ParallelME: A Parallel Mobile Engine to Explore Heterogeneity in Mobile Computing Architectures

Guilherme Andrade[1(✉)], Wilson de Carvalho[1], Renato Utsch[1], Pedro Caldeira[1], Alberto Alburquerque[1], Fabricio Ferracioli[4], Leonardo Rocha[2], Michael Frank[3], Dorgival Guedes[1], and Renato Ferreira[1]

[1] Department of Computer Science, Federal University of Minas Gerais, Belo Horizonte, Brazil
gnandrade@dcc.ufmg.br
[2] Department of Computer Science, Federal University of São João del Rei, São João del Rei, Brazil
[3] LG Electronics, San Jose Lab, Santa Clara, USA
[4] LG Electronics, São Paulo, Brazil

Abstract. Following the evolution of desktops, mobile architectures are currently witnessing growth in processing power and complexity with the addition of different processing units like multi-core CPUs and GPUs. To facilitate programming and coordinating resource usage in these heterogeneous architectures, we present *ParallelME*, a *Parallel M*obile *E*ngine designed to explore heterogeneity in mobile computing architectures. *ParallelME* provides a high-level library with a friendly programming language abstraction for developers, facilitating the programming of operations that can be translated into low-level parallel tasks. Additionally, these tasks are coordinated by a runtime framework, which is responsible for scheduling and controlling the execution on the low-level platform. ParallelME's purpose is to explore parallelism with the benefit of not changing the programming model, through a simple programming language abstraction that is similar to sequential programming. We performed a comparative analysis of execution time, memory and power consumption between *ParallelME*, OpenCL and RenderScript using an image processing application. ParallelME greatly increases application performance with reasonable memory and energy consumption.

1 Introduction

Mobile phones are no longer devices for basic communication between people, but have become much more sophisticated devices. Current models include a range of sensors that are capable of collecting a wide variety of data about the user and the environment, and many applications are being proposed which involve processing this large data volume either on the device or in the cloud. These application's compute demands have put pressure on the hardware architecture to significantly increase the processing power while also maintaining the power consumption at

P.-F. Dutot and D. Trystram (Eds.): Euro-Par 2016, LNCS 9833, pp. 447–459, 2016.
DOI: 10.1007/978-3-319-43659-3_33

a reasonable level. A recent trend in the desktop scenario is the utilization of different types of processing units (PUs), in the so-called heterogeneous systems - computers which include multi-core CPUs as well as other special purpose processors - GPUs being a favorite among them. Nowadays, many cell phone SoCs are also equipped with multi-core CPUs, such as Qualcomm Snapdragon and NVIDIA Tegra 3, and high-performance GPUs, such as Mali from ARM, and Adreno from Qualcomm.

In this new context, it becomes necessary for applications of different domains to achieve better performance and hence have better user experience by exploring, in a coordinated and efficient way, all the available PUs taking full advantage of their processing capabilities. In this direction it is important to create high-level programming abstractions which allow programmers to define the operations in some simple, intuitive way, while its parallel execution on different PUs can be achieved at run-time with minimal programmer interference. To accomplish that, we propose a specialized run-time framework, providing mechanisms for manipulating the generated low-level tasks. The framework is responsible for creating specific tasks to be executed in specific processing units, for managing the dependencies and for scheduling them across the devices in an efficient way. The proper use of different processing units in a heterogeneous system is a well studied problem in the literature when it comes to conventional computer architectures, for which different run-time environments [8,12] exist. However, it is a new and challenging scenario for mobile architectures.

The challenges in dealing with different processing units on a mobile architecture involve mainly the limited resources available for coding and the absence of facilitating run-time environments. The vast majority of mobile developers use programming languages, frameworks and APIs, supported by the device operating system, with a high level abstraction. In order to access and use different computational resources on a device architecture (i.e. GPU), programmers must be able to work at a lower level of abstraction, which requires advanced knowledge. This major effort to link low-level programming with high-level abstraction leads to a high cost of application programming.

In this work we present *ParallelME*, a *Parallel M*obile *E*ngine designed for exploring heterogeneity in mobile architectures. *ParallelME* provides a high-level library with a friendly programming language abstraction for developers, enabling programs to easily describe complex operations and translate them into low-level tasks. These tasks are manipulated in a coordinated manner by our proposed run-time framework, which is responsible for scheduling and controlling the execution on the low-level platform. Therefore, in our work, we explore two challenging points in the context of mobile architectures: (1) Expressing high-level parallel operations; and (2) Exploring different processing units in the low-level system. Despite being originally proposed for the mobile scenario, *ParallelME* is a generic parallel Java extension, coupled with a OpenCL run-time framework, which can easily be used for developing to desktop systems as well.

