Performance Analysis of Paralldroid Generated Programs

Alejandro Acosta
La Laguna University, Spain
Email: aacostad@ull.es

Francisco Almeida
La Laguna University, Spain
Email: falmeida@ull.es

***Abstract*—The advent of emergent System-on-Chip (SoCs) and multiprocessor System-on-Chip (MPSocs) opens a new era on the small mobile devices (Smartphones, Tablets, ...) in terms of computing capabilities and applications to be addressed. The efficient use of such devices, including the parallel power, is still a challenge for general purpose programmers due to the very high learning curve demanding very specific knowledge of the devices. While some efforts are currently being made, mainly in the scientific scope, the scenario is still quite far from being the desirable for non-scientific applications where very few of them take advantage of the parallel capabilities of the devices. We develop a performance analysis in several SoCs using Paralldroid. Paralldroid (Framework for Parallelism in Android), is a parallel development framework oriented to general purpose programmers for standard mobile devices. Paralldroid presents a programming model that unifies the different programming models of Android. The user just implements a Java application and introduces a set of Paralldroid annotations in the sections of code to be optimized. The Paralldroid system automatically generates the native C, OpenCL or Renderscript code for the annotated section. The Paralldroid transformation model involves source-to-source transformations and skeletal programming.**

***Keywords*-Renderscript, source-to-source transformation, Android.**

I. Introduction

SoCs (Systems on Chip [1]) have been the enabling technology behind the evolution of many of todays ubiquitous technologies, such as Internet, mobile wireless technology, and high definition television. The information technology age, in turn, has fuelled a global communications revolution. With the rise of communications with mobile devices, more computing power has been put in such systems. The technologies available in desktop computers are now implemented in embedded and mobile devices. We find new processors with multicore architectures and GPUs developed for this market like the Nvidia Tegra [2] with two and five ARM cores and a low power GPU, the Qualcomm snapdragon [3], the Samsung Exynos [4] and the OMAP 5 [5] platform from Texas Instruments that also goes in the same direction.

On the other hand, software frameworks have been developed to support the building of software for such devices. The main actors in this software market have their own platforms: Android [6] from Google, iOS [7] from Apple and Windows phone [8] from Microsoft are contenders in the smartphone market. Coding applications for such devices is now easier. But the main problem is not creating energy-efficient hardware but creating efficient, maintainable programs to run on them [9].

Conceptually, from the architectural perspective, the model can be viewed as a traditional heterogeneous CPU/GPU with a unified memory architecture, where memory is shared between the CPU and GPU and acts as a high bandwidth communication channel. In the non-unified memory architectures, it was common to have only a subset of the actual memory addressable by the GPU. Technologies like Algorithmic Memory [10], GPUDirect and UVA (Unified Virtual Addressing) from Nvidia [11] and HSA from AMD [12] are going in the direction of an unified memory system for CPUs and GPUs in the traditional memory architectures. Memory performance continues to be outpaced by the ever increasing demands of faster processors, multiprocessor cores and parallel architectures.

Under this scenario, we find a strong divorce among traditional mobile software developers and parallel programmers, the first tend to use high level frameworks like Eclipse for the development of Java programs, without any knowledge of parallel programming (Android: Eclipse + Java, Windows: Visual Studio + C#, IOS: XCode + Objective C), and the latter that use to work on Linux, doing their programs directly in OpenCL closer to the metal. The first take the advantage of the high level expressiveness while the latter assume the challenge of high performance programming. Paralldroid tries to help bring these to worlds.

The Paralldroid system [13] is a development framework that allows the automatic development of Native, Renderscript and OpenCL applications for mobile devices (Smartphones, Tablets, ...). The developer fills and annotates, using his/her sequential high level language, the sections on a template that will be executed in native, Renderscript and OpenCL language. Paralldroid uses the information provided in these annotations to generate a new program that incorporates the code sections to run over the CPU or GPU. Paralldroid can be seen as a proof of concept where we show the benefits of using generation patterns to abstract the developers from the complexity inherent to parallel programs [14].

The advantages of this approach are well known:

- Increased use of the parallel devices by non-expert users
- Rapid inclusion of emerging technology into their systems
- Delivery of new applications due to the rapid develop-

ment time
- Unifies the different programming models of Android

We find the novelty of our proposal in the simultaneous generation of code for different programming models. The heterogeneity of the Android programming models allows the programmer to obtain the best performance, implementing each section of the application using the programming model that better fits to his/her code. Paralldroid allows to generate code for each programming model, facilitating the development of efficient heterogeneous applications. In this paper we present a comparative performance analysis between the generated code by Paralldroid and the adhoc implementation. We have implemented a set of testing problems with different inherent features. The analysis provides an overview of the performance obtained with a very low development effort. Several implementations have been generated for each problem, and these have been tested in two different devices.

