Enabling Android-based devices to high-end GPGPUs

Raffaele Montella[1,2], Carmine Ferraro[1], Sokol Kosta[3,4], Valentina Pelliccia[1], Giulio Giunta[1]

[1] Department of Science and Technologies - University of Napoli Parthenope
{raffaele.montella, carmine.ferraro, valentina.pelliccia, giulio.giunta}@uniparthenope.it
[2] Computation Institute - University of Chicago
montella@uchicago.edu
[3] Sapienza University of Rome
kosta@di.uniroma1.it
[4] CMI, Aalborg University Copenhagen
sok@cmi.aau.dk

Abstract. The success of Android is based on its unified Java programming model that allows to write platform-independent programs for a variety of different target platforms. In this paper we describe the first, to the best of our knowledge, offloading platform that enables Android devices with no GPU support to run Nvidia CUDA kernels by migrating their execution on high-end GPGPU servers. The framework is highly modular and exposes a rich Application Programming Interface (API) to the developers, making it highly transparent and hiding the complexity of the network layer. We present the first preliminary results, showing that not only GPGPU offloading is possible but it is also promising in terms of performance.

Keywords: Virtualization, GPGPU, CUDA, Cloud, Android, Offloading

1 Introduction

Nowadays, we use mobile devices to write documents, browse the internet, explore maps, watch and edit videos, play games, and perform all the tasks we used to run on powerful computers, and many more. Users are so attached to their smart mobile devices, that researchers have found that they would prefer to use their own devices even for working, a concept known as *Bring Your Own Device (BYOD)*[5]. To keep up with users' demands and applications' needs for ever increasing computational resources, devices are being equipped with a myriad of additional hardware, sensors, and extra features. Nevertheless, to always be updated, users are "forced" to replace their devices very frequently, which increases their costs. Furthermore, advances in battery technology have not been

[5] http://www.dell.com/en-uk/work/learn/mobility-byod

able to follow the fast smartphone developing race, making energy consumption a bottleneck for most users. To address these problems, developers and companies have developed solutions that offload the heavy operations from the low-powered mobile devices to more resourceful remote machines, usually residing on the cloud [6], [4]. Recently, researchers have proposed the need for offloading not only the CPU tasks but also the operations performed by the GPU [9], [2]. Unfortunately, the NVIDIA CUDA's GPGPU proprietary approach collides with Android's open source philosophy, limiting the CUDA availability only on a couple of brand-specific Android products, hindering this way the mass diffusion of GPU computation in the Android developers' community.

In this paper, we design and develop an offloading framework that supports transparent method offloading for Android applications, with focus on the GPU task offloading.

The rest of the paper is organized as follows: in Section 2 a short description of some related works; in Section 3 we describe the high-level architecture of the framework, identifying its main components; in Section 3.1 we give an overview of the GPU virtualization technique we designed and used in our system; in Section 4 we present the design and implementation details of the GPGPU task offloading component; in Section 5 we show the first preliminary results of and Android application exploiting GPU offloading; and finally, in Section 6 we conclude the paper with remarks on future directions.

2 Related works

With the proliferation of mobile devices in the recent years, it was clear that the low computational capabilities and the limited battery capacity were obviously a bottleneck to the final users. To this end, researchers have proposed different solutions to alleviate the burden of the low-powered devices by migrating the heavy computations from low-powered devices to more powerful machines.

CloneCloud [3] uses an offline static analysis of the apps' binary to determine the pieces that are better to offload to the cloud. The developer should then use the output of the analysis to build a database of offloading decisions, which is later loaded on the device. On runtime, the framework looks up the database to decide where to execute the code. ThinkAir [6] is a method-based offloading framework for Android devices. The smartphone is associated with a virtual machine (VM) on a private or public cloud and the offloadable methods are sent to the VM for remote execution. ThinkAir also supports dynamic resource allocation, exploiting the power of the cloud whenever the offloaded method can be executed in parallel on multiple clones. Comet [4] is a platform that works on top of the Android's Dalvik Virtual Machine and performs task offloading at thread level. Comet uses the Distributed Shared Memory to achieve its lightweight synchronization and offloading procedure. All the above described works focus on offloading of CPU computations. Only recently researchers have started to propose the first ideas about GPU code offloading on mobile devices [9], [2]. The rest of this section focuses on the available GPU virtualization technologies and

