Document downloaded from:

http://hdl.handle.net/10251/155249

This paper must be cited as:

Prades, J.; Reaño González, C.; Silla Jiménez, F. (2019). On the Effect of using rCUDA to Provide CUDA Acceleration to Xen Virtual Machines. Cluster Computing. 22(1):185-204. https://doi.org/10.1007/s10586-018-2845-0

The final publication is available at

https://doi.org/10.1007/s10586-018-2845-0

Copyright Springer-Verlag

Additional Information

Noname manuscript No.
(will be inserted by the editor)

On the Effect of using rCUDA to Provide CUDA Acceleration to Xen Virtual Machines

Javier Prades · Carlos Reaño · Federico Silla

Received: date / Accepted: date

Abstract Nowadays, many data centers use virtual machines (VMs) in order to achieve a more efficient use of hardware resources. The use of VMs provides a reduction in equipment and maintenance expenses as well as a lower electricity consumption. Nevertheless, current virtualization solutions, such as Xen, do not easily provide graphics processing units (GPUs) to applications running in the virtualized domain with the flexibility usually required in data centers (i.e., managing virtual GPU instances and concurrently sharing them among several VMs). Therefore, the execution of GPU-accelerated applications within VMs is hindered by this lack of flexibility. In this regard, remote GPU virtualization solutions may address this concern.

In this paper we analyze the use of the remote GPU virtualization mechanism to accelerate scientific applications running inside Xen VMs. We conduct our study with six different applications, namely CUDA-MEME, CUDASW++, GPU-BLAST, LAMMPS, a triangle count application, referred to as TRICO, and a synthetic benchmark used to emulate different application behaviors. Our experiments show that the use of remote GPU virtualization is a feasible approach to address the current concerns of sharing GPUs among several VMs, featuring a very low overhead if an InfiniBand fabric is already present in the cluster.

Keywords Virtualization · CUDA · Xen · InfiniBand · HPC · Performance

J. Prades, C. Reaño, F. Silla
Universitat Politecnica de Valencia
Valencia, Spain
E-mail: japraga@gap.upv.es, E-mail: carregon@gap.upv.es,
E-mail: fsilla@disca.upv.es

1 Introduction

Virtual machines (VMs) have demonstrated to provide economic savings to data centers, the main reason being that several VMs can be concurrently executed in a single cluster node thus sharing its CPUs as well as other subsystems and, therefore, increasing overall resource utilization. Acquisition and maintenance costs are therefore reduced because a smaller amount of servers is required to address the same workload, thus reducing also energy consumption needs. In this way, the use of VMs is the basis for cloud computing services like the ones provided by Amazon and other PaaS (Platform as a Service) providers.

The benefits provided by VMs have caused that virtualization solutions such as KVM [3], Xen [9], VMware [8], or VirtualBox [5] become very popular. Actually, the benefits reported by the use of VMs have motivated that leading processor manufacturers such as Intel or AMD have increasingly incorporated more support for virtualization into their chip designs [38]. Moreover, although VMs were known in the past for reducing 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 [11]. This has led some authors to suggest using VMs in the context of high-performance computing (HPC) [48].

However, despite the many advances accomplished in the field of VMs, they still do not support efficiently the current trend of using the CUDA[1] compute plat-

[1] CUDA (Compute Unified Device Architecture) is a technology created by NVIDIA which comprises a parallel compute platform (CUDA-enabled graphics processing units) as well as an application programming interface (API) and a compiler.

form to use the graphics processing units (GPUs) as accelerators, which allows significantly reducing the time required to execute applications from areas as different as data analysis (Big Data) [44], chemical physics [31], computational algebra [45], image analysis [47], finance [41], biology [10], and artificial intelligence [20], to name just a few. In this regard, there have been several recent achievements in order to virtualize GPUs, like the new GRID K1 GPU by NVIDIA [4], which can be shared among up to eight VMs and is mainly intended for desktop virtualization, although it can also be used for executing CUDA programs. Nevertheless, given that this device only features 192 CUDA cores per GPU, its applicability to scientific computing is very limited[2]. Other examples of including virtualization support into GPUs are the recent KVMGT[3] technology by Intel [40] and the new Multiuser GPU by AMD [1], which provide virtualization support for Intel and AMD GPUs, respectively. Unfortunately, these solutions do not support CUDA acceleration. Therefore, the lack of efficient support for CUDA-compatible GPUs in current virtualization solutions makes that applications running in the virtualized domain cannot easily access these GPUs for acceleration purposes. From the point of view of cloud computing providers such as Amazon, this means that the investment made in GPUs cannot be amortized as fast as possible as it will be further described in next section.

In this paper we explore the use of the remote GPU virtualization mechanism in order to provide CUDA acceleration to applications running inside Xen VMs. The main motivation is that GPU virtualization solutions such as V-GPU [7], DS-CUDA [29], rCUDA [37, 36], vCUDA [39], GridCuda [16], GVirtuS [12], GViM [13], Shadowfax [24], or Shadowfax II [6] may be used in VM environments in order to address their current limitations with respect to GPUs. These GPU virtualization frameworks detach GPUs from nodes, thereby allowing applications to access virtualized GPUs regardless of the exact computer where they are being executed. Thus, the detaching features of remote GPU virtualization solutions may turn them into an easy and efficient way to overcome the current limitations of VMs regarding the use of GPUs as accelerators.

The aim of this study is to assess the overhead that applications experience when accessing GPUs outside their Xen VM by using the remote GPU virtualization approach. To that end, we investigate two different scenarios. In the first one, an application within a VM accesses a GPU located in the same computer hosting it. In the second scenario we assume that a high performance network fabric such as InfiniBand (IB) is available in the cluster and the application running inside the VM accesses a GPU located in another cluster node. In this study we use the rCUDA remote GPU virtualization middleware because it was the sole solution able to run the applications considered in our analysis. As can be seen, the main contribution of this paper is testing the performance of the rCUDA remote GPU virtualization middleware in the context of Xen VMs.

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 CUDA acceleration to VMs. Later, Section 3 introduces rCUDA in more detail whereas Section 4 presents the experimental setup used in this paper. Section 5 analyzes the network performance observed by Xen VMs when making use of the virtual network connecting VMs within a host as well as the performance of the InfiniBand interconnect. Section 6 uses these results to analyze the performance of rCUDA when used from the inside of Xen VMs. Next, Section 7 addresses the main goal of this paper: studying the throughput of real GPU-accelerated applications when executed within Xen VMs. A synthetic benchmark is also leveraged in order to emulate several interesting features of application behaviour. Finally, Section 8 summarizes the main conclusions of our work.

2 Providing CUDA GPUs to Virtual Machines

Providing CUDA acceleration to VMs can be accomplished by making use of the PCI passthrough technique [43, 46]. This mechanism is based on the use of the virtualization extensions widely available in current HPC servers, which allow assigning a GPU, 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, only one of the VMs can access the GPU. This means, for instance, that for the CG1 VM instances of Amazon, which make use of the M2050

[2] In addition to the GRID K1 GPU, NVIDIA has also brought to market the GRID K2 model, which features 1536 CUDA cores per GPU and 4 GB of memory. However, this amount of resources per GPU is still noticeable smaller than the ones available in current NVIDIA Tesla K20 and K40 GPUs, featuring, respectively, 2496 and 2880 CUDA cores and 5GB and 12GB of memory. Therefore, using the GRID K2 device for providing acceleration to scientific applications instead of providing desktop virtualization would deliver a significantly lower performance than current mainstream GPUs used in HPC servers, such as the K20 or K40 GPUs.

[3] KVMGT is the open source implementation of Intel's GPU Virtualization Technology for KVM VMs.

NVIDIA GPU, only one of the VMs being executed in a given host can be assigned the GPU at the same time. This exclusive assignment of the GPU to a single VM causes an underutilization of resources because the computation capabilities of the GPU cannot be leveraged by other VMs when the VM that owns the GPU does not use it. Furthermore, this exclusive assignment means that the amount of CG1 VM instances that can be concurrently in execution in a given node is limited by the amount of GPUs installed in that node. This limits the economic profit that a data center can obtain from the underlying hardware, causing that the initial investment requires more time to be amortized.

To address the concern about the exclusive assignment of GPUs to VMs, there have been several attempts, like the one proposed in [14], which dynamically changes on demand the GPUs assigned to VMs. However, these techniques present two important concerns: (1) a high time overhead is generated given that, in the best case, two seconds are required to change the assignment between GPUs and VMs; (2) These techniques do not address the impossibility of sharing GPUs among several VMs simultaneously.

For these reasons, several software-based GPU sharing mechanisms have appeared, such as, for example, V-GPU, DS-CUDA, rCUDA, vCUDA, and GridCuda. Basically, these middleware proposals share a GPU by virtualizing it, so that these middleware systems 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[4] (CUDA [26] in the case of NVIDIA GPUs). In general, CUDA-based virtualization frameworks aim to offer the same API as the NVIDIA CUDA Runtime API [27] does.

Figure 1 depicts the architecture usually deployed by these GPU virtualization solutions, which follow a distributed client-server approach. The client part of the middleware is installed in the domain (either native or virtual)[5] executing the application requesting GPU services, whereas the server side runs in the domain owning the actual GPU. Communication between client and server may be based on shared-memory mechanisms or on the use of a network fabric, depending on the exact features of the GPU virtualization middleware and the underlying system configuration.

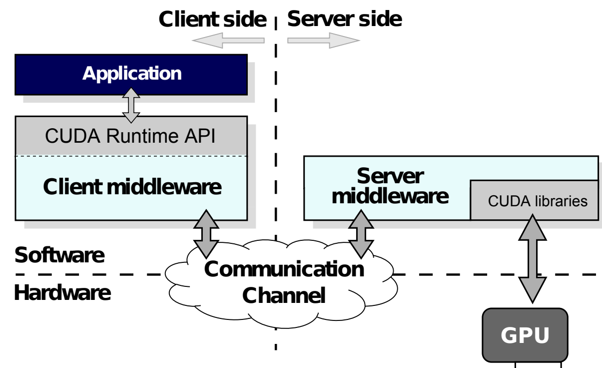


Fig. 1 Typical architecture used by GPU virtualization solutions.

The architecture depicted in Figure 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 middleware. In the server side, 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 GPU virtualization solutions provide GPU services in a transparent way and, therefore, applications are not aware that their requests are actually serviced by a virtual GPU instead of by a local one. The following piece of code shows an example of a CUDA program that we will use to further explain how the architecture in Figure 1 works.

```
1  #include <cuda.h>
2  #include <stdio.h>
3  const int N = 8;

5  // Function that will be executed in the GPU
6  __global__ void my_gpu_function(int *a, int *b)
7  {
8    b[threadIdx.x] = a[threadIdx.x] * 2;
9  }

11 int main()
12 {
13   int a[N] = {0, 1, 2, 3, 4, 5, 6, 7};
14   int *ad, *bd;
15   const int isize = N*sizeof(int);

17   // Perform some computations in the CPU
18   CPU code 1
19   CPU code 2
20   ...


23   // Allocate GPU memory
24   cudaMalloc( (void**)&ad, isize );
25   cudaMalloc( (void**)&bd, isize );

27   // Copy data to GPU memory
28   cudaMemcpy( ad, a, isize,
                 cudaMemcpyHostToDevice );
```

[4] In order to interact with the virtualized GPU, some kind of interface is required so that the application can access the virtual device. This interface could be placed at different levels. For instance, it could be placed at the driver level. However, GPU drivers usually employ low-level protocols which, additionally, are proprietary and strictly closed by GPU vendors. Therefore, a higher-level boundary must be used. This is why the GPU API is commonly selected for placing the virtualization boundary, given that these APIs are public.

