Performance Analysis of Parallel Languages on Android

CS 598SVA - Heterogeneous Computing

Abdul Dakkak Cuong Manh Pham Prakalp Srivastava

{dakkak, pham9, psrivas2}@illinois.edu

Abstract

Mobile devices have become ubiquitous with one of their main selling points shifting from performance to energy efficiency. Given that specialized hardware decrease energy utilizations for certain workloads, mobile devices have become heterogeneous to compete in the efficient energy market. But as the number of different heterogeneous devices increases, so does the programming complexity. Programming languages, such as OpenCL, which expose low level architectural details — either via the standard or implementation extensions — results in optimized code for certain architectures and not others. RenderScript achieves performance portability by both hiding architectural details, performing off-line optimizations to an IR, and delaying backend code generation until runtime. To aid the runtime, the IR is decorated with hints to allow the runtime to make intelligent decisions to determine where code is to be executed. We show, by (i) rewriting the Parboil benchmark suite to run on Android, and (ii) comparing the performance and energy utilization of Java, threaded Java, C, OpenMP, OpenCL, and RenderScript, that RenderScript strikes a balance by hiding enough of the architectural details while still achieving both good performance and energy utilization.

Our code is publicly available at `https://github.com/cmpham/RSBench`.

1. Introduction

Heterogeneous computing promises to address the rising power dissipation problem of today's traditional homogeneous multi-core systems. It provides the ability to integrate a variety of processing elements, such as large and small general purpose cores, GPUs, DSPs, and custom or semi-custom hardware into a single system. Applications that can efficiently use the full range of available hardware reap significant energy savings over conventional processors by executing portions of the computation on devices that perform the computational patterns efficiently. This promise of performance along with power efficiency has led mobile devices such as smartphones and tablets, which deal with a variety of applications with limited battery life, to move towards heterogeneous designs.

However, heterogeneity of hardware resources also has led to a diverse landscape of different programming models, run-time systems, profiling and debugging tools for application development. Since low-level performance and energy optimizations for one device or programming model often do not work with another programming model or device, one gets specialization of code which requires hand-tuning by experts for each programming model or device, e.g., an optimization for the GPU may degrade performance on a DSP and vice-versa. This is highly inefficient and unproductive: we cannot expect applications to use a separate language for each class of compute unit. If one is to expect applications to use the full range of available hardware (to maximize performance or energy efficiency or both) then programming environment has to provide common abstractions to facilitate this.

The industry and the research community have been actively working on abstractions to mitigate this problem. The recent development of RenderScript [11, 28] provides a framework for running computationally intensive tasks at a high performance by using a specialized runtime for parallelizing work across all processors available on the device, such as multi-core CPUs, GPUs, or DSPs. RenderScript is therefore removing the burden of load balancing and memory management from programmers to the run-time, unlike other solutions such as OpenCL, where programmers have more control over the execution semantics of the application (a programmer needs to determine where code is run and which part of the memory hierarchy data would reside). In this fashion, RenderScript is making the computationally intensive part of the application, that needs to be accelerated on specialized hardware, performance portable across the various hardware compute units. Also, since the application is not dependent on the existence and availability of a specific accelerator, the application is portable across SoCs with varying combinations of compute units.

While such portability is a noble goal, RenderScript achieves it at the cost of hiding hardware details from programmers that are critical to good performance on these accelerators. For example, in GPUs, the placement of data at various levels of memory hierarchy is critical for peek performance. It is this reason that most programming lan-

guages for GPUs allow programmers fine-grained control over memory management. RenderScript too can use GPUs for acceleration, but hides the memory hierarchy from programmers. In the RenderScript model, application developers only define the part of the application that needs to be accelerated, and the granularity at which data needs to be partitioned. The rest of the responsibilities of memory hierarchy management, work distribution among different compute units, and device selection is handled by the RenderScript compiler and run-time. This raises the question: "how effective is the RenderScript compiler and run-time, and is it able to achieve these goals?"

In this report we answer this question by performing a comprehensive performance analysis of RenderScript and compare it against other existing alternatives, namely serial Java, threaded Java, native C, OpenMP, and OpenCL. The report is divided as follows: section 2 introduces RenderScript, section 3 describes the benchmarks written for Android and the dataset sizes used for the analysis, section 4 gives some internals of how we structured our implementations of these benchmarks, section 5 shows results from our analysis which examines the performance along with processor and energy utilizations for the different implementations, section 6 describes related work, section 7 details the next logical steps that should be taken to enhance the benchmark suite and our insights of the results, finally, section 8 gives our conclusions.

2. Background

OpenCL and RenderScript have both been proposed for devloping compute kernels that are harware oblivious. They are designed to accelerate data parallel computation intensive parts of application code by allowing workloads to be executed on multiple processing cores, GPUs, DSPs or a combination of these. Here we provide a brief background of OpenCL and RenderScript.

2.1 OpenCL

OpenCL was initiated by Apple Inc. as a vendor neutral alternative to CUDA and is now managed by the Khronos Group [15]. An OpenCL application is composed of two parts: an OpenCL host program and a set of one or more kernels. The kernels, written in restricted C99 syntax, specify functions that are to be executed in data parallel fashion. The OpenCL host code is a C API that allows programmers to specify the device used for kernel execution, setting up the OpenCL environment, allocating memory, copying data, and executing the kernels.

To allow execution on different devices, OpenCL kernels are stored in the program as C strings. The OpenCL runtime embeds a compiler which is invoked and compiles the OpenCL code at run-time. This extra compilation step results in large overhead. Future OpenCL implementations have the ability to perform some off line compilation to a virtual instruction set [16] which should alleviate this problem.

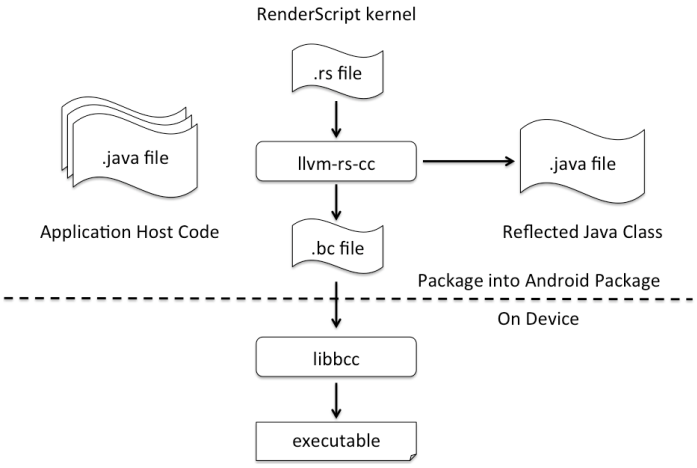


Figure 1: RenderScript Compilation Flow

In the ideal case, OpenCL allows application developers to write data parallel computation intensive parts of their application for an abstract hardware model, without using low level hardware specific function calls. In reality, developers need to target specific hardware to get tangential performance benefit. Furthermore, developers often use hardware specific parameters such as the size of OpenCL work-group, shared memory size either to obtain better performance on specific hardware or because the OpenCL API is verbose that it is inconvenient to extract that data pragmatically. This harms the portability of OpenCL application kernel code.

2.2 RenderScript

RenderScript is Android's official computing framework and was released in 2011 [11]. The motivation behind it is similar to that of OpenCL: to provide performance and portability across SoC architectures for Android devices. It accomplishes portability across devices by not exposing any architectural details to the programmer.

A RenderScript application consists of three parts: (1) Java application host code written by the developer that runs on Dalvik VM, (2) RenderScript code written in restricted C99 syntax containing one or more kernels, and (3) auto-generated Java code that helps the application host code to communicate with the RenderScript kernel code, allowing functions such as memory binding between the host program and the kernels.

RenderScript compilation flow is shown in Figure 1. First, `llvm-rs-cc` utility is used to compile RenderScript kernels to LLVM [18] bitcode files. The LLVM IR provide support for a wide range of hardware devices including CPUs, GPUs and DSPs. As part of the RenderScript compilation step, a Java class is generated for each RenderScript file via the `llvm-rs-cc` tool. The generated Java classes essentially avoids the user having to write JNI code. Then

Benchmark	Implementations					
	NC	OMP	J	JT	OCL	RS
VectorAdd	C	C	C	C	C	C
SGEMM	C	C	C	C	C	C
Stencil	C	C	C	C	C	C
CUTCP	N	N	C	C	C	C
MRIQ	N	N	C	C	C	C
TPACF	B	B	C	C	C	C
Histogram	C	B	C	C	C	C
BFS	N					
MRIG	N					
SAM	N					
SPMV	N					
LBM	N					

Table 1: Parboil Benchmark Porting Status. **NC**: Native C; **OMP**: Native C with OpenMP; **JT**: Threaded Java; **OCL**: OpenCL; **RS**: RenderScript; **C**: Completed; **N**: No Implementation; **B**: a bug causes the benchmark to crash.

after, the application host code, the reflected Java classes, and RenderScript bitcode are bundled together into the Android application package (`*.apk` file). During execution, the RenderScript runtime invokes `libbcc`, the RenderScript back-end compiler, to translate bitcode into machine assembly.

