Providing CUDA Acceleration to KVM Virtual Machines in InfiniBand Clusters with rCUDA

Ferran Pérez, Carlos Reaño(✉), and Federico Silla

DISCA, Universitat Politècnica de València, 46022 Valencia, Spain
ferpelo@upv.es, carregon@gap.upv.es, fsilla@disca.upv.es

Abstract. There is a trend towards using graphics processing units (GPUs) not only for graphics visualization, but also for accelerating scientific applications. But their use for this purpose is not without disadvantages: GPUs increase costs and energy consumption. Furthermore, GPUs are generally underutilized. Using virtual machines could be a possible solution to address these problems, however, current solutions for providing GPU acceleration to virtual machines environments, such as KVM or Xen, present some issues. In this paper we propose the use of remote GPUs to accelerate scientific applications running inside KVM virtual machines. Our analysis shows that this approach could be a possible solution, with low overhead when used over InfiniBand networks.

Keywords: CUDA · KVM · Virtualization · InfiniBand · HPC

1 Introduction

Virtual machine (VM) technologies such as KVM [1], Xen [7], VMware [6], and VirtualBox [3] appeared several years ago in order to address some of the concerns present in computing. One of the issues addressed by VMs was data and/or process isolation. That is, without the use of VMs, in a computing cluster providing service to different institutions and companies, each of the nodes of the cluster should only host processes from a single owner if data or process isolation is a requirement. This guarantee for data security, in addition to some other concerns, led in general to low CPU and system utilization, presenting the additional indirect drawbacks of an unnecessarily increased power consumption as well as an increased hardware acquisition cost and higher space and cooling requirements. Virtualization technologies addressed all these concerns by creating virtual computers that are concurrently executed within a single cluster node thus sharing the CPU in that node as well as other subsystems and, therefore, increasing overall resource utilization. Acquisition and maintenance costs are also reduced because a smaller amount of computers are required to address the same workload, thus reducing also energy consumption needs. Finally, data isolation is expected because different VMs manage separate address spaces,

M. Jelasity and E. Kalyvianaki (Eds.): DAIS 2016, LNCS 9687, pp. 82–95, 2016.
DOI: 10.1007/978-3-319-39577-7_7

thus making not possible that a process being executed in one VM addresses memory belonging to other VM.

The importance that VMs have acquired in data centers can be understood just by considering all the support included for them in current mainstream multicore processors from Intel or AMD. Actually, although VMs were known in the past to noticeably reduce application performance with respect to executions in the native (or real) domain, the virtualization features included in current CPUs allow VMs to execute applications with a negligible overhead [15].

However, despite the many achievements accomplished in the field of VMs, they still do not efficiently support the current trend of using graphics processing units (GPUs). This trend allows that many high-performance computing (HPC) clusters deployed in current datacenters and other computing facilities benefit from server configurations that include several multicore CPU sockets and one or more GPUs. In this way, these heterogeneous configurations noticeably reduce the time required to execute applications from areas as different as data analysis (Big Data) [36], chemical physics [32], computational algebra [13], and finance [16], to name just a few. Unfortunately, the lack of efficient GPU support in current VMs makes that, when using these virtualization technologies, applications being executed in the virtualized domain cannot easily access GPUs in the native domain.

The reason why VMs cannot take advantage of the benefits of using GPUs is mainly due to the fact that current GPUs do not feature virtualization capabilities. Furthermore, in those cases where it is possible for applications within VMs to access real GPUs, by using the PCI passthrough mechanism [35], for example, accelerators cannot be efficiently shared among the several VMs concurrently running inside the same host computer. These limitations impede the deployment of GPGPU computing (general-purpose computing on GPUs) in the context of VMs. Fortunately, GPU virtualization solutions such as V-GPU [5], dOpenCL [22], DS-CUDA [30], rCUDA [14,31], vCUDA [33], GridCuda [26], SnuCL [23], GVirtuS [17], GViM [19], VOCL [37], and VCL [12] may be used in VM environments, such as KVM, VirtualBox, VMware, or Xen, in order to address their current concerns with respect to GPUs. These GPU virtualization frameworks detach GPUs from nodes, thus allowing applications to access virtualized GPUs independently from the exact computer they are being executed at. In this regard, the detaching features of remote GPU virtualization frameworks may turn them into an easy and efficient way to overcome the current limitations of VM environments regarding the use of GPUs.

In this paper we explore the use of remote GPU virtualization in order to provide CUDA acceleration to applications running inside KVM VMs. The aim of this study is to analyze which is the overhead that these applications experience when accessing GPUs outside their VM. For this study we make use of the rCUDA remote GPU virtualization framework because it was the only solution that was able to run the tested applications.

The rest of the paper is organized as follows. Section 2 thoroughly reviews previous efforts to provide GPU acceleration to applications being executed inside VMs and further motivates the use of general GPU virtualization frameworks

to provide acceleration features to VMs. Later, Sect. 3 introduces in more detail rCUDA, the remote GPU virtualization framework used in this study. Next, Sect. 4 addresses the main goal of this paper: studying the performance of real GPU-accelerated applications when executed within KVM VMs. Finally, Sect. 5 summarizes the main conclusions of our work.

