Silla Jiménez, F.; Iserte Agut, S.; Reaño González, C.; Prades, J. (2017). On the Benefits of the Remote GPU Virtualization Mechanism: the rCUDA Case. Concurrency and Computation Practice and Experience. 29(13):1-17. https://doi.org/10.1002/cpe.4072

CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 2010; **00**:1–29
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/cpe

On the Benefits of the Remote GPU Virtualization Mechanism: the rCUDA Case

F. Silla[1*], S. Iserte[2], C. Reaño[1], and J. Prades[1]

[1] *Universitat Politècnica de València, Spain*
[2] *Universitat Jaume I, Spain*

SUMMARY

Graphics Processing Units (GPUs) are being adopted in many computing facilities given their extraordinary computing power, which makes it possible to accelerate many general purpose applications from different domains. However, GPUs also present several side effects, such as increased acquisition costs as well as larger space requirements. They also require more powerful energy supplies. Furthermore, GPUs still consume some amount of energy while idle and their utilization is usually low for most workloads.
In a similar way to virtual machines, the use of virtual GPUs may address the aforementioned concerns. In this regard, the remote GPU virtualization mechanism allows an application being executed in a node of the cluster to transparently use the GPUs installed at other nodes. Moreover, this technique allows to share the GPUs present in the computing facility among the applications being executed in the cluster. In this way, several applications being executed in different (or the same) cluster nodes can share one or more GPUs located in other nodes of the cluster. Sharing GPUs should increase overall GPU utilization, thus reducing the negative impact of the side effects mentioned before. Reducing the total amount of GPUs installed in the cluster may 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 the computing facility. Another benefit of this technique is that cluster throughput, measured as jobs completed per time unit, is noticeably increased when this technique is used. In this regard, cluster throughput can be doubled for some workloads. Furthermore, in addition to increase overall GPU utilization, total energy consumption can be reduced up to 40%. This may be key in the context of exascale computing facilities, which present an important energy constraint. Other benefits are related to the cloud computing domain, where a GPU can be easily shared among several virtual machines. Finally, GPU migration (and therefore server consolidation) is one more benefit of this novel technique.

Received . . .

KEY WORDS: GPGPU; CUDA; GPU virtualization; rCUDA; Slurm; virtual machine; cloud computing; InfiniBand; GPU migration; Xen

*Correspondence to: Universitat Politècnica de València. Departamento de Informática de Sistemas y Computadores (DISCA). Edificio 1G. 46022 Valencia, Spain. E-mail: fsilla@disca.upv.es

Contract/grant sponsor: 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

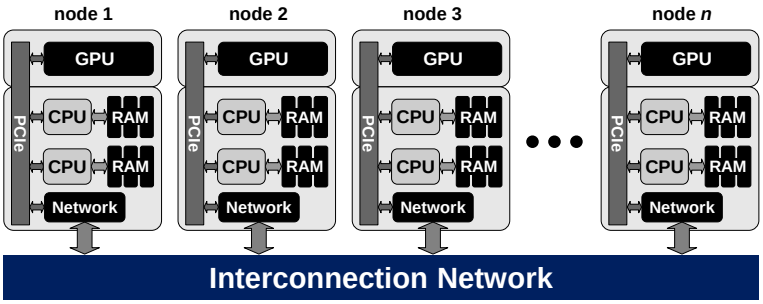


Figure 1. Example of a GPU-accelerated cluster.

1. 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) [1] noticeably ease this task. In this context, GPUs significantly reduce the execution time of applications from domains as different as Big Data [2], chemical physics [3], computational algebra [4], image analysis [5], finance [6], and biology [7], for instance.

Current computing facilities typically include one or more GPUs in the nodes of the cluster. Depending on the exact cluster configuration, GPUs may be present only at some of the nodes or they may be installed at every node. Figure 1 shows an example of a deployment, composed of n nodes, where each node includes one GPU. These nodes could be, for instance, SYS1027-TRF Supermicro servers containing two Xeon processors and one NVIDIA Tesla GPU. Additionally, the interconnection network could be an FDR InfiniBand fabric. Notice, however, that 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 thus 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 cause that those GPUs remain idle for some periods of time, thus reducing overall GPU utilization and making that the initial economic investment done during cluster acquisition requires more time to be amortized.

Another example of the concerns associated with the use of GPUs in clusters is related to the way that job schedulers such as Slurm [8] 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

MINECO and FEDER. The authors are grateful for the generous support provided by Mellanox Technologies and the equipment donated by NVIDIA Corporation.

DOI: 10.1002/cpe

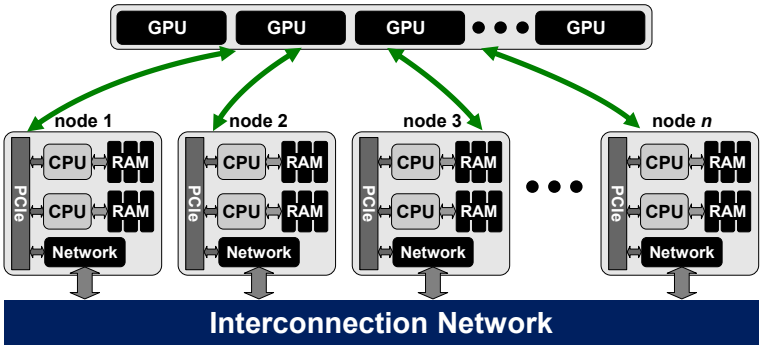


Figure 2. Logical configuration of a cluster when the remote GPU virtualization technique is used.

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 on it, causing that overall GPU utilization is, in general, low. This fact not only reduces the effective computing power of clusters 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 some of 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 GPUs are concurrently shared by all the nodes of the computing facility in a transparent way to applications. Figure 2 shows the new cluster envision after applying the remote GPU virtualization mechanism. In the new cluster configuration GPUs are logically detached from nodes and a pool of GPUs is created. GPUs in this pool can be accessed from any node in the cluster. Furthermore, a given GPU may concurrently serve more than one application. This sharing of GPUs not only increases overall GPU utilization but also allows to create cluster configurations where not all the nodes in the cluster own a GPU at the same time that all the nodes in the cluster can execute GPU-accelerated applications. This cluster configuration would reduce the costs associated with the acquisition and later use of GPUs. In this regard, the total energy required to operate a computing facility may be decreased, thus loosening the big energy concerns of future exascale computing installations.

In this paper we explore some of the benefits that the remote GPU virtualization mechanism provides to clusters. We present this exploration in the context of the rCUDA middleware, given that it is the most modern remote GPU virtualization solution and also the one that provides the best performance, as it will be shown later in the paper. The rest of the paper is organized in the following way: Section 2 presents a review of the remote GPU virtualization technique. Later, Section 3 introduces the rCUDA technology in more detail, given that it will be the one used in this work to quantify the benefits of the remote GPU virtualization mechanism. Next, Section 4 introduces six of the benefits of this virtualization technique. Finally, Section 5 concludes the paper. Notice that this paper is an extension of a previous workshop paper [9].

Prepared using cpeauth.cls
DOI: 10.1002/cpe

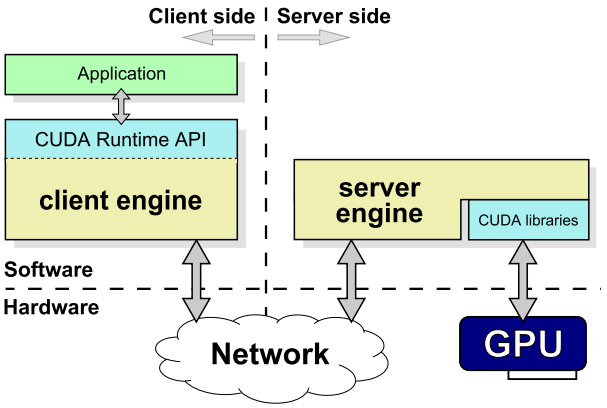


Figure 3. Organization of remote GPU virtualization frameworks.

2. REMOTE GPU VIRTUALIZATION SOLUTIONS

Frameworks such as CUDA [1] assist programmers in using GPUs for general-purpose computing. Several remote GPU virtualization solutions exist for this framework, such as GridCuda [10], DS-CUDA [11], gVirtuS [12], vCUDA [13], GViM [14], and rCUDA [15]. Basically, these middleware proposals share a GPU by virtualizing it. In this way, these middleware solutions provide applications with virtual instances of the real device, which can therefore be concurrently shared. Usually, these GPU sharing solutions place the virtualization boundary at the API level (CUDA in the case of NVIDIA GPUs). In general, CUDA-based virtualization solutions aim to offer the same API as the NVIDIA CUDA Runtime API [16] does.

Figure 3 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. 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 original application, which is not aware that its request has been served by a remote GPU instead of a local one.

CUDA-based GPU virtualization solutions may be classified into two types: (1) those intended to be used in the context of virtual machines and (2) those devised as general purpose virtualization solutions, to be used in native domains (notice that these latter solutions may also be used within virtual machines). Frameworks in the first category usually make use of shared-memory mechanisms in order to transfer data from main memory inside the virtual machine to the GPU in the native domain, whereas the general purpose virtualization solutions in the second type make use of the network fabric in the cluster to transfer data from main memory in the client side to the remote GPU located in the server. This is why these latter solutions are commonly known as remote GPU virtualization solutions.