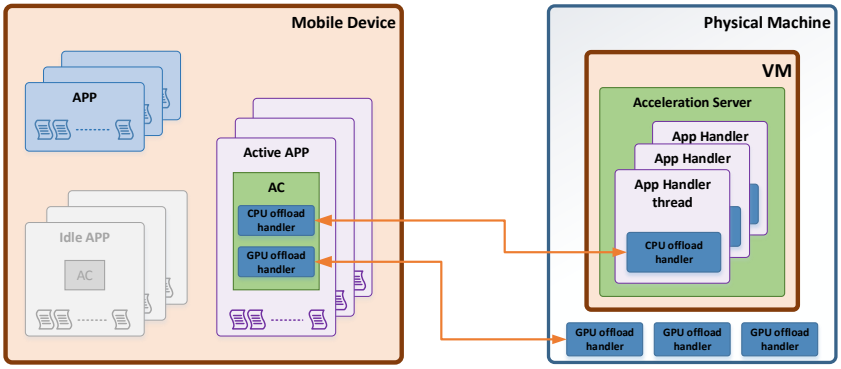

Fig. 1. The architecture of the offloading framework, showing the AC, AS, CPU offload handler, and GPU offload handler.

frameworks, to better create the picture of our solution. JCUDA [11] is a Java framework which enables CUDA kernels invocation delegating the responsibility of generating the Java-CUDA bridge codes and host-device data transfer calls to the compiler. The JCUDA implementation handles data transfers of primitives and multidimensional arrays of primitives between the host and device. JCUDA works thanks to the Java Native method invocation, where C functions are declared as native external methods dealing with out-of-the-sandbox unmanaged memory. While JCUDA offers a remarkable solution to the problem of Java CUDA kernel development, it cannot be applied in the Android context because the CUDA compiler support is mandatory at runtime.

3 Offloading Service Architecture

The offloading framework is highly modular and is composed of two main entities: the Acceleration Client (AC) and the Acceleration Server (AS). The AC is an Android library that specifies the Application Programming Interface (API) that developers can use to implement method offloading without having to deal with the underlying networking solutions. The AS is a server application that runs on a remote machine and receives offloaded methods from the AC for remote execution. The AC defines two main sets of API, specialized to handle CPU and GPU instructions. The API dealing with CPU offloading is inspired by the ThinkAir framework [6], while the API dealing with GPU offloading is a totally novel implementation and Android-adapted of the GVirtuS system [7].

Figure 1 shows a device on the left with multiple applications, some of which make use of the AC to migrate the heavy tasks. We can notice the two different modules implemented by the AC that are used to handle the CPU and GPU tasks offloading. On the right side of the figure we can see the Acceleration Server running on a Virtual Machine (VM) with the same operating system as the device (Android OS e.g.). When an AC connects with the AS, the AS starts a

new thread (App Handler) to serve the respective application, making it possible to support multiple applications on one VM. The CPU offload handler on the AS receives the migrated code, executes it, and sends the result of the execution back to the AC. When the AC has to offload GPU operations, it connects to the GPU offload handler on the remote physical machine, which is part of the GPU virtualized framework. In the rest of the paper we describe in more details the architecture of the offloading framework, with emphasis on the GPU offloading component.

3.1 GPU Virtualization Architecture

The GPU virtualization architecture is inspired by our previous work GVirtuS, which is a generic virtualization framework for virtualization solutions based on a split-driver model [1]. GVirtuS is a framework that offers virtualization support for generic libraries on traditional x86 computers. In the current state, GVirtuS supports the main GPGPU accelerator libraries such as CUDA and OpenCL, with the advantage of being independent from all the underlying involved technologies: i.e. hypervisor, communicator, and target of virtualization. GVirtuS is composed of two main components: the *front-end* and the *back-end*, which are installed on the client device and on the GPU-capable remote device, respectively.

The front-end is transparent to the application developers, allowing applications to make CUDA or OPENCL calls without any adaption on the source code. If the client device is not able to execute the GPGPU calls, the front-end intercepts and transmits them to the back-end, which runs them on the remote machine and sends the result back to the front-end, which then forwards it to the caller application.