[5] The native domain refers to a scenario where virtualization is not used, that is, a real computer is leveraged. On the other hand, the virtual domain refers to the virtual machine.

```
30    // Run function in the GPU
31    my_gpu_function<<<1, N>>>(ad, bd);

33    // Copy results from GPU memory
34    cudaMemcpy( b, bd, isize,
                  cudaMemcpyDeviceToHost );

36    // Free GPU memory
37    cudaFree( ad );
38    cudaFree( bd );

40    return 0;
41 }
```

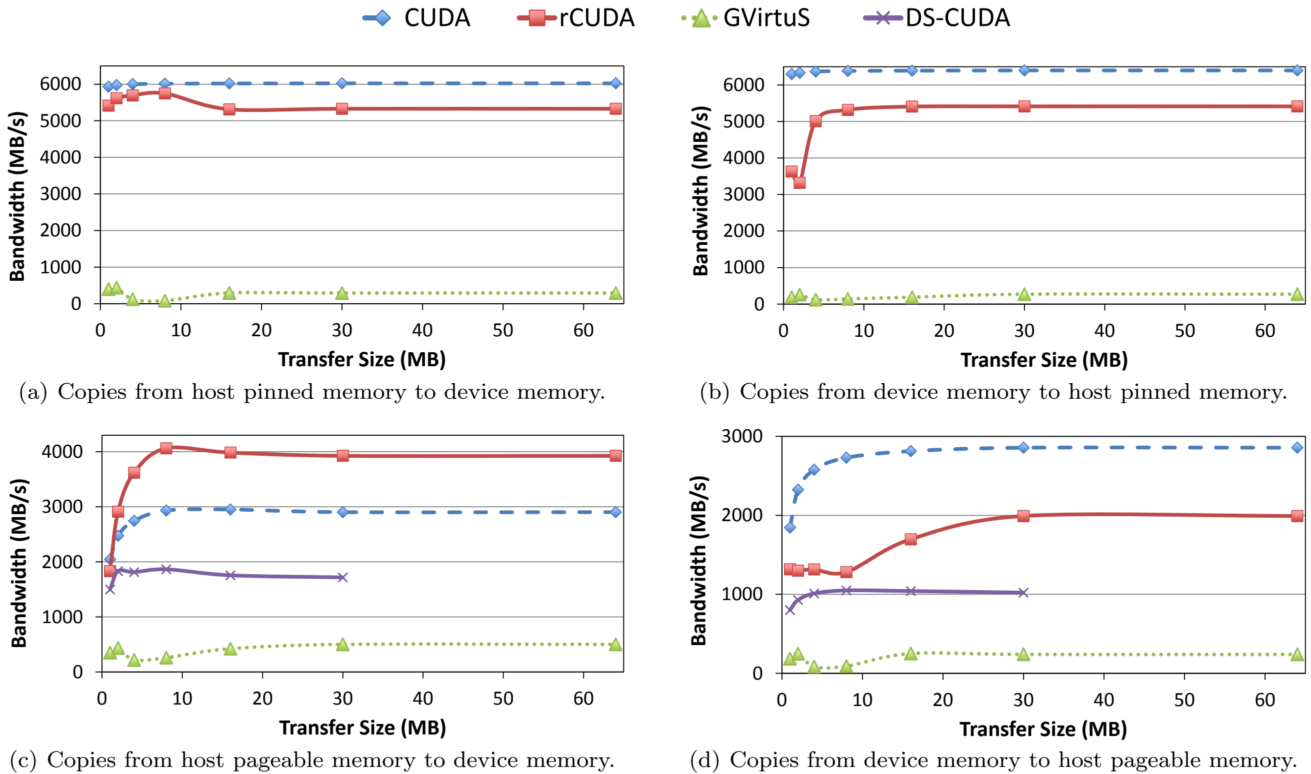
When the previous program is executed, not using a GPU virtualization framework, the CUDA library is loaded. However, when the program is executed to make use of a virtual GPU, the original CUDA library by NVIDIA is not loaded but another library with the same name is loaded. This other library contains a set of wrappers to the original CUDA functions that take care of the virtualization process. In this way, all the CPU code is executed in the same way as before but as soon as a call to a CUDA function is performed, the appropriate wrapper in the second library is called. For example, when the cudaMalloc function in line 24 is called, the wrapper for that function receives the arguments and forwards them to the middleware server, along with the function code assigned to the cudaMalloc function. Once the function code and the arguments arrive at the middleware server, the actual cudaMalloc function is executed in the real GPU by making use of the received arguments and the result of the call (status code) is collected. This status code is sent back to the client middleware, which is waiting for it. Upon reception of the status code, the client middleware delivers it to the application which continues with the execution of the program. The rest of CUDA calls shown in the example code in lines 25, 28, 31, 34, 37, and 38 are processed in a similar way.

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 solutions, to be used in native domains (notice that these latter solutions may also be used within VMs). Middleware systems 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 solutions 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 middleware systems.

Regarding the first type of GPU virtualization solutions mentioned above, several frameworks have been developed, as for example vCUDA, GViM, gVirtuS, and Shadowfax. The vCUDA technology, intended for Xen VMs, only supports the old CUDA version 3.2 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, targeting Xen VMs, 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 6.5 and implements only a small portion of its API. Despite being designed for VMs, 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. Notice that among the virtualization frameworks decribed in this group, only the gVirtuS solution is publicly available.

In the second type of virtualization solutions mentioned above, which provide general purpose GPU virtualization, one can find rCUDA, V-GPU, GridCuda, DS-CUDA, and Shadowfax II. rCUDA, further described in Section 3, features CUDA 8.0 and provides specific communication support for TCP/IP compatible networks as well as for InfiniBand and RoCE fabrics. V-GPU is a recent tool supporting CUDA 4.0. Unfortunately, the information provided by the V-GPU authors is fuzzy 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 supports the old CUDA version 2.3. Moreover, there is currently no publicly available version of GridCuda that can be used for testing. Regarding DS-CUDA, it integrates an old 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. Among these remote GPU virtualization solutions, only the DS-CUDA and rCUDA frameworks are publicly available.

In order to provide a comprehensive comparison among the different GPU virtualization solutions described in this section, Figure 2 presents a performance comparison of the three publicly available GPU virtualization solutions: DS-CUDA, rCUDA, and gVirtuS. This figure also shows the performance of CUDA as the baseline reference. The widely used `bandwidthTest` benchmark



(a) Copies from host pinned memory to device memory.

(b) Copies from device memory to host pinned memory.

(c) Copies from host pageable memory to device memory.

(d) Copies from device memory to host pageable memory.

Fig. 2 Performance comparison among three different GPU virtualization solutions: gVirtuS, DS-CUDA, and rCUDA. The comparison is performed in terms of attained bandwidth. The performance of CUDA is also depicted. Tests have been carried out in native domains with the hardware and software settings described in Section 4.

from the NVIDIA CUDA Samples [25] has been employed. The reason for using bandwidth for measuring performance is that, when transferring data between main memory and GPU memory, data copy sizes are, in general, large (in the order of MB) as it will be shown in Section 7. These large data transfers are mostly influenced by attained bandwidth, which turns out to be the most limiting factor regarding the performance of these solutions. Consequently, other metrics such as latency are less relevant in this context.

The testbed employed for carrying out the performance experiments is the one described in Section 4, although no virtual machine has been used in order to simplify the experiments. In this way, the bandwidth test was run in a native domain whereas the server side of the middleware systems was executed in a remote computer. The InfiniBand FDR network technology was used to connect both computers. Therefore, both the rCUDA and DS-CUDA middleware systems made use of the InfiniBand Verbs API. In the case of gVirtuS, given that it is not able to take advantage of the InfiniBand Verbs API, TCP/IP over InfiniBand was used.

One additional consideration to be made regarding the experiments shown in Figure 2 is that the three GPU virtualization middleware systems analyzed support different versions of CUDA. Thus, each of the frameworks has been analyzed with the respective version of CUDA supported. In this regard, it is important to remark that, in order to avoid introducing additional noise in this particular test, we have previously compared the bandwidth achieved by the three versions of CUDA used and the result is that differences in performance for the bandwidth test are negligible from one CUDA version to another.

Results in Figure 2 deserve some discussion. First, it can be seen that CUDA achieves the highest performance when pinned memory is used (Figures 2(a) and 2(b)), attaining a bandwidth around 6000 MB/s. Notice that this bandwidth is reduced for copies using pageable memory (Figures 2(c) and 2(d)). Second, Figure 2 shows that rCUDA outperforms the other two remote GPU virtualization solutions. Actually, for copies from host to device memory using pageable memory rCUDA also performs better than CUDA. This is a well-known effect thoroughly described in previous works on rCUDA [36] and is due to the use of an efficient

pipelined communication based on the use of internal pre-allocated pinned memory buffers. On the other hand, notice that both rCUDA and DS-CUDA make use of the InfiniBand Verbs API, thus having access to the large bandwidth available in this interconnect. However, although rCUDA is able to struggle an important fraction of the available bandwidth, DS-CUDA presents a relatively poor performance. Therefore, it must be assumed that the difference in bandwidth is due to the different way that both GPU virtualization solutions manage the InfiniBand interconnect. Also notice that DS-CUDA supports neither memory copies larger than 32MB nor the use of pinned memory. Futhermore, notice that the performance of gVirtuS is extremely low. One may think that this is due to the fact that gVirtuS is using TCP/IP over InfiniBand, which clearly achieves lower performance than the InfiniBand Verbs API. However, according to our measurements with the iperf tool [2], when TCP/IP is used over InfiniBand FDR, a bandwidth around 1190 MB/s is achieved, which is a noticeably larger bandwidth than the one attained by gVirtuS. Hence, the low performance of this middleware is not only due to the use of TCP/IP over InfiniBand but also to the way it internally manages communications.

As a final consideration for this review section, it is important to remark that although remote GPU virtualization has traditionally introduced a non-negligible overhead, given that applications do not 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, the rCUDA middleware is able to achieve 98% [34] of the native bandwidth of the Tesla K40 GPU when making use of FDR dual-port network adapters (providing 12.5GB/s of effective bandwidth) [22]. In the case of using the previous generation of these technologies, NVIDIA Tesla K20 GPUs and InfiniBand FDR single-port network adapters (6GB/s of effective bandwidth) [21], Figure 2 shows that bandwidth attained by rCUDA is very close to that of CUDA, except for copies from device to host memory using pageable memory, which still need some refinement. Therefore, when using remote GPU virtualization solutions, the path communicating main memory in the computer executing the application and the remote accelerator presents, in general, a bandwidth similar to that initially attained by the original CUDA approach of using local GPUs.

3 rCUDA: Remote CUDA

As already mentioned in the introduction section, we use in this study the rCUDA middleware given that it was the only one able to run the applications analyzed in this paper, as well as being the most up-to-date solution, providing also the best performance among the different publicly available GPU virtualization solutions. In this section we introduce rCUDA in more detail.

