Improving the Management Efficiency of GPU Workloads in Data Centers through GPU Virtualization

Journal:	*Concurrency and Computation: Practice and Experience*
Manuscript ID	Draft
Editor Selection:	Special Issue Submission
Wiley - Manuscript type:	Special Issue Paper
Date Submitted by the Author:	n/a
Complete List of Authors:	Iserte, Sergio; Universitat Jaume I Prades, Javier; Technical University of Valencia, Department of Computer Engineering Silla, Federico; Technical University of Valencia, Department of Computer Engineering Reaño, Carlos; Queen's University Belfast, School of Electronics, Electrical Engineering and Computer Science
Keywords:	CUDA, HPC, virtualization, InfiniBand, rCUDA, Slurm

Received: Added at production | Revised: Added at production | Accepted: Added at production

DOI: xxx/xxxx

ARTICLE TYPE

Improving the Management Efficiency of GPU Workloads in Data Centers through GPU Virtualization

Sergio Iserte*[1] | Javier Prades[2] | Carlos Reaño[3] | Federico Silla[2]

[1]Dpto. de Ingeniería y Ciencia de los Computadores, Universitat Jaume I, Castellón de la Plana, Spain
[2]Departamento de Informática de Sistemas y Computadores, Universitat Politècnica de València, València, Spain
[3]School of Electronics, Electrical Engineering and Computer Science, Queen's University Belfast, Belfast, UK

Correspondence
*Sergio Iserte, Dpto. de Ingeniería y Ciencia de los Computadores, Universitat Jaume I, Castellón de la Plana, Spain. Email: siserte@uji.es

Abstract

Graphics Processing Units (GPUs) are currently used in data centers to reduce the execution time of compute-intensive applications. However, the use of GPUs presents several side effects, such as increased acquisition costs as well as larger space requirements. Furthermore, GPUs require a non-negligible amount of energy even while idle. Additionally, GPU utilization is usually low for most applications.
In a similar way to the use of virtual machines, using virtual GPUs may address the concerns associated with the use of these devices. In this regard, the remote GPU virtualization mechanism could be leveraged 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.
However, in the same way as job schedulers map GPU resources to applications, virtual GPUs should also be scheduled before job execution. Nevertheless, current job schedulers are not able to deal with virtual GPUs. In this paper we analyze the performance attained by a cluster using the rCUDA middleware and a modified version of the Slurm scheduler, which is now able to assign remote GPUs to jobs. Results show that cluster throughput, measured as jobs completed per time unit, is doubled at the same time that total energy consumption is reduced up to 40%. GPU utilization is also increased.

KEYWORDS:
CUDA; HPC; virtualization; InfiniBand; data centers; Slurm; rCUDA; GPU

1 | INTRODUCTION

The use of GPUs (Graphics Processing Units) has become a widely accepted way of reducing the execution time of applications. The massive parallel capabilities of these devices are leveraged to accelerate specific parts of applications. Programmers exploit GPU resources by off-loading the computationally intensive parts of applications to them. In this regard, 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] or OpenCL (Open Computing Language)[2] noticeably ease this task. The net result is that these accelerators are used to significantly reduce the execution time of applications from domains as different as data analysis (Big Data)[3], chemical physics[4], computational algebra[5], image analysis[6], finance[7], and biology[8] to name only a few.

Many current data centers leveraging GPUs typically include one or more of these accelerators in every node of the cluster. Figure 1 shows an example of such a deployment, composed of n nodes, each of them containing two CPU sockets and one GPU. The example in Figure 1 might be a

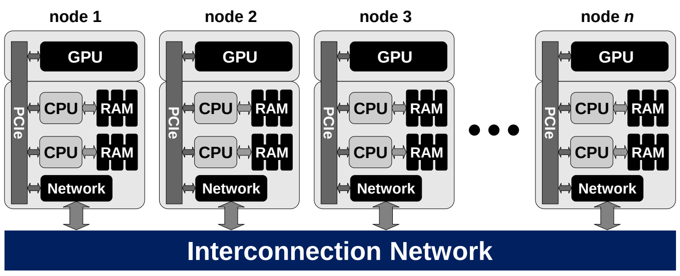


FIGURE 1 Example of a GPU-accelerated cluster.

representation of a typical cluster configuration composed of n SYS1027-TRF Supermicro servers interconnected by an FDR InfiniBand network. Each of the servers in Figure 1 may include, for instance, two Xeon E5-2620 v2 processors and one NVIDIA Tesla K20 GPU. However, the use of GPUs in such a deployment is not exempt from side effects. For instance, let consider the execution of an 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 busy. In other words, the execution of non-accelerated applications in some nodes may prevent other applications from making use of the accelerators installed in those nodes, forcing those GPUs to remain idle for some periods of time. The consequence will be that the initial hardware investment will require more time to be amortized whereas some amount of energy is wasted because idle GPUs still consume some power[1].

Another important concern associated with the use of GPUs in clusters is related to the way that workload managers such as Slurm[9] perform the accounting of resources in a cluster. These workload managers use a fine granularity for resources such as CPUs or memory, but not for GPUs. For instance, workload managers can assign CPU resources in a per-core basis, thus being able to manage a shared usage of the CPU sockets present in a server among several applications. This per-core assignment increases overall CPU utilization, speeding up the amortization of the initial investment in hardware. In the case of memory, workload managers 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, workload managers use a per-GPU granularity. In this regard, GPUs are assigned to applications in an exclusive way. Therefore, a given GPU cannot be shared among several applications even in the case that this GPU has enough resources to allow the concurrent execution of those applications. This per-GPU assignment causes that, in general, overall GPU utilization is low because few applications present enough computational concurrency to keep GPUs in use all the time.

In order to address these concerns, 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. This allows that decoupled (or virtual) GPUs are concurrently shared by all the nodes of the computing facility. Furthermore, given that the remote GPU virtualization mechanism allows GPUs to be transparently used from any node in the cluster, it is possible to create cluster configurations where not all the nodes in the cluster own a GPU. This feature would not only reduce the costs associated with the acquisition and later use of GPUs, but would also increase the overall utilization of such accelerators because workload managers would assign the acquired GPUs concurrently to several applications as far as GPUs present enough resources for all of them. In general, remote GPU virtualization presents many benefits, as shown in in[10].

Notice, however, that workload managers need to be enhanced in order to manage virtual GPUs. This enhancement would basically consist in replacing the current per-GPU granularity by a finer granularity that should allow GPUs to be concurrently shared among several applications. Once this enhancement is performed, it is expected that overall cluster performance is increased because the concerns previously mentioned would be reduced. It would also be possible to consider attaching a server owning several GPUs to a cluster that does not include GPUs. In this way, upgrading a non-accelerated cluster so that it includes GPUs would become an easy and inexpensive process.

In this paper we present a study of the performance of a cluster that makes use of the remote GPU virtualization mechanism along with an enhanced workload manager able to assign virtual GPUs to waiting jobs. To that end, we have made use of the rCUDA[11] remote GPU virtualization middleware along with a modified version[12] of the Slurm workload manager, which is now able to dispatch GPU-accelerated applications to nodes not owning GPUs while assigning such applications as many GPUs from other nodes as they require. A preliminary version of this work was already presented in[13]. Notice, however, that in this paper we further investigate the performance results reported by the integration of rCUDA and Slurm.

[1]Although GPUs present a favorable performance/power ratio while being used, they still require non-negligible amounts of energy while idle. For instance, idle NVIDIA Tesla K20 and K40 GPUs require, respectively, 25 and 20 watts. On the contrary, new NVIDIA Tesla K80 GPUs have significantly reduced the amount of energy consumed in the idle state although they still present a non-negligible power consumption while being active without performing computations.

