Universidade Federal de Minas Gerais

Source-to-source compiler definitions

Report submitted to LGE as part of the Project *Scheduling heterogeneous tasks on heterogeneous devices environments.*

Belo Horizonte, MG
March, 2016

A source-to-source compiler prototype

1 Introduction

This report presents all the definitions and activities performed to accomplish the development of source-to-source compiler prototype. It begins with a detailed description of our proposed framework for parallel programming in Android, its modules and how they interact with the source-to-source compiler. Following sections describes the compiler structure, the tools used to create the prototype and the example application that was chosen to be the first translated code. The report is then finished with details of a more complex visual recognition application that will be used to validate future versions of our compiler, along with the bibliographical studies that served as basis for our work.

2 Framework structure

Our framework, namely ParallelME (Parallel Mobile Engine) was conceived as a complete infrastructure for parallel programming in Android. It is comprised of a programming abstraction, a runtime environment and a source-to-source compiler. The programming abstraction was designed to provide an easy-to-use and generic programming model for parallel applications in Android using Java, though inspired by ideas found in the Scala collections library [1]. The runtime environments specified for our framework are RenderScript and ParallelME runtime (an OpenCL-based runtime), though only the former is used in the first version of our compiler. This source-to-source compiler was developed to be a bridge between the programs written on the programming abstraction and the target runtime environment.

To be able to transform the user code on a valid RenderScript or ParallelME runtime code in C, the source-to-source compiler must have a well defined input. In order to make the framework easier to use, avoid unnecessary code translation and focus on parallelization features, it was divided in three parts, called modules. These modules are defined by the role each of them play on the framework, from interaction with user code to the compiler's core, as follows:

- **User library:** the user library is a Java API with support for the programming abstraction proposed. It corresponds to a collections' package composed of several classes whic support different collections types. These classes are handled directly by the user in his Android application project.
- **Internal library:** the internal library is an ANSI C version of those high level collection classes found in the user library. These classes are handled by the source-to-source compiler to produce the output code on the selected runtime environment.

- **Source-to-source compiler:** the source-to-source compiler corresponds to the compiler code itself.

The source-to-source compiler plays an important role integrating these modules. Its responsibilities are not only to validate user code, but also to put together the user code, the user library and internal library producing a valid output code. The relation between these three modules and how they integrate with the source-to-source compiler is illustrated in Figure 1.

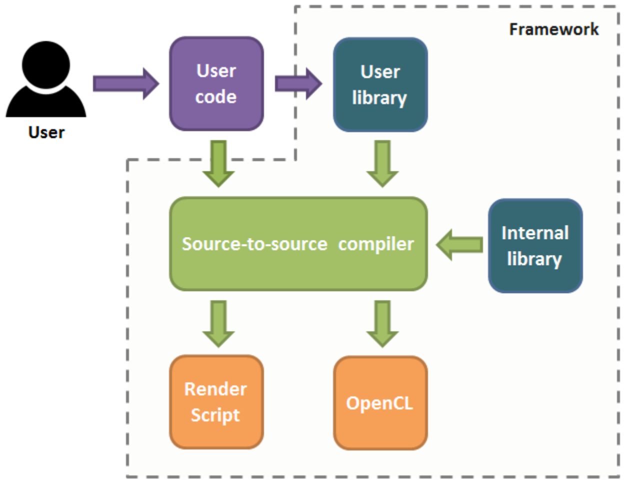

Figure 1: Modules' overview

2.1 User library

The user library holds the programming model proposed for ParallelME. It is comprised of a series of collection's classes as part of a package named **userlibrary**. These collections will represent common data structures like arrays, lists, hash sets and hash maps capable of handling an abstract data type. The abstract data type corresponds to the user class which is used to store user code. All user data to be stored on the respective collection is provided by the parametrized abstract type through the constructor.

So far, the user library contains three functional classes: HDRImage, BitmapImage and FlatArray. The first and second classes offers support for image processing, while the former is a generic bi-dimensional array which is flattened to a single dimension using Morton encoding or Row Major representation.

