ParallelME

Reference Manual for ParallelME framework.

September, 2016

ParallelME: Parallel Mobile Engine

Contents

1 Overview **3**
1.1 System Architecture . . . 4

2 User-library **7**
2.1 Programming Abstraction . . . 7
2.2 Data binding . . . 9
2.3 Operations . . . 10
2.3.1 Foreach . . . 10
2.3.2 Map . . . 11
2.3.3 Reduce . . . 12
2.3.4 Filter . . . 12
2.4 Debugging Support . . . 13

3 Run-time **14**
3.1 Run-time API Phases . . . 15
3.2 Execution Engine . . . 16

4 Compiler **16**
4.1 First Pass . . . 17
4.2 Second Pass . . . 19
4.3 Translation Steps . . . 19
4.3.1 Execution type analysis . . . 20
4.3.2 Definition of a common interface for low-level run-times . . . 21
4.3.3 Run-time specific translation . . . 22
4.3.4 Original user-code translation . . . 23
4.4 Translated Code Integration Architecture . . . 24
4.5 Translated Code Execution Strategy . . . 25
4.5.1 Foreach and Map . . . 26
4.5.2 Reduce . . . 26
4.5.3 Filter . . . 28

5 How to use **30**
5.1 ParallelME Compiler . . . 30
5.2 ParallelME User-library . . . 30
5.3 Limitations . . . 31

1 Overview

The development of new technologies is setting a new era, characterized, among other factors, by the emergence of sophisticated mobile devices. Current models include an extensive range of sensors that are capable of collecting a wide variety of user and environment data. The constant growth of data, coupled with more sophisticated applications, has pushed for significant advances in mobile computing architectures, increasing processing capacity, while also maintaining energy consumption at a reasonable level. Currently, many mobile phones have followed the same trend of desktop architectures, adopting different types of processing units (PUs), becoming so-called heterogeneous systems. These new mobile phones include multi-core CPUs, such as Qualcomm Snapdragon, NVIDIA Tegra 3, as well as other special purpose processors — GPUs being a favorite among them — such as ARM Mali and Qualcomm Adreno.

In this new context, it is expected that applications of different domains achieve higher performance levels, and hence have a better user experience, by exploring, in a coordinated and efficient way, all the available PUs by taking full advantage of their processing capabilities. Thus, different parallel frameworks have been proposed to provide some degree of abstraction to make the use of different PUs simpler for mobile application developers. Among those frameworks in the mobile operating system (Mobile OS) Android, it is important to highlight RenderScript [1] and OpenCL [2]. Those two 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, preventing their popularization among mobile developers.

In order to address the main weaknesses of these frameworks, ParallelME [3], a Parallel Mobile Engine 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. This framework was developed as a result of a LGE research project conducted by the Department of Computer Science of UFMG from 2015 to 2016. The primary goal of the research effort was to reduce the complexity of high-performance code development for Android devices, addressing the main weaknesses of RenderScript and OpenCL.

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. That is achieved through a source-to-source compiler, that translates the high-level operations in Java to a high-performance representation, creating low-level tasks that are coordinated during execution by both platforms RenderScript and the OpenCL-based ParallelME run-time. Those platforms are chosen dynamically during user application run-time, depending on the hardware and software resources available on the target mobile system. This three-layer structure, with an intuitive programming abstraction, a versatile source-to-source compiler, and a powerful run-time environment is the key advantage of ParallelME, combining a wide support of target devices, reduced programming effort, and efficient coordinated usage of the available resources.

In order to facilitate the understanding of the framework structure, the relation between its components is illustrated in figure 1. This image shows how ParallelME modules relate to each other and how the user interacts with the framework.

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 [4]

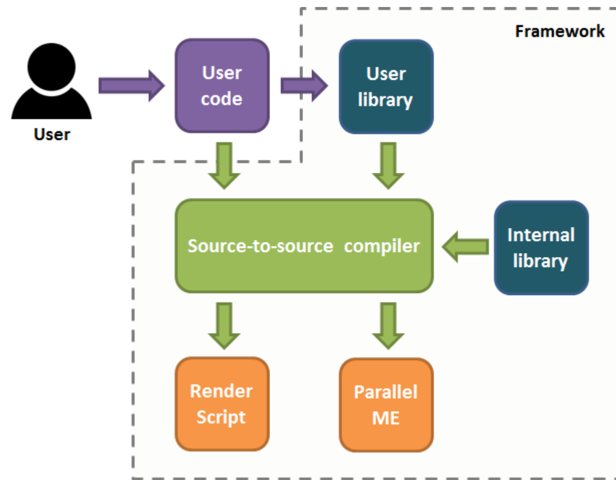

Figure 1: ParallelME overview

and is designed to provide an easy-to-use and generic programming model for parallel applications in Java for Android. This programming abstraction offers a collection-driven approach for code development of specific types of data sets, also introducing special functions to iterate, reduce, perform data-transformation and create sub-sets in these collections. These operations are executed, after translated by ParallelME compiler, in high-performance parallel run-times at NDK level. Once 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 user applications can be debugged at SDK level in Java using the regular development infrastructure.

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 session.

ParalelME source-to-source compiler provides a mechanism for translating user code to a low-level parallel implementation in the specified high-performance run-times. It takes as an input Java code written with the user-library and translates it to a new version integrated with both RenderScript and ParallelME run-time. This translation is performed during development time, leaving the choice of which target run-time to execute to the user application execution time. The output code generated by ParallelME will evaluate during user application execution if the hardware supports OpenCL and perform execution of high-performance code in ParallelME run-time. In case there is no support for OpenCL, ParallelME will transparently switch to RenderScript, which is supported by a greater range of Android devices

1.1 System Architecture

ParallelME is a framework centralized in code translation from a high-level programming abstraction in Java to high-performance run-times in C and C++. Such run-times, being RenderScript and ParallelME run-time, are located outside the Android VM, thus they must be integrated to the

user application. For this reason, ParallelME follow well-defined architectural guidelines in order to translate and integrate code, providing transparent execution and memory handling for user applications in high-performance environments.

