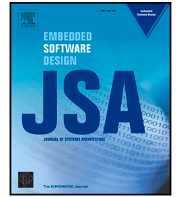
Journal of Systems Architecture 151 (2024) 103138

Contents lists available at ScienceDirect

Journal of Systems Architecture

journal homepage: www.elsevier.com/locate/sysarc



NGS: A network GPGPU system for orchestrating remote and virtual accelerators

Javier Prades [a,*], Carlos Reaño [b], Federico Silla [a]

[a] Departament d'Informàtica de Sistemes i Computadors, Universitat Politècnica de València, 46002 Valencia, Spain
[b] Departament d'Informàtica, Escola Tècnica Superior d'Enginyeria (ETSE), Universitat de València, 46100 Burjassot, Spain

A R T I C L E I N F O

Keywords:
Scheduling
GPU
CUDA
Virtualization

A B S T R A C T

In General-Purpose computing on Graphics Processing Unit (GPGPU), the use of CPUs is combined with that of GPUs. CPUs are used for sequential code, while GPUs are used for parallel code. GPGPU has been enabled by two key factors: (i) the massively parallel architecture of GPUs, which allows thousands of single cores to run parallel code; and (ii) the development of platforms, such as CUDA, that simplify implementing code for GPUs. GPGPU has established itself as the standard computing system in most computing fields due to the great improvements it brings. However, its use is not without problems, such as GPU underutilization, high cost, power consumption, etc. In this paper we present NGS (Network GPGPU System) to address the underutilization of GPUs in computing centers. NGS orchestrates the concurrent access to GPGPU resources from different nodes of the cluster by leveraging the remote GPU virtualization mechanism and the NVML library by NVIDIA. In this way, NGS enables different nodes of the cluster to access remote GPUs as if they were local at the same time that this access is guaranteed to be carried out without collisions. The main novelty is that NGS offers a global and standard solution independent of the computing environment used. Experimental results show up to 4x improvements compared to popular approaches.

1. Introduction

Using GPUs (Graphics Processing Units) to accelerate computing has become tremendously popular during the last two decades. The beginnings of GPU computing date back to the early 2000s when several studies demonstrated the computational capabilities offered by GPUs in the field of linear algebra [1,2]. This trend was consolidated in what it is known today as GPGPU (General-Purpose computing on Graphics Processing Unit) [3]. In GPGPU, the use of the CPU is combined with the use of GPU. Thus, the CPU is used in the parts of the execution where sequentiality and logical operations predominate, while the GPU is used in the most computationally intensive parts. GPGPU has been enabled by two key factors. Firstly, the massively parallel architecture of GPUs together with their high memory access bandwidth, which allows the thousands of single cores available on GPUs to perform parallel floating point operations at high speed and with very high energy efficiency. The second key factor is the development of platforms, such as CUDA (Compute Unified Device Architecture) [4,5] and OpenCL (Open Computing Language) [6,7], which have allowed developers to program and compile code to be executed on GPUs in an efficient and relatively simple way.

GPGPU is currently widely used in high-performance computing (HPC) and artificial intelligence (AI) due to the enormous computational power that these devices have achieved. According to the latest update of the Top500 (June 2023) [8], 8 of the 10 most powerful supercomputers in the world use GPU acceleration. GPUs have been systematically improving their performance year after year. In the decade from 2010 to 2020 they multiplied by 20 times their computational capacity in FP64 operations growing at an annual rate of 35% which is still maintained today [9]. In terms of power consumption, GPUs have very high power consumption. For example, the NVIDIA H100 PCIe GPU has a power consumption of up to 350 W [10]. Nevertheless, they are extraordinarily efficient devices due to their high computational power. This is evident in the updates of the Green500 list [11], where the top 10 supercomputers always incorporate the latest generation GPUs, rendering configurations with previous-generation GPUs obsolete.

As we have seen so far, GPGPU has established itself as the standard computing system in most computing fields due to the great improvements it brings. However, the use of GPGPU is not without problems. There are some challenges that must be faced in order to improve the use of this technology. One of the main challenges presented by this

* Corresponding author.
E-mail addresses: japraga@gap.upv.es (J. Prades), carlos.reano@uv.es (C. Reaño), fsilla@disca.upv.es (F. Silla).

https://doi.org/10.1016/j.sysarc.2024.103138
Received 30 November 2023; Received in revised form 23 March 2024; Accepted 2 April 2024
Available online 5 April 2024
1383-7621/© 2024 The Authors. Published by Elsevier B.V. This is an open access article under the CC BY license (http://creativecommons.org/licenses/by/4.0/).

technology is the difficulty of achieving optimal utilization of GPUs in data centers. In general, studies [12,13] show an underutilization of GPUs in data centers, one of the reasons being that compute clusters often incorporate GPUs in all or most of their nodes in order to easy the scheduling of GPU-accelerated applications. As a consequence of populating most nodes with GPUs, accelerators are often idle, as nodes also execute CPU-only applications. Furthermore, many applications do not make an intense use of the GPU during the whole application execution time. The low GPU utilization, along with the high cost of these devices, results in slow cost recovery. Related to that, there is also a great leap in computing power and energy efficiency every few years. Thus, rapid cost recovery is key to leveraging the competitive advantage that each new generation of GPUs presents over the previous one.

As can be seen, it is necessary to establish mechanisms that increase the utilization of GPUs in data centers. There are different approaches to meet this challenge. We have identified the following: (i) use of GPU virtualization mechanisms, (ii) use of improved GPU job scheduling solutions and, finally, (iii) a combination of both. Next we present these three approaches in more detail:

- GPU virtualization. There are several CUDA-compatible technologies based on GPU virtualization, such as MPS (Multi-Process Service) [14], vGPU (Virtual GPU) [15], MIG (Multi-Instance GPU) [16]. In general, these technologies allow us to create multiple virtual instances from a physical GPU. Through these virtual instances, a single physical GPU can concurrently provide GPU acceleration services to more than one application. There are also solutions based on remote GPU virtualization, such as rCUDA [17], gVirtuS [18,19] or AVEC [20,21]. These solutions go a step further and allow us to use virtual GPU instances over the network, regardless of the physical location where the actual GPU resides, thus noticeably increasing GPU utilization.
- GPU-job scheduling. Due to the rise in the use of GPU compute acceleration in today's data centers, most general-purpose schedulers like Kubernetes [22], OpenPBS [23] or Slurm [24] support the use of GPUs in order to improve the overall utilization of GPUs in computing clusters. In addition there is also, especially within the academic research field, a multitude of GPU scheduling solutions tailored to specific tasks. Some examples can be found in [25–27]. However, current GPU-job schedulers do not consider the use of remote GPU virtualization despite the great benefits provided by this mechanism.
- GPU virtualization combined with GPU scheduling. On the one hand, we have the use of schedulers such as Slurm or Kubernetes with MIG, vGPU or MPS technologies. But their use does not provide a general solution; different approaches are needed for different scenarios: HPC, Cloud Computing, etc. On the other hand, there are also works where the use of remote GPU virtualization is combined with the use of GPU scheduling [12,28], showing that applying both mechanisms at the same time reports large increments in GPU utilization. However, the mentioned studies are still far away from providing a general solution to industry.

As a response to the needs described above, in this work we propose NGS (Network GPGPU System), a general-purpose solution to address the underutilization of GPUs in computing centers. The name of our proposal, NGS, is a direct reference to the well-known protocol NFS (Network File System) used in distributed file systems in a computer network environment. NFS allows different nodes connected to the network to access remote files as if they were local. Furthermore, NFS allows concurrent access to these files without collisions. In a similar manner, NGS offers GPGPU services through the interconnection network. NGS allows different nodes connected to the network to share the system's physical GPUs in a robust and efficient way according to an established usage policy previously defined by the system administrator.

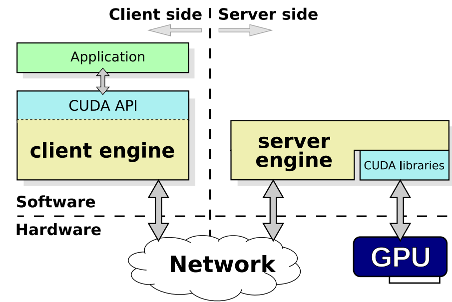


Fig. 1. Architecture of the rCUDA remote GPU virtualization framework.

To that end, NGS works on top of a remote GPU virtualization solution such as rCUDA, for instance, and orchestrates the concurrent access to GPGPU resources from different nodes. In this regard, NGS takes demands for GPGPU resources from several requesters and provides the best fit to maximize GPU utilization according to a given metric. Notice that requesters might be virtual machines, containers, regular applications, etc.

The main novelty of our proposal is that NGS is a general solution for coordinating the concurrent demands for GPGPU resources regardless of the exact location of the GPUs provided with respect to requesters. Notice that NGS works in a very different way to job schedulers. Slurm, for instance, takes requests for computing resources and forwards the scheduled job to the node that satisfies the demand. That is, Slurm moves jobs to the nodes where GPU resources are located. On the contrary, NGS fulfills demands for GPGPU resources from requesters by bringing GPGPU resources to them. That is, a requester being executed in a given node will be brought GPU resources that are physically located in other nodes of the cluster. Moreover, it is important to remark that using NGS on top of a remote GPU virtualization mechanism provides a much more flexible usage of the GPGPU resources in the cluster than using the MIG or vGPU solutions. In summary, NGS solves demands for GPGPU resources from requesters by providing the intelligence that is missing in remote GPU virtualization solutions.

The main contributions of this paper are: (i) introducing the NGS solution, (ii) an in-depth description of its architecture, implementation and operation; and (iii) an extensive performance evaluation.

The rest of the paper is organized as follows. Section 2 presents a review of the technologies on which NGS is based, mainly the rCUDA remote GPU virtualization framework and the NVIDIA NVML library [29]. Then, Section 3 presents related work. Next, Section 4 shows in detail the architecture of NGS, including both its implementation and operation modes. Later, a performance evaluation of NGS is shown in Section 5. Finally, Section 6 summarizes the main conclusions of this work and discusses potential future work directions.