2 Remote GPU Virtualization Solutions

Providing acceleration services to VMs is, basically, the same problem as sharing a GPU among the VMs concurrently running in the host computer or, in a more general perspective, sharing a GPU among several computers.

Sharing accelerators among several computers has been addressed both with hardware and software approaches. On the hardware side, maybe the most prominent solution was NextIO's N2800-ICA [2], based on PCIe virtualization [24]. This solution allowed to share a GPU among eight different servers in a rack within a two-meter distance. Nevertheless, this solution lacked the required flexibility because a GPU could only be used by a single server at a time, thus preventing the concurrent sharing of GPUs. Furthermore, this solution was expensive, what maybe was one of the reasons for NextIO going out of business in August 2013. GPU manufacturers have also tried to tackle the problem with virtualization enabled accelerators, but so far this GPUs are oriented towards the graphics acceleration usage, not compute. Nvidia GRID[1] only supports CUDA enabled VMs under Citrix XenServer and even in this case, the GPU resources are not shared between the VMs, each one has a reserved disjoint segment of the accelerator. The AMD FirePro S-series[2] solution is more flexible, but it only supports OpenCL and it is still oriented towards graphics.

A cheaper and more flexible solution for sharing accelerators, in the context of a server hosting several VMs, is PCI passthrough [35,38]. This mechanism is based on the use of the virtualization extensions widely available in current HPC servers, which allow to install several GPUs in a box and assign each of them, in an exclusive way, to one of the VMs running at the host. Furthermore, when making use of this mechanism, the performance attained by accelerators is very close to that obtained when using the GPU in a native domain. Unfortunately, as this approach assigns GPUs to VMs in an exclusive way, it does not allow to simultaneously share GPUs among the several VMs being concurrently executed at the same host. In order to address this concern, there have been several attempts, like the one proposed in [21], which dynamically changes on demand the GPUs assigned to VMs. However, these techniques present a high time overhead given that, in the best case, two seconds are required to change the assignment between GPUs and VMs.

As a flexible alternative to hardware approaches, several software-based GPU sharing mechanisms have appeared, such as V-GPU, dOpenCL, DS-CUDA, rCUDA, SnuCL, VOCL, VCL, vCUDA, and GridCuda, for example. Basically,

[1] http://www.nvidia.com/object/nvidia-grid.html.
[2] http://www.amd.com/en-us/solutions/professional/virtualization.

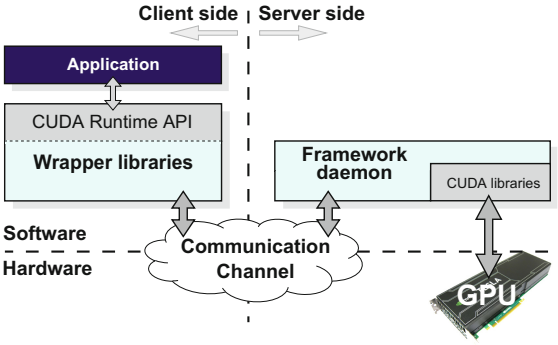


Fig. 1. Architecture usually deployed by GPU virtualization frameworks.

these software proposals share a GPU by virtualizing it, so that they provide applications (or VMs) with virtual instances of the real device, which can therefore be concurrently shared. Usually, these GPU sharing solutions place the virtualization boundary at the API level (Application Programming Interface), which can either be OpenCL [18] or CUDA [29] in the GPGPU field. Nevertheless, we will focus on CUDA-based solutions because CUDA is more widely used. In general, CUDA-based virtualization frameworks aim to offer the same API as the NVIDIA CUDA Runtime API [10] does.

Figure 1 depicts the architecture usually deployed by these virtualization solutions, which follow a distributed client-server approach. The client part of the middleware is installed in the computer (either native or virtual) executing the application requesting GPU services, whereas the server side runs in the native domain owning the actual GPU. Communication between client and server sides may be implemented by means of shared-memory mechanisms if both ends are located at the same physical computer or by using the network fabric if they are placed at different computers. The architecture depicted in Fig. 1 is used in the following way: the client middleware receives a CUDA request from the accelerated application and appropriately processes and forwards it to the server. There, the middleware receives the request and interprets and forwards it to the GPU, which completes the execution of the request and returns the execution results to the server middleware. Finally, the server sends back the results to the client middleware, which forwards them to the accelerated application. Notice that remote GPU virtualization frameworks provide GPU services in a transparent way and, therefore, applications are not aware that their requests are actually serviced by a remote GPU instead of by a local one.

