Automatic Generation of OpenCL Code for ARM Architectures

Sergio Afonso(✉), Alejandro Acosta, and Francisco Almeida

Universidad de La Laguna, San Cristóbal de La Laguna, Spain
{safonsof,aacostad,falmeida}@ull.es

Abstract. The efficient exploitation of the increasing computational capabilities of mobile devices is still a challenge. The heterogeneity of Systems on Chip (SoC) makes necessary a very specific knowledge of their hardware in order to harness their full potential. OpenCL is a well known standard for cross-platform usage of accelerator devices. We follow an annotation-based approach for solving the problem of high development cost of OpenCL programming for mobile devices. With our approach, the programmer can select from different programming models the one that offers the best performance for each section of the application. Computational results show that our automatically-generated OpenCL code can outperform Renderscript when running on the GPU of Android devices, making it the best choice for a range of parallel algorithms.

Keywords: Parallelizing compiler · Source-to-source translation · Annotation based · OpenCL · Android · ARM

1 Introduction

Technologies previously only available in desktop computers are now implemented in embedded and mobile devices. In this scenario, we can find that new processors integrating multicore architectures, GPUs and DSPs are being developed for this market. The Nvidia Tegra [15], the Qualcomm Snapdragon [16] and the Samsung Exynos [17] are some examples of platforms that go in this direction. Conceptually, the architectural model can be viewed as a traditional heterogeneous CPU/GPU system where memory is shared between processing units and acts as a high-bandwidth communication channel.

In non-unified memory architectures, it is common to have a subset of system memory addressable by the GPU. Technologies like Algorithmic Memory [12], GPUDirect [14] and Unified Virtual Addressing from Nvidia and HSA from AMD [5] are working towards a unified memory system for CPUs and GPUs on top of traditional memory architectures. At the same time, memory performance continues to be outpaced by the ever increasing demand of faster processors.

Many frameworks have been created to support the development of software for these devices. The main companies competing in this market have their own platforms: Android from Google [9], iOS from Apple [7] and Windows Phone from Microsoft [13]. Each of these platforms provide a high-level development framework that makes easier the creation of applications. However, they are more geared towards fast development of interactive applications than to reduce the difficulty of efficiently exploiting the underlying parallel architecture. Given the high heterogeneity existent among these devices, the creation of tools is needed to improve the development productivity while exploiting the computational capabilities of their different architectures.

Android provides three development models with distinct features that have to be used in different parts of the application in order to get the best overall performance. In [20] a detailed comparative of these models was presented and the necessity of a unified programming model for Android is highlighted.

- **Java:** A very comprehensive API is provided, so it is the easiest model to program. Most Android applications are written in Java, so Android developers should be familiar with this language.
- **Renderscript:** It is designed for computationally intensive tasks, mainly SPMD. It requires to learn a new language based on C.
- **Native C:** It provides access to native libraries, suffering from less runtime overhead than Java, which sometimes compensates the extra development cost that it supposes.

Paralldroid [1,2,4] is a development framework that allows the automatic creation of Native C and Renderscript applications—sequential and parallel—for mobile devices. Under Paralldroid, the developer annotates the main components of each Java class that has to be optimized. It uses the information provided by the annotations to generate a new program that incorporates the code sections to run on the CPU or GPU, using a specified target language. Therefore, Paralldroid unifies the different programming models of Android.

Paralldroid is an evolution of other annotation-based approaches to automatic parallelization, such as OpenMP or OpenACC, because it is higher level. OpenMP and OpenACC annotations still require the developer to annotate blocks of code inside the algorithm's implementation, whilst Paralldroid separates more clearly the implementation from the parallel semantics by applying annotations to classes, fields and methods. Our goal is to obtain comparable performance to these approaches while making it easier for the developer.

In this paper we present a new backend system for Paralldroid to support the generation of OpenCL code. OpenCL is a native library and programming language for writing high performance applications for heterogeneous systems, supporting many kinds of accelerators [10]. It provides a mechanism for parallel programming and a low-level API for communicating data and handling the different computing devices present in the hardware platform. Currently, OpenCL is not supported by the Android implementations provided by Google. However, given the heterogeneity of the mobile ecosystem, some manufacturers offer OpenCL in their devices, so it is still interesting to generate OpenCL for those.