2. Background

In this section we briefly describe the two components on which the implementation of NGS is based. These third-party components are the rCUDA remote GPU virtualization framework and the NVML library developed by NVIDIA. It is important to remark that the NGS solution is not tied to the rCUDA middleware. In this regard, NGS can also work with other remote GPU virtualization solutions, such as gVirtuS for instance. However, given that rCUDA supports more CUDA functions and features than GVirtuS, we will use rCUDA in this work.

2.1. rCUDA: remote CUDA

The rCUDA [17] remote GPU virtualization middleware allows a CUDA application running on a node which does not have a physical

GPU to access a GPU installed on a remote node. Fig. 1 shows the architecture of rCUDA. As it can be seen in the figure, rCUDA follows a distributed client–server approach. In the client side, there is a library providing the same API as CUDA. It is installed in the node running the application requesting GPU services. In the server side, a daemon is running in the node with the physical GPU installed. Communication between the client side and the server is carried out over the network.

Based on the architecture shown in Fig. 1, the execution flow of an application using rCUDA is as follows. Whenever the application performs a CUDA request, the client side intercepts the request, processes it, and forwards it to the server side. The daemon in the server side proceeds with the request, running it on the GPU. Once the request is finished, the server returns the results to the client, which forwards them to the application. It should be noted that this process is transparent to the application, which is not aware that the request was executed on a remote GPU. This transparent process also includes the source code of the application, which is not modified to use rCUDA. Actually, if the application was compiled to make use of the CUDA library in a dynamic way (linking to the library at run time), it is not necessary to recompile the application. The same binary can be used with rCUDA.

In order to use rCUDA, different environment variables must be defined in the client node prior to the execution of the application. In addition to force the use of the rCUDA library instead of the CUDA one by properly setting the `LD_LIBRARY_PATH` to point to the folder with the rCUDA binary, it is necessary to define the exact amount of remote GPUs that will be provided to the application as well as where those physical GPUs are located. That is, in which nodes of the cluster they are installed. As mentioned above, in the nodes with physical GPUs to be remotely used by the application, the rCUDA server should be in execution before the application is launched.

The environment variable `RCUDA_DEVICE_COUNT` is used to define the amount of accessible remote GPUs. Moreover, the `RCUDA_DEVICE_[ID]` variable is used to indicate the specific location of each of these remote GPUs. The location is specified by the IP:ID pair, where IP refers to the IP address of the server (or its name) and ID corresponds to the CUDA identifier of the GPU within that server. Finally, the memory requested on each GPU is defined by the environment variable `RCUDA_RESERVED_GPU_MEMORY_[ID]`. For example, to access remote GPUs 0 and 3, located on server 192.168.1.32 and reserving 1 GiB and 2 GiB of memory on them, respectively, we will define the following environment variables on the client side:

```
export RCUDA_DEVICE_COUNT=2;
export RCUDA_DEVICE_0=192.168.1.32:0;
export RCUDA_DEVICE_1=192.168.1.32:3;
export RCUDA_RESERVED_GPU_MEMORY_0=1024;
export RCUDA_RESERVED_GPU_MEMORY_1=2048;
```

Once the environment variables are defined, the application is launched and rCUDA takes care of connecting to the rCUDA server in the node with IP address 192.168.1.32 and forwarding every CUDA call to it.

2.2. NVIDIA NVML library

The NVIDIA Management Library (NVML) [29] provides a C-based API for monitoring and managing various states of the NVIDIA GPU devices. The runtime part of this library is included with the NVIDIA driver, and the NVIDIA Software Development Kit (SDK) provides the headers and the stub libraries to use it.

This library provides important information about the GPU states. Some of these parameters are: Error Correction Code (ECC), GPU and memory utilization, active compute process, clocks and Performance State (P-State), temperature and fan speed, power management, etc. In addition, it also allows users to activate or deactivate different modes of use of the GPU, such as the exclusive or concurrent computing mode, or the persistent driver loading mode.

As we will see in the next section, NGS uses the NVML library to gather the state of the system's GPUs in real time. Using this information, NGS is capable of distributing workloads between different GPUs efficiently.

3. Related work

As discussed in Section 1, from our point of view the best strategies to maximize the use of GPUs in current computing clusters are the ones based on: (i) GPU virtualization, (ii) GPU-job schedulers, or (iii) the combination of these two strategies. There are numerous works that exploit all these strategies.

In [30] the authors demonstrate that using NVIDIA MPS and MIG technologies the overall simulation throughput of full-GPU reservoir simulators can be improved significantly without any modifications to the software. In [31] the authors propose MISO, a technique that combines the use of MPS to predict the best partitioning of modern A100 and H100 GPUs using NVIDIA MIG. Using this technique authors reduce the execution time in deep learning workloads. These works demonstrate the virtualization capabilities offered by the latest generation of NVIDIA GPUs. However, these works lack the generality of our proposal. MIG is only available from the Ampere architecture. It also does not offer remote GPU virtualization. Even so, these works open up future research based on combining them with our proposal NGS, which looks promising.

ASTRAEA [25] and MARBLE [26] propose the use of two GPU job schedulers. They focus on improving performance during deep learning training. VAPOR [27] is a GPU sharing scheduler for distributed deep learning workloads on GPU clusters. Like ASTRAEA, VAPOR minimizes interference between jobs that share GPUs. The goal is to maximize GPU usage and reduce execution time. All these solutions are based on the development of custom GPU-job schedulers for deep learning-based workloads.

In [32,33] the feasibility of using remote virtualization of GPUs in virtualized environments is demonstrated. In these works, the use of the rCUDA framework is exploited to provide GPU acceleration in virtual machines under the Xen hypervisor. Results show that sharing GPUs between virtual machines improves system productivity by maximizing GPU utilization. Finally, [12,28] combine the use of remote GPU virtualization with general purpose schedulers, such as Slurm and the TSUBAME2.5 supercomputer scheduler. These works show how remote GPU virtualization offers a higher degree of freedom when scheduling GPU jobs. This results in an increase in overall system productivity. In both cases the proposals require ad hoc modifications to the schedulers to support remote GPU virtualization.

Next, we would like to further discuss and highlight the differences of the proposed solution with regards to existing works. In first place, one feature we would like to deeper discuss and highlight is that, in contrast to other works, NGS monitors GPU utilization at runtime. This enables the implementation of active scheduling policies, which is usually not possible with other solutions. In this sense, existing works that apply some kind of active scheduling policy are usually based on static data. For instance, these works profile a particular application. Then, based on the results of this profiling, they implement a scheduling policy, which varies, for example, depending on the specific kernel run on the GPU by the application. However, this scheduling is limited to static data and to the applications profiled. In contrast, our proposal uses data obtained at runtime. Apart from benefiting from a better scheduling, this makes NGS more general, and thus it can be used with any application.

This generality of our proposal is precisely another point we would like to highlight. Unlike other solutions, NGS is not developed for specific NVIDIA GPU architectures or models. Thus, NGS works for any NVIDIA GPU architecture and model, which makes it a general

approach. Additionally, the generality of our proposal can be further extended. In particular, the NGS daemon responsible for the active monitoring of GPUs could be adapted to other GPU manufacturers such as AMD or Intel. To that end, it would be only necessary to use the monitoring library provided by each manufacturer.

Finally, a very important feature that we would like to highlight is that, in addition to local physical GPUs, NGS also supports remote virtual GPUs. Contrary to other solutions, this support is transparent and does not require ad hoc modifications to the schedulers. Thus, thanks to NGS, an application running on a node without local physical GPUs installed could safely and transparently use remote virtual GPUs for improving its performance. Note that NGS decouples CPUs and GPUs, therefore applications can use CPUs and GPUs regardless of where they are physically installed.

4. NGS: Network GPGPU system

As mentioned above, NGS leverages a remote GPU virtualization solution and the NVML library in order to orchestrate the concurrent access to the GPGPU resources of the cluster from different nodes. To that end, it accepts requests for GPGPU resources and looks for the fit that maximizes GPU utilization according to some criteria. Although the idea is quite simple, the implementation gets much more complicated. This section provides an in depth insight of the NGS solution by presenting the internal architecture of NGS along with the way it works.

4.1. Architecture of NGS

NGS has a design based on a client–server architecture. It is therefore made up of two fundamental pieces, the *NGS server* and the *NGS client*. In addition, NGS also has a small daemon called *NGS monitor* which is executed in the nodes where physical GPUs exist. Next we present these components in more detail:

- NGS server. A single instance of the NGS server runs across the entire cluster. It is usually launched as a system daemon in one of the management nodes. All the intelligence of the NGS service is concentrated in the NGS server. The NGS server performs two fundamental tasks: (i) serving the requests coming from the different NGS clients and (ii) maintaining the status of the system GPUs by continuously communicating with the NGS monitor daemons running in the nodes with GPUs.
- NGS client. This lightweight program is used by requesters in order to communicate with the NGS server from any node in the cluster. There are three types of communications between NGS clients and the NGS server: (i) GPU resource request communications, (ii) GPU resource release communications and (iii) management communications, such as: show the status of the GPUs in the system, change the GPU scheduling policy, view the active jobs on each GPU, etc.
- NGS monitor. This small daemon is in execution in the nodes where the GPUs are physically installed. This daemon is in charge of registering the GPUs into the NGS server and monitoring the GPU status in real time and forwarding it to the NGS server. The monitoring of the GPUs is done by using the NVML library revisited in Section 2. The monitoring task is carried out at the default monitoring period of 10 s. Nevertheless, the frequency of monitoring can be set to other values.

Fig. 2 shows the typical configuration of a cluster using NGS. As we can see, there is a frontend or login node, a management node and several computing nodes. All of them can exchange data by means of an interconnection network. The NGS daemon must be a single process in the cluster and can be run in any node. In our example it is located in a management node. In the computing nodes where GPUs exist, the NGS monitor must be launched. Finally, the NGS client is used at any node in the cluster in order to request/release GPU resources or execute management tasks. Notice that, contrary to the NGS server or the NGS monitor, which are daemons, the NGS client is a regular program that is run when needed during the short time required to accomplish its function.

