Int J Parallel Prog (2017) 45:1142–1163
DOI 10.1007/s10766-016-0462-1

On the Virtualization of CUDA Based GPU Remoting on ARM and X86 Machines in the GVirtuS Framework

Raffaele Montella[1] · Giulio Giunta[1] · Giuliano Laccetti[2] · Marco Lapegna[2] · Carlo Palmieri[1] · Carmine Ferraro[1] · Valentina Pelliccia[1] · Cheol-Ho Hong[3] · Ivor Spence[3] · Dimitrios S. Nikolopoulos[3]

Received: 13 March 2016 / Accepted: 22 September 2016 / Published online: 13 October 2016

Abstract The astonishing development of diverse and different hardware platforms is twofold: on one side, the challenge for the exascale performance for big data processing and management; on the other side, the mobile and embedded devices for data

✉ Raffaele Montella
raffaele.montella@uniparthenope.it

Giulio Giunta
giulio.giunta@uniparthenope.it

Giuliano Laccetti
giuliano.laccetti@unina.it

Marco Lapegna
marco.lapegna@unina.it

Carlo Palmieri
carlo.palmieri@uniparthenope.it

Carmine Ferraro
carmine.ferraro@uniparthenope.it

Valentina Pelliccia
valentina.pelliccia@uniparthenope.it

Cheol-Ho Hong
c.hong@qub.ac.uk

Ivor Spence
i.spense@qub.ac.uk

Dimitrios S. Nikolopoulos
d.nikolopoulos@qub.ac.uk

1 University of Naples Parthenope, Naples, Italy

2 University of Naples Federico II, Naples, Italy

3 Queen's University of Belfast, Belfast, Northern Ireland, UK

collection and human machine interaction. This drove to a highly hierarchical evolution of programming models. GVirtuS is the general virtualization system developed in 2009 and firstly introduced in 2010 enabling a completely transparent layer among GPUs and VMs. This paper shows the latest achievements and developments of GVirtuS, now supporting CUDA 6.5, memory management and scheduling. Thanks to the new and improved remoting capabilities, GVirtus now enables GPU sharing among physical and virtual machines based on x86 and ARM CPUs on local workstations, computing clusters and distributed cloud appliances.

Keywords GPGPU · HPC · ARM · Cloud · Virtualization

1 Introduction

In the challenge for the enormous benefits of exascale applications, the Top500 ranking and its greener counterpart, the Green500 list, an impressive improvement is shown in the performance-power ratio of large-scale high performance computing (HPC) facilities over the last 5 years. Furthermore, a trend clearly visible in these two lists is the adoption of hardware accelerators to obtain unprecedented levels of raw performance with reasonable energy costs, which hints that future Exaflop systems will most likely leverage some sort of specialized hardware [41].

The virtualization currently provided by popular open source hypervisors (XEN, KVM, Virtual Box) does not allow software based transparent use of accelerators as CUDA based GPUs. VMWare and XEN support GPU on the basis of hardware virtualization provided natively by NVIDIA GRID devices instead [16].

High performance internet of things (HPIoT) and high performance cloud computing (HPCC) are typical examples of highly heterogeneous computing systems, where different devices and computing units coexist in the same software environment [8]. They can be described as highly parallel internet-based models providing virtualized and standard resources as a service over the Internet.

In this paper the evolution of our GVirtuS [12], generic virtualization service, enabling transparent GPGPU virtualization [26] and remoting [13] for low-power processors for, but not limited to, the acceleration of scientific applications is presented [29]. In the latest GVirtuS incarnation the architecture independence was enforced, in order to make it work with both CUDA and OpenCL on Intel and ARM architecture, as well as with a clear roadmap heading to Power architectures compatibility. The rest of the paper is organized in the following way: Sect. 2 is a brief technical introduction about GVirtuS, its design, architecture and implementation; Sect. 3 is a detailed description of the GVirtuS new features and how the heterogeneous architectures support has been enabled; Sect. 4 is about the experiment setup for different scenarios; Sect. 5 shows the evaluation results; in Sect. 6 the current version of GVirtuS with other notable related works are compared and contrasted; finally, Sect. 7 is about conclusions and future directions of this promising research.

2 GVirtuS: A Tool to Virtualize Heterogeneous Architectures

GVirtuS is a generic virtualization framework for virtualization solutions based on a split-driver model [1]. GVirtuS offers virtualization support for generic libraries such as accelerator libraries (CUDA, OpenCL), with the advantage of independence from all involved technologies: hypervisor, communicator and target of virtualization. This feature is possible thanks to the plug-in design of the framework, enabling the choice of different communicator or different stub-libraries mocking the virtualization target. GVirtuS is transparent for developers: no changes are required in the software source code to virtualize and execute and there is no need to recompile an already compiled executable.

Low-power processors as ARM or Intel technologies are employed in diverse and different environments for the resolution of highly complex scientific problems, because their low cost and reduced cooling needs. On the other hand, the use of ARM CPUs in HPC infrastructures is a cutting edge technology, but, apparently, not ready for the prime time. At present, most scientific applications are too demanding of high performance to run on the current generation of ARM CPUs, even when integrated with GPUs. To accelerate the use of ARM in science production, remoting capabilities in GVirtuS have been improved in order to share high-end GPU devices hosted on x86 machines with low power/low cost ARM based computing clusters. This implies important challenging issues from the architectural point of view, partially mitigated by the GVirtuS modular design. Some requirements had to be set firmly in order to make it possible, as the use of an ARM CPUs with endianness and word length coherent to the x86 ones.

2.1 Architecture, Design and Implementation

GVirtuS strictly depends on CUDA APIs version because the nature of the transparent virtualization and remoting. In this paper we show our results in GVirtuS development relying on the the CUDA 6.5 APIs. The use of this version is motivated by the following issues:

- After the release of the CUDA 3.0 APIs, the library design no longer fits the same split-driver approach used by GVirtuS and other similar products;
- The CUDA 6.5 APIs unchain the CUDA power on tiny low power ARM architecture: CUDA applications can be compiled directly on the ARM board if ad hoc libraries available from NVIDIA are installed;
- CUDA is strictly proprietary and not open source, making the use of a virtualization/remoting layer non trivial. The GVirtuS development is framed in a wider big picture where the target application requirements are CUDA 6.5 compliant.

Since the first public release, the GVirtuS development has been characterized by two main goals: providing a fully transparent virtualization/remoting solution; reducing the overhead of virtualization and remoting to make the performance of the virtualized solution as close as possible to the bare metal execution.