The main contributions of this paper are:

- The importance of OpenCL in desktop systems is well known. Now, this programming model is extended to mobile devices. The code generation

methodology proposed allows OpenCL code to be transparently executed from Android applications written in Java.
- The support of OpenCL opens the possibility to extend Paralldroid to platforms other than Android. The only requirements that this platform needs to meet are support for Java and an OpenCL driver.
- We analyze the performance of the different programming models supported to prove the benefits of our tool. Computational results show that this new backend improves the performance of Paralldroid-generated programs when ran in GPUs.
- Our new approach lets high-level Java application developers take advantage of more efficient GPU executions without modifying the annotated Java source code. The improvements in performance come from the use of a lower-level library for heterogeneous computing and, as a result, the increase in complexity of the code generation process.

Some tools that generate parallel code from an extension to Java have been presented in [8,18,19]. In all those cases, the Java syntax is modified to introduce new syntactic elements into the language. The main disadvantage of this approach is that those new elements are not compatible with the definition of Java, so a standard Java compiler cannot compile the source code with these extensions. Paralldroid definitions are compatible with Java, because they are a set of new annotations that a standard Java compiler can just ignore. However, the semantics of parallel methods are not preserved in that case. In [11], authors present a Domain Specific Language for generating Renderscript code. It is specific for image processing languages, and it has the downside of requiring the user to learn a new programming language. Our proposal, in contrast, is based on the main language for Android, and our target users know this language.

This paper is structured as follows: Sect. 2 introduces the development models in Android and the different alternatives it offers for exploiting mobile devices, and some of the difficulties associated to each development model are shown. Section 3 gives an overview of the methodologies proposed by Paralldroid. Section 4 presents our new backend to Paralldroid to support the generation of OpenCL code. The performance of our automatically generated OpenCL code is validated in Sect. 5 using four different image-processing applications. We measure execution times of a sequential Java implementation and of Renderscript and OpenCL implementations automatically generated by Paralldroid. We finish with some conclusions and future work in Sect. 6.

2 The Development Model in Android

Android is a Linux based operating system mainly designed for mobile devices such as smartphones and tablets. Android applications are written in Java, and the Android Software Development Kit (SDK) provides the libraries and tools needed to build, test, and debug applications. Starting in version 5.0 applications run in the Android Run Time (ART), which manages system resources allocated to each application.

Besides the development of Java applications, Android provides packages of development tools and libraries to develop native applications: The Native Development Kit (NDK). The NDK enables to implement parts of the application running in ART using native programming languages such as C and C++. Native code communicates with the main class written in Java by using the Java Native Interface (JNI). Files of native source code are compiled using the GNU compiler (GCC). Note that using native code does not result in an automatic performance increase, but it always increases application complexity. Hence, its use is only recommended in CPU-intensive operations that don't allocate much memory, such as signal processing and physics simulations. Native code is also useful for porting an existing native library to Android. We can access OpenCL from the native context if the OpenCL runtime libraries are present in the device.

In order to exploit the high computational capabilities on current devices, Android provides Renderscript, which is a high performance computation API and a programming language based on the C language (C99 standard). Renderscript allows the execution of parallel applications under several types of processors such as the CPU, GPU or DSP, selecting one of them at runtime depending on the hardware's features. Renderscript (`.rs` files) codes are compiled using an LLVM compiler based on `Clang`. Moreover, it generates a set of Java wrapper classes around the Renderscript code. Again, the use of Renderscript code does not result in an automatic performance increase, but it is useful for applications that do image processing, mathematical modelling, or any operations that require lots of parallel computation.

3 Paralldroid

Paralldroid is designed to ease the development of parallel applications on the Android platform. We assume that mobile platforms feature a classical CPU and other kind of *co-processor*, like a GPU, that can be exploited through OpenCL or Renderscript. The way Paralldroid does this is by transforming the original Java source code into another code that, preserving the same semantics, is executed in a more efficient way. The generation of code on other languages is also required in order to take advantage of all the programming models in Android, but in each algorithm the best programming model to use can be a different one due to their different features. This is why the target language is something the user explicitly indicates when using Paralldroid.

Directive based parallelism has been successfully used in applications for High Performance Computing (HPC) systems for years, and Paralldroid takes the same approach in the mobile application development world.

