Received 26 April 2016; | Revised 6 June 2016; | Accepted 6 June 2016

DOI: xxx/xxxx

SPECIAL ISSUE PAPER

Redesigning the rCUDA Communication Layer for a Better Adaptation to the Underlying Hardware

Carlos Reaño* | **Federico Silla**

[1]Universitat Politècnica de València, Spain

Correspondence
*Carlos Reaño, Universitat Politècnica de València, Departamento de Informática de Sistemas y Computadores (DISCA), Edificio 1G, 46022 Valencia, Spain. Email: carregon@gap.upv.es

Summary

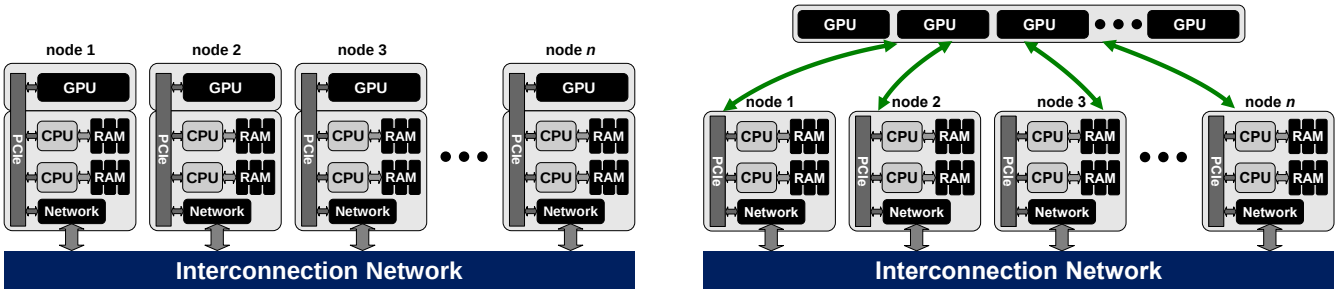
The use of Graphics Processing Units (GPUs) has become a very popular way to accelerate the execution of many applications. However, GPUs are not exempt from side effects. For instance, GPUs are expensive devices which additionally consume a non-negligible amount of energy even when they are not performing any computation. Furthermore, most applications present low GPU utilization. To address these concerns, the use of GPU virtualization has been proposed. In particular, remote GPU virtualization is a promising technology that allows applications to transparently leverage GPUs installed in any node of the cluster.

In this paper the remote GPU virtualization mechanism is comparatively analyzed across three different generations of GPUs. The first contribution of this study is an analysis about how the performance of the remote GPU virtualization technique is impacted by the underlying hardware. To that end, the Tesla K20, Tesla K40 and Tesla P100 GPUs along with FDR and EDR InfiniBand fabrics are used in the study. The analysis is performed in the context of the rCUDA middleware. It is clearly shown that the GPU virtualization middleware requires a comprehensive design of its communication layer, which should be perfectly adapted to every hardware generation in order to avoid a reduction in performance. This is precisely the second contribution of this work: redesigning the rCUDA communication layer in order to improve the management of the underlying hardware. Results show that it is possible to improve bandwidth up to 29.43%, which translates into up to 4.81% average less execution time in the performance of the analyzed applications.

KEYWORDS:
GPGPU, CUDA, Virtualization, HPC, InfiniBand

1 | INTRODUCTION

Accelerators are increasingly being used in current High Performance Computing (HPC) deployments and also in data centers in order to reduce the execution time of applications. In this regard, several kinds of accelerators have been considered by industry. First, Graphics Processing Units (GPUs) have been widely adopted in many supercomputers and data centers as a way to increase computing power. Among the many different GPU



(a) Example of a GPU-accelerated cluster.

(b) Logical configuration of a cluster when the remote GPU virtualization technique is used.

FIGURE 1 Comparison, from a logical point of view, of two cluster configurations: with and without remote GPU virtualization.

flavors and vendors, the enterprise solutions provided by NVIDIA, such as the recent Tesla P100 GPU [1], or the previous Tesla K40 [2] or Tesla K20 [3] devices, have acquired an important market share. The second type of accelerators that have been introduced in many computing facilities are those based on the Xeon Phi architecture [4] by Intel. Those accelerators are composed of a few dozens of simple x64 cores augmented with vector units able to speed up mathematical computations. This kind of accelerators are also referred to as Many Integrated Cores (MICs). Finally, Field-Programmable Gate Arrays (FPGAs) are the third type of accelerators that are recently being considered by researchers worldwide [5 6]. Among the three mentioned kinds of accelerators, in this paper we will focus on GPUs. Many different applications have already been adapted to use GPUs. For instance, applications from domains as different as data analysis (Big Data) [7], computational fluid dynamics [8], chemical physics [9], computational algebra [10], image analysis [11], finance [12], biology [13], and artificial intelligence [14], among others, are using GPUs in order to reduce their execution time.

GPUs, however, are expensive devices that require a non-negligible amount of energy even when not performing computations[1]. Furthermore, for most applications their utilization is relatively low, therefore avoiding a fast amortization of the initial economic investment carried out when building the computing facility. In order to address these concerns, GPU virtualization has been proposed by different authors [15 16 17]. GPU virtualization allows GPUs to be virtualized and concurrently shared among several applications in a transparent way. That is, no modification is required to the source code of applications. In this way, GPU utilization is noticeably increased at the same time that cluster throughput (measured as the amount of completed jobs per time unit) is boosted, thus allowing a larger revenue to data center owners. Overall energy consumption is also reduced because, for a given workload composed of a certain amount of jobs to be executed, GPU virtualization allows that workload to be completed in a smaller amount of time than when no GPU virtualization is leveraged.

An evolution of GPU virtualization consists of locating the virtual GPUs anywhere in the cluster, so that a given application and the GPUs used by that application do not necessarily have to be located at the same cluster node. This evolution adds a lot of flexibility to GPU usage because it detaches GPUs from nodes, thus creating a pool of GPUs that can be concurrently used from any node of the cluster. Figure 1 depicts this idea. In Figure 1 (a) a cluster composed of n nodes is shown, each node containing two Xeon processors and one NVIDIA Tesla GPU. This is a common configuration in computer deployments although, depending on the exact cluster configuration, GPUs may be present only at some of the nodes.

[1]Notice that current GPUs spend very little energy in the idle state. However, when a GPU is assigned to an application, the GPU immediately starts spending a non-negligible amount of energy even if the GPU is not performing any computation. In this regard we differentiate among the idle state (almost no energy consumption) and the active but non-computing state, where the GPU actually may require up to 50W depending on the exact GPU model.

Figure 1 (b) shows the new cluster envision after applying the remote GPU virtualization mechanism. In the new cluster configuration, GPUs are logically detached from nodes and a pool of GPUs is created. GPUs in this pool can be accessed from any node in the cluster. Furthermore, a given GPU may concurrently serve more than one application. This sharing of GPUs not only increases overall GPU utilization but also allows to create cluster configurations where not all the nodes in the cluster own a GPU but all the nodes in the cluster can execute GPU-accelerated applications. This cluster configuration would reduce the costs associated with the acquisition and later use of GPUs. In this regard, the total energy required to operate a computing facility may be decreased, thus loosening the big energy concerns of future exascale computing installations, for instance.

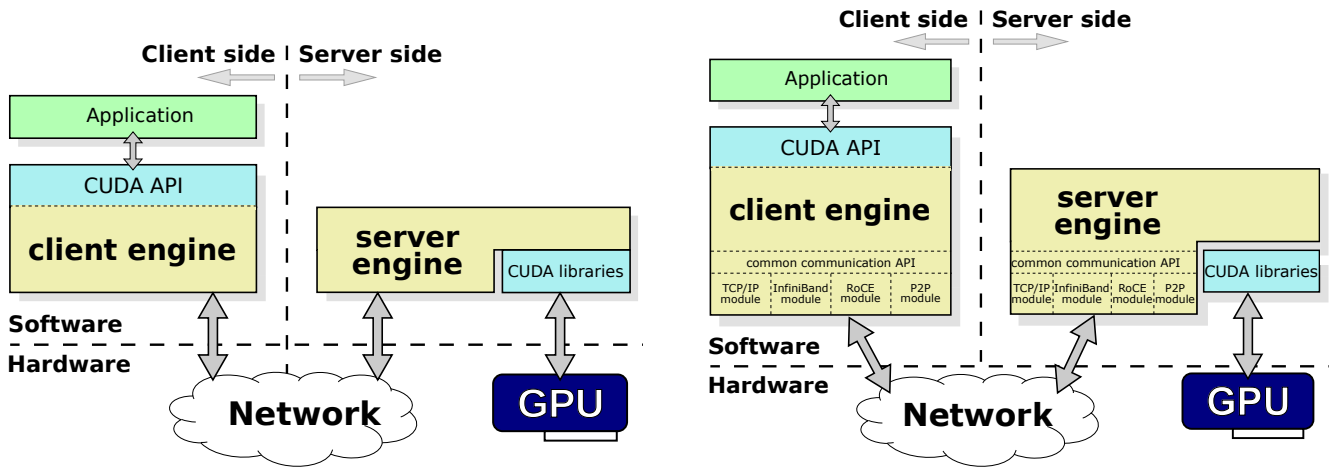
The idea shown in Figure 1 (b) is referred to as remote GPU virtualization because the GPU is not local to the node that executes the application but the GPU is now located in a remote node (typically within the same cluster for the sake of performance). Obviously, in this new GPU virtualization flavor, the performance of the underlying network connecting the application and the remote GPU is key in order to reduce the overhead of the GPU virtualization framework. Additionally, the exact characteristics of the remote GPU are also very important in order to reduce the mentioned overhead because different GPUs must be utilized in a different way by the remote GPU virtualization framework.