The front-end/back-end communication is abstracted by the communication interface concretely implemented by each communicator component. This issue is critical,

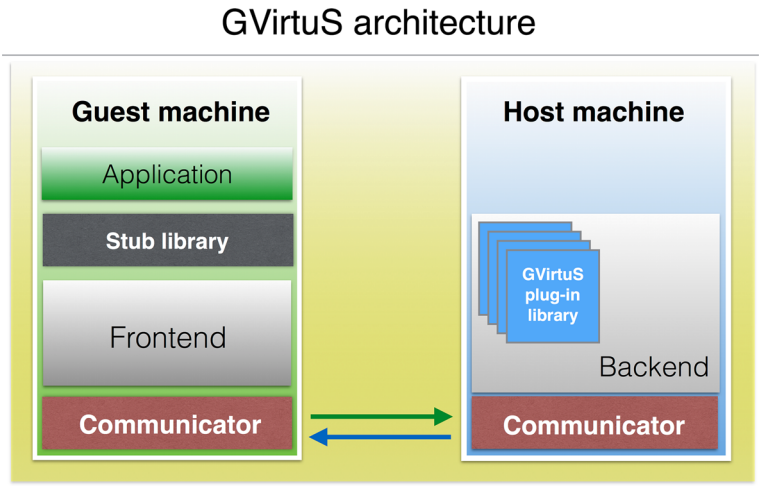

Fig. 1 The GVirtuS approach to the split-driver model

especially when the virtualized resources need to be thread-safe, as in 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 Handler class provides the base functionalities for each stub function management. The back-end is executed on the host machine behaving as a server component running as a user with enough privileges to interact with the CUDA driver. The back-end accepts a new connection spawning 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:

- Obtains a reference to the single Frontend instance;
- Uses Frontend class methods for setting the parameters;
- Invokes the Frontend handler method specifying the remote procedure name;
- 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 subclass from Frontend, Backend and Handler classes. For CUDA runtime virtualization the handler is implemented as a collection of functions and a jump table for a specified service. As in GVirtuS predecessor gVirtuS, in the case of CUDA runtime virtualization, the front-end has been implemented as a dynamic library based on the interface of the original libcudart.so library. Beginning with the second generation of GVirtuS component, the virtualization is focused on CUDA, but not limited to it. Thanks to the GVirtuS modularity and technology/architecture independence, the plug-ins for openCL and, partially, openGL have been developed. 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 (Fig. 1).

2.2 The Front-End

The front-end leverages on the driver's APIs supported by the platform running on a virtual machine instance or on a remote physical machine and is implemented as a stub-library. A stub-library is a virtualization of the real APIs library on the client operating system where the application is launched (typically a virtual machine or a physical one without GPU support). The stub-library implements the functionality of the host machine (GPU capable) on the guest machine. The role of the front-end is to intercept calls to the functions of APIs supported, transfer to the back-end the parameters passed to functions through the use of the selected communicator and wait for the execution result from the back-end. This result is made possible by the stub-library that provides the driver APIs abstraction to the guest application. When a client application calls a function, the stub-library intercepts the call and packs the serialized parameters in a buffer data structure. The front-end sends to the back-end the serialized buffer and the name of the function called through the communicator waiting for the response. For each method of the APIs there is a corresponding method for the management and the execution in the front-end.

2.3 The Back-End

The back-end is the main component of the GVirtuS framework and runs on the host machine (GPU capable). 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. When it receives a request, the back-end creates a new process and loads the plug-in needed for the requested function execution. After this operation, the back-end is ready for a new request from another guest machine. The new process reads the name of the API called, calls the associated method for managing the API required, allocates the space for the parameters of the method required and inserts the value from the parameters passed in the buffer from the front-end. The back-end calls the real API on the host machine through direct access to the driver of the physical device and saves the result in another buffer. Finally, the buffer result is passed to the front-end of the guest machine through the Communicator. To each method of the APIs corresponds a method for the management and the execution in the back-end.

2.4 The Communicator

The communicator is an important component of the GVirtuS framework connecting the front-end guest machine to the back-end host machine. The communicator is independent of hypervisor and virtualized technology. The communicators have strict high-performance requirements because they are used in system-critical components split-driver model compliant. The communicator provides a secure, high-performance, direct communication mechanism between the two sides of virtualization or remoting. The choice of the communicator depends on the physical machine connectivity, in both

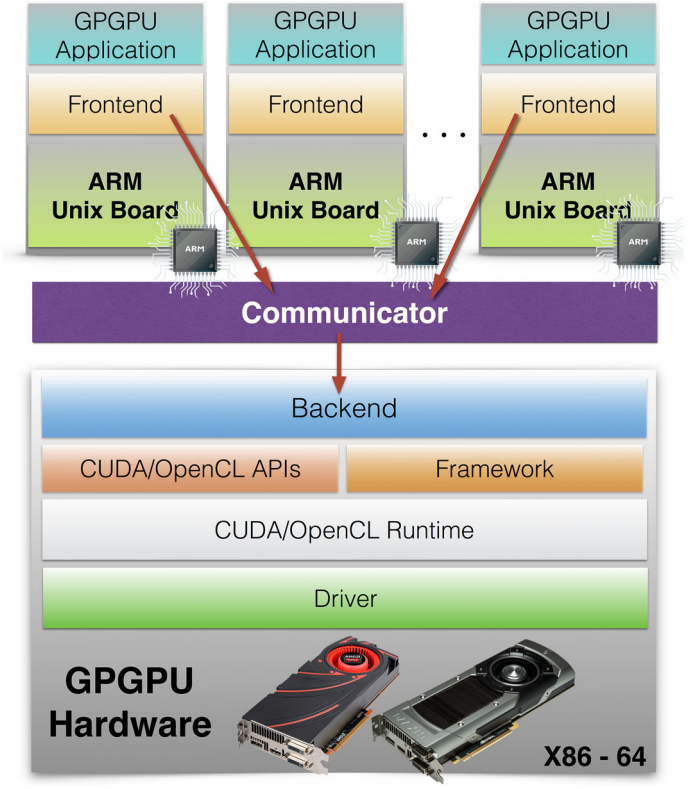

Fig. 2 The GVirtuS architecture

host and guest machines, because it influences the virtualization performance. GVirtuS provides several communicator implementations, including the 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 can access a virtual resource physically connected to a remote host in a transparent way. In practice, the communicator serializes the buffer structure and implements the transmission between host and guest.

3 Remoting and Novelty Introduced Features

In order to fit the GPGPU/x86/ARM application into our generic virtualization system, the back-end on the x86 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 have been mapped. GVirtuS as NVIDIA CUDA remoting and virtualization tool achieves good results in terms of performances and system transparency.

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 one or more high-end CUDA enabled GPGPU devices.

The diagram (Fig. 2) shows the computing architecture (ARM, x86_64) and the acceleration model (CUDA, OpenCL) independence. GVirtuS currently supports a growing subset of NVIDIA CUDA features. Thanks to the GVirtuS modular design, some new features have been developed, such as the GPU scheduling.

3.1 Unified Virtual Addressing (UVA) Management

The unified memory, introduced with CUDA version 6.x. simplifies the programming model by enabling applications to access CPU and GPU memory without the need to manually copy data from one to the other, and makes it easier to add support for GPU acceleration in a wide range of programming languages. When there is no distinction between a host and device pointer, CUDA runtime can identify where the data are stored and the correct value of the pointer. Essentially, in a unified virtual address any space allocated through the cudaMalloc, cudaMallocManaged, cudaHostAlloc or cudaMallocHost functions is mapped in a single unified space. As a consequence, for example, the direction of the copy in a cudaMemcpy function becomes obsolete so it is replaced by cudaMemcpyDefault. To support these features in GVirtuS, maps and lists have been used (Fig. 3).

Anytime a call to a function is made from the cudaMalloc family, the result pointer is stored in a list, so the nature of the pointer can be easily identified. When a call to cudaMemcpy is performed, the front-end can correctly identify the direction even when the cudaMemcpyDefault flag is selected. Direction mismatch is also avoidable, but this feature is not provided by CUDA, so it has not been used for this project. The nature of the pointers is determined by querying the list where the device pointers are stored. To support the UVA any time a managed pointer is allocated, this is stored in a map along with its size and a host pointer allocated through malloc function from glibc. When a managed pointer is involved in the execution, GVirtuS runtime takes care that data are passed to the back-end and stored on the device. Moreover, GVirtuS runtime takes care that the processed data are available on the front-end after the execution. When a managed pointer has to be used, the GVirtuS runtime searches for a match on the pointer map ensuring the coherence between the two virtualization/remoting address spaces. The memory pinned by the managed pointer is copied from the front-end to the back-end bounded with the valid device pointer. Finally, the GVirtuS runtime pushes in a stack the value of the host pointer.