3. Benchmarks

We extend the Parboil benchmark suite to run on Android devices. Table 1 shows the porting status of each version of the benchmarks in the Parboil Benchmark Suite.

In this section, we give an overview of the benchmarks implemented along with the dataset sizes we used when profiling the results. While the Parboil benchmark suite represents scientific workloads, we expect them to be representative of future Android applications — given that we already see laptops using Android as their OS.

3.1 VectorAdd

The VectorAdd benchmark adds two floating point vectors with $8K$ elements. Compared to other benchmarks, VectorAdd has a very high memory to compute ratio. The benchmark is therefore not a fit for parallelization, but we use it to examine the overhead behavior.

3.2 SGEMM

The SGEMM benchmark multiplies two matrices A and B produces an output C. Matrix A is of dimension 128×96 and B is of dimension 96×160. Matrix A is stored in the row major format, while B is stored in the column major format — we therefore do not need to transpose B to make effective use of the cache.

The OpenMP code uses the `#pragma omp parallel for shared(A, B, C) collapse(2)` pragma, in which, based on the processor utilization, the Android compiler was not able to parse as valid OpenMP code. Therefore, the OpenMP code for SGEMM is equivalent to the serial C code.

Language	Line Count
Java	7549
RenderScript	1000
JNI/C++	2048
OpenCL	480

Table 2: Lines-of-code project breakdown per language.

3.3 TPACF

The TPACF benchmark analyzes the angular distribution of astronomical objects. The algorithm computes the distances between all pairs of coordinates in a dataset and then performs histogramming. The results are collected in three histograms which are then cross-correlated to find the statistical spatial distribution of the astronomical bodies. For our analysis, we use 100 datasets, each containing 487 coordinates.

3.4 MRIQ

The MRIQ benchmark computes a non-uniform 3D inverse-Fourier-transform, representing a calibration matrix. The calibration matrix is used to perform 3D image reconstruction from MRI data which is presented in a non-Cartesian space. The input dataset is of size $32 \times 32 \times 32$ containing trajectory information in 3D as direction parameter in 2D.

3.5 Stencil

The Stencil benchmark computes a 7 point stencil of an input volume. The input volume has dimension $128 \times 128 \times 32$. Each kernel performs a standard 7 point stencil: accessing the 6 adjacent voxels, scaling and then adding them to the current voxel. The result is then placed in the output buffer.

3.6 Histogram

The Histogram benchmark computes the histogram of an input image. The input image is of size 996×1040 and compute a histogram of size 256. Each bin in the histogram saturates at the value 255.

Unlike the other parallel implementations, which use atomics, both the threaded Java and OpenMP implementations use privatizations. Private histogram copies are allocated, each thread operates on its own private copy, and once the threads finish, the master thread aggregates the results.

4. Implementation

Table 2 shows the numbers of lines-of-code (LOC) broken down into different types of programming languages in the project. This section describes how we can achieve good code reuse while still having a sound and understandable program structure.

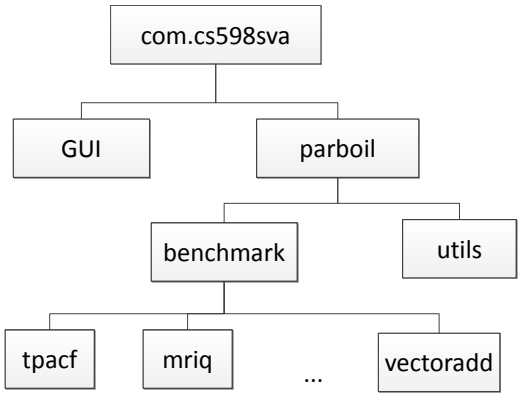


Figure 2: The Java package structure.

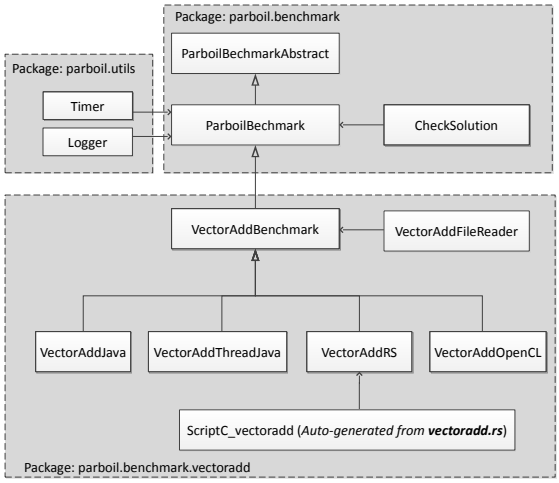


Figure 3: Class diagram of the VectorAdd benchmark.

4.1 Framework Structure

We design our framework to serve two purposes. First, as shown in Figure 2, we separated the UI logic from the benchmark logic, furthermore each benchmark is compartmentalized into its own Java package — this allows for benchmarks to be worked on independently. Second we made extensive use of subclassing to facilitate code reuse — we would like to only write one file input handler for each benchmark, for example. Figure 3 provides a closer view at the class hierarchy for the VectorAdd benchmark.

The `parboil` package is the core Java package of the project. Common utility functions such as timers and loggers are placed in the `utils` package and reused across benchmarks.

Each benchmark's base class has an input and output reader code to initialize the data, and code to check if the output result matches the expected output. The implementation of each benchmark requires one to define just the `compute` method. This design keeps code simple and avoids code duplication.

4.2 Utility Functions

A few utility functions were identified to be common across benchmarks. This section briefly discusses some of them.

4.2.1 Timer

The `Timer` class uses `SystemClock` to get real-time clock information with nanosecond resolution. `Timer` objects are collected into a dynamic list with each measuring a particular execution segment, e.g., allocation time, setup time, and compute time. `Timer.start(category, message)` starts the timer and sets its label (here category is a user defined category such as "Compute" or "Setup", while message is a message that further refines the category such as "Allocating temporary data structures"). At the end of the execution segment the `Time.stop()` method is called.

Unlike Parboil, which outputs the times to `stdout`, we output our data into a SQLite database. This affords us a few things. First, since data is outputted in the specified columns, we do not have to re-parse the output data. Second, timing information can be shared easily by copying the database. Finally, we can store more than just timing information – for example we also store which machine the time has been taken on as well as which runtime is being used. Since writing to flash is expensive, all timing data is stored in memory, and then after the benchmark has finished, it is written into the database.

4.2.2 Processor and Power Utilization

Unlike programming desktops, where one mainly improves software by increasing either features or performance, mobile programmers develop for increase features, performance, and battery life. In fact, one of the main selling points for heterogeneous programming on mobile devices, is the increase in battery life. Increasingly, hardware vendors, such as Apple or Samsung, sell new mobile hardware by advertising longer battery life. Finding a balance between energy consumption and performance is a balancing act that a programmer would like to delegate to the compiler or runtime.

Since Android does not offer a way to capture processor usage information programmatically, we use the Trepn [23] tool by Qualcomm to capture the data and save it into a `csv` file. This tool is limited to Qualcomm based chipsets. Trepn reads internal processor counters as well as power rail information, both of which are not available programmatically and are more accurate than reading the information from the `/proc` kernel file system.

Trepn is an external application that is run outside of our benchmark framework. To pass messages to Trepn, we make use of Android's `Intent` framework. The `Intent` framework allows one to pass messages between applications. Trepn records the time a message is received (which corresponds to time blocks in our code) and we developed scripts to

correlate Trepn's data dump with both the load and power usage.

We set Trepn to read the counters every $100ms$ and measure the load and power usage separately to decrease the overhead of the profiler. To reduce overhead, Trepn measures the processor usage information every $100ms$, both the frequency and the load are measured sequentially, we therefore needed to correct that when parsing the `csv` file.

First, we parse each processor reading along with the time-stamps for reading the file. Next, we interpolate the measured data (we use linear interpolation), and evaluate the interpolated value at the application state times (these are the times Trepn received signals from our application and correspond to timed blocks of code). We then multiply the load by the frequency, and rescale all the CPU and GPU data (we perform the rescaling on the CPU and GPU separately). Trepn can have measurement errors, resulting in infinite numbers. To make sure that these do not skew the plots, we clip the range of possible processor reading to be between 1 and 0.99th quantile of the data. While efforts have been taken to reduce the profiler's overhead, the overhead is still around 10%.

4.3 Compute Kernels

A RenderScript kernel is a map operation, taking a sequence of input and producing one output. For example, in the VectorAdd benchmark, the kernel takes two elements (one from each vector) adds the results and produces an output. With sequential accesses, one does not need to index into an array to access elements. In general, this results in code that's simpler to understand than OpenCL.

