ParallDroid: A Framework for Parallelism in Android^{TM}.

Alejandro Acosta[1] Francisco Almeida[2] Vicente Blanco[3]

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 currently are being made some efforts, mainly in the scientific scope, the scenario is still quite far from being the desirable for non-scientific applications where very few applications take advantage of the parallel capabilities of the devices. We propose ParallDroid (Framework for Parallelism in Android^{TM}), a parallel development framework oriented to general purpose programmers for standard mobile devices. Paralldroid allows the rapid development of Native Android applications. The user just implements a Java class and introduces Paralldroid annotations. The Paralldroid system automatically generates the C/C ++/OpenCL native code for this class. Paralldroid is provided as a plugin integrated in the eclipse IDE, and it works transparently to the user. The ParallDroid transformation model involves source-to-source transformations and skeletal programming. A proof of concept is presented to test the feasibility, productivity and efficiency of the approach on synthetic applications.**

Keywords— **SoC, Android, source-to-source transformation.**

I. Introduction

System on Chip (SoC [1]) has 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}4 [3] platform from Texas Instruments that Platform also goes in the same direction.

On the other hand, software frameworks have been developed to support the build of software for such devices. The main actors in this software market have their own platform: 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 it can be viewed as traditional heterogeneous CPU/GPU architecture where memory performance continues to be outpaced by the ever increasing demands of faster processors, multiprocessor cores and parallel architectures. In particular, embedded memory performance has become the limiting factor in overall system performance for SoC designs. 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.

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. 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 OpenCL parallel applications for mobile devices (Smartphones, Tablets, ...). The developer fills and annotate, using her sequential high level language, the sections on a template that will be executed in parallel. ParallDroid uses the information provided in this template to generate a new parallel program that incorporates the code sections to run over the GPU. The approach does not pose a parallelization of code in the traditional sense (for example through the loop parallelization), nor is a classical parallel skeleton, where the user fills the sequential sections of a parallel skeleton. In our case, starting from the specification given by the programmer, the parallel execution pattern is generated dynamically. 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:

[1]Dpt. Statistics and Computer Science, La Laguna University, Spain, e-mail: aacostad@ull.es

[2]Dpt. Statistics and Computer Science, La Laguna University, Spain, e-mail: falmeida@ull.es

[3]Dpt. Statistics and Computer Science, La Laguna University, Spain, e-mail: vblanco@ull.es

- 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

We find the novelty of our proposal in the application domain of our framework. Parallelism is often restricted to the scientific domain applications [14], [15], however ParallDroid is oriented to general purpose programmers developing applications demanding high performance computing that they are no necessary scientific applications. We refer, for example, to applications like those coming from augmented reality, video and image processing, etc.

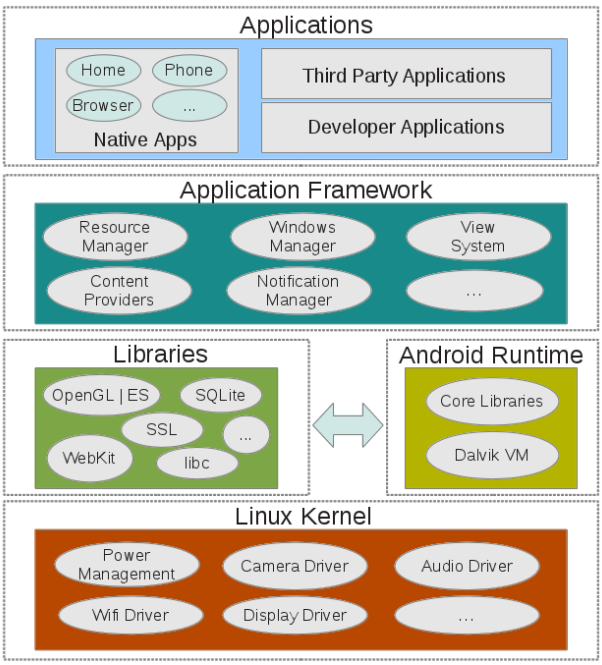
We chose to focus Paralldroid to Java developers that use Eclipse to develop for the Android that seek to exploit the GPU integrated into the device. Two main reasons lead us to take that decision: Android is an Open Source platform with a very high level of market penetration and there is has a large community of developers using Java and Eclipse as the development paradigm.

