SLURM Support for Remote GPU Virtualization: Implementation and Performance Study

Sergio Iserte, Adrián Castelló,
Rafael Mayo, Enrique S. Quintana-Ortí
Universitat Jaume I de Castelló
Castelló de la Plana, Spain
{siserte,adcastel,mayo,quintana}@uji.es

Federico Silla, Jose Duato,
Carlos Reaño, Javier Prades
Universitat Politècnica de València
València, Spain
{fsilla,jduato}@disca.upv.es,
{carregon,japraga}@gap.upv.es

***Abstract*—SLURM is a resource manager that can be leveraged to share a collection of heterogeneous resources among the jobs in execution in a cluster. However, SLURM is not designed to handle resources such as graphics processing units (GPUs). Concretely, although SLURM can use a generic resource plugin (GRes) to manage GPUs, with this solution the hardware accelerators can only be accessed by the job that is in execution on the node to which the GPU is attached. This is a serious constraint for remote GPU virtualization technologies, which aim at providing a user-transparent access to all GPUs in cluster, independently of the specific location of the node where the application is running with respect to the GPU node.**

In this work we introduce a new type of device in SLURM, "rgpu", in order to gain access from any application node to any GPU node in the cluster using rCUDA as the remote GPU virtualization solution. With this new scheduling mechanism, a user can access any number of GPUs, as SLURM schedules the tasks taking into account all the graphics accelerators available in the complete cluster. We present experimental results that show the benefits of this new approach in terms of increased flexibility for the job scheduler.

***Keywords*—HPC cluster; job scheduler; resource management; remote GPU virtualization;**

I. Introduction

Graphics processing units (GPUs) have remarkably evolved during the last few years, from being just graphics coprocessors to become powerful general-purpose accelerators, profusely adopted in high performance computing (HPC) systems. In addition to the favorable performance/cost ratio of GPUs, this evolution has been further stimulated by considerable advances in GPU programmability, with the introduction of frameworks such as CUDA [1], OpenCL [2] and OpenACC [3]. As a result, GPU computing (also known as GPGPU) is nowadays successfully exploited in areas as diverse as finance [4], chemical physics [5], computational algebra [6], health-care equipment [7], computational fluid dynamics [8], and image analysis [9], among others.

On the other hand, the deployment of GPUs is hampered by their high acquisition and maintenance (including energy) costs, as well as the limited amount of (GPU-appealing) data-parallelism for many applications. In this sense, *remote GPU virtualization* offers an alluring means to increase utilization of the GPUs installed in a cluster, which can potentially yield a faster amortization of the total costs of ownership (TCO) for this type of equipment. Concretely, GPU virtualization logically decouples the GPUs in the cluster from the nodes they are located in, thus opening a path to share the accelerators among all the applications that request GPGPU services, independently of whether the node(s) these applications are mapped to are equipped with a GPU. In consequence, the GPUs can be accessed from any node in the cluster, the amount of these accelerators can be reduced, and their utilization rate can be significantly improved.

Currently, there exist several frameworks for remote GPU virtualization, based either on the CUDA or OpenCL application programming interfaces (APIs): rCUDA [10], [11], GVirtuS [12], DS-CUDA [13], vCUDA [14], GViM [15], GridCuda [16], V-GPU [17], SnuCL [18], dOpenCL [19], VOCL [20], and VCL [21]. In these frameworks, applications invoking CUDA/OpenCL kernels are not aware that their requests are intercepted by the corresponding GPU virtualization middleware and redirected to a real GPU, which is generally located in a remote node of the cluster.

Although remote GPU virtualization has demonstrated very low overhead with respect to a configuration with a local GPU [22], due to its novelty, this technology is not yet supported by the job schedulers that are commonly encountered in production clusters (e.g., SLURM [23], PBSPro [24], MOAB [25], TORQUE [26], LSF [27], OAR [28], MAUI [29], LoadLever [30], Condor [31], and Sun Grid Engine [32]). In particular, a common job scheduler in production today only deals with real GPUs so that, when a job requests a number of nodes equipped with one (or more) GPU(s), the scheduler will try to map that job to nodes that actually own the requested number of GPUs, thus impairing the benefits of GPU virtualization.

Nevertheless, it should be possible to modify the scheduler, so that it becomes aware of the fact that the assignment should no longer be constrained by the GPU kernels having to be executed in the same node where the invoking application is mapped to. The goal is thus to create a GPU virtualization-aware job scheduler which in turn allows applications to leverage all the cluster GPUs, independently of their location.

In this paper we present an extension to the SLURM resource manager in order to support remote GPU virtualization. The choice of SLURM is motivated by this scheduler being distributed as open-source as well as its portability and interconnect independence, which makes it suitable for a variety of cluster architectures. Moreover, its scalability and robustness, as well as its administrator-friendly characteristics further improve SLURM's appeal. We finally note the widespread of SLURM in HPC clusters. Indeed, this job scheduler is currently used in five of the ten top systems in the Top500 list, including the system currently ranked in the first position [33].

