IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received June 7, 2018, accepted July 5, 2018, date of publication July 12, 2018, date of current version August 7, 2018.

Digital Object Identifier 10.1109/ACCESS.2018.2855261

GSaaS: A Service to Cloudify and Schedule GPUs

SERGIO ISERTE[1], RAÚL PEÑA-ORTIZ[2], JUAN GUTIÉRREZ-AGUADO[2], JOSE M. CLAVER[2], (Senior Member, IEEE), AND RAFAEL MAYO[1]

[1]Department of Computer Science and Engineering, Universitat Jaume I, 12071 Castelló de la Plana, Spain
[2]Department of Computer Science, Universitat de València, 46100 Burjassot, Spain

Corresponding author: Sergio Iserte (siserte@uji.es)

This work was supported by the MINECO and FEDER under Projects TIN2014-53495-R and TIN2017-82972-R.

ABSTRACT Cloud technology is an attractive infrastructure solution that provides customers with an almost unlimited on-demand computational capacity using a pay-per-use approach, and allows data centers to increase their energy and economic savings by adopting a virtualized resource sharing model. However, resources such as graphics processing units (GPUs), have not been fully adapted to this model. Although, general-purpose computing on graphics processing units (GPGPU) is becoming more and more popular, cloud providers lack of flexibility to manage accelerators, because of the extended use of peripheral component interconnect (PCI) passthrough techniques to attach GPUs to virtual machines (VMs). For this reason, we design, develop, and evaluate a service that provides a complete management of cloudified GPUs (cGPUs) in public cloud platforms. Our solution enables an effective, anonymous, and transparent access from VMs to cGPUs that are previously scheduled and assigned by a full resource manager, taking into account new GPU selection policies and new working modes based on the locality of the physical accelerators and the exclusivity when accessing them. This easy-to-adopt tool improves the resource availability through different cGPUs configurations for end-users, whilst cloud providers are able to achieve a better utilization of their infrastructures and offer more competitive services. Scalability results in a real cloud environment demonstrate that our solution introduces a virtually null overhead in the deployment of VMs. Besides, performance experiments reveal that GPU-enabled clusters based on cloud infrastructures can benefit from our proposal not only exploiting better the accelerators, but also serving more jobs requests per unit of time.

INDEX TERMS Cloud computing, platform virtualization, networking, GPU cloudification, GPU resource management.

I. INTRODUCTION

The adoption of cloud computing in data centers offers new computational possibilities. From the end-user perspective, the cloud offers on-demand resources which can be easily provisioned and decommissioned on-the-fly. From the point of view of the cloud provider, virtualization, flexible resource availability, and shareability can lead to important economic competitiveness and a better exploitation of the infrastructure [1]. Although many types of resources have been adapted to this paradigm (i.e: CPUs, volatile/permanent memory devices, networks,. . .), others, such as GPUs, are not so flexible regarding cloud computing.

Hong *et al.* [2] discuss about how heterogeneous computing with GPUs can benefit the cloud computing model and support the idea of a need for a paradigm shift. Traditionally, the access to GPGPU in the cloud has been aimed to scientific applications. However, many areas would benefit from this approach, such as video encoding, sequencing in bioinformatics, scene rendering in remote gaming, or machine learning.

PCI passthrough [3] is the most common solution for providing GPU-enabled VMs, where a VM is configured with exclusive access to the PCI port of the accelerator. The PCI bandwidth is increasingly becoming the bottleneck at the multi-GPU system level, driving the need for new technology like NVIDIA NVLink [4], which provides higher bandwidth, more links, and improved scalability for multi-GPU/CPU system configurations. In this scenario, the number of GPUs that can be assigned to a VM is limited by the number of GPUs available in the physical node where the VM is allocated. Moreover, the GPU cannot be shared with other VMs which may lead to an under utilization of the accelerators.

Despite the rigidness of this solution, it is commonly adopted by cloud providers like Amazon Web Services (AWS) [5], Google Cloud [6] and Microsoft Azure [7], which are the current flagships in Infrastructure as a Service (IaaS).

Google Compute Engine provides GPUs that you can add to your virtual machine instances [8], whilst AWS and Azure offer GPU-capable VMs with support to CUDA and OpenCL through Amazon Elastic Compute Cloud (EC2) [9] and Azure VMs [10], respectively. Moreover, AWS and Google Cloud support the NVIDIA GPU Cloud (NGC) [11]. NGC provides researchers and data scientists with simple access to a comprehensive catalog of GPU-optimized software tools for deep learning and high performance computing (HPC) based on GPU-accelerated containers that take full advantage of NVIDIA GPUs. However, the deployed containers can only make use of local available GPUs and are oriented to specific deep learning applications.

More flexible approaches have been proposed to provide GPU access to non-GPU-enabled clients with the aim of accelerating graphics performance of applications. For instance, NVIDIA GRID GPU [12] and the Intel KVMGT [13] technology that implement complete GPU virtualization. Amazon EC2 Elastic GPU solution [14] allows end-users to easily attach low-cost graphics acceleration to a wide range of EC2 instances over the network. However, Elastic GPUs are only suited for applications that require a small or intermittent amount of additional GPU for graphics acceleration, and that use OpenGL graphics support and Microsoft Windows Server 2012 R2 or above.

Regarding GPGPU, several works have leveraged GPUs in a more general fashion. GVirtuS [15] and its predecessor gVirtuS [16] supply virtual GPUs (vGPUs) accessible from any virtual machine in a cloud-based cluster, but they only consider the private cloud paradigm. gCloud [17] is a similar solution, but it is not yet integrated in a cloud computing manager, and the application source code has to be adapted to run in the virtual environment. The solution presented by Jun *et al.* [18] shows some of its bottlenecks for intensive applications, and details of its integration in a cloud infrastructure are not given. Vgris [19] has not been tested on cloud infrastructures and only allows local access to GPUs. More recently, the benefits of the remote access to the GPUs using remote CUDA (rCUDA) [20] are evaluated in [21] presenting the tool as a real CUDA-enabling solution in cloud environments as well as in clusters. rCUDA does not virtualize GPUs, but provides remote access to them, henceforth referred as remote GPUs (rGPUs). Although some of these projects are showing a high rate of matureness, they have neglected their integration in real cloud platforms and the management of the GPU resources.

The main contribution of this paper is the design, implementation, and evaluation of a reliable service capable of deploying GPU-enabled VMs through a secure and flexible access to physical GPUs in public or private clouds; however security requirements can be relaxed in the private scenario. The proposed solution relies on:

- A new architectural component to provide VMs access to cGPUs in multitenant environments, hiding the real location of the accelerators and detaching their traffic from the VMs traffic.
- A complete cGPU resource management and scheduling system, which is an extension of our previous general purpose GPU management system [22]. It has been improved by including the necessary logic to support new working modes based on the locality of the physical GPUs and the exclusivity when accessing them. Furthermore, additional GPU selection policies, which take into account those new working modes, have been incorporated.

The developed prototype has been integrated in a cloud platform, OpenStack, and the evaluation process tackles a performance analysis that not only cover different type of applications, but also a thorough assessment of the infrastructure regarding cGPU-enabled VMs deployment time. Moreover, we demonstrate the benefits of integrating a shared service for cGPU resources in production environments through a series of simulations in different workload scenarios. Results suggest that our service can improve the availability and utilization of this type of accelerators.

The rest of the paper is organized as follows. Section II introduces the technologies used in this work. Section III describes the basis of the proposed service and the supported cGPUs working modes. Section IV presents a thorough evaluation that has been performed in order to justify the appropriateness of our work. Section V evaluates possibilities offered by the proposed service through simulating a batch of jobs which compete for the GPU resources. Finally, some concluding remarks and future work are drawn in Section VI.

II. BACKGROUND

This section describes the three main technologies used in this work, which are the following: the cloud platform selected to deploy our prototype, OpenStack; the framework adopted for accessing remotely to the GPUs from the virtual machines, rCUDA; and the service, GPGPUMS, integrated into the cloud infrastructure to manage registered GPUs.

