Progress Report 1: Performance Analysis of RenderScript

Abdul Dakkak Cuong Manh Pham Prakalp Srivastava
University of Illinois at Urbana-Champaign
{dakkak, pham9, psrivas2}@illinois.edu

Progress Summary

This section summarizes our progress to date. Further information on each task is described in the subsequent sections.

We have successfully setup our development infrastructure, which allows us to efficiently collaborate and develop on both the emulator and hardware devices. phones. Due to the diversity of the Android versions (4.2, 4.3, and 4.4), the SDKs (API-17, API-18, and API-19), and the hardware configurations of different devices, we had a chance to evaluate the portability of our framework. For example, we did learn that the datasets generated for traditional computers are often too large for mobile devices, thus we needed to generate different datasets for our mobile benchmarks. Also, due the wide range of hardware resources, a dataset that is suitable for a tablet may be too big for a phone. Therefore, we will need to have a clever way to select suitable datasets and make fair comparison between devices.

In term of porting the Parboil benchmarks, we have completed the overall framework and three benchmarks, namely VectorAdd, Stencil, and Segmm, and close to finish three more benchmarks, namely, CUTCP, TPACF, MRIQ. The status of the benchmarks porting are summarized in Table 1. We spent a significant amount of effort to make the benchmark framework robust and convenient to work on. For example, the framework contains a Timer utility that allows us to record time of different segments of each benchmark. We also implemented a database back-end to store the benchmark results for later analysis.

In term of measurement and analysis, for each implemented benchmark, we compare three versions, Java, threaded Java, and RenderScript implementations, across two physical devices: a Samsung Galaxy Nexus phone and a Google Nexus 7 tablet. The results are detailed in the *Analysis* section.

Benchmark	Porting status
Matrix Multiply	
– Java	Completed
– RenderScript	Completed
Stencil	
– Java	Completed
– RenderScript	Completed
VectorAdd	
– Java	Completed
– Threaded Java	Completed
– RenderScript	Completed
CUTCP	
– Java	Completed
– Threaded Java	Partial
– RenderScript	Partial
MRI-Q	
– Java	Completed
– Threaded Java	Partial
– RenderScript	Partial
TPACF	
– Java	Completed
– RenderScript	Partial
Histogram	Next phase
Breadth-First Search	Next phase
MRI-Gridding	Next phase
Sum of Absolute Differences	Next phase
Spare-Matrix Vector Multiply	Next phase
Lattice-Boltzmann Method	TBD

Table 1. Parboil Benchmark Porting Status

Methodology

In our proposal we stated that if we were able to run OpenCL on the Android devices, then we will benchmark against OpenCL. Sadly, we were not able to find a way to run OpenCL on the devices. Instead, we opted to implement a serial version of the code in Java along with a multithreaded version and compare both against RenderScript. This is added work, since we now implement 3 versions of the same benchmark, but since we usually start with the serial Java version, make it multithreaded, and then port the multithreaded code into RenderScript.

All the implementations share a common interface, which means our framework takes the code, runs it, and the information is stored in a database on the device which we use for analysis.

RenderScript Kernels

Currently, the benchmarks we have ported have naive RenderScript kernel implementations. Our aim has been to write the RenderScript kernels allowing maximum parallelization and let the RenderScript runtime perform locality optimizations (such as, deciding how to group parallel threads when running on a GPU), if any. We believe that this is the right approach as the purpose of RenderScript was to keep the programmer oblivious to the low level hardware details.

We would be working on finding techniques of improving performance of RenderScript kernels. However, we have realized that it has several challenges.

Firstly, the RenderScript programming environment prevents the standard locality optimizations by the programmer, such as, tiling. This is primarily because these optimizations require knowledge of memory sizes, e.g., cache sizes, scratchpad sizes on GPUs, details which we cannot use as we are not aware of the processing unit RenderScript kernel is running on.

Secondly, since RenderScript is relatively new, we could not find many examples of improving performance of RenderScript kernels. Also, not having detailed documentation on RenderScript makes it difficult to write high performance RenderScript kernels. For example, there are multiple ways to pass read-only data to RenderScript runtime. However, we are not sure of the overheads involved with each approach. With more experience and experiments, we believe we would have better understanding of such optimizations and could improve RenderScript performance.

