM and others to connect processors and peripherals together
C99	An informal name for ISO/IEC 9899:1999 which is version of the C programming language
CAD	Computer-Aided Drafting

Glossary

CPU	Central Processing Unit
Dalvik	Google implementation of the Java Virtual Machine (JVM) used in the Android operating System
DFT	Discrete Fourier Transform
DIF	Decimation in Frequency
DIT	Decimation in Time
DDR	Double Data Rate memory also referred to as DDR SRAM is a memory commonly used in a computer
DSP	Digital Signal Processing or Digital Signal Processor
DUT	Device under Test
ECC	Error checking and correction, in the context of this work usually associated with a feature available in memory devices
FFT	Fast Fourier Transform
FFTW	Fast Fourier Transform in the West, a software library which computes FFTs
FLOPS	Floating Operations per Second
FPGA	Field Programmable Gated Array
FPU	Floating point unit, a hardware device capable executing floating point computations
GDDR	Graphics DDR, see DDR
GPGPU	General Purpose computing on Graphics Processing Units

Glossary

GPU	Graphics Processing Unit
IP	Intellectual Property, in the context of this work, describes a hardware implementation which can be integrated in a SOC.
JIT Compiler	Just-In-Time compiling is compilation done during execution of a program
HPC	High Performance Computing
IO	Input and output
IR	Intermediate Representation
LTE	Long-Term Evolution, A 4G communication standard
NDK	Native Development Kit, in the Android context this means having access via C/C++ interface to the device.
OpenCL	Open Compute Language, is a programming extension which allows a subset of C99 which through some extensions allows one to take advantage of Heterogeneous computing.
ORNL	Oak Ridge National Laboratory
OS	Operating System
PCI	Peripheral Component Interconnect, A bus standard for connecting computers and their peripherals
PCIe	PCI Express, A revision of PCI which has improved throughput and lower IO count.
SIMD	Single Instruction Multiple Data, a computer instruction which operates on data operands at once

SMX	Streaming Multiprocessors, a term used by NVIDIA to describe a SIMD processing units inside a GPU
SOC	System-on-Chip
SSE	Streaming SIMD Extensions are extensions implemented on both AMD and Intel processors to improve performance of multimedia applications.
TLB	Table look aside buffer

Chapter 1

Introduction

In recent years, the explosion of smartphone usage has brought an amazing amount of computing power to a typical consumer's pocket. Demand for these devices has dramatically reduced the cost and increased the performance of mobile system-on-chips (SOCs). Most Android smartphones and tablets now have quad-core processors with a graphics processing unit (GPU) integrated into a single SOC. This rise of the mobile platform means that applications, which were once limited to either customized field-programmable gate array (FPGA) or a powerful computer, have become viable on mobile SOCs. One such application is digital signal processing (DSP).

The mobile space's SOC architecture is far less defined compared to the standard desktop or laptop computer. In the traditional desktop computer market there are only two main processor vendors, Intel and AMD, and two main GPU vendors, NVIDIA and AMD. This leads to standardized architectures, which are well understood and supported. However, this is not true in the mobile space as there are many processor vendors such as Apple, Samsung, Qualcomm, and Texas Instruments. In addition, there are also many mobile GPU vendors including NVIDIA, ARM Mali, Arduino, and PowerVR. One of the disadvantages of a diverse landscape of vendors

in the mobile market is that many details are not openly disclosed or published, therefore the only way to understand performance is through benchmarking. This is especially true when talking about general computing performance of mobile GPUs.

One key to getting performance out of a mobile SOC, especially energy-efficient performance, is utilizing both the CPU and GPU. Until recently, this required use of device specific frameworks which were fairly involved, but Google has released a framework called RenderScript Compute which enables one to access processors, GPU, and even DSP cores as generic compute units. This framework is similar to both OpenCL and CUDA which have been used extensively in both desktop and High Performance Computing (HPC) spaces to utilize CPU, GPU, and other DSP hardware.

One drawback of heterogeneous computing is that it complicates the programming model. To overcome this, software technologies are utilized which abstract this diverse and complex hardware. This abstraction enables programmers to use familiar constructs and languages with minimal performance cost. Although this abstraction does not completely solve the additional complexity of heterogeneous hardware, it can mitigate many of the issues and allow features such as code portability.

1.1 Motivation and Approach

Given the proven advantages of heterogeneous computing in both the desktop and HPC space, it seems that the next logical step is to apply these techniques to the mobile environment. Mobile computing offers many challenges and opportunities when looking at heterogeneous computing. Challenges include limited computing resources, restrictive programming environments, and sparsely documented hardware and software interfaces. The opportunities include dynamic and rapidly evolving hardware architectures, tightly coupled hardware resources (many times in a single

SOC), and huge demand for energy-efficient computing,

The approach taken for this work is to focus on technologies and techniques which could be used in production software and utilized on consumer-class devices. In terms of hardware, this means limiting platforms to tablets and smartphones. In software, this means writing code on portable frameworks which can be deployed in the Google Play Store and used across devices with minimal or no tuning or tweaking.

Chapter 2

Heterogeneous Computing Hardware

Heterogeneous computing can be broadly defined as using varied computational hardware such as GPUs, DSPs, or custom FPGAs in conjunction with a CPU to complete a task. These platforms have the advantage that the strengths of multiple hardware architectures can be combined to more efficiently solve a problem. A disadvantage is that the complexity in the system is increased by multiple computation devices, this has traditionally been the job of the programmer to address. A common example of heterogeneous computing, is moving parallel operations to a GPU where the architecture allows for parallel computation with faster local memory, thus allowing the CPU to focus on servicing IO and providing flow control for the rest of the program.

2.1 Typical Desktop Computer Architecture

To describe the architecture of a mobile SOC or heterogeneous computer, it is useful to understand a modern desktop computer and how both the CPU and GPU may be

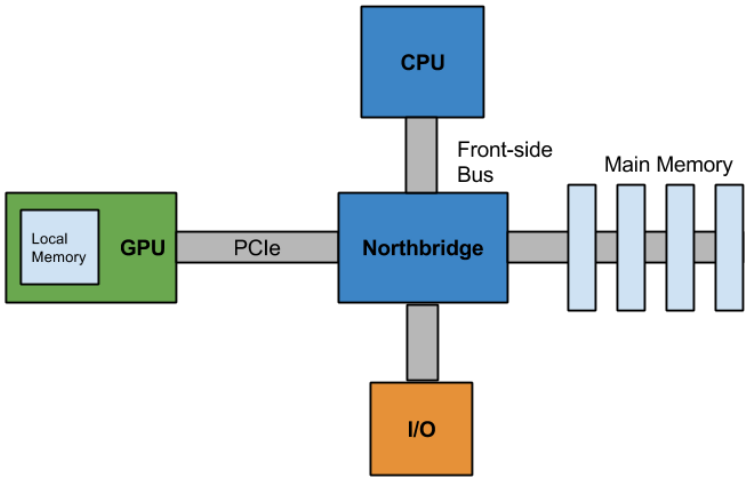


Figure 2.1: Typical CPU / GPU Architecture

exploited for DSP applications. In Figure 2.1 the CPU and GPU are connected via a PCI-Express Bus (PCIe). The memory is connected to the CPU via the northbridge. The GPU's memory is separate and integrated on the PCI card itself. Traditionally, computation is performed on the CPU, and the GPU is reserved for graphics only.

In the case of either a single-threaded or multi-threaded DSP algorithm, there are many calculations to be done on a relatively small amount of memory. Thus the memory bandwidth and cache sizes are rarely the limiting factor. The limiting factor is often the number of cores which can perform the computation [1]. To address this problem both AMD and Intel have added Streaming SIMD Extensions (SSE). These extensions have been used by such projects as FFTW to improve performance [2].

Recently, applications have begun using both CPU and GPU for calculations to improve performance. In this case, for DSP calculations, the data which is to be operated on is moved via PCIe bus to the graphics card. Then the large parallel GPU quickly does computation and the results need to be copied back to main memory. In the case of DSP, the bus link between CPU and GPU is often the limiting factor [1]. Both the GPU architecture and middleware involved are explored more deeply

in subsequent sections.

2.2 GPU Architecture

2.2.1 General

Modern GPUs are essentially programmable parallel computers which are capable of handling both floating point and integer math on large datasets in a very efficient way; however, GPUs did not always operate this way. Originally GPUs were designed to be fixed function special purpose devices for graphics [1]. These devices, although providing very powerful graphics acceleration, did not give programmers, graphics or otherwise, much flexibility to optimize hardware utilization. Depicted in Figure 2.2 is a typical graphics pipeline showing the various steps that take place on a GPU to take a set of vertices and convert them to a screen image [3].

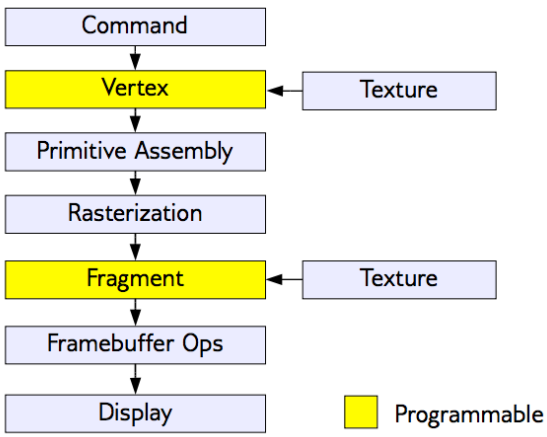


Figure 2.2: Typical Graphics Pipeline

There are many stages involved in the rendering of modern computer graphics as shown in Figure 2.2 of which two, depicted as programmable, are the most impor-

tant in the development of GPU programmability. The first operation is known as vertex operation which does the following on a per-vertex basis: transformation into screenspace, texture coordinate generation/transformation, and applying lighting effects. Next, the fragment stage using color information from vertices and textures, computes each fragment's color; other effects such as lighting can be applied at this stage. Both of these operations are highly parallelizable as they are applied on a per- vertex/fragment basis on a large numbers of objects. These computations have always been configurable *i.e.* a programmer could choose from a handful of lighting effects. However as graphics engines became more complex, it became clear that fixed-function hardware could not support the plethora of desired effects; therefore programmability support was required.

The process of changing from fixed-function hardware to a programmable platform started slowly with per-vertex operations and per-fragment operations moving from configurable fixed hardware implementations to having independent instruction sets which programmers could configure with shader programs. One limitation of separate vertex and fragment instruction sets is applications can over utilize vertex hardware and underutilize fragment hardware or vise versa leading to underutilization of the hardware [1].

This limitation gave rise to the foundation of the modern GPU with the unified shader model. The unified shader model allows the same hardware to be used for both vertex and fragment operations. This not only solves the partitioning problem expressed above but also gives rise to the possibility of these cores to be used for applications which are no longer graphics specific. Along with software support, this has enabled the General Purpose GPU (GPGPU) generation of devices to become common practice in many problem spaces such as 3D CAD, simulation toolkits, and even high performance computing (HPC).

In addition to the above major architectural shader change, many newer gen-

erations of GPUs have begun to support single and double precision floating point operations. This improvement makes GPUs an especially relevant option for scientific computing including signal processing. The main advantage of GPUs compared to traditional CPUs for signal processing is that they have much more raw FLOP/S. Not only do they have greater raw performance, in most cases their performance per watt, usually measured as FLOPS/Watt, is 30-50% better [4].

2.2.2 NVIDIA Tesla K20X

A good example of a state-of-the-art GPU, which is designed from the ground up to target GPGPU, is the NVIDIA Tesla K20X. This GPU has been used in several HPC installations including Oak Ridge National Laboratory's (ORNL) Titan supercomputer (see Figure 2.3). A similar Kepler architecture was utilized in the NVIDIA Tegra K1 SOC. The architecture used by the Tesla K20X, and to be discussed in this section, is the Kepler GK110 [5].

Depicted in Figure 2.3 is the overall block diagram of the Kepler architecture. The GPUs interconnect to the main CPU and memory of the system is a 16-lane PCIe Gen II bus with a maximum throughput of 8 GB/s. This link feeds the 6 memory controllers on the Tesla K20X feed 6 GB GDDR5 memory with a peak bandwidth of 250 GB/s. Computation is performed by up to 15 streaming multiprocessors (SMX) (14 for Tesla K20X) which all share a common L2 cache.

Each of the aforementioned SMX processing units (see Figure 2.4) contains 192 single precision CUDA cores, 64 doubleprecision units, 32 special function units (SFU), and 32 load/store units (LD/ST). This hardware is utilized by 4 warp schedulers each capable of dispatching 2 instructions per GPU clock cycle. Each warp is capable executing 32 "threads" simultaneously. Another interesting feature of the SMX processing units is the separate and configurable caches. First, the 48K

Figure 2.3: Kepler Block Diagram

read-only cache is static for the life of a thread. Second, the 64 KB cache can be partitioned between L1 cache and shared memory cache in a 48/16KB spilt in favor of either L1 or shared memory; or additionally in an even 32 KB split between the two caches.

These specifications can be misleading if one does not translate them into classic CPU terms. For example what NVIDIA calls a "warp" is similar to a 32-bit floating-point or integer SIMD instruction [6]. This implies that CUDA cores are more like traditional ALUs rather than having the complexity of a CPU core. This leads to the important realization that special attention must be given when programming for these architectures to use as many of the 32 "threads" of a warp as possible and organize memory accesses to optimize for warp-level locality.

Figure 2.4: SMX Block Diagram

2.3 Heterogeneous Supercomputer Architectures

An interesting case study for how heterogeneous computing is being applied to areas in which purely CPU computing has recently dominated is supercomputers. A good example of this seat change is the upgrade of the Jaguar supercomputer to the Titan supercomputer at ORNL. The Jaguar supercomputer was composed of 18,688 nodes

in 4,672 Cray XT5 blades each of which contains 2 x 2.3 GHz hex-core AMD Opteron processors and 16 GB of DDR2 memory [7]. From Jaguar to Titan the XT5 blades were upgraded to XK7 blades. This change upgraded processors to AMD 16-core 6200 series Opteron with 32 GB of DDR3 memory and paired each processor with a Tesla K20X GPU with 6GB of GDDR5 memory. ORNL says that by relying on its "CPU cores to guide simulations and allowing its new NVIDIA GPUs to do the heavy lifting, Titan will enable researchers to run scientific calculations with greater speed and accuracy" [8].