In this paper we analyze the performance of remote GPU virtualization over three different generations of NVIDIA GPUs, namely the Tesla K20, the Tesla K40, and the Tesla P100 GPUs. Given that these GPUs feature different PCIe versions, we augment our analysis by using the appropriate InfiniBand technology for each of these GPUs. In this regard, we make use of the FDR and EDR InfiniBand fabrics. The purpose of this study is to analyze how the internal configuration and also overall performance of the remote GPU virtualization technique is impacted by the underlying hardware. Notice that a preliminary version of this analysis has already been published [18]. After that thorough analysis, we then present how we have redesigned the rCUDA communication layer in order to improve the management of the underlying hardware. Furthermore, in this paper we also analyze how those improvements in attained data copy bandwidth translate int a reduction in the execution time of applications.

The rest of the paper is structured as follows. Section 2 presents a revision of different GPU virtualization solutions. That section also presents a summary of the rCUDA remote GPU virtualization middleware, which is the one used in this work to carry out the comparative performance analysis. Next, Section 3 briefly introduces the main characteristics of the three GPU generations used in this analysis as well as the accompanying InfiniBand networks. Then, Section 4 presents the first contribution of this paper by analyzing the performance of rCUDA in the context of the three GPU generations presented in Section 3. Later, Section 5 presents the second contribution of this paper: a redesigned communication layer for rCUDA, comparing the performance to the previous version. Finally, Section 6 introduces the most important conclusions of this work and also outlines future work that will be carried out in regard to the conclusions obtained in this paper.

2 | BACKGROUND ON REMOTE GPU VIRTUALIZATION SOLUTIONS

GPU virtualization can be addressed both from a hardware approach and from a software perspective. With respect to the hardware approach, there have been several recent achievements in order to virtualize GPUs, like the new GRID GPUs by NVIDIA [19], which can be shared among up to 64 virtual machines running in the same cluster node where the GPU is installed (remote GPU access is not allowed). It is important to remark



(a) General organization of remote GPU virtualization frameworks.

(b) Organization of the rCUDA middleware.

FIGURE 2 Default architecture of remote GPU virtualization solutions. A detail of the internal of rCUDA is shown.

that sharing an NVIDIA GRID GPU is only intended for desktop virtualization. In this regard, when a CUDA program is executed, the GPU must be assigned to a single virtual machine in an exclusive way, thus hindering the possibility of concurrently sharing the GPU among several virtual machines.

On the other hand, several software-based GPU sharing mechanisms have appeared, such as, for example, DS-CUDA [16], rCUDA [17], vCUDA [20], GridCuda [21], GVirtuS [15], GViM [22], Shadowfax [23], or Shadowfax II [24]. Basically, these middleware proposals share a GPU by virtualizing it, so that these frameworks provide applications (or virtual machines) 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[2] (CUDA [25] in the case of NVIDIA GPUs). In general, CUDA-based virtualization frameworks aim to offer the same API as the NVIDIA CUDA Runtime API [26] does.

Figure 2 (a) 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 cluster node executing the application requesting GPU services, whereas the server side runs in the node 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. The architecture depicted in Figure 2 (a) 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.

[2] 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.

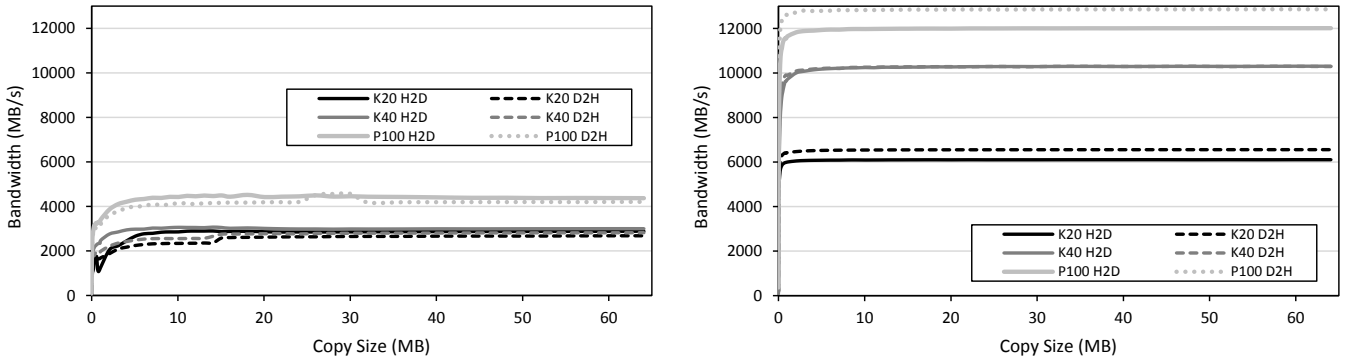
Different GPU virtualization solutions feature different characteristics. For instance, the vCUDA technology, intended for Xen virtual machines, only supports an old CUDA version 3.2 and implements an unspecified subset of the CUDA Runtime API. Moreover, its communication protocol presents a considerable overhead, because of the cost of the encoding and decoding stages, which causes a noticeable drop in overall performance. GViM, also targeting Xen virtual machines, 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 virtual machines, 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 virtual machines 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 a similar way, 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 version 4.1 of CUDA 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.

Regarding rCUDA, this middleware supports version 8.0 of CUDA, the latest available one at the time of writing this paper, 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 API (except for graphics functions) and also provides support for the libraries included within CUDA, such as cuFFT, cuBLAS, or cuSPARSE. rCUDA provides specific support for different interconnects. This 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, as shown in Figure 2 (b). Currently, four modules are available: one intended for TCP/IP compatible networks, another one specifically designed for InfiniBand, which makes use of the RDMA feature of InfiniBand, a third one intended for RoCE networks, which also leverages RDMA features, and a fourth one for peer-to-peer (P2P) memory copies between remote GPUs over InfiniBand fabrics, using RDMA too.

Among the several publicly available remote GPU virtualization frameworks, we used for the performance analysis presented in this paper the rCUDA middleware given that it was the only one able to run the considered benchmarks, as well as being the most up-to-date solution, providing also the best performance when compared to other solutions [27].

3 | BACKGROUND ON GPU AND INFINIBAND HARDWARE USED

This section presents the basic information about the GPUs and network fabrics considered in our analysis. Regarding the GPUs, we have used in our study the NVIDIA Tesla K20, K40, and P100 GPUs. The Tesla K20 GPU [3] comprises 2,496 CUDA cores working at 706 MHz as well as 5 GB of GDDR5 on-board memory providing a bandwidth of 208 GB/s. Additionally, it supports PCI Express Gen 2 x16. On the other hand, the NVIDIA Tesla K40 GPU [2] includes 2,880 CUDA cores working at 745 MHz in addition to 12 GB of GDDR5 memory with a bandwidth of 288 GB/s. The K40



(a) Copies using pageable host memory.

(b) Copies using pinned host memory.

FIGURE 3 Performance (bandwidth) of the Tesla K20, K40 and P100 GPUs. The `bandwidthTest` benchmark from the NVIDIA CUDA Samples was used. "H2D" refers to copies from host to device, while "D2H" refers to copies from device to host.

GPU has a PCI Express Gen 3 x16 system interface. Finally, the Tesla P100 model[1] is the most recent one among the GPUs used in this study. It comprises 3,584 CUDA cores featuring the NVIDIA Pascal architecture. The P100 GPU includes 16 GB of on-package memory providing 732 GB/s and presents a PCI Express Gen 3 x16 interface[3].

Figure 3 presents a bandwidth comparison among the three GPU models. To analyze the bandwidth, we have used the `bandwidthTest` benchmark from the NVIDIA CUDA Samples[28]. This benchmark measures the attained bandwidth when data is moved to/from the GPU (typically referred to as *device* in the CUDA argot). Additionally, we have executed the benchmark with the `shmoo` option, which allows bandwidth to be measured for a large range of memory copy sizes. Both pinned and pageable host memory has been considered as well as moving data in the two possible directions: from host to device (referred to as "H2D" in the figure) and also from device to host ("D2H" in the figure).

It can be seen in the figure that the maximum bandwidth achieved when pageable host memory is involved in the data movement is noticeably lower than the one obtained when using pinned host memory. Moreover, both the K20 and K40 GPUs perform similar when pageable host memory is used. Regarding the use of pinned host memory, the figure shows that the K20 GPU presents the lowest performance because the use of PCIe Gen 2 is considerably limiting maximum bandwidth. On the other hand, although both the K40 and P100 models present a PCIe Gen 3 x16 interface, it can be seen in Figure 3 (b) that NVIDIA has significantly improved performance for the P100 model, although in both cases attained bandwidth is still far away from the maximum theoretical bandwidth of PCIe Gen3 x16, which is 15.75 GB/s.

In addition to compare the three GPU generations from a purely bandwidth perspective, they can also be compared from a computational performance point of view. To that end, we have used the MAGMA suite[29 30]. MAGMA is a dense linear algebra library similar to LAPACK but for heterogeneous architectures. We have utilized release 2.2.0 along with the `dpotrf_gpu` benchmark, which computes the Cholesky factorization for different matrix sizes (from 1K to 10K elements per dimension, in 1K increments).

