Remote GPU Virtualization: Is It Useful?

Federico Silla*, Javier Prades*, Sergio Iserte†, and Carlos Reaño*
*Universitat Politècnica de València, Spain
Email: fsilla@disca.upv.es, japraga@gap.upv.es, carregon@gap.upv.es
†Universitat Jaume I, Spain
Email: siserte@uji.es

***Abstract*—Graphics Processing Units (GPUs) are currently used in many computing facilities. However, GPUs present several side effects, such as increased acquisition costs as well as larger space requirements. Also, GPUs still require some amount of energy while idle and their utilization is usually low.**

In a similar way to virtual machines, using virtual GPUs may address the mentioned concerns. In this regard, remote GPU virtualization allows to share the GPUs present in the computing facility among the nodes of the cluster. This would increase overall GPU utilization, thus reducing the negative impact of the increased costs mentioned before. Reducing the amount of GPUs installed in the cluster could also be possible.

In this paper we explore some of the benefits that remote GPU virtualization brings to clusters. For instance, this mechanism allows an application to use all the GPUs present in a cluster. Another benefit of this technique is that cluster throughput, measured as jobs completed per time unit, is doubled when this technique is used. Furthermore, in addition to increasing overall GPU utilization, total energy consumption is reduced up to 40%. This may be key in the context of exascale computing facilities, which present an important energy constraint.

***Keywords*-GPU, CUDA, GPU virtualization, rCUDA, SLURM, virtual machine, cloud computing, InfiniBand**

I. Introduction

Currently, the massive parallel capabilities of GPUs (Graphics Processing Units) are leveraged to accelerate specific parts of applications. In this regard, programmers exploit GPU resources by off-loading the computationally intensive parts of applications to them. To that end, although programmers must specify which parts of the application are executed on the CPU and which parts are off-loaded to the GPU, the existence of libraries and programming models such as CUDA (Compute Unified Device Architecture) [11] noticeably ease this task. In this context, GPUs significantly reduce the execution time of applications from domains as different as Big Data [28], chemical physics [21], computational algebra [29], image analysis [17], finance [26], and biology [1] for instance.

Current computing facilities typically include one or more GPUs at every node of the cluster. However, using GPUs in such a configuration is not exempt from side effects. For example, let us consider the execution of a distributed MPI (Message Passing Interface) application which does not require the use of GPUs. Typically, this application will spread across several nodes of the cluster flooding the CPU cores available in them. In this scenario, the GPUs in the nodes involved in the execution of such an MPI application would become unavailable for other applications because all the CPU cores in those nodes would be devoted to the non-accelerated MPI application. This would force those GPUs to remain idle for some periods of time.

Another example of the concerns associated with the use of GPUs in clusters is related to the way that job schedulers such as Slurm [31] perform the accounting of resources in a cluster. These job schedulers use a fine granularity for resources such as CPUs or memory, but not for GPUs. For instance, job schedulers can assign CPU resources in a per-core basis, thus being able to share the CPU sockets present in a server among several applications. In the case of memory, job schedulers can also assign, in a shared approach, the memory present in a given node to the several applications that will be concurrently executed in that server. However, in the case of GPUs, job schedulers use a per-GPU granularity. In this regard, GPUs are assigned to applications in an exclusive way. Hence, a GPU cannot be shared among several applications even when it has enough resources to allow the concurrent execution of those applications, causing that overall GPU utilization is, in general, low. This fact not only reduces the effective computing power of computing facilities but also causes that a non-negligible amount of energy is wasted, being both aspects key concerns in the context of exascale computing.

In order to address the side effects related to the use of GPUs, the remote GPU virtualization mechanism could be used. This software mechanism allows an application being executed in a computer which does not own a GPU to transparently make use of accelerators installed in other nodes of the cluster. In other words, the remote GPU virtualization technique allows physical GPUs to be logically detached from nodes, thus allowing that decoupled (or virtual) GPUs are concurrently shared by all the nodes of the computing facility in a transparent way to applications. This not only increases overall GPU utilization but also allows to create cluster configurations where not all the nodes in the cluster own a GPU, thus reducing the costs associated with the acquisition and later use of GPUs. In this regard, the total energy required to operate a computing facility would be decreased, thus loosening the big energy concerns of future exascale computing facilities.

In this paper we explore some of the benefits that the remote GPU virtualization mechanism provides to clusters. To that end, Section II presents a review of this virtualization technique and introduces the rCUDA technology, which will be used in this work to quantify the benefits of the remote GPU virtualization mechanism. Later, Section III introduces four of its benefits. Finally, Section IV concludes the paper.

II. Remote GPU Virtualization

Frameworks such as CUDA [11] assist programmers in using GPUs for general-purpose computing. In addition, several remote GPU virtualization solutions exist for this framework, such as GridCuda [15], DS-CUDA [18], gVirtuS [4], vCUDA [25], GViM [5], and rCUDA [19].

Figure 1 depicts the architecture underlying most of these virtualization solutions, which follow a client-server distributed approach. The client part of the middleware is installed in the cluster node executing the application requesting GPU services, whereas the server side runs in the computer owning the actual GPU. Generally, the client middleware offers the same application programming interface (API) as does the NVIDIA CUDA Runtime API [12]. In this way, the client receives a CUDA request from the accelerated application and appropriately processes and forwards it to the remote server. In the server node, the middleware receives the request and interprets and forwards it to the GPU, which completes the execution of the request and provides the execution results to the server middleware. In turn, the server sends back the results to the client middleware, which forwards them to the initial application, which is not aware that its request has been served by a remote GPU instead of a local one.

