J Supercomput
DOI 10.1007/s11227-014-1119-8

Android™ development and performance analysis

Alejandro Acosta · Francisco Almeida

Abstract The advent of emergent systems on chip and MPSocs opens a new era in the small mobile devices (Smartphones, Tablets,…) in terms of computing capabilities and applications to be addressed. Currently, these devices have multicore processors and GPUs which provide high computational power. The efficient use of such devices, including the parallel power, is still a challenge for general-purpose programmers. In the last years Android has become the dominant platform in the small mobile devices. In addition, it has a large community of developers. For application development, Android provides two development kits, the Software Development Kit and Native Development Kit. To exploit the high computational capabilities on current devices, Android provides Renderscript, an API that allows the execution of parallel applications and it is designed to be used in applications that require high computing power. The development model used involves an important impact in the performance of the applications. In this paper, we address the evaluation of the performance on Android platforms. A set of benchmark applications has been implemented to evaluate the performance of the different development models. Sequential and parallel versions of the different development kits are considered in the computational experience. This benchmark and the computational experience achieved are greatly helpful to the programmer for understanding sources of overhead and bottlenecks in the developed code.

Keywords Renderscript · Android · SoC · Performance

A. Acosta (✉) · F. Almeida
Dept. Estadística, I.O. y Computación, ETS Ingeniería Informática,
La Laguna University, Santa Cruz de Tenerife, Spain
e-mail: aacostad@ull.edu.es

F. Almeida
e-mail: falmeida@ull.edu.es

Published online: 21 February 2014

1 Introduction

Systems on chip (SoCs [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, and the OMAP^{TM} 5 [3] 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 [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 creating efficient and maintainable programs to run on them [9] is still an unsolved problem.

Conceptually, the architectural 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 unified virtual addressing (UVA) 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.

Android is an open source platform with a very high level of market penetration and it has a large community of developers [13]. Usually the applications are developed in Java, using the development tools proposed by the platform. Although the actual Java compiler provides a performance similar to the compiler for native languages like C or C++, Android includes development tools to implement code sections of Android applications in native languages. Android also provides the Renderscript language to achieve high-performance computing in the devices.

The native C code is executed in Android using the Java Native Interface (JNI) provided by Java. Several studies have been conducted on the performance of applications using JNI [14–16], and over the increased yield obtained when using Renderscript [17–19], but all these studies do not consider the different optimization parameters available on the programming models.

In this paper, we present a comparative performance analysis between the different programming models in Android. We have implemented a set of testing problems with different inherent features. The main contribution of the paper is that the experimental analysis provides an overview on the behaviour of the programming models, so this experience can be used when eventually solving similar problems. The future developer can refer to these results to select the programming model to use according to

the problem to be implemented to obtain the best performance. Several implementations have been generated for each problem, and these have been tested in an Asus Transformer Prime TF201 device.

The paper is structured as follows: in Sect. 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 Sect. 3, we compare the performance of the different programming models in Android using different optimization parameters. One of the set of testing problems considered is based on the Renderscript image-processing benchmark (transforming an image to Grayscale, to convolve 3×3 and 5×5 and levels) and the other one is a general convolve implementation developed by ourselves. Four different versions (corresponding to different programming models in Android) have been implemented. A Java version, a Native C implementation and two Renderscript implementations (sequential and parallel). The results show how the optimization parameters affect the performance of the different programming models.

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.

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 Fig. 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) [20]. 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 section in the right of Fig. 1 shows the compilation and execution model of an application where part of the code has been written using the NDK. The Native .c code 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 [21]. Most devices support the ARMv7-a instruction set [21]. ARMv7 version extends the ARMv5 instruction set and includes support for the Thumb-2 instruction set [22] and the VFP hardware FPU instructions [22]. According to [20], 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 do not 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