[3]The Tesla P100 GPU is also available in another form factor which makes use of the NVIDIA NVLink interface, although in this study we have considered the PCI Express version.

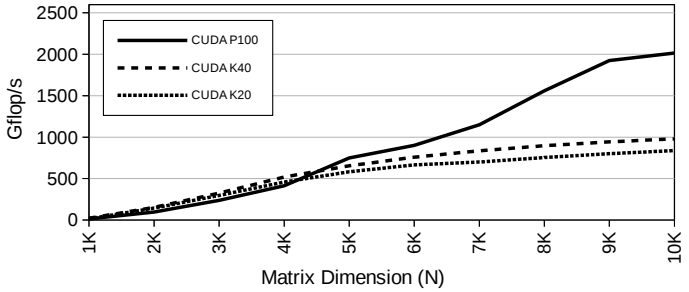


FIGURE 4 Performance (computational power) of the Tesla K20, K40 and P100 GPUs. The `dpotrf_gpu` benchmark from the MAGMA suite was used.

Figure 4 shows the performance (in terms of computational power) of the three GPU models. Computational power is expressed in terms of Gflop/s. The noticeable computing performance increment of the P100 GPU with respect to the K20 and K40 models is clearly shown in the figure. This increment was actually expected, given the higher amount of CUDA cores of the P100 GPU along with the much faster on-package memory.

With regard to the network fabric, Figure 5 depicts the bandwidth attained by both the FDR and EDR InfiniBand interconnects, which theoretically provide 56 Gbps and 100 Gbps, respectively. The figure shows the achieved performance when RDMA write, RDMA read, and send operations are used. The `ib_write_bw`, `ib_read_bw`, and `ib_send_bw` benchmarks from the Mellanox OFED software distribution were used to gather bandwidth measurements.

Notice that InfiniBand adapters provide the InfiniBand Verbs 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 data transfers with minimum involvement of the CPUs. rCUDA employs both semantics, selecting one or the other depending on the exact communication to be carried out.

Figure 5 shows that the maximum bandwidth provided by FDR InfiniBand (which makes use of a PCIe Gen 3 x8 interface) is 6 GB/s, whereas the maximum bandwidth provided by EDR InfiniBand (featuring a PCIe Gen 3 x16 system interface) is over 10 GB/s. As can be seen in Figures 3 (b)

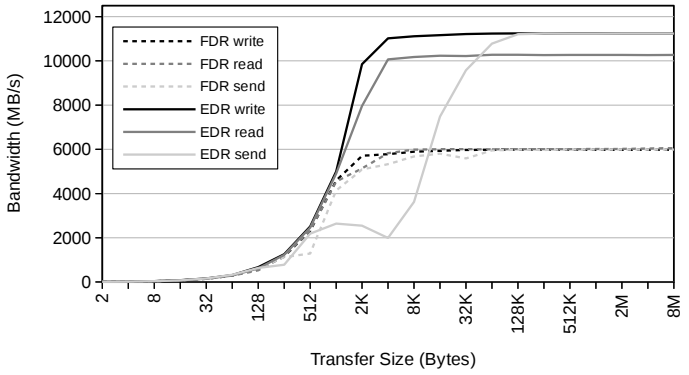


FIGURE 5 Performance (bandwidth) of the FDR and EDR InfiniBand interconnects. The `ib_write_bw`, `ib_read_bw`, and `ib_send_bw` benchmarks from the Mellanox OFED software distribution were used.

and 5 , the Tesla K20 GPU provides a transfer bandwidth similar to that of FDR InfiniBand adapters, while the bandwidth featured by the Tesla K40 and P100 GPUs is close to that of the EDR InfiniBand interconnect. For this reason, in the experiments with rCUDA in the next section, the K20 GPU will be used along with the FDR InfiniBand network fabric, whereas the K40 and the P100 GPUs will be used in conjunction with the EDR InfiniBand adapters.

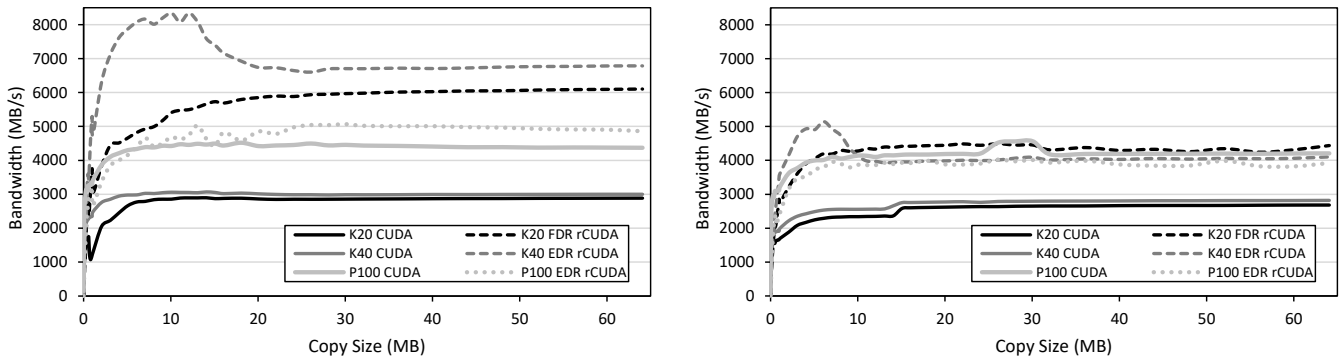
4 | COMPARATIVE PERFORMANCE ANALYSIS

This section section presents the first contribution of this paper: a comparative performance analysis of the rCUDA remote GPU virtualization middleware using the Tesla K20, K40, and P100 GPUs over the FDR and EDR InfiniBand fabrics. The comparative analysis will be carried out in terms of attained bandwidth between the client and server sides of the rCUDA middleware as well as in terms of application performance.

The experiments shown in this section have been done using three pairs of nodes. The first pair of nodes are equipped with the FDR InfiniBand fabric. One of these two nodes owns a Tesla K20 GPU. In a similar way, the second pair of nodes include EDR InfiniBand adapters and one Tesla K40 GPU. The third pair of nodes is similar to the second one, although it comprises a Tesla P100 GPU instead of the K40 one. All the nodes are 1027GR-TRF Supermicro servers featuring two Intel Xeon E5-2620v2 processors (Ivy Bridge) operating at 2.1 GHz and 32 GB of DDR3 memory at 1600 MHz. Linux CentOS 7.3 was used along with CUDA 8.0.

4.1 | Impact on attained bandwidth

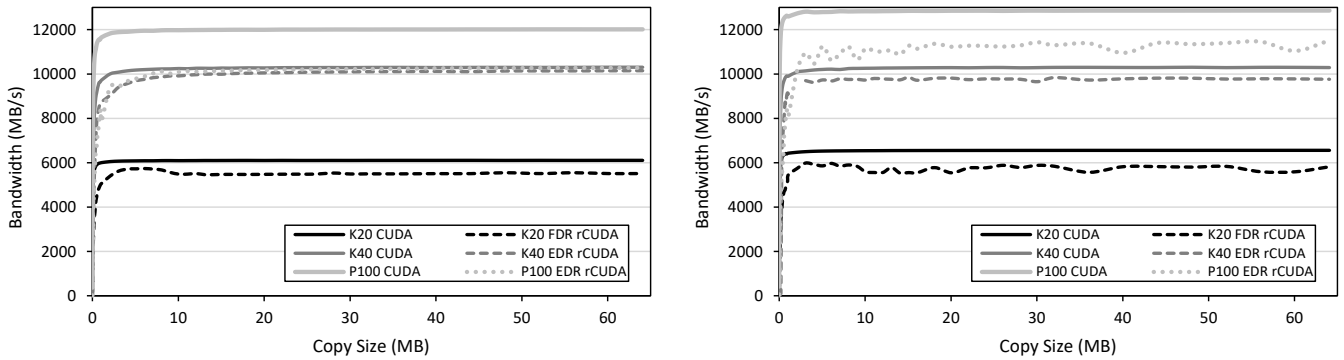
Figure 6 and Figure 7 present the bandwidth attained between the client and server sides of the rCUDA middleware. The bandwidth test from the NVIDIA CUDA Samples was executed in the node without GPU whereas the node with the GPU was used as the remote GPU server. Figure 6 (a) shows attained bandwidth when data stored in pageable RAM memory in the client node is moved to GPU memory in the remote node, while Figure 6 (b) presents the reverse direction. Results using pinned host memory are shown in Figures 7 (a) and 7 (b), respectively.



(a) Host-to-device transfers using pageable host memory.

(b) Device-to-host transfers using pageable host memory.

FIGURE 6 Performance (bandwidth) of the rCUDA framework when used with the Tesla K20, K40, and P100 GPUs. Data is transferred between the client node pageable host memory and the remote node device memory.



(a) Host-to-device transfers using pinned host memory.