The paper is structured as follows, in section II we introduce the development models in Android and the different alternatives to exploit the devices, some of the difficulties associated to the development models are shown. In section III we present the Paralldroid Framework, the performance of Paralldroid is validated in section IV using five different applications, transform a image to grayscale, convolve 3x3 and 5x5, levels and a general convolve implementation. Seven different versions have been compared, the ad-hoc Java, Native C and Renderscript versions, and the generated Native C and Renderscript versions. The computational results prove the increase of performance provided by Paralldroid at a low cost of development. We finish the paper with some conclusions and future lines of research.

II. The development model in Android

Android is a Linux based operating system mainly designed for mobile devices such as smartphones and tablet devices, although it is also used in embedded devices as smart TVs and media streamers. It is designed as a software stack that includes an operating system, middleware and key applications.

Android applications are written in Java, and the Android Software Development Kit (SDK) provides the API libraries and developer tools necessary to build, test, and debug applications in a Software Development Kit (SDK). The central section of Figure 1 shows the compilation and execution model of a Java Android application. The compilation model converts the Java `.java` files to Dalvik-compatible `.dex` (Dalvik Executable) files. The application runs in a Dalvik virtual machine (Dalvik VM) that manages the system resources allocated to this application (through the Linux kernel)

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 the Dalvik VM using native-code languages such as C and C++. This native code is executed using the Java Native Interface (JNI) provided by Java. The right-hand section of

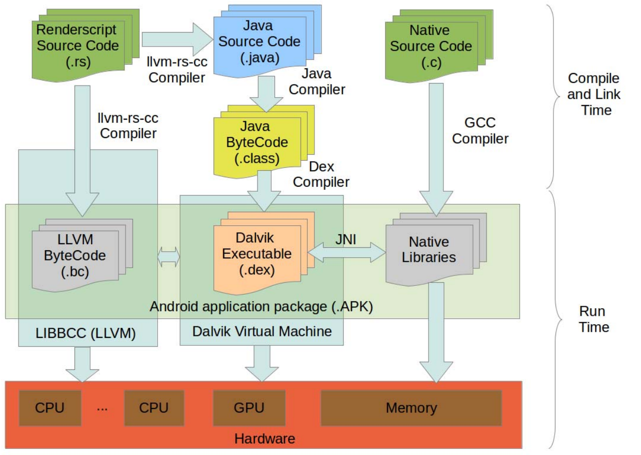


Fig. 1. Compilation and execution model of an application in Android

Figure 1 shows the compilation and execution model of an application where part of the code has been written using the NDK. The Native `.c` is compiled using the GNU compiler (GCC). The compiler used the default ARM architecture, in this case the code is optimized for ARM-based CPUs that supports the ARMv5TE instruction set [15]. Most devices support the ARMv7-a instruction set [15]. ARMv7 version extends the ARMv5 instruction set and includes support for the Thumb-2 instruction set [16] and the VFP hardware FPU instructions [17]. Note that using native code does not result in an automatic performance increase, but always increases application complexity, its use is recommended in CPU-intensive operations that don't allocate much memory, such as signal processing, physics simulation, and so on. Native code is useful to port an existing native code to Android, not for speeding up parts of an Android application. Some devices support OpenCL for executions on GPU. OpenCL code is implemented on the context of Native Development Kit (NDK).

To exploit the high computational capabilities on current devices, Android provides Renderscript, it is a high performance computation API at the native level and a programming C language (C99 standard). Renderscript allows the execution of parallel applications under several types of processors such as the CPU, GPU or DSP, performing an automatic distribution of the workload across the available processing cores on the device. The left-hand section of Figure 1 shows the compilation and execution model used by Renderscript. Renderscript (`.rs` files) codes are compiled using a LLVM compiler based on `Clang` [18], moreover, it generates a set of Java classes wrapper around the Renderscript code. Again, the use of Renderscript code does not result in an automatic performance increasing. It is useful for applications that do image processing, mathematical modelling, or any operations that require lots of mathematical computation.

