CUDA virtualization and remoting for GPGPU based acceleration offloading at the edge

Mentone, Antonio; Di Luccio, Diana; Landolfi, Luca; Kosta, Sokol; Montella, Raffaele

Published in:
The 12th International Conference on Internet and Distributed Computing Systems

DOI (link to publication from Publisher):
10.1007/978-3-030-34914-1_39

Publication date:
2019

Document Version
Accepted author manuscript, peer reviewed version

CUDA virtualization and remoting for GPGPU based acceleration offloading at the edge

Antonio Mentone[1 [0000−0001−7546−4000]], Diana Di Luccio[1 [0000−0002−0810−2250]], Luca Landolfi[1 [0000−0002−7973−2209]], Sokol Kosta[2 [0000−0002−9441−4508]], and Raffaele Montella[1 [0000−0002−4767−2045]]

[1] University of Naples "Parthenope", Science and Technologies Department, Napoli, Italy
{antonio.mentone001,luca.landolfi}@studenti.uniparthenope.it
{diana.diluccio,raffaele.montella}@uniparthenope.it
[2] Aalborg University Copenhagen, Department of Electronic Systems, Denmark
sok@cmi.aau.dk

Abstract. In the last decade, GPGPU virtualization and remoting have been among the most important research topics in the field of computer science and engineering due to the rising of cloud computing technologies. Public, private, and hybrid infrastructures need such virtualization tools in order to multiplex and better organize the computing resources. With the advent of novel technologies and paradigms, such as edge computing, code offloading in mobile clouds, deep learning techniques, etc., the need for computing power, especially of specialized hardware such as GPUs, has skyrocketed. Although many GPGPU virtualization tools are available nowadays, in this paper we focus on improving GVirtuS, our solution for GPU virtualization. The contributions in this work focus on the CUDA plug-in, in order to provide updated performance enabling the next generation of GPGPU code offloading applications. Moreover, we present a new GVirtuS implementation characterized by a highly modular approach with a full multithread support. We evaluate and discuss the benchmarks of the new implementation comparing and contrasting the results with the pure CUDA and with the previous version of GVirtuS. The new GVirtuS yielded better results when compared with its previous implementation, closing the gap with the pure CUDA performance and trailblazing the path for the next future improvements.

Keywords: HPC · GPGPU · Cloud Computing · Virtualization.

1 Introduction

In a grid environment, computing power is offered on-demand to perform large numerical simulations on a network of machines, potentially extended all over the world [5]. Virtualization techniques represent a good solution to the problem of executing generic complex high performance scientific software on a grid, and have inspired a novel computing paradigm in which virtualized resources are spread in a cloud of real high performance hardware infrastructures [6].

This model, well known as Cloud Computing, is *an internet-based model providing a convenient on demand access to a shared pool of configurable computing resources which can be rapidly assigned and released with a minimal management effort or service provider interaction* [9].

Latest-generation supercomputers take advantage of GPUs (Graphics Processing Units) processing power in order to speed up calculations. GPUs are parallel microprocessors attached to a graphics card. They are extremely flexible, completely programmable and able to achieve extremely high performance during the parallel processing of large sets of data. GPUs' high performance can also be used for general purpose scientific computing and starting from this definition the GPGPU technology is born [2]. GPGPU is exploited by using parallel programming environment such as **OpenCL** and **CUDA** [12].

In this paper, we consider the case of GVirtuS (GPU Virtualization Service) [10], one of the state-of-the-art solutions that allows sharing the power of a GPGPU among different applications running concurrently on a single machine. GVirtuS uses a virtualization approach: the virtualized service is transparent to the users running a GPGPU application, and there is little overhead compared to bare metal GPGPU setup. The fields of application are many, including Internet of Things, mobile code offloading, and others [11, 13].

In this work, we present the evolution of GVirtuS[3], featuring a redesigned architecture and a general *code refactoring*. In order to modernize the framework and improve its performance, we have adopted and integrated new technologies and multi-threading techniques. We have also improved and restructured the build process, in order to improve software portability and simplify the management of external libraries.

The rest of the paper is organized as follows: Section 2 describes related work, Section 3 contains a description of the software architecture and the main design choices; Section 4 presents the new implementation of the GPU virtualization; Section 5 describes the performance tests and the obtained results; and finally, Section 6 draws conclusions and discusses some of the future planned developments.