RenderScript does not unify memory, but provides APIs from within Java to allocate data and copy to and from the runtime. We use `copyFromUnchecked` to perform unsafe copies between Java and RenderScript, otherwise the runtime allocate temporary buffers and checks whether the conversion between the Java type and RenderScript type is valid.

The RenderScript targets the 4.4 Android platforms and is compiled with `renderscript.support.mode=false` which allows the compiler to use some optimizations which were not available in previous Android versions.

Since Android applications must be hosted within Dalvik VM, we write JNI to interface our code with the native — C, OpenMP, and OpenCL. To avoid having to write another timer, we split the native code into JNI functions corresponding to timed blocks and our Java timer is used to time the native code. Aside from wrapping the code via JNI so it is callable from within Java, little modification was done to the Parboil version of the C and OpenMP code.

The OpenCL kernels are lifted from Parboil as well, but we rewrite the OpenCL host code using the C++ OpenCL API, since it simplifies some of the code and affords more code reuse opportunities. We use the `base` implementation of OpenCL — this a platform agnostic unoptimized implementation.

Name	CPU	GPU	Memory
GalaxyNexus	ARMv7, 2 cores, 1200 Mhz, SIMD NEON	PowerVR-SGX 540	694Mb
Nexus 5	Qualcomm Snapdragon S4 Pro 1.5GHz	Adreno 320 400MHz	2Gb
Nexus 7	Qualcomm Snapdragon 800 2.26GHz	Adreno 330 450MHz	2Gb
Nexus 10	DualCore 1.7GHz Cortex-A15	Mali T604	2Gb
SM-T900	QuadCore 1.9GHz Cortex-A15	Mali T628	3Gb

Table 3: The device hardware specifications used for the analysis.

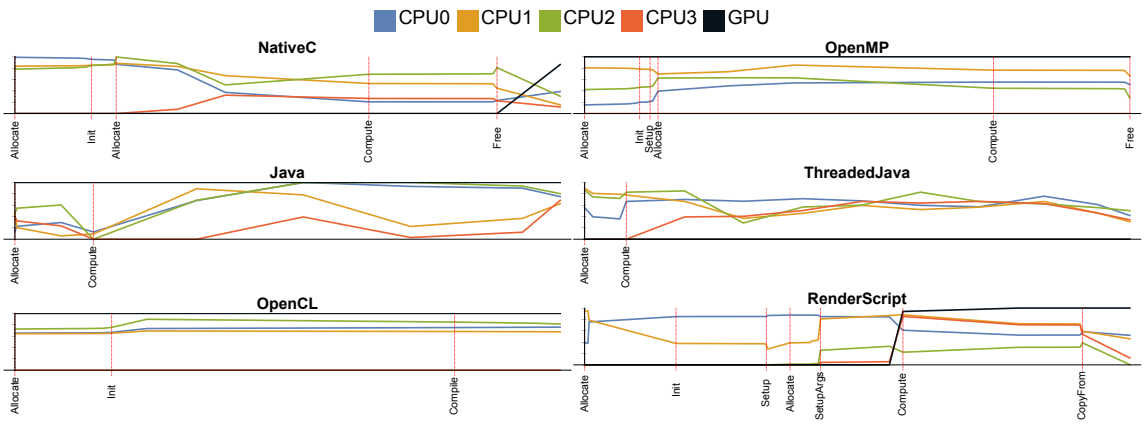
To allow devices with no OpenCL support to make use of the C and OpenMP implementations, we generate 3 libraries (one for C, OpenMP, and OpenCL). Each library is then loaded and called from with its class implementation. The `-O3 -ftree-vectorize -mvectorize-with-neon-quad` compile option is set when compiling the libraries, this allows the compiler the opportunity to autovectorization code.

5. Analysis

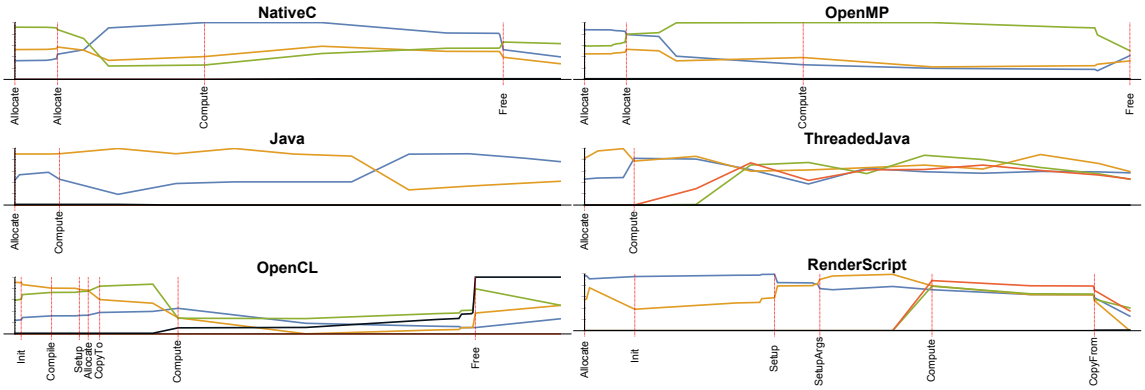
This section evaluates each benchmark on five devices shown in Table 3, all running Android $4.4.2$ (KitKat). These devices capture the low, mid, and high end mobile and tablet devices that are currently available in the market. Only the Nexus 5 and Nexus 7 devices can be modified to install an unofficial OpenCL implementation and only those two devices are fully supported by Trepn. Each benchmark contains multiple implementations, which are analyzed to determine trade-offs between processor utilization, performance, and energy usage. We conclude the section by discussing the programability of RenderScript and compare it against established programming models such as: native C, OpenMP, and OpenCL.

5.1 Processor Utilization

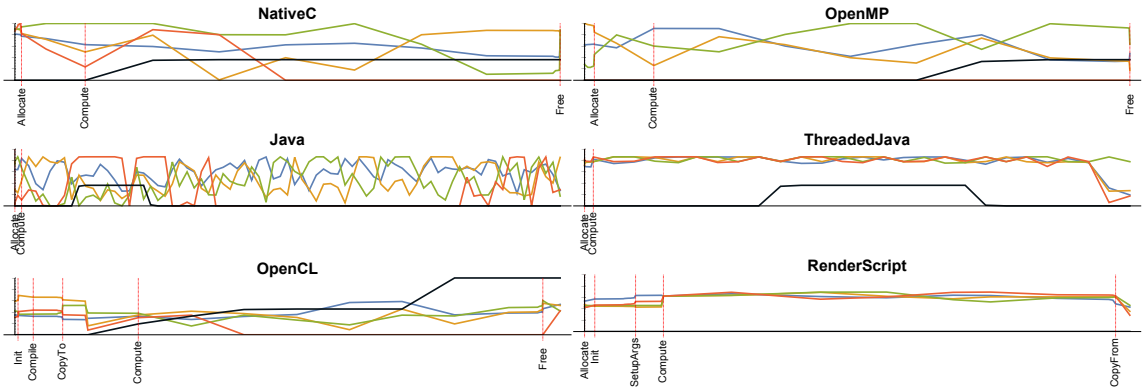
Mobile devices use dynamic voltage frequency scaling (DVFS) to match performance to power utilization. The frequency is increased for the processors when the load goes over a certain threshold. Both CPUs and GPUs make use of frequency scaling. But unlike CPUs (which typically have very fine-grained frequencies — operating at around 10 different frequencies), GPUs have coarse-grained frequencies (operating at only 4 different frequencies).



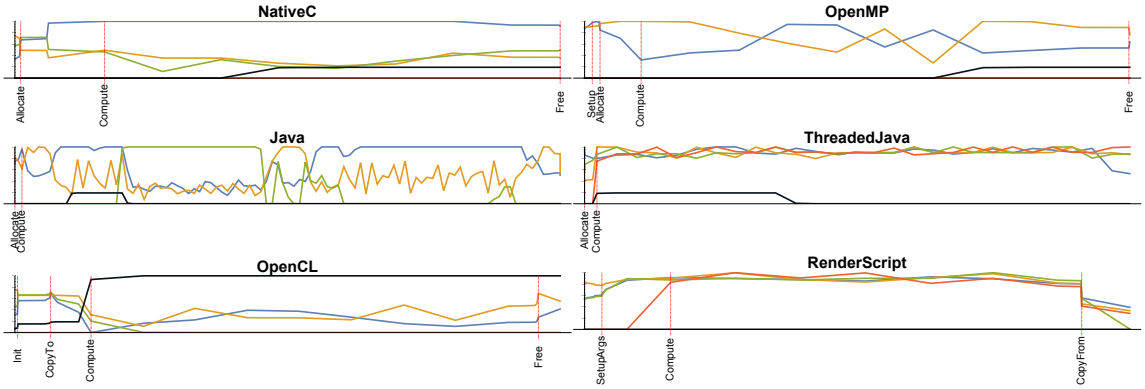
(a) VectorAdd on Nexus 5



(b) VectorAdd on Nexus 7



(c) Sgemm on Nexus 5



(d) Sgemm on Nexus 7

Figure 4: Processor utilization of VectorAdd and SGEMM for both Nexus 5 and Nexus 7. The x axis is time, and the y axis is normalized to the peek utilization for CPUs or GPUs across implementations.