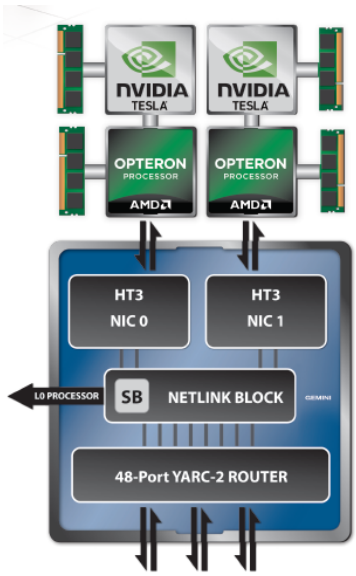


Figure 2.5: XK7 Blade Block Diagram

Shown in Table 2.1 is not only a ten-fold increase in computing power (FLOP/s) but also a four-fold increase in energy efficiency (FLOPS/Watt). This ability of GPUs to drive power efficiency is clear as the first 15 supercomputers on the June 2014 Green 500 [9] are GPU based. From this case study and others, GPUs and, more generally, heterogeneous computers will have a central role in coming years.

Table 2.1: ORNL Jaguar Versus Titan

	Jaguar	Titan
Date of Measurement	November 2011	June 2014
Top500 Rank	3rd	2nd
Performance	1,759.0 TFLOP/S	17,590.0 TFLOP/S
Green500 Rank	149th	43rd
Energy Efficiency	0.58 GFLOPS/Watt	2.1 GFLOPS/Watt

2.4 Mobile SOC Architecture

In general, mobile architectures are very different from that of general-purpose or high-performance computers. First, the entire system is integrated together, usually on a single chip *i.e.* SOC. Inside this chip, it is very common place to integrate many vendors' intellectual property (IP). This IP can include computing resources such as GPUs or DSPs, communication hardware such as LTE modems, IO peripherals such as ADC or display controllers, and much more.

2.4.1 Qualcomm Snapdragon

Qualcomm's Snapdragon SOCs are a dominate force in today's smartphone and tablet marketplaces. This dominance comes from its highly integrated design, particularly with cellular modems for 4G LTE [10]. The tight integration of all parts of the design is critical to Snapdragon's impressive power consumption numbers. This dominance is so prevalent that even competitors like Samsung's consumer-level devices still use Snapdragon SOC inside their designs as in the Samsung Galaxy S4 [11]. This work will focus on two specific implementations of Qualcomm's Snapdragon processor, namely the Snapdragon 600 (sometimes advertised as S4 Pro) and Snapdragon 800.

Table 2.2: Comparison of Snapdragon 600 and 800

	Snapdragon 600	Snapdragon 800
Processor	1.51GHz Quad Kait 300	2.26GHz Quad Kait 400
GPU	Adreno 320	Adreno 330
Memory Interface	2ch 533MHz LPDDR3	2ch 800MHz LPDDR3
Release Date	2013	Q2 2013
Example Device	Nexus 7 Tablet	Nexus 5 Phone

Qualcomm's SOC designs are unique in that they are not simply an integration of an ARM core with many peripherals. In fact, Qualcomm simply licenses the instruction set from ARM and does their own implementation. This allows Qualcomm to tailor their designs to the needs of their SOCs as they deem fit. This holistic design approach continues as Qualcomm purchased AMD's hand-held graphics division in 2009 and rebranded their existing imagination cores to Adreno [12]. Add to this Hexagon DSP IP which can be tailored to various applications such as audio, video, and communication processing and the Snapdragon SOC is an amazingly complex and powerful heterogeneous platform.

An unfortunate fact about Qualcomm's SOCs is that many details are proprietary and therefore not publicly available. It is clear that they believe in Heterogeneous computing and have integrated many different computation devices into their SOC. The first being Krait CPU cores which implements the ARMv7 instruction set. The Krait architecture is the successor of Scorpion and has several improvements including enhanced floating point performance and full out-of-order instruction execution. The floating point improvements can be seen in the improved VeNum SIMD which is very similar to NEON SIMD (see 2.4.2).

The second computing device inside Qualcomm's Snapdragon SOC is the Adreno GPU. The two varieties found in the 600 and 800 series are the Adreno 320 and 330, respectively. In July 2012, Adreno 320 was found to have performance comparable

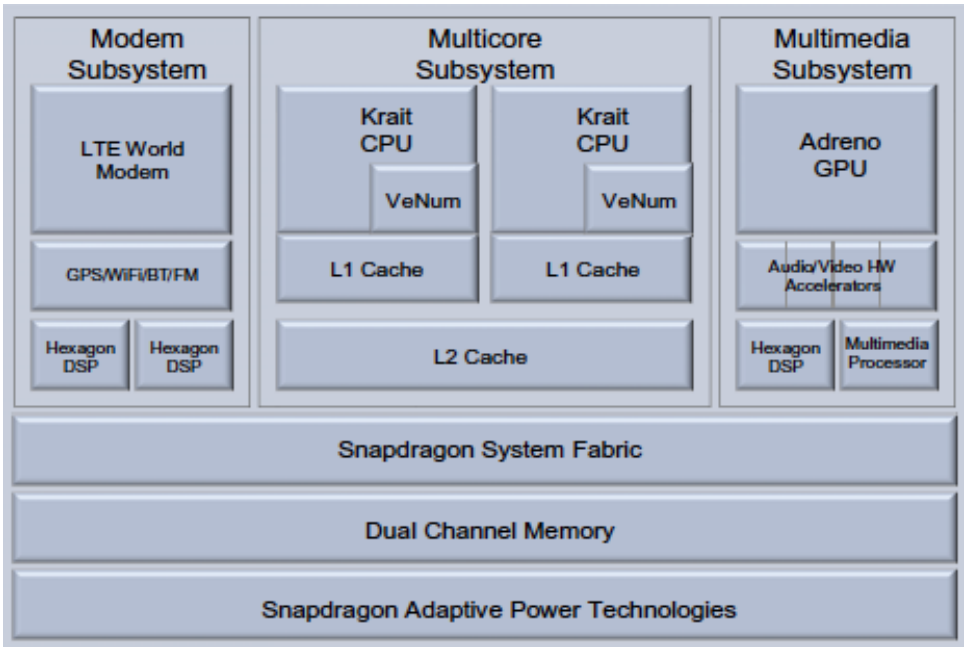


Figure 2.6: Snapdragon S4 Architecture

to PowerVR's GPUs which are integrated into Apple's iPad [13]. The Adreno 330 is claimed to improve performance by 50% over the 320 [14]. Both Adreno 320 and 330 support a Direct X9 unified shader model, which implies a rich programmable pipeline, and both 320 and 330 have OpenCL drivers, [15] although many times these drivers are hidden from end-user devices.

The Hexagon DSP found Qualcomm SOCs serve many purposes such as communication processing and multimedia processing [16]. This DSP provides parallel operation by exposing both VLIW and SIMD. In the communication case, this DSP is preassigned for communications and the user does not have access. On the other hand, the multimedia processor is open for the user to utilize [16]. Qualcomm provides libraries and a SDK for interfacing and many companies are developing device-specific libraries for multimedia applications. It appears there is not currently Renderscript Compute support and therefore this technology was not utilized in the body of this work.

Table 2.3: Comparison of Exynos 4412 Quad and Exynos 5250 Dual

	Exynos 4412 Quad	Exynos 5250 Dual
Processor	1.4GHz Cortex-A7	1.7GHz Cortex-A15
GPU	ARM Mali-400MP4	ARM Mali-T604MP4
Memory Interface	2 ch 400 MHz LPDDR2	2 ch 800 MHz LPDDR3
Release Date	2012	Q3 2012
Example Device	Samsung Note II	Nexus 10 Tablet

2.4.2 Samsung Exynos

Samsung's approach to mobile SOCs is very different to that of Qualcomm's; where Qualcomm brings many parts of the IP under one company, Samsung integrates many different IP vendors into a single SOC. This allows Samsung to get many advanced features from the latest ARM processors and leverage many vendors' domain- specific knowledge into their products. This work will focus on two specific SOCs: the Exynos 4412 Quad and the Exynos 5250 Dual as shown in Table 2.3.

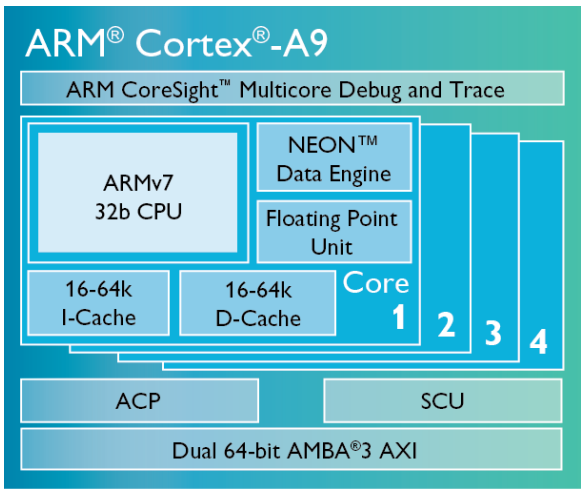


Figure 2.7: Cortex A9 Architecture

The ARM Cortex-A9 found in the Samsung Exynos 4412 shown in Figure 2.7

is a quad core processor supporting the ARMv7 32-bit instruction set. On top of the general ARM instruction set, this processor supports both NEON SIMD and Jazelle RCT. Jazelle RCT provides hardware acceleration for JIT computation of Java byte code [17]. The NEON SIMD can operate on 32 registers holding signed or unsigned integers, 32-bit or 64-bit floating point; additionally, these registers can be interpreted as 16 128-bit float values [18]. These instructions can be used in vectorized instructions for sign/unsigned integers and single precision floating point numbers. A couple of caveats for NEON should be covered for completeness. First, normal floating point operations and NEON floating point operations share the same register set, therefore, the transition from normal FPU operations and NEON operations carries some configuration overhead. Second, NEON floating point operations are not fully IEEE-754 compliant and some operations require software support for compliance [18]. The Coretex A9 CPU core architecture is shown in Figure 2.8.

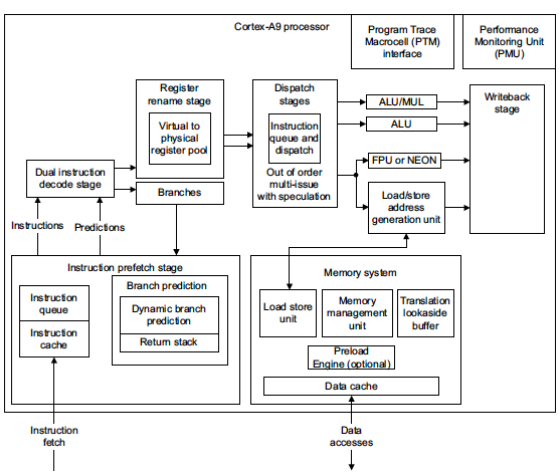


Figure 2.8: Cortex A9 Architecture CPU Core Architecture

These processing elements are fed by a 64-bit AXI bus providing data to and from both memory and peripherals cellular modem or GPU. The memory link is supplemented by a 1 MB L2 cache shared among the cores. Each core of the 4

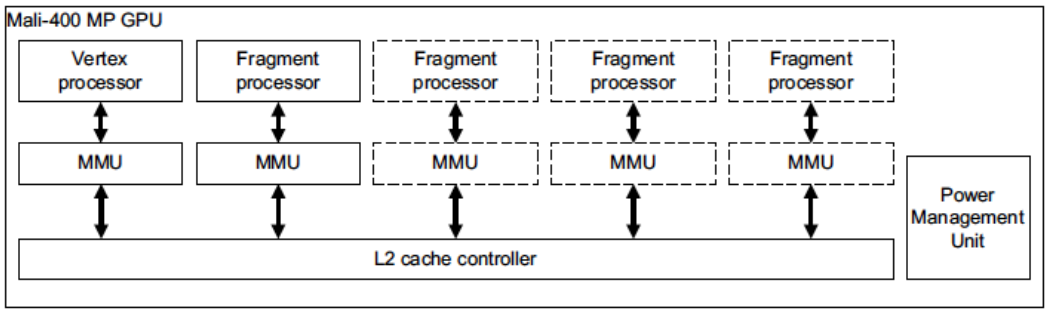


Figure 2.9: Udgard Block Diagram

cores has both a 32 KB instruction and a 32KB data cache. The instructions are dynamically branch predicted and feed to a dual decoder. Finally, these decoded instructions are reordered to maximize ALU utilization.

Also integrated on the Exynos 4412 SoC is the ARM Mali-400MP4. This GPU is based on ARM's Utgard architecture which implements separate vertex and fragment processors [19]. In the Mali-400MP4, there are 4 fragment processors and a single vertex processor. With no unified shader or OpenCL support, it is unlikely that Renderscript Compute is able to utilize this as a compute resource.

Another variant provided by Samsung is the Exynos 5250 which contains dual Cortex-A15 cores shown in Figure 2.10. The A15 is targeted for better overall performance and less power efficiency than the A7. Some enhanced features include ECC on both L2 and L1 cache as well as double the bus width via the AXI bus [20]. The new AXI bus implements some interesting new features from the perspecive of heterogeneous computing, namely Accelerator Coherency Port (ACP). This allows coherency between GPU and CPU L2 caches which when both are involved in a computation has huge performance benefits. It is interesting that the Jazelle RCT is noticeably absent in this version of the Exynos hardware.

