Towards a Unified Heterogeneous Development Model in Android$^{\text{TM}}$

Alejandro Acosta and Francisco Almeida

La Laguna University, Spain
aacostad@ull.es

Abstract. The advent of emergent SoCs and 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 propose Paralldroid (Framework for Parallelism in Android), 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.

1 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] and the OMAP$^{\text{TM}}$5 [3] platform 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 [4] from Google, iOS [5] from Apple and Windows phone [6] from Microsoft are contenders in the smartphone market. Other companies like Samsung [7] and Nokia [8] have been developing proprietary frameworks for low profile devices. 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) [11] and HSA [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, 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. The work developed in this paper tries to help bring these to worlds.

We propose the Paralldroid system, a development framework that allows for 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 [13].

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 development time and unify the different programming models of Android.

We find the novelty of our proposal in the 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.

The paper is structured as follows, in section 2 we introduce the development model in Android and the different alternatives to exploit the devices, some of the difficulties associated to the development model are shown. In section 3 we present the Paralldroid Framework, the performance of Paralldroid is validated in section 4 using five different applications, transform a image to grayscale, convolve 3x3 and 5x5, levels and a general convolve implementation. Five 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.

2 The Development Model in Android

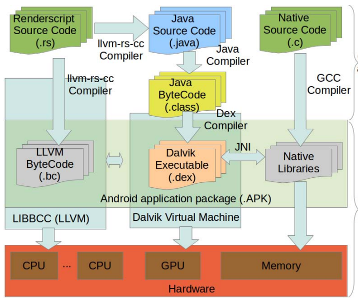
Android is a Linux based operating system mainly designed for mobile devices such as mobile phones and tablet computers, although it is also used in embedded devices as smart TVs and media streamers. It is designed as a software stack that includes an operating system, middleware and key applications.

Applications are executed in Android under a Dalvik virtual machine (Dalvik VM). The Dalvik VM executes files in the Dalvik Executable (`.dex`) format which is optimized for minimal memory footprint. The VM is register-based, and runs classes compiled by a Java language compiler that have been transformed into the `.dex` format by the included `dx` tool (see “Compile and Link time” in Figure 1(a)). The Dalvik Executable files with any data and resource files are packaged into the Android Application Package (`.apk`). All the code in a single .apk file is considered to be one application and is the file that Android-powered devices use to install the application. Android relies on Linux for core system services such as security, memory management, process management, network stack, and driver model. The kernel also acts as an abstraction layer between the hardware and the rest of the software stack.

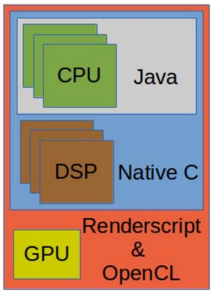
Android applications are written in Java, and the Android Software Development Kit (SDK) provides the API libraries and developer tools necessary to build, test, and debug applications in a Software Development Kit (SDK). The central section of Figure 1(a) shows the compilation and execution model of 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 VM that manages the system resources allocated to this 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 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. Using Native code may introduce benefits to certain classes of applications, in the form of reuse of existing code and in some cases increased speed. The section in the right side of Figure 1(a) shows the compilation and execution model of an application where part of the code has been rewritten using the NDK. The compilation process of the Java code is done using the SDK. The Native `.c` is compiled using the GNU compiler (GCC). More recent versions of the NDK allow the use of the Clang compiler to compile Native code, although the default is still to use the GCC [14] compiler. The Dalvik-compatible code of the application is executed in the Dalvik VM. When the Dalvik-compatible code calls a

Native routine, the VM will allocate the resources needed on the system and will allow the Native code to be executed. All the resources used by the native code are controlled by the VM. Note that using native code does not result in an automatic performance increase due to the JNI overload, but always increases the 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 the Native Development Kit (NDK).



(a) Android compilation and execution model



(b) Heterogeneity in android devices

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

To exploit the high computational capabilities on current devices, Android provides Renderscript, it is a high performance computation API at the native level (similar to CUDA) 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 section in the left side of Figure 1(a) shows the compilation and execution model used by Renderscript. Renderscript (`.rs` files) codes are compiled using `llvm-rs-cc`, a `Clang` based compiler [15] that produces byteCode (`.bc` files) for a Low-Level Virtual Machine (LLVM), moreover, it generates a set of Java classes wrapper around the Renderscript code. These Java classes, are compiled together with the rest of the Java classes of the application. To execute the Renderscript (`.bc` files) bytecode is used a LLVM (`libbcc`). 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.

