Performance Analysis of Parallel Languages on Android

CS 598SVA - Heterogeneous Computing

Abdul Dakkak Cuong Manh Pham Prakalp Srivastava

{dakkak, pham9, psrivas2}@illinois.edu

To do notes is enabled. Disable it to hide all notes

Abstract

– Abstract –

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 code on the device which optimized for it. 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. The differences are so deep that programmers are often experts on only one class of device, e.g., an expert GPU programmer will not have much DSP expertise 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 we want applications to use the full range of available hardware to maximize performance or energy efficiency or both, the programming environment has to provide common abstractions for available hardware compute units.

The industry and the research community have been trying to solve this problem. The recent development of RenderScript [9, 22] 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 the programmer to the run-time, unlike other solutions such as OpenCL, where the programmer has more control over the execution semantics of the application (*the programmer decides which part of application would run on which device and using which part of the memory hierarchy*). 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 the programmer that are critical to good performance on these accelerator. For example, in GPUs, the placement of data at various levels of memory hierarchy is critical to good performance. It is this reason that most programming languages for GPUs, allow the programmer unlimited control over memory management. RenderScript too can use GPUs for acceleration, but completely hides the memory management from the programmer. 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, while the rest of the responsibilities of memory management and work distribution among different compute units is handled by RenderScript compiler and run-time. This raises an important question of "how effective is the RenderScript compiler and run-time?", which we plan to answer by doing a comprehensive performance analysis of RenderScript.

2. Motivation and Background

Assuming Introduction has a sentence or two about the programmability problem.

Heterogeneity of hardware resources has led to a diverse landscape of different programming models, run-time systems, profiling and debugging tools for application development. The differences are so deep that programmers are often experts on only one class of device, e.g., an expert GPU programmer will not have much DSP expertise 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 we want applications to use the full range of available hardware to maximize performance or energy efficiency or both, the programming environment has to provide common abstractions for available hardware compute units.

OpenCL and RenderScript have been proposed to solve this problem. They are designed to accelerate data parallel computation intensive part of application code by allowing workload to be executed on multiple processing cores, GPUs, DSPs or a combination of these. Here we give a brief overview of OpenCL and RenderScript.

2.1 OpenCL

OpenCL was initiated by Apple Inc. and is managed by the Khronos Group [**?**]. It allows the 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.

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 program identifies the device on which the OpenCL kernel would be executed, sets up the environment, allocate memory on the device, copy data into the device memory and enqueue the kernel execution on the device.

In the Android world, where all application execute on Dalvik virtual machine, an application using OpenCL has a third component as well. This is the Java host code which would initialize the application, read in the inputs and use JNI calls to invoke OpenCL host code.

OpenCL is designed to be portable across a wide range of devices. However, developers often use hardware specific parameters such as the size of OpenCL work-group, shared memory size to obtain better performance on specific hardware. This harms the portability of OpenCL application kernel code.

2.2 RenderScript

RenderScript was released by Google as an official computing framework in 2011 [**?**]. The motivation behind RenderScript is to provide performance and portability across SoC architectures to RenderScript applications.

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 application host code to communicate with RenderScript kernel code, allowing functions such as memory binding between the host program and the kernels.

RenderScript compilation flow is shown in Figure **??**. First, `llvm-rs-cc` utility is used to compile RenderScript kernels to LLVM **??** bitcode (`*.bc extension`) files. LLVM bitcode is LLVM intermediate representation having back-end support for a wide range of hardware devices including general purpose processors, GPUs and DSPs. All OpenCL compilers too use a subset of LLVM IR as intermediate representation, thus making it a natural choice for RenderScript IR which has portability as one of its primary goal. During compilation of RenderScript kernels, `llvm-rs-cc` also generates the corresponding reflected Java classes of the kernels. Thereafter, the application host code, the reflected Java classes and bitcode are bundled together into the Android application package (`*.apk` file), which is installed on the Android device. During execution, the RenderScript runtime invokes `libbcc`, RenderScript back-end compiler to translate bitcode into appropriate machine code.

2.3 Why RenderScript evaluation is required?

RenderScript 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 the programmer to the run-time, unlike other solutions such as OpenCL, where the programmer has more control over the execution semantics of the application (*the programmer decides which part of application would run on which device and using which part of the memory hierarchy*). 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 the programmer that are critical to good performance on these accelerator. For example, in GPUs, the placement of data at various levels of memory hierarchy is critical to good performance. It is this reason that most programming languages for GPUs, allow the programmer unlimited control over memory management. RenderScript too can use GPUs for acceleration, but completely hides the memory management from the programmer. 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, while the rest of the responsibilities of memory management and work distribution among different compute units is handled by RenderScript compiler and run-time. This raises an important question of "how effective is the RenderScript compiler and run-time?", which we plan to answer by doing a comprehensive performance analysis of RenderScript.

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
MRI-Q	N	N	C	C	C	C
TPACF	B	B	C	C	C	C
Histogram	C	B	C	C	C	C
BFS	N					
MRI-G	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.

3. Benchmarks

We extend the Parboil benchmark suite and extend it 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 represent 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

VectorAdd, adds two floating point vectors with $8K$ elements. Compared to other benchmarks, vector add 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

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

The OpenMP code use the `#pragma omp parallel for shared(A, B, C) collapse(2)` pragma, 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.

3.3 TPACF

TPACF analyzes the angular distribution of astronomical objects. The algorithm computes the distance between all pairs of coordinates in a dataset and then performs histogramming. The results are collecting in 3 histograms which are then cross correlated to find the statistical spatial distribution of the astronomical bodies. For our analysis, use 100 datasets each containing 487 coordinates.

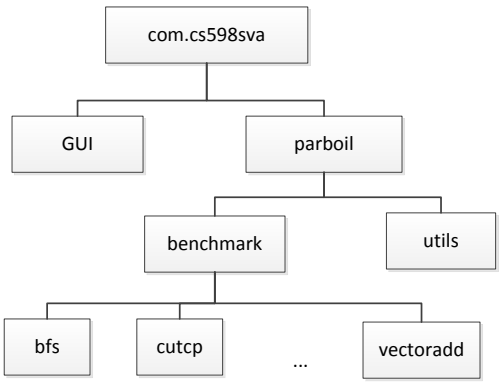


Figure 1: The Java package structure.

3.4 MRIQ

MRIQ 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 non-Cartesian space. The input dataset of size $32 \times 32 \times 32$ containing trajectory information in 3D as direction parameter in 2D.