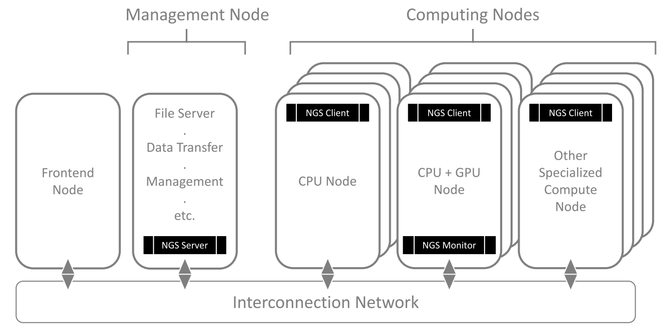


Fig. 2. Typical cluster configuration with NGS enabled.

4.2. How NGS works

As we have seen in Section 1, NGS works on top of a remote GPU virtualization solution (rCUDA in our case) and ensures that GPU computing resources are accessible from any requester, regardless of where the GPUs are physically located. In addition, NGS makes efficient use of GPUs by maximizing their utilization by following a pre-established usage policy. Furthermore, NGS guarantees that GPGPU resources in the cluster are shared among requesters in a robust way. For instance, it guarantees that when GPGPU resources in the cluster are exhausted, no additional request is accepted until some resources are previously released.

4.2.1. NGS start up

Continuing with the example in Fig. 2, the first step to start up the NGS system will be to start the NGS server in a management node. Afterwards, the NGS monitors will be started in the nodes that incorporate GPUs in order to monitor their status. By default all GPUs installed on each node are monitored. However, it is possible to select only some of them by leveraging the `CUDA_VISIBLE_DEVICES` environment variable provided by NVIDIA in the CUDA platform. The NGS monitors will contact the NGS server in an initial communication in order to registered the GPUs into the NGS system. Later, new connections will be established periodically in order to update the status of the GPUs in the NGS server in real time.[1]

4.2.2. Operation of NGS

In general, the operation of NGS is based on allocating and deallocating GPU computing resources on demand so that those resources are used by requesters thanks to the underlying GPU remote virtualization framework. Resources are requested or released through the use of the NGS client program, which transmits this information to the NGS server. The resources are requested by indicating the number of GPUs needed and also the available memory required on the GPUs.[2] An example of a resource request is shown below:

[1] Note that the underlying GPU remote virtualization framework must also be active at this point. In our case, we started the rCUDA servers together with the NGS monitors.

[2] Notice that requests can be much more complex. For instance, it is possible to define the type or model of GPU. It is also possible to specify conditions referring to their physical location. For example, it is possible to request two or more GPUs located on the same remote node, etc.

```
./NGS_client -S $SERVER -alloc -g 2 -m 1024
```

where $SERVER is the NGS server IP address or hostname, the number of GPUs to be allocated is 2 and the memory required per GPU is 1 GiB. The NGS server receives the resource request from the NGS client and then searches among all available GPUs for the candidates that best match the request. When the match is found, the NGS server sends back to the client the rCUDA environment variables (similar to those shown in Section 2.1) in order to enable the requester to use the remote GPUs. Two additional environment variables (not belonging to rCUDA but to NGS) are also sent: an operation identifier $NGS_OP_ID and an error code $NGS_ERROR.

When the requester does not need the GPUs anymore, the resources are released by indicating the NGS operation identifier. An example of such release could be:

```
./NGS_client -S $SERVER -dealloc -i $NGS_OP_ID
```

When this information reaches the NGS server, the GPU resources associated with the operation identifier $NGS_OP_ID will be released.

Concurrently with the request/release of GPU resources, the NGS server will update the status of the GPUs in the system. To that end, the NGS server will periodically communicate with the NGS monitors. In this way, the NGS server has information on the actual status (memory occupancy, usage percentage, temperature, etc.) of the GPUs in real time and can make scheduling decisions according to that information.

Finally, remember that the NGS client is also used to change options or query the NGS status. These queries are sent to the NGS server and it returns the result. For example, we can change the GPU monitoring period or the preset usage policy. It is also possible to query the status of the GPUs in the system.

4.3. Policies and working modes within NGS

When a request for GPGPU resources arrives at the NGS server, several policies can be applied during the process of selecting the best fit for that request. Additionally, in case there is no possible fit at the time of the request (for instance more GPUs are required by the requester than GPUs available at that time) it is possible to wait until the resources are available or, on the contrary, return an error.

In this section we present the available scheduling policies as well as the blocking and non-blocking modes that can be selected by the requester when placing the request.

4.3.1. Scheduling policies

There are two types of scheduling policies in NGS:

- Passive scheduling policies. When choosing GPUs to resolve a resource request, the NGS server relies exclusively on the amount of processes previously associated to the GPUs and on the previous reserved memory. Based on this information, the NGS server selects the candidate GPUs.
- Active scheduling policies. The NGS server relies on the real-time status of the GPUs (reported by the NGS monitors). That is, the candidate GPUs to resolve a resource request are chosen based on the percentage of GPU utilization or actual memory occupancy.

Currently NGS has the following passive policies:

- Round Robin (RR): GPUs are filled following a RR scheme.
- Performance (PERF): GPUs are filled by choosing the ones with fewer processes.
- Energy saving policy (ECO): GPUs are filled until all their memory is occupied. In this way, until a GPU memory is completely filled, the NGS server does not consider another GPU.

Active policies are under development. Currently there is only one active policy, called Best Fit (BF). In this policy the GPUs are chosen according to the minimum average GPU utilization percentage, reported by the NGS monitors, in a given period of time (by default last 5 min).

4.3.2. Blocking and non-blocking modes

Requests to NGS can be resolved in blocking and non-blocking modes, depending on the NGS configuration at request time:

- Blocking requests. When the blocking mode is used, the client is blocked until the requested GPUs are available and assigned. That is, if at the time of the request there are not enough available GPUs to satisfy the request, the client will be blocked until GPUs are available and the request can be serviced. Currently, requests are served in order of arrival following a FIFO strategy although future versions of NGS could leverage better approaches in order to maximize GPU utilization. Notice that an exception to the FIFO order is requests that inevitably cannot be serviced at any time. Instead of blocking the client, for these requests the server just sends an error code ($NGS_ERROR variable). An example of this could be a system with a total of 20 GPUs where a client requests for 21 GPUs.
- Non-blocking requests. This type of requests never block the client. They are resolved immediately by one of these three types of response: (i) if the requested resources are available they are assigned, (ii) if they are not available, an error code is sent indicating that they are not available, and finally (iii) if the requested resources are impossible to satisfy, an error code is sent indicating this condition. Notice that in the (ii) case, it is the requester who decides what to do if demanded resources are not available at that time.

Notice that blocking and non-blocking modes are orthogonal to scheduling policies. In this regard, these modes to not modify the selection process in the NGS server but influence the way that the NGS client proceeds when the demanded resources are not available: in the blocking mode the NGS client blocks until resources are granted whereas in the non-blocking mode the NGS client never blocks and the requester decides how to proceed in case the demanded resources are not available.

Finally, it should be noted that there is some configuration complexity. Depending on the number of physical GPUs in the system and the possible simultaneous requesters, it might happen that the aggregated GPGPU resources demanded by all requesters exceed the available resources. This could be a problem when using blocking requests, since an unsatisfied blocking request can block the whole system. All of these configuration details are the responsibility of the system administrator.

4.4. Trust model within NGS

An important concern that must be discussed is the trust model within NGS. That is, if security issues are not considered, it could be possible for users to misuse previous GPU allocations provided by NGS to execute new jobs demanding remote GPU resources. That could be carried out very easily just by appropriately setting the necessary environment variables prior to the execution of the accelerated application. Moreover, it could be even possible to aggregate the remote GPU resources provided by several previous allocations by NGS and execute applications using all those resources. In both cases, the consequences could be catastrophic because malicious users could use GPU resources that NGS has previously assigned to other users. This could also imply, for instance, that authorized users would see how their applications abort due to lack of GPU memory resources.

In order to avoid the security concerns mentioned above, a very simple mechanism can be implemented within NGS and rCUDA. This solution is the following: when a user requests GPU resources, NGS provides the set of environment variables to be used by the application, as mentioned in previous sections. Then, NGS will additionally provide a new environment variable consisting of a string where all the resources granted to the user are encoded. Moreover, NGS will also encode a grant identifier within that string and will store that identifier into its internal data. We could see this new environment variable as a

single-use key: once the user receives the result of the allocation from NGS and launches the application, the client side of rCUDA will use the key to decode the remote GPU resources granted to the user and will check whether those resources match the environment variables set by the user. Additionally, the rCUDA client will coordinate with NGS and check that the grant identifier has not been previously used. When NGS is asked for a grant identifier, NGS deletes that identifier from its internal data so that later queries using that identifier will provide negative answers. With this simple trust model, NGS guarantees that malicious users cannot use more resources than those granted. Additionally, it also ensures that they cannot use the result from previous allocations for future executions.

Finally, it is important to remark that this trust model is a proposal and it has not been incorporated into the current version of NGS. Therefore, in the performance evaluation carried out in Section 5, the possible overhead introduced by this trust model has not been considered. Nevertheless, the implementation of the trust model is a very interesting point that will probably require further research to minimize its overhead. Several possibilities can be devised to that end. For instance, in order to reduce the overhead of the trust model on the response time of the main NGS server, a separate daemon could be implemented to keep track of single-use keys. This separate daemon could even be executed in a different node to further reduce the overhead of the main NGS server response time. Another possibility could be to create small instances of this additional daemon and place them into the nodes of the system. In this way, when a user submits a request to the main NGS server, just after providing the allocation result, the main NGS server would provide the single-use key to the small daemon running in the node from which the request to NGS was submitted. When the application starts execution and the rCUDA client is loaded in that node, the rCUDA client would contact the small daemon with very little latency. This option will both reduce the overhead of the trust model on the main NGS server and reduce the overall time required to check the single-use key. Other possibilities to reduce the overhead of the trust model could also be considered.

4.5. Use cases for NGS

We have previously mentioned that the demands for GPGPU resources can be placed by virtual machines, regular applications, etc. Depending on the exact nature of the requester, a different use case of NGS is carried out.