As ParallelME run-time and RenderScript 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* allowed 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.

In order to comply with RenderScript, *ParallelME Java Layer* wraps RenderScript's automatically created *Reflected Layer*. *ParallelME Java Layer* is shown in Figure 2, being this image similar to that presented in the original RenderScript documentation [5]. 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, being composed of a *RenderScript Code* layer and a *Graphics and Compute Engine* layer. The *RenderScript Code* layer is composed of C99 code that represents the application user-defined behaviour. In ParallelME, this layer is created by the source-to-source compiler in order to store those definitions created at the high-level programming abstraction. Once compiled by RenderScript framework, *RenderScript Code* is then 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 [1], it does not create new memory allocations at run-time level, thus requiring that segments of memory handled by user applications are always allocated at Android VM level in Java.

As ParallelME run-time is entirely implemented at Android NDK level, thus working outside the Android VM, it has a simplified stack at Android framework level. In this sense, the *ParallelME Java Layer* is used to integrate the Android application code with this low-level run-time, as shown in Figure 3. In the same way as in the integration with RenderScript's *Reflected Layer*, *ParallelME Java Layer* plays an important role performing memory allocation, reads, writes and binding with the low-level platform. ParallelME run-time, in turn, is composed of three layers: a *JNI Layer*, a *Kernel and user code* layer and, finally, the *heterogeneous scheduling engine* at the bottom layer.

ParallelME JNI Layer corresponds to the translated code that is adherent to the Java Native Interface (JNI) [6] specification. This layer allows the interoperability between the Java code that executes in the Android VM and the C and C++ codes that executes at native level. These native codes compose the kernel of ParallelME run-time and the translated user code.

Kernel and user code layer is composed of predefined functions that compose the run-time kernel and allows the execution of the user code that is translated by ParallelME compiler from the high-level programming abstraction in Java. This layer is ultimately responsible for providing transparent access to the low-level heterogeneous scheduling engine at the bottom layer.

Finally, *ParallelME Heterogeneous Scheduling Engine* is responsible for executing the translated user code in the available processing units. In order to accomplish that, this layer is composed of tasks that are distributed for parallel execution in the heterogeneous hardware using a scheduling algorithm.

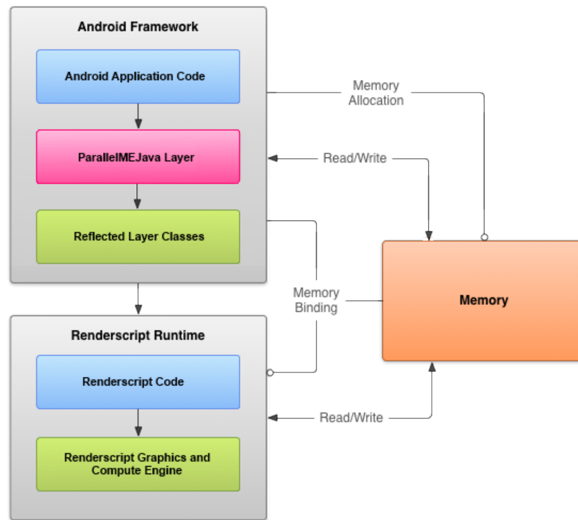

Figure 2: RenderScript integration architecture

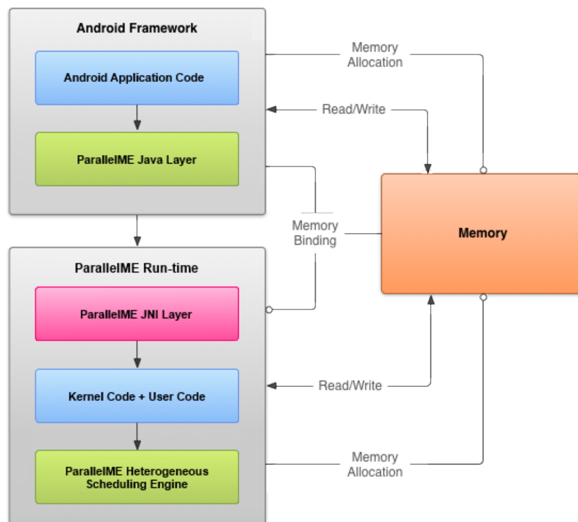

Figure 3: ParallelME run-time integration architecture

2 User-library

The user-library corresponds to ParallelME's front-end interface, establishing a bridge between user application code and the rest of the framework's components. Designed to provide an intuitive programming abstraction, the user-library is composed of a set of classes that define the data-structure oriented model provided by ParallelME. This programming model was conceived in order to introduce the minimum amount of new concepts in users' applications, keeping the programming abstraction similar to what sequential programmers expect, hiding all the complexity of low-level parallel operations from its users. Its API also serves to guide and syntactically restrict users into what is allowed by the framework, reducing the complexity of the analysis performed by ParallelME compiler during the translation to low-level code.

2.1 Programming Abstraction

The programming abstraction proposed in the user-library was inspired by ideas found in the Scala Collection Framework [4], providing an easy-to-use and generic programming model for creating parallel applications in Android. In the same way as proposed in the Scala Collection Framework, ParallelME User-Library was designed to provide 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 was 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 [7; 8; 9; 10], 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[4] was analyzed in depth and many of its concepts were borrowed.

