Using RenderScript and RCUDA for Compute Intensive tasks on Mobile Devices: a Case Study

Roelof Kemp, Nicholas Palmer, Thilo Kielmann, Henri Bal
VU University
De Boelelaan 1081A
Amsterdam, The Netherlands
{rkemp, palmer, kielmann, bal}@cs.vu.nl

Bastiaan Aarts, Anwar Ghuloum
NVIDIA
2701 San Tomas Expressway
Santa Clara, CA, USA
{baarts, aghuloum}@nvidia.com

Abstract: The processing power of mobile devices is continuously increasing. In this paper we perform a case study in which we assess three different programming models that can be used to leverage this processing power for compute intensive tasks. We use an imaging algorithm and compare a reference implementation of this algorithm based on OpenCV with a multi threaded RenderScript implementation and an implementation based on computation offloading with Remote CUDA. Experiments show that on a modern Tegra 3 quad core device a multi threaded implementation can achieve a 2.2 speed up factor at the same energy cost, whereas computation offloading does neither lead to speed ups nor energy savings.

1 Introduction

Over the last two years we observed that mobile processors not only increased in clock speed, but also made the step to multicore. With the introduction of dual core processors for mobile devices in 2010 and quad core processors in 2011, the available raw compute power increased significantly. When hardware shifts the horizon of compute power, there is always software that takes advantage of it. Today, for example, high end mobile devices offer a gaming experience near or better than console quality.

With the increase of processing power came also an increase in complexity of the mobile hardware, such as the aforementioned multicore processors, but also dynamic frequency and voltage scaling, power gating and more. Whereas some complexity is transparent to software, others require software to be rewritten in order to take full advantage of it, which in turn adds complexity to the software.

This is especially true for multicore processors, that can only unleash their processing power if applications are written to execute in multiple threads. While the default programming languages on the major mobile platforms (Android: Java, iOS: Objective-C,

Windows Phone: C#) offer multi threading by default, it is still up to the developer to use it. Moreover, it is likely that if a certain app really needs performance it will be using either a lower level language – such as C – or a specialized high performance language such as Google's RenderScript [Ren].

Next to performance gains through the use of multi threading and specific languages, it has been noted that over the years the gap in computation power between mobile and non mobile devices got smaller. Despite this improvement, the fundamental constraints of a mobile device with respect to size, heat dissipation and power supply remained and therefore non mobile devices continue to offer more processing power than their mobile counterparts. A technique that takes advantage of this difference is *computation offloading* where heavy computation is transferred to non mobile devices to decrease execution time and/or save energy [LWX01].

In this paper we assess three different programming models; low level language, specialized language and computation offloading, using a case study for a High Dynamic Range photography app on a quad core mobile device. We compare the resulting implementations based on execution time and energy usage.

We find that the specialized compute language implementation leads to speed ups of max 2.2 times on a quad core device at the same energy cost; the use of computation offloading, however, has no benefits both in execution time and energy usage.

The remainder of this paper is organized as follows. Section 2 discusses the hardware and software as well as the programming models that we use in our case study. Then in Section 3 we detail the implementations for multicore and computation offloading, after which we discuss the methodology of our experiments in Section 4. Then in Section 5 we discuss the results of the experiments and we conclude in Section 6.

2 Background

2.1 Hardware

For this study we use the NVIDIA Tegra 3 Developer tablet, the only available mobile quad core processor during our study. The Tegra 3 SoC has a 4-PLUS-1 architecture which has either a low power core active if the load is low, or 1-4 normal cores if the load is higher. The maximum clock speed – 1.4 GHz – of the device can only be reached when one of the four normal cores is active; as soon as multiple cores are active the clock speed is reduced to 1.3 GHz.

On the developer device it is possible to explicitly turn cores on or off with *hotplug*. Furthermore, the device has several power rails on which different components are placed. For each power rail the amperage and power consumption can be read out in real time, both in software and with a special breakout board that can be connected to a regular computer. The NVIDIA Tegra 3 Developer tablet runs the Android 4.0.3 operating system.