CUDA-based GPU virtualization frameworks may be classified into two types: (1) those intended to be used in the context of VMs and (2) those devised as general purpose virtualization frameworks to be used in native domains, although the client part of these latter solutions may also be used within VMs. Frameworks in the first category usually make use of shared-memory mechanisms in order to transfer data from main memory inside the VM to the GPU in

the native domain, whereas the general purpose virtualization frameworks in the second type make use of the network fabric in the cluster to transfer data from main memory in the client side to the remote GPU located in the server. This is why these latter solutions are commonly known as remote GPU virtualization frameworks.

Regarding the first type of GPU virtualization frameworks mentioned above, several solutions have been developed to be specifically used within VMs, as for example vCUDA [33], GViM [19], gVirtuS [17], and Shadowfax [28]. The vCUDA technology supports only an old CUDA version (v3.2) and implements an unspecified subset of the CUDA Runtime API. Moreover, its communication protocol presents a considerable overhead, because of the cost of the encoding and decoding stages, which causes a noticeable drop in overall performance. GViM is based on the obsolete CUDA version 1.1 and, in principle, does not implement the entire CUDA Runtime API. gVirtuS is based on the old CUDA version 2.3 and implements only a small portion of its API. For example, in the case of the memory management module, it implements only 17 out of 37 functions. Furthermore, despite it being designed for KVM VMs, it requires a modified version of KVM. Nevertheless, although it is mainly intended to be used in VMs, granting them access to the real GPU located in the same node, it also provides TCP/IP communications for remote GPU virtualization, thus allowing applications in a non-virtualized environment to access GPUs located in other nodes. Regarding Shadowfax, this solution allows Xen VMs to access the GPUs located at the same node, although it may also be used to access GPUs at other nodes of the cluster. It supports the obsolete CUDA version 1.1 and, additionally, neither the source code nor the binaries are available in order to evaluate its performance.

In the second type of virtualization framework mentioned above, which provide general purpose GPU virtualization, one can find rCUDA [14,31], V-GPU [5], GridCuda [26], DS-CUDA [30], and Shadowfax II [4]. rCUDA, further described in Sect. 3, features CUDA 7.0 and provides specific communication support for TCP/IP compatible networks as well as for InfiniBand fabrics. V-GPU is a recent tool supporting CUDA 4.0. Unfortunately, the information provided by the V-GPU authors is unclear and there is no publicly available version that can be used for testing and comparison. GridCuda also offers access to remote GPUs in a cluster, but supporting an old CUDA version (v2.3). Although its authors mention that their proposal overcomes some of the limitations of the early versions of rCUDA, they later do not provide any insight about the supposedly enhanced features. Moreover, there is currently no publicly available version of GridCuda that can be used for testing. Regarding DS-CUDA, it integrates a more recent version of CUDA (4.1) and includes specific communication support for InfiniBand. However, DS-CUDA presents several strong limitations, such as not allowing data transfers with pinned memory. Finally, Shadowfax II is still under development, not presenting a stable version yet and its public information is not updated to reflect the current code status.

It is important to notice that although remote GPU virtualization has traditionally introduced a non-negligible overhead, given that applications do not

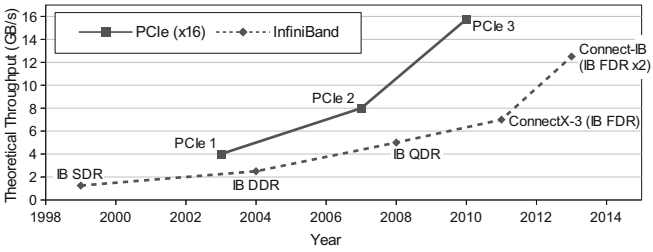


Fig. 2. Comparison between the theoretical bandwidth of different versions of PCI Express x16 and those of commercialized InfiniBand fabrics and network adapters.

access GPUs attached to the local PCI Express (PCIe) link but rather access devices that are installed in other nodes of the cluster (traversing a network fabric with a lower bandwidth), this performance overhead has significantly been reduced thanks to the recent advances in networking technologies. For example, as depicted in Fig. 2, the theoretical bandwidth of the InfiniBand network is 12.5 GB/s when using the Mellanox Connect-IB dual-port adapters [8]. This bandwidth is very close to the 15.75 GB/s of PCIe 3.0×16. This makes that the bandwidth achieved by InfiniBand Connect-IB network adapters and that of the NVIDIA Tesla K40 GPU are very close. Moreover, the previous generation of these technologies (NVIDIA Tesla K20 GPU and InfiniBand ConnectX-3 network adapter), provides performance figures that are also very close: the Tesla K20 GPU used PCIe 2.0, which achieves a theoretical bandwidth of 8 GB/s, whereas InfiniBand ConnectX-3 (which uses PCIe 3.0×8) provides 7 GB/s. As a result, when using remote GPU virtualization solutions in both hardware generations (Tesla K40 & Connect-IB and Tesla K20 & ConnectX-3), the path communicating the main memory in the computer executing the application and the remote accelerator presents a similar bandwidth in all of its stages. This bandwidth is very close to the one initially attained by the traditional approach using local GPUs, as shown in Fig. 3. These small differences in bandwidth cause that the initial non-negligible performance overhead of remote GPU virtualization solutions is now noticeably reduced, thus boosting the performance of GPU virtualizing frameworks. Furthermore, when remote GPU virtualization solutions are considered at the cluster level, it has been shown in [20] that they provide a noticeable reduction on the execution time of a given workload composed of a set of computing jobs. Moreover, important reductions in the total energy required to execute such workloads are also achieved [9]. All these aspects have definitively turned remote GPU virtualization frameworks into an appealing option, hence motivating the use of such frameworks in order to provide GPU services to applications being executed inside VMs. As mentioned before, we will make use of the rCUDA framework in our study because it was the sole solution being able to run the tested applications. In next section we present additional information on rCUDA relevant to the work presented in this paper.