SLURM has been extended many times in order to integrate new features, some of them related to GPUs. For example, an integer programming-based heterogeneous CPU-GPU cluster scheduler was introduced in SLURM in [34]. The authors also propose there the use of GPU ranges. Such a feature can be very useful to runtime autotuning applications and systems that can make use of a variable number of GPUs. However, that work does not consider the use of virtual GPUs, decoupled from the CPU cores.

Our remote GPU virtualization support for SLURM can be easily adapted to any GPU virtualization framework. In this paper we focus on rCUDA, since this package supports the most up-to-date CUDA version and, in addition, it has been reported to offer remarkable high performance [10] [11].

The rest of the paper is organized as follows. Section II briefly introduces SLURM and rCUDA. The main contributions of this work follow next, with the description of the tools in Section III, and the evaluation of the framework in Section IV. Finally, Section V outlines a few conclusions.

II. Background

In this section we review the most relevant characteristics of rCUDA and SLURM.

A. *The rCUDA architecture*

rCUDA is a framework that provides transparent access to any GPU installed in a cluster, independently of the location of the application requesting GPGPU services. Thus, rCUDA is useful in a number of scenarios: *i)* in a cluster equipped with rCUDA, the designers can reduce the total number of GPUs in the system, improving the utilization rate of the power-hungry hardware accelerators; *ii)* rCUDA can also be leveraged to significantly accelerate the data-parallel computations of a conventional cluster, by adding only a reduced pool of accelerators to the system, much smaller than the total number of nodes; and *iii)* rCUDA increases the number of GPUs that can be accessed by an application, from only the local accelerators to all GPUs available in the cluster. In summary, in many practical cases, in exchange for a slight increase of the execution time of GPU-enabled applications, considerable savings can be achieved in energy consumption, maintenance, space, and cooling with rCUDA.

The rCUDA framework is split into two major software modules, as depicted in Figure 1:

- The client middleware consists of a collection of wrappers that replace the NVIDIA CUDA Runtime (provided by NVIDIA as a shared library) in the client (GPU-less) node, and some accelerated libraries such as cuBLAS, cuFFT and cuSPARSE. These wrappers are in charge of forwarding the API calls from the applications requesting acceleration services to the server middleware, and retrieving the results, providing applications with the illusion of a direct access to a local GPU.
- The server middleware runs as a service on one or more cluster nodes, equipped with one or more GPUs each. The middleware receives, interprets, and executes the API calls from the clients over a real GPU, employing a different process to serve each remote execution over an independent GPU context, thus enabling GPU multiplexing.

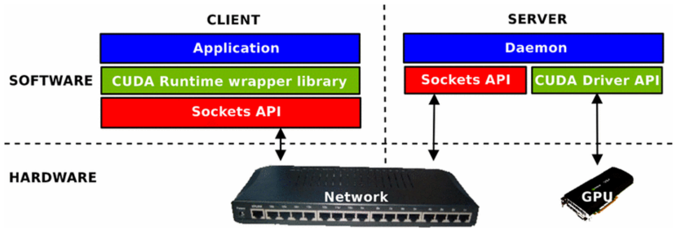

Fig. 1. Overview of the general architecture of the rCUDA virtualization solution.

rCUDA is organized as a client-server distributed architecture; see Figure 1. The client middleware runs in the same cluster node as the application demanding GPU acceleration services, while the server middleware runs in the cluster node where the physical GPU resides.

rCUDA accommodates several underlying client-server communication technologies, thanks to its modular, layered architecture, which supports runtime-loadable network-specific communication libraries.

This software currently provides communication modules for Ethernet and InfiniBand based networks. Furthermore, regardless of the specific communication technology, data transfers between rCUDA clients and servers are pipelined for performance, using preallocated buffers of pinned memory.

B. *SLURM*

SLURM consists of a daemon that runs on each computing node (`slurmd`), a central daemon that runs on the management node (`slurmctld`), and several command line utilities (`srun`, `scancel`, `sinfo`, `squeue`, and `scontrol`). The daemons manage *nodes*, the compute resource in SLURM; *partitions*, which group nodes into logical disjoint sets; *jobs* or allocations of resources assigned to a user for a specified amount of time; and *job steps*, which are sets of (possibly parallel) tasks within a job. Each job in the priority-ordered queue is allocated nodes within a single partition. Once an allocation request fails, no lower priority jobs for that partition will be considered for a resource allocation. Once a job is assigned a set of nodes, the user is able to initiate parallel work in the form of job steps in any configuration within the allocation. For instance, a single job step may be started that utilizes all nodes allocated to the job, or several job steps may independently use a portion of the allocation.

III. Integration of GPU virtualization via rCUDA into SLURM

We next describe the main code modifications that were added to SLURM in order to accommodate GPU virtualization via rCUDA, and how SLURM runs after these changes.