Current virtualization frameworks provide different features. For example, DS-CUDA supports CUDA 4.1 and includes specific communication support for InfiniBand, although it presents several severe limitations like not allowing data transfers with pinned memory. Regarding the vCUDA technology, it supports the old CUDA 3.2 version and implements an unspecified subset of the CUDA runtime API. Moreover, its communication protocol presents a considerable overhead because of the costs of the encoding and decoding stages, which cause a noticeable drop in overall performance. On the other hand, GViM is based on the old CUDA version 1.1 and, in principle, does not implement the entire runtime API. Similarly, the gVirtuS approach is based on the old CUDA 2.3 version and implements only a small portion of the runtime API. Furthermore, it only provides TCP/IP communications between clients and servers, thus reducing the effective bandwidth in networks such as InfiniBand. GridCuda, supports CUDA 3.2 and has no public version that may be used for testing and comparison. In the case of rCUDA, it is binary compatible with CUDA 7.0 and implements the entire CUDA Runtime and Driver APIs (except for graphics functions). It provides support for the libraries included within CUDA (cuBLAS, cuFFT, etc). Additionally, it supports several underlying interconnection technologies by making use of network-specific communication modules. Currently two communication modules are available: TCP/IP and InfiniBand. The former can be used in any TCP/IP compatible network whereas the latter makes use of the high performance InfiniBand Verbs API available in the InfiniBand network adapters. Furthermore, as shown in [23], rCUDA outperforms the rest of available remote GPU virtualization solutions. For these reasons, we use this middleware in our study.

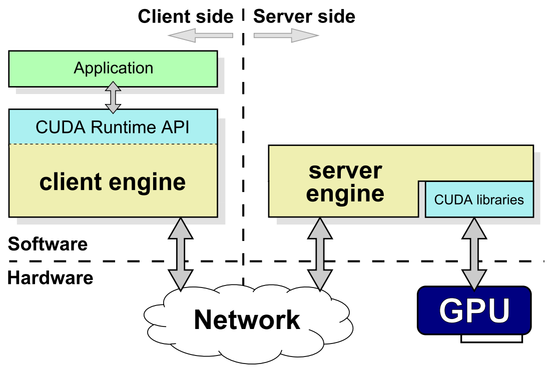


Fig. 1: Organization of remote GPU virtualization frameworks.

III. Benefits of Using Remote GPU Virtualization

In this section we introduce four of the benefits that the remote GPU virtualization mechanism presents. Namely, these benefits, which will be further described and analyzed in the next subsections, are the following ones:

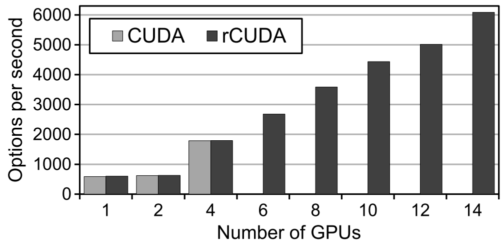
1) More GPUs are available for a single application.
2) Cluster throughput is increased at the same time that energy consumption is reduced. Overall GPU utilization is also increased.
3) Cluster upgrades are made easier and cheaper just by attaching GPU servers to a non-GPU cluster.
4) Several virtual machines can concurrently access the same GPU in a shared manner.

The next subsections further describe and analyze these benefits by including a performance evaluation for each of them. To that end, the testbed leveraged is based on the use of 1027GR-TRF Supermicro servers, each of them including two Intel Xeon E5-2620 v2 processors (six cores with Ivy Bridge architecture) operating at 2.1 GHz and 32 GB of DDR3 memory at 1600 MHz. They also have a Mellanox ConnectX-3 VPI single-port FDR InfiniBand adapter connected to a Mellanox Switch SX6025 (InfiniBand FDR compatible) to exchange data at a maximum rate of 56 Gb/s. Furthermore, an NVIDIA Tesla K20 GPU is installed at each node.

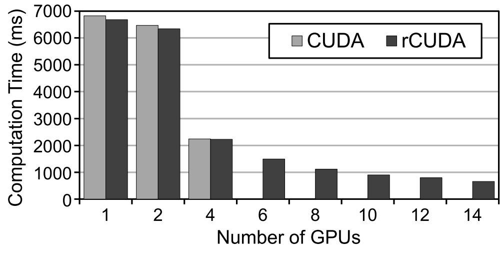
Regarding the software configuration of the cluster, Linux CentOS 6.4 was used along with CUDA 7.0 and Mellanox OFED 2.4-1.0.4 (InfiniBand drivers and administrative tools). For those experiments involving a job scheduler, Slurm version 14.11.0 was used. It was configured to use the `backfill` scheduling policy. In this way, jobs can overtake others. Finally, for those applications requiring the MPI library, version 2.0b of the MVAPICH2 implementation of MPI, specifically tuned for InfiniBand, was used.

Benefit 1: More GPUs for a Single Application

When using CUDA, an MPI application can be distributed across several nodes in the cluster in order to make use of the GPUs installed in those nodes. However, a shared-memory application based on the use of threads can only run in a single node and therefore it can only benefit from the GPUs installed in that node. On the contrary, when rCUDA is leveraged, an application being executed in a single node can use all the



(a) Total options per second computed.