Processor utilization is measured by multiplying the processor load and frequency information collected via Trepn. The data is measured while the system is connected to a power source, which avoids the device going to sleep (it does add the extra overhead of the device communicating debug information with the development machine). Since Trepn has a $100ms$ measure granularity, the kernel code is run 100 times (5 times for MRIQ and TPACF), this allows us to visualize the trend in resource utilization.

As per standard Android applications, the UI runs in a separate UI thread independent from the computation. Trepn, which has a high overhead (10% on average), also runs as a background process. These background processes are echoed in the plots — a single threaded Java code, for example, should not utilize more than one core, but our plots show that more than one core is active.

The VectorAdd benchmark has a very high memory to compute ratio, therefore the processor utilization plot (figure 4) shows that not all CPUs were fully utilized during the compute phase. For the Nexus 5 device, while the graphs show a GPU was being used during the OpenMP, RenderScript, and Java, this is an error in the measurement — we believe this is due to the UI thread utilizing the GPU or some other interferences. As expected, the GPU is not being utilized on the Nexus 7 except for OpenCL.

As discussed in the benchmark section, the Android GCC compiler was not able to interpret the OpenMP `pragma` code and therefore SGEMM ran in a single thread on both the Nexus 5 and Nexus 7 (figure 4). While OpenCL does utilize the GPU, because of the size of the matrix, the GPU has lower occupancy and therefore does not achieve peek performance. Both the RenderScript and ThreadedJava code make full utilization of all the cores.

MRIQ is embarrassingly parallel, and we can see full utilization for both the CPUs and GPUs in figure 5. Abrupt dips in the plots show positions where a kernel launch occurs. It is also interesting to note that the CPU on the Nexus 5 was not fully utilized for both ThreadedJava and RenderScript. This is due to either the load being low for a high frequency or the frequency being chosen to be low.

The TPACF code is divided into two parts. The first is serial allocation and populating data that, for the purpose of code reuse, is done on the Java side using a single thread. Once that completes, we then execute the code on different datasets in parallel. The utilization plots (figure 5) show this behavior.

For Histogram (figure 6), we see that memory starts to play a major role in processor utilization. The GPU was being utilized in the memory copy as it was in the computation, for example.

For Stencil (figure 6), we see that RenderScript did not fully utilize all available CPU cores. This is because the stencil kernel is memory access bound, performing around 10 flops (ignoring index calculations). We again see that memory copy (`SetupArgs` for RenderScript and `CopyTo` for OpenCL) resulted in a substantial amount of resources being wasted.

Throughout the benchmarks, we see that RenderScript did not utilize the GPUs on the tested devices. This is due to RenderScript requiring full IEEE 754-2008 compliance, which the GPUs are not able to provide. Inserting the `#pragma rs_fp_imprecise` pragma into the RenderScript kernel allows for relaxed IEEE compliance and should allow for GPU acceleration. Similar options are available in OpenCL and C, but since this option is not available in Java, we chose to disable it. Further investigations would require us to analyze the results with relaxed precision.

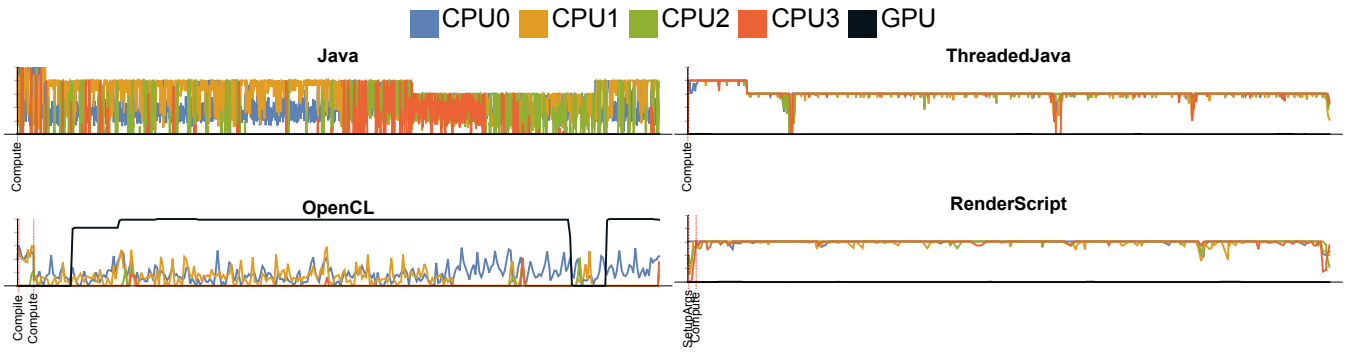
5.2 Performance

Performance is measured by timing code within regions of code while the device is plugged into the development machine. Each compute part of an implementation is run 5 times with the minimum presented. We consider two cases — one where the kernel code is run once (figure 7) and therefore the overhead (memory, compilation, and initialization) has a big impact on perfomance, and one where the kernel is run 100 times (figure 8) (or 5 for both TPACF and MRIQ) and the overhead has little impact.

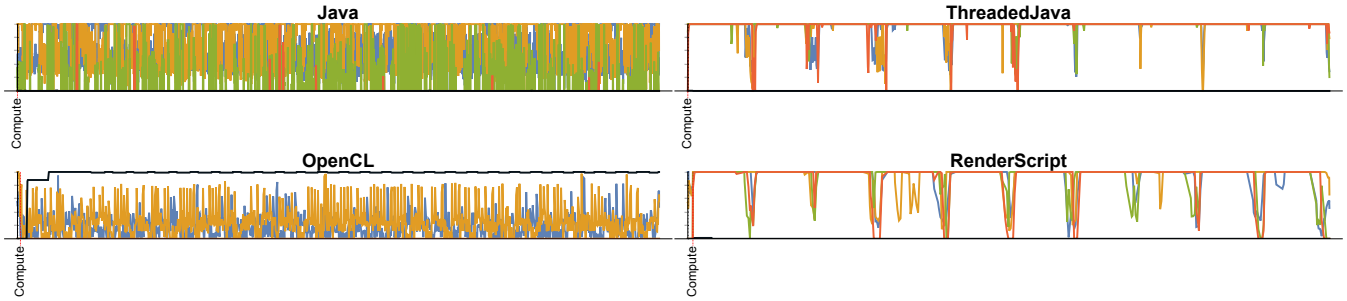
For each device, the plot show the time taken to execute sections of the code normalized to the Java execution time. These times correspond to the x-axis of the processor utilization times discussed in the previous section (e.g. figure 4) — Trepn is not running while collecting these timing results. Not all benchmarks were run on the GalaxyNexus, since the device, being low end, takes a long time to execute the benchmarks.

In figure 7 the compute code is only executed once, it is clear that the overhead of RenderScript on the Nexus 10 (and to some extent the SM-T900) device is consistently high. We suspect that the Nexus 5 and Nexus 7 are using a more recent version of the RenderScript library compared to the Nexus 10. As one would expect, when a kernel is executed only once, then overhead dominates the time. Thefore, for such executions, it is not a good policy to off-load the computation to either RenderScript or OpenCL — i.e. the programmer still needs to understand which sections of the program are very hot and could benefit by not being hosted in Java.

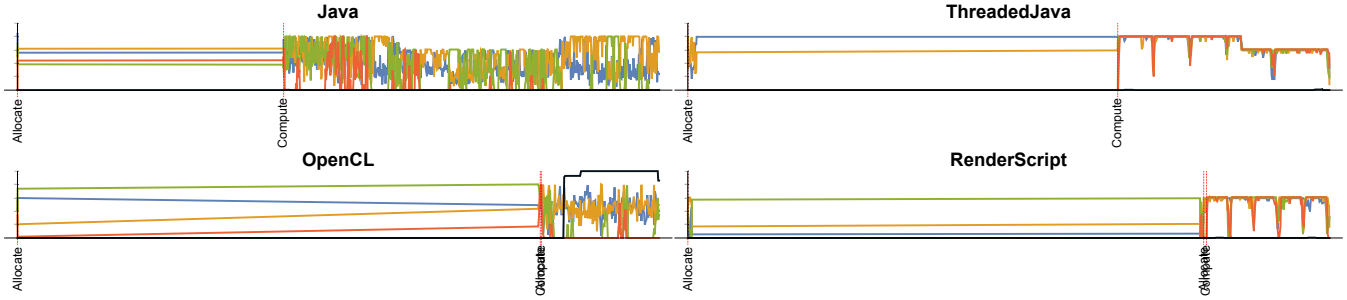
In figure 8, we look at the performance if memory management is optimized and the kernel code is executed many times. Code with a high memory to compute ratio, such as VectorAdd (and SGEMM to some extent), do not perform well using either RenderScript or OpenCL (this is due to poor occupancy in the OpenCL case). For code that has irregular accesses or with a low memory to compute ratio, we find that RenderScript's compute time is similar to that of OpenCL, but is better when also considering overhead time. Both RenderScript and OpenCL outperform the OpenMP implementation in all benchmarks.



