

UNIVERSIDADE FEDERAL DE MINAS GERAIS
Instituto de Ciências Exatas
Programa de Pós-Graduação em Ciência da Computação

Wilson de Carvalho Moreira Júnior

Parallel Programming Models for Mobile Devices

Belo Horizonte
2017

Wilson de Carvalho Moreira Júnior

Parallel Programming Models for Mobile Devices

Final Version

Thesis presented to the Graduate Program in Computer Science of the Federal University of Minas Gerais in partial fulfillment of the requirements for the degree of Master in Computer Science.

Advisor: Renato Antônio Celso Ferreira
Co-Advisor: Dorgival Olavo Guedes Neto

Belo Horizonte
2017

Ficha catalográfica elaborada pela Biblioteca do ICEx - UFMG

Moreira Júnior, Wilson de Carvalho.

M838p Parallel programming models for mobile devices. /
Wilson de Carvalho Moreira Júnior. – Belo Horizonte,
2017.
xxv, 75 f.: il.; 29 cm.

Dissertação (mestrado) - Universidade Federal de
Minas Gerais – Departamento de Ciência da Computação.

Orientador: Renato Antônio Celso Ferreira.
Coorientador: Dorgival Olavo Guedes Neto

1. Computação – Teses. 2. Programação paralela
(Computação). 3. Dispositivos móveis. 4. Dispositivos
heterogêneos. I. Orientador. II. Coorientador. III. Título.

CDU 519.6*31(043)



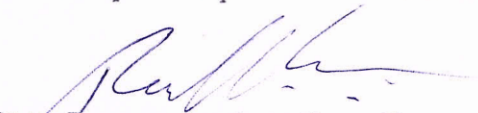
UNIVERSIDADE FEDERAL DE MINAS GERAIS
INSTITUTO DE CIÊNCIAS EXATAS
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

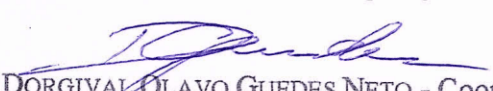
FOLHA DE APROVAÇÃO

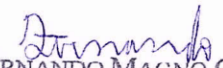
Parallel programming models for mobile devices

WILSON DE CARVALHO MOREIRA JÚNIOR


Dissertação defendida e aprovada pela banca examinadora constituída pelos Senhores:


PROF. RENATO ANTÔNIO CELSO FERREIRA - Orientador
Departamento de Ciência da Computação - UFMG


PROF. DORGIVAL OLAVO GUEDES NETO - Coorientador
Departamento de Ciência da Computação - UFMG


PROF. FERNANDO MAGNO QUINTÃO PEREIRA
Departamento de Ciência da Computação - UFMG


PROF. GUIDO COSTA SOUZA DE ARAÚJO
Instituto de Computação - UNICAMP


PROF. LEONARDO CHAVES DUTRA DA ROCHA
Departamento de Ciência da Computação - UFSJ

Belo Horizonte, 23 de fevereiro de 2017.

I dedicate this work to my wife Raphaella for being my true love and for carrying our baby Nicolas who will be born in a few months. Hope our paths continue in the same direction forever.

Acknowledgments

I need to recognize all those people that helped me (consciously or not) during my path in the pursue of my MSc:

my wife Raphaella, for her support, patience and love that seems to grow since 2008, despite this boring nerd that she married;

my parents, brother and sisters, who always showed through examples and kind words that the continuous education is the most dignified path to be taken;

my former friends of self-education Euler Teixeira, Josemar Figueiredo and Marcos Abi-Ackel, for their friendship and willingness to discuss about computer science and electrical engineering;

my advisor Prof. Renato Ferreira and co-advisor Prof. Dorgival Guedes, for their patience, guidance and ideas that led me to accomplish this work;

my cats Nutella e Viçosa, for their company, meows and purrs during study nights. They certainly made this time happier;

my partners in the ParallelME project: Alberto Cavalcante, Dorgival Guedes, Fabrício Ferracioli, Guilherme Andrade, Michael Frank, Pedro Caldeira, Renato Ferreira and Renato Utsch;

those that shared some time and a pleasant conversation with me during my lunch time at UFMG: Jeronimo Barros, Julio Jeha, Marcelo Henrique, Marcos Carvalho, Prof. Fernando Quintão, Sara Carvalho and Waner Miranda;

my friend Maurício, for constantly inviting me and my wife to attend his wonderful dinners and lunches to talk about Life, the Universe and Everything;

my friends back at the video-game days in the 90s: I would never become a computer science geek without you;

my friend Leandro Collares, who donated his precious time to patiently revise this work in days he could perfectly enjoy a good movie.

“See first, think later, then test. But always see first. Otherwise, you will only see what you were expecting. Most scientists forget that.”
(Douglas Adams)

Resumo

A densidade de componentes eletrônicos em um único chip tem crescido por meio século. Mantendo esta tendência por longos anos, a indústria de microprocessadores tem continuamente lançado produtos mais poderosos, permitindo o desenvolvimento de aplicações mais complexas e que exigem maior capacidade computacional. Especialmente nos últimos dez anos, a direção tomada fabricantes para atender à crescente demanda por recursos computacionais das aplicações recentes e manter uma densidade de potência reduzida é aumentar o número de unidades de processamento (PUs) num mesmo empacotamento físico. Tais processadores são conhecidos hoje como arquiteturas multi-núcleo. Adicionalmente ao aumento no número de núcleos, arquiteturas *desktop* e servidor tem adotado diferentes tipos de PUs nas chamadas arquiteturas heterogêneas: computadores que incluem CPUs multi-núcleo e também outros processadores de propósito especial – sendo GPUs um favorito dentre eles.

A criação de modelos de programação de alto nível para facilitar o desenvolvimento de aplicações e do uso coordenado de PUs em arquiteturas heterogêneas são tópicos largamente discutidos em sistemas *desktop* e servidor. Entretanto, no emergente cenário de arquiteturas de dispositivos móveis, poucas avaliações e discussões foram feitas até o momento. Desta forma, este trabalho tem como objetivo analisar o atual cenário de programação paralela em plataformas móveis heterogêneas — focando no sistema operacional Android — e apresentar contribuições para reduzir a complexidade de desenvolver aplicações paralelas para dispositivos móveis heterogêneos.

Desta forma, o presente trabalho analisa *frameworks* de baixo nível para programação paralela em Android, apresentando um compilador de código fonte para código fonte capaz de traduzir código criado na abstração proposta para representações em *frameworks* de mais baixo nível. Esta abordagem trouxe ganhos de desempenho e consequente redução de consumo de energia das aplicações.

Palavras-chave: Programação Paralela, Dispositivos Móveis, Dispositivos Heterogêneos

Abstract

The density of electronic components on a single chip has shown steady increase for half a century. Keeping up with this tendency for many years, the microprocessors industry has continuously released more powerful products, allowing the design of more complex and demanding applications. Specially in the last ten years, the direction taken by manufacturers to meet the increasing demand of computing resources of modern applications and to keep a low power density is to increment the number of processing units (PUs) in single physical packages. These processors are currently known as multi-core architectures. In addition to the increment in number of cores, desktop and server architectures have also adopted different types of PUs in the so-called heterogeneous architectures: computers which include multi-core CPUs as well as other special purpose processors — GPUs being a favorite among them.

The creation of high-level programming models to facilitate the development of parallel applications and the coordinated usage of processing units in heterogeneous architectures are broadly discussed topics in desktop and server systems. However, in the emerging scenario of mobile architectures, there have been few evaluations and discussions so far. For this reason, the main goal of this work is to analyze the overall scenario of parallel programming in mobile heterogeneous platforms – focusing on Android OS – and present contributions to reduce the complexity of developing parallel applications for mobile heterogeneous devices.

Thus, this work analyses low-level frameworks for parallel programming in Android, presenting a source-to-source compiler to translate the code created in the proposed programming abstraction to representations in lower-level frameworks. This approach provides performance gains and consequently reduces applications' energy consumption.

Keywords: Parallel Programming, Mobile Devices, Heterogeneous Devices

List of Figures

2.1	Paralldroid structure	24
2.2	Paralldroid experimental results	25
2.3	Android software stack	28
3.1	ParallelME framework overview	31
3.2	Application algorithm overview.	33
4.1	Number of lines of code evaluation	51
5.1	Compiler overview	54
5.2	Parse tree for previously presented code	55
5.3	RenderScript integration architecture	63
5.4	ParallelME Run-time integration architecture	64
5.5	Translated code integration architecture	65
5.6	Symbols for graph representation	66
5.7	Sequential and parallel graphs for Foreach and Map operations	67
5.8	Sequential Reduce	68
5.9	Parallel Reduce	69
5.10	Parallel Reduce for Array class	69
5.11	Sequential Filter	70
5.12	Parallel Filter	70
5.13	Execution time comparison	72
5.14	Power consumption comparison	74
A.1	ParallelME Run-time Execution Flow	83

List of Tables

5.1	Total energy required compared to sequential Java baseline	74
-----	--	----

Contents

1	Introduction	13
1.1	Goal	14
1.2	Scope	15
2	Background	18
2.1	Programming for GPUs and Heterogeneous Architectures	18
2.1.1	Aparapi	18
2.1.2	OpenCL	19
2.1.3	Rootbeer	20
2.2	Parallel and Heterogeneous Programming in Mobile Devices	21
2.2.1	RenderScript	22
2.2.2	Pyjama	22
2.2.3	Paralldroid	24
2.3	Functional Programming	26
2.4	Android Mobile Operating System	27
2.5	Summary	29
3	Parallel Mobile Engine	30
3.1	Overview	30
3.2	User Library	31
3.3	Internal library	31
3.4	Run-time System	32
3.5	Source-to-source Compiler	32
3.6	User Library and Compiler Evaluation	33
3.7	Summary	34
4	User Library	35
4.1	Programming Abstraction	35
4.2	Data binding	38
4.3	Operations	39
4.3.1	Foreach	39
4.3.2	Map	40
4.3.3	Reduce	41
4.3.4	Filter	42

4.4	Debugging Support	43
4.5	Evaluation	44
4.5.1	Comparison of ParallelME User Library and Low-level Frameworks	45
4.5.1.1	ParallelME User Library Implementation	45
4.5.1.2	ParallelME Run-time Implementation	47
4.5.1.3	Implementation for RenderScript	49
4.5.2	Sources Lines of Code	50
4.6	Summary	52
5	Compiler	53
5.1	Overview	53
5.2	First Pass	54
5.3	Second Pass	56
5.4	Translation Steps	58
5.4.1	Analysis of Different Execution Forms	58
5.4.2	Common interface for target run-time systems	59
5.4.3	Run-time system specific translation	60
5.4.4	Original user code translation	61
5.5	Translated Code Integration Architecture	62
5.5.1	RenderScript	62
5.5.2	ParallelME Run-time	63
5.5.3	Integration of user-application and target run-time systems	65
5.6	Translated Code Execution Strategy	66
5.6.1	Foreach and Map	67
5.6.2	Reduce	67
5.6.3	Filter	69
5.7	Limitations	71
5.8	Evaluation	72
5.8.1	Execution Time	72
5.8.2	Power and Energy Consumption	73
5.9	Summary	75
6	Conclusion and Future Works	76
6.1	Known Limitations and Future Works	77
	Bibliography	78
	Appendix A Run-time	82
A.1	Run-time API Phases	82
A.2	Execution Engine	84

Chapter 1

Introduction

Following the cost reduction and improvements in fabrication techniques of electronic components throughout the years, the density of such components on a single silicon chip has shown steady increase for half a century. This trend, which was considered exponential when first observed by Moore [1965], is well known among computer scientists as Moore's Law. Keeping up with this tendency for long years, microprocessors' manufacturers have competed for market share by continuously increasing the number of components on a single chip, releasing more powerful products year after year.

To sustain the increment of microelectronic components necessary to design more capable hardware and keep a low power density, manufacturers have released processors with multiple processing units (PUs) on a single chip [Parkhurst et al., 2006]. Such sophisticated devices, presently known as multi-core architectures, have almost completely replaced single-core architectures in desktop, laptop and server platforms.

The same path taken by desktop, laptop and server platforms towards multiple PUs is being followed by mobile devices. Mobile devices no longer lend themselves to basic communication between people only. They have become far more sophisticated devices. Current models include a range of sensors that are capable of collecting a wide variety of user and environment data. Many applications are being proposed which involve processing this large data volume either on the device or in the cloud. These applications' computing demands have put pressure on the hardware architecture to significantly increase the processing power, while also maintaining the power consumption at a reasonable level. A recent trend in the desktop scenario is the utilization of different types of 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. This trend has also been followed by the industry of mobile devices and many current cell phones are equipped with multi-core CPUs and GPUs.

In this new context, programmers have been pushed to use parallel programming from server to mobile applications in order to produce software that achieves higher performance levels and provides a better user experience by exploring all the available PUs, taking full advantage of modern devices' processing capabilities. The use of parallel programming techniques for software development is normally recognized as a complex task

that can implicate in the introduction of errors during its execution. Such complexity refers to what McCool et al. [2012] calls the **serial illusion**: a mental model of the computer as a machine that executes operations sequentially. According to these authors, this model was introduced decades ago and continues to be used despite the successive improvements in processors to make them more parallel internally. Moreover, programmers have become overly dependent on the serial illusion.

Different languages and frameworks have been proposed in order to provide some degree of abstraction and reduce the complexity of using parallel programming techniques. Recent languages such as R, Python, Clojure and Scala have popularized the usage of functional programming techniques and immutable data structures. Such structures eliminate the possibility of side-effects and write concurrency, paving the way for creating programs that can be executed in multiple PUs with less effort. In this sense, application development has been reshaped from popular imperative programming to functional programming style. Conversely, high-performance frameworks such as OpenCL [Khronos, 2016] and RenderScript [Google, 2016d] – the latter only supported in the mobile operating system (Mobile OS) Android – have offered APIs for using heterogeneous architectures, providing programming interfaces that allow the development of applications that can run in CPUs and GPUs.

1.1 Goal

The study and understanding of programming abstractions and coordinated usage of processing units in heterogeneous architectures are broadly discussed topics with regard to desktop systems. However, in the new scenario of mobile architectures, few evaluations and analysis have been carried out so far. Thus, the main goal of this work is to analyze the overall scenario of parallel programming in mobile heterogeneous platforms - focusing on Android OS - and present an alternative programming abstraction to reduce the complexity of creating applications that take advantage of multiple PUs existing in current mobile devices. In order to do that, this work analyses low-level frameworks for parallel programming in Android, presenting a source-to-source compiler to translate the code created in the proposed programming abstraction to low-level frameworks. This approach provides performance gains and consequently reduces applications' energy consumption.

The programming abstraction, the source-to-source compiler and the run-time environment presented in this work are parts of ParallelME, a Parallel Mobile Engine framework conceived during a 15-month project in a joint effort between *Universidade Federal de Minas Gerais* and *LG Electronics*. It provides an easy-to-use programming abstraction

in Java for creating high-performance applications in Android, allowing the user code to run transparently in lower-level frameworks like RenderScript and OpenCL.

ParallelME [Andrade et al., 2016] was designed to explore heterogeneity in Android devices, automatically coordinating the usage of computing resources while maintaining the programming effort similar to what sequential programmers expect. ParallelME distinguishes itself from other frameworks by its high-level library that contains a friendly collection-based programming abstraction in Java with a debugging feature in the same language, and the ability to efficiently coordinate the usage of resources in heterogeneous mobile architectures with task schedulers. Being built as a modular framework, ParallelME is divided in three well-defined parts: (i) a programming abstraction (User Library), (ii) a source-to-source compiler and (iii) a run-time system.

ParallelME User Library is composed of different data-structures for handling collections of numerical and image data. These collections provide an intuitive programming abstraction that supports the introduction of user-defined operations in the Java high-level programming model. Such programming model is recognized by the source-to-source compiler that translates the user code written in the proposed programming abstraction to low-level representations in RenderScript and the OpenCL-based ParallelME Run-time. These representations are transparently integrated into the user application and produce parallel high-performance applications with low effort.

1.2 Scope

This work provides two contributions to ParallelME: the programming abstraction presented in the User Library and the source-to-source compiler. These components are responsible for establish a bridge between users and the low-level run-times targeted. Designed to reduce the complexity of developing high-performance and low-energy consumption applications, both components were developed to be used in Android mobile devices.

According to the International Data Corporation [2016], Android is the prevailing operating system with 87.6% of presence in mobile devices reported on the second quarter of 2016. The mobile OS leadership position and its open-source nature [Google, 2016a] are evident incentives for producing programming solutions for such platform. Android programming tools are divided in two very different development kits: the System Development Kit (SDK) and the Native Development Kit (NDK). At the SDK level, the programmer has access to the high-level and ubiquitous Java language, with an extensive set of libraries and tools. Conversely, at the NDK level, APIs provide access to native lan-

guages such as C and C++. In this sense, it is possible to find high and low-level parallel programming frameworks in Android, presenting SDK and NDK interfaces, respectively.

Several frameworks [Giacaman et al., 2013, Acosta and Almeida, 2014, Pratt-Szeliga et al., 2012, Gupta et al., 2013, Khronos, 2016, Google, 2016d] are worth mentioning when evaluating tools for developing high-performance applications in Android. These important initiatives present positive and negative aspects that influenced the development of this work. The aforementioned frameworks have, in their own way, positive features regarding the reduced complexity of creating parallel code. In one case [Google, 2016d], the framework provides a highly integrated model to the Android environment. Conversely, as a general rule, all remaining frameworks fail to deliver powerful debugging mechanisms, restricting code debugging to console messages. In this sense, low programming complexity, extensibility, high-performance of applications, easy integration with Android environment and sophisticated debugging features were key aspects considered during the development of ParallelME User Library and ParallelME Compiler.