All the pointers present in the stack after the computation are transferred from the back-end to the front-end, so that the processed data are available on the front-end side. At this stage, no significative overhead is introduced by the identification process. Nevertheless, in the UVA case a significant overhead is introduced, because any pointer involved in the calculation must be enforced by coherence in both directions (Fig. 4).

3.2 GPU Scheduling

The GPU scheduler of GVirtuS enables fair and efficient use of virtualized GPUs among multiple cloud users. The GPU scheduler multiplexes back-end processes

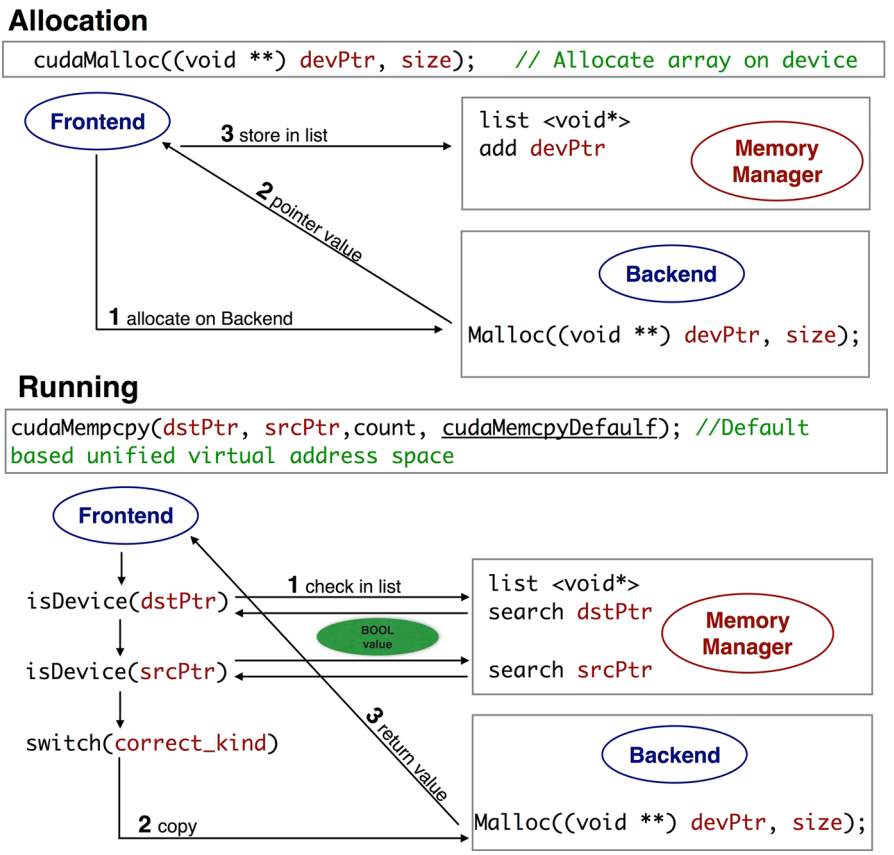

Fig. 3 Automatic memory management

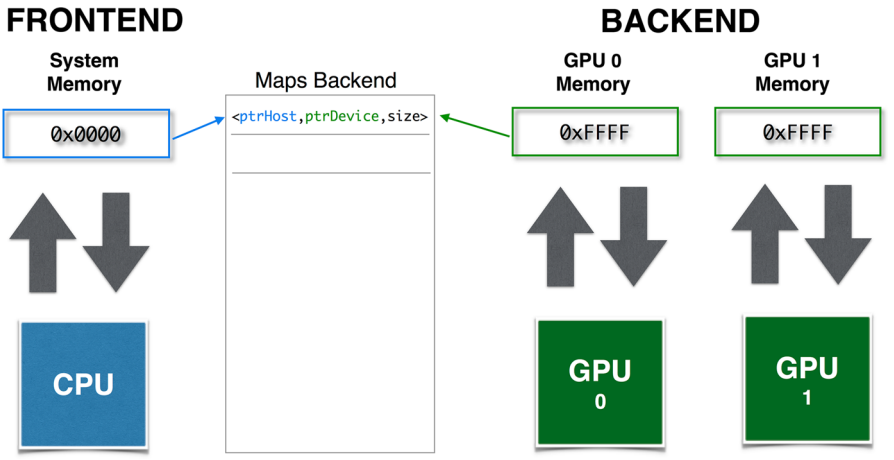

Fig. 4 Unified virtual address with back-end maps

spawned by the GVirtuS Backend driver; GVirtuS creates a back-end process whenever a connection between the split-drivers is established, and terminates it after the connection between them is closed. The scheduler maintains a run queue to accommodate runnable back-end processes and selects one of them to execute according to its fairness policy. It then gives a token to the chosen process in order to allow the process to exclusively access GPU devices during its time slice.

As a fairness policy, the GPU scheduler adopts the Credit scheduling policy, which is a proportional fair-share algorithm employed in the Xen hypervisor as a CPU scheduler. In this scheduling policy, the global credit accounting function periodically (30 ms) assigns a certain amount of credits to each back-end process in proportion to the GPU weight. The accounting function then decides the priority of each process based on the remaining credit amount. Similarly to the Xen hypervisor, the GPU scheduler maintains two priorities: UNDER and OVER. If the credit value of a back-end process is positive, its priority is set to UNDER. Otherwise, the priority becomes OVER. The accounting function then sorts the back-end processes into priority order (UNDER and OVER) in the run queue; for simplicity and fast sorting speed, the scheduler does not sort them based on the credit amount.

The scheduler selects the next back-end process to run in the head of the run queue at every scheduling instance; it implements the O(1) scheduling concept that can select the next process within a fixed amount of time. While the chosen process is using GPU devices, the credit value of the process is decreased at a fixed rate. After the time slice, the back-end process is put at the tail of its priority list. Before the global credit accounting function executes, its priority is maintained regardless of its current credit amount in order to reduce the frequency of sorting. This whole procedure reflects the motivation of credit scheduling, which focuses on efficient scheduling decision and simple implementation.

As depicted in Fig. 5, the GPU scheduler is placed in the user space of the host OS rather than in the kernel space in order to communicate with back-end processes more efficiently. It utilizes POSIX shared memory and real time signal mechanisms for synchronous and asynchronous communication respectively. When a back-end process is created, it notifies the GPU scheduler of its process ID by a signal. The GPU scheduler then inserts the process in the GPU run queue. When a timer alarm event is sent to the GPU scheduler, the scheduler decides the next back-end process to execute based on the Credit scheduling policy. The scheduler then revokes a token from the previous process and delivers it to the next process via the shared memory. In our implementation, the time slice of a running back-end process is configured to 6 ms, which can be adjusted by the administrator.