With a data-driven generic programming model, the Scala collection library adopts a smooth approach for parallelism that allows the creation of parallel operations by simply inserting a method call. This simple and effective approach eliminates the necessity of structural changes in user application, thus reducing significantly the complexity of developing applications that can use the popular multi-core architecture present in most of computer devices existing nowadays. An example of the simplicity of Scala collection library approach on parallelism is shown in Listing 1. 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 1: Parallelization feature in the Scala collection library

The Scala code presented in Listing 1 can be easily parallelized since it does not include any external dependencies or state changes that could incur in concurrency issues. This approach, which drives users to pursue data immutability, is necessary when using functional programming.

Ultimately, codes produced without mutable-state structures can be easily transformed into parallel code as in the example. Being fully implemented in Java to be easily integrated into the Android SDK, ParallelME user-library incorporates some concepts of data-immutability and uses a similar approach as presented by the Scala collection library for operating collections. Besides Java choice can be easily justified by its integration with Android Development Tools, Java is currently one of the most popular languages worldwide and its usage allows user-library to reach a higher number of users, since ParallelME may have future extensions for desktop systems.

In order to suppress the debugging shortcomings found in other high-performance Android frameworks [7; 8; 9; 10; 1; 2], the user-library provides a sequential implementation that can be entirely executed at Android SDK level in Java. This sequential implementation is intended to be used only for debugging purposes, allowing applications to be easily debugged and validated with more sophisticated tools present at SDK level, which are those tools used to debug regular applications. Once the code is validated, the user must simply add a *par()* method call, similarly to Scala, informing ParallelME compiler that the code should 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 mentioning the current version supported by Android (Java 7) which does not support an important element for code simplification: lambda expressions. Lambda expressions allows functionality to be passed as an argument, creating a simple code where a more complex structure should be created to support it. Still referencing the example code in Listing 1, the code *v =¿ 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. To exemplify the functionality transmission feature, the example code in Listing 2 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 2: Functionality transmission in user-library

The restricted expressiveness of Java 7, compared to Scala, is evident when analyzing the differences of codes in Listing 2 and Listing 1, being Java clearly more verbose. The most recent version of Android (Marshmallow) was released with full support for Java 8 [11], meaning that lambda expressions are now available in the Android development environment, helping 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 created in Java 8 by the instantiation of an anonymous class with a single method implementation [12], reflecting exactly the structure defined for functionality transmission in the user-library. The code shown in Listing 2, where the expression that starts with the *Map* object creation in line 1 with the subsequent method implementation is an expanded version of a lambda expression. Code in Listing 3 shows the same functionality 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 3: 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 High Dynamic Range (HDR) types. These classes integrates to the regular Android API through functions for data input and output, allowing users to move data to and from ParallelME. Also, these classes allow the development of complex tasks from 4 basic operations: filter (sub-setting), foreach (iteration), map (data-transformation) and reduce (reduction), following the same working principle and being applied in the user code as an anonymous class followed by an implementation of a method named *function* which will ultimately store the user code, as exemplified in Listing 2.

2.2 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. 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[1], memory must always be allocated at SDK level and than bind to the user kernel in C. Though this is not a restriction to ParallelME run-time, since memory may also be allocated at NDK level, the user-library must provide a unique programming interface for both low-level frameworks. Thus, ParallelME user-library requires that memory is allocated by users at SDK level, providing previously-allocated data structures in input-bind operations. Thus, input-bind operations are always defined in constructors, where users must provide an object during user-library collections instantiation with the data that will handled by the collection, like shown in line 6 of Listing 4.

Though Renderscript restriction in memory allocation is the same for output-bind operations, the user-library provides functions that allocate at SDK level the necessary memory before calling

```
int[] data = new int[42];
// Initializing input data with some valid values
for (int i=0; i<data.length; i++) {
    data[i] = i;
}
Array<Int32> array = new Array<Int32>(data, Int32.class); // Input-bind
array.par().foreach(...); // Some operation
int[] result1 = array.toJavaArray(); // Output-bind type 1
int[] result2 = new int[42];
array.toJavaArray(result2); // Output-bind type 2
```

Listing 4: Input and output data binding in ParallelME user-library

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 4. Assignment in line 8 automatically creates the memory allocation for a given object at SDK level, being this task executed by the user-library. On the other hand, the assignment in line 10 demands that the necessary memory to store the collection data is allocated by the user. 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 JDK, otherwise users may face execution errors.

2.3 Operations

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 responsible for providing such features are respectively *foreach*, *map*, *filter* and *reduce*. They follow the form $A \overset{f}{\mapsto} B$, where 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.

2.3.1 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 by the following equation, where A is the collection and f is the user code:

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

Two code examples are provided in Listing 5, 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.

```java
array.par().foreach(new Foreach<T>() {
    @Override
    public void function(T element) {
        // User code
        ...
    }
});
```

```java
array.par().foreach(element -> {
    // User code
    ...
});
```

Listing 5: Foreach operation compliant with Java 7 (left) and Java 8 (right)

2.3.2 Map

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

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

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 where its elements are the result of each individual user function call during the iteration of the original collection. In this sense, all changes performed on a given element are discarded after the user function execution, unless it is 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 6 shows two abstract types in line 1 and 3: R and T. The R type corresponds to the user function return type, while T corresponds to the current collection element type. These types may be different when the operation is used to perform data-transformation.

```java
Array<R> result = array.par().map(R.class, new Map<R, T>() {
    @Override
    public R function(T element) {
        // User code
        ...
        return ...;
    }
});
```

Listing 6: Map operation implemented with anonymous class (Java 7)

Listing 7 shows the equivalent code presented in Listing 6, 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 7: Map operation implemented with lambda expression (Java 8)

2.3.3 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 it may had inside the function are used in the next iteration.