The methodologies proposed by Paralldroid can be defined in two points:

– **Annotation methodology:** The `Target` annotation creates a data environment that allows the memory management and the execution of code in the target context. Elements inside a class (fields and methods) can be used to define the data and execution models in the target context. Paralldroid

Table 1. Paralldroid annotations

Annotation	Applied to	Parameters	Scope
@Target	Classes	Value	—
@Map	Fields, method parameters	Value	@Target
@Declare	Fields, methods	—	@Target
@Parallel	Methods	—	@Target
@Input	Method parameters	—	@Parallel
@Output	Method parameters	—	@Parallel
@NumThreads	Methods, method parameters	Field	@Parallel
@Index	Method parameters	—	@Parallel

defines a set of annotations that are applied to the class fields and method definitions. These annotations allow the creation of a device data environment, specify how a variable is mapped in the device data environment (data model) and also specify how a section of code is executed in the device environment (execution model), see Table 1.

– **Generation methodology:** The Paralldroid code generation process is integrated in the OpenJDK Java compiling process. It adds a set of stages in which the Paralldroid annotations are detected and new ASTs are generated according to these annotations. For each implementation to generate from a single annotated Java source, a translator class has to be created, which takes the original AST as input and outputs another AST. To add support for a new language, only is needed a new translator for the modified Java code and the target language. This makes Paralldroid easily extensible.

4 OpenCL Code Generation

The OpenCL standard represents the most important effort to create a common high performance programming interface for heterogeneous devices. The main issue of OpenCL is the complexity of its programming model, which makes it difficult to use and to keep the maintainability of the application.

The annotation methodology proposed by Paralldroid simplifies the complexity associated to OpenCL. Based on a Java class definition, the programmer can add a set of annotations to generate OpenCL code that can be executed transparently, because it is integrated into the Java workflow. This simplifies the development of OpenCL powered Android applications and helps this standard to have a major adoption on the Android development community.

Figure 1 shows the different sets of translations classes of Paralldroid. As with any of the other target languages of Paralldroid, the way to generate new ASTs from the original source code is to create a translation class for each output AST. The Java AST translator generates the modified Java code that manages

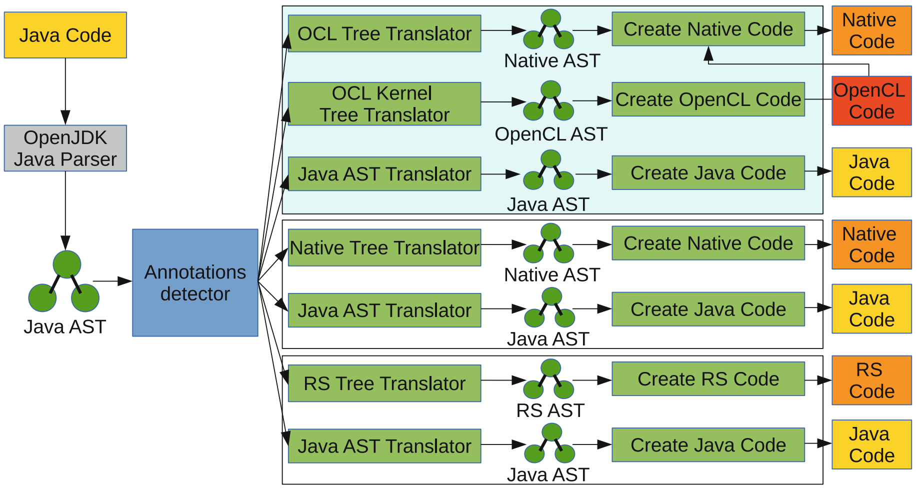
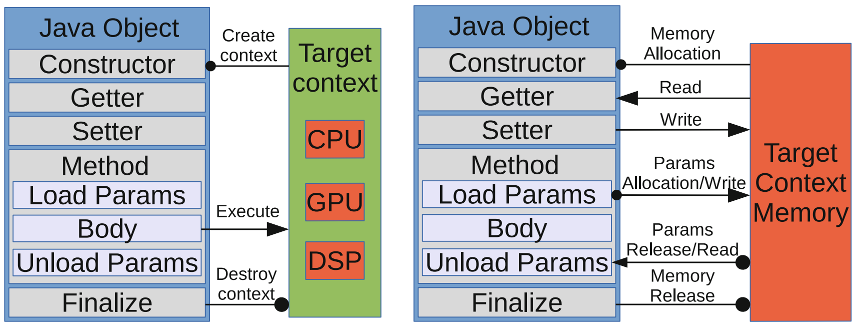