III. Paralldroid

Paralldroid is designed as a framework to ease the development of future parallel applications on Android platforms. We assume that the mobile platforms will be provided with a classical CPU and with some kind of production processor like a GPU that can be exploited thorough OpenCL or Renderscript. In the proposed translation model, the developers define their problem as Java code in the Android SDK and add a set of directives. These directives are an extension of OpenMP 4.0 [19]. From this code definition, currently we can generate automatically the Native C, Native OpenCL and Renderscript code to be executed in the parallel device. The approach followed is based in a source-to-source translation process. The model implemented can be divided in three different modules (Figure 2): front-end, middle-end and back-end. These modules are integrated as a plugin into the Eclipse building process, thereby, when Eclipse builds a code our modules will be executed automatically.

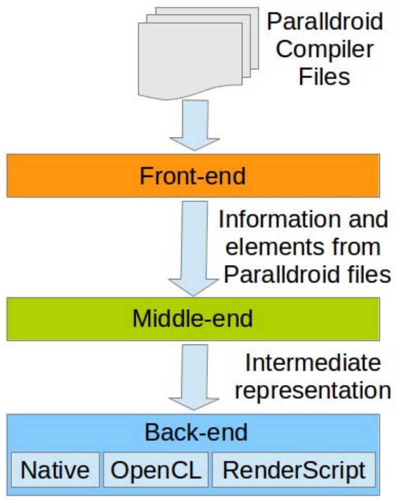

Paralldroid Compiler Files
Front-end
Information and elements from Paralldroid files
Middle-end
Intermediate representation
Back-end
Native
OpenCL
RenderScript

Fig. 2. Translation process modules.

The front-end is the first module and is responsible for checking that the code written by the user is under the Paralldroid language syntax and semantics. The syntax and semantic analysis is supported on the library Java Development Tools (JDT) [20] that allows to obtain and manipulate all the elements of a Java class (Annotations, Methods, Fields, ...) easily. The front-end module is launched during the compilation step performed by Eclipse, it invokes the middle-end module when needed and also calls to the back-end module after the generation of the intermediate code.

The middle-end takes charge of identifying directives defined by the user and analyse the Java code associated to these directives. All information and elements extracted by this module are stored using an intermediate representation that will be used by the next module. The middle-end is invoked by the front-end module by demand.

The back-end takes over the generation of the target code starting from the intermediate representation. The generation is divided in two phases, the generation of the native code and the modification of the original code to allow its access to the native code generated. To access to the native code generated, several modifications in the original code are performed. The entire process is transparent to the user. This module is invoked by the front-end after finishing with the intermediate code generation.

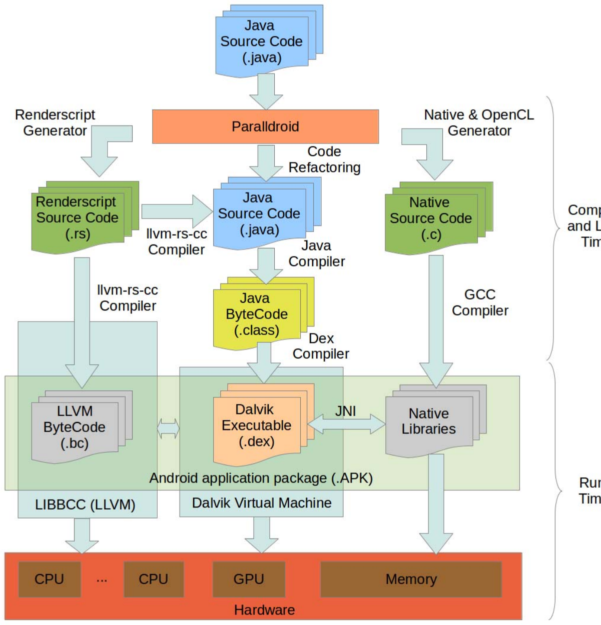


Fig. 3. The development model in Paralldroid

In Figure 3 you can see as the process of generation is integrated in the Android execution model (Figure 1). The Paralldroid generation process is in the top level and analyzes the Java code looking for directives. The files that do not contain directives are compiled directly (central section). If Paralldroid finds a directive for Native or OpenCL code generation, this code is generated and the Java code is modified to access to generated Native code (right section). The same process is used to generate Renderscript code (left section). The increment of productivity under this approach is clear, moreover when considering that Paralldroid not only generates the OpenCL or Renderscript codes but the Native C JNI implementation. The current version of Paralldroid imposes some constraints that could be overcomed in the future. We only support primitive type variables, the code associated to directives must be Java and C99 compatible.

