Enabling GPU support for the COMPSs-Mobile framework

Francesc Lordan[1,2], Rosa M. Badia[1,3], and Wen-Mei Hwu[4]

[1] Department of Computer Sciences, Barcelona Supercomputing Center(BSC-CNS)
[2] Department of Computer Architecture, Universitat Politècnica de Catalunya(UPC)
[3] Artificial Intelligence Research Institute, Spanish National Research Council(CSIC)
[4] Coordinated Science Lab, University of Illinois, Urbana-Champaign (UIUC)
{francesc.lordan,rosa.m.badia}@bsc.es

Abstract. Using the GPUs embedded in mobile devices allows for increasing the performance of the applications running on them while reducing the energy consumption of their execution. This article presents a task-based solution for adaptative, collaborative heterogeneous computing on mobile cloud environments. To implement our proposal, we extend the COMPSs-Mobile framework – an implementation of the COMPSs programming model for building mobile applications that offload part of the computation to the Cloud – to support offloading computation to GPUs through OpenCL. To evaluate our solution, we subject the prototype to three benchmark applications representing different application patterns.

Keywords: programming model, heterogeneous computing, collaborative computing, GPGPU, OpenCL, Mobile Cloud Computing, Android

1 Introduction

Graphical Processing Units (GPUs) employ SIMT architecture to achieve higher instruction execution rates compared to multi-core CPUs while saving energy through simpler control logic. During the last decade, heterogeneous systems combining multi-core CPU, GPU, and other accelerators have become ubiquitous thanks to the general-purpose computing on GPU (GPGPU) frameworks. Even some system-on-chips (SoCs) already have integrated them on the same die; for instance, the Qualcomm Snapdragon and the NIVIDA Tegra. Both target mobile devices where energy efficiency is a major issue and CPU computing power, highly constrained.

The most widely used programming models for developing applications for GPGPU are OpenCL [9] and CUDA [14]. Both present the hardware as a parallel platform allowing programmers to be agnostic to the actual parallel capabilities of the underlying hardware. On the one hand, these frameworks offer a multi-platform programming language to describe the computation to perform on the computing device; and, on the other hand, they provide an API to handle the parallel platform (launching computations, managing memory, and querying actual hardware details for high-performance purposes).

In this article, we propose a solution to enable applications running on a mobile device to exploit the heterogeneous resources of a distributed system. The internal computing devices within the mobile (CPU, GPU and other accelerators) and external resources in the Cloud (either nearby cloudlets or VM instances hosted on Cloud providers such as Amazon) collaborate to shorten the execution time, reduce the energy footprint and improve the user experience. For that purpose, we based our work on COMPSs-Mobile [11], an implementation of the COMPSs [12] programming model specifically designed for Mobile Cloud environments.

COMPSs is a task-based programming model that automatically exploits the parallelism inherent in an application. Developers code in a sequential fashion, being totally unaware of the underlying infrastructure and without using any specific API. At execution time, a runtime system detects the tasks that compose the application and orchestrates their execution on the available resources (local computing devices or remote nodes) guaranteeing the sequential consistency of the application.

Given that CUDA is a proprietary platform exclusive for devices equipped with the Tegra SoC – considering only embedded devices –, we opted for building our prototype on OpenCL: an open standard widely adopted by processor manufacturers, and thus, by a wide range of users. However, the proposed architecture does not lose any generality, and CUDA support could be easily added.

The contribution presented in this work consists on enabling COMPSs-Mobile applications to benefit from the computing resources within the mobile device other than the cores of CPU. For that purpose, we extend the COMPSs programming model to allow developers to declare the availability of OpenCL kernels that implement a task. Regarding the COMPSs-Mobile system, we revisit the policy to assign computing resources to each task, so it considers offloading parts of the computation to any embedded OpenCL device. To ease the interaction of the runtime system with the devices, we construct a generic computing platform leveraging on OpenCL. This platform orchestrates the execution of tasks on a settable OpenCL device; it submits the necessary commands to execute the corresponding kernels and manage the content of the memory of the device for kernels to operate on correct values. Thus, our solution hides from the programmer all the parallel platform management details (no need of invoking the API of the GPGPU framework) while the application user profits from their use. Finally, we conducted several tests to evaluate the behavior of the resulting prototype in different situations and measure the potential benefits of our proposal on Android applications.

The article goes on presenting the related work in Section 2. Section 3 introduces the extended COMPSs programming model, while Section 4 gives insights on the runtime implementation to support the execution of tasks on GPGPU devices. In Section 5, we describe the applications used for evaluating the performance of the solution and present the obtained results. Finally, to wrap up the article, we expose the conclusions and future directions of our research in Section 6.

2 Related Work

To the best of our knowledge, COMPSs-Mobile is the first framework targeting mobile devices to bring together adaptative, heterogeneous computing and computation offloading to the Cloud.

Regarding adaptative heterogeneous computing on mobile devices, Android already provides a natively integrated framework for running computationally intensive tasks at high performance: RenderScript [2]. Programming with a C99-derived language, developers write code portable across the computing devices available on the SoC. At execution time, the RenderScript toolkit parallelizes the work considering the availability of the resources (load balancing) and manages the memory. Although RenderScript achieves performances similar to OpenCL or CUDA, it can not exploit remote resources.

Beyond mobile computing, there exist other programming models/languages aiming to ease the development of task-based applications with GPU support. OmpSs [7] and StarPU [3] are two programming models that leverage on OpenMP pragmas to declare either CPU or GPU task implementations. Conversely, PaRSEC [4] allows programmers to describe the application as a DAG compactly represented in a format called JDF. For each task, JDF indicates the execution space, the parallel partitioning of the data, how the method operates on the parameters and the method to call to execute the task (allowing one CPU implementation and one for the GPU).

Regarding automatic computation offloading to Cloud resources from mobile devices, there exist several other frameworks that consider CPU task offloading. Some examples are AlfredO [16], Cuckoo [8], MAUI [6], CloneCloud [5] and ThinkAir [10]. However, they only consider CPU code offloading; developers need to deal explicitly with GPGPU frameworks to exploit the computing power of GPUs and manually balance the load across the computing devices.

Although COMPSs-Mobile does not currently offload GPU code to remote nodes, other frameworks already have implemented it. To exploit GPGPUs on mobile devices without a GPU, Ratering et al. [15] propose using virtual OpenCL devices as the interface to compute clouds. For CUDA-enabled applications, rCUDA [17] takes a driver-split approach where the driver manages all the necessary details to execute the kernels on the local or remote GPU. A complete framework for computation offloading is the result of the RAPID[13] EU project, which allows CPU and GPU code offloading; however, none of the proposed offloading frameworks automatically deals with load balancing.