Lastly, RenderScript programming environment has certain idiosyncratic restrictions which make it difficult to use. For instance, the input and output *Allocation* to a RenderScript kernel have to be of same dimensions. Such restrictions are counter-intuitive and add to the complexity of using RenderScript.

Performance Analysis

We tried to mirror some of the API decisions found in Parboil, this is the same for the timer code. We implemented a class that allows us to time sections of code and store dump them into the database.

```
timer.start("IO", "Reading inputs");
float[] inputData = ReadVectorData("input.data");
timer.stop();
```

Unlike Parboil's timers however, our interface does not aggregate the data. This is left for the analysis pass. The reason is that we feel like Parboil loses some information about which sections are hot in the code by aggregating all the data for you.

Power Analysis

Many hours were wasted trying to find a way to programatically read power data. A way that may work in debug mode, but other than that it seems like every vendor offers their own counter and there is no unified Android API to measure how much power is being drawn at a specific point in time. If we have extra time, we will revisit this later in the project.

Analysis

The emulator is used to perform the debugging (although it does not seem to be able to run RenderScript code). To run the code, we are using two devices for the analysis. The first is a Nexus 7 with the following specs:

```
CPU: Qualcomm Snapdragon S4 Pro, 1.5GHz
GPU: Adreno 320, 400MHz
Memory - 2 GB
Storage 32 GB
```

The second is a Samsung Galaxy Nexus with the following specs:

```
CPU: ARMv7, 2 cores, 1200 Mhz, SIMD NEON
Memory: 694, JVM max: 96 MB
GPU: PowerVR SGX 540.
```

Preliminary Results

While we do have implementations for 3 out of the 4 benchmarks we promised in the proposal schedule, our results only show part of those. The data has not been fully analyzed, but we can give some preliminary insights into the expected results.

The IO times are around 1000 times more than the compute times on Android devices. Therefore, in the plots we chose not to include the IO times, since that heavily skews the plots, and using a log scale it makes the other data almost the same.

The plots record the time it took to allocate the data (which should be similar across implementations), the compute time, and for RenderScript the setup time — this is the time it takes to load the RenderScript code as you prepare for computation (it may include jitting the code for example).

Vector addition, shown in figure 1, is a simple kernel and we see that the serial version outperforms both RenderScript and the threaded Java version. For the RenderScript, we suspect this is overhead from translating between the Java data arrays and the RenderScript arrays. For the threaded version, we use a thread for each element calculation (in the same style as you'd do in CUDA), and therefore you result in huge runtime penalty because of launching many threads. Performing thread coarsening would decrease this impact, but that has not been done.

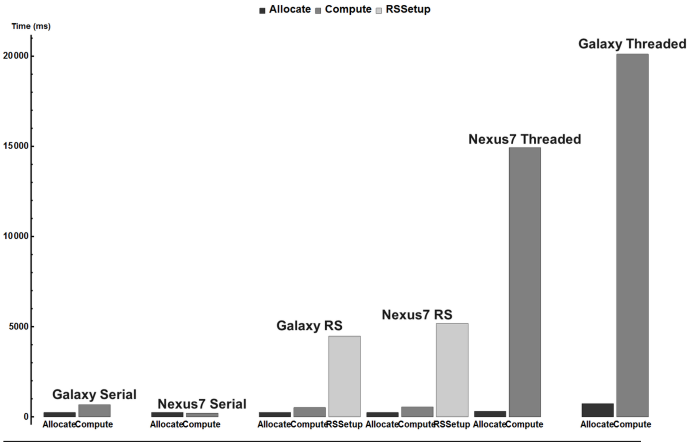

Figure 1. VectorAdd Benchmark.