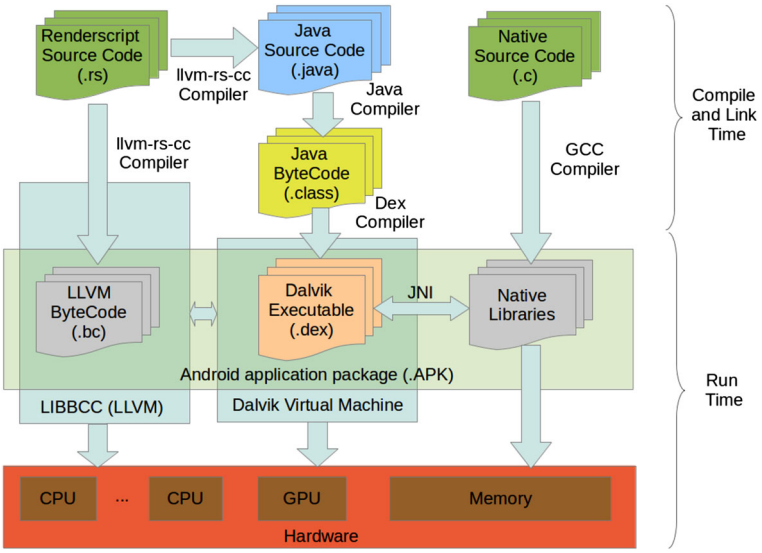


Fig. 1 Compilation and execution model of an application in Android

Android application. Some devices support OpenCL for executions on GPUs. The OpenCL code is implemented in the context of the Native Development Kit (NDK).

To exploit the high computational capabilities of current devices, Android provides Renderscript [23]; 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 section on the left of Fig. 1 shows the compilation and execution model used by Renderscript. Renderscript (.rs files) codes are compiled using a LLVM compiler based on Clang [24]; moreover, it generates a set of Java classes wrapper around the Renderscript code. Again, according to [23] the use of Renderscript code does not result in an automatic performance increase. It is useful for applications that do image processing, mathematical modelling, or any operations that require lots of mathematical computation.

3 Computational results

To compare the performance of the different programming models in Android, we consider five different applications; four of these applications are based on the Renderscript image-processing benchmark [25] (transforming an image to Grayscale, levels and to convolve 3×3 and 5×5) and the other one is an additional General convolve implementation developed by ourselves. In this case, we vary the sizes of the convolve kernels in the range 3×3, 5×5, 7×7 and 9×9. These problems are representative of a large variety of problems commonly solved when using Android. Most of the devices that support Android are designed to get information around them; the camera is one

Table 1 Execution times of the problems implemented with the default configuration

Problems	Java	Native C	Renderscript	
			Sequential	Parallel
Grayscale	286	254	144	100
Levels	650	999	338	133
Convolve				
3×3	1,975	3,503	526	195
5×5	4,775	12,287	1,365	405
General convolve				
3×3	2,337	4,492	505	195
5×5	5,779	12,096	936	323
7×7	10,768	23,195	1,542	473
9×9	17,497	37,248	2,350	686

of the most important component. For this reason, the image-processing algorithms are really important in this context. The test problems that we use here can be used as the base of more complex applications like augmented reality. Another important characteristic of the cases of study used, is the different granularity of each problem, this allows us analyse the behaviour of each programming model with different levels of granularity. In all cases, we implemented four versions of code, a Java version, a Native C implementation, two Renderscript implementations (sequential and parallel). We executed these codes on a SoC device running Android, an Asus Transformer Prime TF201 (ASUS TF201). This device is composed of a NVIDIA Tegra 3 holding a Quad-core ARM V7 Cortex-A9 processor (1,400 MHz, up to 1.5 GHz in single-core mode), 1 GB of RAM memory and a GPU NVIDIA ULP GeForce. The Android version used is 4.1 with the NDK *r*9. In all cases, the Java version will be used as the reference to calculate the speedup. For all the problems we used a image of size $1{,}600 \times 1{,}067$.

Table 1 shows the execution times in milliseconds for all the problems proposed. The Native implementation is compiled for the default ARM architecture; in this case the code is optimized for ARM-based CPUs that supports the ARMv5TE instruction set. The Renderscript executions used the default floating point precision, that in the applications follow the specifications defined by the IEEE 754-2008 standard [26]. The 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 kernel size is higher. As expected, the Renderscript implementations get the best results for all the problems.