3 Programming Model

COMP Superscalar (COMPSs) is a framework that aims to ease the development and execution of parallel applications atop distributed infrastructures. The core of the framework is the COMPSs Programming Model (PM) which abstracts away the parallelization and distribution concerns by offering a sequential, infrastructure-agnostic way of programming. The PM considers applications as composites of invocations to pieces of software whose execution is to be orchestrated aiming to exploit the parallelism inherent in the application. These computations are encapsulated as methods, called Core Elements (CEs) .

During application development, programmers write their code in a sequential fashion with no references to any COMPSs-specific API or the underlying infrastructure. At execution time, calls to CE methods are transparently replaced by asynchronous tasks whose execution is to be orchestrated by the runtime system. To define CE methods, developers create an interface, called Core Element Interface (CEI), where they declare those methods along with some meta-data in the form of directives. To pick a method as a CE, the programmer annotates the method declaration on the CEI with *@Method* indicating the class containing the method implementation. The code snippet in Figure 1 reproduces a simple example of a COMPSs application. Subfigure 1(a) shows the sequential code of the application which runs N simulations and selects the best one. The CEI presented in Subfigure 1(b) selects two methods to become a CE: *runSimulation* and *getBest*.

For the runtime system to determine the dependencies between CE invocations, developers specify how each CE operates on the accessed data (its parameters) by adding (*@Parameter*) directive indicating the parameter type – which can be automatically inferred at execution time – and directionality (in, out, inout). The *runSimulation* CE is a void method with no parameters that updates the content of the callee instance with the result of the simulation considering its initial value; therefore, the only datum on which the method operates is the callee. COMPSs considers the object from which the method is invoked as an implicit *INOUT* access. Conversely, *getBest* is a static method which compares two *Sim* objects and returns the one whose simulation obtained a better performance. Consequently, the developer declares the CE on the CEI with two *IN* parameters.

```
public Sim checkSimulation(int N) {
    Sim best = null;
    for (int i=0; i < N; i++) {
        Sim s = new Sim();
        s.prepareSimultation(...);
        s.runSimulation();
        best = Sim.getBest(best, s);
    }
    return best;
}
```

(a) Application main code

```
public interface SampleCEI {
    @Method(declaringClass="Sim")
    void runSimulation();

    @Method(declaringClass = "Sim")
    Sim getBest(
        @Parameter(direction = IN)
        Sim s1,
        @Parameter(direction = IN)
        Sim s2
    );
}
```

(b) Core Element Interface

Fig. 1. Sample application code written in Java

Often, several algorithms exist to achieve the same functionality with different requirements and complexity; for instance, the MergeSort and RadixSort algorithms sort a set. COMPSs supports these cases, but all the versions of the same CE need to be homonymous – *sort* – and share parameters and access patterns. To declare multiple versions for a CE, the programmer adds as many *@Method* directives as different versions and in each one indicates the implementing class as shown in the code snippet in Figure 2. The runtime creates a new task for the CE regardless the called method and selects the implementation to run according to the running host and input data characteristics.

```
@Method (declaringClass = "containing.package.RadixSort")
@Method (declaringClass = "containing.package.MergeSort")
void sort (
    @Parameter(direction = INOUT)
    int[] values
);
```

Fig. 2. Sort method CE declaration with two possible versions implemented in RadixSort and MergeSort classes respectively.

3.1 Extension for GPU support

Likewise, different versions can target different computing architectures; programmers can implement the same CE to run on a CPU core or GPU threads. To indicate OpenCL implementations of a method, programmers annotate the method declaration with *@OpenCL*. In this case, instead of pointing out the class implementing the method, programmers indicate the file (attached as an application resource) containing the OpenCL code of the kernel.

As with native language implementations, the runtime determines which version is to run and makes all the management to enable its execution. In the case of an OpenCL implementation, this includes the copy of input values into the GPU memory, the kernel invocation, the monitoring of the execution, and the collection of output values.

Unlike CPU-oriented languages, where programmers describe the computation to run on a single core, the sequential code in OpenCL and CUDA runs concurrently on several execution threads known as work-items. For each work-item to operate on a specific subset of the input/output data, they are uniquely identified according to the coordinates within a 3D grid. Developers are to specify the number of work-items through the dimensions of this grid (*global_work_size*) and the offset (*global_work_offset*) used to calculate the global ID of the work-item regarding the original coordinates. Besides, the library partitions the grid in several work-groups whose dimensions are defined as another 3D grid (*local_work_size*). Each work-item within a work-group has a local ID, and programmers can synchronize the progress of the work-items within a work-group. For the COMPSs runtime system to invoke the kernel automatically, developers indicate these three values on the CEI by adding three attributes to the *@OpenCL* directive. However, the actual value of these variables – specially *global_work_size* – may depend on the input values or its size. For that purpose, COMPSs allows simple algebraic expressions using the values and dimensions of the parameters as variables. To refer to a parameter, the developer uses the keyword *par* along with its index – starting by 0 –; for instance, *par0* would refer to the first parameter of the invocation; *par1*, to the second one; and so on so forth. If the parameter is a number, COMPSs can use its value; if the parameter is an array, it can use the value of one of its positions or its length. For multi-dimensional arrays, developers can refer to the length of any of its dimensions using the dimensional identifiers x, y and z respectively to indicate the first, second and third dimension. The default value for *global_work_offset* is (0, 0,... 0) and *NULL* for the *local_work_size*, in which case the OpenCL implementation determines how to break the global work-items into appropriate work-group instances.

Another important characteristic of OpenCL is that kernels do not return values. To work around the constraint that OpenCL kernels must be void functions, COMPSs assumes the return value, if any, to be the last parameter of the kernel; therefore, the kernel implementations of a CE with return value have an additional parameter compared to the native language implementations. As opposed to native methods, where the return value is created within the method code, the memory space for the return value of OpenCL implementations needs to be allocated prior the invocation of the kernel. The runtime is to manage the allocation of result values automatically when it decides to run an OpenCL kernel. Again, the amount of memory to allocate depends on each CE and, likely, on the input values; therefore, programmers need to specify the number of elements within each dimension of the return value with an algebraic expression. The actual number of bytes is inferred according to the return type of the declaration.

Figure 3 depicts an example of a COMPSs application performing a matrix multiplication. The actual computation of the operation is encapsulated within a CE, *multiply*, implemented either as a regular method and an OpenCL kernel. Note that, while the Java version has two parameters (A and B) and returns value, the OpenCL implementation is a void method with three parameters (a, b and c) .