Figure 1: Example of three input images with different exposure levels and the resulting HDR image. Images from http://en.wikipedia.org/High_dynamic_range_imaging

2.2 Software

The application that we focus on during the case study is a demo Camera app – similar to the standard Android 4.0 Camera app – with enhanced photography functionality, such as Negative Shutter Lag and High Dynamic Range (HDR) photography.

More specific, we focus on *exposure fusion*, a computer vision algorithm within the HDR process. This algorithm performs the computationally expensive operation of fusing together multiple images taken from roughly the same spot and the same time with different exposure levels, in such a way that the result image shows details in both dark and bright regions, which can greatly improve the end-user experience of capturing images in scenes with details in both the darker and brighter areas. An example of such a scene and the resulting HDR image is shown in Figure 1.

A detailed specification of the algorithm can be found in [MKVR07]. Relevant for this paper is that the computation in this algorithm is based on matrix, filter and pyramid operations, are all data parallel operations. We use a *control script* around these operations to control how the output of one operation is used as input for another operation.

2.3 Programming Models

Android suppports multiple programming models and languages (see Table 1). Using the Android's Software Development Kit (SDK) Java is the default language, but one can switch to the Native Development Kit (NDK), which uses C/C++ through the Java Native Interface (JNI) for better performance.

Furthermore, with the introduction of multicore processors and the expectation of GPGPU computing becoming available on mobile devices the RenderScript programming model was introduced. RenderScript allows programmers to write kernels that at runtime are automatically run in parallel on the hardware selected by the RenderScript runtime. Until recently RenderScript could only execute code on CPUs, but the Nexus 10 now supports

Table 1: Programming Models on Android

	Language	OpenCV	Automatic Parallelism
SDK	Java	Yes	No
	RenderScript	No	Yes
NDK	C/C++	Yes	No
	OpenGL	No	Yes
Computation Offloading	RCUDA	Yes	Yes

execution on the GPU. Before there was hardware available on which RenderScript can run on the GPU, it was already possible to do general computational jobs on the GPU by writing OpenGL shaders, provided the algorithm and data can be expressed in graphics operations and graphics data.

In addition to the programming models provided by the Android platform that ultimately execute code on the device itself, one can also use a programming model that uses the communication means of a mobile device to offload computation to another better suited device, such as a desktop machine. An example of such a programming model is the Remote CUDA (RCUDA) computation offloading framework.

Of major importance for the subject of our field study, the HDR algorithm, is the availability of the common open source imaging library OpenCV [Ope]. The OpenCV library is written in C++, and has Java wrappers so that it can be used from both the SDK and the NDK on Android. OpenCV is not only used on mobile devices, but also on desktop systems. On desktop systems there are many kernels, for which OpenCV has GPGPU implementations written in CUDA[CUD] to employ data parallelism on the imaging data. With OpenCV's default Android build however, data parallelism is turned off, because no current mobile hardware running Android supports CUDA. However, with the RCUDA computation offloading framework we are able to run the OpenCV library *with CUDA support* on Android in combination with a CUDA enabled device hosting a server. The implementation of RCUDA that we used is similar to the framework described by Duato et al. [DIM+10], however their framework is targeted at HPC cluster systems.

The demo Camera application we use in our field study comes with an implementation built on top of OpenCV in the NDK, which has the drawback of being single threaded. We use this implementation as reference to two new implementations: an implementation with RenderScript that should automatically use all the available cores on the Tegra 3 and an implementation with RCUDA to study the impact of computation offloading. [1]

The focus of the remainder of the paper is on execution times and energy usage of the various implementations. A more qualitative comparison between the SDK, NDK and RenderScript can be found in [QZL12].

[1] Although we have an OpenGL implementation of the algorithm, the mapping of the algorithm to graphics primitives and the limited memory of the GPU resulted in an OpenGL algorithm that does not produce the same results as the others, so we chose to leave this out in the remainder of the paper. We did not make an implementation in Java, because it is essentially the same as the native implementation, however with the addition of overhead due to the Java wrappers in OpenCV if used from the SDK.