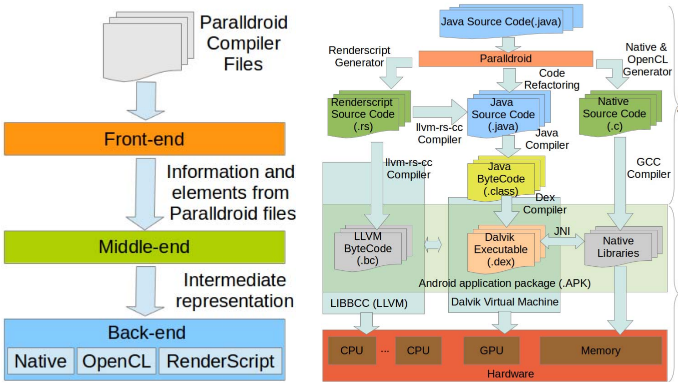
As you can see in Figure 1(b) the heterogeneity is inherent to the Android environment. The devices that support Android usually have a heterogeneous architecture composed of CPUs and a GPU and also de development model is heterogeneous. The different programming models in Android allow the programmer to obtain the best performance of each component of these devices, implementing each section of the application using the programming model that better fits to his/her code and using the components that obtain the best performance.

3 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 [16]. 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(a)): front-end, middle-end and back-end. These modules are integrated as a plugin into the Eclipse building process.

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) [17] 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 analyze 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.

In figure 2(b) you can see as the process of generation is integrated in the Android execution model (figure 1(a)). 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



(a) Translation process modules.

(b) Paralldroid compilation and execution model

Fig. 2. Translation process and Compilation model in Paralldroid

```
package es.ull.pcg.paralldroid.tests;
public class Grayscale  {
   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);
   }
}
```

Fig. 3. GrayScale problem with Paralldroid directives

JNI implementation also. 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.

3.1 Paralldroid Directives

The directives supported by Paralldroid are an extension of OpenMP 4.0, its latest release candidate [18] includes directives for accelerator devices. The set of directives and clauses supported by Paralldroid are:

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

– `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` clause maps a variable from the current Java environment to the target data environment. We have different types of map. `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`.

Target Create a device data environment and execute 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.

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 loop is distributed between available threads.
- `Rsvector` is an extension to the OpenMP standard. It is used for Renderscript code generation and indicates the input and output vectors used.

Teams should be used in the context of a `target` directive, this directive is responsible for teams or groups of threads. The clauses supported are:
- `Num_teams` indicates the number of teams
- `Num_thread` indicates the 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.

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.

Figure 3 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

```
#pragma version(1)
#pragma rs java_package_name(es.ull.pcg.paralldroid.tests)
int width, height;
int *scrPxs, *outPxs;
void root(const int *v_in, int *v_out, uint32_t x_lidrstadkd) {
   int x, sum, 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);
}
```

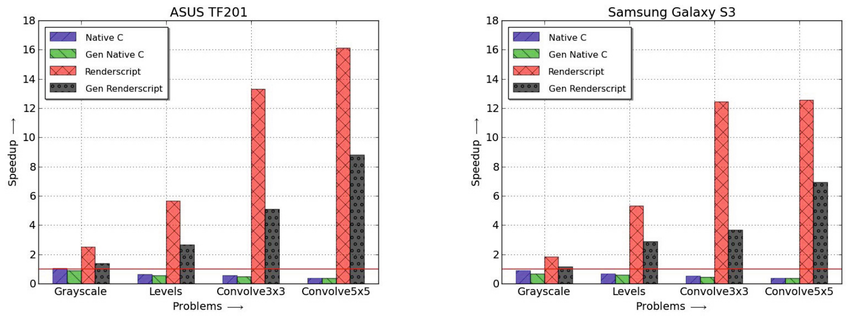
Fig. 4. Generated Renderscript version of GrayScale problem

with some extension. Figure 4 shows the Renderscript code generated by Paralldroid. The variables mapped by `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 `parallel for` directive are defined inside of root function.

4 Computational Results

To validate the performance of the code generated by our framework, we consider five different applications, four of these applications are based on the Renderscript image-Processing benchmark [19] (transforming a image to grayscale, to convolve 3x3 and 5x5 and levels) and the other one is an additional general convolve implementation developed by ourselves. In all cases, we implemented five versions of code, the ad-hoc version from a Java developer, an ad-hoc native C implementation, ad-hoc Renderscript implementation, and the versions automatically generated by Paralldroid, the generated native C and Renderscript code. 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. The GPUs of these devices do not support OpenCL or Renderscript executions, so in this case, the GPUs can not be used as accelerators. 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.

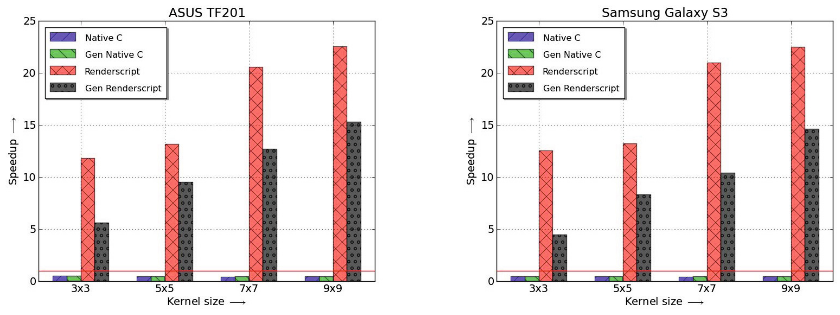
Figure 5 shows the speedup relative to the ad-hoc java version for the Renderscript benchmark problems using a image of size 1600×1067. In this case the native C versions do not present differences between ad-hoc version and generated version. In all cases, the ad-hoc Renderscript version gets the best results,



(a) ASUS TF201 (b) Samsung Galaxy SIII

Fig. 5. Speedup for the Renderscript benchmark problems (image 1600 × 1067)

due to the fact that this version 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 computacional load of the instances solved involves an important impact in the performance, problems with more computational load get a better speedup.