2 Related work

A comprehensive survey about GPGPU virtualization and remoting techniques is discussed in [7]. GPU virtualization solutions such as GVirtuS have been implemented by other research projects such as rCUDA [17, 15, 4] and Distribuited-Shared CUDA (DS-CUDA) [14]. They use an approach similar to GVirtuS, providing CUDA API wrappers on the front-end application in the guest OS while the back-end in the host OS accesses the CUDA devices. We now discuss some of the differences between these solutions.

CUDA Toolkit supported version: all GPGPU computing solutions mentioned above implement their functionalities using the CUDA Runtime API.

[3] https://github.com/gvirtus/GVirtuS

None of them supports rendering specific graphic APIs, such as OpenGL[4] and Direct3D[5].
Communicator: the component that connects guest and host systems. GVirtuS supports several communicator protocols: TCP/IP sockets, WebSocket, Unix Sockets, VMSocket (for KVM based virtualization), and VMCI (for VMWare based virtualization). By default, rCUDA and DS-CUDA use InfiniBand Verbs and TCP/IP sockets as fallback.
Plug-in architecture: GVirtuS supports CUDA and OpenCL, while rCUDA and DS-CUDA only support NVIDIA CUDA.
Transparency: using GVirtus and rCUDA, CUDA enabled software is able to run on the remote GPUs without further changes to the source code. In order to enable DS-CUDA support, an application must include DS-CUDA specific extensions and it must be compiled using DS-CUDA specific tool-chain.
License: GVirtuS and DS-CUDA are both open source projects: the former is licensed under the Apache 2.0, while the latter is licensed under the GPLv3. rCUDA is proprietary software, but it is distributed for free under specified terms and conditions of use.
ARM support: GVirtuS supports x86_64 and ARM hardware platforms. It supports all combinations of ARM and x86_64 on the front-end and the back-end (e.g. it is possible to run code from ARM front-end on a x86_64 back-end, and vice-versa). rCUDA also supports ARM and x86_64 platforms in a manner similar to GVirtuS [1], while DS-CUDA supports ARM front-ends but only x86_64 back-ends [8].

3 System architecture and Design

GVirtuS is a generic virtualization framework for virtualization solutions. GVirtuS offers virtualization support for generic libraries such as accelerator libraries (CUDA, OpenCL), with the advantage of independence from all the involved technologies: hypervisor, communicator, and target virtualization. This feature is possible thanks to the plug-in design of the framework, enabling the choice of different communicators or different stub-libraries which mock the virtualization targets. GVirtuS is transparent for developers: no changes in the software source code are required to virtualize and execute applications, and there is no need to recompile an already compiled executable.

3.1 Architecture

The virtualization system of GVirtuS is based on a *split driver approach* with two main components, **front-end** and **back-end**. The front-end component is deployed on the lightweight machines that don't have a GPU, while the back-end component is hosted on the real machine that accesses directly the GPU device.

[4] https://www.opengl.org/
[5] https://docs.microsoft.com/en-gb/windows/win32/direct3d

A hypervisor concurrently deploys the applications requiring access to the GPU accelerators as VM appliances. The device is under control of the hypervisor. An access to the GPU is routed via the front-end/back-end layers under control of a management component, and data are moved from GPU to guest VM application, and vice-versa. The front-end and the back-end layers implement the decoupling between the hypervisor and the communication layer. A key property of the proposed system is its ability to execute CUDA kernels and OpenCL with an overall performance similar to that obtained by real machines with direct access to the accelerators. This has been achieved by developing a component that provides a high performance communication between virtual machines and their hosts. The choice of the hypervisor deeply affects the efficiency of the communication between the guest and host machines and then between the GPU virtualization front-end and back-end. GVirtuS provides efficient communication for VMware[6] and KVM/Qemu[7] hypervisors.

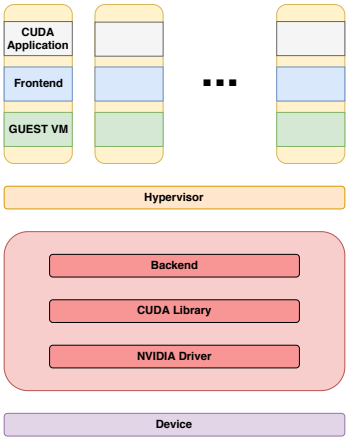

Fig. 1. Block diagram of the GVirtuS architecture.