The Exynos 5250 also contains an improved Mali-T604MP4 which is based on

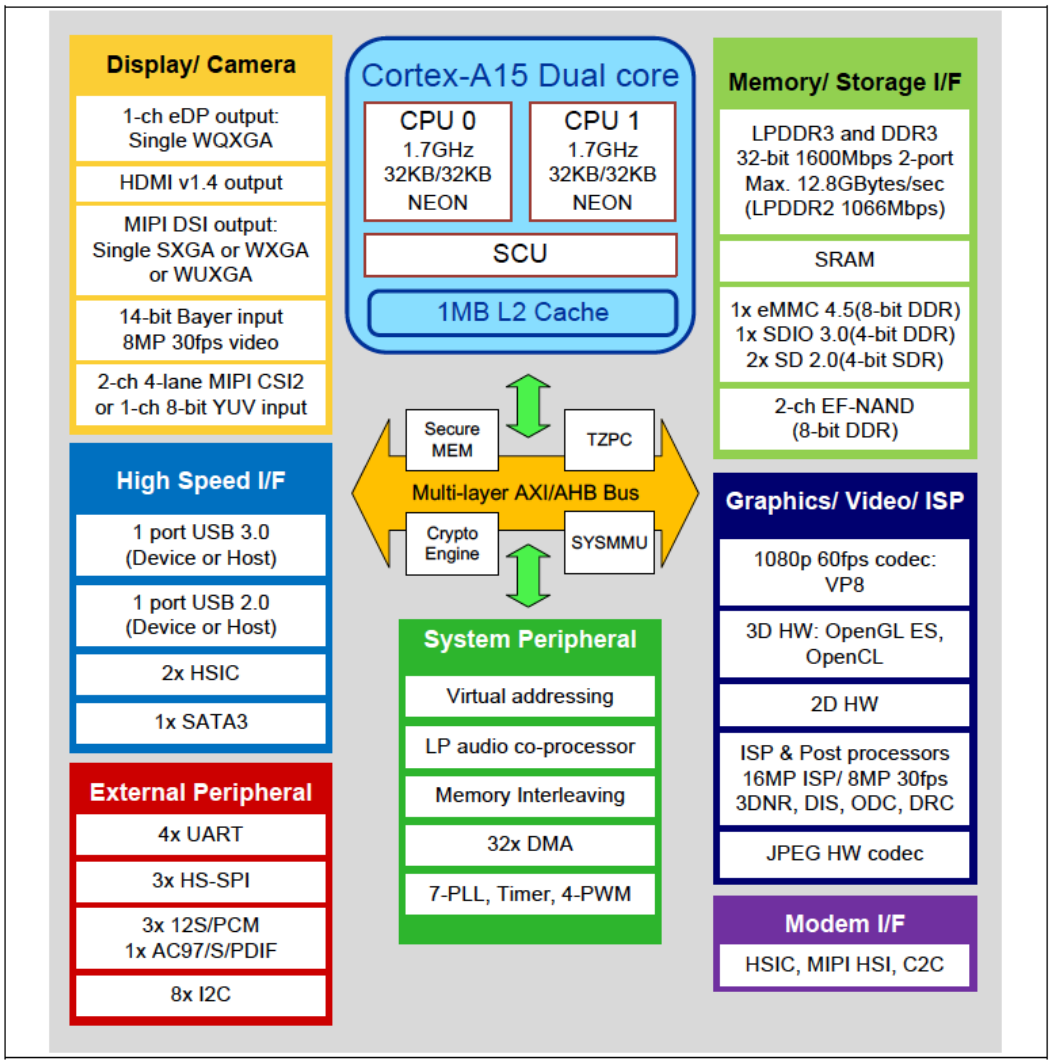


Figure 2.10: Exynos 5250 Block Diagram

the new ARM Midgard architecture. This architecture implements both vertex and fragment shaders via a unified architecture [19]. With the unified shader model and full OpenCL 1.1 support this Mobile GPU has become one of the most widely used heterogeneous computing experiments.

Chapter 3

Heterogeneous Software Framework

One of the biggest challenges to mass adoption of heterogeneous computing is the complexity it adds to the software developer's programming model. Until the last couple decades, heterogeneous computing was limited to a set of experts which either developed for specific applications, like graphics or for custom platforms such as FPGAs or DSPs. In order for heterogeneous computing to be widely adopted it needs to allow development to occur at high-level languages which can target multiple platforms. These high-level languages will allow more developers to develop applications more rapidly than is possible with lower-level languages.

There are many similarities between the evolution of the GPU hardware (see Section 2.2) and the evolution of GPGPU software. Like GPU hardware, originally the only interfaces available to the programmer reflected the special purpose hardware which backed them. As GPUs implemented shader models and later unified shader models, this interface slowly evolved. Early attempts such as BrooksGPU and Sh [1] acted as translators to either OpenGL or DirectX. These third-party tools paved the

way for vendors to begin providing interfaces to expose their parallel programming potential.

The first vendor programming language was NVIDIA's CUDA in 2006 [6] followed quickly by OpenCL which was started by Apple but quickly became a managed by the Khronos working group in 2008 [21]. With these two competing technologies GPGPU programming started to become more common place. Now many high-level tools such as Matlab parallel toolbox and Solidworks design tools use GPUs to accelerate their applications.

3.1 CUDA and OpenCL

CUDA and OpenCL are very similar in programming models and features. In many cases there are one to one relations between CUDA's and OpenCL's programming constructs (see Figure 3.1). CUDA is recognized as a good platform for heterogeneous computing. Its main limitation is that it can only support hardware made by NVIDIA and was not designed with the idea of generic hardware and is instead optimized for NVIDIA architecture.

CUDA term	OpenCL term
host CPU	host
streaming multiprocessor (SM)	compute unit (CU)
scalar core	processing element (PE)
host thread	host program
thread	work-item
thread block	work-group
grid	NDRange
shared memory	local memory
constant memory space	constant memory
texture memory space	constant memory

Figure 3.1: CUDA versus OpenCL contructs

OpenCL, in contrast to CUDA, was designed from the ground up to support different vendors' hardware and even different hardware platforms. OpenCL is standard which is published by a Khornos working group. The Khornos group has about 15 active standards [21] including well known ones such as OpenGL and OpenCL and lesser-known standards like OpenVX (computer vision) or StreamInput (sensor fusion). Its members include NVIDIA, AMD, Altera, and Qualcomm. These members provide technical support to different working groups and out of these standards, APIs, conformance tests, and other supporting documentation/tools have been developed. OpenCL is an open platform for not only application developers, but also other higher level frameworks such as WebCL or Aparapi. Even Renderscript Compute utilizes OpenCL drivers to interact with some GPUs (this can be observed through logging available in the debugger).

To leverage either CUDA or OpenCL one begins by writing the target parallel code in what is known as a kernal (see Figure 3.2). This kernel, like a normal C function, contains input and output parameters. However unlike a normal C function, it is not directly passed an index in these structures. Instead this index is encoded as a thread ID or thread index. This kernel is then invoked on a set of data. This process in OpenCL is called a work group.

In typical programs device memory and host memory are not shared. Data to be operated on in a work group must first be allocated and copied to the device. When the data is copied to the device and the work group is properly setup, then a user needs to simply invoke this kernel. This call is asynchronous allowing the host program to continue with non-parallel tasks or begin setting up subsequent parallel tasks. It is important to note that within these threads it is possible to operate in a data-parallel manner, *e.g.* same function on different data, as above; in addition to a task-parallel manner, *e.g.* different functions on same or different data. Once results are required for program execution the host program executes a synchronization or

```
// Kernel definition
__global__ void VecAdd(float *A, float *B, float *C)
{
    int i = threadIdx.x;
    C[i] = A[i] + B[i];
}

int main()
{
    ...
    // Kernel invocation with N threads
    VecAdd<<<1, N>>>(A, B, C);
    ...
}
```

Figure 3.2: Example Kernel

barrier function to ensure that the parallel computation is ready to be read back. Once this blocking function returns, data can be read back and reported to the user. In addition, to the above steps in both CUDA and OpenCL there are some minor device initialization steps required. In the OpenCL case this can be complex if one wants to target a specific device class or run different kernels based off a device class.

Finally, once both the host program and the device kernels have developed both CUDA and OpenCL, provide compilation and runtime tools. In the CUDA case it is typical to be target a specific device thus the nvcc compiler is invoked to compile kernels into binary cubin objects or into PTX code (NVIDIA assembly) [6]. The host program is then sent through a typical compiler where it links to the NVIDIA's runtime and driver environment. When a program is executed, cubins are loaded to the device via the host program, or in the case of PTX code, the driver may JIT compile into binary cubins and load code onto the device. OpenCL's workflow is similar except it is much more common to leave CL kernels as either C kernel code or as LLVM IR. In both cases the kernel code is JIT compiled by the OpenCL runtime to the target compute device(s) [22], enabling the final binary generation to

take place during runtime allows for code portability.

Both of these runtimes have been used extensively for numerical and scientific computing. As described in [23] code portability given by OpenCL does not translate to performance portability. In this work the authors explored the porting of code originally designed for the NVIDIA fermi architecture to the AMD's TeraScale2 architecture. Their conclusion was that an auto-tuning routine was needed for work group optimization. This performance portability is not a problem unique to OpenCL as the author references several examples of this on both CPU only libraries and CUDA based libraries.

Despite this performance portability issue, CUDA and OpenCL dominate the heterogeneous computing market on HPC and desktop environments. This domination however has not translated in the mobile SOC space. Some of this can be attributed to NVIDIA only releasing CUDA for ARM in June 2013 and with older devices having little to no OpenCL support. This, however, is changing as most devices now at the vendor level have OpenCL support. However, this support does not translate to a reliable API which the programmer can write against.

3.2 RenderScript Compute

One solution to both the performance portability issue and to the lack of universal device support has come from Google in the form of Renderscipt Compute. This framework allows both programming model simplicity and easy Android application deployment. Like OpenCL, it is not vendor specific and is intended to be run on different GPUs and other unique compute cores such as DSPs or processor SIMD extensions. RenderScript Compute guarantees that code written will run on any device running Android versions 2.3 and above. This is accomplished by being able to be run RenderScript Compute on the system processor [24] if no other hardware is

available. Google's main reason for creating a new language, rather than leveraging an existing standard such as OpenCL, is the perceived belief that OpenCL application would lead to only certain devices being optimized for [24].

To provide this device agnostic performance, Renderscipt Compute abstracts work scheduling rather than allowing programmers to choose how work is scheduled. This abstraction allows code to be performance portable with the cost of the programmer not being able to optimize and get the absolute peak performance out of a particular architecture. Another drawback is that Renderscript Compute is currently only available on Android and therefore not applicable to the desktop domain where CUDA and OpenCL dominate. Despite its drawbacks Renderscript Compute is gaining popularity in the mobile space.

As with typical Android applications, everything starts from the MainActivity class. A developer can simply call a Java implementation directly from this entry point. Once this is configured the application can be packaged into an Android package (APK) and subsequently loaded onto a device. The application then runs on DalvikVM, Google's implementation of the Java virtual machine [24].

Next, the algorithm is rewritten in RenderScript compute. This involves RenderScript (rs) files, which are written in a C99-like syntax [25]. This is non-trivial as the algorithm must be refactored in a parallel way. It is beyond the scope of this study to describe the techniques required to convert a serial implementation to a parallel one. Once a rs file is written, it is analyzed and a reflected ScriptC Java class is generated by the Android compiler. This allows the exposed RenderScript implementations to be called from the MainActivity. Once the application is ready for packaging all of the Java code is packaged as normal. The rs file(s) are packaged as well in the APK. When an application is run, the rs files are JIT compiled and the reflected interface is implemented for target hardware.

In general it is not known to the application whether the rs code is being executed on a GPU, DSP, or on the main CPU. This maintains the single APK for any supported device and saves the programmer from being concerned with device compatibility.

A more detailed look at Renderscript Compute implementation is done in Section 5.1 and results of signal processing algorithms performance may be found in Chapter 6. However, comparisons between Renderscipt Compute and OpenCL have been conducted by [26]. In this work, code is generated for both OpenCL and Renderscript Compute and compared. It can be seen that Renderscript Compute and OpenCL trade performance benefits back and forth with OpenCL code leading to the conclusion that Renderscript Compute holds its own in performance versus other frameworks and gives the benefit of portability. Another interesting study of Renderscipt Compute is done in [27] where Renderscript Compute is compared to pure Java SDK and Android NDK (Native C/C++ Android implementation). In this study, it is found that Renderscript Compute can provide performance on par with NDK developed applications while providing better portability.

Chapter 4

Signal Processing Algorithms

Signal processing is the theory and practice of transforming signals from real world sensors and processing it such that a useful output is obtained. In the context of this work the focus is on digital signal processing. To simplify this broad and rich area both the DFT and FFT were chosen to represent the class of computations and workloads, which a platform must perform to implement these algorithms.

Both the DFT and FFT take time domain data and convert it to the frequency domain. This transform has applications including communications, image processing, and mathematics. Many algorithms can be expressed in terms of DFT (and hence FFTs) such as discrete cosine transform, fast convolution, and partial differential equations [28].

4.1 Discrete Fourier Transform

The discrete Fourier transform (DFT) algorithm was chosen to start benchmarking mobile SOC performance. The formulation is as follows:

$$X_k = \sum_{n=0}^{N-1} x_n \cdot e^{-j2\pi kn/N}, k = 0, 1, ..., N-1 \tag{4.1}$$

The above formulation has a computational complexity of $O(N^2)$ where N is the number of point DFT taken [29]. In practical applications, a fast Fourier transform (FFT) is preferred over a DFT due to it superior computational complexity of $O(nlog_2(n))$ [30]. This work begins with the DFT as there are many competing implementations of FFT which have large platform dependent performance. This adds another element of uncertainty to the results. Notice that although the above has the time complexity of $O(N^2)$, each inner sum has no data dependency to any other inner sum therefore making this algorithm embarrassingly parallel.

4.2 Fast Fourier Transform

The FFT is defined as any algorithm which calculates the DFT transform in a faster method than presented in the previous section. There is a large array of implementations available for the FFT. In general, these algorithms have $O(nlog_2(n))$ and match DFT results with no adverse effects. This work focuses on two algorithms: a classic decimation in time (DIT) radix-2 FFT and a radix-2 Stockham FFT. In addition, the assumption of is the input size of the data is a power of 2. While these algorithms have adaptations that allow for efficient computation of any size, such a comprehensive implementation is beyond the scope of this work [2].

Most FFT algorithms are derivations and reformulations on the ideas presented here. First 4.1 is rewritten as

$$X_k = \sum_{n=0}^{N-1} x_n \cdot W_N^{-kn}, k = 0, 1, ..., N-1 \tag{4.2}$$

where

$$W_N = e^{-j2\pi kn/N} \tag{4.3}$$

W_N terms are known as twiddle factors and some important identities can be observed.

$$\begin{aligned} W_N^2 &= W_{N/2} \\ W_N^{k+N/2} &= -W_N^k \\ W_{N/2}^{k+N/2} &= W_N^k \end{aligned} \tag{4.4}$$

The critical concept is the divide and conquer strategy which is composed of 3 steps [28]:

1. Divide the problem into two or more subproblems of smaller size.

2. Solve each subproblem recursively by the same algorithm. Apply the boundary condition to terminate the recursion when the sizes of the subproblems are small enough.