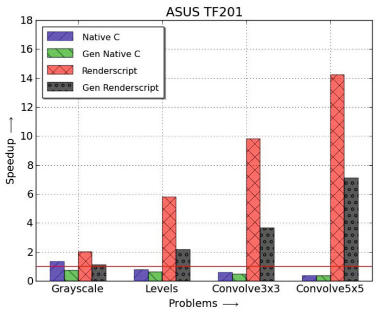


(a) ASUS TF201 (b) Samsung Galaxy SIII

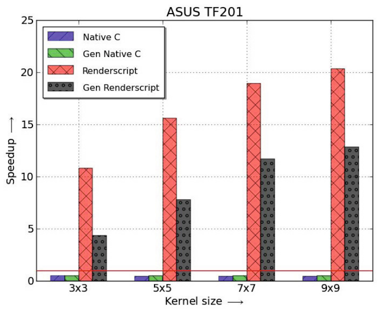
Fig. 6. Speedup for a general convolve implementation (image 1600 × 1067)

In Figure 6 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 kernels in the range 3 × 3, 5 × 5, 7 × 7 and 9 × 9. As on the previous case, 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.

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



(a) Speedup for the Renderscript benchmark problems



(b) Speedup for a general convolve implementation

Fig. 7. Speedup for 800 × 600 image size in ASUS TF201

of size 1600 × 1067, the ad-hoc Renderscript version gets the best results. The native C versions provide similar results, for the grayscale problems the ad-hoc native version get a better result than the obtained by the generated native version.

In general, the ad-hoc versions get higher performances but their implementations are more complex.

5 Conclusion

We propose Paralldroid, 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 developer that has no knowledge on parallel programming. 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 consider in our agenda extending the set of annotations supported by Paralldroid to include additional parallel directives as the parallel regions. Another future line of work is the use of Paralldroid to parallelize basic libraries used for the Android programmers so that they 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 (September 2012), http://www.ieee-socc.org/
2. NVIDIA: NVIDIA Tegra mobile processors: Tegra2, Tegra 3 and Tegra 4, http://www.nvidia.com/object/tegra-superchip.html
3. Texas Instruments: OMAP^{TM} Mobile Processors : OMAP^{TM} 5 platform, http://www.ti.com/omap5
4. Google: Android mobile platform, http://www.android.com
5. Apple: iOS: Apple mobile operating system, http://www.apple.com/ios
6. Microsoft: Windows Phone: Microsoft mobile operating system, http://www.microsoft.com/windowsphone
7. Samsung: Bada: Samsung mobile operating system, http://developer.bada.com
8. Nokia: Nokia Belle: lastest Nokia symbian platform, http://www.developer.nokia.com/
9. Reid, A.D., Flautner, K., Grimley-Evans, E., Lin, Y.: SoC-C: efficient programming abstractions for heterogeneous multicore systems on chip. In: Altman, E.R. (ed.) Proceedings of the 2008 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems, CASES 2008, Atlanta, GA, USA, pp. 95–104. ACM (October 2008)
10. Memoir Systems: Algorithmic Memory ^{TM} technology, http://www.memoir-systems.com/
11. Nvidia: GPUDirect Technology, http://developer.nvidia.com/gpudirect
12. Anandtech: AMD Outlines HSA Roadmap: Unified Memory for CPU/GPU in 2013, HSA GPUs in 2014, http://www.anandtech.com/show/5493/
13. Peláez, I., Almeida, F., Suárez, F.: DPSKEL: A skeleton based tool for parallel dynamic programming. In: Wyrzykowski, R., Dongarra, J., Karczewski, K., Wasniewski, J. (eds.) PPAM 2007. LNCS, vol. 4967, pp. 1104–1113. Springer, Heidelberg (2008)
14. Android: Native development kit (ndk), http://developer.android.com/tools/sdk/ndk/index.html
15. Clang: a c language family frontend for llvm, http://clang.llvm.org/
16. OpenMP: The openmp api specification for parallel programming, http://openmp.org/wp/openmp-specifications/
17. Eclipse: Eclipse jdt, http://www.eclipse.org/jdt/
18. OpenMP: Openmp application program interface, version 4.0-rc 2, http://www.openmp.org/mp-documents/OpenMP_4.0_RC2.pdf/
19. AOSP: Android open source project, http://source.android.com/