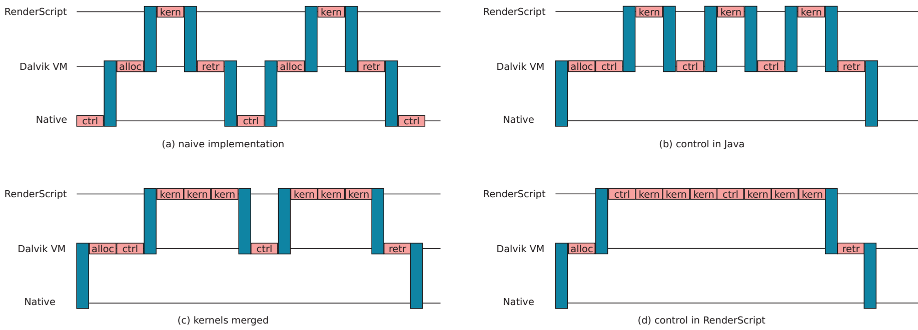

Figure 2: Schematic overview of the optimizations of the RenderScript implementation. ctrl: control script, alloc: memory allocation, kern: execution of a kernel, retr: retrieval of the results. The vertical boxes indicate context switching (JNI) overhead. The boxes are not proportional to the actual execution time.

3 Implementations

3.1 RenderScript

RenderScript is a host/device language similar to languages such as OpenCL and CUDA. RenderScripts host code is written in Java, whereas the device code is written in C99. Memory allocations can only be done in host code. Device code can be started single threaded or multi threaded. If device code is multi threaded it executes a particular kernel for each element of a 1, 2, or 3 dimensional array. RenderScript automatically distributes the elements over the available processors and also does load balancing. While the kernel code is written in C99, it is compiled to an intermediate byte code which, at runtime, gets compiled to the appropriate instructions for the hardware that the RenderScript runtime selects.

To port the reference implementation to RenderScript we initially replaced the OpenCV calls with RenderScript equivalents, while leaving the control code as is. This naive port allowed for easy debugging of the RenderScript kernels because we could switch between OpenCV and RenderScript at kernel level and therefore debug kernels individually instead of debugging the entire algorithm.

Once all kernels were correctly ported we started optimizing the code. The first naive port suffered from the overhead of continuous transitions to different execution environments (see Figure 2-(a)). Each kernel invocation starts in the native environment, then goes to the virtual machine using JNI where Java code is used to retrieve the data from the native environment and allocate it for the RenderScript environment. Once the data is available to RenderScript, the kernel executes (in parallel) and afterwards the Java code retrieves the resulting data from the RenderScript environment and passes it to the native environment. Then the control code selects the next operation and the same process happens over again, until the control code is finished and the final HDR picture is computed.

To reduce the overhead related to the transition from native to Java and back with JNI, we moved the control code to Java such that we only have a single transition to Java at the start of the control function. Then the entire control code is executed in Java and only when the result is computed the program goes back to the native environment (see Figure 2-(b)). By using references to the memory allocations, this implementation did not have to copy intermediate data back and forth to RenderScript, except from the initial input images and the final output image.

We noticed that the transition from Java to RenderScript also added overhead to the algorithm and therefore we started to merge as many kernels together as possible (see Figure 2-(c)). As an example, instead of executing an *add* kernel followed by a *multiply* kernel one can create a single *add-multiply* kernel. This optimization enabled the reduction of the number of kernel invocations drastically, albeit that the kernels themselves are larger.

Finally, we further reduced the transition overhead from Java to RenderScript by moving the control script to RenderScript (see Figure 2-(d)). While the Java code still does all the memory allocations, it will transfer the control to RenderScript which starts computing until the final image is computed without any context switches.

The comparison of the resulting optimized RenderScript implementation versus the reference NDK implementation is discussed in Section 5.

3.2 RCUDA

Next to an implementation with RenderScript that exploits data parallelism on the device itself, we also implemented exposure fusion with computation offloading using Remote CUDA. RCUDA does classic computation offloading by offering a proxy on the client side that forwards calls to a server. RCUDA operates at the CUDA abstraction layer and therefore executes computation on the GPGPU of the host in parallel.

On the mobile device there is a modified version of the CUDA shared library (libcuda.so) that can communicate over TCP with a remote cudaserver. On top of libcuda.so the regular CUDA runtime (libcudart.so) and CUDA libraries can be run (cufft, NPP, etc.).