When the requester is a virtual machine, that virtual machine can place the request during the startup stage. Later, the virtual machine will release the acquired GPGPU resources during shutdown. This can be done by making use of configuration files. This is similar to how the */etc/fstab* file is used during boot of an NFS system in order to mount the NFS partitions and then this partitions are released during shutdown. As can be seen, in this use case the GPGPU resources are granted to the requester (the virtual machine) during all the time that the virtual machine is active.

Another use case for NGS is associated with regular applications. In this case, the GPGPU resources will be requested just before launching an application that requires GPU acceleration and will be released just after finishing the execution of such application. To achieve that, a *launcher* could be used. That launcher wraps the command that runs the GPU-accelerated application and incorporates, prior to the command, the request for NGS resources (the call to the NGS client). Afterwards, once the application is completed, the launcher will call again the NGS client in order to release the GPGPU resources. Notice that from the point of view of efficiency in the use of GPU resources, this use case is much more efficient because GPU resources are only requested when strictly necessary. In the use case of the virtual machine above, GPU resources are granted to the virtual machine but they might not be in use all the time by applications.

More use cases for NGS are possible. For instance, it is possible to start a virtual machine without demanding GPGPU resources and ask for those resources from the inside of the virtual machine when GPU-accelerated applications are executed in the virtual domain.

4.6. Implementation

The implementation of all NGS components has been carried out using the C++ programming language to obtain a fast, powerful tool with low consumption of system resources.

The NGS server is the critical piece of the entire system since it is the most complex program. The NGS server follows the typical multi-threaded server structure in which each incoming request is served by a new thread. Once the request is serviced, the thread is destroyed. The information corresponding to each of the system's GPUs is organized into objects. To reduce access time to these objects within critical sections, some information is replicated across multiple objects.

The design of the NGS server has focused on the speed of responding to requests. To that end, we have made a large effort in order to reduce the execution time of internal critical sections within the NGS to the minimum possible.[3] Notice that execution time of these critical sections will become key when the amount of concurrent requests to the NGS server is large.

The NGS monitor is a daemon that runs on the nodes where the physical GPUs are installed. A process is created[4] per each GPU that is responsible for maintaining the updated status of the main parameters of the GPU on the server. Some of these parameters are: percentage of utilization, occupied memory, running processes, temperature, power consumed, etc.

Finally, the NGS client program follows a single-thread structure. A connection is set up with the NGS server to obtain/release GPU resources, or to change some server parameter.

5. Evaluation

In this section we evaluate our proposal. We use different metrics to assess the overhead it introduces to the system both in performance and resource consumption. First, we evaluate the overhead of making a generic request. We also perform an in-depth study on the parallel processing capacity. Next, we analyze the consumption of computational resources, focusing on average CPU utilization and memory consumption footprint. Finally, we compare our proposal with the widely used Slurm [24] resource scheduler. As mentioned in the previous section, note that in this section we do not study the overhead introduced by the trust model of NGS because it has not been implemented in this initial version of NGS.

5.1. Test bench

Our test bench consists of a total of nine nodes interconnected by a Gigabit Ethernet network. The hardware configuration is as follows:

- One node where the NGS server daemon runs. This node has a 16-core AMD EPYC 7282 processor and 128 GB of DDR4 memory.
- Four nodes, each equipped with an NVIDIA A100 GPU, a 16-core AMD EPYC 7282 processor and 128 GB of DDR4 memory. In these nodes run the NGS GPU monitoring processes.[5]
- Four nodes from which requests to NGS are made. Each of these four nodes is equipped with two 6-core Intel Xeon E5-2620 v2 processors and 32 GB of DDR3 memory.

[3] Critical sections imply sequential execution where only one thread of execution can advance. Therefore, it is crucial to obtain a good level of parallelism to reduce the percentage of execution time within these sections.

[4] In this case it was decided to create processes and not threads for greater robustness, since in the case of using threads, an error in one thread can end up propagating to the rest of the threads and ruin the monitoring of all the GPUs on the node.

[5] In this work we have performed a large scale evaluation of our system, reaching a total of 1024 GPUs. For this purpose, we have launched up to 256 NGS monitoring processes in each of these nodes.

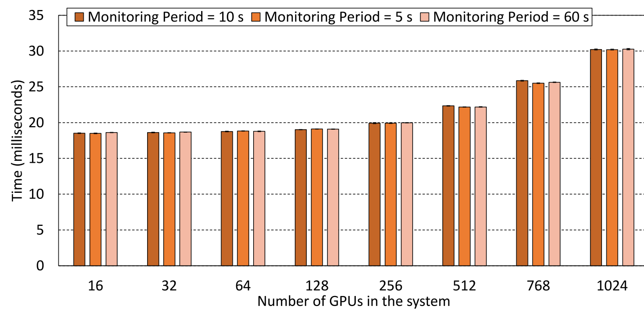


Fig. 3. Runtime overhead introduced by NGS when making a generic request for different monitoring periods.

Finally, in terms of software, all nodes have the Ubuntu 20.04 LTS operating system installed. The GPU nodes also have CUDA 11.3 installed with the NVIDIA driver version 465. 19.01.

5.2. Runtime overhead introduced by NGS

In this section we evaluate the runtime overhead that the overall system presents due to the use of NGS. As we have explained in Section 4, NGS can be used to provide GPGPU resources to applications by using a launcher that will wrap the GPU-accelerated application command with a request and a release of GPU resources. This introduces a small runtime overhead when requesting or releasing GPUs. We use a *generic request* to evaluate this overhead. This generic request consists of using the launcher to allocate a generic GPU,[6] run an empty job, i.e. an empty bash script, and release the GPU.

5.2.1. Overhead introduced by a generic request

Fig. 3 shows the execution time of a generic request under different conditions. We can observe the effect of varying the monitoring period, shown with different shades of brown. As we explained in Section 4, this monitoring period indicates the time between updates of the state of the system GPUs. Fig. 3 shows that varying this monitoring period does not influence the time needed to serve a request. Thus, the results are very similar regardless of the selected monitoring period: 10 (NGS default value), 5 or 60 s. This is because NGS is able to update the state of GPUs and resolve requests at the same time with almost no interference.[7] Fig. 3 also shows how the number of GPUs present in the system affects the time needed to solve a generic request. Or, in other words, the overhead introduced by NGS in the execution time of applications. As we can see, the execution time of a generic request varies from about 18.5 ms when we only have 16 GPUs in the system to about 30 ms when the number of GPUs is 1024.

To find the correlation between the number of GPUs in the system and the time needed to serve a generic request, Figs. 4 and 5 show the breakdown of a generic request on both the client and server sides. These figures show the main operations occurring on each side and their respective execution times.[8] In Figs. 4(a) and 5(a) we can see the execution time of the following operations at the client side:

- Pre-initialization. In this phase tasks are performed prior to the generic request. These tasks include the generation of a unique time stamp from which we create an auxiliary file to store the environment variables that NGS will provide us. The time cost of the creation of the NGS generic request process by the operating system is also accounted for in this phase.
- Create connection. In this phase the time consumed within the generic request process to establish the TCP connection with the NGS server daemon is accounted for.
- Send data. Time consumed during the operation of sending data to the NGS server (i.e. the *send* function). The data sent consists of a few bytes. Once the connection has been established, a packet is sent with information about the resources requested, such as number of GPUs, type of GPUs, etc. At the end, when the request finishes, a second packet is sent with information about the resources released.
- Recv data. Time consumed during the operation of receiving data from the NGS server (i.e. the *recv* function). This data contains the environment variables necessary for the correct operation of rCUDA, i.e. the environment variables indicating the GPU to be used.
- Job management. In this phase the empty job is executed. First, the environment variables received in the previous phase are exported. Then, an empty bash script is executed. Finally, the time cost of creating the NGS resource release process by the operating system is also accounted for.

Figs. 4(b) and 5(b) show the main tasks that occur on the server side:

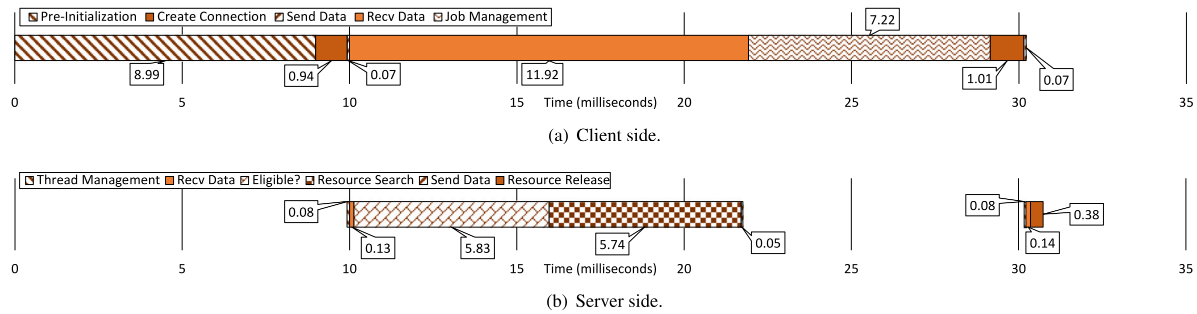
- Thread management. In the server, once a new connection is accepted, a new thread is generated to attend it. The socket of the connection is passed to this thread to communicate with the client and thus parallelize work. In this phase the time cost of creating and managing this thread is measured.
- Recv data. Time consumed during the operation of receiving the data sent by the client (i.e. the *recv* function). This data contains information about the resources requested options.
- Eligible? This phase evaluates whether the requested resources can be served or not. In other words, it is evaluated whether all the free GPUs in the system could satisfy the request. If this is not possible, an error is returned to the client indicating that the request cannot be satisfied. An example would be: a system consisting of only two GPUs where a request for three GPUs is received. Fig. 4(b) shows the general case, where it is assumed that the request can be served.
- Resource search. In this phase, the time required to obtain the exact resources requested by the client is accounted for.
- Send data. Time consumed during the operation of sending data to the client (i.e. the *send* function). The data sent consists of a few bytes and corresponds to the rCUDA variables that indicate where to find the requested resources.
- Resource Release. Time consumed in releasing resources that are no longer used.