In Table 2 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 and the image is loaded 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 Java and Native versions use more memory due to the transformation of the Java object that represents the image into an array of pixels. These transformations get

Table 2 Java heap size

Implementation	Heap size (MB)	
	Base	Execution
Java	9.75	13.42
Native C	9.75	13.42
Renderscript sequential	9.75	9.75
Renderscript parallel	9.75	9.75

Table 3 CPU activity

Implementation	CPU			
	1	2	3	4
Java	ON	OFF	OFF	OFF
Native C	ON	OFF	OFF	OFF
Renderscript sequential	ON	OFF	OFF	OFF
Renderscript parallel	ON	ON	ON	ON

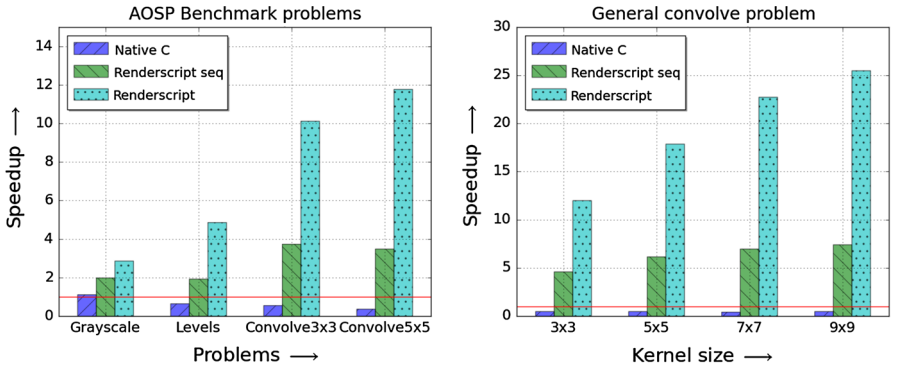
the best performance on the Java implementations. The Renderscript versions do not transform the Java object of the image and do not need extra memory. Note that, for all Renderscript versions the memory used on the Renderscript context is not represented on the Java heap memory. This memory must be added to the values shown in the table.

Table 3 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 active.

Figure 2 shows the speedup relative to the Java version for the problems proposed. In this case, the Native C versions do not obtain good results since we compiled the code for the default ARM architecture. These results can be improved using the compiler of the specific architecture. The Renderscript parallel version is faster than the Renderscript sequential version since the parallel version takes advantage of the quad core processor. In all cases, the Renderscript versions get the best results. In this case, the computational load of the instances solved involves an important impact in the performance, problems with more computational load get a better speedup.

As we show in Table 1; Fig. 2, the results obtained with the native implementation have the worst results. It has been previously mentioned that the device used supports the ARMv7-a instruction set. In what follows, we try also to analyse the impact on the running times of using specific code generations for the target architecture.

In Table 4 we compare the execution times in milliseconds of the Native-code implementations using the ARMv5 and ARMv7 instruction set. In this case, the code executed using the ARMv7 instruction set gets the best performance since the ARMv7 version extends the ARMv5 and includes support for the Thumb-2 instruction set and



(a) Renderscript benchmark problems **(b)** General Convolve problem

Fig. 2 Speedup of the problems implemented with default configuration

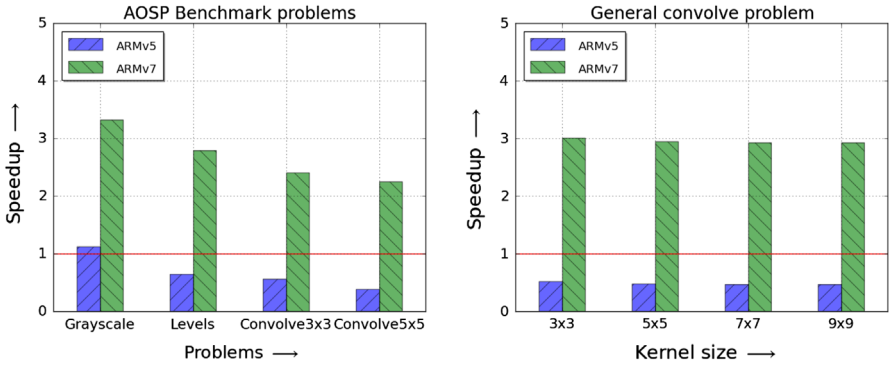
Table 4 Execution times of the native-code implementation using ARMv5 and ARMv7 instruction set