Porting the exposure fusion algorithm from the reference NDK implementation to a RCUDA implementation is trivial. All the OpenCV kernels that are used in the NDK already have a CUDA based implementation, only the package names differ between the regular and the GPGPU implementation – cv::<kernel> for the regular and cv::gpu::<kernel> for the GPGPU implementation. Furthermore, the data type of the matrices the kernels operate on have to be changed from cv::Mat to cv::gpu::GpuMat and these matrices have to be initialized somewhat differently using a function called upload, because data now resides in GPU memory. Data can be retrieved from a GpuMat using the download function.

The exposure fusion algorithm is the first algorithm of substantial size that has been used with RCUDA and therefore we expected to identify performance bottlenecks in RCUDA. Because the computation offloading in RCUDA is at the CUDA call abstraction level, a reasonably sized algorithm, such as our exposure fusion algorithm, can easily include

thousands of calls. For each call a synchronous request is sent to the server that, depending on the call, immediately returns a response and executes the request asynchronously, or first executes the request and thereafter returns a response. Either way the client blocks until the response arrives.

Because of the sheer number of calls, even a low network latency of 1 ms would add communication overhead of multiple seconds to the process. Therefore we changed as many requests as possible from synchronous to asynchronous, we cached the results of some calls that were called repeatedly on the client side, thereby reducing the number of CUDA calls over the network and the overhead of the calls. Furthermore we noticed that for the remaining synchronous calls, Nagle's algorithm [Nag84] in TCP – buffering small messages into a large message for a certain time – caused much overhead, as described in [DIM+10]. Turning this off with TCP_NODELAY increased performance dramatically.

In addition to the above latency based optimizations to RCUDA we also introduced compression to the client-server protocol, to reduce the amount of data that needs to be transferred at the cost of a additional computation. We use the zlib compression library, which supports 10 different compression levels, ranging from easy compression at a low cost to very complex compression at a high cost.

In the experiments section (Section 5) we assess the impact of latency, bandwidth and compression on the execution time and energy usage of the RCUDA implementation of the exposure fusion algorithm on a Tegra 3 device.

4 Methodology

4.1 Targets and Variables

The key targets of our experiments are the execution time and the energy usage of a particular implementation under particular circumstances. Next to these main targets, we also collect data about the CPU load, the CPU frequency and the temperature, to be able to investigate unexpected results. With this information we can for instance see if using multiple cores indeed leads to all processors being active, or if temperature causes the frequency to be scaled down thereby lengthening execution time.

Next to a specific implementation there are several other variables that will impact the execution time and energy usage of the exposure fusion algorithm. The more pixels an image has, the longer the execution takes; the higher the latency or the lower the bandwidth, the longer computation offloading takes. The more complex the compression, the more computation is required, but also the less data has to be sent.

For varying the latency we artificially increase the latency on the server side using *netem* [H+05]. With *trickle* [Eri05] we manipulated the bandwidth for both the up and downlink of the server. We used the *hotplug* feature of linux to measure the impact of the number of active cores for the multicore implementation.

For each combination of settings we repeated the experiment 30 times. Whereas the results

in general are very consistent, we inspect the data for explainable outliers. We use the additional information such as temperature, CPU usage and CPU frequency to determine whether we discard the outlier for the final results. In all experiments we have at least 26 valid data points.

For the RCUDA experiments we remove the result of the first execution, because it includes a one time overhead of initializing the libraries. Furthermore, for the RCUDA experiments we use ethernet over USB to connect the mobile device to the network, to prevent interference artifacts from the wireless network and make the experiments repeatable.

The Tegra 3 Developer Tablet supports hardware monitoring through various monitoring applications in combination with a PM292 breakout board. The main advantage of hardware monitoring is that it does not interfere with a running application, it does not take cycles from the CPU nor consumes energy from the battery. The main disadvantage of using hardware monitoring is that we cannot easily correlate the data we gather with the execution of a particular part of an application, because the gathered data will be timestamped with the time on the external machine, not the tablet. To overcome this issue we can use software monitoring, such as Power Tutor [ZTQ+10]. In our experiments we use software monitoring where we read out the current power consumption values from the power rail that hosts the quad core.