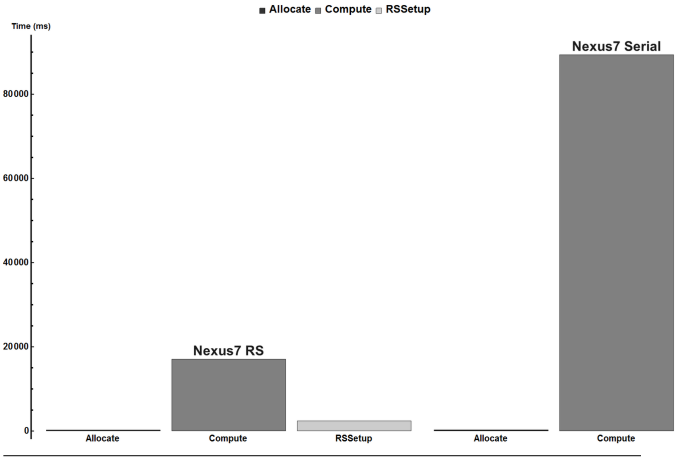

Figure 2. Matrix Matrix Multiplication Benchmark.

The large RenderScript setup time might either be because this benchmark does not take too much time to run and therefore the difference is apparent, or because it is the first benchmark we run and the RenderScript initial load time is added. Further investigation is required, but in general we plan on extending our benchmark framework so it performs multiple runs, discarded the time for the first run, and computes the mean time.

Matrix multiplication, shown in figure 2, performs matrix multiplication. The data is stored in column major order and we perform the computation in column major order. One possible optimization is to transpose the data beforehand to get better locality when performing computation. Regardless, we see a 4x performance improvement in SGEMM. We have not implemented a threaded Java version of the SGEMM benchmark, which is why it's missing from the plot.

Note, the semantics of Java's single precision floating point operations are slightly different from those of C, so do not expect to always use single precision for all the benchmarks (this was seen in the initial porting of TPACF).

The final benchmark implemented is stencil shown in figure 3. Again, we did not implement the threaded Java version of this code, but we can see that the RenderScript version outperforms the serial

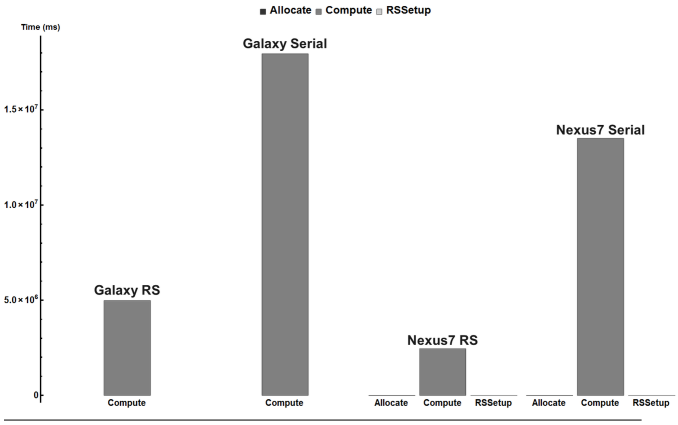

Figure 3. Stencil Benchmark.

code. RenderScript has not scratch pad memory constructs, so we were not able to use fine grained memory management to achieve high throughput. We further suspect that this code is not fully optimized to use all the RenderScript constructs available. Finally, the serial versions have an added (negligible) `RSSetup` time, this is either due to bad parsing of the output data or an incorrect labeling of a timer in the source code.

Complications

Our main complication came from the fact that we have never developed Android applications before. We approached it as if we were developing Java applications, but quickly realized that there are some constraints placed by the Android environment.

Furthermore, because the Android software and hardware is fragmented, references we found on how to solve certain problems either did not pertain to the hardware we were using or the Android version we are targeting. This is the main reason why we were not able to capture power information from the device, since each device has a different file system path that stores those counters.

Finally, the datasets used by Parboil each have a different format. This means that for each benchmark we have to write a file reader and writer to read in the proper benchmarks. The datasets are also quite large — a few GB. At first, we were bundling the data with the application which resulted in the Android application to balloon into a 1GB application. We have now moved the data to be read from the SD card. One possible alternative is to generate random data (according to some distribution). While this is simple for benchmarks such as matrix multiplication, it is not clear how one would generate meaningful data for the MRI benchmark.