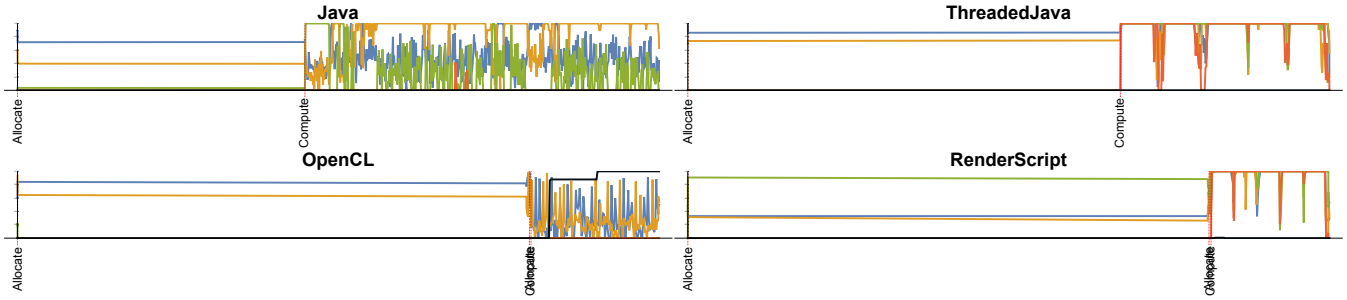
(a) MRIQ on Nexus 5



(b) MRIQ on Nexus 7

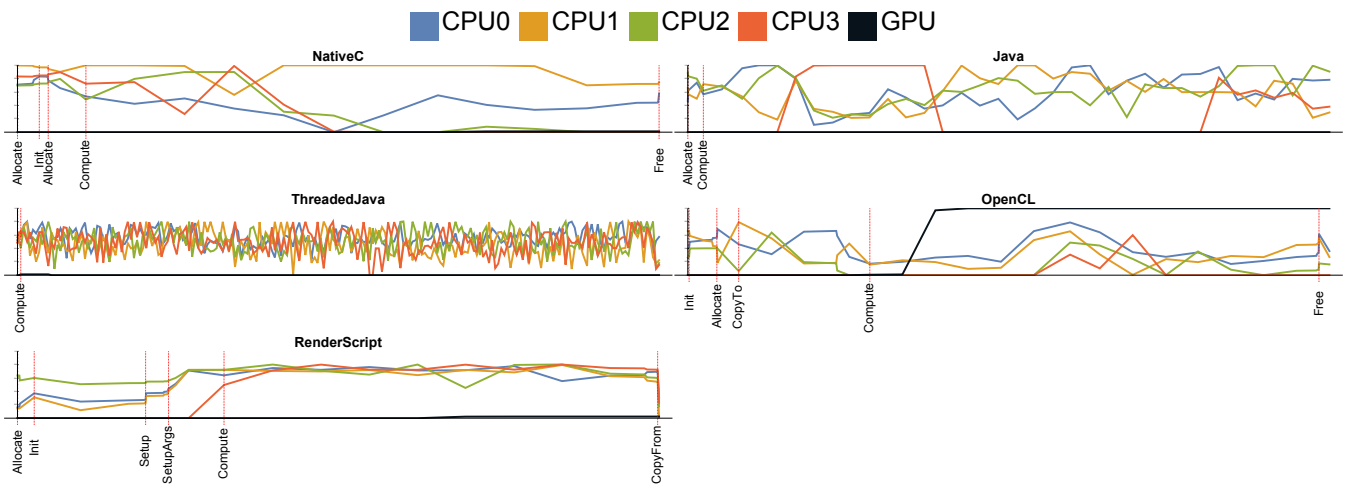


(c) TPACF on Nexus 5

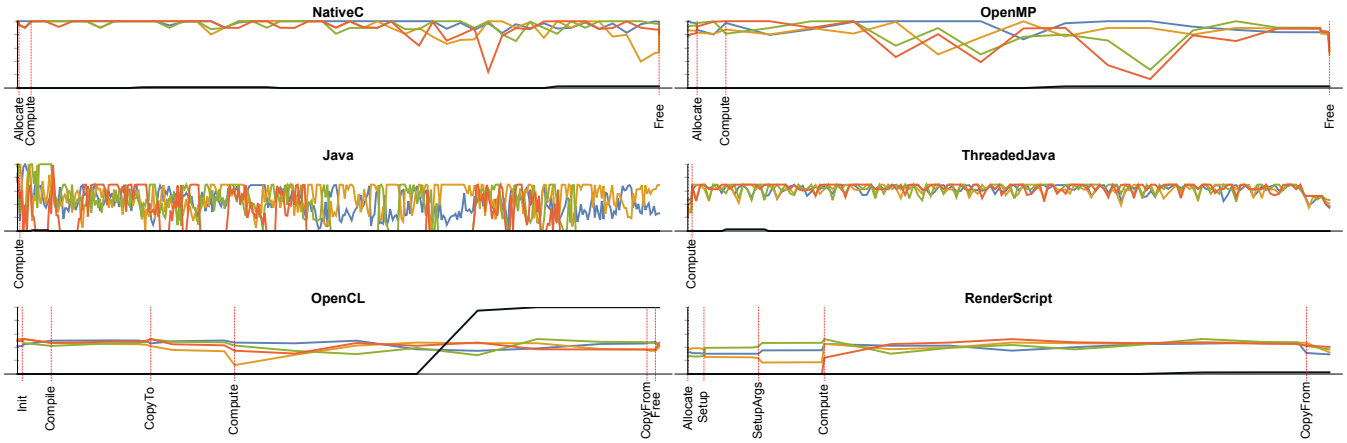


(d) TPACF on Nexus 7

Figure 5: Processor utilization of MRIQ and TPACF for both Nexus 5 and Nexus 7. The x axis is time, and the y axis is normalized to the peek utilization for CPUs or GPUs across implementations.



(a) Histogram on Nexus 5



(b) Stencil on Nexus 5

Figure 6: Processor utilization of Histogram and Stencil for both Nexus 5. The x axis is time, and the y axis is normalized to the peek utilization for CPUs or GPUs across implementations.

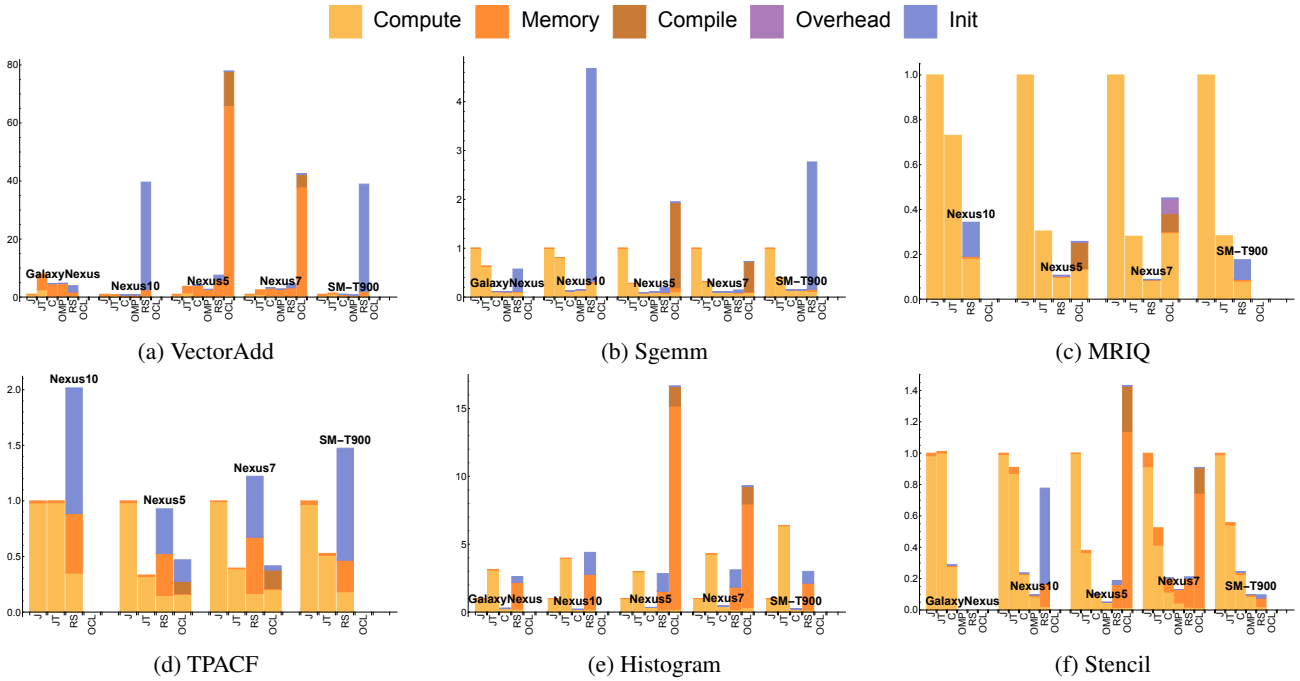
As expected, SGEMM's OpenMP timing is similar to that of sequential C, confirming our hypothesis that the C compiler was not able to interpret the OpenMP pragma. Because of the privatization, which requires allocation of histogram within each thread, the threaded Java implementation performs poorly and is worse than the serial Java implementation. Consistently, OpenCL results in better speedups on the Nexus 5 versus the Nexus 7 when compared to the on-board CPU.