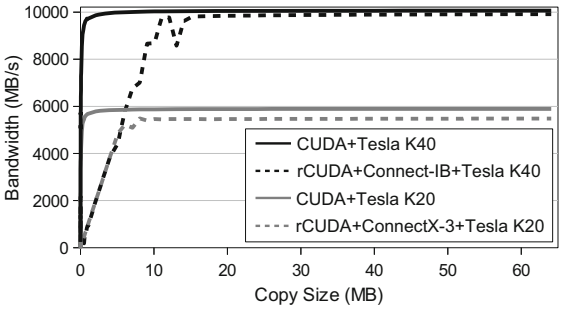


Fig. 3. Bandwidth test for copies between pinned host memory and GPU memory, using CUDA and a remote GPU virtualization framework (rCUDA) over InfiniBand employing different cards: ConnectX-3 single port and Connect-IB dual port. Both cards employ PCIe 3.0. The former makes use of 8 PCIe lanes whereas the latter uses 16 PCIe lanes. The CUDA accelerators used in the plot are an NVIDIA Tesla K20 (PCIe 2.0×16) and an NVIDIA Tesla K40 (PCIe 3.0×16). Notice that the Tesla K20 GPU and the ConnectX-3 network adapter went into the market at the same time, approximately. The same holds for the Tesla K40 GPU and the Connect-IB network adapter.

3 rCUDA: Remote CUDA

The rCUDA middleware supports version 7.0 of CUDA, being binary compatible with it, which means that CUDA programs using rCUDA do not need to be modified. Furthermore, it implements the entire CUDA Runtime API (except for graphics functions) and also provides support for the libraries included within CUDA, such as cuFFT, cuBLAS, or cuSPARSE. Furthermore, the rCUDA middleware allows a single rCUDA server to concurrently deal with several remote clients that simultaneously request GPU services. This is achieved by creating independent GPU contexts, each of them being assigned to a different client [31].

rCUDA additionally provides specific support for different interconnects [31]. Support for different underlying network fabrics is achieved by making use of a set of runtime-loadable, network-specific communication modules, which have been specifically implemented and tuned in order to obtain as much performance as possible from the underlying interconnect. Currently, two modules are available: one intended for TCP/IP compatible networks and another one specifically designed for InfiniBand.

Regarding the InfiniBand communications module, as explained by the rCUDA developers in [31], it is based on the InfiniBand Verbs (IBV) API. This API offers two communication mechanisms: the channel semantics and the memory semantics. The former refers to the standard send/receive operations typically available in any networking library, while the latter offers RDMA operations where the initiator of the operation specifies both the source and destination of a data transfer, resulting in zero-copy transfers with minimum involvement of the CPUs. rCUDA employs both IBV mechanisms, selecting one or the other depending on the exact communication to be carried out [31].

Moreover, independently from the exact network used, data exchange between rCUDA clients and remote GPUs located in rCUDA servers is pipelined so that higher bandwidth is achieved, as explained in [31]. Internal pipeline buffers within rCUDA use preallocated pinned memory given the higher throughput of this type of memory.

4 Impact of KVM Virtual Machines on Real Applications

In order to gather performance figures, we have used a testbed composed of two 1027GR-TRF Supermicro nodes running the CentOS 6.4 operating system. Each of the servers includes two Intel Xeon E5-2620 v2 processors (six cores with Ivy Bridge architecture) operating at 2.1 GHz and 32 GB of DDR3 SDRAM memory at 1600 MHz. They also own a Mellanox ConnectX-3 VPI single-port InfiniBand adapter[3], which uses a Mellanox Switch SX6025 (InfiniBand FDR compatible) to exchange data at a maximum rate of 56 Gb/s. The Mellanox OFED 2.4-1.0.4 (InfiniBand drivers and administrative tools) was used at both servers. Furthermore, the node executing the rCUDA server includes an NVIDIA Tesla K20 GPU (which makes use of a PCIe 2.0×16 link) with CUDA 7.0 and NVIDIA driver 340.46. On the other side, at the client node, the OFED has been configured in order to provide 2 virtual instances (virtual functions) of the InfiniBand adapter. One of these virtual functions will be used in the tests, for comparison purposes, by applications being executed at the native domain of the node whereas the other virtual function will be provided to a VM by using the PCI passthrough mechanism in order to assign it to the virtualized computer in an exclusive way. In addition to the use of virtual functions (virtual instances), we will also make use, for comparison purposes, of the original non-virtualized InfiniBand adapter, which will be referred to as physical function in the experiments. On the other hand, the VM has been created using qemu-kvm version 0.12.1.2, installed from the official CentOS repositories and has later been configured to have 16 cores and 16 GB of RAM memory.