3.5 Stencil

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

3.6 Histogram

Histogram computes the histogram of an input image. The input image used 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

4.1 Framework Structure

In order to facilitate the development of the benchmarks, we designed a framework, which contains (i) a hierarchy of the benchmarks, e.g., all different implementations of the same benchmark are group together, and (ii) utilities functions that support measuring and recording the results. Figure 1 depicts

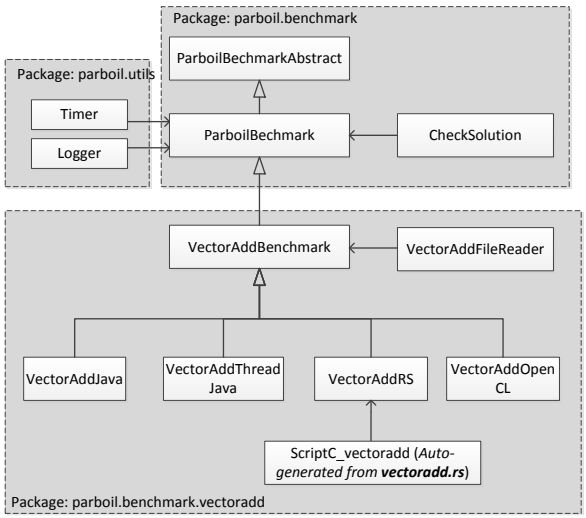


Figure 2: Class diagram of the VectorAdd benchmark.

the overview of the framework by mean of the Java package structure. Figure 2 provides a closer view at the organization of the classes related to the VectorAdd benchmark.

At a high level, we split the development of the graphical user interface (GUI), from the development of each of the benchmarks, under the `GUI` package and the `parboil` package in Figure 1, respectively. At this stage, the GUI is simply to allow users to start running the benchmarks, and to display the status of the benchmark execution, e.g., in which stage a benchmark is running, and whether a benchmark has executed successfully or failed at the end. In the future, more features will be added, such as displaying the final scores of the device.

The `parboil` Java package is the core of the project. In this package, we identified the common utility functions, such as timing- and logging-related functions, that are used across all the benchmarks. Those functions are grouped in classes, e.g., `Timer` and `Logger`, and implemented in the `utils` package. Each benchmark, e,g., `bfs` or `cutcp`, is implemented in a separate package under the `benchmark` package. Figure 2 depicts the class diagram of the VectorAdd benchmark. Other benchmarks have the same class structure. In order to implement a benchmark, we just need to inherit for the `ParboiBenchmark` class, which layouts a common skeleton. For each benchmark, we have to implement a class to read the input, for example `VectorAddFileReader` class in this case. Each computational kernel, e.g., for Java, threaded Java, RenderScript, or OpenCL, is implemented in a separate class, e.g., `VectorAddJava`, `VectorAddThreadedJava`, `VectorAddRS`, or `VectorAddOpenCL`, respectively. The implementation classes inherit from the parent class of the benchmark — `VectorAddBenchmark` in this case. This hierarchical design maximize the code reuse between benchmarks, and between computation kernel. It also allows for a consistent interface to run the benchmarks.

4.2 Utility Functions

4.2.1 Timer

The `Timer` class is an important utility in our benchmark. It determines the accuracy and flexibility of our measurement. The `Timer` class utilizes Android's `SystemClock`, which provides real-time clock at nanosecond resolution. Each `Timer` object consists of a dynamic list of `TimerElement` objects. Each `TimerElement` object is a measure of a particular execution segment, e.g., allocation time, setup time, and compute time. In order to create a new `TimerElement`, we just need to invoke the `Timer.start(category, message)` method at the beginning of the execution segment we wish to measure (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, we need to invoke the `Time.stop()` method. The timestamp and elapsed time will be automatically computed and recorded. The `Timer` class also can dump all the recorded `TimerElement` to a SQLite database or serialize it to the JSON format for conveniently storing and parsing the results.

If a benchmark is run multiple times, for example, we run the compute part 100 times for each small benchmark, and five times for each big benchmark in our currently analysis, then the `Timer` has a function to aggregate the results to output the average time across runs. Currently, we do not exercise that code and we leave the statistical analysis to the parser and visualizer.

4.2.2 Output to Database

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 extensive, all timing data is stored in memory and after the benchmark has run it is inserted into the database.

4.2.3 Load/Power Profiler

Since Android does not offer a way to capture processor usage information programatically, we use the Trepn [18] tool by Qualqomm to capture the data and save it into a `csv` file. While the tool is limited to Qualcomm based chipsets, it is reads internal processor counters as well as power rail information, both of which are not available programatically 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

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

Table 2: RSBench project line count breakdown.

use of Android's `Intent` framework. The `Intent` framework allows one to pass message between applications. Trpen records the time a message is recieved (which correspond 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 seperatly to decrease the overhead of the profiler. To reduce overhead, Trepn measure the processor usage information every $100ms$, both the frequency and the load are measured sequentially, we therefore need to correct that when parsing the `csv` file.

First, we parse each processor reading along with the time stamp for reading the file. Next, we interpolate the measured data (we use linear interpolant) and evaluate the interpolant at the application state times (these are the times Trepn recieved a signal 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 seperatly). Trepn can have measurment 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 the the 0.99th quantile of the data. While efforts have been taken to reduce the profiler's overhead, it still ocupies around a 10% overhead.

4.3 Implementing Computational Kernels

Table 2 shows the numbers of line-of-code (LOC) broken down into different types of programming languages in the RSBench project. The next sub-sections describe the process of implementing different types of computational kernels for each benchmark.

4.3.1 RenderScript Kernels

By design, each RenderScript kernel produces an *element* of the output. It is up to developers to define the granularity of an element. For example, in the VectorAdd benchmark, an output element can be one, or two, or three array elements of the output vector. This granularity essentially determines the parallelism of the written RenderScript code. In our benchmarks, we select the finest granularity of the output element, e.g., one array element in this case, to implement our RenderScript kernel.

In order to support this model, the RenderScript framework provides APIs to (i) define the type of `Element` for both input and output of RenderScript kernels, and (ii) pack `Elements` into `Allocation`, which is the data structure that is used to pass data back and forth between regular Java code and RenderScript code.

A significant effort of our work is to determine efficient ways to convert input data from original structure to the `Allocation` structure with appropriate `Element` type. This conversion is also reflected at runtime through the *setup* time category of RenderScript execution.

After data has been packed into `Allocation` objects, we move to write RenderScript code in C99 format. The code needs to be placed in a specific file with the $.rs$ extension and a proper header, so that the Android Development Tools (ADT) can automatically generate a wrapper Java class for it. The auto-generated Java class provides interfaces to invoke the RenderScript code from Java code.

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

4.3.2 Native Kernels

The Android operating system requires that C, OpenMP, and OpenCL kernels have to be invoked through Java Native Interface (JNI). The C and OpenMP versions of the code have been taken from Parboil with minimal modifications. Aside from wrapping the code via JNI so it is callable from within Java, little modification was done to the C and OpenMP code.