The biggest performance gain comes by not using the JVM, however. Aside from typical JVM overhead, we notice that these kernels are array access extensive. Since Java's semantics guarantee array accesses are within bounds, an overhead is incurred for the JVM having to perform implicit checks when arrays are accessed. Java's floating point semantics also do not match modern hardware (which implement the IEEE 754 standard), this introduces more overhead where the JVM needs to perform extra checks and convert to Java's IEEE representation. These overheads do not manifest themselves in our native implementations, and also use unsafe casts in both our RenderScript and native code to avoid Java performing checks to convert datatypes between Java and RenderScript or Java and native code.

Because of the tooling, however, in many cases writing RenderScript is easier that writing C with JNI bindings. One does not need to interact with the Java garbage collector, as you do need to when accessing raw Java memory in JNI.

5.3 Energy Utilization

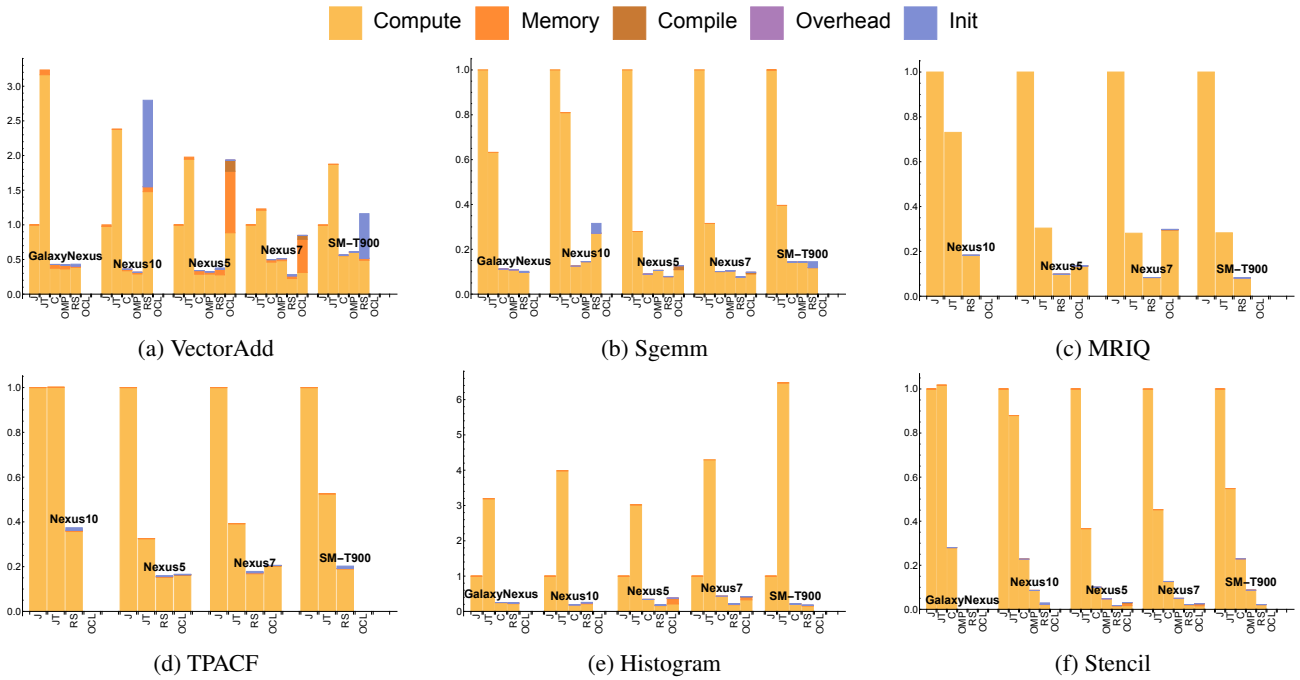
Since mobile devices employ DVFS, the energy utilization of the devices at a specific time is governed by the operating frequency of the processor. One can model [9, 30] the battery usage at time t by using the following formula:



(a) VectorAdd (b) Sgemm (c) MRIQ

(d) TPACF (e) Histogram (f) Stencil

Figure 7: Runtime across devices where kernel is executed once. The runtime is normalized to the Java execution time (lower is better). J : Java, JT : JavaThreaded, C : Native C, OMP: OpenMP, OCL : OpenCL, and RS : RenderScript.



(a) VectorAdd (b) Sgemm (c) MRIQ

(d) TPACF (e) Histogram (f) Stencil

Figure 8: Runtime across devices where kernel is executed multiple times. The runtime is normalized to the Java execution time (lower is better). J : Java, JT : JavaThreaded, C : Native C, OMP: OpenMP, OCL : OpenCL, and RS : RenderScript.

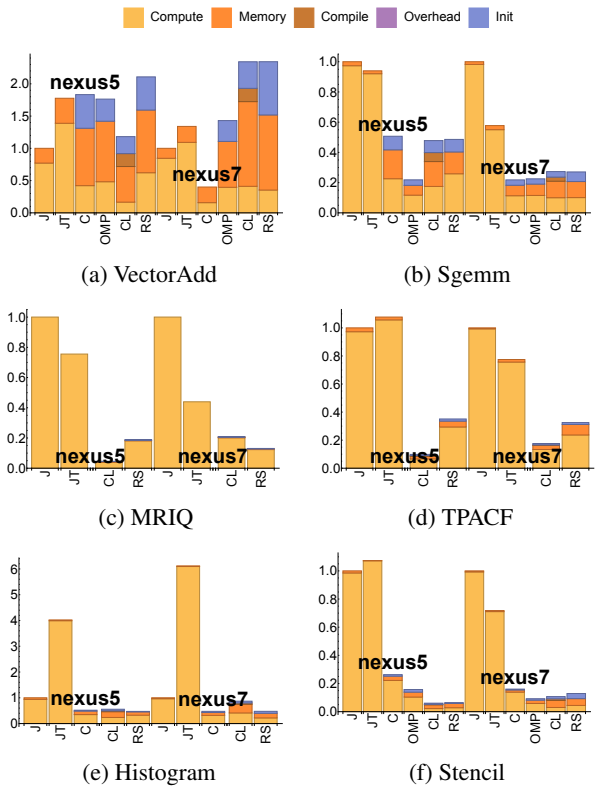
$$P(t) = \beta_v(\beta(t)) + G_u(t)\, G_v(G_f(t)) + \sum_{i=1}^{N} C_{u_n}(t)\, C_{v_n}(C_{f_n}(t))$$

where N is the number of CPU cores, β is the brightness of the LCD screen at time t, $\beta_v(br)$ is the power used for the specified brightness level, $G_f(t)$ and $C_{f_n}(t)$ are the operating frequencies at time t, $G_v(f)$ and $C_{v_n}(f)$ are the power draws for the processors at the specified voltage, and $G_u(t)$ and $C_{u_n}(t)$ are the processor utilizations at time t. Other terms, such as GPS, wireless, and other sensors, can be measured or modeled, but for this analysis we turned them off.

The issues with using a model is determining the power draw at the specified frequency (which is not specified by the processor's manufacturer) and the method of reading the CPU and GPU counters varies from device to device. As a result, we again use Trepn to read the hardware counters giving us the battery usage (in uW) at each time interval. Battery usage consists of all power-consuming components, e.g., screen, CPU, GPU, and memory. If, for example, the computation takes longer then more power is consumed by other components, such as the screen. Therefore, the energy consumed by a computation is not only dependent on of the load, but also the time it takes to perform a computation. We place the device in airplane mode (to disable GPS, wireless, and other sensors) and disconnect the device from a power source to collect the data. Similar to the when measuring the processor utilization, we only consider the case where the kernel is executed many times.