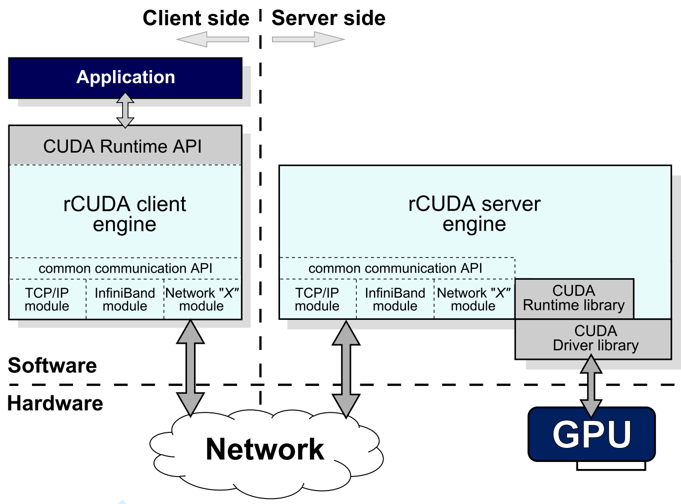


FIGURE 2 Architecture of the rCUDA remote GPU virtualization middleware.

Paper layout is the following: Section 2 presents the basic required background on the rCUDA remote GPU virtualization framework and on the modified Slurm workload manager to understand the rest of the paper. Later, Section 3 presents a thorough performance study of a cluster using rCUDA and the modified version of Slurm. Section 4 presents a review of the state-of-the-art on the remote GPU virtualization and workload manager areas. Finally, Section 5 concludes the paper.

2 | BACKGROUND ON rCUDA AND SLURM

The purpose of this section is twofold. First, this section presents the required background on the rCUDA remote GPU virtualization framework so that the reader can understand the performance evaluation presented in Section 3. Notice that this section is focused on the rCUDA middleware. A complete discussion on the remote GPU virtualization mechanism as well as a description of the available frameworks can be found in Section 4. Second, this section also presents the required background on the Slurm workload manager in order to follow Section 3. A summary of the main changes applied to Slurm so that virtual GPUs provided by rCUDA can be managed by Slurm is also provided. A detailed description of the applied changes can be found in [12]. Additionally, a complete discussion about workload managers can be found in Section 4.

2.1 | Background on rCUDA

Frameworks such as CUDA[1] assist programmers in using GPUs for general-purpose computing. In addition, several remote GPU virtualization solutions exist for this framework, such as GridCuda[14], DS-CUDA[15], gVirtuS[16], vCUDA[17], GViM[18], and rCUDA[11]. Current virtualization frameworks provide different features. This section focuses on describing the main characteristics of the rCUDA framework. A complete discussion on the state-of-the-art on remote GPU virtualization frameworks can be found in Section 4.

Figure 2 depicts the architecture of the rCUDA framework, which is similar to that of most of these virtualization solutions, as shown in Figure 11. The rCUDA framework follows 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. The client middleware offers the same application programming interface (API) as does the NVIDIA CUDA Runtime[19] and Driver[20] APIs (except for graphics functions). It is binary compatible with CUDA 9.0 and also provides support for the libraries included within CUDA (cuBLAS, cuFFT, cuDNN, etc). Every time the accelerated application performs a CUDA call, the client side of rCUDA receives the request from the 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.

The rCUDA framework supports several underlying interconnection technologies by making use of network-specific communication modules. Currently, three communication modules are available: TCP/IP, InfiniBand, and RoCE. The former can be used in any TCP/IP compatible network whereas the two latter make use of the high performance InfiniBand Verbs API available in the InfiniBand and RoCE network adapters. In order to maximize performance, rCUDA has been perfectly tuned to the InfiniBand Verbs API[21]. Furthermore, as shown in [22], rCUDA outperforms the rest of available remote GPU virtualization solutions. Additionally, rCUDA has been applied to different areas with very good results[23] [24] [25] [26]. rCUDA can also be used in cloud computing scenarios[27] [28]. For these reasons, we use this middleware in our study.

Using rCUDA requires to set three environment variables prior to application execution: `RCUDA_DEVICE_COUNT`, `RCUDA_DEVICE_j`, and `RCUDA_NETWORK`. The first variable indicates the amount of remote virtual GPUs accessible to the application. For example, if two remote 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` remote 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 to inform the rCUDA client about the location of the virtual GPUs assigned to the application. In case the GPU server owns several accelerators, it is possible to declare which of those accelerators is assigned to the application. To that end, the `RCUDA_DEVICE_j` variable should include the GPU identifier inside the server. For instance, the commands "`export RCUDA_DEVICE_0=192.168.0.3:2`" and "`export RCUDA_DEVICE_1=192.168.0.4:0`" would assigne the application GPUs 2 and 0, respectively, of servers with IP addresses "`192.168.0.3`" and "`192.168.0.4`". Finally, the `RCUDA_NETWORK` environment variable sets the communication module to be used during the execution of the application. For instance, the command "`export RCUDA_NETWORK=IB`" should be used in order to leverage the InfiniBand Verbs API.

2.2 | Background on Slurm

Current workload managers do not support the virtual GPUs provided by frameworks such as rCUDA due to their novelty. In this regard, workload managers are only able to deal with real GPUs. Therefore, when a job includes within its computing requirements one or more GPUs per node, current workload managers will try to map that job to nodes owning the requested amount of real GPUs. Nevertheless, it is possible to enhance current workload managers so that they become aware of virtual GPUs. This would make the assignment of GPUs more flexible because any available GPU across the cluster might be assigned to a job, regardless of the exact GPU and job locations. In this way, by increasing the awareness of workload managers, they would provide support for virtual GPUs, hence allowing the scheduling process to enjoy a larger degree of freedom.

In this paper we make use of an extended version of the Slurm workload manager[12] which supports the rCUDA middleware. Selecting Slurm among the many available job schedulers was based on its open-source nature, at the same time that Slurm has demonstrated to be portable and interconnect independent, thus making it suitable for many different cluster architectures. A complete discussion on workload managers can be found in Section 4.

Next we present the six main modifications to the Slurm workload manager in order to make it virtual-GPU aware (the reader may refer to[12] for a thorough description of the modifications done to Slurm):

1. The GRes module, which manages the allocation and deallocation of consumable generic resources such as GPUs, has been augmented so that all GPUs in the cluster can be accessed from all the nodes. Additionally, GPUs in the cluster can be shared among different jobs.
2. Two new plug-ins have been implemented. On the one hand, the new GRes plug-in "`gres/rgpu`" is responsible for the declaration of the remote GPUs as a generic resource, which will be referred to as rGPU. On the other hand, the select plug-in "`select/cons_rgpu`" will perform tasks related to selection and scheduling of the new rGPUs.
3. Several internal data structures within Slurm have been modified with new attributes in order to maintain the required information about the new rGPUs.
4. The RPC packages within Slurm have been augmented with additional fields intended to carry the rGPU information required by Slurm.
5. The job submission commands within Slurm have been modified so that they accept the new parameters related to the use of the new rGPU resources.
6. In order to connect the scheduling process with the rCUDA middleware, the Slurm scheduler has to set the three rCUDA environment variables mentioned in the previous section.

In addition to the previous modifications, some policy must be followed during the scheduling process in order to select one GPU or another from the many GPUs available in the cluster. In this work we have followed a round-robin approach while giving a higher priority to those rGPUs located in the same node that will execute the application. Other selection policies were also considered although performance results did not vary significantly.