A. OPENSTACK

OpenStack [23] is a well known cloud infrastructure that can be used to provide private, public, or hybrid IaaS. It relies on a set of logical elements such as scheduler, networking, compute, storage, image manager, etc. Each one of these elements is implemented as a set of distributed components that interchange information using a message middleware, RabbitMQ in our deployment. The management of virtual machines and networks can be done through a web-based interface (OpenStack Dashboard), through the command line client tools, or through HTTP APIs. The `Nova` service allows the provisioning of compute instances, whilst the `Neutron` service gives network connectivity among them by creating networks, subnetworks, routers, load balancers, etc. Particularly, these shared services have been relevant for the deployment of our service. Nevertheless, OpenStack includes an extensive list of services, such as `Storage`, which provides volumes that can be mounted in the instances

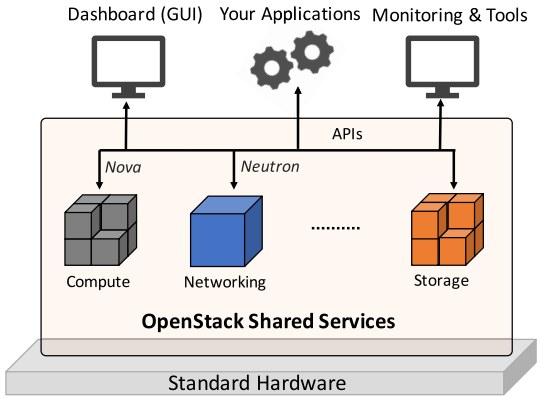


FIGURE 1. OpenStack shared services overview.

as block storage. Figure 1 depicts how these shared services are interconnected in OpenStack.

This cloud platform can be deployed over different underlying infrastructures, i.e., using physical or virtual machines, or a hybrid approach [24]. It also supports the most recent hypervisors, and its architecture offers flexibility to create an adaptable cloud, integrating legacy systems, and non proprietary hardware and software requirements. KVM [25] is one of the most extended, configurable, open source, and Linux compliant solutions among current hypervisors (VMware, Xen, vBox, etc). Besides, KVM supports the widest set of OpenStack features. For these reasons, OpenStack and KVM have been selected to deploy our prototype.

Commonly, cloud infrastructures enable VMs to access exclusively to local GPUs by using PCI passthrough. This limitation can be overcome by integrating accelerators with native support for virtualization, vGPUs, or by providing mechanisms to cloudify existing GPUs, cGPUs.

B. rCUDA

rCUDA [26] is a middleware that enables access to any NVIDIA GPU device present in a cluster from every compute node. 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 API.

Therefore, applications are not aware that they are executed on top this middleware. The integration of remote GPGPU virtualization with resource management systems (RMS) such as SLURM [27] completes this appealing technology, making accelerator-enabled clusters more flexible and energy-efficient.

rCUDA has been extensively tested on clusters and on private virtual environments [21] where the physical location on the resources can be exposed without risk. However, it does not fulfill several mandatory features when integrating in a cloud environment, such as anonymous location of the GPUs, autonomous configuration, GPGPU RMS, security issues, networking performance, etc.

C. GPGPUMS

Authors in [22] presented the general purpose GPU management system (GPGPUMS), a module for OpenStack that is in charge of managing remote access to a set of GPUs registered in the cloud infrastructure. This development leverages rCUDA to grant remote access to the GPUs in a provider network from any VM. For this purpose, it handles the petitions of deploying GPU-enabled VMs by scheduling the access to the accelerators.

GPGPUMS offers several working modes, being the most relevant for our work the *exclusive* and *shared* modes. A single VM has access to the assigned GPUs in the exclusive mode, whilst GPUs can be assigned to more than one VMs or several times to the same VM in the shared mode. In this regard, GPU memory is partitioned and each partition is referred as a rGPU. A rGPU monopolizes the whole GPU memory in the exclusive mode. However, the original implementation of GPGPUMS did not take into account GPU locality neither was designed to operate in a multitenant environment. In other words, it configured the VMs to access the accelerators through a secure shell command and it did not anonymize the location of the accelerators, what is inadmissible in public clouds.

These drawbacks motivate us to consider cloudification as the basis for GPGPUMS improvement and evolution as a cloud service by defining and implementing: i) new components required to deal with multitenancy in public clouds, and ii) new working modes that take into account not only the exclusivity when accessing the accelerators, but also their locality. For this reason, our solution turns rGPUs into cGPUs, being the later more appropriate for public cloud environments.

III. GPU SCHEDULING AS A SERVICE

GPU Scheduling as a Service (GSaaS) aims to provide a management layer between the GPUs and the cloud infrastructure. In this regard, GSaaS is completely integrated into the OpenStack architecture as a new shared service (see Figure 2) in charge of cloudifying and scheduling the access from VMs to physical GPUs.

Figure 3 shows the involved technologies in GSaaS. As described in Section II, rCUDA enables the access to GPUs that are previously scheduled and assigned by GPGPUMS. Additionally, resource-oriented distributed virtual routing (RODVR) is the key component that permits multitenancy by hiding GPU location and providing access to them as cGPUs. GSaaS allows a better integration in the cloud and prevents unauthorized accesses to physical GPUs, because it is impossible to access the accelerators from VMs that have been launched outside the proposed service.

The GSaaS components integration into the control plane of the cloud infrastructure is depicted in Figure 4.

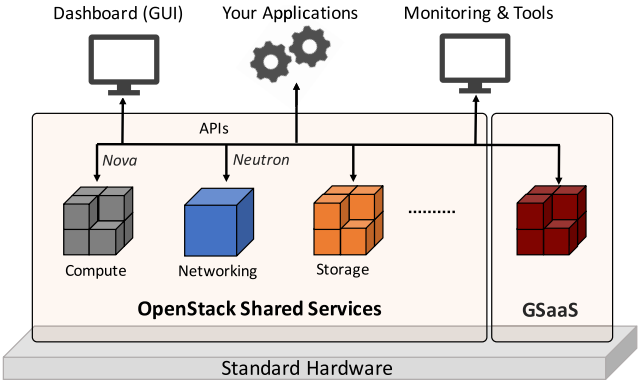


FIGURE 2. GSaaS: A service to cloudify and schedule GPU access in OpenStack.

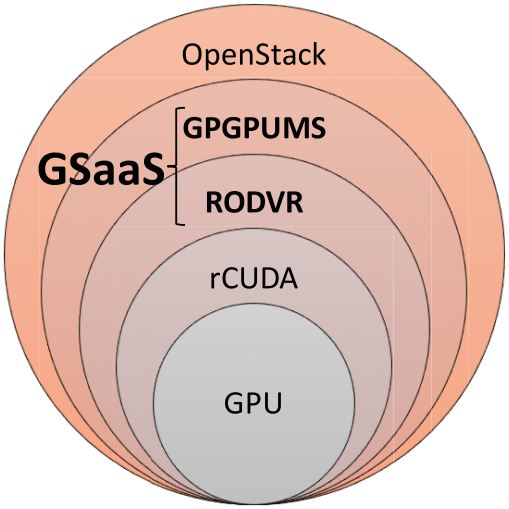


FIGURE 3. Technologies integration scheme.

Compute nodes are standard OpenStack computation nodes that allocate tenant virtual machines. It is worth noting that these nodes do not include GPUs devices, while *compute-GPU* nodes have physical GPUs and can allocate VMs that use them locally as cGPUs. On the other hand, *compute-rGPU* nodes have an additional *GPU network* that allows their hosted VMs to access remote cGPUs provided by the *compute-GPU* nodes. This dedicated GPU network avoids interference with the VM or management traffic in OpenStack. An *RODVR Endpoint* is deployed in each node (*compute-GPU* or *compute-rGPU*) willing to host cGPU-enabled VMs, whilst rCUDA-server daemons are started only in *compute-GPU* nodes. The GSaaS service orchestrates the deployment of cGPU-enabled VMs by using the GPGPUMS module and interacting with the cloud infrastructure and with the *RODVR Endpoints* through the OpenStack management network.

While RODVR is responsible for the abstraction of the real GPU location inside the VM configuration, GPGPUMS selects and assigns the most appropriate cGPUs according to new policies. This setup can also facilitate the migration of the VM to a different compute node, with (*compute-GPU*) or without (*compute-rGPU*) physical GPUs, transparently for the VM.

A. BOOTING A cGPU-ENABLED VM

The boot process of a cGPU-enabled VM starts with an user request sent to GSaaS and requires a set of activities, as depicted in Figure 5. The request provides information related to the VM instance (image, flavor, network interfaces, etc.) and related to the GPUs (number, memory, access mode, etc). Initially, GPU requirements are checked by GPGPUMS and if these are satisfied then a call to OpenStack `Nova` service is performed to check if the VM requirements can be fulfilled.