3. Obtain the solution for the original problem by combining the solutions to the subproblems.

In the case of the DIT radix-2 FFT the division begins by splitting the even and odd indices into two separate sums.

$$\begin{aligned} X_k &= \sum_{n=0}^{N/2-1} x_{2n} \cdot W_N^{-2nk} + \sum_{n=0}^{N/2-1} x_{2n+1} \cdot W_N^{-(2n+1)k} \\ &= \sum_{n=0}^{N/2-1} x_2 n \cdot W_N^{-2nk} + W_N^{-k} \cdot \sum_{n=0}^{N/2-1} x_{2n+1} \cdot W_N^{-2nk} \\ &= \sum_{n=0}^{N/2-1} x_{2n} \cdot W_{N/2}^{-nk} + W_N^{-k} \cdot \sum_{n=0}^{N/2-1} x_{2n+1} \cdot W_{N/2}^{-nk} \\ &= \sum_{n=0}^{N/2-1} y_n \cdot W_{N/2}^{-nk} + W_N^{-k} \cdot \sum_{n=0}^{N/2-1} z_n \cdot W_{N/2}^{-nk} \end{aligned} \tag{4.5}$$

Notice this leaves two equations in the same form as 4.2. Namely

$$\begin{aligned} Y_k &= \sum_{n=0}^{N/2-1} y_n \cdot W_N^{-kn} \\ Z_k &= \sum_{n=0}^{N/2-1} z_n \cdot W_N^{-kn} \end{aligned} \tag{4.6}$$

which allows recursive division of the problem until sums become single values. Once the problem has been recursively divided the solutions can be combined as follows. By substituting Y_k and Z_K in 4.5 it can be obtained

$$X_k = Y_k + W_N^{-k} \cdot Z_k \tag{4.7}$$

Furthermore by applying the identities in 4.4 the following can be applied to reduce the computations by 2.

$$\begin{aligned} X_{k+N/2} &= \sum_{n=0}^{N/2-1} y_n \cdot W_{N/2}^{(k+N/2)n} + W_N^{k+N/2} \cdot \sum_{n=0}^{N/2-1} z_n \cdot W_{N/2}^{(k+N/2)n} \\ &= \sum_{n=0}^{N/2-1} y_n \cdot W_{N/2}^{kn} - W_N^k \cdot \sum_{n=0}^{N/2-1} z_n \cdot W_{N/2}^{kn} \\ &= Y_k - W_N^{-k} \cdot Z_k \end{aligned} \tag{4.8}$$

These combinations recursively applied up the chain yield the final frequency result. Together these two equations are known as a butterfly and are often represented by the following graphical notation

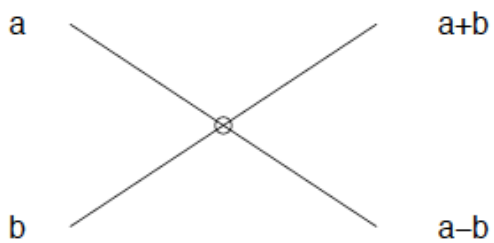


Figure 4.1: Single Butterfly

Applying this notation to an 8-point FFT yields the flow graph representation shown in Figure 4.2.

An important consideration that arises out of this graphical representation is that a reorder is required to output a natural order FFT result. This reorder is referred to as bit-reversal. A table showing the reversal required for an 8-point FFT is shown in Figure 4.3.

With the above concepts combined a DIT radix-2 FFT can be implemented. These same concepts with the butterflies applied in reverse yields a decimation in frequency (DIF) FFT. Also, if the problem is divided into 4 parts rather than 2 a

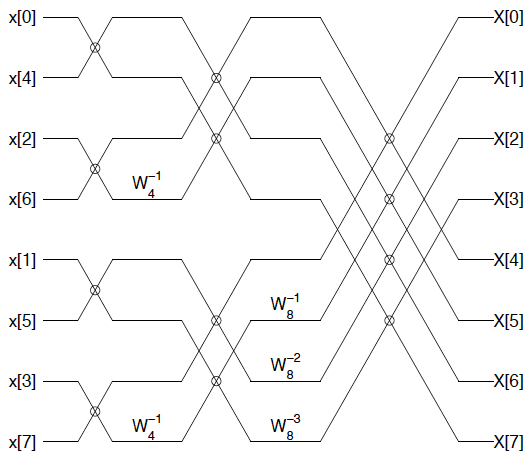


Figure 4.2: 8 point FFT Flow Diagram

n	binary	bit-rev	n'
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

Figure 4.3: 8 point FFT Bit Reversal

radix-4 algorithm emerges. In more complex libraries often a combination of algorithms and techniques are used to optimize a given solution [2]. A more complete derivation may be found in [28] or [30].

A variation on the above radix-2 FFT is done by the Stockham autosort algorithm. The basic idea is to avoid bit-reversal by integrating this piecewise while doing butterfly operations. The cost of this is doubling the size of the memory re-

quired since at each butterfly stage the calculation is done out-of-place rather than in-place as above. Derivation of this algorithm is described in both [28] and [31]. The Stockham autosort algorithm implemented in this work is a radix-2 DIF FFT. This implementation is referred to as "stockham FFT" in the rest of the text.

Chapter 5

Evaluation Methodology

The following sections outline the methodology used for evaluating Renderscript Compute's performance on signal processing algorithms. In order to test on mobile devices, an app is developed to benchmark three algorithms: DFT, radix-2 FFT, and Stockham FFT. Next, a summary of devices used is presented and some differences highlighted. Then the methodology of testing these devices, both for runtime performance and for power consumption, is explained. Finally, a brief explanation of units chosen and calculation methods employed is provided.

5.1 App Implementation

The app was developed utilizing a combination of Java, Android libraries, and Renderscript Compute framework. The Android user interface code itself is kept very simple with few buttons or controls. This interface allows one to configure a test and then dispatch an AsyncTask to do the actual processing thus allowing the MainActivity thread to respond to the Android OS. Otherwise, Android will detect this as a non-responsive program and prompt the user to close the application. Inside this

AsyncTask is where all of the computation setup, execution, and result saving occur.

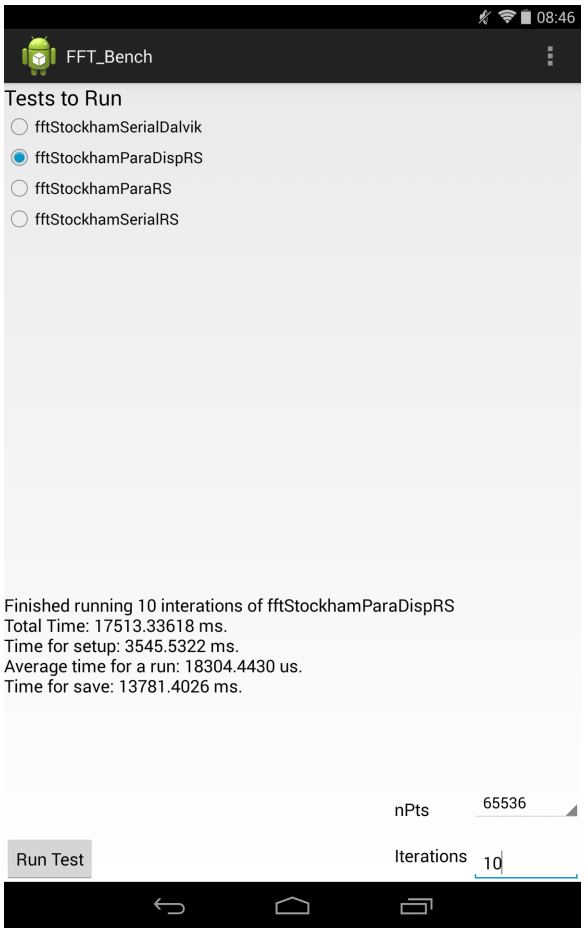


Figure 5.1: Application Interface

The simple user interface shown in Figure 5.1 shows that different tests can be selected and two parameters varied: the number of points FFT (nPts) and the number of iterations. When running quick tests, a text summary is displayed showing the average time taken per iteration. For benchmarking, a more automated function is available which automates data collection by iterating over the following: a subset of algorithms and all power of two input sets between limits. In addition, it performs each of these combinations many times. This data on a per run basis is exported for later analysis.

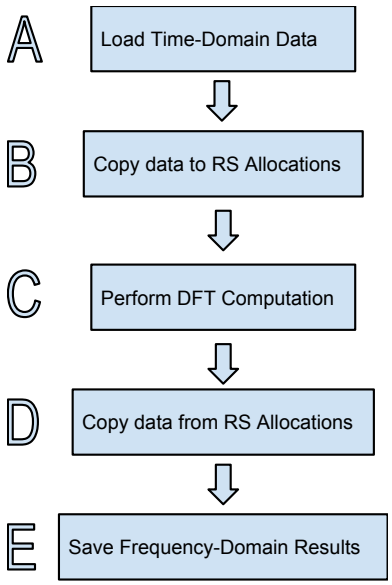


Figure 5.2: Application Steps

The general test harness where the computation takes place is outlined in Figure 5.2. The process begins by loading the time domain data (step A). For testing this data is loaded from a comma separated value (csv) file, but in real implementations could be read from a sensor such as a microphone. This data, if the application is implemented in RenderScript Compute, needs to be loaded into structures known as allocations (step B). Next, the Fourier transform computation is done (step C). Then, if using RenderScript Compute, the data is copied back from allocations into normal Java data structures (step D). Finally, the frequency data is saved to a csv file for off-line verification of algorithm correctness. To understand performance implications, step C has been written in several different implementations.

Only steps B through D are counted as part of the execution time (in the case of a Java-only implementation only step C is considered). The benchmarking is done in a very conservative way where the data is fully read back before the next computation is started. This prevents algorithms from gaining advantage by streaming data. In a library implementation, streaming of results to hide latency is a common practice

but was considered an additional complexity not desired for comparisons done in this work.

5.1.1 DFT Implementations

The first implementation of the DFT is named dftSerialDalvik. In this implementation, computation is done in a serial manner via Android Java virtual machine, Dalvik. The second and third implementations, dftJavaThreads_2 and dftJavaThreads_4 are strictly in Java again except they use Java's threading implementation to take advantage of the multicore processor. The computation is evenly divided by the number of threads. As the name suggests, the only difference between these implementations is the number of threads utilized.

The fourth implementation, dftSerialRS, is the same serial algorithm as the first but is implemented in RenderScript Compute. In this case the time domain data must be copied to RenderScript Compute via the allocation structure. Once this is complete, a single non-blocking invocation of the serial RenderScript Compute kernel is done. The AsyncTask is then blocked when the data is read out from the allocation. This implementation allows for the Renderscript JIT compiler to utilize data parallel constructs like SIMD instructions but does not allow any task parallelism.

The fifth implementation, dftPartialParlRS part C is divided between Java and Renderscript Compute such that the inner real and imaginary calculations are done with both task and data parallelism. The task parallelism is enabled in Renderscipt Compute by invoking a for_each call with both input and output allocation structures. This for_each call allows independent and parallel calculation of the inner loop iterations. Special attention needs to be taken when using these constructs as they can create data race conditions if calculations are not independent. These calculations are retrieved from the resulting allocation and summed in the Java code. The

summing is done for each frequency bin in the resulting frequency data.

The sixth RenderScript Compute implementation is dftFullParlRS where all computation is executed in RenderScript Compute including both inner real and imaginary computations and the summation of the frequency bins. Allowing the summation to be done in Renderscript Compute has the advantage that only the final summed frequency value needs to copied back to Java rather than N values.

5.1.2 Radix-2 FFT Implementations

In the radix-2 implementations many of the names and concepts carry over from DFT implementations therefore each one is only discussed briefly. There are two Dalvik implementations: the first one, fftRecurSerialDalvik, utilizes recursion to make the code more compact. This implementation is useful only for understanding the mechanics of the code. A more optimized Dalvik implementation, fftRad2SerialDalvik, is presented which includes optimizations such as calculating twiddle factors outside the loops and referencing them.

The optimized Dalvik implementation was translated to Renderscript Compute in fftRad2SerialRS to allow data parallelism. In fftRad2FullParaRS, each stage's outer loop is implemented using the for_each construct to allow task parallelism. Finally, to minimize interactions between Renderscript Compute and the Java runtime, fftRad2DispParaRS allows Java to make a single invocation to Renderscript Compute. In Renderscript Compute C code a similar for_each methodology is utilized to implement the same algorithm as fftRad2FullPara.

5.1.3 Stockham FFT Implementation

Building on the concepts explored in both DFT and Radix-2 FFT, implementations Stockham FFTs are implemented. First, fftStockhamSerialDalvik implements the algorithm in a straight-forward manner including not precalculating twiddle factors. This implementation is then translated to Renderscipt Compute, in fftStockhamSerialRS, to allow data parallelism in addition to optimizations such as twiddle precalculation. Next, the serial implementation is translated into a task parallel as well as data parallel implementation in fftStockhamParaRS. Finally, fftStockhamParaDispRS restructures the algorithm to allow for a single for_each invocation instead of two nested loops; with the additional complexity of calculating the previous indexes inside the actual kernel.

5.2 Devices Evaluated

The devices selected are meant to represent the capability of high-end smartphones available in late 2013 [11]. The Nexus 4 is utilized to show the generational improvements from 2012 to 2014 [32]. It should be noted that many of the Renderscript Compute functions may be implemented using CPU rather than either GPU or SIMD due to limited driver support. Another challenge is that most of the smartphones released in the United States have Qualcomm chips due to the company's IP dominance in 4G LTE which made it difficult to find newer Samsung SOCs; however, the Note II has an Exynos SOC, and although not the newest, can be used for comparison purposes.

The devices are evaluated with software versions which are up to date or near up to date; it should be noted that Nexus devices get updates months ahead of other devices therefore are usually at least one OS version ahead. In Section 6.2 the

Table 5.1: Devices for Evaluation

Name	Nexus 4	Nexus 5	Nexus 7 (2013)	Galaxy S4	Galaxy Note 2
SOC	Snapdragon S4 Pro	Snapdragon 800	Snapdragon S4 Pro	Snapdragon 600	Exynos 4412 Quad
Processor	Quad 1.51 GHz Krait 300	Quad 2.26 GHz Krait 400	Quad 1.51 GHz Krait 300	Quad 1.9 GHz Krait 300	Quad 1.6 GHz Cortex-A9
GPU	Adreno 320	Adreno 330	400 MHz Adreno 320	Adreno 320	ARM Mali-400MP
Memory	2 GB RAM	2 GB RAM	2 GB RAM	2 GB RAM	2 GB RAM
OS Version	4.2.2	4.4.2/4.4.4	4.4.2/4.4.4	4.3/4.4.2	4.1.2/4.4.2

difference of OS versions used in the study are shown to not be critical to performance characteristics. Since the software is implemented within the Android sandbox no rooting or custom configuration is required; devices are left in a configuration which one would expect a typical user to have their phone *i.e.*, utilizing carrier/vendor provided Android versions.