3.2 Design

The front-end/back-end communication is abstracted by the *Communication* interface concretely implemented by each communicator component. The methods implemented by concrete communicator classes support request preparation, input parameters management, request execution, error checking, and output data recovery. The back-end is executed on the host machine: it is a server program that runs as a user with enough privileges to interact with the CUDA driver.

[6] https://www.vmware.com/
[7] https://www.linux-kvm.org

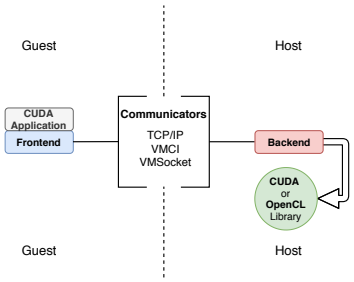

Fig. 2. The GVirtuS components.

The back-end accepts new connections and spawns a new process to serve the front-end requests. The CUDA enabled application running on the virtual or remote machine requests GPGPU resources to the virtualized device using the stub-library. Each function in the stub-library follows these steps:

1. Obtains a reference to the single `Frontend` instance;
2. Uses `Frontend` class methods for setting the parameters;
3. Invokes the `Frontend` handler method specifying the remote procedure name;
4. Checks the remote procedure call results and handles output data.

GVirtuS strictly depends on the CUDA API version, because of the nature of the transparent virtualization and remoting. Given that CUDA is a proprietary solution and not open source, the use of a virtualization/remoting layer becomes inherently non trivial.

3.3 A novel approach

With the **third generation of GVirtuS** many new features have been added. The loading process of the *Communicator* has changed. Before, the *Communicator* was part of a static library and it was linked at compile time. In the new version, it is loaded at run time using a dynamic loading technique, the same used with the plug-in libraries. Moreover, the back-end can now use several `Communicator` objects: in this way the server can listen on multiple endpoints, each of them using a different communication protocol.
When the server is started, it launches a new process for each type of communicator indicated in the configuration file. When a process receives a request, it creates a new thread that serves it, and then it keeps listening for new requests from the clients. Thanks to this new thread model, each module and dynamic library is now loaded at startup and it is not unloaded after a request is served, whereas the old version loaded and unloaded modules for each different request. As a result, the overall overhead is reduced for subsequent requests, as we show in section 5.2.

There are several other features that have been added, such as JSON configuration file, signal state handlers, exceptions hierarchy, and much more. In

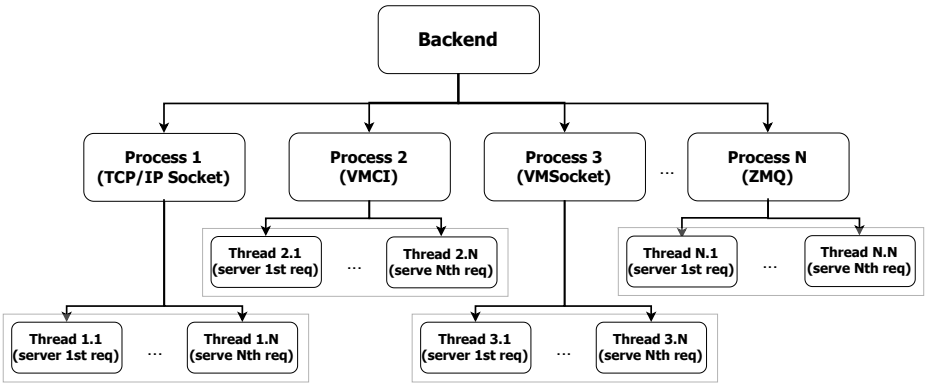

Fig. 3. The new back-end design.

order to support these changes, the overall architecture of the framework has undergone a substantial re-design.

4 Implementation details

Several new technologies have been used in the new version of GVirtuS.
Web-Sockets have been added to the Communicator suite, facilitating real-time data transfers from and to the server. **GTest** and **GMock**, powered by Google, have made possible to write test units quickly and easily. An important effort has been made to update the framework to the latest standards, such as the new **C++1z** standard[8], and many tools like **JSON** (JavaScript Object Notation), **TLS** (Transport Layer Security), and **zlib**[9] (DEFLATE data compression algorithm). Finally, an initial effort to introduce *asynchronous I/O* has been made, using the **libuv** library[10] (Node.js engine written in C). However, this feature is not fully tested and is not meant to be used in production yet.