(b) Execution time.

Fig. 2: Performance of the MontecarloMultiGPU Sample by NVIDIA with a varying number of GPUs when using CUDA and rCUDA.

GPUs in the cluster, thus boosting its performance. In this case, the only limitation is the ability of the programmer to code the application in the proper way so that it takes advantage of as many GPUs as they are available.

Figure 2 shows the performance of the MontecarloMultiGPU Sample by NVIDIA when executed in a single node owning 4 GPUs with CUDA and also when executed in a cluster making use of up to 14 GPUs with rCUDA. The CUDA executions have been performed in a node based on the Supermicro SYS7047GR-TRF server, populated with four NVIDIA Tesla K20 GPUs. Given that CUDA can only use the GPUs installed in the same node that is executing the application, only up to 4 GPUs can be used for the CUDA executions. On the contrary, when rCUDA is used, many additional GPUs can be provided to the application. Figure 2 shows how the use of a large amount of GPUs contributes to reduce total execution time. Notice also that for 1 and 2 GPUs, execution time with rCUDA is slightly lower than with CUDA. This is mainly due to the higher bandwidth attained by rCUDA for moving data to/from the GPU, as shown in [24].

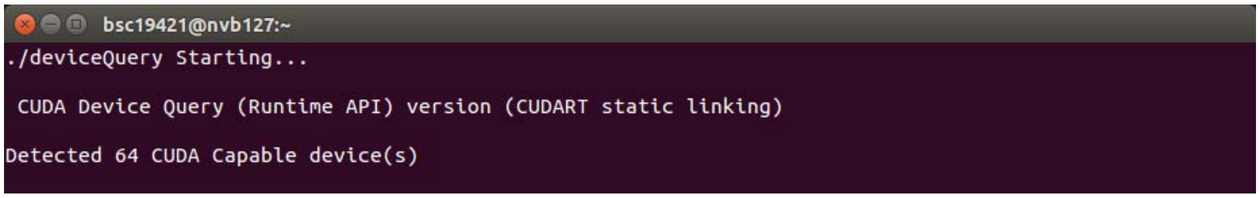
On the other hand, Figure 3 depicts part of the output provided by the execution of the deviceQuery sample by NVIDIA. In this case, all the 64 GPUs installed in one of the clusters owned by the Barcelona Supercomputing Center were provided to the application.

Benefit 2: Increased Cluster Throughput

When the remote GPU virtualization mechanism is used in a cluster, GPUs can be concurrently shared among several applications as far as there are enough memory resources available in the GPUs for the applications being executed. Additionally, given that a GPU can be used by applications being executed in a node other than the one where the GPU is installed, when all the CPU cores in the node owning the GPU are busy with a non-accelerated application, the GPU can still be used from a remote node. These features contribute to a higher GPU utilization, what translates into an increased cluster throughput (measured in jobs per time unit) and a reduced energy consumption.

In order to quantify the benefits of these features, in this subsection we study the impact that using the remote GPU virtualization mechanism has on the performance of a small cluster. To that end, we have executed several workloads in the cluster by submitting a series of randomly selected job requests to the Slurm queues. After job submission, several parameters have been measured, such as total execution time of the workloads, energy required to execute them, and GPU utilization. We have considered two different scenarios for workload execution. In the first one, the cluster uses CUDA and therefore applications can only use those GPUs installed in the same node where the application is being executed. In this scenario, an unmodified version of Slurm has been used. In the second scenario we have made use of rCUDA and therefore an application being executed in a given node can use any of the GPUs available in the cluster. Moreover, we have modified Slurm [6] so that it is possible to schedule the use of remote GPUs. These two scenarios will allow to compare the performance of a cluster using CUDA with that of a cluster using rCUDA. A 16-node cluster has been used for executing the workloads. The characteristics of the nodes are the ones mentioned before. One additional node has been leveraged in order to execute the central Slurm daemon responsible for scheduling jobs (the `slurmctld` process).

Several workloads have been considered in order to provide a more representative range of results. The workloads are composed of the following applications (see Table I): GPU-BLAST [27], LAMMPS [2], mCUDA-MEME [16], GROMACS [22], BarraCUDA [14], MUMmerGPU [8], GPU-LIBSVM [3], and NAMD [20]. They have been selected from the list of NVIDIA's Popular GPU-Accelerated Applications Catalog [13] because of their different characteristics. The versions of NAMD and GROMACS used in this study do not



```
bsc19421@nvb127:~
./deviceQuery Starting...

 CUDA Device Query (Runtime API) version (CUDART static linking)

Detected 64 CUDA Capable device(s)
```

Fig. 3: Screenshot of the deviceQuery Sample by NVIDIA when used with rCUDA after assigning 64 GPUs to an application.

TABLE I: Applications used in this study. Configuration details for each application

Application	Configuration	Execution time (s)	Memory per GPU
GPU-Blast	1 process with 6 threads in 1 node	21	1599 MB
LAMMPS	4 single-thread processes in 4 different nodes	15	876 MB
mCUDA-MEME	4 single-thread processes in 4 different nodes	165	151 MB
GROMACS	2 processes with 12 threads each one in 2 nodes	167	
BarraCUDA	1 single-thread process in 1 node	763	3319 MB
MUMmerGPU	1 single-thread process in 1 node	353	2104 MB
GPU-LIBSVM	1 single-thread process in 1 node	343	145 MB
NAMD	4 processes with 12 threads each one in 4 nodes	241	

make use of GPUs and therefore they are intended to contribute to a higher degree of heterogeneity of the workloads.