Regarding the first type of GPU virtualization solutions mentioned above, several solutions have been developed to be specifically used within virtual machines, such as for example vCUDA, GViM,

DOI: 10.1002/cpe

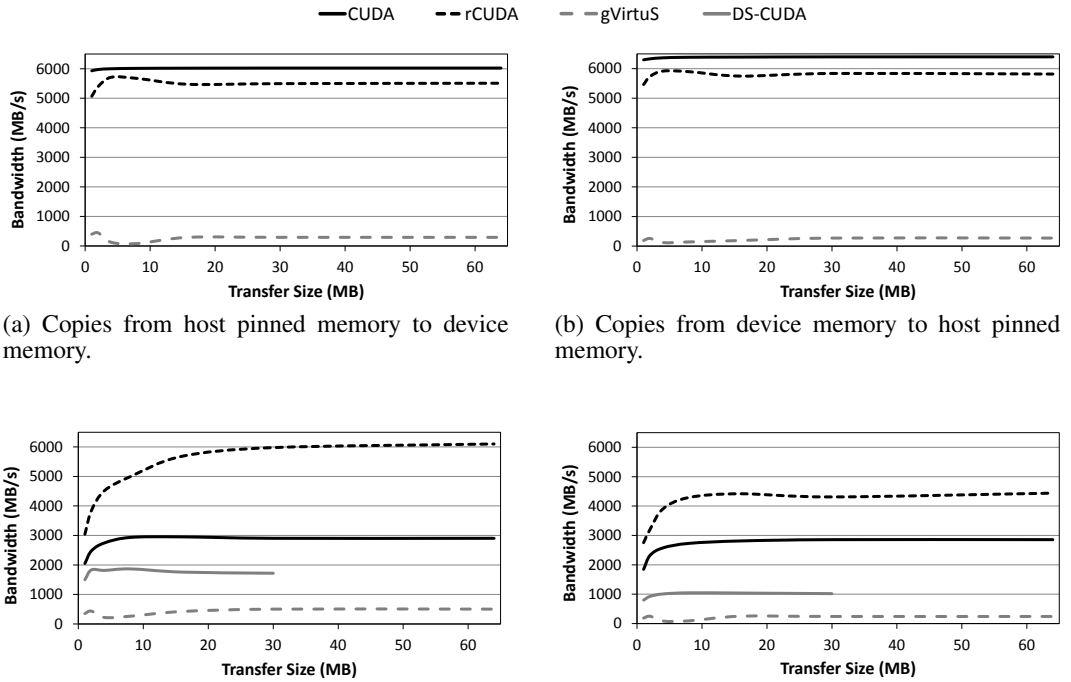
gVirtuS, and Shadowfax [17]. The vCUDA technology, intended for Xen virtual machines, only supports an old CUDA version (v3.2) and implements an unspecified subset of the CUDA Runtime API. Moreover, its communication protocol presents a considerable overhead because of the cost of the encoding and decoding stages. This overhead causes a noticeable drop in overall performance. GViM, targeting Xen environments, is based on the obsolete CUDA version 1.1 and, in principle, does not implement the entire CUDA Runtime API. gVirtuS is based on the old CUDA version 2.3 and implements only a small portion of its API. Despite being designed for virtual machines, it also provides TCP/IP communications for remote GPU virtualization, thus allowing applications in a non-virtualized environment to access GPUs located in other nodes. Regarding Shadowfax, this solution allows Xen virtual machines to access the GPUs located at the same node, although it may also be used to access GPUs at other nodes of the cluster. It supports the obsolete CUDA version 1.1 and, additionally, neither the source code nor the binaries are available in order to evaluate its performance.

In the second type of virtualization solutions mentioned above, which provide general purpose GPU virtualization, one can find rCUDA, GridCuda, DS-CUDA, and Shadowfax II [18]. rCUDA, further described in Section 3, features CUDA 7.5 and provides specific communication support for TCP/IP compatible networks as well as for InfiniBand fabrics (rCUDA uses the InfiniBand Verbs API in order to leverage the RDMA features of this network). GridCuda also offers access to remote GPUs in a cluster, but supports an old CUDA version (v2.3). Moreover, there is currently no publicly available version of GridCuda that can be used for testing. Regarding DS-CUDA, it integrates a more recent version of CUDA (4.1) and includes specific communication support for InfiniBand by making use of the InfiniBand Verbs API. However, DS-CUDA presents several strong limitations, such as not allowing data transfers with pinned memory. Finally, Shadowfax II is still under development, not presenting a stable version yet and its public information is not updated to reflect the current status of the source code.

In order to provide a comprehensive comparison among the different GPU virtualization solutions described in this section, Figure 4 presents a performance analysis of three publicly available GPU virtualization solutions: DS-CUDA, rCUDA, and gVirtuS. This figure also shows the performance of CUDA as the baseline reference. The widely used `bandwidthTest` benchmark from the NVIDIA CUDA Samples [19] has been employed. The testbed employed for carrying out the performance experiments is based on two Supermicro servers as the ones described in Section 4. The bandwidth test (along with the client side of the different frameworks) was run in one of the computers whereas the server side of the middleware solutions was executed in another computer which owns a Tesla K20 GPU. The InfiniBand FDR network technology was used to connect both computers. Therefore, both the rCUDA and DS-CUDA solutions made use of the InfiniBand Verbs API. In the case of gVirtuS, given that it is not able to take advantage of the InfiniBand Verbs API, TCP/IP over InfiniBand was used.

Results in Figure 4 deserve some discussion. First, it can be seen that CUDA achieves the highest performance when pinned memory is used (Figures 4(a) and 4(b)), attaining a bandwidth around 6000 MB/s. Notice that this bandwidth is reduced to the half for copies using pageable memory (Figures 4(c) and 4(d)). Second, Figure 4 shows that rCUDA outperforms the other two remote GPU virtualization solutions. Actually, for copies using pageable memory rCUDA also performs better than CUDA. This is a well-known effect thoroughly described in previous works on rCUDA [15] and

DOI: 10.1002/cpe



(a) Copies from host pinned memory to device memory.

(b) Copies from device memory to host pinned memory.

(c) Copies from host pageable memory to device memory.

(d) Copies from device memory to host pageable memory.

Figure 4. Performance comparison among three publicly available CUDA GPU virtualization solutions: gVirtuS, DS-CUDA, and rCUDA. The comparison is performed in terms of attained bandwidth. The performance of CUDA is also depicted for comparison purposes.

is due to the use of an efficient pipelined communication based on the use of internal pre-allocated pinned memory buffers. On the other hand, notice that both rCUDA and DS-CUDA make use of the InfiniBand Verbs API, thus having access to the large bandwidth available in this interconnect. However, although rCUDA is able to struggle an important fraction of the available bandwidth, DS-CUDA presents a relatively poor performance. Therefore, it must be assumed that the difference in bandwidth is due to the different way that both GPU virtualization solutions manage the InfiniBand interconnect. Also notice that DS-CUDA supports neither memory copies larger than 32MB nor the use of pinned memory. On the other hand, notice that the performance of gVirtuS is extremely low. One may think that this is due to the fact that gVirtuS is using TCP/IP over InfiniBand, which clearly achieves lower performance than the InfiniBand Verbs API. However, according to our measurements with the iperf tool [20], InfiniBand FDR provides around 1190 MB/s when TCP/IP over InfiniBand is used. This bandwidth is noticeably larger than the one attained by gVirtuS. Hence, the low performance of this middleware is not due to the use of TCP/IP over InfiniBand but to the way it internally manages communications.

As a final consideration for this review section, it is important to remark that although remote GPU virtualization has traditionally introduced a non-negligible overhead, given that applications do not access GPUs attached to the local PCI Express (PCIe) link but rather access devices that are installed in other nodes of the cluster (traversing a network fabric with a lower bandwidth), this performance overhead has significantly been reduced thanks to the recent advances in networking

DOI: 10.1002/cpe

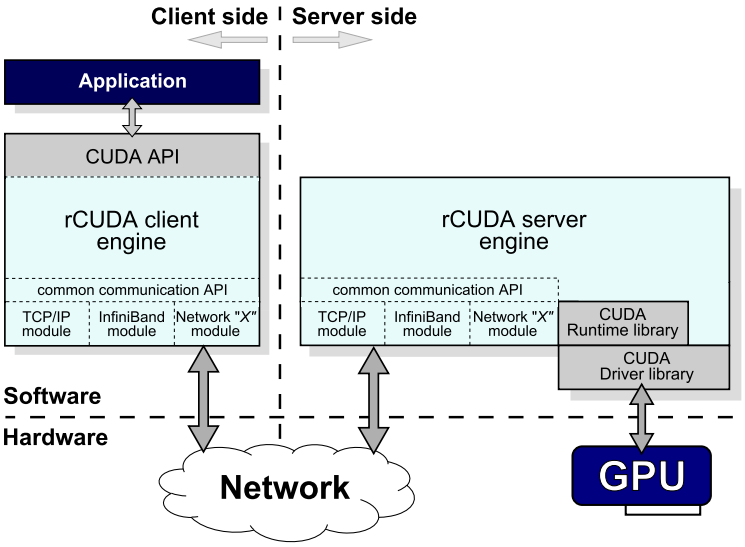


Figure 5. rCUDA layered and modular architecture.

technologies as well as a careful design of the remote virtualization solution, as shown in Figure 4 for the rCUDA framework. The reader may refer to [21] for a deeper analysis.