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

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

Listing 8 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.

2.3.4 Filter

The *filter* operation provides the means for creating sub-sets. In this sense, the user function decides the criteria that will be used to filter elements of a given collection. For 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.

```java
T result = array.par().reduce(new
    Reduce<T>() {
    @Override
    public T function(T elem1, T elem2)
        {
        // User code
        ...
        return ...;
    }
});
```

```java
T result = array.par().reduce((elem1,
    elem2) -> {
    // User code
    ...
    return ...;
});
```

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

Filter 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.

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

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

Listing 9 shows an implementation with an anonymous class in Java 7, while Listing 10 shows its equivalent version with a lambda expression in Java 8. The abstract type T represents the collection element type, which is also resulting collection element type associated to the *result* variable. In this code, the user function returns a boolean value. In this sense, all elements returning true in the user function call will be added to the resulting collection and all those which returned false will be discarded.

```java
Array<T> result = array.par().filter(new Filter<T>() {
    @Override
    public boolean function(T element) {
        // User code
        ...
        return ...; // boolean return
    }
});
```

Listing 9: Filter operation implemented with anonymous class (Java 7)

2.4 Debugging Support

One of the greatest limitations of low-level code development in Android is the restricted debugging support. As previously mentioned, one of the key goals of ParallelME was to enable code debugging

```java
Array<T> result = array.par().filter(element -> {
    // User code
    ...
    return ...; // boolean return
});
```

Listing 10: Filter operation implemented with lambda expression (Java 8)

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 generated 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 11.

```java
array.foreach(new Foreach<T>() {
    @Override
    public void function(T element) {
        // User code
        ...
    }
});
```

```java
array.foreach(element -> {
    // User code
    ...
});
```

Listing 11: 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 to mimic the parallel code that is created by ParallelME compiler. In this sense, it contains some features that were created exclusively to maintain functionality compatibility, like for example the (x, y) coordinates for pixels in images. Also, as the sequential implementation was created 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.

3 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

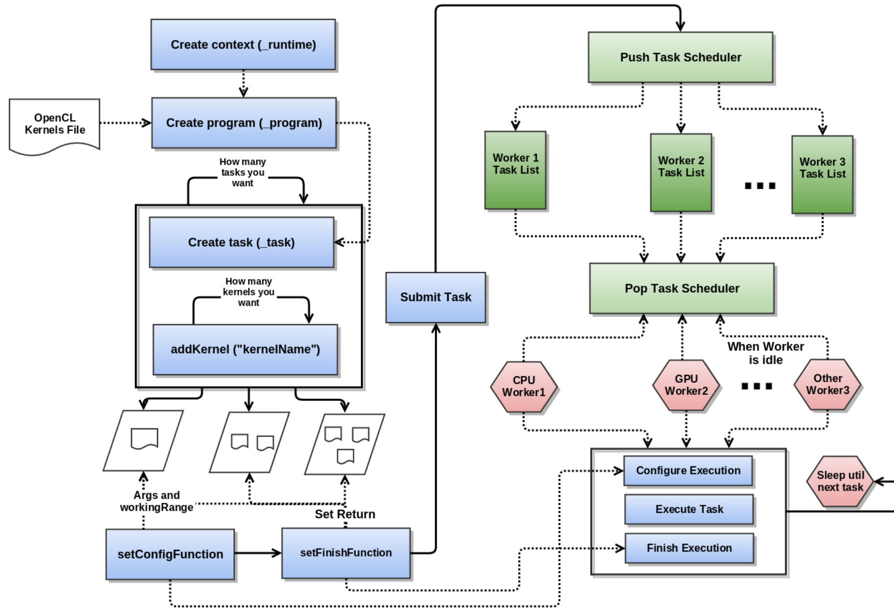

Figure 4: ParallelME Run-time Execution Flow

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 4 gives an overall picture of the entire framework, which is further described below.

3.1 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 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.

3.2 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.

4 Compiler

ParallelME source-to-source compiler was developed with the incorporation of ANTLR, a powerful parser generator [13] 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 to user application execution. In order to do that, the user code provided in user-library operations is translated to C code compatible with each low-level run-time, being integrated to the Java application through an intermediary layer that is created to wrap both low-level run-times, including a mechanism capable of choosing the appropriate run-time during application execution.

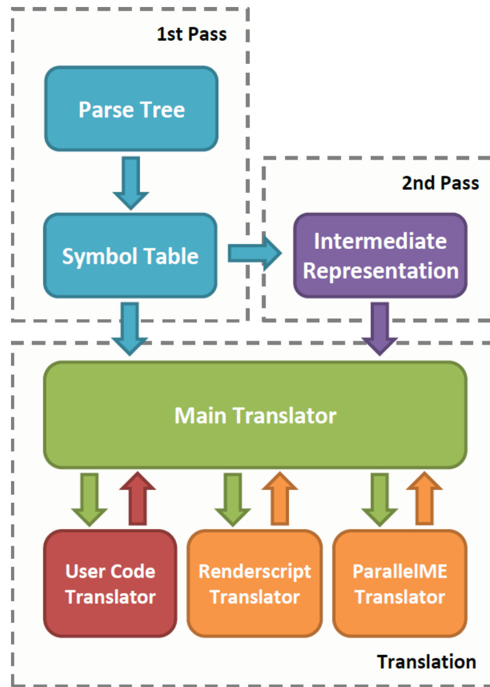

Figure 5: Compiler overview

ParallelME compiler was designed to completely decouple run-time specific features from code translation tasks as shown in figure 5, 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, create RenderScript and ParallelME run-time implementations and integrate all the translated code with the original user class in Java.