The paper is structured as follows, in section II 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 III we present the ParallDroid Framework using a synthetic application to illustrate it, the performance of ParallDroid is validated in section IV using a matrix multiplication problem and a filtering median problem on colour image data. Three different versions have been compared, the Java original version, and the C Native and OpenCL automatic generated versions. The computational results prove the increase of performance provided by ParallDroid at a low cost of development, where the parallelism is hidden to the sequential developers. We finish the paper with some conclusions and future lines of research.

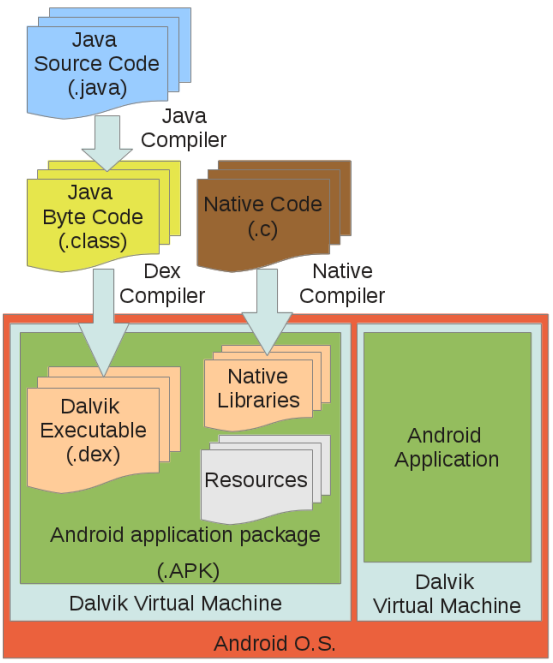
II. The develpment model in Android

Android™[16] is a software stack for mobile devices that includes an operating system, middleware and key applications (see Figure 1(a)). The Android SDK provides the tools and APIs necessary to begin developing applications on the Android platform using the Java programming language. It has a large community of developers writing applications to extend the functionality of devices.

The SDK tools compile the application code into an Android package (`.apk`) that holds the compiled bytecodes (`.dex`) that will be executed on the Dalvik processing 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 Figure 1(b)). Each application lives in its own security sandbox and by default they run in isolation from other applications. Every Android application runs in its own process, with its own instance of the Dalvik VM. Dalvik has been written so that a device can run multiple VMs efficiently. 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. The Dalvik VM is supported on the Linux kernel for underlying functionality such as threading and low-level memory management.



(a) Software stack



(b) Execution model

Fig. 1. The Android software stack and the execution model

Another available development tool is the Android Native Development Kit (NDK). NDK allows to embed components that make use of native code in An-

droid applications. The NDK enables to implement parts of the application running in the Dalvik VM using native-code languages such as C and C++ (see Figure 1(b)). This can provide benefits to certain classes of applications, in the form of reuse of existing code and in some cases increased speed. The Android framework provides several ways to use native code, we use the Java Native Interface (JNI), where developers can implement functions in native code and to load them in the Java code using the function `System.loadLibrary()`. Using native code does not result in an automatic performance increase due to the JNI overload, 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.

To exploit the high computational capabilities on current devices, the Android SDK provides RenderScript, is a high performance 3D graphics rendering computing API at the native level (similar to CUDA) and a programming C language (C99 standard). The main goal is providing the Android developers with a low level API for high performance. During the development process, the C99 code is compiled to an intermediate code that is inserted into the Android package of the application. When the application runs, the intermediate code is compiled and optimized for the device. This makes the code to be portable and improves the performance due to the native code. The RenderScript runtime would decide where the code should be executed (CPU, GPU or any other processing unit available) transparently to the programmer. Currently only the CPU is used and the load is distributed among the CPUs of the system.

The development of applications to exploit the computational capacities in these devices is a complex task. Writing native applications through the NDK requires the knowledge of the native language and the Java Native Interface (JNI) also. RenderScrip partly simplifies the development of applications, however it is still a low level API that developers must learn to use [16]. This is one of the reasons that motivated us to develop ParallDroid.