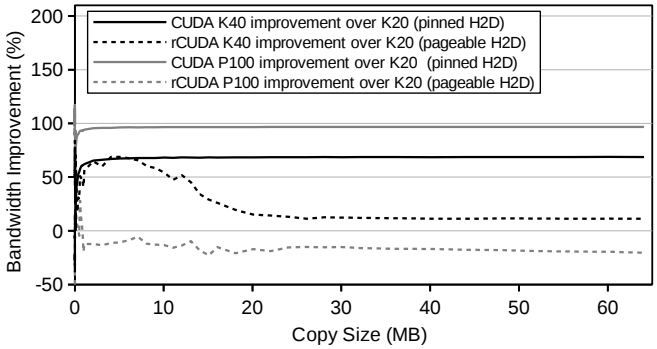
(b) Device-to-host transfers using pinned host memory.

FIGURE 7 Performance (bandwidth) of the rCUDA framework when used with the Tesla K20, K40, and P100 GPUs. Data is transferred between the client node pinned host memory and the remote node device memory.

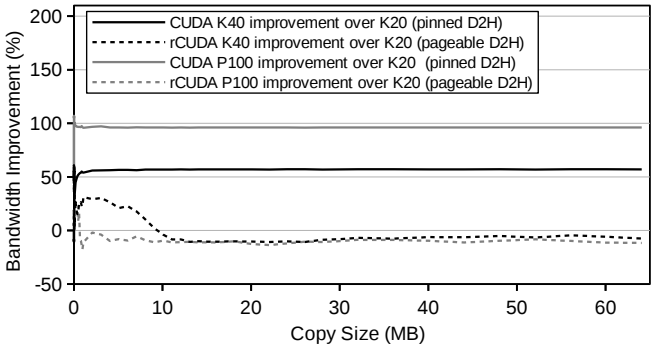
It can be seen in Figure 6 (a) that the bandwidth achieved when using rCUDA over the FDR InfiniBand fabric with a remote Tesla K20 GPU is twice the bandwidth obtained in the traditional CUDA case when the benchmark is executed in the same node where the GPU is located. The reason for this extraordinary performance improvement of rCUDA over CUDA is due to the internal use within rCUDA of an efficient communications pipeline based on the use of pinned memory [31]. This kind of memory presents higher bandwidth, as previously shown in Figure 3 (b). Actually, it can be seen that bandwidth shown in Figure 6 (a) for rCUDA, which is based in the internal usage of pinned memory, is the same as for CUDA when pinned memory is leveraged, as shown in Figure 3 (b). Additionally, Figure 5 shows that the bandwidth obtained also matches the maximum bandwidth provided by the FDR InfiniBand interconnect.

Regarding the Tesla K40 GPU, it can be seen in Figure 6 (a) that rCUDA attains much more bandwidth than CUDA, as in the previous case for the Tesla K20 GPU. However, in this case the maximum bandwidth achieved is far away from the maximum bandwidth obtained when pinned memory is used for the K40 GPU, as shown in Figure 3 (b). More precisely, in the case of the K40 GPU, rCUDA attains a bandwidth of 7 GB/s whereas CUDA achieves a bandwidth of 10 GB/s for pinned memory in Figure 3 (b). Additionally, the interconnect between the client and the server sides, EDR InfiniBand, is not a limiting factor for the bandwidth obtained because, as shown in Figure 5 , EDR InfiniBand provides a bandwidth higher than 10 GB/s. In conclusion, the reason for the relatively lower bandwidth when using a remote K40 GPU with respect to the case of using a remote K20 GPU must be in the internal design of the rCUDA framework.

Figure 6 (a) also shows the attained bandwidth when moving data stored in pageable RAM in the client to the memory of a remote P100 GPU. In this case, maximum bandwidth is even lower than the one obtained with the previous GPU models (Tesla K20 and K40 GPUs) despite still using EDR InfiniBand and despite having more bandwidth in the P100 GPU when pinned memory is moved to/from the device (see Figure 3 (b)). This noticeably lower bandwidth clearly points out that the internal architecture of the rCUDA middleware does not properly scale with the increasingly higher bandwidths provided by the underlying hardware.



(a) Host-to-device transfers using pageable host memory with rCUDA, and pinned host memory with CUDA.



(b) Device-to-host transfers using pageable host memory with rCUDA, and pinned host memory with CUDA.

FIGURE 8 Performance (bandwidth) improvement of CUDA and rCUDA when using the Tesla K40 and P100 GPUs with respect to the Tesla K20 GPU.

In transfers from pinned host memory to device memory, however, the bandwidth obtained seems not to be limited by the middleware, and results are close to the maximum bandwidth of the underlying hardware, as shown in Figure 7 (a). The same happens in the reverse direction, device-to-host copies using pinned host memory, as can be observed in Figure 7 (b).

When data is copied from the remote GPU to the client node using pageable host memory, Figure 6 (b), it can be seen that the internal design of the rCUDA middleware again becomes a clear limiting factor of the performance given that attained bandwidth is much lower than the bandwidth of the interconnect (FDR or EDR InfiniBand, shown in Figure 5).

To better reflect the loss in scalability detected in rCUDA when using pageable host memory, Figure 8 shows the bandwidth improvement of using the Tesla K40 and P100 GPUs with respect to the Tesla K20 GPU. As explained before, rCUDA internally uses pinned host memory to perform this kind of copies. For that reason, in the case of CUDA the figure shows the bandwidth attained when using pinned host memory. As we can observe, when using CUDA the bandwidth improves with each new generation of GPU. On the contrary, rCUDA performance is reduced in spite of having more bandwidth available. Therefore, the internal communications architecture of the rCUDA middleware seems to be limiting the improvement.

As commented in Section 2, rCUDA has four different communication modules: one for TCP/IP compatible networks, another one for InfiniBand networks, a third one for RoCE networks, and a fourth one for peer-to-peer (P2P) memory copies between remote GPUs also over InfiniBand fabrics. In the performance tests previously presented, we were using the module for InfiniBand networks. Despite there is no performance loss with respect to CUDA, as shown in Figure 6 , we have seen that it is possible to achieve more bandwidth (see Figure 8). In order to determine if the limiting factor is actually in the rCUDA communications module, Figure 9 presents performance results when using the rCUDA communications module for P2P copies.

Figure 9 shows the bandwidth of CUDA and rCUDA for the Tesla K20, K40, and P100. In the experiments with CUDA, the two GPUs involved in the data transfer are located in the same server while in the tests with rCUDA, each GPU is located in a different remote server. In the case of the Tesla K20 and K40, it can be seen that the bandwidth attained by rCUDA is close to the maximum one provided by the underlying hardware, 6 GB/s

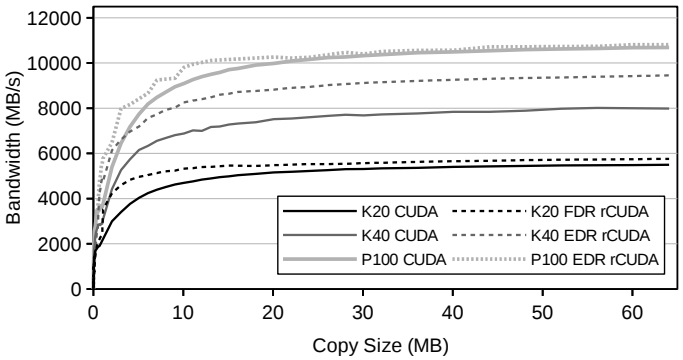


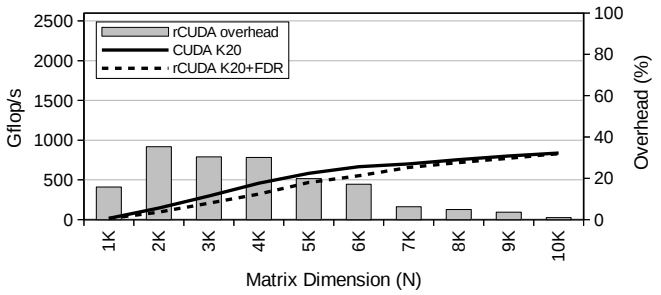
FIGURE 9 Bandwidth obtained for different transfer sizes when data is transferred between remote GPUs located at different cluster nodes as the consequence of executing a P2P memory copy CUDA function.

and 10 GB/s, respectively (see Figure 3 (b)). In the case of the Tesla P100 (offering 12 GB/s), the hardware limiting the bandwidth is not the GPU, as it was in the case of the Tesla K20 and K40. The limiting factor here is the network card, 11 GB/s, (see Figure 5). In this way, the loss in scalability shown in Figure 8 when using rCUDA is not present for P2P memory copies. As commented, the reason is that this kind of copies between GPUs employs a different internal communication architecture within rCUDA, described in further detail in [32].

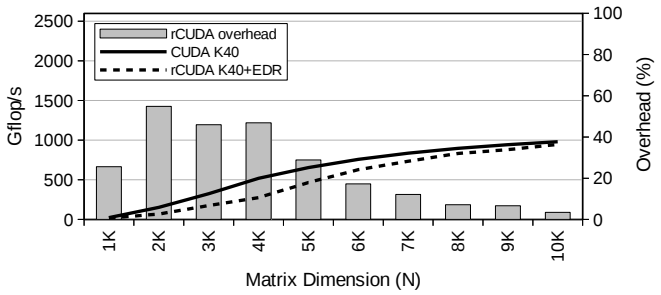
4.2 | Impact on application performance

In order to analyze the impact on application performance of the three generations of GPUs, we have used again the same Cholesky factorization benchmark from the MAGMA suite previously employed in Section 3. Figure 10 presents the execution performance (in Gflop/s) for different matrix sizes. Figure 10 (a) shows the Gflop/s achieved by this benchmark for the Tesla K20 GPU. Both CUDA and rCUDA (over an FDR InfiniBand interconnect) have been considered. The figure also shows the overhead of rCUDA with respect to CUDA. It can be seen that overhead decreases as matrix size increases. This is mainly due to the fact that kernel execution time in the GPU increases its significance with respect to data movement to/from the GPU as matrix size increases.