4.1 First Pass

The compiler's first pass is responsible for lexical and syntax analysis, creating the symbol table in the end of its execution. It was developed with ANLTR (**AN**other **T**ool for **L**anguage **R**ecognition), a powerful parser generator for reading, processing, executing, or translating structured text or binary files [14] [13]. It is widely used to build languages, tools and frameworks, providing the means for automatically generating parsers from a given grammar definition.

For ParallelME compiler, a Java grammar was applied to ANTLR in order to generate the Java parsing infrastructure, creating classes that are used by the compiler to build and walk parse trees. Such infrastructure is composed of specific classes inherited from those automatically generated by ANTLR, which are used 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 12 is a declaration of a user-library object and Figure 6 shows its equivalent parse tree.

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

Listing 12: User-library object declaration

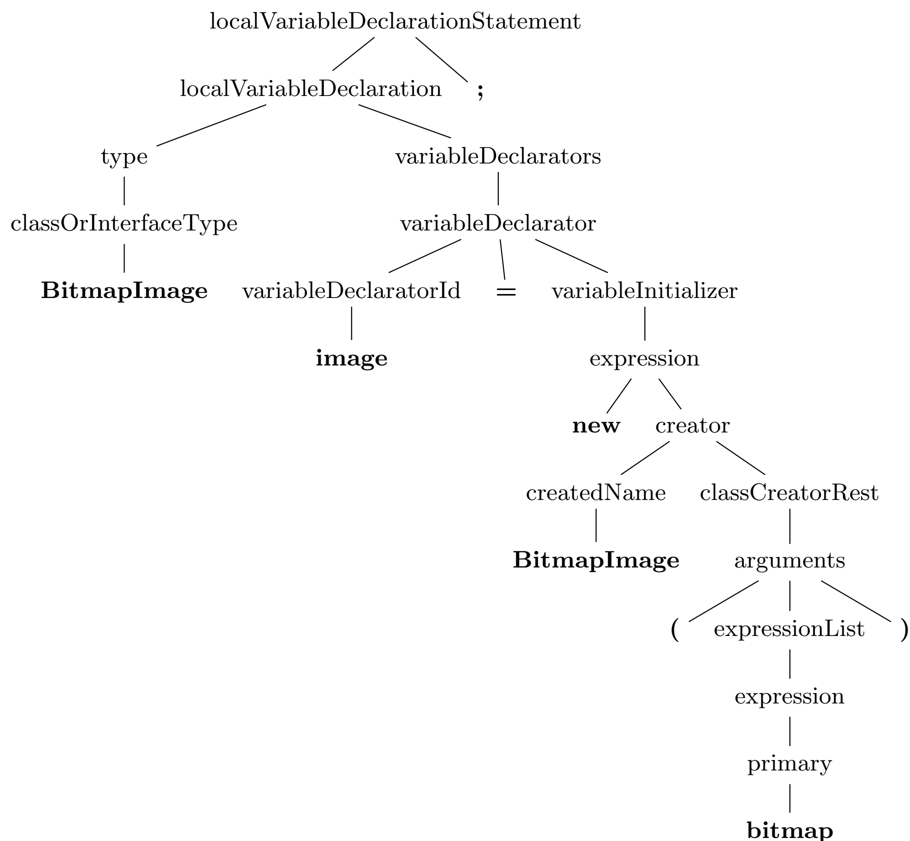

Figure 6: Parse tree for code presented in listing 12

The symbol table conceived for ParallelME compiler was designed as an hierarchical and scope-driven structure, meaning that all symbols stored on the table are in the same hierarchical scope as in the original input code. This allows that the necessary scope-level analysis that is performed during the second pass occurs with a reasonable computing cost.

All operations related to the symbol table creation are implemented in *ScopeDrivenListener* 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 walk 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 show in listing 13.

```
<RootSymbol> $
  <ClassSymbol> UserClass
    <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 13: Scope-driven symbol table

4.2 Second Pass

The second pass is responsible for a deeper analysis on the user code. During this phase, the compiler will perform another evaluation on 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 necessary the intermediate representation that will be used for code translation to both target run-times.

The intermediate representation was designed to represent, besides variables, literals and other features, those abstract tasks that describes in a high level how the programming abstraction works. Such tasks are input binds, operations and output binds calls performed in user-library classes. They represent, respectively, the transfer of data from Java environment to low-level run-times, user operations that are performed in the collection data and, lastly, the retrieval of data which was processed in low-level run-times.

After all data necessary for building the in-memory intermediate representation is collected in the second pass, it is sent to appropriate run-time translators that are responsible for creating the equivalent low-level code. Listing 14 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 15, 16 and 17.

4.3 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 5 sequential steps.

```java
class UserClass {
    public Bitmap someMethod(Bitmap inputBitmap) {
        BitmapImage image = new BitmapImage(inputBitmap); // Input-bind
        image.par().foreach(new Foreach<Pixel>() { // Operation
            @Override
            public void function(Pixel pixel) {
                pixel.rgba.red += 1;
                pixel.rgba.green += 2;
                pixel.rgba.blue += 3;
            }
        });
        Bitmap returnBitmap = image.toBitmap(); // Output-bind
        return returnBitmap;
    }
}
```

Listing 14: Input bind, operation and output bind at user-library level

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

Listing 15: Intermediate representation for input-bind

These steps are presented in the following sections, being described in the same order as they are executed by ParallelME compiler.

4.3.1 Execution type analysis

The execution type analysis 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 execute sequentially. This step ensures that the user code does not include references for variables of other scopes that may create concurrency issues, thus invalidating parallel execution. In this sense, the user code provided is checked for every operation and their execution type is marked as sequential or parallel so next compiler steps can perform a proper translation.

In current version of ParallelME compiler, the following rule is considered to evaluate an operation execution type: 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.