4 Runtime Support Implementation

To parallelize and distribute the computation, COMPSs-Mobile replaces the CE invocations by asynchronous tasks whose execution is orchestrated by the runtime toolkit. Also, accesses to data generated on remote nodes need to fetch the value. To instrument the application, COMPSs-Mobile extends the Android application building process and adds an extra step: Parallelization. For code instrumentation, the framework leverages on Javassist [1] to replace the original Java bytecode with an instrumented version. Thus, when the user runs the application, the instrumented calls are executed and invoke the runtime toolkit.

```
package es.bsc.compss.matmul;

public class Matmul {
    public static void main(String[] args) {
        int[][] A;
        int[][] B;
        int[][] C;
        ...
        C = multiply(A, B);
        ...
    }

    public static int[][] multiply(int[][] A, int[][] B) {
        // Matrix multiplication code
        // C = AB
        ...
        return C;
    }
}
```

(a) Application Java code

```
__kernel void multiply (
    __global const int *a,
    __global const int *b,
    __global int *c)
{
    //Matrix multiplication code
    // C = AB
    ...
}
```

(b) OpenCL code in matmul.cl

```
public interface CEI {
    @OpenCL(kernel="matmul.cl", globalWorkSize="par0.x,par1.y", resultSize="par0.x,par1.y")
    @Method(declaringClass="es.bsc.compss.matmul.Matmul")
    int[][] multiply (
        @Parameter(direction = IN)
        int[][] A,
        @Parameter(direction = IN)
        int[][] B
    );
}
```

(c) Core Element Interface

Fig. 3. Example of a matrix multiplication with two implementations: one in OpenCL and one as a regular method.

4.1 COMPSs-Mobile Runtime Architecture

The main purpose of the toolkit is to orchestrate the execution of CE invocations (tasks) to fully exploit the available computing resources (local devices or remote nodes) while guaranteeing sequential consistency. Since several applications can share computing resources and data values, the runtime library consists of two parts.

On the one hand, the application-private part of the runtime controls those aspects of the execution related to the application. In other words, it detects CE invocations and creates new asynchronous tasks, monitors the private values they access (objects) and hosts the execution of the tasks. On the other hand, the orchestrator is in charge of handling all those aspects of the execution that might affect several applications; namely, accesses to shared data (files) and managing the usage of the available computing devices. While each COMPSs-Mobile application instantiates the application-private part of the runtime, there is only one single instance of the former deployed in an Android device running as an Android service on a separate process.

The *Analyzer* processes tasks upon their detection; private and public data registers identify the accessed data values and assign a unique ID to the cor-

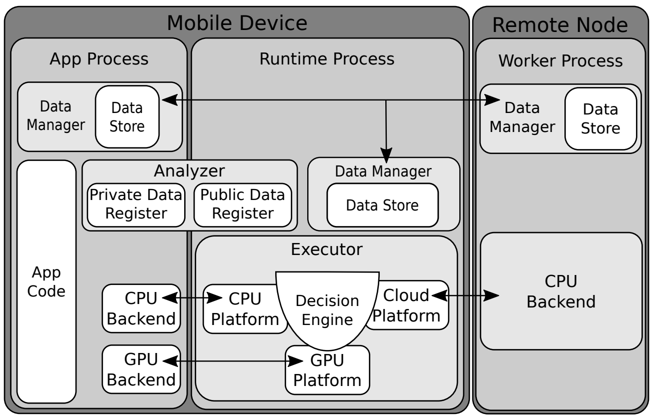

Fig. 4. Runtime system architecture

responding version. These IDs allow the runtime to detect and enforce data dependencies among tasks. After that, tasks move forward to the *Executor* for their execution. To decide which resources should host the execution, the runtime relies on the concept of *Computing Platform*: a logical grouping of computing resources capable of running tasks. The decision is made by the *Decision Engine (DE)*, which is unaware of the actual computing devices supporting the platform nor the details of their interaction. The *DE* polls each of the available platforms – configured by the user beforehand – for a forecast of the expected end time, energy consumption and economic cost of the execution. According to a configurable heuristic, the *DE* picks the best platform to run the task and requests its execution. The selected platform is responsible for monitoring the data dependencies of the task and scheduling both the execution of the task on its resources and the obtaining and preparation of any necessary value. To achieve these duties, each platform can turn to different strategies: centralizing the management on the orchestrator process, centralizing it in a remote resource or distributed across multiple resources. Regardless the approach followed to solve the scheduling, all platforms delegate the execution of tasks on a *Platform Backend* hosted off the orchestrator. For platforms handling local resources, the backend runs on the application-private part of the runtime since both the application code and any possible object are private elements of the application. Otherwise, the backend is a service running on a remote node.

In order to support data value delivery, each process hosts a common data repository, the *Data Manager (DM)*. The *DM* is asynchronous; either *Computing Platforms*, *Platform Backends* or the instrumented code of the application subscribe to the existence or value of datums (using the unique ID assigned by the *Analyzer*), and the *DM* notifies all the subscribers upon the publication of the value. The local instance of the *DM* is responsible for handling the fetching of requested values if they are located in a different process.

Figure 4 contains a diagram of the runtime architecture.

4.2 OpenCL Platform

In order to enable the execution of tasks on GPU devices, we implemented a *Computing Platform* with its corresponding OpenCL *Platform Backend* running on the application process. Each such platform maps to one computing device of an OpenCL platform, whose names are provided by the users when setting up the available platforms.

Upon the submission of a new task execution, the platform subscribes to the existence of all the input values and continues to monitor the status update from the *DM* until the task is dependency-free. To properly manage the lifecycle of several concurrent tasks, the platform has an event-based *Task Scheduler* that leverages on the out-of-order mode of OpenCL. The out-of-order mode allows OpenCL users to enqueue commands with no specific order of execution; users explicitly enforce order constraints across commands using events that OpenCL returns upon the command submission. Once the last dependency is satisfied, the platform orders its backend to fetch all the input values in a remote location through the *DM* and run the kernel.

Once the *DM* in the application process has all the input values loaded on the host memory, the backend needs to copy the input values from the host memory to the memory of the GPU device before launching the kernel on the GPU. For that purpose, it creates a memory buffer for each parameter and enqueues buffer copies for every value read by the kernel. Immediately after that, it enqueues the kernel invocation depending on the ordered copies to enforce the completion of the copies before the kernel executes. For each parameter updated during the kernel execution, the backend enqueues a memory copy command depending on the kernel execution to retrieve the new value. For detecting the end of the kernel and collecting all the outputs, the backend subscribes a listener for the kernel execution event and one for each value-to-read. Once it finishes, the backend stores the results on the local *DM*.