5 Performance evaluation

5.1 Workstation setup

The workstation used for testing is equipped with a double Intel®**Xeon**®**E5-2609** v3 @ 1.90 GHz, a six-core hyper-threaded CPU with **15 MB** cache, and **32 GB** of DDR4 RAM. The GPU sub-system is composed of two NVIDIA GeForce GTX TITAN X GM200[11]. They are equipped with 3072 CUDA cores

[8] https://isocpp.org/std/status
[9] https://www.zlib.net/
[10] https://github.com/libuv/libuv
[11] https://www.geforce.com/hardware/desktop-gpus/geforce-gtx-titan-x/specifications

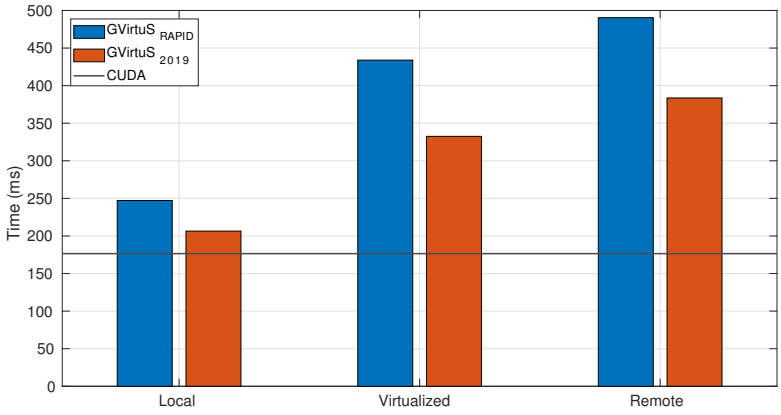

Fig. 4. Matrix multiplication test. Three physical setups are showed for each framework. The bare-metal CUDA performance is also showed as a benchmark line.

and **12 GB** of GDDR5 memory. The CUDA cores run at 1000 MHz, and the graphic memory runs at 1753 MHz. The testing system has been built on top of the CentOS 7 Linux operating system, the NVIDIA CUDA/OpenCL driver, and the SDK/Toolkit version 9.0.

5.2 Benchmarks

In this section, we show the benchmarks executed to test the performance of the latest version of GVirtuS, namely GVirtuS_{2019} in the rest of the paper, against the bare CUDA and against the previous implementation of the framework, namely $\text{GVirtuS}_{\text{RAPID}}$. The performance has been measured using a program that computes **matrix multiplication**, a classic but highly relevant problem in scientific computing. The size of the real valued matrices (using 32 bit floating point arithmetic) used for testing are 320×320 and 320×640. First, we perform the matrix multiplication on the bare-metal CUDA setup, which is used as a benchmark for the other results achieved in the other physical setups, which are the following:

Local The front-end and back-end is running on the same machine.
Virtualized The front-end is running inside a virtual machine.
Remote The front-end is running on a remote host and communicates with the back-end using TCP/IP sockets.

Figure 4 presents the results of the experiments, where the execution times using three physical setups for each framework are shown. Each column presents the average execution time of 1000 runs of the matrix multiplication program using the matrices described above. Notice that in the presented experiments

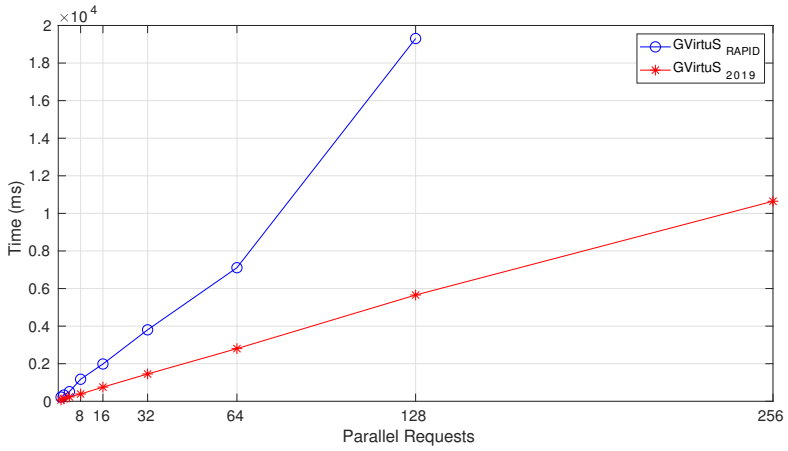

Fig. 5. Average times of parallel requests to the old and new version of GVirtuS.