ParallelME's purpose is to explore parallelism with the benefit of not changing the programming model, through a simple programming language abstraction that is similar to sequential programming.

We performed a comparative analysis of execution time, memory and power consumption between *ParallelME*, OpenCL and RenderScript using an image processing application. Our experiments have shown that ParallelME greatly increases application performance up to 32.34 times, with reasonable energy consumption and reducing memory usage significantly, while maintaining a simple programming interface.

The remainder of this paper is organized as follows. In Sect. 2 we present a description of the main parallel programming frameworks currently available for mobile architectures. In Sect. 3 we present all components of ParallelME, on which we highlight the language used and the low level platforms explored. In sequence, in the Sects. 3.1 and 3.3 we explain the features of the proposed high-level library and the details of the low-level execution mechanism, respectively. We evaluate the comprehensiveness and applicability of our abstract language and the run-time performance, presenting the results in Sect. 4. Finally, we conclude and discuss some future work in the last section.

2 Related Works

In this section we present a detailed description of the main parallel programming frameworks available, to the best of our knowledge, for developing efficient application for mobile architectures. The frameworks addressed here are Pyjama [9], Aparapi [11], Rootbeer [14], Paralldroid [7] and RenderScript [6].

Pyjama's [9] focus is on bringing OpenMP's programming model to Java, with optimizations for Graphical User Interface (GUI) applications. Pyjama was built with this requirement in mind: it changes the way threads are generated by its own source to source compiler implementation. In OpenMP, the main thread that triggers parallel computation becomes the master one for the processing, whereas in Pyjama a new master thread is created for that, since the GUI one has to be kept running. It does not exactly present a new parallel programming model, but its changes in benefit of GUI applications are really interesting for developing mobile apps and its source to source compiler model is an effective way of introducing its own changes to the Java language, as it keeps things simple to the user while retaining control over how the parallelization is implemented. The amount of information the user provides and the way it is done is key to the effectiveness of the programming model adopted by the framework. On the other hand, Pyjama does little to address the complexity of OpenMP programming in general: a developer still has to learn his ways through it and is fully responsible for the good execution of his code.

Aparapi [11] is an AMD project, which focuses on making GPU programming easier. Aparapi was presented in 2011 as an API to express data parallel workloads in Java. It avoids changing the basic structure of the language itself, so instead of writing a parallel-for, for example, the developer extends a kernel

base class to write his function. The code is compiled to Java bytecode and then parallelized by converting to OpenCL at runtime, reverting back to Java Thread Pool if necessary. It is possible to iterate over objects, not only primitive types. Comparing to programming with OpenCL or CUDA directly, Aparapi offers several advantages, though it does not tackle important problems. It is much easier to create and maintain new code, but debugging is still difficult and searching for execution bottlenecks remains as hard as on other environments. Compared with our own goals, Aparapi is limited in its parallelization scope, regarding both the hardware it optimizes code for and the level of complexity of its functions.

The idea behind Rootbeer [14] is similar to Aparapi's: transform regular Java code into GPU parallel code with minimum effort. Rootbeer is more robust than Aparapi, for it supports a broader set of instructions in Java language, excluding mostly dynamic method invocation, reflection and native methods. Rootbeer was created to make development on GPUs easier, but it never guaranteed performance improvement by treating the Java code written by the programmer. This means that even though it makes it easier to program, the task of code optimization still befalls heavily on the programmer, so much knowledge and time is still required to create efficient code and the programmer has to deal on his own with the performance issues his program's structure creates after the code conversion.

Paralldroid [7] is a framework to ease parallel programming on Android devices, aiming to make source to source translation of OpenMP-like annotated code. The user writes his Java code directing the framework on what should be done in parallel on each occasion, and the framework writes the low level code. Though it has limitations on what it converts, as they only support primitive type variables, conversions are made to RenderScript, OpenCL and Native C code. Akin to Pyjama, Paralldroid also uses OpenMP-like directives as basis for its programming model. These are complex to learn and require knowledge many programmers might not have.