A. Changes to SLURM and rCUDA

The following list of modifications were necessary to extend the SLURM-rCUDA suites with the sought-after functionality:

1) New attributes were added to several data structures in SLURM, in order to maintain certain information about the GPUs which is required by jobs, partitions and nodes.
2) The GRes module of SLURM was modified to allow that all cluster GPUs were accessible to all nodes, which implies that GPUs are to be shared among the nodes. This module manages the allocations and deallocations of generic resources such as the GPUs.
3) Two new SLURM plug-ins were implemented: The GRes plug-in "`gres/rgpu`" declares remote GPUs as a new generic resource in the system. The select plug-in "`select/cons_rgpu`" is responsible for job resource selection and scheduling. The code of this plug-in is based on the "`select/cons_res`" plug-in, and therefore similar behaviour can be expected from it.
4) The job submission commands were augmented with new parameters in order to specify new features to configure rgpu options.
5) Additional fields were introduced in the RPC packages in order to transfer the rGPU information used by SLURM to schedule the jobs.
6) Finally, two rCUDA environment variables had now to be set during the scheduling, in order to enable the use of rCUDA software:
    - RCUDA_DEVICE_COUNT, used by an rCUDA client to learn how many GPUs exist.
    - RCUDA_DEVICE_X, used by an rCUDA client to acquire the IP of the nodes where the rGPUs are installed.

With these changes, SLURM allows the user to submit jobs to the system queue(s) under three different working modes:

- **Original (SLURM)**: The behavior of SLURM is analogous to version 2.6.2.
- **Exclusive (rCUDAex)**: SLURM decouples GPUs from nodes, but they remain accessible only to one job at a time.
- **Shared (rCUDAsh)**: Nodes and GPUs are decoupled and the graphics accelerators can now be shared by several jobs. This mode is automatically selected whenever a certain amount of GPU memory is requested.

B. Operation of the GPU virtualization-aware SLURM

Once the controller (`slurmctld`) is launched, this process checks the total amount of configured rGPUs in each partition of the cluster. After this initial step, SLURM awaits for jobs to be scheduled. We will show the operation using `srun` as a workhorse, but the behaviour is also common for `salloc` and `sbatch`. For example, consider the command:

```
srun -N1 --gres=rgpu:4:1G job.sh
```

which submits a request to SLURM to allocate 1 node (preferably with a local GPU) and 4 remote GPUs with 1 Gbyte of memory each, in order to run the script `job.sh`.

The request could also be more precise, and ask for a specific node of the cluster:

```
srun -w "node3" --gres=rgpu:4:1G job.sh
```

In this example, SLURM is requested to allocate node `node3`, no matter whether it is equipped with GPUs: In case it is not, the node will thus have to access remote GPUs to execute the job.

Alternatively, the request can ask for all the memory of a GPU ensuring that the job will be executed exclusively. An easy way to achieve this behaviour is:

```
srun --rcuda-mode=excl
 --gres=rgpu:4 job.sh
```

Let us illustrate the new capabilities with an additional example. Assume we have a small cluster with 4 nodes (*node[0-3]*) in which only *node0* and *node1* are equipped with one GPU each, and each GPU has 4 Gbytes of memory; *node2* and *node3* are thus GPU-less. Moreover, consider we want to run two different jobs on this small cluster, each requesting 2 nodes and 2 GPUs, but only 1 Gbyte of memory per GPU. Submitting the jobs in exclusive mode, the 2 available GPUs will be assigned to one job, while the other will be enqueued until the first one finalizes. On the other hand, with the shared mode both jobs will run concurrently sharing the two GPUs. In the first case, two nodes will be idle while executing the jobs (one after another). In the second case, the four nodes are simultaneously used. These alternative configurations thus allow to share the GPU and increase the number jobs in execution (throughput), or boost the performance of a single job.

Upon encountering a new submission, SLURM commences the scheduling procedure. First, the select plug-in "`select/cons_rgpu`" checks whether the job requires rGPUs. If that is the case, the "`rgpu_mode`" will be activated for this job. (Otherwise, SLURM just proceeds as usual; i.e., without the GPU virtualization-aware modification.)

Depending on the resources available in the cluster and the job characteristics, it will be then enqueued or executed. When a job is submitted, it is immediately enqueued and its requirements are checked. If the cluster has enough resources for it, the scheduler eventually submits the job to execution. When the job is running, the resources that were requested need to be allocated. This is carried out by the new GRes module, where the implied data structures are updated with scheduling information. Currently our selection algorithm implements a policy which prioritizes rGPUs located on the execution node itself. If it is impossible to fulfil this request, the algorithm looks for more rGPUs in the rest of the nodes of the cluster. This search is performed by iterating an ordered list with all the nodes hosting rGPUs, until the request is satisfied.

Upon completion of the execution, all resources allocated to it are released, becoming then available for other jobs.