The interaction between the front-end and the back-end is performed through a *communicator* component, which is a pluggable module, meaning that it only presents a public API to the interested components without exposing the underlying communication technology. The communicator has an important role in the overall architecture, given that is responsible for the data transmission.

The back-end is the main component of the GVirtuS framework and runs on the GPU-capable host machine. The back-end daemon runs on the host operating system in the user or superuser space, depending on the specifics of applications and security policies waiting for an incoming connection from the front-end. The daemon implements the back-end functionality dealing with the physical device driver and performing the host-side virtualization.

4 Design and Implementation of Android GPU Offloading Framework

In this section we present the design of GVirtus4J, the Android GPU offloading framework, which is inspired by the architecture of GVirtuS described Section 3.1. A CUDA program is composed by two distinct elements: *i)* the CUDA application

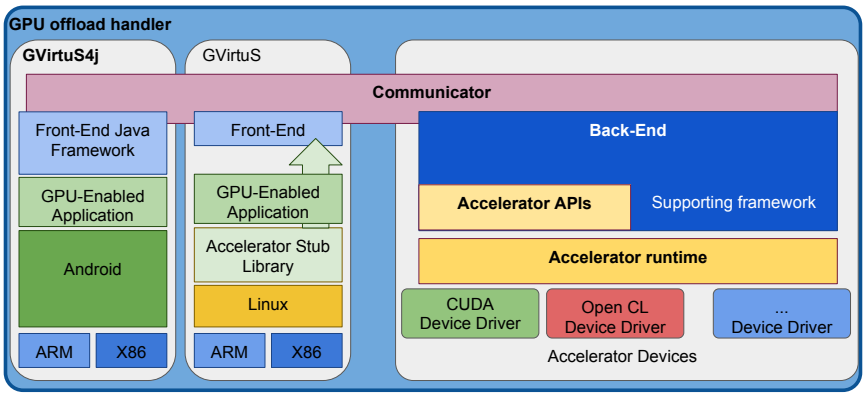

Fig. 2. The GVirtuS4J architecture.

programming interface functions[6], such as the memory management, and *ii)* the kernel functions[7], which are written in CUDA language. As such, we implement GVirtuS4J along two axes: *i)* Encapsulating the CUDA API functions and *ii)* Handling the kernel functions.

To deal with the first problem, we build the front-end of GVirtuS4J as a Java/Android library that developers can include in their applications. We include the CUDA C++ methods with the same original signature, whenever possible. Following an approach similar to JCUDA [11], Android developers can write Java code that directly calls CUDA kernels by using our API. Differently from JCUDA, GVirtuS4J does not delegate the responsibility of generating the Java-CUDA bridge code and host-device data transfer calls to the compiler, but it behaves as local wrapper of the standard CUDA library. In the remote side, we use the original GVirtuS back-end.

The communication between the Android GVirtuS4J front-end and the back-end is implemented using a customized TCP/IP communication protocol (see Figure 2). When a CUDA function is called in the Java/Android code, a socket connection with the back-end server is created and a buffer for the data to be sent is allocated. Then, the CUDA function name together with the data (the parameters passed to the function) are serialized and sent to the back-end. The latter assigns them to the appropriate variables and invokes the corresponding CUDA function. The processing result is returned to the Java/Android front-end, which forwards it to the caller object.

CUDA Kernels can be written using the CUDA instruction set architecture, called *Parallel Thread Execution* (PTX), or using a high-level programming language such as C. In both cases, kernels must be compiled into binary code by the *NVIDIA CUDA Compiler* (NVCC) to execute on the device. NVCC compiles applications written in C/C++ and CUDA by splitting the code in two parts:

[6] http://docs.nvidia.com/cuda/cuda-runtime-api/
[7] https://devblogs.nvidia.com/parallelforall/easy-introduction-cuda-c-and-c/