5.3 Physical Setup

5.3.1 Run-Time Measurements

The run time of each test is measured and averaged over many iterations and tested with different power of two input sizes. The device under test (DUT) is put into airplane mode *i.e.*, there is no network connectivity and restarted to minimize software running in the background. Additionally, the device's display brightness is set to a minimum, externally powered and externally air cooled to prevent thermal throttling. The automated testing of many algorithms versus varying input sizes can take hours to ensure that plenty of samples are taken to minimize OS software's effect on average run time.

5.3.2 Power Consumption Measurements

Measuring the power of these highly integrated consumer devices can only conveniently be done at the system level, which means measuring the power consumed at the USB power input. In order to minimize the amount power consumed, a similar setup to the above is utilized. Also, the devices are fully charged and left sitting idle for a minimum of 30 minutes before the tests are conducted to ensure as stable of power state as possible.

The current is measured via a logging multimeter at a rate of 1 Hz and is averaged over a number of iterations of the algorithm. The voltage has been verified to stay a constant 4.97V which is approximated to 5V. The voltage and current are multiplied for measured power. Next, the difference is then taken from the measured power by the idle power to give delta power. Both power numbers are listed in Appendix A. This measurement is then used in derived units such as joules per calculation or MFLOPS/W.

5.4 Units

The primary unit chosen for comparing performance is FLoating point OPerations per Second (FLOP/S). The number of FLOP/S done per point is determined by counting the number of operations of floating point operations then dividing by the execution time. For example in Figure 5.3 a DFT implementation is shown.

Here it can be assumed that addition, multiplication, sin, and cos functions all take a single FLOP. In addition, it is assumed that the compiler has done constant propagation so that 2π is precomputed. The number of FLOPs per point is thus calculated to be 24. Finally, it can be seen that the algorithm complexity of a DFT is $O(N^2)$, therefore, for a given input size of N the $FLOPs = 24N^2$. Similarly radix-2

```
for (int k = 0; k < nPts; k++)
{
  outReal[k] = 0;
  outImag[k] = 0;
  for (int t = 0; t < nPts; t++)
  {
    outReal[k] +=  inReal[t]*cos(2*PI*t*k/nPts)
                  +inImag[t]*sin(2*PI*t*k/nPts);
    outImag[k] += -inReal[t]*sin(2*PI*t*k/nPts)
                  +inImag[t]*cos(2*PI*t*k/nPts);
  }
}
```

Figure 5.3: C99 Implementation of DFT algorithm

FFT and stockham FFT can both be shown to both have $5Nlog_2(N)$ FLOPs.

This unit of measure is an industry standard but is not without its flaws. In order to be consistent, this number must stay constant regardless of the architecture ran on. Thus, even though a SIMD architecture might be able to do four floating-point additions in a single instruction, the FLOPs are counted as four FLOPs so that the speedup can be observed between implementations. In addition this number is keep constant for any given implementation. Finally this number does not directly express any integer math (required usually for index calculation) or memory usage. Even though these are not directly expressed by this unit, these latencies are accounted for since they add execution time and therefore lower effective FLOP/S. Finally, the results are presented in terms of MFLOP/S which equates to millions of FLOPs per second.

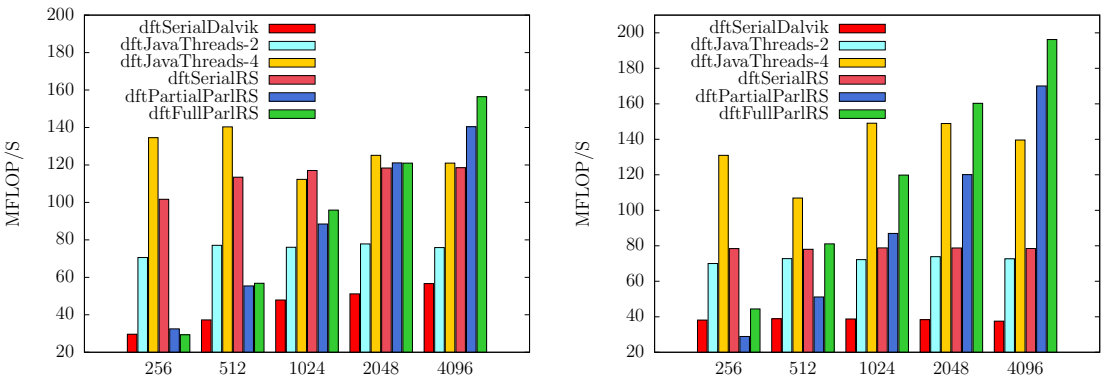
Other units are available but are used less in the main body of text. These are joules per calculation and average runtime. Average runtime is synonymous with execution time and is the average time taken for a particular algorithm to compute a given input size. Joules per calculation is equally straight forward since it is the

delta power multiplied by average runtime. Both of these calculations are available in Appendix A.

Chapter 6

Results

6.1 DFT Results

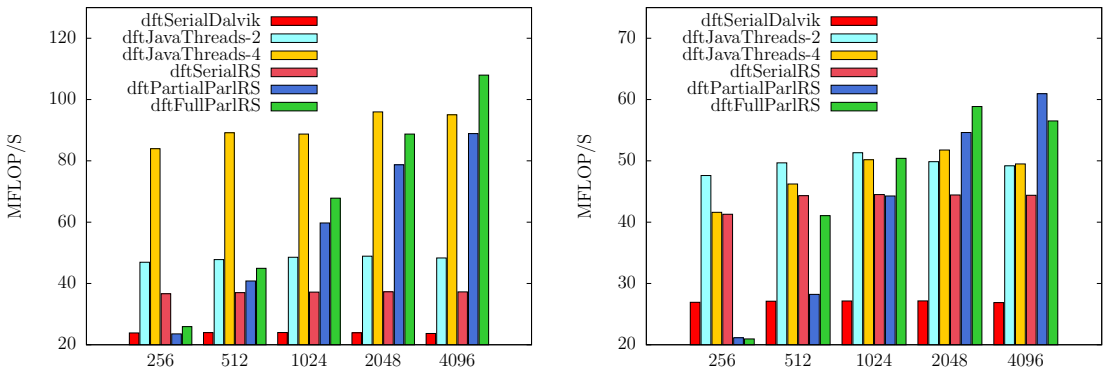


(a) Nexus 5

(b) Nexus 7

Figure 6.1: MFLOP/S versus DFT Implementation

The results shown in the Figures 6.1a, 6.1b, 6.2a, and 6.2b display the MFLOP/S for the different implementations across different devices. In all cases the RenderScript Compute implementations require a large number of points to amortize the overhead of copying into and out of allocations. In these larger cases Renderscript



(a) Note II (b) Nexus 4

Figure 6.2: MFLOP/S versus DFT Implementation

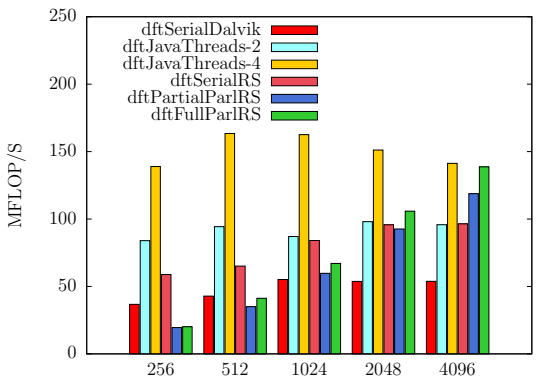


Figure 6.3: GalaxyS4 MFLOP/S versus DFT Implementation

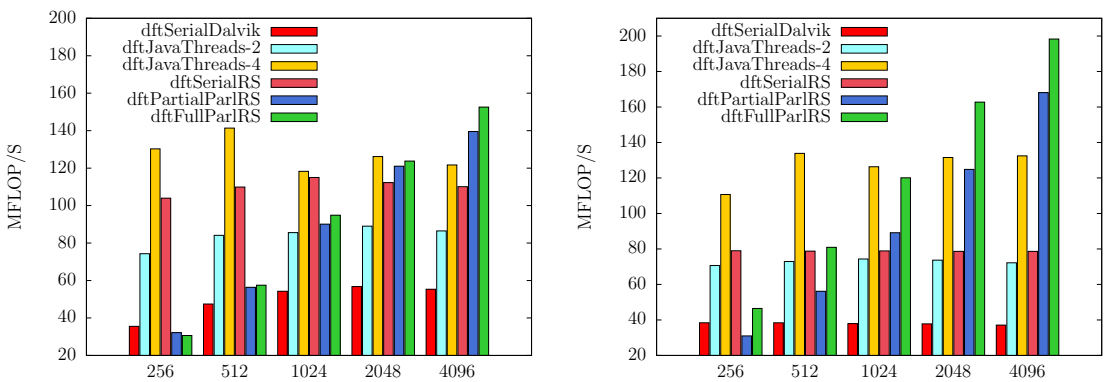
Compute enables energy efficient computing.

The Nexus 5 and Nexus 7 show fairly similar results which can be attributed to their similar Snapdragon chipsets. The Nexus 7 achieves the best performance which can be attributed to its higher clocked GPU. However, achieving this peak performance comes at the cost of energy efficiency. The results here show a larger than expected overhead for RenderScript Compute when dealing with smaller DFT calculations.

One outlier that can be observed is the Galaxy S4 where the dftJavaThreads_4 consistently outperforms other implementations. It is not known whether this is a driver issue within the S4 specific build of Android or some unknown hardware limitation. It is worth noting that although RenderScript Compute was not able to improve overall performance it did improve energy efficiency in the dftPartParlRS case.

Finally, the Note II, which is the only non-Snapdragon processor, showed significant power saving when using RenderScript Compute. This could be due to NEON's SIMD instructions are more power optimized for these floating point operations. This could also be a result of more efficient memory transfers since both CPU and NEON share cache and memory. Another explanation for this impressive performance comes from both the Note II and the GalaxyS4 having significantly higher idle power. This higher idle power may bias the delta power and therefore artificially inflate the power savings.

6.2 Effect of Android Versions



(a) Nexus 5 OS 4.4.4

(b) Nexus 7 OS 4.4.4

Figure 6.4: MFLOP/S versus DFT Implementation

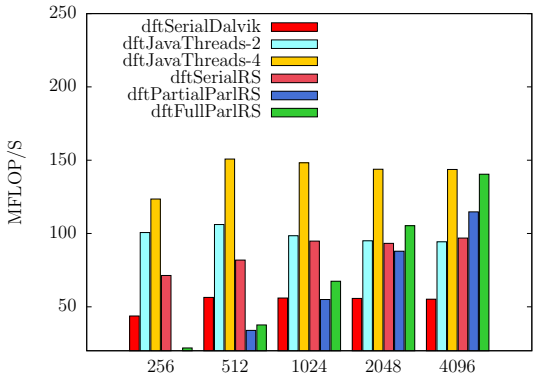


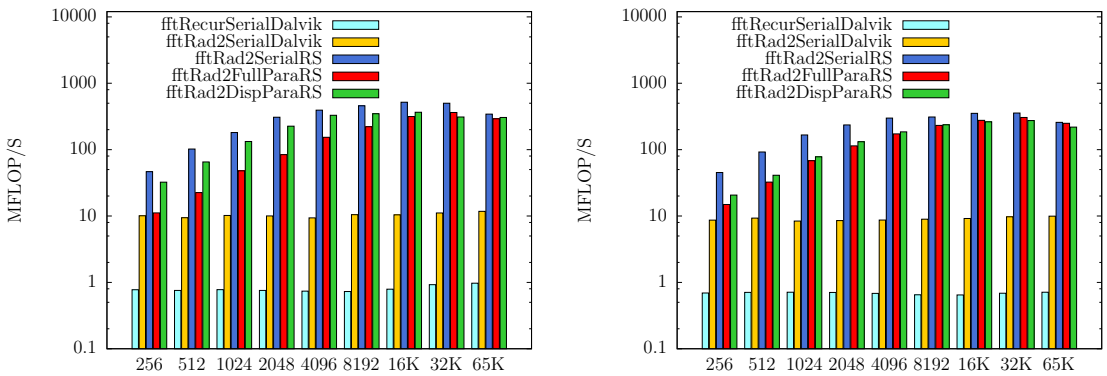
Figure 6.5: Galaxy S4 OS 4.4.2 MFLOP/S versus DFT Implementation

In Figures 6.4a, 6.4b, and 6.5 DFT results are shown from different Android versions than originally developed on to if any performance differences can be seen. It is clear that the performance has remained constant. This is quite different from the results presented in [33] where performance of Google's internal benchmarks improve over 2.5 times from Android version 4.0 to 4.2. One conclusion which can be drawn is that results from Android version 4.3 to 4.4.4 onwards are very comparable. Even though not all permutations were run on every OS version, the performance appears to stay stable.

6.3 Radix-2 FFT Results

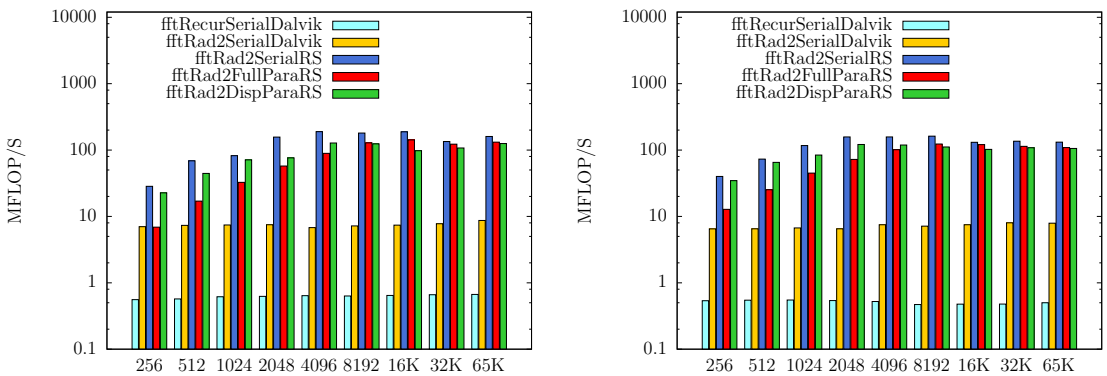
Both Radix-2 FFTs and Stockham have surprisingly good results compared to DFTs. Since the number of FLOPs is significantly reduced by the change in time complexity from N^2 to $Nlog_2(N)$ one would expect the performance in MFLOP/S to stay constant. This is only the case when looking at Dalvik implementation which actually reduces performance.