If the request is valid, the VM is created by providing a script to `cloud-init`, which is a multi-distribution package that handles early initialization of the cloud instance. This script is executed during the VM boot stage and sets rCUDA environment variables with the number of available cGPUs and their masked locations. These variables are needed to run the CUDA application in the VM.

Once the `Nova` agent starts the specified image (active state[1]), the GSaaS service waits until the network is up in the machine (deployment stage). This ensures that the `Neutron` agent in the compute node has created all the networking infrastructure for the VM.

Then, concurrently to the VM boot process, GSaaS invokes the GPGPUMS module to select the GPUs, and immediately thereafter, each GPU remote access is configured via the corresponding RODVR endpoint. The GPU information provided by GPGPUMS is passed to RODVR that inserts rules in the compute node with the aim of routing packets from the VM to the assigned GPUs. Due to these rules, packets that leave a virtual interface attached to the machine with a masked IP and port are rerouted to the real address of the host where the rCUDA daemon is running. When both stages (selection and configuration) have finished the CUDA application in the VM is ready to be used.

Equation 1 defines the total boot time of a VM when assigning x cGPUs. The parameters of the equation correspond to the activities in Figure 5, where the *VM boot time* is the sum of both *checking times (GPU and VM)*, the *VM activation & creation times* and the higher time between the concurrent activities (*VM deployment* and *GPUs selection & RODVR configuration*).

$$\begin{aligned} &cGPU-enabledVM\ Boot\ Time(x) \\ &\quad = CheckGPU\&VMTime \\ &\qquad + VMCreationTime + VMActivationTime \\ &\qquad + max(VMDeploymentTime,\ \ GPUsSelectionTime(x) \\ &\qquad + RODVRconfigurationTime(x)) \end{aligned} \tag{1}$$

[1] https://wiki.openstack.org/wiki/VMState

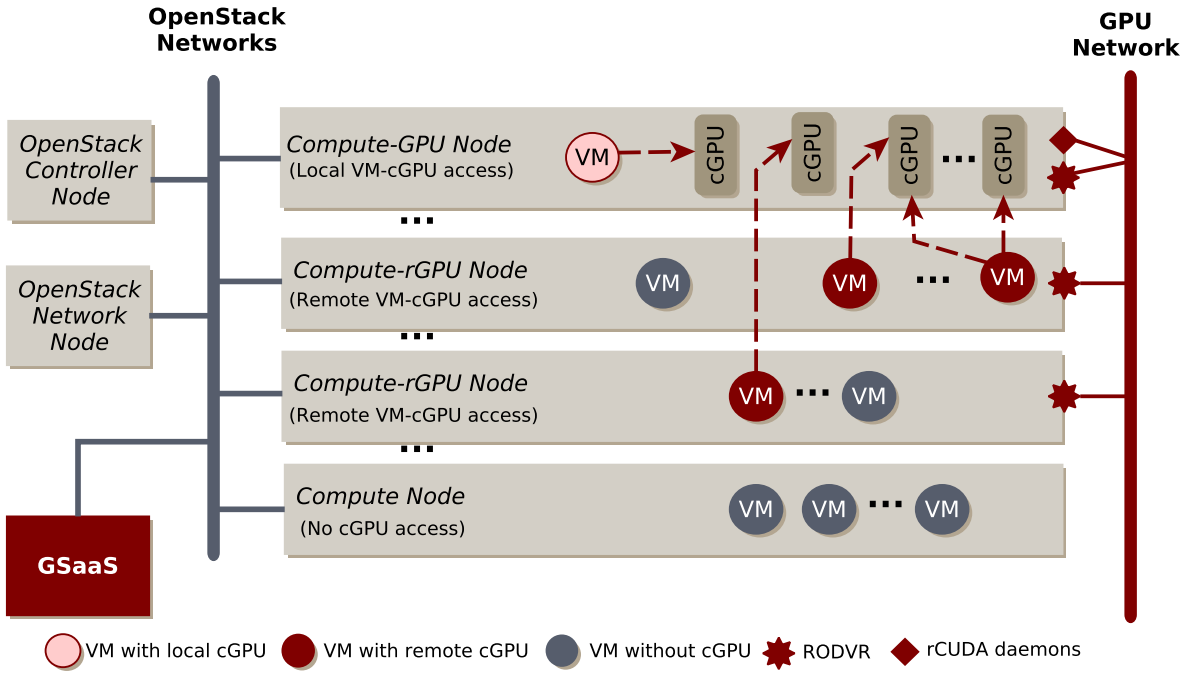


FIGURE 4. GSaaS components integration diagram, showing proposed node types and interconnection networks.

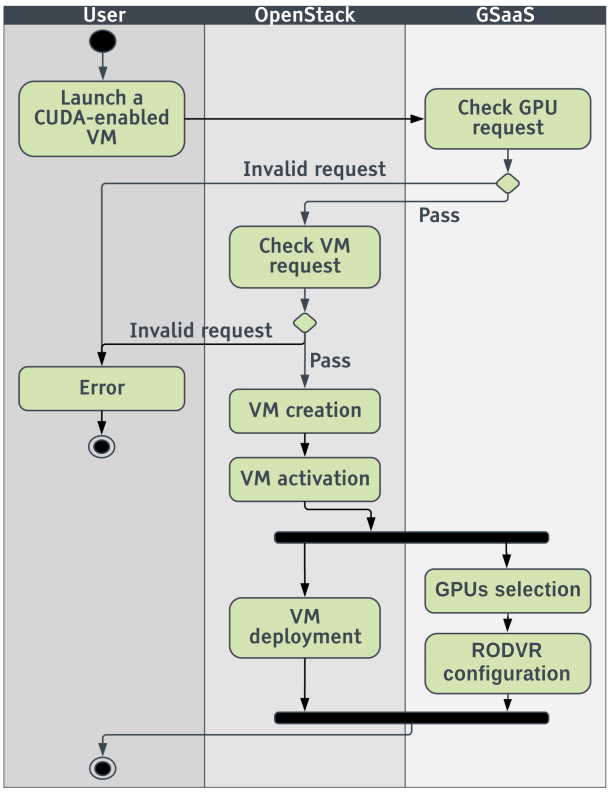


FIGURE 5. Activity diagram during the launch of a cGPU-enabled VM.

B. GPU SELECTION POLICIES

GPGPUMS includes a GPU selection policy to host cGPUs based on a *first fit* algorithm, which selects the first GPU in the pool of GPUs (result of a previous filter that takes into account the user's locality preferences) with enough available memory to fulfill the request. In cases of low need for GPU memory, this policy has as a result a performance cut since GPUs are massively oversubscribed by the cGPUs. For this reason, we moved to other policies more aimed to performance instead of energy-saving or resource-conservative as it is *first fit*. Thanks to the modularity and the plugin-based architecture of GPGPUMS, we developed a new *least load fit* policy that checks the available memory in the whole pool of GPUs and selects the device with the maximum amount of free memory. Algorithm 1 describes how decisions are performed in our *least load fit* implementation. First of all,

Algorithm 1 GPU Selection Policy Based on a Least Load Fit Strategy

```
1:  function Select_GPU(job)
2:      gpu_sorted_list = gpus.sort_desc_by(avail_mem)
3:      for each gpu in gpu_sorted_list do
4:          new_mem = gpu.avail_mem + job.req_mem
5:          if new_mem ≤ gpu.total_mem then
6:              gpu.update_status(job)
7:              return gpu
8:          end if
9:      end for
10:     return null
11: end function
```

the GPUs are listed and sorted by their current available memory (line 2). Then, the list of GPUs is iterated (line 3). For each one, the algorithm calculates the total sum of memory

needed in that GPU if the job was assigned to it (line 4), by retrieving the current amount of available memory in the GPU (*gpu.avail_mem*) and the memory requested by the job (*job.req_mem*). If the sum of memory fits in the GPU memory (line 5), the job is assigned to that GPU and its attributes are updated (line 6), for instance, the GPU available memory will be reduced by the job requested memory. Finally, the selected GPU is returned to the GPGPUMS runtime (line 7). Otherwise, if no GPU had enough memory to host the job (the cGPU request), the algorithm would return *NULL* and the request could not be satisfied (line 10).

C. CAPABILITIES AND USAGE EXAMPLES

Users are expected to use the command `gsaas` with different arguments in order to deploy their cGPU-enabled VMs. Table 1 illustrates the working modes that are provided.

TABLE 1. cGPU-enabled VMs working modes.