4 Scenarios and Prototypal Applications

Designing a test plan for the application scenario described in this paper is a complex issue because, while many legacy CUDA enabled applications and widely accepted performance test cases are available for the x86_64 architecture, the same is not true for ARM. So we had to face the lack of standard testing guidelines for CUDA and ARMs due to the weak available support for this technology, relatively new for CUDA

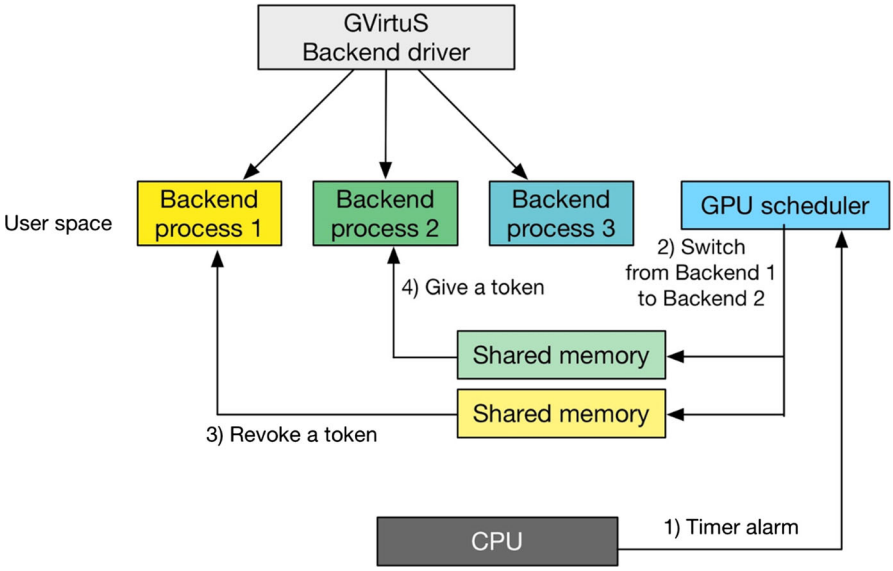

Fig. 5 Procedure of GPU back-end process switching from back-end process 1–2

environment. From the hardware point of view, our test setup involves a Maxwell based development workstation and a cluster built of 3 ARM based high-end single board computers (SBCs).

In order to test performance for x86_64/x86_64/GPU two different evaluation software have been used: CUDASW++ [23] and SRAD [36]. The first is a bioinformatics software for Smith–Waterman protein database searches, while the latter is a diffusion method for ultrasonic and radar imaging applications based on partial differential equations. Both applications take advantage of the massively parallel CUDA architecture of NVIDIA.

ARM/x86/GPU performance tests have been produced developing an ad hoc MPI Matrix Multiplication software [40] enabling the GVirtuS behaviour investigation setting up a scenario where a x86 machine is used as an accelerator node of a high-end ARM based cluster. The Matrix Multiplication software used is a matrix–matrix multiply routine (GEMM, GEneral Matrix to Matrix Multiplication) achieving better performance if compared with other usual implementations. This routine uses a LU, QR, and Cholesky factorizations gaining up to 80–90.

4.1 The Development Workstation

The performance test system has been built on top of the Ubuntu 14.04 Linux operating system, the NVIDIA CUDA Driver, and the SDK/Toolkit version 6.5 hosted on a workstation equipped by an i7-940@2.93 GHz 12Gb RAM. The GPU subsystem is enforced by two NVIDIA GeForce Titan X 12Gb RAM powered by the Maxwell chipset and summing up 3072 CUDA cores.

4.2 The Cluster Based on High-End ARM Single Board Computer

In order to face with a real next generation high performance computing scenario, an experimental cluster made by 3 NVIDIA Jetson TK1 computing nodes has been set up, connected by a dedicated Gigabit Ethernet network to the developing workstation mimicking an accelerator server. Each computing node relies on 4-PLUS-1 Cortex A15 r3 CPU architecture, that delivers higher performance and is more power efficient than the previous generation, and a Kepler GPU architecture that utilizes 192 CUDA cores to deliver advanced graphics capabilities, GPU computing with NVIDIA CUDA 6.x support, breakthrough power efficiency and performance for the next generation of gaming and GPU-accelerated computing applications.

4.3 Smith–Waterman Sequence Alignment

The Smith–Waterman algorithm has been available for more than 25 years. It is based on a dynamic programming approach that explores all the possible alignments between two sequences; as a result it returns the optimal local alignment. Unfortunately, the computational cost is very high, requiring a number of operations proportional to the product of two-sequence length. Furthermore, the exponential growth of protein and DNA databases makes the Smith–Waterman algorithm unrealistic for searching similarities in large sets of sequences [21]. The alignment of two sequences is based on the computation of an alignment matrix. The number of its columns and rows is given by the number of the residues in the query and database sequences respectively. The computation is based on a substitution matrix and on a gap-penalty function. The CUDASW++ [22] has been used as evaluation software. It is a publically available open source software for Smith–Waterman protein database searches on Graphics Processing Units with CUDA. This software has been added to the NVIDIA Tesla Bio Workbench.

4.4 Rodinia Performance Study Application

The Rodinia software suite is widely accepted by the GPGPU scholars as a test of CUDA performances and capabilities. It uses the Berkeleys dwarf taxonomy to choose the applications developed using CUDA and OpenMP. Each dwarf represents a set of algorithms with similar computation and data movement. Even though programs representing a particular dwarf may have varying characteristics, they share strong underlying patterns. The dwarves are defined at a high level of abstraction to allow reasoning about the program behaviors [37]. The Rodinia software suite focuses on Structured Grid, Unstructured Grid, Combinational Logic, Dynamic Programming, fast Fourier transform (FFT), N-Body, Monte Carlo and Dense Linear Algebra dwarves. It targets on GPUs and multicore CPUs as a starting point in developing a broader treatment of heterogeneous computing. Rodinia benchmark suite enables users to evaluate heterogeneous systems including both accelerators, such as GPUs, and multicore CPUs. The parallel computing on GPGPU application chosen to test GVirtuS is speckle reducing anisotropic diffusion (SRAD) [39].

SRAD is a diffusion method for ultrasonic and radar imaging applications based on partial differential equations (PDEs). It is used to remove locally correlated noise, known as speckles, without destroying important image features. SRAD consists of several pieces of work: image extraction, continuous iterations over the image (preparation, reduction, statistics, computation 1 and computation 2) and image compression.

4.5 Matrix Multiplication

Our implementation of Matrix Multiplication takes advantage of shared memory already used to evaluate the performances ARM CUDA enabled software offloaded on remoted GPUs [24].

In this implementation each task (MPI process or thread) is responsible for computing number_of_rows/number_of_task rows of the matrix C (Algorithm 1). Every block of CUDA thread is responsible for the computing of 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 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 [3]. In order to easily verify the correct execution of the code the software performs:

$$RAND(n \times n) * EYE(n \times n) = RAND(n \times n) \tag{1}$$

The choice of this strategy comes from the easy scalability and evaluation, and because it does not need synchronization mechanism to avoid race condition, this comes from the spawn of the data amongst the tasks. We propose two implementations of this test: one for MPI process and one for POSIX thread. The main process (Rank 0 in MPI) takes care of distributing the data amongst the workers and collecting them after the computing process is ended.

Writing a basic dense Matrix–Matrix Multiplication kernel is a fairly simple exercise (see the CUDA Programming Guide for details). Achieving this high level of performance, on the other hand, requires more careful optimization. Volkov and Demmel [40] used a block algorithm similar to those used for vector computers, using GPU registers and per-block shared memory to store the data blocks. As the GPU has an unusually large register file, registers can be used as the primary scratch space for the computation. Furthermore, assigning small blocks of elements to each thread, rather than a single element to each thread, boosts efficiency much as strip-mining boosts efficiency on vector machines. Finally, the non-blocking nature of loads on the GPU makes it possible to do software prefetching, which is useful for hiding memory latency [11] (Fig. 6).