3. rCUDA: REMOTE CUDA

As already mentioned in the introduction section, we use in this study the rCUDA middleware given that it is the most up-to-date solution, providing also the best performance among the different publicly available GPU virtualization solutions, as shown in the previous section. Furthermore, it was the only framework able to run the applications analyzed in this paper. In this section we present rCUDA in more detail. Figure 5 depicts a detailed view of the architecture of the rCUDA middleware.

The rCUDA middleware supports version 7.5 of CUDA, being binary compatible with it, which means that CUDA programs do not need to be modified for using rCUDA. Furthermore, it implements the CUDA Runtime and Driver APIs (except for graphics functions) and also provides support for the libraries included within CUDA, such as cuFFT, cuBLAS, or cuSPARSE. Moreover, the rCUDA middleware allows a single rCUDA server to concurrently deal with several remote clients that simultaneously request GPU services. This is achieved by creating independent GPU contexts, each of them being assigned to a different client [15]. These independent GPU contexts also provide robustness against the failure of one of the clients.

rCUDA provides specific support for different interconnects. Support for different underlying network fabrics is achieved by making use of a set of runtime-loadable, network-specific communication modules, which have been specifically implemented and tuned in order to obtain as much performance as possible from the underlying interconnect. Currently, two modules are available: one intended for TCP/IP compatible networks and another one specifically designed for InfiniBand.

Prepared using cpeauth.cls
DOI: 10.1002/cpe

Regarding the InfiniBand communications module, it is based on the InfiniBand Verbs API. This API offers two communication mechanisms: the channel semantics and the memory semantics. The former refers to the standard send/receive operations typically available in any networking library, while the latter offers RDMA operations where the initiator of the operation specifies both the source and destination of a data transfer, resulting in zero-copy transfers with minimum involvement of the CPUs. rCUDA employs both mechanisms, selecting one or the other depending on the exact task to be carried out [15].

Moreover, regardless of the exact network used, data exchange between rCUDA clients and GPUs managed by rCUDA servers is pipelined so that higher bandwidth is achieved. Internal pipeline buffers within rCUDA use pre-allocated pinned memory given the higher throughput of this type of memory.

Using rCUDA is very simple. It just requires to set three environment variables prior to application execution: `RCUDA_DEVICE_COUNT`, `RCUDA_DEVICE_j`, and `RCUDAPROTO`. The first variable indicates the amount of GPUs accessible by the application. For example, if two GPUs are assigned to the application, then the command "`export RCUDA_DEVICE_COUNT=2`" should be executed. The second environment variable, `RCUDA_DEVICE_j`, indicates, for each of the n GPUs assigned to the application, in which cluster node the GPU with identifier j is located. For instance, in the previous example, the commands "`export RCUDA_DEVICE_0=192.168.0.1`" and "`export RCUDA_DEVICE_1=192.168.0.2`" should be executed. Finally, the `RCUDAPROTO` environment variable sets the communication module to be used during the execution of the application. For instance, the command "`export RCUDAPROTO=IB`" should be used in order to leverage the InfiniBand Verbs API. In case of using the TCP/IP communication module, the command "`export RCUDAPROTO=TCP`" should be executed.

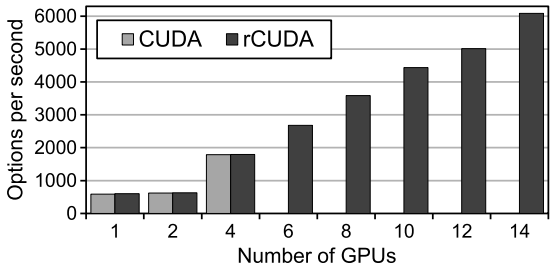
4. BENEFITS OF USING REMOTE GPU VIRTUALIZATION

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

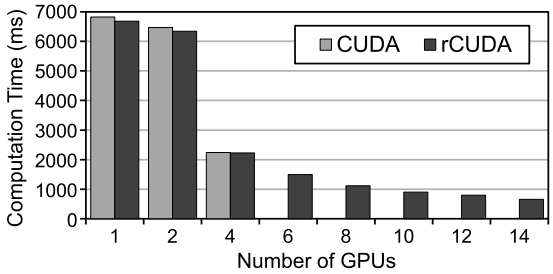
1. More GPUs are available for a single application.
2. Busy CPU sockets in a server do not hinder the use of the GPUs at that server.
3. Cluster throughput is increased at the same time that energy consumption is reduced. Overall GPU utilization is also increased.
4. Cluster upgrades are made easier and cheaper just by attaching GPU servers to a non-GPU cluster.
5. Several virtual machines can concurrently access the same GPU in a shared manner.
6. GPU jobs can be easily migrated across the cluster in order to consolidate them into fewer servers.

The next subsections describe and analyze these benefits. A performance evaluation is included for most 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

DOI: 10.1002/cpe



(a) Total options per second computed.



(b) Execution time.

Figure 6. Performance of the MontecarloMultiGPU Sample by NVIDIA with a varying number of GPUs when using CUDA and rCUDA.

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 7.5 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 Available 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 parallel 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 GPUs in the cluster, thus boosting its performance. In this case, the only limitation to increase application performance would be 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 6 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

DOI: 10.1002/cpe

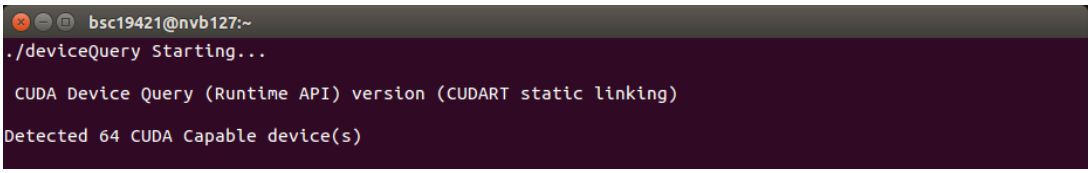


Figure 7. Screenshot of the deviceQuery Sample by NVIDIA when used with rCUDA after assigning 64 GPUs to an application.

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 the 4 GPUs inside the Supermicro SYS7047GR-TRF server can be used for the CUDA executions. On the contrary, when rCUDA is used, many additional GPUs can be provided to the application. Figure 6 shows how the use of a larger 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 Section 2, as well as the faster synchronization of rCUDA with respect to CUDA, as shown in [22].

On the other hand, Figure 7 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, showing the possibilities that remote GPU virtualization brings.

Benefit #2: Busy CPUs in a Server Do Not Hinder The Use of The GPUs at That Server

Users of a cluster tend to require as many computing resources as possible for executing their applications in order to reduce application execution time. Requiring as many resources as possible may happen in several ways. For instance, it is quite common that users submitting a non GPU-accelerated shared-memory parallel application to the job scheduler queues in the system request for their application as many CPU cores as available in the node. In practice, this requirement translates into the application using all the CPU cores of the cluster node where the application has been launched. In this way, during the execution of such an application, no other application can be executed in that node due to the lack of CPU cores.

In a similar way, users may also submit to the job scheduler queues requests for executing non-accelerated hybrid MPI shared-memory applications. These applications span over several nodes of the cluster, usually flooding all the CPU cores present at each of the nodes. As in the previous example, during the time that one of these hybrid applications is being executed, no other application can fit into the nodes that the former application is using because there is no CPU core available.

Although the execution of the mentioned applications may lead to a reduced application execution time and therefore an overall high CPU utilization, when the nodes involved in their execution include one or more GPUs, these accelerators will remain idle during the time that these non-accelerated applications are being executed. The accelerators in those nodes become unavailable for other applications because, in order to use them, it is required to launch an application in those nodes. However, this is not possible because that application would require at least one

DOI: 10.1002/cpe

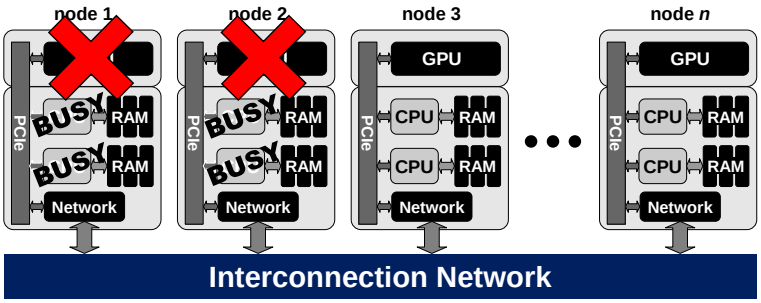


Figure 8. GPUs in nodes 1 and 2 are not available because all the CPU cores at those nodes are busy with the execution of non-accelerated applications.

available CPU core but all the CPU cores have been allocated to the non-accelerated application and therefore the job scheduler will not forward any application to that node. This condition is depicted in Figure 8, where the GPUs in nodes 1 and 2 of the cluster are not available because all the CPU cores at those nodes are busy with the execution of non-accelerated applications. Notice that these blocked GPUs do not only cause a temporal reduction on the overall computing power of the cluster but they still consume some amount of energy. For instance, the NVIDIA Tesla K20 GPU consumes 25W during the idle state. In a similar way, the NVIDIA Tesla K40 GPU wastes 20W while in this condition.