Problems	ARMv5	ARMv7
Grayscale	254	86
Levels	999	233
Convolve		
3×3	3,503	822
5×5	12,287	2,127
General convolve		
3×3	4,492	776
5×5	12,096	1,961
7×7	23,195	3,674
9×9	37,248	5,982

the VFP hardware FPU instructions. Figure 3 shows the speedup relative to the Java version for the Native-code implementation using both instruction sets. As we can see, the use of the Thumb-2 and the VFP hardware FPU instructions have an important impact in the performance. In all problems, the ARMv7 executions improved the result obtained by the ARMv5 executions.

When compiling the Native C code, we can select the version of the compiler used. By default the compiler version is GCC 4.6 but the programmers can use a different one. Table 5 shows the execution times in milliseconds of the different Native compiler versions. In this case, we compare two different compilers: the GNU Compiler Collection (GCC) and Clang. For each compiler we change the version used, 4.4.3, 4.6 (used as default), 4.7 and 4.8 for GCC compiler and 3.1, 3.2 and 3.3 for Clang. These executions are compiled for the ARMv7 architecture. The results obtained with the different versions of each compiler are similar. When we compare the two compilers, the Clang versions get the best results.

Figure 4 shows the speedup relative to the Java version for the different Native compiler versions. The Clang compiler does not present differences between the versions executed. The results obtained with the GCC compiler are similar, but in this case the differences are bigger. Using the GCC compiler, the default configuration (GCC 4.6)

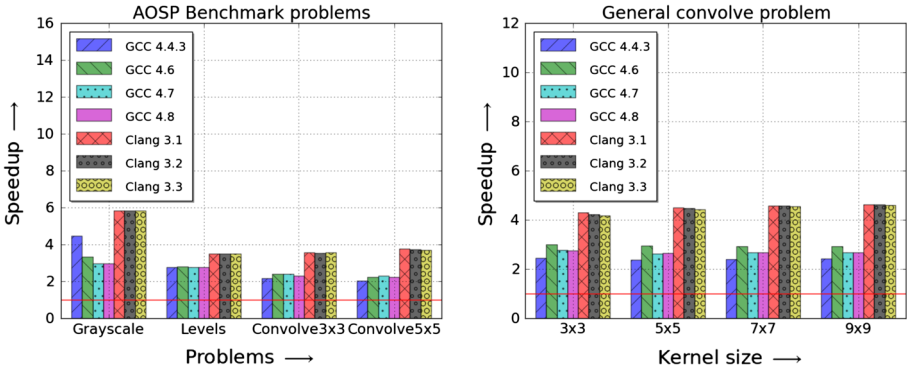


(a) Renderscript benchmark problems **(b)** General Convolve problem

Fig. 3 Speedup of the Native-code implementations using the ARMv5 and ARMv7 instruction set

Table 5 Execution times for the Native compiler versions

Problems	GCC				Clang		
	4.4.3	4.6	4.7	4.8	3.1	3.2	3.3
Grayscale	64	86	96	96	49	49	49
Levels	234	233	234	235	186	185	185
Convolve							
3×3	905	825	819	858	553	560	553
5×5	2,353	2,129	2,072	2,153	1,272	1,276	1,287
General convolve							
3×3	955	779	842	847	545	553	561
5×5	2,421	1,960	2,211	2,191	1,282	1,291	1,304
7×7	4,475	3,676	4,010	4,012	2,350	2,360	2,373
9×9	7,249	5,983	6,520	6,546	3,776	3,783	3,802