The rCUDA middleware supports version 8.0 of CUDA, being binary compatible with it, which means that CUDA programs do not need to be modified for using rCUDA. Furthermore, it implements the entire CUDA Runtime API (except for graphics functions). rCUDA also provides support for the CUDA Driver API. Additionally, is also supports the libraries included within CUDA, such as cuFFT, cuBLAS, or cuSPARSE. Moreover, 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 [37]. These independent GPU contexts also provide robustness against the failure of one the clients.

rCUDA additionally provides specific support for different interconnects [37]. 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, three modules are available: one intended for TCP/IP compatible networks, another one specifically designed for InfiniBand, and a third one intended for RoCE networks.

Regarding the InfiniBand and RoCE communications modules, they are 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 task to be carried out [37].

Moreover, regardless of the exact network used, data exchange between rCUDA clients and GPUs managed by rCUDA servers is pipelined so that higher bandwidth is achieved, as explained in [36]. Internal pipeline

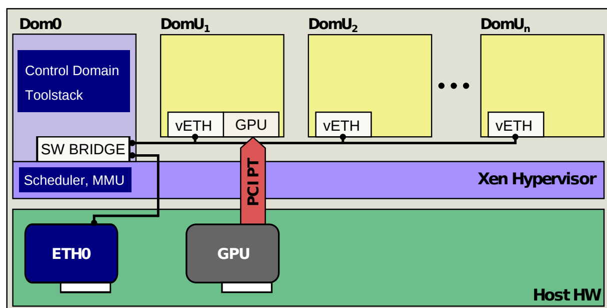
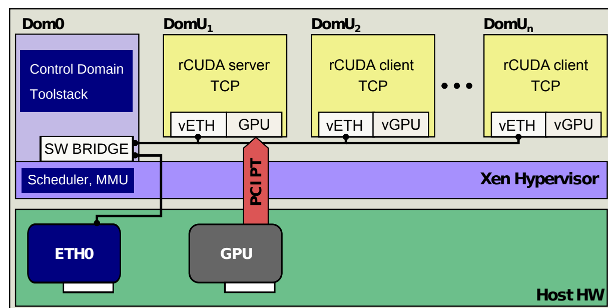
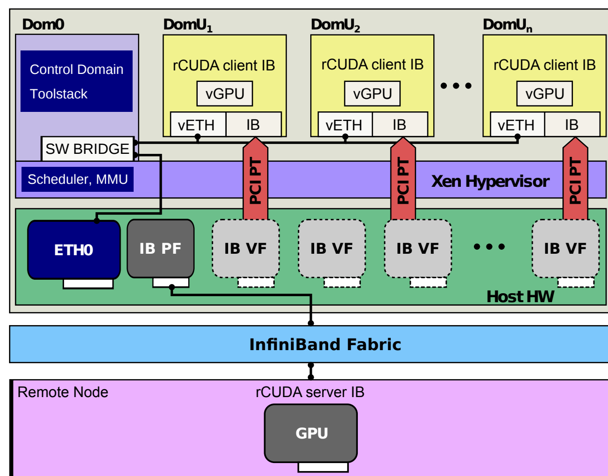


Fig. 3 Typical configuration of a Xen-based system showing how the Ethernet adapter and the GPU available in the host are provided to VMs. The GPU is exclusively assigned to a single VM by making use of the PCI passthrough mechanism. Network connectivity among VMs and between VMs and the external network is provided by means of a software bridge that connects the internal virtual network to the real Ethernet adapter.



(a) Testbed using the virtual network within Xen.



(b) Testbed using InfiniBand to access a remote GPU.

Fig. 4 Testbeds used in the experiments presented in this paper, which make use of rCUDA to provide GPU access to VMs. **(a)** In a single-node testbed, VMs employ the virtual network to access the rCUDA server by means of the TCP/IP protocol stack. **(b)** When an InfiniBand fabric is available, VMs use such interconnect to access a remote rCUDA server.

buffers within rCUDA use pre-allocated pinned memory given the higher throughput of this type of memory.

Finally, notice that previous works such as [36, 34] have already measured the bandwidth attained by rCUDA. However, the results presented in this paper are new, mainly because the context of the performance measurements is different. In previous works the focus was on assessing the performance of rCUDA on native domains, whereas in this work the focus is on stating the throughput of rCUDA on Xen VMs. Notice also that the analysis presented in Section 7 regarding real applications is, again, focused on the use of rCUDA within Xen VMs, what was not previously studied in other works. Finally, notice that although in this paper we focus on analyzing the effect of using rCUDA in Xen VMs, rCUDA can also be used with other hypervisors. For instance, in [30] the rCUDA middleware was used in the context of the KVM hypervisor. Other VM environments such as VMware or VirtualBox could also be leveraged.

4 Testbeds Used in The Experiments

In this work we consider several scenarios in order to provide Xen VMs with access to CUDA accelerators by using the rCUDA middleware. Figure 3 depicts a typical Xen configuration, showing a computer hosting several VMs. It can be seen in the figure that the host hardware comprises, among other devices, an Ethernet network adapter and a GPU. On top of the hardware, a thin software layer (the Xen hypervisor) is installed. Above the hypervisor we can find the VMs (Dom0 and DomU$_i$). Notice that the Dom0 VM is a predefined VM using the Xen Linux kernel and behaves as the configuration and management interface to the hypervisor. The rest of VMs (from DomU$_1$ to DomU$_n$) are unprivileged VMs that can be provided to users. Figure 3 shows how the Ethernet adapter and the GPU are provided to VMs. On the one hand, the Ethernet adapter is owned by the Dom0 VM, which provides connectivity to the rest of VMs by using a software Ethernet bridge, thus creating a virtual network among the VMs. On the other hand, the GPU is assigned, in an exclusive way, to one of the VMs by making use of the PCI passthrough (PT) mechanism. In this manner, this VM is the only one that may access the GPU, as mentioned in Section 2. It is noteworthy the small flexibility that this configuration provides regarding the use of GPUs, given that only one of the VMs can access the GPU.

Once revisited the typical configuration of a Xen-based system, we can describe the testbeds used in the experiments presented in this paper. Notice that we are considering the use of the rCUDA remote GPU virtu-

alization solution in two different scenarios: one where VMs access a GPU located at the same host executing the VMs and another one where the InfiniBand fabric is already present in the cluster and therefore VMs access a GPU installed in another cluster node by making use of this high performance interconnect. Figure 4(a) depicts the first scenario whereas Figure 4(b) presents the second one.

In the first scenario, one of the VMs will have exclusive access to the GPU by making use of the PCI passthrough mechanism. This VM will grant GPU access to the other VMs by using the rCUDA middleware: the rCUDA server will be executed in the VM owning the GPU whereas the other VMs will use the rCUDA client to access the GPU across the virtual Xen network. TCP/IP based communications will be used in this scenario to communicate the rCUDA clients with the rCUDA server. Accordingly, VMs running the rCUDA client will have one or several virtual instances (vGPU) of the real GPU, which is physically connected to the VM DomU_1. Moreover, the VM DomU_1 will be able to use either the real GPU or its virtual instances. Finally, notice that the real GPU can only be assigned to a DomU_i VM because NVIDIA does not provide support for the Xen Linux kernel used in the Dom0 VM.

Regarding the second scenario, shown in Figure 4(b), which uses the InfiniBand fabric already present in the cluster to access a GPU in another node, the firmware in the InfiniBand adapter has been changed, according to the directions in Mellanox User's Guide [23], in order to provide several virtual instances (virtual functions, VF) of the InfiniBand adapter, in addition to the real instance (physical function, PF). Each of these virtual functions will be provided, in an exclusive way, to a Xen VM by using the PCI passthrough mechanism. Moreover, given that an InfiniBand network is available, communication between the rCUDA clients in the VMs and the remote rCUDA server will be based on the use of the high performance InfiniBand Verbs API. Notice that in all the experiments involving the InfiniBand fabric, the remote GPU server is executed in a remote computer which has not been virtualized and also whose InfiniBand network adapter makes use of the original firmware which does not provide virtualization features. Similarly to the scenario shown in Figure 4(a), VMs will have one or several virtual instances of the real GPU, which is physically located in the remote node. Finally, it is important to remark that, although in this work we only consider sharing a single GPU, the rCUDA middleware also allows sharing multiple GPUs.

In addition to the two scenarios depicted in Figure 4, a third scenario that could also be considered would consist of a remote rCUDA server accessed through the 1Gbps Ethernet network usually available in the cluster instead of leveraging the InfiniBand interconnect. Notice, however, that although this configuration is also valid, and VMs would have access to GPUs, the low performance of the 1Gbps Ethernet network would significantly increase the execution time of applications being executed inside VMs. Actually, as shown in [35], the performance of applications using remote GPUs across the 1Gbps Ethernet interconnect is noticeably reduced with respect to the use of a local GPU with CUDA. Therefore, in this work we will not consider this third scenario.

The testbed used in this paper to explore the use of the remote GPU virtualization mechanism inside Xen VMs is composed of three 1027GR-TRF Supermicro nodes. One of them will host the Xen VMs whereas the other two nodes will not make use of VMs. In one of the native domains we will execute the rCUDA server as shown in Figure 4(b) and the other native domain will be used for several comparison purposes. 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 have a Mellanox ConnectX-3 VPI single-port InfiniBand adapter connected to a Mellanox Switch SX6025 (InfiniBand FDR compatible) to exchange data at a maximum rate of 56 Gb/s. Furthermore, an NVIDIA Tesla K20 GPU is installed at each node.

Regarding the software configuration, SUSE Linux Enterprise Server 11 SP3 (x86_64) was used in the three servers, with kernel version 3.0.76-0.11. Additionally, in the node hosting the VMs, Xen version 4.2.2 was used. The same kernel version was used in the Dom0 and all the DomU domains, although for Dom0 the kernel was recompiled in order to activate the Xen options. Moreover, the Mellanox OFED 2.3-1.0.1 (InfiniBand drivers and administrative tools) was used, along with CUDA 6.5 and NVIDIA driver 340.29. Finally, VMs were configured to have 4 cores and 12 GB of RAM memory.

5 Network Performance Observed by Xen VMs

When making use of remote GPU virtualization solutions, the bandwidth characteristics of the communication path between main memory, in the client computer, and GPU memory, in the GPU server, greatly influence the performance of data transfers between them. In this section we present the bandwidth numbers achieved by the interconnects used in our study. These results will help us to better understand the behavior of rCUDA

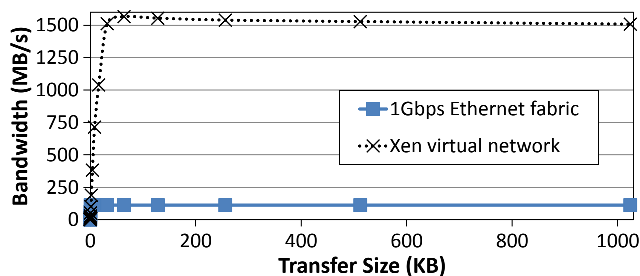
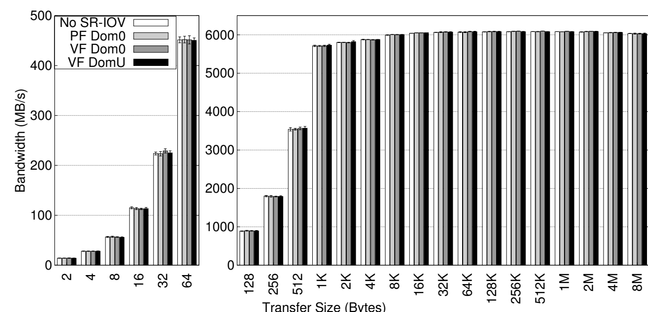
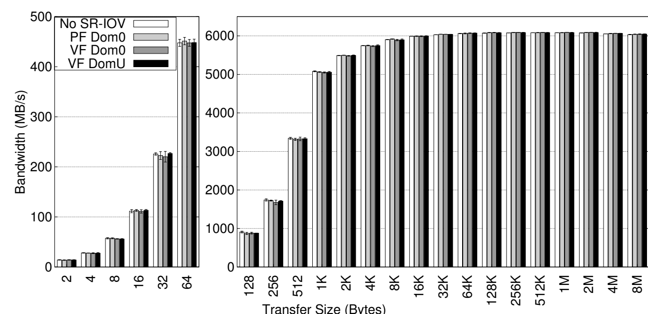


Fig. 5 Bandwidth attained by the virtual network among Xen VMs.