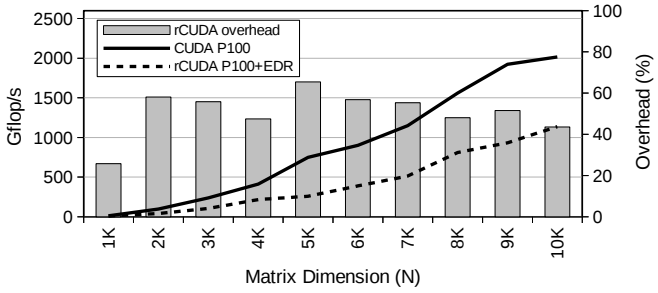
Figure 10 (b) shows the execution results and overhead when a Tesla K40 GPU is used. The overhead in this case follows the same trend as in the case of the Tesla K20 GPU. On the contrary, this tendency is not followed when the Tesla P100 GPU is used, as depicted in Figure 10 (c). This figure shows that overall overhead of rCUDA with respect to CUDA is noticeably increased. The reason is two fold: on the one hand, kernel execution times are reduced because of the larger amount of CUDA cores of the P100 GPU. This reduction in the execution time causes that the communications time has more weight than in the previous cases for the Tesla K20 and K40 GPUs. On the other hand, given that effective bandwidth achieved by rCUDA for the P100 GPU is lower than in the previous cases, the larger contribution to overhead of communications because of the reduction of kernel execution time is additionally exacerbated by a longer communication time due to the smaller effective bandwidth. As a consequence, performance of the MAGMA Cholesky factorization benchmark drops when executed with rCUDA on a remote Tesla P100 GPU.



(a) Using a Tesla K20 GPU with both CUDA and rCUDA over FDR InfiniBand.



(b) Using a Tesla K40 GPU with both CUDA and rCUDA over EDR InfiniBand.



(c) Using a P100 GPU with both CUDA and rCUDA over EDR InfiniBand.

FIGURE 10 Performance comparison when running MAGMA `dpotrf_gpu` test with CUDA and rCUDA for the three GPUs under study. Primary Y-axis shows Gflop/s and secondary Y-axis rCUDA overhead with respect to CUDA. Lines refer to values in the primary Y-axis (i.e., Gflop/s), while bars refer to values in the secondary Y-axis (i.e., overhead).

5 | COMMUNICATION LAYER REDESIGN

In the previous section we have seen that the communication layer of the rCUDA middleware must be properly redesigned in order to obtain the maximum performance of each hardware generation. In this manner, we have found out that there is still room for improvement in the communication layer. This section presents the second contribution of this paper: a redesigned communication layer for rCUDA.

After redesigning the communications layer within rCUDA, a comparative analysis will be carried out in terms of attained bandwidth between the client and server sides of the rCUDA middleware as well as in terms of application performance. For the experiments in this section, we have used the same hardware as in the previous section. The performance results will be compared to the ones of the previous communication layer.

5.1 | The new communication layer

In the previous section we have seen that the current communication layer obtained very good performance when using pinned host memory, as shown in Figure 7 . In this regard, in this kind of copies the attained bandwidth was near to the available bandwidth of the underlying hardware. On the contrary, when using pageable host memory (see Figure 6), the bandwidth obtained seemed to be limited by the rCUDA middleware and results were far from the maximum bandwidth of the underlying hardware.

Furthermore, this behavior seemed to be exacerbated by the use of new hardware generation, as reflected in Figure 8 . In this way, we have observed that when using CUDA the bandwidth improves with each new GPU generation. However, rCUDA performance is reduced in spite of

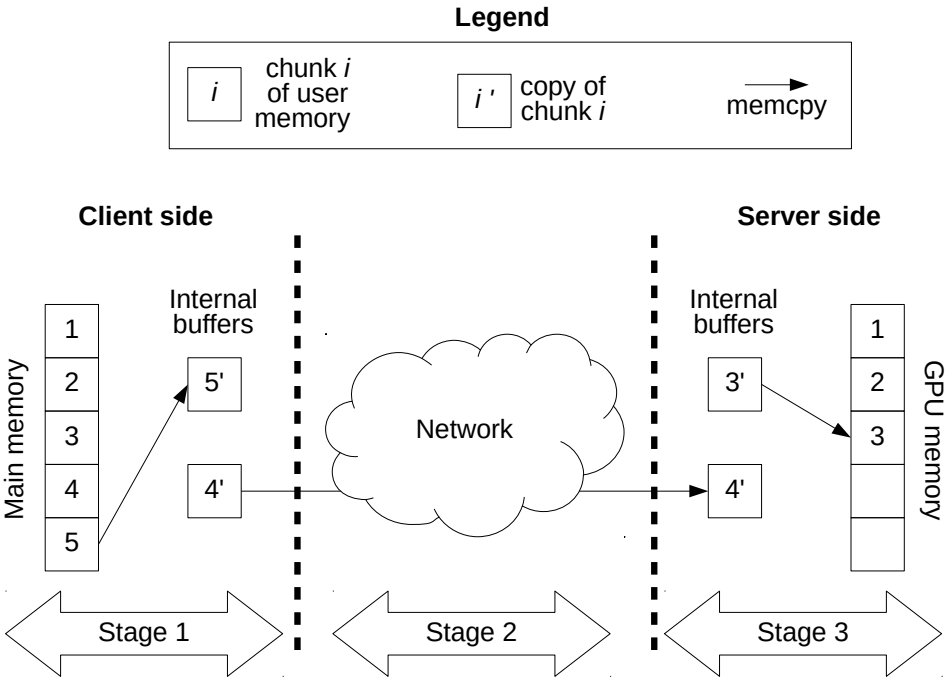


FIGURE 11 Pipeline of the previous communication layer.

having more bandwidth available. Therefore, we have concluded that the internal communications architecture of the rCUDA middleware seems to be limiting the improvement.

As already explained, rCUDA internally uses pinned host memory to perform pageable host memory copies. The current communication layer, thoroughly described in previous works on rCUDA [31], implements a pipelined communication based on the use of internal pre-allocated pinned memory buffers. This pipeline worked efficiently in previous hardware generations. However, as we have seen, newer generations have newer requirements.

In order to better understand how the pipeline works, Figure 11 shows the three stages of the pipeline involved in the process of transferring data from main memory in the source computer to the GPU memory in the target node:

- Stage 1: copy from main memory to one of the internal pre-allocated pinned memory buffers in the client side
- Stage 2: copy from one of the internal pre-allocated pinned memory buffers in the client side to one in the server side
- Stage 3: copy from one of the internal pre-allocated pinned memory buffers in the server side to the GPU memory

After carefully analyzing the pipeline of the previous communication layer shown in Figure 11 , we found that it was limited in the following way: the number of internal pre-allocated pinned memory buffers used in the pipeline for copying pageable host memory was fixed and hard-coded. This was not the case for the size of the internal buffers, which was configurable by the user.

In addition, we detected that both the optimal number and the size of the internal memory buffers presented different values depending on the total size of the data to be transferred. In this manner, it was required for these values to be configurable and also adaptive to the copy size. To overcome these limitations it was necessary to completely redesign and re-implement the communication layer, as the initial code did not considered the aforementioned limitations in its design.

After that process, the new communication layer includes the following improvements: (i) variable number of internal buffers and (ii) dynamically adaptive number and size of the internal memory buffers. The first one allows the number of internal pre-allocated pinned memory buffers used in the pipelined for copying pageable host memory to be variable and configurable by the user through environment variables.