Table I provides additional information about the applications used in this study, such as the exact execution configuration used for each of the applications, showing the amount of processes and threads used for each of them. It can be seen that LAMMPS, mCUDA-MEME, GROMACS, and NAMD are MPI applications that will spread across several nodes in the cluster. On the contrary, the other four applications will execute in a single node. Additionally, some of the applications also make use of threads. For instance, it can be seen in the table that the GPU-Blast application uses a single process composed of 6 threads. During execution, each of these threads will use a different CPU core. In a similar way, the NAMD application will be distributed across 4 different nodes of the cluster (4 processes) and 12 threads will be launched at each node. Therefore, the NAMD application will make use of 4 entire nodes. In a similar way, the GROMACS application will keep busy two entire nodes while being executed.

Table I also shows the execution time for each application, which ranges from 15 up to 763 seconds for LAMMPS and BarraCUDA, respectively. Applications can be classified according to their execution time. In this regard, GPU-Blast, LAMMPS, mCUDA-MEME, and GROMACS require less than 170 seconds to complete execution (they are "short" applications) whereas BarraCUDA, MUMmerGPU, GPU-LIBSVM, and NAMD require more than 240 seconds to be executed ("long" applications).

In addition to execution time, Table I also shows the GPU memory required by each application. For those applications composed of several processes, the amount of GPU memory depicted in Table I refers to the individual needs of each particular process. Notice that the amount of GPU memory is not specified for the GROMACS and NAMD applications because we are using non-accelerated versions of these applications. The reason for this choice is simply to increase the heterogeneity degree of the workloads by using some CPU-only applications, as it could be the case in many data centers.

The previous applications have been combined in order to create three different workloads as shown in Table II. Workload labeled as "Set 1" is composed of 400 instances randomly selected from applications GPU-Blast, LAMMPS, mCUDA-MEME, and GROMACS. The exact amount of instances for each application is shown in the table. Additionally, the exact sequence of the applications within the workload is also randomly set. In a similar way, workload labeled as "Set 2" is composed of 400 instances of applications BarraCUDA, MUMmerGPU, GPU-LIBSVM, and NAMD. Finally, a third workload, referred to as "Set 1+2", has been created with instances from all the applications.

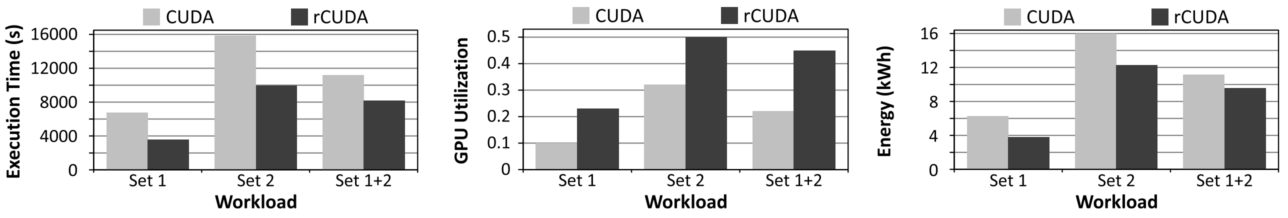
TABLE II: Workload composition

	Workload		
Application	Set 1	Set 2	Set 1+2
GPU-Blast	112		57
LAMMPS	88		52
mCUDA-MEME	99		55
GROMACS	101		47
BarraCUDA		112	51
MUMmerGPU		88	52
GPU-LIBSVM		99	37
NAMD		101	49
Total	400	400	400

Figure 4 shows the performance results. The figure shows, for each of the workloads depicted in Table II, the performance when CUDA is used along with the original Slurm workload manager (results labeled as "CUDA") as well as the performance when rCUDA is used in combination with the modified version of Slurm (label "rCUDA"). Figure 4(a) shows total execution time for each of the workloads. Figure 4(b) depicts the averaged GPU utilization for all the 16 GPUs in the cluster, whereas Figure 4(c) shows total energy required for completing workload execution.

As can be seen in Figure 4(a), workload "Set 1" presents the smallest execution time, given that it is composed of the applications requiring the smallest execution times. Furthermore, using rCUDA reduces execution time for the three workloads. In this regard, execution time is reduced by 48%, 37%, and 27% for workloads "Set 1", "Set 2", and "Set 1+2", respectively. Regarding GPU utilizacion, Figure 4(b) shows that the use of remote GPUs helps to increase overall GPU utilization. Actually, when rCUDA is used with "Set 1" and "Set 1+2", average GPU utilization is doubled with respect to the use of CUDA. Finally, total energy consumption is reduced accordingly, as shown in Figure 4(c), by 40%, 25%, and 15% for workloads "Set 1", "Set 2", and "Set 1+2", respectively.

Several are the reasons for the benefits obtained when GPUs are shared across the cluster. First, as already mentioned, the execution of the non-accelerated applications makes that GPUs in the nodes executing them remain idle when CUDA is used. On the contrary, when rCUDA is leveraged, these GPUs can be used by applications being executed in other



(a) Total execution time of the workloads. (b) Average GPU utilization. (c) Total energy consumed.