(a) InfiniBand RDMA write bandwidth.



(b) InfiniBand send bandwidth.

Fig. 6 InfiniBand bandwidth tests using ConnectX-3 network cards executed in the different scenarios under study.

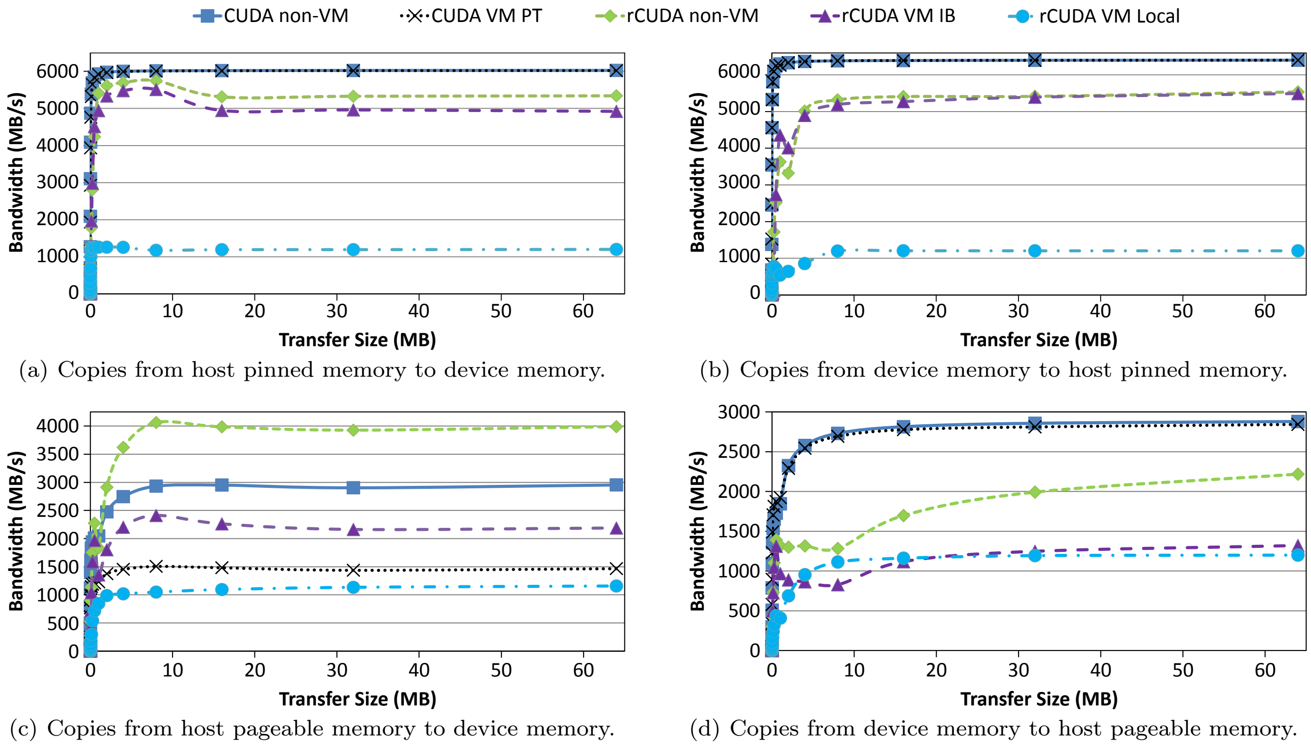
within Xen VMs when used in conjunction with GPU-accelerated applications, later analyzed in Sections 6 and 7.

In this paper we consider the two scenarios shown in Figure 4. In the first one, see Figure 4(a), the virtual network among VMs is used to exchange data among rCUDA clients and servers using TCP/IP. We have analyzed the performance of this network by using the `iperf` tool [2]. Figure 5 shows the bandwidth attained by this network as transfer size increases. The figure also includes, for comparison purposes, the performance of the widely available 1 Gbps Ethernet when used from the inside of a VM. In this case, the virtualization features included in the Xen framework have been leveraged in order to provide the Ethernet adapter to the VM. It can be seen in the figure that the virtual network provides much higher bandwidth than the Ethernet one, achieving even higher bandwidth than the 10Gbps Ethernet. In this regard, notice that starting from transfer sizes equal to 32KB, the performance of the virtual network almost reaches 1600 MB/s.

With respect to the second scenario, shown in Figure 4(b), a wider analysis is required, given the different possibilities that the use of the InfiniBand cards brings in this context. In this scenario, InfiniBand Verbs are used over virtual instances of the InfiniBand network card in order to communicate the rCUDA client and server. For this reason, the performance of the virtualized InfiniBand network card is next compared to the performance of the non-virtualized one. The `ib_read_bw`, `ib_write_bw`, and `ib_send_bw` benchmarks from the Mellanox OFED software distribution were used in order to mimic the use that the rCUDA framework makes of the InfiniBand fabric [36]. In this regard, the `ib_read_bw` and `ib_write_bw` tests use the memory semantics (i.e., RDMA read and RDMA write, respectively), whereas the `ib_send_bw` test makes use of the channel semantics (i.e., send/receive).

Figure 6 shows the bandwidth achieved by the InfiniBand card for different transfer sizes in the scenario depicted in Figure 4(b). The behavior of the memory semantics (RDMA) is shown in Figure 6(a), where only results for the RDMA write case are presented, given that the RDMA read benchmark provided very similar performance. Figure 6(b) shows the channel semantics bandwidth (non-RDMA) when using the `ib_send_bw` benchmark. For comparison purposes, in the experiments carried out with this scenario, we have also considered the performance attained when the tests are executed in the Dom0 VM using both the physical function of the InfiniBand adapter, labeled as *"PF Dom0"*, and also one of the virtual functions, labeled as *"VF Dom0"*. In a similar way, results labeled as *"VF DomU"* refer to the use of a virtual function of the InfiniBand adapter card from the inside of a regular DomU VM. Finally, results labeled as *"No SR-IOV"* have been included for comparison purposes and refer to the use of a non-virtualized InfiniBand card from a native domain. Notice that the bars shown in the figure represent the average of 10 executions of the bandwidth tests configured to perform 20,000 repetitions at each run. Furthermore, this information is complemented, for each transfer size, with the 95% confidence intervals (although these intervals are quite small and can be only distinguished for some of the transfer sizes).

As we can see in Figure 6, the shapes of the bandwidth attained in all the cases under study are, in gen-



(a) Copies from host pinned memory to device memory.

(b) Copies from device memory to host pinned memory.

(c) Copies from host pageable memory to device memory.

(d) Copies from device memory to host pageable memory.

Fig. 7 Bandwidth tests for copies between host and device memory, using CUDA and the rCUDA middleware. Tests have been carried out in the different scenarios depicted in Figure 4 as well as in native domains.

eral, quite similar. Therefore, we can conclude that both the verbs using channel semantics (non-RDMA) and the ones using memory semantics (RDMA) provide similar bandwidth regardless of whether the network card is virtualized or not and also regardless of whether the network card is used from a Dom0 VM or from a $DomU_i$ VM.

6 Performance of rCUDA within Xen VMs

In this section we explore the performance of the rCUDA middleware when used in the context of Xen VMs.

We employ the performance of CUDA as the baseline reference in this analysis, since minimizing the overhead with respect to the performance of CUDA is the goal of any remote GPU virtualization solution. Therefore, we will make use of the `bandwidthTest` benchmark from the NVIDIA CUDA Samples [25] to transfer data from main memory in the client VM to the Tesla K20 GPU (located either in other VM or in a remote real server). In order to use the proper hardware configuration for the baseline CUDA reference, we made use of a configuration which uses the GPU local to the node executing the benchmark, in the traditional way and within a native domain (no Xen VM). Results for this case are referred to as *"CUDA non-VM"* in Figure 7. In a similar way, when CUDA is used in $DomU_1$ in the scenario depicted in Figure 3 by using the PCI passthrough mechanism, the label *"CUDA VM PT"* is used. Regarding the performance of rCUDA, the label *"rCUDA non-VM"* refers to the performance of the rCUDA middleware when used between native domains (no Xen VM involved) making use of the InfiniBand network. These curves are included for comparison purposes. When Xen VMs are involved in the tests, the label *"rCUDA VM IB"* refers to the performance of rCUDA when used in the scenario shown in Figure 4(b). Finally, the performance of rCUDA in the scenario depicted in Figure 4(a) is denoted by the label *"rCUDA VM Local"*.

Figure 7 presents bandwidth results for copies in the host-to-device[6] direction and also for the opposite direction, using both pinned and pageable host memory. Results in Figure 7 are the average from 10 executions of the CUDA `bandwidthTest` test configured to perform 1000 repetitions at each execution. The 95% confidence intervals were also computed, although in this case the variability is very small and thus the value of these intervals is, in average, 1.95%, what suggested

[6] In this work, we will refer to main memory as *host memory* or just *host*, while GPU memory will be referred as *device memory* or simply *device*, according to the well-established usage defined in the CUDA ecosystem.

not to include them in the figures given that these small confidence intervals were going to be hardly visible.

