Enabling GPU Virtualization in Cloud Environments

Sergio Iserte, Francisco J. Clemente-Castelló, Adrián Castelló,
Rafael Mayo and Enrique S. Quintana-Ortí

Department of Computer Science and Engineering, Universitat Jaume I, Castelló de la Plana, Spain

Keywords: Cloud Computing, GPU Virtualization, Amazon Web Services (AWS), OpenStack, Resource Management.

Abstract: The use of accelerators, such as graphics processing units (GPUs), to reduce the execution time of compute-intensive applications has become popular during the past few years. These devices increment the computational power of a node thanks to their parallel architecture. This trend has led cloud service providers as Amazon or middlewares such as OpenStack to add virtual machines (VMs) including GPUs to their facilities instances. To fulfill these needs, the guest hosts must be equipped with GPUs which, unfortunately, will be barely utilized if a non GPU-enabled VM is running in the host. The solution presented in this work is based on GPU virtualization and shareability in order to reach an equilibrium between service supply and the applications' demand of accelerators. Concretely, we propose to decouple real GPUs from the nodes by using the virtualization technology rCUDA. With this software configuration, GPUs can be accessed from any VM avoiding the need of placing a physical GPUs in each guest host. Moreover, we study the viability of this approach using a public cloud service configuration, and we develop a module for OpenStack in order to add support for the virtualized devices and the logic to manage them. The results demonstrate this is a viable configuration which adds flexibility to current and well-known cloud solutions.

1 INTRODUCTION

Nowadays, many cloud vendors have started offering virtual machines (VMs) with graphics processing units (GPUs) in order to provide GPGPU (general-purpose GPU) computation services. A few relevant examples include Amazon Web Services (AWS)[1], Penguin Computing[2], Softlayer[3] and Microsoft Azure[4]. In the public scope, one of the most popular cloud vendors is AWS, which offers a wide range of preconfigured instances ready to be launched. Alternatively, owning the proper infrastructure, a private cloud can be deployed using a specific middleware such as OpenStack[5] or Opennebula[6].

Unfortunately, sharing GPU resources among multiple VMs in cloud environments is more complex than in physical servers. On one hand, instances in public clouds are not easily customizable. On the other, although the instances in a private cloud can be customized in many aspects, when referring to GPUs the number of options is reduced. As a result, neither vendors nor tools offer GPGPU services. Remote virtualization has been recently proposed to deal with the low-usage problem. Some relevant examples include rCUDA (Peña, 2013), DS-CUDA (Kawai et al., 2012), gVirtus (Giunta et al., 2010), vCUDA (Shi et al., 2012), VOCL (Xiao et al., 2012), and SnuCL (Kim et al., 2012). Roughly speaking these virtualization frameworks enable cluster configurations with fewer GPUs than nodes. The goal is that GPU-equipped nodes act as GPGPU servers, yielding a GPU-sharing solution that potentially achieves a higher overall utilization of the accelerators in the system.

The main goals of this work are to study current cloud solutions in an HPC GPU-enabled scenario, and to analyze and improve them by adding flexibility via GPU virtualization. In order to reach this goal, we select rCUDA, a virtualization tool that is possibly the more complete and up-to-date for NVIDIA GPUs.

The rest of the paper is structured as follows. In Section 2 we introduce the technologies used in this work; Section 3 summarizes related work; the effort

[1]https://aws.amazon.com
[2]http://www.penguincomputing.com
[3]http://www.softlayer.com
[4]https://azure.microsoft.com
[5]https://www.openstack.org
[6]http://opennebula.org

to use AWS is explained in Section 4; while the work with Openstack is described in Section 5; finally, Section 6 summarizes the advances and Section 7 outlines the next steps of this research.

2 BACKGROUND

2.1 The rCUDA Framework

rCUDA (Peña et al., 2014) is a middleware that enables transparent access to any NVIDIA GPU device present in a cluster from all compute nodes. The GPUs can also be shared among nodes, and a single node can use all the graphic accelerators as if they were local. rCUDA is structured following a client-server distributed architecture and its client exposes the same interface as the regular NVIDIA CUDA 6.5 release (NVIDIA Corp.,).With this middleware, applications are not aware that they are executed on top of a virtualization layer. To deal with new GPU programming models, rCUDA has been recently extended to accommodate directive-based models such as OmpSs (Castelló et al., 2015a) and OpenACC (Castelló et al., 2015b). The integration of remote GPGPU virtualization with global resource schedulers such as SLURM (Iserte et al., 2014) completes this appealing technology, making accelerator-enabled clusters more flexible and energy-efficient (Castelló et al., 2014).