Figure 9 shows the energy consumed by each implementation for both Nexus 5 and Nexus 7 (Trepn is not able to read hardware counters for the other devices). While, as discussed in the previous section, RenderScript performs better, it does make full utilization of all CPU cores. OpenCL, on the other hand, does not make use of all cores and only fully utilizes GPUs. Since OpenCL has a similar time profile as RenderScript on the Nexus 5, we observe that it is more energy efficient compared to RenderScript. For the Nexus 7, in some benchmarks RenderScript has better energy utilization while OpenCL is better in others.

Therefore, which implementation to choose to reduce energy consumption is device- and compute-pattern-dependent. Compared to the other implementations, RenderScript has lower energy consumption for many of the benchmarks. It is also clear that the LCD screen makes a big impact on power usage, in MRIQ, for example, even though the threaded Java code utilizes all the CPU cores, because it is able to complete quicker it uses less energy overall compared to the serial Java code.



(a) VectorAdd (b) Sgemm (c) MRIQ (d) TPACF (e) Histogram (f) Stencil

Figure 9: Energy usage for both the Nexus 5 and Nexus 7 normalized to Java's energy utilization (lower is better).

5.4 Input/Output

Throughout the analysis, we have ignored IO times which is in fact the biggest performance bottleneck in these benchmarks. The main bottle neck is read speeds, since unlike desktops which have a hard drive read speed of 80-100MB/s, mobile devices typically use use flash which have a 7-25MB/s read speed. In reality, however, computational data for mobile devices is not stored on disk and is usually either streamed from the cloud or captured from onboard sensors and therefore is available in RAM. Therefore, to keep with the typical usage, and to not skew the plots, we explicitly removed the IO performance times from our graphs.

5.5 RenderScript Programmability

This section presents our analysis of RenderScript's programmability. This analysis is based on our experience working with RenderScript, and is is highly subjective. Some of the limitations presented are due to RenderScript being a less mature framework (compared to CUDA and OpenCL) and is actively being developed. Therefore, we expect that RenderScript will evolve to fix many of the identified limitations and fix the unnecessary idiosyncratic restrictions we encountered.

5.5.1 Performance Portability

The main objective of RenderScript is to give acceptable compute performance (comparable to OpenCL) on a variety of hardware. The runtime behavior of our RenderScript implementations is consistent across the devices we tested on. Reproducibility of bugs is another important factor, we always got the same runtime errors, e.g., segmentation fault or out-of-memory error, regardless of the devices that the code is running on.

RenderScript does not expose any device specific features — it is not even possible to choose which device RenderScript runs on. Regardless, RenderScript was able to achieve good performance and work unchanged across devices. OpenCL, on the other hand, required us to read some device specific parameters and change some of the run parameters, such as work-group sizes, in order to avoid runtime errors.

5.5.2 Debug Support

Since RenderScript is natively supported in Android, it offers a set of convenient built-in debugging facilities, such as the `rsDebug` functions, detailed runtime exceptions, and IDE-navigable compilation errors. The `rsDebug` function forces RenderScript to execute on the CPU and interacts with the Android's logging facilities, e.g., logcat, to offer a convenient way to present printed values from within the kernel (OpenCL and CUDA offer similar facilities, but they are less convenient to use). For programmers accustomed to `printf` debugging style in C/C++, this interface provides a familiar debugging model. For developers expecting a breakpoint debugging workflow, the Eclipse environment has no facilities for that currently.

5.5.3 Memory Operations

The `Allocation` interface is intuitive to express data and execution parallelism. The `Allocation` interface gives programmers the ability to express parallelism granularities, i.e., via packing output and/or input into `Element` objects. The interface is intuitive and is similar to the already familiar OpenCL memory interface. Within the kernel, utility functions (`rsGetElementAt_*`) make indexing into a multi-dimensional arrays platform agnostic (e.g. the strides should not be assumed to be the same across architectures). The helper functions also allow one to not perform complex index arithmetic, mapping a set of coordinates to access a 1D array.

5.5.4 Familiar Language

RenderScript kernel is C99-based, and therefore does not require any learning of syntax. Classifying that as a feature is slightly biased towards programmers already familiar with C programming. For many programmers, a transition from programming in Java to programming in C might not be simple — especially if the two languages are within the same project. Similarly, for C programmers, having a language that is C like but employing different semantics might be confusing.

5.5.5 Multi-Dimensional Parallelism

Unlike the familiar CUDA and OpenCL models where threads are partitioned into blocks which are then partitioned into a grid, RenderScript has only one level partitioning. A RenderScript kernel allows an `Allocation` to specify `X`, `Y`, and `Z` dimensions, and the workload would be distributed across hardware units using these dimensions. We found that the current implementation lacks support for launching 3D kernels.

5.5.6 Lack of Synchronization Intrinsics

Because RenderScript does not allow one to group threads into blocks, there is only a global synchronization operation, i.e.,using `syncAll`, which is called from within the Java code. Coming from a GPU programming background, this means no support for `__syncthread` as in CUDA or the OpenCL equivalent `barrier` function. The RenderScript kernels therefore cannot perform fine grained synchronizations and share data between a group of threads. For some benchmarks, this model is not convenient. In the `CUTCP` benchmark, for example, we had to rethink how the algorithm operates to port the OpenCL implementation to RenderScript.

5.5.7 Non-Unified Memory

A RenderScript kernel requiers explicit allocation and copy for all data that it accesses. The buffers then have to be copied back to make them available to Java. Safe casting between Java and C requires some checking — since Java does not conform to the same IEEE standard that RenderScript conforms to. Use use unsafe casting which results in little overhead when the processors are in the same coherence domain. The current implementation of RenderScript does not perform these optimizations. Furthermore, the runtime, via function overloading, should be able to hide explicit buffer creation and copies, by detecting the type passed into the function and converting it into an `Allocation` buffer if it is passed in a Java array.

5.5.8 Lack of a Standardization

RenderScript is similar to CUDA in this respect — the reference implementation and documentation being the standard. Yet unlike CUDA, RenderScript's documentation tend to be incomplete and insufficient. Atomic instructions such as `rsAtomicInc`, for example, are claimed to be supported (in both the documentation and the header files) for several data types, but a runtime exception is raised when using the function with types other than `int32_t`. The lack of specification results in some errors being unintuitive — one has to read the runtime code to determine what some of the errors mean as well as possible causes. The lack of standard body also

means that it is unlikely that RenderScript would be ported to desktop platforms and adopted on descrete GPUs.

5.5.9 Yet Another Parallel Framework

Aside from having full control over the language [3], it is not clear why Google did not adopt OpenCL or OpenCL's SPIR layer. As can be seen, the RenderScript language borrows many elements from the CUDA/OpenCL programming model, with some tooling support. Many of the "features" of RenderScript can be implemented via a library that interacts with OpenCL and there is no need for a new language. Furthermore, many hardware vendors already support OpenCL [20].

One reason for a new language could be that the compiler and runtime can prevent compute code that run on the GPU from possibly crashing the driver, e.g., by using too much resources or using unsafe memory accesses, since crashing the GPU driver on mobile devices is equivalent to crashing the device. Yet, again, this feature can be implemented by enhancing the OpenCL compiler and runtime to detect and raise an exception if illegal memory accesses occur. It is therefore unclear why a new programming language was necessary.

6. Related Work

Related work can be divided into 3 categories : works that explore performance portability, works that examine the programming models and how to abstract some of the low level details them, and, finally, works that examine computational patterns on mobile platforms.

6.1 Performance Portability

In term of programming model, RenderScript is similar to OpenCL [15] and CUDA [1]. While CUDA is restricted to NVIDIA GPU devices, OpenCL is vendor neutral. OpenCL performance and performance portability has been extensively evaluated. Most evaluations compare OpenCL against CUDA on GPUs [4, 10, 12, 17, 25–27]. Since OpenCL and CUDA have a similar platform, memory, and programming model on discrete GPUs, a one-to-one analysis is fair. Using benchmarks, such as Parboil, [4, 25–27] show that on NVIDIA GPUs, CUDA achieves better performance than OpenCL. The studies also show that OpenCL provides a sufficient interface for developers to express more architectural details to improve the performance of their applications.

In [8, 10, 17] the authors show that most OpenCL kernels can obtain comparable performance with CUDA kernels when properly optimized. Furthermore, the authors show that OpenCL achieves stable performance across the tested platforms. The studies illustrate, however, that cases exist where OpenCL does not handle architectural specifics well, such as memory layout and number of processing cores. In order to improve the portability of applications, recent OpenCL iterations have added the option to delegate the workgroup size to the runtime. We are not aware of any study that evaluates the optimality of this feature.