MODE	DESCRIPTION
Remote-Exclusive	A VM uses GPUs located in different nodes and monopolizes the use of the assigned GPUs.
Remote-Shared	A VM uses GPUs located in different nodes, but these GPUs can be shared with other VMs.
Local-Shared	A VM uses GPUs provided by the compute node in which it is allocated, but these GPUs can be shared with other VMs.
Local-Exclusive	A VM uses GPUs provided by the compute node in which it is allocated, and it monopolizes the use of the assigned GPUs.

The following list of examples illustrates the usage of the command and the working modes (information related to the VM is omitted for simplicity):

- `$ gsaas -l -ncgpus = 2`: The basic invocation to launch a cGPU-enabled VM. The parameter `-l` stands for *launch* and `-ncgpus` determines the number of cGPUs associated to the VM. By default, cGPUs are only accessible by one VM (exclusive mode).
- `$ gsaas -l -ncgpus = 2 -poolmem = 2048 -mode = shared`: In this case, a user requests 2 cGPUs in shared mode (`-mode`) with a GPU-memory limitation of 2 GB (`-poolmem`). In other words, the scheduler (GPGPUMS) is in charge of managing the GPU memory in order to meet the request, by assigning 2 cGPUs of 2 GB to the VM.
- `$ gsaas -l -ncgpus = 8 -poolmem = 4096 -mode = shared -locality = local`: A user can also define the location of the VM with respect to the cGPUs. By default, VMs are deployed in compute hosts determined by the `Nova` service of OpenStack. However, if we have the special interest in deploying a VM in the same compute node where the physical GPUs are hosted, we will use the argument `-locality` in order to indicate it. In this example, a user is launching a VM with access to 8 cGPUs of 4 GB of GPU memory each one, in the same compute node where the accelerators are plugged.
- `$ gsaas -l -ncgpus = 2 -locality = Xlocal`: In exclusive mode, locality has the same meaning. In this case, the VM is deployed in the host where the 2 GPUs are placed, having exclusive access to them.
- `$ gsaas -t -id = <VMid>`: With the argument `-t`, which stands for *terminate*, together with the identifier of a VM (`-id`), the user is able to stop and destroy an existent cGPU-enabled VM.

When requirements of launching arguments cannot be met, the deployment of the VM is aborted with an error (see *Check GPU request* activity in Figure 5).

IV. EXPERIMENTAL RESULTS

This section presents a set of experiments that evaluates and demonstrates the benefits provided by GSaaS. The experiments range from a detailed analysis of the VM deployment time to the scalability study of different kind of applications like multi-GPU or distributed computation.

A. EXPERIMENTAL SETUP

The experimental setup used to evaluate the features and the performance of our proposal is depicted in Figure 6. It is based on *OpenStack Ocata* with *Neutron* and the *ml2* plugin to define cloud networking,[2] and adopts a hybrid cloud infrastructure approach [24]. Virtual and physical machines run an Ubuntu 16.04 operating system.

The master node (*master*) and the three *compute-rGPU* nodes (*compute[0–2]-rGPU*) have 2 Intel XEON E5-2630-v3 sockets (8 cores at 2.40 GHz each) with 32 GB of DDR3-2200 SDRAM and a 2 TB hard drive. Each processor provides 16 virtual CPUs (vCPUs) due to the hyper-threading technology, and therefore 32 vCPUs are available per node.

The *master* node hosts the virtualized OpenStack controller and network nodes. Each virtualized node is configured with 8 vCPUs, but the controller has 10 GB of RAM memory and the network node has 6 GB of RAM. Moreover, the GSaaS elements have been deployed in a VM with 2 vCPUs and 2 GB of RAM as part of the hybrid infrastructure.

The *Compute-GPU* node has 2 Intel XEON E5-2603-v4 sockets (6 cores at 1.70 GHz) for a total of 12 cores with 32 GB of DDR-2133 SDRAM and a 1 TB hard drive. Furthermore, it is equipped with 4 NVIDIA Quadro M4000 GPUs with 8 GB of memory each. The *compute-GPU* node can allocate tenant VMs which make use of its local GPUs.

The 3 networks (*Management*, *VM*, and *GPU*) use 10 GbE network cards respectively connected through 3 NetGear proSAFE Plus (XS708E) switches.

B. PERFORMANCE METRICS

Metrics provided by the experimental setup can be classified into 3 main groups according to their origin: metrics from tenant VMs, metrics from their hosts, and metrics from

[2]https://docs.openstack.org/ocata/networking-guide/

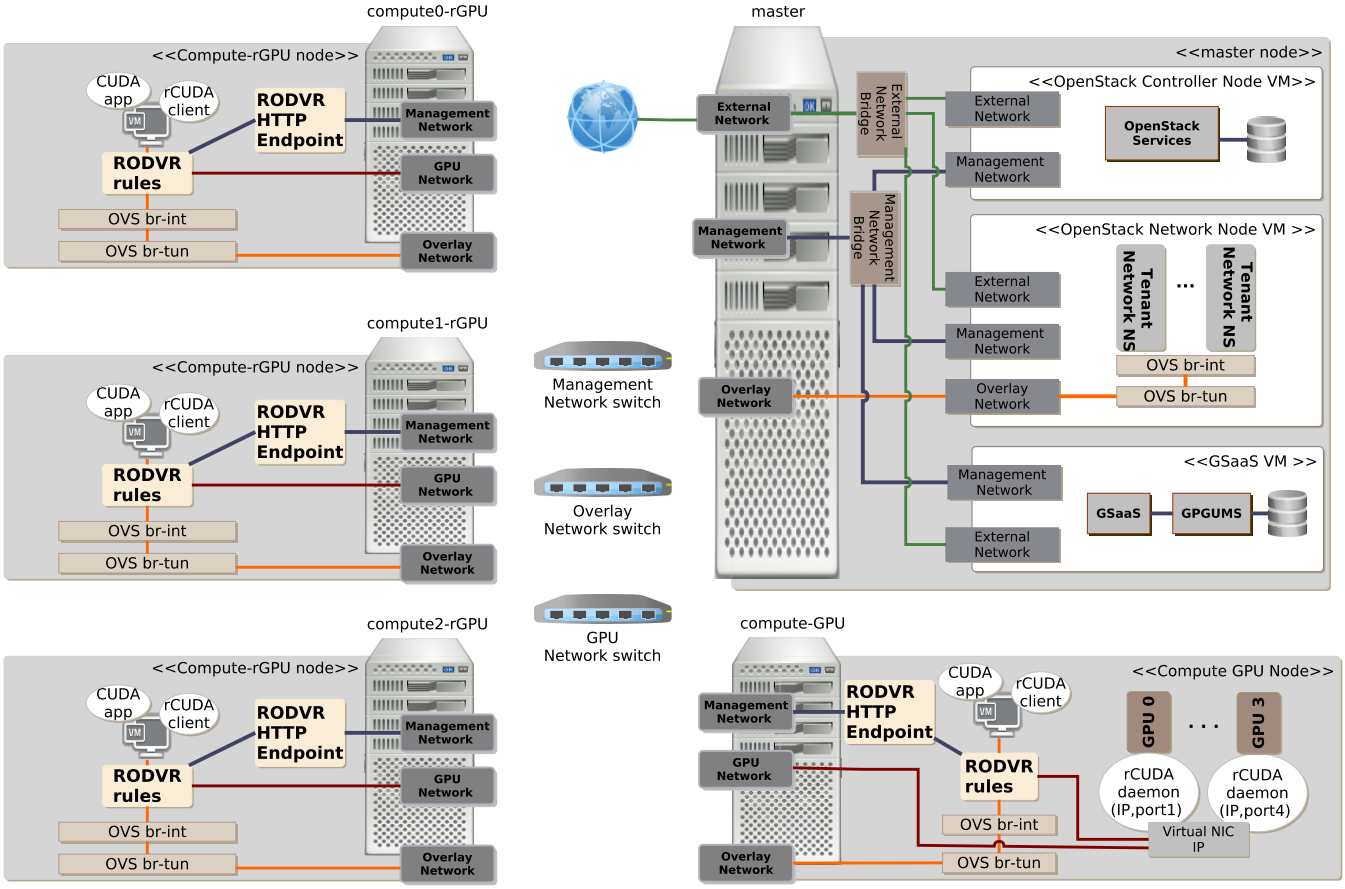


FIGURE 6. Experimental setup deployed in a hybrid cloud infrastructure based on OpenStack.

the GPUs (using `nvidia-smi` tool). With the purpose of detecting overhead conditions, memory and processor utilization is measured for each physical and virtual machine. Moreover, the detailed deployment time when evaluating the booting process, or the execution time for each performance test, are collected to characterize the user utilization pattern. Furthermore, the performance of each GPU is metered by the memory and CPU utilization of the running processes, the power consumption, and the usage rate.