The more interesting case is when FFTs are implemented in Renderscript Com-



(a) Nexus 5 (b) Nexus 7

Figure 6.6: MFLOP/S versus Radix-2 FFT Implementation



(a) Note II (b) Nexus 4

Figure 6.7: MFLOP/S versus Radix-2 FFT Implementation

pute, the serial implementation MFLOP/S increases by 2 times. This can be explained by the nature of the FFT algorithms chosen. The accumulation steps are moved from the inner-most loop of the algorithm to occurring at each stage of the divide and conquer strategy. This allows the data dependency from the accumulation to be moved to the highest level *i.e.*, the $log_2(N)$ loop. Thus inside of each loop there are N operations which have no data dependency, thus data parallelism can take full advantage of the deep pipelines available in both CPU and GPU SIMD instructions.

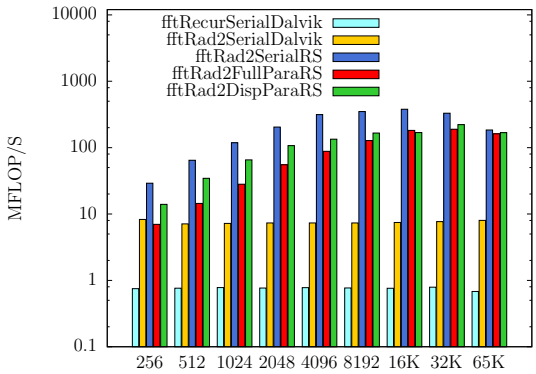
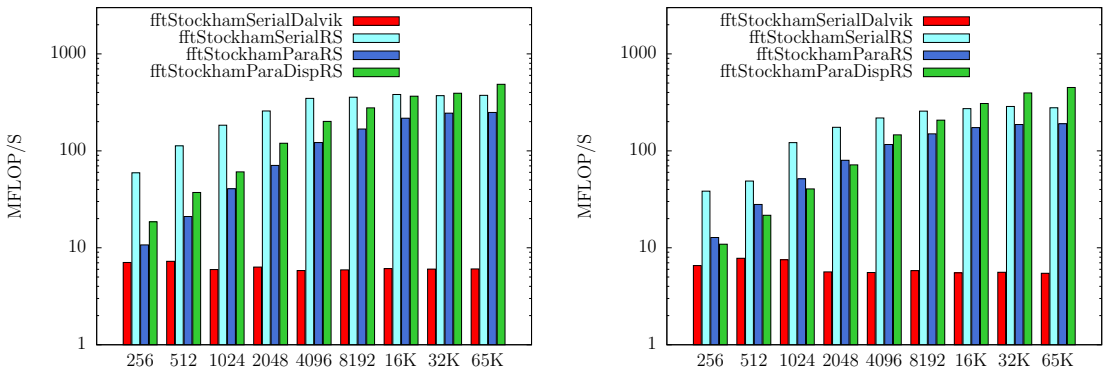


Figure 6.8: Galaxy S4 MFLOP/S versus Radix-2 FFT Implementation

This data parallelism is very dominate, so much so that only in very large point FFT does task parallelism show an advantage. This effect is so dominate that task parallelism actually hinders performance in all Radix-2 FFT cases. This is due to the overhead of synchronization and added complexity in coding does not provide the needed advantage to amortize the cost. In cases larger than 16K, performance starts to decrease. Although a single factor does not stand out, some contributing factors may be bit-reversal having a larger effect on performance, and memory accesses exceed some bound. In addition, it can be seen in the power numbers that initiating task parallelism has a significant energy cost.

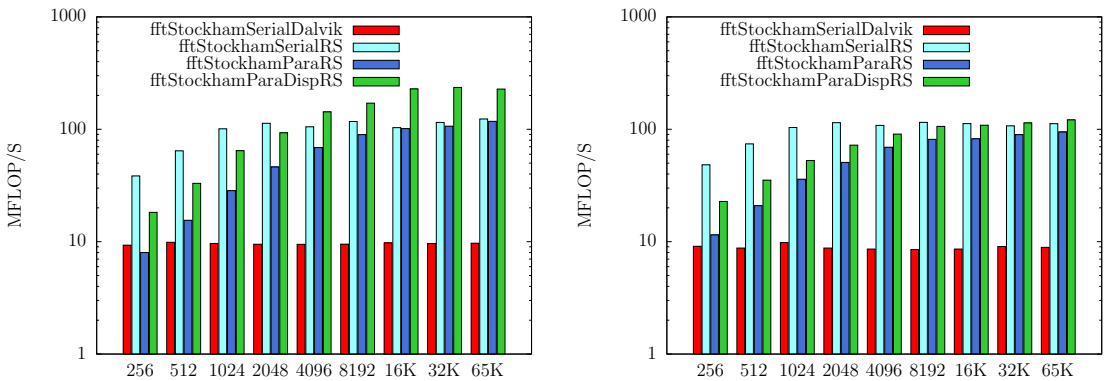
6.4 Stockham FFT Results

The Stockham FFT results are similar to the Radix-2 FFT results in terms of impressive speedup when compared to DFT results. Again, the data parallelism dominates performance overall, but with no bit-reversal and easier memory strides, task parallelism starts to show some advantage around 16K point FFT. Until this point there are not enough computations to amortize the additional cost and complexity of a task-parallel implementation brings. This task parallelism in addition to extra mem-



(a) Nexus 5 (b) Nexus 7

Figure 6.9: MFLOP/S versus Stockham FFT Implementation



(a) Note II (b) Nexus 4

Figure 6.10: MFLOP/S versus Stockham FFT Implementation

ory transition affect the power domain very heavily and are a significant detriment to energy-efficient performance.

The main advantage of the Stockham algorithm is that it does not require bit-reversal. The downside being that twice the memory is required to allow out-of-place computation within each stage. This access pattern is more convenient for coding as Renderscript Compute for_each statements naturally have an input allocation and an output allocation thus allowing for Stockham's task parallel nature to be

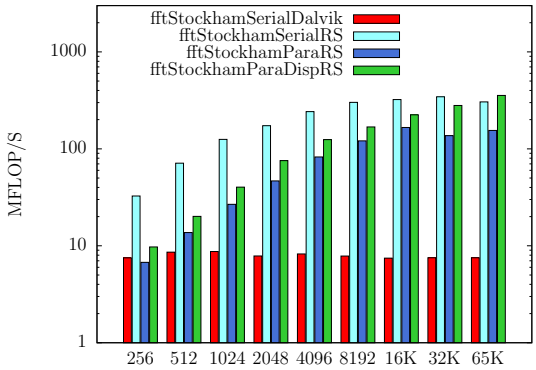


Figure 6.11: Galaxy S4 MFLOP/S versus Stockham FFT Implementation

directly expressed by input paramters rather than using an indirect index. This is also a contributing factor in this implementation being better suited for task parallel implementation. Although, the peak performance reached by radix-2 16K was not exceeded, the Stockham FFT showed very good performance with larger point FFT datasets. This implementation seemed to scale better than radix-2 and allowed Renderscript Compute to utilize both task-parallel and data-parallel abilities effectively.

Chapter 7

Conclusion

These results clearly show that use of the RenderScript Compute on mobile SOCs running Android can improve performance of signal processing applications, but great care must be taken in the implementation approach to in order to reach optimal solutions. In addition, it shows that use of GPU and SIMD hardware can improve energy efficiency as well as performance of digital signal processing workloads as demonstrated by both DFT and FFT implementations. In the cases examined here, task-parallel implementations are important only when computing very large FFTs.

The best FFT performance results were obtained with the Nexus 5 during a 16K point radix-2 SerialRS FFT. In this case, over 500 MFLOP/S were obtained. In the energy efficiency realm, the Nexus 5 with the same SerialRS FFT implementation obtained about 280 MFLOPS/W putting it just barely ahead of the Note II. These numbers are impressive and show that with improved data-parallelism, newer architectures with improved SIMD and GPUs can pull away from older platforms. With newer generations implementing more complex GPUs and additional DSP resources in SOCs these numbers should get better over time on new hardware with little to no code changes needed.

When comparing DFT numbers it is also clear, even with non-optimized algorithms, a 30% overall energy savings is obtained. Specifically, this number is calculated by comparing Java Threads to Renderscript Compute numbers.

7.1 Comparison of Results

Looking at the results presented in this work in comparison to other systems these power numbers seem very promising. One interesting comparison is between the results presented versus the June 2014 Green500 list which ranks the most energy-efficient supercomputers [9]. In terms of performance per watt, would rank 409th spot in the DFT case or 284th spot in the FFT case. This is not a direct comparison as this paper is not running the same benchmarks nor are all the same factors being considered such as cooling, network communication, or AC power supply efficiency.

Other efforts to offload processing from the CPU to heterogeneous processors have yielded similar results. In [34] the authors explore offloading machine learning classification to a DSP located inside a SOC. This lead to a 17% overall power reduction, but when system overhead is excluded the offloaded classification algorithm savings becomes 50%. In the case of DFTs the results presented here fall slightly short at 40% but in the FFT case it is likely this number is slightly higher but since threaded versions of FFT algorithms were not measured a direct number cannot be computed.

Another comparison with similar workloads can be found in [4] where several implementations of the FFT are examined with desktop CPUs and GPUs. This study finds that GPUs can be as high as 5-8 GFLOPS/Watt and for Intel CPUs can range from 1 to 4 GFLOPS/Watt. This study focused on datasets where the number of point FFT was 64K or greater. These results show that Renderscript Compute running on Mobile SOCs is competitive but still slightly lower than tradi-

tional CPUs and GPUs. This suggests that where the high performance capabilities of a supercomputer or desktop CPU are not feasible or required that a SOC solution is ideal given its comparable performance per watt and its smaller overall footprint-/form factor. As mobile SOC's performance improve this current energy-efficiency gap should close making this architecture more relevant for larger workloads.

One main advantage of the results presented in this work is as expected heterogeneous computing performance does seem to portable across devices. This is not the case in the higher performing CPU/GPU implementations shown in [2], [4], and [23]. This advantage is largely gained by employing Renderscript Compute but is nonetheless important when considering implementation details for an algorithm. It is unfortunate that this advantage is only available on Android and not realized in a more generalized way. Additional testing may be needed on additional non-Snapdragon processors to confirm this result.

7.2 Future Work

To continue the work presented, here the FFT algorithm should be explored in more depth and its dominant performance drivers understood. This exploration is probably easier in a framework like OpenCL which can be controlled with much finer granularity than is possible in Renderscript Compute. Comparing this work with other heterogeneous frameworks such as OpenCL, NDK, Aparapi, or even CUDA would be interesting. Another avenue for further research is to investigate larger radixes and combining different radixes; in [4] a 35-55% reduction in power is possible when employing multiple radixes.

Another avenue for future work would be to implement these algorithms with real-world sensors such as a microphone or a software-defined radio. This would provide an interesting case study and uncover more implementation details which are not

always clear when working on benchmarking exclusively. A less ambitious effort might be to convert the implementations presented here into a more traditional FFT framework where the optimal algorithm can be chosen given input parameters such as size and deadline. Then such parameters such as latency and energy efficiency can be balanced against the performance requirements of the application. Since performance is fairly portable the heuristics involved would be much simpler than in FFTW where benchmarking is performed on target systems to pick the optimal implementation [2].

Appendices

Appendix A

Tabular Performance Results

Table A.1: Nexus 5 Power Calculations for 4096 DFT

	Power (Watt)	Delta From Idle	Energy for Calc (Joule)	MFLOPS/Watt
idle	0.44	N/A	N/A	N/A
dftSerialDalvik	2.26	1.82	12.90	31.20
dftJavaThreads-2	3.44	3.00	15.92	25.29
dftJavaThreads-4	3.9	3.46	11.52	34.95
SerialRS	2.12	1.76	5.97	67.49
dftPartParlRS	3.35	2.91	8.35	48.24
dftFullParlRS	3.60	3.16	8.12	49.56

Table A.2: Nexus 5 MFLOP/S across DFT Implementations

npts	dftSerialDalvik	dftJavaThreads-2	dftJavaThreads-4	dftSerialRS	dftPartialParlRS	dftFullParlRS
256	29.648709	70.563661	134.547819	101.671881	32.483767	29.333532
512	37.278284	77.091729	140.340308	113.482251	55.426447	56.817990
1024	47.892940	76.038869	112.312331	117.039457	88.487426	95.895378
2048	51.172411	77.849500	125.117514	118.352219	121.107444	120.990993
4096	56.663193	75.894972	120.978269	118.513155	140.421062	156.423626

Table A.3: Nexus 5 Power Calculations for 65K FFT

	Power (Watt)	Delta From Idle	Energy for Calc (Joule)	MFLOPS/Watt
idle	0.46	N/A	N/A	N/A
fftRecurDalvik	1.66	1.20	6.49	0.81
fftRad2SerialDalvik	1.73	1.27	0.57	9.27
fftRad2SerialRS	1.67	1.21	0.02	283.23
fftRad2FullParaRS	1.92	1.46	0.03	199.60
fftRad2DispParaRS	1.87	1.41	0.02	215.80
fftStockhamSerialDalvik	1.49	1.03	0.89	5.86
fftStockhamSerialRS	1.94	1.48	0.02	252.62
fftStockhamParaRS	1.96	1.50	0.03	166.06
fftStoickhamDispRS	2.80	2.34	0.03	207.36

Table A.4: Nexus 5 MFLOP/S across Radix-2 FFT Implementations

npts	fftRecurSerialDalvik	fftRad2SerialDalvik	fftRad2SerialRS	fftRad2FullParaRS	fftRad2DispParaRS
256	0.775061	10.090321	46.514478	11.130071	32.340249
512	0.756804	9.426928	101.999638	22.553625	65.130047
1024	0.776327	10.202388	180.704456	47.985428	132.509169
2048	0.755632	10.015599	307.576366	84.110747	225.357674
4096	0.737877	9.367130	393.474297	153.470041	329.427433
8192	0.728129	10.460356	456.793807	221.098483	347.493516
16K	0.788805	10.418416	517.257874	315.314850	365.735163
32K	0.922688	11.105336	499.999267	362.526434	310.476559
65K	0.971227	11.746795	342.423775	291.355003	304.864570

Table A.5: Nexus 5 MFLOP/S across Stockham FFT Implementations

npts	fftStockhamSerialDalvik	fftStockhamSerialRS	fftStockhamParaRS	fftStockhamParaDispRS
256	7.063551	59.335522	10.711526	18.542883
512	7.252408	112.582993	21.021827	37.213651
1024	5.964673	183.901152	40.670520	60.654786
2048	6.333529	257.708621	70.696285	119.677093
4096	5.821344	347.861345	121.683543	201.105041
8192	5.921162	357.438948	167.900823	277.657179
16K	6.101891	381.194753	216.900712	365.746175
32K	6.028334	370.820613	244.783325	392.690804
65K	6.057563	373.522957	248.601028	485.761244

