Virtualizing CUDA enabled GPGPUs on ARM clusters

Raffaele Montella[1], Giulio Giunta[1], Giuliano Laccetti[2], Marco Lapegna[2]
Carlo Palmieri[1], Carmine Ferraro[1], Valentina Pelliccia[1]

[1] University of Napoli Parthenope, Department of Science and Technologies, Napoli, Italy
{raffaele.montella, giulio.giunta, carlo.palmieri
carmine.ferraro, valentina.pelliccia }@uniparthenope.it
[2] University of Napoli Federico II, Department of Mathematics and Applications, Napoli, Italy
{giuliano.laccetti, marco.lapegna }@unina.it

Abstract. Tiny ARM based devices are the backbone of the Internet of Things technologies, nevertheless the availability of high performance multicore lightweight CPUs pushed the High Performance Computing to hybrid architectures leveraging on diverse levels parallelism. In this paper we describe how to accelerate inexpensive ARM-based computing nodes with high-end CUDA enabled GPGPUs hosted on x86 64 machines using the GVirtuS general-purpose virtualization service. We draw the vision of a possible hierarchical remote workload distribution among different devices. Preliminary, but promising, performance evaluation data suggests that the developed technology is suitable for real world applications.

1 Introduction

In the quest for the enormous benefits that Exascale applications promise, the Top500 ranking and its greener counterpart, the Green500 list, show an impressive 6× improvement in the performance-power ratio of large-scale high performance computing (HPC) facilities over the last five years. Furthermore, a trend clearly visible in these two lists is the adoption of hardware accelerators to attain unprecedented levels of raw performance with reasonable energy costs, which hints that future Exaflop systems will most likely leverage some sort of specialized hardware [13].

On the other hand, virtualization technologies are currently widely deployed, as their use yields important benefits such as resource sharing, process isolation, and reduced management costs. Thus, it is straightforward that the usage of virtual machines (VMs) in HPC is an active area of research. VMs provide an improved approach to increase resource utilization in HPC clusters, as several different customers may share a single computing node with the illusion that they own it in an exclusive way.

Cloud computing offers an appealing elastic infrastructure, but whether and how it can provide the effective high performances required by most e-science applications is still a research issue. Especially in the field of parallel computing applications, virtual clusters instanced on cloud infrastructures suffers from the poorness of message passing performances between virtual machine instances running on the same real machine and also from the impossibility to access hardware specific accelerating devices as GPUs Recently, scientific computing has experienced on general purpose graphics processing units to accelerate data parallel computing tasks [12].

One the most successful GPU based accelerating system is provided by nVIDIA and relies on the CUDA programming paradigm supporting high level languages tools. An active research field is currently focused on suitably exploiting special purpose processors as accelerators for general-purpose scientific computations in the most efficient and cost-effective way [14].

Presently, virtualization issued by popular hypervisors (XEN, KVM, Virtual Box, VMWare) does not allow a transparent use of accelerators as CUDA based GPUs, as virtual/real machines and guest/host real machines communication issues rise serious limitations to the overall potential performance of a cloud computing infrastructure based on elastically allocated resources.

In this paper we present the updated component GVirtuS (Generic Virtualization Service) as results in GPGPUs transparent virtualization targeting mainly the use of nVIDIA CUDA based accelerator boards through virtual machines instanced to accelerate scientific computations [1]. In the latest GVirtuS incarnation we enforced the architecture independence making it working with both CUDA and OpenCL on Intel and ARM architecture as well with a clear roadmap heading to Power architectures compatibility. The rest of the paper is organized in the following way: the following section, the number two, is about how GVirtuS works on different architectures while the following section, the number three, is dedicated to the architecture, the design and the implementation of the latest version of GVirtuS; the forth section is experiment setup for different scenario; the preliminary evaluation results are shown in the section number five; in the section seven the current version of GVirtuS is compared and contrasted with other notable related works; finally the last section, the number seven, is about the conclusions and the future directions of this promising research.

2 GVirtuS on heterogeneous architectures

An ARM port of GVirtuS is motivated raised from different application fields such as High Performance Internet of Things (HPIoT) and cloud computing. In HPC infrastructures, ARM processors are used as computing nodes often provided by tiny GPU on chip or integrated on the CPU board [5].