6.2 Programming Model Abstractions

There are many recent proposals aiming at abstracting parallel software development to either make it easier to develop, less error prone, or to have better utilization heterogeneous cores on mobile devices. Aparapi [5] allows one to off load computation to OpenCL by inserting Java source code annotations around the compute kernels. In [7] the authors extend the Aparapi model, introducing Android-Aparapi, and targets Android devices.

Intel [6] has extended the Thread Building Blocks (TBB) [24] model to run on heterogeneous devices. In this model, one encodes parallelism as a DAG of tasks. The tasks are then schedule to run in parallel on available hardware (maintaining the dependencies that the programmer encoded). Qualcomm MARE [22] is similar to Intel's TBB extension. In [14], the authors summarize the programming model of common task based parallelism implementations.

Other work examined whether a source-to-source translator is practical to port OpenCL code to RenderScript. In [29], the authors presents such a tool along with challenges of this process: the most notable is differences in the execution models of the two. More recently, in [21] the authors develop a domain specific language that allows one to develop image processing kernels, but target C, CUDA, OpenCL, or RenderScript. In [2], the authors propose the Paralldroid language which allows one to target C and RenderScript using programmer's annotation of Java source code. The results show that the auto-generated RenderScript code often achieves higher performance than the auto-generated C code.

6.3 Computation on Mobile Devices

Studies examining the computation behavior on mobile devices are scarce. In [13] the authors compare RenderScript, Remote CUDA (RCUDA), and C for image processing applications. The results show that on an NVIDIA Tegra 3 GPU RenderScript outperforms both the RCUDA and C. Furthermore, this performance gap increases as the size of the datasize increases.

However, we are not aware of any work that provides a systematic evaluation of RenderScript's performance and performance portability. The only available tool that does some comparison is the CompuBench mobile for RenderScript [19] commercial benchmark application. This application is targeted at comparing RenderScript performance across devices, rather than understanding the results, and the benchmarks used (around 4 applications) are all image processing based.

7. Future Work

Redoing the analysis with relaxed floating points enabled for RenderScript kernels and determining what impact that

has on both the performance and energy utlization would be the first step in understanding the results better. Fixing bugs that prevented us from running some benchmarks, and completing the other benchmark implementations would allow us to have a benchmark suite that has been well studied on desktop and would allow us to have more insights into which computational patterns have a clear mapping to RenderScript. Finally, all the devices used are ARM based. Measuring the performance on Intel or NVIDIA based CPUs or GPUs would give us more insight about when to use CPUs or GPUs for computation.

8. Conclusion

Unlike CUDA or OpenCL which are specifically marketed to get maximal performance, RenderScript markets itself as a language that would give good performance across devices. For performance portability, while maintaining the semantics and reducing energy utilization, we believe that RenderScript delivers on its promise.

Furthermore, unlike the C or OpenMP which rely on static analysis and compiler intelligence to get free speedups, and OpenCL which is not officially supported, we see future versions of RenderScript offering free speedups for programmers. And while we would like RenderScript to expose more information about the running hardware, this is because we are more familiar with CUDA and OpenCL programming where such facilities are available. We think that we are in the minority, and most programmers do not need full control over the hardware. Furthermore, exposing some hardware detail would complicate the language and would cause more overhead by the runtime. This entices us to suggest RenderScript as the language, over C or OpenMP, when writing compute intensive codes on Android devices.

References

[1] *NVIDIA CUDA Compute Unified Device Architecture - Programming Guide*, 2014. URL `http://developer.download.nvidia.com/compute/cuda/6_0/NVIDIA_CUDA_Programming_Guide_6.0.pdf`.

[2] A. Alejandro and F. Almeida. Performance analysis of parall-droid programs. *Annals of Multicore and GPU Programming*, 1(1), 2014.

[3] R. Amadeo. Googles iron grip on android: Controlling open source by any means necessary, May 2014. URL `http://tinyurl.com/k6gfeaj`.

[4] R. Amorim, G. Haase, M. Liebmann, and R. W. dos Santos. Comparing cuda and opengl implementations for a jacobi iteration. In *High Performance Computing & Simulation, 2009. HPCS'09. International Conference on*, pages 22–32. IEEE, 2009.

[5] Aparapi. Api for parallel java, May 2014. URL `https://code.google.com/p/aparapi/`.

[6] R. Barik, R. Kaleem, D. Majeti, B. T. Lewis, T. Shpeisman, C. Hu, Y. Ni, and A.-R. Adl-Tabatabai. Efficient mapping of irregular c++ applications to integrated gpus. 2014.

[7] H.-S. Chen, J.-Y. Chiou, C.-Y. Yang, Y. jui Wu, W. chung Hwang, H.-C. Hung, and S.-W. Liao. Design and implementation of high-level compute on android systems. In *Embedded Systems for Real-time Multimedia (ESTIMedia), 2013 IEEE 11th Symposium on*, pages 96–104, Oct 2013. .

[8] R. Dolbeau, F. Bodin, and G. C. de Verdiere. One opencl to rule them all? In *Multi-/Many-core Computing Systems (MuCoCoS), 2013 IEEE 6th International Workshop on*, pages 1–6. IEEE, 2013.

[9] M. Dong and L. Zhong. Self-constructive high-rate system energy modeling for battery-powered mobile systems. In *Proceedings of the 9th international conference on Mobile systems, applications, and services*, pages 335–348. ACM, 2011.

[10] J. Fang, A. L. Varbanescu, and H. Sips. A comprehensive performance comparison of cuda and opencl. In *Parallel Processing (ICPP), 2011 International Conference on*, pages 216–225. IEEE, 2011.

[11] Google. Renderscript, May 2014. URL `http://developer.android.com/guide/topics/renderscript`.

[12] K. Karimi, N. G. Dickson, and F. Hamze. A performance comparison of cuda and opencl. *arXiv preprint arXiv:1005.2581*, 2010.

[13] R. Kemp, N. Palmer, T. Kielmann, H. Bal, B. Aarts, and A. Ghuloum. Using renderscript and rcuda for compute intensive tasks on mobile devices: a case study. 2013.

[14] D. Khaldi, P. Jouvelot, C. Ancourt, and F. Irigoin. Task parallelism and data distribution: An overview of explicit parallel programming languages. In *Languages and Compilers for Parallel Computing*, pages 174–189. Springer, 2013.

[15] Khronos. Khronos opencl 2.0 specification, May 2014. URL `https://www.khronos.org/opencl/`.

[16] Khronos. The standard portable intermediate representation for device programs, May 2014. URL `http://www.khronos.org/spir`.

[17] K. Komatsu, K. Sato, Y. Arai, K. Koyama, H. Takizawa, and H. Kobayashi. Evaluating performance and portability of opencl programs. In *The fifth international workshop on automatic performance tuning*, 2010.

[18] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.

[19] K. Ltd. Compubench mobile for renderscript, May 2014. URL `https://compubench.com/`.

[20] Mali. Mali opencl sdk, 2004. URL `http://malideveloper.arm.com/develop-for-mali/sdk/mali-opencl-sdk/`. [Online; accessed 25-Mayruary-2014].

[21] 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*, pages 1–6, March 2014. .

[22] Q. D. Network. Parallel computing (mare), May 2014. URL `https://developer.qualcomm.`

com/mobile-development/maximize-hardware/parallel-computing-mare.

[23] T. Profiler. Qualcomm.

[24] J. Reinders. *Intel threading building blocks: outfitting C++ for multi-core processor parallelism*. O'Reilly Media, Inc., 2007.

[25] R. V. van Nieuwpoort and J. W. Romein. Correlating radio astronomy signals with many-core hardware. *International journal of parallel programming*, 39(1):88–114, 2011.

[26] T. I. Vassilev. Comparison of several parallel api for cloth modelling on modern gpus. In *Proceedings of the 11th International Conference on Computer Systems and Technologies and Workshop for PhD Students in Computing on International Conference on Computer Systems and Technologies*, pages 131–136. ACM, 2010.

[27] R. Weber, A. Gothandaraman, R. J. Hinde, and G. D. Peterson. Comparing hardware accelerators in scientific applications: A case study. *Parallel and Distributed Systems, IEEE Transactions on*, 22(1):58–68, 2011.

[28] Wikipedia. Renderscript — Wikipedia, the free encyclopedia, 2004. URL http://en.wikipedia.org/wiki/Renderscript. [Online; accessed 25-Mayruary-2014].

[29] C.-y. Yang, Y.-j. Wu, and S. Liao. O2render: An opencl-to-renderscript translator for porting across various gpus or cpus. In *Embedded Systems for Real-time Multimedia (ESTIMedia), 2012 IEEE 10th Symposium on*, pages 67–74. IEEE, 2012.

[30] L. Zhang, B. Tiwana, Z. Qian, Z. Wang, R. P. Dick, Z. M. Mao, and L. Yang. Accurate online power estimation and automatic battery behavior based power model generation for smartphones. In *Proceedings of the eighth IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pages 105–114. ACM, 2010.