Fig. 4: Performance results from the 16-node 16-GPU cluster.

nodes of the cluster. The second reason for the improvements shown in Figure 4 is related to the usage that applications make of GPUs. As Table I showed, some applications do not completely exhaust GPU memory resources. For instance, applications mCUDA-MEME and GPU-LIBSVM only use about 3% of the memory present in the NVIDIA Tesla K20 GPU. However, the unmodified version of Slurm (combined with CUDA) will allocate the entire GPU for executing each of these applications, thus causing that almost 100% of the GPU memory is wasted during application execution. On the contrary, when rCUDA is used, GPUs can be shared among several applications provided that there is enough memory for all of them. Obviously, GPU cores will have to be multiplexed among all those applications, what will cause that all of them execute slower. However, one interesting point of view related to the slower execution of the applications sharing a GPU is that despite the slower execution of each individual application, the entire workload is completed earlier, as shown in Figure 4. This means that (1) the time spent by applications waiting in the Slurm queues is reduced and (2) the execution of each individual application is completed earlier.

Benefit 3: Cheaper Cluster Upgrade

The use of GPUs in a cluster usually puts several burdens on the physical configuration of the nodes in the cluster. For instance, nodes owning a GPU need to include larger power supplies able to provide the energy required by the accelerators. Also, GPUs are not small devices and therefore they require a non-negligible amount of space in the nodes where they are installed. These requirements make that installing GPUs in a cluster which did not initially include them is sometimes expensive (power supplies need to be upgraded) or simply impossible (nodes do not have enough physical space for the GPUs). However, the workload in some data centers may evolve towards the use of GPUs. At that point, the concern is how to address the introduction of GPUs in the computing facility.

One possible solution to the concern above is acquiring some amount of servers populated with GPUs and divert the execution of accelerated applications to those nodes. The Slurm workload manager would automatically take care of dispatching the GPU-accelerated applications to the new servers. However, although this approach is feasible, it presents the limitation that GPU jobs will probably have to wait for long until one of the GPU-enabled servers is available even though GPU utilization is usually low. Another concern is that MPI accelerated applications will only be able to span to as many nodes as GPU-enabled servers were acquired. Given these concerns, a better approach would be to acquire some amount of servers populated with GPUs and use rCUDA to execute accelerated applications at any of the nodes in the cluster while using the GPUs in the new servers. This solution would not only increase overall GPU utilization with respect to the use of CUDA in the previous scenario but would also allow MPI applications to span to as many nodes as required because MPI processes would be able to remotely access GPUs thanks to rCUDA. In summary, the remote GPU virtualization mechanism allows clusters which did not initially include GPUs to be easily and cheaply updated for using GPUs by attaching to them one or more computers containing GPUs. In this way, the original nodes will make use of the GPUs installed in the new nodes, which will become GPU servers. Slurm would be used to schedule the use of the GPUs in the new servers.

In order to analyze the performance of these two possible solutions, we have substituted one of the nodes in the cluster by a node containing four GPUs. This node is based on the Supermicro SYS7047GR-TRF server, populated with four NVIDIA Tesla K20 GPUs and one FDR InfiniBand network adapter. Furthermore, in order to additionally consider the use of parallel shared-memory applications in order to increase the heterogeneity of the workloads, we have modified the workloads used in the previous experiments by modeling shared-memory applications with two and four threads that require two and four GPUs, respectively. To that end, two different flavors of the LAMMPS and mCUDA-MEME applications have been used, as shown in Table III: (1) "LAMMPS long 2p" and "mCUDA-MEME long 2p" consist of two single-threaded processes that are forced to be executed in the same node. These instances of the applications will model the use of two-thread shared-memory applications, (2) "LAMMPS long 4p" and "mCUDA-MEME long 4p" consist of four single-threaded processes that will be forced to execute in the same node. They will model the use of four-thread shared-memory applications. One additional flavor of these applications will model single-thread shared-memory applications. This additional flavor is composed by the "LAMMPS short" and "mCUDA-MEME short" cases shown in Table III which make use of one single-threaded process. Furthermore, small input data sets are used for the "LAMMPS short" and "mCUDA-MEME short" cases whereas the multi-threaded flavors use a large input data set in order to lengthen their execution time.

TABLE III: Composition of two additional workloads

Application	Workload WL 1	Workload WL 2
GPU-Blast	41	48
LAMMPS short	39	46
LAMMPS long 2p	20	10
LAMMPS long 4p	20	10
mCUDA-MEME short	39	46
mCUDA-MEME long 2p	20	10
mCUDA-MEME long 4p	20	10
GROMACS	40	40
BarraCUDA	40	47
MUMmerGPU	41	47
GPU-LIBSVM	40	46
NAMD	40	40
Total	400	400

Figure 5 shows the performance results when a server with four GPUs has been attached to a cluster without GPUs. The original cluster is composed of 15 nodes (same node configuration as in the previous subsections, but GPUs have been removed). Results show that decoupling GPUs from nodes with rCUDA allows applications to make a much more flexible usage of the resources in the cluster and therefore execution time is reduced as well as energy consumption.