5 Experiments

5.1 Multi Core

In our first experiment we compare the execution times of the NDK implementation and the RenderScript implementation, while varying the image size and the number of active cores. The RenderScript execution times include both the (serial) host code and (parallel) device code. Since the computation scales with the number of pixels we expect a linear relation between the image size and the execution time. Furthermore, we expect RenderScript to perform up to 4 times better than the NDK implementation, because it can make use of all the available cores. We also expect that the RenderScript runtime adds some overhead, due to the host code that is serial.

Figure 3 shows the results of both the reference and the RenderScript implementation while varying the image size. We found that indeed the execution time has a linear relation with the image size for both implementations. Furthermore, RenderScript does not achieve a 4x speedup, indicating that the usage of the RenderScript runtime introduces overhead. To get a better idea about the runtime overhead, we performed a second experiment where we varied the number of active cores for the RenderScript implementation with *hotplug*.

The results of this experiment are shown in Figure 4, where we normalized the RenderScript execution times with respect to the reference execution time to calculate the speedup. From this figure we can see that the image size does not impact the scalability of RenderScript significantly. The overhead of using RenderScript on a single core is 25.9%

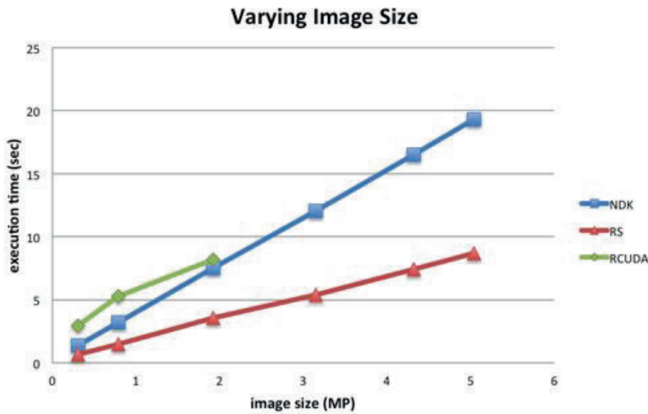

Figure 3: Execution times for RCUDA, RenderScript (RS) and the reference implementation (NDK) while varying the image sizes. The RCUDA execution times are measured without any additional latency and bandwidth constraints.

on average and increases to 45.7% on four cores. Although increasing the number of cores leads to an increase in overhead, the speedup factor increases too – up to 2.2x on four cores. This means that we have not yet reached an assymptot in the speedup graph, and although the current hardware prevents us from running the algorithm on more than four cores, adding more cores can possibly lead to even lower execution times and thus higher speedups.

We also perform the same experiment without explicitly turning on or turning off cores with hotplug, but rather letting the default governor activate cores when needed, such as would happen in real world scenarios. We find that it takes a constant time period for the governor to turn on all four cores (about 0.5 seconds). With small problem sizes (such as VGA resolution), the activation time for the other cores wastes the possible speedup severely, whereas the the activation time is hardly noticeable for an image size of 5 MP. A possible solution to improve the execution time with small problem sizes in real world scenarios is that the governor could offer an interface to applications such that they can explicitly instruct the governor to turn on multiple cores if some compute intensive job is about to start.

Now that we have seen that the RenderScript implementation improves the execution time by using multiple cores, we analyze what the impact of using multiple cores on the energy usage is. On the one hand we expect RenderScript to consume less energy, because of the shorter execution time, on the other hand we expect RenderScript to consume more energy because it uses multiple cores. If we only consider the energy usage of the quad core, we find that RenderScript's shorter execution time with a higher power draw results in energy usage equal to the NDK's longer execution time with lower power draw (see Figure 5). This is surprising given the fact that some of the energy of RenderScript is spent on overhead and one would therefore expect that RenderScript would consume more energy.

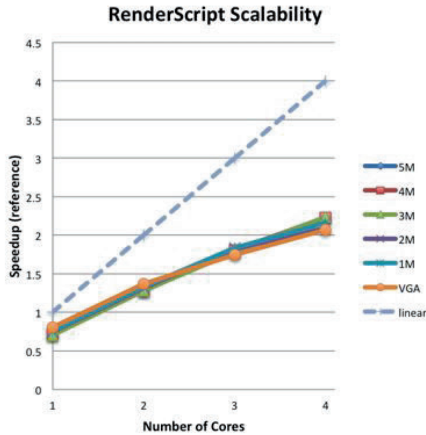

Figure 4: Although the RenderScript implementation does not reach linear speedup, adding cores improves the speedup compared to the native reference implementation.