IV. Performance Evaluation

In this section we perform an evaluation of the new SLURM module and report the results obtained with it. For this purpose we have configured a cluster with the upgraded, GPU-aware version of SLURM; we have studied the scalability of several scientific applications; we have executed the application in two different ways, focusing our attention on the performance and the throughput; and we have tested the system while reducing the number of resources in order to compare the results between physical GPUs and virtualized remote GPUs.

A. Setup

All the tests were executed on a 9-node cluster, where 8 of them are compute nodes while the last one is used as a front-end. Each node is equipped with a Supermicro 1027GF-TRF motherboard, two Intel Xeon E5-2620 (Ivy Bridge) hexacore processors at 2.1 GHz, and 32 Gbytes of DDR3 SRAM memory at 1.6 GHz. Each node is also endowed with an NVIDIA Tesla K20 GPU connected trough a PCIe 2.0 x16 to the motherboard. The cluster nodes communicate via a Mellanox SX6025 (Infiniband FDR-compatible) switch, and each node is equipped with a Mellanox ConnectX-3 VPI single-port (InfiniBand FDR-compatible) card. The theoretical bandwidth for this network configuration is 56 Gbytes/s.

In the software stack the testbed system operates under a CentOS 6.4; the communication network runs the Mellanox OFED 2.1-1.0.0 driver; and the GPUs use CUDA 5.5 and the NVIDIA driver 331.62. The GPU virtualization support is based on rCUDA 4.1.

B. Applications

In order to implement a real heterogeneous workload, we decided to use the following applications, selected because of their appealing properties:

1) GPU-Blast: This is one of the most used bioinformatic tools, implemented as a multi-thread application to be run using a single process which accesses a single GPU only[1].

2) LAMMPS: This is a classic molecular dynamics simulator to model atoms. From the implementation perspective, it is a multi-thread and multi-process application which needs at least one GPU to host their processes, but can benefit from the use of multiple GPUs[2].

[1]http://archimedes.cheme.cmu.edu

[2]http://lammps.sandia.gov

TABLE I. Maximum performance on applications

Application	Configuration	Execution time (s)
GPU-Blast	1 process with 6 threads	21
LAMMPS	5 single-thread processes in 5 different nodes	15
MCUDA-MEME	4 single-thread processes in 4 different nodes	165
GROMACS	2 processes, with 12 threads each one, in 2 nodes	167

3) MCUDA-MEME: This a bioinformatic application based on the MEME algorithm. With properties similar to LAMMPS, MCUDA-MEME is multi-thread and multi-process, though in this code each process needs a GPU. In consequence, they must run in different nodes[3].

4) GROMACS: As LAMMPS, GROMACS is a molecular dynamics simulator, but this application does not need GPU acceleration[4]. This multi-thread and multi-process application that contributes a higher degree of heterogeneity to our experimental workload.

All the GPU-requesting applications are in the NVIDIA's applications catalog [35].

C. Workloads

The combination of rCUDA and SLURM allows the GPU to be shared by several jobs. In order to do this, it is necessary to provide the maximum amount of GPU memory requested for each job. This corresponds to the memory requested by the job, the GPU memory needed by the GPU driver, and the GPU memory needed for the rCUDA daemon. Thus, the maximum amount of memory that our application consumes will be given by: $App_{MaxMem} + NVIDIA\ Driver + (rCUDAd \cdot number\ of\ threads)$.

In our concrete case, the driver occupies around 10 Mbytes and the rCUDA daemon 63 Mbytes. This number is multiplied by 6 when GPU-Blast is running though. The third column of Table III shows the configuration of a job prepared to use rCUDA. The last field of the `gres` clause indicates the amount of memory requested by the job, taking into account the operators.

We have generated three workloads of different theoretical duration. The workloads have been generated randomly, but reproducible by setting the same seed to the random function. The generator is given a quantity of theoretical minutes and it appends a new random job that will be in charge of one of the four applications. The application chosen each time, is a random process where the probability of being selected is 0.25. Notice that the theoretical time should be similar to that obtained by executing a workload in a sequential mode. In our case jobs are overlapped so the execution time, as will be shown in the next section, is considerably lower. As reported in Table I, each application has its own execution time, and this value is used to add jobs to the workload until the sum of the times exceeds the previous given time.

Table II contains a detailed description, for each duration, of the quantity of instances of each application. The order of the jobs is independent of the type of workload. The

[3]http://sites.google.com/site/yongchaosoftware/Home/cuda-meme

[4]http://www.gromacs.org

TABLE II. WORKLOAD COMPOSITION

App	Workload		
	2 hours	4 hours	8 hours
GPU-Blast	12	43	81
LAMMPS	18	47	90
MCUDA-MEME	18	36	77
GROMACS	23	42	79
Total	71	168	327

modifications only affect the launching parameters depending on the nature of the experiment.