Among those frameworks, it is important to highlight RenderScript [Google, 2016d] and OpenCL [Khronos, 2016]. Code produced with both frameworks share good results of performance and low energy consumption [Kemp et al., 2013, Wang et al., 2013, Andrade et al., 2016]. Such low-level frameworks provide tools for creating generic multi-threaded functions that can be executed in both GPUs and CPUs. Although they share the same goal, they provide different features and programming interfaces, with limitations being evident in the complex programming abstraction and the restricted coordination of resources [Andrade et al., 2016], preventing their popularization among mobile developers. For this reason, these frameworks were used as low-level platforms for ParallelME. Thus, code created in the proposed programming abstraction with the User Library is translated by ParallelME Compiler to representations in RenderScript and the OpenCL-based ParallelME Run-time. These representations are then transparently integrated to the user application with little programming effort.

The programming abstraction proposed, encapsulated in ParallelME User Library, has deep roots in the Scala Collection Framework [Prokopec et al., 2011], consisting of an extensible set of data structures with intrinsic support for parallelism. In the current version, this library supports four types of operations: iteration, data-transformation, sub-setting and reduction. Currently, these operations can be executed independently in different types of numerical arrays and images, but can also be used to create more complex solutions with sequences of operations.

The following five chapters are organized in the following order:

- An introduction to the current scenario of heterogeneous programming, parallel programming for mobile devices and functional programming;
- An overview of ParallelME framework;

- The detailed description of the proposed programming abstraction (ParallelME User Library) and its evaluation;
- The detailed description of the proposed source-to-source compiler (ParallelME Compiler) and its evaluation;
- Conclusion and future works.

Chapter 2

Background

This chapter presents the background necessary to understand the context and contributions of this work. Initially, an overview of tools designed to reduce the complexity of developing applications for heterogeneous and parallel architectures is described, providing an analysis of main frameworks and tools proposed in this regard for mobile, server and desktop platforms. The overview is followed by an evaluation of functional programming and the reasons that makes it a good fit for developing highly scalable parallel applications. Finally, it presents Android, the target operating system that was used to develop and test all the contributions presented in this work.

2.1 Programming for GPUs and Heterogeneous Architectures

Several frameworks and tools have been developed in order to reduce the complexity of programming for GPUs and heterogeneous architectures. From more elaborate code transformations [Gupta et al., 2013, Pratt-Szeliga et al., 2012] to an API that provides compatibility with a wide range of devices [Khronos, 2016], frameworks presented in the following sections introduce higher-level abstractions designed to reduce the complexity of programming for these architectures.

2.1.1 Aparapi

Aparapi [Gupta et al., 2013, AMD, 2016] (A Parallel API) is an AMD project designed to allow Java developers to take advantage of GPUs by executing data parallel fragments in these processing units. The framework was presented in 2011, introducing

an extensible kernel structure to allow users to insert code that is lately translated to OpenCL and executed in parallel in the GPU.

```
1 final int square = new int[size];
2 final int in = new int[size];
3
4 for (int i=0; i<size; i++) {
5     square[i] = in[i] * in[i];
6 }
```

Listing 1: Original Java code. Extracted from Frost [2014]

```
1 final int square = new int[size];
2 final int in = new int[size];
3
4 new Kernel() {
5     @Override
6     public void run() {
7         int i = getGlobalID();
8         square[i] = in[i] * in[i];
9     }
10 }.execute(size);
```

Listing 2: Aparapi’s equivalent code. Extracted from Frost [2014]

The code presented in Listing 1 is an example of a sequential iteration in Java that was developed using the regular *for* construct offered by the language. It calculates the square of each element of an input array, storing the result in another array with the same size, being this size determined by an external parameter. This code was rewritten in Aparapi’s programming model in Listing 2, demonstrating the creation of an equivalent operation that will be transformed into parallel workloads by the framework.

Structured in basically two elements, an API for expressing data parallel workloads and a run-time component, Aparapi presents a simple and effective approach for programming in heterogeneous architectures. The API offers to users a Java class (*Kernel*, shown in Listing 2) that can be implemented with the code that will be used to create parallel workloads in the GPU by overriding the *run* method. After being translated to Java Bytecode, the run-time component translates those compatible workloads to OpenCL in order to enable its execution in GPUs.

2.1.2 OpenCL

OpenCL (Open Computing Language) [Khronos, 2016] is an open standard for parallel programming of heterogeneous systems that is currently supported by numerous processors and hardware vendors [AMD, 2016, Intel, 2016, Apple, 2016, NVIDIA, 2016b].

It was designed to support both data and task-based parallelism, consisting of an API that provides abstractions for coordinating parallel computations across heterogeneous processors.

With a wide range of supported devices (personal computers, servers, mobile and embedded devices), the OpenCL standard contains a core specification that any OpenCL-compliant device must implement and a set of extensions that are platform-specific. Being a two-layer framework, it is divided in two main components: OpenCL Runtime and OpenCL Compiler. With a subset of ISO C99 as programming language, developers can write functions (also called *kernels*) that can be executed in statically-chosen processing units.

At kernel level, functions are developed with *global* memory allocations that can have fragments shared by simultaneous calls at runtime, thus allowing parallel reads and writes. A matrix multiplication code is shown in Listing 3 which presents a function that can execute in parallel performing both reads and writes with no need for concurrency control mechanisms.

```
1 // Multiplies A*x, leaving the result in y.
2 // A is a row-major matrix, meaning the (i,j) element is at A[i*ncols+j].
3 __kernel void matvec(__global const float *A, __global const float *x,
4                     uint ncols, __global float *y) {
5     size_t i = get_global_id(0); // Global id, used as the row index.
6     __global float const *a = &A[i*ncols]; // Pointer to the i'th row.
7     float sum = 0.f; // Accumulator for dot product.
8     for (size_t j = 0; j < ncols; j++) {
9         sum += a[j] * x[j];
10    }
11    y[i] = sum;
12 }
```

Listing 3: Matrix multiplication in OpenCL

2.1.3 Rootbeer

Similarly to Aparapi, Rootbeer [Pratt-Szeliga et al., 2012] is a framework designed to perform code transformation from Java to GPU code with focus on CUDA platform [NVIDIA, 2016a]. Being described as a production quality software, Rootbeer is presented as a ready-to-use tool that can be used by researchers needing to increase Java code performance with the GPU support.

Rootbeer supports all features of the Java programming language, with the exception of dynamic method invocation, reflection and native methods. With a programming interface close to that proposed by Aparapi, Rootbeer requires that developers implement a *Kernel* interface and a single method (*gpuMethod*) in order to produce Java code

```
1 public class ArraySum implements Kernel {
2     private int[] source;
3     private int[] ret;
4     private int index;
5     public ArraySum(int[] src, int[] dst, int i) {
6         source = src; ret = dst; index = i;
7     }
8     public void gpuMethod() {
9         int sum = 0;
10        for (int i = 0; i < array.length; ++i) {
11            sum += array[i];
12        }
13        ret[index] = sum;
14    }
15 }

1 public class ArraySumApp {
2     public int[] void sumArrays(List<int[]> arrays) {
3         List<Kernel> jobs = new ArrayList<Kernel>();
4         int[] ret = new int[arrays.size()];
5         for (int i = 0; i < arrays.size(); ++i) {
6             jobs.add(new ArraySum(arrays.get(i), ret, i));
7         }
8         Rootbeer rootbeer = new Rootbeer();
9         rootbeer.runAll(jobs);
10        return ret;
11    }
12 }
```

Listing 4: Rootbeer’s syntax for GPU programming. Extracted from Pratt-Szeliga et al. [2012]

capable of being executed in GPUs, as shown in Listing 4.

In order to perform code transformation from Java to CUDA, Rootbeer takes an approach similar to Aparapi’s, transforming Java Bytecode to the target GPU platform. In this sense, Rootbeer takes a program previously compiled by the Java Compiler and converts it into a new version integrated to CUDA platform. In order to do that, the framework cross-compiles kernels written in Java to CUDA and integrates them to the remaining parts of the Java application through JNI calls.

2.2 Parallel and Heterogeneous Programming in Mobile Devices

So far have been few evaluations and little discussions about the creation of high-level programming models for heterogeneous mobile architectures. However, it is possible to find several high-level frameworks that have been designed to facilitate the development of parallel applications with different processing units in these architectures. Since this work focuses on Android OS, the frameworks presented in the following sections are

specific for this platform.

2.2.1 RenderScript

RenderScript [Google, 2016d] is a framework designed by Google to perform data-parallel computations in Android devices, transparently running user code in heterogeneous architectures. As RenderScript is an integral part of Android, it is present in all recent versions of the operating system.

The framework is a way to write performance-critical code that can run effortlessly on different processors selected from those available at run-time, like the device's CPU, DSP or even GPU. However, where this code ultimately runs is a question that depends on the framework's scheduler, which is not completely controlled by the developer.

Following the same principle of OpenCL, RenderScript also requires the implementation of user kernels in C99 and explicit memory handling. Even though RenderScript bears similarities with the mobile version of OpenCL, it provides a much easier programming interface that can be handled directly in Java using the Android SDK, reducing considerably the complexity of integrating the kernel with the user application in Java.

Although RenderScript's programming complexity is reduced due to its high level of integration with Android, the programmer is still required to handle complex tasks to use the API. The execution context must be created, data allocations must be specified and memory binding must be performed in order to be able to run user kernels. Therefore, there is a rudimentary, yet necessary memory handling that must be performed by programmers.

2.2.2 Pyjama

OpenMP is an API that supports multithreaded programming in C, C++ and Fortran. The main goal is to enable a program to be executed in parallel without losing its sequential logic: the same program should be able to run sequentially without code changes. This is accomplished by the addition of precompiler directives to the sequential program.

Pyjama's [Giacaman et al., 2013] focus is to bring OpenMP's programming model to Java, with changes regarding the optimization of applications with Graphical User In-

```

1  int main(int argc, char **argv) {
2      int a[100000];
3      #pragma omp parallel for
4      int i;
5      for (i = 0; i < N; i++)
6          a[i] = 2 * i;
7      return 0;
8  }

```

Listing 5: Standard OpenMP code, image extracted from Giacaman et al. [2013]

terface (GUI). In traditional scientific or data processing applications, after data input the program controls the execution flow until an output is generated. However, in GUI apps, a thread is responsible for handling user's input (such as interactions with a smartphone's screen) and the program's flow follows according to what the user wants to do. Usually, in graphical applications, any processing happens in the background while the GUI thread runs constantly in parallel, so responsiveness can be maintained.

In order to deal with these requirements, Pyjama changes the way threads are generated with its own source-to-source compiler implementation. In OpenMP the main thread that triggers parallel computation becomes the master thread for processing, whereas in Pyjama a new master thread is created for that, since the GUI thread has to be kept running.

```

1  public void actionPerformed() {
2      //omp parallel freequithread
3      {
4          processImage(file);
5          //omp gui
6          updateProgressBar();
7      }
8      showImage(file);
9  }

```

Listing 6: Original Java code with Pyjama's annotations, extracted from Giacaman et al. [2013]

```

1  public void actionPerformed() {
2      _ompEnqueue("_omp_workRegion_0", "_omp_cont_pt_0");
3      private void _omp_workRegion_0() {
4          processImage(file);
5          if (false == EventQueue.isDispatchThread()) {
6              SwingUtilities.invokeLater(new Runnable() {
7                  public void run() {
8                      updateProgressBar();
9                  }
10             });
11         }
12     }
13 }
14 private void _omp_cont_pt_0() {
15     showImage(file);
16 }

```

Listing 7: Transformed code, extracted from Giacaman et al. [2013]

While it does not exactly present a new parallel programming model, Pyjama’s focus on GUI-related aspects is useful for mobile application development. Its source-to-source compiler model is an effective way of introducing changes to the Java language, keeping a simple API while retaining control of how the parallelization is implemented.

2.2.3 Paralldroid

Paralldroid [Acosta and Almeida, 2014] is a framework created to reduce the complexity of parallel programming in Android devices. It provides an OpenMP-like directive-based programming abstraction developed to be non-invasive regarding parallel programming concepts. Thus, the user code is filled with Paralldroid directives and then submitted to a source-to-source translator that creates native C code, OpenCL compatible code or RenderScript code accordingly to user definitions.

The translation model defined for Paralldroid involves source-to-source transformation and skeletal programming. It is divided in three modules: front-end, middle-end and back-end, as shown in Figure 2.1. The front-end module performs the validation of user code according to Paralldroid proposed language syntax and semantics. After that, the front-end calls the middle-end followed by the back-end module. The middle-end is responsible for checking the user-defined directives and the Java code associated with them, creating an intermediate representation that is used by the back-end module. The back-end module is then responsible for generating the output code in Native C, OpenCL or RenderScript, performing the necessary modifications in user code in Java to allow it to access one of the output code platforms.

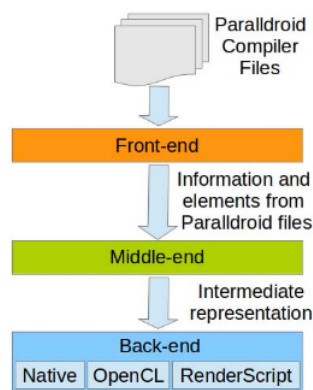


Figure 2.1: Paralldroid structure. Image extracted from Acosta and Almeida [2014]

The proposed language for Paralldroid contains directives that are an extension of OpenMP 4.0, being used in regular Java implementations to identify those code sections

that must be parallelized. In this sense, it uses regular sequential high-level constructs in Java and introduces Paralldroid directives to inform the necessary transformations that must be performed by the translation model, as shown in the example code in Listing 8.

```

1  public void grayscale() {
2      int pixel, sum, x;
3      int[] scrPxs = new int[width*height];
4      int[] outPxs = new int[width*height];
5      bitmapIn.getPixels(scrPxs, 0, width, 0, 0, width, height);
6      // pragma Paralldroid target lang (rs) map(to : scrPxs, width, height) map (from :
   ↪ outPxs)
7      // pragma Paralldroid parallel for private (x, pixel, sum) rsvector(scrPxs, outPxs)
8      for (x = 0; x < width*height; x++) {
9          pixel = scrPxs[x];
10         sum = (int)(((pixel) & 0xff) * 0.299f);
11         sum += (int)(((pixel) >> 8) & 0xff) * 0.587f);
12         sum += (int)(((pixel) >> 16) & 0xff) * 0.114f);
13         outPxs[x] = (sum) + (sum << 8) + (sum << 16) + (scrPxs[x] & 0xff000000);
14     }
15     bitmapOut.setPixels(outPxs, 0, width, 0, 0, width, height);
16 }

```

Listing 8: Paralldroid usage example. Code extracted from Acosta and Almeida [2014].

The framework’s tests presented by Acosta and Almeida [2014] used a Java baseline and did not include GPUs, as the devices used were not compatible with OpenCL or RenderScript. Consequently, results mention only CPU speedups. As it can be seen in Figure 2.2, even though Native C provides improvements over the baseline, significant speedups are obtained only in RenderScript, with the generated code’s performance falling well below the handmade implementations. The results show, however, that significant speedups can be obtained with little coding effort.

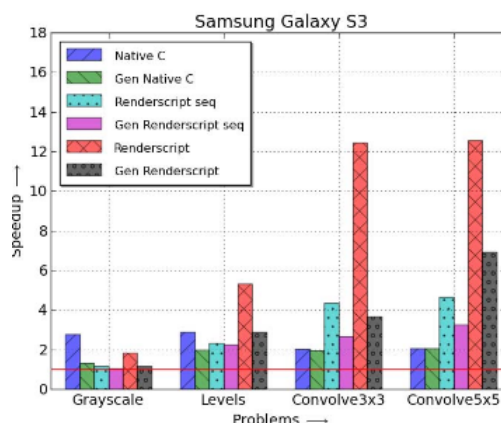


Figure 2.2: Paralldroid experimental results. Image extracted from Acosta and Almeida [2014]

2.3 Functional Programming

According to Odersky et al. [2011], functional programming is guided by two main ideas: that functions are first class values and that operations of a program should map input values to output values rather than change data in place. The ideas of functional programming date back to the 1930s and hinge on the formal system of mathematical logic known as lambda calculus [Church, 1941].

In functional languages functions are first class values, *i.e.*, they can be manipulated in the same way as values like numbers and characters. In other words, they can be passed as parameters to other functions, be returned as result of other functions and be store in variables. When functions are first class values, functionality can be transmitted throughout the code and higher levels of abstractions can be created. In this sense, if we take as an example a map function, it can have as input arguments different types of functions to be applied over a set of elements like in steps 2.1d and 2.1e of the following expression:

$$\text{Given } A = \begin{bmatrix} a^1 \\ \dots \\ a^n \end{bmatrix} \quad (2.1a)$$

$$\text{and functions } f(x) = x + 1 \quad (2.1b)$$

$$\text{and } f(z) = z - 2. \quad (2.1c)$$

$$\text{We can apply } A \xrightarrow{f(x)} B \quad (2.1d)$$

$$\text{and } A \xrightarrow{f(z)} C. \quad (2.1e)$$

The idea of considering operations that map input to output values instead of changing data in place means that functions should not have side effects, similarly to mathematical functions. In this sense, whenever a function is called no action other than producing a result is performed by this function, meaning that it will not retain state – a concept known as referential transparency. Such concept implicates that every function call could be replaced by its result without affecting the program’s semantics. This whole idea can be easily understood by analysing functions 2.1b and 2.1c as well as their respective calls in 2.1d and 2.1e. In the example, both calls produce new arrays B and C that are the result of applying functions $f(x)$ and $f(y)$ over all elements of A .