RenderScript [6] is a framework designed by Google to perform data-parallel computation in Android devices, and was formally introduced in Android SDK API Level 11 (aka Android 3.0 Honeycomb). RenderScript is a means of writing performance critical code that can run natively on different processors, selected from those available at runtime. This could be the device CPU, a DSP, or even the GPU. Where it ultimately runs on is a question that depends on many factors that are not readily available to the developer.

3 ParallelME

ParallelME was conceived as a complete infrastructure for parallel programming for mobile architectures, exploring different processing units in a coordinated manner. Our purpose is focused on mobile computing, so we implemented ParallelME through available and widely used platforms for mobile systems. It is composed of three main components: (1) a programming abstraction, (2) a source-to-source compiler and (3) a run-time framework.

The programming abstraction was inspired by ideas found in the Scala collections library and is designed to provide an easy-to-use and generic programming model for parallel applications in Java for Android. The language was extended by means of specialized classes for which we also provided a sequential implementation. This means that the applications can run as they are written using the regular development infrastructure, though they will be very inefficient.

Our source-to-source compiler, however, provides a mechanism for replacing the use of the sequential libraries by extracting the portions of the code that comprise the parallel tasks and generating them either as RenderScript or as OpenCL [5]. The compiler also incorporates the appropriate bindings of the application code with the parallel tasks, thus creating a parallel application and improving the overall performance.

The run-time framework was developed using OpenCL and is responsible for setting up the application to allow several parallel tasks to be specified and queued for execution on different devices. It allows different criteria in deciding the PU in which each task should run at run-time, creating a novel level of control and flexibility which can be explored in order to achieve certain goals as far as resource utilization, which can further improve overall performance.

The first and current version of ParallelME is focused on the mobile operating system (Mobile OS) Android. In addition to being the most used Mobile OS in the world [4], Android is also an open source Linux-based system. In the following subsections we further describe each of the components of our proposed framework.

3.1 Programming Abstraction

General purpose mobile applications are commonly implemented in Java for Android OS through the use of the SDK (Software Development Kit [2]). Even though it is possible to use native Android resources with lower level programming platforms and language through NDK (Native Development Kit [1]), the use of Android SDK is preponderant among programmers. In this sense, ParallelME programming abstraction was designed to be supported by Android SDK in Java. In order to provide a generic and effective programming model for parallel applications, we considered the highest level of abstraction possible regarding the limitations of the current Java version supported by Android.

ParallelME programming abstraction was inspired in ideas found on the Scala collection library [15], which is a library consisted of a set of different data structures with native support for parallel processing. The goal of ParallelME was to create a similar data-structure oriented library that could be used to produce parallel code with the minimum effort by its user. In ParallelME, this set of data structures is called *User-Library*.

The User-Library is composed of a series of classes as part of a collections' package. These collections represent common data structures like arrays, lists, hash sets and hash maps capable of handling an abstract data type. The abstract data type corresponds to the user class which is used to store user data. All user data is stored on the respective collection through one of its data insertion methods.

So far, the User-Library contains three functional classes: Array, BitmapImage and HDRImage. The former class is a generic single-dimensional array, while the last two classes offer support for bitmap and HDR image processing.

Though Scala is a language which differs deeply from Java, ParallelME provides a similar approach for operations that iterate through an entire collection with no restriction of processing order. Given a bitmap object and a *foreach* operation to iterate through all its pixels and perform a simple modification in its color (i.e. add 1 to each RGBA color space parameter), the proposed abstraction for ParallelME is presented in Listing 1.

```
BitmapImage image = new BitmapImage(bitmap);
image.par().foreach(new UserFunction<Pixel>(){
    @Override
    public void function(Pixel pxl) {
        pxl.rgba.red = pxl.rgba.red + 1;
        pxl.rgba.green = pxl.rgba.green + 1;
        pxl.rgba.blue = pxl.rgba.blue + 1;
        pxl.rgba.alpha = pxl.rgba.alpha + 1;
    }
});
bitmap = image.toBitmap();
```

```
BitmapImage image = new BitmapImage(bitmap);
image.par().foreach(pxl -> {
    pxl.rgba.red = pxl.rgba.red + 1;
    pxl.rgba.green = pxl.rgba.green + 1;
    pxl.rgba.blue = pxl.rgba.blue + 1;
    pxl.rgba.alpha = pxl.rgba.alpha + 1;
});
bitmap = image.toBitmap();
```