To sum up, there are 4 scenarios established with 6 workloads in each one. Moreover, these workloads are divided into 2 groups, depending on whether they request GPUs or rGPUs. Both groups include 3 different workload volumes with respect to the theoretical time.

D. Experimentation

Let us start by stating that SLURM was configured with the scheduling policy `backfill` so that jobs can overtake others. Furthermore, the selection of consumable resources has been performed with the `cons_rgpu` policy, which allows the request of remote GPUs.

We leveraged MPI implementation MVAPICH2, specially tuned for InfiniBand technology. We used `salloc` command to submit multiprocess jobs, since `srun` needs the application linked to SLURM's implementation of PMI library.

In the experiments carried out in this work, we designed three distinguishable types of experiments, each with a different purpose.

In the first one jobs are expected to attain the highest possible performance. The data have been extracted from a scalability analysis, carried out in order to estimate the maximum performance during the execution of each application in our system. This analysis involved the execution of applications with a variety of configurations regarding number of processes, when possible. Although, our cluster is equipped with 8 GPUs, we restricted the MCUDA-MEME application to work with only 4 GPUs because our goal was to reduce the resources of our cluster to 4 GPUs or less, and launch experiments with this hardware configuration. On the contrary, LAMMPS is configured with 5 GPUs because of its implementation. Table III summarizes the best configuration and execution time for each application. Note that looking for the highest performance rate is understandable when there are few jobs to execute. However, our workloads are composed of a considerable quantity of jobs, and therefore, with this configuration a lot of resources are wasted due to some jobs having to wait for the rest of the resources requested.

The second configuration adds several degrees of freedom in order to avoid the restriction of each process having to be run in a different node. We recognize that this will likely cause a non optimal mapping of resources to the jobs but, on the other hand, it will also yield a higher global throughput. Table IV shows how applications, such as GPU-Blast and LAMMPS, do not force their processes to run in different nodes (the argument `-N` was removed). However,

TABLE III. SLURM LAUNCHING PARAMETERS TO GET THE UNITARY MAXIMUM PERFORMANCE

Application	Launch with CUDA	Launch with rCUDA
GPU-Blast	-N1 -n1 -c6 –gres=gpu:1	-N1 -n1 -c6 –gres=rgpu:1:1686M
LAMMPS	-N5 -n5 -c1 –gres=gpu:1	-N5 -n5 -c1 –gres=rgpu:5:3275M
MCUDA-MEME	-N4 -n4 -c1 –gres=gpu:1	-n4 -c1 –gres=rgpu:4:163M
GROMACS	-N2 -n2 -c12	-N2 -n2 -c12

TABLE IV. SLURM LAUNCHING PARAMETERS TO IMPROVE THE GLOBAL THROUGHPUT

Application	Launch with CUDA	Launch with rCUDA
GPU-Blast	-n1 -c6 –gres=gpu:1	-n1 -c6 –gres=rgpu:1:1686M
LAMMPS	-n5 -c1 –gres=gpu:1	-n5 -c1 –gres=rgpu:5:3275M
MCUDA-MEME	-N4 -n4 -c1 –gres=gpu:1	-n4 -c1 –gres=rgpu:4:163M
GROMACS	-N2 -n2 -c12	-N2 -n2 -c12

the MCUDA-MEME and GROMACS configurations remain without change. The explanation for MCUDA-MEME is that this application needs 4 GPUs (one per process); LAMMPS is able to use the same GPU for different processes, while MCUDA-MEME is not. GROMACS boosts its execution time if two or more instances share a node due to the pinning configuration between threads and cores. That is why we decided that each instance is given the exclusive access to the two requested nodes. After these changes, we observe a higher throughput of jobs per minute.

The second row of Table V reveals that some resources are underutilized. Hence, we planned to reduce the number of GPUs in the cluster progressively, from 8 to 4–6 GPUs. For this purpose, we modified the command to submit an application in order to match the new "platform". This was a minor change in the submission of LAMMPS instances using rCUDA on a 4-GPU cluster. In particular, as our cluster no longer has 5 GPUs, we cannot request 5 rGPUs so that the launching parameters for LAMMPS in this scenario was `-n5 -c1 --gres=rgpu:4:3275M`.

Finally, we experienced an overflow of SLURM's controller daemon when submitting such a huge burst of jobs in a very short period of time. To tackle this, we delayed the submissions in order to avoid the saturation. The delay was increased as the workload grew, so during the submission of the first jobs the delay was close to zero.

E. Results

The following charts show the whole execution time using bars, while the lines represent the throughput (number of jobs per minute) of the system.