Fig. 1. Paralldroid translator classes. Our contribution is highlighted.

the data and execution models of OpenCL and forwards the implementation of methods to the target context, according to the methodology explained in Fig. 2. However, there is a noticeable difference between the set of translators for OpenCL and the others. That difference is the fact that there is one extra translator class. The OpenCL context is not directly accessible from the Java context so, in addition to generating Java and OpenCL code, native code has also to be created to work as a bridge between the two contexts. The OpenCL Kernel Translator, is also unlike all other translators in that it can only generate code for annotated methods, so it is not an "standalone" translator. This means that it has to be called from other translator when a parallel method is found. The OpenCL Kernel Translator outputs OpenCL C code that is inserted into the native code as a string literal. This complex model is hidden by Paralldroid; the programmer must only create a Java class and use the Paralldroid annotations.



(a) Execution model (b) Data model

Fig. 2. Execution and data models.

4.1 Execution Model

As shown in Fig. 2(a), the execution model consists of three basic operations, which are creating and releasing the OpenCL context and the execution of kernels. These operations are carried out in the constructor, finalizer and parallel methods of the class, respectively.

- **Static initializer:** Every class annotated with `@Target(OPENCL)` has many of its methods defined as native, so the library containing the implementation of those has to be loaded so that the user can call them.
- **Constructor:** The first time an instance of the class is created, in the constructor the OpenCL shared objects are initialized (context, command queue, ...), and the OpenCL C kernels are compiled.
- **Parallel methods:** The signature of the generated `@Parallel` methods differs from the original methods in that the `@Index` parameters have been stripped, since they are assigned at runtime by the OpenCL driver. Moreover, the method body is substituted by a kernel execution enqueued in native code. The actual code of the method is translated to an OpenCL C kernel that is embedded in the native code as a string constant.
- **Methods:** All `@Declare` methods are removed from the Java class and only accessible from the target context. Every other method can also be called from Java. For methods to be callable from the target context, they have to be defined in native code and in OpenCL C code as support functions. This makes it possible to call them from sequential and parallel methods.
- **Finalizer:** All the shared OpenCL objects that were created in the first instantiation of the class have to be released when the last instance of the class is garbage-collected.

4.2 Data Model

The data model of our approach to automatically offloading computation to accelerators is shown in Fig. 2(b). The user annotates fields and method parameters in order to specify the data movements between the different contexts.

- **Constructor:** The first time an instance of the class is created, all static fields are initialized according to the default values the user might have provided. Then, each time an instance is created, the native context has to be initialized with the same values that were used in the constructor. These two things are achieved by creating two native initialization functions that take as arguments the set of initialized fields in each case.
- **Fields:** Fields annotated as `@Declare` are deleted from the Java class and only exist in the native context. The rest of fields, however, need to be accessible from external Java code, even though they exist in the native context. We accomplish this by automatically generating getter and setter methods depending on the specified annotations. When a field is annotated as `@Map(TO)` or `@Map(TOFROM)`, a setter is generated, and when it is annotated

as `@Map(TOFROM)` or `@Map(FROM)`, a getter is generated. As arrays are represented in OpenCL as memory objects, these methods enqueue the required memory operations into the OpenCL command queue and transform the data format from Java to OpenCL and vice versa.

- **Methods:** When a native method that receives arrays as arguments is called, a conversion is needed between the Java and native formats and between the native and OpenCL formats. Data transfers are performed according to the `@Map` annotation applied to each array before and after running the body of the method. The semantic in this case is the same as that of fields.
- **Finalizer:** All memory allocated when initializing the instance is released when the garbage collector deletes it. When the last instance of the class is being deleted, then also global objects and native static fields are released.

```
 1 @Target(OPENCL)
 2 public class GrayScale {
 3   @Declare
 4   private float gMonoMult[] =
 5    {0.299f, 0.587f, 0.114f};
 6   @Map(TO)
 7   private int width;
 8   @Map(TO)
 9   private int height;
10
11   public GrayScale(int width, int height){
12     this.width = width;
13     this.height = height;
14   }
15   @Parallel
16   public void test(@Map(TO) int[] srcPxs,
17    @NumThreads @Map(FROM) int[] outPxs,
18    @Index int x){
19     int acc;
20
21     acc =  (int)(((srcPxs[x]    ) & 0xff)
22         * gMonoMult[0]);
23     acc += (int)(((srcPxs[x]>> 8) & 0xff)
24         * gMonoMult[1]);
25     acc += (int)(((srcPxs[x]>>16) & 0xff)
26         * gMonoMult[2]);
27
28     outPxs[x] = (acc) + (acc << 8)
29               + (acc << 16)
30               + (srcPxs[x] << 24);
31   }
32 }
```