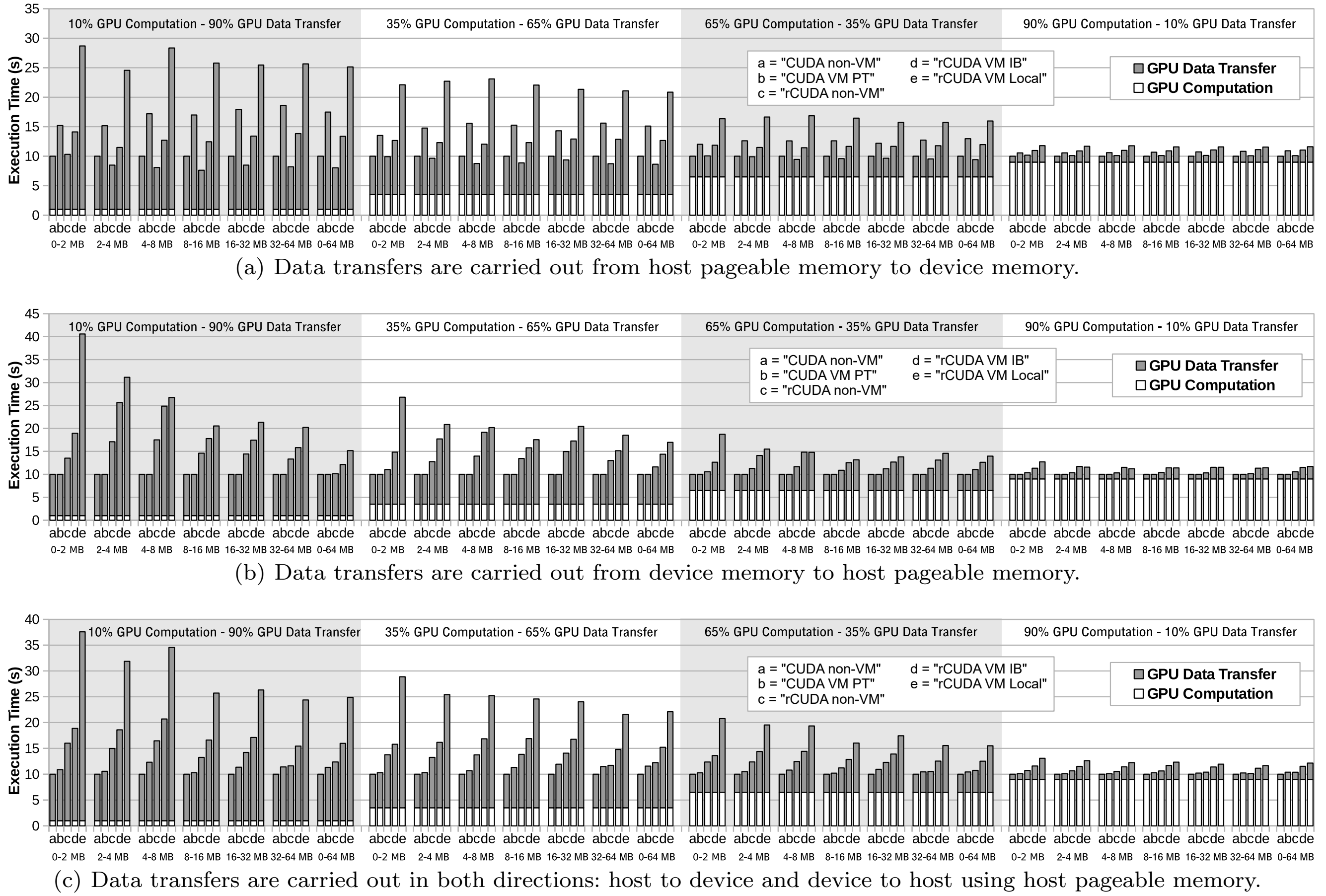
The bandwidth results for pinned memory, presented in Figures 7(a) and 7(b), show that the bandwidth attained for CUDA copies in the native domain and in the Xen VM using PCI passthrough present almost the same performance. In the case of rCUDA using InfiniBand to communicate with a remote GPU server, a slightly smaller bandwidth is achieved. Finally, when rCUDA is used employing the virtual network among VMs, maximum bandwidth for the CUDA memory copies is slightly lower than the one obtained when using the `iperf` tool, shown in previous section.

Regarding the use of pageable memory, it can be seen in Figures 7(c) and 7(d) that in the case of copies from host memory to device memory, there is an important difference between the performance achieved by CUDA in the native domain and that obtained in the Xen VM using the PCI passthrough mechanism, since performance in the former doubles the bandwidth in the latter. Nevertheless, this effect does not appear in the opposite direction (Figure 7(d)), where both cases present almost the same performance. Regarding the use of rCUDA when the InfiniBand network is leveraged, the ratio between the performance obtained in the native domain and that in the VM follows the same trend as for CUDA: the native domain attains twice the performance achieved in the VM. With respect to the performance of rCUDA when the virtual network is used along with TCP/IP based communications, Figure 7(c) shows that this scenario achieves the lowest bandwidth, as it was expected from the results shown in Figure 5. On the other hand, when the device-to-host direction is considered, results are quite different. First, the performance of the baseline CUDA and that of CUDA when used within a VM with PCI passthrough are very similar. Second, the performance of rCUDA in the native and virtualized domains follow the same trend as for the host-to-device direction, but now performance is noticeably reduced. Third, the bandwidth results of rCUDA when the virtual network is used are similar to the performance achieved in the opposite direction.

In summary, we can conclude that the bandwidth attained by PCI passthrough is almost identical to the one achieved by CUDA, except for copies from host pageable memory to device memory, where the bandwidth is reduced to the half. On the other hand, rCUDA over the Xen virtual network results in a very stable behavior in all the scenarios, the bandwidth being limited by the network performance (see Figure 5). Finally, the bandwidth obtained by rCUDA over an InifiniBand network is very close to that of CUDA when using pinned host memory, regardless of whether accessing the remote GPU from a VM or from a native domain. In the case of pageable host memory, the bandwidth when accessing the GPU from a VM is reduced to the half of the one obtained by rCUDA without using VM. This reduction in the performance when involving the VM is more evident in the case of copies from device memory to host memory, where the bandwidth obtained by rCUDA using the VM is the same, regardless of using the Xen virtual network or the InfiniBand one.

Next we analyze the effect of using rCUDA within Xen VMs by making use of a synthetic application. The purpose of using a synthetic application is to be able to modulate the amount of data transferred between host and device as well as controlling the amount of computations carried out in the GPU. In this way, it is possible to analyze the effect of rCUDA on application performance when the application features different percentages of communications and computations. For instance, it is possible to mimic communication intensive applications by setting the amount of data transfers to last 90% of the application execution time while keeping computations to only 10% of execution time. On the contrary, it is feasible to model compute intensive applications by setting the percentage of time devoted by the application to data transfers to only 10% whereas setting the percentage of time used for computations to 90% of execution time. An intermediate case where 35% of the execution time is devoted to computations in the GPU whereas 65% of the execution time is used for data transfers is also possible. The opposite case, with 65% of execution time used for computations and 35% of execution time employed in data transfers would complete a thorough analysis with such a synthetic application.

Figure 8 shows the performance results when the synthetic application is used with several computation and transfer percentages in the same scenarios previously described for Figure 7 (namely, *"CUDA non-VM"*, *"CUDA VM PT"*, *"rCUDA non-VM"*, *"rCUDA VM IB"*, and *"rCUDA VM Local"*). Figure 8(a) depicts the performance results when the data transfers performed by the application follow the host-to-device direction. Figure 8(b) shows the results when data transfers are carried out in the opposite direction. Figure 8(c) presents results when data transfers in both directions are used. Furthermore, results in Figures 8(a), 8(b), and 8(c) take into account the size of the data transfer, given that, as shown in Figure 7, attained bandwidth depends on the exact data transfer size. This is why different size intervals are used for each figure. Each of the size intervals shown in the figures (each of the bars) is the average of 5 repetitions. Each of the repetitions makes



(a) Data transfers are carried out from host pageable memory to device memory.

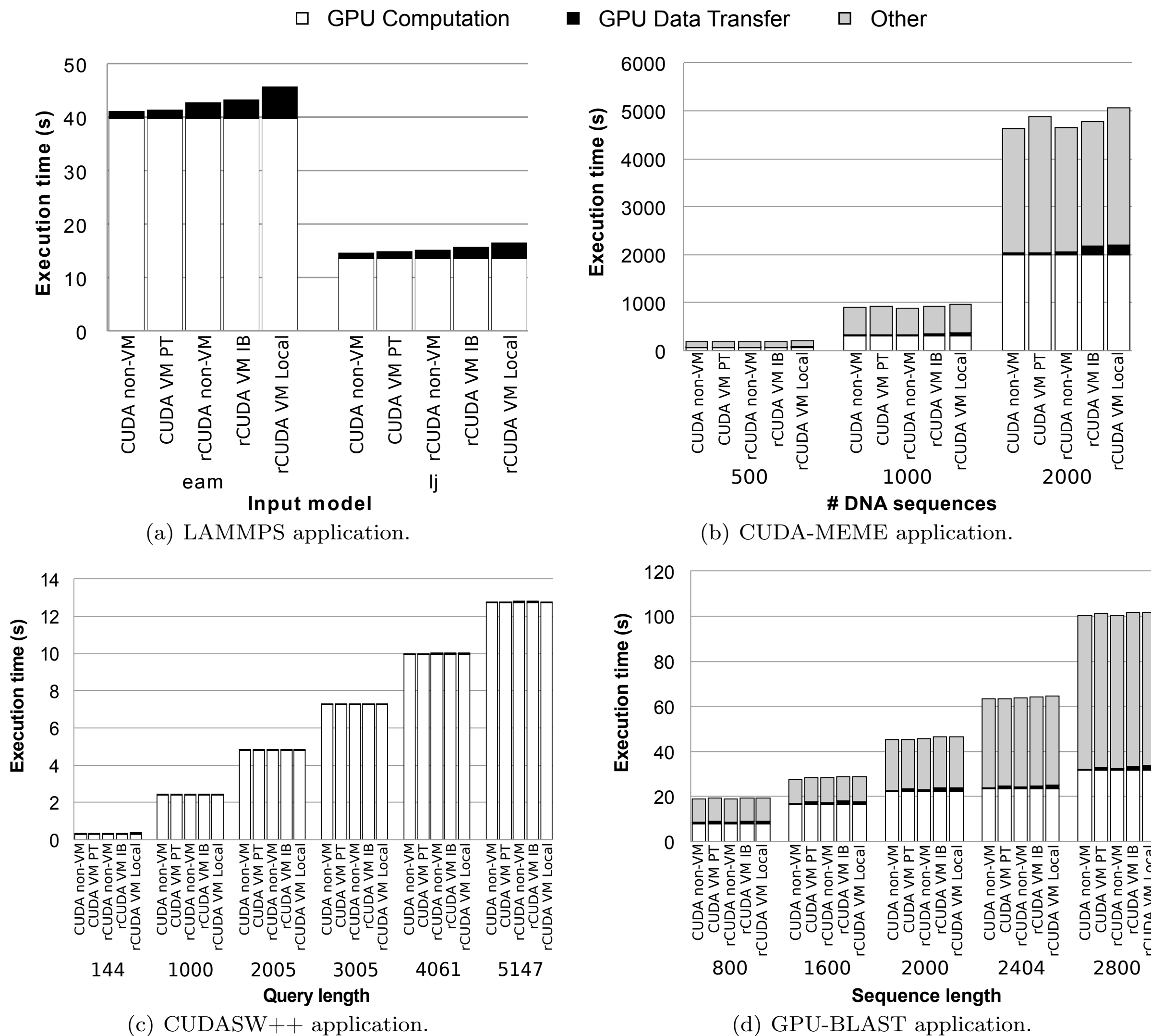
(b) Data transfers are carried out from device memory to host pageable memory.

(c) Data transfers are carried out in both directions: host to device and device to host using host pageable memory.

Fig. 8 Performance of a synthetic application where the percentage of execution time devoted to data transfers to/from the GPU and the percentage of execution time used for computations in the GPU are set by the user. Notice that these percentages are initially established for the executions using CUDA with a local GPU (case "a") by defining the amount of data to be transferred. For the rest of scenarios, this initial amount of data to be transferred is kept constant, thus producing a deviation of the initial percentages. Furthermore, for each size interval, the exact size of data transfers is randomly set.

use of a randomly chosen data transfer size. Finally, notice that the percentages of application execution time devoted to data transfers and GPU computations are initially set for the native CUDA scenario using a local GPU (case "a" in the figures). This is achieved by setting the amount of data to be transferred to/from the GPU. In the rest of scenarios, that very same amount of data is transferred. However, given that the underlying communication channel to/from the GPU is different, a deviation from the initial percentages is produced. This deviation will allow us to determine the overhead introduced by each scenario.