Several pure and impure functional programming languages have been proposed since Lisp [McCarthy, 1960], the first functional programming language introduced in the late 1950s. *Pure* and *impure* are terms used to describe, respectively, languages that follow

strict functional programming main ideas and those that include imperative programming features like mutable data structures and variables.

Concepts related to immutability introduced by functional programming are of special interest in this work, since they help to reduce the complexity of developing parallel programs. The use of immutable data structures favors the creation of functions without side effects, as it is not possible to change data in such structures. For this reason, it fosters the creation of functions that follow principles of functional programming, meaning that they will map the output to structures other than those used as input, which in turn will remain intact. Thus, code developed with immutable data structures and functions can be parallelized with a reduced effort, since concurrent writes are not created by the programmer.

One of the languages that incorporate functional as well as imperative programming features is Scala [EPFL, 2016, Odersky and Rompf, 2014], which has a mixture of object oriented and functional programming languages constructs. The language provides seamless interoperability with Java, as it compiles to Java bytecodes and executes in the Java Virtual Machine (JVM) environment. With an extensive set of mutable and immutable data structures for imperative and functional programming styles, Scala offers native support for parallelism in its collection library [Prokopec et al., 2011]. Though not directly supported by the target platform of this work (Android OS), the way immutable data structures are integrated with a parallel run-time in Scala is particularly important for this work, as described in the next chapter.

2.4 Android Mobile Operating System

The history of the Android Mobile Operating System dates back to 2003 when the Android Inc. was founded, and 2005 when the company was bought by Google. Since then, Android OS market share has increased continuously and achieved 87% of presence in mobile devices worldwide [International Data Corporation, 2016].

As shown in Figure 2.3, Android OS was developed as a multi-layered software stack. From bottom to top, the open-source system is comprised of the following layers [Google, 2016b]:

- Linux kernel: controls the fundamental hardware features;
- Hardware Abstraction Layer (HAL): provides standard interfaces with device hardware capabilities to the Java API framework;

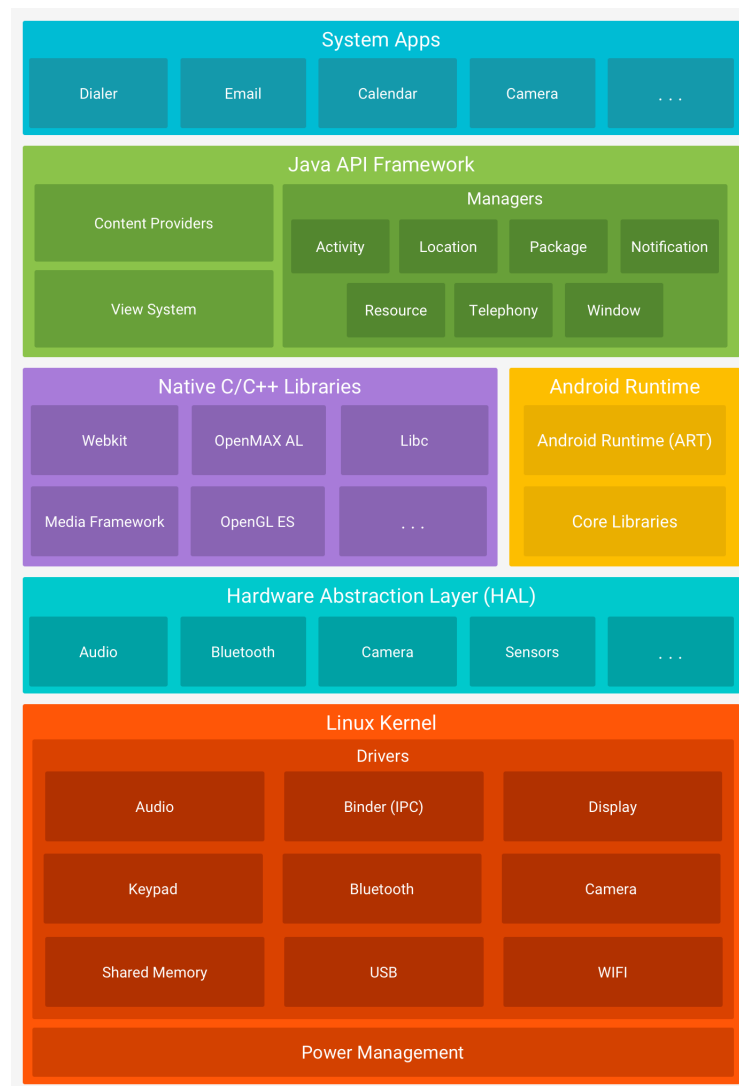


Figure 2.3: The Android software stack. Extracted from Google [2016b]

- **Android Runtime:** is a specific virtual machine for running special bytecode format for mobile devices;
- **Native C/C++ Libraries:** allows Android Native Development kit to access low-level features;
- **Java API Framework:** allows users to provide the fundamental building blocks for creating applications with a higher level API in Java;
- **System Apps:** contains a set of core applications for e-mail, SMS messaging, keyboard controls, web browsing, contacts and more.

Two layers of the Android stack are of particular interest for this work: the Java API Framework and Native C/C++ Libraries. They are part, respectively, of the Android Software Development Kit (SDK) and the Android Native Development Kit (NDK). Such

development kits are sets of programs and APIs for writing user applications in a high-level language (SDK) or native languages (NDK).

The Android SDK is normally used to write regular Java applications that do not require more features than those provided by the Android Java API Framework. By using the Android SDK, programmers are comfortably situated in a higher level of abstraction that offers automatic memory management (garbage collection) and a diverse set of UI components for building applications.

The Android NDK, in turn, is used to develop low-level applications in native languages like C and C++. These programs can be integrated into the Android SDK using Java Native Interface (JNI) calls [Oracle, 2016a]. Even though the NDK presents a much higher level of complexity compared to SDK, at the NDK level programmers have direct access to device drivers, memory allocation and are able to interact freely with the calling VMs from outside of the Java application. In this sense, it provides a interesting platform for programmers willing to avoid several levels of power-consuming software.

2.5 Summary

This chapter provided the bulk of concepts and motivations necessary to understand the contributions of this work, which will be presented in the next chapter. It provided an overview of previous attempts to reduce the necessary effort to develop applications for multi-core and heterogeneous architectures, and presented tools and frameworks designed for this particular goal. It also presented functional programming as an approach to be followed by sequential programmers in order to produce applications that can be automatically translated to parallel code with less effort. Lastly, it presented an overview of the Android OS in order to facilitate the understanding of the target mobile platform used in this work. The next chapter will introduce ParallelME and the major contributions of this work to this open-source project.

Chapter 3

Parallel Mobile Engine

This chapter presents ParallelME (Parallel Mobile Engine), an open-source framework designed to provide an easy-to-use programming abstraction for creating applications with higher performance and reduced energy consumption in Android devices. The framework is divided in three parts: *(i)* a programming abstraction (User Library), *(ii)* a run-time system and *(iii)* a source-to-source compiler. A description of these components and how they relate to each other is provided in the following sections.

3.1 Overview

ParallelME distinguishes itself from other frameworks by its high-level library, with a friendly collection-based programming abstraction in Java, and the ability to efficiently coordinate the usage of resources in heterogeneous mobile architectures with task schedulers. This is achieved through a source-to-source compiler that translates the high-level operations in Java to a low-level representation, creating tasks that are coordinated during execution by RenderScript and the OpenCL-based ParallelME Run-time.

ParallelME chooses the run-time platform dynamically during execution depending on the hardware and software resources available on the target mobile system. The framework combines reduced programming effort, a wide support of target devices, low energy consumption and efficient coordinated-usage of the available resources.

In order to facilitate the understanding of the framework, the relationships between its components are illustrated in Figure 3.1. This image shows how ParallelME modules relate to each other and how the user interacts with the framework.

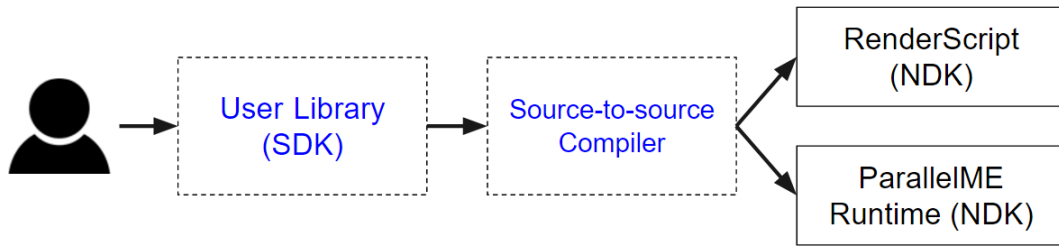


Figure 3.1: ParallelME framework overview

3.2 User Library

The User Library holds the programming abstraction proposed by ParallelME, being the high-level API that is directly handled by the user code. It was inspired by the Scala collections library [Prokopec et al., 2011] and has been designed to provide an easy-to-use and generic programming model for parallel applications in Java for Android.

The programming abstraction was devised to offer an intuitive transition for parallel programming. For this reason, it introduces concepts of data-parallelism using an abstraction that can be easily understood by sequential programmers. The User Library offers a collection-driven approach for code development. It supports specific types of data sets and introduces special functions to iterate, reduce, perform data-transformation and create sub-sets. These operations are executed after being translated by ParallelME Compiler in high-performance parallel run-times at NDK level. As debugging at NDK level is not an easy task, a fully functional sequential implementation in Java is also provided for each collection. It means that the User Library allows user applications to be debugged at SDK level using Android’s regular development infrastructure in Java.

An in-depth description of ParallelME User Library and all its contributions are presented in Chapter 4. Furthermore, that chapter presents a comparative analysis of the effort necessary to write code using the User Library, regular Java and the target run-times.

3.3 Internal library

The internal library is an integration feature which is composed of static Java and C code. It is comprised of different routines, like data-binding and data-allocation operations, used to store predefined code that is later handled by the compiler during the integration of user code and ParallelME Run-time. As it is simply a set of routines that

is stored along with the compiler source-code, it will be described as part of the compiler in its own section.

3.4 Run-time System

The run-time system of ParallelME was developed in C and C++ using OpenCL. It is responsible for setting up the application to allow several parallel tasks to be specified and queued for execution on different processing units, providing a transparent scheduling mechanism that allows tasks to be executed in CPUs or GPUs.

Given that ParallelME Run-time is not one of the contributions of this work, its detailed description is presented in Appendix A. Due to its performance and low-energy consumption [Andrade et al., 2016], this low-level framework is a key component of ParallelME. As the contributions of this work and ParallelME Run-time components are intimately connected, the understanding of the latter is required to comprehend the former.

3.5 Source-to-source Compiler

ParallelME source-to-source compiler provides a mechanism for translating user code to a low-level parallel implementation in the specified target run-times. The compiler takes as input Java code written with the User Library and translates it to a new version integrated with both RenderScript and ParallelME Run-time.

The translation is performed during compilation time, while the target run-time is chosen at user-application execution time. The output code generated by ParallelME evaluates during execution if the hardware supports OpenCL and runs code in ParallelME Run-time. Otherwise, if there is no support for OpenCL, ParallelME will transparently choose RenderScript, which is supported by all recent Android devices.

An in-depth description of ParallelME Compiler and all its contributions are presented in Chapter 5. Moreover, that chapter presents a comparative analysis of the performance of code translated by ParallelME Compiler and manually-optimized code in both target run-times.

3.6 User Library and Compiler Evaluation

In order to evaluate the proposed programming abstraction and the source-to-source compiler, an image processing application was used. This application uses a specific tone mapping algorithm developed to process high dynamic range (HDR) images.

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. Print-outs, CRT or LCD monitors and projectors all have a limited dynamic range that is inadequate to reproduce the full range of light intensities present in natural scenes. Tone mapping addresses the problem of strong contrast reduction from the scene radiance to the displayable range while preserving the image details and color appearance that are important to appreciate the original scene content. In this work the implementation proposed by Reinhard et al. [2002] was used.

Reinhard's tone mapping algorithm first computes the luminance map, which is taken into the log scale. That luminance is then adjusted with the logarithmic summation of luminance to normalize the output of the luminance map (from zero to one). With the normalized luminance map, a saturation of the radiance map is achieved by dividing each of the radiance maps by the luminance map (normalizing its color channel by its luminance). The saturated radiance is then scaled with the luminance map values. The result is then clamped to the range $[0, 1]$ to keep the values in the displayable range.

Figure 3.2 shows an overview of the application, in which each box represents an operation of the proposed algorithm.

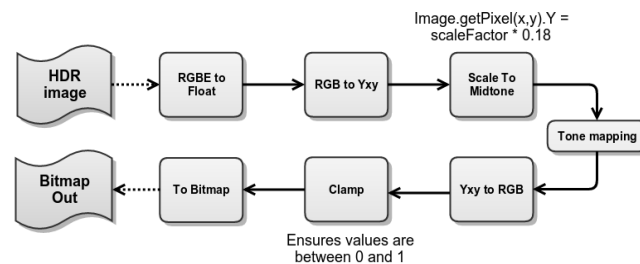


Figure 3.2: Application algorithm overview.

3.7 Summary

The chapter presented an overview of ParallelME, an open-source framework that offers a complete infrastructure for the development of parallel applications for heterogeneous mobile devices. The three-layer structure of the framework was briefly described in order to provide a basic understanding of its components and their relationship. Furthermore, the proposed application that is used to evaluate the contributions of this work in the following chapters was also presented.

Chapter 4

User Library

This chapter introduces ParallelME User Library, the module responsible for providing the framework’s programming abstraction. The library, its relationship with other components of ParallelME and an evaluation performed with the proposed application are described in the following sections.

4.1 Programming Abstraction

Inspired by concepts presented in the Scala Collection Framework [Prokopec et al., 2011], the programming abstraction proposed in the User Library provides an easy-to-use and generic programming model for creating parallel applications in Android. ParallelME User Library was designed to offer a smooth path towards parallel programming, transforming sequential code into parallel code without introducing complex concepts.

Since the first prototypes that later became the current ParallelME User Library, the goal has been to develop the simplest programming abstraction possible. Such abstraction should allow users to produce parallelizable code without explicit references to parallel control mechanisms. Even after analyzing different proposals for parallel programming in Android [Gupta et al., 2013, Pratt-Szeliga et al., 2012, Giacaman et al., 2013, Acosta and Almeida, 2014], there was a lack of a sufficiently simple, notation-free, and low-complexity programming abstraction that could be adopted by ParallelME. For this reason, the Scala Collection Library [Prokopec et al., 2011] was analyzed in depth and many of its concepts were borrowed.

With a data-driven generic programming model, the Scala collection library allows the injection of user-defined code with lambda expressions in some of its supported operations like *foreach*, *map*, *reduce*, *filter* and *fold*. In order to offer users a smooth approach to parallelism, this injected user code is transferred to pre-defined parallel skeletons with a unique method call. This simple and effective approach eliminates the necessity of structural changes in user application. Also, it reduces significantly the complexity of

developing applications that can use the popular multi-core architecture present in most of current computer devices. An example of the Scala collection library simple approach to parallelism is shown in Listing 9. This code presents a map operation performed on an array to add one to each of its elements and return a new array named *result*. The left-side code shows the sequential operation, while the right-side shows the parallel version of the same algorithm, which is created by simply adding a call to the method *par* present in the array object.

```
val result = array.map(v => v + 1) | val result = array.par.map(v => v + 1)
```

Listing 9: Parallelization feature in the Scala collection library

The Scala code presented in Listing 9 is driven by data immutability and allows data-parallelism. Since the user code does not include any external dependencies or state changes that could incur in concurrent write issues, it can be easily transformed in parallel code as shown in the example. In order to introduce such concepts for programming in mobile devices, ParallelME User Library incorporates ideas of data-immutability.

In order to be smoothly integrated to the Android SDK, ParallelME User Library is fully implemented in Java. Choosing Java as the implementation language results from its integration with Android Development Tools and its popularity worldwide. Consequently, it is possible for the User Library to reach a higher number of users, since ParallelME may have future extensions for desktop and server systems.

All the evaluated Android frameworks [Gupta et al., 2013, Pratt-Szeliga et al., 2012, Giacaman et al., 2013, Acosta and Almeida, 2014, Google, 2016d, Khronos, 2016] presented debugs shortcomings, restricting code debugging to printing console messages. In order to overcome these limitations, ParallelME User Library provides a sequential implementation that can be entirely executed at Android SDK level in Java. This sequential implementation, intended to be used only for debugging purposes, allows applications to be easily debugged and validated with more sophisticated tools present at SDK level. These are tools used to debug regular Android applications. Once the code is validated, the user must simply add a *par()* method call, similarly to Scala, informing ParallelME Compiler that the code must be translated to a parallel high-performance version in the low-level run-times.

Much of the syntactic simplicity present in Scala code cannot be expressed in Java, especially concerning the current version supported by Android (Java 7) which does not support lambda expressions – an important element for code simplification. Lambda expressions allows functionality to be passed as an argument, creating a simple code instead of a more complex structure. Still referencing the example code in Listing 9, the code $v \Rightarrow v + 1$ is a lambda expression that is used to transparently pass a *add one*

functionality to the *map* operation that would otherwise be coded in a separate class or method.