Nevertheless, for most massively parallel processing applications, as scientific computing, are too compute intensive to run well on the current generation of ARM chips with integrated GPU.
In this context we developed the idea to share one or more regular high-end GPU devices hosted on a small number of x86 machines with a good amount of low power/low cost ARM based computing sub-clusters better fitting into the HPC world.

From the architectural point of view this is a big challenge for reasons of word size, endianess, and programming models. For our prototype we used the 32-bit ARMV6K processor supporting both big and little endian so we had to set the little endian mode in order to make data transfer between the ARM and the x86 full compliant. Due to the prototypal nature of the system all has been set to work using 32 bits. The solution is the full recompilation of the framework with a specific reconfiguration of the ARM based system. As we will migrate on 64 bits ARMs this point will be revise.

In order to fit the GPGPU/x86 64/ARM application into our generic virtualization system we mapped the back-end on the x86 64 machine directly connected to the GPU based accelerator device and the front-end on the ARM board(s) using the GVirtuS tcp/ip based communicator. GVirtuS as nVidia CUDA virtualization tool achieves good results in terms of performances and system transparency.

The CUDA applications are executed on the ARM board through the GVirtuS front-end. Thanks to the GVirtuS architecture, the front-end is the only component needed on the guest side.
This component acts as a transparent virtualization tool giving to a simple and inexpensive ARM board the illusion to be directly connected to a high-end CUDA enabled GPGPU device or devices (Fig. 1).

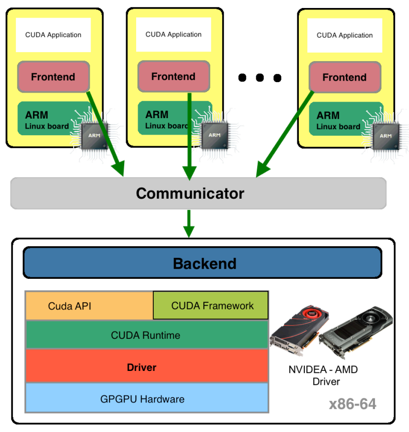

Fig. 1. The GVirtuS architecture. This diagram points out about the computing architecture (ARM, x86_64) and acceleration model (CUDA, OpenCL) independence.

In the work presented in this paper, we chose to design and implement a GVirtuS plugin implementing CUDA 6.5 APIs. This has been strongly motivated by several issues:

Since CUDA version 4, the library design appears to no longer fit with the split driver approach leveraged by GVirtuS and other similar products, so a consistent update is needed;
CUDA applications can be compiled directly on the ARM board with the installation of ad hoc libraries available from nVidia.
While OpenCL is intrinsically open and all interfaces are public and well documented, nVidia devices at now offer a limited support to it [7].

In this paper we focus on acceleration services provided by the nVIDIA in order to support the CUDA programming model on heterogeneous architectures. More exactly, we target two main goals: to provide a fully transparent virtualization solution (CUDA enabled software has to be executed in a virtual environment without any further modification of binaries or source code) and to reduce the overhead of virtualization so that the performance of the virtualized solution is as close as possible to the one of the bare metal execution.

The GVirtuS framework is implemented as a collection of C++ classes. The utility library contains the common software components used by both frontend and backend. This library provides the support for communication between the frontend and the backend. The communicator component is implemented behind an interface called Communicator. GVirtuS already provides several Communicator subclasses such as TCP/IP, Unix sockets, VMSocket (high performance communicator for KVM based virtualization) and VMCI (VMWare efficient and effective communication channel for VMWare based virtualization). The Communicator interface strongly resembles the POSIX socket interface. This choice is due to the fact that using a well-known interface facilitates to understand and implement the communicator interface.

The frontend library allows the development of the frontend component and contains a single class, called Frontend. There is only one instance of this class for each application using the virtualized resource. This instance is in charge of the backend connection, which is based on an implementation of a Communication interface. This is a critical issue especially when the virtualized resources have to be thread-safe as in the case of GPUs providing CUDA support. The methods implemented in this class support request preparation, input parameters management, request execution, error checking and output data recovery.

The backend library structure is slightly more complex than the frontend and it is composed of three classes:

- Backend is a partially abstract class, which contains methods for starting the backend server and for installing new handlers as new instances of subclasses of the Handler class;

- Handler is a completely abstract class with a single method used to execute a service request (handler).