The only difference between Figs. 4 and 5 is the amount of GPUs present in the system, 1024 and 256, respectively. If we analyze in depth Figs. 4(a) and 5(a), which show what happens on the client side, we can clearly see that the main difference resides in the data reception operation "Recv Data". The rest of the phases are executed in very

[6] NGS allows a large number of modifiers when requesting GPUs. We can choose the number of GPUs, the type, the amount of memory in the GPUs, etc. In this work by a generic GPU we refer to a single GPU regardless of type, memory, etc. and using a round robin policy.

[7] When in NGS it is required to update the state of a GPU, this operation may interfere with the scheduling operation (same critical region, implies sequential execution). However, the NGS design seen in Section 4 makes this collision time minimal and no overheads are observed in the experimental results.

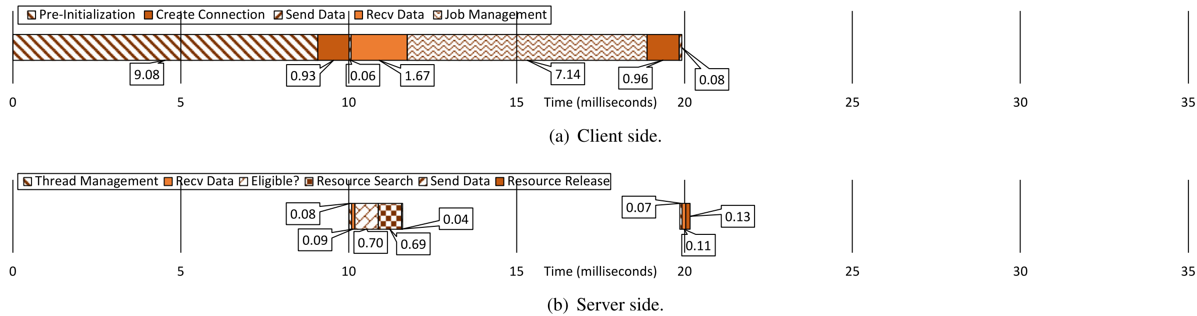
[8] Note that client and server are different nodes and therefore have different clocks. However, in Figs. 4 and 5 we used a common clock. To achieve this, we have manually synchronized both clocks. The synchronization has been done during the initial connection phase, where on the client side the *connect* function is performed and on the server side the *accept* function. Therefore, we assume that both operations conclude at the same time on both sides, ignoring the network time of a few microseconds. This time is negligible given the scale of the graph in milliseconds.



(a) Client side.

(b) Server side.

Fig. 4. Breakdown of a generic request in NGS with 1024 GPUs in the system and a monitoring period of 10 s.



(a) Client side.

(b) Server side.

Fig. 5. Breakdown of a generic request in NGS with 256 GPUs in the system and a monitoring period of 10 s.

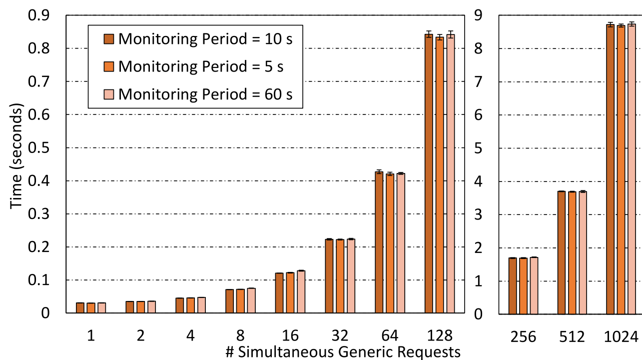


Fig. 6. Runtime overhead introduced by NGS with 1024 GPUs when making multiple simultaneous requests for different monitoring periods.

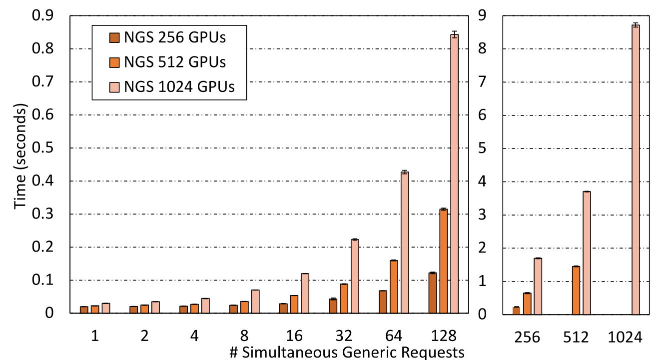


Fig. 7. Runtime overhead introduced by NGS with different number of GPUs when making multiple simultaneous requests. The GPU monitoring period used is the default one (i.e. 10 s).

similar times regardless of the number of GPUs in the system. However, on the server side shown in Figs. 4(b) and 5(b), we see three phases that reduce their duration if there are fewer GPUs in the system. These phases are: "Eligible?", "Resource Search" and "Resource Release". The rest of the phases remain approximately constant and are therefore not related to the number of GPUs in the system. This behavior is very interesting and shows that the duration of the "Eligible?", "Resource Search" and "Resource Release" phases is closely related to the number of GPUs in the system. The greater the amount of GPUs in the system, the longer the duration of these phases. This is because the time involved in exploring the different GPU structures to allocate or release resources increases with the number of GPUs in the system. The increase in the duration of the data reception phase on the client side is a direct consequence of what happens on the server side. This is because the data will not be sent to the client until it is computed on the server.

5.2.2. Parallel processing capacity

In this section we evaluate how NGS behaves when solving multiple simultaneous requests. As we saw in Section 4, the NGS design tries to maximize the parallel computation of the different phases that take place inside it.

Figs. 6 and 7 show the time required to serve batches of simultaneous generic requests of different sizes. These batches range in size from a single request to 1024 simultaneous requests. As we can see from Fig. 6, the variation of the monitoring period has no effect on the time required to serve multiple requests. However, as we can see in Fig. 7, as the number of GPUs in the system increases, the time needed to serve the batches of requests increases. These results are consistent with the analysis carried out in the previous section and we can attribute them to the same causes. Therefore, we can conclude that the time needed to serve concurrent generic requests does not depend on the monitoring period, but on the number of GPUs in the system.

We can also observe that the processing time of these batches is less than what we would obtain by sequentially serving the requests in these batches. In this regard, Fig. 8 shows the speedup obtained when processing concurrent requests with respect to their sequential execution time. The left side of the figure shows the results for NGS configured with 1024 GPUs, whereas the right side shows the results

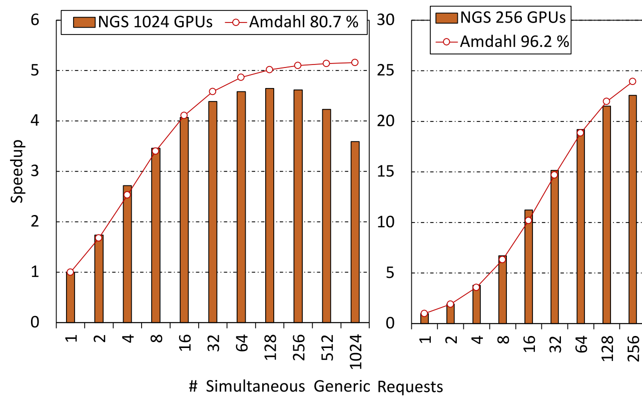


Fig. 8. Speedup obtained by executing concurrent generic requests compared to sequential execution time. The left side shows the results for NGS configured with 1024 GPUs. The right side shows the results for NGS configured with 256 GPUs.

for 256 GPUs. In each of the configurations in Fig. 8, we reach the maximum limit of concurrent requests that the NGS system can serve without waiting for the release of resources. Thus, in the 1024 GPUs configuration we reach 1024 concurrent requests, while in the 256 GPUs configuration we reach 256 concurrent requests. The maximum speedup achieved is 4.64x and 22.58x, for 1024 and 256 GPUs, respectively. There is a big difference in the acceleration obtained. It is due to the percentage of the execution time in which calculations are performed within the critical region. Note that these calculations have to be performed sequentially because only one thread has access to the critical region. The phases that cannot be executed in parallel are: (i) "Thread Management", in which the main thread is in charge of creating a new thread to attend the new request, and (ii) "Resource Search" phase, in which NGS looks for resources to serve the request. The sum of these two phases represents 19.3% of the execution time in the configuration with 1024 GPUs, and 3.8% with 256 GPUs.[9] For comparison purposes, we have added to Fig. 8 a red line corresponding to the theoretical speedup calculated using Amdahl's law. Theoretically, if NGS is configured with 1024 GPUs, 80.7% of the execution time can be improved by parallel processing. The percentage increases to 96.2% when NGS is configured with 256 GPUs. The term improvement corresponds to the number of simultaneous requests. As we can see, the results show that NGS is capable of achieving the theoretical speedup up to a certain point. Beyond that point, NGS begins to lose efficiency. This is due to the large number of threads and communications in the system, which increase proportionally to the number of simultaneous requests.

5.3. Resource consumption analysis

In this section we evaluate the resource consumption of NGS. To do this, we measure CPU usage and total memory used. First, we analyze the resources required for the operation of NGS. Next, we study the resources required to serve generic requests.

5.3.1. Resource consumption of NGS while not serving requests

As we saw in Section 4, NGS monitors the state of the GPUs in real time. The objective of this monitoring is to efficiently allocate resources according to the selected policy. Therefore, running NGS (without serving resource requests) has a computational cost. This cost depends on the GPUs that NGS manages. Fig. 9 shows the CPU and memory consumption of the NGS server as we increase the number of GPUs managed. The analysis shows 150 s of execution with the default monitoring period (10 s). In this experiment, every 30 s the number of GPUs in the system increases by 256 units. Thus, during the interval [0,30) s, the number of GPUs will be 0. In the interval [30,60), it will be 256. In the interval [60,90), 512 and so on until reaching 1024 GPUs. Results show that without GPUs the CPU consumption is approximately 3% of a CPU core. Regarding memory, the usage is close to 4300 KiB. With each batch of 256 GPUs, the CPU usage increases by approximately 2% of a CPU core and the size of the resident memory by approximately 2300 KiB. At the end of the test, with 1024 GPUs, the CPU usage is 11% of a CPU core and the memory used is 13 500 KiB. Note also that every time we add a batch of 256 GPUs, there is a 2% consumption spike in CPU usage that lasts between 1 and 2 s.