Listing 1.1. GrayScale in Paralldroid

```
public class GrayScale {
  static {
    System.loadLibrary("grayscale");
  }
  private static int instanceCount = 0;
  private long instanceDataPtr;
  private float[] gMonoMult =
   {0.299F, 0.587F, 0.114F};
  private int width;
  private int height;
  public GrayScale(int width, int height){
    this.width = width;
    this.height = height;
    if (instanceCount == 0) initJNI();
    ++instanceCount;
    initGrayScale(gMonoMult, width, height);
  }
  public native void test(int[] srcPxstest,
   int[] outPxstest);
  public native void setWidth(int width);
  public native void setHeight(int height);
  protected void finalize(){
    destroyGrayScale();
    --instanceCount;
    if (instanceCount == 0) releaseJNI();
  }
  private native void initGrayScale(
   float[] gMonoMult, int width, int height);
  private native void destroyGrayScale();
  private static native void initJNI();
  private static native void releaseJNI();
}
```

Listing 1.2. Generated Java code

4.3 Paralldroid Example

Listing 1.1 shows a Java implementation for the algorithm of conversion of an image to gray scale using Paralldroid. The `@Target` directive (line 1) specifies that the class has to create an OpenCL context definition and that the elements of the class have to be defined in that context. Lines 3 to 9 define its fields. The constructor is defined in lines 11 to 14. The method `test` (lines 16 to 31) defines the algorithm to transform an image to gray scale. The `@Parallel` directive specifies that this method will be executed in parallel. `srcPxs` and `outPxs` are the vectors which contain the input image and output buffer, respectively. Note the usage of the appropriate `@Map` directive parameter in each of them. The `@NumThreads` directive applied to an array means that the parallel method will be executed with as many threads as elements there are in the array, but it is also possible to specify an integer variable. The `@Index` directive defines the index used in the parallel execution, which is used to access the elements of the input and output vectors. The value of this variable is assigned at runtime, and its values range from zero to the number of threads minus one.

Listing 1.2 shows the code generated by our Java translator, as described in Sects. 4.1 and 4.2. The library that it loads is obtained from compiling the native code that we also generate. A set of fields have been added. `instanceCount` lets us initialize and release native global variables before the first instance is created and after the last one is deleted, respectively. `instanceDataPtr` is a field only accessed from the native code that keeps a reference to a dynamically allocated struct holding the native instance data. The constructor (lines 11 to 17) is modified to call the native global and instance initializer function.

4.4 Error Handling

A new methodology for error handling has been developed as part of this new backend for Paralldroid. This methodology was designed to ease the detection and handling of errors that could occur in the target context to make the application fail gracefully, notify or solve these problems at runtime. This methodology could be adapted to other target languages of Paralldroid providing the user with a seamless and unified way of handling errors that occur in the target contexts.

An `OpenCLException` class was created, which is a `RuntimeException` that holds specific data of the OpenCL error. This exception contains an OpenCL error code that could be used to troubleshoot the reason of the problem, and a message with either the name of the file and line number where the error was detected or the compilation log in case the error occured when compiling the OpenCL C code at runtime.

After every call to a function of the OpenCL API in the generated native code, the error code returned by the function is checked and an `OpenCLException` is raised if there was an error. These exceptions can be handled from the calling Java code, without any need of knowing what the native code is actually doing.