The remote GPU virtualization mechanism may be useful in the previous scenarios, as shown in Figure 9. When non-accelerated applications block the use of the GPUs in one or several nodes of the cluster, frameworks such as rCUDA may allow to use the blocked GPUs by allocating them to applications being executed in other nodes of the cluster. In this way, the free CPU cores that were missing in the previous scenarios will now be located in other nodes. The net result is that, in addition to increase overall CPU utilization, GPU utilization is also increased. On the other hand, remember that the rCUDA framework makes use of the rCUDA server in order to provide access to remote GPUs. This server, which is run as a daemon, must be executed in one of the CPU cores of the node owning the GPU. Given that in the scenarios considered above all the CPU cores in the nodes with blocked GPUs are being used for the execution of the non-accelerated application, one may wonder whether the execution of the rCUDA server would introduce an important overhead,

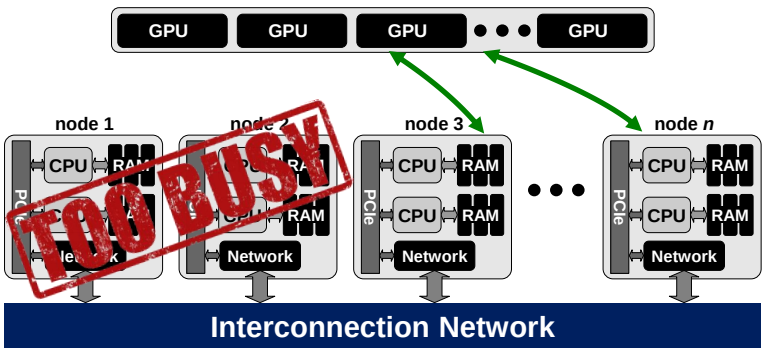


Figure 9. The remote GPU virtualization mechanism allows GPUs in nodes with busy CPU cores to be used by applications being executed in other nodes of the cluster.

DOI: 10.1002/cpe

which in turn would penalize the execution time of the non-accelerated application. Nevertheless, such an analysis is beyond the scope of this paper and, additionally, has already been addressed in [15].

Benefit #3: 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 another cluster node, as described in the previous Benefit #2. 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 jobs 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 executing the workloads. 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 [23] so that it is possible to schedule the use of remote GPUs. These two scenarios will allow us 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 same as the ones mentioned before (two Xeon E5-2620 v2 sockets with one NVIDIA Tesla K20 GPU and one FDR InfiniBand adapter). One additional node (the 17^{th} node) has been leveraged in order to execute the Slurm controller 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 [24], LAMMPS [25], mCUDA-MEME [26], GROMACS [27], BarraCUDA [28], MUMmerGPU [29], GPU-LIBSVM [30], and NAMD [31]. They have been selected from the list of NVIDIA's Popular GPU-Accelerated Applications Catalog [32] because of their different characteristics. The versions of NAMD and GROMACS used in this study do not make use of GPUs and therefore they are intended to contribute to a higher degree of heterogeneity of the workloads.

Table I provides 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

Prepared using cpeauth.cls
DOI: 10.1002/cpe

Table I. Configuration details for each of the applications used in the workloads employed to test cluster performance.

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	–

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. Furthermore, as both the NAMD and GROMACS applications do not make use of GPUs, the concern mentioned in the previous Benefit#2 about the use of the accelerators will appear.

Table I also shows the execution time for each application, which ranges from 15 up to 763 seconds for LAMMPS and BarraCUDA, respectively. In this regard, applications can be classified according to their execution time. For instance, 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.

In summary, the eight applications used in this study present different characteristics, not only regarding the amount of processes and threads used by each of them and their execution time but they also present different GPU usage patterns, what includes both memory copies to/from GPUs and also kernel executions. Therefore, although the set of applications considered is finite, it may provide a representative sample of a workload typically found in current data centers. Actually, the set of applications in Table I could be considered from two different points of view. In the first one, the exact computations performed by each application would receive the main focus. In this point of view, some applications address similar problems, like LAMMPS, GROMACS, and NAMD. However, in the second point of view, the exact problem addressed by each application is not the focus but applications are seen as processes that keep CPUs and GPUs busy during some amount of time and require some amount of memory. Now the focus is the amount of resources required by

DOI: 10.1002/cpe

Table II. Slurm launching parameters

Application	Launch with CUDA	Launch with rCUDA
GPU-Blast	-N1 -n1 -c6 –gres=gpu:1	-n1 -c6 –gres=rgpu:1:1599M
LAMMPS	-N4 -n4 -c1 –gres=gpu:1	-n4 -c1 –gres=rgpu:4:876M
mCUDA-MEME	-N4 -n4 -c1 –gres=gpu:1	-n4 -c1 –gres=rgpu:4:151M
GROMACS	-N2 -n2 -c12	-N2 -n2 -c12
BarraCUDA	-N1 -n1 -c1 –gres=gpu:1	-n1 -c1 –gres=rgpu:1:3319M
MUMmerGPU	-N1 -n1 -c1 –gres=gpu:1	-n1 -c1 –gres=rgpu:1:2104M
GPU-LIBSVM	-N1 -n1 -c1 –gres=gpu:1	-n1 -c1 –gres=rgpu:1:145M
NAMD	-N4 -n48 -c1	-N4 -n48 -c1

each application and the time that those resources are kept busy. From this second perspective, the set of applications in Table I becomes even more representative.

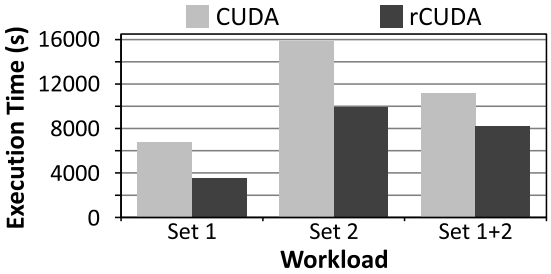
Table II displays the Slurm parameters used for launching each of the applications. The use of real and virtual GPUs has been considered in the table. In the first case, CUDA will be used (column labeled "Launch with CUDA"). In the second case, remote GPUs can be shared among several applications. To that end, the amount of memory required at each GPU must be specified in the submission command, as shown in the column labeled as "Launch with rCUDA". On the other hand, it can be seen by comparing the parameters in the two columns that when CUDA is used it is required that MPI applications are mapped to different nodes (parameter "-Ni" where i is the amount of different nodes requested) whereas this requirement is removed when rCUDA is employed.

The previous applications have been combined in order to create three different workloads as shown in Table III. 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.

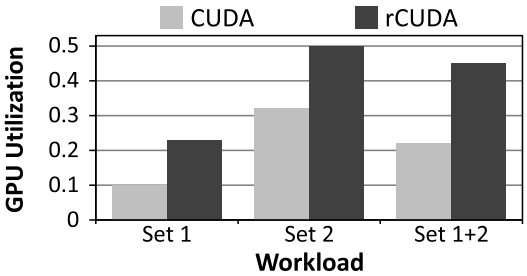
Table III. Workload composition

Application	Workload		
	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

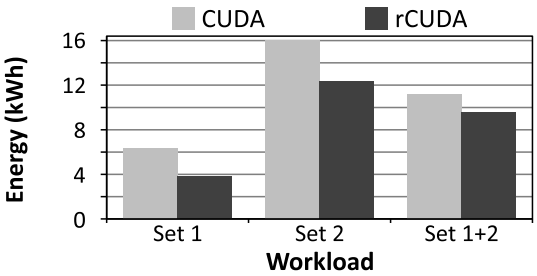
DOI: 10.1002/cpe



(a) Total execution time of the workloads.



(b) Average GPU utilization.



(c) Total energy consumed during the execution of the workloads.

Figure 10. Performance results from the 16-node 16-GPU cluster.

Figure 10 shows the performance results. Remember that a small cluster composed of 16 nodes with one GPU at each node is being used. The figure shows, for each of the workloads depicted in Table III, the performance when CUDA is used along with the original Slurm job scheduler (results labeled as "CUDA") as well as the performance when rCUDA is used in combination with the modified version of Slurm (label "rCUDA"). Figure 10(a) shows total execution time for each of the workloads. Figure 10(b) depicts the averaged GPU utilization for all the 16 GPUs in the cluster. Data for GPU utilization has been gathered by polling each of the GPUs in the cluster once every second and afterwards averaging all the samples after completing workload execution. An in-house Python script based on the pyNVML library was used for polling the GPUs. In a similar way, Figure 10(c) shows total energy required for completing workload execution. Energy has been measured by polling once every second the power distribution units (PDUs) present in the cluster. Used PDU units are APC AP8653 PDUs, which provide individual energy measurements for each of the servers connected to them. After workload completion, the energy required by all servers was aggregated to provide the measurements in Figure 10(c).