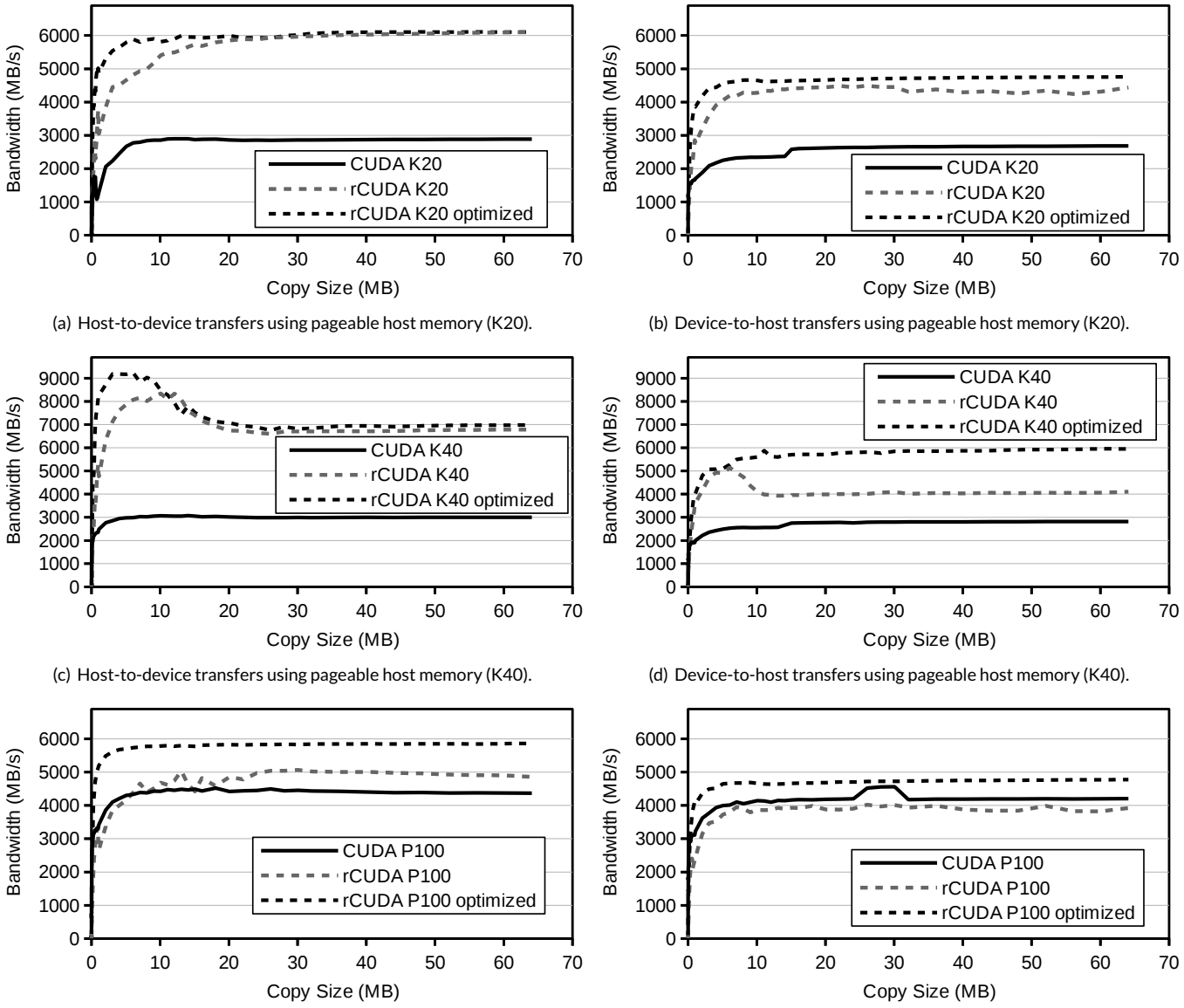
The second improvement permits the number and size of the internal pre-allocated pinned memory buffers used in the pipelined for copying pageable host memory, a part from being variable and configurable by the user, to be also defined taking into account the copy size. In this manner, it is possible for the user to define the number of buffers to be used in terms of copy size ranges. For instance, it is possible to use one number of memory buffers for copy sizes between 1 byte and 1 Kilobyte, a different number of memory buffers for copy sizes between 1 Kilobyte and 1 Megabyte, and so on. Notice that the copy size ranges can also be configured by the user. The same improvement applies to the size of the internal buffers, which can also be configured depending on the transfer size.

Notice that for implementing this second improvement it is also necessary that the new communication layer dynamically adapts the number and the size of the internal memory buffers depending on the size of the data to be copied. By default, it uses pre-defined values, but as previously mentioned it is possible for the user to easily modify those values through environment variables. This allows for a finer tuning of the pipeline depending on the underlying hardware.

5.2 | Impact on attained bandwidth

Figure 12 presents the bandwidth attained by rCUDA when using the new communication layer (labeled as "rCUDA optimized") compared to the previous results shown in Figure 6 , which are labeled simply as "rCUDA". CUDA bandwidth is also shown as a reference. The same hardware as in previous sections is used. Only results for transfers using pageable host memory are shown, as the aim of the new communication layer is improving this kind of copies. As already explained, transfers using pinned host memory already attained almost maximum performance, as shown in Figure 7 .

As we can observe, when using rCUDA over the FDR InfiniBand fabric with a remote Tesla K20 GPU for host-to-device transfers, Figure 12 (a), the bandwidth is specially improved for small and medium copy sizes (up to 20 MB), with an average improvement of 4.97%. For larger copy sizes (over 20 MB), the bandwidth achieved by the optimized version is similar to the previous one. This improvement is possible thanks to the new feature included in the new communication layer which allows to dynamically adapt the number and the size of the internal memory buffers. In the previous version of rCUDA, however, those values where fix for all copy sizes, so they were usually chosen to obtain the maximum bandwidth for large copy sizes. With the new version, we can select different number and size of the internal buffers and use the optimal value for each copy size. Similar conclusions can be obtained when using rCUDA over the EDR InfiniBand fabric with a remote Tesla K40 GPU for host-to-device transfers, Figure 12 (c), presenting an average improvement of 3.86%.



(a) Host-to-device transfers using pageable host memory (K20).

(b) Device-to-host transfers using pageable host memory (K20).

(c) Host-to-device transfers using pageable host memory (K40).

(d) Device-to-host transfers using pageable host memory (K40).

(e) Host-to-device transfers using pageable host memory (P100).

(f) Device-to-host transfers using pageable host memory (P100).

FIGURE 12 Optimized performance (bandwidth) of the rCUDA middleware when used with the Tesla K20, K40, and P100 GPUs. Data is transferred between the client node host memory and the remote node device memory.

In the case of host-to-device transfers over the EDR InfiniBand fabric with a remote Tesla P100 GPU, Figure 12 (e), the bandwidth is increased not only for small and medium copy sizes, but also for large ones, obtaining an average improvement of 21.47%. As commented in previous sections, the bandwidth loss was more notorious when using newer hardware generations, such as the P100 GPU.

Regarding the opposite copy direction, device-to-host transfers, bandwidth improvement is even higher, as this kind of copies is more sensitive to the number and size of the internal buffers. In fact, for configuring this kind of copies, we have defined more copy size ranges than before, with different number and different size of internal buffers for these ranges in order to achieve the maximum performance for each copy size. The average

TABLE 1 Average bandwidth improvement obtained in the results shown in Figure 12 .

Scenario	H2D Copies	D2H Copies
FDR + K20	4.97%	9.44%
EDR + K40	3.86%	29.43%
EDR + P100	21.47%	20.67%
Average	10.10%	19.85%
Average bandwidth improvement: 14.97%		

improvement attained was again higher when using newer hardware. Thus, when using FDR and the K20, Figure 12 (b), the average improvement was 9.44%. When using EDR and the K40, Figure 12 (d), the average improvement was 29.43%. Lastly, when using EDR and the P100 GPU, Figure 12 (f), the average improvement was 20.67%.

As a summary, Table 1 shows the average bandwidth improvement obtained in the results shown in Figure 12 . As we have mentioned, the improvement is higher for device-to-host transfers than for host-to-device ones, presenting the former an average bandwidth improvement for all the scenarios considered in the analysis of 19.85%, while the average improvement in the latter is 10.10%. Finally, the average bandwidth improvement obtained considering both copy directions is 14.97%.

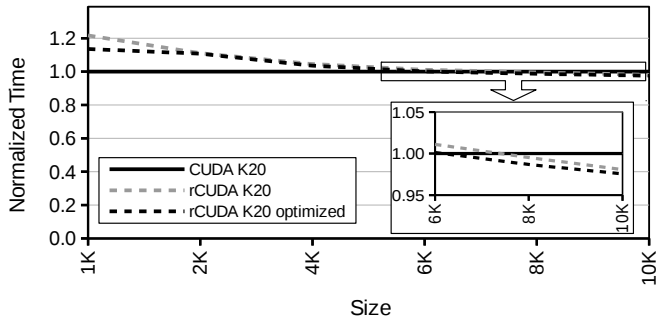
5.3 | Impact on application performance

This section compares the previous and the new communication layer in terms of application performance. For that purpose, we have selected two applications which mainly transfer data using pageable host memory (the kind of transfers improved by the new communication layer). The reason behind this selection is to show the actual impact on application performance of the high increase in bandwidth shown in the previous version when using the new communication layer. Notice that the MAGMA application used in previous sections mainly transfers data using pinned host memory. As mentioned before, this kind of copies already achieved near to maximum performance with the previous communication layer (see Figure 7). Therefore, results for the MAGMA application are similar as the ones shown in Figure 10 regardless of using the previous or the new communication layer. For this reason, results for MAGMA have been omitted in this section.

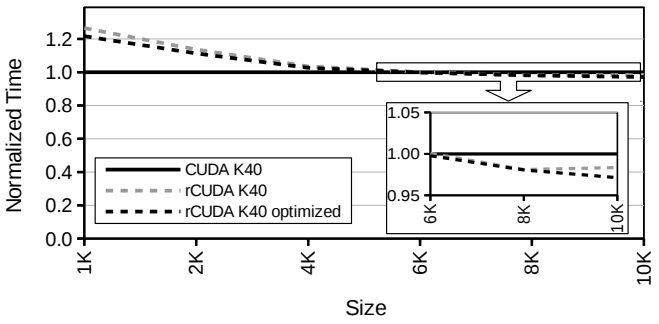
5.3.1 | cuBLAS

The NVIDIA CUDA Basic Linear Algebra Subroutines library (cuBLAS) [33] is a fast GPU-accelerated implementation of the standard Basic Linear Algebra Subprograms (BLAS) [34] routines, which provide standard building blocks for performing basic vector and matrix operations. In this section we have used an application using the cuBLAS library for comparing the performance of the new and the previous communication layers.