Even though Java 7 does not support lambda expressions, the User Library was implemented to support functionality transmission, enabling users to inject code in its proposed operations. In order to exemplify the functionality transmission feature, the example code in Listing 10 shows the equivalent version of a parallel *add one* function using a *map* operation in ParallelME User Library in Java 7. Though the Java 7 examples with anonymous classes are also valid in Java 8, for the sake of simplicity, implementations using anonymous classes will be referenced as *Java 7* code in this document, while implementations using lambda expression as *Java 8* code, since the last feature is only valid in the most recent versions of Java.

```

1  Array<Int32> result = array.par().map(Int32.class, new Map<Int32, Int32>() {
2      @Override
3      public Int32 function(Int32 v) {
4          v.value = v.value + 1;
5          return v;
6      }
7  });

```

Listing 10: Functionality transmission in User Library

The restricted expressiveness of Java 7, compared to Scala, is evident when analyzing the differences of codes in Listing 10 and Listing 9, being Java clearly more verbose. The most recent version of Android (Marshmallow) was released with full support for Java 8 [Google, 2016c], meaning that lambda expressions are now available in the Android development environment, which helps reduce code verbosity. Although still not implemented in ParallelME Compiler, the programming abstraction proposed by ParallelME User Library has natural support for lambda expressions once they follow its specifications.

Lambda expressions are defined in Java 8 by the instantiation of an anonymous class with a single method implementation [Oracle, 2016b], corresponding exactly to the structure defined for functionality transmission in the User Library. The code shown in Listing 10 depicts a functionality transmission in ParallelME User Library, in which the *map* expression is followed by a functionality injection that is performed with the instantiation of a *Map* object. Similarly, code shown in Listing 11 shows the same functionality transmission using a lambda expression in Java 8.

```

1  Array<Int32> result = array.par().map(Int32.class, (v -> {
2      v.value = v.value + 1;
3      return v;
4  }));

```

Listing 11: Lambda expression in ParallelME User Library

Currently in the first version, the User Library contains a generic single-dimensional array class capable of handling numeric data types and two image classes for Bitmap and HDR types. These classes are integrated into the regular Android API through functions for data input and output, allowing users to move data to and from ParallelME. Furthermore, these classes enable the development of complex tasks from four basic operations: filter (sub-setting), foreach (iteration), map (data-transformation) and reduce (reduction). These operations follow the same working principle and are applied in the user code as an anonymous class followed by an implementation of a method named *function*. Ultimately, this method will store the user code, as exemplified in Listing 10.

4.2 Data binding

Once the User Library creates an abstraction layer between the SDK infrastructure and the high-performance run-times, there must be ways to transport data from the regular Java code to the User Library and backwards. For this reason, the User Library also offers specific functions for data input and output, complying with different memory handling approaches adopted by both low-level run-times (RenderScript and ParallelME Run-time). In this sense, a standard abstraction for data binding was created in User Library collections with two operations: input and output data binding.

Due to Renderscript architectural restrictions [Google, 2016d], memory must always be allocated at SDK level and then bound to the user kernel in C. Thus, ParallelME User Library requires that memory is allocated by users at SDK level, providing previously-allocated data structures in input-bind operations. Therefore, input-bind operations are always defined in constructors, in which users must provide an object during User Library collections instantiation with the data that will be handled by the collection, as shown in line 6 of Listing 12.

```
1  int[] data = new int[42];
2  // Initializing input data with some valid values
3  for (int i=0; i<data.length; i++) {
4      data[i] = i;
5  }
6  Array<Int32> array = new Array<Int32>(data, Int32.class); // Input-bind
7  array.par().foreach(...); // Some operation
8  int[] result1 = array.toArray(); // Output-bind type 1
9  int[] result2 = new int[42]; // Memory allocation at SDK level
10 array.toArray(result2); // Output-bind type 2
```

Listing 12: Input and output data binding in ParallelME User Library

Even though RenderScript restriction in memory allocation is the same for output-bind operations, the User Library provides functions that allocate the necessary memory at SDK level before calling the low-level run-time. Thus, output-bind operations are defined as method calls, allowing two different types of assignments, as shown in lines 8 and 10 of Listing 12. Assignment in line 8 automatically creates the memory allocation for a given object at SDK level, being this task executed by the User Library. Conversely, the assignment in line 10 demands that the necessary memory to store the collection data is allocated at SDK level like the example in line 9. This last function was created in order to allow users to reuse previously allocated objects and increase performance, which can be specially helpful when dealing with images. It is important to note, though, that the memory size must be precisely the same of the object being copied from the low-level run-time to the VM, otherwise users may face execution errors.

4.3 Operations

ParallelME User Library operations were designed to allow the development of more complex tasks from basic operations that provide features for iteration, data transformation, sub-setting and reduction. The operations follow the form $A \xrightarrow{f} B$, in which f corresponds to the user code which will be applied to the collection A to return B , being B the same collection with new values, an entirely new collection, an empty set or a single collection element.

4.3.1 Foreach

The *foreach* operation provides the means for iterating over all elements of a given collection applying the user code in each one. It allows a side-effect based iteration, meaning that the user can change the value of a given element during the iteration with the guarantee that this new value will be stored in the collection to be used in future operations. *Foreach* is expressed in the following form, where A is the collection and f is the user code:

$$\text{Given } A = \begin{bmatrix} a^1 \\ \dots \\ a^n \end{bmatrix}, \quad A = f(A)$$

Two code examples are provided in Listing 13, with an implementation performed with an anonymous class in the left side and with a lambda expression in the right side. Both implementations are equivalent and are compliant with Java 7 and Java 8, respectively. The type T shown in lines 1 and 3 corresponds to the element type that is handled by the collection, being a numerical type or pixel for Array class or a pixel for image classes.

<pre> 1 array.par().foreach(new Foreach<T>() { 2 @Override 3 public void function(T element) { 4 // User code 5 ... 6 } 7 }); </pre>	<pre> 1 array.par().foreach(element -> { 2 // User code 3 ... 4 }); </pre>
---	---

Listing 13: Foreach operation compliant with Java 7 (left) and Java 8 (right)

4.3.2 Map

The *map* operation provides a way of applying a user function over all elements of a given collection, returning a new collection as the result. It is mathematically expressed by in the form below, where A is the input collection, f is the user code and B is the resulting collection:

$$A = \begin{bmatrix} a^1 \\ \dots \\ a^n \end{bmatrix} \xrightarrow{f} B = \begin{bmatrix} b^1 \\ \dots \\ b^n \end{bmatrix}$$

Though it may be seen at first as similar to *foreach*, a *map* operation does not allow side-effects, meaning that the collection is considered an immutable data structure while applying the user function. For this reason, a *map* operation outputs a new collection with the same size whose elements are the result of each individual user function call during the iteration of the original collection. Consequently, all changes performed on a

given element are discarded after the user function execution, unless they are used as the function's return.

Another important feature of *map* operation is that it can be used to perform data-transformation, meaning that an array of integers can be turned into an array of floats using a *map*. The example code shown in Listing 14 shows two abstract types in line 1 and 3: *R* and *T*. The *R* type refers to the user function return type, while *T* refers to the current collection element type. These types may be different when the operation is used to perform data-transformation.

```
1 Array<R> result = array.par().map(R.class, new Map<R, T>() {
2     @Override
3     public R function(T element) {
4         // User code
5         ...
6         return ...;
7     }
8 });
```

Listing 14: Map operation implemented with anonymous class (Java 7)

Listing 15 shows the equivalent code presented in Listing 14, but this time implemented with a lambda expression in Java 8. It is important to note that the lambda expression implicitly recognizes the collection element type (*T*) and for this reason the only abstract type that must be provided is the element return type *R*.

```
1 Array<R> result = array.par().map(R.class, element -> {
2     // User code
3     ...
4     return ...;
5 });
```

Listing 15: Map operation implemented with lambda expression (Java 8)

4.3.3 Reduce

The *reduce* operation is an aggregation function designed to combine in pairs all the elements of a collection returning a single summary value. In the User Library, the summary value represents an element of the type handled by a given collection.

Reduce does not allow side-effects, meaning that the collection is considered an immutable data structure when applying the user function. In this sense, all changes performed on a given element are discarded after that instance of the user function is executed, unless it is used as the function's return. When an element is used as a return, all changes undergone the function scope are propagated to the next iteration.

The operation has the following form, in which A is the input collection, f is the user code with a combiner function and b is the summary value:

$$A = \begin{bmatrix} a^1 \\ \dots \\ a^n \end{bmatrix} \xrightarrow{f} b$$

Listing 16 shows implementations with an anonymous class in Java 7 (left) and its equivalent version with a lambda expression in Java 8 (right). The abstract type T represents the collection element type, which is also the summarized value type associated to the *result* variable. The user function, acting as a combiner, receives a pair of input parameters (*elem1* and *elem2*), performs its operations and returns a single value. Each return value will be used as an input for the next iteration on the collection, forming a new pair of input parameters with a collection element not visited yet by the reduction iterator.

<pre> 1 T result = array.par().reduce(new ↪ Reduce<T>() { 2 @Override 3 public T function(T elem1, T elem2) { 4 // User code 5 ... 6 return ...; 7 } 8 }); </pre>	<pre> 1 T result = array.par().reduce((elem1, ↪ elem2) -> { 2 // User code 3 ... 4 return ...; 5 }); </pre>
--	--

Listing 16: Reduce operation compliant with Java 7 (left) and Java 8 (right)

4.3.4 Filter

The *filter* operation provides a way to create sub-sets. In this sense, the user function decides the criteria that will be used to filter elements of a given collection. In order to accomplish that, the code iterates over all the collection elements and, based on the return of the user function, decides if a given element will be part of the resulting collection or not.

Filter does not allow side-effects, meaning that the collection is considered an immutable data structure when applying the user function. Therefore, all changes performed on a given element are discarded after that instance of the user function is executed.

The operation is expressed by the following form, where A is the input collection, f is the user function with a boolean result and B is the resulting collection:

$$A = \begin{bmatrix} a^1 \\ \dots \\ a^n \end{bmatrix} \xrightarrow{f} B = \begin{bmatrix} b^1 \\ \dots \\ b^m \end{bmatrix} \vee B = \emptyset, \quad \text{where } B \subseteq A$$

Listing 17 shows an implementation with an anonymous class in Java 7, whereas Listing 18 shows its equivalent version with a lambda expression in Java 8. The abstract type T represents the collection element type, which is also associated to the *result* variable. In this code, the user function returns a boolean value. In this sense, all elements that return true in the user function call will be added to the resulting collection. Otherwise, they will be discarded.

```

1  Array<T> result = array.par().filter(new Filter<T>() {
2      @Override
3      public boolean function(T element) {
4          // User code
5          ...
6          return ...; // boolean return
7      }
8  });

```

Listing 17: Filter operation implemented with anonymous class (Java 7)

```

1  Array<T> result = array.par().filter(element -> {
2      // User code
3      ...
4      return ...; // boolean return
5  });

```

Listing 18: Filter operation implemented with lambda expression (Java 8)

4.4 Debugging Support

One of the greatest limitations of low-level code development in Android is the restricted debugging support. Even when the RenderScript framework is considered, the available tools for code debugging are equivalent to inserting *print line* commands throughout the user code. As previously mentioned, one of the key goals of ParallelME was to overcome such limitations and enable code debugging in the high-level development environment of Android SDK. For this reason, all User Library classes have a sequential implementation of all operations, allowing users to find errors in application logic using regular Android development tools.

The sequential implementation that enables code debugging can be used by removing the *par* call that instructs ParallelME Compiler to generate low-level code. Without

this call the operation is ignored by the compiler, being directly accessed in the User Library class and providing the user code to the sequential implementation, as shown in Listing 19.

```
1 array.foreach(new Foreach<T>() {
2     @Override
3     public void function(T element) {
4         // User code
5         ...
6     }
7 });
```

```
1 array.foreach(element -> {
2     // User code
3     ...
4 });
```

Listing 19: Debugging Foreach operation in Java 7 (left) and Java 8 (right)

The code created to allow application debugging at SDK level in Java was developed exclusively to provide users a to mimic the code that is automatically created by ParallelME Compiler. Therefore, it contains some features designed exclusively to maintain functionality compatibility, e.g, the (x, y) coordinates for pixels in images. Furthermore, as the sequential implementation was developed in Java, it is much slower even considering the sequential code created by the compiler in any of the low-level frameworks. Thus, the sequential version of ParallelME User Library must be used exclusively for debugging purposes at Android SDK level.

4.5 Evaluation

This section describes three implementations of the same tone mapping application referred in section 3.6. Implementations using the proposed ParallelME User Library and in the low-level frameworks RenderScript and the OpenCL-based ParallelME Run-time are presented in detail. The complexity of programming in each of these platforms is compared using a Source Lines of Code (SLOC) approach and a regular Java implementation as the base-line.

4.5.1 Comparison of ParallelME User Library and Low-level Frameworks

In order to evaluate the effectiveness and lower-complexity of the proposed programming abstraction, the following sections describe in detail implementations using ParallelME User Library, RenderScript and the OpenCL-based ParallelME Run-time. These different implementations were used in the comparison with the Java base-line application. The ParallelME User Library, RenderScript and the OpenCL-based ParallelME Run-time implementations were also employed to evaluate the performance of the code generated by the proposed source-to-source compiler, which is presented in the next chapter.

4.5.1.1 ParallelME User Library Implementation

In the example application *HDRImage*, a ParallelME built-in class for HDR image processing was used to implement all necessary operations that compose the tone mapping algorithm. This class offers a pixel-based programming abstraction that encapsulates each of its RGB elements and XY coordinates in a *Pixel* object, facilitating the development with a simple and straightforward interface. A segment of the application code is shown in Listing 20.

In order to use *HDRImage* or any ParallelME class, it is necessary to provide the source data as an array of a primitive Java type. In this case, an array of bytes is provided in the constructor along with image width and height, performing an input-bind operation as shown in line 1. *HDRImage* needs an array of bytes with one byte for each color (red, green and blue) for all pixels in the source image.

Operations that need to iterate over elements were implemented using *foreach* and *reduce* methods. These operations are shown in lines 2 and 16, being preceded by a *par* method call and followed by the creation of homonym anonymous classes implementing the *function* method. This method is ultimately responsible for holding the user code that may be written with the usage of Java primitive types, control flow statements, assignments, arithmetic and unary operators, as well as *java.lang.Math* operations.

A call to *par* method indicates to the compiler that the following operation must be translated to both target run-times, creating parallel or sequential low-level code according to the analysis of different execution forms presented in section 5.4.1. Conversely, the removal of the *par* method call allows the developer to execute the user code sequentially at

```

1  HDRImage image = new HDRImage(data, width, height); //Input
2  image.par().foreach(p -> { // Scale to Yxy
3      float result0, result1, result2;
4      float w;
5      result0 = result1 = result2 = 0.0f;
6      result0 += 0.5141364f * p.rgba.red;
7      ...
8      if (w > 0.0) {
9          p.rgba.red = result1; // Store value
10         ...
11     } else {
12         p.rgba.red = 0.0f; // Store value
13         ...
14     }
15 });
16 // Gets max value and sum logs
17 Pixel ret = image.par().reduce((p1, p2) -> {
18     ...
19     if (p1.rgba.red > p2.rgba.red) {
20         p1.rgba.alpha += p2.rgba.alpha;
21         return p1;
22     } else {
23         p2.rgba.alpha += p1.rgba.alpha;
24         return p2;
25     }
26 });
27 sum = ret.rgba.alpha;
28 ...
29 Bitmap bitmap = image.toBitmap(); // Data output

```

Listing 20: Tone mapping with ParallelME User Library

SDK level in Java. This is an useful feature for debugging applications before submitting them to ParallelME Compiler, overcoming both RenderScript and OpenCL frameworks limited debugging tools.

The *foreach* operation enables developers to perform changes in collections' elements in the user code provided. This feature is used in the example application, among other uses, to perform RGB to Yxy color space conversion, storing modifications in pixel parameters that are used in next steps of the algorithm, as shown in lines 9 and 12.

The *reduce* operation in line 16 is performed with the image data modified on the previous *foreach*. Unlike *foreach*, *reduce* operations do not store modifications in elements' properties unless the modified element is used as a function return value. It works by iterating over all collections' elements, providing the user pairs (p1 and p2 in this example) that will be used to perform the reduction. The user is then responsible for modifying and choosing which element will be provided as the first element (p1) of the next iteration, iterating until all elements are visited and producing a single result in the end.

After all operations are performed on the *HDRImage* object, the data processed in the target run-time must be returned to the user application in the SDK. This is achieved with an output-bind operation. In this example, the output-bind is performed by calling the *toBitmap* method, as shown in line 29. This method call produces a *Bitmap* object which can be handled at SDK level with the regular Android API.

The algorithm in Listing 20 shows the simplicity of ParallelME programming ab-

straction and how easy it is to create parallel operations that perform computations in collections.

4.5.1.2 ParallelME Run-time Implementation

ParallelME Run-time is an OpenCL-based tool designed primarily to provide an API for transparently exploring heterogeneous architectures. This run-time wraps the OpenCL framework and includes heuristic schedulers to improve the usage of computing resources in mobile devices. It has been discussed in a previous work [Andrade et al., 2016] and directly compared to OpenCL CPU and GPU implementations. In that work, ParallelME Run-time was 32.34 times faster than the baseline application written in Java. It was also 85% faster than OpenCL CPU and 34% faster than OpenCL GPU. These significant gains of performance were achieved with equivalent memory consumption (2% difference) and total energy consumption (1% difference) when compared to the best OpenCL results.