The OpenCL kernels are lifted from the parboil benchmark suite with no modifications. We use the `base` implementation of OpenCL — this a platform agnostic unoptimized implementation. Unlike the Parboil benchmark suite, we write the OpenCL host code using the C++ OpenCL API, this simplifies some of the code and affords us more code reuse oportunities.

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.

Native kernels are compiled with the `-O3 -ftree-vectorize -mvectorize-with-neon-quad` option allowing the compiler the autovectorization oportunity.

5. Analysis

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 hetrogenous 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

Name	CPU	GPU	Memory
GalaxyNexus	ARMv7, 2 cores, 1200 Mhz, SIMD NEON	PowerVR-SGX 540	694Mb
Nexus5	Qualcomm Snapdragon S4 Pro 1.5GHz	Adreno 320 400MHz	2Gb
Nexus7	Qualcomm Snapdragon 800 2.26GHz	Adreno 330 450MHz	2Gb
Nexus10	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.

consumption and performance is a balancing act that a programmer would like to deligate to the compiler or runtime.

In this section evaluate each benchmark on 5 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 on the market. Only the Nexus 5 and Nexus 7 devices can be modified to allow one to install an unofficial OpenCL implementation and only those two devices are fully supported by Trepn. Each benchmarks contains multiple implementation which is analyzed to determine tradeoff between processor utlization, 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 vary fine frequencies — operating at around 10 different frequencies), GPUs have coarse grained frequencies (operating at only 4 different frequencies).

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. The information gives us insight as to what parts of the code are active in each section of the code, how that impacts performance and battery.

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.

VectorAdd has a very high memory to compute ratio, therefore the processor utilization plot (figure 3) shows that not all CPUs are fully utilized during the compute phase. For the Nexus 5 device, while the graphs show a GPU is 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 runs in a single thread on both the Nexus 5 and Nexus 7 (figure 3). While OpenCL does utilize the GPU, because of the size of the matrix, there GPU has lower occupancy and therefore does not achieve peek performance. Both the RenderScript and ThreadedJava code make full utilization of all the cores.

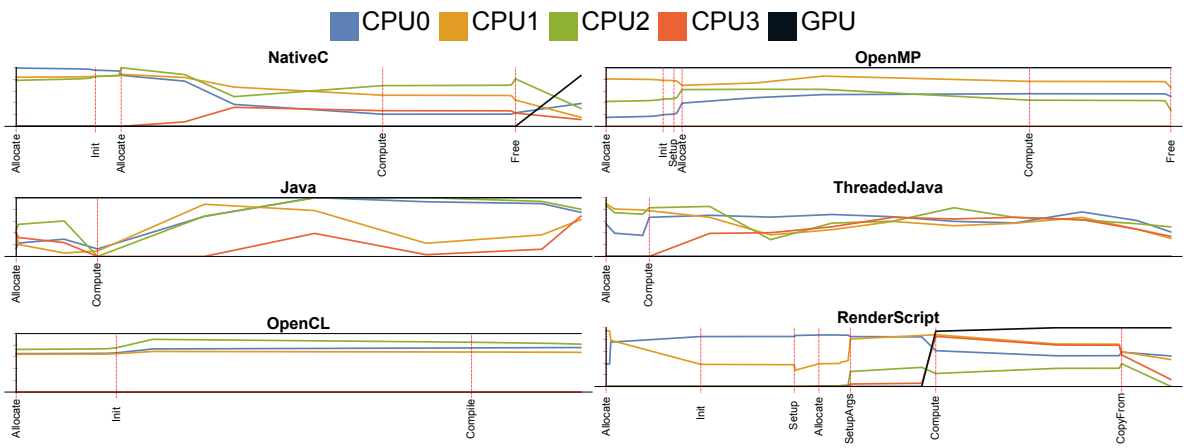
MRIQ is embarrassingly parallel and we can see full utilization for both the CPUs and GPUs in figure 4. Abrupt dips in the plots show positions where a kernel launch occurs. It is also interesting to note that on the Nexus 5 the CPU is not fully utilized for both ThreadedJava and RenderScript. This is 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 sake of code reuse, is done on the Java side using a single thread. Once that complete, we then execute the code on different datasets in parallel. The utilization plots (figure 4) show this behavior.

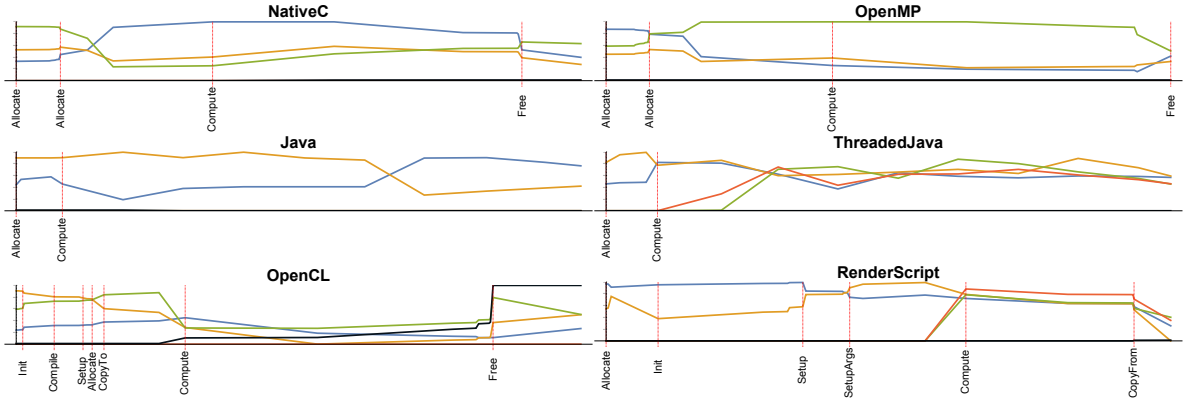
For Histogram (figure 5) we see that memory starts to play a major role in processor utilization. The GPU is being utilized in the memory copy as it is in the computation, for example.

For Stencil (figure 5) we see that RenderScript does not fully utilize all 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) result in the a substantial amount of resources being wasted.