III. ParallDroid

A. The ParallDroid developing model

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. In the proposed translation model, the developers define their problem as a Java class in the Android SDK. From this class definition, we can generate automatically the OpenCL/C code to be executed in the parallel device.

Just for illustration purposes, we consider the operation of multiplying a vector by a scalar (scalar multiplication). The code in Figure 2 shows a sequential code to solve this operation using the SDK, we can see the loop of lines 12-13 in routine `scalarMultiplication` where the vector is traversed, and the goal is to process this operation in parallel. Following the OpenCL programming model, in ParallDroid each element of the vector can be computed in parallel.

```
public class ScalarMultiplication {
   public int scalar;
   public int[]vector;

   // Construct the array
   public ScalarMultiplication(int size) {
      vector = new int[size];
      // Initialize array and scalar
      // ....
   }

   public void scalarMultiplication() {
      for (int i = 0; i < vector.size(); i++)
         vector[i] = vector[i] * scalar;
   }
}
```

Fig. 2. Secuential scalar multiplication.

The code of Figure 3 shows the specification for the scalar multiplication operation in ParallDroid. The class is annotated as `@Parallel` to denote a parallel class to be processed by ParallDroid. The developer must annotate the routine `scalarMultiplication` to be launched as a Kernel with the directive `@Kernel`. The arguments of a kernel routine are doubled meaning, the number of arguments indicate the number of dimensions for the execution of the Kernel in the device, and the name of the arguments reference the thread identification when running in parallel. In this case, since we deal with a one dimensional array, the routine in line 13 receives just one argument and the variable `i` identifies the work to be done by thread `i` in the kernel. Note that, as usual in GPU programming, the loop is removed.

```
@Parallel
public class ScalarMultiplication {
   public int scalar;
   public int[]vector;

   // Construct the array
   public ScalarMultiplication(int size) {
      vector = new int[size];
      // Initialize array
      // ....
   }

   @Kernel
   public void scalarMultiplication(int i) {
      vector[i] = vector[i] * scalar;
   }
}
```

Fig. 3. ParallDroid scalar multiplication developed by end-user.

The increment of productivity under this approach is clear, moreover when considering that ParallDroid not only generates the OpenCL code but the C Native JNI implementation. Current version of ParallDroid imposes some constraints that could be over-

comes in the future. In the annotated class, we only support primitive type variables, the code inside the kernel must be Java and C99 compatible, more than 3 dimensions are not allowed in the kernel, and depending on the hardware, we may have limitations in the number of threads on each dimension.

B. The translation process

From the fields defined in the original annotated Java class, ParallDroid generates automatically the native code OpenCL/C and the Java implementation necessary to launch the native implementation from the NDK. Figure 4 shows the generation model where the native OpenCL/C file and a refactorized Java code that invokes the native code are shown. The Java development tools (JDT) provide the syntax tree (AST) of the annotated class defined by the user, the AST eases the parsing and code generation. This way, the process of code generation is transparent to the end user.

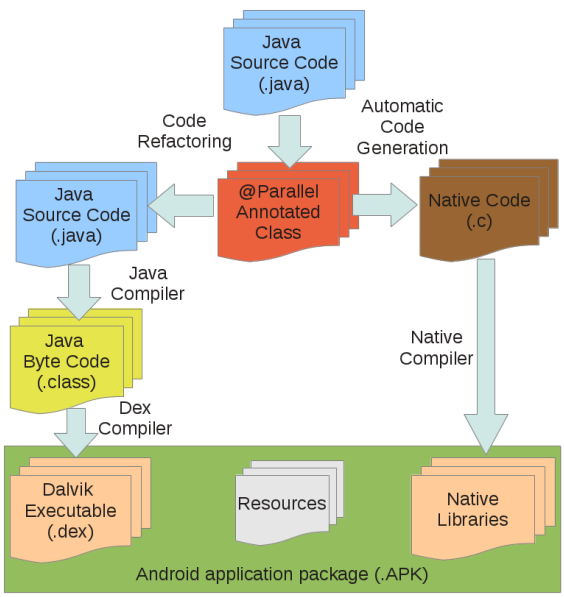


Fig. 4. The development model in ParallDroid

For the scalar multiplication example, ParallDroid generates the code presented in Figure 5, where the method `init` (line 12) allocates the memory in the native scope. Calls to the methods of lines 14-16 on Figure 6, allow getting the values holding this variables in the Java object to the native scope, and calls to lines 47-48 allows to restore the new values calculated in the native scope into the variables of the Java objects.