To better exploit locality, the backend monitors the content of the device memory. By keeping track of the buffer containing each data value and the writing event, the backend can discover the existence of another buffer with the value. Using the existing buffer as a parameter of the kernel and enforcing its execution to wait upon the corresponding writing event, the scheduler avoids the overhead of creating and filling a new buffer. Hence, the backend notices the existence of those values computed on the device when the producing kernel invocation is enqueued and internally bypasses the existence notification of the *DM* to hand over the scheduling of the kernel to OpenCL.

We expect COMPSs-Mobile applications to have a high degree of parallelism and a sufficient number of coarse-grain tasks so that data transfers (from the remote nodes and to the GPU memory) can overlap with the execution of other tasks. When generating a forecast of completion time for tasks, the runtime considers the execution time and the wait for resources (memory allocation for result values happens during this waiting period, and the actual transfer time is negligible). Cost is only related to the number of bytes transferred from remote nodes. The energy model considers the consumption during the execution and the energy spent on transfers from remote nodes; transfers from and to the device memory are also negligible. The platform uses the statistical data from the profiling of previous executions to predict the forecasts.

5 Performance Evaluation

To validate our proposal, we ported three applications to Android following the COMPSs programming model: Digits Recognition (DR), Bézier Surface (BS) and Canny Edge Detection (CED). DR is a Convolutional Neural Network trained to recognize digits out of hand-written numbers. The algorithm applies eight processing steps to a set of images. We merge the processing of all the images on each step within a task; thus, the application becomes a sequence of tasks. BS is a mathematical spline that interpolates a surface given a set of control points. The application splits the output surface, and each task computes the result values within a chunk independently of each other. Finally, CED is an image-processing algorithm for edge detection where each frame goes through a four-stage process (Gaussian filter, Sobel filter, non-maximum suppression and hysteresis) each one encapsulated within a CE. We apply the algorithm to 30 frames of 354x626 pixels producing a workload composed of 30 parallel chains of four tasks. We selected these applications and implementations because of the diversity of patterns presented by their workloads as shown in Figure 5.

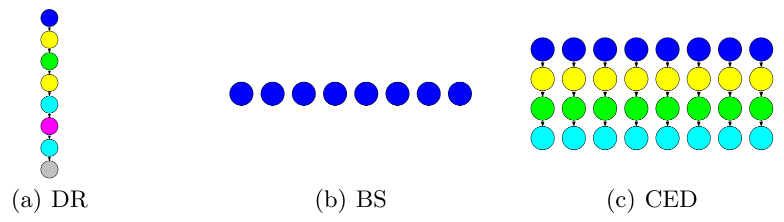
(a) DR (b) BS (c) CED

Fig. 5. Applications' dependency graphs

The experiments to evaluate the behavior of our prototype run on a OnePlus One smartphone equipped with a Qualcomm SnapDragon 801 processor (a Krait 400 quad-core CPU at 2.5 GHz and an Adreno 330 GPU). We appraise different configurations using two Computing Platforms operating on the local resources: the CPU Platform using the cores of the CPU (varying the number of available ones) and the OpenCL Platform leveraging on the Adreno device.

5.1 OpenCL Platform Performance

The first test aims to check the proper behavior of the OpenCL platform and evaluate the impact of the implemented optimizations. For that purpose, we executed the three applications considering six possible scenarios: CPU, GPU, R1CPU, R4CPU, RGPU, RGPUO. The CPU and GPU scenarios execute an Android-native version of the application; while CPU runs the application sequentially on the CPU, GPU naively[5] offloads the computation to the GPU through OpenCL. On the remaining four scenarios, the developer codes the application following the COMPSs programming model and the final user sets up the runtime to force the runtime to execute on a specific computing platform. On R1CPU and R4CPU, the runtime uses only the CPU platform exploiting one and four cores respectively. On RGPU and RGPUO, the runtime offloads all the tasks to the GPU through the OpenCL platform. The former disables all the optimizations obtaining a behavior similar to the GPU scenario, while the latter enables all the optimizations (reusing memory buffers and overlapping transfers with other kernel executions).

For each scenario, we measured the execution time and its energy consumption. Within the execution time, we distinguish the amount time spent on the execution of tasks (*Tasks*) from the overhead surrounding the computation (*Overhead*). This experiment focuses on isolating the part of this overhead corresponding to transfers between main and devices memories (*Ov. Mem.*) to evaluate the benefits of the optimizations implemented on the GPU backend. Regarding the energy consumption, we only separate the energy used for computing the tasks (*Tasks*) from the energy consumed by the whole system including the screen (*System*).

Digits Recognition Charts in Figure 6 depict the results obtained from processing 512 images with the Digits Recognition application. It is plain to see that GPU allows a significant improvement both on time and energy regardless of using COMPSs. Comparing CPU to GPU scenarios, the execution time shrinks from 18,516 ms to 4,358 ms (23.53%) – 1,531 ms of which correspond to memory transfers –; and the energy consumption, from 36.48 J to 8.68 J (27.8%). R1CPU and R4CPU present a behavior similar to the CPU case since the application has no task-level parallelism; however, on both cases, the runtime incurs a negligible overhead (31 ms and 0.02 J) caused by the inter-process communication among the runtime components. Likewise, the overhead appears on both scenarios where the runtime uses the GPU. Besides this overhead, the application performs as on GPU when the platform optimizations are disabled. When enabled, the runtime reuses the memory values generated by one task as the input of the succeeding one; thus allows to reduce the overhead of data copies from and to the device memory from 1,531 ms to 5 ms. The optimizations implemented for the management of the device memory allow COMPSs-Mobile to speed up the execution of the application on GPUs even when they have no task level parallelism. Despite the improvement on the execution time, these optimizations have a low impact on the energy consumption (0.56 J) since the source of the most significant part of it is the actual computation of the kernels.