It can be seen in Figures 8(a), 8(b), and 8(c) that the overhead of rCUDA greatly depends on the percentage of data transfers carried out during the execution of the application. In this regard, the overhead introduced by rCUDA when the application devotes 90% of its execution time to data transfers is much higher than when only 10% of the execution time is devoted to data transfers. Additionally, as transfer sizes become larger, the overhead introduced by rCUDA is reduced. This result is consistent with the results shown in Figure 7. In a similar way, Figure 8(a) shows the effect of having less bandwidth in the *"CUDA VM PT"* scenario than in the *"CUDA non-VM"* configuration, as already pointed out in Figure 7(c). Another interesting remark about Figure 8(a) is that application performance is better for the *"rCUDA non-VM"* case than for the *"CUDA non-VM"* scenario. The reason is that rCUDA achieves higher bandwidth than CUDA in the native domains, as already shown in Figure 7(c). Finally, the lower bandwidth of rCUDA for the device-to-host data copies, shown in Figure 7(d), is also visible in Figure 8(b).



(a) LAMMPS application.

(b) CUDA-MEME application.

(c) CUDASW++ application.

(d) GPU-BLAST application.

Fig. 9 Execution time of several applications when executed in different local and remote scenarios. Execution time is broken down into three components: GPU computation, GPU data transfer, and Other.

7 Impact of Xen VMs on Real Applications

In previous sections we have studied how the performance of the Xen virtual network and that of the InfiniBand ConnectX-3 network cards, when used from the inside of Xen VMs, influence the performance of the rCUDA remote GPU virtualization middleware. In order to do so, we used synthetic benchmarks that allowed us to focus on specific characteristics of the virtualization solution. In this section we study how the performance of these interconnects, along with the use of Xen VMs, influence the execution time of real applications. Remember that this is the actual goal of our work: to explore the use of the remote GPU virtualization mechanism in order to provide CUDA acceleration to applications running inside Xen VMs, and characterizing such exploration by using as a metric the overhead that applications experience when accessing GPUs outside their Xen VM by using a remote GPU virtualization framework. We consider two different types of applications in this section: those making use of a single GPU and those that offload computations to more than one GPU. In next section we analyze the first kind of applications. In Section 7.2 we will present performance results for the second type of applications.

7.1 Applications Using One GPU

The applications analyzed in this section are LAMMPS [15], CUDA-MEME [18], CUDASW++ [19], and GPU-BLAST [42], listed in the NVIDIA GPU-Accelerated Applications Catalog [28]:

– LAMMPS is a molecular dynamics simulator that can be used as a parallel particle simulator at the atomic, mesoscopic, or continuum scale. For our tests we use the release from Dec. 9, 2014, and benchmarks `in.eam` and `in.lj`, with a factor scale of 5 in all three dimensions.
– CUDA-MEME is a parallel formulation and implementation of the MEME motif discovery algorithm using the CUDA programming model. In particular, we have used its latest release, version 3.0.15, for our study, along with the test cases available in the application website [17].
– CUDASW++ is a bioinformatics software for Smith-Waterman protein database searches that takes advantage of the massively parallel CUDA architecture of NVIDIA Tesla to perform sequence searches. In particular, we have used its latest release, version 3.1, with the latest Swiss-Prot database and the example query sequences available in the application's website.
– GPU-BLAST has been designed to accelerate the gapped and ungapped protein sequence alignment algorithms of the NCBI-BLAST implementation using GPUs. It is integrated into the NCBI-BLAST code and produces identical results. We use the release 1.1 in our experiments, where we have followed the installation instructions for sorting a database and creating a GPU database. We then use the query sequences that come with the application package to search the database.

Figure 9 shows the execution time of these four applications when executed in the same scenarios as in the previous section: execution with CUDA with a local GPU in a native domain (*"CUDA non-VM"*) and within a Xen VM accessing the GPU in the host by making use of PCI passthrough (*"CUDA VM PT"*). In the case of rCUDA, the three scenarios considered (*"rCUDA non-VM"*, *"rCUDA VM IB"*, and *"rCUDA VM Local"*) refer to the ones already described in the previous section. Every experiment has been performed 10 times, so that the figures show the averaged results. Furthermore, the 95% confidence intervals were computed, but they are so small that their inclusion in the figures provided no additional important information. In addition to execution time, the plots in the figure also include a breakdown of the execution time, which is split into three different components: (1) time required to transfer data to/from the GPU (*"GPU Data Transfer"*), (2) time spent carrying out computations in the GPU (*"GPU Computation"*), and (3) time spent in tasks not involving the GPU, such as CPU computations and I/O (*"Other"*).

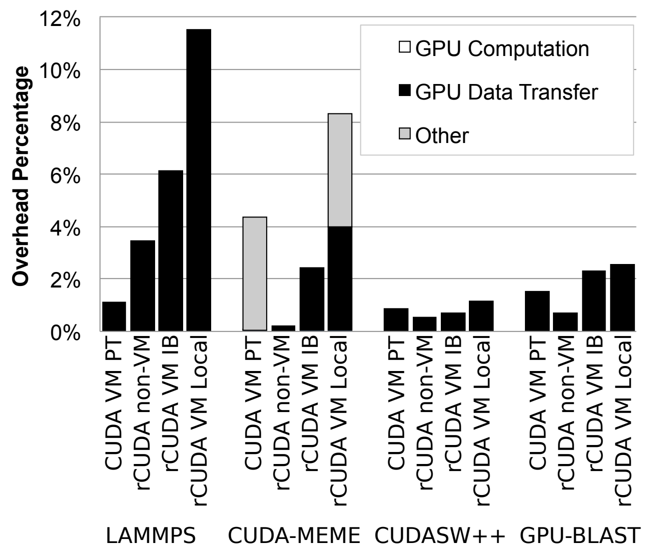
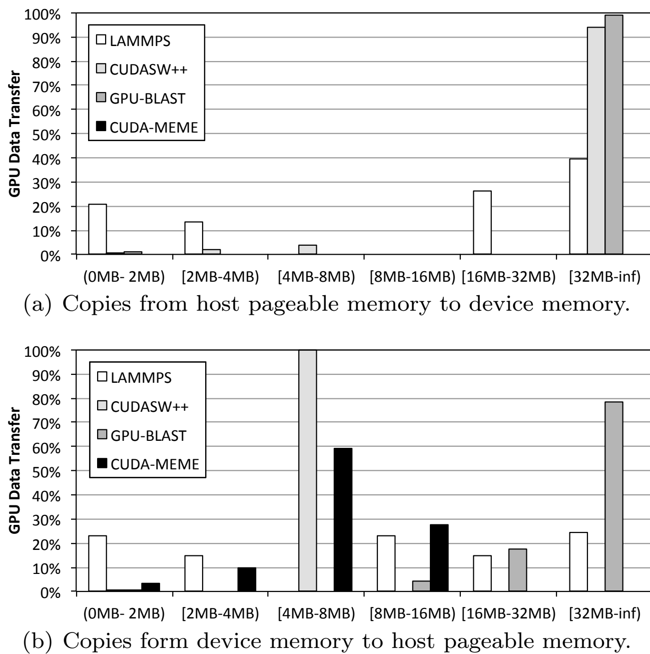


Fig. 10 Average overhead with respect to executions with CUDA in a native domain for the four applications depicted in Figure 9.

Execution times presented in Figure 9 show that the four applications have a similar behavior, spending a very small portion of time for transferring data to the GPU, and spending the rest of the time performing computations either in the CPU or in the GPU. More specifically, in the case of GPU-BLAST and CUDA-MEME applications, they present periods of time in which the GPU is not used. On the contrary, both LAMMPS and CUDASW++ keep the GPU busy for almost all the execution time.

Figure 10 shows the average overhead with respect to executions with CUDA in a native domain for the four applications. This figure shows that rCUDA overhead in LAMMPS, CUDASW++ and GPU-BLAST applications is mainly due to data transfers between main memory and GPU memory. Additionally to the overhead of transfers, the CUDA-MEME application also presents a performance decrease when using a VM that makes use of the PCI passthrough technique. As we can see, this additional overhead is not due to the increase of GPU data transfer time, but to the time spent in other tasks by the PCI passthrough technique (referred to as *"Other"* in the figure), which are out of the scope of this paper.

In general, the overhead of rCUDA is mainly due to data transfers between main memory and GPU memory. This was expected because once data is in the GPU memory, GPU computations require the same amount of time to be completed as in a native environment. In average, in our experiments, the overhead of running GPU-accelerated applications in a Xen VM with respect to a native domain is 2%, 2.8%, and 5.8% when



(a) Copies from host pageable memory to device memory.

(b) Copies form device memory to host pageable memory.

Fig. 11 Histograms showing the percentage of transferred data according to message size.

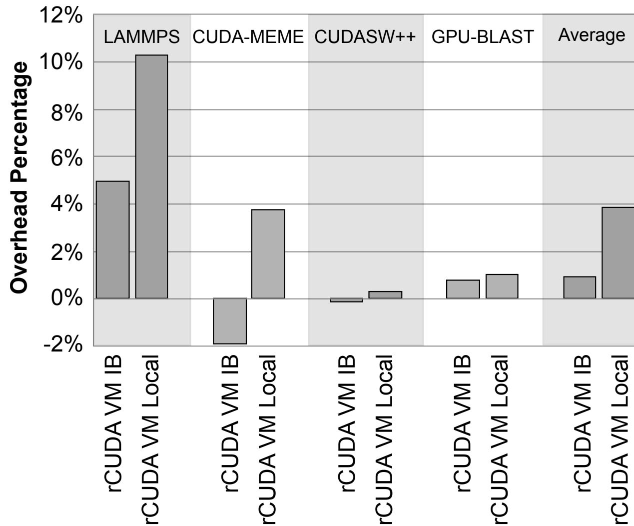


Fig. 12 Average overhead experienced by applications with respect to executions with CUDA using the PCI passthrough from the inside of a VM.

using PCI passthrough, rCUDA over an InfiniBand fabric, and rCUDA over the Xen virtual network, respectively.

Once the main cause of the overhead has been studied, a deeper analysis is necessary to characterize the behavior of each application. In this regard, as shown in Figure 7, time required for data transfers varies depending on the copy direction (to or from GPU memory) and the memory type (pageable or pinned memory). In order to analyze the influence of these different transfer bandwidths on application execution time, Table 1 presents the total amount of data transferred in each direction, as well as the memory type. As we can observe, none of the applications uses pinned memory. Additionally, given that bandwidth attained for data copies also depends on transfer size, Figure 11 depicts how the total amount of transferred data shown in Table 1 is split into different message sizes in order to be actually transferred. Putting all this information together, Table 1 shows that CUDASW++ and GPU-BLAST mainly copy data from main memory to GPU memory, more than 90% of these transfers being done with messages greater than 32MB, as depicted in Figure 11(a). On the other hand, according to Table 1, the majority of the transfers in the CUDA-MEME application are from GPU memory to main memory, and almost 90% of the transferred data is copied in message sizes between 4 and 16MB, as shown in Figure 11(b). Finally, the LAMMPS application presents similar percentage of transfers in both copy directions, with 80% of data transferred in messages of size larger than 2MB.