2.2 Amazon Web Services

AWS (Amazon Web Services, 2015) is a public cloud computing provider, composed of several services, such as cloud-based computation, storage and other functionality, that enables organizations and/or individuals to deploy services and applications on demand. These services replace company-owned local IT infrastructure and provide agility and instant elasticity matching perfectly with enterprise software requirements.

From the point of view of HPC, AWS offers high performance facilities via instances equipped with GPUs and high performance network interconnection.

2.3 OpenStack

OpenStack (OpenStack Foundation, 2015) is a cloud operating system (OS) that provides Infrastructure as a Service (IaaS). OpenStack controls large pools of compute, storage, and networking resources throughout a datacenter. All these resources are managed through a dashboard or an API that gives administrators control while empowering their users to provision resources through a web interface or a command-line interface. OpenStack supports most recent hypervisors and handles provisioning and life-cycle management of VMs. The OpenStack architecture offers flexibility to create a custom cloud, with no proprietary hardware or software requirements, and the ability to integrate with legacy systems and third party technologies.

From the HPC perspective, OpenStack offers high performance virtual machine configurations with different hardware architectures. Even though in OpenStack it is possible to work with GPUs, the Nova project does not support this architecture yet.

3 STATE-OF-THE-ART

Our solutions to the deficiencies exposed in the previous section relies on GPU virtualization, sharing resources in order to attain a fair balance between supply and demand. While several efforts with the same goal have been initiated in the past, as exposed next, none of them is as ambitious as ours. The work in (Younge et al., 2013) allows the VM managed by the Xen hypervisor to access the GPUs in a physical node, but with this solution a node cannot use more GPUs than those locally hosted, and an idle GPU cannot be shared with other nodes. The solution presented by gVirtus (Giunta et al., 2010) virtualizes GPUs and makes them accessible for any VM in the cluster. However, this kind of virtualization strongly depends on the hypervisor, and so does its performance. Similar solution is presented in gCloud (Diab et al., 2013). While this solution is not yet integrated in a Cloud Computing Manager, its main drawback is that the application's code must be modified in order to run in the virtual-GPU environment. A runtime component to provide abstraction and sharing of GPUs is presented in (Becchi et al., 2012), which allows scheduling policies to isolate and share GPUs in a cluster for a set of applications. The work introduced in (Jun et al., 2014) is more mature; however, it is only focused on compute-intensive HPC applications.

Our proposal goes further, not only bringing solutions for all kind of HPC applications, but also aiming to boost flexibility in the use of GPUs.

4 GPGPU SHARE SERVICE WITH AWS

4.1 Current Features

4.1.1 Instances

An instance is a pre-configured VM focused on an specific target. Among the large list of instances offered by AWS, we can find specialized versions for general-purpose (T2, M4 and M3); computer science (C4 and C3); memory (R3); storage (I2 and D2) and GPU capable (G2). Each type of instance has its own purpose and cost (price). Moreover, each type offers a different number of CPUs as well as network interconnection, which can be: low, medium, high or 10Gb. For our study, we worked in the AWS availability zone US EAST (N. VIRGINIA). The instances available in that case present the features reported in Table 1.

Table 1: Shown HPC instances available in US EAST (N. VIRGINIA) in June 2015.

Name	vCPUs	Memory	Network	GPUs	Price
c3.2xlarge	8	15 GiB	High	0	$ 0.42
c3.8xlarge	32	60 GiB	10 Gb	0	$ 1.68
g2.2xlarge	8	15 GiB	High	1	$ 0.65
g2.8xlarge	32	60 GiB	10 Gb	4	$ 2.6

For the following experiments, we select C3 family instances, which are not equipped with GPUs, as clients; whereas instances of the G2 family will act as GPU-enabled servers.

4.1.2 Networking

Table 1 shows that each instance integrates a different network. As the bandwidth is critical when GPU virtualization is applied, we first perform a simple test to verify the real network bandwidth.

Table 2: IPERF results between selected instances.

Server	Client	Network	Bandwidth
g2.8xlarge	c3.2xlarge	High	1 Gb/s
g2.8xlarge	c3.8xlarge	10Gb	7.5 Gb/s
g2.8xlarge	g2.2xlarge	High	1 Gb/s
g2.8xlarge	g2.8xlarge	10Gb	7.5 Gb/s

To evaluate the actual bandwidth, we executed the IPERF[7] tool between the instances described in Table 1, with the results shown in Table 2. From this experiment, we can derive that network "High" corresponds to a 1 Gb interconnect while "10 Gb" has a real bandwidth of 7.5 Gb/s. Moreover, it seems that the bandwidth of the instances equipped with a "High" interconnection network is constrained by software to 1 Gb/s since the theoretical and real bandwidth match perfectly. The real gap between sustained and theoretical bandwidth can be observed with the 10 Gb interconnection, which reaches up to 7.5 Gb/s.