All the experiments were executed 50 times in order to obtain trustworthy statistical measurements.

C. INFRASTRUCTURE DEPLOYMENT EVALUATION

This section analyzes the scalability of deploying CUDA-enabled VMs from two different points of view: a VM with multiple cGPUs, and multiple single-cGPU VMs.

First, *cGPUs scalability* has been evaluated by assessing the detailed temporal cost of booting a VM depending on the number of cloudified GPUs assigned to it. Specifically, we have performed a series of experiments that measure each time considered in Equation 1 when booting a VM with 0 to 75 cGPUs in 25-cGPU steps. The VM is booted using a flavor of 1 vCPU, 768 MB RAM and each cGPU requiring 100 MB of memory. Results are shown in Table 2 and Figure 7.

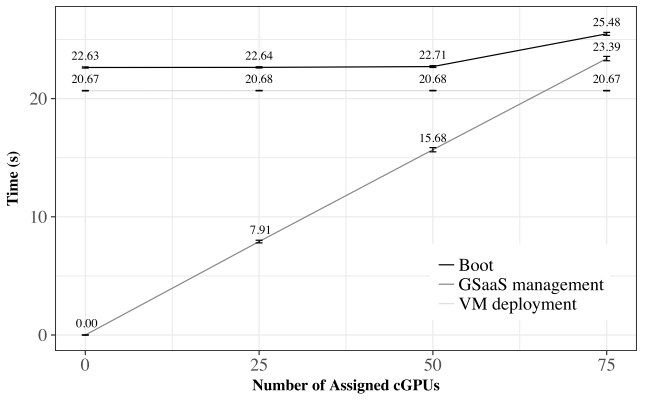


FIGURE 7. Boot time of a virtual machine when increasing the number of assigned cGPUs.

The boot time of a cGPU-enabled VM is the elapsed time between the invocation of a `gsaas` command and the moment when the requested cGPUs are ready to be used. GSaaS checking times are not affected by the number of cGPUs as can be seen in Table 2. VM creation and VM activation times show a small increase of time with the number of cGPUs. The boot time is basically determined by the

TABLE 2. Detailed time (seconds) of GSaaS booting a VM with a different number of assigned cGPUs.

# cGPUs	GSaaS checking	VM creation	VM activation	VM deployment	GSaaS management (Selection & RODVR)	Boot Time
0	-	0.6817	1.2808	20.6710	-	22.6335
25	0.0047	0.6740	1.2626	20.6755	7.9014	22.6368
50	0.0047	0.7397	1.2588	20.6769	15.6715	22.7101
75	0.0046	0.7515	1.3121	20.6745	23.3833	25.4815

maximum between the time required by OpenStack to deploy the VM and the time required by GSaaS to select GPUs and configure the access to each cGPU in the compute node where the VM is allocated (named GSaaS management time). As can be seen in Figure 7, the GSaaS management time increases linearly with the number of cGPUs to configure. However, the VM deployment time is bigger than the GSaaS management time up to around 62 cGPUs. The overhead when assigning 50 cGPUs is almost negligible, since it represents only an increase of 0.2% in the boot time with respect to the base case of booting the VM without cGPU. In the case when 75 cGPUs are assigned, the GSaaS management time is bigger than the VM deployment time and the boot time has an overhead of 12%.

Conversely, *VMs scalability* has been evaluated by studying the deployment of multiple single-cGPU VMs when varying from 15 to 90 in 15-VM steps. Experiments use the three *compute-rGPU* nodes (*compute[0–2]-rGPU*) to fairly allocate the VMs, and they wait for 10 seconds between consecutive invocations to the `gsaas` command. Each VM is booted using the same flavour as in the previous experiment. Results are shown in Figure 8. The Y-axis represents a zoomed-in region of the time, which indicates an almost negligible increment in the time, mainly due to the performance deterioration of the compute nodes that host all the VMs.

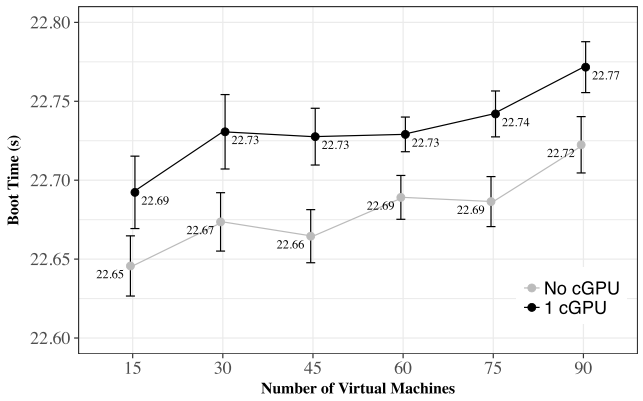


FIGURE 8. Boot time of non-GPU-enabled VMs compared to cGPU-enabled VMs.

D. PERFORMANCE EVALUATION

Three different applications have been chosen in order to study different aspects of our proposal. First, the importance of the VM-GPU tenant locality and the GPU shareability is analyzed. Then, a multi-GPU scenario, where all the cGPUs of a VM work together to solve a problem, is examined. Finally, a distributed CUDA-enabled application uses the message passing interface (MPI) paradigm to parallelize a problem in several processes.

1) GPU LOCALITY AND SHAREABILITY

In this experiment we leverage CUSHAW [28], a well-established leading next-generation sequencing read alignment CUDA compatible software package. This problem allows us to check the behavior of GSaaS when there is a significant amount of data to be transferred from host to device and vice versa. The VM sends to the GPU the human reference genome (around 2.4GB) and the sequenced chromosome (in our case 240MB for the sample human chromosome Chr1) to be analyzed. When the alignment finishes it receives the result (around 200 MB).

With this application we analyze the effect of the cGPU locality and the impact of sharing accelerators. For this purpose, 2 instances of CUSHAW are concurrently executed in a VM equipped with 4 vCPUs, 8 GB of memory and 2 cGPUs. While the *exclusive* cGPUs match 1-to-1 the GPUs, in the *shared* mode, each cGPU allocates 4 GB of memory of the same physical GPU. Furthermore, we distinguish between two deployment locality options: *local* if the VM is deployed in the same host where the GPUs are installed (*compute-GPU*) or *remote* if they are in a non-GPU compute node (*compute-rGPU*).

Figure 9 depicts the total execution time of both CUSHAW instances running concurrently. Results of the exclusive scenario show an increment of 5% in execution time when the experiment is performed using remote access instead of

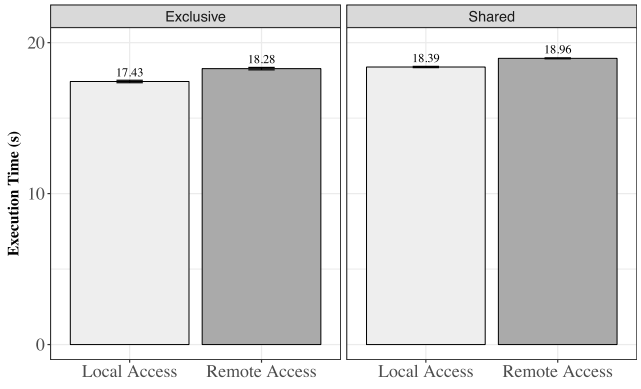


FIGURE 9. Mean execution time of *CUSHAW* for exclusive and shared modes using local and remotes deployment localities.

local access, because the VM has to reach other compute host to transfer the data to and from the accelerator. The shared scenario presents a similar general behavior.

Worth noting is that the execution time in the shared scenario does not double the time obtained in the exclusive scenario, because there are periods with different GPU usage. Therefore, it is possible to execute two CUSHAW works simultaneously during the low usage periods without a real drawback when considering remote locality or GPU shareability.

All in all, since the exclusive mode allocates for a cGPU the whole memory of a physical GPU, memory-based policies such as *least load fit* or *first fit* have no difference in the resource scheduling and the resource waste is inevitable. However, if we combine the locality with exclusive mode, we can appreciate variations in the performance of the applications.