Applications	**HtoD pageable**		**DtoH Pageable**	
	GB	**%**	**GB**	**%**
LAMMPS	3	59	2	41
CUDASW++	0.195	98	0.004	2
GPU-BLAST	1.3	79	0.356	21
CUDA-MEME	0.048	0	100	100

Table 1 Data transfers in the applications under analysis

With the data gathered in this analysis, we can complete our study and conclude that when comparing the overhead of PCI passthrough and rCUDA (used from the inside of a VM), their behavior with respect to each other mainly depends on the type of data transfers they mostly perform (pageable host-to-device or pageable device-to-host), which present a very different performance as shown in Figures 7(c) and 7(d). In this regard, for applications in which copies from host to device have a bigger weight, PCI passthrough performs worse. On the contrary, for applications that mainly transfer data from device to host, then rCUDA performs worse. That is, there is a direct dependency of application performance on the bandwidth attained for each copy direction. This result points out the impact on application performance of the bandwidth attained by the underlying network connecting main memory and GPU memory.

Finally, notice that current cloud computing providers use the PCI passthrough mechanism to provide applications with CUDA acceleration. However, the average

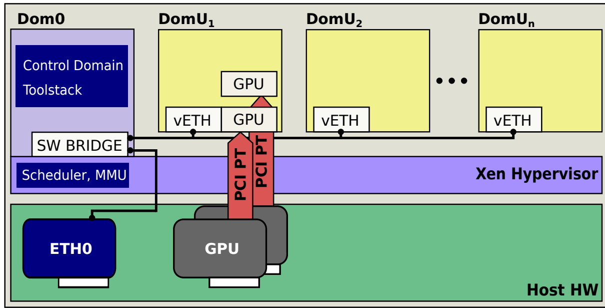


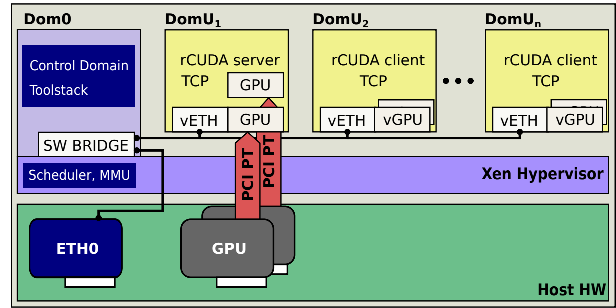
Fig. 13 Configuration of a Xen-based system showing two GPUs assigned to one of the VMs. The GPU assignment is carried out by making use of the PCI passthrough mechanism. Therefore, both GPUs can only be used by the VM owning them.

overheads shown in Figure 10 are computed with respect to executions with CUDA in a native domain. Therefore, in order to provide the right perspective, it is advisable to use as the baseline reference the performance of applications when using the PCI passthrough from the inside of a VM instead. In this regard, Figure 12 shows the average overhead experienced by applications when using the rCUDA middleware using as the baseline reference their performance when executed using the PCI passthrough mechanism in order to access the GPUs. As can be seen the overhead is very low when an InfiniBand network is available. In the cases of CUDA-MEME and CUDASW++ the execution time is even lower than the obtained with the PCI passthrough mechanism. In the case that rCUDA is used through the virtual network, we can see that the overhead increases with respect to the previous scenario but this overhead remains low, less than 4% on average.

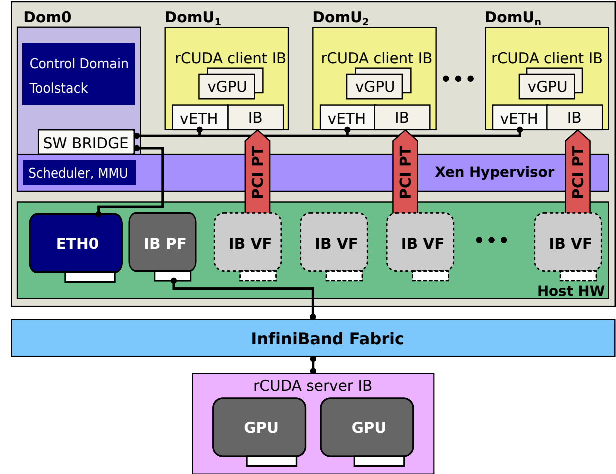
7.2 Applications Using Multiple GPUs

In the previous section we have presented an analysis of four different applications that offloaded their computations to one GPU. However, there are applications that can make use of several GPUs in order to further reduce their execution time. In this section we present performance results for applications using two GPUs.

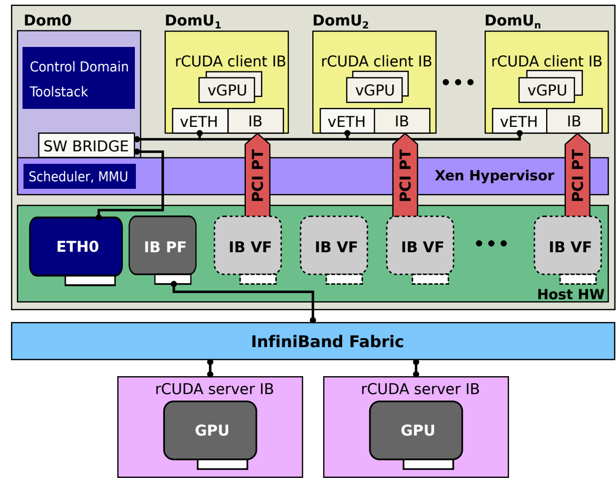
Several system configurations can be used when several GPUs are leveraged in the context of Xen VMs and rCUDA. Figures 13 and 14 show four of these configurations. Figure 13 depicts the simplest one, where a Xen VM is assigned two GPUs by making use of the PCI passthrough mechanism and the GPUs are accessed by means of CUDA. This configuration is similar to that depicted in Figure 3 although two GPUs are used now. On the other hand, Figure 14 shows the configurations when rCUDA is used within Xen VMs in order to ac-



(a) Testbed using the virtual network within Xen.



(b) Testbed using the InfiniBand fabric to access two remote GPUs located in the same remote node.

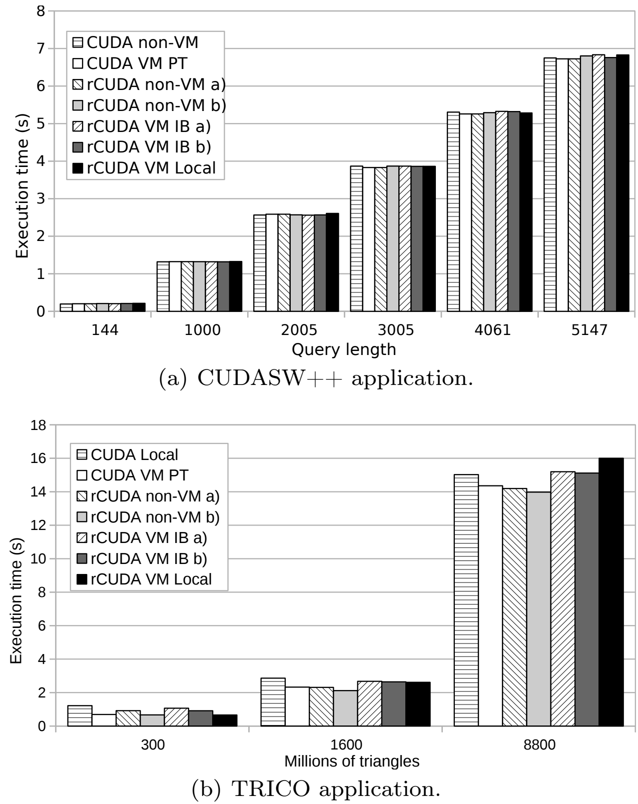


(c) Testbed using the InfiniBand fabric to access two remote GPUs located in two remote nodes.

Fig. 14 Testbeds used with rCUDA. Two GPUs are provided to VMs. (a) In a single-node scenario, VMs use the virtual network (TCP/IP) to access the rCUDA server running in one of the VMs. (b/c) When an InfiniBand fabric is available in the cluster, VMs use such interconnect in order to access the remote GPUs, which can be located either in the same (b) or in different (c) remote nodes.

cess two GPUs. Figure 14(a) shows the scenario where rCUDA is used to access the two GPUs located at the same host that executes the VMs. This configuration is similar to that presented in Figure 4(a) but two GPUs are leveraged now. Notice that the two GPUs are assigned, by making use of the PCI passthrough technique, to one of the VMs, where the rCUDA server is being executed. Furthermore, when the InfiniBand network is present in the cluster, two additional configurations are feasible: (1) both GPUs are located in the same remote node and (2) two remote nodes are used, each with one GPU. Figures 14(b) and 14(c) depict, respectively, these configurations, which are similar to the one shown in Figure 4(b) although two GPUs are used now. Finally, in the experiments carried out in this section, GPUs are also used in native domains. In the case of CUDA, the two GPUs are installed at the node running the application. In the case of rCUDA the two GPUs can be located in one remote node or in two remote nodes. All these scenarios will be considered in the performance tests carried out in this section.

Two applications will be used as test cases in this section: the CUDASW++ application already used in the previous section and the TRICO (TRIangle COunt) application [32]. TRICO is a CUDA implementation of a parallel algorithm for counting triangles (i.e. 3-cycles) in large graphs which additionally is able to take advantage of all the GPUs available in the node where it is being executed.



(a) CUDASW++ application.

(b) TRICO application.

Fig. 15 Performance of two applications when executed in different local and remote scenarios involving Xen VMs.

Figure 15(a) shows the performance results of the CUDASW++ application when executed using two GPUs. Label *"CUDA non-VM"* refers to the execution with CUDA with two local GPUs in a native domain whereas the case for the application being executed within a Xen VM and accessing the GPUs in the host by making use of PCI passthrough is referred to as *"CUDA VM PT"*. In the case of rCUDA, executions in a native domain (no VM involved) are referred to as *"rCUDA non-VM a)"* when both GPUs are located in the same remote node. When both GPUs are located in different remote nodes, the label *"rCUDA non-VM b)"* is used. In a similar way, when using rCUDA within a Xen VM, label *"rCUDA VM Local"* refers to the scenario depicted in Figure 14(a) where the virtual network provided by Xen is used to access the GPUs located at the same host executing the VM. Finally, labels *"rCUDA VM IB a)"* and *"rCUDA VM IB b)"* refer to the scenarios depicted in Figures 14(b) and 14(c), respectively, where the InfiniBand fabric is present in the cluster and therefore one or two remote GPU servers are used. It can be seen in Figure 15(a) that the performance of rCUDA when two GPUs are used is similar to that of CUDA in all the scenarios considered. On the other hand, Figure 15(b) presents the performance results for the TRICO application when two GPUs are used. In this case, a higher variability in the execution time of the application is observed, being the worst case the execution for 8800 millions of triangles with rCUDA when the virtual network provided by Xen is used with TCP/IP (scenario depicted in Figure 14(a)).

8 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 Xen VMs. We have considered two different scenarios: (1) in those clusters not leveraging an InfiniBand interconnect, a VM grants GPU access to the other VMs concurrently running in the same host, and (2) in those clusters were an InfiniBand fabric is already present, VMs access a remote GPU located in another node of the cluster.

First, we have used synthetic benchmarks to characterize the performance attained when using different underlying network fabrics. Afterwards, we have stud-

ied the impact on execution time of running scientific applications inside the virtualized domain.

The main conclusion from our exploration is that remote GPU virtualization solutions are a feasible option to provide CUDA acceleration services to Xen VMs. Our experiments showed that the overhead of executing accelerated applications within Xen VMs with respect to currently available approaches (i.e., PCI passthrough) greatly depends on the internals of each application, being negligible (0.92% on average) when the cluster already includes an InfiniBand interconnect and very low (3.84% on average) in the case of using the internal virtual network within Xen.