[5] OpenCL commands are synchronous, and all the input and output data is copied to and from the device memory on every kernel execution

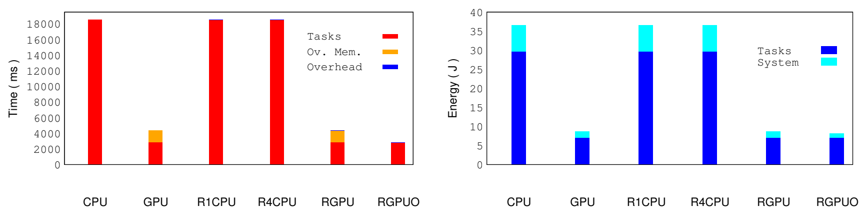

Fig. 6. Execution time (left) and energy consumption (right) obtained from the Digits Recognition runs

Bézier Surface Interpolling a surface of 1024 x 1024 points using 256 x 256 blocks with the Bézier Surface application presents results similar to DR as shown in Figure 7. Although GPU computes the tasks 2.99 times faster than the CPU (2,672 ms vs. 7,984 ms), the memory transfers overhead (337 ms) slows down the application. It only achieves a 2.65x lower execution time (3,009 ms) and a 50.45% reduction of the energy consumption (15.73 J vs. 7.8 J). As with DR, the runtime incurs a little overhead (39 ms and 0.02 J) when comparing CPU to RCPU and GPU to RGPU.

Unlike DR, tasks in BS have no dependencies; thus, the runtime can exploit the parallelism and use the four cores of the CPU at a time speeding up the execution of the kernels up to 2.72x (2,939 ms). The reduction of the CPU frequency to control the temperature of the processor and the thread oversubscribing with the runtime threads separates the obtained performance from the optimal. These measures increase the energy consumption of the tasks which grows from 15.74 J to 19.65 J. Since BS tasks have no dependencies, they never read values generated by other tasks; therefore, the runtime cannot reuse values already transferred for preceding tasks. However, the computation of one task can overlap with the transfers of output/input values of the preceding and succeeding ones. This optimization allows the runtime to reduce the time spent on memory transfers from 337 ms to 3 ms on the RGPUO scenario. On the RGPUO scenario, BS lasts 2,714 ms and consumes 7.68 J.

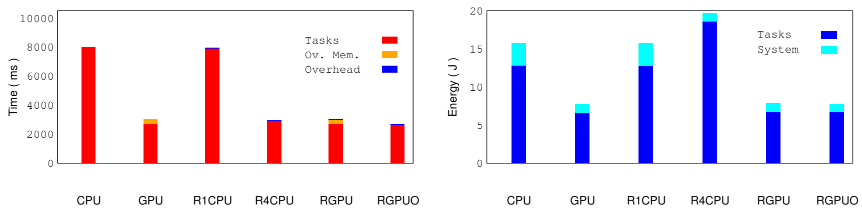

Fig. 7. Execution time (left) and energy consumption (right) obtained from the Bezier Surface runs

Canny Edge Detection In this case, the GPU device processes the 30 frames in 420 ms, 11.95x faster than the CPU; and again, the data transfers worsen the application performance adding a 324 ms overhead. In overall, the application takes 5,020 ms to run in the CPU scenario and consumes 9.39 J; while for the GPU case, it needs 744 ms and 1.33 J respectively. The runtime adds an overhead of 34 ms and 0.02 J slightly noticeable when comparing CPU and GPU to R1CPU and RGPU, respectively.

In this case, the application presents task-level parallelism and dependencies among tasks; thus, the GPU can apply both optimizations. The GPU reuses the output of some tasks as the input of its successors; thus, the runtime reduces the number of transfers. Besides, the remaining transfers can overlap with the computation of other dependency-free tasks. Enabling these optimizations allows the runtime to reduce the 324 ms overhead caused by memory transfers to 1 ms. On the RGPUO scenario, the application lowers the execution time to 455 ms and its energy consumption to 1.22 J.

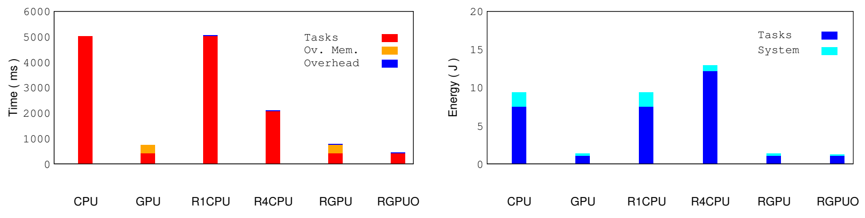

Fig. 8. Execution time (left) and energy consumption (right) obtained from the Canny Edge Detection runs

5.2 Load balancing policies

The second experiment studies the impact of extending the resource-assignment policies on the execution time and energy consumption of the application. For that purpose, we run the COMPSs-Mobile version of each application with different task granularity using every possible combination of resources. For the heterogeneous scenarios - i.e., using both computing platforms -, we compare the results of three different policies: Static, DynPerf and DynEn. Static is a predetermined load distribution that mimics what application developers could easily devise to minimize the execution time. The load arrangement employed on each execution depends on the application workflow, the number of tasks and the time they require to run on each device; further details on the division applied on each application are provided on the corresponding subsection. With the same purpose, the DynPerf policy automatically decides which computing platform executes the task according to the earliest end time forecasted by the platforms. Conversely, DynEn aims to find a balance between reducing the execution time and the additional energy that it incurs. For that purpose, the policy takes into account not only the end time of the task but also the energy spent on its processing; the policy would pick a later end time if for each sacrificed ms the application can save 5 mJ.

Digits Recognition DR is an application where a set of images go through a 7-stage process. Each stage is encapsulated in a task; thus, their granularity depends on the number of images to process. In this experiment, we use three different input sets composed of 128, 256 and 512 images. Since DR has no task-level parallelism, we dismiss all those configurations using more than one core of the CPU. All the CEs that compose the application take less time and energy to run on the GPU device than on the CPU; therefore, the Static policy for this application consists of submitting all the tasks to the GPU.