- Process is the class serving the requests from a single frontend; it receives the requests from the frontend and executes them using the method of the Handler subclass.

The backend is executed on the host machine. It waits for connections from frontends. As a new connection is incoming it spawns a new thread for serving the frontend requests. The CUDA enabled application running on the virtual machine requests services 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.

In order to implement the nVIDIA/CUDA stack split-driver using GVirtuS a developer has to implement the Frontend, Backend and the Handler subclasses. For CUDA runtime virtualization the Handler is implemented as a collection of functions and a jump table for the specified service. The frontend has been implemented as a dynamic library based on the interface of the original libcudart.so library.

As an improvement over the nVIDIA/CUDA virtualization offered by gVirtuS, we used the general-purpose virtualization service GVirtuS to provide virtualization support for CUDA, openCL and openGL. The CUDA driver implementation is similar to the CUDA runtime except for the low-level ELF binary management for CUDA kernels. A slightly different strategy has been used for openCL and openGL support. The openCL library provided by nVIDIA is a custom implementation of a public specification. That means that the openCL GVirtuS wrap is independent of the openCL implementation. Based on the implemented split driver a software using openCL running on a virtual machine could use nVIDIA or ATI accelerated devices physically connected to other remote machines in a transparent and dynamic way. As in the case of openCL, openGL wrap is a custom implementation of a public specification. In order to support remote visualization in a high performance way a VNC based screen virtualization had to be implemented.

3 Architecture, design, implementation

The latest implementation of GVirtuS extends and generalizes a previously developed GPU virtualization solution proposed in our past works. The main motivation of the first generation of GVirtuS was to address the limitations of transparently employing accelerators such as CUDA-based GPUs in virtualization environments. GVirtuS could be considered as a generic virtualization framework for facilitating the development of split- drivers for virtualization solutions.

The brightest GVirtuS feature is the independence from all involved technologies: the hypervisor, the communicator and the target of the virtualization, as demonstrated later in this work.

Using a plug-in based design, GVirtuS offers virtualization support for generic libraries such as accelerator libraries (OpenCL, OpenGL, CUDA) and parallel file systems and communication libraries (MPI / OpenMP).

GVirtuS could be seen as an abstraction layer for generic virtualization in HPC on cloud infrastructures.

In GVirtuS the split-drivers are abstracted away, while offering developers abstractions of common mechanisms, which can be shared for implementing the desired functionality.

In this way, developing a new virtualization driver is simplified, as it is based on common utilities and communication abstractions. For each virtualized device the frontend and the backend are cooperating, while both of them are completely independent from the communicator. Developers can focus their efforts on virtual device and resource implementation without taking care of the communication technology.

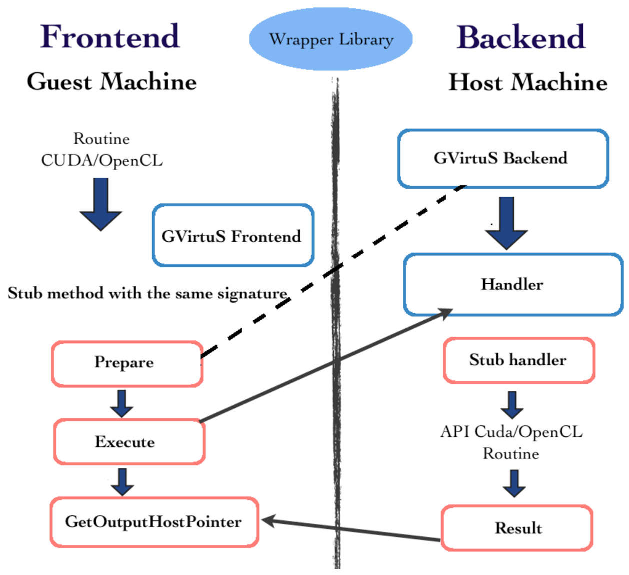

Fig. 2. The GVirtuS approach to the split driver model.

3.1 The front-end

The front-end is a kernel module that uses the driver APIs supported by the platform. The interposer library provides the familiar driver API abstraction to the guest application. It collects the request parameters from the application and passes them to the back- end driver, converting the driver API call into a corresponding front-end driver call.