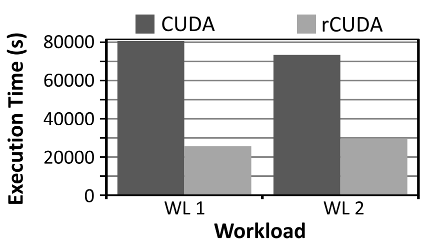
Benefit 4: Virtual Machines Can Easily Access GPUs

Providing CUDA acceleration to virtual machines can be accomplished by making use of the PCI passthrough technique [30]. This mechanism is based on the use of the virtualization extensions widely available in current high performance computing (HPC) servers, which allow assigning a GPU, in an exclusive way, to one of the virtual machines running at the host. Furthermore, when making use of this mechanism, the performance attained by accelerators is very close to that obtained when using the GPU in a native domain. Unfortunately, as this approach assigns GPUs to virtual machines in an exclusive way, it does not allow simultaneously sharing GPUs among the several virtual machines being concurrently executed at the same host. This issue constrains the use of GPUs in the cloud computing domain.

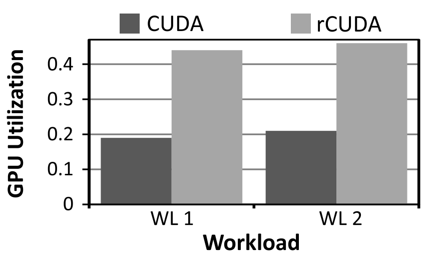
With the remote GPU virtualization mechanism it is possible to concurrently assign a given GPU to several virtual machines, so that the applications being executed inside them can share the GPU resources. Two different scenarios can be considered: one where virtual machines access a GPU located at the same host executing the virtual machines and another one where the InfiniBand fabric is already present in the cluster and therefore virtual machines access a GPU installed in another cluster node. Figure 6(a) depicts the first scenario whereas Figure 6(b) presents the second one.

In the first scenario, one of the virtual machines will have exclusive access to the GPU by making use of the PCI passthrough mechanism. This virtual machine will grant GPU access to the other virtual machines by using the rCUDA middleware: the rCUDA server will be executed in the virtual machine owning the GPU whereas the other virtual machines will use the rCUDA client to access the GPU across the Xen virtual network. TCP/IP based communications will be used in this scenario to communicate the rCUDA clients with the rCUDA server. Accordingly, virtual machines running the rCUDA client will have one or several virtual instances (vGPU) of the real GPU, which is physically connected to the virtual machine DomU_1. Moreover, the virtual machine DomU_1 will be able to use either the real GPU or its virtual instances. Notice that the rCUDA server can only be installed in one of the DomU_i virtual machines given that NVIDIA does not provide support for the Xen Linux kernel used in the Dom0 virtual machine.

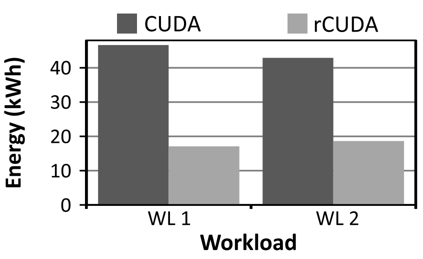
Regarding the second scenario, shown in Figure 6(b), which uses the InfiniBand fabric already present in the cluster to access a GPU in another node, the firmware in the InfiniBand adapter must be changed, according to the directions in Mellanox User's Guide [10], in order to provide several virtual instances (virtual functions, VFs) of the InfiniBand adapter, in addition to the real instance (physical function, PF). Each of these virtual functions will be provided, in an exclusive way, to a Xen virtual machine by using the PCI passthrough mechanism. Moreover, given that an InfiniBand network is available, communication between the rCUDA clients in the virtual machines and the remote rCUDA server will be based on the use of the high performance InfiniBand Verbs API. Notice that in the later experiments involving the InfiniBand fabric, the remote GPU server is executed in a remote computer which has not been virtualized and also whose InfiniBand network adapter makes use of the original firmware which does not provide virtualization features. Similarly to the scenario shown in Figure 6(a), virtual machines will have one or several virtual instances of the real GPU, which is physically located in the remote node. Finally, it is important to remark that, although



(a) Total execution time of the workloads.

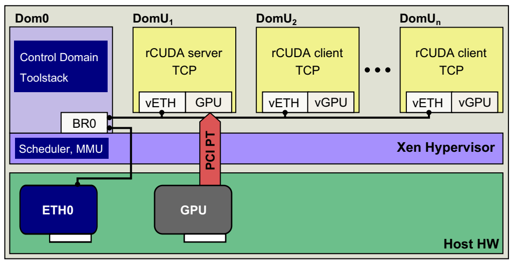


(b) Average GPU utilization.

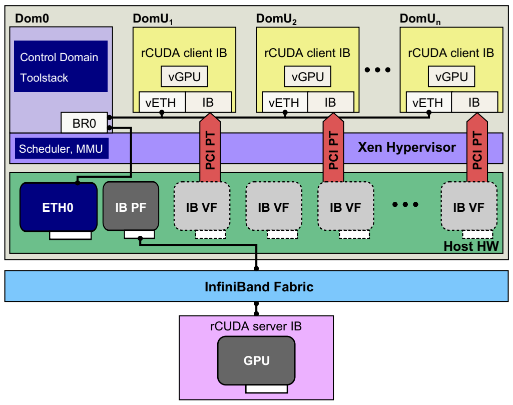


(c) Total energy consumed

Fig. 5: Performance results when a server with 4 GPUs is attached to a 15-node cluster without GPUs.



(a) Testbed using the virtual network within Xen.



(b) Testbed using InfiniBand to access a remote GPU.