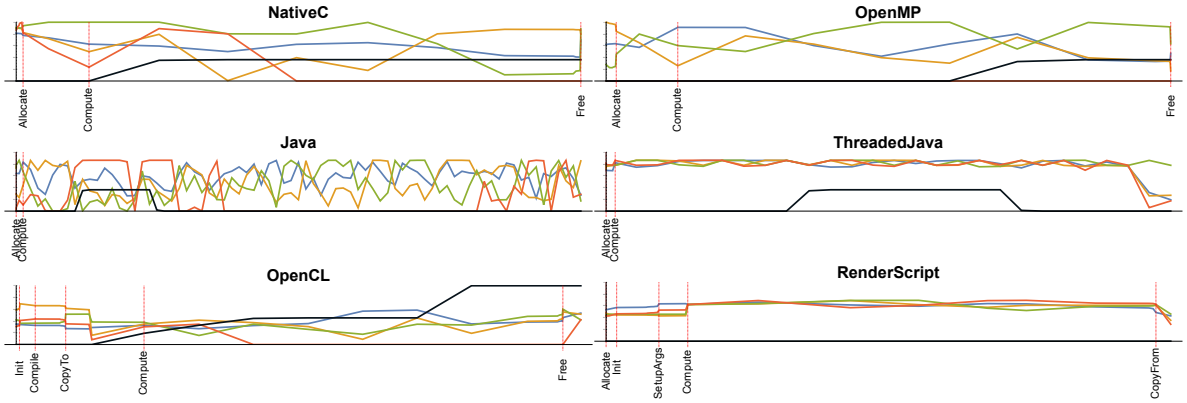
Throughout the benchmarks we see that RenderScript does not utilize the GPU. This is due to RenderScript requiring full IEEE 754-2008 compliance, which the GPU is



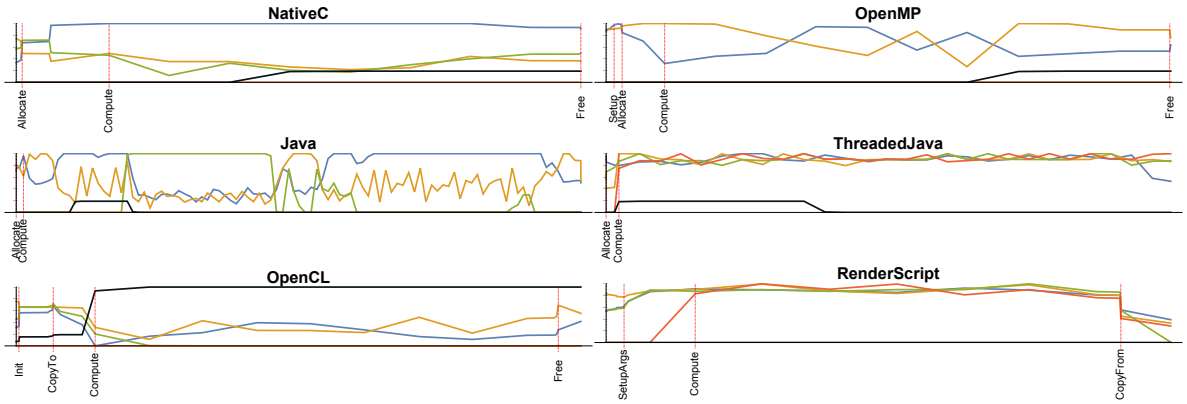
(a) VectorAdd on Nexus 5



(b) VectorAdd on Nexus 7

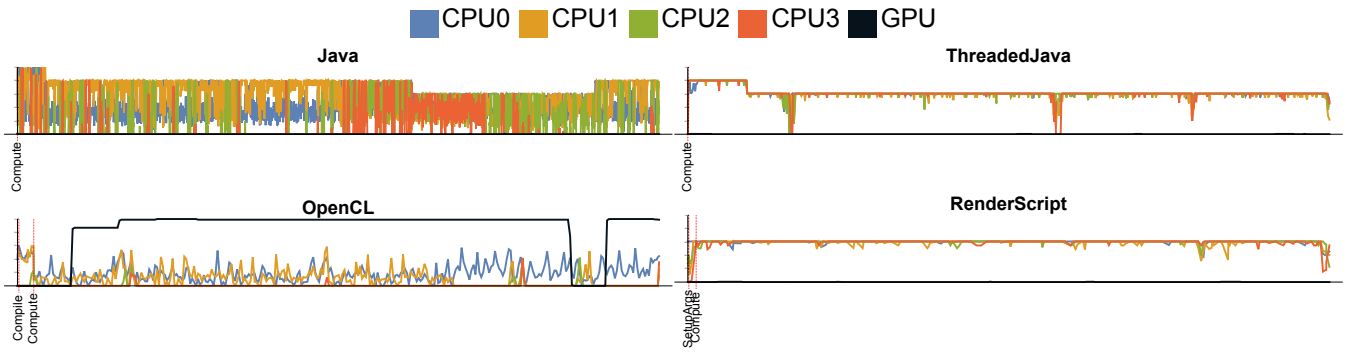


(c) Sgemm on Nexus 5

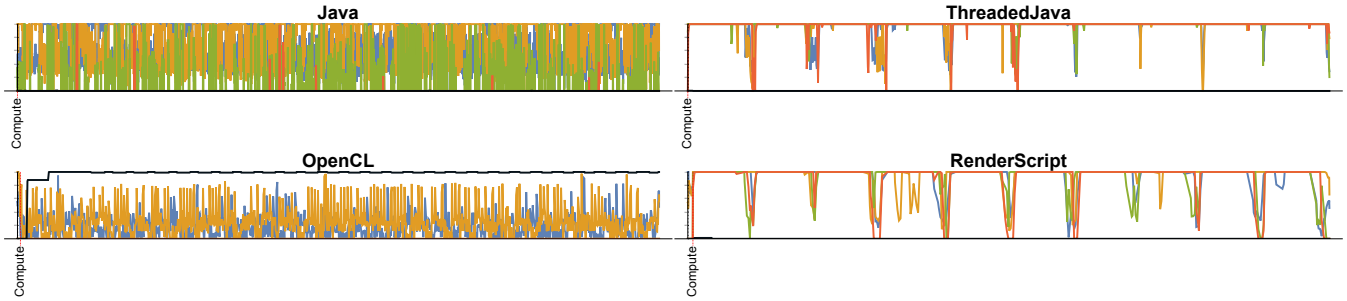


(d) Sgemm on Nexus 7

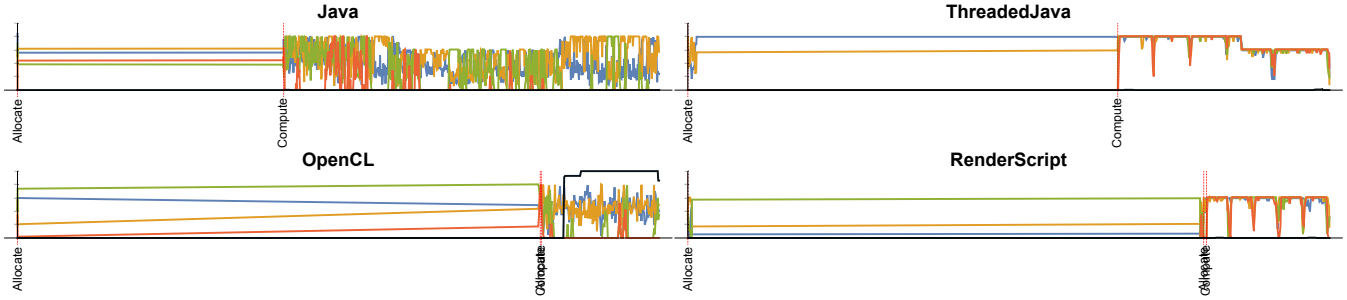
Figure 3: Processor utilization of VectorAdd and SGEMM for both Nexus 5 and Nexus 7. The x axis is time, and the y axis is normalized to the peek utilization for CPUs or GPUs across implementations.