Once these changes are implemented, Slurm users are able to submit jobs to the system queues in three different modes:

1. **CUDA**: no change is required to the original way of launching jobs.

2. **rCUDA shared**: the job will use the new rGPU resources, which will be shared with other jobs. The required amount of GPU memory should be specified as a parameter to the job submission command. For instance, "`srun -rcuda-mode=shar -gres=rgpu:4:100M job.sh`" will submit a job named "`job.sh`" requesting 4 virtual GPUs having each of them at least 100 MB of available memory. These GPUs may be shared with other jobs.

3. **rCUDA exclusive**: the job will use the new rGPU resources but will not share them with other jobs. For instance, "`srun -rcuda-mode=excl -gres=rgpu:4 job.sh`" will submit a job named "`job.sh`" requesting the exclusive use of 4 virtual GPUs.

3 | PERFORMANCE ANALYSIS

In this section we study the impact that using the remote GPU virtualization mechanism in combination with Slurm has on the performance of a data center. To that end, we have executed several workloads in a cluster by submitting a series of job requests to the Slurm queues. After job submission we have measured several parameters such as total execution time of the workloads, energy required to execute them, GPU utilization, etc. 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, the modified version of Slurm has been used 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 the throughput of a cluster using rCUDA.

In the following subsections we present the performance analysis. In this regard, we first present the cluster configuration and the workloads used in the experiments. After that, we analyze the performance of combining rCUDA with Slurm in different cluster configurations.

3.1 | Cluster Testbed

The testbed used in this study is comprised of 1027GR-TRF Supermicro servers. Each of the servers includes two Intel Xeon E5-2620 v2 processors (six cores with Ivy Bridge architecture) operating at 2.1 GHz and 32 GB of DDR3 SDRAM memory at 1600 MHz. They also have a Mellanox ConnectX-3 VPI single-port FDR InfiniBand adapter connected to a Mellanox Switch SX6025 (FDR InfiniBand compatible) to exchange data at a maximum rate of 56 Gb/s. Furthermore, an NVIDIA Tesla K20 GPU is installed in each node.

In order to analyze how the obtained performance results depend on cluster size, we have considered three cluster sizes for the experiments: 4 nodes, 8 nodes, and 16 nodes. Obviously, the cluster configuration composed of 16 nodes is the most representative one (although it is still smaller than most data centers). However, these three sizes will allow us to study the different trends of the performance metrics. In all the three cluster sizes mentioned, one additional node has been leveraged. This additional node, which does not include a GPU, will be used as the Slurm management node and will execute the central Slurm daemon responsible for scheduling jobs (the `slurmctld` process).

Regarding the software configuration of the cluster, Linux CentOS 6.4 was used along with Mellanox OFED 2.4-1.0.4 (InfiniBand drivers and administrative tools). Slurm version 14.11.0 was used. The modifications described in Section 2.2 were applied to Slurm. It was configured to use the `backfill` scheduling policy. In this way, jobs can overtake others. Finally, version 2.0b of the MVAPICH2 implementation of MPI, specifically tuned for InfiniBand, was used for those applications requiring the MPI library.

3.2 | Workloads

Several workloads have been considered in order to provide a more representative range of results. The workloads are composed of applications (see Table 1) selected because of their different characteristics from the list of NVIDIA's Popular GPU-Accelerated Applications Catalog[29].

- GPU-BLAST[30] has been designed to accelerate the gapped and ungapped protein sequence alignment algorithms of the NCBI-BLAST (http://www.ncbi.nlm.nih.gov) implementation using GPUs.

- LAMMPS[31] is a classic molecular dynamics simulator that can be used to model atoms or, more generically, as a parallel particle simulator at the atomic, mesoscopic, or continuum scale.

- mCUDA-MEME[32] is a parallel CUDA implementation of the MEME algorithm, used for discovering motifs in a group of related DNA or protein sequences.

- GROMACS[33] is a molecular dynamics simulator, as LAMMPS. Although this package can use GPUs, in this study we will use a non-accelerated version in order to achieve a higher degree of heterogeneity in our experimental workloads.

TABLE 1 Applications used in this study. Configuration details for each application

Application	Configuration	Is an MPI application?	Total amount of GPUs	Execution time (s)	Memory per GPU
GPU-Blast	1 6-thread process in 1 node	no	1	21	1599 MB
LAMMPS	4 1-thread processes in 4 nodes	yes	4	15	876 MB
mCUDA-MEME	4 1-thread processes in 4 nodes	yes	4	165	151 MB
GROMACS	2 12-thread processes in 2 nodes	yes	2	167	-
BarraCUDA	1 1-thread process in 1 node	no	1	763	3319 MB
MUMmerGPU	1 1-thread process in 1 node	no	1	353	2104 MB
GPU-LIBSVM	1 1-thread process in 1 node	no	1	343	145 MB
NAMD	4 12-thread processes in 4 nodes	yes	4	241	-

- BarraCUDA[34] is a sequence mapping software that uses GPUs to accelerate the inexact alignment of short sequence reads to a particular location on a reference genome.
- MUMmerGPU is the GPU implementation of MUMmer[35], which is a system for rapidly aligning entire genomes, whether in complete or draft form.
- GPU-LIBSVM is a modification of the original LIBSVM[36] algorithm that exploits the CUDA framework to significantly reduce processing time. LIBSVM is an integrated software intended for vector classification, regression and distribution estimation.
- NAMD[37] is a parallel molecular dynamics simulator designed for high-performance simulation of large biomolecular systems. Although this application is able to use GPUs, the version of NAMD used in our study does not make use of them and therefore is intended to contribute to a higher degree of heterogeneity of the workloads.

Table 1 provides additional information about the applications used in this work, 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 in the table 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, the GPU-Blast application uses a single process composed of 6 threads. During execution, each of these threads will use a different CPU core, although all of them will make use of the same GPU. 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. Notice that we have configured the execution of the considered applications as shown in Table 1 according to a previous scalability analysis (not shown) which was carried out in advance to find out the best configuration for each application.

Table 1 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 1 also shows the GPU memory required by each application. For those applications composed of several processes, the amount of GPU memory depicted in Table 1 refers to the individual needs of each 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.

In summary, the eight applications used 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 1 could be considered from two different point of views. 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 specific 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 each application and the time that those resources are kept busy. From this second perspective, the set of applications in Table 1 becomes even more representative.

Table 2 displays the Slurm parameters used for launching each of the applications with CUDA and rCUDA. In the first case, CUDA will be used (column labeled "Launch with CUDA"). In the second case, remote GPUs can either be used in an exclusive or shared way. In the first approach, the column labeled as "Launch with rCUDA exclusive" shows that the `rcuda-mode` parameter is set to `excl` and no GPU memory is declared. In the second approach, the column labeled as "Launch with rCUDA shared" shows that the amount of memory required at each GPU must be specified

TABLE 2 Slurm launching parameters

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

in the submission command. On the other hand, it can be seen by comparing the parameters in the three columns that, when CUDA is used, different processes of an MPI application must be mapped to different nodes (parameter "-Ni" where i is the amount of requested nodes) so that each process can use a different GPU[2]. On the contrary, 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 3. Workload labeled as "Set 1" is composed of 400 instances from applications GPU-Blast, LAMMPS, mCUDA-MEME, and GROMACS. Notice that these applications are the ones with the shortest execution times. The exact amount of instances for each application is shown in the table. In a similar way, workload labeled as "Set 2" is composed of 400 instances of applications BarraCUDA, MUMmerGPU, GPU-LIBSVM, and NAMD (these applications are the "long" applications). Finally, a third workload, referred to as "Set 1+2", has been created with instances from all the applications. Notice that, for each of the workloads, the instances from different applications as well as the exact sequence of instances within the workload are randomly set. However, once workloads are set, they remain constant across the different experiments presented in this section. This means that the amount of instances of each application and the exact sequence of these instances is not modified across experiments.