the matrices remain the same during all the executions. However, randomizing the values of the matrices still yields the same results. As it can be seen from the figure, the new release of GVirtuS has improved significantly from its previous version, performing 20% better in the *Local* setup and 25% better in the *Virtualized* and the *Remote* setups.

Figure 5 shows the average response time for an increasing number of parallel requests to the GVirtuS back-end, and compares the performance between the old and the new implementation. The new GVirtuS exhibits remarkably better overall performance due to the modern multi-threaded architecture. In particular, it can sustain a higher load than the old version. In our tests, the old framework was indeed not able to sustain more than 128 concurrent requests. Moreover, GVirtuS_{2019} performance exhibits a linear growth with respect to the number of parallel requests, while the old GVirtuS degrades almost quadratically. To perform the parallel tests we have used the **GNU Parallel power tool** [18].

Finally, Figure 6 shows the impact of the caching system presented in 3.3 on the performance of GVirtuS_{2019} compared to $\text{GVirtuS}_{\text{RAPID}}$. The first request is served in a time comparable to the old version, but each subsequent request greatly benefits from the reduced overhead due to the now eliminated loading/unloading of the dynamic libraries.

6 Conclusions and Future directions

In this paper, we presented the third generation of the GVirtuS framework, a GPU virtualization and sharing service. The main aim of this upgrade was to improve the performance of the framework and to make it compliant with the latest technological standards. We have reported the results of an extensive

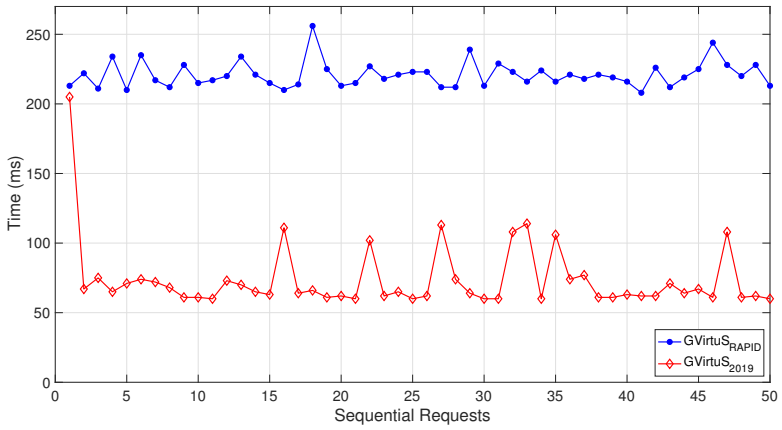

Fig. 6. After the first request is served, response times are reduced in GVirtuS$_{2019}$.

testing process. We compared the performance of GVirtuS$_{2019}$ against the previous version, GVirtuS$_{\text{RAPID}}$. The results clearly show the performance boost achieved by GVirtuS$_{2019}$ compared to its predecessor. Moreover, considering benchmarks available in literature [3] and experiments performed with rCUDA, which we could not present in this paper due to its license restrictions[12], rCUDA performs slightly better than GVirtuS$_{2019}$. On the other hand, GVirtuS still offers advantages compared to a proprietary solution like rCUDA: it has an open source license, it supports a vast series of communication protocols, and provides OpenCL back-end support.

As future work, we will explore the performance of GVirtuS on high performance networks like 10G Ethernet and add the support for the Infiniband protocol [16].