Further analysis of the measurements reveals that the power draw of the quad core CPU does scale linear with the number of active cores, but also includes a fixed draw, and can be roughly approximated by the following formula: $P_{quadcore} = 500mW + n * 500mW$

Using more cores therefore results in better power efficiency. In the case of the RenderScript implementation however, what is won in efficiency due to the use of multiple cores is wasted on the RenderScript runtime overhead, resulting in an equal energy usage to the reference implementation.

From the experiments with RenderScript we conclude that the exposure fusion algorithm benefits from a multicore implementation, leading to a maximum speed up of 2.2x on four cores, while using an equal amount of energy on the CPU. Although the energy usage for the CPU is equal, RenderScript will likely lead to device wide energy savings, because the shorter the algorithm has to run, the shorter other components, such as the screen have to be turned on. Furthermore, for smaller size jobs the use of RenderScript will not automatically lead to good speed ups, because it takes some time for the CPU governor to switch from single core to quad cores.

5.2 Computation Offloading

In this section we turn our attention to the experiments we performed with the computation offloading implementation based on Remote CUDA. Our primary focus is how latency, bandwidth and compression level impact both execution time and energy usage.

In our first experiment we optimized the circumstances to get the lowest possible execution times for computation offloading. This means that we did not put limits on the bandwidth and latency and used compression level 1, as we show in later experiments this turns out

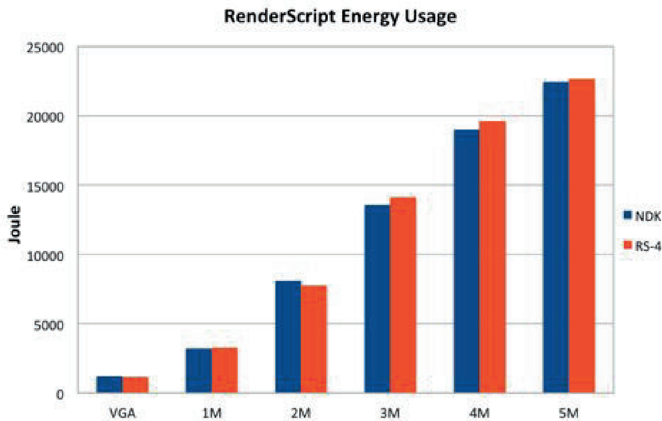

Figure 5: Energy used by RenderScript on 4 cores compared to the reference implementation.

to be the optimal compression level value. Figure 3 shows the results of this experiment. Because of GPU memory limits on our offloading laptop, which has a 512 MB Fermi based GPU, we could not run the algorithm for images with a size larger than 2 MP. Although the RCUDA implementation for all our measurements is slower than the reference implementation, we see the difference between the two implementations decreasing when the image size increases. Future experiments with a host with larger GPU memory have to prove whether this is a trend and the RCUDA implementation is faster than the NDK implementation with sufficient large images.

Because the remote GPGPU computation is much faster than the computation of the reference implementation, much of the total execution time is determined by the communication. The time spent in communicating depends on the available bandwidth and the latency. We performed a second experiment with RCUDA in which we vary both the bandwidth and the latency. The results of this experiment are shown in Figure 6. We observe that increases in the latency and decreases in the bandwidth to real world values lead to dramatic increases in execution time, well above what is acceptable for an algorithm like exposure fusion (for instance a latency well above 50 ms is common in 3G networks [HXT+10]). Therefore we can conclude that for this particular algorithm the RCUDA computation offloading framework is not a competitive alternative to on device computation. This is partially due to the abstraction level of the RCUDA framework – a low abstraction level leads to many messages, sensitive to latency – and partially due to the fact that the algorithm blows up the data, making the algorithm sensitive to bandwidth. For instance, single 1MP images, which as compressed JPEG images are typically below 200 kB, get converted to 9 MB float arrays in the algorithm. Other computation offloading frameworks that operate at a higher abstraction level (such as [CBC+10, KPKB10] that operate on the method level), could reduce the number of messages to a single response/reply and the data to JPEG compressed images.