Listing 1: Programming abstraction for ParallelME

Code in Listing 1 shows two variations of the same operation where left and right side are semantically identical. The first and last lines respectively are the data input and output operations of the User-Library class. They correspond to the class constructor where the user provides the input data (in this case a bitmap object) and the data retrieval method, on which the user gets back the data processed on ParallelME (in this case, the data replaces the original bitmap). The code between data input and output corresponds to the user code with the color modification proposed. It contains a call to method *par* on object *image*, which indicates that a parallel operation will be performed on the next method call. It is followed by a *foreach* iterator containing the user code and where operations have different syntax on each side. Left side code represents a version compatible with Java version supported by Android (Java 7) [2] during the development of the first version of ParallelME, with no support for lambda expressions [10]. Right side code was created using a lambda expression, which is available in Java 8. It demonstrates the simplicity and objectivity of our syntax when applied together with a lambda expression and shows how ParallelME will be used as a more recent version of Java is incorporated in Android.

3.2 Source-to-Source Compiler

The ParallelME source-to-source compiler has been developed with the incorporation of ANTLR, a powerful parser generator [13] widely used to build languages, tools and frameworks. We used an ANTLR Java grammar to create a parser that was used to build and traverse a parse tree, which in turn was the basis for ParallelME compiler. Our compiler takes as input Java code written

with the User-Library and translates it into C code compatible with RenderScript or OpenCL frameworks, depending on the chosen runtime. Besides that, the compiler also integrates the translated code with the Java application, performing modifications to the original user code to support the proposed run-time environment.

In this sense, our compiler is composed of 3 macro steps:

- **User-Library detection:** detects where the user code was written with User-Library classes. These locations will point to the code that must be translated to the target runtime.
- **Memory binding:** detects where the user is providing the data that will be processed in order to transmit it to the runtime environment.
- **Conversion to RenderScript/OpenCL:** the user code in the iterators' body of the User-Library classes must be translated to a RenderScript or OpenCL compliant version in order to produce the behaviour specified on the high level programming abstraction.

In favor of creating a first version of ParallelME on a reasonable schedule, we defined some rules and limitations on the input and output code. These rules and limitations are related to the user code provided to the iterator and the user class that is parametrized on User-Library collections. For both user code in iterators and parametrized classes we defined that only Java primitive types, their equivalent wrapper classes and User-Library collections can be used. These restrictions reduced considerably the compiler complexity, yet allowing a high degree of flexibility on user code creation.

Once our User-Library was designed in Java, the user code written with our programming abstraction is 100 % compliant with Java 7 language specification. This allowed us to rely on the Java compiler (javac) for syntactic and semantic evaluation of code, reducing responsibilities of ParallelME compiler and increasing its robustness.

3.3 Run-Time Framework Details

The run-time component of ParallelME is responsible for coordinating, in an efficient way, all processing units available on the mobile architecture. It organizes and manages low level tasks generated by the Compiler component, from definitions expressed by developers in User-Library. The implementation is in C++ and was inserted in the Android system using the NDK toolkit. As previously explained, OpenCL was adopted as a low-level parallel platform to manipulate available processing units. In general, the dynamics of the runtime involves the following phases: (1) Identifying the computing resources available on mobile architecture; (2) Creating tasks and their input and output parameters; (3) Arranging task's data according to their parameters; (4) Submitting them for execution; (5) Instantiating routines of a scheduling policy; (6) Assigning tasks to processing units defined by the scheduling policy. The first four phases correspond to user API phases and must be performed to instantiate

the system and the tasks. The run-time internal engine is composed of the two remaining phases. Figure 1 gives an overall picture of the entire framework, which is further described below.