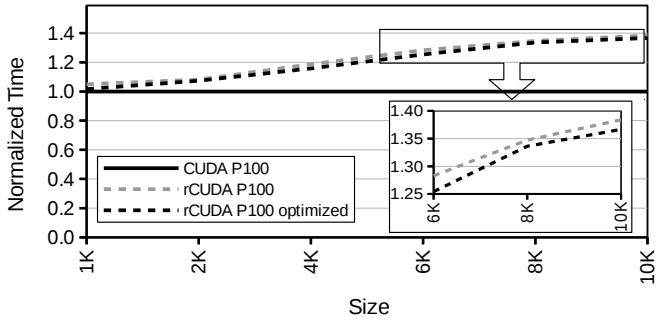
Figure 13 shows the normalized execution time of these experiments for the optimized communication layer, labeled as "rCUDA optimized", compare to the one for the previous communication layer, labeled as "rCUDA". Results for both of them are normalized with respect to the execution time using CUDA and a local GPU. As we can observe, the increase in performance provided by the new communication layer when using the FDR InfiniBand fabric with a remote Tesla K20 GPU (see Figure 13 (a)) is higher for short/medium problem sizes than for large ones. The same happens



(a) Using a Tesla K20 GPU with both CUDA and rCUDA over FDR InfiniBand.

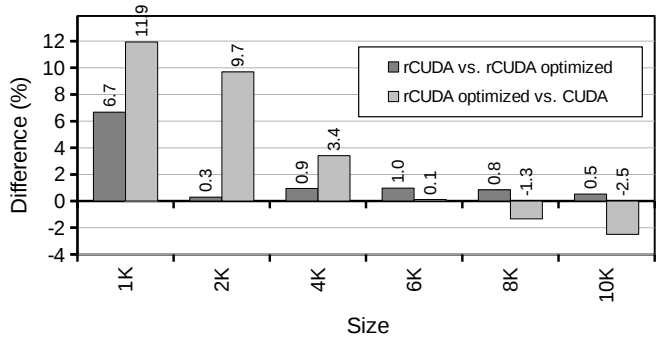


(b) Using a Tesla K40 GPU with both CUDA and rCUDA over EDR InfiniBand.

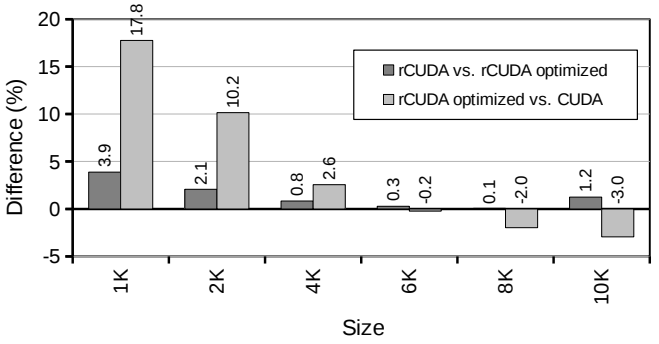


(c) Using a Tesla P100 GPU with both CUDA and rCUDA over EDR InfiniBand.

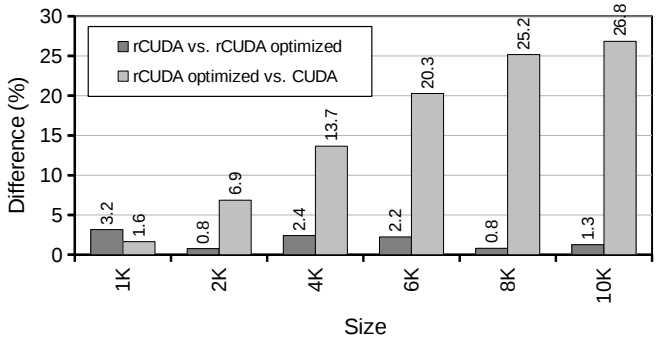
FIGURE 13 Performance comparison when running cuBLAS test with CUDA and rCUDA for the three GPUs under study.



(a) Using a Tesla K20 GPU with both CUDA and rCUDA over FDR InfiniBand.



(b) Using a Tesla K40 GPU with both CUDA and rCUDA over EDR InfiniBand.



(c) Using a Tesla P100 GPU with both CUDA and rCUDA over EDR InfiniBand.

FIGURE 14 Performance comparison when running cuBLAS test with CUDA and rCUDA for the three GPUs under study.

when using the EDR InfiniBand fabric with a remote Tesla K40 GPU, as shown in Figure 13 (b). When using the EDR InfiniBand fabric with a remote Tesla P100 GPU (see Figure 13 (c)), the improvement of using the new communication layer is more regular.

To better analyze these results, Figure 14 presents the percentage execution time difference of using the previous communication layer compared to the new one, bars labeled as "rCUDA vs. rCUDA optimized". As it can be seen, the previous communication layer of rCUDA requires on average 1.71%, 1.39% and 1.77% more execution time for the three scenarios under analysis FDR + K20, EDR + K40 and EDR + P100, respectively. As it was the case for the bandwidth, the scenario involving the newest hardware, EDR InfiniBand and Tesla P100, presents the highest improvement. As expected, the impact of improving the bandwidth on a real application is reduced by the fact that the main task of real applications is not transferring data to the GPU. Therefore, the weight of data transfers in a real application is usually very low in comparison to oder tasks such as doing computations in the GPU.

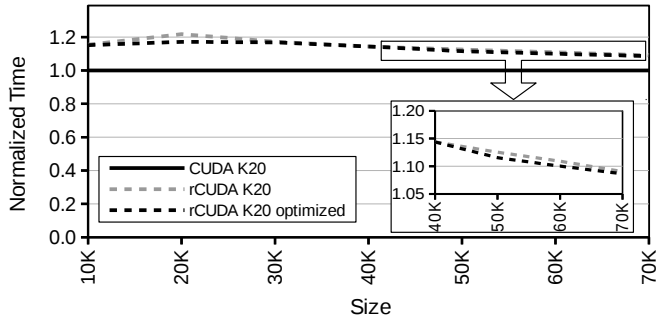
Figure 14 also shows the percentage difference of using rCUDA with the new communication layer compared to using CUDA and a local GPU, bars labeled as "rCUDA optimized vs. CUDA". In this case, the average overhead of rCUDA with respect to CUDA for the same three scenarios under analysis is 3.56%, 4.22% and 15.74%, respectively. Notice that in the latter scenario, EDR InfiniBand fabric with a remote Tesla P100 GPU, the overhead is higher than in the other two scenarios. The reason is that in this scenario we have improved the GPU in comparison to the other two scenarios, however, the underlying InfiniBand network has not been improved from EDR to HDR. We expect that results with HDR InfiniBand will result into a lower overhead, comparable to the ones obtained in the other two scenarios.

5.3.2 | cuFFT

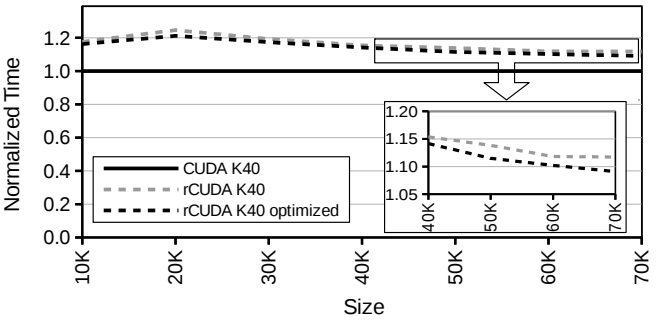
The NVIDIA CUDA Fast Fourier Transform library (cuFFT) [35] provides GPU-accelerated implementations of the Fastest Fourier Transform in the West (FFTW) library [36], which is a software library for computing discrete Fourier transforms (DFT). In this section we have used an application using the cuFFT library for comparing the performance of the new and the previous communication layers.

Figure 15 shows the normalized execution time of these experiments for the optimized communication layer, labeled as "rCUDA optimized", compare to the one for the previous communication layer, labeled as "rCUDA". Results for both of them are normalized with respect to the execution time using CUDA and a local GPU. As we can observe, in these experiments, the increase in performance provided by the new communication layer is similar in each scenario regardless of the copy size. It can also be seen that using the new communication layer provides more benefits as newer hardware is used. Thus, the improvement obtained when using the FDR InfiniBand fabric with a remote Tesla K20 GPU (see Figure 15 (a)) is lower than the one obtained when using the EDR InfiniBand fabric with a remote Tesla K40 GPU (see Figure 15 (b)). The latter, in turn, presents lower improvement than when using the EDR InfiniBand fabric with a remote Tesla P100 GPU (see Figure 15 (c)).