DOI: 10.1002/cpe

As can be seen in Figure 10(a), workload "Set 1" presents the smallest execution time, given that it is composed of the shortest applications. 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 utilization, Figure 10(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 10(c), by 40%, 25%, and 15% for workloads "Set 1", "Set 2", and "Set 1+2", respectively. These results about reducing energy are very important given that energy consumption is an important concern in current computing facilities and will be key in future exascale systems.

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. This is the case for the GROMACS and NAMD applications, which span over 2 and 4 nodes, respectively, hindering the use of the GPUs at those nodes. On the contrary, when rCUDA is leveraged, these GPUs can be used by applications being executed in other nodes of the cluster. Notice that this remote usage of GPUs belonging to nodes with busy CPUs will be more frequent as cluster size increases because more GPUs will be blocked by non-accelerated applications (also depending on the exact workload). Another example is the execution of LAMMPS and mCUDA-MEME, which require 4 nodes with one GPU. While these applications are being executed with CUDA, those 4 nodes cannot be used by any other application from Table I: on the one hand, the other accelerated applications cannot access the GPUs in those nodes because they are busy and, on the other hand, the non-GPU applications (GROMACS and NAMD) cannot use those nodes because they require all the CPU cores and LAMMPS and mCUDA-MEME already took one core. However, when GPUs are shared among several applications, GPUs assigned to LAMMPS and mCUDA-MEME can concurrently be assigned to other applications that will run in any available CPU in the cluster, thus increasing overall throughput. This concurrent usage of the GPUs brings to a second cause for the improvements shown in Figure 10, as explained next.

The second reason for the improvements shown in Figure 10 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 original 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. This concern is also present for other applications in Table I. Moreover, if NVIDIA Tesla K40 GPUs were used instead of the NVIDIA Tesla K20 devices employed in this study, then this memory underutilization would be worse because the K40 model features 12 GB of memory. 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. In this regard, Figure 11 presents the execution times for the GPU-accelerated applications in Table I when several instances of the same application

DOI: 10.1002/cpe

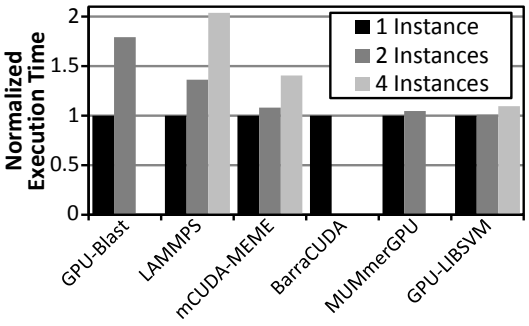


Figure 11. Normalized execution time when several concurrent instances of the same application are executed with CUDA.

are concurrently executed in a GPU[†]. Executions in Figure 11 have been manually constrained to a single node using CUDA without the use of Slurm. For some of the applications only two concurrent instances were executed due to their larger memory requirements. In a similar way, BarraCUDA does not allow the concurrent execution of other instances due to its high memory requirements. As shown, executing several instances of the same application reports a speed up for all of them: LAMMPS achieves the smallest one whereas GPU-LIBSVM obtains significant benefits. In summary, sharing a GPU among several applications reduces total execution time. This reduction makes that combining rCUDA with the modified version of Slurm results in important reductions in the time required to complete workload execution.

Another possible point of view related to sharing GPUs among applications is that all the applications sharing the GPU execute slower because they have to share the GPU cores. However, despite the slower execution of each individual application, the entire workload is completed earlier, as shown in Figure 10. 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. As a consequence, data center users increase their satisfaction about the service received.

Benefit #4: 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 a computing facility that did not include accelerators at acquisition time.

[†]It is also possible to analyze concurrent executions when the applications concurrently using the GPU are different. However, using several instances of the same application generates a higher pressure on the system because all the instances will try to synchronously perform the same operations.

DOI: 10.1002/cpe

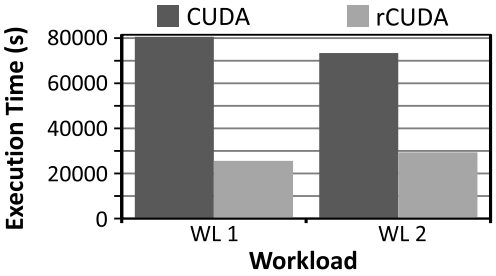
Table IV. Composition of two additional workloads

Application	Workload	
	WL 1	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

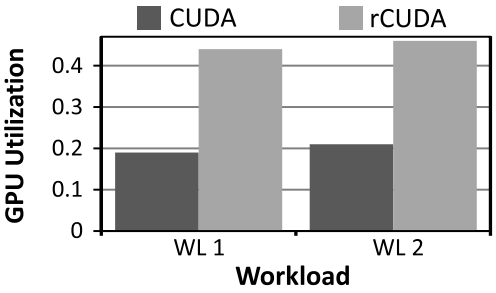
One possible solution to the concern above is acquiring some amount of servers populated with GPUs and divert the execution of accelerated applications to those nodes. The Slurm workload manager would automatically take care of dispatching the GPU-accelerated applications to the new servers. However, 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 accelerated MPI 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. The modified version of 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 testbed 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 IV: (1) "LAMMPS long 2p" and "mCUDA-MEME long 2p" consist of two single-threaded processes that are forced to

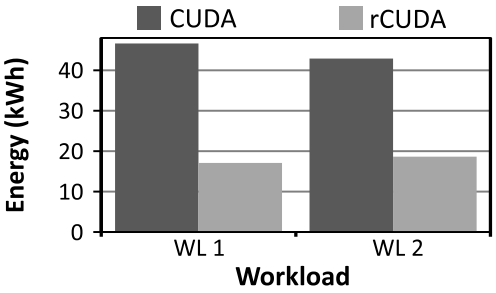
DOI: 10.1002/cpe



(a) Total execution time of the workloads.



(b) Average GPU utilization.



(c) Total energy consumed during the execution of the workloads.

Figure 12. Performance results when a server with 4 GPUs is attached to a 15-node cluster without GPUs.

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 IV 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.

Figure 12 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.

Prepared using cpeauth.cls
DOI: 10.1002/cpe

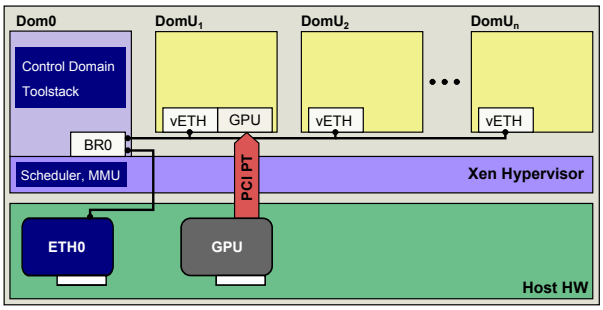


Figure 13. Typical configuration of a Xen-based system showing how the Ethernet adapter and the GPU available in the host are provided to VMs.

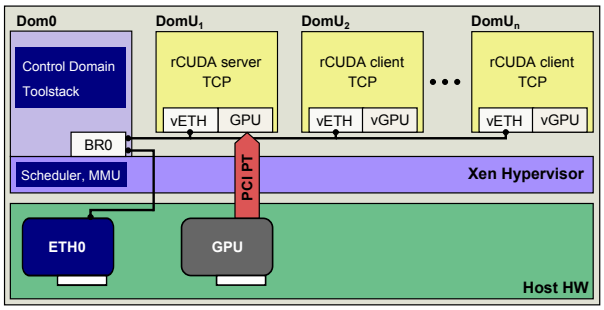
Benefit #5: Virtual Machines Can Easily Access GPUs

Providing CUDA acceleration to virtual machines (VMs) is usually accomplished by making use of the PCI passthrough technique [33] [34]. 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 VMs running at the host. Moreover, 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. Figure 13 depicts a typical VM deployment based on the Xen hypervisor, showing a computer hosting several VMs. It can be seen in the figure that the host hardware comprises, among other devices, an Ethernet network adapter and a GPU. On top of the hardware, a thin software layer (the Xen hypervisor) is installed. Above the hypervisor we can find the VMs (Dom0 and DomU_i). Notice that the Dom0 VM is a predefined VM using the Xen Linux kernel and behaves as the configuration and management interface to the hypervisor. The rest of VMs (from DomU_1 to DomU_n) are unprivileged VMs that can be provided to users. Figure 13 shows how the Ethernet adapter and the GPU are provided to VMs. On the one hand, the Ethernet adapter is owned by the Dom0 VM, which provides connectivity to the rest of VMs by using a software Ethernet switch, thus creating a virtual network among the VMs. On the other hand, the GPU is assigned to one of the VMs by making use of the PCI passthrough (PCI PT) mechanism. For other hypervisors, such as the KVM one, the overall deployment is similar although the exact configuration details differ. The reader may refer to [35] for a complete discussion on the KVM case.