Table A.6: Nexus 7 Power Calculations for 4096 DFT

	Power (Watt)	Delta From Idle	Energy for Calc (Joule)	MFLOPS/Watt
idle	0.47	N/A	N/A	N/A
dftSerialDalvik	1.44	0.97	10.43	38.61
dftJavaThreads-2	2.15	1.69	11.91	43.10
dftJavaThreads-4	3.82	3.36	11.02	41.60
dftSerialRS	1.48	1.01	7.57	77.55
dftPartParlRS	2.89	2.42	6.83	70.23
dftFullParlRS	3.08	2.61	6.31	75.14

Table A.7: Nexus 7 MFLOP/S across DFT Implementations

npts	dftSerialDalvik	dftJavaThreads-2	dftJavaThreads-4	dftSerialRS	dftPartialParlRS	dftFullParlRS
256	38.157787	70.029564	130.962864	78.407976	28.875785	44.418639
512	38.893769	72.750416	106.906644	78.009374	51.179175	81.106820
1024	38.763168	72.199403	149.086635	78.823015	86.991683	119.842964
2048	38.362537	73.876438	148.923419	78.765656	120.091737	160.304636
4096	37.511348	72.681080	139.626389	78.446471	170.070952	196.233355

Table A.8: Nexus 7 Power Calculations for 65K FFT

	Power (Watt)	Delta From Idle	Energy for Calc (Joule)	MFLOPS/Watt
idle	0.46	N/A	N/A	N/A
fftRecurDalvik	1.69	1.22	9.01	0.58
fftRad2SerialDalvik	1.75	1.29	0.68	7.71
fftRad2SerialRS	1.66	1.20	0.02	214.57
fftRad2FullParaRS	1.88	1.42	0.03	175.62
fftRad2DispParaRS	1.88	1.42	0.03	153.57
fftStockhamSerialDalvik	1.79	1.33	1.27	4.12
fftStockhamSerialRS	1.95	1.49	0.03	186.62
fftStockhamParaRS	2.00	1.53	0.04	124.01
fftStockhamDispRS	2.95	2.49	0.03	180.95

Table A.9: Nexus 7 MFLOP/S across Radix-2 FFT Implementations

npts	fftRecurSerialDalvik	fftRad2SerialDalvik	fftRad2SerialRS	fftRad2FullParaRS	fftRad2DispParaRS
256	0.691231	8.670396	45.178984	14.921702	20.638720
512	0.706520	9.303562	92.013984	32.444122	41.075882
1024	0.711745	8.396543	166.275680	68.086587	77.982784
2048	0.704886	8.525422	235.755459	114.123663	131.604775
4096	0.682244	8.686473	298.681983	172.568114	185.430556
8192	0.649510	8.955872	310.330007	232.065446	236.535864
16K	0.646939	9.172333	352.082780	276.436312	262.501057
32K	0.685993	9.737022	355.982145	305.159349	274.189786
65K	0.711709	9.952480	257.385957	249.436212	218.481670

Table A.10: Nexus 7 MFLOP/S across Stockham FFT Implementations

npts	fftStockhamSerialDalvik	fftStockhamSerialRS	fftStockhamParaRS	fftStockhamParaDispRS
256	6.559622	38.387407	12.763192	10.900667
512	7.794736	48.827753	27.993130	21.641815
1024	7.529932	121.609277	51.485963	40.517825
2048	5.644437	174.812333	79.875945	71.723976
4096	5.557881	218.352638	116.276295	146.281038
8192	5.808150	256.966829	149.778569	207.133500
16K	5.518352	272.213389	173.704478	307.841347
32K	5.589455	286.920807	186.946221	395.852439
65K	5.461713	278.177763	190.339680	450.548613

Table A.11: Nexus 4 Power Calculations for 4096 DFT

	Power (Watt)	Delta From Idle	Energy for Calc (Joule)	MFLOPS/Watt
idle	0.81	N/A	N/A	N/A
dftSerialDalvik	2.05	1.24	18.50	21.76
dftJavaThreads-2	2.87	2.06	16.82	23.94
dftJavaThreads-4	2.53	1.72	13.95	28.86
dftSerialRS	2.06	1.25	11.29	35.65
dftPartParlRS	2.82	2.01	13.24	30.40
dftFullParlRS	2.08	1.27	9.01	44.68

Table A.12: Nexus 4 MFLOP/S across DFT Implementations

npts	dftSerialDalvik	dftJavaThreads-2	dftJavaThreads-4	dftSerialRS	dftPartialParlRS	dftFullParlRS
256	26.932603	47.604843	41.610159	41.293358	21.146330	20.949174
512	27.100823	49.675926	46.219924	44.321634	28.226731	41.058905
1024	27.144085	51.330540	50.177103	44.496391	44.266282	50.407259
2048	27.149429	49.871583	51.774873	44.438836	54.618074	58.858819
4096	26.875563	49.200107	49.496457	44.387643	60.961229	56.515107

Table A.13: Nexus 4 Power Calculations for 65K FFT

	Power (Watt)	Delta From Idle	Energy for Calc (Joule)	MFLOPS/Watt
idle	0.83	N/A	N/A	N/A
fftRecurDalvik	2.17	1.33	13.98	0.38
fftRad2SerialDalvik	2.14	1.31	0.87	6.05
fftRad2SerialRS	2.02	1.19	0.05	110.19
fftRad2FullParaRS	2.04	1.21	0.06	89.81
fftRad2DispParaRS	2.27	1.44	0.07	73.37
fftStockhamSerialDalvik	2.22	1.39	0.82	6.42
fftStockhamSerialRS	2.30	1.47	0.07	76.47
fftStockhamParaRS	2.41	1.58	0.09	59.75
fftStockhamDispRS	2.41	1.58	0.07	76.74

Table A.14: Nexus 4 MFLOP/S across Radix-2 FFT Implementations

npts	fftRecurSerialDalvik	fftRad2SerialDalvik	fftRad2SerialRS	fftRad2FullParaRS	fftRad2DispParaRS
256	0.536700	6.493987	40.050647	12.792387	34.549457
512	0.546964	6.511659	73.121039	25.242393	65.241506
1024	0.549607	6.700139	116.686716	44.898483	83.928044
2048	0.540441	6.504367	157.391477	72.247641	121.334237
4096	0.522087	7.491612	157.378613	101.190753	119.085882
8192	0.471002	7.142956	161.786075	123.415982	111.398229
16K	0.477458	7.489439	130.832439	120.559229	102.374225
32K	0.477724	8.020720	136.006439	113.756852	108.295561
65K	0.500693	7.902821	131.401426	108.857433	105.604190

Table A.15: Nexus 4 MFLOP/S across Stockham FFT Implementations

npts	fftStockhamSerialDalvik	fftStockhamSerialRS	fftStockhamParaRS	fftStockhamParaDispRS
256	9.102963	48.265870	11.490457	22.809076
512	8.777755	74.104311	20.901847	35.254481
1024	9.807452	103.748786	35.942448	52.700537
2048	8.770024	114.459874	50.748478	72.258957
4096	8.598758	108.369739	69.179641	90.573417
8192	8.515318	115.160446	81.355835	106.025538
16K	8.597892	112.295618	82.299067	108.585688
32K	9.043073	107.292354	89.517276	113.904559
65K	8.899769	112.243590	94.613380	121.257910

Table A.16: Note II Power Calculations for 4096 DFT

	Power (Watt)	Delta From Idle	Energy for Calc (Joule)	MFLOPS/Watt
idle	0.74	N/A	N/A	N/A
dftSerialDalvik	1.21	0.47	7.99	50.39
dftJavaThreads-2	1.49	0.745	6.21	64.84
dftJavaThreads-4	1.97	1.225	5.19	77.57
dftSerialRS	1.24	0.495	5.35	75.26
dftpartParlRS	1.24	0.50	2.26	177.81
dftFullParlRS	1.53	0.785	2.93	137.56

Table A.17: Note II MFLOP/S across DFT Implementations

npts	dftSerialDalvik	dftJavaThreads-2	dftJavaThreads-4	dftSerialRS	dftPartialParlRS	dftFullParlRS
256	23.860194	46.923150	83.975654	36.654952	23.559976	25.920633
512	23.954676	47.782000	89.177264	37.006388	40.803269	44.935762
1024	23.991214	48.555488	88.733909	37.174758	59.730903	67.826925
2048	23.914782	48.890360	95.981327	37.284912	78.714535	88.731563
4096	23.682152	48.307561	95.028577	37.254692	88.903183	107.983208

Table A.18: Note II Power Calculations for 65K FFT

	Power (Watt)	Delta From Idle	Energy for Calc (Joule)	MFLOPS/Watt
idle	0.79	N/A	N/A	N/A
fftRecurDalvik	1.40	0.61	4.80	1.09
fftRad2SerialDalvik	1.45	0.65	0.39	13.31
fftRad2SerialRS	1.39	0.60	0.02	266.04
fftRad2FullParaRS	1.41	0.62	0.02	212.58
fftRad2DispParaRS	1.44	0.65	0.03	192.73
fftStockhamSerialDalvik	1.30	0.51	0.27	19.08
fftStockhamSerialRS	2.02	1.23	0.05	100.32
fftStockhamParaRS	2.14	1.34	0.06	87.51
fftStockhamDispRS	2.32	1.53	0.04	148.74

Table A.19: Note II MFLOP/S across Radix-2 FFT Implementations

npts	fftRecurSerialDalvik	fftRad2SerialDalvik	fftRad2SerialRS	fftRad2FullParaRS	fftRad2DispParaRS
256	0.557557	7.000386	28.379421	6.893453	22.627244
512	0.570747	7.337801	68.900037	17.036027	44.431193
1024	0.616802	7.425077	82.245119	32.606138	71.026712
2048	0.625058	7.496950	156.849889	57.438280	76.314356
4096	0.640530	6.774709	189.102917	89.088453	127.705957
8192	0.633012	7.202211	180.395384	128.510206	124.029491
16K	0.645960	7.389627	188.359342	142.402822	97.692685
32K	0.661341	7.759719	134.274288	122.560005	107.516235
65K	0.667552	8.688884	159.991233	131.614122	125.323712

Table A.20: Note II MFLOP/S across Stockham FFT Implementations

npts	fftStockhamSerialDalvik	fftStockhamSerialRS	fftStockhamParaRS	fftStockhamParaDispRS
256	9.306673	38.456580	8.001913	18.225543
512	9.873537	64.307555	15.498847	33.036438
1024	9.636432	100.992351	28.456846	64.501286
2048	9.493388	113.142532	46.144382	93.007267
4096	9.471609	105.243897	68.725339	142.881520
8192	9.495371	117.309748	89.638932	170.950777
16K	9.727916	103.382986	101.449586	228.589329
32K	9.620879	115.113213	106.632293	235.508739
65K	9.687646	123.504353	117.617224	227.803493

Table A.21: Galaxy S4 Power Calculations for 4096 DFT

	Power (Watt)	Delta From Idle	Energy for Calc (Joule)	MFLOPS/Watt
idle	0.75	N/A	N/A	N/A
dftSerialDalvik	2.80	2.05	15.34	26.24
dftJavaThreads-2	3.65	2.90	12.19	33.03
dftJavaThreads-4	3.70	2.95	8.41	47.89
dftSerialRS	2.60	1.85	7.73	52.12
dftPartParlRS	2.40	1.65	5.59	72.01
dftFullParlRS	3.15	2.40	6.97	57.80

Table A.22: Galaxy S4 MFLOP/S across DFT Implementations

npts	dftSerialDalvik	dftJavaThreads-2	dftJavaThreads-4	dftSerialRS	dftPartialParlRS	dftFullParlRS
256	36.740575	83.975654	138.945583	58.820643	19.492676	20.118496
512	42.869011	94.338821	163.456898	65.101987	35.016731	41.252744
1024	55.120519	87.021764	162.548921	84.110374	59.726650	67.094550
2048	53.767097	98.033069	151.137013	95.725761	92.636356	105.828800
4096	53.798490	95.797957	141.283802	96.424245	118.816707	138.715342

Table A.23: Galaxy S4 Power Calculations for 65K FFT

	Power (Watt)	Delta From Idle	Energy for Calc (Joule)	MFLOPS/Watt
idle	0.76	N/A	N/A	N/A
fftRecurDalvik	1.93	1.17	9.01	0.58
fftRad2SerialDalvik	1.77	1.01	0.66	7.93
fftRad2SerialRS	1.83	1.07	0.03	172.98
fftRad2FullParaRS	1.72	0.95	0.03	169.39
fftRad2DispParaRS	1.81	1.04	0.03	160.91
fftStockhamSerialDalvik	1.74	0.98	0.68	7.69
fftStockhamSerialRS	1.90	1.13	0.02	268.98
fftStockhamParaRS	1.72	0.96	0.03	161.62
fftStockhamDispRS	2.73	1.97	0.03	180.72

Table A.24: Galaxy S4 MFLOP/S across Radix-2 FFT Implementations

npts	fftRecurSerialDalvik	fftRad2SerialDalvik	fftRad2SerialRS	fftRad2FullParaRS	fftRad2DispParaRS
256	0.749878	8.273401	29.094278	6.991235	13.986258
512	0.761638	7.104508	64.549822	14.420849	34.478455
1024	0.778319	7.219143	118.407904	28.114784	65.423553
2048	0.767618	7.332787	204.215311	55.172536	107.019267
4096	0.776401	7.342273	314.339499	88.059745	134.025625
8192	0.768441	7.334499	349.518333	127.538628	165.678871
16K	0.760087	7.445567	378.485530	181.871241	168.458780
32K	0.790696	7.669912	329.713880	189.250961	222.012618
65K	0.681305	8.019037	184.463197	161.516548	167.952578

Table A.25: Galaxy S4 MFLOP/S across Stockham FFT Implementations

npts	fftStockhamSerialDalvik	fftStockhamSerialRS	fftStockhamParaRS	fftStockhamParaDispRS
256	7.517179	32.624630	6.748815	9.723667
512	8.598311	71.116685	13.707622	20.109065
1024	8.728223	125.184421	26.772438	40.269828
2048	7.853967	173.367191	46.658755	75.670655
4096	8.245793	242.190120	82.479630	124.379323
8192	7.844008	301.394055	120.800508	168.260763
16K	7.465871	322.295666	165.907918	224.526159
32K	7.524284	344.302515	136.715288	279.920598
65K	7.530650	304.908734	154.861770	355.595901