The example bellow illustrates how some of the proposed collections are instantiated to handle an image, using a native support to RGB color space:

```
BitmapImage image = new BitmapImage(bitmap);
HDRImage image = new HDRImage(resources, resourceId);
FlatArray<RGB> array = new FlatArray<RGB>(RGB.class, width, height, data);
```

Listing 1: Instantiation of collection classes

The user is able to run parallel operations with any collection by calling *par* method followed by a call to an iteration method. The code presented on Listing 2 is an example of how parallel operations are described using a *foreach* iterator. The call to *par* method in the user code is recognized by the source-to-source compiler and then translated to an equivalent version in RenderScript or ParallelME runtime.

```
image.par().foreach(new UserFunction<Pixel>() {
    @Override
    public void function(Pixel pixel) {
        pixel.rgba.red = 0;
        pixel.rgba.green = 0;
        pixel.rgba.blue = 0;
    }
});
```

Listing 2: Executing a parallel operation on a collection

2.2 Internal library

The purpose of the internal library is to integrate the Java environment with the target runtime environment. Thus, this library contains predefined code that is necessary to support a given target runtime, integrating it with the user library programming abstractions.

For ParallelME runtime, the internal library code is stored in a resource folder named ParallelME which is automatically exported when compiling code to this target runtime. This library was written to integrate ParallelME runtime with OpenCL and also RenderScript. More details can be found on the Runtime Environment report.

For RenderScript runtime, the internal library code is generated automatically during code translation and is integrated to its respective runtime definition class source code.

2.3 Source-to-Source Compiler

The source-to-source compiler takes as an input the Java code written using the user library and translates it into C code compatible with RenderScript or ParallelME runtime. Besides that, the compiler must also integrate the translated code with the Java application performing modifications on the original Java code in order to support the target runtime environment.

In this sense, our compiler is comprised of 3 macro steps which are presented here and will be detailed in the proper compiler section of this report:

- **User library detection:** it must detect where the user code is accessing user-library classes. These locations will point to the code that must be translated to the target runtime.
- **Perform memory binding:** it must detect where the user is providing the data that will be processed in order to transmit it to the runtime environment.
- **Convert user code to RenderScript/ParallelME runtime:** the user code written on the iterators' body of the user-library classes must be translated to a RenderScript or ParallelME runtime compliant version in order to produce the behaviour specified on the high level programming abstraction.

In favor of creating a prototype on a reasonable schedule, we defined some rules and limitations on the input and output code. These definitions reduced considerably the complexity of the implementation, making it possible to produce a viable prototype in a short period of time.

2.3.1 Input definition

One of the first tasks performed by a compiler is to check for inconsistencies regarding the syntax and semantics of the code written in a given language [2]. The development of a fully functional syntax and semantics analyzer would greatly increase the complexity of our prototype, so we had to find viable alternatives to perform these tasks.

As our user library was designed in Java to be introduced in the user application with no changes on the programming language, though introducing a new programming abstraction, the code produced with its use is 100% compliant with Java 7 language specification (actual version supported by Android). The guarantee that the code produced by our user will meet Java standards, allowed us to rely on the Java compiler (javac) for syntactic and semantic evaluation on the Java language level. It means that before taking a given code to our compiler, this code must be compiled with *javac* in order to check that it is fully compliant with Java specification. This strategy, along with the time and complexity reduction, contributed to increase the compiler robustness, since future modifications on the language specification can be easier to incorporate as the necessary implementation on our framework is also reduced.

Another decision taken to reduce the implementation effort was to introduce some limitations on the input code in Java that may be written by the user. These limitations are related to the parametrized class that is provided to the collection using generic data types. Up to this moment, the parametrized class may only be formed by Java primitive types.

2.3.2 Output definition

The output code must be able to run on RenderScript or ParallelME runtime. To accomplish that, the compiler must translate the parametrized user class along with the code in the iterators body and integrate it with the internal library, creating the necessary apparatus to reproduce the original application behaviour specified by the user on the target framework.

3 Compiler Prototype

ParallelME compiler prototype was developed in two iterations. The first iteration produced a compiler compliant with RenderScript runtime and was used to integrate the fundamental definitions and decisions created during the analysis and specification phases. The second version and final prototype was built to support both RenderScript and ParallelME runtime, thus it was redesigned with a more complex structure, including an intermediate representation level.