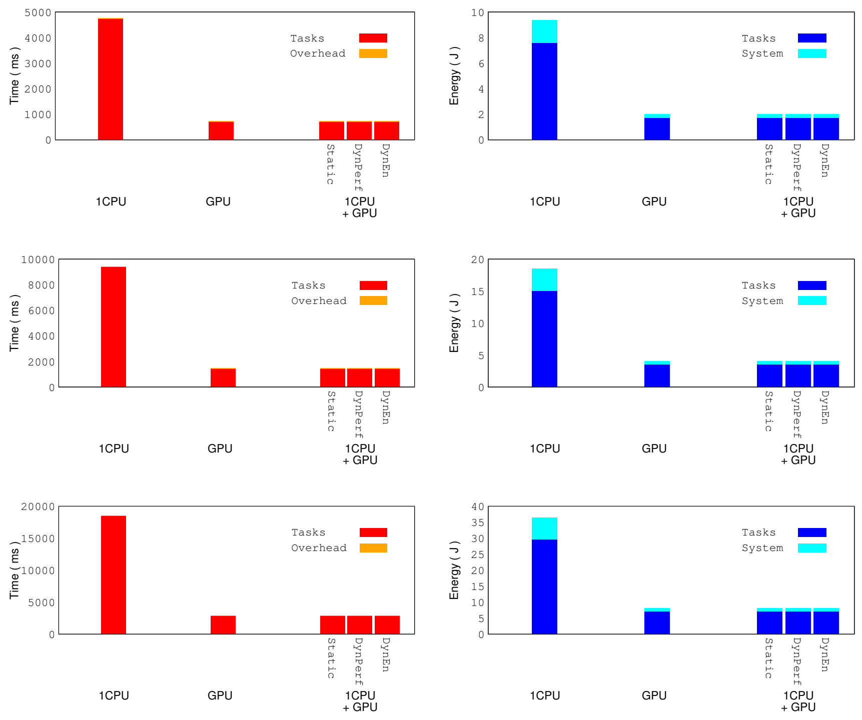

Fig. 9. Execution time (left) and energy consumption (right) for Digit Recognition runs using 128, 256 and 512 images (from top to bottom)

Charts in Figure 9 show the execution time (left) and energy consumption (right) when processing 128, 256 and 512 images (from top to bottom). Despite the difference in the magnitude of the values, the application behaves alike regardless the input size. As we double it, almost does so the execution time and the energy consumption whether if the application runs on the CPU (4,762 ms and 9.333 J for 128 images; 9,410 ms and 18.490 J for 256 images, and 18,547 ms and 36.493 J for 512 images) or on the GPU (731 ms and 2.015 J, 1,446 ms and 4.065 J, and 2,862 ms and 8.127 J respectively for processing 128, 256 and 512 images). Given that the GPU is faster and less energy-consuming than the CPU and that the application presents no task-level parallelism, submitting all the executions to the GPU is the optimal solution either from the performance or the energy point of view. Hence, both dynamic policies schedule all the executions to the GPU as expected. Despite all the employed configurations use the COMPSs-Mobile runtime which incurs an overhead, it is important to notice that dynamically deciding where to run a task adds no significant overhead compared to those cases where the runtime handles a homogeneous system or the decision is statically set beforehand.

Bézier Surface BS is an application whose task-granularity and parallelism depends on the partitioning of the output. For this experiment, the application computes a fixed-size surface of 1024x1024 points varying the size of the chunk computed by a task from a 1024x1024 block – 1 task –, through 256x256 – 4 tasks – and 512x512 blocks – 16 tasks–, right up to blocks of 128x128 points – 64 tasks. Figure 10 depicts the execution time (left) and energy consumption (right) of running the application with the four granularities (top to bottom). Considering the number of tasks, the number of CPU cores and the ratio between the time to run a task on a GPU and a CPU – the more CPU cores are used, the higher the speedup is; 3x, 3.4x, 3.9 and 4.3x respectively for using 1, 2, 3 and 4 cores –, it is easy for the application developer to find the number of tasks to assign to each computing device to minimize the application execution time. For instance, in the case of a 128x128 block size output using a single core of the CPU, the speedup is 3.03x; thus, the optimal load balancing from a temporal point of view is to run 48 tasks on the GPU while the CPU core processes 16. For the Static policy in this experiment, we assume the application developer to be fully aware of the number of CPU cores to use, the granularity of the task and the corresponding speedup and code the application to balance the load using this knowledge.

From a temporal point of view, the Static policy balances the load in such a way that the execution time is minimal. As with DR, DynPerf behaves like Static in all executions (as expected) achieving the optimal performance with no significant overhead due to taking the decision dynamically. Regarding energy consumption, running all the tasks on the GPU is the optimal solution in all four cases (7.825 J, 7.741 J, 7.684 J and 7.538 J respectively for 1024, 512, 256 and 128). The cause of this reduction in the energy is the better performance of the GPU when processing smaller chunks – 2,692 ms to compute the surface in one single block vs. 2,621 ms to compute 64 blocks, 40.96 ms each –; the CPU behaves alike – 8,035 ms vs. 7, 934 ms.

For those cases with a coarse granularity, the low number of tasks and the big difference in the energy consumption of the computing devices lead the DynEn

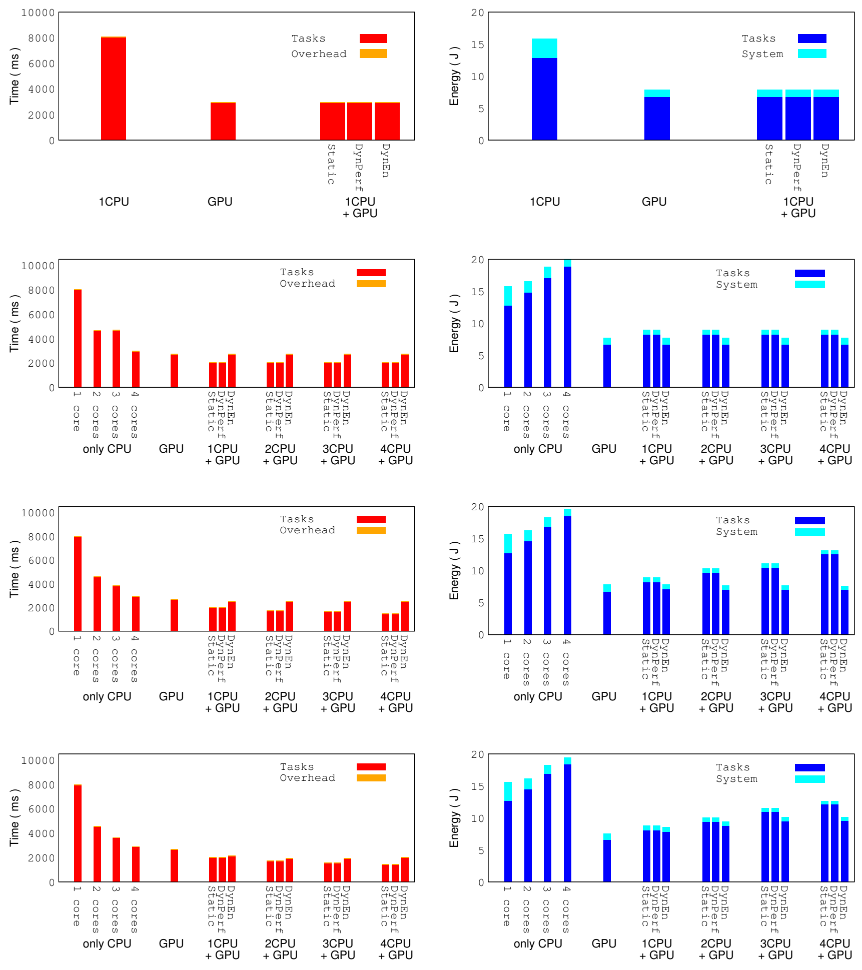

Fig. 10. Execution time (left) and energy consumption (right) for Bézier Surface runs using block sizes of 1024x1024, 512x512, 256x256 and 128x128 (from top to bottom)