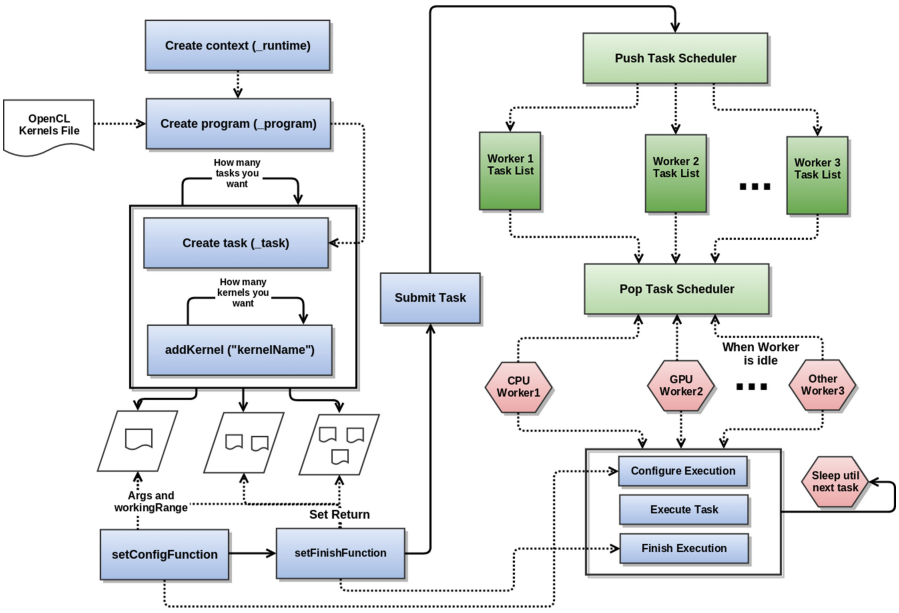


Fig. 1. Execution mechanism dynamics

Run-Time API Phases. The first run-time phase is divided into two steps: (1) detection of the available resources in a specific mobile architecture; and (2) instantiation of all necessary structures related to the framework. In the first step, the framework identifies the available processing units, also called devices. A context is created for each device, which corresponds to specific implementations of OpenCL routines, provided by different suppliers such as NVIDIA, Intel, AMD and Qualcomm. In the second step, the framework instantiates a system thread for each of these contexts. These threads, called *Worker Threads*, are responsible for managing devices using specific OpenCL routines. These configurations are performed by instantiating the run-time constructor.

The second phase corresponds to the creation of tasks that are executed by processing units. In this phase, a source file containing one or more OpenCL kernels is built, and each kernel present in the compiled file composes one or more tasks. When more than one kernel is assigned to a task, they are executed in the order they were instantiated. It is important to emphasize that these tasks are generic and not linked - at this moment - to any device. When submitted to execution, these tasks will be scheduled and executed on a specific device.

The third phase is responsible for preparing the data that will be manipulated by each task. As OpenCL buffers are device-specific, it would be very expensive to set up tasks' data before the scheduler decides where the task will run on. In order to deal with this, we propose a mechanism in which the task configuration is done through callbacks: before sending the task to the scheduler, the user specifies configuration callbacks (that can be lambda functions) that set up task data. Only after the scheduler decides where the task will run, the configuration function is called with the target device specified through a parameter. This avoids the cost of copying task data to multiple devices. Also, in this step, it is possible to configure for each kernel the amount of threads or work units involved (OpenCL work range) in its execution.

Finally, the last phase related to Environment Setup corresponds to submitting tasks to execution. The run-time routine responsible for performing it must be called for each created task.

Execution Engine. After task submission, the run-time engine is responsible for scheduling and executing tasks across the devices. Once a task is submitted, the scheduler routine *Push Task* allocates it to a specific device task list, following a particular scheduling policy. *Worker threads* remain on a sleeping state if there is no task to be executed. However, when a task is submitted, a signal awakens *worker threads* which in turn call the routine *Pop Task*. This routine is responsible for retrieving a task from the task list, following a particular scheduling policy. When a *worker thread* receives the task returned by *Pop task* routine it starts the execution process.

For task execution, first the run-time framework executes the *Configure Execution* callback, responsible for allocating buffers of kernel parameters, assigned in the task, on the specific device. After this allocation, the kernel is also assigned to the device memory in order to start its execution. A *worker thread* waits for the execution to finish and then calls the *Finish Execution* callback, which is responsible for retrieving the output buffers. The *worker thread* then goes back to sleeping state if there is no more tasks in its execution lists.