As the preliminary version was conceived to put together many concepts, we had to prioritize which features would be part of it. In this sense, we worked on the most fundamental characteristics that could produce a compiler with the best return on investment considering our time and effort. With this first version, which was built over a very simple syntax and semantic analysis, we were able to validate the programming abstraction, propose modifications and also evaluate the viability of the compiler itself.

For the second version a new structure, more complex and robust, was created in order to decouple runtime features from code translation tasks. Therefore a two pass compiler with an intermediate representation was created with complete separation of responsibilities between each compiler's module. The in-memory intermediate representation level was conceived over the abstract concepts that defines the integration between the programming abstraction and the runtime target. The final prototype version of ParallelME compiler is shown in figure 2 and will be described in detail in the following sections.

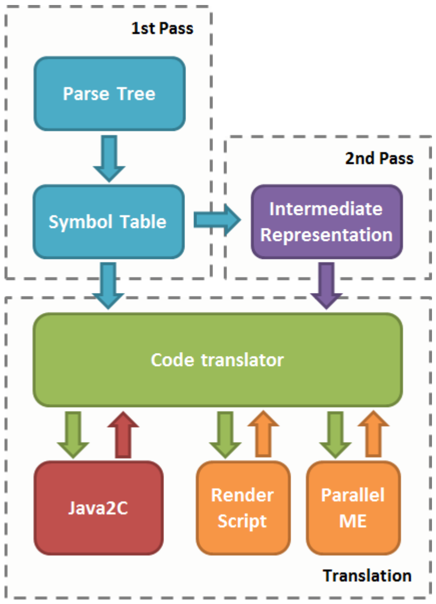

Figure 2: Compiler overview

3.1 First Pass

The compiler's first pass is responsible for syntax analysis and was developed using ANTLR. ANLTR (**AN**other **T**ool for **L**anguage **R**ecognition) is a powerful parser generator for reading, processing, executing, or translating structured text or binary files [3] [4]. It is widely used to build languages, tools, and frameworks. Starting from a Java grammar, ANTLR generates a parser that is used on the compiler to build and walk parse trees. The incorporation of ANTLR allowed us to save implementation time, being adopted since the first version of ParallelME compiler.

On ParallelME compiler, *JavaParser* is a Java parser class generated by ANTLR from a Java grammar. It allows the representation and operations' execution on a given source code parse tree. The code in Listing 3 is a declaration of a user-library object followed by its parse tree visual representation in Figure 3:

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

Listing 3: User library object declaration

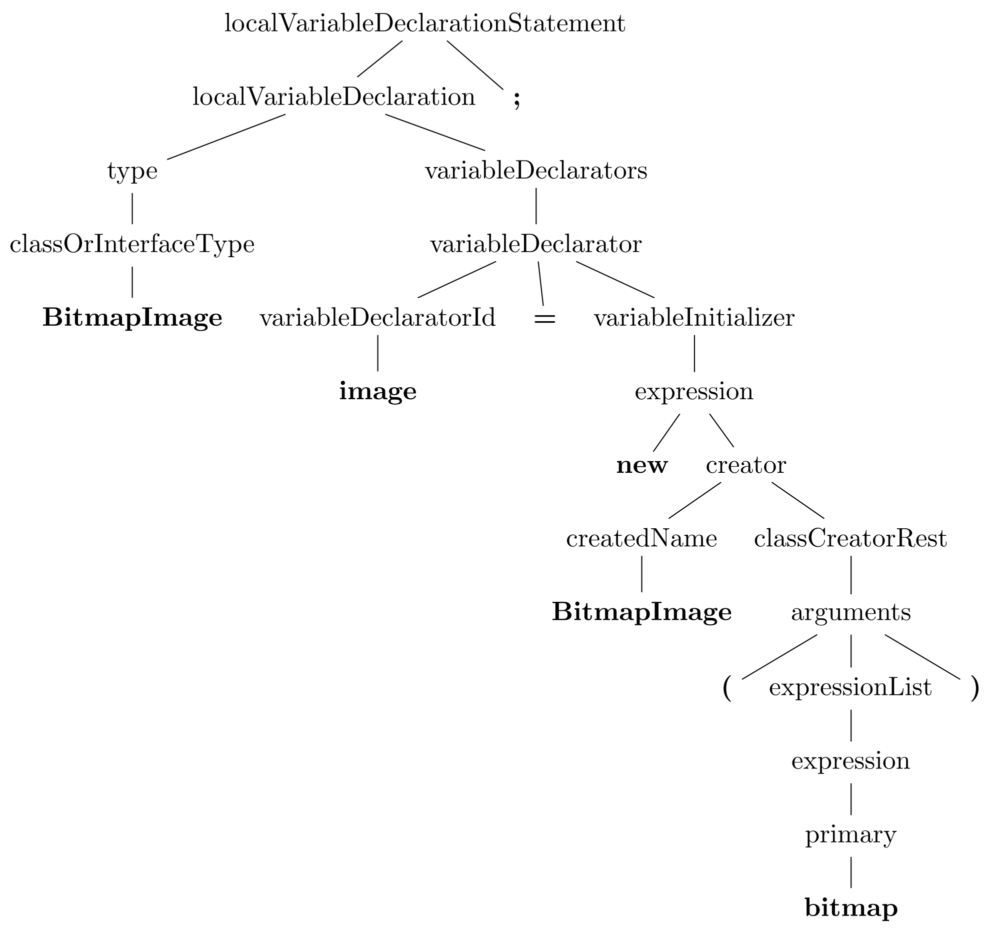

Figure 3: Listing **??** matching Parse Tree

To walk in a parse tree using ANTLR it is necessary to implement a visitor or a listener. It is possible to visit each node of the parse tree using both, though each has its own particularities. To implement a customized visitor or listener, it is necessary to follow a predefined contract created by ANTLR as an interface or class. For this reason, the *ScopeDrivenListener* class was implemented by extending ANTLR *JavaListener* class.

ScopeDrivenListener class was created to hold most of the complexity of the symbol table creation during the parse tree walk. The remaining code of the first pass was implemented in class *CompilerFirstPassListener*, which extends *ScopeDrivenListener*.

The symbol table conceived for the final prototype of ParallelME compiler was a major improvement compared to the first version. Besides being a recursive data structure like the first version, the symbol table is now scope-driven and stores more data that is necessary for second pass, increasing the overall compiler performance. A scope-drive symbol table means that all symbols stored on the table are in the same hierarchical scope as in the original Java source code given as input. This allows scope level analysis during the second pass with a reasonable computing cost.

All operations related to the symbol table creation are implemented in *ScopeDrivenListener* class. As this class derives from ANTLR *JavaListener* class, it must implement *enter* and *exit* methods that correspond to the entrance and exit of each node of the parse tree. Some nodes are represented in figure 3 and an example method is presented in listing 4.

```
@Override
public void enterLocalVariableDeclaration(
                JavaParser.LocalVariableDeclarationContext ctx) {
        this.createVariable(ctx.variableDeclarators(), ctx.type(),
                        this.getCurrentStatementAddress());
}
```

Listing 4: ScopeDrivenListener implementation of a JavaListener method

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 5.

```
<RootSymbol> $
  <ClassSymbol> BitmapLoaderTest,
    <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, , RGB, []
        <MethodSymbol> function, void, [<VariableSymbol> pixel, RGB, ]
          <VariableSymbol> pixel, RGB,
          <MethodBodySymbol>
          <VariableSymbol> foo, RGB,
          <CreatorSymbol> $fooCreator, RGB, foo, , []
          <VariableSymbol> w, float, null
```

Listing 5: Scope-driven symbol table

3.2 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 data for code translation for a target runtime. This data will have the form of an intermediate representation, which is an in-memory high level representation of those operations that can be expressed with the user library.

Given a programming task for an operation over a collection on some target runtime, the user must transfer the data from Java environment to the runtime (input bind operation), iterate over this data and finally, after all iterations are done, retrieve data from runtime to Java environment (output bind operation). These abstract tasks describes in a high level how the programming abstraction works and how the intermediate representation was designed.