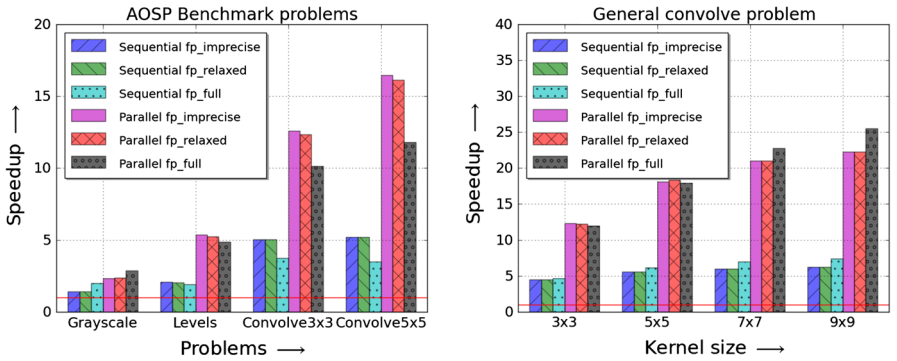


(a) Renderscript benchmark problems **(b)** General Convolve problem

Fig. 4 Speedup for the Native compiler versions

Table 6 Execution times for the Renderscript floating point options

Problems	Renderscript					
	Sequential			Parallel		
	Imprecise	Relaxed	Full	Imprecise	Relaxed	Full
Grayscale	201	200	144	122	121	100
Levels	315	316	338	121	124	133
Convolve						
3×3	392	393	526	157	160	195
5×5	917	918	1,365	290	296	405
General convolve						
3×3	522	522	505	190	191	195
5×5	1,034	1,034	936	320	315	323
7×7	1,798	1,798	1,542	512	512	473
9×9	2,818	2,818	2,350	786	786	686



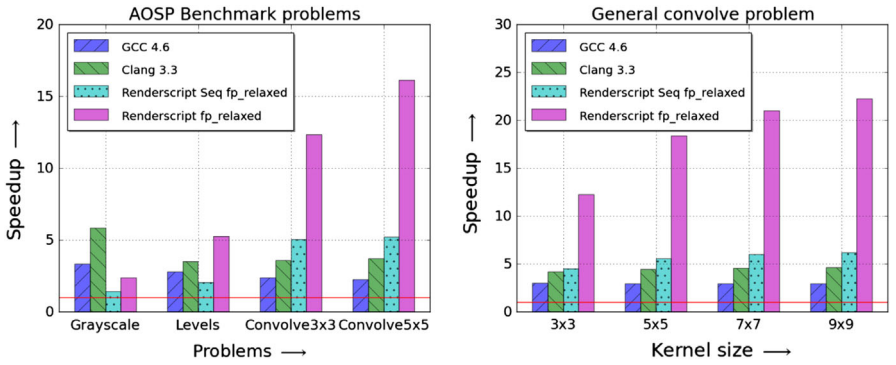
(a) Renderscript benchmark problems **(b)** General Convolve problem

Fig. 5 Speedup of the Renderscript floating point options

obtains a good performance, however, the Clang compiler presents the best speedups in all problems.

In the Renderscript executions we can change the floating point precision. It has three different floating point options. Full is the default option. The applications follow the specifications defined by the IEEE 754-2008 standard [26]. Relaxed, the applications do not require strict IEEE 754-2008 compliance and can tolerate less precision. Imprecise applications do not have stringent precision requirements. Some architectures allow additional optimizations when the Relaxed precision is active. Some applications can be used in Relaxed mode without any side effects. Table 6 shows the execution times for each floating point options in the Renderscript executions. In this case, the Renderscript benchmark problems take advantage of these optimizations and get best results when using the Relaxed mode.

In Fig. 5 we can see the speedup relative to the Java version of floating point options in the Renderscript executions. The Renderscript benchmark problems get the



(a) Renderscript benchmark problems **(b)** General Convolve problem

Fig. 6 Speedups of the optimized version of each programming model

best results using the imprecise mode, but the different between imprecise and relaxed mode is small. In these cases, the best option is to use the relaxed mode. In the general convolve problem the results are different, for the finest granularity executions (kernel size 3×3 and 5×5) the best results are obtained using the relaxed mode. Full mode gets the best results for the high granularity executions (kernel size 7×7 and 9×9).

Finally, Fig. 6 summarizes the speedups obtained with the optimized version of each programming model. The Renderscript parallel version gets the best speedup for most of the problems. For the finest granularity problem (Grayscale) the Native implementations improve the results obtained with Renderscript. When the computational load of the problems is increased, the Renderscript versions show a better performance.