```
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 16: Intermediate representation for Foreach operation

User code that references final variables from other scopes or does not contain any reference for variables of other scopes is considered code that can be 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' analysis. Though this analysis 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 considered in current version of ParallelME compiler in order to reduce the complexity of the semantic analysis.

The fragment code presented in listing 18 contains a reference for a non-final class variable (*sum*) in user code. This variable is assigned to a new value in line 5, meaning that if this code is executed in parallel, concurrent reads and writes may cause inconsistencies on the variable result. In this case, unless the code is executed sequentially or a critical section is created, the variable value is unpredictable in the end of the operation. For cases like this, 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 a parallel code.

4.3.2 Definition of a common interface for low-level run-times

In order to create the mechanism for dynamic selection of low-level 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. Once this contract will be used at user application level, it is defined as a Java interface that is created by ParallelME compiler with the information from the intermediate representation.

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

Listing 17: Intermediate representation for output-bind

```
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 18: User code referencing a non-final variable outside of its scope

The communication between user application and low-level 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 used to be handled by the dynamic run-time selection mechanism.

An example interface is show in Listing 19. It was created based on the code shown in Listing 14, 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 second pass and is assigned to each element in the intermediate representation, being independent for input-binds, operations and output-binds. In this sense, 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.

One interface is created for each user class that references user-library objects, being named after the original class name with and additional *Wrapper* suffix. Also, 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.

4.3.3 Run-time specific translation

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

```java
public interface UserClassWrapper {
    boolean isValid();

    void inputBind1(Bitmap bitmap);

    void foreach1();

    Bitmap outputBind1();
}
```

Listing 19: Interface for integrating user application and low-level run-times

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 low-level implementation based on the existing parameters of the intermediate representation.

4.3.4 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 14 shown in Listing 20.

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

```
class UserClass {
    private UserClassWrapper PM_parallelME; // Common interface reference

    public UserClass(RenderScript PM_mRS) { // New constructor
        this.PM_parallelME = new UserClassWrapperImplPM();
        if (!this.PM_parallelME.isValid())
            this.PM_parallelME = new UserClassWrapperImplRS(PM_mRS);
        }