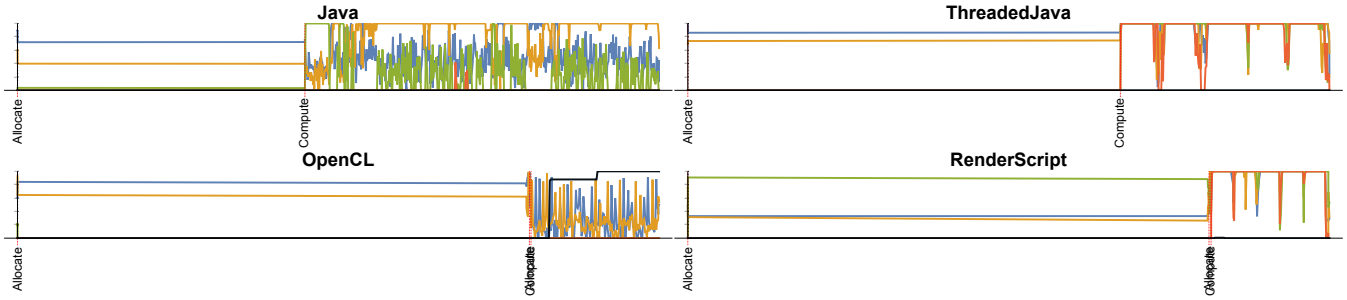
(a) MRIQ on Nexus 5



(b) MRIQ on Nexus 7

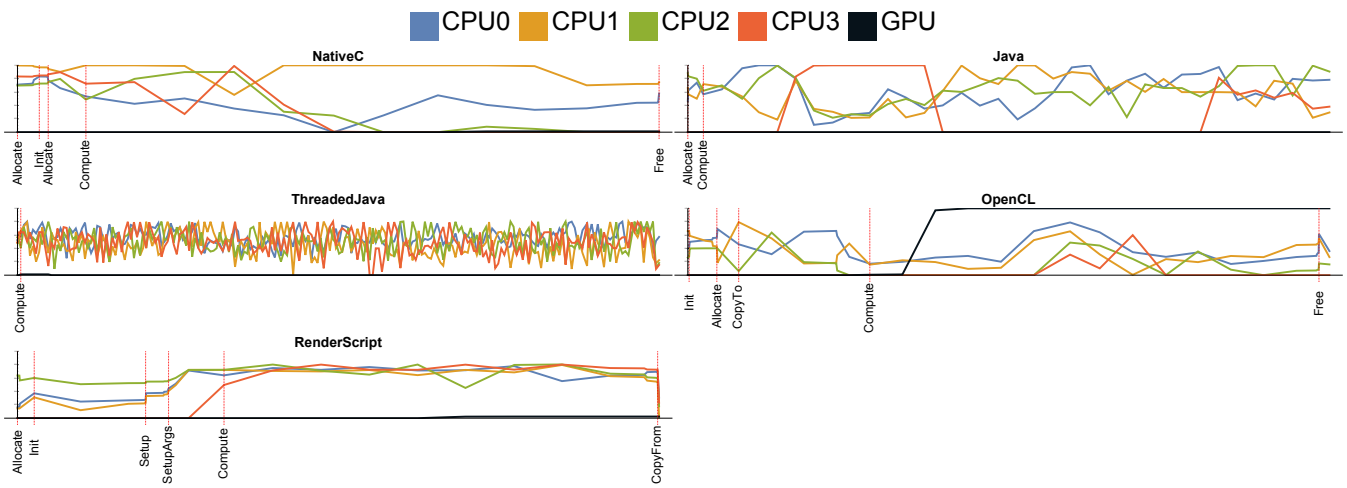


(c) TPACF on Nexus 5

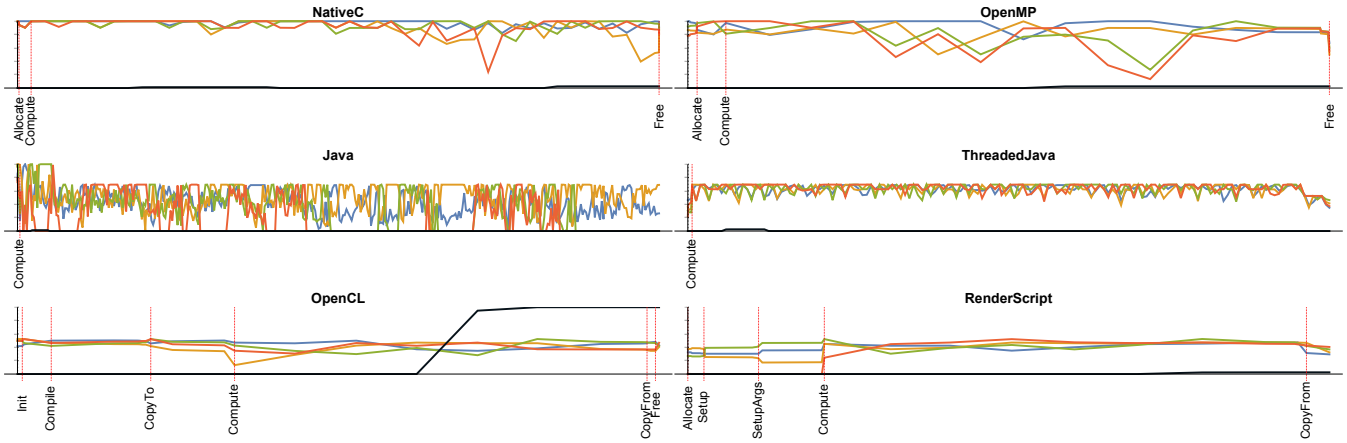


(d) TPACF on Nexus 7

Figure 4: 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 5: 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.