policy to schedule the execution of all tasks on the GPU. On finer-grained scenarios, the heterogeneous systems and the GPU present a different behavior. In the case of 256x256, 1 task is computed on the CPU; thus allows the application to reduce 167 ms despite an increase of 381 mJ. Using more CPU cores increases both the execution time and the energy consumption of each task run on the CPU (by 72 ms and 116 mJ); DynEn dismisses executing more tasks on the CPU to avoid their growth. Using smaller blocks reduces the difference in time and energy; thus gives more freedom to the *DE* and allows more diverse schedulings as shown by the four heterogeneous cases using 128x128 blocks. With the GPU and one core of the CPU at its disposal, DynEn assigns 12 tasks to the CPU (requiring 2,130 ms and 7.70 mJ to run), while DynPerf assigns 16 tasks to the CPU (1,983 ms and 8.09 mJ). For the heterogeneous case using 2 CPU cores, DynEn assigns 18 tasks to the CPU vs. the 23 assigned by DynPerf. Again the growth on the execution time and energy consumption due to the concurrent exploitation of multiple cores cuts the number of tasks assigned to the CPU; DynEn and DynPerf assign 18 and 27 tasks to the CPU with three available CPU cores. For the same reason, when using all the computing devices of the phone, DynEn reduces the number of tasks assigned to the CPU to 16 while DynPerf assigns 30 to it. Thus, DynEn shrinks the energy consumption from 12.09 J to 9.5 J while DynPerf shortens the execution time 570 ms.

Canny Edge Detection Instead of using different input sizes, for the third application, we always process a 30-frames video. However, we consider two different workload divisions that the developer could easily implement: Task Partitioning, where the GPU runs the first two tasks of each frame and the CPU the last two; and Data Partitioning, where one device processes the whole frame. Figure 11 shows the execution time (left) and energy consumption (right) obtained when running the application and compares them to the ones obtained with DynPerf and DynEn.

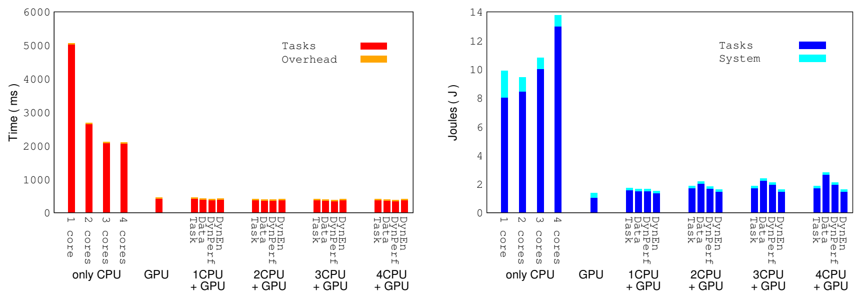

Fig. 11. Execution time (left) and energy consumption (right) obtained from the Canny Edge Detection runs

Task Partitioning achieves lower energy consumptions while Data Partitioning offers better performance. The behavior of Task Partitioning does not change when it has more than one core at its disposal. The time to process the first two tasks of a frame on the GPU – 12 ms – is higher than what it takes to execute the last two – 9 ms and 5 ms respectively, but the executions corresponding to different iterations can overlap. With one CPU core available, the application takes 451 ms and consumes 1.71 J, vs. 412 ms and 1.87 J when using two or more CPU cores.

Data Partitioning assigns the whole processing of a frame to the same computing unit. The problem of this approach is that the number of frames assigned to the CPU does not progress according to the number of available cores – 2, 4, 4, 4 frames, respectively for 1 to 4 cores– due to the performance loss when using multiple cores simultaneously. Using one core, the application takes 427 ms and 1.67 J. When using two or more cores, the execution time shows no improvement – 399 ms with 2 and 4 cores available; indeed, using 3 cores worsens the execution time to 410 ms –; however, the energy consumption reflects the usage of more cores and increases according to the number of used cores – 2.19 J, 2.40 J and 2.79 J.

DynPerf avoids this effect and schedules the executions similarly to Task Partitioning but adjusting the load imbalances. When only one core is available, DynPerf assigns 4 non-maximum suppressions and 1 hysteresis to the GPU to balance the 2 ms difference. Thus, the execution time is reduced to 411 ms consuming only 1.65 J. Conversely, when using more cores, the runtime fills their idle time with Gaussian filter tasks. With two cores at its disposal, the DE decides to run two of them on the CPU reducing the execution time to 395 ms with an energy consumption of 1.84 J; with more cores available, it assigns 6 Gaussian filter tasks to the CPU achieving a 379 ms execution time (79 FPS) with an energy consumption of 2.12 J.

DynEn tends to schedule more tasks on the GPU to avoid the higher consumption of the CPU. Hence, with one available core, the DE submits only 14 non-maximum suppressions and 27 hystereses to the GPU; thus obtaining an execution time of 422 ms and an energy consumption of 1.51 J – the GPU alone achieves 455 ms and 1.22 J). From two cores on, the number of non-maximum suppressions assigned to the CPU raises to 24 to shrink the execution time to 409 ms (73 FPS) with an energy consumption of 1.61 J.

6 Conclusions and Future Work

COMPSs-Mobile is a framework to develop applications targetted to Mobile-Cloud environments. Its programming model, COMPSs, allows developers to parallelize their applications automatically with no need of modifying the code. Through an annotated interface, programmers select the methods whose invocations are replaced by asynchronous tasks. A runtime toolkit executed along with the application detects the data dependencies among these tasks and orchestrates their execution on the underlying infrastructure to exploit the application parallelism while guaranteeing the sequential consistency of the application.

This article introduces an extension to the COMPSs programming model to allow the implementation of these tasks as OpenCL kernels to run on GPUs, FPGAs or any other accelerator. Thus, applications following the model could make the most of the heterogeneous systems composing the infrastructure and use all the available computing devices collaboratively. Beyond inherent parallelism exploitation, the proposed extension helps COMPSs to ease the development of applications by hiding away from the developer all the details related to the handling of the OpenCL platform – managing the content of the device memory and kernels submission – and the load balancing.