When a callback is received from the front-end driver, it delivers the response messages to the application. In GVirtuS the front-end runs on the virtual machine instance and it is implemented as a stub library. A stub library is a virtualization of the physical driver library on the guest operating system. The stub library implements the driver functionality in the guest operating system in cooperation with the back-end running on the host operating system.

In this work we describe the implementation of the front-end plugins implementing CUDA version 6.5 and OpenCL.

3.2 The CUDA front-end

The CudaRTFrontend Class establishes connections with the back-end and executes the cuda routine through the compiled library libGvirtus-frontend. The constructor method creates an object of the class Frontend from the libGvirtus-frontend library using the method GetFrontend using a factory/instance design pattern. This instance of the class will be alive through all the life cycle of the application and it will be used any time we need a method from the CudaRTFrontend class. The stub methods all have a common schema. Every stub follows the same interface of the handled Cuda routine. The description of this method is significant for the explanation of any other method. The first step is to get the unique instance of the GVirtuS Frontend class. This task is accomplished by the constructor method. The Prepare method resets the input buffer that will contain the parameters to send to the back-end. After that all the parameters are inserted into the buffer. The execute method forwards the request for the routine using the name of the routine as parameter. If the method is successfully executed, we can get the output parameters. At last the method GetExitCode returns the exit code of the routine executed by the back-end. Using cudaGetDeviceCount and cudaSetDevice it is possible to get info abot all the devices attached to the physical machine. This simple explicative schema is common to all the stubs coded.

3.3 The OpenCL front-end

The OpenCLFrontend Class establishes connections with the back-end and executes the OpenCL routine through the compiled library libGvirtus-frontend.

The constructor method creates an object of the class Frontend from the libGvirtus-frontend library using the method GetFrontend using a factory/instance design pattern. This instance of the class will be alive through all the life cycle of the application and it will be used any time we need a method from the OpenCLFrontend class.

All the stubs methods have a common schema. Every stub follows the same interface of the handled OpenCL routine. In the code snippet below is shown the stub for OpenCL::GetDeviceIDs(). The description of this method is significant for the explanation of any other method.

The first step is to get the unique instance of the GVirtus' Frontend class. This task is accomplished by the constructor method.

The Prepare method reset the input buffer that will contain the parameters to send to the back-end. After that all the parameters are inserted in to the input buffer. The execute method forward the request for the routine using the name of the routine as parameter. If the method is successfully executed so we can get the output parameters. At last the method GetExitCode returns the exit code of the routine executed by the backend.

The clGetDeviceIDs routine can be used to obtain the list of available devices on a platform.

This simple explicative schema is common to all the stubs coded.

3.4 The back-end

The backend is a component serving frontend requests through the direct access to the driver of the physical device. This component is implemented as a server application waiting for connections and responding to the requests submitted by front-ends. In an environment requiring shared resource access (as it is very common in cloud computing), the back-end must offer a form of resource multiplexing. Another source of complexity is the need to manage multithreading at the guest application level.
A *daemon* runs on the host operating system in the user space or super user space depending on the specifics of applications and security policies. The daemon implements the back-end functionality dealing with the physical device driver and performing the host-side virtualization.

3.5 The CUDA back-end

An ad-hoc file named gvirtus.properties configures GVirtuS. GVirtuS only handles two parameters at this stage of development, communicator and plugin. The first parameter selects which kind of communicator has to be used choosing from a list of available communication mechanisms, the second one selects which plugin must be loaded. The main task of GVirtuS back-end is to start a communication in server mode and waiting then accepting new incoming connections. It handles the loading of plugins previously installed. CudaRTHandler class contains all the methods needed in order to serve the requests of cuda routine execution. In the CudaRTHandler class there is a table, mspHandlers, associating function pointers to the name of the routines, so any routine can be handled in the right way. As in the front-end there is a stub method for each cuda method, in the back-end there is a function managing the execution of each method.

3.6 The OpenCL back-end

As previously stated about the CUDA plug-in, GVirtuS back-end invokes the GetHandler method in order to create a new instance of OpenclHandler class containing all the methods needed in order to serve the requests of OpenCL routine execution. In this class it's possible to find all the methods to handle the execution of OpenCL routines. In the OpenclHandler class there is a table, mpsHandlers, associating function pointers to the name of the routines, so any routine can be handled in the right way. As in the front-end there is a stub method for each OpenCL method, in the back-end there is a function managing the execution of each method.