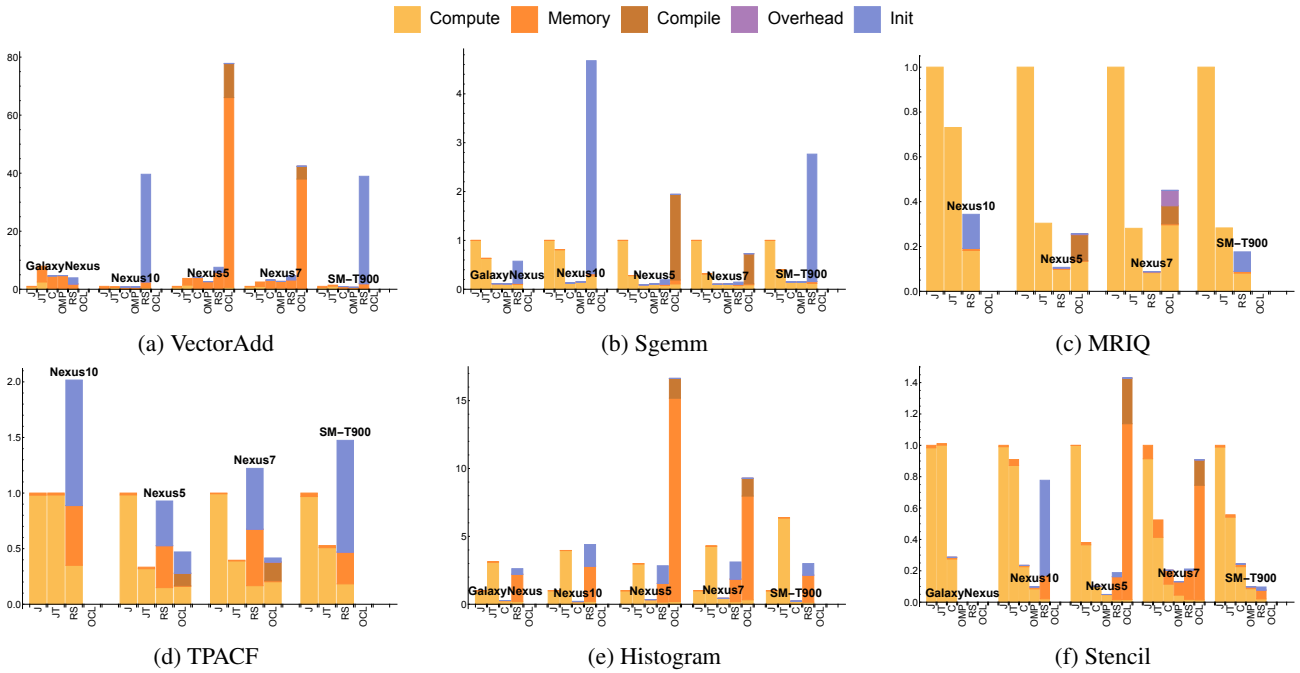
not able to provide. Inserting the `#pragma rs_fp_imprecise` pragma into the RenderScript kernel allows for relaxed IEEE compliance and should allow for GPU acceleration. Similar options are available in OpenCL and C, but since this option is not available in Java we chose to disable it. Further investigations would require us to analyze the results with relaxed precision.

5.2 Performance

The performance measurements are collected by measuring the time spent within each section of the 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 6) and therefore the overhead (memory, compilation, and initialization) have an impact, and one where the kernel is run 100 times (figure 7) (or 5 for both TPACF and MRIQ) and the overhead has little impact.

For each device, the plot show the time 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 3) — Trepn is not running while collecting these timing results. Not all benchmarks were run on the GalaxyNexus, this is due to the device being low end resulting in a long time to execute some of the benchmarks.

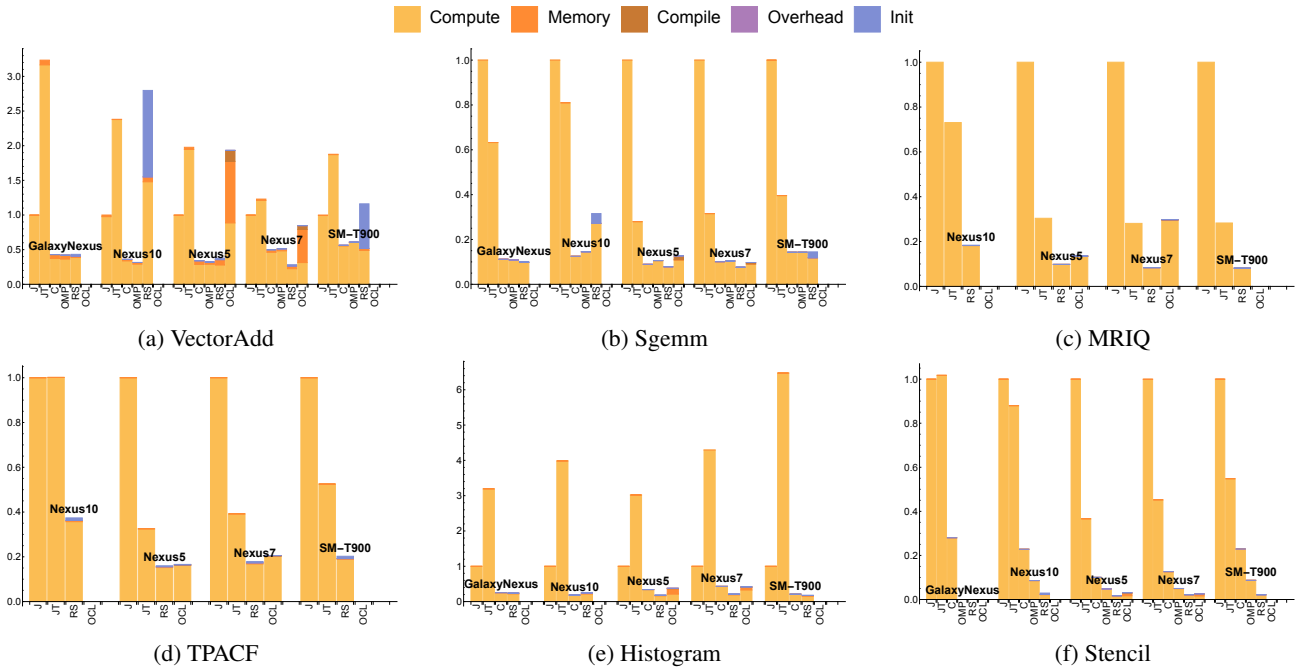
In figure 6 the compute code is only executed once, it is clear that the overhead of RenderScript on the Nexus 10 (and to some extent the SM-T900) device is consistently high. We suspect that the Nexus 5 and Nexus 7 are using a more recent version of the RenderScript library compared to the Nexus 10. As one would expect, a kernel is executed only once is not a good fit to be of-loaded to either RenderScript or OpenCL. This is due to overhead playing a big roll



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

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

Figure 6: 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 7: 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.

with overhead time being order of magnitude bigger than the compute time — 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 7, 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 see RenderScript's compute time to be similar to OpenCL, but is better when also considering overhead time. Both RenderScript and OpenCL outperform the OpenMP implementation in all benchmarks as well.