In Fig. 10 we also show the resources required for the operation of NGS. In this case we show how the variation in the period with which the GPUs are monitored affects the percentage of CPU usage.[10] As we can see, CPU consumption increases if the monitoring period is reduced and vice versa. This was expected, since by reducing the monitoring period we increase the computations because we have to update the state of the GPUs more times in the same interval. Regarding memory consumption, we found that it is independent of the monitoring period. Thus, variations in the monitoring period do not alter the memory consumed by NGS.

Finally, we also have to consider the resource consumption of the GPU monitoring in the nodes where GPUs are physically located. In these nodes there is one NGS monitor process for each physical GPU. The CPU resources required for this process are negligible (about 1% of a CPU core at each monitoring point. I.e. using the default monitoring period, we have a 1% consumption spike every 10 s). The memory consumption of this process is about 7000 KiB.

5.3.2. Resource consumption to serve generic requests

During the operation of NGS, the server must serve requests coming from the different NGS clients, which are running in the computing nodes of the cluster. In this section we evaluate the resource consumption of NGS when serving batches of generic requests.[11] To that end, we launch concurrent batches of requests for 15 s and measure the memory and CPU used.

In Fig. 11 we can observe the memory consumption of NGS when serving different number of concurrent generic requests. This Figure shows the results for 256, 512 and 1024 GPUs. In Fig. 11(a) we can see the average memory consumption, while Fig. 11(b) shows the maximum memory used. As we can observe, the memory consumed increases (i) with the number of concurrent requests, and also (ii) with the number of GPUs. In the worst case, serving 1024 concurrent requests with 1024 GPUs, NGS presents a maximum memory consumption around 160 MiB. In our test bench, the node running the NGS server is equipped with 128 GiB of memory. This represents a memory consumption of only 0.12%.

Fig. 12 shows the percentage of CPU usage of NGS when serving different amounts of concurrent generic requests. Again, the figure shows the results for configurations of 256, 512 and 1024 GPUs. The average CPU consumption is shown in Fig. 12(a), whereas the maximum CPU consumption is shown in Fig. 12(b). The maximum CPU consumption follows the same pattern as the memory consumption in

[9] Note that when processing a generic request it is also possible that GPU status updates are performed. In this case, there is also a small access time to the critical region. However, this has not been considered because, in general, it is negligible. In addition, it will be very difficult to calculate its impact.

[10] Note that in Fig. 10 the average CPU consumption is shown. We measure each of the configurations for 30 s and then calculate the average over those 30 s.

[11] The resource consumption experienced when performing a generic request on the client side is negligible. Note that the request on the client consists only of creating a TCP connection to the NGS server and transmitting a few bytes to request and release resources.

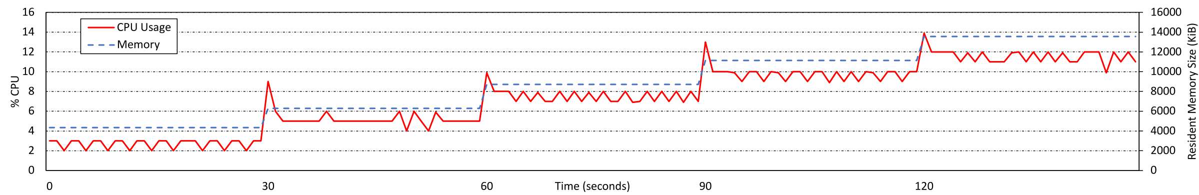


Fig. 9. CPU and memory usage of the NGS server as the number of GPUs managed by the system increases. Every 30 s, a batch of 256 GPUs is added to the system.

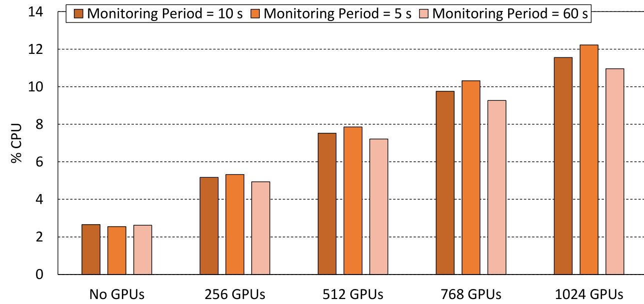


Fig. 10. Average CPU utilization in NGS server with different number of GPUs and different monitoring periods. Notice that the percentage is related to the capacity of a single CPU core.

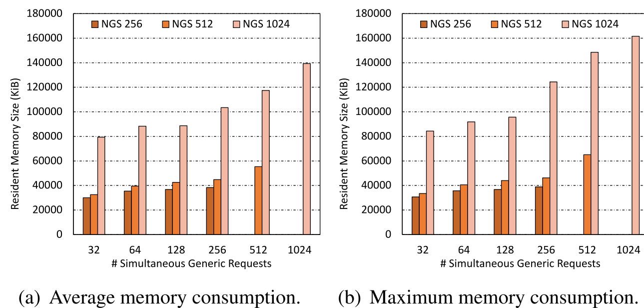


Fig. 11. Memory consumed by NGS when serving batches of generic requests with different number of GPUs and default monitoring period (i.e. 10 s).

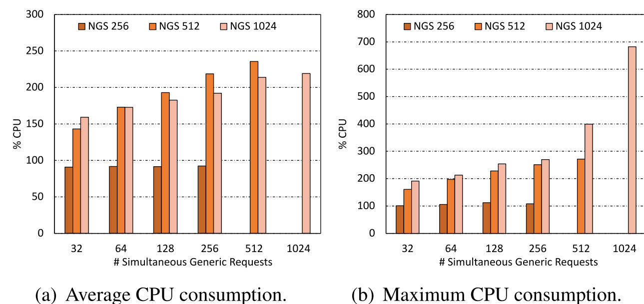


Fig. 12. Percentage of CPU consumed by NGS when serving batches of generic requests with different number of GPUs and default monitoring period (i.e. 10 s).

Fig. 11(b). Thus, the CPU consumption increases with the number of concurrent requests and also with the number of GPUs. However, this is not entirely true for average CPU consumption. With 256 GPUs, the average CPU consumption is constant regardless of the request batch size. With 512 and 1024 GPUs, the average CPU consumption increases with the number of concurrent requests. But unexpectedly, the highest average CPU consumption is obtained with 512 GPUs and not with 1024 GPUs.

There are probably several reasons working together for this anomalous behavior. One of them might be that, as this only happens with average values (Fig. 12(a)), one of the reasons for this anomalous behavior is probably related to the fact that increasing the number of GPUs results in increasing the "Resource Search" phase seen in Figs. 4(b) and 5(b). This phase, being inside a critical section of the code, will force the sequential execution, consequently the average CPU consumption will be reduced.

Regardless of the reasons for the anomalous behavior, in the worst case, serving 1024 concurrent requests with 1024 GPUs, NGS presents a maximum CPU consumption of up to 700%.[12] In our test bench, the node running the NGS server is equipped with 16-cores (32 execution threads). This represents a maximum CPU consumption of up to 22% of the overall CPU resources. Notice also that this CPU consumption lasts for a very short period of time (while scheduling is in progress).

As a summary of the results in this section, having analyzed both NGS operation consumption with no requests and also consumption under intense batches of generic requests, we can establish that NGS in our system has a low memory consumption at all times. It also presents a low CPU consumption in the general case (NGS operation with no request), and a consumption between 3% and 8% with peaks up to 20% in punctual moments of high workload (Fig. 7 shows the duration of the request batches).

5.4. Comparison with Slurm

In this section we compare the overhead introduced by NGS at run time with the overhead introduced by a widely used job scheduler such as Slurm. It is important to remark that NGS and Slurm have different purposes and also work in a different way, although from a simplistic point of view both them carry out a scheduling task. On the one hand, Slurm is a job scheduler (this includes GPU jobs) that looks for the best fit according to the job requirements and, once found, forwards the job to the selected node. The computing resources in that node (CPU and GPU) are leveraged to complete job execution. Once finished, the results will be brought back to the node from which the request for Slurm was initially carried out. On the other hand, NGS does not move jobs to selected nodes by the scheduling process. Actually, the scheduling stage does not selects nodes but portions of GPU computing resources. In this regard, a requester (job, container, virtual machine, etc.) places a demand for some GPU resources of the cluster and then NGS looks for the best fit of those GPU resources. Once found, NGS assigns that portion of resources to the requester, which is granted access by NGS from the requesting node. That is, the requester remotely accesses the GPU resources from the node where the requester was in execution when it placed the demand. Thus, the job is executed on the node without GPU using a remote GPU installed in other node of the cluster. When the job finishes, the access to the remote is released. Anyway, despite Slurm and NGS have different purposes, we believe that comparing their overheads is a good way to put the results of previous sections into context.

[12] Note that the value 100% is equivalent to the use of one *logical* CPU (i.e. execution thread) in the system. The use of more CPUs is expressed with values higher than 100%. In our test bench, the node running the NGS server is equipped with 16-cores (32 execution threads). Thus, the maximum CPU consumption is 3200%.

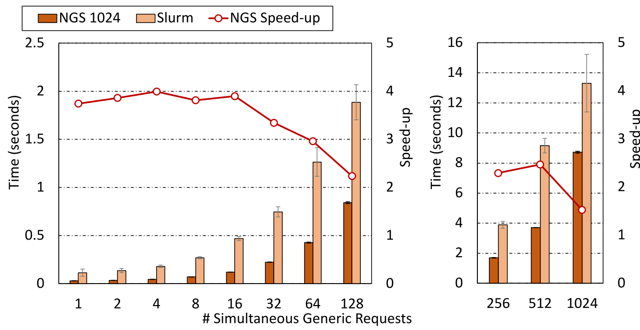


Fig. 13. Comparison of the runtime overhead introduced by NGS and Slurm when serving multiple concurrent requests with 1024 GPUs.

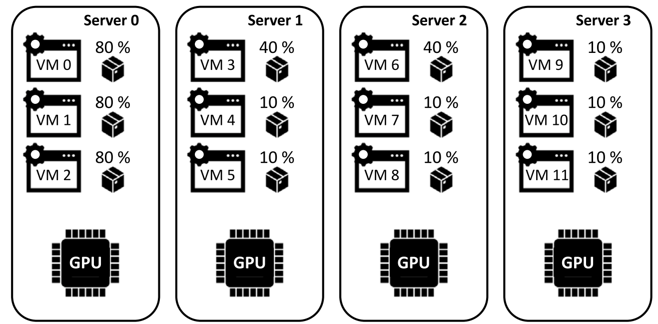


Fig. 14. Configuration of the testbed used for carrying out the performance comparison of NGS and NVIDIA MIG. VM refers to virtual machine, while the percentages indicate the computational intensity of the load in the GPU.