3.7 The communicator component

A *communicator* is a key piece of software in GVirtuS stack because it connects the guest and host operating systems. The communicators have strict high-performance requirements, as they are used in system-critical components such as split-drivers. Additionally, in a virtual machine environment the isolation between host and guest and among virtual machines is a design requirement. Consequently, the communicator main goal is to provide secure high-performance direct communication mechanisms between guest and host operating systems.

In GVirtuS the communicators are independent of hypervisor and virtualized technology. Additionally, novel communicator implementations can be provided independently from the cooperation protocols between front-end and back-end.

GVirtuS provides several communicator implementations including a TCP/IP communicator. The TCP/IP communicator is used for supporting virtualized and distributed resources. In this way a virtual machine running on a local host could access a virtual resource physically connected to a remote host in a transparent way. However, in the some application scenarios the TCP/IP based communicator is not feasible because of the following limitations:

- The performance is strongly impacted by the protocol stack overhead.
- In a large size public or private cloud computing environment the use of the network could be restricted for security and performances reasons.
- For protection reasons a virtual machine may be unaware of the network address of its hosting machine.

For addressing these potential limitations, GVirtuS open solution allows for future protocols to be simply integrated into the architecture without any frontend or backend modification.

4 Scenarios and prototypal applications

The main contribution of this paper is an experimental evaluation of the possibilities that state-of-the-art technology offers in today's HPC facilities, as well as low-power alternatives offer for the acceleration of scientific applications using remote graphics processors.

Our test setup involves a Tesla based development workstation, GPU provided Amazon Web Service Elastic Cloud Computing instances and inexpensive ARM based single board computers as tiny computing nodes of an experimental Beowulf cluster.

4.1 The development workstation

Workstation Genesis GE-i940 Tesla equipped with an i7- 940 2,93 133 GHz fsb, Quad Core hyper-threaded 8 Mb cache CPU and 12Gb RAM. The GPU subsystem is enforced by one nVIDIA Quadro FX5800 4Gb RAM video card and two nVIDIA Tesla C1060 4 Gb RAM summing up 720 CUDA cores. The testing system has been built on top of the Ubuntu 12.04 Linux operating system, the nVIDIA CUDA Driver, and the SDK/Toolkit version 6.5. Due to the obsolescence of the Tesla devices, this machine is used just as a development workstation providing really poor computing capabilities in class 1.x.

4.2 The Amazon Elastic Cloud Computing GPU machine

Amazon Elastic Compute Cloud (Amazon EC2) is a web service that provides resizable compute capacity in the cloud. It is designed to make web-scale cloud computing easier for developers.

Amazon EC2's simple web service interface allows you to obtain and configure capacity with minimal friction. It provides you with complete control of your computing resources and lets you run on Amazon's proven computing environment. Amazon EC2 reduces the time required to obtain and boot new server instances to minutes, allowing you to quickly scale capacity, both up and down, as your computing requirements change. Amazon EC2 changes the economics of computing by allowing you to pay only for capacity that you actually use. Amazon EC2 provides developers the tools to build failure resilient applications and isolate themselves from common failure scenarios.

We used an AWS g2.2xlarge instance intended for graphics and general-purpose GPU compute applications. Those virtual machines run on High Frequency Intel Xeon E5-2670 (Sandy Bridge) Processors. The g2 instances are provided by high-performance NVIDIA GPUs, each with 1,536 CUDA cores and 4GB of video memory. Each GPU features an on-board hardware video encoder designed to support up to eight real-time HD video streams (720p@30fps) or up to four real-time full HD video streams (1080p@30fps). This kind of instance supports low-latency frame capture and encoding for either the full operating system or select render targets, enabling high-quality interactive streaming experiences. Nevertheless we used this instance in order to setup a remote elastic virtual GPGPU environment used locally by tiny ARM based devices and/or regular x86_64 machines. The AWS g2 GPU instance provides CUDA computing capabilities 2.x thanks a CUDA Grid K250 device.