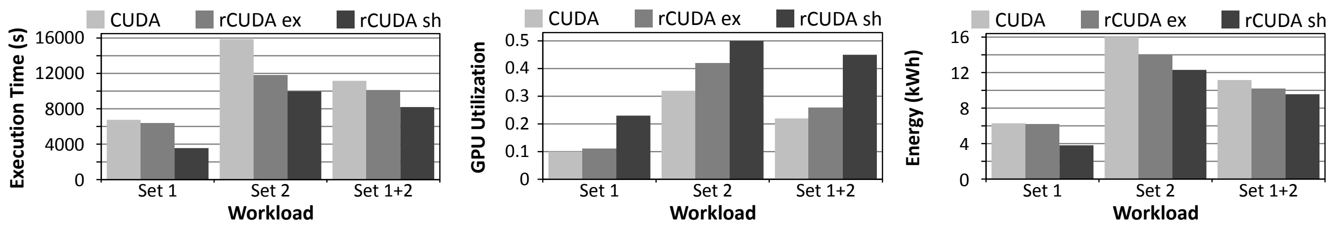
3.3 | Initial Performance Analysis: n nodes with 1 GPU each

This first experiment considers the simplest scenario consisting of a cluster with n nodes each of them owning one GPU. The three cluster sizes mentioned in Section 3.1 were used. Figure 3 shows the performance results for the 16-node case. The other two cluster sizes provided similar trends. The figure shows, for each of the workloads depicted in Table 3, 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. In this case, label "rCUDAex" refers to the results when remote GPUs are used in an exclusive way by applications whereas label "rCUDAsh" refers to the case when remote GPUs are shared among several applications. Among both rCUDA uses, the shared approach is the most interesting option. The exclusive case is considered in this paper only for comparison purposes. Figure 3(a) shows total execution time for each of the workloads.

TABLE 3 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

[2]In our cluster testbed there is only one GPU per node.



(a) Total execution time of the workloads.

(b) Average GPU utilization.

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

FIGURE 3 Performance results from the 16-node 16-GPU cluster.

Figure 3(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. The `nvidia-smi` command was used for polling the GPUs. In a similar way, Figure 3(c) shows total energy required for completing workload execution. Energy has been measured by polling once every second the power distribution units (PDUs) present the cluster. Used 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 3(c).

As can be seen in Figure 3(a), workload "Set 1" presents the smallest execution time, given that it is composed of the applications with the smallest execution times. Furthermore, using rCUDA in a shared way 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. Notice also that the use of remote GPUs in an exclusive way also reduces execution time. In the case for "Set 2" this reduction is more noticeable because, when CUDA is used, the NAMD application (with 101 instances in the workload) spans over 4 complete nodes thus blocking the GPUs in those nodes, which cannot be used by any accelerated application during the entire execution time of NAMD (241 seconds). On the contrary, when "rCUDAex" is leveraged, the GPUs in those four nodes are accessible from other nodes and therefore they can be used by other applications being executed at other nodes. Regarding GPU utilizacion, Figure 3(b) shows that the use of remote GPUs helps to increase overall GPU utilization. Actually, when "rCUDAsh" 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 3(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 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 1: 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 but LAMMPS and mCUDA-MEME already took one core. However, when GPUs are shared among several applications, GPUs assigned to LAMMPS and mCUDA-MEME can also 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 3.

The second reason for the improvements shown in Figure 3 is related to the usage that applications make of GPUs. As Table 1 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. This concern is also present for other applications in Table 1. 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 instead of the 5 GB available in the Tesla K20 device. On the contrary, when rCUDA is used in a shared way, 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 4 presents execution times for the applications in Table 1 when several instances of the same application are concurrently executed in a GPU[3]. Executions in Figure 4 have been manually constrained to a single node using CUDA without

[3]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.

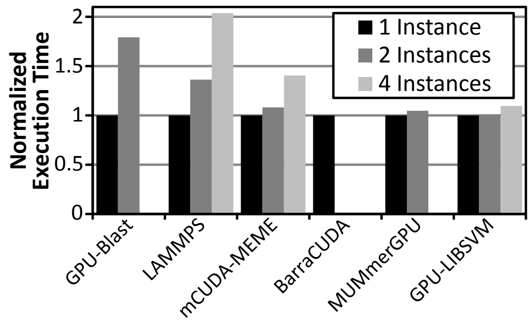


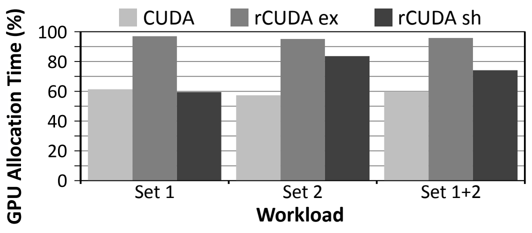
FIGURE 4 Normalized execution time when several concurrent instances of the same application are executed with CUDA.

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.

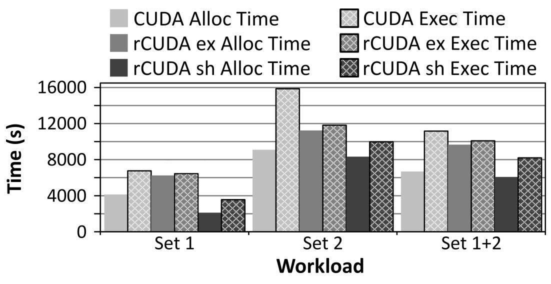
Another possible point of view related to sharing GPUs among applications is that all the applications sharing the GPU execute slower because they have to share the GPU cores. However, despite of the slower execution of each individual application, the entire workload is completed earlier, as shown in Figure 3. This means (1) that 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 regarding the service received.

One additional metric that could be analyzed is the time that GPUs remain allocated to applications. Figure 5 presents the time that any GPU in the cluster is assigned to an application and also compares that time with total execution time of the workload. It can be seen that the use of "rCUDAex" increases the percentage of time that GPUs are assigned to applications up to 96% whereas this percentage is reduced for "rCUDAsh" to values equal to 59%, 83%, and 74% for "Set 1", "Set 2", and "Set 1+2" respectively. These lower percentages point out that, when rCUDAsh is leveraged, execution time of workloads is dominated by the non-GPU applications because accelerated ones take advantage of available remote GPU resources to complete their execution before all non-accelerated applications have finished.

Finally, as shown in Figure 4, the BarraCUDA application presents high GPU memory requirements, which reduce the opportunity to share its GPU with other applications. Thus, executions of BarraCUDA behave in a similar way to the "rCUDAex" mode. In order to analyze the stability of the results when this behavior is reduced, two new workloads have been created. Table 4 shows these new workloads. It can be seen that the presence of the heavy BarraCUDA application has been noticeably reduced at the same time that the presence of lighter applications such as GPU-LIBSVM has been increased. Figure 6 shows the performance results for these new workloads. It can be seen that although the workloads have been noticeably modified, the trends of the results are similar to those in Figure 3. Results for cluster sizes of 4 and 8 nodes also showed similar trends. Moreover, Figure 7 depicts the GPU allocation time with the new workloads. Again, the same trend as with the previous workloads is followed. This suggests that the performance improvements shown in this section can be expected for many other workloads and cluster sizes. Finally, once it has been seen that rCUDA exclusive performs better than CUDA but worse than rCUDA shared mode, henceforth we will continue the performance study taking only into account rCUDA shared.