A. *Paralldroid Directives*

The directives supported by Paralldroid are an extension of OpenMP 4.0 [21] that includes directives for accelerators. The set of directives supported by Paralldroid are:

1) Target Data: Creates a device data environment. This directive is responsible for mapping the data to the context of the device. The clauses supported are:

Listing 1. GrayScale problem with Paralldroid directives.

```
public void grayscale() {
   int pixel, sum, x;
   int [] scrPxs = new int[width*height];
   int [] outPxs = new int[width*height];
   bitmapIn.getPixels(scrPxs, 0, width, 0, 0, width, height);

   // pragma Paralldroid target lang(rs) map(to:scrPxs,width,height) map(from:outPxs)
   // pragma Paralldroid parallel for private(x,pixel,sum) rsvector(scrPxs,outPxs)
   for(x = 0; x < width*height; x++) {
      pixel = scrPxs[x];
      sum = (int)(((pixel) & 0xff) * 0.299f);
      sum += (int)(((pixel >> 8 ) & 0xff) * 0.587f);
      sum += (int)(((pixel >> 16) & 0xff) * 0.114f);
      outPxs[x] = (sum) + (sum << 8) + (sum << 16) + (scrPxs[x] & 0xff000000);
   }

   bitmapOut.setPixels(outPxs, 0, width, 0, 0, width, height);
}
```

- `lang` is an extension to the OpenMP standard, we use it to indicate the target language that we want generate: Renderscript, Native or OpenCL.
- `map` maps a variable from the current Java environment to the target data environment. We have different types of mapping:
  - `alloc` declares that on entry to the region each new corresponding list item has an undefined initial value.
  - `to` declares that on entry to the region each new corresponding list item is initialized with the original list item's value.
  - `from` declares that on exit from the region the corresponding list item's value is assigned to each original list item.
  - `tofrom` declares that on entry to the region each new corresponding list item is initialized with the original list item's value and that on exit from the region the corresponding list item's value is assigned to each original list item.

  If the programmer does not specify a map type, the default map type is `tofrom`.

2) Target: Creates a device data environment and executes the construct on the same device. This directive is responsible for mapping the data and executing the code associated to the directive in the device. The clauses have the same function as in the `Target Data` case.

3) Parallel for: Should be used in the context of a `target` directive, this directive is applied to a for loop and is responsible for distributing the load of the for loop between the threads available on the device. The clauses supported are:

- `private` indicates that each thread has a private copy of the variables.
- `firstprivate` is the same that `private` but the variables are initialized.
- `shared` indicates that all threads share the variables.
- `colapse` is used for nested loops, the load of all nested loops is distributed between available threads.
- `rsvector` is an extension to the OpenMP standard. It is used for the renderscript code generation and indicates the input and output vectors used.

4) Teams: should be used in the context of a `target` directive, this directive is responsible of teams or groups of threads. The clauses supported are:

- `num_teams` indicates the number of teams
- `thread_limit` indicates the maximum number of threads of each team.
- `private` indicates that each team has a private copy of the variables. These variables are shared between all threads in a teams.
- `firstprivate` is the same that `private` but the variables are initialized.
- `shared` indicates that all teams shared the variables.

5) Distribute: should be used in the context of a `teams` directive, this directive is similar to the `parallel for` directive but in this case distributes the load of the for loop between the teams available on the device. Clauses are similar to the `parallel for` case.

Listing 1 shows a Java implementation for the grayScale problem. This problem has a loop that traverses the pixels array of the image and gets the colour to transform to grayscale. On top of the loop, we add the Paralldroid directives to generate Renderscript code (`target lang(rs)`) and parallelize the loop ((`parallel for`)). As you can see, these directives have the OpenMP 4.0 syntax with some extension. For the definitions of the annotations we used comments instead of Java annotations because the Java annotation system just supports annotations to appear on class/method/field/variable declarations (Java Specification Request JSR 250 [22]), there is no support for the annotation of statements. Listing 2 shows the Renderscript code generated by Paralldroid. The variables

Listing 2. Generated Renderscript version of GrayScale problem.

```
#pragma version(1)
#pragma rs java_package_name(com.Paralldroid.grayscale)

int width;
int height;
int * scrPxs;
int * outPxs;

void root(const int *v_in, int *v_out, uint32_t x_lidrstadkd) {
   int x;
   int sum;
   int pixel;
   x = x_lidrstadkd;

   pixel=scrPxs[x];
   sum=(int)(((pixel) & 0xff) * 0.299f);
   sum+=(int)(((pixel >> 8) & 0xff) * 0.587f);
   sum+=(int)(((pixel >> 16) & 0xff) * 0.114f);

   outPxs[x]=(sum) + (sum << 8)+(sum << 16)+(scrPxs[x] & 0xff000000);
}
```

mapped by the `target` directive (`scrPxs, outPxs, width, height`) are defined in the Renderscript context. The loop is replaced by a root function that will be executed in parallel, private variables of the `parallel for` directive are defined inside the root function.