As expected, the SGEMM OpenMP timing is similar to that of C, confirming our hypothesis that the compiler was not able to interpret the OpenMP pragma. Because of the privatization, which requires an allocation in a 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.

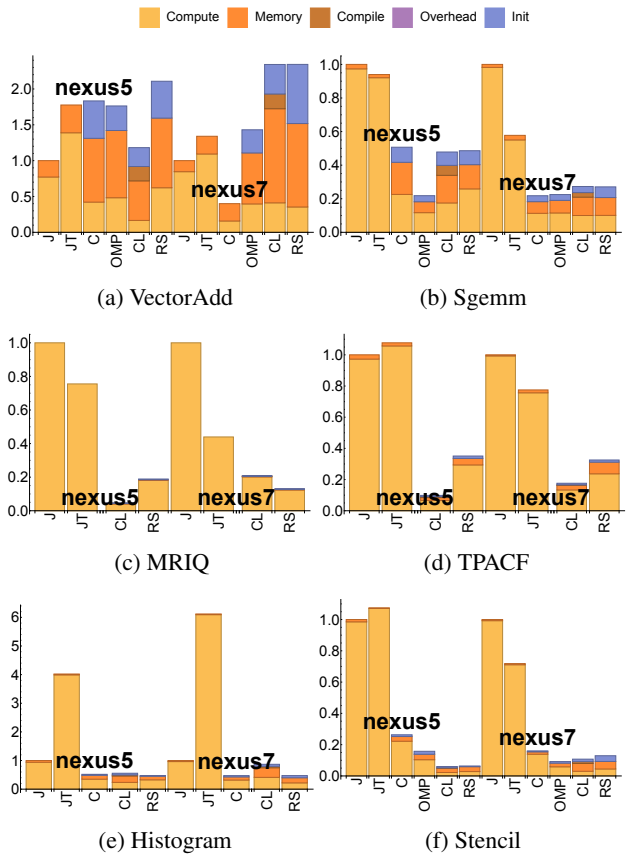
The biggest performance gain comes by not using the JVM, however. Aside from typical JVM overhead, we notice that these kernels are array access extensive. Since Java's semantics guarantee array accesses are within bounds, an overhead is incurred. Java's floating point semantics also do not match modern hardware (which implement the IEE 754 standard), this introduces more overhead where the JVM needs to perform extra checks. These overheads do not manifest themselves in our native implementations. We also use unsafe casts to reduce the overhead in the native implementations.

5.3 Energy Utilization

Since, mobile devices employ DVFS the energy utilization of the device at a specific time is governed by the operating frequency of the processor. One can model [7, 24] the battery usage of the power draw at time t by

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

where N 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, and $G_v(f)$ and $C_{v_n}(f)$ are the power draws for the processors at the specified voltage, $G_u(t)$ 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 turn them off.



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

Figure 8: Energy usage for both the Nexus 5 and Nexus 7 normalized to Java's energy utilization (lower is better).

The issues with using a model is determining the 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 is 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 8 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

utilize the GPU. 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, 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 any 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 power overall compared to the serial Java code.

5.4 Input/Output

Throughout the analysis, we have ignored IO times which is in fact the biggest performance bottleneck in these benchmarks. Unlike desktops which have a hard drive read speed of 80-100MB/s, Android devices use flash which have a 7-25MB/s read speed. In reality, however, computational data for mobile devices are not stored on disk and are usually either streamed from the cloud or captured from onboard sensors and therefore are available in RAM. To keep with the typical usage, and to not skew the plots, we therefore 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 eclipse platform to offer a continent 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. As described in section 4.3, this `Allocation` interface gives the programmer 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 features 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 model 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 a 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 (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

Before RenderScript execution starts, all heap data to be used by the kernel has to copied into `Allocation` buffers. The buffer has to be copied back to make it 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. But in the case of unsafe copies, these can be avoided when the kernel is executing 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 to a `Allocation` buffer if what is passed is a Java array.

5.5.8 Lack of a Standardization

RenderScript is similar to CUDA in this respect — the reference implementation is 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 support (in both the documentation and the header files) several data types, but a runtime exception is raises when use with types other than `int32_t`. The lack of specification results in some errors being unintuitive — one has to read the runtime to determine what the error refers to and what caused it. It also means that it is unlikely that RenderScript would be ported to desktop platforms.

5.5.9 Yet Another Parallel Framework

Aside from having full control over the language, 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 implementing via a library that interacts with OpenCL and there is no need for a new language. One reason could be that the compiler and runtime can prevent compute code that run on the GPU from possibly crashing the driver (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.

6. Related Work

In term of programming model, RenderScript is similar to OpenCL [13] and CUDA. While CUDA is designed specifically for NVIDIA GPU devices, OpenCL's goal is similar to RenderScript, which is aiming at simplifying cross-platform parallel programming for heterogeneous systems. In fact, Google had an option to adopt OpenCL, since some Android hardware already has OpenCL SDKs [16], but they opted to create RenderScript. Google justified the choice by arguing that they required not only performance portability and development efficiency, but also a more intuitive programming and distribution model. This decision caused some frustration from the OpenCL community [10] and some hardware vendors [2] who had made big investments on OpenCL.

Since being introduced in 2008, OpenCL performance and performance portability has been extensively evaluated. The most common of such evaluations is OpenCL's performance against CUDA on GPUs [3, 8, 11, 14, 19–21]. On GPUs, OpenCL and CUDA have a similar platform, memory, and programming model, thus an one-to-one analysis is possible. Most studies [3, 19–21], from a wide array of domains, show that CUDA usually achieves better performance (on NVIDIA GPUs) than OpenCL. Another consensus among these studies is that OpenCL provides a sufficient interface for developers to express more architectural details to improve the performance of their applications. For example, studies [14] and [8] show that most OpenCL kernels can obtain comparable performance with CUDA kernels when properly optimized.

According to [14] and [6], OpenCL achieves fairly stable performance across the tested platforms. However, both studies also illustrate some cases, in which 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 versions has an option to let the runtime decides the group size, i.e., the number of concurrent threads, or wrap size in CUDA's term. However, we are not aware of any study that evaluates the optimality of this feature.