2) MULTI-GPU COMPUTATION AND SCHEDULING POLICIES

Multi-GPU applications leverage all the assigned cGPUs to the VM to perform their operations. A classical Multi-GPU application based in the Monte Carlo algorithm, widely used in this kind of tests, is found in the NVIDIA SDK. *MonteCarloMultiGPU*[3] is a single-process application which evaluates fair call price for a given set of European options using a Monte-Carlo approach, taking advantage of all CUDA-capable cGPUs assigned to the VM.

In this regard, we have deployed a VM with 30 vCPUs and 30 GB of RAM. Furthermore, through GSaaS, we have progressively attached to the VM more cGPUs (with 4 GB of memory) in order to evaluate the productivity of *MonteCarloMultiGPU* when sharing GPUs. The VM can own up to 8 cGPUs (each GPU has 8 GB of memory) with this configuration.

Figure 10 depicts the number of options calculated per second (throughput) by *MonteCarloMultiGPU* when assigning an increasing number of cGPUs to the VM and considering different working modes (local and remote) and selection policies (first fit and least load fit).

As in the previous section, local-remote modes do not present any significant improvement. However, when comparing selection policies we can appreciate important differences in the results.

From the perspective of an end-user, *least load fit* policy provides better results up to 4 cGPUs. In this particular case, this policy selects 4 different GPUs to host the first 4 cGPUs, while *first fit* hosts the first 2 cGPUs in the same GPU, the third and forth cGPUs in the second GPU and so on, oversubscribing the accelerators earlier. *Least load fit* prioritizes devices with lower allocation of memory; that is why *least load fit* selects different accelerators (exclusive usage) for the cGPUs until they are finished (up to 4), then from the 5th cGPU on, GPUs are oversubscribed.

[3]http://developer.download.nvidia.com/compute/cuda/1.1-Beta/x86_website/samples.html#MonteCarloMultiGPU

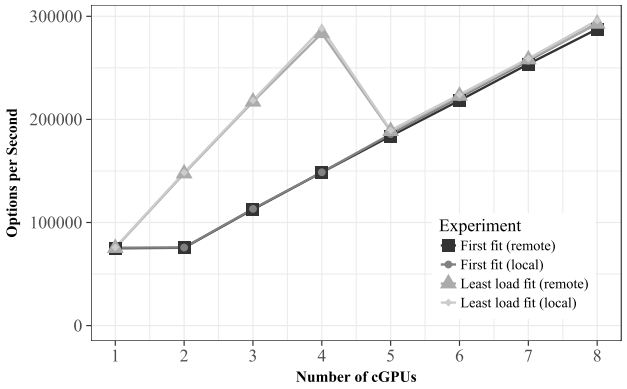


FIGURE 10. MonteCarloMultiGPU throughput using different cGPUs assignation policies.

On the other hand, the *first fit* policy depicts a more appealing approach for a cloud provider. This policy deploys cGPUs in one accelerator while it has enough memory to host them. With the requirement of 4 GB of memory per cGPUs, each physical device can host up to 2 cGPUs, hence we are using few GPUs at the expense of worse performance (fewer options calculated per second). However, depending on the pay-per-use price of the cGPUs, a user can sacrifice the performance if the economic outcome is worth it.

3) DISTRIBUTED GPU-ENABLED COMPUTATION

For this last experiment we have employed a distributed application with support for GPUs. The application, CUDA-MEME [29], is an ultra-fast scalable motif discovery algorithm. Particularly, we have used *mCUDA-MEME*, a further extension of CUDA-MEME in terms of sensitivity and speed, which enables users to use a GPU per MPI process in order to accelerate motif finding. The application was configured to work over the input dataset `nrsf_2000.fasta`, from its test-cases, with a maximum size of 2,000,000 elements.

mCUDA-MEME is capable of using all the GPUs in a host whereas there is spawned, at least, the same amount of MPI processes. This feature limits the grade of parallelism to the number of installed GPUs in the target host. Apart from that, GPUs involved in the resolution of *mCUDA-MEME* are not fully computationally loaded. This experiment provides an insight into the use of GSaaS, which demonstrates its versatility when it comes to improve the performance of an application and the system utilization.

To increase the parallelism with the same hardware infrastructure, we share the GPUs in order to provide access to more cGPUs in 2 different ways. On the one hand, we setup a VM with 12 cGPUs (intranode distributed computation). On the other hand, we deploy up to 12 single-cGPU VMs (internode distributed computation). Both scenarios aim to increase the performance of *mCUDA-MEME* instances by spawning more MPI processes with access to a GPU. The VM for the intranode experiment consisted of 30 vCPUs and 30 GB of RAM, while the internode VMs were equipped with

2 vCPUs and 4 GB of RAM. Each cGPU was configured with 1 GB of memory, and GPGPUMS with the *least load fit* policy.

Figure 11 shows the average time of 50 executions of *mCUDA-MEME* with an increasing number of CUDA-enabled processes running on the same VM (intranode) and on independent VMs (internode). For the sake of clarity, we have omitted the result of 1 cGPU (that is, 3,997.56 seconds) and beyond 12 processes, which did not show further performance improvement. Increasing the number of CUDA-enabled MPI processes, by sharing the 4 underlying physical GPUs, reduces the execution time in both modes. However, offloading the processes in different VMs (internode scenario) provides better results. The intranode scenario processes run on the same compute host (where the VM is running), whilst the internode scenario processes run on different VMs, spread across the 3 compute hosts. Hence, compute hosts have less burden and the performance increases.

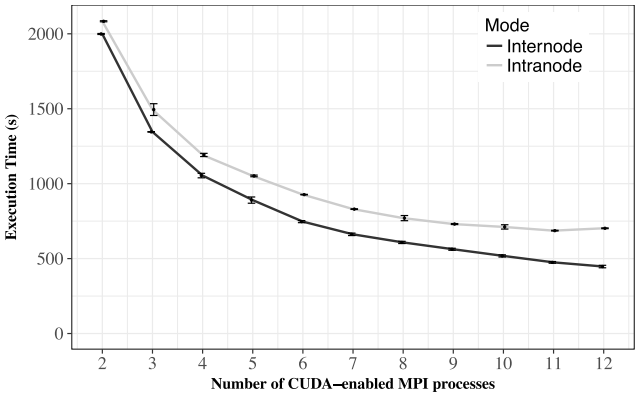


FIGURE 11. mCUDAmeme execution time with different number of CUDA-enabled MPI processes running on the same VM (intranode) and on different VMs (internode).

Worth noting is that the low utilization rate of the accelerators gives us an improvement opportunity. Figure 12 illustrates the average utilization rate of each physical GPU (obtained from the `nvidia-smi` command with a sampling rate of 1 second) during the execution of *mCUDA-MEME* for the configuration of different processes, when considering intranode (Figure 12a) and internode (Figure 12b) modes. Both charts depict a pattern where utilization rate increases a step every 4 processes. Once the 4-process point is exceeded, where each process is using exclusively one of the 4 GPUs available, GPUs are oversubscribed by the subsequent processes. As it is shown in Figure 12b, processes running on different VMs can exploit better the GPUs, utilizing them almost a 20% more than the intranode counterpart.

V. PRODUCTION ENVIRONMENT PROJECTIONS

After experimentally showing the benefits of GSaaS on a real infrastructure, we now supplement the study with the analysis

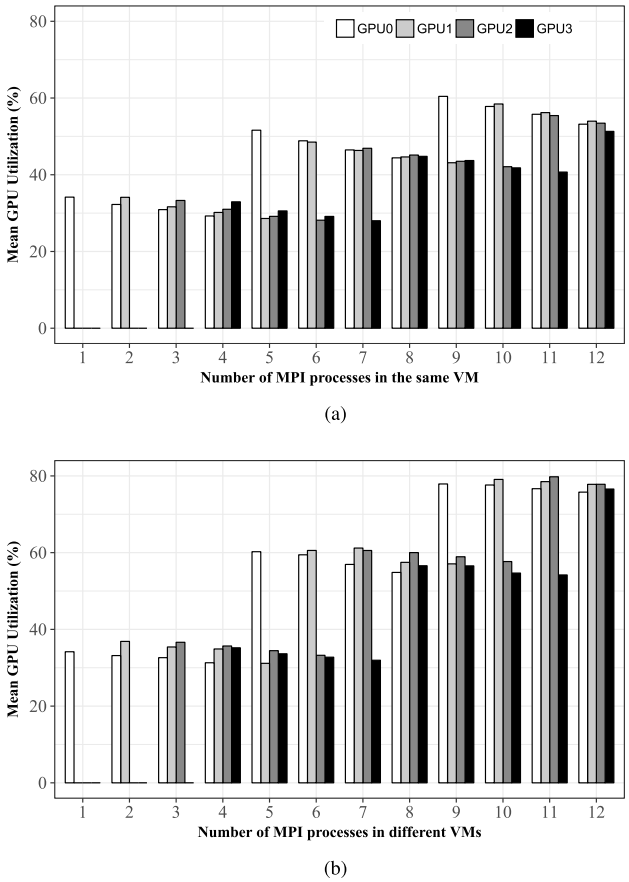


FIGURE 12. GPU utilization rate for both distributed modes. (a) Intranode mode. (b) Internode mode.

of using GSaaS in production environments where physical resources are managed by a job scheduler.