IV. Computational Results

Leaving aside some peculiarities associated to the real time requirements of the smartphones and tables (e.g., power management, network management), we validate the performance of the code generated by Paralldroid using five different applications. Four of these applications are based on the Renderscript image-Processing benchmark [23] (transforming a image to grayscale, to levels and convolve with convolve window of sizes 3x3 and 5x5) and the other one is an additional general convolve implementation developed by ourselves. In all cases, we implemented seven versions of code, the ad-hoc version from a Java developer, an ad-hoc Native C implementation, two ad-hoc Renderscript implementations (sequential and parallel), and the versions automatically generated by Paralldroid, the generated Native C and Renderscript codes. We executed these codes over two SoCs devices running Android, a Samsung Galaxy SIII (SGS3) and an Asus Transformer Prime TF201 (ASUS TF201). The Samsung Galaxy SIII is composed of an Exynos 4 (4412) holding a Quad-core ARM Cortex-A9 processor (1400MHz), 1GB of RAM memory and a GPU ARM Mali-400/MP4. The Asus Transformer Prime TF201 is composed of a NVIDIA Tegra 3 holding a Quad-core ARM Cortex-A9 processor (1400MHz, up to 1.5 GHz in single-core mode), 1GB of RAM memory and a GPU NVIDIA ULP GeForce. Both devices run the Android system version 4.1 with the NDK $r9$. The GPUs of these devices do not support OpenCL or Renderscript executions, so in this case, the GPUs can not be used as accelerators. For Native C implementations, the devices used supports the ARMv7-a instruction set, so we compiled the code using the ARMv7 instructions set. In all cases, the Java version will be used as the reference to calculate the speedup. For all the problems we used two images of size 1600×1067 and 800×600.

To prove the performance obtained with the generated code, we analysed the resources used (as memory or CPU) and the execution time of each problem. The results obtained by the generated code are compared to the ad-hoc versions. This allows to analyse the differences between the generated code and the ad-hoc code.

TABLE I
Execution times for the Renderscript benchmark problems (image 1600×1067) with Asus TF201

Implementation	Execution times (ms)			
	GrayScale	Levels	Convolve	
			3x3	5x5
Ad-hoc Java	336	779	2368	5747
Ad-hoc Native	102	277	974	2510
Generated Native	280	448	1158	2698
Ad-hoc Renderscript sequential	284	373	453	1087
Generated Renderscript sequential	300	390	851	1613
Ad-hoc Renderscript parallel	134	138	178	356
Generated Renderscript parallel	242	292	464	652

Tables I and II show the execution times in milliseconds for all the problems proposed on the Asus architecture. The Ad-hoc Java implementation provides an overview of each problem's granularity. We can see how the GrayScale problem has the finest granularity. For the convolve problems the granularity increases when the convolve window size is higher. The Native implementation get the best results for the finest granularity problems. When the granularity of the problems

TABLE II
EXECUTION TIMES FOR A GENERAL CONVOLVE IMPLEMENTATION (IMAGE 1600×1067) WITH ASUS TF201

Implementation	Execution times (ms)			
	General Convolve			
	3x3	5x5	7x7	9x9
Ad-hoc Java	2827	6926	12864	20933
Ad-hoc Native	920	2311	4344	7045
Generated Native	1085	2490	4502	7233
Ad-hoc Renderscript sequential	595	1223	2111	3304
Generated Renderscript sequential	859	1625	2737	4217
Ad-hoc Renderscript parallel	239	525	625	929
Generated Renderscript parallel	502	727	1013	1367

TABLE III
HEAP SIZE ON ASUS TF201

Implementation	Heap size (MB)	
	Base	Execution
Ad-hoc Java	9,75	13,42
Ad-hoc Native	9,75	13,42
Generated Native	9,75	13,42
Ad-hoc Renderscript sequential	9,75	9,75
Generated Renderscript sequential	9,75	13,42
Ad-hoc Renderscript parallel	9,75	9,75
Generated Renderscript parallel	9,75	13,42

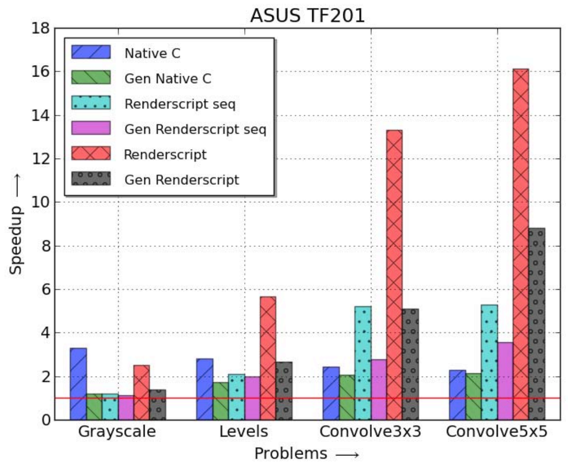
increase, the Renderscript implementations obtain the best results.