4 Evaluation

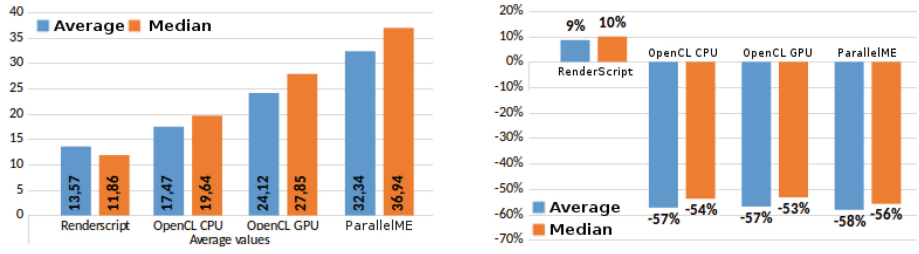
The evaluation of ParallelME was performed on three different metrics: execution time, memory consumption and power consumption. For this purpose, we implemented a HDR tone mapping application using Eric Reinhard's operator [16] with ParallelME. Tone mapping is a technique used in image processing and computer graphics to map one set of colors to another to approximate the appearance of high dynamic range images in a medium that has a more limited dynamic range. Additionally, we implemented and evaluated the same application in different platforms in order to compare and discuss ParallelME performance: (1) Single Threaded Java; (2) OpenCL using CPU; (3) OpenCL using GPU; (4) RenderScript.

In each of the mentioned platforms, along with ParallelME, we ran the application two times. In the first run we measured execution time and power consumption, while in the second we measured memory consumption. We used 5 different HDR images (Clifton Bridge, Clock Building, Crowfoot, Lake Tahoe and Tintern Abbey) [3] and all executions ran each image 1, 2, 5, 10 and 15 times. For ParallelME environment, each tone mapping execution correspond to a task. In this way, in our experiment 1, 2, 5, 10 and 15 tasks were executed concurrently in the ParallelME environment. For these experiments ParallelME used just a simple First Come First Serve scheduler for task distribution.

We also performed each execution from scratch, meaning that after finishing the previous execution, we killed the application process and restarted it anew. We used a LG D855 (LG G3) device set up with Android 5.0, 100 % of screen brightness, flight mode activated and all others applications closed.

4.1 Results and Discussions

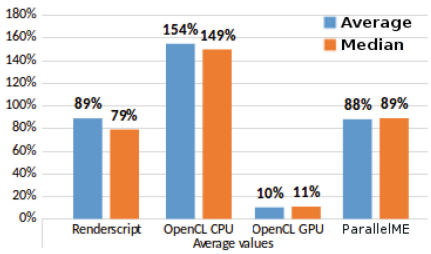
To measure execution time we used Java routines to take the time delay between application start and finish and for power consumption evaluation we used a Monsoon Power Monitor configured to 3.9 V. Power consumption was measured in the following way: (1) average consumption of Android OS with 180 s idle (no application running) and the screen on; (2) average consumption during the algorithm execution. Then we subtracted values (1) from (2) to get the application power consumption, that tells us, in milliwatts (mW), the average power consumption required during execution. Additionally we evaluated and discussed the total energy consumption, which was found by multiplying the average power by the application execution time. Taking the single threaded Java implementation as the basis for comparison, we evaluated execution time, memory consumption, power consumption and total energy consumption for the other implementations. Figures 2 and 3 show the mean and median for all executions.



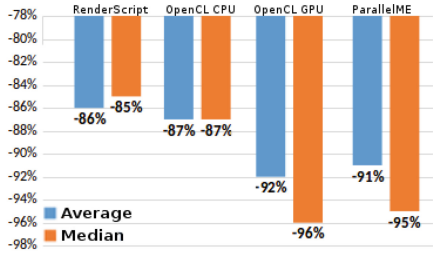
(a) Performance gain. (b) Memory consumption.

Fig. 2. Performance and memory consumption evaluation.

In Fig. 2(a) it is possible to observe that ParallelME presents the best result, increasing application performance, in average, by 32.34 times compared to the single thread Java implementation. The presence of a run-time framework in ParallelME allows tasks to be executed concurrently. Thus all processing units



(a) Power consumption.



(b) Total Energy consumption.

Fig. 3. Power and energy consumption evaluation.

in the mobile architecture are being explored in coordination, reflecting a significant performance increase. In addition, analyzing the Fig. 3(a), we note that ParallelME required 88 % more power to achieve the reported performance. This consumption is acceptable given that different resources in mobile architecture are being used in parallel. However, despite ParallelME requiring more power, the total energy consumption, shown in Fig. 3(b), is 94 % lower than the energy consumed by the single threaded Java implementation, since the application execution time using ParallelME is considerably lower than the Java one. This shows that reducing application execution time with ParallelME also causes a significant reduction on the energy consumed by the application. Also, by comparing the results to RenderScript, which required the same amount of power to increase application performance by just 13 times, ParallelME becomes even more satisfactory when it comes to the total energy consumption.