In order to limit the impact of bandwidth on the execution time we added compression to

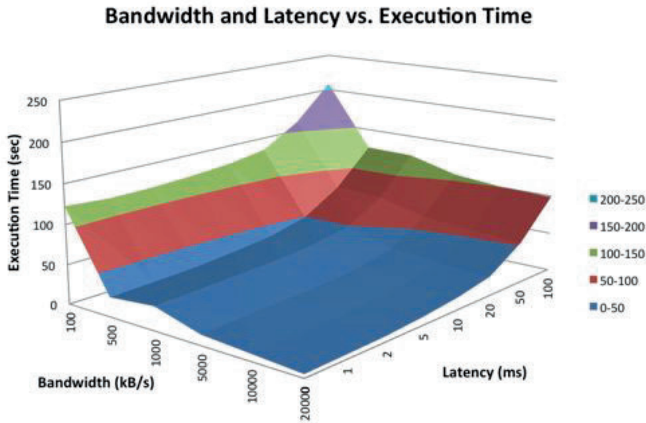

Figure 6: The impact of both bandwidth and latency on the execution time of the RCUDA implementation on a 1MP image.

RCUDA by using the zlib [ZLi] library. The zlib library supports compression levels from 0 to 9, where a higher compression level makes use of better compression techniques, at the cost of more computation. Thus with compression we can trade communication for computation. We performed an experiment where we varied the compression level and bandwidth. We expect that when there is plenty of bandwidth only simple compression will contribute to a lower execution time, whereas at low bandwidth it may be worthwhile to spend additional time on compressing to reduce the data that is sent.

Figure 7 shows the results of this experiment. We find that indeed increasing the compression level in the cases where we have a bandwidth of more than 100 kB/s only slows down the algorithm, whereas with the lowest bandwidth setting we see that an increase in compression level – beyond level 2 – leads to slightly lower execution times. However, compression level 1 is even at low bandwidth an equal choice to compression level 9, indicating that even at the lowest bandwidth that we used in our experiment, putting more effort in compressing data does not improve execution time. If we shift our focus from the execution time to the energy usage, we can see clearly that an increase in compression level, and thus an increase in computation leads to an increase in energy usage of the CPU (see Figure 8). This gives additional reason to only use simple compression. Whereas we expected that computation offloading would reduce the energy consumed by the CPU, we see that without compression RCUDA uses only 5% less energy on the CPU than our reference implementation and with compression it always uses more energy. Whereas these figures only compare energy used on the CPU rail, we should not forget that offloading computation introduces additional energy usage for communication. Since we use Ethernet over USB in our experiments, we did not include the energy usage for communication, because in real world settings a wireless variant will be used for connectivity. However, we can safely conclude that RCUDA computation offloading for the exposure fusion al-

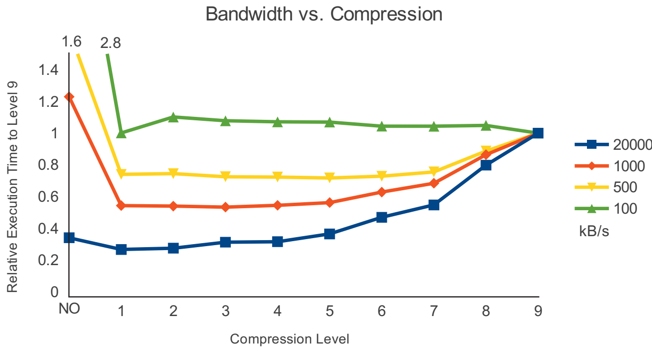

Figure 7: The relative execution time for a specific bandwidth and compression level compared to the execution time with maximum compression on a 1MP image.