Fig. 6: Testbeds used in the experiments presented in this subsection, which make use of rCUDA to provide GPU access to virtual machines. **(a)** In a single-node testbed, virtual machines employ the virtual network to access the rCUDA server by means of the TCP/IP protocol stack. **(b)** When an InfiniBand fabric is available, virtual machines use such interconnect to access a remote rCUDA server.

in this analysis we only consider sharing a single GPU, the rCUDA middleware also allows sharing multiple GPUs.

The testbed used in this subsection to explore the use of the remote GPU virtualization inside Xen virtual machines is composed of three 1027GR-TRF Supermicro nodes as the ones mentioned before. One of them will host the Xen virtual machines whereas the other two nodes will not make use of virtual machines. In one of the native domains we will execute the rCUDA server as shown in Figure 6(b) and the other native domain will be used for several comparison purposes. Regarding the software configuration, SUSE Linux Enterprise Server 11 SP3 (x86_64) was used in the three servers, with kernel version 3.0.76-0.11. Additionally, in the node hosting the virtual machines, Xen version 4.2.2 was used. The same kernel version was used in the Dom0 and all the DomU domains, although for Dom0 the kernel was recompiled in order to activate the Xen options. Finally, virtual machines were configured to have 4 cores and 12 GB of RAM memory. The applications used in this analysis are LAMMPS [2], CUDA-MEME [16], CUDASW++ [9], and GPU-BLAST [27], being all of them listed in the NVIDIA GPU-Accelerated Applications Catalog [13].

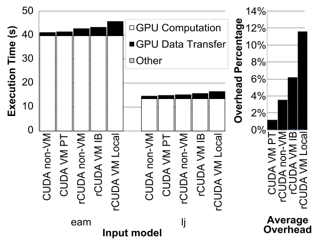
Figure 7 shows the performance of these four applications when executed in the following scenarios:

- Execution with CUDA with a local GPU in a native domain. Results for this scenario are referred to as *"CUDA non-VM"*.
- When CUDA is used in DomU$_1$ by using the PCI passthrough mechanism (rCUDA is not used), the label *"CUDA VM PT"* is used. In this case, the Xen virtual machine will access the GPU in the host by making use of PCI passthrough.
- The label *"rCUDA non-VM"* refers to the performance of the rCUDA middleware when used between native domains (no Xen virtual machine involved) making use of the InfiniBand network.
- When Xen virtual machines are involved in the tests, the performance of applications using rCUDA in the scenario depicted in Figure 6(a) is denoted by the label *"rCUDA VM Local"*.
- When using rCUDA in the scenario shown in Figure 6(b), the performance of applications will be labeled as *"rCUDA VM IB"*.

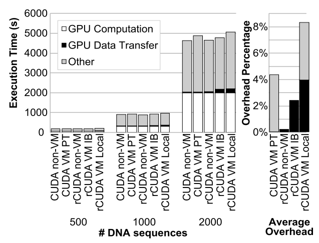
Every experiment has been performed 10 times, so that Figure 7 shows the averaged results. In addition to execution time, the plots in Figure 7 also include a breakdown of the execution time, which is split into three different components: (1) time required to transfer data to/from the GPU (*"GPU Data Transfer"*), (2) time spent making computations in the GPU (*"GPU Computation"*), and (3) time spent in tasks not involving the GPU, such as CPU computations and I/O (*"Other"*). Execution times presented in Figure 7 show that the four applications have a similar behavior, spending a very small portion of time for transferring data to the GPU, and spending the rest of the time making computations either in the CPU or in the GPU. More specifically, in the case of GPU-BLAST and CUDA-MEME applications, they present periods of time in which the GPU is not used. On the contrary, both LAMMPS and CUDASW++ keep the GPU busy for almost all the execution time.

Figure 7 also shows the average overhead with respect to executions with CUDA in a native domain for the four applications. It is shown that rCUDA overhead in LAMMPS, CUDASW++ and GPU-BLAST applications is mainly due to data transfers between main memory and GPU memory. Additionally to the overhead of transfers, the CUDA-MEME application also presents a performance decrease when using a virtual machine that makes use of the PCI passthrough technique. This additional overhead is not due to the increase of GPU data transfer time, but to the time spent in other tasks by the PCI passthrough technique.

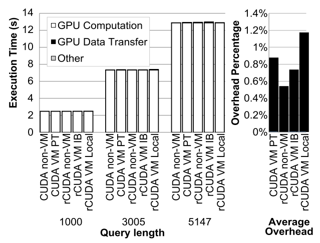
In general, the fact that the overhead of rCUDA is mainly due to data transfers between main memory and GPU memory was expected because once data is in the GPU memory, GPU computations require the same amount of time to be completed as in a native environment. In average, in the experiments, the overhead of running GPU-accelerated applications in a Xen virtual machine with respect to a native domain is 2%, 2.8%, and 5.8% when using PCI passthrough, rCUDA over an InfiniBand fabric, and rCUDA over the Xen virtual network, respectively.



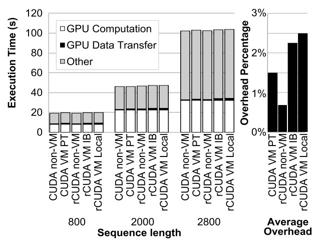
(a) LAMMPS application.



(b) CUDA-MEME application.



(c) CUDASW++ application.



(d) GPU-BLAST application.

Fig. 7: Execution time of several applications when executed in different local and remote scenarios. Execution time is broken down into three components: GPU computation, GPU data transfer, and Other.