In Table III we show the Java heap memory used by the Dalvik virtual machine (Dalvik VM) for each implementation. The Base column indicates the memory used when the application is open an the image is load but the algorithm implemented is not under execution. In all cases the base memory used is the same. The Execution column collects the base memory plus the memory used on the execution of the algorithms implemented. In this case we obtain two different memory usages. The Ad-hoc Java, Ad-hoc Native and all generated versions use more memory due to the transformation of the Java object that represents the image into an array of pixels. These transformations get a best performance on the Ad-hoc Java implementations. The Ad-hoc Renderscript versions do not transform the Java object of the image and do not need extra memory. Note that, for all Renderscript versions the memory used on the Renderscript context is not represented on the Java heap memory. We do not find any reliable tool that allow measure memory used by the Renderscript context. This memory must be added to the values shown in the table.

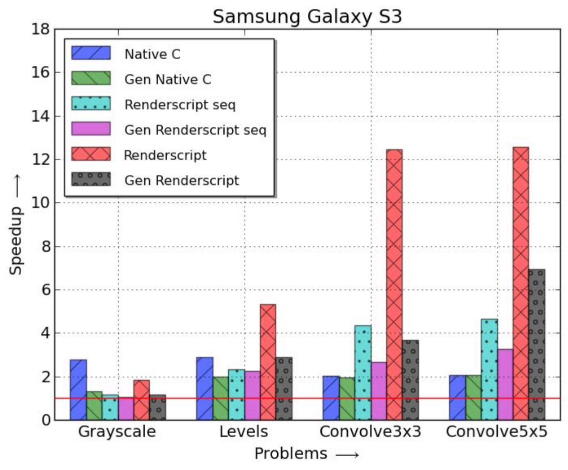
TABLE IV
CPU ACTIVITY ON ASUS TF201

Implementation	CPU			
	1	2	3	4
Ad-hoc Java	ON	OFF	OFF	OFF
Ad-hoc Native	ON	OFF	OFF	OFF
Generated Native	ON	OFF	OFF	OFF
Ad-hoc Renderscript sequential	ON	OFF	OFF	OFF
Generated Renderscript sequential	ON	OFF	OFF	OFF
Ad-hoc Renderscript parallel	ON	ON	ON	ON
Generated Renderscript parallel	ON	ON	ON	ON

Table IV shows the CPU activity when each problem is executed. If the application is open but the algorithm implemented is not under execution, the CPU activity of all cores are in a low energy state. In this state the CPU frequency decreases and some cores are offline to save energy. When the algorithms are executed, as expected, the activity of the CPUs depends on the type of execution. For the sequential executions only one core is active and the remaining of cores are on a low energy state. On the parallel executions all cores are actives.