The tone mapping implementation using ParallelME Run-time was entirely created at NDK level, with a single call from SDK in Java. All the necessary input data was provided in this call, supplying the image byte array, height and width in the same way as previously described for the *HDRImage* class. All application parameters and a reference to the output bitmap object were provided in the single call as shown in Listing 21.

```
1 private native void nativeRunOp(long tonemapperPtr,  
2     int width, int height, byte[] data, float key,  
3     float power, Bitmap bitmap);
```

Listing 21: Native call for ParallelME Run-time in SDK

At NDK level, the application code was split in two parts: ParallelME Run-time calls and user operations. The ParallelME Run-time calls correspond to the code that is necessary to use the run-time itself, creating buffers and tasks, providing application parameters, calling user operations and copying the computed result to the output bitmap. A fraction of those operations are shown in Listing 22.


```

1 void ScheduledTonemapper::tonemap(int width, int height,
2     float key, float power, JNIEnv *env,
3     jbyteArray imageDataArray, jobject bitmap) {
4     size_t workSize = width * height;
5     size_t imageSize = workSize * 4;
6     auto outputBitmap = env->NewGlobalRef(bitmap);
7     auto imageBuffer = std::make_shared<Buffer>(imageSize);
8     imageBuffer->setJArraySource(env, imageDataArray);
9     ...
10    auto task = std::make_unique<Task>(_program);
11    task->addKernel("to_float") // Define user
12        ->addKernel("to_xyxy") // operations' call order
13        ...
14        ->addKernel("to_bitmap");
15    task->setConfigFunction( [=] (DevicePtr &device, ...) {
16        kernelHash["to_float"] // Set user operation
17        ->setArg(0, imageBuffer) // parameters
18        ->setArg(1, dataBuffer)
19        ...
20    });
21    task->setFinishFunction( [=] (DevicePtr &device, ...) {
22        imageBuffer->copyToAndroidBitmap(device->JNIEnv(),
23        outputBitmap);
24        device->JNIEnv()->DeleteGlobalRef(outputBitmap);
25    });
26    _runtime->submitTask(std::move(task)); // Exec. operat.
27 }
28

```

Listing 22: ParallelME Run-time calls

In order to comply with OpenCL requirements at NDK level, user operations must be stored inside a kernel string as shown in Listing 23. That string is then compiled by OpenCL and called by ParallelME Run-time in line 11 of Listing 22.

```

1 const char kernelSource[] =
2     "__kernel void to_xyxy(__global float4 *data) { "
3     " int gid = get_global_id(0); "
4     " float4 result = (float4)(0.0f); "
5     " float4 pixel = data[gid]; "
6     " result.s0 += 0.5141364f * pixel.s0; "
7     ...
8     " float w = result.s0 + result.s1 + result.s2;"
9     " if(w > 0.0f) { "
10    " pixel.s0 = result.s1; // Y "
11    ...
12    " } else { "
13    " pixel = (float4) (0.0f); "
14    " } "
15    " data[gid] = pixel; "
16    "}"
17    ...

```

Listing 23: Kernel string in C99 for OpenCL

Even though the ParallelME Run-time programming interface is considerably less complex than the original OpenCL API, it is easy to observe that developing applications with native code in NDK level is a much more complicated task than using ParallelME User Library. Furthermore, encapsulating user operations in a string creates a big issue for

code validation and debugging, since the string will be only compiled during application run-time.

4.5.1.3 Implementation for RenderScript

Since RenderScript is part of the Android package, it is considerably more integrated to the development environment, being easily included in user application. The framework also offers a much less complex programming interface compared to ParallelME Run-time or OpenCL, with a user-API in Java. For this reason, the low-level programming effort is reduced, being limited to the development of user operations with the RenderScript API in C.

The tone mapping implementation in RenderScript was performed in Java and C. RenderScript Java API offers abstractions to handle input and output data binding as well as calling user operations. Similarly to ParallelME Run-time and OpenCL, user operations are those functions that relate to the data processing, being created in separate kernel files, as shown in Listing 25. Once the user kernel is created, RenderScript automatically creates a Java wrapper class in the Reflected Layer that contains all the necessary tools for integrating the low-level operations with the Android framework in Java. This class, declared in line 4 of Listing 24, is then used to call user operations created in the kernel file.

```
1  RenderScript mRS;
2  Allocation dataAlloc, imageAlloc;
3  ...
4  ScriptC_tonemapper operations; //Operations' wrapper
5  ...
6  Type imageType = new Type.Builder(mRS, Element.
7      RGBA_8888(mRS)).setX(width)
8      .setY(height).create();
9  Type dataType = new Type.Builder(mRS, Element.F32_4(mRS))
10     .setX(width).setY(height).create();
11  imageAlloc = Allocation.createTyped(mRS, imageType);
12  dataAlloc = Allocation.createTyped(mRS, dataType);
13  imageAlloc.copyFrom(data); // Input data
14  // Run user operations
15  operations.forEach_to_float(imageAlloc, dataAlloc);
16  operations.forEach_to_xyxy(dataAlloc, dataAlloc);
17  ...
18  operations.set_tonemapData(dataAlloc);
19  operations.forEach_tonemap(dataAlloc, dataAlloc);
20  operations.forEach_to_bitmap(dataAlloc, imageAlloc);
21  imageAlloc.copyTo(bitmap); // Data output
```

Listing 24: SDK tone mapping algorithm in RenderScript

```

1  float4 __attribute__((kernel)) to_yxy(
2      float4 in, uint32_t x, uint32_t y) {
3      float4 result, out;
4      result.s0 = result.s1 = result.s2 = 0.0f;
5      result.s0 += 0.5141364f * in.s0;
6      ...
7      float w = result.s0 + result.s1 + result.s2;
8      if (w > 0.0f) {
9          out.s0 = result.s1;    // Y
10         ...
11     } else {
12         out = 0.0f;
13     }
14     return out;
15 }
16 rs_allocation logAverageData;
17 int logAverageWidth;
18 float4 __attribute__((kernel)) log_average(
19     float4 in, uint32_t x, uint32_t y) {
20     float sum = 0.0f;
21     float max = 0.0f;
22     ...
23 }

```

Listing 25: Kernel file with user operations in RenderScript

Though the framework offers tools that reduce the complexity of its usage, the user must control data allocations, i.e., the correct memory size and type to store the data. Conversely, debugging tools for user kernels, although far from being easy to use as in Android Java, are considerably better than ParallelME Run-time and OpenCL user kernels, with pre-existing functions that print debugging messages to the standard output. The framework programming interface, though simpler than ParallelME Run-time and OpenCL, is still very complex when compared to ParallelME User Library, requiring a more specialized programmer and offering a steeper learning curve as any low-level framework for parallel programming.

4.5.2 Sources Lines of Code

The analysis of programming complexity was performed by counting Source Lines of Code (SLOC). In order to do that the open-source project CLOC [Danial, 2016] was used in this work. All the source files evaluated were in similar formatting style. Furthermore, blank lines and commented lines were not considered. The count procedure was performed for all source files that were directly created by the programmer, not considering automatic generated code like those created by ParallelME Compiler or those in RenderScript's Reflected Layer.

The goal of the proposed evaluation is to compare the amount of coding effort

necessary to run a given Java application in both target run-times to gain performance and reduce energy consumption as demonstrated by Andrade et al. [2016]. For this reason, the number of lines of code was evaluated in two aspects: (i) the amount of user code necessary to reproduce the image processing algorithm and (ii) the code necessary to run this algorithm in the platform (overhead). Thus, the image processing algorithm corresponds to all the code that is effectively used to process each pixel. Conversely, the overhead code corresponds to all of the remaining code necessary to handle the image and the processing algorithm itself in the platform.

Figure 4.1 shows that ParallelME User Library in Java 7, compared to the Java baseline, presents an equivalent number of lines for both user and overhead code, which in turn leads to similar total program sizes. A smaller overhead is observed in User Library in Java 8, implying in a smaller total lines due to code simplification provided by lambda expressions. When evaluating both target run-times, RenderScript and ParallelME Run-times present equivalent number of lines necessary to write the user code. Conversely, when evaluating the total program size, they present a considerably bigger program (200 and 249 lines, respectively) compared to the baseline (126 lines) and the User Library in Java 7 (129 lines). In this regard, the User Library in Java 7 presents a reduction of 35% in the total program size compared to RenderScript and a reduction of 49% compared to ParallelME Run-time.

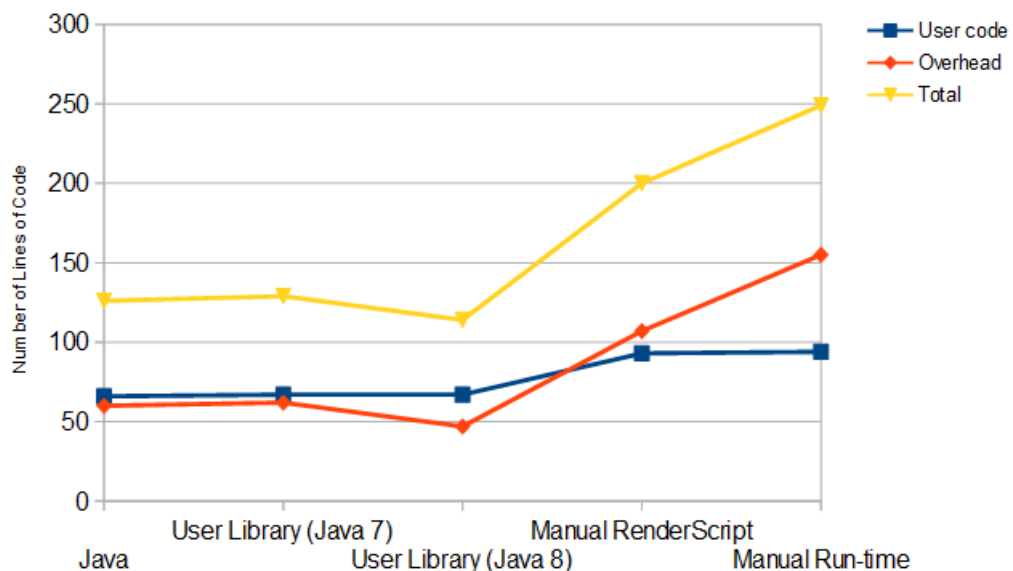


Figure 4.1: Number of lines of code evaluation

The number of lines of code of a program may not directly correlate with its complexity. Many other factors like the programming language or the application architecture itself may influence one's judgement of how difficult it is to understand a program. Consequently, the evaluation only considers aspects that can be compared directly with a tangible metric like the proposed SLOC count. Thus, although the sample application is

small it underestimates, for example, difficulties in programming at NDK level in C. For this reason, the perception of ParallelME User Library complexity by developers may be even more favorable than presented in this work.

4.6 Summary

This chapter presented in detail the ParallelME User Library. It provided an in-depth description of all the features that allows the User Library to bring a higher-level of programming abstraction for parallel programming. It showed three different implementations used to evaluate the effectiveness of the programming abstraction's contributions. Through a Source Lines of Code (SLOC) analysis, it was shown that the programming complexity of ParallelME User Library is considerably lower when compared to RenderScript, ParallelME Run-time and the intuitive sequential Java.

Chapter 5

Compiler

This chapter describes the internals of the proposed source-to-source compiler which is part of ParallelME. A detailed description of the translation process that allows code created with the ParallelME User Library become low-level representations in both target run-times is provided in the next sessions.

5.1 Overview

ParallelME source-to-source compiler was developed with the incorporation of ANTLR (**A**N**O**ther **T**ool for **L**anguage **R**ecognition), a powerful parser generator [Parr, 2013] widely used to build languages, tools and frameworks. An ANTLR Java grammar was used to create a parser capable of building and traversing a parse tree, which in turn was the compiler basis for lexical and syntax analysis.

ParallelME Compiler takes as input Java code written with the User Library and translates it to RenderScript and ParallelME Run-time, leaving the choice of which run-time to use for user application execution. In order to achieve that, the user code provided in User Library operations is translated to C code compatible with each low-level run-time. The translated code is then integrated to the Java application through an intermediary layer created to wrap both low-level run-times, including a mechanism capable of choosing the appropriate run-time during application execution.

The source-to-source was designed to completely decoupled run-time specific features from code translation tasks as shown in Figure 5.1, being divided in three well-defined phases: first pass, second pass and translation. The first pass uses the parse tree to create the symbol table, while the second pass uses the information acquired in the first pass to create the intermediate representation. Finally, the translation phase uses the information acquired in first and second passes to translate the user code. It then creates RenderScript and ParallelME Run-time implementations and integrates all the translated code with the original user class in Java.

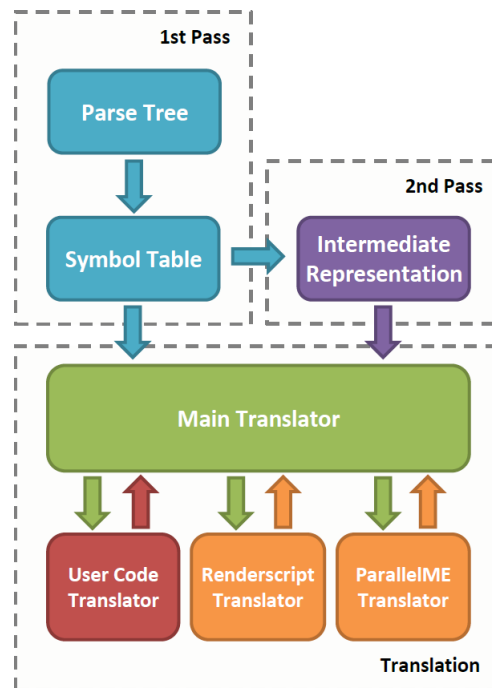


Figure 5.1: Compiler overview

5.2 First Pass

The compiler's first pass is responsible for lexical and syntax analysis, creating the symbol table at the end of its execution. It was developed with ANTLR, a powerful parser generator for reading, processing, executing, or translating structured text or binary files [Parr, 2016] [Parr, 2013]. It is widely used to build languages, tools and frameworks, and allows the automatic generation of parsers from a given grammar definition.

For ParallelME Compiler, a Java grammar was applied to ANTLR in order to generate the Abstract Syntax Tree, creating classes that are used by the compiler to build and traverse parse trees. Such infrastructure is composed of specific classes inherited from those automatically generated by ANTLR, which are employed to perform lexical and syntax analysis in the user code, exploring its parse tree during the first pass and building the symbol table. The code in Listing 26 is a declaration of a User Library object and Figure 5.2 shows its equivalent parse tree.

```
1 BitmapImage image = new BitmapImage(bitmap);
```

Listing 26: User Library object declaration

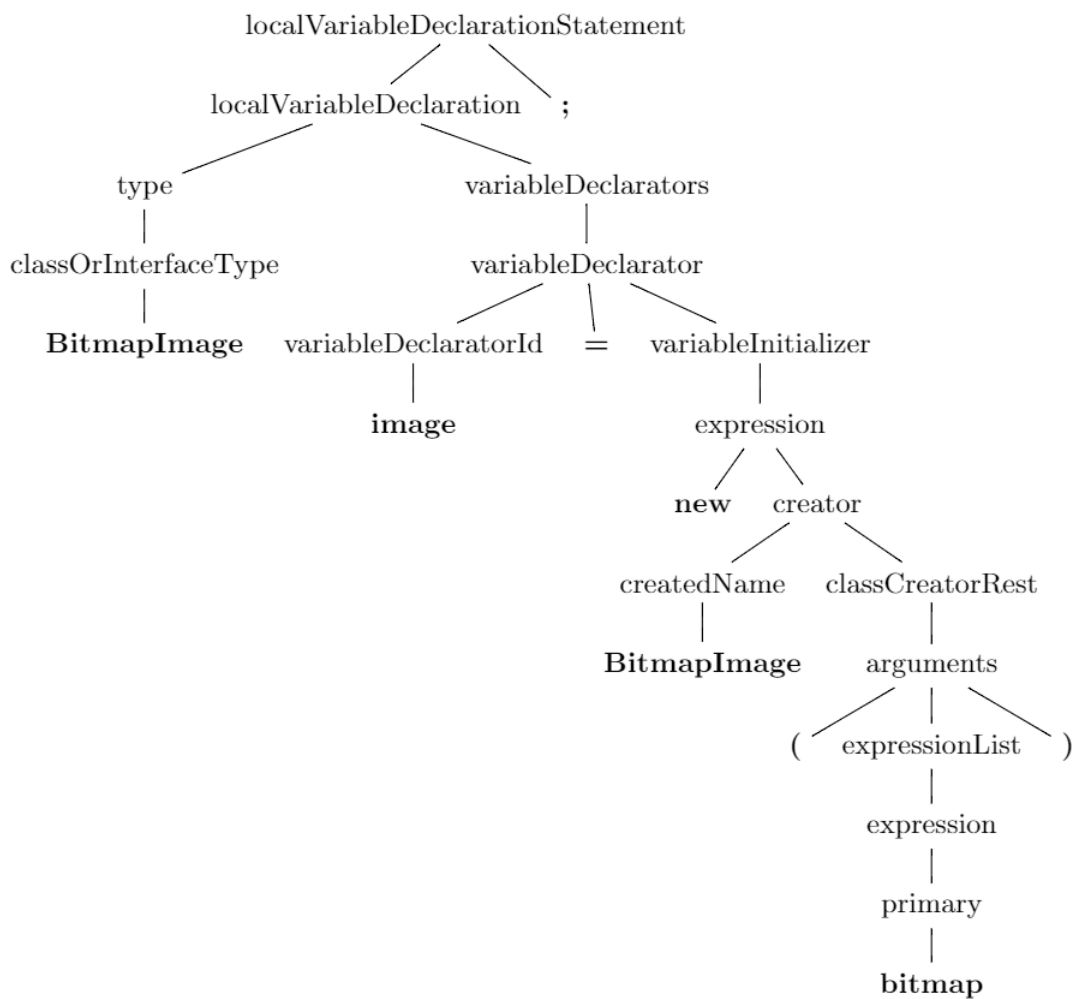


Figure 5.2: Parse tree for code presented in Listing 26

The symbol table conceived for ParallelME Compiler was designed as a hierarchical and scope-driven structure, i.e., all symbols stored on the table are in the same hierarchical scope as in the original input code. This allows the necessary scope-level analysis in the second pass to be performed with a reasonable computing cost.