Algorithm 1 MatrixMul MPI/CUDA

```
1: procedure MAINTASK
2:     for i ← 1,num_of_tasks do
3:         alocal ←a[offset * num_rows_a]
4:         clocal ←c[offset * num_rows_b]
5:         SEND_TO_WORKER(alocal)
6:         SEND_TO_WORKER(b)
7:         SEND_TO_WORKER(clocal)
8:     end for
9:     for i ← 1,num_of_tasks do
10:        COLLECT_FROM_WORKER(i)
11:    end for
12: end procedure
13: procedure WORKERTASK
14:    for a ← 1,num_of_el_a do
15:        for k ← 1,num_of_block do
16:            Csub ← CALCULATE_C_SUBMATRIX(k)
17:        end for
18:        C ← COLLOCATE_CSUB(Csub)
19:    end for
20: end procedure
```

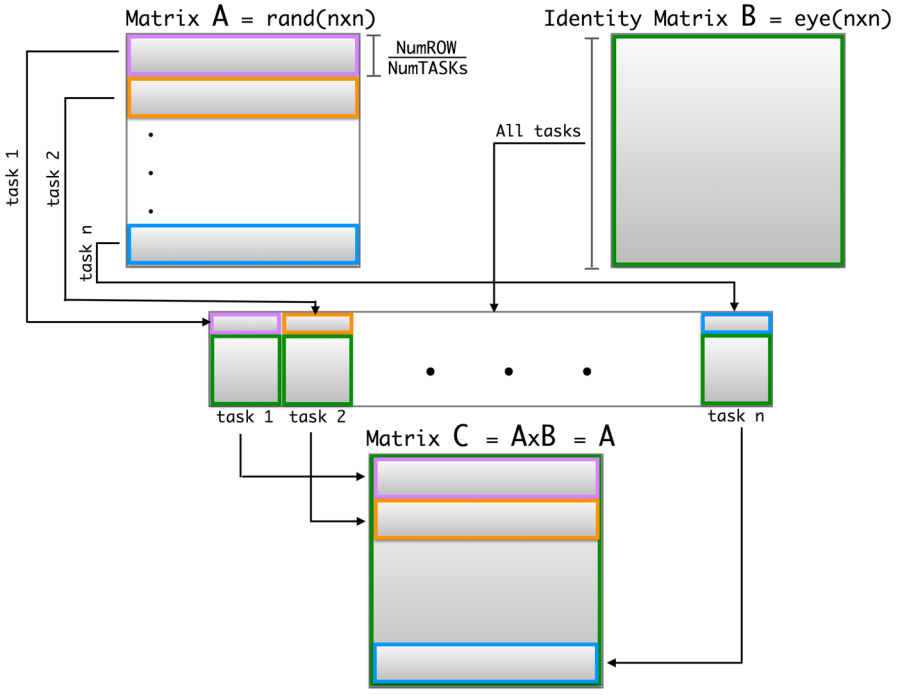

Fig. 6 Implementation of Matrix Multiplication multitask

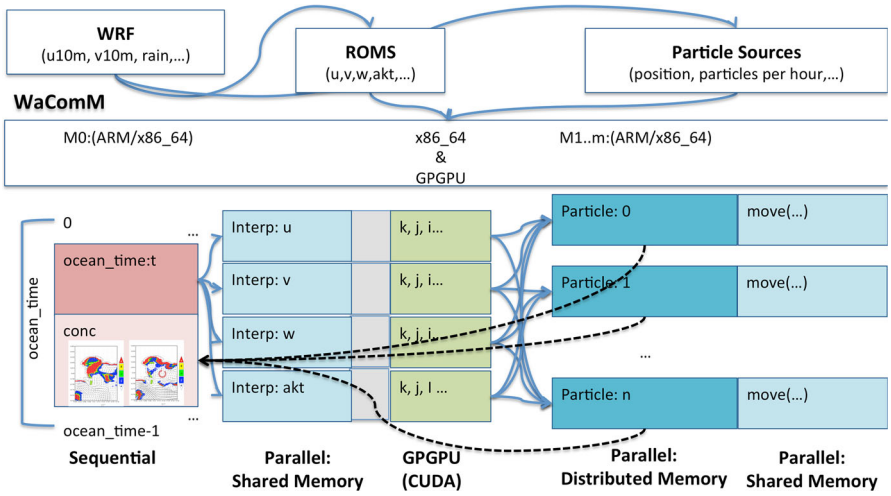

Fig. 7 The WaComM hybrid parallel implementation

4.6 Experiment Design for a Real World Problem

Water quality Community Model (WaComM) is a coastal area decisionmaking tool for mussel farms food quality assessment and prediction. It is based on eulerian/lagrangian methods. WaComM is numerically coupled with marine dynamics models [10] in an offline fashion. WaComM has been developed and tested in X86_64 multicore environments. Due to the intensive demanding computations, the porting to a hierarchically parallel architecture has been designed and partially already implemented. In this scenario we refactored the model code in order to implement distributed memory/shared memory/GPGPU hierarchical parallelisation (Fig. 7).

The use of GVirtuS in order to take advantage of a massive ARM based HPC system with few high-end CUDA equipped accelerator nodes could be the killing application targeting the reduction of total cost of ownership (procurement, powering, cooling) in an application field, the continuous operational real-time environmental modelling, where the on-premises and on-cloud solutions are economically borderline options. A forthcoming paper will discuss about the WaComM architecture, its implementation and the performance assessment in a GVirtuS based, mixed ARM/X86_64/GPGPU environment.

5 Results

From the wall clock time point of view, the execution of an application interacting with a remoted GPU will sustain lower performance than the same in the full availability of a dedicated, local, not virtualized accelerator device. This is due to the need of the split-driver interleaved layers and (in the case of GPU remoting) the network infrastructure. In the best case the virtualization/remoting overhead is partially balanced by the improvements in computation performance. This happens for some

Table 1 SRAD parameters

R	C	y1	y2	x1	x2	L	I
2048	2048	0	31	0	31	0.5	10
2048	2048	0	31	0	31	0.5	100
4096	4096	0	31	0	31	0.5	10
4096	4096	0	31	0	31	0.5	100

application classes characterized by the need for high GPU calculations feeded by a relative poor amount of input and output parameters. But, if we change the point of view from a strict performance oriented to a total costs of ownership perspective, the GPU remoting permits to build a cluster with a reduced number of GPUs. The use of GVirtuS middleware could be definitely effective if the applications are designed explicitly to take advantage from an hybrid architecture computation environment with a consistent costs reduction. The proposed evaluation tests have the target to demonstrate the effectiveness of the designed infrastructure rather than the mere performance that, as previously stated, is affected by many ineluctable components that could be mitigated with future technology improvements. It is possible to evaluate the overhead introduced by GVirtuS faced with the chance to run CUDA code or no CUDA enabled devices. Mainly, the bottleneck is the communication overhead due to the use of the TCP communicator. This results in poor performance, especially stressed out when the GPU remoting is done outside the local dedicated network where the overhead is acceptable.

5.1 GPGPU Virtualization and Remoting

In this section the results of CUDASW++ (test 1) and SRAD (test 2) to test performance for x86_64/x86_64/GPU in three different approaches have been showed.