The translation process is supported in generic skeletal patterns that ease the code generation. In Figure 7 we show the whole translation process for the native class generation and the initialization of references for the `array` variable in the scalar multiplication example. Note how the generic pattern is instantiated with the values corresponding to the instance of generation. There are variable fields in the generic skeletal pattern that are filled with those extracted from the annotated class, in this case,

```
public class ScalarMultiplication {
   public int scalar;
   public int[]vector;

   {  // Initializer method,
      // runs before any other constructor
      System.loadLibrary("ScalarMultiplication");
   }

   // Construct the array
   public ScalarMultiplication(int size) {
      vector = new int[size];
      // Initialize array ...
      init(vector);// Generated automatically
   }

   // Code generated for the native execution
   public native void scalarMultiplication(int i);
   public native void init(int[] vector);
   public native void remove();
   protected void finalize() throws Throwable {
      remove();
   }
}
// Executing the native code
int vSize = ...;
int numThreads = vSize;
ScalarMultiplication sm = new ScalarMultiplication(vSize);
// Launching numThreads Threads
sm.ScalarMultiplication(numThreads);
```

Fig. 5. Java code generated by ParallDroid.

the name of the class (`ScalarMultiplication`), the name of the variable (`vector`) and its type (`int`).

IV. Computational Results

To validate the performance of the code generated by our framework, we consider two different applications, a classical synthetic matrix multiplication, and a median filtering of an image in the spatial domain [17]. In both cases, we implemented three versions of code, the ad-hoc version from a Java developer, ad-hoc C native implementation, and the native OpenCL code automatically generated by ParallDroid. The ParallDroid code generation is hundred percent functional in the Android Emulator, however, the OpenCL libraries are still not available for Android. For that reason, for the validation of the performance, the running times of the three versions have been measured on a system composed of a CPU Intel i3 2130 with 4 cores and a GPU Nvidia GT440.

Without loss of generality we hope that the computational results obtained here will be extrapolated to SoCs supporting Android and OpenCL since the virtual machine, the libraries and architectures will be analogous. To emulate the conditions of the Android system the running times have been measured under a Java Virtual machine. Times are expressed in milliseconds. The input/output data times have not been included in these running times, but the transference times among the different memory systems are included, i. e., the data movement between the Java Virtual Machine and the main memory for the native code, and the time between the main memory and the memory of the GPU. Labels Java, C Native and OpenCL show, in tables and figures, the running time and the ratio with the Java

```
jintArray GENvector;
// The Kernel
const char* scalarMultiplication =
    "__kernel void scalarMultiplication("
       "int scalar, __global int* vector){"
       "int i = get_global_id(0);"
       "  vector[i]=vector[i] * scalar;"
    "}";

// The public native void scalarMultiplication(int i);
void JNICALL Java_ScalarMultiplication_scalarMultiplication(
                        JNIEnv *env, jobject obj, jint i)  {
   jclass c = (*env).GetObjectClass(obj);
   // JNI to C data transformation
   jfieldID IDscalar = (*env).GetFieldID(c, "scalar", "I");
   jint scalar = (*env).GetIntField(obj, IDscalar);
   jint *vector = (*env).GetIntArrayElements(GENvector, 0);
   // Initialize OpenCL specific variables
   ...
   // OpenCL device memory for arrays
   cl_mem d_vector;
   // Initialize OpenCL:GPU devices,command-queue,memory
   d_vector = clCreateBuffer(clGPUContext,
            CL_MEM_READ_WRITE | CL_MEM_COPY_HOST_PTR,
            (*env).GetArrayLength(GENvector)*sizeof(int),
            vector, &errcode);
   // Load and build OpenCL kernel
   kernelLength = strlen(scalarMultiplication);
   clProgram = clCreateProgramWithSource(clGPUContext, 1,
            (const char **)&scalarMultiplication,
            &kernelLength, &errcode);
   clBuildProgram(clProgram, 1, clDevices, NULL, NULL, NULL);
   clKernel=clCreateKernel(clProgram,"scalarMultiplication"...);
   // Launch OpenCL kernel
   clSetKernelArg(clKernel,0,sizeof(int),(void*)&scalar);
   clSetKernelArg(clKernel,1,sizeof(cl_mem),(void*)&d_vector);
   globalWorkSize[0] = i;
   clEnqueueNDRangeKernel(clCommandQue,clKernel,1,NULL,
            globalWorkSize,NULL,0,NULL,NULL);
   // Retrieve result from device
   clEnqueueReadBuffer(clCommandQue,d_vector,CL_TRUE,0,
            (*env).GetArrayLength(GENvector)*sizeof(int),
            vector,0,NULL,NULL);
   ...
   // Free OpenCL device memory for arrays
   clReleaseMemObject(d_vector);
   // C to JNI data Transformation
   (*env).SetIntField(obj, IDscalar, scalar);
   (*env).ReleaseIntArrayElements(GENvector, vector, 0);
}
```