All operations related to the symbol table creation are implemented in *Scope-DrivenListener* class. This class is inherited directly from ANTLR *JavaListener* class and uses *enter* and *exit* methods provided by ANTLR during entrance or exit of each parse tree node.

During the parse tree traversal symbols for classes, methods, variables, User Library variables and creators are inserted in the symbol table. Each symbol stores basic description information from its type representation, like methods' return types and arguments, variables' types, creators' parameters and more. These symbols are stored with as much relevant information as possible to increase the performance of the second pass. An example of the scope-driven symbol table created by ParallelME Compiler is shown in Listing 27.


```

<RootSymbol> $
  <ClassSymbol> UClass
    <UserLibraryVariableSymbol> image, BitmapImage
    <MethodSymbol> load, Bitmap, [<VariableSymbol> res, Resources, ; ...]
      <VariableSymbol> res, Resources
      <VariableSymbol> resource, int, null
      <VariableSymbol> options, BitmapFactory
      <CreatorSymbol> $optionsCreator, BitmapFactory.Options, options
      <VariableSymbol> bitmap, Bitmap
      <CreatorSymbol> $imageCreator, BitmapImage, image
      <CreatorSymbol> $anonObject1, UserFunction, Pixel
      <MethodSymbol> function, void, [<VariableSymbol> pixel, Pixel]
        <VariableSymbol> pixel, Pixel
        <MethodBodySymbol>
          <VariableSymbol> foo, Pixel
          <CreatorSymbol> $fooCreator, Pixel, foo, , []
          <VariableSymbol> w, float

```

Listing 27: Scope-driven symbol table

5.3 Second Pass

The second pass is responsible for a deeper analysis of the user code. During this stage, the compiler performs another evaluation of the source code, this time with the information gathered on the first pass and stored on the symbol table. This analysis will produce as result the intermediate representation that will be used for code translation to both target run-times.

The intermediate representation was designed to represent not only variables, literals or other features, but also abstract tasks that describe in a high level how the programming abstraction works. Such tasks are input binds, operations and output bind calls performed in User Library classes. They represent data transfer from Java environment to low-level run-times, user operations performed in the collection data and the retrieval of processed data from the low-level run-times, respectively.

```

1  class UClass {
2      public Bitmap someMethod(Bitmap inputBitmap) {
3          BitmapImage image = new BitmapImage(inputBitmap); // Input-bind
4          image.par().foreach(new Foreach<Pixel>() { // Operation
5              @Override
6              public void function(Pixel pixel) {
7                  pixel.rgba.red += 1;
8                  pixel.rgba.green += 2;
9                  pixel.rgba.blue += 3;
10             }
11         });
12         Bitmap returnBitmap = image.toBitmap(); // Output-bind
13         return returnBitmap;
14     }
15 }

```

Listing 28: Input bind, operation and output bind at User Library level

Following the collection of all the necessary data for building the in-memory intermediate representation in the second pass, data is sent to appropriate run-time translators that are responsible for creating the equivalent low-level code. Listing 28 shows a code example with input bind (line 3), operation (line 4) and output bind (line 12), while its equivalent intermediate representation is presented in Listings 29, 30 and 31.

```
InputBind {
  Variable {
    name: image
    typeName: BitmapImage
  }
  Parameters {
    Variable {
      name: bitmap
      typeName: Bitmap
    }
  }
}
```

Listing 29: Intermediate representation for input-bind

```
Operation {
  OperationType: Foreach
  ExecutionType: Parallel
  Variable {
    name: image
    typeName: BitmapImage
  }
  UserFunction {
    Variable {
      name: pixel
      typeName: Pixel
    }
    Code {
      pixel.rgba.red += 1;
      pixel.rgba.green += 2;
      pixel.rgba.blue += 3;
    }
  }
}
```

Listing 30: Intermediate representation for Foreach operation

```
OutputBind {
  Variable {
    name: image
    typeName: BitmapImage
  }
  DestinationVariable {
    name: bitmap
    typeName: Bitmap
  }
}
```

Listing 31: Intermediate representation for output-bind

5.4 Translation Steps

The code translation phase is responsible for transforming the intermediate representation into valid code for both target run-times. From receiving the intermediate representation up to the integration of user code with the low-level run-time, the translation phase is comprised of four sequential steps. These steps are presented in the following section in the order they are executed by ParallelME Compiler.

5.4.1 Analysis of Different Execution Forms

The analysis of different execution forms is performed for all operations in order to determine if the user code provided can in fact execute in parallel or if that operation should be executed sequentially. This step ensures that the user code does not include references for variables of scopes other than of the current operation, avoiding the creation of concurrency issues that can invalidate parallel execution. Thus, the user code provided in a given operation is analyzed to determine if its execution form must be sequential or parallel.

In the current version of ParallelME Compiler, the following rule is considered to assess an operation execution form: if a given operation contains user code that references non-final variables from different scopes, the operation will be marked as sequential, otherwise it will be marked as parallel. User code that references final variables from other scopes or does not contain any reference for variables of other scopes is deemed “parallelizable” by ParallelME Compiler.

The rule, although simple, defines precise boundaries in user code and simplifies the translation process. Once the user code makes reference to a variable from an outer scope that may have its value changed, the entire operation is executed sequentially in order to avoid errors in application semantics and to simplify next steps’ analyses. Though that rule can be improved by checking if the non-final variable is in fact being assigned, which will really create a concurrency issue, this approach was not considered in the current version of ParallelME Compiler in order to reduce the complexity of the semantic analysis.

The fragment code presented in Listing 32 is an example of a code that must have a sequential execution form. It contains a reference for a non-final class variable (*sum*) in the user code. This variable is assigned to a new value in line 5, meaning that if this code is executed in parallel, concurrent writes may cause inconsistencies on that variable.

In cases like this, unless the code is executed sequentially or a critical section is created, the variable value is unpredictable in the end of the operation. For this reason, when any external non-final variable is referenced inside the user code, the compiler will translate the operation to a sequential version for both target run-times, issuing a warning that indicates precisely what prevented the creation of parallel code.

```
1  this.sum = 0; // Class variable (not declared as final)
2  image.par().foreach(new Foreach<Pixel>() {
3      @Override
4      public void function(Pixel pixel) {
5          sum += Math.log(0.00001f + pixel.rgba.red); // Variable from different scope
6      }
7  });
```

Listing 32: User code referencing a non-final variable outside of its scope

5.4.2 Common interface for target run-time systems

In order to create the mechanism for dynamic selection of target run-time, it is necessary to define a base contract with common rules to delineate the integration between the user application and the run-time environments. This contract is used at user application level, being defined with a Java interface that is created by ParallelME Compiler with the information from the intermediate representation.

The communication between user application and target run-times is defined by those three abstract elements defined in ParallelME User Library: input-bind, operation and output-bind. Thus, the interface is defined with methods for each of these abstract elements called in the original user code, being later handled by the dynamic run-time selection mechanism.

An example interface is shown in Listing 33. It was created based on the code shown in Listing 28, being composed of one method for verifying the validity of the run-time (*isValid*), a method to define an input-bind (*inputBind1*), a method to define an operation (*foreach1*) and a method to define an output-bind (*outputBind1*). Each of these three last methods is followed by a number which is incorporated to methods' names in order to uniquely identify each user-defined interaction with the User Library. This sequential number is created during the second pass and is assigned to each element in the intermediate representation, being independent for input-binds, operations and output-binds. Whenever a developer interacts with the User Library, a unique identification composed of the element type and its sequential number is created in this interface.

```
1 public interface UClassWrapper {
2     boolean isValid();
3
4     void inputBind1(Bitmap bitmap);
5
6     void foreach1();
7
8     Bitmap outputBind1();
9 }
```

Listing 33: Interface for integrating user application and target run-times

One interface is created for each user class that references User Library objects and named after the original class name with an additional *Wrapper* suffix. Besides, this interface is created in the same package as the original class, thus reducing the amount of modifications necessary in the original user code in order to replace User Library calls by eliminating the necessity of including imports statements in the translated user class.

5.4.3 Run-time system specific translation

Once the contract that defines how the interaction between user application and target run-times is established, the compiler performs user code translation for RenderScript and ParallelME Run-time.

Due to the division of responsibilities in the compiler architecture, each target run-time has its own definition, which is completely independent. Whenever a translation to RenderScript or ParallelME Run-time is necessary, the compiler calls the appropriate run-time definition, which is then responsible for performing the translation with specific *translators*.

A *translator* is a specialized class that is responsible for translating specific types of collections. As the current version of ParallelME User Library supports *BitmapImage*, *HDRImage* and *Array* classes, there are three translators for each target run-time. Each translator receives data from the intermediate representation and outputs the run-time specific code in C for a given operation. Built in order to allow a template-based translation, each translator contains a series of templates that corresponds to the operations' structure in C for a given run-time, thus it creates the target implementation based on the existing parameters of the intermediate representation.

5.4.4 Original user code translation

ParallelME User Library was designed to provide a fully compatible high-level library that can be easily integrated to user applications to produce high-performance code. In this sense, it does not require that a given user class in Java uses exclusively those ParallelME User Library classes. Thus, user classes may be created as a mix between ParallelME User Library classes and others functions that may access different features, frameworks or classes as in any regular Java code. For this reason, the original user code provided as input to ParallelME Compiler receives special treatment.

In order to translate input code, the compiler analyses the user class and locates all those lines that make reference for ParallelME User Library collections and ultimately stores this information in the intermediate layer. With this information, the compiler generates the intermediate layer, with its common interface, and proceeds to modify the original user class by adding or replacing lines of code as shown in the compiler-translated version of Listing 28 shown in Listing 34.

```
1  class UserClass {
2      private UserClassWrapper PM_parallelME; // Common interface reference
3
4      public UserClass(RenderScript PM_mRS) { // New constructor
5          this.PM_parallelME = new UserClassWrapperImplPM();
6          if (!this.PM_parallelME.isValid())
7              this.PM_parallelME = new UserClassWrapperImplRS(PM_mRS);
8      }
9
10     public Bitmap someMethod(Bitmap inputBitmap) {
11         PM_parallelME.inputBind(inputBitmap); // Input-bind
12         PM_parallelME.foreach(); // Operation
13         Bitmap returnBitmap = PM_parallelME.outputBind(); // Output-bind
14         return returnBitmap;
15     }
16 }
```

Listing 34: Compiler-translated version of code presented in Listing 28

New lines of code are added to the original user class in order to integrate it with the intermediate layer. In this way, a new constructor and an object declaration for the common interface are added to handle the mechanism for dynamic selection of run-time. This constructor is then responsible for instantiating the appropriate implementation of the common interface, selecting its RenderScript or ParallelME Run-time version accordingly to the hardware support of the target mobile device.

On the other hand, those User Library calls created in the user class as input-binds, operations or output-binds are replaced by calls to the common interface. These calls, shown in lines 11, 12 and 13, are created by the compiler in order to integrate the original user application with those portions of code that were transferred to high-performance

run-times, thus preserving the initial user code semantics.

5.5 Translated Code Integration Architecture

As ParallelME is a framework that focuses in code translation from a high-level programming abstraction in Java to high-performance run-times in C and C++, it must be able to integrate the user application in the Android VM with RenderScript and ParallelME Run-time. Since both frameworks are located outside the VM, ParallelME follows well-defined architectural guidelines in order to translate and integrate code, providing transparent execution and memory handling for user applications in these high-performance environments.

Once RenderScript and ParallelME Run-time have distinct programming interfaces, ParallelME defines a common layer to enable the communication of Android application code with these different platforms. This layer, known as *ParallelME Java Layer*, defines a single communication protocol with both low-level run-times, maintaining the same interfaces for performing memory operations as well as controlling the execution of these run-times. As it provides transparent access for ParallelME Run-time and RenderScript, the existence of *ParallelME Java Layer* allows the creation of a mechanism for dynamically choosing the high-performance run-time. This mechanism is responsible for initializing automatically the appropriate low-level run-time according to the existing hardware resources of the target mobile device.

5.5.1 RenderScript

In order to comply with RenderScript, *ParallelME Java Layer* wraps RenderScript's automatically created *Reflected Layer*. *ParallelME Java Layer* is shown in Figure 5.3, being this image similar to that presented in the original RenderScript documentation [Google, 2016e]. This layer is called by the Android application code in order to allocate, read and write memory, as well as perform memory binding and triggering the execution of the RenderScript run-time.

RenderScript run-time is located outside the Android framework and comprises a *RenderScript Code* layer and a *Graphics and Compute Engine* layer. The *RenderScript Code* layer is composed of C99 code that represents the user-defined application

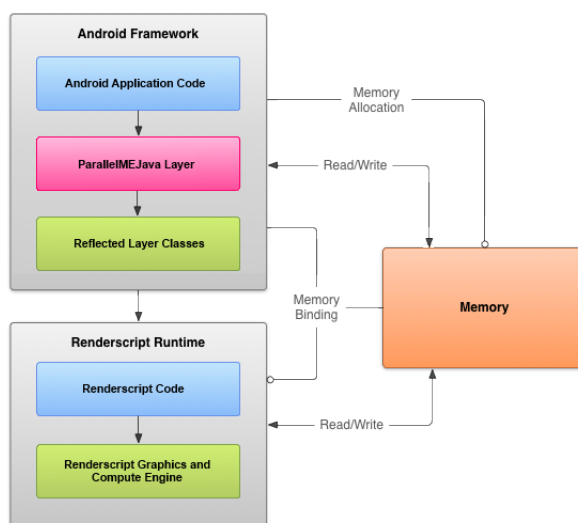


Figure 5.3: RenderScript integration architecture

behaviour. In ParallelME, this layer is created by the source-to-source compiler in order to store definitions created with the high-level programming abstraction. Once compiled by RenderScript framework, *RenderScript Code* is executed by the *Graphics and Compute Engine* during application run-time, performing reads and writes to the memory previously allocated at Android VM. Due to RenderScript’s architectural restrictions [Google, 2016d], it does not create new memory allocations at run-time level, thus requiring that segments of memory handled by user applications be always allocated at Android VM level in Java.

5.5.2 ParallelME Run-time

The run-time component of ParallelME was developed independently of the other parts of the framework, being compatible with mobile, desktop and server architectures. It means that ParallelME Run-time can be used directly without the User Library and the source-to-source compiler, though in this case the user will have to handle its C and C++ API. In this sense, in the Android platform it is entirely handled at NDK level, thus working outside the VM. For this reason, *ParallelME Java Layer* is used by the compiler to integrate the Android application code with this low-level run-time, as shown in Figure 5.4. Although the memory architecture is analogous to that shown for RenderScript in Figure 5.3, ParallelME Run-time does not have limitations regarding memory allocation outside the Android VM, thus it can allocate new memory segments directly at NDK level by calling special functions at JNI level. Similarly to the integration

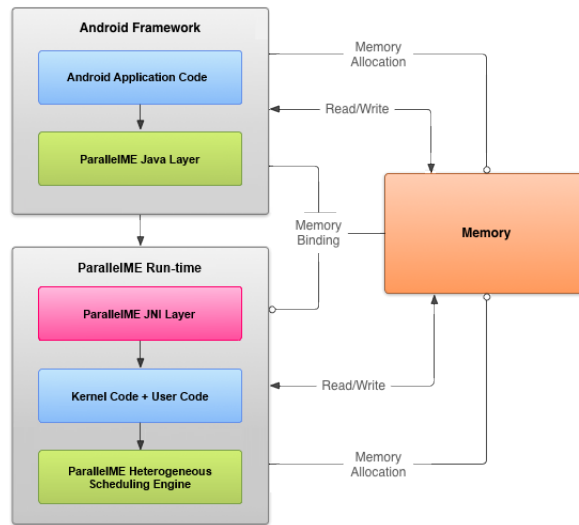


Figure 5.4: ParallelME Run-time integration architecture

with *RenderScript's Reflected Layer*, *ParallelME Java Layer* plays an important role in performing allocation, reads, writes and binding memory between the low-level ParallelME Run-time and the Android VM. ParallelME Run-time is composed of three layers: a *JNI Layer*, a *Kernel and User Code* layer, and the *Heterogeneous Scheduling Engine*.

ParallelME JNI Layer corresponds to the translated code that is adherent to the Java Native Interface (JNI) [Oracle, 2016a] specification. This layer allows the interoperability between the Java code that executes in the Android VM and the C and C++ codes that execute at native level. These native codes compose the kernel of ParallelME Run-time and the translated user code that is produced by ParallelME Compiler.

Kernel and User Code layer is composed of predefined functions that constitute the run-time kernel and allows the execution of user code translated by ParallelME Compiler. Similarly to *RenderScript Code*, the user code is implemented in C99 and is translated from the high-level programming abstraction in Java, being integrated to the run-time kernel. The kernel code is ultimately responsible for providing transparent access to the low-level heterogeneous scheduling engine at the bottom layer, allowing the execution of user code.

Finally, *ParallelME Heterogeneous Scheduling Engine* is responsible for executing the translated user code in the available processing units. It uses a scheduling algorithm responsible for distributing tasks for parallel execution, transparently running the user code in the heterogeneous hardware.

5.5.3 Integration of user-application and target run-time systems

As described in chapter 4, ParallelME offers a collection-driven programming abstraction developed to allow specific data-types and operations to be executed in high-performance run-times. Consequently, it does not provide means for writing an entire Android application solely with tools present in ParallelME User Library. For this reason, ParallelME Compiler must be able to integrate code that will remain on the Android VM to code that will run in high-performance run-times.