    public Bitmap someMethod(Bitmap inputBitmap) {
        PM_parallelME.inputBind(inputBitmap); // Input-bind
        PM_parallelME.foreach(); // Operation
        Bitmap returnBitmap = PM_parallelME.outputBind(); // Output-bind
        return returnBitmap;
    }
}
```

Listing 20: Compiler-translated version of code presented in Listing 14

portions of code that were transferred to high-performance run-times, thus preserving the initial user code semantics.

4.4 Translated Code Integration Architecture

Once ParallelME compiler is a source-to-source compiler, it reads a high-level input language and outputs another high-level language. In the input it receives Java code and generates as output Java and C codes. Since 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 and must be preserved. In this sense, the compiler must interpret the input Java code and integrate it with 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 7 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.

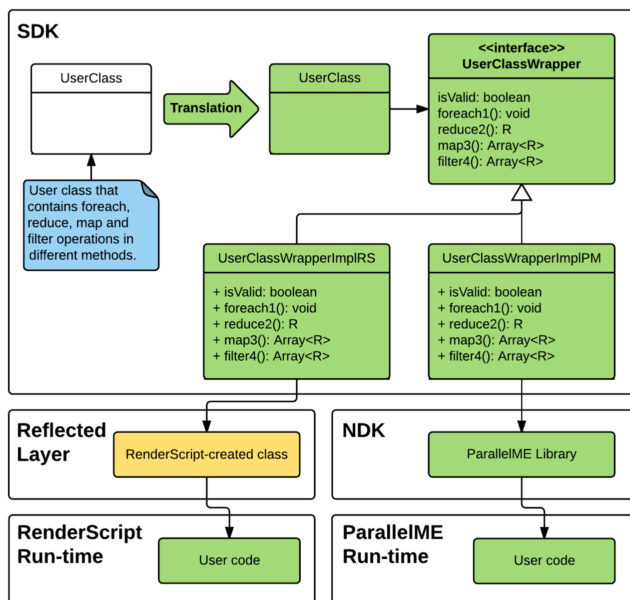

Figure 7: Translated code integration architecture

The user application is connected to run-times through the common interface presented in section 4.3.2. This interface, shown in Figure 7 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.

4.5 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 4.3.1, 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.

In order to facilitate the understanding of execution strategies, the graphical notation proposed by [15] 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 8.

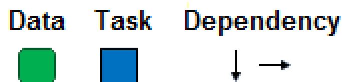

Figure 8

4.5.1 Foreach and Map

Due to their iteration characteristics, both *foreach* and *map* operations can be expressed using the same **map** pattern as described by [15]. 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 9, 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 [15].

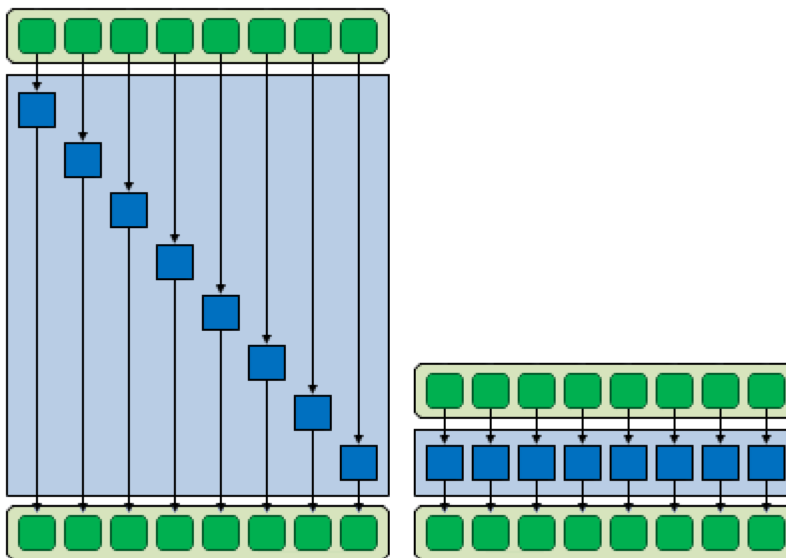

Figure 9: Sequential (left) and parallel (right) graphs for Foreach and Map operations

4.5.2 Reduce

The *reduce* operation is expressed using the homonym pattern described by [15]. 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 10. Once the combine function implies in dependencies from different elements, the parallel

version of this operation is structured in a particular way.

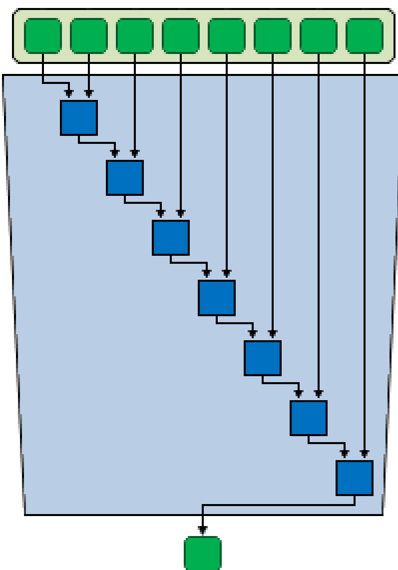

Figure 10: Sequential Reduce

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 11. 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 11, 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 12.

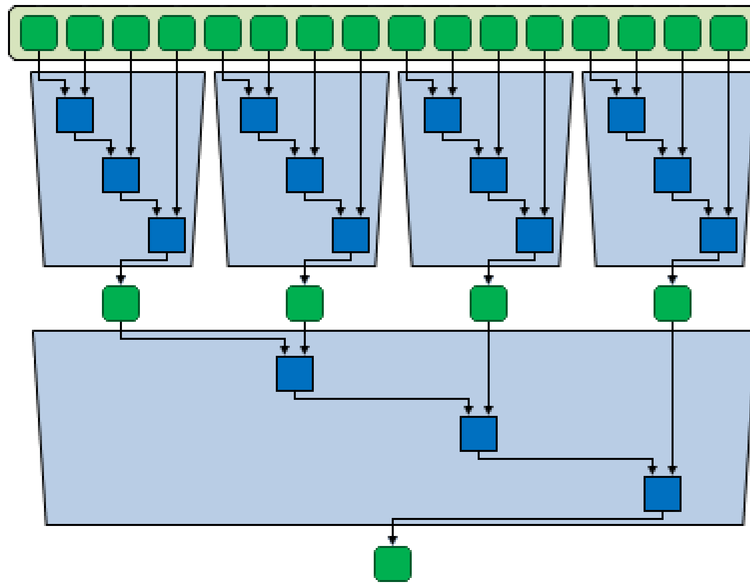

Figure 11: Parallel Reduce

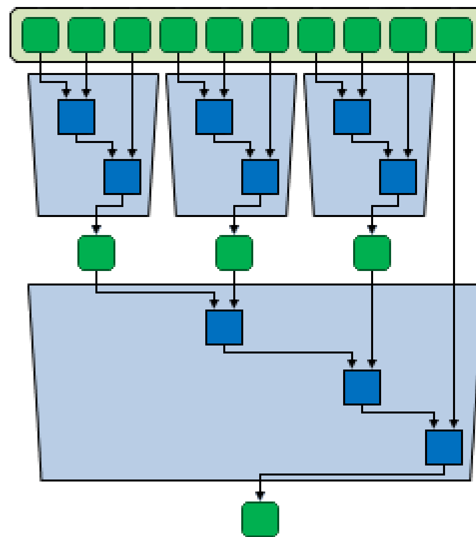

Figure 12: Parallel Reduce for Array class

4.5.3 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 13, an intermediate data-set with the same size of the initial collection. This intermediate data-set is than 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 14. 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.

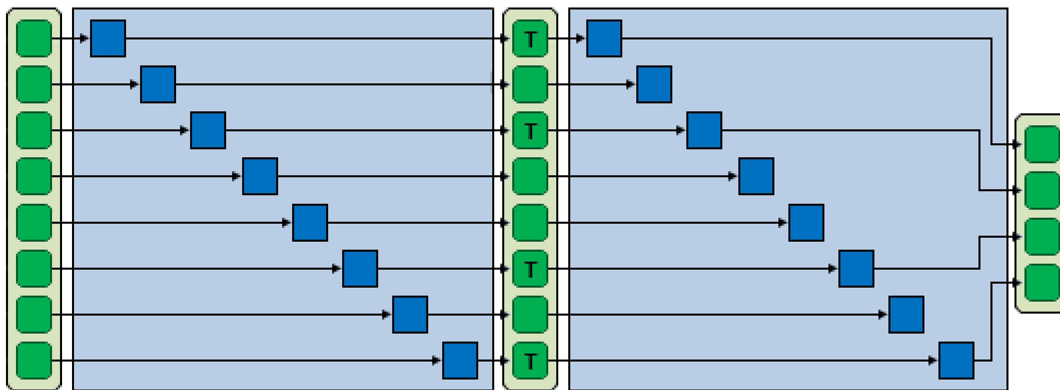

Figure 13: Sequential Filter

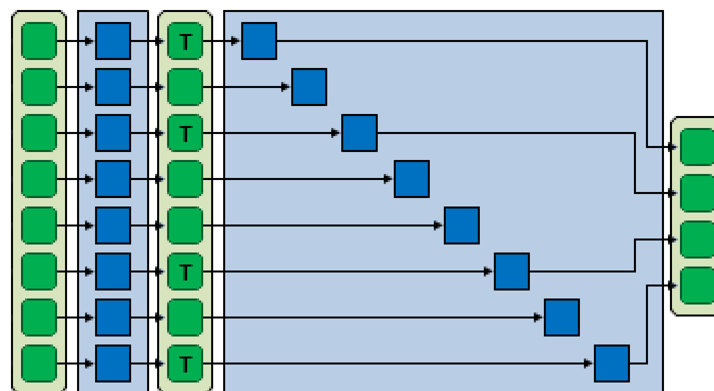

Figure 14: Parallel Filter

5 How to use

5.1 ParallelME Compiler

Before using ParallelME, the source-to-source compiler must be compiled and assembled into a jar file. To perform this operation, you must first install Maven on the computer that will be used compile the project. As long as Maven is installed, open the command line, navigate to *parallelme-compiler* folder and execute the command *mvn clean package*. If everything goes fine, a *parallelme-compiler-VERSION.jar* file will be created in *target* folder.

To use *parallelme-compiler-VERSION.jar* to compile a file, use the following parameters:

- **-f file_or_folder_with_user_code**
- **-o destination_folder**

Example call:

- **java -jar parallelme-compiler-0.1-SNAPSHOT-jar-with-dependencies.jar -f Test.java -o ./output**