[7]http://iperf.fr/

4.1.3 GPUs

An instance relies on a VM that runs on a real node with its own virtualized components. Therefore AWS can leverage a virtualization framework to offer GPU services to all the instances. Although the `nvidia-smi` command indicates that the GPUs installed are NVIDIA GRID K520, we need to verify that these are non-virtualized devices. For this purpose, we execute the NVIDIA SDK `bandwidthtest`. As shown in Table 3, the bandwidth achieved in this test is higher than the network bandwidth, which suggests that the accelerator is an actual GPU.

Table 3: Results of `bandwidthtest` transferring 32MB using pageable memory in a local GPU.

Name	Data Movement	Bandwidth
g2.2xlarge	Host to Device	3,004 MB/s
g2.2xlarge	Device to Host	2,809 MB/s
g2.8xlarge	Host to Device	2,814 MB/s
g2.8xlarge	Device to Host	3,182 MB/s

4.2 Testbed Scenarios

All scenarios are based on the maximum number of instances that a user can freely select without submitting a formal request. In particular, the maximum number for "g2.2xlarge" is 5; for "g2.8xlarge" it is 2. Ant the instances operate the RHEL 7.1 64-bit OS and version 6.5 of CUDA. We design three configuration scenarios for our tests:

- Scenario A (Figure 1(a)) shows a common configuration in GPU-accelerated clusters, with each node populated with a single GPU. Here, a node can access 5 GPUs using the "High" network.
- Scenario B (Figure 1(b)) is composed of 2 server nodes, equipped with 4 GPUs each, and a GPU-less client. This scenario includes a 10Gb network, and the client can execute the application using up to 8 GPUs.

- Scenario C (Figure 1(c)) combines scenarios A and B. A single client, using a 1Gb network interconnection, can leverage 13 GPUs as if they were local.

Once the scenarios are configured from the point of view of hardware, the rCUDA middleware needs to be installed in order to add the required flexibility to the system. The rCUDA server is executed in the server nodes and the rCUDA libraries are invoked from the node that acts as client.

In order to evaluate the network bandwidth using a remote GPU, we re-applied NVIDIA SDK `bandwidthtest`. Table 4 exposes that the bandwidth is limited by the network.

Table 4: Results of `bandwidthtest` transferring 32MB using pageable memory in a remote GPU using rCUDA.

Scenario	Data Movement	Network	Bandwidth
A	Host-to-Device	High	127 MB/s
A	Device-to-Host	High	126 MB/s
B	Host-to-Device	10 Gb	858 MB/s
B	Device-to-Host	10 Gb	843 MB/s

4.3 Experimental Results

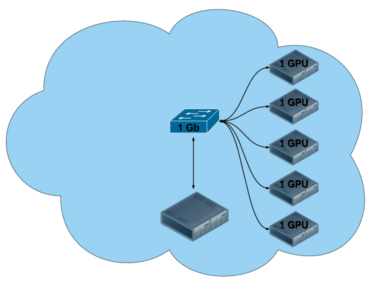
The first application is `MonteCarloMultiGPU`, from the NVIDIA SDK, a code that is compute bound (its execution barely involves memory operations). This was launched with the default configuration, "scaling=weak", which adjusts the size of the problem depending on the number of accelerators. Figure 2 depicts the options per second calculated by the application running on the scenarios in Figure 1 as well as using local GPUs. For clarity, we have removed the results observed for Scenario B as they are exactly the same as those obtained from Scenario C with up to 8 GPUs. In this particular case, rCUDA (remote GPUs) outperforms CUDA (local GPUs) because the former loads the libraries when the daemon is started (Peña, 2013). With rCUDA we can observe differences in the results between both scenarios. Here, Scenario A can increase the throughput because the GPUs do not share the PCI bus with other devices as each node only is equipped with one GPU. On the other hand, when the 4-GPU instances ("g2.8xlarge") are added (Scenario C), the PCI bus constrains the communication bandwidth, hurting the scalability.