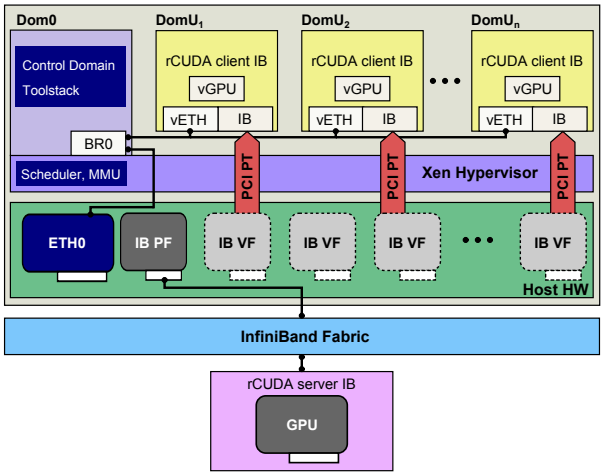
Unfortunately, the PCI passthrough approach assigns GPUs to VMs in an exclusive way and, therefore, it does not allow simultaneously sharing GPUs among the several VMs being concurrently executed at the same host. In the case of Figure 13, VM DomU_1 is the only one that may access the GPU. The rest of VMs hosted in that computer cannot make use of the accelerator until it is detached from DomU_1. Moreover, it is important to remark that at that point only one of the other VMs will be able to use the GPU. It is noteworthy the small flexibility that this configuration provides regarding the use of GPUs, given that only one of the VMs can access the GPU.

In order to address the concern about the exclusive assignment nature of the PCI passthrough mechanism, there have been several attempts, like the one proposed in [36], which dynamically changes on demand the GPUs assigned to VMs. However, these techniques present a high time overhead given that, in the best case, two seconds are required to change the assignment between GPUs and VMs. This issue constrains the use of GPUs in the cloud computing domain.

DOI: 10.1002/cpe



(a) Testbed using the virtual network within Xen.



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

Figure 14. Testbeds used in the experiments presented in this subsection, which make use of rCUDA to provide GPU access to VMs. **(a)** In a single-node testbed, VMs employ the virtual network to access the rCUDA server by means of the TCP/IP protocol stack. **(b)** When an InfiniBand fabric is available, VMs use such interconnect to access a remote rCUDA server.

With the remote GPU virtualization mechanism it is possible to concurrently assign a given GPU to several VMs, so that the applications being executed inside them can share the GPU resources [37]. Two different scenarios can be considered: one where VMs access a GPU located at the same host executing the VMs and another one where the InfiniBand fabric is already present in the cluster and therefore VMs access a GPU installed in another cluster node. Figure 14(a) depicts the first scenario whereas Figure 14(b) presents the second one.

In the first scenario, one of the VMs will have exclusive access to the GPU by making use of the PCI passthrough mechanism. This VM will grant GPU access to the other VMs by using the rCUDA middleware: the rCUDA server will be executed in the VM owning the GPU whereas the other VMs 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, VMs running the rCUDA client will have one or several virtual instances (vGPU) of the real GPU, which is physically connected to the VM DomU_1. Moreover, the VM 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 the DomU_i VMs given that NVIDIA does not provide support for the Xen Linux kernel used in the Dom0 VM.

Prepared using cpeauth.cls
DOI: 10.1002/cpe

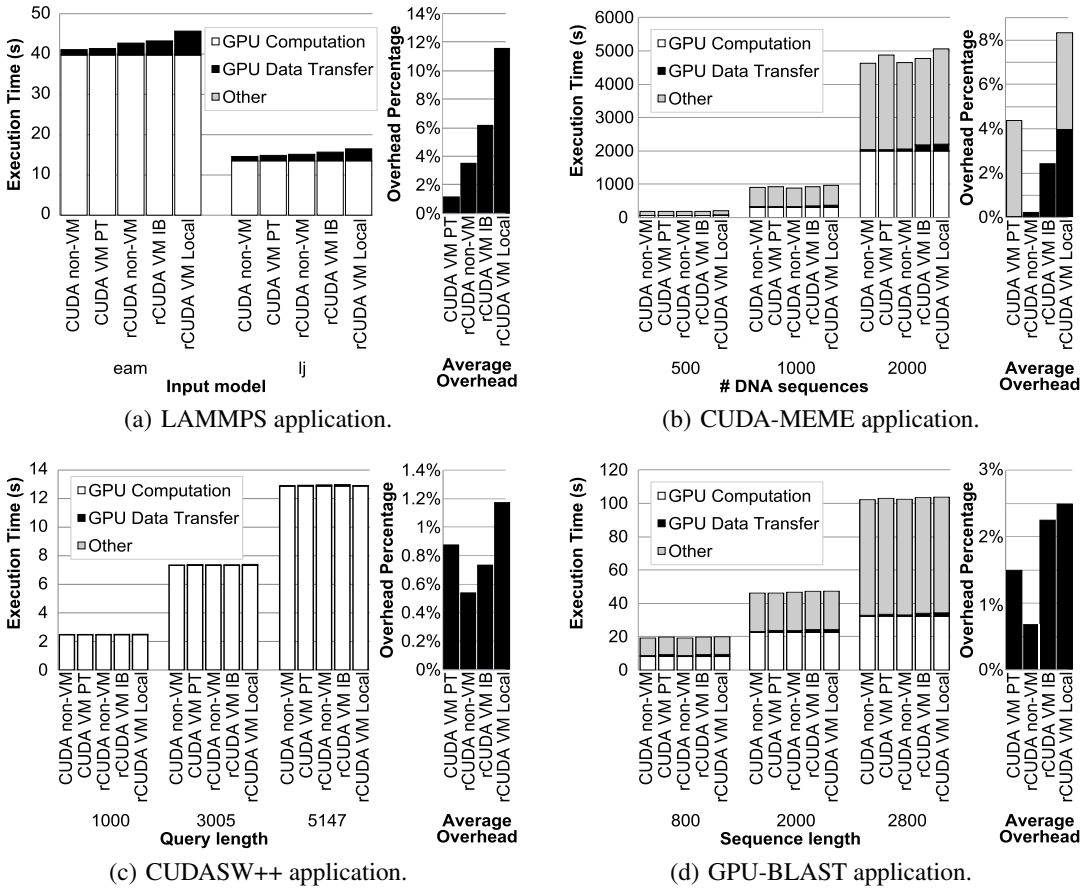
Regarding the second scenario, shown in Figure 14(b), which uses the InfiniBand fabric already present in the cluster to access a GPU in another node, the firmware in the InfiniBand adapter must be changed, according to the directions in Mellanox User's Guide [38], in order to provide several virtual 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 VM by using the PCI passthrough mechanism. Moreover, given that an InfiniBand network is available, communication between the rCUDA clients in the VMs 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 14(a), VMs 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 in this discussion 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 VMs is composed of three 1027GR-TRF Supermicro nodes as the ones mentioned before. One of them will host the Xen VMs whereas the other two nodes will not make use of VMs. In one of the native domains we will execute the rCUDA server as shown in Figure 14(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 VMs, 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, VMs were configured to have 4 cores and 12 GB of RAM memory. The applications used in this analysis are LAMMPS [25], CUDA-MEME [26], CUDASW++ [39], and GPU-BLAST [24], being all of them listed in the NVIDIA GPU-Accelerated Applications Catalog [32]. Figure 15 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 VM involved) making use of the InfiniBand network.
- When Xen VMs are involved in the tests, the performance of applications using rCUDA in the scenario depicted in Figure 14(b) is denoted by the label *"rCUDA VM IB"*.
- When using rCUDA in the scenario shown in Figure 14(a), the performance of applications will be labeled as *"rCUDA VM Local"*.

Every experiment has been performed 10 times, so that Figure 15 shows the averaged results. Furthermore, the plots in Figure 15 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

DOI: 10.1002/cpe



(a) LAMMPS application. (b) CUDA-MEME application.

(c) CUDASW++ application. (d) GPU-BLAST application.

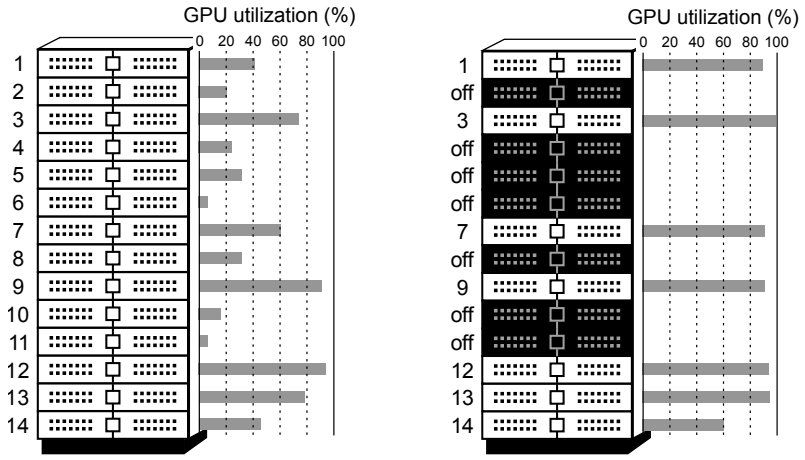
Figure 15. Execution time of several applications when executed in different local and remote scenarios. Execution time is broken down into three components: GPU computation, GPU data transfer, and Other.

spent in tasks not involving the GPU, such as CPU computations and I/O (*"Other"*). Execution times presented in Figure 15 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 15 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 VM 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 VM

Prepared using cpeauth.cls
DOI: 10.1002/cpe



(a) GPU utilization before consolidating the rCUDA servers. All the nodes in the cluster are switched on.