In order to decouple runtime features from syntax and semantic code analysis, the in-memory intermediate representation contains all the necessary information for translating input bind, iteration and output bind operations. This information is collected during the second pass and then passed to the target runtime, which will be responsible for translating this information to an equivalent runtime compliant code. Listing 6 shows an code example with input bind (line 1), iteration (line 2) and output bind (line 10) operations. Its equivalent intermediate representation are presented in listings 7, 8 and 9, while the output code translated to RenderScript is presented in listing **??**.

```
1   BitmapImage image = new BitmapImage(bitmap);
2   image.par().foreach(new UserFunction<Pixel>() {
3       @Override
4       public void function(Pixel pixel) {
5           pixel.rgba.red += 1;
6           pixel.rgba.green += 2;
7           pixel.rgba.blue += 3;
8       }
9   });
10  bitmap = (Bitmap)image.toBitmap();
```

Listing 6: Input bind, iteration and output bind operations on user library level

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

Listing 7: Input Bind Intermediate Representation

3.3 Code Translation

Code translation phase is responsible for transforming the intermediate representation in a valid code for the target runtime. The translation phase is comprised of 8 steps which occur on the

```
Iterator {
    Type: 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 8: Iterator Intermediate Representation

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

Listing 9: Output Bind Intermediate Representation

sequence presented on the following sections.

3.3.1 Iterators' type analysis

This analysis consists in the verification of parallelization possibilities and will mark each iterator as sequential or parallel. Those iterators that contains in its body external non-constant variables will be translated to sequential iterators on the target runtime, while those that do not contain external variables' dependencies or have external variables declared as constants will be transformed into parallel iterators.

The fragment code presented in listing 10 contains a reference for the numerical class variable *sum* inside the user function. This variable has its value changed during the user function execution and clearly, due to concurrency issues, indicates that parallel writes (line 5) may cause inconsistencies on the operation result unless the code is executed sequentially or a critical section is created. For cases like this, when any external non-constant variable is **used** inside the user function, the compiler will translate the user code into a sequential version for the target runtime, issuing a warning for this situation. Listing 11 presents the translated code for RenderScript runtime with the sequential operation in two nested *for* loops (lines 2 and 3).

```
1         this.sum = 0;
2         image.par().foreach(new UserFunction<Pixel>() {
3             @Override
4             public void function(Pixel pixel) {
5                 sum += Math.log(0.00001f + pixel.rgba.red);
6             }
7         });
```

Listing 10: User function with external non-constant variable

```
1         float4 pixel;
2         for(int x = 0; x < gInputDataXSize; ++x) {
3             for(int y = 0; y < gInputDataYSize; ++y) {
4                 pixel = rsGetElementAt_float4(gInput_$imageOut, x, y);
5                 gSum += log(0.00001f + pixel.s0);
6             }
7         }
8         rsSetElementAt_float(gOutput_sum, gSum, 0);
```

Listing 11: RenderScript code for user function with external non-constant variable

3.3.2 Input data binding

The input data binding operation is responsible for creating memory allocations on the target runtime and transfering the user data to this environment. This operation is done by analyzing the user library object creation, since data must be provided through the user library class constructor.

The translated code for listing 3 is shown in listing 12. This fragment presents the Java code that was generated by the compiler to integrate the user application with RenderScript runtime.

```
1         Allocation $imageIn, $imageOut;
2         Type $imageInDataType;
3         $imageIn = Allocation.createFromBitmap(mRS, bitmap,
4             Allocation.MipmapControl.MIPMAP_NONE,
5             Allocation.USAGE_SCRIPT | Allocation.USAGE_SHARED);
6         $imageInDataType = new Type.Builder(mRS, Element.F32_3(mRS))
7                 .setX($imageIn.getType().getX())
8                 .setY($imageIn.getType().getY())
9                 .create();
10        $imageOut = Allocation.createTyped(mRS, $imageInDataType);
```

Listing 12: Java output code for RenderScript input bind

3.3.3 Get methods' translation

One of the key objectives of the user library is to provide a concise and easy-to-use abstraction for dealing with user collection data. Once the user provides the input data for a given user library class, it is desired that he/she may work with this user library object instead of the input data throughout the code. For this reason, some general purpose methods, besides input bind, iterators and output bind are available in user library classes for information retrieval, like *getWidth* and *getHeight* in *BitmapImage* and *HDRImage* classes. Once these methods are used in user code, they must also be translated to equivalent operations on the target runtime. Listing 13 presents user library code in the left side and its translated version to RenderScript runtime.