Section 4.2 describes the required developments on the runtime toolkit to support the execution of OpenCL kernels on the mobile device as well as the optimizations implemented to maximize the performance of the application automatically. The results presented in Section 5 for the three applications using only one core of the CPU or offloading all the computation to the GPU illustrate the potential benefits of using the accelerators embedded on the device instead of a CPU core to compute a task either from the temporal or energetic point of view. For the CED application, the GPU is $\sim$12x faster and consumes an 87% less energy; for BS, GPU is $\sim$3x faster and 54% less energy consuming.

Accelerators may also be part of the remote nodes to which COMPSs-Mobile offloads computation. Although the proposed extension of the programming model already allows developers to write applications that use them, the described runtime system does not support OpenCL code offloading yet. We believe that a natural step forward in our research is to enable this feature to improve the performance of those application taking benefit of computation offloading.

Delegating the load balancing to the runtime system improves the portability of applications. The execution time of a task and its energy consumption depends on the characteristics of the hardware running the task; therefore, the task scheduling is different for each computing infrastructure. The optimal scheduling may not be evident nor easy to implement; dynamic policies can achieve the desired behavior with no strain for the developer, as shown in the CED test case. Besides, they allow the application user to decide whether if the application should aim for the best performance, the lowest energy consumption or finding a balanced solution with no additional cost for the developer.

Another aspect that we would like to work on is dynamic, adaptative computing platform assignment. Currently, upon the task detection, the runtime picks a platform considering the end time, energy consumption and cost forecasts of running the task on each platform based on the profiling data from previous executions. Hence, the runtime makes the decision considering only task-scoped information instead of considering the impact on the execution as a whole. This makes some scheduling decisions hard to explain. For instance, if the runtime has to pick between option A, where the task ends at ms 50 consuming 0.1 J, and B, finishing at ms 250 with 0.07 J consumed; it would choose the former. However, if we contextualize this decision on an execution that finishes at ms 300 regardless the chosen option, the latter would be better. Besides, the forecasts are based on the profiling data from previous executions. If the characteristics of the workload do not meet those of previous workloads, the runtime makes decisions that harm the application performance. We envisage to enhance the platform selection by enabling a mechanism to correct the current scheduling by re-assigning the execution of pending tasks to another platform.

Acknowledgments

This work is partially supported by the Joint-Laboratory on Extreme Scale Computing (JLESC), by the European Union through the Horizon 2020 research and innovation programme under contract 687584 (TANGO Project), by the Spanish Goverment (TIN2015-65316-P, BES-2013-067167, EEBB-2016-11272, SEV-2011-00067) and the Generalitat de Catalunya (2014-SGR-1051).

References

1. Java programming assistant (javassist). http://www.javassist.org
2. Android Developers: Renderscript, `https://developer.android.com/guide/topics/renderscript/compute.html`
3. Augonnet, C., Thibault, S., Namyst, R., Wacrenier, P.A.: StarPU: a unified platform for task scheduling on heterogeneous multicore architectures. Concurrency and Computation: Practice and Experience 23(2), 187–198 (2011), `http://onlinelibrary.wiley.com/doi/10.1002/cpe.1631/full{\%}5Cnhttp://doi.wiley.com/10.1002/cpe.1631`
4. Bosilca, G., Bouteiller, A., Danalis, A., Herault, T., Lemarinier, P., Dongarra, J.: DAGuE: A generic distributed DAG engine for High Performance Computing. Parallel Computing 38(1-2), 37–51 (2012)
5. Chun, B.G., et al.: CloneCloud: Elastic Execution Between Mobile Device and Cloud. In: Proceedings of the Sixth Conference on Computer Systems. pp. 301–314. EuroSys '11, ACM, New York, NY, USA (2011), `http://doi.acm.org/10.1145/1966445.1966473`
6. Cuervo, E., et al.: MAUI: Making Smartphones Last Longer with Code Offload. In: Proceedings of the 8th International Conference on Mobile Systems, Applications, and Services. pp. 49–62. MobiSys '10, ACM, New York, NY, USA (2010), `http://doi.acm.org/10.1145/1814433.1814441`
7. Duran, A., Ayguadé, E., Badia, R.M., Labarta, J., Martinell, L., Martorell, X., Planas, J.: OmpSs: A PROPOSAL FOR PROGRAMMING HETEROGENEOUS MULTI-CORE ARCHITECTURES. Parallel Processing Letters 21(2), 173–193 (2011)
8. Kemp, R., et al.: Cuckoo: A Computation Offloading Framework for Smartphones. In: Gris, M.L., 0001, G.Y. (eds.) MobiCASE. Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering, vol. 76, pp. 59–79. Springer (2010), `http://dblp.uni-trier.de/db/conf/mobicase/mobicase2010.html{\#}KempPKB10`
9. Khronos OpenCL Working Group and others: The opencl specification. version 1(29), 8 (2008)
10. Kosta, S., et al.: Unleashing the Power of Mobile Cloud Computing using ThinkAir. CoRR abs/1105.3 (2011), `http://arxiv.org/abs/1105.3232`
11. Lordan, F., Badia, R.M.: COMPSs-Mobile: Parallel Programming for Mobile-Cloud Computing. In: 2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid). pp. 497–500 (may 2016)
12. Lordan, F., et al.: Servicess: An interoperable programming framework for the cloud. Journal of Grid Computing 12(1), 67–91 (2014), `http://dx.doi.org/10.1007/s10723-013-9272-5`
13. Montella, R., et al.: Enabling Android-Based Devices to High-End GPGPUs. In: Algorithms and Architectures for Parallel Processing, pp. 118–125. Springer International Publishing (2016)
14. Nvidia: Compute unified device architecture programming guide (2007)

15. Ratering, R., Hoppe, H.C.: Accelerating opencl applications by utilizing a virtual opencl device as interface to compute clouds (2011), `https://www.google.ch/patents/US20110161495`
16. Rellermeyer, J.S., et al.: AlfredO: An Architecture for Flexible Interaction with Electronic Devices. In: Proceedings of the 9th ACM/IFIP/USENIX International Conference on Middleware. pp. 22–41. Middleware '08, Springer-Verlag New York, Inc., New York, NY, USA (2008), `http://dl.acm.org/citation.cfm?id=1496950.1496953`
17. Silla, F., et al.: Remote GPU Virtualization: Is It Useful? In: High-Performance Interconnection Networks in the Exascale and Big-Data Era (HiPINEB), 2016 2nd IEEE International Workshop on. pp. 41–48. IEEE (2016)