Schedule

We think we are not too far behind of our initial projection. We had setup around 1-2 weeks to allow us to recover from some set backs. In terms of how far behind we are, we are around 2-3 days behind mainly related to the fact that we still need to do some more work on the infrastructure and complete the CUTCP implementation. To allow us to stay on schedule, we will not be implementing the Lattice Bolzmann Machine (LBM) benchmark.

We have revised the gant chart we included in the proposal to reflect our new projected timeline.

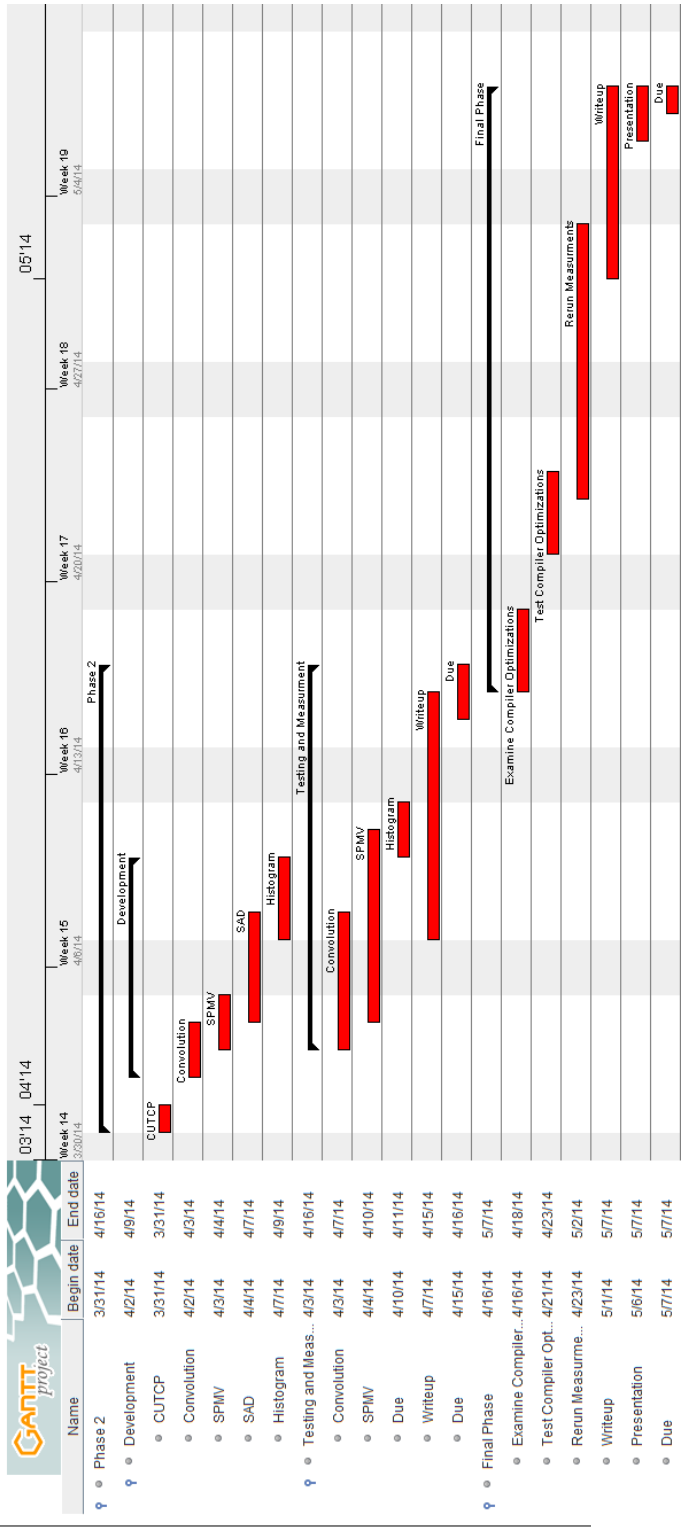

Figure 4. Projected project schedule.