Figure 4 graphically depicts the configurations to be used in the experiments presented in this section. First, the configuration where an accelerated application running in the native domain of the client machine communicates with the remote node using a non-virtualized InfiniBand adapter will be denoted in the performance plots as *PF-Rem* (*P*hysical *F*unction to *Rem*ote node). In a similar way, a configuration where the application is being executed in the native domain of the client node but makes use of the virtual copy (virtual function) of the InfiniBand adapter will be labeled as *VF-Rem* (*V*irtual *F*unction to *Rem*ote node). Finally, when the application is being executed inside the KVM VM, we will refer to this configuration as *VM-Rem* (*V*irtual *M*achine to *Rem*ote

[3] Notice that the new Mellanox InfiniBand Connect-IB network adapters are already available. These adapters feature a PCIe 3.0×16 connector, that, in addition to their dual-port configuration, provides an aggregated bandwidth larger than 12 GB/s. Nevertheless, the Mellanox driver is not ready yet to provide support for VMs. This is why in this work we use the previous ConnectX-3 adapters.

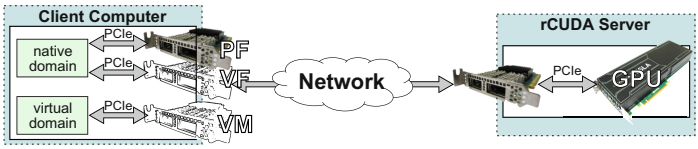


Fig. 4. Testbed used in the experiments presented in this paper. The client node, hosting the KVM VM, owns an InfiniBand ConnectX-3 network adapter, which has been virtualized. The server node also owns an InfiniBand ConnectX-3 network adapter. Notice that the tests in this paper will always use the real instance of this latter adapter, which has not been virtualized.

node). In all the three configurations, the remote node used a non-virtualized InfiniBand card.

The applications analyzed in this section are CUDASW++, GPU-BLAST and LAMMPS, all of them listed in the NVIDIA Popular GPU-Accelerated Applications Catalog [11].

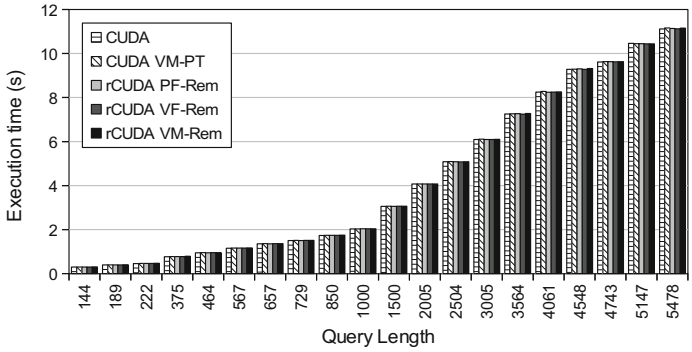
CUDASW++ [27] is a bioinformatics software for Smith-Waterman protein database searches that takes advantage of the massively parallel CUDA architecture of NVIDIA Tesla GPUs to perform sequence searches. In particular, we have used its latest release, version 3.0, for our study, along with the latest Swiss-Prot database and the example query sequences available in the application website[4].

GPU-BLAST [34] has been designed to accelerate the gapped and ungapped protein sequence alignment algorithms of the NCBI-BLAST[5] implementation using GPUs. It is integrated into the NCBI-BLAST code and produces identical results. We use release 1.1 in our experiments, where we have followed the installation instructions for sorting a database and creating a GPU database. To search the database, we then use the query sequences that come with the application package.

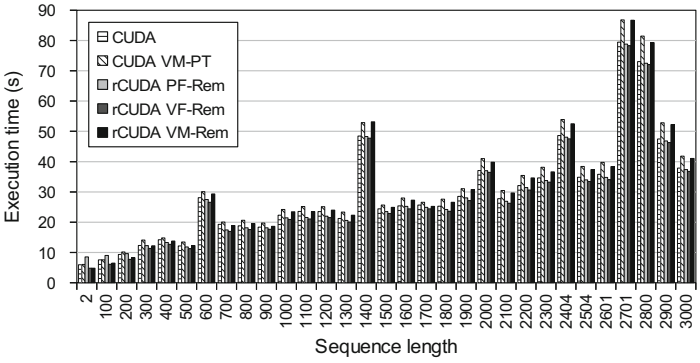
LAMMPS [25] is a classic molecular dynamics simulator that can be used to model atoms or, more generically, as a parallel particle simulator at the atomic, mesoscopic, or continuum scale. For the tests below, we use the release from Feb. 1, 2014, and benchmarks `in.eam` and `in.lj` installed with the application. We run the benchmarks with one processor, scaling by a factor of 5 in all three dimensions (i.e., a problem size of 4 million atoms).