Once the input Java code may have references that are not exclusive to ParallelME User Library elements, this code cannot be simply replaced by a new Java class completely generated by the compiler, as it may have more user-created functions that are not related to ParallelME calls and must be preserved. Consequently, the compiler must interpret the input Java code and integrate those ParallelME calls to the high-performance code in both target run-times.

In order to link RenderScript and ParallelME Run-time to the original user class, the compiler-generated code follows design principles that defines translation rules for integrating these different environments. This integration architecture is shown in Figure 5.5 with all compiler-translated code in the same color as the arrow labeled *translation*. This image illustrates how a user class is modified to incorporate calls for the intermediate layer, which in turn connects the user application to both low-level run-times.

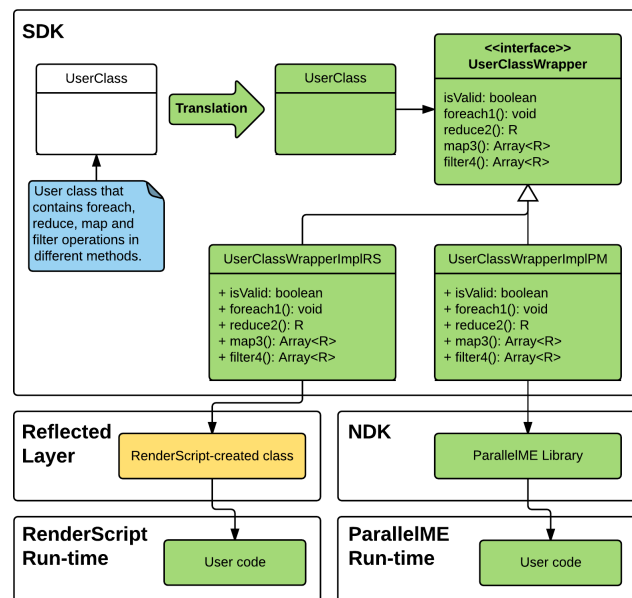


Figure 5.5: Translated code integration architecture

The user application, is connected to run-times through the common interface pre-

sented in section 5.4.2. This interface, shown in Figure 5.5 as *UserClassWrapper*, is used in the translated version of the user class to integrate the user application to both low-level run-times. This translated user class is incremented with a mechanism that dynamically chooses the run-time. In this sense, it instantiates *UserClassWrapperImplRS* for RenderScript or *UserClassWrapperImplPM* for ParallelME Run-time depending on the hardware support for OpenCL. These classes, located at SDK level, integrates with the original user code in each of the low-level run-times through RenderScript’s automatically generated *Reflected Layer* and the ParallelME NDK layer that is created by ParallelME Compiler.

5.6 Translated Code Execution Strategy

The programming abstraction defined by ParallelME User Library was designed to guide developers to produce parallelizable code in a high-level programming model, allowing the compiler to translate this code to predefined execution strategies. However, as stated in section 5.4.1, the user may create code that cannot be parallelized, thus forcing the compiler to translate it to a sequential version in both target run-times. For this reason, the translated code may be sequential or parallel depending on how the user code is developed. Thus, in this section all the details of execution strategies adopted in target run-times are described, showing both sequential algorithms and its respective parallel patterns adopted.



Figure 5.6: Symbols for graph representation

In order to facilitate the understanding of execution strategies, the graphical notation proposed by McCool et al. [2012] is used. In this notation, data, task and dependencies are expressed in a graph that represents an execution overview of a given operation. This graph representation must be read from left to right and from top to bottom. Symbols used to represent graph elements are shown in Figure 5.6.

5.6.1 Foreach and Map

Due to their iteration characteristics, both *foreach* and *map* operations can be expressed using the same **map** pattern as described by McCool et al. [2012]. In this pattern the data collection is processed in a loop where data is processed individually by identical computations.

As it is shown in the sequential graph in Figure 5.7, though the data is computed in individual tasks, the time is unique for each element processed. This approach is only justifiable in cases where there is dependencies among different instances of the loop, like when the same variable is written by two or more of them. On the other side, when the loop body is independent, the time can be compressed and all instances of the loop can be computed in parallel.

Being a pattern in which a set of identical computations are performed on different data without communication, this patterns is also referenced as embarrassing parallelism [McCool et al., 2012].

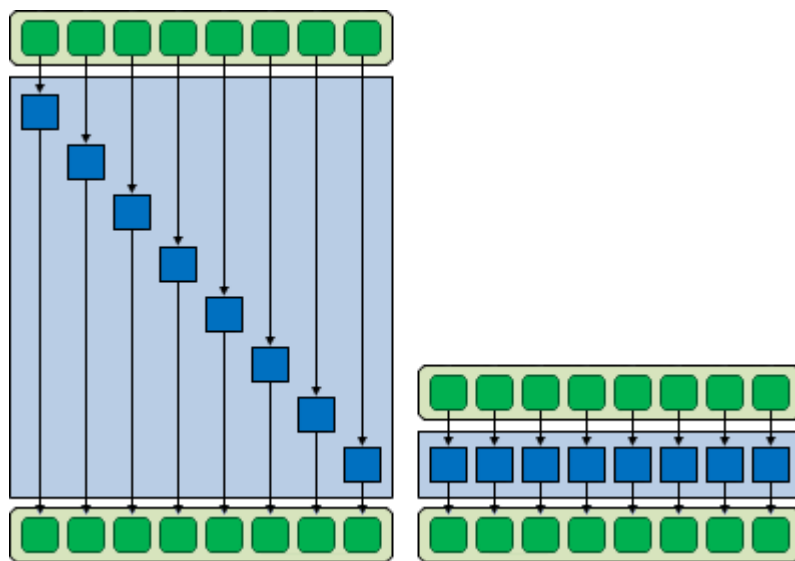


Figure 5.7: Sequential and parallel graphs for Foreach and Map operations

5.6.2 Reduce

The *reduce* operation is expressed using the homonym pattern described by McCool et al. [2012]. This pattern presents a combiner function that is used to combine pairs of elements, being successively applied to the data set until a single summary result is

achieved.

The sequential *reduce* is translated to a loop that performs the combination as shown in Figure 5.8. Once the combine function implies in dependencies from different elements, the parallel version of this operation is structured in a particular way.

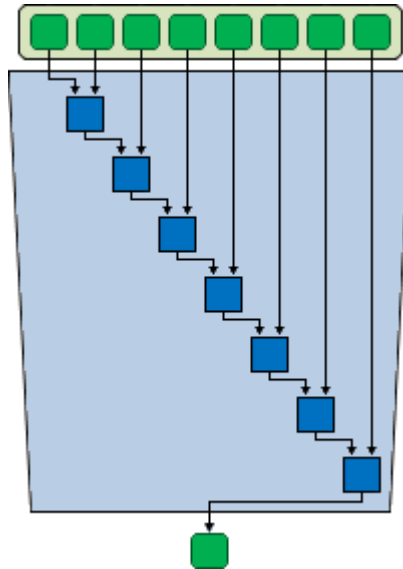


Figure 5.8: Sequential Reduce

Once the *reduce* operation implies in dependencies, the execution strategy adopted for a parallel *reduce* operation requires its division in two levels of computations as shown in Figure 5.9. The first level split the data in groups of the same size, also known as tiles, performing computations of each group in parallel, which in turn sequentially combines the elements to get an intermediate summary result. After these groups are entirely processed, the second level of computation proceeds by sequentially combining the intermediate values, resulting in a single summary value for the entire operation.

In the example shown in Figure 5.9, the data set with 16 elements is divided in 4 groups of 4 elements. The group size is determined differently for User Library classes for image (*BitmapImage* and *HDRImage*) and array (*Array*). For images, the number of groups is defined by the image width, thus each group computes an image column which size is equivalent to the image height. In this sense, the second level of computation sequentially process a *width* number of intermediate results. For *Array* class, the group size is defined as the largest integer less than or equal to the square root of the array length. In cases where the array length is not precisely an integer square root, as 10 for example, the remaining elements in positions greater than the nearest square root are processed sequentially along with the intermediate results. In this sense, for this 10 element array, 9 elements will be processed in 3 parallel groups in the first level of computations, while the remaining element will be processed afterwards in the second level as shown in Figure 5.10.

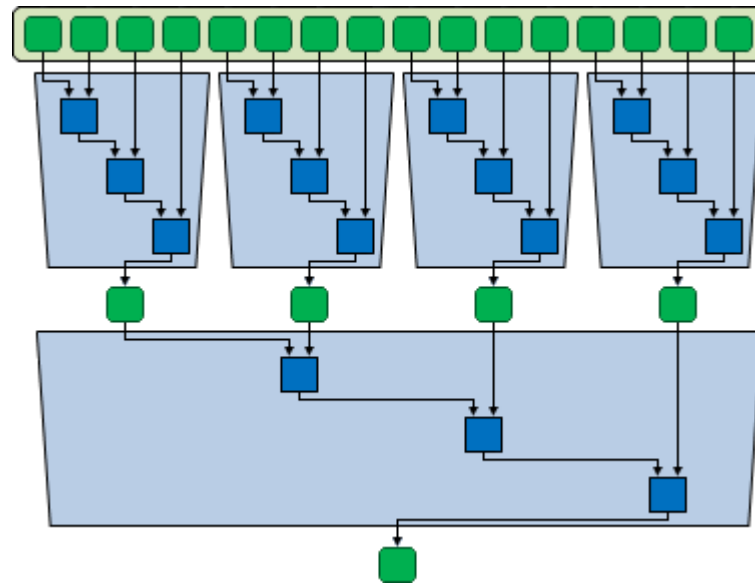


Figure 5.9: Parallel Reduce

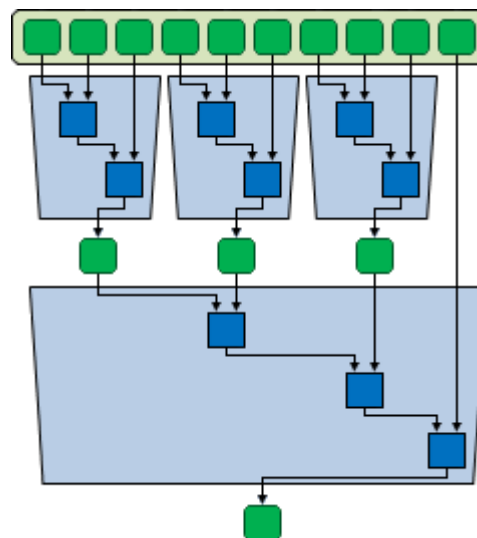


Figure 5.10: Parallel Reduce for Array class

5.6.3 Filter

The *filter* operation consist on the creation of sub-sets of data from a boolean-returning user function, meaning that it may result in an empty set or a collection that is smaller than the initial collection. In this sense, the operation must create a new memory allocation which size will be defined after the user function is evaluated. For this reason, this function is divided in three parts: (i) user function evaluation, (ii) memory allocation and (iii) data copy.

In the sequential version of this operation, shown in Figure 5.11, an intermediate data set with the same size of the initial collection. This intermediate data set is then used to store information about those elements that were evaluated true in the user function. With this intermediate result, it is possible to know what is the size of the output memory necessary to store the operation result, thus the memory is allocated and, finally, those elements which were evaluated true in the user function are copied in the last step of computations.

The parallel version of this algorithm has some similarities with *foreach* and *map* operations in the user function evaluation, as shown in Figure 5.12. Once the intermediate memory allocation is of the same size of the original collection, all elements in the *filter* operation can be evaluated in parallel. After that, the memory is allocated to store the results, being followed by the sequential data copy similarly to the sequential version of this operation. The reason for this last computation being sequential is because filtered elements are kept in the same order of the initial collection, in this way it is required that the elements are copied in its original sequence to the result memory allocation.

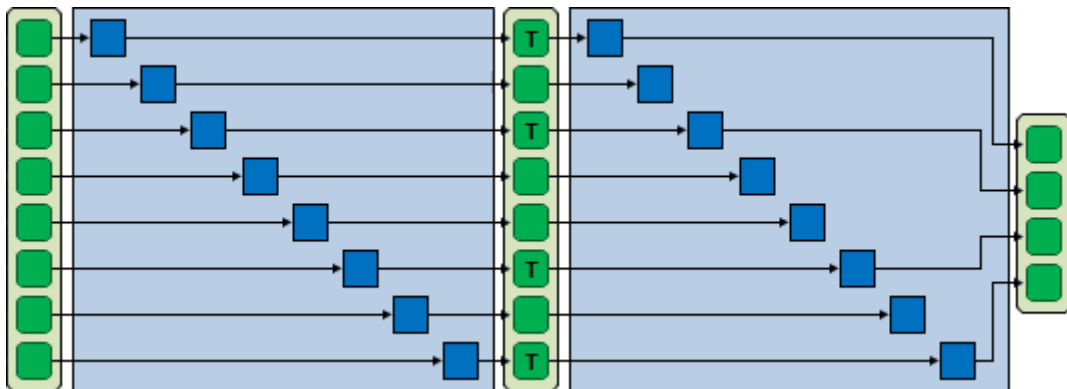


Figure 5.11: Sequential Filter

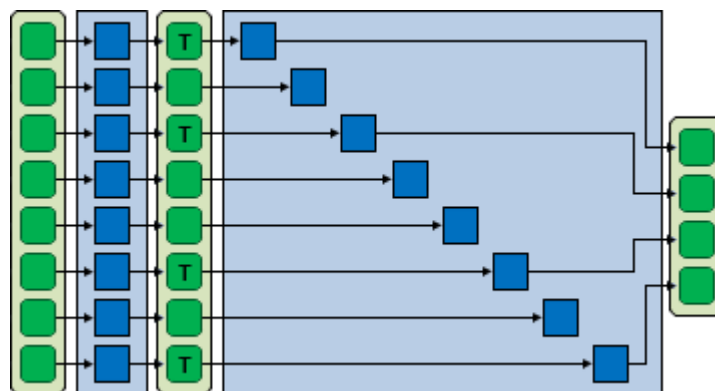


Figure 5.12: Parallel Filter

5.7 Limitations

The current compiler version has some limitations regarding user function code translation. These limitations are related to restricted features of Java to C translation in the compiler back-end. In this sense, the code written by the user inside the user function is not translated to C before being sent to the output file. It means that only C-like code with some restrictions can be written on user function bodies. Thus, the user code provided in this function has some restrictions:

- Only Java primitive types up to 32 bits are allowed, since they are mapped directly to C types during translation;
- The **new** operator is not supported;
- Variables declared outside the user function scope can be used for read and write;
- Variables declared outside the user function scope **without** *final* modifier will imply in sequential code translation for the given operation, even though this variable is not assigned to a new value in the user function;
- Variables declared outside the user function scope **with** *final* modifier will not affect generation of parallel code by the compiler;
- Arrays, even though of primitive types, are **not** supported;
- Strings are **not** supported;
- Method calls are **not** supported;
- Nested user functions are **not** allowed;
- Though lambda expressions (Java 8 feature) are supported in the User Library, they are **not** supported by ParallelME Compiler;
- Variables with names that begin with *PM_* or variables named *x* and *y* are **not** allowed inside user functions. The first limitation is due to temporary variables that are created in translated code, being prefixed with *PM_*. The last two limitations are related to RenderScript, once *x* and *y* are variables used in the Reflected Layer and as parameters in RenderScript's functions.

5.8 Evaluation

The evaluation of the proposed source-to-source compiler was performed using the implementations presented in section 4.5. Execution time, power and energy consumption were evaluated, providing the means for analyzing the effectiveness of the compiler and the quality of the code translated from the proposed high-level programming abstraction.

5.8.1 Execution Time

In order to evaluate execution time, Java routines were used to measure the interval between execution start and finish; the results are shown in Figure 5.13. Due to scale differences, Java application execution time was omitted from this image, averaging from 21 to 30 seconds for 1 image up to 20 images, respectively.

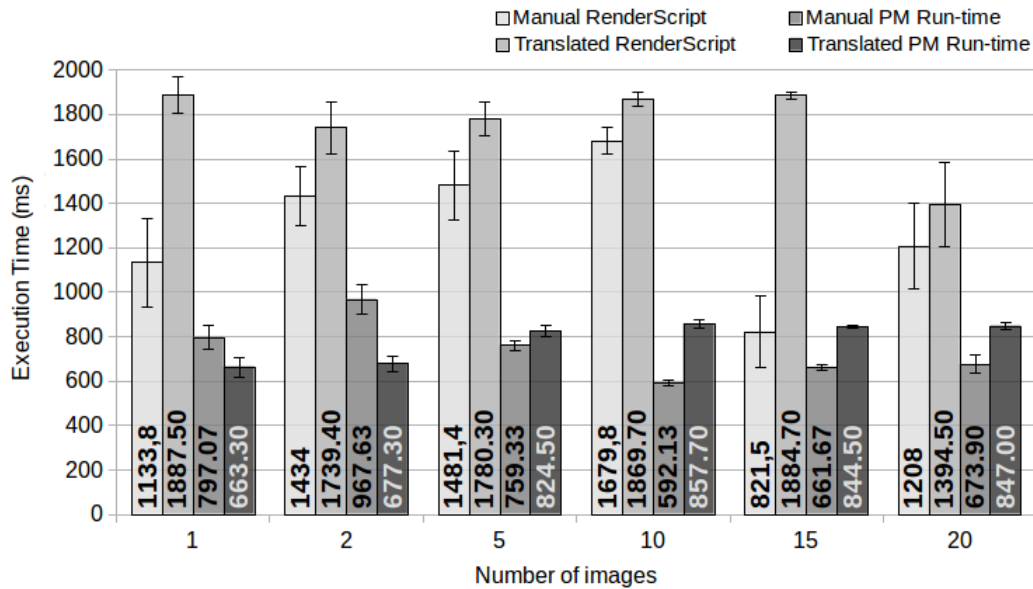


Figure 5.13: Execution time comparison

In general, it is possible to observe that ParallelME Run-time showed significantly higher performance compared to RenderScript. This is easily explained by the fact that ParallelME Run-time uses the processing units present in the mobile architecture in a coordinated way. More specifically, in comparison, the manual RenderScript implementation presented a slightly higher performance compared to the compiler-translated RenderScript code. This is due to the fact that manual RenderScript implementation considered

some programmer-created improvements, such as data buffers optimizations, increasing the parallelism of some operations. Those optimizations are related to the ability of an expert programmer in RenderScript which uses the framework's resources wisely. However, when it comes to automatic generation of RenderScript code performed by ParallelME Compiler, the RenderScript implementation steps were systematized, resulting in some simplifications of the final code. Such systematization cannot capture different forms of optimizations, thus producing a simpler code, with lower performance when compared to a manual implementation created by an expert programmer.