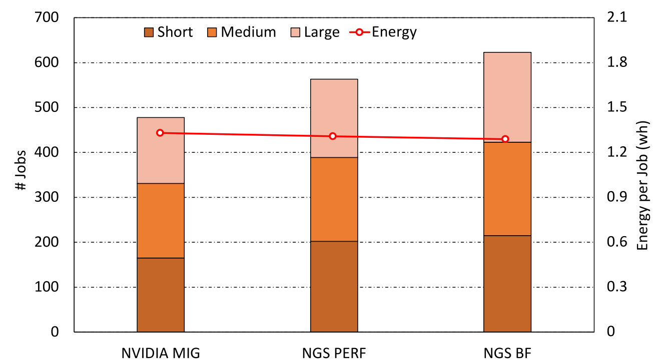


Fig. 15. Number of completed jobs and energy consumption per job with NVIDIA MIG and NGS configurations.

The Slurm setup in our test bench is similar to that of NGS. The main Slurm daemon runs on the same node where the NGS server runs. We use the same four nodes with GPUs and launch jobs from the four additional nodes without GPUs. The jobs submitted to the Slurm queues are empty jobs that require one GPU. Notice that we are only interested in measuring the overhead of scheduling a GPU job with Slurm and the overhead of serving a GPU request with NGS. In this regard, we are not interested on the time required to execute a batch of jobs. This is why we use empty jobs in our test.

Similarly to what we did with NGS in previous sections, we have submitted batches of concurrent GPU jobs to a Slurm setup with 1024 GPUs. These GPU jobs are submitted from the nodes where there are no GPUs. Fig. 13 compares these results with those from NGS (using default monitoring period).

As we can observe in Fig. 13, the overhead introduced by NGS and Slurm are of the same order of magnitude. However, the overhead introduced by NGS is noticeably lower (especially for smaller batch sizes). Fig. 13 also shows the improvement (i.e. speed-up) of NGS over Slurm. We can see that it ranges from up to 4x for smaller batch sizes, to 1.5x for larger ones. In general, as the batch size increases, the speed-up of NGS over Slurm reduces.

5.5. NGS in action: An example

To conclude the analysis of our proposal, we have conducted a final experiment. The objective of this experiment is to assess the performance of NGS in terms of throughput, execution time, GPU utilization, and energy consumption. To achieve that, we compare NGS with a conventional configuration that makes use of the NVIDIA MIG technology.

5.5.1. Testbed for the experiment

The testbed for the experiment deploys virtual machines (VMs) on a set of servers to execute GPU jobs in all the VMs for one hour. The hardware configuration for this experiment involves the four servers equipped with A100 GPUs described in Section 5.1. Each of these servers will host three VMs, as shown in Fig. 14.

The GPU jobs used in the experiment will consist of synthetic tests based on a modified version of the CUDA SDK sample MonteCarloMultiGPU [34]. These GPU jobs aim to mimic different behavioral patterns found in real GPU applications. To ensure diversity, we have modified the MonteCarloMultiGPU sample with varying execution times: short (approximately 10 s), medium (50 s), and large (100 s). We have also included the possibility of having different computational intensities, with average GPU utilizations of 10%, 40%, and 80%. Also, GPU memory occupancy can be 2 GiB, 4 GiB or 8 GiB. Finally, the inter-arrival time of jobs for each virtual machine varies randomly from 0 to 10 s.

The configuration for the experiment is depicted in Fig. 14. VMs on server 0 exclusively run jobs with an average GPU utilization of 80%. On servers 2 and 3, jobs are executed with GPU utilization of 40% and 10%, as shown in Fig. 14. Finally, on server 3, all VMs handle jobs with low GPU computational demands (i.e. 10%). The remaining parameters for the GPU jobs (execution time, memory occupancy and inter-arrival time) are randomly set for each execution.

As previously mentioned, in this experiment we conduct a performance comparison of NGS and NVIDIA MIG. The NVIDIA MIG configuration is based on the division of each physical GPU into three equal virtual MIG instances of type *2g.10gb*. Each of these virtual instances features 10 GiB of memory, which is enough to host the jobs run in the experiments. Furthermore, as the physical GPU is split into three virtual instances, each instance have a computational power equivalent to 28.6% of the original physical GPU.[13]

NGS is configured in blocking mode, employing the PERF (passive) and BF[14] (active) scheduling policies. NGS usage associated with applications is facilitated through a *launcher*, as explained in the second use case of Section 4.5. It is important to remark that when NGS is used in this experiment, VMs have access to any of the four GPUs thanks to the use of rCUDA.

As a summary of our experiment, we run the aforementioned MIG configuration for one hour and measure, among other results, how many jobs have been completed during that time period. In a similar way, we gather performance results for the one-hour execution with NGS PERF and also for the one-hour execution with NGS-BF.

[13] The ideal case would have been generating three virtual instances with a computational capacity equal to 1/3 of the physical GPU. Unfortunately, MIG is quite rigid in this aspect and the closest possible combination to the ideal case is the one we have selected (i.e. 2g.10gb).

[14] The time period for evaluating the average GPU utilization in the BF active policy has been set to 10 s.

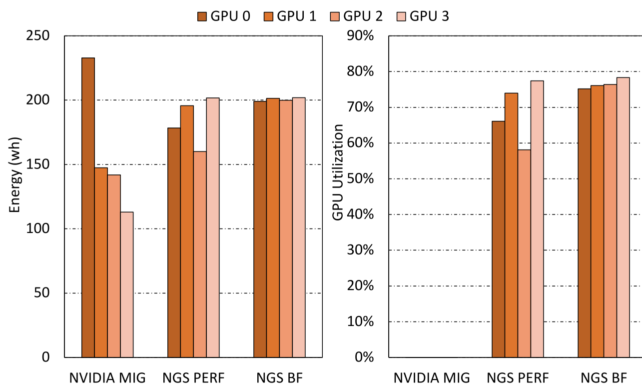


Fig. 16. Energy consumption and GPU utilization for NVIDIA MIG and NGS. Note that for MIG it is not possible to collect GPU utilization data.

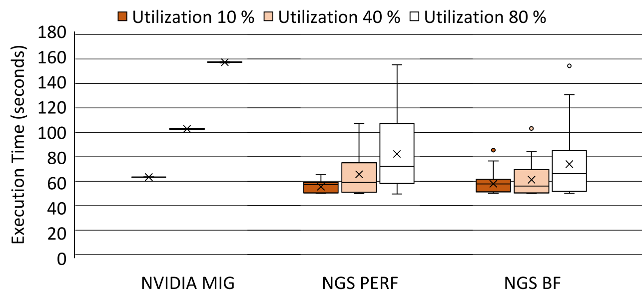


Fig. 17. Box–whisker plot showing the distribution of job execution time for NVIDIA MIG and NGS. Note that this plot only considers medium jobs.

5.5.2. Performance results

The first performance result to consider in our experiment is the throughput of the whole system, in terms of completed jobs during the one-hour period of the experiment. The amount of completed jobs is shown in Fig. 15 broken down into short, medium and large jobs. Results are shown for the different configurations considered (MIG, NGS PERF and NGS BF). Average energy per job is also shown in the figure. As it can be seen, the figure clearly shows that the configurations using NGS provide a much higher number of completed jobs than MIG. Thus, the throughput of NGS PERF and BF is 18% and 30% higher than that of MIG, respectively. Furthermore, it can also be observed that the average energy per job slightly decreases as performance increases.

Fig. 16 shows the energy consumption and GPU utilization for each of the four GPUs in the experiment. In the case of using MIG, it can be seen that GPU 0 consumes more energy due to its higher workload. On the contrary, GPU 3 presents the lowest energy consumption, as expected. Therefore, when using MIG the load is not well balanced between the four GPUs. Interestingly, when using NGS all the GPUs present a more balanced power consumption. This is particularly noticeable when the BF active scheduling policy is used. Similar results are achieved for GPU utilization. Notice that, in this case, GPU utilization results for the MIG case cannot be gathered because of the limitations imposed by NVIDIA.

Finally, Fig. 17 shows a box–whisker plot with job execution times. In the plot, the “X” symbol refers to the average execution time. It can be seen that the MIG configuration provides always the same performance. Actually, these are the expected results because MIG splits the physical GPU into three independent partitions. On the contrary, when NGS is used, average execution time is lower than with MIG although the variability among jobs is much larger. This is expected because, with NGS, GPUs are not split into independent partitions but all the jobs being executed in a given GPU compete for the GPU computing resources. It is interesting to remark that the BF active policy provides lower execution time variability than the PERF passive policy.

6. Conclusions and future work

In this paper we have presented NGS (Network GPGPU System), a general-purpose solution to address the underutilization of GPUs in computing centers. NGS offers GPGPU services through the interconnection network. It enables different nodes of the cluster to access remote GPUs as if they were local. In addition, interconnected nodes can concurrently share these GPUs without collisions. The main novelty of our proposal, compared to other works, is that it offers a global and standard solution independent of the computing environment used. It can be applied in all types of environments: HPC clusters, virtual machines, containers, Cloud Computing, AI scenarios, IoT devices, etc.

After thoroughly describing the architecture of NGS, we have done an extensive performance evaluation. In addition to measure NGS performance, we have also compared NGS to other approaches in order to put the results into context. Experimental results show up to 4x improvements compared to popular approaches.