Figure 5 shows the execution times for these three applications when run in the four different scenarios under analysis: execution with CUDA with a local GPU in a native domain (label *CUDA*) and with rCUDA in the three remote scenarios (labels *rCUDA PF-Rem*, *rCUDA VF-Rem*, and *rCUDA VM-Rem*). Additionally, the performance of these applications when executed within the VM, using the GPU of the host by leveraging the PCI passthrough mechanism, is also analyzed (label *CUDA VM-PT*). Every experiment has been performed ten times, so that the figures show the averaged results. For the experiments involving executions in the native domain, the VM was shut off in order to avoid interferences.

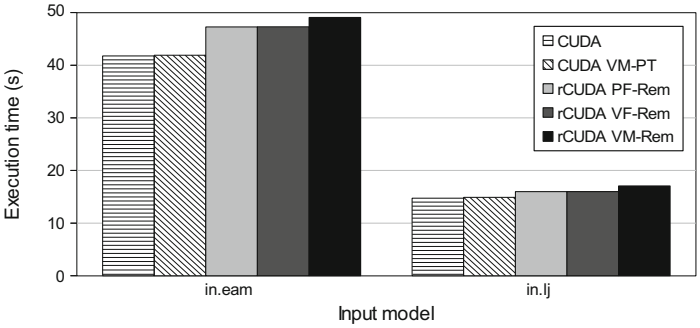
[4] http://cudasw.sourceforge.net.
[5] http://www.ncbi.nlm.nih.gov.



(a) Execution time of CUDASW++ application.



(b) Execution time of GPU-BLAST application.



(c) Execution time of LAMMPS application.

Fig. 5. Execution time with respect to CUDA of the several applications when executed in different local and remote scenarios.

Regarding CUDASW++, Fig. 5(a) shows that execution times of this application in the different scenarios considered are very similar. As expected, executing the application within the VM using a remote GPU (*rCUDA VM-Rem*) introduces the larger overhead with respect to CUDA, 0.804 % on average. When the PCI passthrough mechanism is used from inside the VM (*CUDA VM-PT*),

the application experiences an overhead (0.475 % on average) comparable to the one achieved when executing the application in the native domain using the remote GPU (*rCUDA PF-Rem* and *rCUDA VF-Rem*, 0.323 % and 0.358 % on average, respectively).

Next, Fig. 5(b) presents results for the GPU-BLAST application. It can be seen that this application presents several peaks at sequence lengths equal to 600, 1400, 2000, 2404, 2701, and 2800. Notice that rCUDA mimics the behavior of CUDA for all of these peaks in all the scenarios under study, therefore, analyzing the reasons of this behavior is out of the scope of this paper. Regarding the overhead, in this case we can see an almost constant overhead of about 10 % when VMs are used (*CUDA VM-PT* and *rCUDA VM-Rem*), being the later a little inferior. Surprisingly, in the case of executions using the rCUDA framework from the native domain, the application is slightly accelerated, 0.451 % on average. A deeper analysis of these experiments shows that KVM is introducing some overhead not related with the use of rCUDA or the InfiniBand network. In this way, the application presents periods of time in which the GPU is not used and the overhead is introduced by KVM in tasks not involving the GPU, such as CPU computations and I/O operations.

Finally, Fig. 5(c) shows results for LAMMPS. It can be seen that the lower execution time is achieved in the native domain, as expected. Furthermore, executing the application from the inside of the VM using the GPU in the host (*CUDA VM-PT*) introduces a negligible overhead of 0.3 % and 1.1 % for the in.eam and in.lj input models, respectively. The use of the rCUDA framework, however, introduces a larger overhead. When used from the native domain (*rCUDA PF-Rem* and *rCUDA VF-Rem*) execution time increases, respectively, up to 13 % and 8 % for the in.eam and in.lj benchmarks. Lastly, executing this application inside the VM increases the execution time up to 17 %. The reason for the larger overhead shown in these experiments is that here, unlike in previous applications, the total amount of data transferred to/from the GPU is significantly higher (see Table 1), thus meaning a higher use of the network fabric, which translates in more overhead when using rCUDA.

Table 1. Summary of rCUDA overhead when applications are being executed inside the KVM VM (*rCUDA VM-Rem*), related with data transferred to/from the GPUs, for the applications under analysis.