5 Computational Results

Leaving aside to future researches other relevant metrics for smartphones and tablets (e.g., power management, network management, ...), we validate the performance of the generated code using four different applications. These are based on the Renderscript image processing benchmark [6] (transforming an image to gray scale, changing contrast and saturation levels of an image and convolutions with window sizes 3 × 3 and 5 × 5). In all cases, we implemented two versions of the code: a Java sequential version and a Java version with Paralldroid annotations. From the same annotated Java code two versions were automatically generated by Paralldroid: Renderscript and OpenCL. As it was shown in [3], automatically generated Renderscript code performance was comparable to its handwritten counterpart, so we compare our generated OpenCL code to that generated Renderscript code. Our implementations were tested over a Sony Xperia Z (labelled SXZ) and an Odroid-XU3 (labelled XU3). Sony Xperia Z is based on a Qualcomm APQ8064 Snapdragon S4 Pro SoC with a Quad-core Krait CPU @ 1.5 GHz and an Adreno 320 GPU, whilst Odroid-XU3 is based on a Samsung Exynos 5422 Octa SoC with dual ARM CPUs (Cortex-A15 @ 2GHz and Cortex-A7 @ 1.3 GHz) and an 8-core ARM Mali-T628 MP6 GPU. Both devices have 2GB of RAM shared by CPU and GPU, and support OpenCL execution in their GPU.

In Fig. 3 we observe the speed-ups obtained relative to the sequential Java implementation. We depicted results for our smallest and biggest image sizes and for the finest and coarsest grain algorithms benchmarked. All OpenCL executions are done in the GPU of the device, whilst the operating system can decide at runtime where Renderscript is executed. In all our tests on the XU3, Renderscript executions were carried out on the CPU of the device, which turns out to be faster than the GPU. This may be due to the fact that XU3's GPU is not fully cache coherent, so the OpenCL driver reports that the system contains

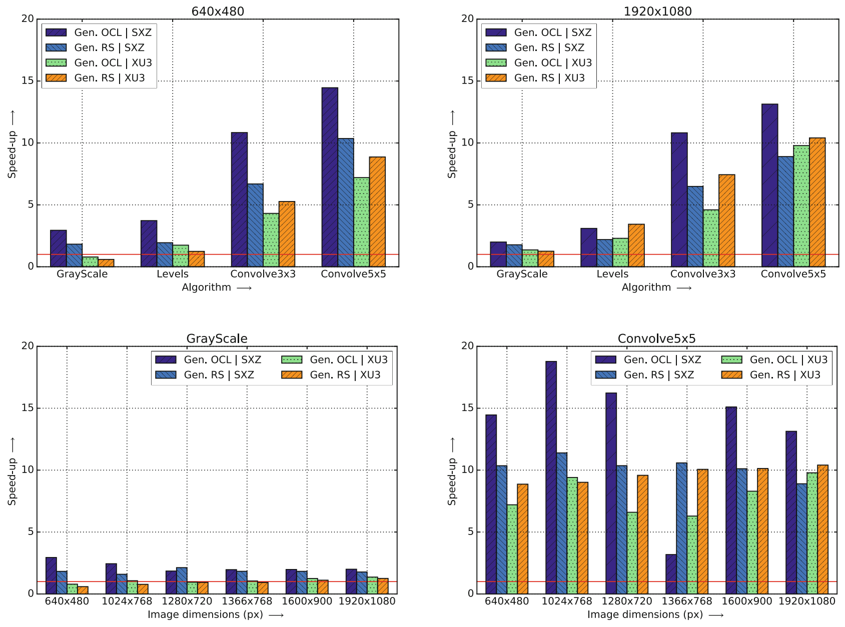


Fig. 3. Speed-up obtained with respect to the sequential Java version

two GPU devices and our generated OpenCL host code only uses one of these partitions of GPU cores. However, in almost every other case where both codes were run in a GPU our generated OpenCL code was faster.

We noticed that the input size was not as relevant as the problem's granularity regarding the performance. Coarser grain problems always experienced higher speed-ups. It is also clear from the graphs that the performance of our generated OpenCL code is more unstable than Renderscript. This could be in part due to the fact that one of the main design goals of Renderscript is to provide stable speed-ups at the expense of peak performance.

6 Conclusion

In this paper we have presented a new methodology for automatically generating OpenCL code for mobile devices. Our approach lets the developer write the whole application in a high-level programming language and, through a simple set of annotations, let the compiler take care of offloading to GPUs. Calling the offloaded code is transparent from the developer's point of view.