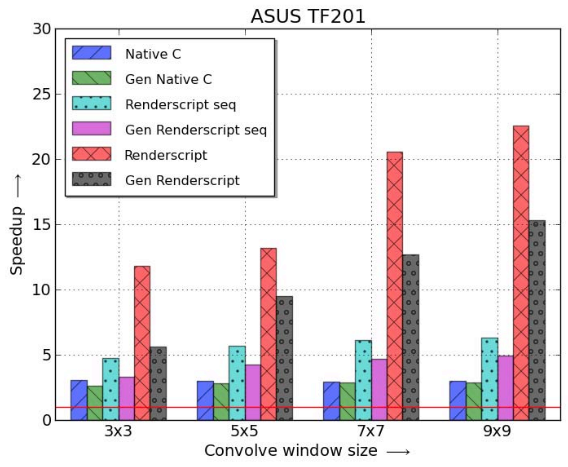
(a) ASUS TF201



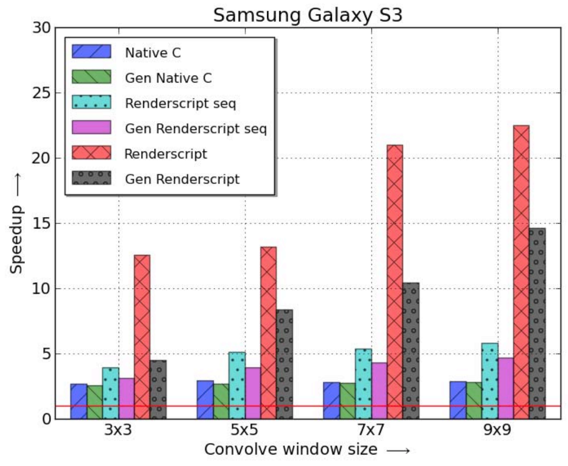
(b) Samsung Galaxy SIII

Fig. 4. Speedup for the Renderscript benchmark problems (image 1600×1067)

Figure 4 shows the speedup relative to the ad-hoc java version for the Renderscript benchmark problems using a image of size 1600×1067. The Native C versions show a big differences between the ad-hoc and generated ones. These differences disappear when the granularity of the problems increase. The ad-hoc Renderscript parallel version is faster than the ad-hoc Renderscript sequential version since the parallel version takes advantage of the quad core processor. The ad-hoc Native C version gets the best results for the finest granularity problem. This is because the Renderscript implementations introduce overhead when create the context, allocate memory and copy values to the new memory context. For the coarse granularity problems, the ad-hoc Renderscript versions get the best results since these versions are optimized and use vector operations. Currently, Paralldroid does not obtain this level of optimization but it provides a positive speedup at a low development effort. In the Renderscript executions, the computational load of the instances solved involves an important impact in the performance, problems with more computational load get a better speedup.



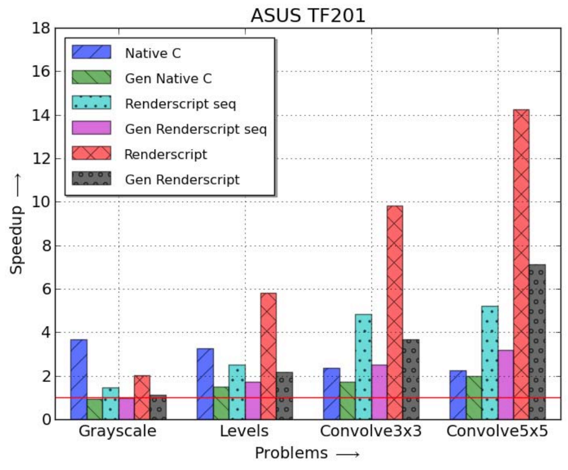
(a) ASUS TF201



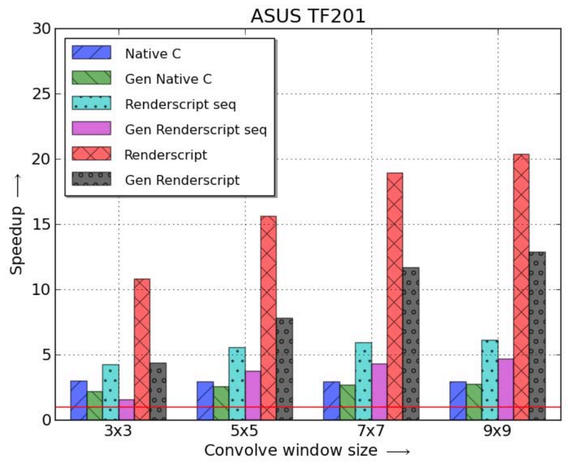
(b) Samsung Galaxy SIII

Fig. 5. Speedup for a general convolve implementation (image 1600×1067)

In Figure 5 we show the speedup relative to the ad-hoc Java version for the general convolve implementation using a image of size 1600×1067. In this case we vary the sizes of the convolve windows in the range 3×3, 5×5, 7×7 and 9×9. As in previous cases, the Native C versions provide a similar result. Again, in the generated Renderscript version we get positive speedups in all the cases but the best results are obtained with the ad-hoc Renderscript version.



(a) Speedup for the Renderscript benchmark problems



(b) Speedup for a general convolve implementation

Fig. 6. Speedup for 800×600 image size in ASUS TF201

Figure 6 shows the executions for all the problems using an image of size 800×600. In this case we only show the speedups on the ASUS TF201 device, but we experimentally tested that the running times provided by the Samsung Galaxy SIII device are similar. As on the executions performed with the image of size 1600×1067, the ad-hoc Renderscript version gets the best results for the coarse granularity problems. The Native C versions provide similar results, for the grayscale problems obtains the best result.