```
int width = image.getWidth();
int height = image.getHeight();
```

```
int width = $imageIn.getType().getX();
int height = $imageIn.getType().getY();
```

Listing 13: User library get method translation (left = original, right = translated)

3.3.4 Iterators' translation

Iterator's translation is performed in two steps: replacing fragments in Java code for an equivalent runtime function call and translating the user code to a runtime equivalent version. For RenderScript, the translated code for the parallel iterator presented in listing 14 is shown in listing 15 and **??**. Listing 15 presents the code that replaced the original user library code in Java application, while listing **??** presents the RenderScript kernel function that was created to perform the same operations as the user function code.

```
        final float max2 = (float)Math.pow(getMaxValue(), 2);
        image.par().foreach(new UserFunction<Pixel>() {
            @Override
            public void function(Pixel pixel) {
                pixel.rgba.red *= (1.0f + pixel.rgba.red / max2) / (1.0f + pixel.rgba.red);
            }
        });
```

Listing 14: User library iterator code (Java)

```
        final float max2 = (float)Math.pow(getMaxValue(), 2);
        function6_script.set_gMax2(max2);
        function6_script.forEach_root($imageOut, $imageOut);
```

Listing 15: Translated iterator code (Java calling RenderScript)

```
float gMax2;
float4 __attribute__((kernel)) root(float4 pixel, uint32_t x, uint32_t y) {
    pixel.s0 *= (1.0f + pixel.s0 / gMax2) / (1.0f + pixel.s0);
    return pixel;
}
```

Listing 16: Translated iterator code (RenderScript kernel)

3.3.5 Output data binding

Output data binding translation is performed by replacing user library specific method calls to equivalent runtime code. The fragment shown in listing 17 is translated to listing 18 after compilation.

```
    Bitmap bitmap = image.toBitmap();
```

Listing 17: User library output bind code (Java)

```
    bitmap = Bitmap.createBitmap($imageIn.getType().getX(),
        $imageIn.getType().getY(),
        Bitmap.Config.ARGB_8888);
    function0_script.forEach_root($imageOut, $imageIn);
    $imageIn.copyTo(bitmap);
```

Listing 18: Translated output bind code (Java calling RenderScript)

3.3.6 Imports removal

Imports that references ParallelME user library classes or packages must be removed, since the translated code won't have user library classes, once they will be replaced by runtime code.

3.3.7 Imports introduction

All those necessary imports for proper runtime execution will be included on the output code.

3.4 Limitations

The current compiler version is limited regarding the target runtime which is only RenderScript for this prototype. It also has some limitations regarding user function code translation. 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, user function code is restricted to the following cases:

- Only Java primitive types up to 32 bits are allowed, since they are mapped directly to C types during translation;
- Variables declared outside the user function scope can be used for read and write operations;

- Variables declared outside the user function scope **without** *final* modifier will imply in sequential code translation for the given iterator;
- Variables declared outside the user function scope **with** *final* modifier will imply in parallelizable code translation for the given iterator;
- Arrays are **not** supported;
- Strings are **not** supported;
- Method calls are **not** supported;
- Nested user functions are **not** allowed;
- Operations with Bitmaps and HDR images are fully supported in Java sequential version and the compiler generated version, whereas there is still work to be done on compiler generated code for FlatArray data type (although it works on Java sequential version).

3.5 Usage

Before using ParallelME project, it must be compiled and packed 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-1.0-SNAPSHOT-jar-with-dependencies.jar -f Test.java -o ./output**

3.6 Planned Implementations

Some implementations are already planned for the remaining of this project:

- Add support for Java2C;
- Add support for nested user functions;
- Add full support for FlatArray;
- Add support for ParallelME runtime.

References

[1] A. Prokopec, P. Bagwell, T. Rompf, and M. Odersky, “A generic parallel collection framework,” in *Euro-Par 2011 Parallel Processing*, pp. 136–147, Springer, 2011.

[2] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools (2Nd Edition)*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2006.

[3] “Antlr - parser generator.” `http://www.antlr.org/`. Accessed: 2015-12-15.

[4] T. Parr, *The Definitive ANTLR 4 Reference*. Pragmatic Bookshelf, 2nd ed., 2013.