It is necessary to emphasize the results of the OpenCL GPU implementation. In this case, the total energy consumption is only slightly smaller than the ParallelME energy consumption, although power consumption was just 10 % higher than the single thread Java. This also happened because ParallelME runtime is much faster. Also, in this situation, despite the significant increase in performance and low power consumption of OpenCL, GPU resources are fully occupied during application execution. Because of this, the phone screen may freeze, as the screen stops being updated, compromising user experience.

To measure memory consumption we plugged the phone to a computer using a USB cable and adb connection, allowing it to run in debug mode and, in this way, to track memory. Following this strategy, Fig. 2(b) presents in graphical format the average values of the proposed scenario.

RenderScript application consumes more memory to execute; 9 % on average. On the other hand, the improvements achieved by OpenCL version are clear, reducing memory consumption by 57 % on average. ParallelME is based on OpenCL and provides, besides a significant performance improvement and reduction in power consumption, a significant reduction in memory usage compared to RenderScript and single thread Java implementations.

5 Conclusions

In this work we presented *ParallelME*, a framework designed to explore heterogeneity in mobile computing architectures. The proposed engine provides a high-level library with a friendly programming language abstraction for users and a run-time mechanism which is responsible for scheduling and controlling the execution on the low-level platform. We described all three *ParallelME* components - programming abstraction, source-to-source compiler and run-time environment - showing it was designed to deal with parallel operations from high-level implementation to execution on heterogeneous architectures.

We measured execution time, memory and power consumption of *ParallelME* comparing to other implementations of an image processing application. We show that *ParallelME* increases application performance by 32.34 times, which is the best result, with reasonable energy consumption and significantly reducing memory usage.

This work can be extended in different ways. The User-Library can be extended with several different collections and operations to increase ParallelME's usability. The system run-time API makes it possible to insert different scheduling strategies and others schedulers may explore different aspects of the underlying environment, such as reducing energy consumption. It is also possible, at the level of the proposed compiler, to insert features to extract code information that can be used by the scheduler to improve kernel execution assignment in the available processing units.

References

1. Android NDK. http://developer.android.com/tools/sdk/ndk
2. Android SDK. http://developer.android.com/sdk
3. HDR images collection. https://www.cs.utah.edu/~reinhard/cdrom/results.html
4. IDC analyses. http://www.idc.com/prodserv/smartphone-os-market-share.jsp
5. OpenCL by khronos group. https://www.khronos.org/opencl/
6. RenderScript. https://developer.android.com/guide/topics/renderscript
7. Acosta, A., Almeida, F.: Performance analysis of paralldroid generated programs. In: 2014 22nd Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP). IEEE (2014)
8. Augonnet, C., Thibault, S., Namyst, R., Wacrenier, P.: StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures (2011)
9. Giacaman, N., Sinnen, O. et al.: Pyjama: OpenMP-like implementation for Java, with GUI extensions. In: Proceedings of the 2013 International Workshop on Programming Models and Applications for Multicores and Manycores. ACM (2013)
10. Gosling, J., Joy, B., Steele Jr., G.L., Bracha, G., Buckley, A.: The Java Language Specification, Java SE 7 Edition. 1st edn. (2013)
11. Gupta, K.G., Agrawal, N., Maity, S.K.: Performance analysis between Aparapi (a parallel api) and Java by implementing sobel edge detection algorithm. In: Parallel Computing Technologies (PARCOMPTECH). IEEE (2013)
12. Kunzman, D.: Charm++ on the cell processor (2006)
13. Parr, T.: The Definitive ANTLR 4 Reference. Pragmatic Bookshelf, 2nd edn. (2013)

14. Pratt-Szeliga, P.C., Fawcett, J.W., Welch, R.D.: Rootbeer: seamlessly using GPUs from java. In: 2012 IEEE 14th International Conference on High Performance Computing and Communication & 2012 IEEE 9th International Conference on Embedded Software and Systems (HPCC-ICESS). IEEE (2012)
15. Prokopec, A., Bagwell, P., Rompf, T., Odersky, M.: A generic parallel collection framework. In: Proceedings of the 17th International Conference on Parallel Processing - Volume Part II, EUROPAR 11 (2011)
16. Reinhard, E., Stark, M., Shirley, P., Ferwerda, J.: Photographic tone reproduction for digital images. ACM Trans. Graph. (TOG) **21**, 267–276 (2002). ACM