Nevertheless, overhead percentages are not the only result to keep from this exploration. Another important conclusion is that remote GPU virtualization solutions provide to data center managers the configuration flexibility that Xen currently lacks. In this manner, remote GPU virtualization frameworks not only provide the possibility to concurrently offer GPU acceleration services to several VM instances being executed in the same host, but they also provide the possibility of offering differentiated services to different data center users, given that cluster administrators keep complete control on how GPUs are shared among users. For example, it could be possible to create two groups of users for a given application: a smaller group including those users willing to pay more money in order to achieve higher application performance (i.e., not sharing GPUs) and a bigger group composed of those users preferring to wait some more time for their application to complete execution but at a lower economic cost (i.e., sharing GPUs among VMs).

As for future work, we plan to analyze the effect on application performance of sharing the available GPUs among several VMs. In this regard, it is necessary to develop a scheduler that coordinates the use of GPU memory among the several VMs sharing the GPUs. This scheduler is required in order to ensure that applications do not experience out-of-memory issues due to the fact that several of them are allocating GPU memory at the same time. Migrating GPU jobs [33] will be a useful technique in order to better coordinate the use of GPU memory resources among VMs. Finally, a new communication layer within rCUDA based on the use of shared memory will also be investigated. The purpose of this new shared-memory based communication layer is to avoid using the virtual network provided by the Xen hypervisor, thus attaining higher performance.

Acknowledgements This work was funded by the Generalitat Valenciana under Grant PROMETEO/2017/077. Authors are also grateful for the generous support provided by Mellanox Technologies Inc.

References

1. (2015) AMD Multiuser GPU, hardware-based virtualized solution. http://www.amd.com/Documents/Multiuser-GPU-Datasheet.pdf, accessed 19 October 2015
2. (2015) iperf3: A TCP, UDP, and SCTP network bandwidth measurement tool. https://github.com/esnet/iperf, accessed 19 October 2015
3. (2015) Kernel-based Virtual Machine, KVM. http://www.linux-kvm.org, accessed 19 October 2015
4. (2015) NVIDIA GRID Technology. http://www.nvidia.com/object/grid-technology.html, accessed 19 October 2015
5. (2015) Oracle VM VirtualBox. http://www.virtualbox.org/, accessed 19 October 2015
6. (2015) Shadowfax II - scalable implementation of GPGPU assemblies. http://keeneland.gatech.edu/software/keeneland/kidron, accessed 19 October 2015
7. (2015) V-GPU: GPU virtualization. https://github.com/zillians/platform_manifest_vgpu, accessed 19 October 2015
8. (2015) VMware virtualization. http://www.vmware.com/, accessed 19 October 2015
9. (2015) Xen Project. http://www.xenproject.org/, accessed 19 October 2015
10. Agarwal PK, Hampton S, Poznanovic J, Ramanthan A, Alam SR, Crozier PS (2013) Performance modeling of microsecond scale biological molecular dynamics simulations on heterogeneous architectures. Concurrency and Computation: Practice and Experience 25(10):1356–1375
11. Felter W, Ferreira A, Rajamony R, Rubio J (2014) An Updated Performance Comparison of Virtual Machines and Linux Containers. In: IBM Research Report
12. Giunta G, Montella R, Agrillo G, Coviello G (2010) A GPGPU Transparent Virtualization Component for High Performance Computing Clouds. In: Proc. of the Euro-Par Parallel Processing, Euro-Par, pp 379–391
13. Gupta V, Gavrilovska A, Schwan K, Kharche H, Tolia N, Talwar V, Ranganathan P (2009) GViM: GPU-accelerated virtual machines. In: Proc. of the ACM Workshop on System-level Virtualization for High Performance Computing, HPCVirt, pp 17–24
14. Jo H, Jeong J, Lee M, Choi DH (2013) Exploiting GPUs in Virtual Machine for BioCloud. BioMed research international 2013

15. Laboratories SN (2013) LAMMPS Molecular Dynamics Simulator. http://lammps.sandia.gov/, accessed 19 October 2015
16. Liang TY, Chang YW (2011) GridCuda: A Grid-Enabled CUDA Programming Toolkit. In: Proc. of the IEEE Advanced Information Networking and Applications Workshops, WAINA, pp 141–146
17. Liu Y (2014) CUDA-MEME. https://sites.google.com/site/yongchaosoftware/mcuda-meme, accessed 19 October 2015
18. Liu Y, Schmidt B, Liu W, Maskell DL (2010) CUDA-MEME: Accelerating motif discovery in biological sequences using CUDA-enabled graphics processing units. Pattern Recognition Letters 31(14):2170–2177
19. Liu Y, Wirawan A, Schmidt B (2013) CUDASW++ 3.0: accelerating Smith-Waterman protein database search by coupling CPU and GPU SIMD instructions. BMC Bioinformatics 14(1):1–10
20. Luo GH, Huang SK, Chang YS, Yuan SM (2014) A parallel Bees Algorithm implementation on GPU. Journal of Systems Architecture 60(3):271–279
21. Mellanox (2013) ConnectX-3 VPI Single and Dual QSFP+ Port Adapter Card User Manual 1.7. http://www.mellanox.com/related-docs/user_manuals/ConnectX-3_VPI_Single_and_Dual_QSFP_Port_Adapter_Card_User_Manual.pdf, accessed 19 October 2015
22. Mellanox (2014) Connect-IB Single and Dual QSFP+ Port PCI Express Gen3 x16 Adapter Card User Manual. http://www.mellanox.com/related-docs/user_manuals/Connect-IB_Single_and_Dual_QSFP+_Port_PCI_Express_Gen3_%20x16_Adapter_Card_User_Manual.pdf, accessed 19 October 2015
23. Mellanox (2014) Mellanox OFED for Linux User Manual. http://www.mellanox.com/related-docs/prod_software/Mellanox_OFED_Linux_User_Manual_v2.3-1.0.1.pdf, accessed 19 October 2015
24. Merritt AM, Gupta V, Verma A, Gavrilovska A, Schwan K (2011) Shadowfax: Scaling in Heterogeneous Cluster Systems via GPGPU Assemblies. In: Proc. of the International Workshop on Virtualization Technologies in Distributed Computing, VTDC, pp 3–10
25. NVIDIA (2013) The NVIDIA GPU Computing SDK Version 5.5
26. NVIDIA (2015) CUDA C Programming Guide 7.5. http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf, accessed 19 October 2015
27. NVIDIA (2015) CUDA Runtime API Reference Manual 7.5. http://docs.nvidia.com/cuda/pdf/CUDA_Runtime_API.pdf, accessed 19 October 2015
28. NVIDIA (2015) NVIDIA Popular GPU-Accelerated Applications Catalog. http://www.nvidia.com/content/gpu-applications/PDF/GPU-apps-catalog-mar2015.pdf, accessed 19 October 2015
29. Oikawa M, Kawai A, Nomura K, Yasuoka K, Yoshikawa K, Narumi T (2012) DS-CUDA: A Middleware to Use Many GPUs in the Cloud Environment. In: Proc. of the SC Companion: High Performance Computing, Networking Storage and Analysis, SCC, pp 1207–1214
30. Pérez F, Reaño C, Silla F (2016) Providing CUDA Acceleration to KVM Virtual Machines in InfiniBand Clusters with rCUDA. In: 16th International Conference Distributed Applications and Interoperable Systems (DAIS), Springer International Publishing, pp 82–95
31. Playne DP, Hawick KA (2009) Data Parallel Three-Dimensional Cahn-Hilliard Field Equation Simulation on GPUs with CUDA. In: Proc. of the Parallel and Distributed Processing Techniques and Applications, PDPTA, pp 104–110
32. Polak A (2016) Counting triangles in large graphs on GPU. IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW) pp 740–746
33. Prades J, Silla F (2017) Turning GPUs into Floating Devices over The Cluster: The Beauty of GPU Migration. In: Proc. of the 6th Workshop on Heterogeneous and Unconventional Cluster Architectures and Applications (HUCAA)
34. Reaño C, Silla F (2016) Reducing the performance gap of remote GPU virtualization with InfiniBand Connect-IB. In: 2016 IEEE Symposium on Computers and Communication (ISCC), pp 920–925
35. Reaño C, Mayo R, Quintana-Ortí E, Silla F, Duato J, Peña A (2013) Influence of InfiniBand FDR on the performance of remote GPU virtualization. In: Proc. of the IEEE International Conference on Cluster Computing, CLUSTER, pp 1–8
36. Reaño C, Silla F, Shainer G, Schultz S (2015) Local and Remote GPUs Perform Similar with EDR 100G InfiniBand. In: Proceedings of the Industrial Track of the 16th International Middleware Conference, ACM, Middleware Industry '15, pp 4:1–4:7
37. Reaño C, Silla F, Duato J (2017) Enhancing the rCUDA Remote GPU Virtualization Framework: From a Prototype to a Production Solution. In: Proceedings of the 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Comput-

ing, IEEE Press, CCGrid '17, pp 695–698
38. Semnanian A, Pham J, Englert B, Wu X (2011) Virtualization Technology and its Impact on Computer Hardware Architecture. In: Proc. of the Information Technology: New Generations, ITNG, pp 719–724
39. Shi L, Chen H, Sun J (2009) vCUDA: GPU accelerated high performance computing in virtual machines. In: Proc. of the IEEE Parallel and Distributed Processing Symposium, IPDPS, pp 1–11
40. Song J, et al (2014) KVMGT: a Full GPU Virtualization Solution. In: KVM Forum
41. Surkov V (2010) Parallel option pricing with Fourier space time-stepping method on graphics processing units. Parallel Computing 36(7):372–380
42. Vouzis PD, Sahinidis NV (2011) GPU-BLAST: using graphics processors to accelerate protein sequence alignment. Bioinformatics 27(2):182–188
43. Walters JP, Younge AJ, Kang DI, Yao KT, Kang M, Crago SP, Fox GC (2014) GPU-Passthrough Performance: A Comparison of KVM, Xen, VMWare ESXi, and LXC for CUDA and OpenCL Applications. In: Proc. of the IEEE International Conference on Cloud Computing, CLOUD
44. Wu H, Diamos G, Sheard T, Aref M, Baxter S, Garland M, Yalamanchili S (2014) Red Fox: An Execution Environment for Relational Query Processing on GPUs. In: Proc. of the International Symposium on Code Generation and Optimization, CGO
45. Yamazaki I, Dong T, Solcà R, Tomov S, Dongarra J, Schulthess T (2014) Tridiagonalization of a dense symmetric matrix on multiple GPUs and its application to symmetric eigenvalue problems. Concurrency and Computation: Practice and Experience 26(16):2652–2666
46. Yang CT, Wang HY, Ou WS, Liu YT, Hsu CH (2012) On implementation of GPU virtualization using PCI pass-through. In: Proc. of the IEEE Cloud Computing Technology and Science, CloudCom, pp 711–716
47. Yuancheng Luo D (2008) Canny edge detection on NVIDIA CUDA. In: Proc. of the Computer Vision and Pattern Recognition Workshops, CVPR Workshops, pp 1–8
48. Zhang J, Lu X, Arnold M, Panda D (2015) MVAPICH2 over OpenStack with SR-IOV: An Efficient Approach to Build HPC Clouds. In: Proc. of the IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, CCGrid, pp 71–80