The second application, LAMMPS[8], is a classical molecular dynamics simulator that can be applied at the atomic, meso, or continuum scale. From the implementation perspective, this multi-process application employs at least one GPU to host its processes, but can benefit from the presence of multiple GPUs. Figure 3(a) shows that, for this application, the use of remote GPUs does not offer any advantage over the original CUDA. Furthermore, for the execution on remote GPUs, the difference between both networks is small, although, the results observed with the "High" network are worse than those obtained with the "10 Gb" network. In execution of LAMMPS on a larger problem (see Figure 3(b)), CUDA still performs better, but the interesting point is the execution time when using remote GPUs. These are almost the same even with different networks, which indicates that the transfers turn the interconnection network into a bottleneck. For this type of application, enlarging the problem size compensates the negative effect of a slower network.

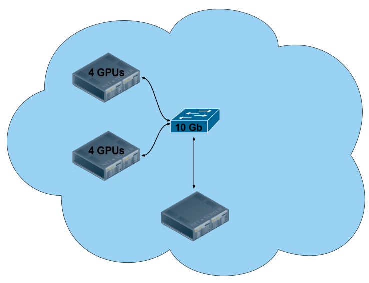
4.4 Discussion

The previous experiments reveal that the AWS GPU-instances are not appropriate for HPC because neither the network nor the accelerators are powerful enough to deliver high performance when running compute-intensive parallel applications. As (Peña, 2013) demonstrates, network and device types are critical factors to performance. In other words, AWS is more oriented toward offering a general-purpose service than to provide high performance. Also, AWS fails in offering flexibility as it enforces the user to choose between a basic instance with a GPU and a powerful instance with 4 GPUs. Table 1 shows that the resources of the "g2.8xlarge" are quadrupled, but so is the cost per hour. Therefore, in the case of having other necessities (instances type), using GPU virtualization technology we could in principle attach an accelerator to any type of instance. Furthermore, reducing the budget spent in cloud services is possible by customizing the resources of the available instances. For example, we can work on an instance with 2 GPUs for only \$ 1.3 by launching 2 "g2.2xlarge" and using remote GPUs, avoiding to pay the double for features that we do not need in "g2.8xlarge". In terms of GPU-shareability, AWS reserves GPU-capable physical machines which will be waiting for a GPU-instance request. Investing in expensive accelerators to keep them in a standby state is counter-productive. It makes more sense to dedicate less devices, accessible from any machine, resulting in a higher utilization rate.

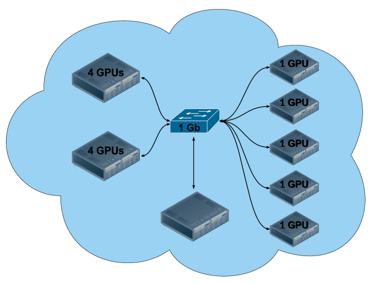
[8]http://lammps.sandia.gov



(a) 5 remote GPUs over 1GbE.



(b) 8 remote GPUs over 10GbE.



(c) 13 remote GPUs over 1GbE.

Figure 1: Configurations where an AWS instance, acting as a client, employs several remote GPUs.

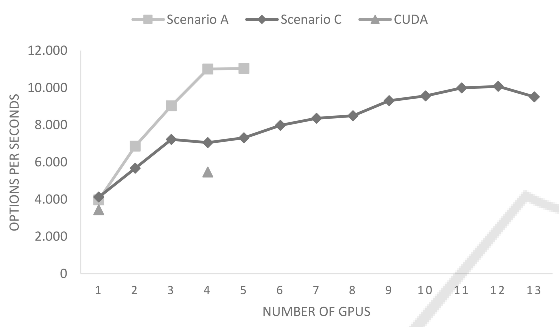
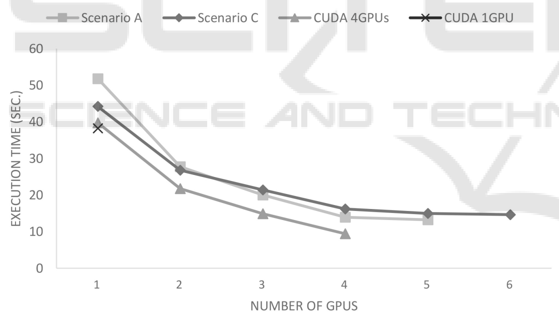
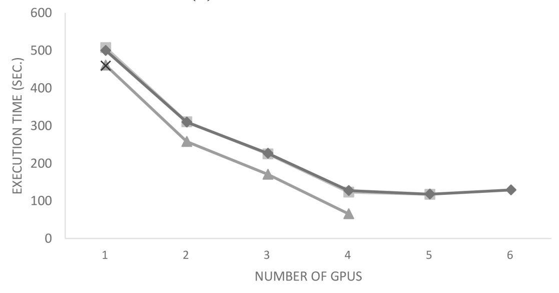


Figure 2: NVIDIA SDK `MonteCarloMultiGPU` execution using local (CUDA) and remote (Scenarios A and C) GPUs.