(a) Percentage of GPU allocation time.

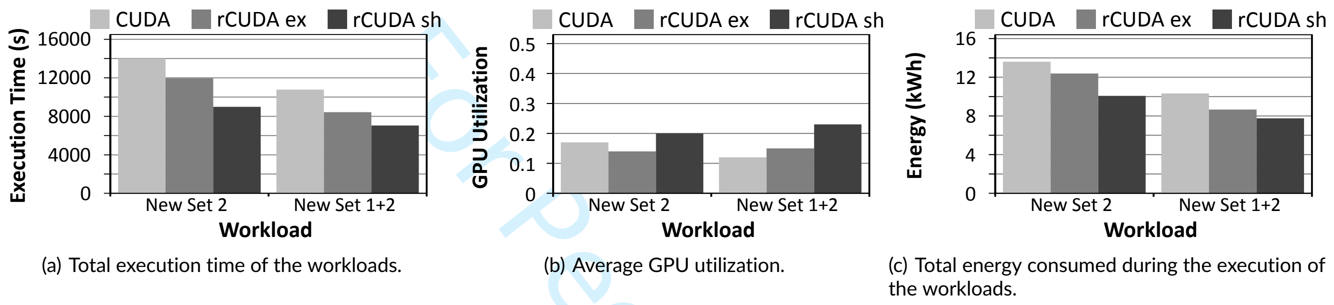


(b) Total GPU allocation time.

FIGURE 5 GPU allocation time in the 16-node 16-GPU cluster.

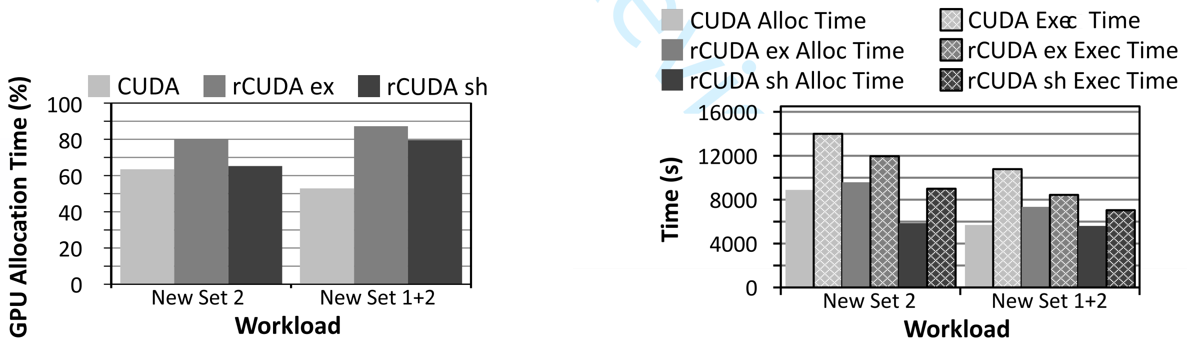
TABLE 4 Composition of new workloads

Application	Workload			
	Set 2	Set 1+2	**New Set 2**	**New Set 1+2**
GPU-Blast	-	57	-	**50**
LAMMPS	-	52	-	**50**
mCUDA-MEME	-	55	-	**50**
GROMACS	-	47	-	**50**
BarraCUDA	112	51	**40**	**20**
MUMmerGPU	88	52	**100**	**50**
GPU-LIBSVM	99	37	**160**	**80**
NAMD	101	49	**100**	**50**
Total	400	400	**400**	**400**



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

FIGURE 6 Performance of the 16-node 16-GPU cluster with the new workloads.



(a) Percentage of GPU allocation time. (b) Total GPU allocation time.

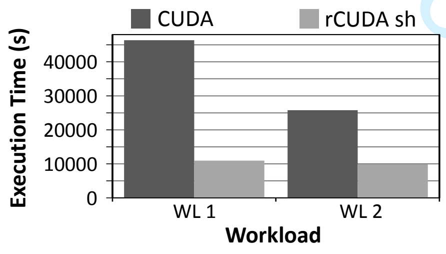
FIGURE 7 GPU allocation time in the 16-node 16-GPU cluster with the new workloads.

3.4 | Introducing Additional Heterogeneity in the Cluster

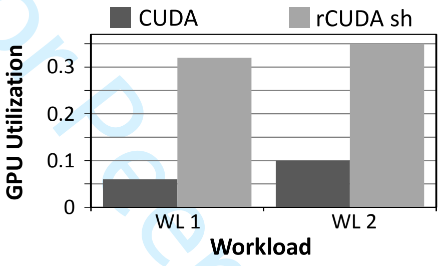
Current data centers execute a large variety of applications. Some of them address highly parallel problems that can benefit from the use of several GPUs. In these cases, the MPI library can be used in order to distribute the application processes across several nodes in the cluster so that each process makes use of the GPU installed in the node executing that process. The net result is that the application is concurrently making use of several GPUs thanks to the MPI library. However, using the MPI library for some applications may not be the best option. For instance, if communications among processes are too intense, then the use of a messaging library would noticeably reduce overall performance. In these cases, instead of following a distributed approach for designing the application, it could be programmed by leveraging the shared memory paradigm. In this context, the application would be divided into threads and each thread would be responsible for submitting its kernels to the GPU. However, in order to efficiently execute such a shared-memory parallel application, as many GPUs as threads should be available in the computer executing

TABLE 5 Composition of workloads for non-uniform heterogeneous clusters

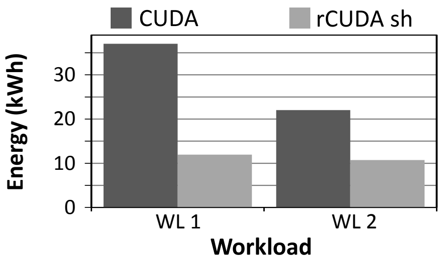
Application	Workload	
	WL 1	WL 2
GPU-Blast	41	48
LAMMPS short	39	46
LAMMPS long 2p	20	10
LAMMPS long 4p	20	10
mCUDA-MEME short	39	46
mCUDA-MEME long 2p	20	10
mCUDA-MEME long 4p	20	10
GROMACS	40	40
BarraCUDA	40	47
MUMmerGPU	41	47
GPU-LIBSVM	40	46
NAMD	40	40
Total	400	400



(a) Total execution time of the workloads.



(b) Average GPU utilization.



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

FIGURE 8 Performance results from a non-uniform cluster with fifteen 1-GPU nodes and one 4-GPU node.

the application. In our cluster example, for instance, where only one GPU is available in each node, it would not be possible to execute this kind of applications. One possible solution could be to augment the cluster with one or more servers featuring several GPUs. This would create a non-uniform heterogeneous cluster.

In order to model this scenario, we have replaced one of the nodes in our 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. Additionally, in order to model the use of these parallel shared-memory applications, we have modified the workloads used in the experiments by modeling 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 5: (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 by using the launching parameters -N1 -n2 -c1. These instances of the application 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 execute in the same node by using the launching parameters -N1 -n4 -c1. 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 5 which make use of one single-threaded process with launching parameters -N1 -n1 -c1. 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. The idea behind the "2p" and "4p" flavors of LAMMPS and mCUDA-MEME is that they necessarily require the 4-GPU server for their execution when a cluster not using rCUDA is employed. On the contrary, when rCUDA is used, all the threads will be placed in the same node but these threads will be able to use any of the GPUs in the cluster, thus loosening up the initial limitation of having to wait for the 4-GPU node and thus speeding up the execution of the workload.