Three test scenarios are presented:

- *No virtualization* the CUDA code is executed using regular CUDA libraries. This is a measurement of the blank.
- *Localhost remoting* the CUDA code is executed using a remoted CUDA device hosted on the same machine. This test verifies the effectiveness of GVirtuS libraries.
- *Virtualization* the CUDA code is executed on a virtual machine hosted on the same physical host where the CUDA devices are connected to the PCIe bus.

The Test 1 leverages on CUDASW++ version 2.0.11 executed with parameters -query P01008.fasta -db uniprot_sprot.fasta -use_single 0. The database used for the test is uniprot_sprot.fasta. This is the last release of the Swiss-Prot database released by UniProt, a scientific community with a comprehensive, high-quality and freely accessible resource of protein sequence and functional information. The database contains 550,552 sequence entries [2]. The query used for the test is P01008.fasta. This is an example query sequence suggested by the CUDASW++ documentation.

Table 2 SRAD performances

Size	Iterations	No virtualization (s)	Localhost remoting (s)	Virtualization (s)
2048 × 2048	10	0.451	0.626	2.969
2048 × 2048	100	1.119	3.458	27.872
4096 × 4096	10	0.948	2.005	2.997
4096 × 4096	100	3.781	12.283	27.946

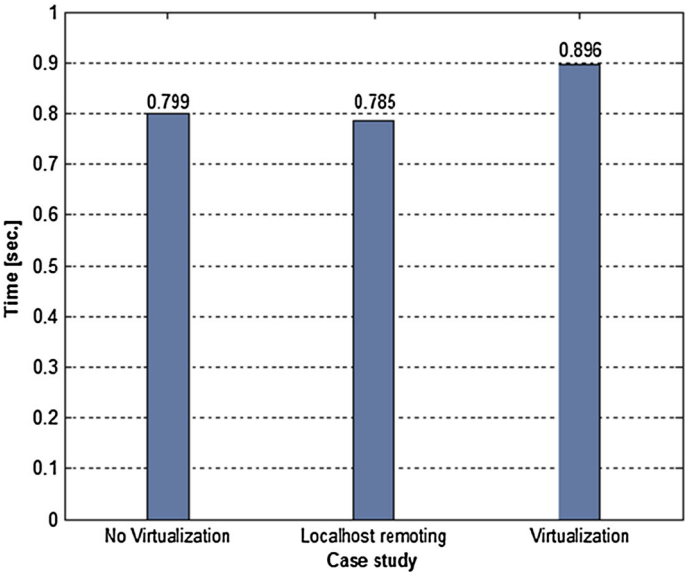

Fig. 8 Execution time of SWCUDA++ in the three different approach

The test 2 leverages on benchmark provided by Rodinia SRAD casted with the parameters shown in Table 1: the first parameter, R, is the number of rows in the domain; the second parameter, C, is the number of columns in the domain. Currently, the GPU implementation of SRAD only supports a dimension of kernel that can be divided by 16. The kernel has square shape. The parameters from third to sixth represent respectively the y1, y2, x1, x2 positions of the speckle. The seventh parameter is the lambda value (L). The last parameter is I, the number of iterations.

The tests ensure the effectiveness of the GVirtuS framework because the results of the execution through CUDA and through GVirtuS coincide. As of the performances in the execution of CUDASW++, no significative overhead is introduced by the use of GVirtuS in the Localhost remoting scenario, while in the virtualization scenario the overhead introduced has to be correlated to the virtual environment. The execution of SRAD is impacted by the involvement of GVirtuS as shown in Table 2. The reason of this behaviour has to be found in the data intensive nature of this test so the bottleneck

Table 3 Matrix Multiplication performances using internal and remoted GPU

Size	NP1 (s)	NP2 (s)	NP3 (s)	NP1 GVirtuS (s)	NP2 GVirtuS (s)	NP3 GVirtuS (s)
800×800	2.812	1.948	1.895	1.238	1.383	2.115
1600×1600	9.813	9.004	5.846	2.201	2.411	2.795
3200×3200	46.754	39.061	30.388	5.341	5.280	6.571

is represented by the TCP/IP communicator. Furthermore the TCP/IP communicator is not intended for performance purpose (Fig. 8).

5.2 High-End ARM GPU Cluster

The results of MPI Matrix Multiplication program showing the performance test results for a ARM/x86/GPU setup are presented in this section. In this experiment the MPI Matrix Multiplication program has been used, in order to investigate about the behavior of GVirtuS in a scenario where a x86 machine is used as an accelerator node of a high-end ARM based cluster. In our setup each computing node is provided by an on-board K20A NVIDIA CUDA enabled GPU with 192 cores, while the accelerator node is powered by a couple of NVIDIA Titan X. This benchmark has been performed with two problem size: $1600 \times 1600 \times 800$ and $3200 \times 3200 \times 1600$. The experiment compares the performance of the on-board GPU and GVirtuS remoted on both problems size (Table 3). The ARM based cluster is built on 3 nodes each provided by 4 CPU cores. The MPI Matrix Multiplication program uses MPI, but it is not OpenMP enabled, so runs were performed using up to 3 MPI computing processes.

Results demonstrate the use of GVirtuS remoted CUDA acceleration is convenient especially when the problem size increases: the weight of the latency due to the communication decreases, as expected. The overall performances are improved by the MPI parallel approach when the CUDA is used locally, but the limited amount of node memory and number of nodes prevented to investigate more in this direction.

When the number of MPI processes increases over 2, benchmarks are no more suitable for classic parallel programming efficiency and speedup analysis, but could be useful for some speculations about GVirtuS and its use in GPU remoting. When GVirtuS will fully support the multithreading, the use of the Matrix Multiplication enabled for both distributed and shared memory could provide a better performance test for this kind of applications (Fig. 9).

6 Related Works

GPU virtualization solutions to GPGPU as GVirtuS have been implemented in research projects as Remote CUDA (rCUDA) [34] and Distributed-Shared CUDA (DS-CUDA) [17]. They all use an approach similar to GVirtuS, providing CUDA API wrappers on

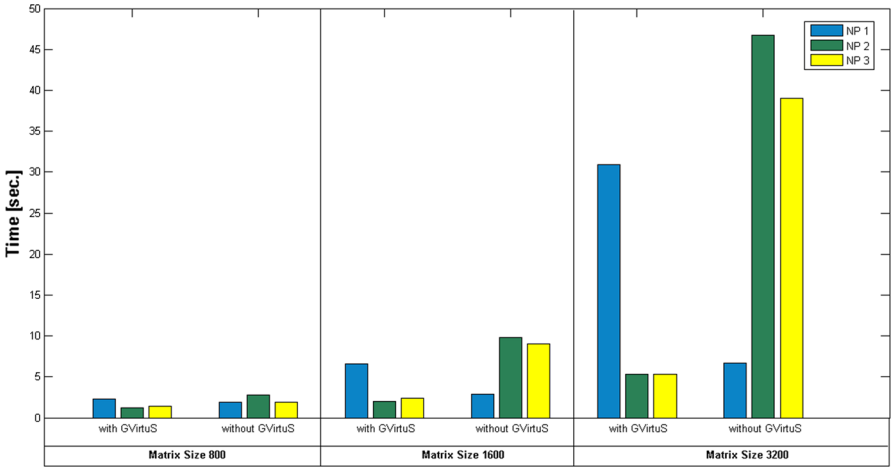

Fig. 9 Implementation of Matrix Multiplication multitask

the front-end application in the guest OS while the back-end in the host OS accesses to the CUDA devices.