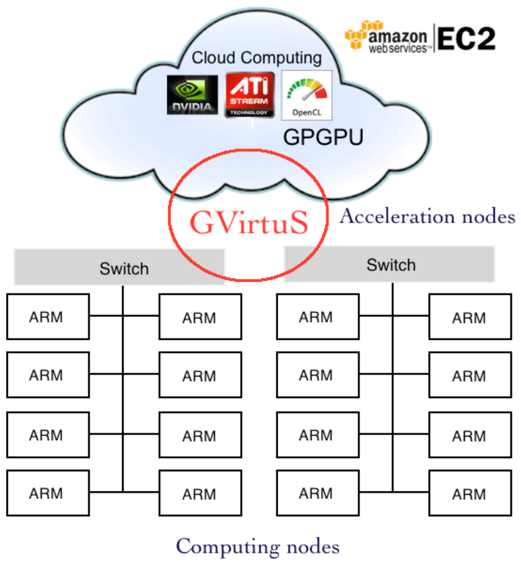

Fig. 3. Sharing high-end GPU accelerated devices hosed by x86_64 machines among inexpensive Beowulf clusters based on ARM.

4.3 The inexpensive ARM based computing node

UDOO is a multi development platform solution for Android, Linux, Arduino™ and Google ADK 2012. The board is designed to provide a flexible environment that allows exploring the new frontiers of the Internet of Things. UDOO allows you to switch between Linux and Android in a few seconds, simply by replacing the Micro SD card and rebooting the system. UDOO is compatible with all the sketches, tutorials and resources available on the Arduino community as well as all the shields, sensors and actuators for Arduino DUE available on the market. We used an UDOO Quad single computer board equipped by a Freescale i.MX 6 ARM Cortex-A9 CPU Quad core 1GHz with a custom version of Ubuntu 12.04 Linux as GVirtuS consumer. This board is supported by 1GB DDR3 RAM, Ethernet RJ45 (10/100/1000 MBit), SATA interface and a boot device base on Micro SD. The UDOO Quad single board computer has an integrated graphics capabilities, each processor provides 3 separated accelerators for 2D, OpenGL® ES2.0 3D and OpenVG™. In this work we intentionally neglected the on-board GPU for computing tasks.

Our main goal is to access a remote Cuda GPGPU on an arm powered SoC in order to provide or expand its Cuda capabilities. The following scenarios were tested:

- Backend on X86_64/Frontend on X86_64;
- Backend on X86/Frontend on X86;
- Backend on AWS X86/Frontend on X86;
- Backend on AWS X86/Frontend on ARM 32bit.

4.4 The benchmark experiment setup

Matrix multiplication is an implementation of matrix multiplication that does take advantage of shared memory. In this implementation, each thread block is responsible for computing one square sub-matrix Csub of C and each thread within the block is responsible for computing one element of Csub. Csub is equal to the product of two rectangular matrices: the sub-matrix of A of dimension (A.width, block_size) that has the same row indices as Csub, and the sub-matrix of B of dimension (block_size, A.width) that has the same column indices as Csub. In order to fit into the device's resources, these two rectangular matrices are divided into as many square matrices of dimension block_size as necessary and Csub is computed as the sum of the products of these square matrices. Each of these products is performed by first loading the two corresponding square matrices from global memory to shared memory with one thread loading one element of each matrix, and then by having each thread compute one element of the product. Each thread accumulates the result of each of these products into a register and once done writes the result to global memory.

Vector addition is a very basic sample that implements element by element vector addition. Used to test the simplest cuda functionalities.

Sorting networks implements bitonic sort and odd-even merge sort, algorithms belonging to the class of sorting networks. While generally subefficient on large sequences compared to algorithms with better asymptotic algorithmic complexity (i.e. merge sort or radix sort), may be the algorithms of choice for sorting batches of short- or mid-sized arrays.

5 Preliminary evaluations

After running several tests, we can for sure assert the effectiveness of the designed infrastructure. Coming to performances we can evaluate the overhead introduced by GVirtuS, but we have to evaluate also the chance to run CUDA code on no CUDA enabled devices.

The main bottleneck is the communications overhead, in the test environment we only used TCP communicator, resulting in poor performances when we tested our infrastructure over the Internet, while backend and frontend were on the same machine or on the same local network the resulted overhead is acceptable.

For more details about performances you can look at the following tables.

5.1 GPGPU Virtualization

This experiment setup involves the development machine hosting a virtual machine running inside the user space provided by a hypervisor. In this contest we are neglecting the optimization of the communicator component focusing on the interoperability of the TCP/IP channel.