Appendix B

Selected Sample Source Code Snippets

B.1 dftFullParlRS

B.1.1 dftFullParlRS.java

```
1  ...
2  public void run ()
3  {
4      int n = inReal.length;
5      mRS_inReal.copyFrom(inReal);
6      mRS_inImag.copyFrom(inImag);
7
8      for (int k = 0; k < n; k++)
9      {  // For each output element
10         m_Script.set_k(k);
11         m_Script.forEach_sumkern(mRS_count, mRS_sumOut);
12
13         m_Script.bind_sumIn(mRS_sumOut);
14         m_Script.invoke_sumFloat2();
15     }
16     mRS_outReal.copyTo(outReal);
17     mRS_outImag.copyTo(outImag);
18 }
19 ...
```

B.1.2 dftFullParl.rs

```
1  #pragma version(1)
2  #pragma rs java_package_name(com.example.dftFullParl)
3
4  float *inReal;
5  float *inImag;
6
7  int nPts;
8  int k;
9
10 float2* sumIn;
11 float* outReal;
12 float* outImag;
13
14 void sumkern (const int *in, float2 *out)
15 {
16     int t = *in;
17
18     out->x =  inReal[t]*cos(2*PI * t * k / nPts) + inImag[t]*sin(2*PI * t * k / nPts); // Real
19     out->y = -inReal[t]*sin(2*PI * t * k / nPts) + inImag[t]*cos(2*PI * t * k / nPts); // Imag
20 }
21
22 void sumFloat2 ()
23 {
24     int i;
25     int length=nPts;
26     outReal[k] = 0;
27     outImag[k] = 0;
```

```
28
31      for (i=0;i<length; i++)
32      {
33          outReal[k] += sumIn[i].x;
34          outImag[k] += sumIn[i].y;
35      }
36  }
```

B.2 fftRad2SerialRS

B.2.1 fftRad2SerialRS.java

```
1   ...
2   public void run ()
3   {
4       // Copying incoming data is done in bitRevOrder
5
6       m_Script.invoke_bitRevOrder();
7       m_Script.invoke_calcTwiddle(outReal.length);
8       m_Script.invoke_fftRad2Serial();
9
10      mRS_outReal.copyTo(outReal);
11      mRS_outImag.copyTo(outImag);
12  }
13  ...
```

B.2.2 fftRad2Serial.rs

```
1   #pragma version(1)
2   #pragma rs java_package_name(com.example.fftRad2Serial)
3
4   #include "mathUtils.rsh"
5
6   float *inReal;
7   float *inImag;
8   float *outReal;
9   float *outImag;
10  float2 *twiddle;
11  float2 *currTwid;
12
13  void bitRevOrder ()
14  {
15      unsigned int N = rsAllocationGetDimX(rsGetAllocation(inReal));
16      unsigned int nBits = uiLog2(N);
17      unsigned int inIdx;
18      for (int i=0;i<N;i++)
19      {
20          inIdx = bitRevIdx(i, nBits);
21          outReal[i] = inReal[inIdx];
22          outImag[i] = inImag[inIdx];
```

```
23      }
24  }
27
28  void calcTwiddle (unsigned int N)
29  {
30      for (int k=0; k<N/2; k++)
31      {
32          twiddle[k].x = (cos(2*k*PI/N));
33          twiddle[k].y = (sin(2*k*PI/N));
34      }
35  }
36
37  void fftRad2Serial ()
38  {
39      unsigned int N = rsAllocationGetDimX(rsGetAllocation(outReal));
40      unsigned int nStages = uiLog2(N);
41      unsigned int L;
42      float2 a,b,cmplxTmp,out;
43
44      // Both bitRevOrder and calcTwiddle should have been called from java
45      for (int stage = 0; stage < nStages; stage++)   // each power of two
46      {
47          L = 1<<(stage+1);
48
49          for (int k=0;k<N;k+=L) // Might need to stop at N-L
50          {
51              for (int n=0;n<L/2;n++)
52              {
53                  a.x = outReal[n+k];
54                  a.y = outImag[n+k];
55
56                  b.x = outReal[n+k+L/2];
57                  b.y = outImag[n+k+L/2];
58
59                  cmplxTmp = cmplxMult(b, twiddle[n*(N/L)]);
60
61                  // Butterflies
62                  out = a+cmplxTmp;
63                  outReal[n+k] = out.x;
64                  outImag[n+k] = out.y;
65
66                  out = a-cmplxTmp;
67                  outReal[n+k+L/2] = out.x;
68                  outImag[n+k+L/2] = out.y;
69              }
70          }
71      }
72  }
```

B.3 fftStockhamSerialRS

B.3.1 fftStockhamSerialRS.java

```
1  ...
2  public void run ()
3  {
4  mRS_inCmplx.copyFrom(inCmplx);
5
6  m_Script.invoke_fftStockhamSerial();
7
8  mRS_outCmplx.copyTo(outCmplx);
9  }
10 ...
```

B.3.2 fftStockhamSerial.rs

```
1  #pragma version(1)
2  #pragma rs java_package_name(com.example.fftStockhamSerial)
3
4  #include "mathUtils.rsh"
5
6  float2 *inCmplx;
7  float2 *outCmplx;
8
9  void fftStockhamSerial ()
10 {
11     unsigned int N=rsAllocationGetDimX(rsGetAllocation(outCmplx));
12     unsigned int nStages = uiLog2(N);
13     unsigned int L0,r0,L,r;
14     unsigned int omegaIdx;
15     float2 b;
16
17     for (int stage = 0; stage < nStages; stage++) // each power of two
18     {
19         // inner loop invariants
20         L = 1<<(stage+1);
21         r=N/L;
22         L0=L/2;
23         r0=N/L0;
24
25         // Copy inCmplx->outCmplx
26         for (int idx=0;idx<N;idx++)
27         {
28             outCmplx[idx] = inCmplx[idx];
29         }
30
31             for (int j=0; j<L0;j++)
32         {
33                 omegaIdx = ((float)j/(float)L)*N;
34             for (int k=0;k<r;k++)
35             {
36                 b = cmplxMult(omega[omegaIdx], outCmplx[j*r0+k+r]);
```

```
37
40                  inCmplx[j*r+k] = outCmplx[j*r0+k]+b;
41                  inCmplx[(j+L0)*r+k] = outCmplx[j*r0+k]-b;
42              }
43          }
44      }
45      for (int idx=0;idx<N;idx++)
46      {
47          outCmplx[idx] = inCmplx[idx];
48      }
49  }
```

B.4 fftStockhamParaDispRS

B.4.1 fftStockhamParaDispRS.java

```
1   ...
2   public void run ()
3   {
4       mRS_inCmplx.copyFrom(inCmplx);
5
6       m_Script.invoke_fftStockhamParaDisp(m_Script,mRS_inCmplx, mRS_outCmplx);
7
8       mRS_inCmplx.copyTo(outCmplx);
9   }
10  ...
```

B.4.2 fftStockhamParaDisp.rs

```
1   #pragma version(1)
2   #pragma rs java_package_name(com.example.fftStockhamParaDisp)
3
4   #include "mathUtils.rsh"
5
6   unsigned int g_N;
7
8   unsigned int g_L0;
9   unsigned int g_L;
10  unsigned int g_r;
11
12  void fftStockhamParaDisp (rs_script stockhamKern, rs_allocation allocInCmplx, rs_allocation allocOutCmplx)
13  {
14      unsigned int nStages = uiLog2(g_N);
15      struct rs_script_call rs_call;
16      for (int stage = 0; stage < nStages; stage++)   // each power of two
17      {
18          g_L = 1<<(stage+1);
19          g_L0 = g_L>>1;
20          g_r=g_N/g_L;
21
```

```
22          rs_call.strategy = RS_FOR_EACH_STRATEGY_DONT_CARE;
23          rs_call.xStart = 0;
26          rs_call.xEnd = g_L0*g_r;
27          rs_call.yStart = 0;
28          rs_call.yEnd = 0;
29          rs_call.zStart = 0;
30          rs_call.zEnd = 0;
31          rs_call.arrayStart = 0;
32          rs_call.arrayEnd = 0;
33
34          if ((stage % 2) == 0)
35          {
36              rsForEach(stockhamKern, allocInCmplx, allocOutCmplx, NULL, 0, &rs_call);
37          }
38          else
39          {
40              rsForEach(stockhamKern, allocOutCmplx, allocInCmplx, NULL, 0, &rs_call);
41          }
42      }
43  }
44  void root (const float2 *in, float2 *out, const void *usrL, uint32_t i)
45  {
46      float2 b;
47      unsigned int j, omegaIdx;
48
49      j = floor((float)i / (float)g_r);
50
51      omegaIdx = ((float)j/(float)g_L)*g_N;
52
53      b = cmplxMult(omega[omegaIdx], in[j*g_r+g_r]);
54
55      out[0]        = in[j*g_r]+b;
56      out[g_L0*g_r] = in[j*g_r]-b;
57  }
```

References

[1] J. Owens, M. Houston, D. Luebke, S. Green, J. Stone, and J. Phillips, "Gpu computing," *Proceedings of the IEEE*, vol. 96, no. 5, pp. 879–899, May 2008.

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

[3] S. Chaudhuri. (2010) Cs148: Rendering - iii. [Online]. Available: "http://graphics.stanford.edu/courses/cs148-10-summer/docs/08_rendering3.pdf"

[4] Y. Ukidave, A. Ziabari, P. Mistry, G. Schirner, and D. Kaeli, "Quantifying the energy efficiency of fft on heterogeneous platforms," in *Performance Analysis of Systems and Software (ISPASS), 2013 IEEE International Symposium on*, April 2013, pp. 235–244.

[5] "Nvidias next generation cuda compute architecture: Kepler gk110," White Paper, Nvidia, 2012.

[6] *CUDA C Programming Guide*, PG-02829-001_v6.5, NVIDIA, 2014.

[7] N. C. for Computational Sciences. (2010) Jaguar. [Online]. Available: http://www.nccs.gov/computing-resources/jaguar/

[8] O. R. L. C. Facility. (2010) Ornl debuts titan supercomputer. [Online]. Available: https://www.olcf.ornl.gov/wp-content/themes/olcf/titan/Titan_Debuts.pdf

[9] CompuGreen. (2014) The green 500. [Online]. Available: http://www.green500.org/

[10] Qualcomm. (2011) Snapdragon s4 processors: System on chip solutions for a new mobile age. [Online]. Available: http://www.qualcomm.com/media/documents/files/snapdragon-s4-processors-system-on-chip-solutions-for-a-new-mobile-age.pdf

[11] (2014) Comparision of smartphones. [Online]. Available: http://en.wikipedia.org/wiki/Comparison_of_smartphones

[12] Qualcomm. (2009, Jan) Qualcomm acquires handheld graphics and multimedia assets from amd. [Online]. Available: https://www.qualcomm.com/news/releases/2009/01/20/qualcomm-acquires-handheld-graphics-and-multimedia-assets-amd

[13] A. L. Shimpi and B. Klug. (2012, Jul) Qualcomm's quadcore snapdragon s4 (apq8064/adreno 320) performance preview. [Online]. Available: http://www.anandtech.com/show/6112/qualcomms-quadcore-snapdragon-s4-apq8064adreno-320-performance-preview

[14] "Qualcomm snapdragon 800 processors," White Paper, Qualcomm, 2013.

[15] "The rise of mobile gaming on android: Qualcomm snapdragon technology leadership," White Paper, Qualcomm, 2014.

[16] "Qualcomm snapdragon 600 processors," White Paper, Qualcomm, 2013.

[17] *Cortex-A9 Technical Reference Manual*, DDI0388I, ARM, 2012.

[18] *Cortex-A9 NEON Media Processing Engine*, DDI0409I, ARM, 2012.

[19] *ARM Mali GPU OpenGL ES Application Optimization Guide*, DUI0555C, ARM, 2013.

[20] *ARM Cortex-A15 MPCore Processor Technical Reference Manual*, DDI0438I, ARM, 2013.

[21] "Khronos overview," Presentation, Khronos, 2012.

[22] *AMD Accelerated Parallel Processing OpenCL Programming Guide*, AMD, 2013.

[23] P. Du, R. Weber, P. Luszczek, S. Tomov, G. Peterson, and J. Dongarra, "From cuda to opencl: Towards a performance-portable solution for multi-platform gpu programming," *Parallel Comput.*, vol. 38, no. 8, pp. 391–407, Aug. 2012. [Online]. Available: http://dx.doi.org/10.1016/j.parco.2011.10.002

[24] Google. (2014) Android developers homepage. [Online]. Available: http://developer.android.com/

[25] R. J. Sams. (2011) Introducing renderscript. [Online]. Available: http://android-developers.blogspot.com/2011/02/introducing-renderscript.html

[26] R. Membarth, O. Reiche, F. Hannig, and J. Teich, "Code generation for embedded heterogeneous architectures on android," in *Design, Automation and Test in Europe Conference and Exhibition (DATE), 2014*, March 2014, pp. 1–6.

[27] X. Qian, G. Zhu, and X.-F. Li, "Comparison and analysis of the three programming models in google android," in *First Asia-Pacific Programming Languages and Compilers Workshop (APPLC)*, 2012.

[28] A. G. E. Chu, *Inside the FFT Black Box.* CRC Press, 2000.

[29] B. H. E. Kamen, *Fundamentals of signals and systems.* Prentice Hall, 1997.

[30] I. Selsnick. (2014) El 713: The fast fourier transform. [Online]. Available: http://eeweb.poly.edu/iselesni/EL713/zoom/fft

[31] C. Van Loan, *Computational Frameworks for the Fast Fourier Transform.* Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 1992.

[32] Google. (2014) "nexus homepage". [Online]. Available: http://www.google.com/nexus/

[33] R. J. Sams. (2013) Evolution of renderscript performance. [Online]. Available: http://android-developers.blogspot.com/2013/01/evolution-of-renderscript-performance.html

[34] C. Shen, S. Chakraborty, K. R. Raghavan, H. Choi, and M. B. Srivastava, "Exploiting processor heterogeneity for energy efficient context inference on mobile phones," in *Proceedings of the Workshop on Power-Aware Computing and Systems*, ser. HotPower '13. New York, NY, USA: ACM, 2013, pp. 9:1–9:5. [Online]. Available: http://doi.acm.org/10.1145/2525526.2525856