(a) Run size = 100.



(b) Run size = 2000.

Figure 3: Execution time of LAMMPS.

5 GPGPU SHARED SERVICE IN OPENSTACK

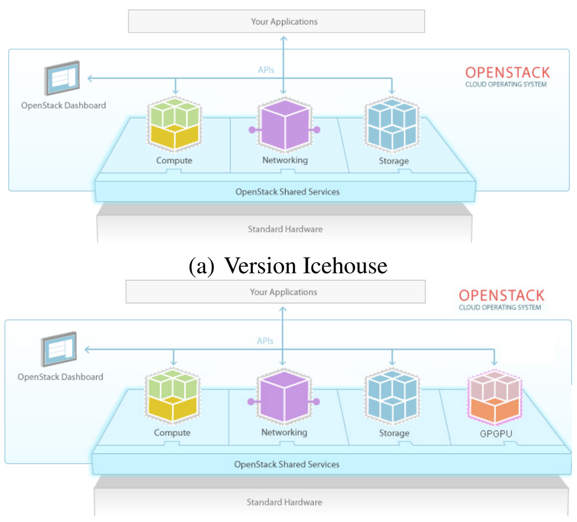
5.1 Managing Remote GPUs with OpenStack

The main idea is to evolve from the original OpenStack architecture (see Figure 4(a)) to a solution where a new shared service, responsible for the GPU accelerators, becomes integrated into the architecture (see Figure 4(b)). This new service brings more flexibility when managing GPUs and new working modes for GPGPU computation in the Cloud. As illustrated in Figure 5, we alter the original OpenStack Dashboard with a new parser, which splits the HTTP query in order to make use of both the GPGPU API for GPU-related operations, and the Nova API for the rest of the computations. The new GPGPU Service grants access to GPUs in a VM, but also allows the creation of "GPU-pools", consisting of a set of independent GPUs (logically disattached from the nodes) that can be assigned to one or more VMs.

Thanks to the modular approach of the GPGPU Service, the Nova Project does not need to be modified, and the tool can be easily ported to other Cloud Computing solutions.

5.2 Working Modes

The developed module allows users to set up any of the scenarios displayed in Figure 6. The users are given two configuration options to decide whether a physical GPU will be completely reserved for an instance (first column) or the instance will address a partition of the GPU as if it were a real device (second column). We refer to this as the "mode", with possible values being: "exclusive" or "shared". Let us assume there are 4 GPU devices in the cluster (independently of where they were hosted). An example of these scenarios is shown in Figure 6. There, in the "exclusive"



(a) Version Icehouse

(b) With GPGPU module

Figure 4: OpenStack Architecture.

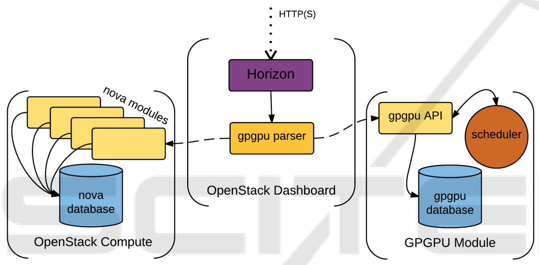


Figure 5: Internal Communication among modules.

mode the instance monopolizes all the GPUs; while in the "shared" mode, the GPUs are partitioned. As a result of sharing the GPU memory, the instance will be able to work with up to 8 GPUs, provided that each partition can be addressed as an independent GPU.

Moreover, the users are also responsible for deciding whether a GPU (or a pool) will be assigned to other instances. This behavior is refereed as "scope", and it determines that a group of instances is logically connected to a pool of GPUs. Working with the "public" scope (bottom row of Figure 6) implies that the GPUs of a pool can be used simultaneously by all the instances linked to it. Again, the GPU pool can be composed of "exclusive" or "shared" GPUs.