IV. Conclusions

In this paper it has been shown that the use of the remote GPU virtualization technique provides several benefits to computing facilities. For instance, the improvements attained in execution time for a batch of jobs have been quantified. The associated reduction in energy consumption has also been presented. These features may be interesting in the context of exascale computing facilities given that one of the walls in this area is the hard power consumption limitation.

Notice, however, that the remote GPU virtualization mechanism can also be useful for migrating the GPU jobs from one GPU server to another. It is quite complex to perform this migration in an efficient way when this virtualization mechanism is not being used, but on the contrary it is very simple when the rCUDA technology is used due to the fact that rCUDA intercepts all the CUDA calls and tracks the state of the memory areas used by the application in the GPU. Migrating GPU jobs would be an inexpensive and efficient way of consolidating GPU servers, so that as many GPU jobs as possible are packed together, switching off those GPU servers not required. This would be a means of further reducing the total energy consumed in exascale computing facilities.

Acknowledgment

This work was funded by Generalitat Valenciana under Grant PROMETEOII/2013/009 of the PROMETEO program phase II. The author from Universitat Jaume I was supported by project TIN2014-53495-R from MINECO and FEDER. The authors are grateful for the generous support provided by Mellanox Technologies and the equipment donated by NVIDIA Corporation.

References

[1] P. K. Agarwal et al., "Performance modeling of microsecond scale biological molecular dynamics simulations on heterogeneous architectures", CCPE Journal, 2013

[2] W. M. Brown et al., "Implementing molecular dynamics on hybrid high performance computers: Particle-particle particle-mesh," *Computer Physics Communications*, 2012.

[3] C. C. Chang et al., "LIBSVM: A library for support vector machines," *ACM Trans. Intell. Syst. Technol.*, 2011

[4] G. Giunta et al., "A GPGPU transparent virtualization component for HPC clouds," *Euro-Par 2010*

[5] V. Gupta *et al.*, "GViM: GPU-accelerated virtual machines," in *HPCVirt*, 2009.

[6] S. Iserte et al. "SLURM support for remote GPU virtualization: implementation and performance." *SBAC-PAD* 2014

[7] H. Jo et al., "Exploiting GPUs in Virtual Machine for BioCloud," BioMed research international, 2013

[8] S. Kurtz et al., "Versatile and open software for comparing large genomes," *Genome Biology*, 2004

[9] Y. Liu et al., "CUDASW++ 3.0: accelerating smith-waterman protein database search by coupling CPU and GPU SIMD instructions," BMC Bioinformatics 14 (1) (2013) 1–10

[10] Mellanox, Mellanox OFED for Linux User Manual (2015).

[11] NVIDIA, *CUDA C Programming Guide 7.0*, 2015

[12] NVIDIA, *CUDA Runtime API 7.0*, 2015

[13] NVIDIA, "GPU Applications," http://www.nvidia.com/object/gpu-applications.html, 2015.

[14] P. Lam et al., "BarraCUDA - a fast short read sequence aligner using graphics processing units", *BMC Research Notes*, 2012

[15] T. Y. Liang et al., "GridCuda: A Grid-Enabled CUDA Programming Toolkit," *WAINA 2011*

[16] Y. Liu et al., "CUDA-MEME: Accelerating motif discovery in biological sequences using CUDA-enabled graphics processing units," *Pattern Recognition Letters*, 2010

[17] D. Y. Luo, "Canny edge detection on NVIDIA CUDA", Computer Vision and Pattern Recognition Workshops, 2008

[18] M. Oikawa et al., "DS-CUDA: A Middleware to Use Many GPUs in the Cloud Environment," *SC 2012*

[19] A. J. Peña et al., "A complete and efficient CUDA-sharing solution for HPC clusters," *Parallel Computing*, vol. 40, 2014

[20] J. C. Phillips et al., "Scalable molecular dynamics with namd," *Journal of Computational Chemistry*, 2005

[21] D. P. Playne et al., "Data parallel three-dimensional cahn-hilliard field equation simulation on GPUs with CUDA", PDPTA 2009

[22] S. Pronk et al., "Gromacs 4.5: a high-throughput and highly parallel molecular simulation toolkit," *Bioinformatics*, 2013

[23] C. Reaño et al., "A performance comparison of CUDA remote GPU virtualization frameworks," *Cluster 2015*

[24] C. Reaño et al., "Local and Remote GPUs Perform Similar with EDR 100G InfiniBand," *Middleware 2015*

[25] L. Shi et al.,"vCUDA: GPU accelerated high performance computing in virtual machines," *IPDPS 2009*

[26] V. Surkov, "Parallel option pricing with fourier space time-stepping method on graphics processing units", Parallel Computing, 2010

[27] P. D. Vouzis et al., "GPU-BLAST: Using GPUs to accelerate protein sequence alignment," *Bioinformatics*, 2010

[28] H. Wu et al. "Red Fox: An Execution Environment for Relational Query Processing on GPUs", CGO 2014

[29] I. Yamazaki et al., "Tridiagonalization of a dense symmetric matrix on multiple GPUs and its application to symmetric eigenvalue problems", CCPE Journal, 2014

[30] C.-T. Yang et al., "On implementation of GPU virtualization using PCI pass-through, " CloudCom, 2012

[31] A. Yoo et al., "Simple linux utility for resource management," *Job Scheduling Strategies for Parallel Processing*, 2003