Figure 8 depicts the performance results when the workloads in Table 5 are executed in a non-homogeneous cluster composed of 15 nodes owning one GPU and one additional node populated with 4 GPUs. It can be seen that decoupling GPUs from nodes with rCUDA provides large

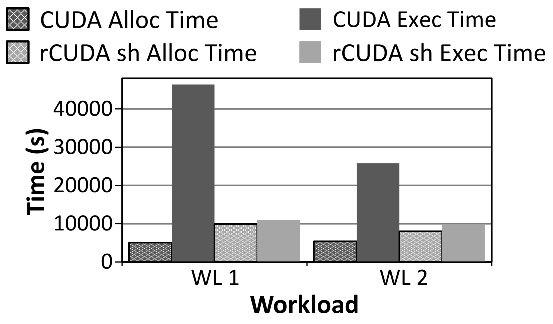


FIGURE 9 GPU allocation time in a non-uniform cluster with fifteen 1-GPU nodes and one 4-GPU node.

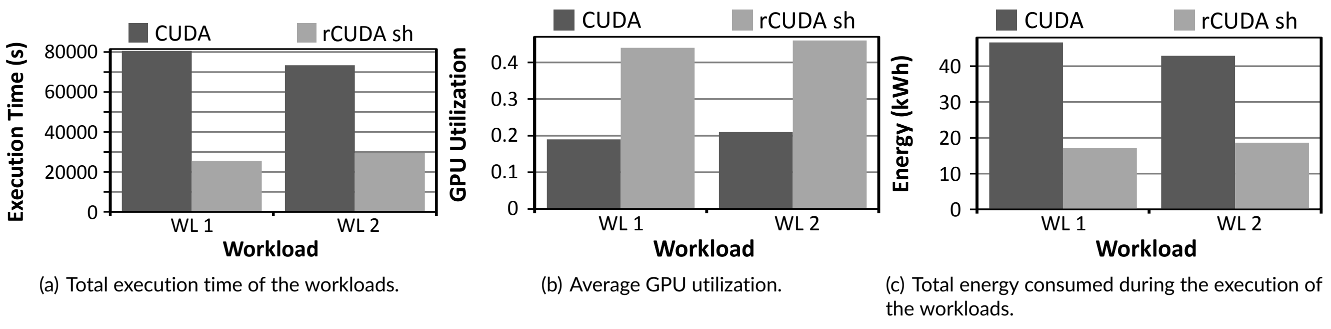
benefits because applications requiring 2 (or 4) GPUs can start execution as soon as there are enough available resources in remote GPUs across the cluster. Contrariwise, when CUDA is used, applications spend a lot of time waiting for the 4-GPU node. This wait can be seen in Figure 9 which shows the small percentage of GPU-allocation time when CUDA is used for these workloads. This small GPU allocation time suggests that applications spend most of the time waiting for resources.

3.5 | Attaching GPUs to a non-GPU Cluster

The use of GPUs in a cluster usually puts several burdens in 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 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 that MPI applications 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 that clusters which did not initially include GPUs can 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 manage the use of the GPUs in the new severs.

Figure 10 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 (the same as in the previous section, but GPUs have been removed). The 4-GPU server is the same as in previous section.



(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 when a server with 4 GPUs is attached to a non-GPU cluster.

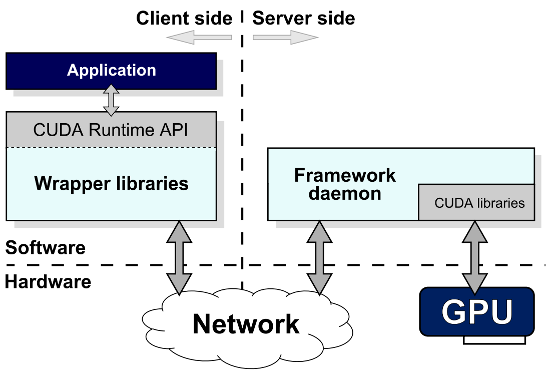


FIGURE 11 Architecture usually deployed by GPU virtualization frameworks.

Results are similar to those presented in the previous section and 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.

4 | RELATED WORK

In this section an overview of the state-of-the-art is provided. We first review other works related to GPU virtualization and then a review about workload managers is addressed.

4.1 | GPU Virtualization

Sharing accelerators among several computers has been addressed both with hardware and software approaches. On the hardware side, maybe the most prominent solution was NextIO's N2800-ICA[38], based on PCIe virtualization[39]. This solution allowed to share a GPU among eight different servers in a rack within a two-meter distance. Nevertheless, this solution lacked from the required flexibility because a GPU could only be used by a single server at a time, thus preventing the concurrent sharing of GPUs. Furthermore, this solution was expensive, what may be one of the reasons for NextIO going out of business in August 2013.

As a flexible alternative to hardware approaches, several software-based GPU sharing mechanisms have appeared, such as V-GPU, DS-CUDA, rCUDA, vCUDA, and GridCuda, for example. Basically, these software proposals share a GPU by virtualizing it, so that they 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, thus offering the same API as the NVIDIA CUDA Runtime API[19] does.

Figure 11 depicts the architecture usually deployed by these virtualization solutions, which follow a distributed client-server approach and is very similar to the architecture depicted for the rCUDA middleware in Figure 2.

CUDA-based GPU virtualization frameworks 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 frameworks to be used in native domains, although the client part of these latter solutions may also be used within a virtual machine. 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 frameworks 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 frameworks.

Regarding the first type of GPU virtualization frameworks mentioned above, several solutions have been developed to be specifically used within virtual machines, as for example vCUDA[17], GViM[18], gVirtuS[16], and Shadowfax[40]. The vCUDA technology supports only 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, which causes a noticeable drop in overall performance. GViM 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. For example, in the case of the memory management module, it implements only 17 out of 37 functions. Furthermore, despite it is designed for KVM virtual machines, it requires a modified version of KVM. Nevertheless, although it is mainly intended to be used in virtual machines, granting them access to the real GPU located in the same node, 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

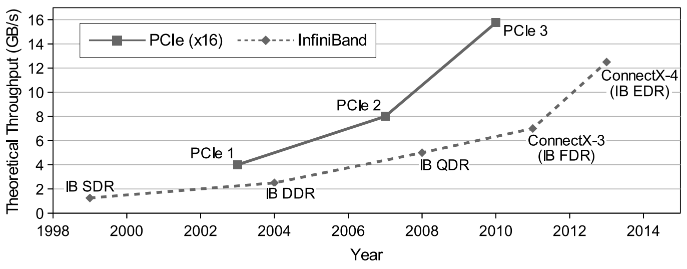


FIGURE 12 Comparison between the theoretical bandwidth of different versions of PCI Express x16 and those of commercialized InfiniBand fabrics and network adapters.

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 frameworks mentioned above, which provide general purpose GPU virtualization, one can find rCUDA[11], V-GPU[41], GridCuda[14], DS-CUDA[15], and Shadowfax II[42]. rCUDA, was already described in Section 2.1. V-GPU is a recent tool supporting CUDA 4.0. Unfortunately, the information provided by the V-GPU authors is fuzzy and there is no publicly available version that can be used for testing and comparison. GridCuda also offers access to remote GPUs in a cluster, but supporting 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. 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 code status.