5.3 User Interface

In order to deal with the new features, several modifications have been planned in the OpenStack Dashboard, they have not been implemented yet, though. First of all, the Instance Launch Panel should be extended with a new field, where the user could assign an existing GPU pool, create a new one, or keep the

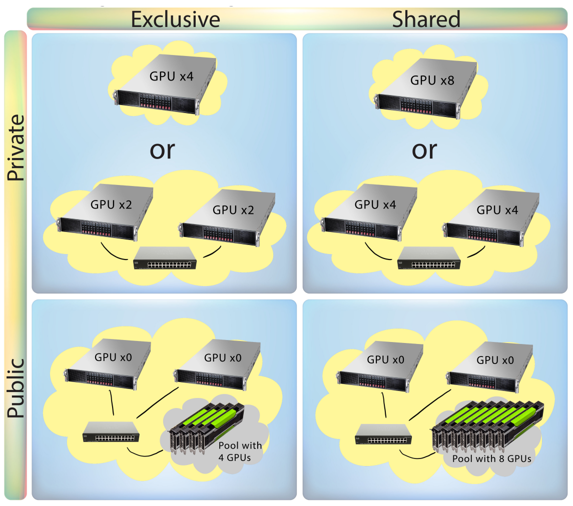
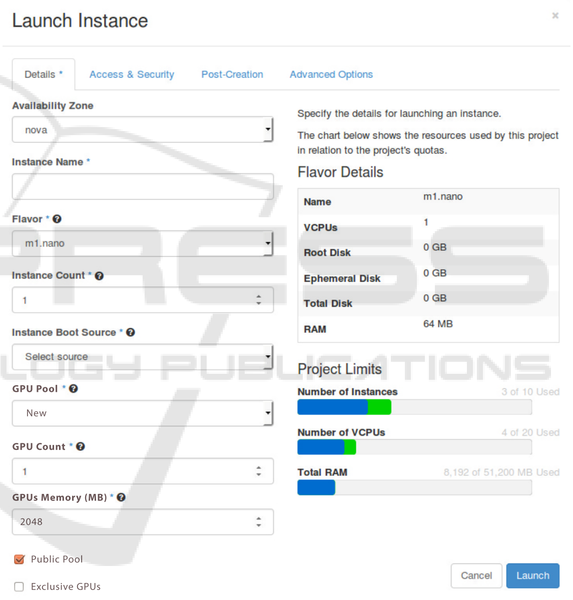


Figure 6: Examples of working modes.


Launch Instance

Details * | Access & Security | Post-Creation | Advanced Options

Availability Zone	nova
Instance Name *	
Flavor *	m1.nano
Instance Count *	1
Instance Boot Source *	Select source
GPU Pool *	New
GPU Count *	1
GPUs Memory (MB) *	2048

[✓] Public Pool
[ ] Exclusive GPUs

Specify the details for launching an instance.

The chart below shows the resources used by this project in relation to the project's quotas.

Flavor Details

Name	m1.nano
VCPUs	1
Root Disk	0 GB
Ephemeral Disk	0 GB
Total Disk	0 GB
RAM	64 MB

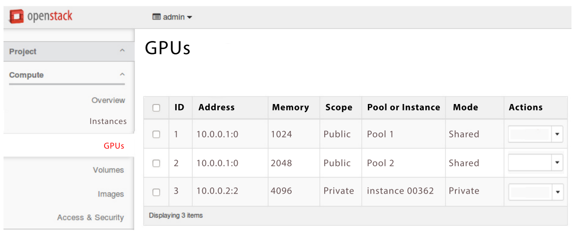
Project Limits

Number of Instances	3 of 10 Used
Number of VCPUs	4 of 20 Used
Total RAM	8,192 of 51,200 MB Used

Cancel Launch

Figure 7: Launching Instances and assigning GPUs.

instance without accelerators of this kind. When the option "New GPU Pool" is chosen, fields for the pool configuration would appear (see Figure 7). Furthermore, a new panel with the existent GPUs displays all the information related to GPUs (see Figure 8).



GPUs

ID	Address	Memory	Scope	Pool or Instance	Mode	Actions
1	10.0.0.1:0	1024	Public	Pool 1	Shared	
2	10.0.0.1:0	2048	Public	Pool 2	Shared	
3	10.0.0.2:2	4096	Private	instance 00362	Private	

Displaying 3 items

Figure 8: GPU Information Panel.

5.4 Experimental Results

Following tests were executed on 2 sets of nodes using a 1Gb Ethernet network. All the nodes composed by an Intel Xeon E7420 quadcore processor, at 2.13 GHz, and 16 GB DDR2 RAM at 667 MHz.

The first set, in charge of providing the cloud environment, consisted of 3 nodes. To deploy an IaaS, we used OpenStack Icehouse version, and QEMU/KVM 0.12.1 as the hypervisor. A fully-featured OpenStack deployment requires at least three nodes: a controller manages the compute nodes where the VMs are hosted; a network node manages the logic virtual network for the VMs, and one or more compute nodes run the hypervisor and VMs.