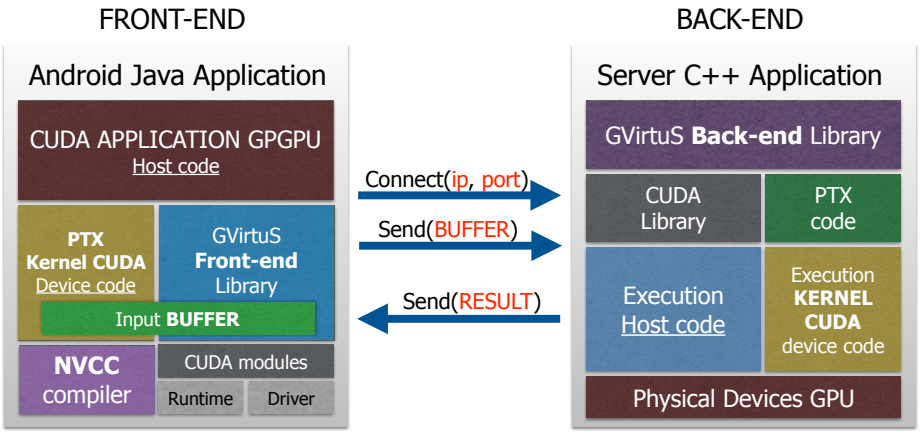

Fig. 3. Representation of a PTX source code offloaded by GVirtuS4J front-end to the back-end for remote execution.

the pure C/C++ part, which will be compiled by traditional C/C++ compilers such as GCC, and the CUDA part, which will be compiled by the NVCC into a PTX file. To be able to include CUDA kernels in Java/Android and to execute them remotely, we send the PTX file containing the kernels to the backend, as shown in Figure 3.

5 Preliminary Results

To evaluate our prototype, we performed some preliminary tests which proved that *i)* GPU code offloading is feasible and *ii)* convenient under the right circumstances. Choosing a performance testing suite accepted by the community was not possible,

$A \times B$	CPU (PA64, s)	GPU* (PA64, s)	CPU (AZP8, s)	GPU* (AZP8, s)
$A[128, 192] \times B[128, 128]$	0.3	6.3	0.6	6.3
$A[256, 384] \times B[256, 256]$	3.6	19.9	7.4	20.2
$A[512, 768] \times B[512, 512]$	60.7	79.1	81.0	**75.5**
$A[768, 1152] \times B[768, 768]$	223.6	**169.2**	288.5	**170.1**
$A[1024, 1536] \times B[1024, 1024]$	505.4	**309.6**	685.2	**302.5**
$A[1280, 1920] \times B[1280, 1280]$	1071.0	**479.7**	1531.3	**466.7**
$A[1536, 2304] \times B[1536, 1536]$	1772.3	**686.2**	2611.8	**708.6**

Table 1. Preliminary Results: Time in seconds, *GPU offload. PA64: PineA64+; AZP8: Ausus ZenPad 8. In **bold** when the offload GPU execution beats the local CPU performance.

due to the lack of such suite, given that the GPU code offloading for Android devices is a novel approach. We chose one of the NVIDIA's CUDA SDK 6.5 samples, which are included with the CUDA distribution[8]. Precisely, in this paper we show the results of the *Matrix Multiplication*. The choice was motivated by its clarity of exposition on illustrating various CUDA programming principles, which makes it easy to clearly present the needed modifications for making it work with GVirtuS4J. Moreover, performing linear algebra operations is a common task assigned to GPGPUs [10]. The preliminary tests have been performed with a varying problem size as shown in Table 1, where the size of matrix A is given by 4*block_size, 6*block_size and the size of matrix B is given by 4*block_size, 4*block_size (bloc_size=32). The Accelerator server (Dual Xeon 6-core E5-2609v3 1.9Ghz - 8GB DDR4) we used for this experiment is equipped by two NVIDIA Titan X CUDA enabled devices. We executed performance tests using a PineA64+ (Single Board Computer: Cortex A53 64bit quad) and an Asus ZenPad 8 (Tablet: Qualcomm 64bit quad core, 1.2GHz, 1GB RAM). The connection between the front-end and the back-end was realized through a traditional WiFi infrastructure.

The experiment demonstrates the GPU offloading is convenient as the problem size increases. The break point of offloaded GPU beating the local CPU is related to the device (single board computer and tablet behaves differently), the algorithm used, and the network condition. These results have to be intended as very preliminary and a more extensive performance test and evaluation suite is needed in order to define the range of problem size reflecting on the feasibility of the proposed approach. Exploiting the result asset of this promising research is a keystone for the next level of distributed applications in which the scenario is not limited to mobile devices but to all embedded technologies.