Fig. 6. OpenCL/C code generated by ParallDroid.

Java Annotated Code

```
@Parallel
public class ScalarMultiplication {
        public int scalar;
        public int[]vector;

        ...
}
```

Skeletal Generation Pattern

```
$className = "ScalarMultiplication";
$arrayName = "vector";
$arrayType = "int";

$generateCode = "
    j$(arrayType)Array GEN$(arrayName);

    JNIEXPORT void JNICALL Java_$(className)_init(JNIEnv *env,
                                                  jobject obj,
                                                  j$(arrayType)Array $(arrayName)) {
        GEN$(arrayName) = (j$(arrayType)Array)(*env).NewGlobalRef($(arrayName));
        (*env).DeleteLocalRef($(arrayName));
    }
";
```

Native Generated Code

```
jintArray GENvector;

JNIEXPORT void JNICALL Java_ScalarMultiplication_init(JNIEnv *env,
                                                      jobject obj,
                                                      jintArray vector) {
   GENvector = (jintArray)(*env).NewGlobalRef(vector);
   (*env).DeleteLocalRef(vector);
}
```

Fig. 7. The generation process from skeletal generation pattern

sequential version. Since the two problems are two-dimensional, the execution of the OpenCL implementation launches sequentially kernels on one of the

Size	Java	
	Time	Ratio
500x500	262	1x
1000x1000	6958	1x
2000x2000	58152	1x
Size	C Native	
	Time	Ratio
500x500	160	1.6x
1000x1000	6591	1.1x
2000x2000	57142	1x
Size	OpenCL	
	Time	Ratio
500x500	673	0.4x
1000x1000	5099	1.4x
2000x2000	27747	2.1x

TABLE I
PERFORMANCE OF THE MATRIX MULTIPLICATION

dimensions, i.e., for matrices of size n, we launch n times the kernel using n threads each. Other combinations of kernel/thread may produce different running times.

The annotated code for the matrix product can be seen in Figure 8, it implements a two-dimensional kernel where each thread is intended to calculate the value of an element `c[i][j]` from the resulting matrix. Square matrices of sizes 500, 1000, 2000 have been tested. Table I and Figures 10 show the time in milliseconds and the performance obtained. We observe that for the small problem the C Native version performs better with a ratio of *1.6x* but when the size of the problem increases the OpenCL implementation is better. A ratio of *2.1x* is obtained. This ratio seems to increase on the OpenCL code when the size of the problem is incremented.

```
@Parallel
public class MatrixMultiplication {
   private int size;
   private int[] A, int[] B, int[] C;
   ...

   @Kernel
   public void MatrixMult(int i, int j) {
      int value = 0;
      for (int k = 0; k < size; ++k) {
         int elementA = A[j * size + k];
         int elementB = B[k * size + i];
         value += elementA * elementB;
      }
      C[j * size + i] = value;
   }
}
```

Fig. 8. ParallDroid Kernel for a Matrix Multiplication.