In general, the ad-hoc versions get higher performances but their implementations are more complex. Native C code is a good options for problems with finest granularity. When the granularity increases the best options is Renderscript.

V. Conclusion

We develop a performance analysis in several SoCs using Paralldroid. Paralldroid is a framework for the automatic generation of Native C, Renderscript and OpenCL applications for Android. The Java code annotated by the user is automatically transformed in a native C or Renderscript version. The generation process is automatic and transparent for the Java developers, the implementation details of target parallel programming language are hidden to the developer. Although there is still opportunity for the optimization in terms of the memory transfer among the different devices and in the use of vector operations, the validation tests performed on five different problems prove that the results are quite promising. With a very low development effort the running times are significantly reduced. Paralldroid also contributes to increase the productivity in the parallel developments due to the low effort required. For the near future we plan to introduce further optimizations in the Renderscript generations. We will also focus now on extending the annotations set and using Paralldroid for the parallelization of basic libraries used for Android programmers that could take advantage of the parallel execution.

Acknowledgment

This work has been supported by the EC (FEDER) and the Spanish MEC with the I+D+I contract number: TIN2011-24598

References

[1] SoCC, "IEEE International System–on–Chip Conference," Sep. 2012. [Online]. Available: http://www.ieee-socc.org/
[2] NVIDIA, "Tegra mobile processors: Tegra2, Tegra 3 and Tegra 4." [Online]. Available: http://www.nvidia.com/object/tegra-superchip.html
[3] Qualcomm, "Snapdragon mobile processors." [Online]. Available: http://www.qualcomm.com/snapdragon
[4] Samsung, "Exynos mobile processors." [Online]. Available: http://www.samsung.com/global/business/semiconductor/minisite/Exynos/
[5] Texas Instruments, "OMAP^{TM}Mobile Processors : OMAP^{TM}5 platform." [Online]. Available: http://www.ti.com/omap5
[6] Google, "Android mobile platform." [Online]. Available: http://www.android.com
[7] Apple, "iOS: Apple mobile operating system." [Online]. Available: http://www.apple.com/ios
[8] Microsoft, "Windows Phone: Microsoft mobile operating system." [Online]. Available: http://www.microsoft.com/windowsphone
[9] A. D. Reid, K. Flautner, E. Grimley-Evans, and Y. Lin, "SoC-C: efficient programming abstractions for heterogeneous multicore systems on chip," in *Proceedings of the 2008 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems, CASES'08*, E. R. Altman, Ed. Atlanta, GA, USA: ACM, Oct. 2008, pp. 95–104.
[10] Memoir Systems, "Algorithmic Memory ^{TM}technology." [Online]. Available: http://www.memoir-systems.com/
[11] Nvidia, "GPUDirect Technology." [Online]. Available: http://developer.nvidia.com/gpudirect
[12] Anandtech, "AMD Outlines HSA Roadmap: Unified Memory for CPU/GPU in 2013, HSA GPUs in 2014." [Online]. Available: http://www.anandtech.com/show/5493/
[13] A. Acosta and F. Almeida, "Towards an unified heterogeneous development model in android," in *Eleventh International Workshop HeteroPar'2013: Algorithms, Models and Tools for Parallel Computing on Heterogeneous Platforms*, 2013.
[14] I. Peláez, F. Almeida, and F. Suárez, "Dpskel: A skeleton based tool for parallel dynamic programming," in *Seventh International Conference on Parallel Processing and Applied Mathematics, PPAM2007*, 2007.
[15] ARM, "Architecture reference manuals." [Online]. Available: http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.subset.architecture.reference
[16] ——, "Thumb-2 instruction set." [Online]. Available: http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0344c/Beiiegaf.html
[17] ——, "Vfpv3 architecture." [Online]. Available: http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0344c/Beiiegaf.html
[18] CLANG, "A C language family frontend for LLVM." [Online]. Available: http://clang.llvm.org/
[19] OpenMP, "The OpenMP API specification for parallel programming." [Online]. Available: http://openmp.org/wp/openmp-specifications/
[20] Eclipse, "Eclipse Java development tools (JDT)." [Online]. Available: http://www.eclipse.org/jdt/
[21] OpenMP, "OpenMP application program interface, version 4.0." [Online]. Available: http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf
[22] JSRs: Java Specification Requests, "JSR 250: Common Annotations for the Java Platform." [Online]. Available: http://jcp.org/en/jsr/detail?id=250
[23] AOSP, "Android Open Source Project." [Online]. Available: http://source.android.com/