Table 1. Backend on X86_64/Frontend on X86_64.

Test	With GVirtuS	Without GVirtuS
Matrix Multiplication	0.139 sec	0.092 sec
Vector Addition	0.063 sec	0.059 sec
Sorting Networks	8.787 sec	8.539 sec

Table 1 reports the results of running both the frontend and the backend on the same machine. The Table 2 demonstrate the behavior of the GVirtuS setup using a 32 bit environment. The limitation issued by our development machine avoided any test sharing CUDA devices with ARM based boards.

Table 2. Backend on X86/Frontend on X86.

Test	With GVirtuS	Without GVirtuS
Matrix Multiplication	0.149 sec	0.098 sec
Vector Addition	0.067 sec	0.057 sec
Sorting Networks	8.676 sec	8.482 sec

5.1 Elastic remote GPGPU sharing

We explored a scenario in which GPUs are hosted elastically on a public cloud in a infrastructure as a service fashion. We setup the backend on a g2.x2large AWS EC2 instance.

Table 3. Backend on AWS X86/Frontend on X86.

Test	With GVirtuS	Without GVirtuS
Matrix Multiplication	38.236 sec	0.098 sec
Vector Addition	3.298 sec	0.057 sec
Sorting Networks	2min24.648 sec	8.482 sec

The Table 3 represents the results of our test suite executed on a local machine sharing CUDA enabled GPUs available on the cloud. The Table 4 is about the same tests, but executed locally on an ARM based single board computer.

Table 4. Backend on AWS X86/Frontend on ARM 32bit.

Test	With GVirtuS	Without GVirtuS
Matrix Multiplication	50.607 sec	N/A (Can’t run)
Vector Addition	3.368 sec	N/A (Can’t run)
Sorting Networks	4min17.547 sec	N/A (Can’t run)

6 Related works

While GVirtuS is a transparent and VMM independent framework to allows an instanced virtual machine to access GPUs implementing various communicator components (TCP/IP, VMCI for VMware, VMSocket for KVM) to connect the front-end in guestOS and back-end in hostOS, rCUDA [3], GViM [4] and vCUDA [2] are three recent research projects on CUDA virtualization in GPU clusters and virtual machines as GPGPU library.

They all use an approach similar to GVirtuS.

The rCUDA framework creates virtual CUDA-compatible devices on those machines without a local GPU to enable a remote GPU-based acceleration, and communicate using the sockets API between the front-end and back-end. rCUDA can be used in the virtualization environment, but in the earlier versions, it requires the programmer to rewrite the CUDA applications to avoid the use of the CUDA C extensions, and requires to change the standard compile options to separate the host and device code into different files. Recently rCUDA adopted a transparent virtualization model really similar to GVirtuS including the ARM support, nevertheless the source code is not publically available and the license in really restrictive about compare and contrast performance evaluation.

GViM is a Xen-based system, allows virtual machines to access a GPU through XenStore between a front-end executed on the VM and a back-end on the Xen Domain0. GViM requires modification to the guest VMs running on the virtualized platform, a custom kernel module must be inserted to the guestOS.

vCUDA is characterized by a general architecture really similar to GVirtuS. In particular the design of communicator leverages on VMRPC. The authors believe RPC-based optimization that exploits the inherent semantics of the application, such as data presentation, would perform better than those merely optimizing data transfer channels as it happens in GVirtuS. vCUDA supports suspend/resume functionality of CUDA applications.

7 Conclusions

In this paper has been presented our preliminary results about the design and the implementation of an updated CUDA wrapper library as GVirtuS framework plugin in order to accelerate sub-clusters of inexpensive low power demanding ARM based boards using high end GPGPU devices. In a previous work we chose OpenCL as parallel programming computing model because it's independence from any kind of architectural constraints, nevertheless the now days CUDA rules on GPGPUs. The most challenging result achieved by our work described in this paper is the implementation of a base tool unchaining the development of really distributed and heterogenic hardware architectures and software applications. The experiments we performed demonstrate how is convenient the path we followed as trailblazer in the hunt for the next big thing in the off the shelf commodity high performance computing clusters. We setup a sub-cluster made by high performance ARM based boards provided by multicore ARM 64bit CPUs and high bandwidth network interfaces experiencing important improvements from the ARM side, but even a better scalability because a more performing communication. The second experiment