It is important to notice 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 technologies. For example, as depicted in Figure 12, the theoretical bandwidth of the InfiniBand network is 12.5 GB/s when using the most recent Mellanox EDR InfiniBand adapters. This bandwidth is very close to the 15.75 GB/s of PCIe 3.0 x16. This makes that the bandwidth achieved by EDR InfiniBand network adapters and that of the NVIDIA Tesla K40 GPU are very close. Moreover, the previous generation of these technologies (NVIDIA Tesla K20 GPUs and FDR InfiniBand network adapters), provides performance figures that are also very close: the Tesla K20 GPU used PCIe 2.0, which achieves a theoretical bandwidth of 8 GB/s, whereas FDR InfiniBand (which uses PCIe 3.0 x8) provides 7 GB/s. As a result, when using remote GPU virtualization solutions in both hardware generations (Tesla K40 & EDR InfiniBand and Tesla K20 & FDR InfiniBand), the path communicating the main memory in the computer executing the application and the remote accelerator presents a similar bandwidth in all of its stages. This bandwidth is similar to the one initially attained by the traditional approach using local GPUs, as shown in[43]. This has turned remote GPU virtualization frameworks into an appealing option.

4.2 | Workload Managers and Job Schedulers

In addition to Slurm, there is a myriad of job schedulers, such as PBSPro[44], LoadLeveler[45], Condor[46], MOAB[47], LSF[48], DPCS[49], Quadrics RMS[50], BPROC[51], TORQUE[52], OAR[53], MAUI[54], Sun Grid Engine[55], etc. An in-depth analysis can be found in[56]. We briefly review in the following the most important ones in order to support our choice for Slurm.

The Portable Batch System (PBS)[44] is a commercial scheduler originally developed in NASA. Recently, support for GPU scheduling was introduced with two different flavors: (a) a simple approach in which just a single GPU job can be run at a time in a node and (b) an advanced approach that allows several GPU jobs to be concurrently run in a node. In any case, sharing a GPU among several jobs is not allowed. Although PBS is portable, it mainly presents the concern of being single threaded and hence exhibits poor performance on large clusters.

LoadLeveler[45] is a commercial tool by IBM and supports very few non-IBM systems, thus reducing its portability to general clusters. Furthermore, it presents a very poor scalability, requiring around 20 minutes to execute a trivial 8000-task 500-node assignment[9].

Condor[46] is a parallel job manager developed at University of Wisconsin. It was the basis for LoadLeveler and presents an elaborated checkpoint/restart feature and an interesting advertising system that allows servers to announce their available resources and consumers to disclose their requirements, so that a broker can later perform matches among them. Although the source code of Condor is available, thus making it a good candidate for introducing virtual GPU awareness, this batch system is less used than Slurm, which is used in several of systems in TOP500 list[57].

MOAB[47] is a commercial scheduler derived from the PBS one. It provides the usual features in this kind of systems (backfilling, FCFS, preemption, advance reservation, etc) but given that is is just a scheduler, it requires to be complemented with a resource manager system.

Load Sharing Facility (LSF)[48] is a proprietary batch system and parallel job manager created by Platform Computing. Although it has interesting features such as the possibility of being fed with thermal data from the computing nodes and thus try to balance the workload among servers, it is not open-source, thus hindering the possibility of making any change on it.

DPCS (Distributed Production Control System)[49] was developed at Lawrence Livermore National Laboratory (LLNL) and provides support only for a few computing systems. Furthermore, it requires an underlying infrastructure for parallel job management.

Quadrics RMS[50] is intended for Unix systems leveraging the Quadrics Eln interconnects, thus making its usability quite limited. Moreover, it is proprietary, thus making it impossible to use it for the purposes of the work presented in this paper.

The Beowulf Distributed Process Space (BPROC)[51] is a suit of libraries and utilities that allow to start processes in a Beowulf-style cluster. However, achieving scalability with this tool can be difficult.

The rest of schedulers mentioned above present similar concerns. On the contrary, Slurm is a simple, highly scalable and portable resource management system which, additionally, is open-source and widely used in the HPC community. These characteristics made us to select Slurm as the target for the modifications to achieve virtual GPU awareness. Actually, Slurm has also been previously used for including new features. For example, in[58] an integer programming based heterogeneous CPU-GPU cluster scheduler was proposed for Slurm. However, this work did not consider the use of virtual GPUs. Also, in[59] the use of GPU ranges is proposed. Such a feature can be very useful to runtime auto-tuning applications and systems that can make use of a variable number of GPUs. However, this work does not consider the use of virtual GPUs which are decoupled from the CPU cores. Furthermore, in[60] the reliability of job schedulers, with a focus on Slurm, is analyzed and two proposals are made. Finally, in[61] the features provided by Slurm are enhanced with multi-core/multi-threaded support. As can be seen, Slurm has been extended many times in order to consider new technology trends.

5 | CONCLUSIONS

In this paper we have carried out a thorough performance evaluation of a cluster using a modified version of Slurm which is able to schedule the use of the virtual GPUs provided by the rCUDA middleware. The main idea is that the rCUDA middleware decouples GPUs from the nodes where they are installed, therefore making the scheduling process much more flexible at the same time that a better usage of resources is achieved.

Results from the execution of 7 different workloads composed of 8 applications in 3 different cluster configurations suggest that cluster performance can be noticeably increased just by modifying the Slurm scheduler and introducing rCUDA in the cluster. It is also expected that as GPUs feature larger memory sizes, the benefits presented in this work will become also larger.

ACKNOWLEDGEMENTS

This work was funded by Generalitat Valenciana under Grant PROMETEO/2017/077. The author from Universidad Jaume I was supported by projects TIN2014-53495-R, TIN2015-65316-P, and TIN2017-82972-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. NVIDIA . *CUDA C Programming Guide 7.0*. 2015.

2. Group KOW. *OpenCL 1.2 Specification*. 2011.

3. Wu H, Diamos G, Sheard T, et al. Red Fox: An Execution Environment for Relational Query Processing on GPUs. In: CGO'14. ; 2014: 44–54.

4. Playne DP, Hawick KA. Data Parallel Three-Dimensional Cahn-Hilliard Field Equation Simulation on GPUs with CUDA. In: PDPTA'09. ; 2009: 104–110.

5. Yamazaki I, Dong T, Solca 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.

6. Luo Y, Duraiswami R. Canny edge detection on NVIDIA CUDA. In: CVPRW'08. ; 2008: 1-8.

7. Surkov V. Parallel Option Pricing with Fourier Space Time-stepping Method on Graphics Processing Units. *Parallel Comput.* 2010; 36(7): 372–380.

8. 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.

9. Yoo A, Jette M, Grondona M. SLURM: Simple Linux Utility for Resource Management. In: Lecture Notes in Computer Science. Springer Berlin Heidelberg. 2003 (pp. 44-60).

10. Silla F, Iserte S, Reaño C, Prades J. On the benefits of the remote GPU virtualization mechanism: The rCUDA case. *Concurrency and Computation: Practice and Experience* 2017; 29(13): e4072.

11. Reaño C, Silla F, Shainer G, Schultz S. Local and Remote GPUs Perform Similar with EDR 100G InfiniBand. In: Middleware Industry '15. ACM; 2015; New York, NY, USA: 4:1–4:7

12. Iserte S, Castello A, Mayo R, et al. Slurm Support for Remote GPU Virtualization: Implementation and Performance Study. In: ; 2014: 318-325.

13. Iserte S, Prades J, no CR, Silla F. Increasing the Performance of Data Centers by Combining Remote GPU Virtualization with Slurm. In: ; 2016: 98-101.