6 Conclusions and Future Directions

In this paper we presented the design and implementation of GVirtuS4J, the first, to the best of our knowledge, GPGPU CUDA-based offloading framework for Android devices. We first presented the big picture of the generic offloading framework, describing its main components, with focus on the detailed description of the GPGPU related subsystem that deals with the GPU offloading process. We showed through preliminary experiments that the system is working correctly. Moreover, we showed that if the problem is complex enough, offloading to a GPU enabled machine reduces its execution time.

Currently, we are still working on the development of the proposed platform in three time-based future directions: *i)* in a short term we will improve the efficiency and architectural design of the GPU offload handler, leveraging on a more extensive and robust test suite thanks to the availability of a more up-to-date hardware infrastructure; *ii)* as medium term goal we will release the GVirtuS4J framework as open source software under Apache 2.0 license, offering

[8] http://docs.nvidia.com/cuda/cuda-samples/

to a selected group of early adopters the availability of cloud shared GPGPUs computing resources to test their applications, to explore framework's possible enhancements, and to contribute on improving the overall quality; *iii)* as a long term goal we will leverage on a solid and stable GPGPU offloading infrastructure to implement real-world applications in the field of distributed computing and high-performance internet of things [5].

Acknowledgments This research has been supported by the Grant Agreement number: 644312 - RAPID - H2020-ICT-2014/H2020-ICT-2014-1 "Heterogeneous Secure Multi-level Remote Acceleration Service for Low-Power Integrated Systems and Devices".

References

1. Armand, F., Gien, M., Maigné, G., Mardinian, G.: Shared device driver model for virtualized mobile handsets. In: Proceedings of the First Workshop on Virtualization in Mobile Computing. pp. 12–16. ACM (2008)
2. Choi, K., Lee, J., Kim, Y., Kang, S., Han, H.: Feasibility of the computation task offloading to gpgpu-enabled devices in mobile cloud. In: Cloud and Autonomic Computing (ICCAC), 2015 International Conference on. pp. 244–251 (Sept 2015)
3. Chun, B.G., Ihm, S., Maniatis, P., Naik, M., Patti, A.: Clonecloud: elastic execution between mobile device and cloud. In: Proceedings of the sixth conference on Computer systems. pp. 301–314. ACM (2011)
4. Gordon, M.S., Jamshidi, D.A., Mahlke, S., Mao, Z.M., Chen, X.: Comet: Code offload by migrating execution transparently. In: Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12). pp. 93–106. USENIX, Hollywood, CA (2012), https://www.usenix.org/conference/osdi12/technical-sessions/presentation/gordon
5. Kantarci, B., Mouftah, H.T.: Trustworthy sensing for public safety in cloud-centric internet of things. Internet of Things Journal, IEEE 1(4), 360–368 (2014)
6. Kosta, S., Aucinas, A., Hui, P., Mortier, R., Zhang, X.: Thinkair: Dynamic resource allocation and parallel execution in the cloud for mobile code offloading. In: INFOCOM, 2012 Proceedings IEEE. pp. 945–953 (March 2012)
7. Montella, R., Giunta, G., Laccetti, G., Lapegna, M., Palmieri, C., Ferraro, C., Pelliccia, V.: Virtualizing cuda enabled gpgpus on arm clusters. In: Parallel Processing and Applied Mathematics, pp. 3–14. Springer (2016)
8. Reiter, A., Zefferer, T.: Power: A cloud-based mobile augmentation approach for web- and cross-platform applications. In: Cloud Networking (CloudNet), 2015 IEEE 4th International Conference on. pp. 226–231 (Oct 2015)
9. Silva, F.A., Rodrigues, M., Maciel, P., Kosta, S., Mei, A.: Planning mobile cloud infrastructures using stochastic petri nets and graphic processing units. In: 2015 IEEE 7th International Conference on Cloud Computing Technology and Science (CloudCom). pp. 471–474 (Nov 2015)
10. Volkov, V., Demmel, J.W.: Benchmarking gpus to tune dense linear algebra. In: High Performance Computing, Networking, Storage and Analysis, 2008. SC 2008. International Conference for. pp. 1–11. IEEE (2008)
11. Yan, Y., Grossman, M., Sarkar, V.: Jcuda: A programmer-friendly interface for accelerating java programs with cuda. In: Euro-Par 2009 Parallel Processing, pp. 887–899. Springer (2009)