In production clusters, users submit their resource requests in the shape of jobs. An increase in the number of completed jobs per unit of time has been proved in many studies by combining a remote access GPU technology with an RMS. For instance, Iserte *et al.* [30] show how a HPC cluster can boost its throughput by combining rCUDA and Slurm.

If the submitted jobs instanced VMs requiring a certain amount of GPUs, the scheduler should be able to manage those jobs and the underlying resources. In order to analyze the behavior of our framework in a production environment, we have developed a simulator based on *SimPy*.[4]

The simulator implements a job scheduler combining *SimPy* features such as: resource management and waiting for other processes. Moreover, the simulator has been developed to take decisions depending on the available GPUs and on the request in order to determine whether a job is executed or not. For this purpose, the simulator is provided with a naive job scheduling policy, sufficiently satisfactory to demonstrate the benefits of GSaaS in production environments. This policy implements a *first come first served (FCFS)* algorithm

[4]https://simpy.readthedocs.io

without backfilling, in other words, not allowing jobs to overtake others.

In order to evaluate the GSaaS approach, we simulated the execution of a workload of jobs, where each job requested a quantity of single-GPU nodes for a given time. The workloads were based on the Lublin-Feitelson model [31], a complex statistical model that derives from actual traces and considers, among other features: the resources requested by the jobs, the job execution times and the jobs interarrival times. In an effort to emulate our 4-GPU infrastructure, jobs were restricted to use a maximum of 4 nodes (with 1 GPU each one). Additionally, the GPU memory partitions were limited to 1 GB with the aim of deploying up to 32 cGPU-enabled VMs ($8\ GB \times 4\ GPUs$), instead of the 4 physical GPUs without the cloudified approach. Furthermore, the experiment assumes that the maximum quantity of GPU memory needed by a job is 1 GB.

Table 3 contains the estimated processing time of the workloads with different cGPU configurations. Results show a reduction in time of a 50% with the 2X increment of accelerators.

TABLE 3. Expected execution time of a workload with different cGPU configurations.

# cGPUs	Assigned memory	Expected execution time
4	8GB	53.35 hours
8	4GB	26.73 hours
16	2GB	12.71 hours
32	1GB	6.89 hours

Nevertheless, when a physical GPU is shared among different processes, the performance of these processes is degraded. A previous work [21] analyzed this degradation when several programs are running at the same time in a GPU. This study involves a set of commonly used scientific applications, where in the worst case, two instances of the same application running concurrently, experienced an increment in the execution time of 1.75X. This set of applications includes mCUDA-MEME, where two instances need a 10% more time to finish, while 3 concurrent instances complete their execution using 1.4X the original individual time.

As scientific applications in production environments do not usually spend all the execution time running on the GPU (they have different CPU-GPU computational stages and also spend time transferring data between them), executing concurrent instances do not imply to double or triple the completion time. Although, sharing resources has proved to benefit the global throughput of the system, the performance reduction cannot be dismissed. For this reason, we have included a *sharing factor overhead* that increments the running time of the jobs by a defined factor.

Figure 13 shows the speedup obtained after applying: a null sharing overhead factor (1X), a higher overhead of 1.5X and a maximum theoretical sharing overhead of 2X, which doubles the job execution time. It represents the estimated speedups of sharing GPUs among VMs regarding different overhead factors. The simulation reaches up to 4 concurrent jobs running over the physical accelerator (16 cGPUs), each one requesting 2 GB of GPU memory. Results indicate an speedup in the range of 2-4X.

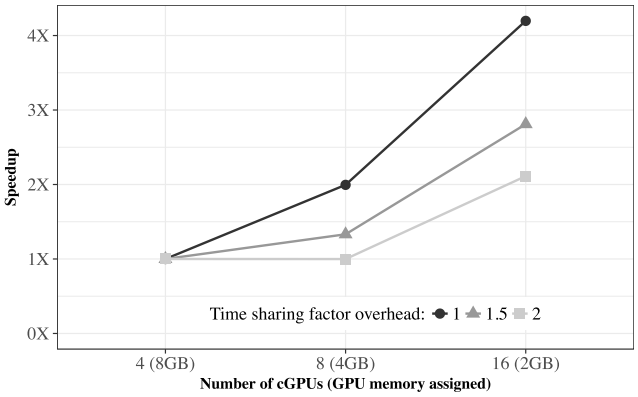


FIGURE 13. Simulated speedup when applying different time sharing factors overhead.

VI. CONCLUSIONS AND FUTURE WORK

Although many application areas would benefit from suitable cloud solutions based on GPUs, cloud providers are only offering instances with exclusive access to local GPUs by using PCI passthrough. This limitation can be overcome by providing cGPUs from existing devices that do not support native virtualization, as we extensively demonstrate in this paper.

This work designs, develops and evaluates GSaaS, a service to cloudify and schedule the access to physical GPUs from VMs, aimed to public cloud infrastructures. The proposed service relies on a set of 3 distributed components, integrated between the cloud infrastructure and the physical GPUs. In this sense, rCUDA enables remote access to the GPUs, which are previously cloudified and scheduled by RODVR and GPGPUMS, respectively. These components offer 4 cGPU working modes: local-exclusive, local-shared, remote-exclusive, and remote-shared.

The main benefits achieved by GSaaS are: adaptive scheduling of GPU resources, decoupling of the interface between the client and the rCUDA server, hiding the real location of the resources, preventing GPU unauthorized accesses, and detaching VM traffic from GPU traffic by using a dedicated network. Besides, the proposed solution automates the configuration of its distributed components.

A GSaaS prototype has been evaluated in an actual cloud deployment based on OpenStack. We have demonstrated its versatility in different scenarios where GSaaS can be leveraged to scale-up applications, facilitate the provision of accelerators or increase the utilization rate of the GPU. Deployment scalability experiments denote that our solution introduces low overhead, since the deployment time is only increased 0.2% when assigning 50 cGPUs to a VM. Performance experiments reveal the importance of VM-GPU tenant

locality and GPU shareability in different scenarios. Results show that the application performance is barely affected, and the proposed service can exploit better the GPUs, increasing their utilization up to a 20% in a distributed scenario. Finally, through a simulation, we unveil the potential of resource management tools like GSaaS when dealing with many requests.

We are exploring the usage and adaptation of the components provided by the networking plane of the cloud infrastructure to provide access to scheduled cGPUs as future work.

ACKNOWLEDGMENT

The authors would like to thank the anonymous reviewers for their valuable, insightful comments that improve the quality of this paper.

REFERENCES

[1] T. Mastelic, A. Oleksiak, H. Claussen, I. Brandic, J.-M. Pierson, and A. V. Vasilakos, "Cloud computing: Survey on energy efficiency," *ACM Comput. Surv.*, vol. 47, no. 2, 2014, Art. no. 33.

[2] C.-H. Hong, I. Spence, and D. S. Nikolopoulos, "GPU virtualization and scheduling methods: A comprehensive survey," *ACM Comput. Surv.*, vol. 50, no. 3, 2017, Art. no. 35.

[3] J. P. Walters *et al.*, "GPU passthrough performance: A comparison of KVM, Xen, VMWare ESXi, and LXC for CUDA and OpenCL applications," in *Proc. IEEE 7th Int. Conf. Cloud Comput. (CLOUD)*, Jun./Jul. 2014, pp. 636–643.