Application	rCUDA VM-Rem overhead	Total amount of data transfers to/from GPU
CUDASW++	0.804 %	0.20 GB
GPU-BLAST	2.835 %	1.76 GB
LAMMPS	16.68 %	5.00 GB

As a summary, in Table 1 we present rCUDA overhead when applications are being executed inside the KVM VM (labeled as *rCUDA VM-Rem* in previous

figures). As we can observe, the overhead is directly related with total amount of data transferred to/from the GPUs. Thus, we can conclude that the more data transfers, the more overhead.

5 Conclusions

In this paper we have analyzed the use of the remote GPU virtualization mechanism in order to provide acceleration services to scientific applications running inside KVM virtual machines in InfiniBand clusters. This approach overcomes the issues of other methods, such as the PCI passthrough mechanism, allowing the concurrent sharing of GPUs by multiple virtual machines.

The main conclusion from the paper is that remote GPU virtualization frameworks could be a feasible option to provide acceleration services to KVM virtual machines. In this manner, our experiments have shown that the performance experienced by GPU-accelerated applications, running inside a virtual machine and accessing a remote GPU, mainly depends on data transferred to/from the remote GPU: the more data the application transfers to the remote GPU, the more overhead it will show with respect to using a local GPU.

Acknowledgment. This work was funded by Generalitat Valenciana under Grant PROMETEOII/2013/009 of the PROMETEO program phase II. The authors are grateful for the generous support provided by Mellanox Technologies and the equipment donated by NVIDIA Corporation.

References

1. Kernel-based Virtual Machine. http://www.linux-kvm.org. Accessed: Jan 2016
2. NextIO, N2800-ICA — Flexible and manageable I/O expansion and virtualization. http://www.nextio.com/. Accessed: Mar 2012
3. Oracle VM VirtualBox. http://www.virtualbox.org/. Accessed: Jan 2016
4. Shadowfax II - scalable implementation of GPGPU assemblies. http://keeneland.gatech.edu/software/keeneland/kidron. Accessed: Jan 2016
5. V-GPU: GPU virtualization. http://www.zillians.com/products/vgpu-gpu-virtualization/. Accessed: Jan 2016
6. VMware virtualization. http://www.vmware.com/. Accessed: Jan 2016
7. Xen Project. http://www.xenproject.org/. Accessed: Jan 2016
8. Mellanox, Connect-IB Single and Dual QSFP+ Port PCI Express Gen3x16 Adapter Card User Manual (2013). http://www.mellanox.com/related-docs/user_manuals/Connect-IB_Single_and_Dual_QSFP+_Port_PCI_Express_Gen3_x16_Adapter_Card_User_Manual.pdf
9. rCUDA: Virtualizing GPUs to reduce cost and improve performance (2014). http://www.rcuda.net
10. CUDA API Reference Manual 7.0 (2015). https://developer.nvidia.com/cuda-toolkit
11. NVIDIA Popular GPU-Accelerated Applications Catalog (2015). http://www.nvidia.es/content/tesla/pdf/gpu-accelerated-applications-for-hpc.pdf

12. Barak, A., Ben-Nun, T., Levy, E., Shiloh, A.: A package for OpenCL based heterogeneous computing on clusters with many GPU devices. In: 2010 IEEE International Conference on Cluster Computing Workshops and Posters (CLUSTER WORKSHOPS), pp. 1–7. IEEE (2010)
13. Barrachina, S., Castillo, M., Igual, F.D., Mayo, R., Quintana-Ortí, E.S., Quintana-Ortí, G.: Exploiting the capabilities of modern GPUs for dense matrix computations. Concurrency Comput.: Pract. Experience **21**(18), 2457–2477 (2009)
14. Duato, José, Igual, Francisco D., Mayo, Rafael, Peña, Antonio J., Quintana-Ortí, Enrique S., Silla, Federico: An efficient implementation of GPU virtualization in high performance clusters. In: Lin, Hai-Xiang, Alexander, Michael, Forsell, Martti, Knüpfer, Andreas, Prodan, Radu, Sousa, Leonel, Streit, Achim (eds.) Euro-Par 2009. LNCS, vol. 6043, pp. 385–394. Springer, Heidelberg (2010)
15. Felter, W.: An updated performance comparison of virtual machines and linux containers. IBM Research Report (2014)
16. Gaikwad, A., Toke, I.M.: GPU based sparse grid technique for solving multidimensional options pricing PDEs. In: Proceedings of the 2nd Workshop on High Performance Computational Finance, WHPCF 2009, pp. 6: 1–6: 9. ACM, New York (2009)
17. Giunta, G., Montella, R., Agrillo, G., Coviello, G.: A GPGPU transparent virtualization component for high performance computing clouds. In: D'Ambra, P., Guarracino, M., Talia, D. (eds.) Euro-Par 2010, Part I. LNCS, vol. 6271, pp. 379–391. Springer, Heidelberg (2010)
18. Group, K.O.W: OpenCL 1.2 Specification (2011)
19. Gupta, V., Gavrilovska, A., Schwan, K., Kharche, H., Tolia, N., Talwar, V., Ranganathan, P.: GViM: GPU-accelerated virtual machines. In: Proceedings of the 3rd ACM Workshop on System-level Virtualization for High Performance Computing, pp. 17–24. ACM (2009)
20. Iserte, S., Gimeno, A.C., Mayo, R., Quintana-Ortí, E.S., Silla, F., Duato, J., Reaño, C., Prades, J.: SLURM support for remote GPU virtualization: Implementation and performance study. In: 26th IEEE International Symposium on Computer Architecture and High Performance Computing, SBAC-PAD 2014, Paris, France, 22–24 October, pp. 318–325 (2014)
21. Jo, H., Jeong, J., Lee, M., Choi, D.H.: Exploiting GPUs in virtual machine for BioCloud. BioMed Res. Int. **2013**, 1–11 (2013)
22. Kegel, P., Steuwer, M., Gorlatch, S.: dopencl: Towards a uniform programming approach for distributed heterogeneous multi-many-core systems. In: 2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops PhD Forum (IPDPSW), pp. 174–186, May 2012
23. Kim, J., Seo, S., Lee, J., Nah, J., Jo, G., Lee, J.: SnuCL: An OpenCL framework for heterogeneous CPU/GPU clusters. In: Proceedings of the 26th ACM International Conference on Supercomputing, ICS 2012, pp. 341–352. ACM, New York (2012)
24. Krishnan, V.: Towards an integrated IO and clustering solution using PCI express. In: 2007 IEEE International Conference on Cluster Computing, pp. 259–266. IEEE (2007)
25. Laboratories, S.N.: LAMMPS Molecular Dynamics Simulator (2013). http://lammps.sandia.gov/
26. Liang, T.Y., Chang, Y.W.: GridCuda: a grid-enabled CUDA programming toolkit. In: 2011 IEEE Workshops of International Conference on Advanced Information Networking and Applications (WAINA), pp. 141–146. IEEE (2011)