Conversely, when it comes to comparing compiler-translated ParallelME Run-time code and its manually-created equivalent implementation, we observed a performance variability. The considerations highlighted in the previous paragraph, about automatic code generation and manually-created code also apply in this case. However, considering that ParallelME Run-time distributes tasks among processing units through a scheduling strategy, it is possible to assume that the scheduler decision, which is a heuristic algorithm, can accentuate the performance differences observed. Thus, it is possible to conclude that the cases where compiler-translated run-time code showed better performance are a reflection of scheduler's best choices.

Observing the execution time results it is possible to note that there is a small performance loss in the compiler-translated code when compared to the manual implementation. Nonetheless, comparing the programming effort to write code using ParallelME User Library and the effort of manually implementing in a low-level platform like ParallelME Run-time or RenderScript, the small performance difference ends up being a minor issue. As analyzed in section 4.5.2, the effort necessary to write code using ParallelME User Library, measured in lines of code, is 49% lower than ParallelME Run-time, meaning that ParallelME Compiler presents significant results.

5.8.2 Power and Energy Consumption

For power consumption evaluation, it was used a Monsoon Power Monitor configured to 3.9V. The average consumption of the whole system was measured in milliwatts (mW) during the algorithm execution. Additionally, the total energy consumption, which was found by multiplying the average power by the application execution time, is also evaluated and discussed.

Analyzing Figures 5.14 and Table 5.1, it is possible to note that there is not a significant difference between ParallelME Run-time automatically generated and manual implementations when it comes to power and energy consumption. Conversely, implemen-

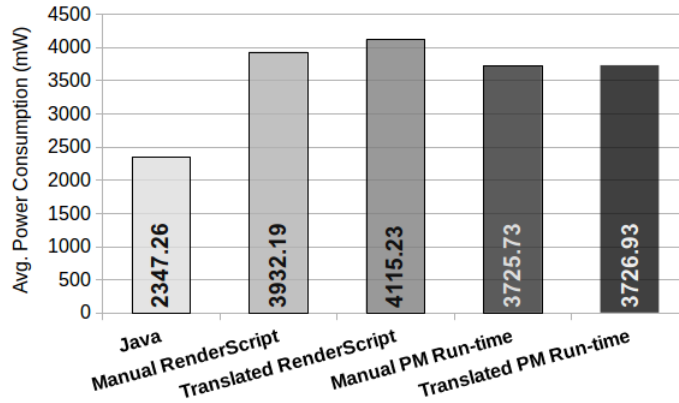


Figure 5.14: Power consumption comparison

	Required Energy (%)
Manual RenderScript	-93.99
Translated RenderScript	-94.57
Manual PM Run-time	-98.12
Translated PM Run-time	-98.03

Table 5.1: Total energy required compared to sequential Java baseline

tations running RenderScript and ParallelME Run-time achieved similar power consumption, while the sequential Java implementation presented lower power consumption. These results show that implementations with higher required power explored more processing resources during execution. Thus, it is possible to conclude that RenderScript and ParallelME Run-time benefited from parallel execution on multi-core CPU and GPU, achieving higher power consumption, while Java implementation executed on a single CPU core, showing lower power consumption.

Furthermore, the total energy required for both ParallelME Run-time applications, shown in Table 5.1, is 98% lower than the energy consumed by the sequential Java baseline. This shows that reducing execution time with a smart resource usage, like in ParallelME Run-time, causes a significant decline in the energy consumed by the application. In that sense, ParallelME User Library and ParallelME Compiler become even more satisfactory, since they are able to combine low programming complexity with high performance and substantial reduction in energy consumption.

5.9 Summary

This chapter presented the proposed source-to-source compiler capable of translating code from the high-level programming model of the User Library to target run-times. ParallelME Compiler was described in depth, from syntactic analysis to the parallelism strategies adopted in translated code. Finally, the low-level code generated automatically by the compiler from ParallelME User Library was evaluated in three different performance aspects (execution time, power and energy consumption), showing that the translated code reached performance levels close to the best implementation created manually by an expert programmer.

Chapter 6

Conclusion and Future Works

Due to a continuous battle for market share, processor vendors have successfully increased processing capacity, reduced energy consumption and kept a regular miniaturization rate in modern chips. Following these constant improvements, current mobile devices incorporated many of the architectural features that were previously exclusive of server and personal computers. Consequently, mobile phones have become powerful devices with different types of processing units, becoming highly complex heterogeneous systems.

Android platform - the dominant mobile operating system in use nowadays - have limited programming tools for developing parallel applications that can benefit from current heterogeneous devices. For this reason, this work presented two important contributions for reducing the complexity of developing parallel applications for that platform. Thus, it was developed ParallelME (Parallel Mobile Engine), an open-source framework for parallel programming in heterogeneous Android devices. The framework was described in detail, focusing on those two contributions that are in the scope of this work: the programming abstraction present in ParallelME User Library and the source-to-source compiler known as ParallelME Compiler.

The primary goal of this work was to introduce a programming model capable of offering users a high-level abstraction for developing parallel applications in heterogeneous mobile architectures. The programming abstraction was built with the incorporation of elements of functional programming and ideas borrowed from the Scala Collection Library. It provides a high-level programming model in Java that can be translated by the source-to-source compiler. The proposed programming model was able to provide a smooth path to parallel programming in heterogeneous mobile devices, allowing developers to use an intuitive collection-driven solution and transparently create high-performance code with the source-to-source compiler.

Both contributions were validated through a performance and coding complexity analysis performed in a test application. The results showed that the code generated automatically by the proposed compiler from the high-level programming model presented a small loss of performance when compared to the best manual implementation possible. On the other hand, the analysis of power and energy consumption contrasted with the programming effort necessary to produce the code presented a significant gain for

compiler-translated code, showing the most energy-efficient strategy with a programming effort equivalent to sequential Java and half of the programming effort compared to a direct low-level implementation. In this sense, the contributions presented in this work proved to facilitate code development, reducing considerably the complexity of developing high-performance code while maintaining a low-level of energy consumption, both highly desirable achievements specially when considering mobile architectures.

6.1 Known Limitations and Future Works

Even though the contributions of this work are comprehensive and proposes an effective programming model for heterogeneous devices, many limitations can be overcome and new features can be added in future works:

- Limitations regarding the programming model, that restrict user-code in User Library collections to be used only with primitive data types should be reviewed. Ideally, the compiler should be ready to translate any Java-compatible code to the low-level frameworks, allowing more complex applications to be developed with the proposed programming model.
- Since ParallelME Run-time is actually an asynchronous task dispatcher, a new programming model created specially for asynchronous or reactive programming, like the Actor Programming Model [Agha, 1986], should be considered.
- Instead of generating code from a high-level programming model, ParallelME Compiler could generate code directly from Java bytecode. Even though this is apparently a more complex task, it could be a path to provide a mixed programming model in ParallelME User Library that is compatible with Java and Scala, since both languages are compiled to bytecode.
- All the OpenCL work developed in ParallelME Run-time is readily compatible with desktop and server platforms. Thus, this work can be extended in order to increase performance and reduce energy consumption on devices other than mobile.
- A distributed version of ParallelME should be considered. For mobile, server or desktop platforms, a cluster could be set-up with the underlying ParallelME Run-time and a new (or derived) programming abstraction could be created to allow the development of more complex solutions. The Spark project [Zaharia et al., 2010] could provide good insights for that work.

Bibliography

- Alejandro Acosta and Francisco Almeida. Performance analysis of Paralldroid generated programs. In *2014 22nd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, pages 60–67. IEEE, 2014.
- Gul Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, 1986.
- AMD. Aparapi github page. <http://aparapi.github.io/>, visited on 10/2016, 2016.
- AMD. Intro OpenCL Tutorial. <http://developer.amd.com/tools-and-sdks/opencl-zone/opencl-resources/introductory-tutorial-to-opencl/>, visited on 10/2016, 2016.
- Guilherme Andrade, Wilson de Carvalho, Renato Utsch, Pedro Caldeira, Alberto Alburquerque, Fabricio Ferracioli, Leonardo Rocha, Michael Frank, Dorgival Guedes, and Renato Ferreira. ParallelME: A Parallel Mobile Engine to Explore Heterogeneity in Mobile Computing Architectures. In *European Conference on Parallel Processing*, pages 447–459. Springer, 2016.
- Apple. OpenCL for macOS. <https://developer.apple.com/opencl/>, visited on 10/2016, 2016.
- Alonzo Church. *The calculi of lambda-conversion*. Princeton University Press, 1941.
- Al Danial. CLOC: counting lines of code. <http://cloc.sourceforge.net/>, visited on 09/2016, 2016.
- EPFL. The Scala Programming Language. <http://www.scala-lang.org/>, visited on 10/2016, 2016.
- Gary Frost. Aparapi: Using gpu/apus to accelerate java workloads, 2014.
- Nasser Giacaman, Oliver Sinnen, 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.
- Google. Android Open Source Project. <https://source.android.com/>, visited on 10/2016, 2016a.

- Google. Android Platform Architecture. <https://developer.android.com/guide/platform/index.html/>, visited on 10/2016, 2016b.
- Google. Java 8 Language Features. <https://developer.android.com/preview/j8-jack.html>, visited on 09/2016, 2016c.
- Google. RenderScript. <http://developer.android.com/guide/topics/renderscript>, visited on 09/2016, 2016d.
- Google. RenderScript Overview. <https://stuff.mit.edu/afs/sipb/project/android/docs/guide/topics/renderscript/compute.html#overview>, visited on 09/2016, 2016e.
- Krishan Gopal Gupta, Nisha Agrawal, and Samrit Kumar Maity. Performance analysis between aparapi (a parallel API) and JAVA by implementing sobel edge detection Algorithm. In *Parallel Computing Technologies (PARCOMPTECH), 2013 National Conference on*, pages 1–5. IEEE, 2013.
- Intel. Intel SDK for OpenCL Applications. <https://software.intel.com/en-us/intel-opencl/>, visited on 10/2016, 2016.
- International Data Corporation. Smartphone OS Market Share, 2016 Q2. <http://www.idc.com/prodserv/smartphone-os-market-share.jsp>, visited on 10/2016, 2016.
- Roelof Kemp, Nicholas Palmer, Thilo Kielmann, Henri E. Bal, Bastiaan Aarts, and Anwar M. Ghuloum. Using renderscript and RCUDA for compute intensive tasks on mobile devices: a case study. In *Software Engineering 2013 - Workshopband*, 2013.
- Khronos. OpenCL by khronos group. <https://www.khronos.org/opencl/>, visited on 09/2016, 2016.
- John McCarthy. Recursive functions of symbolic expressions and their computation by machine, part i. *Commun. ACM*, 3(4):184–195, apr 1960. ISSN 0001-0782. doi: 10.1145/367177.367199. URL <https://doi.org/10.1145/367177.367199>.
- Michael McCool, James Reinders, and Arch Robison. *Structured Parallel Programming: Patterns for Efficient Computation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2012. ISBN 9780123914439, 9780124159938.
- Gordon E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8), April 1965.
- NVIDIA. CUDA Parallel Computing Platform. <http://www.nvidia.com/cuda/>, visited on 10/2016, 2016a.

- NVIDIA. NVIDIA OpenCL SDK. <https://developer.nvidia.com/opencv/>, visited on 10/2016, 2016b.
- Martin Odersky and Tiark Rompf. Unifying functional and object-oriented programming with scala. *Commun. ACM*, 57(4):76–86, April 2014. ISSN 0001-0782. doi: 10.1145/2591013. URL <http://doi.acm.org/10.1145/2591013>.
- Martin Odersky, Lex Spoon, and Bill Venner. *Programming in Scala: A Comprehensive Step-by-Step Guide, 2Nd Edition*. Artima Incorporation, USA, 2nd edition, 2011.
- Oracle. Java Native Interface Specification. <http://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/jniTOC.html>, visited on 09/2016, 2016a.
- Oracle. Oracle Java Documentation. <https://docs.oracle.com/javase/tutorial/java/java00/lambdaexpressions.html>, visited on 09/2016, 2016b.
- Jeff Parkhurst, John Darringer, and Bill Grundmann. From single core to multi-core: Preparing for a new exponential. In *Proceedings of the 2006 IEEE/ACM International Conference on Computer-aided Design, ICCAD '06*, pages 67–72, New York, NY, USA, 2006. ACM. ISBN 1-59593-389-1.
- Terence Parr. *The definitive ANTLR 4 reference*. Pragmatic Bookshelf, 2nd edition, 2013.
- Terence Parr. ANTLR - Parser Generator. <http://www.antlr.org/>, visited on 09/2016, 2016.
- Philip C Pratt-Szeliga, James W Fawcett, and Roy D Welch. Rootbeer: Seamlessly using GPUs from Java. In *High Performance Computing and Communication & 2012 IEEE 9th International Conference on Embedded Software and Systems (HPCC-ICES), 2012 IEEE 14th International Conference on*, pages 375–380. IEEE, 2012.
- Aleksandar Prokopec, Phil Bagwell, Tiark Rompf, and Martin Odersky. A Generic Parallel Collection Framework. In *Proceedings of the 17th International Conference on Parallel Processing - Volume Part II, EUROPAR 11*, 2011.
- Erik Reinhard, Michael Stark, Peter Shirley, and James Ferwerda. Photographic tone reproduction for digital images. In *ACM Transactions on Graphics (TOG)*, volume 21. ACM, 2002.
- Guohui Wang, Yingen Xiong, Jay Yun, and Joseph R Cavallaro. Accelerating computer vision algorithms using opencl framework on the mobile gpu-a case study. In *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*, pages 2629–2633. IEEE, 2013.

Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. In *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'10, pages 10–10. USENIX Association, 2010.

Appendix A

Run-time

The run-time component of ParallelME 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 when deciding in which PU a given task should run during execution, creating a novel level of control and flexibility that can be explored in order to achieve certain goals for improving overall performance. ParallelME run-time is responsible for coordinating, in an efficient way, all processing units available in the mobile architecture. It organizes and manages low level tasks generated by ParallelME Compiler, from definitions expressed by developers in the User Library. It was developed in C++ and integrated in Android using the NDK toolkit.

OpenCL was adopted as a low-level parallel platform to manipulate available processing units. In general, the dynamics of the run-time involves the following phases: (1) Identify the computing resources available in the mobile architecture; (2) Create tasks with their input and output parameters; (3) Arrange task's data accordingly to their parameters; (4) Submit tasks for execution; (5) Instantiate scheduling policy routines and; (6) Assign tasks to processing units defined by the scheduling policy. The first four phases correspond to user API phases and must be performed to initialize the run-time system and its tasks. The run-time core engine is composed of the two last phases. Figure A.1 gives an overall picture of the entire framework, which is further described below.

A.1 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

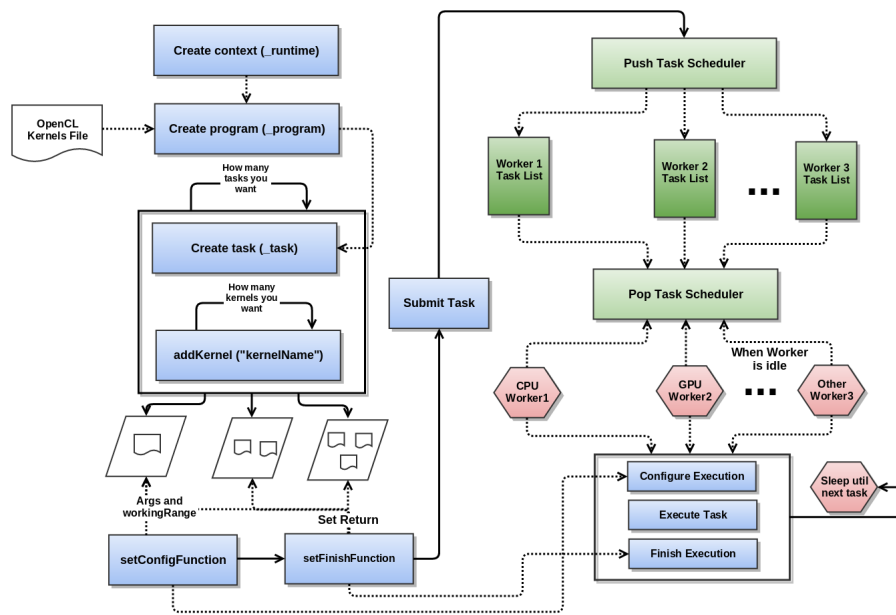


Figure A.1: ParallelME Run-time Execution Flow

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.

A.2 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.