Table 4 shows the main differences on the CUDA toolkit supported, the implementation of various communicator components to connect the front-end and back-end, the re-compiling needed, the concurrent remote usage of CUDA devices in a transparent way, the support for x86 and ARM processors and, finally, the type of license.

- *CUDA Toolkit supported* all GPGPU computing solutions mentioned implement the functions in the CUDA Runtime API, but the graphic relevant APIs, such as OpenGL and Direct3D interoperability, are not supported. A common restriction for GVirtuS and DS-CUDA is the asynchronous APIs implemented as aliases to their synchronous counterparts.
- *Communicator* a communicator is a key piece because it connects the guest and host operating systems. One of the main differences lies in the use of the communication technique. In GVirtuS communicators are independent from hypervisor, virtualized technology and from the cooperation protocols between front-end and back-end. 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). By default, rCUDA and DS-CUDA use InfiniBand Verbs, and TCP sockets in case the network infrastructure does not support InfiniBand in the guest and host communication.
- *Plug-in architecture* while GVirtuS is a general-purpose virtualization service with a plug-in architecture, which can load modules of CUDA and OpenCL and use different GPU devices, rCUDA and DS-CUDA allow to manage only NVIDIA GPUs. The main aim of GVirtuS is to provide a flexible tool capable to adapt itself to any possible scenario, GVirtuS competitors aim is just to provide NVIDIA support.

Table 4 CUDA virtualization features comparison table

GPU Vir.	RT	DRV	Comm	Rebuild	Plug-in	License
GVirtuS	6.5		TCP/IP, SHMem, ...		Yes	LGPL
rCUDA	5.5	Yes	IB, TCP/IP			Proprietary
DS-CUDA	4.5		IB, TCP/IP	Needed		GPL

- *Computing architecture* in the last years, the use of remote GPUs and low-power processors for acceleration of scientific applications has become an important case study. GVirtuS is a tool to virtualize heterogeneous architectures. It is based on a split-driver model independent of the computing architecture ARM and x86_64. DS-CUDA is going to use Android tablets and smartphones to run the executable CUDA file [24]. rCUDA carried experimental study on scientific applications with different hardware platforms [5].
- *Transparency and re-compiling* the main goal of GVirtuS is to provide a fully transparent virtualization solution, that is CUDA enabled software has to be executed without any further modification of binaries and the source code of applications does not need to be modified in order to use remote GPUs. Transparency is an important common feature of the presented virtualization GPUs systems. DS-CUDA needed of a re-compiling in order to build an executable for the application program, this latter has a DS-CUDA preprocessor dscudacpp to handle CUDA C/C++ extensions.
- *Licence* GVirtuS and DS-CUDA are open source projects, the former is licensed under the LGPL (Lesser General Public License), while the latter is licensed under the GPLv3 (General Public License version 3). The rCUDA technology is own by the Parallel Architectures Group from Universitat Politecnica de Valencia (Spain). The Software is distributed for free under specified terms and conditions of use.

7 Conclusions and Future Directions

In this paper were presented our results about the design and the implementation of an updated CUDA wrapper library for the GVirtuS framework in order to accelerate subclusters of inexpensive low power demanding ARM based boards. We used high-end GPGPU devices providing an experimental evaluation of the possibilities that state-of-the-art technology offers in nowadays HPC facilities [32], as well as low-power alternatives offer for the acceleration of scientific applications using remote graphics processors.

The performed experiments demonstrate how convenient is the path we followed as trailblazer in the hunt for the next big thing in the off-the-shelf commodity high performance computing clusters. The latest GVirtuS release tested in a x86_64 virtualization and remoting performs enough to consider feasible the use of our approach in real world production applications, especially if enhanced with an Infiniband communicator component. This because with the availability of the needed hardware testbed,

the communication plug-in component will evolve in order to support the Infiniband network because it is expected that the higher bandwidth allows remote GPU virtualization frameworks to experience communication performances similar to the PCIe on the path between the local GPGPU and the remote GPU resource [31,33]. Due to the unavailability of real world applications fitting the available ARM cluster, GVirtuS has been tested using an ad hoc distributed memory Matrix Multiplication software [14] and accelerated CUDA kernels working on local or x86 remoted high-end GPU device [18].

On short and medium term, we are working on the GVirtuS over all improvement in order to implement a production service for GPGPU computation offloading dedicated to high end server machines and mobile devices. A custom Java/Android friendly front-end implementation will enable to GPGPU computing the most part of low-power integrated systems and devices. The final destination of this research is provisioning a full production software environment for advanced earth system simulations and analysis based on science gateways, workflow engines and high performance cloud computing [28], giving a support for the next generation of scientific dissemination tools [30] and the smart city management in case of extreme weather events.

Acknowledgements This research has been supported mainly 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", in part by the project IZS ME04/12 RC/C78C120017001 "Mapping Escherichia Coli and Salmonella pollution in mussel farm areas and model prediction comparisons", in part by the University of Naples Parthenope - Department of Science and Technologies "Weather/marine extreme event simulation with Galaxy-ES (Earth System) scientific workflow engine and cloud computing tools" Research Project, and in part by the University of Naples Federico II - Department of Mathematics "Approcci Innovativi per la Risoluzione di Modelli di Interesse nelle Simulazioni Computazionali" Research Project Grant Agreement.

References

1. Armand, F., Gien, M., Maign, 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. Bairoch, A.M., Apweiler, R., Wu, C.H., Barker, W.C., Boeckmann, B., Ferro Rojas, S., Gasteiger, E., et al.: The universal protein resource (UniProt). Nucleic Acids Res. **33**(Database issue), D154–D159 (2005)
3. Bell, N., Garland, M.: Efficient sparse matrix-vector multiplication on CUDA. NVIDIA Technical Report NVR-2008-004, Nvidia Corporation (2008)
4. Caruso P.G. Laccetti, Lapegna, M.: A performance contract system in a grid enabling, component based programming environment. In: Advances in Grid Computing-EGC 2005, LNCS, vol. 3470, pp. 982–992. Springer (2005)
5. Castello, A., Duato, J., Mayo, R., Pena, A.J., Quintana-Ort, E.S., 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)
6. Dagum, L., Enon, R.: OpenMP: an industry standard API for shared-memory programming. IEEE Comput. Sci. Eng. **5**(1), 46–55 (1998)
7. 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: IEEE 10th International Symposium on Proceedings of Parallel and Distributed Processing with Applications (ISPA), pp. 863–864 (2012)