27. Liu, Y., Wirawan, A., Schmidt, B.: CUDASW++ 3.0: accelerating Smith-Waterman protein database search by coupling CPU and GPU SIMD instructions. BMC Bioinform. **14**(1), 1–10 (2013)
28. Merritt, A.M., Gupta, V., Verma, A., Gavrilovska, A., Schwan, K.: Shadowfax: scaling in heterogeneous cluster systems via GPGPU assemblies. In: Proceedings of the 5th International Workshop on Virtualization Technologies in Distributed Computing, VTDC 2011, pp. 3–10. ACM, New York (2011)
29. NVIDIA: CUDA C Programming Guide 7.0 (2015)
30. Oikawa, M., Kawai, A., Nomura, K., Yasuoka, K., Yoshikawa, K., Narumi, T.: DS-CUDA: a middleware to use many GPUs in the cloud environment. In: Proceedings of the 2012 SC Companion: High Performance Computing, Networking Storage and Analysis, SCC 2012, pp. 1207–1214. IEEE Computer Society, Washington, DC (2012)
31. Peña, A.J., Reaño, C., Silla, F., Mayo, R., Quintana-Ortí, E.S., Duato, J.: A complete and efficient CUDA-sharing solution for HPC clusters. Parallel Comput. **40**(10), 574–588 (2014)
32. Playne, D.P., Hawick, K.A.: Data parallel three-dimensional cahn-hilliard field equation simulation on GPUs with CUDA. In: PDPTA, pp. 104–110 (2009)
33. Shi, L., Chen, H., Sun, J.: vCUDA: GPU accelerated high performance computing in virtual machines. In: IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2009, pp. 1–11. IEEE (2009)
34. Vouzis, P.D., Sahinidis, N.V.: Gpu-blast: Using graphics processors to accelerate protein sequence alignment. Bioinformatics **27**(2), 182–188 (2010)
35. Walters, J.P., Younge, A.J., Kang, D.I., Yao, K.T., Kang, M., Crago, S.P., Fox, G.C.: GPU-Passthrough performance: a comparison of KVM, Xen, VMWare ESXi, and LXC for CUDA and OpenCL applications. In: 7th IEEE International Conference on Cloud Computing (CLOUD 2014) (2014)
36. Wu, H., Diamos, G., Sheard, T., Aref, M., Baxter, S., Garland, M., Yalamanchili, S.: Red fox: an execution environment for relational query processing on GPUs. In: Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2014, pp. 44: 44–44: 54. ACM, New York (2014)
37. Xiao, S., Balaji, P., Zhu, Q., Thakur, R., Coghlan, S., Lin, H.,Wen, G., Hong, J., Chun Feng, W.: Vocl: An optimized environment for transparentvirtualization of graphics processing units. In: Proceedings of the 1st Innovative Parallel Computing (InPar) (2012)
38. Yang, C.T., Wang, H.Y., Ou, W.S., Liu, Y.T., Hsu, C.H.: On implementation of GPU virtualization using PCI pass-through. In: 2012 IEEE 4th International Conference on Cloud Computing Technology and Science (CloudCom), pp. 711–716. IEEE (2012)