[4] *NVIDIA NVLink Fabric*. Accessed: Jun. 1, 2018. [Online]. Available: https://www.nvidia.com/en-us/data-center/nvlink/

[5] *AWS: Amazon Web Services*. Accessed: Feb. 6, 2018. [Online]. Available: http://aws.amazon.com

[6] *Google Cloud Platform*. Accessed: Feb. 6, 2018. [Online]. Available: https://cloud.google.com/

[7] *Microsoft Azure*. Accessed: Jun. 1, 2018. [Online]. Available: https://azure.microsoft.com/

[8] *GPUs in Google Compute Engine*. Accessed: Feb. 6, 2018. [Online]. Available: https://cloud.google.com/compute/docs/gpus/

[9] *Amazon Elastic Compute Cloud EC2*. Accessed: Feb. 6, 2018. [Online]. Available: http://aws.amazon.com/ec2

[10] *Microsoft Azure: GPU Optimized Virtual Machine Sizes*. Accessed: Jun. 1, 2018. [Online]. Available: https://docs.microsoft.com/en-us/azure/virtual-machines/windows/sizes-gpu/

[11] *NVIDIA GPU Cloud: GPU-Accelerated Containers*. Accessed: Jun. 1, 2018. [Online]. Available: https://www.nvidia.com/en-us/gpu-cloud/

[12] *NVIDIA GRID Technology*. Accessed: Feb. 20, 2018. [Online]. Available: www.nvidia.com/object/grid-technology.html

[13] *Intel Graphics Virtualization Technology (Intel GVT)*. Accessed: Feb. 20, 2018. [Online]. Available: https://01.org/igvt-g/blogs/wangbo85/2017/intel-gvt-g-kvmgt-public-release-q22017

[14] *Amazon EC2 Elastic GPUs*. Accessed: Jun. 1, 2018. [Online]. Available: https://aws.amazon.com/ec2/elastic-gpus/

[15] R. Montella *et al.*, "On the virtualization of CUDA based GPU remoting on ARM and X86 machines in the GVirtuS framework," *Int. J. Parallel Program.*, vol. 45, no. 5, pp. 1142–1163, Oct. 2016.

[16] G. Giunta, R. Montella, G. Agrillo, and G. Coviello, "A GPGPU transparent virtualization component for high performance computing clouds," in *Proc. Euro-Par Conf. Parallel Process.* Springer, 2010, pp. 379–391.

[17] K. M. Diab, M. M. Rafique, and M. Hefeeda, "Dynamic sharing of GPUs in cloud systems," in *Proc. IEEE 27th Int. Parallel Distrib. Process. Symp. Workshops PhD Forum (IPDPSW)*, May 2013, pp. 947–954.

[18] T. J. Jun, V. Q. Dung, M. Yoo, D. Kim, H. Cho, and J. Hahm, "GPGPU enabled HPC cloud platform based on openstack," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal.*, 2014.

[19] Z. Qi, J. Yao, C. Zhang, M. Yu, Z. Yang, and H. Guan, "VGRIS: Virtualized GPU resource isolation and scheduling in cloud gaming," *ACM Trans. Archit. Code Optim.*, vol. 11, no. 2, 2014, Art. no. 17.

[20] A. J. Peña, C. Reaño, F. Silla, R. Mayo, and E. S. Quintana-Ortí, and J. Duato, "A complete and efficient CUDA-sharing solution for HPC clusters," *Parallel Comput.*, vol. 40, no. 10, pp. 574–588, 2014.

[21] F. Silla, S. Iserte, C. Reaño, and J. Prades, "On the benefits of the remote GPU virtualization mechanism: The rCUDA case," *Concurrency Comput., Pract. Exper.*, vol. 29, no. 13, p. e4072, 2017.

[22] S. Iserte, F. J. Clemente-Castelló, A. Castelló, R. Mayo, and E. S. Quintana-Ortí, "Enabling GPU virtualization in cloud environments," in *Proc. 6th Int. Conf. Cloud Comput. Services Sci. (CLOSER)*, Rome, Italy, 2016, pp. 249–256.

[23] *OpenStack: The Open Source Cloud Operating System*. Accessed: Feb. 6, 2018. [Online]. Available: http://www.openstack.org/software/

[24] E. Chirivella-Perez, J. Gutiérrez-Aguado, J. M. Claver, and J. M. A. Calero, "Hybrid and extensible architecture for cloud infrastructure deployment," in *Proc. 15th IEEE Int. Conf. Comput Inf. Technol.*, Oct. 2015, pp. 611–617.

[25] I. Habib, "Virtualization with KVM," *Linux J.*, vol. 2008, no. 166, Feb. 2008, Art. no. 8.

[26] A. J. Pe na, "Virtualization of accelerators in high performance clusters," Ph.D. dissertation, Univ. Jaume I, Castellón de la Plana, Spain, 2013.

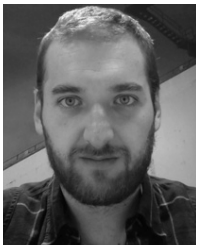
[27] S. Iserte *et al.*, "SLURM support for remote GPU virtualization: Implementation and performance study," in *Proc. IEEE 26th Int. Symp. Comput. Archit. High Perform. Comput. (SBAC-PAD)*, Oct. 2014, pp. 318–325.

[28] Y. Liu, B. Schmidt, and D. L. Maskell, "CUSHAW: A CUDA compatible short read aligner to large genomes based on the Burrows–Wheeler transform," *Bioinformatics*, vol. 28, no. 14, pp. 1830–1837, 2012.

[29] Y. Liu, B. Schmidt, W. Liu, and D. L. Maskell, "CUDA–MEME: Accelerating motif discovery in biological sequences using CUDA-enabled graphics processing units," *Patter Recognit. Lett.*, vol. 31, no. 14, pp. 2170–2177, Oct. 2010.

[30] S. Iserte, J. Prades, C. Reaño, and F. Silla, "Increasing the performance of data centers by combining remote GPU virtualization with Slurm," in *Proc. 16th IEEE/ACM Int. Symp. Cluster, Cloud Grid Comput. (CCGrid)*, May 2016, pp. 98–101.

[31] U. Lublin and D. G. Feitelson, "The workload on parallel supercomputers: Modeling the characteristics of rigid jobs," *J. Parallel Distrib. Comput.*, vol. 63, no. 11, pp. 1105–1122, 2003.



SERGIO ISERTE received the B.S. and M.S. degrees in computer engineering in 2011 and the M.S. degree in intelligent systems from Universitat Jaume I, Spain, in 2014, where he is currently pursuing the Ph.D. degree in computer science. His research interests are mainly in high-throughput computation which include parallel shared and distributed runtimes, resource management, in clusters and clouds, and energy-efficient high-performance systems.

RAÚL PEÑA-ORTIZ received the Ph.D. degree in computer science from the Universitat Politècnica de València, Spain. He has around 20 years' experience in software engineering and quality, especially in the application of scientific research knowledge to fix industrial problems related to web-based applications and their ideal deployments, publishing various papers in leading journals and conferences. He worked in several universities, research centers, and software companies, participating in more than 50 national and international research and innovation projects. He is currently an Adjunct Lecturer with the Universitat de València, Spain. His research is aimed at designing and evaluating distributed architectures based on cloud and fog computing.

JUAN GUTIÉRREZ-AGUADO received the Ph.D. degree in computer science from the Universitat de València, where he is currently an Associate Professor. He has taught undergraduate and graduate courses on image processing, programming, mobile devices, server-side programming, and cloud computing. He has authored or co-authored of journal papers in computer vision, image processing, and recently in cloud infrastructure architectures and monitoring. His current research focuses on distributed and cloud computing.

JOSE M. CLAVER (M'00–SM'14) received the M.Sc. degree in physics from the Universitat de València (UV) and the Ph.D. degree in computer science from the Universitat Politècnica de València, Spain. He was with the Electronics and Computer Architecture Department, University of Castilla-La Mancha, Spain, from 1985 to 1991, and also with the Department of Computer Science, Universitat Jaume I, Spain, from 1991 to 2007. Since 2007, he has been a Professor with the Department of Computer Science, UV. He has taught undergraduate and graduate courses on computer architecture, embedded systems, high-speed networks, and parallel computing. He has authored or co-authored over 60 research publications on these subjects. His research interests include parallel and cloud computing, computer architecture, computer networks, and embedded systems.

RAFAEL MAYO received the B.S. and Ph.D. degrees in computer science from the Universitat Politècnica de València in 1991 and 2001, respectively. Since 2002, he has been an Associate Professor with the Department of Computer Science and Engineering, Universitat Jaume I. His research interests include the optimization of numerical algorithms for general processors and for specific hardware, and their parallelization on both message-passing parallel systems (mainly clusters) and shared-memory multiprocessors. Nowadays, he is involved in several research efforts on HPC energy-aware systems, cloud computing, and HPC system and development tools.