Increasing the Performance of Data Centers by Combining Remote GPU Virtualization with Slurm

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

***Abstract*—The use of Graphics Processing Units (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. Using the virtual GPUs provided by the remote GPU virtualization mechanism may address the concerns associated with the use of these devices. However, in the same way as workload managers map GPU resources to applications, virtual GPUs should also be scheduled before job execution. Nevertheless, current workload managers are not able to deal with virtual GPUs. In this paper we analyze the performance attained by a cluster using the rCUDA remote GPU virtualization middleware and a modified version of the Slurm workload manager, which is now able to map remote virtual GPUs to jobs. Results show that cluster throughput is doubled at the same time that total energy consumption is reduced up to 40%. GPU utilization is also increased.**

***Keywords*-GPGPU; CUDA; HPC; virtualization; InfiniBand; data centers; Slurm; rCUDA**

I. Introduction

GPUs (Graphics Processing Units) are used to reduce the execution time of applications. However, the use of GPUs is not exempt from side effects. For instance, when an MPI (Message Passing Interface) application not requiring GPUs is executed, it will typically spread across several nodes of the cluster using all the CPU cores available in them. At that point, the GPUs in the nodes executing the MPI application will not be available for other applications given that all the CPU cores are busy.

Another important concern related to the use of GPUs in clusters is has to do with the way that workload managers like Slurm [11] track the accounting of resources. These workload managers use a fine granularity for resources such as CPUs but not for GPUs. In this regard, they can assign individual CPU cores to different applications, what allows a shared usage of the CPU sockets present in a server among several applications. On the contrary, workload managers use a per-GPU granularity. That is, the entire GPU is assigned to an application in an exclusive way, thus hindering the possibility of sharing these accelerators among several applications even if those accelerators present enough resources for all the applications.

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. 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 those GPUs concurrently to several applications as far as GPUs have enough resources for all of them. Notice, however, that workload managers need to be enhanced in order to deal with 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.

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 used the rCUDA [7] remote GPU virtualization middleware along with a modified version of Slurm [3], which is now able to dispatch GPU-accelerated applications to nodes not owning a GPU and, therefore, the use of a remote (or virtual) GPU must be scheduled.

II. Performance Analysis

In this section we study the impact that using the remote GPU virtualization mechanism has on the performance of a data center. To that end, we have executed several workloads in a cluster by submitting a series of randomly selected 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

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

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

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, a modified version of Slurm [3] 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 that of a cluster using rCUDA.

In order to present the performance analysis, we first present the cluster configuration and the workloads used in the experiments.

A. *Cluster Configuration*

The testbed used in this study is based on the use of a cluster composed of 16 1027GR-TRF Supermicro servers. Each of the 16 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 (InfiniBand FDR compatible) to exchange data at a maximum rate of 56 Gb/s. Furthermore, an NVIDIA Tesla K20 GPU is installed at each node. One additional node (without GPU) has been leveraged to execute the central Slurm daemon responsible for scheduling jobs.

B. *Workloads*

Several workloads have been considered in order to provide a more representative range of results. The workloads are composed of the following applications: GPU-BLAST [10], LAMMPS [1], mCUDA-MEME [6], GROMACS [9], BarraCUDA [5], MUMmerGPU [4], GPU-LIBSVM [2], and NAMD [8]. Table I provides additional information about the applications used in this work, such as the exact execution configuration used for each of the applications, their execution time, and the GPU memory required by each application. For those applications composed of several processes or threads, the amount of GPU memory depicted in Table I refers to the individual needs of each particular thread or process. Notice that the amount of GPU memory is not specified for the GROMACS and NAMD applications because we are using non-accelerated versions of these applications. The reason for this choice is simply to increase the heterogeneity degree of the workloads by using some CPU-only applications, as it could be the case in many data centers. The previous applications have been combined in order to create three different workloads as shown in Table II.

As can be seen, the eight applications used present different characteristics, not only in the amount of processes and threads used by each of them and their execution time but they also present different GPU usage patterns, what includes both memory copies to/from GPUs and also kernel executions. Therefore, although the set of applications considered is finite, it may provide a representative sample of a workload typically found in current data centers. Actually, the set of applications in Table I could be considered from two different 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 exact problem addressed by each application is not the focus but applications are seen as processes that keep CPUs and GPUs busy during some amount of time and require some amount of memory. Now the focus is the amount of resources required by each application and the time that those resources are kept busy. From this second perspective, the set of applications in Table I becomes even more representative.