The code in Figure 9 shows the ParallDroid kernel for the median filtering problem considering colour image data. Again it is a two-dimensional kernel, for each pixel (x, y), the filtering is applied by replacing each entry with the median of the neighbouring entries (*window*) on each dimension RGB. Image data of sizes 1024×768, 3872×2592 and windows of size 3×3, 6×6, 12×12, 24×24 have been tested. In

this case, the difficulty of the problem is incremented with the size of the problem and with the size of the window. Table II and Figures 11 show the time in milliseconds and the performance. We observe the same behaviour than in the former problems for small problems the C Native version is the best, and when the difficulty of the problem increases, the OpenCL implementation overcomes them. In this case, since we are dealing with larger sized problems the performance is incremented reaching a value of *5.4x* for the improvement ratio.

This computational experience proved that ParallDroid offers good performances with a very low cost of implementation where the parallelism is hidden to the sequential developer. As expected, in most of the cases, the C Native implementation improves to the Java implementation so, ParallDroid is an useful tool also for the automatic generation of the native code. As usual, to obtain a substantial gain from the parallel implementation the size of the problem must be large enough. At this point a comparative with RenderScript would be mandatory. However since RenderScript compiles the native code in running time a fair comparative between both environment will not be fair until the API Android/OpenCL be provided.

```
@Parallel
public class MedianFilter {
   private int window, height, width;
   private int r[], g[], b[];
   ...

   @Kernel
   public void filter(int x, int y) {
      int disp;
      int valueR = valueG = valueB = 0;
      for (int i = 0; i < window; i++) {
         disp = ((x+i)*height);
         for (int j = 0; j < window; j++) {
            valueR += r[disp+(y+j)];
            valueG += g[disp+(y+j)];
            valueB += b[disp+(y+j)];
         }
      }
      r[((x)*height)+y] = valueR/(window*window);
      g[((x)*height)+y] = valueG/(window*window);
      b[((x)*height)+y] = valueB/(window*window);
   }
}
```

Fig. 9. ParallDroid Kernel for a Filtering Problem.

Size	Execution	3x3	
		Time	Ratio
1024x768	Java	57	1x
	C Native	22	2.6x
	OpenCL	74	0.8x
3872x2592	Java	607	1x
	C Native	275	2.2x
	OpenCL	353	1.7x
Size	Execution	6x6	
		Time	Ratio
1024x768	Java	97	1x
	C Native	42	2.3x
	OpenCL	87	1.1x
3872x2592	Java	1137	1x
	C Native	542	2.1x
	OpenCL	401	2.8x
Size	Execution	12x12	
		Time	Ratio
1024x768	Java	255	1x
	C Native	131	1.9x
	OpenCL	116	1.7x
3872x2592	Java	3220	1x
	C Native	1554	2.1x
	OpenCL	601	5.4x
Size	Execution	24x24	
		Time	Ratio
1024x768	Java	680	1x
	C Native	472	1.4x
	OpenCL	258	2.6x
3872x2592	Java	8333	1x
	C Native	5212	1.6x
	OpenCL	1585	5.3x

TABLE II
Performance of the Filtering Median Problem

V. Conclusion and Future Work

We propose ParallDroid, a framework to generate parallel OpenCL applications for Android. The Java code annotated by the user is automatically transformed in a native parallel 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, the validation test performed on two 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 memory transfers. We will also focus now 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: TIN2008-06570-C04-03 and TIN2011-24598

References

[1] SoCC, "IEEE International System–on–Chip Conference," http://www.ieee-socc.org/, Sept. 2012.
[2] NVIDIA, "NVIDIA Tegra mobile processors: Tegra2 and Tegra 3," http://www.nvidia.com/object/tegra-superchip.html.
[3] Texas Instruments, "OMAP^{TM}Mobile Processors : OMAP^{TM}4 platform," http://www.ti.com/omap4.
[4] Google, "Android mobile platform," http://www.android.com.
[5] Apple, "iOS: Apple mobile operating system," http://www.apple.com/ios.