Figure 2 compares the results obtained in a cluster of 8 GPUs, for jobs submitted with two different objectives. The first one (left) aims to get the maximum unitary performance (see Table III), whilst the second (right) looks for throughput (see Table IV). The chart on the left reflects a similar behaviour in both modes. Although the rCUDA mode can share GPUs, this overloading with the restriction of using a specific number of nodes to execute a job, delays the global progress. For example, the parallel execution of 12 instances (because of the 12 cores of a node) in the same 4 nodes takes about 1,884 seconds. On the contrary, the serial execution requires 165

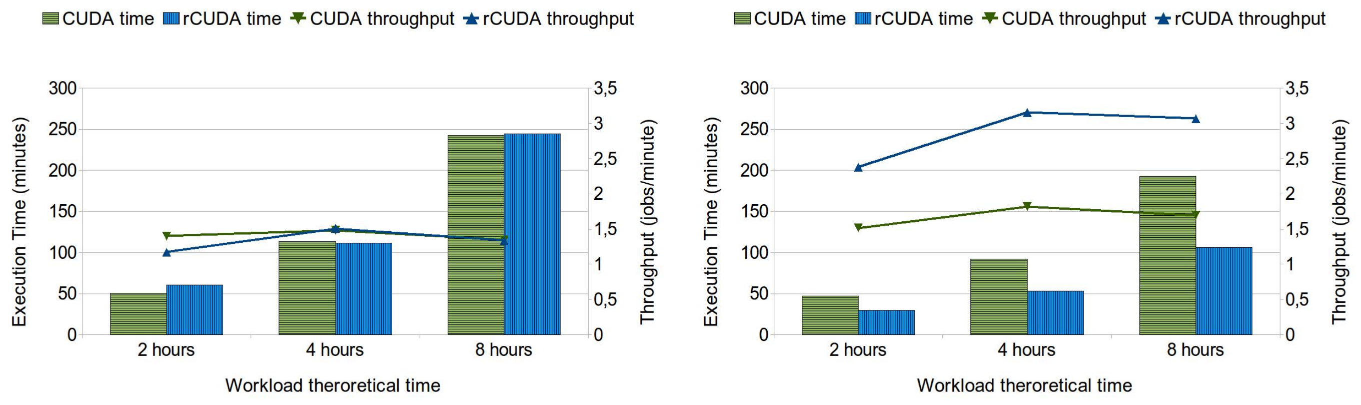

Fig. 2. Comparation between CUDA and rCUDA in a 8-GPU cluster configuring the launch of the jobs at maximum performance (see Table III) (left) and configuring the launch to improve the throughput (see Table IV) (right).

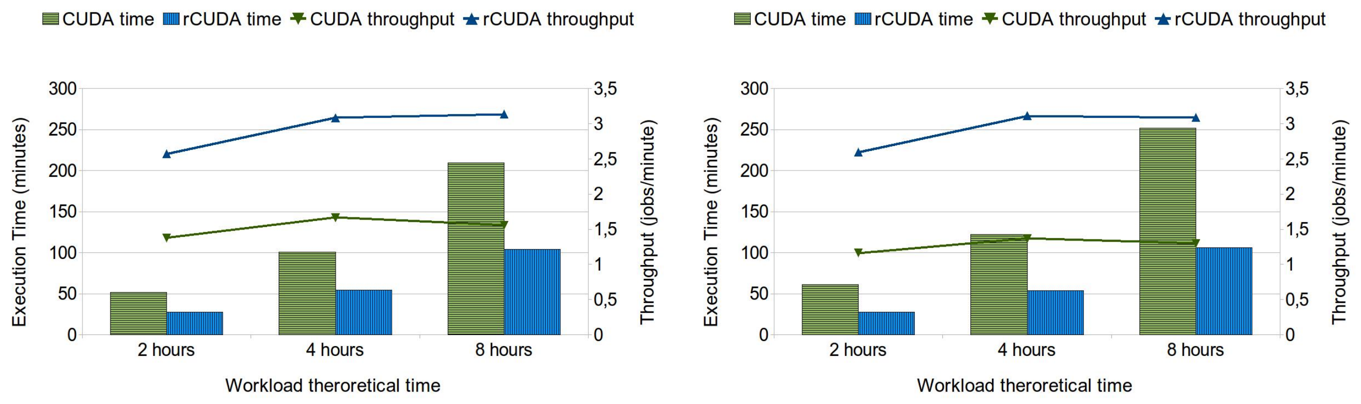

Fig. 3. Comparation between CUDA and rCUDA in a 6-GPU cluster (left) and in a 4-GPU cluster (right).

seconds × 12 instances = 1,980 seconds, which is quite similar to the parallel result.

At first glance, in see Figures 2–right and 3–both we appreciate that rCUDA reduces the execution time, in several tests in a factor larger than 2×, This is well illustrated with the execution times on a 4-GPU cluster. Even more remarkably, the rCUDA mode keeps the throughput index independently of the GPUs in the cluster being 4, 6 or 8), opposite to the CUDA mode. Removing GPUs from some nodes of our cluster causes a very negative effect in the CUDA mode, as shown the same charts, increasing the execution time and reducing the global throughput.