Table III displays the Slurm parameters used for launching each of the applications. The use of real and virtual GPUs has been considered in the table. Notice that once Slurm has been enhanced, Slurm users are able to submit jobs to the system queues in three different modes: (1) CUDA:

Table II: 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

Table III: Slurm launching parameters

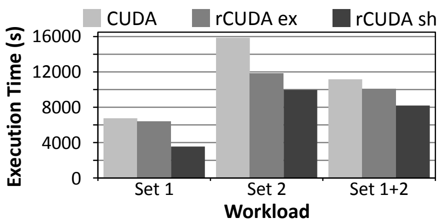
Application	Launch with CUDA	Launch with rCUDA exclusive	Launch with rCUDA shared
GPU-Blast	-N1 -n1 -c6 –gres=gpu:1	-n1 -c6 –rcuda-mode=excl –gres=rgpu:1	-n1 -c6 –rcuda-mode=shar –gres=rgpu:1:1599M
LAMMPS	-N4 -n4 -c1 –gres=gpu:1	-n4 -c1 –rcuda-mode=excl –gres=rgpu:4	-n4 -c1 –rcuda-mode=shar –gres=rgpu:4:876M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

no change is required to the original way of launching jobs. (2) rCUDA shared: the job will use the remote virtual GPUs, which will be shared with other jobs, and (3) rCUDA exclusive: the job will use the new remote virtual GPUs but it will not share them with other jobs. In the first case, CUDA will be used (column labeled "Launch with CUDA"). In the second and third cases, rCUDA will be leveraged. 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 in the submission command. In the third case, the column labeled as "Launch with rCUDA exclusive" shows that no GPU memory is declared because the GPU assigned to a given job will be not shared with other jobs.

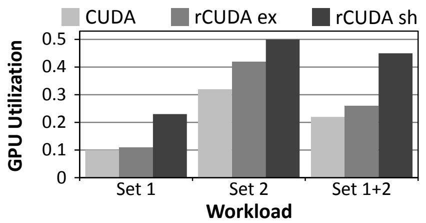
C. Performance Analysis

Figure 1 shows, for each of the workloads depicted in Table II, the performance when CUDA is used along with the original Slurm job scheduler (results labeled as "CUDA") as well as the performance when rCUDA is used in combination with the modified version of Slurm. 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 can be shared among several applications. Among both rCUDA uses, the shared one is the most interesting one. The exclusive case is considered in this paper only for comparison purposes. Figure 1(a) shows total execution time for each of the workloads. Figure 1(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 1(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 1(c).

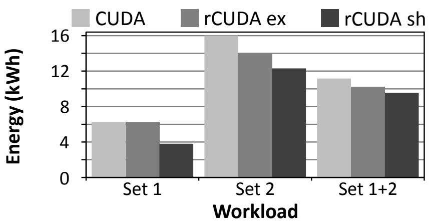
As can be seen in Figure 1(a), workload "Set 1" presents the smallest execution time, given that it is composed of the applications requiring the smallest execution times. Furthermore, using rCUDA in a shared way reduces execution time for the three workloads. In this regard, execution time is reduced 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 1(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



(a) Total execution time of the workloads.



(b) Average GPU utilization.



(c) Total energy consumed.

Figure 1: Performance results from the 16-node 16-GPU cluster.

use of CUDA. Finally, total energy consumption is reduced accordingly, as shown in Figure 1(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 I: on the one hand, the other accelerated applications cannot access the GPUs in those nodes because they are busy and, on the other hand, the non-GPU applications (GROMACS and NAMD) cannot use those nodes because they require all the CPU cores and LAMMPS and mCUDA-MEME already took one core. However, when GPUs are shared among several applications, GPUs assigned to LAMMPS and mCUDA-MEME can 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 1.

The second reason for the improvements shown in Figure 1 is related to the usage that applications make of GPUs. As Table I showed, some applications do not completely exhaust GPU memory resources. For instance, applications mCUDA-MEME and GPU-LIBSVM only use about 3% of the memory present in the NVIDIA Tesla K20 GPU. However, the unmodified version of Slurm (combined with CUDA) will allocate the entire GPU for executing each of these applications, thus causing that almost 100% of the GPU memory is wasted during application execution. This concern is also present for other applications in Table I. Moreover, if NVIDIA Tesla K40 GPUs were used instead of the NVIDIA Tesla K20 devices employed in this study, then this memory underutilization would be worse because the K40 model features 12 GB of memory instead of the 5 GB of the Tesla K20 devices. 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.

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 1. 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 about the service received.

III. 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 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.

Acknowledgment

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

References

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

[2] C. C. Chang et al., “Libsvm: A library for support vector machines,” *ACM Trans. Intell. Syst. Technol.*, 2011

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

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

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

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

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

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

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

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

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