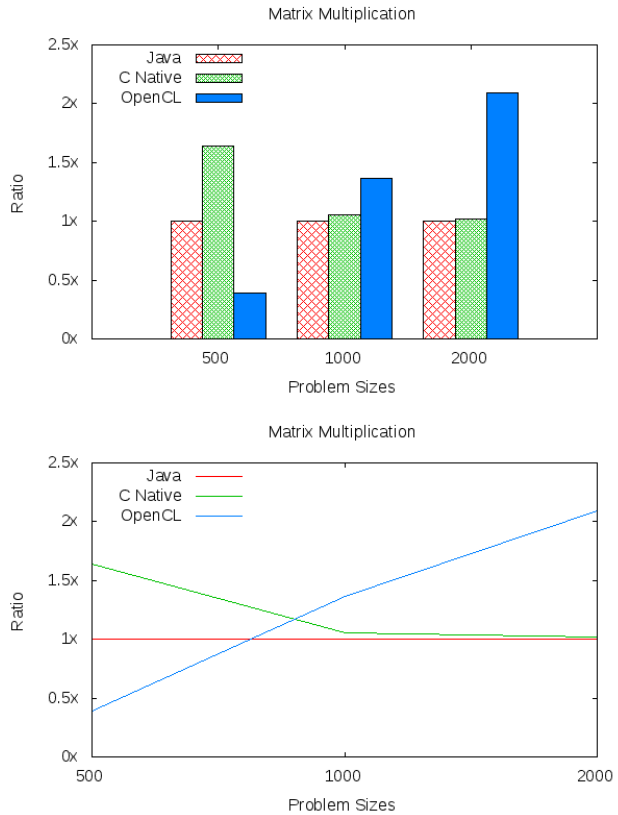


Fig. 10. Performance of Matrix Multiplication

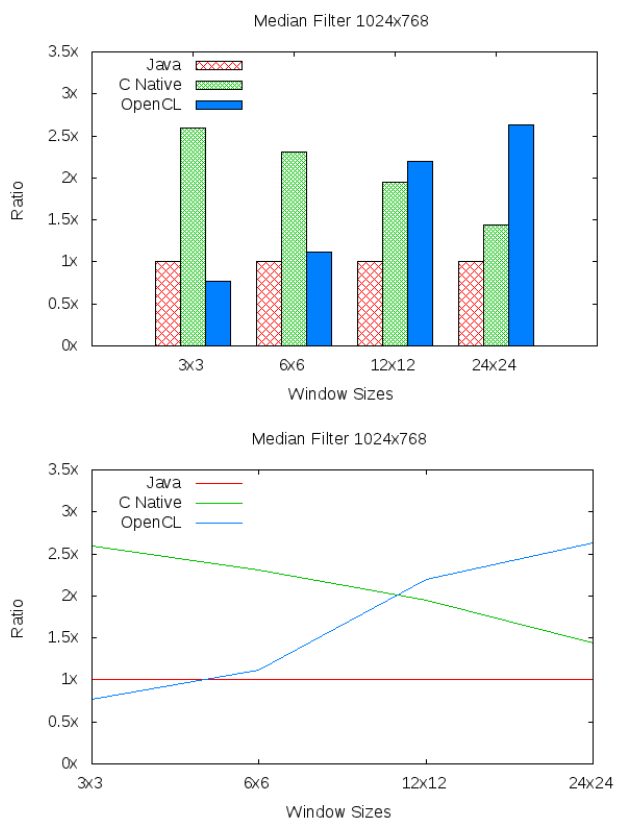


Fig. 11. Performance of Filtering Median Problem, 1024x768

[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] Alastair D. Reid, Krisztián Flautner, Edmund Grimley-Evans, and Yuan 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*, Erik R. Altman, Ed., Atlanta, GA, USA, Oct. 2008, pp. 95–104, ACM.

[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] 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.

[14] Ruymán Reyes and Francisco de Sande, "Automatic code generation for gpus in llc," *The Journal of Supercomputing*, vol. 58, no. 3, pp. 349–356, 2011.

[15] BSC, "Montblanc project: High peformance computing with embedded and mobile devices," `http://www.montblanc-project.eu/`.

[16] Google, "Android developers," `http://developer.android.com/index.html`.

[17] J. Astola and P. Kuosmanen, *Fundamentals of nonlinear digital filtering*, Electronic engineering systems series. CRC Press, 1997.