There are many recent proposals aiming at making parallel software development easier and better utilizing multicores and GPUs available in mobile devices. Chen et al. [5] introduces a new programming model, called Android-Aparapi, which is based on the Aparapi model [4] to allow Android applications to run parallel Java code on GPU via OpenCL. Qualcomm has been developing an Android programming library and API, called MARE (Multicore Asynchronous Runtime Environment) [17], to facilitate writing parallel C code via Android Native Development Kit (NDK). Other studies focused on further simplifying the development of RenderScript code. In order to quickly reuse OpenCL legacy code in Android environments, Yang et al. [23] presents a source-to-source translator from OpenCL to RenderScript. The authors present several challenges of

this process, the most notable one is the differences in the execution models of the two standards. More recently, Acosta et al. [1] proposes a parallel development framework, called Paralldroid, for improving the parallelism of programs running in the Android platform. The framework automatically generates parallel code in C and RenderScript based on programmers' annotation in Java code. The results show that the auto-generated RenderScript code often achieves higher performance than the auto-generated Native C code.

In term of benchmarking RenderScript, study [12] compares three different programming models, namely RenderScript, Remote CUDA (RCUDA), and C++, of an image processing library in Android platform. The results on a Tegra 3 quad core device evidently show that RenderScript outperforms both the RCUDA and C++ implementations. Furthermore, this performance gap increases as the size of the input 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 we might be able to leverage is CompuBench mobile for RenderScript [15]. But since this benchmark is a commercial product and does not offer source, we would not be able to perform thorough analysis using this it. So in this regard, we will be providing the first open source benchmark for RenderScript that can be evaluated against different language paradigms and hadware targets.

7. 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.

Furthermore, unlike the C or OpenMP versions 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 choose RenderScript, over C or OpenMP, when writing compute intensive kernels.

8. 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's 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.

References

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

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

[3] 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.

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

[5] 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. .

[6] 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.

[7] 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.

[8] 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.

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

[10] V. Hindriksen. Google blocked opencl on nexus with android 4.3, May 2014. URL `http://tinyurl.com/qeeupgr`.

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

[12] 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.

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

[14] 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.

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

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

[17] Q. D. Network. Parallel computing (mare), May 2014. URL `https://developer.qualcomm.com/mobile-development/maximize-hardware/parallel-computing-mare`.

[18] T. Profiler. Qualcomm.

[19] 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.

[20] 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.

[21] 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.

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

[23] 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.

[24] 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.

Todo list

To do notes is enabled. Disable it to hide all notes . . 1

Assuming Introduction has a sentence or two about the programmability problem. 1

9. Appendix

Listing 1: Bitcode representation of RenderScript kernel

```
@blockDim = common global i32 0, align 4

; Function Attrs: nounwind readonly
define i32 @root(i32 %x, i32 %y) #0 {
  %1 = load i32* @blockDim, align 4, !tbaa !5
  %2 = mul i32 %1, %x
  %3 = add i32 %2, %y
  %4 = tail call fastcc i32 @ave(i32 %3, i32 %3)
  ret i32 %4
}

; Function Attrs: nounwind readnone
define internal fastcc i32 @ave(i32 %a, i32 %b) #1 {
  %1 = add nsw i32 %b, %a
  %2 = sdiv i32 %1, 2
  ret i32 %2
}

!\23pragma = !{!0, !1}
!\23rs_export_var = !{!2}
!\23rs_object_slots = !{}
!\23rs_export_foreach_name = !{!3}
!\23rs_export_foreach = !{!4}

!0 = metadata !{metadata !"version", metadata !"1"}
!1 = metadata !{metadata !"java_package_name", metadata !"com.cs598sva.rsbench.sgemm"}
!2 = metadata !{metadata !"blockDim", metadata !"5"}
!3 = metadata !{metadata !"root"}
!4 = metadata !{metadata !"58"}
!5 = metadata !{metadata !"int", metadata !6}
!6 = metadata !{metadata !"omnipotent char", metadata !7}
!7 = metadata !{metadata !"Simple C/C++ TBAA"}
```

Listing 2: NVVM representation of CUDA kernel

```
define i32 @ave(i32 %a, i32 %b) {
entry:
  %add = add nsw i32 %a, %b
  %div = sdiv i32 %add, 2
  ret i32 %div
}

define void @simple(i32* %data) {
entry:
  %0 = call i32 @llvm.nvvm.read.ptx.sreg.ctaid.x()
  %1 = call i32 @llvm.nvvm.read.ptx.sreg.ntid.x()
  %mul = mul i32 %0, %1
  %2 = call i32 @llvm.nvvm.read.ptx.sreg.tid.x()
  %add = add i32 %mul, %2
  %call = call i32 @ave(i32 %add, i32 %add)
  %idxprom = sext i32 %add to i64
  %arrayidx = getelementptr inbounds i32* %data, i64 %idxprom
  store i32 %call, i32* %arrayidx, align 4
  ret void
}

declare i32 @llvm.nvvm.read.ptx.sreg.ctaid.x() nounwind readnone

declare i32 @llvm.nvvm.read.ptx.sreg.ntid.x() nounwind readnone

declare i32 @llvm.nvvm.read.ptx.sreg.tid.x() nounwind readnone

!nvvm.annotations = !{!1}
!1 = metadata !{void (i32*)* @simple, metadata !"kernel", i32 1}
```

Listing 3: SPIR representation of OpenCL kernel

```
; Function Attrs: nounwind readnone
define i32 @ave(i32 %a, i32 %b) #0 {
entry:
  %add = add nsw i32 %b, %a
  %div = sdiv i32 %add, 2
  ret i32 %div
}

; Function Attrs: nounwind
define void @simple(i32 addrspace(1)* nocapture %data) #1 {
```

```
entry:
  %call = tail call i32 bitcast (i32 (...)* ↩
      @get_global_id to i32 (i32)*)(i32 0) #3
  %call1 = tail call i32 bitcast (i32 (...)* ↩
      @get_global_size to i32 (i32)*)(i32 0) #3
  %mul = mul nsw i32 %call1, %call
  %call2 = tail call i32 bitcast (i32 (...)* ↩
      @get_local_id to i32 (i32)*)(i32 0) #3
  %add = add nsw i32 %mul, %call2
  %add.i = shl nsw i32 %add, 1
  %div.i = sdiv i32 %add.i, 2
  %arrayidx = getelementptr inbounds i32 addrspace↩
      (1)* %data, i32 %add
  store i32 %div.i, i32 addrspace(1)* %arrayidx, ↩
      align 4, !tbaa !1
  ret void
}

declare i32 @get_global_id(...) #2

declare i32 @get_global_size(...) #2

declare i32 @get_local_id(...) #2

!opencl.kernels = !{!0}

!0 = metadata !{void (i32 addrspace(1)*)* @simple}
!1 = metadata !{metadata !"int", metadata !2}
!2 = metadata !{metadata !"omnipotent char", ↩
    metadata !3}
!3 = metadata !{metadata !"Simple C/C++ TBAA"}
```