The Paralldroid framework has proven to ease the development of such automatic code generation tool due to its extensible design based on translator classes. It also provides us with the added value of letting the programmer choose a different target language for each class in the application, or testing and deciding the one that gives the best performance for a particular problem.

Results show that our generated OpenCL code achieves the best performance in most of the benchmarks where the GPU was used to run Renderscript computations. The differences with respect to Java code are clear, even though differences in the code are very small. Our approach greatly reduces the costs of developing high performance code for mobile devices.

There is still room for improvement in our proposal for automatic generation of OpenCL code. There are a number of optimizations that we can add to make the generated code run faster and use the available resources more efficiently:

- To reduce data transfer overheads between the CPU and accelerator devices.
- Implementing task parallelism by executing kernels asynchronously and managing a runtime dependency graph. This could improve the occupancy of the GPU when running complex heterogeneous workloads.
- Usage of a global OpenCL context shared by all generated classes. Currently we create an OpenCL context for each of the generated classes, even though it would be better to have a single set of global OpenCL objects, such as the context or the command queue, shared throughout the whole application. This could result in a smaller overhead, since all these objects refer to the same hardware.

Acknowledgement. This work was supported by the EC (ERDF), the NESUS IC1315 COST Action, the Spanish Ministry of Education and Science through the TIN2011-24598 project, and the Spanish CAPAP-H network.

References

1. Acosta, A., Afonso, S., Almeida, F.: Extending paralldroid with object oriented annotations. Parallel Comput. **57**, 25–36 (2016). http://www.sciencedirect.com/science/article/pii/S0167819116300126
2. Acosta, A., Almeida, F.: Towards a unified heterogeneous development model in android. In: 11th International Workshop HeteroPar 2013: Algorithms, Models and Tools for Parallel Computing on Heterogeneous Platforms (2013)
3. Acosta, A., Almeida, F.: Paralldroid: performance analysis of GPU executions. In: Lopes, L., et al. (eds.) Euro-Par 2014. LNCS, vol. 8806, pp. 387–399. Springer, Cham (2014). doi:10.1007/978-3-319-14313-2_33
4. Acosta, A., Almeida, F.: Performance analysis of paralldroid generated programs. In: 2014 22nd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, pp. 60–67 (2014)
5. Anandtech: AMD Outlines HSA Roadmap: Unified Memory for CPU/GPU in 2013, HSA GPUs in 2014. http://www.anandtech.com/show/5493/
6. AOSP: Android Open Source Project. http://source.android.com/
7. Apple: iOS: Apple mobile operating system. http://www.apple.com/ios
8. Dubach, C., Cheng, P., Rabbah, R., Bacon, D.F., Fink, S.J.: Compiling a high-level language for GPUs: (via language support for architectures and compilers). SIGPLAN Not. **47**(6), 1–12 (2012)
9. Google: Android mobile platform. http://www.android.com
10. Khronos Group: The open standard for parallel programming of heterogeneous systems. https://www.khronos.org/opencl/
11. Membarth, R., Reiche, O., Hannig, F., Teich, J.: Code generation for embedded heterogeneous architectures on android. In: DATE, pp. 1–6 (2014)
12. Systems, M.: Algorithmic Memory ^{TM}Technology. http://www.memoir-systems.com/
13. Microsoft: Windows Phone: Microsoft mobile operating system. http://www.microsoft.com/windowsphone
14. NVIDIA: GPUDirect Technology. http://developer.nvidia.com/gpudirect
15. NVIDIA: Tegra mobile processors: Tegra 2, Tegra 3 and Tegra 4. http://www.nvidia.com/object/tegra-superchip.html
16. Qualcomm: Snapdragon mobile processors. http://www.qualcomm.com/snapdragon
17. Samsung: Exynos mobile processors. http://www.samsung.com/global/business/semiconductor/minisite/Exynos/
18. Valentin, C., Christian, S., Pierre, K., François, K.P., Jean-François, R.: Parallel object programming with Java. http://gridgroup.hefr.ch/popj/doku.php
19. Viry, P.: Ateji PX for Java-parallel programming made simple. Ateji White Paper (2010)
20. Qian, X., Guangyu Zhu, X.F.L.: Comparison and analysis of the three programming models in Google android. In: 1st Asia-Pacific Programming Languages and Compilers Workshop (APPLC), June 2012