(b) Cluster state after consolidating the rCUDA servers. Seven nodes have been switched off.

Figure 16. Usage of rCUDA server migration in a cluster in order to consolidate GPU jobs and reduce energy.

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.

Benefit #6: GPU Migration: Towards Server Consolidation

Maximizing resource utilization is one of the goals pursued when running a data center. By maximizing the utilization of the different resources in the computing facility a larger revenue can be achieved, thus causing a faster amortization of the initial acquisition costs as well as making a larger profit afterwards.

Resource utilization in data centers evolves over time, depending on the exact workload applied at every moment. Therefore, at some point in time, the utilization of the GPUs in the cluster may be similar to that depicted in Figure 16(a). This figure shows a small cluster composed of 14 nodes, each of them including one GPU. Next to each node, the utilization of its GPU is displayed. It can be seen that some nodes present a high GPU utilization. For instance, node 9 and node 12 are using their GPUs at 90% approximately. On the contrary, some other nodes present a very low GPU utilization, such as node 6 or node 11, whose GPUs are almost not used.

In this scenario where some nodes of the cluster present a very low GPU utilization, it would be useful to gather the GPU jobs being executed in those nodes into other nodes. That is, it would be useful to consolidate the GPU jobs into a smaller number of servers, so that those nodes that become free can be switched off, thus reducing the energy consumption of the data center. Figure 16(b) depicts this consolidation of GPU jobs, where jobs generating a lower GPU utilization, such as the ones in nodes 2, 4, 5, 6, 8, 10, and 11 have been migrated to other nodes. After job migration, the nodes sourcing the movement of jobs have been switched off, thus consuming a negligible amount of energy.

Carrying out the migration of jobs using GPUs requires to migrate the process being executed at the CPU as well as the GPU part of the application. Migrating the CPU part of an application has been achieved in the past by many different frameworks. However, migrating the GPU part of

DOI: 10.1002/cpe

a CUDA application is much more complex because of two reasons: (1) kernels being executed in the GPU run asynchronously with the CPU and therefore when migration is triggered, the kernel in the GPU could be under execution, and (2) the GPU memory allocated to the application, which is not tracked by the operating system, must be copied to the destination GPU.

Addressing (1) above is easy. Once migration has been triggered, the migration framework could just execute a synchronization call (such as the `cudaDeviceSynchronize` function) in order to wait for the completion of the kernels being executed in the GPU. However, addressing (2) is not so easy given that the map of memory used by the application at the GPU is not included in the system tables stored by the operating system for this process. Therefore, unless some additional support is implemented, it is not possible to retrieve which are the memory regions used by the application at the GPU.

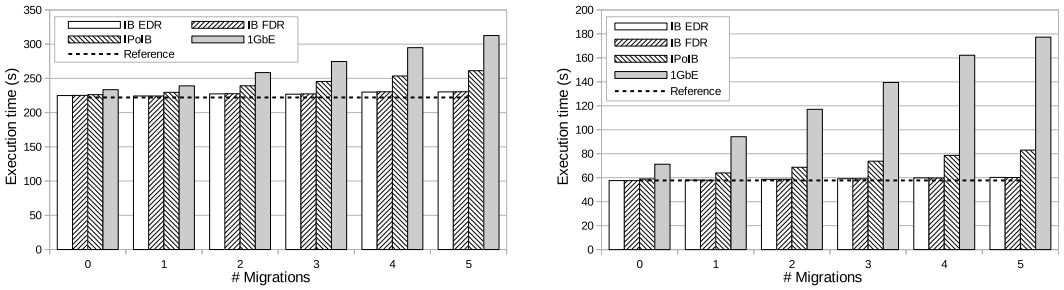
In order to implement this additional support, the GPU memory management calls executed by the application (such as the `cudaMalloc` and `cudaFree` functions) could be intercepted. By intercepting them, it is possible to gather the required information for retrieving the memory regions used at the GPU. This is the usual approach followed by other frameworks that provide support for migrating applications that make use of GPUs, such as CheCUDA[‡] [40].

Nevertheless, although obtaining the required information about memory regions used at the GPU makes it possible to migrate CUDA applications from one node to another, it is still possible to make this migration even more effective when the remote GPU virtualization mechanism is being leveraged. In this regard, when remote GPU virtualization is not used, migrating a CUDA application to another cluster node means that the destination node has (1) enough CPU cores available for the application being migrated, (2) enough main RAM memory for hosting the application data, and (3) enough GPU memory to hold the data stored at the source GPU. Finding a target node within the cluster that complies with these three requirements may not be difficult. However, when the remote GPU virtualization technique is used, another approach could be followed in order to make the migration process more efficient. This new approach is based on the fact that the remote GPU virtualization mechanism detaches GPUs from nodes, from a logical point of view, and therefore the CPU and GPU parts of the application may be migrated independently from each other to different destination nodes. In this manner, it would not be necessary that the three requirements described above are satisfied by a given node but these requirements could be split into two sets: finding a node that satisfies requirements (1) and (2) and finding another node that complies with requirement (3). The first set of requirements is intended for the migration of the CPU part of the application whereas the second set is devoted to the GPU part.

The new proposed migration approach would make it easier to find better node candidates than when the three requirements must be satisfied by the same node. Furthermore, given that the GPU part of the application was probably running in a node different from the CPU part, it would even be possible that only one of the parts (CPU or GPU) needs to be migrated. In this context, selecting whether to migrate the CPU part or the GPU part of an application would be based on the current cluster status and optimization policies. In any case, by using the remote GPU virtualization mechanism it would be possible to consolidate both CPU and GPU servers at the same time that (1)

[‡]The CheCUDA framework does not provide remote GPU virtualization features as rCUDA does. The CheCUDA framework is only intended for making it possible to checkpoint applications that use GPUs.

Prepared using cpeauth.cls
DOI: 10.1002/cpe



(a) Execution time of a synthetic application. (b) Execution time of the GPU-BLAST application.

Figure 17. Evolution of application execution time when rCUDA is leveraged for executing the applications using a remote GPU and an increasing amount of migrations are forced during application execution.

migration is carried out faster because the amount of data to be moved is probably smaller (likely only one of the CPU or GPU parts is migrated), and (2) given that the target node only has to comply with a subset of the conditions above, finding a better destination candidate should be much easier than in the original approach.

The rCUDA middleware supports the migration of the GPU part of CUDA applications. In this regard, with the rCUDA framework it is possible to select one of the multiple jobs using a given GPU and move it to another GPU in the same or in other node of the cluster. This process is transparent to the application using the GPU, which is not aware of the migration.

Figure 17 shows the evolution of the execution time of two applications when up to 5 migrations are forced during their execution. Only the GPU part of the applications is migrated. Figure 17(a) depicts such evolution for a synthetic application whereas Figure 17(b) shows the evolution for the GPU-BLAST [24] application. Several nodes are used in these experiments. The characteristics of these nodes are the same as the ones mentioned before: two Xeon E5-2620 v2 sockets with one NVIDIA Tesla K20 GPU and one FDR InfiniBand adapter. EDR InfiniBand has also been considered. Given that source and destination GPUs are located at different cluster nodes, several interconnects and communication protocols are considered: RDMA over EDR InfiniBand, RDMA over FDR InfiniBand, TCP/IP over InfiniBand, and TCP/IP over 1 Gb Ethernet. On the other hand, label “Reference” in Figure 17 refers to the execution time of the applications when CUDA is used (rCUDA is not used for executing the applications in this case).

The synthetic application used in Figure 17(a) performs the multiplication of a vector by a scalar. To that end, it initially allocates GPU memory for 1000 randomly-sized arrays and fills them by copying data from host memory to GPU memory. Then the application launches the necessary kernels to apply the multiplication to the 1000 vectors and finally results are copied back from GPU to host memory and GPU memory is then released. The aggregated volume of memory used at the GPU for the 1000 arrays is 700 MB. When migration is triggered, the rCUDA framework performs 1000 allocations of GPU memory at the destination GPU, performs 1000 memory copies between source and destination GPUs, and then carries out 1000 memory releases at the source GPU, which is freed and thus no longer related to the execution of the application. It can be seen in the figure that, as expected, the use of RDMA over InfiniBand provides the smallest migration overhead given the superior features of this communication mechanism.

Figure 17(b) shows a similar study for the GPU-BLAST application. In this case, the application holds 1300 MB of data in 9 regions of GPU memory. Therefore, every time the application is

DOI: 10.1002/cpe

migrated, the rCUDA framework must allocate 9 memory regions in the destination GPU, must copy the 9 regions from source to destination GPUs, and finally must release the 9 regions at the source GPU. It can be seen that migration overhead is negligible when RDMA is used.

5. 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.