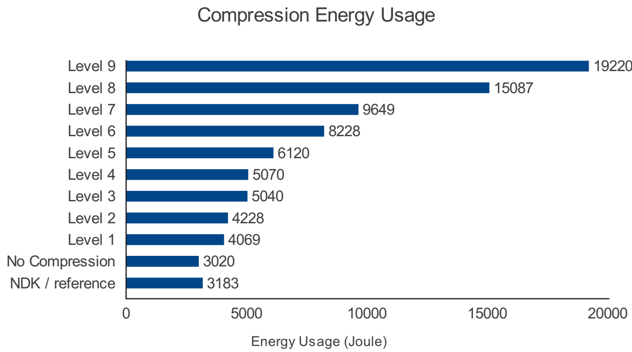

Figure 8: Energy usage at different compression levels for RCUDA with a 1MP image.

gorithm on a Tegra-3 device does not lead to better execution times nor to lower energy usage if the additional communication cost is taken into account.

6 Conclusions

In this paper we discussed and evaluated two alternative programming models to a native implementation for compute intensive tasks on mobile devices using a case study with the exposure fusion algorithm used for HDR photography. We found that using RenderScript, a multicore programming model, we can improve execution times up to 2.2 times while keeping energy usage on the CPU similar and reducing energy usage on the system as a whole. The other programming model we examined, Remote CUDA for computation offloading, did not lead to speed ups nor to energy savings, but the case study taught us several lessons that we applied in optimizing the Remote CUDA environment, such

as selecting the right TCP settings as well as improving on caching and asynchronous execution.

We also found that for short run compute intensive tasks the power of a multicore processor is not optimally used, because of the time it takes to switch from single core to quad core and therefore we recommend additional API calls for developers, such that they intentionally can turn on multiple cores just before a compute intensive task starts, instead of waiting on the processor governor to do so.

References

[CBC+10] E. Cuervo, A. Balasubramanian, D. Cho, A. Wolman, S. Saroiu, R. Chandra, and P. Bahl. MAUI: making smartphones last longer with code offload. In *Proceedings of the 8th international conference on Mobile systems, applications, and services*, pages 49–62. ACM, 2010.

[CUD] NVIDIA, C Programming Best Practices Guide, CUDA Toolkit 4.2.

[DIM+10] J. Duato, F. Igual, R. Mayo, A. Peña, E. Quintana-Ortí, and F. Silla. An efficient implementation of GPU virtualization in high performance clusters. In *Euro-Par 2009–Parallel Processing Workshops*, pages 385–394. Springer, 2010.

[Eri05] M.A. Eriksen. Trickle: A userland bandwidth shaper for unix-like systems. In *Proc. of the USENIX 2005 Annual Technical Conference, FREENIX Track*, 2005.

[H+05] S. Hemminger et al. Network emulation with NetEm. In *Linux Conf Au*, pages 18–23, 2005.

[HXT+10] J. Huang, Q. Xu, B. Tiwana, Z.M. Mao, M. Zhang, and P. Bahl. Anatomizing application performance differences on smartphones. In *Proceedings of the 8th international conference on Mobile systems, applications, and services*, pages 165–178. ACM, 2010.

[KPKB10] Roelof Kemp, Nicholas Palmer, Thilo Kielmann, and Henri Bal. Cuckoo: a Computation Offloading Framework for Smartphones. In *MobiCASE '10: Proc. of The 2nd International Conference on Mobile Computing, Applications, and Services*, 2010.

[LWX01] Zhiyuan Li, Cheng Wang, and Rong Xu. Computation offloading to save energy on handheld devices: a partition scheme. In *Proceedings of the 2001 international conference on Compilers, architecture, and synthesis for embedded systems*, CASES '01, pages 238–246, New York, NY, USA, 2001. ACM.

[MKVR07] T. Mertens, J. Kautz, and F. Van Reeth. Exposure fusion. In *Computer Graphics and Applications, 2007. PG'07. 15th Pacific Conference on*, pages 382–390. IEEE, 2007.

[Nag84] J. Nagle. Congestion control in IP/TCP internetworks. *ACM SIGCOMM Computer Communication Review*, 14(4):11–17, 1984.

[Ope] OpenCV. http://opencv.willowgarage.com/wiki/.

[QZL12] Xi Qian, Guangyu Zhu, and Xiao-Feng Li. Comparison and Analysis of the Three Programming Models in Google Android. In *First Asia-Pacific Programming Languages and Compilers Workshop (APPLC)*, 2012.

[Ren] Google RenderScript. http://developer.android.com/guide/topics/renderscript.

[ZLi] ZLib. http://zlib.net/.

[ZTQ+10] 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.