We have analysed the percentage of time that the GPUs were allocated. The percentages have been extracted from the rCUDA executions and are given in the Table V. The coloured cells facilitate understanding of the results: the dark green cells mean that a GPUs has been allocated for a "long" time, and the red cells have the opposite meaning. Overall, we have a reliable indication that compaction of the jobs in the available GPUs is well conducted.

Regarding the execution times and the percentages of allocation, we can conclude that by reducing the number of GPUs and using the existing GPUs in a remote way, the jobs make the most of these resources, while the workload execution suffers little. The advantage of this technique lies in that it reduces the costs of hardware, which also implies a reduction in the energy consumption. Additionally, it can process more jobs per minute, which implies that our cluster has a higher profit-earning capacity.

TABLE V. PERCENTAGE OF ALLOCATED TIME

GPUs	Workload	GPU0	GPU1	GPU2	GPU3	GPU4	GPU5	GPU6	GPU7
8 GPUs Max Perf	2 hours	90.12	90.12	90.12	90.12	85.60	55.66	55.22	55.19
	4 hours	90.78	90.78	88.12	88.12	62.53	38.74	34.41	9.70
	8 hours	93.30	93.30	93.09	93.08	92.14	84.45	84.19	83.64
8 GPUs	2 hours	79.16	79.16	79.16	79.16	76.82	56.31	43.24	25.75
	4 hours	94.86	94.86	94.86	91.67	87.72	77.11	44.41	32.67
	8 hours	90.27	90.27	90.27	87.03	86.31	78.06	68.54	57.92
6 GPUs	2 hours	86.34	86.34	86.34	86.34	70.69	68.88	-	-
	4 hours	93.48	93.48	93.48	92.37	77.14	58.44	-	-
	8 hours	97.09	97.09	97.09	94.40	86.12	85.10	-	-
4 GPUs	2 hours	99.57	98.17	98.17	98.17	-	-	-	-
	4 hours	99.29	99.04	99.04	99.04	-	-	-	-
	8 hours	97.67	97.67	97.67	97.67	-	-	-	-

V. CONCLUSIONS

In this paper we have integrated new functionality to accommodate GPU virtualization in the SLURM job scheduler.

With these modifications, and using a remote GPU virtualization tool as rCUDA, the GPUs in an HPC cluster are logically decoupled from the nodes in which they are installed, so that they can then be accessed by jobs running in any cluster node. The additions/changes to SLURM include the definition of a new SLURM GRes resource, a reduced number of new resources that the scheduler can manage, and modifications (mainly in the form of options) to several SLURM commands.

We have also performed an extensive evaluation of the new functionality on a real cluster. For this purpose, we have defined a collection of synthetic workloads in order to demonstrate the functionality of our SLURM version as well as the real possibilities of reducing the number of GPUs installed on a cluster and its impact on performance.

Our current version of SLURM adopts scheduling decisions involving remote GPUs based on the amount of GPU memory required for the job. In a future implementation we plan to experiment with different scheduling algorithms, in order to take into account not only the amount of GPU memory required by a job, but also the actual computational intensity of the workload jobs, the GPU computational power, and the network distance between the application nodes and the remote GPU nodes to be assigned to a job. A detailed study on the performance-TCO trade-off is part of future work.

Acknowledgments

The researchers at UPV were supported by the the Generalitat Valenciana under Grant PROMETEOII/2013/009 of the PROMETEO program phase II. Researchers at UJI were supported by MINECO, by FEDER funds under Grant TIN2011-23283, and by the Fundacion Caixa-Castell Bancaixa (Grant P11B2013-21). The authors are also grateful for the generous support provided by Mellanox Technologies.

References

[1] NVIDIA. *The NVIDIA CUDA API Reference Manual Version 5*, NVIDIA 2012

[2] Khronos OpenCL Working Group. OpenCL 1.2 Specification, http://www.khronos.org/registry/cl/specs/opencl-1.2.pdf [3 June 2013]

[3] OpenACC. Directives for accelerators. http://www.openacc-standard.org/ [3 June 2014]

[4] A. Gaikwad and I. M. Toke. GPU based sparse grid technique for solving multidimensional options pricing PDEs, *Proceedings of the 2nd Workshop on High Performance Computational Finance* 2009, pp. 6:1–6:9

[5] D. P. Playne and K. A. Hawick. Data parallel three- dimensional CahnHilliard field equation simulation on GPUs with CUDA, in International Conference on Parallel and Distributed Processing Techniques and Applications, H. R. Arabnia, Ed., 2009, pp. 104110.

[6] S. Barrachina, M. Castillo, F. D. Igual, R. Mayo, E. S. Quintana-Orti and G. Quintana-Orti. Exploiting the capabilities of modern GPUs for dense matrix computations, Concurr. Comput. : Pract. Exper., vol. 21, no. 18, pp. 24572477, 2009.