setup involved x86_64 GPU accelerated back-ends running in Amazon Elastic Computer Cloud. In this scenario some other actors will get playing as the use of MPICH [9] for ARM to ARM and ARM to x86_64 message passing, the OpenMP[8] for intra ARM board parallelism and, above all, one or more GPU devices hosted on the accelerator node have to be multiplexed by several ARM processes.
The final destination of this research is the provisioning of a full production software environment for advanced earth system simulations and analysis based on science gateways, workflow engines and high performance cloud computing [10,7] giving a support for the next generation of scientific dissemination tools [11].

Acknowledgments. This research was supported in part 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. Giunta, G., Montella, R., Agrillo, G., & Coviello, G. (2010). A GPGPU transparent virtualization component for high performance computing clouds. In *Euro-Par 2010-Parallel Processing* (pp. 379-391). Springer Berlin Heidelberg.
2. Shi, L., Chen, H., Sun, J., & Li, K. (2012). vCUDA: GPU-accelerated high-performance computing in virtual machines. *Computers, IEEE Transactions on*, *61*(6), 804-816.
3. Duato, J., Pena, A. J., Silla, F., Mayo, R., & Quintana-Ortí, E. S. (2010, June). rCUDA: Reducing the number of GPU-based accelerators in high performance clusters. In *High Performance Computing and Simulation (HPCS), 2010 International Conference on* (pp. 224-231). IEEE.
4. Gupta, V., Gavrilovska, A., Schwan, K., Kharche, H., Tolia, N., Talwar, V., & Ranganathan, P. (2009, March). GViM: GPU-accelerated virtual machines. In *Proceedings of the 3rd ACM Workshop on System-level Virtualization for High Performance Computing* (pp. 17-24). ACM.
5. Montella, R., Giunta, G., & Laccetti, G. (2014). Virtualizing high-end GPGPUs on ARM clusters for the next generation of high performance cloud computing. *Cluster computing*, *17*(1), 139-152.
6. Di Lauro, R., Giannone, F., Ambrosio, L., & Montella, R. (2012, July). Virtualizing general purpose GPUs for high performance cloud computing: an application to a fluid simulator. In *Parallel and Distributed Processing with Applications (ISPA), 2012 IEEE 10th International Symposium on* (pp. 863-864). IEEE.
7. Laccetti, G., Montella, R., Palmieri, C., & Pelliccia, V. (2014). The High Performance Internet of Things: Using GVirtuS to Share High-End GPUs with ARM Based Cluster Computing Nodes. In *Parallel Processing and Applied Mathematics* (pp. 734-744). Springer Berlin Heidelberg.
8. Dagum, L., & Menon, R. (1998). OpenMP: an industry standard API for shared-memory programming. *Computational Science & Engineering, IEEE*, *5*(1), 46-55.

9. Gropp, W. (2002). MPICH2: A new start for MPI implementations. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface* (pp. 7-7). Springer Berlin Heidelberg.
10. Montella, R., Brizius, A., Elliott, J., Kelly, D., Madduri, R., Maheshwari, K., ... & Foster, I. (2014, November). FACE-IT: a science gateway for food security research. In *Proceedings of the 9th Gateway Computing Environments Workshop* (pp. 42-46). IEEE Press.
11. Pham, Q., Malik, T., Foster, I., Di Lauro, R., & Montella, R. (2012). SOLE: linking research papers with science objects. In *Provenance and Annotation of Data and Processes* (pp. 203-208). Springer Berlin Heidelberg.
12. Foster, I., Zhao, Y., Raicu, I., & Lu, S. (2008, November). Cloud computing and grid computing 360-degree compared. In *Grid Computing Environments Workshop, 2008. GCE'08* (pp. 1-10). Ieee.
13. Raicu, I., Foster, I. T., & Beckman, P. (2011, June). Making a case for distributed file systems at exascale. In *Proceedings of the third international workshop on Large-scale system and application performance* (pp. 11-18). ACM.
14. Yang, C. T., Huang, C. L., & Lin, C. F. (2011). Hybrid CUDA, OpenMP, and MPI parallel programming on multicore GPU clusters. *Computer Physics Communications*, *182*(1), 266-269.