Following the input code translation, three folders namely **java**, **jni** and **rs** are created in the output destination folder informed in the parameter **-o**. These folders contains, respectively, the modified user code and the intermediate layer in Java, C and C++ files created for ParallelME runtime and RenderScript files. Once ParallelME compiler is not integrated (yet) to Android Studio, these output folders and their files must be integrated to user application manually. Thus, they must be copied to the *app/src/main* folder of the user application, replacing the original Java input classes by those generated by the compiler.

After the translated code is placed with the original user application code, the developer must call the compiler-created constructor for its original input classes. The new constructor requires a RenderScript object, which in turn requires a context to be created. For an example of RenderScript object creation, refer to the method *onCreate* in *MainActivity* class of **ToneMapping** application in the **samples** repository or to RenderScript's documentation[1].

Prior to compiling in Android Studio and deploying the ParallelME compiler-generated source code, it is also required that the **jni** files are compiled by a special command. Thus, the user must open a command line prompt, go to the *app/src/main* folder and execute the command **ndk-build**. This command is responsible for reading those **.mk** files generated by ParallelME compiler and create binaries for those C and C++ files.

5.2 ParallelME User-library

In order to develop programs with the user-library in Android Studio, the user must first clone it from GitHub[2]. After that, to include ParallelME user-library in user application, the steps bellow must be followed:

-
-

[1] https://developer.android.com/reference/android/renderscript/RenderScript.html
[2] https://github.com/ParallelME/userlibrary

5.3 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.

References

[1] "RenderScript." `https://developer.android.com/guide/topics/renderscript`, 2016.

[2] "OpenCL by khronos group." `https://www.khronos.org/opencl/`, 2016.

[3] G. Andrade, W. de Carvalho, R. Utsch, P. Caldeira, A. Alburquerque, F. Ferracioli, L. Rocha, M. Frank, D. Guedes, and R. Ferreira, "Parallelme: A parallel mobile engine to explore heterogeneity in mobile computing architectures," in *European Conference on Parallel Processing*, pp. 447–459, Springer, 2016.

[4] A. Prokopec, P. Bagwell, T. Rompf, and M. Odersky, "A Generic Parallel Collection Framework," in *Proceedings of the 17th International Conference on Parallel Processing - Volume Part II, EUROPAR 11*, 2011.

[5] Google, "RenderScript Overview." `https://stuff.mit.edu/afs/sipb/project/android/docs/guide/topics/renderscript/compute.html#overview`, visited on 09/2016, 2016.

[6] Oracle, "Java Native Interface Specification." `http://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/jniTOC.html`, visited on 09/2016, 2016.

[7] K. G. Gupta, N. Agrawal, and S. K. 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*, pp. 1–5, IEEE, 2013.

[8] P. C. Pratt-Szeliga, J. W. Fawcett, and R. D. Welch, "Rootbeer: Seamlessly using GPSUs from Java," in *High Performance Computing and Communication & 2012 IEEE 9th International Conference on Embedded Software and Systems (HPCC-ICESS), 2012 IEEE 14th International Conference on*, pp. 375–380, IEEE, 2012.

[9] N. Giacaman, O. 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.

[10] A. Acosta and F. Almeida, "Performance analysis of paralldroid generated programs," in *2014 22nd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, pp. 60–67, IEEE, 2014.

[11] developer.android.com, "Java 8 Language Features." `https://developer.android.com/preview/j8-jack.html`, 2016. Accessed: 2016-06-25.

[12] Oracle, "Oracle Java Documentation." `https://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html`, 2016. Accessed: 2016-06-25.

[13] T. Parr, *The definitive ANTLR 4 reference*. Pragmatic Bookshelf, 2nd ed., 2013.

[14] "ANTLR - Parser Generator." `http://www.antlr.org/`, 2016. Accessed: 2015-12-15.

[15] M. McCool, J. Reinders, and A. Robison, *Structured Parallel Programming: Patterns for Efficient Computation*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1st ed., 2012.