The second set, composed of 4 nodes, were auxiliary servers with a Tesla C1060 GPU each. The OS was a Centos 6.6; the GPUs used CUDA 6.5; and rCUDA v5.0 as GPU virtualization framework.

We have designed 6 different set-ups which can be divided into 2 groups: exclusive and shared GPUs. The exclusive mode provides, at most, 4 accelerators. The number of available GPU in shared mode will depend on the partition size. In this case, we halved the GPU memory, resulting 8 partitions that can be addressed as independent GPUs. For each group, we deployed virtual clusters of 1, 2 and 3 nodes, where the application processes were executed. The instances were launched with the OpenStack predefined flavor m1.medium, which determines a configuration of VMs consisting of 2 cores and 4 GB of memory.

MCUDA-MEME (Liu et al., 2010) was the application selected to test the set-ups. Thus is an MPI software, where each process must have access to a GPU. Therefore, the number of GPUs determines the maximum number of processes we can launch. Figure 9 compares the execution time of the application with different configurations over different setups. We used up to 3 nodes to spread the processes and launched up to 8 processes (only 4 in exclusive mode), one per remote GPU. We can first observe that the performance is higher with more than one node, because the traffic network is distributed among the nodes. In addition, the shortest execution time is obtained by both modes (exclusive and shared) when running their maximum number of processes with more than one node. This seems to imply that it is not worth to scale (increase) the number of resources, because the performance growth rate is barely increasing. Although, the time is lower when the GPUs are shared, the setup cannot take advantage of an increase in the number of devices.

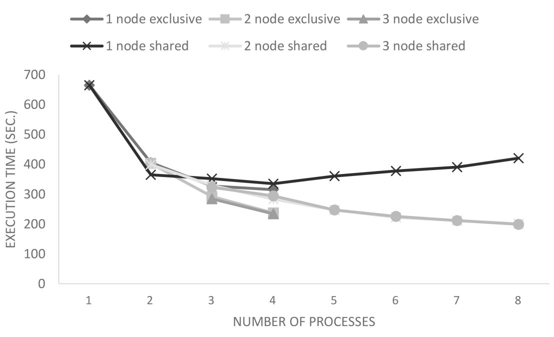


Figure 9: Scalability results of MCUDAMEME with a different number of MPI processes.

5.5 Discussion

The network interconnect restricts the performance of our executions. The analysis in (Peña, 2013) reveals that improving the network infrastructure can make a big different for GPU virtualization.

The most remarkable achievement is the wide range of possible configurations and the flexibility to adapt a system to fit the user requirements. In addition, with this virtualization technology, the requests for GPU devices can be fulfilled with small investment in infrastructure and maintenance. Energy can be saved not only thanks to the remote access and the ability to emulate several GPUs using only a few real ones, but also by consolidating the accelerators in a single machine (when possible), or turning down nodes when their GPUs are idle.

6 CONCLUSIONS

We have presented a complete study of the possibilities offered by AWS when it comes to GPUs. The constraints imposed by this service motivated us to deploy our own private cloud, in order to gain flexibility when dealing with these accelerators. For this purpose, we have introduced an extension of OpenStack which can be easily exploited to create GPU-instances as well as manage the physical GPUs to better profit.

As we expected, due to the limited bandwidth of the interconnects used in the experimentation, the performances observed for the GPU virtualized scenarios in the tests were quite low. On the other hand, we have created new operation modes that open interesting new ways to leverage GPUs in situations where having access to a GPU is more important than having a powerful GPU to boost the performance.

7 FUTURE WORK

The first item in the list of pending work is an upgrade of the network to an interconnect that is more prone to HPC. In particular, porting the setup and the tests to an infrastructure with an Infiniband network will shed light on the viability of this kind of solutions. Similar reasons, motivate us to try other Cloud vendors which better support for HPC. Looking for situations where performance is less important than flexibility will drive us to explore alternative tools to easily deploy GPU-programming computer labs.

Finally, an interesting future work is to design new strategies in order to decide where a remote GPUs is created and assigned to a physical device Concretely, to innovate scheduling policies can enhance the flexibility offered by the GPGPU module for OpenStack.

ACKNOWLEDGEMENTS

The authors would like to thank the IT members of the department Gustavo Edo and Vicente Roca for their help. This research was supported by Universitat Jaume I research project (P11B2013-21); and project TIN2014-53495-R from MINECO and FEDER. The initial version of rCUDA was jointly developed by Universitat Politècnica de València (UPV) and Universitat Jaume I de Castellón until year 2010. This initial development was later split into two branches. Part of the UPV version was used in this paper and it was supported by Generalitat Valenciana under Grants PROMETEO 2008/060 and Prometeo II 2013/009.