4 Conclusion

We conducted a performance comparison between the different programming models available in Android. We analysed the optimization parameters of each programming model and how these parameters affect the performance. Each programming model has been analysed under different conditions using five test problems. Four of these applications are based on the Renderscript image-processing benchmark and the other one is an general convolve implementation developed by ourselves. These problems have different characteristics and cover a wide range of situations that developers can find when implementing their applications. The results show that the optimization parameters have an important impact in the performance. In the Native implementations the use of the suitable instruction set significantly improves the results. The granularity of the problems also affects the performance. For coarse-grained problems with a high computational load, the Renderscript implementations are the best options. In this case, the computational load has an important effect in the performance; the problems with more computational load got the best speedup. However, for the finest granularity problems, the Native implementations improved the results obtained by the Renderscript implementations.

Acknowledgments 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 (2014) IEEE International System-on-chip conference. http://www.ieee-socc.org/. Accessed Feb 2014
2. NVIDIA. Tegra mobile processors: Tegra 2, Tegra 3, Tegra 4 and Tegra K1. http://www.nvidia.com/object/tegra-superchip.html. Accessed Feb 2014
3. Texas Instruments. OMAP™ Mobile Processors : OMAP™ 5 platform. http://www.ti.com/omap5. Accessed Feb 2014
4. Google. Android mobile platform. http://www.android.com. Accessed Feb 2014
5. Apple. iOS: Apple mobile operating system. http://www.apple.com/ios. Accessed Feb 2014
6. Microsoft. Windows Phone: Microsoft mobile operating system. http://www.windowsphone.com/. Accessed Feb 2014
7. Samsung. Bada: Samsung mobile operating system. http://developer.bada.com. Accessed Feb 2014
8. Nokia. Nokia Asha: Nokia mobile operating system designed for low-end borderline smartphones. http://www.developer.nokia.com/. Accessed Feb 2014
9. Reid AD, Flautner K, Grimley-Evans E, Lin Y (2008) SoC-C: efficient programming abstractions for heterogeneous multicore systems on chip. In: Altman ER (ed) Proceedings of the 2008 international conference on compilers, architecture, and synthesis for embedded systems, CASES'08. ACM, Atlanta, pp 95–104
10. Memoir Systems: Algorithmic Memory™ technology. http://www.memoir-systems.com/. Accessed Feb 2014
11. Nvidia: GPUDirect Technology. http://developer.nvidia.com/gpudirect. Accessed Feb 2014
12. Anandtech. AMD Outlines HSA Roadmap: Unified Memory for CPU/GPU in 2013, HSA GPUs in 2014. http://www.anandtech.com/show/5493/. Accessed Feb 2014
13. Android. Android, the world's most popular mobile platform. http://developer.android.com/about. Accessed Feb 2014
14. Cinar O (2012) Pro Android C++ with the NDK. Apress
15. Wilson S, Kesselman J (2000) Java platform performance—strategies and tactics. Addison-Wesley, USA
16. Kurzyniec D, Sunderam V (2001) Efficient cooperation between java and native codes jni performance benchmark. In: The 2001 international conference on parallel and distributed processing techniques and applications
17. Guihot H (2012) Pro Android Apps performance optimization. Apressus Series, Apress
18. Sams J. Levels in renderscript. http://android-developers.blogspot.com.es/2012/01/levels-in-renderscript.html. Accessed Feb 2014
19. Sams J. Evolution of renderscript performance. http://android-developers.blogspot.com.es/2013/01/evolution-of-renderscript-performance.html. Accessed Feb 2014
20. Android: NDK. http://developer.android.com/tools/sdk/ndk/index.html. Accessed Feb 2014
21. ARM: Architecture reference manuals. http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.subset.architecture.reference. Accessed Feb 2014
22. ARM: VFPv3 architecture and thumb-2 instruction set. http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0344c/Beiiegaf.html. Accessed Feb 2014
23. Android: Renderscript. http://developer.android.com/guide/topics/renderscript/compute.html. Accessed Feb 2014
24. CLANG. A C language family frontend for LLVM. http://clang.llvm.org/. Accessed Feb 2014
25. AOSP: Android Open Source Project. http://source.android.com/. Accessed Feb 2014
26. IEEE. 754-2008 - IEEE Standard for Floating-Point Arithmetic. http://dx.doi.org/10.1109/IEEESTD.2008.4610935