Finally, we detail next some potential future work lines that are interesting from our point of view:

- NGS active policies. NGS monitors GPU utilization at runtime. This enables the design of active scheduling policies for GPU jobs, which are not usually provided by a job scheduler. Implementing and evaluating these policies is another future research line that we would like to explore.
- Support for other GPU manufacturers. Currently, NGS supports NVIDIA GPUs. A future work will consider extending this support to other GPU manufacturers such as AMD or Intel.
- NGS request semantics. Requests can be enhanced by including additional parameters such as user priority and request priority for a given user. This could also consider job preemption. Thus, lower priority jobs would be preempted to allow higher priority ones to immediately obtain the resources required [35].
- Job schedulers. NGS offers a lot of flexibility. A future research line includes integrating NGS with popular job schedulers to further show its benefits.
- Real applications. In this paper we have used synthetic tests. As future work we plan to perform experiments with real applications.
- Security. In this work we have discussed a potential trust model for NGS. In the future we plan to actually implement it and evaluate its influence in performance.

CRediT authorship contribution statement

Javier Prades: Writing – original draft, Investigation, Funding acquisition. **Carlos Reaño:** Writing – review & editing, Validation, Supervision, Formal analysis. **Federico Silla:** Formal analysis, Funding acquisition, Supervision, Validation, Writing – review & editing.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

Javier Prades was supported by Margarita Salas Postdoctoral Fellowship (UP2021-036) from Universitat Politècnica de València Funded by Ministerio de Universidades and the European Union-Next Generation EU.

References

[1] J. Krüger, R. Westermann, Linear algebra operators for GPU implementation of numerical algorithms, ACM Trans. Graph. 22 (3) (2003) 908–916, http://dx.doi.org/10.1145/882262.882363.
[2] J. Bolz, I. Farmer, E. Grinspun, P. Schröder, Sparse matrix solvers on the GPU: Conjugate gradients and multigrid, ACM Trans. Graph. 22 (3) (2003) 917–924, http://dx.doi.org/10.1145/882262.882364.
[3] D. Luebke, M. Harris, N. Govindaraju, A. Lefohn, M. Houston, J. Owens, M. Segal, M. Papakipos, I. Buck, GPGPU: General-purpose computation on graphics hardware, in: Proceedings of the 2006 ACM/IEEE Conference on Supercomputing, SC '06, Association for Computing Machinery, New York, NY, USA, 2006, pp. 208–es, http://dx.doi.org/10.1145/1188455.1188672.
[4] J. Nickolls, I. Buck, M. Garland, K. Skadron, Scalable parallel programming with CUDA: Is CUDA the parallel programming model that application developers have been waiting for? Queue 6 (2) (2008) 40–53, http://dx.doi.org/10.1145/1365490.1365500.
[5] NVIDIA, CUDA toolkit - free tools and training, 2023, URL https://developer.nvidia.com/cuda-toolkit.
[6] P. Du, R. Weber, P. Luszczek, S. Tomov, G. Peterson, J. Dongarra, From CUDA to OpenCL: Towards a performance-portable solution for multi-platform GPU programming, Parallel Comput. 38 (8) (2012) 391–407, http://dx.doi.org/10.1016/j.parco.2011.10.002, URL https://www.sciencedirect.com/science/article/pii/S0167819111001335. APPLICATION ACCELERATORS IN HPC.
[7] OpenCL, Opencl overview - the khronos group inc, 2023, URL https://www.khronos.org/opencl.
[8] TOP500 project, Top500 the list, 2023, URL https://www.top500.org/.
[9] W.J. Dally, S.W. Keckler, D.B. Kirk, Evolution of the graphics processing unit (GPU), IEEE Micro 41 (06) (2021) 42–51, http://dx.doi.org/10.1109/MM.2021.3113475.
[10] NVIDIA, NVIDIA H100 PCIe product brief, 2022, URL https://www.nvidia.com/content/dam/en-zz/Solutions/gtcs22/data-center/h100/PB-11133-001_v01.pdf.
[11] TOP500 project, Green500 the list, 2023, URL https://www.top500.org/lists/green500.
[12] S. Iserte, J. Prades, C. Reaño, F. Silla, Improving the management efficiency of GPU workloads in data centers through GPU virtualization, Concurr. Comput. Pract. Exp. 33 (2) (2021) http://dx.doi.org/10.1002/cpe.5275.
[13] G. Yeung, D. Borowiec, A. Friday, R. Harper, P. Garraghan, Towards GPU utilization prediction for cloud deep learning, in: 12th USENIX Workshop on Hot Topics in Cloud Computing, HotCloud 20, USENIX Association, 2020, URL https://www.usenix.org/conference/hotcloud20/presentation/yeung.
[14] NVIDIA, Multi-process service (MPS), 2023, URL https://docs.nvidia.com/deploy/mps/index.html.
[15] NVIDIA, Virtual GPU software - user guide, 2023, URL https://docs.nvidia.com/grid/latest/pdf/grid-vgpu-user-guide.pdf.
[16] NVIDIA, NVIDIA multi-instance GPU - user guide, 2023, URL https://docs.nvidia.com/datacenter/tesla/pdf/NVIDIA_MIG_User_Guide.pdf.
[17] F. Silla, S. Iserte, C. Reaño, J. Prades, On the benefits of the remote GPU virtualization mechanism: The rCUDA case, Concurr. Comput.: Pract. Exper. 29 (13) (2017) e4072, http://dx.doi.org/10.1002/cpe.4072, arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/cpe.4072. URL https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.4072. e4072 cpe.4072.
[18] GVirtuS, A GPGPU transparent virtualization component for high performance computing clouds, 2023, URL https://github.com/gvirtus/GVirtuS.
[19] G. Giunta, R. Montella, G. Agrillo, G. Coviello, A GPGPU transparent virtualization component for high performance computing clouds, in: P. D'Ambra, M. Guarracino, D. Talia (Eds.), Euro-Par 2010 - Parallel Processing, Springer Berlin Heidelberg, Berlin, Heidelberg, 2010, pp. 379–391.
[20] J. Kennedy, B. Varghese, C. Reaño, AVEC: Accelerator virtualization in cloud-edge computing for deep learning libraries, in: 5th IEEE International Conference on Fog and Edge Computing, ICFEC, 2021, pp. 37–44.
[21] J. Kennedy, V. Sharma, B. Varghese, C. Reaño, Multi-tier GPU virtualization for deep learning in cloud-edge systems, IEEE Trans. Parallel Distrib. Syst. 34 (7) (2023) 2107–2123, http://dx.doi.org/10.1109/TPDS.2023.3274957.
[22] Z. Rejiba, J. Chamanara, Custom scheduling in kubernetes: A survey on common problems and solution approaches, ACM Comput. Surv. 55 (7) (2022) http://dx.doi.org/10.1145/3544788.
[23] OpenPBS, Openpbs open source project, 2023, URL https://www.openpbs.org/.
[24] Slurm, Slurm workload manager, 2023, URL https://slurm.schedmd.com/documentation.html.
[25] Z. Ye, P. Sun, W. Gao, T. Zhang, X. Wang, S. Yan, Y. Luo, ASTRAEA: A fair deep learning scheduler for multi-tenant GPU clusters, IEEE Trans. Parallel Distrib. Syst. 33 (11) (2022) 2781–2793, http://dx.doi.org/10.1109/TPDS.2021.3136245.
[26] J. Han, M.M. Rafique, L. Xu, A.R. Butt, S.-H. Lim, S.S. Vazhkudai, MARBLE: A multi-GPU aware job scheduler for deep learning on HPC systems, in: 2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing, CCGRID, 2020, pp. 272–281, http://dx.doi.org/10.1109/CCGrid49817.2020.00-66.
[27] X. Zhu, L. Gong, Z. Zhu, X. Zhou, Vapor: A GPU sharing scheduler with communication and computation pipeline for distributed deep learning, in: 2021 IEEE Intl Conf on Parallel & Distributed Processing with Applications, Big Data & Cloud Computing, Sustainable Computing & Communications, Social Computing & Networking, ISPA/BDCloud/SocialCom/SustainCom, 2021, pp. 108–116, http://dx.doi.org/10.1109/ISPA-BDCloud-SocialCom-SustainCom52081.2021.00028.
[28] P. Markthub, A. Nomura, S. Matsuoka, Using rCUDA to reduce GPU resource-assignment fragmentation caused by job scheduler, in: 2014 15th International Conference on Parallel and Distributed Computing, Applications and Technologies, 2014, pp. 105–112, http://dx.doi.org/10.1109/PDCAT.2014.26.
[29] NVIDIA, NVIDIA management library NVML, 2023, URL https://developer.nvidia.com/nvidia-management-library-nvml.
[30] R. Gandham, Y. Zhang, K. Esler, V. Natoli, Improving GPU throughput of reservoir simulations using NVIDIA MPS and MIG, in: Fifth EAGE Workshop on High Performance Computing for Upstream, Vol. 2021, European Association of Geoscientists & Engineers, 2021, pp. 1–5, http://dx.doi.org/10.3997/2214-4609.2021612025, (1) URL https://www.earthdoc.org/content/papers/10.3997/2214-4609.2021612025.
[31] B. Li, T. Patel, S. Samsi, V. Gadepally, D. Tiwari, MISO: Exploiting multi-instance GPU capability on multi-tenant GPU clusters, in: Proceedings of the 13th Symposium on Cloud Computing, SoCC '22, Association for Computing Machinery, New York, NY, USA, 2022, pp. 173–189, http://dx.doi.org/10.1145/3542929.3563510.
[32] J. Prades, F. Silla, Made-to-measure GPUs on virtual machines with rCUDA, in: The 47th International Conference on Parallel Processing, ICPP 2018, Workshop Proceedings, Eugene, OR, USA, August 13-16, 2018, ACM, 2018, pp. 19:1–19:8, http://dx.doi.org/10.1145/3229710.3229741.
[33] J. Prades, C. Reaño, F. Silla, On the effect of using rCUDA to provide CUDA acceleration to Xen virtual machines, Clust. Comput. 22 (1) (2019) 185–204, http://dx.doi.org/10.1007/s10586-018-2845-0.
[34] V. Podlozhnyuk, M. Harris, Monte Carlo option pricing, 2012, URL https://developer.download.nvidia.com/compute/DevZone/C/html_x64/4_Finance/MonteCarloMultiGPU/doc/MonteCarlo.pdf.
[35] J. Prades, F. Silla, GPU-job migration: The rCUDA case, IEEE Trans. Parallel Distrib. Syst. 30 (12) (2019) 2718–2729, http://dx.doi.org/10.1109/TPDS.2019.2924433.