[7] S. S. Stone, J. P. Haldar, S. C. Tsao, W. Hwu, Z.P . Liang, and B. P. Sutton. Accelerating advanced MRI reconstructions on GPUs, in Proceedings of the 2008 conference on Computing Frontiers (CF08), pp. 261272. ACM, New York, 2008.

[8] E. H. Phillips, Y. Zhang, R. L. Davis, and J. D. Owens. Rapid aerodynamic performance prediction on a cluster of graphics processing units, in Proceedings of the 47th AIAA Aerospace Sciences Meeting, no. AIAA 2009-565, Jan. 2009.

[9] Y. C. Luo and R. Duraiswami. Canny edge detection on NVIDIA CUDA, in Computer Vision on GPU, 2008.

[10] Antonio J. Peña et al. An efficient implementation of GPU virtualization in high performance clusters, in Euro-Par Workshops, 2009

[11] Antonio J. Peña et al. Performance of CUDA virtualized remote GPUs in high performance clusters, in ICPP, 2011

[12] Giunta, Giulio et al. A GPGPU transparent virtualization component for high performance computing clouds, Euro-Par, 2010

[13] Oikawa, Minoru et al. DS-CUDA: a middleware to use many GPUs in the cloud environment, in SC, 2012

[14] Lin Shi et al. vCUDA: GPU accelerated high performance computing in virtual machines, IPDPS, 2009

[15] Gupta, Vishakha et al. GViM: GPU-accelerated virtual machines, in HPCVirt, 2009

[16] Tyng-Yeu Liang et al. GridCuda: a grid-enabled CUDA programming toolkit, WAINA, 2011

[17] Zillians, Inc., V-GPU: GPU virtualization, 2013, http://www.zillians.com/vgpu

[18] Kim, Jungwon et al. SnuCL: an OpenCL framework for heterogeneous CPU/GPU clusters, ICS, 2012

[19] Kegel, P. et al. dOpenCL: towards a uniform programming approach for distributed heterogeneous multi-many-core systems, 2012

[20] Shucai Xiao et al. VOCL: an optimized environment for transparent virtualization of graphics processing units. InPar, 2012

[21] Barak, A. et al. A package for OpenCL based heterogeneous computing on clusters with many GPU devices, CLUSTER, 2010

[22] C. Reano, R. Mayo, E.S. Quintana-Orti F. Silla, J. Duato A.J. Pena. Influence of InfiniBand FDR on the Performance of Remote GPU Virtualization, CLUSTER 2013

[23] A. B. Yoo, M. A. Jette and M. Grondona. SLURM: Simple Linux Utility for Resource Management, in Job Scheduling Strategies for Parallel Processing, in Dror G. Feitelson, Larry Rudolph, and Uwe Schwiegelshohn, Eds., pp. 4460. Springer Verlag, 2003, Lect. Notes Comput. Sci. vol. 2862.

[24] B. Nitzberg , J. M. Schopf , J. P. Jones. PBS Pro: Grid computing and scheduling attributes, Grid resource management: state of the art and future trends, Kluwer Academic Publishers, Norwell, MA, 2004

[25] Moab Workload Manager Documentation, http://www.adaptivecomputing.com/resources/docs/

[26] Torque Resource Manager Documentation, http://www.adaptivecomputing.com/resources/docs/

[27] LSF (Load Sharing Facility) Features and Documentation, http://www.platform.com/workload-management/high-performance-computing

[28] N. Capit, G.D. Costa, Y. Georgiou, G. Huard, C. Martin, G. Mounie, P. Neyron, and O. Richard. A batch scheduler with high level components, in 5th Int. Symposium on Cluster Computing and the Grid, Cardiff, UK, 2005, pp. 776783, IEEE.

[29] B. Bode, D.M. Halstead, R. Kendall, Z. Lei, and D. Jackson. The Portable Batch Scheduler and the Maui Scheduler on Linux Clusters, in Proceedings of the 4th Annual Showcase and Conference (LINUX-00), Berkeley, CA, 2000, pp. 217224, The USENIX Association.

[30] S. Kannan, M. Roberts, P. Mayes, D. Brelsford, and J. F. Skovira. Workload Management with LoadLeveler. IBM, rst ed., Nov 2001. ibm.com/redbooks

[31] T. Tannenbaum, D. Wright , K. Miller , M. Livny. Condor: a distributed job scheduler, Beowulf cluster computing with Linux, MIT Press, Cambridge, MA, 2001

[32] W. Gentzsch. Sun Grid Engine: Towards Creating a Compute Power Grid, in Proc. First IEEE International Symposium on Cluster Computing and the Grid (1st CCGRID01), Brisbane, Australia, May 2001, pp. 3539, IEEE Computer Society (Los Alamitos, CA).

[33] http://slurm.schedmd.com

[34] Seren Soner, Can zturan. Integer Programming Based Heterogeneous CPU-GPU Cluster Scheduler for SLURM Resource Manager

[35] http://www.nvidia.com/object/gpu-applications.html