References

1. Castelló, A., Duato, J., Mayo, R., Peña, A.J., Quintana-Ortı, E., Roca, V., Silla, F.: On the use of remote gpus and low-power processors for the acceleration of scientific applications. In: The Fourth International Conference on Smart Grids, Green Communications and IT Energy-aware Technologies (ENERGY). pp. 57–62 (2014)
2. Di Lauro, R., Giannone, F., Ambrosio, L., Montella, R.: Virtualizing general purpose gpus for high performance cloud computing: an application to a fluid simulator. In: 2012 IEEE 10th International Symposium on Parallel and Distributed Processing with Applications. pp. 863–864. IEEE (2012)
3. Duato, J., Pena, A.J., Silla, F., Fernandez, J.C., Mayo, R., Quintana-Orti, E.S.: Enabling cuda acceleration within virtual machines using rcuda. In: 2011 18th International Conference on High Performance Computing. pp. 1–10. IEEE (2011)

[12] http://www.rcuda.net/pub/rCUDA_TOS.pdf

4. Duato, J., Pena, A.J., Silla, F., Mayo, R., Quintana-Ortí, E.S.: rcuda: Reducing the number of gpu-based accelerators in high performance clusters. In: 2010 International Conference on High Performance Computing & Simulation. pp. 224–231. IEEE (2010)
5. Foster, I., Zhao, Y., Raicu, I., Lu, S.: Cloud computing and grid computing 360-degree compared. arXiv preprint arXiv:0901.0131 (2008)
6. Giunta, G., Montella, R., Agrillo, G., Coviello, G.: A gpgpu transparent virtualization component for high performance computing clouds. In: European Conference on Parallel Processing. pp. 379–391. Springer (2010)
7. Hong, C.H., Spence, I., Nikolopoulos, D.S.: Gpu virtualization and scheduling methods: A comprehensive survey. ACM Computing Surveys (CSUR) **50**(3), 35 (2017)
8. Martinez-Noriega, E.J., Kawai, A., Yoshikawa, K., Yasuoka, K., Narumi, T.: Cuda enabled for android tablets through ds-cuda (2013)
9. Mell, P.: The nist definition of cloud computing v15. http://csrc. nist. gov/groups/SNS/cloud-computing/ (2009)
10. Montella, R., Coviello, G., Giunta, G., Laccetti, G., Isaila, F., Blas, J.G.: A general-purpose virtualization service for hpc on cloud computing: an application to gpus pp. 740–749 (2011)
11. Montella, R., Ferraro, C., Kosta, S., Pelliccia, V., Giunta, G.: Enabling android-based devices to high-end gpgpus. In: Carretero, J., Garcia-Blas, J., Ko, R.K., Mueller, P., Nakano, K. (eds.) Algorithms and Architectures for Parallel Processing. pp. 118–125. Springer International Publishing, Cham (2016)
12. Montella, R., Giunta, G., Laccetti, G., Lapegna, M., Palmieri, C., Ferraro, C., Pelliccia, V., Hong, C.H., Spence, I., Nikolopoulos, D.S.: On the virtualization of cuda based gpu remoting on arm and x86 machines in the gvirtus framework. International Journal of Parallel Programming **45**(5), 1142–1163 (2017)
13. Montella, R., Kosta, S., Oro, D., Vera, J., Fernndez, C., Palmieri, C., Di Luccio, D., Giunta, G., Lapegna, M., Laccetti, G.: Accelerating linux and android applications on low-power devices through remote gpgpu offloading. Concurrency and Computation: Practice and Experience **29**(24), e4286 (2017). https://doi.org/10.1002/cpe.4286, https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.4286, e4286 cpe.4286
14. Oikawa, M., Kawai, A., Nomura, K., Yasuoka, K., Yoshikawa, K., Narumi, T.: Ds-cuda: A middleware to use many gpus in the cloud environment. pp. 1207–1214 (2012)
15. Reaño, C., Silla, F.: A performance comparison of cuda remote gpu virtualization frameworks. In: 2015 IEEE International Conference on Cluster Computing. pp. 488–489. IEEE (2015)
16. Reaño, C., Silla, F.: Reducing the performance gap of remote gpu virtualization with infiniband connect-ib. In: 2016 IEEE Symposium on Computers and Communication (ISCC). pp. 920–925. IEEE (2016)
17. Reaño, C., Silla, F., Shainer, G., Schultz, S.: Local and remote gpus perform similar with edr 100g infiniband. In: Proceedings of the Industrial Track of the 16th International Middleware Conference. p. 4. ACM (2015)
18. Tange, O., et al.: Gnu parallel-the command-line power tool. The USENIX Magazine **36**(1), 42–47 (2011)