Other benefits of this novel virtualization mechanism have also been explored. Perhaps the most significant one may be GPU migration. In this manner, we have shown that migrating GPU jobs from one GPU server to another is quite complex to perform in an efficient way when the remote GPU virtualization mechanism is not being used. On the contrary, GPU job migration 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.

REFERENCES

1. NVIDIA. *CUDA C Programming Guide 7.5* 2016.
2. Wu H, Diamos G, Sheard T, Aref M, Baxter S, Garland M, Yalamanchili S. Red Fox: An Execution Environment for Relational Query Processing on GPUs. *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '14, ACM, 2014; 44:44–44:54.
3. Playne DP, Hawick KA. Data parallel three-dimensional cahn-hilliard field equation simulation on GPUs with CUDA. *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications, PDPTA*, 2009.
4. Yamazaki I, Dong T, Solc R, Tomov S, Dongarra J, Schulthess T. Tridiagonalization of a dense symmetric matrix on multiple GPUs and its application to symmetric eigenvalue problems. *Concurrency and Computation: Practice and Experience* 2014; **26**(16):2652–2666.
5. Yuancheng Luo D. Canny edge detection on NVIDIA CUDA. *Computer Vision and Pattern Recognition Workshops, 2008. CVPRW '08. IEEE Computer Society Conference on*, IEEE, 2008; 1–8.
6. Surkov V. Parallel option pricing with fourier space time-stepping method on graphics processing units. *Parallel Computing* 2010; **36**(7):372 – 380.
7. Agarwal PK, Hampton S, Poznanovic J, Ramanthan A, Alam SR, Crozier PS. Performance modeling of microsecond scale biological molecular dynamics simulations on heterogeneous architectures. *Concurrency and Computation: Practice and Experience* 2013; **25**(10):1356–1375.
8. Yoo AB, Jette MA, Grondona M. *SLURM: Simple Linux Utility for Resource Management*. Springer Berlin Heidelberg: Berlin, Heidelberg, 2003; 44–60.
9. Federico S, Javier P, Iserte S, Reaño C. Remote GPU virtualization: Is it useful? IEEE Computer Society, 2016; 41–48.
10. Liang TY, Chang YW. GridCuda: A Grid-Enabled CUDA Programming Toolkit. *Advanced Information Networking and Applications (WAINA), 2011 IEEE Workshops of International Conference on*, IEEE, 2011; 141–146.

DOI: 10.1002/cpe

11. Oikawa M, Kawai A, Nomura K, Yasuoka K, Yoshikawa K, Narumi T. DS-CUDA: A Middleware to Use Many GPUs in the Cloud Environment. *Proceedings of the 2012 SC Companion: High Performance Computing, Networking Storage and Analysis*, SCC '12, IEEE Computer Society: Washington, DC, USA, 2012; 1207–1214.
12. Giunta G, Montella R, Agrillo G, Coviello G. *A GPGPU Transparent Virtualization Component for High Performance Computing Clouds*. Springer, 2010.
13. Shi L, Chen H, Sun J. vCUDA: GPU accelerated high performance computing in virtual machines. *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, IEEE, 2009; 1–11.
14. Gupta V, Gavrilovska A, Schwan K, Kharche H, Tolia N, Talwar V, Ranganathan P. GViM: GPU-accelerated virtual machines. *Proceedings of the 3rd ACM Workshop on System-level Virtualization for High Performance Computing*, ACM, 2009; 17–24.
15. Peña AJ, Reaño C, Silla F, Mayo R, Quintana-Orti ES, Duato J. A Complete and Efficient CUDA-Sharing Solution for HPC Clusters. *Parallel Computing* 12/2014 2014; **40**:574–588.
16. CUDA API Reference Manual 7.5. https://developer.nvidia.com/cuda-toolkit 2016.
17. Merritt AM, Gupta V, Verma A, Gavrilovska A, Schwan K. Shadowfax: Scaling in Heterogeneous Cluster Systems via GPGPU Assemblies. *Proceedings of the 5th International Workshop on Virtualization Technologies in Distributed Computing*, VTDC '11, ACM: New York, NY, USA, 2011; 3–10.
18. Shadowfax II - scalable implementation of GPGPU assemblies. http://keeneland.gatech.edu/software/keeneland/kidron. Accessed: 2015-05-20.
19. NVIDIA. *The NVIDIA GPU Computing SDK Version 5.5* 2013.
20. iperf3: A TCP, UDP, and SCTP network bandwidth measurement tool. https://github.com/esnet/iperf 2016.
21. Reaño C, Silla F, Shainer G, Schultz S. Local and Remote GPUs Perform Similar with EDR 100G InfiniBand. *Proceedings of the Industrial Track of the 16th International Middleware Conference*, Middleware Industry '15, 2015.
22. Reaño C, Silla F, Castello A, Peña AJ, Mayo R, Quintana-Orti ES, Duato J. Improving the user experience of the rCUDA remote GPU virtualization framework. *Concurrency and Computation: Practice and Experience* 2015; **27**(14):3746–3770.
23. Iserte S, Castelló A, Mayo R, Quintana-Orti ES, Silla F, Duato J, Reaño C, Prades J. Slurm Support for Remote GPU Virtualization: Implementation and Performance Study. *Computer Architecture and High Performance Computing (SBAC-PAD), 2014 IEEE 26th International Symposium on*, 2014; 318–325.
24. Vouzis PD, Sahinidis NV. GPU-BLAST: Using graphics processors to accelerate protein sequence alignment. *Bioinformatics* 2010; .
25. Brown WM, Kohlmeyer A, Plimpton SJ, Tharrington AN. Implementing molecular dynamics on hybrid high performance computers: Particle-particle particle-mesh. *Computer Physics Communications* 2012; **183**(3):449 – 459.
26. Liu Y, Schmidt B, Liu W, Maskell DL. CUDA-MEME: Accelerating motif discovery in biological sequences using CUDA-enabled graphics processing units. *Pattern Recognition Letters* 2010; **31**(14):2170 – 2177.
27. Pronk S, Pll S, Schulz R, Larsson P, Bjelkmar P, Apostolov R, Shirts MR, Smith JC, Kasson PM, van der Spoel D, *et al.*. Gromacs 4.5: a high-throughput and highly parallel open source molecular simulation toolkit. *Bioinformatics* 2013; **29**(7):845–854.
28. Klus P, Lam S, Lyberg D, Cheung M, Pullan G, McFarlane I, Yeo G, Lam B. Barracuda - a fast short read sequence aligner using graphics processing units. *BMC Research Notes* 2012; **5**(27).
29. Kurtz S, Phillippy A, Delcher A, Smoot M, Shumway M, Antonescu C, Salzberg S. Versatile and open software for comparing large genomes. *Genome Biology* 2004; **5**(2).
30. Chang CC, Lin CJ. Libsvm: A library for support vector machines. *ACM Trans. Intell. Syst. Technol.* May 2011; **2**(3):27:1–27:27.
31. Phillips JC, Braun R, Wang W, Gumbart J, Tajkhorshid E, Villa E, Chipot C, Skeel RD, Kal L, Schulten K. Scalable molecular dynamics with namd. *Journal of Computational Chemistry* 2005; **26**(16):1781–1802.
32. NVIDIA Popular GPU-Accelerated Applications Catalog. http://www.nvidia.es/content/tesla/pdf/gpu-accelerated-applications-for-hpc.pdf 2016.
33. Walters JP, Younge AJ, Kang DI, Yao KT, Kang M, Crago SP, Fox GC. GPU-Passthrough Performance: A Comparison of KVM, Xen, VMWare ESXi, and LXC for CUDA and OpenCL Applications. *7th IEEE International Conference on Cloud Computing (CLOUD 2014)*, 2014.
34. Yang CT, Wang HY, Ou WS, Liu YT, Hsu CH. On implementation of GPU virtualization using PCI pass-through. *Cloud Computing Technology and Science (CloudCom), 2012 IEEE 4th International Conference on*, IEEE, 2012; 711–716.

DOI: 10.1002/cpe

35. Pérez F, Reaño C, Silla F. Providing CUDA Acceleration to KVM Virtual Machines in InfiniBand Clusters with rCUDA. *Proceedings of the International Conference on Distributed Applications and Interoperable Systems*, 2016.
36. Jo H, Jeong J, Lee M, Choi DH. Exploiting GPUs in Virtual Machine for BioCloud. *BioMed research international* 2013; **2013**.
37. Prades J, Reaño C, Silla F. CUDA Acceleration for Xen Virtual Machines in Infiniband Clusters with rCUDA. *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '16, 2016.
38. Mellanox. Mellanox OFED for Linux User Manual 2015.
39. Liu Y, Wirawan A, Schmidt B. CUDASW++ 3.0: accelerating Smith-Waterman protein database search by coupling CPU and GPU SIMD instructions. *BMC Bioinformatics* 2013; **14**(1):1–10.
40. Takizawa H, Sato K, Komatsu K, Kobayashi H. CheCUDA: A Checkpoint/Restart Tool for CUDA Applications. *Proceedings of the 2009 International Conference on Parallel and Distributed Computing, Applications and Technologies*, 2009.

DOI: 10.1002/cpe