REFERENCES

Amazon Web Services (2015). Amazon web services. http://aws.amazon.com. Accessed: 2015-10.

Becchi, M., Sajjapongse, K., Graves, I., Procter, A., Ravi, V., and Chakradhar, S. (2012). A virtual memory based runtime to support multi-tenancy in clusters with GPUs. In *21st Int. symp. on High-Performance Parallel and Distributed Computing*.

Castelló, A., Duato, J., Mayo, R., Peña, A. J., Quintana-Ortí, E. S., Roca, V., and Silla, F. (2014). On the use of remote GPUs and low-power processors for the acceleration of scientific applications. In *The Fourth Int. Conf. on Smart Grids, Green Communications and IT Energy-aware Technologies*, pages 57–62, France.

Castelló, A., Mayo, R., Planas, J., and Quintana-Ortí, E. S. (2015a). Exploiting task-parallelism on GPU clusters via OmpSs and rCUDA virtualization. In *I IEEE Int. Workshop on Reengineering for Parallelism in Heterogeneous Parallel Platforms*, Helsinki (Finland).

Castelló, A., Peña, A. J., Mayo, R., Balaji, P., and Quintana-Ortí, E. S. (2015b). Exploring the suitability of remote GPGPU virtualization for the OpenACC programming model using rCUDA. In *IEEE Int. Conference on Cluster Computing*, Chicago, IL (USA).

Diab, K. M., Rafique, M. M., and Hefeeda, M. (2013). Dynamic sharing of GPUs in cloud systems. In *Parallel and Distributed Processing Symp. Workshops & PhD Forum, 2013 IEEE 27th International*.

Giunta, G., Montella, R., Agrillo, G., and Coviello, G. (2010). A GPGPU transparent virtualization component for high performance computing clouds. In *Euro-Par, Parallel Processing*, pages 379–391. Springer.

Iserte, S., Castelló, A., Mayo, R., Quintana-Ortí, E. S., Reaño, C., Prades, J., Silla, F., and Duato, J. (2014). SLURM support for remote GPU virtualization: Implementation and performance study. In *Int. Symposium on Computer Architecture and High Performance Computing*, Paris, France.

Jun, T. J., Van Quoc Dung, M. H. Y., Kim, D., Cho, H., and Hahm, J. (2014). GPGPU enabled HPC cloud platform based on OpenStack.

Kawai, A., Yasuoka, K., Yoshikawa, K., and Narumi, T. (2012). Distributed-shared CUDA: Virtualization of large-scale GPU systems for programmability and reliability. In *The Fourth Int. Conf. on Future Computational Technologies and Applications*, pages 7–12.

Kim, J., Seo, S., Lee, J., Nah, J., Jo, G., and Lee, J. (2012). SnuCL: an OpenCL framework for heterogeneous CPU/GPU clusters. In *Int. Conf. on Supercomputing (ICS)*.

Liu, Y., Schmidt, B., Liu, W., and Maskell, D. L. (2010). CUDA-MEME: Accelerating motif discovery in biological sequences using CUDA-enabled graphics processing units. *Pattern Recognition Letters*, 31(14).

NVIDIA Corp. *CUDA API Reference Manual Version 6.5*.

OpenStack Foundation (2015). OpenStack. http://www.openstack.org. Accessed: 2015-10.

Peña, A. J. (2013). *Virtualization of accelerators in high performance clusters*. PhD thesis, Universitat Jaume I, Castellon, Spain.

Peña, A. J., Reaño, C., Silla, F., Mayo, R., Quintana-Ortí, E. S., and Duato, J. (2014). A complete and efficient CUDA-sharing solution for HPC clusters. *Parallel Computer*, 40(10).

Shi, L., Chen, H., Sun, J., and Li, K. (2012). vCUDA: GPU-accelerated high-performance computing in virtual machines. *IEEE Trans. on Comput.*, 61(6).

Xiao, S., Balaji, P., Zhu, Q., Thakur, R., Coghlan, S., Lin, H., Wen, G., Hong, J., and Feng, W. (2012). VOCL: An optimized environment for transparent virtualization of graphics processing units. In *Innovative Parallel Computing*. IEEE.

Younge, A. J., Walters, J. P., Crago, S., and Fox, G. C. (2013). Enabling high performance computing in cloud infrastructure using virtualized GPUs.