14. Liang TY, Chang YW. GridCuda: A Grid-Enabled CUDA Programming Toolkit. In: WAINA. ; 2011: 141–146.

15. Oikawa M, Kawai A, Nomura K, Yasuoka K, Yoshikawa K, Narumi T. DS-CUDA: A Middleware to Use Many GPUs in the Cloud Environment. In: SCC '12. ; 2012: 1207–1214.

16. Giunta G, Montella R, Agrillo G, Coviello G. A GPGPU Transparent Virtualization Component for High Performance Computing Clouds. In: Springer Berlin Heidelberg; 2010: 379–391.

17. Shi L, Chen H, Sun J. vCUDA: GPU accelerated high performance computing in virtual machines. In: IPDPS'09. ; 2009: 1–11.

18. Gupta V, Gavrilovska A, Schwan K, et al. GViM: GPU-accelerated Virtual Machines. In: HPCVirt'09. ; 2009: 17–24.

19. NVIDIA . *CUDA Runtime API 7.0*. 2015.

20. NVIDIA . *CUDA Driver API 7.0*. 2015.

21. Reaño C, Silla F. Tuning remote GPU virtualization for InfiniBand networks. *The Journal of Supercomputing* 2016; 72(12): 4520–4545.

22. Reaño C, Silla F. A Performance Comparison of CUDA Remote GPU Virtualization Frameworks. In: ; 2015: 488-489.

23. ImbernÃşn B, Prades J, GimÃľnez D, Cecilia JM, Silla F. Enhancing large-scale docking simulation on heterogeneous systems: An MPI vs rCUDA study. *Future Generation Computer Systems* 2018; 79: 26 - 37.

24. Silla F, Prades J, Reaño C. Leveraging rCUDA for Enhancing Low-Power Deployments in the Physics Domain. In: ICPP '18. ; 2018: 17:1–17:8.

25. Reaño C, Prades J, Silla F. Exploring the Use of Remote GPU Virtualization in Low-Power Systems for Bioinformatics Applications. In: ICPP '18. ; 2018: 8:1–8:8.

26. Prades J, Reaño C, Silla F, Imbernón B, Pérez-Sánchez H, Cecilia JM. Increasing Molecular Dynamics Simulations Throughput by Virtualizing Remote GPUs with rCUDA. In: ICPP '18. ; 2018: 9:1–9:8.

27. Pérez F, Reaño C, Silla F. Providing CUDA Acceleration to KVM Virtual Machines in InfiniBand Clusters with rCUDA. In: Jelasity M, Kalyvianaki E., eds. *Distributed Applications and Interoperable Systems*; 2016: 82–95.

28. Prades J, Reaño C, Silla F. CUDA Acceleration for Xen Virtual Machines in Infiniband Clusters with rCUDA. In: PPoPP '16. ; 2016: 35:1–35:2.

29. NVIDIA . GPU Applications. http://www.nvidia.com/object/gpu-applications.html; 2015.

30. Vouzis PD, Sahinidis NV. GPU-BLAST: Using graphics processors to accelerate protein sequence alignment. *Bioinformatics* 2010.

31. 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.

32. 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.

33. Pronk S, PÃąll S, Schulz R, et al. GROMACS 4.5: a high-throughput and highly parallel open source molecular simulation toolkit. *Bioinformatics* 2013; 29(7): 845-854.

34. Klus P, Lam S, Lyberg D, et al. BarraCUDA - a fast short read sequence aligner using graphics processing units. *BMC Research Notes* 2012; 5(1): 1–7.

35. Kurtz S, Phillippy A, Delcher A, et al. Versatile and open software for comparing large genomes. *Genome Biology* 2004; 5(2).

36. Chang CC, Lin CJ. LIBSVM: A Library for Support Vector Machines. *ACM Trans. Intell. Syst. Technol.* 2011; 2(3): 27:1–27:27.

37. Phillips JC, Braun R, Wang W, et al. Scalable molecular dynamics with NAMD. *Journal of Computational Chemistry* 2005; 26(16): 1781–1802.

38. NextIO, N2800-ICA — Flexible and manageable I/O expansion and virtualization. http://www.nextio.com/; .

39. Krishnan V. Towards an integrated IO and clustering solution using PCI express. In: CLUSTER'07. ; 2007: 259–266.

40. Merritt AM, Gupta V, Verma A, Gavrilovska A, Schwan K. Shadowfax: Scaling in Heterogeneous Cluster Systems via GPGPU Assemblies. In: VTDC'11. ; 2011: 3–10.

41. V-GPU: GPU virtualization. http://www.zillians.com/products/vgpu-gpu-virtualization/; .

42. Shadowfax II - scalable implementation of GPGPU assemblies. http://keeneland.gatech.edu/software/keeneland/kidron; .

43. Reaño C, Silla F, Shainer G, Schultz S. Local and Remote GPUs Perform Similar with EDR 100G InfiniBand. In: Middleware Industry '15. ACM; 2015: 4:1–4:7.

44. Nitzberg B, Schopf JM, Jones JP. PBS Pro: Grid Computing and Scheduling Attributes. In: Kluwer Academic Publishers; 2004: 183–190.

45. Kannan S, Roberts M, Mayes P, Brelsford D, Skovira J. *Workload Management with LoadLeveler*. IBM, rst ed. . 2001.

46. Tannenbaum T, Wright D, Miller K, Livny M. Beowulf Cluster Computing with Linux. In: MIT Press. 2002 (pp. 307–350).

47. Moab Workload Manager Documentation. http://www.adaptivecomputing.com/resources/docs/; .

48. LSF (Load Sharing Facility) Features and Documentation. http://www.platform.com/workload-management/high-performance-computing; .

49. Distributed Production Control System. http://www.llnl.gov/icc/lc/dpcs_overview.html; .

50. Quadrics Resource Management System. http://www.quadrics.com/website/pdf/rms.pdf; .

51. Beowulf Distributed Process Space. http://brpoc.sourceforge.net; .

52. Torque Resource Manager Documentation. http://www.adaptivecomputing.com/resources/docs/; .

53. Capit N, Da Costa G, Georgiou Y, et al. A batch scheduler with high level components. In: . 2. ; 2005: 776-783 Vol. 2.

54. Bode B, Halstead DM, Kendall R, Lei Z, Jackson D. The Portable Batch Scheduler and the Maui Scheduler on Linux Clusters. In: ALS'00. USENIX Association; 2000: 27–27.

55. Gentzsch W. Sun Grid Engine: towards creating a compute power grid. In: ; 2001: 35-36.

56. Georgiou Y. *Resource and Job Management in High Performance Computing*. PhD Thesis, Joseph Fourier University, France . 2010.

57. Slurm Workload Manager. http://slurm.schedmd.com; .

58. Soner S, Ozturan C. Integer Programming Based Heterogeneous CPU-GPU Cluster Scheduler for SLURM Resource Manager. In: HPCC-ICESS'12. ; 2012: 418-424.

59. Soner S, Ozturan C. Extending Slurm with Support for GPU Ranges. In: ; 2013.

60. Sabin G, Sadayappan P. On Enhancing the Reliability of Job Schedulers. In: HAPCW. ; 2005.

61. Balle SM, Palermo DJ. Enhancing an Open Source Resource Manager with Multi-core/Multi-threaded Support. In: JSSPP'07. Springer-Verlag; 2008; Berlin, Heidelberg: 37–50.

How to cite this article: S. Iserte., J. Prades, C. Reaño, and F. Silla (2019), Improving the Management Efficiency of GPU Workloads in Data Centers through GPU Virtualization, *Concurrency and Computation: Practice and Experience, 2019;00:1–6.*