8. Di Lauro, R., Lucarelli, F., Montella, R.: SIaaS-sensing instrument as a service using cloud computing to turn physical instrument into ubiquitous service. In: 2012 IEEE 10th International Symposium on Parallel and Distributed Processing with Applications. IEEE, pp. 861–862 (2012)
9. Foster, I., Zhao, Y., Raicu, I., Lu, S.: Cloud computing and grid computing 360-degree compared. In: IEEE Grid Computing Environments Workshop GCE 08, pp. 1–10 (2008)
10. Giunta, G., Mariani, P., Montella, R., Riccio, A.: pPOM: a nested, scalable, parallel and Fortran 90 implementation of the Princeton Ocean Model. Environ. Model. Softw. **22**(1), 117–122 (2007)
11. Garland, M., Le Grand, S., Nickolls, J., Anderson, J., Hardwick, J., Morton, S., Phillips, E., Zhang, Y., Volkov, V.: Parallel computing experiences with CUDA. IEEE Micro **28**(4), 13–27 (2008)
12. Giunta, G., Montella, R., Agrillo, G., Coviello, G.: A GPGPU transparent virtualization component for high performance computing clouds. In: EuroPar 2010 Parallel Processing, LNCS, vol. 6271, no. 2, pp. 379–391. Springer (2010)
13. Giunta, G., Montella, R., Laccetti, G., Isaila, F., Blas, F.: A GPU accelerated high performance cloud computing infrastructure for grid computing based virtual environmental laboratory. Adv. Grid Comput. 35–43 (2011)
14. Gropp, W.: MPICH2: a new start for MPI implementations. In: Recent Advances in Parallel Virtual Machine and Message Passing Interface 2002, LNCS, vol. 2474, p. 7. Springer (2002)
15. Gupta, V., Gavrilovska, A., Schwan, K., Kharche, H., Tolia, N., Talwar, V., Ranganathan, P.: GViM: GPU-accelerated virtual machines. In: Proceedings of the 3rd ACM Workshop on System-Level Virtualization for High Performance Computing, pp. 17–24. ACM (2009)
16. Herrera, A.: NVIDIA GRID: Graphics Accelerated VDI with the Visual Performance of a Workstation. Nvidia Corp, Santa Clara (2014)
17. Kawai, A., Yasuoka, K., Yoshikawa, K., Narumi, T.: Distributed-shared CUDA: virtualization of large-scale GPU systems for programmability and reliability (2012)
18. Karunadasa, N.P., Ranasinghe, D.N.: Accelerating high performance applications with CUDA and MPI. In: 2009 International Conference on Industrial and Information Systems (ICIIS), pp. 331–336. IEEE (2009)
19. Kehne, J., Metter, J., Bellosa, F.: GPUswap: enabling oversubscription of GPU memory through transparent swapping. In: Proceedings of the 11th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, pp. 65–77. ACM (2015)
20. Laccetti, G., Montella, R., Palmieri, C., Pelliccia, V.: 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 2013, LNCS, vol. 8384, pp. 734–744. Springer, Berlin, Heidelberg (2013)
21. Ligowski, L., Rudnicki, W.: An efficient implementation of Smith–Waterman algorithm on GPU using CUDA, for massively parallel scanning of sequence databases. In: IEEE International Symposium on Parallel and Distributed Processing 2009, IPDPS 2009, pp. 1–8. IEEE (2009)
22. Liu, Y., Schmidt, B., Maskell, D.L.: CUDASW++ 2.0: enhanced Smith–Waterman protein database search on CUDA-enabled GPUs based on SIMT and virtualized SIMD abstractions. BMC Res. Notes **3**(1), 93 (2010)
23. Manavski, S.A., Valle, G.: CUDA compatible GPU cards as efficient hardware accelerators for Smith–Waterman sequence alignment. BMC Bioinf. **9**(2), 1 (2008)
24. Martinez-Noriega, E.J., Josafat, E., Kawai, A., Yoshikawa, K., Yasuoka, K., Narumi, T.: CUDA Enabled for Android Tablets through DS-CUDA (2013)
25. Montella, R., Foster, I.: Using hybrid grid/cloud computing technologies for environmental data elastic storage, processing, and provisioning. In: Handbook of Cloud Computing, pp. 595–618. Springer, USA (2010)
26. 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. In: International Conference on Parallel Processing and Applied Mathematics, pp. 740–749. Springer, Berlin, Heidelberg (2011)
27. Montella, R., Giunta, G., Laccetti, G.: Virtualizing high-end GPGPUs on ARM clusters for the next generation of high performance cloud computing. Cluster Comput. **17**(1), 139–152 (2014)
28. Montella, R., Kelly, D., Xiong, W., Brizius, A., Elliott, J., Madduri, R., Maheshwari, K., et al.: FACE IT: A science gateway for food security research. Concurr. Comput. Pract. Exp. **27**(16), 4423–4436 (2015)
29. Montella, R., Giunta, G., Laccetti, G., Lapegna, M., Palmieri, C., Ferraro, C., Pelliccia, V.: Virtualizing CUDA enabled GPGPUs on ARM clusters. In: Parallel Processing in and Applied Mathematics 2015, LNCS, vol. 9574, Springer, Berlin, Heidelberg (2016)

30. Pham, Q., Malik, T., Foster, I., Di Lauro, R., Montella, R., SOLE: linking research papers with science objects. In: Provenance and Annotation of Data and Processes 2012, LNCS, vol. 7525, pp. 203–208. Springer, Berlin, Heidelberg (2012)
31. Prades, J., Reao, C., Silla, F.: CUDA acceleration for Xen virtual machines in infiniband clusters with rCUDA. In: Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, p. 35. ACM (2016)
32. Rajovic, N., Rico, A., Puzovic, N., Adeniyi-Jones, C., Ramirez, A.: Tibidabo: making the case for an ARM-based HPC system. Fut. Gener. Comput. Syst. **36**, 322–334 (2014)
33. Reao, C., Mayo, R., Quintana-Orti, E.S., Silla, F., Duato, J., Pea, A.J.: Influence of InfiniBand FDR on the performance of remote GPU virtualization. In: Proceedings of the 2013 IEEE International Conference on Cluster Computing, Indianapolis, USA (2013)
34. Reao, C., Silla, F., Pena, A.J., Shainer, G., Schultz, S., Castello, A., Quintana-Orti, E.S., Duato, J.: POSTER: Boosting the performance of remote GPU virtualization using InfiniBand connect-IB and PCIe 3.0. In: 2014 IEEE International Conference on Cluster Computing (CLUSTER), pp. 266–267. IEEE (2014)
35. Shi, L., Chen, H., Sun, J., Li, K.: vCUDA: GPU-accelerated high-performance computing in virtual machines. IEEE Trans. Comput. **61**(6), 804–816 (2012)
36. Shuai, C., Boyer, M., Meng, J., Tarjan, D., Sheaffer, J.W., Skadron, K.: A performance study of general-purpose applications on graphics processors using CUDA. J. Parallel Distrib. Comput. **68**(10), 1370–1380 (2008)
37. Shuai, C., Boyer, M., Meng, J., Tarjan, D., Sheaffer, J.W., Lee, S.-H., Skadron, K.: Rodinia: a benchmark suite for heterogeneous computing. In: Proceedings of the IEEE International Symposium on Workload Characterization—IISWC 2009, pp. 44–54 (2009)
38. Sourouri, M., Gillberg, T., Baden, S.B., Cai, X.: Effective multi-GPU communication using multiple CUDA streams and threads. In: 2014 20th IEEE International Conference on Parallel and Distributed Systems (ICPADS), pp. 981–986. IEEE (2014)
39. Szafaryn, L.G., Skadron, K., Saucerman, J.J.: Experiences accelerating MATLAB systems biology applications. In: Proceedings of the workshop on biomedicine in computing: systems, architectures, and circuits (BiC) 2009. In: Conjunction with the 36th IEEE/ACM International Symposium on Computer Architecture (ISCA) (2009)
40. Volkov, V., Demmel, J.W.: Benchmarking GPUs to tune dense linear algebra. In: International Conference for High Performance Computing, Networking, Storage and Analysis 2008, SC 2008, pp. 1–11. IEEE (2008)
41. Yang, C., Huang, C., Lin, C.: Hybrid CUDA, OpenMP, and MPI parallel programming on multicore GPU clusters. Comput. Phys. Commun. **182**(1), 266–269 (2011)