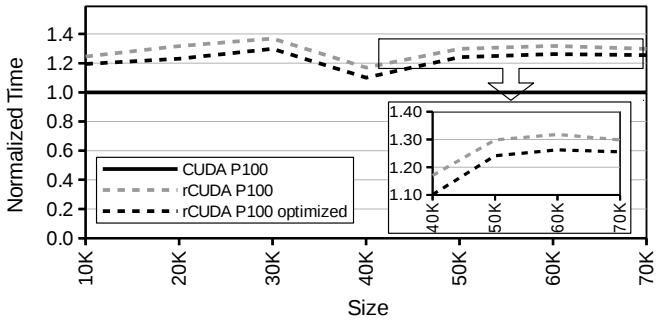
In Figure 16 we can see the percentage execution time difference of using the previous communication layer compared to the new one, bars labeled as "rCUDA vs. rCUDA optimized". The figure shows that the previous communication layer of rCUDA requires on average 0.91%, 1.77% and 4.81% more execution time for the three scenarios under analysis FDR + K20, EDR + K40 and EDR + P100, respectively. As commented, similarly to what happened with the bandwidth, the improvements are more noticeable in the scenarios using newer hardware. Again, as in the case of the



(a) Using a Tesla K20 GPU with both CUDA and rCUDA over FDR InfiniBand.

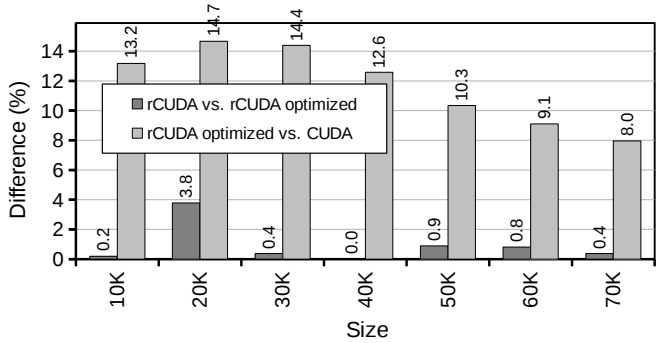


(b) Using a Tesla K40 GPU with both CUDA and rCUDA over EDR InfiniBand.

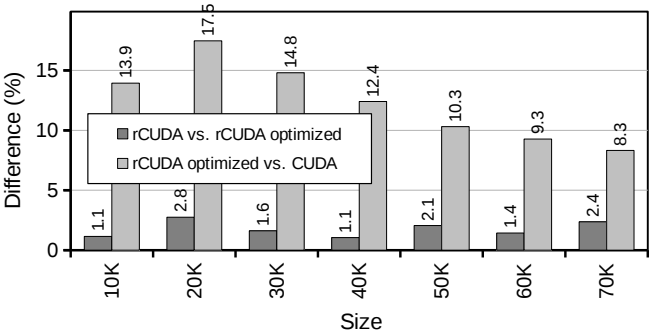


(c) Using a Tesla P100 GPU with both CUDA and rCUDA over EDR InfiniBand.

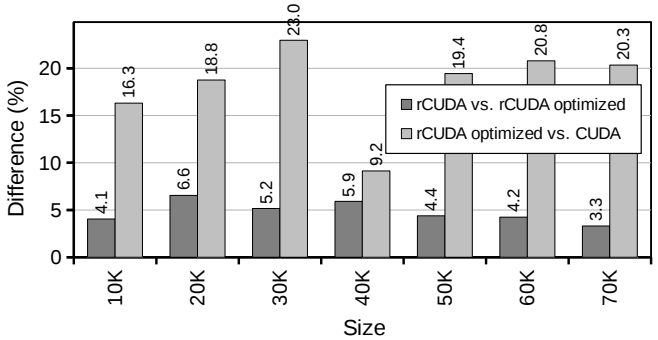
FIGURE 15 Performance comparison when running cuFFT test with CUDA and rCUDA for the three GPUs under study.



(a) Using a Tesla K20 GPU with both CUDA and rCUDA over FDR InfiniBand.



(b) Using a Tesla K40 GPU with both CUDA and rCUDA over EDR InfiniBand.



(c) Using a Tesla P100 GPU with both CUDA and rCUDA over EDR InfiniBand.

FIGURE 16 Performance comparison when running cuFFT test with CUDA and rCUDA for the three GPUs under study.

experiments with cuBLAS presented in the previous section, the improvement achieved is lower than the one shown for the bandwidth. As we have already mentioned, the improvement achieved is not as high as in the case of the bandwidth. The reason is the same as in the cuBLAS experiments presented in the previous section: a real application does more tasks than moving data to and from the GPU memory, such as doing computations in the GPU.

Figure 16 also shows the percentage difference of using rCUDA with the new communication layer compared to using CUDA and a local GPU, bars labeled as "rCUDA optimized vs. CUDA". In this case, the average overhead of rCUDA with respect to CUDA for the same three scenarios under analysis is 11.75%, 12.36% and 18.25%, respectively. In comparison to the experiments with cuBLAS, these values are higher. The reason is that in these experiments, data transfers have more weight than in the cuBLAS ones. This also explains that the new communication layer has obtained better improvement values when compared to the previous communication layer.

5.3.3 | Summary

As a summary, Table 2 shows the average execution time difference obtained in the results shown in Figures 13 and 15 . The table shows that the previous communication layer requires on average 1.62% more execution time for the experiments using cuBLAS. Regarding the experiments using cuFFT the overhead of the previous communication layer is higher, and it needs 2.50% more execution time, on average. The experiments using cuFFT clearly show how the impact of the new communication layer is higher as newer hardware is used.

Table 2 also shows that when using rCUDA with the new communication layer, the average overhead in execution time with respect to CUDA using a local GPU is 7.84% and 14,12%, respectively, for the experiments with cuBLAS and cuFFT. As has already been pointed out, the scenario using the EDR InfiniBand fabric with a remote Tesla P100 GPU presents a higher overhead than in the other two scenarios. As explained before, this is due to the fact that in this scenario we use a newer GPU, the P100, in comparison to the other two scenarios, where we used a K20 and a K40. However, the InfiniBand network is the same as in the scenario with the K40: EDR InfiniBand. We expect that results using a newer InfiniBand fabric, such as HDR, will result into a lower overhead, comparable to the ones obtained in the other two scenarios.

It should also be noted that the overhead of rCUDA in these experiments is relatively high if we compare it to other studies which show that rCUDA overhead is below 5% [17]. As we have mentioned, the applications selected for the experiments in this section mainly transfer data using pageable host memory, being the time devoted to perform computations in the GPU lower than usual. While this behavior is the best one to show the impact of using the new communication layer, it is also the worst case for rCUDA. In general, the more computations in the GPU, the less rCUDA overhead. The reason is that the time spent in computations is the same for CUDA and rCUDA, and it helps to hide the potential overhead of rCUDA because of accessing the GPU through the network.

TABLE 2 Average execution time difference obtained in the results shown in Figures 13 and 15 .

Scenario	cuBLAS			cuFFT		
	FDR + K20	EDR + K40	EDR + P100	FDR + K20	EDR + K40	EDR + P100
rCUDA vs. rCUDA optimized	1.71%	1.39%	1.77%	0.91%	1.77%	4.81%
rCUDA optimized vs. CUDA	3.56%	4.22%	15.74%	11.75%	12.36%	18.25%

6 | CONCLUSIONS

In this paper we have studied the influence of the underlying hardware in the performance of remote GPU virtualization. To that end, we have used three different generations of GPUs: the Tesla K20, the Tesla K40 and the Tesla P100. In addition, we have also used two different InfiniBand network fabrics: FDR and EDR. The remote GPU virtualization middleware employed in the analysis was rCUDA.

The results have revealed that the communication layer of the GPU virtualization middleware must be properly designed in order to obtain the maximum performance of each hardware generation. In this manner, we have found out that there was still room for improvement in the rCUDA middleware communication layer.

After carefully analyzing the previous communication layer, we have concluded that the pipeline used for the data transfers was limited in the following ways. Firstly, the number of buffers used in the pipelined was fixed and hard-coded. In addition, it was required that both the number and the size of the pipeline buffers were configurable and adaptive depending on the total size of the data to be transferred.

As a way to address these limitations, we have completely redesigned the rCUDA communication layer in order to improve the management of the underlying hardware. Results show that it is possible to improve bandwidth up to 29.43%, which translates into up to 4.81% average less execution time in the performance of the applications analyzed.

ACKNOWLEDGMENTS

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. NVIDIA . NVIDIA Tesla P100; Infinite Compute Power for the Modern Data Center http://images.nvidia.com/content/tesla/pdf/nvidia-tesla-p100-PCIe-datasheet.pdfAccessed 2 April 2017; 2017.
2. NVIDIA . Tesla K40 GPU Accelerator Board Specification http://www.nvidia.com/content/PDF/kepler/Tesla-K40-PCIe-Passive-Board-Spec-BD-06902-001_v05.pdfAccessed 25 March 2017; 2013.

3. NVIDIA . Tesla K20 GPU Accelerator Board Specification http://www.nvidia.com/content/pdf/kepler/tesla-k20-passive-bd-06455-001-v07.pdfAccessed 25 March 2017; 2013.

4. Intel Corporation . Intel Xeon Phi Processors http://www.intel.com/content/www/us/en/products/processors/xeon-phi/xeon-phi-processors.htmlAccessed 25 March 2017; 2017.

5. Guo Kaiyuan, Sui Lingzhi, Qiu Jiantao, et al. From Model to FPGA: Software-Hardware Co-Design for Efficient Neural Network Acceleration. In: Hot Chips: A Symposium on High Performance Chips; 2016.

6. Suda Naveen, Chandra Vikas, Dasika Ganesh, et al. Throughput-Optimized OpenCL-based FPGA Accelerator for Large-Scale Convolutional Neural Networks. In: Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays; 2016.

7. Wu Haicheng, Diamos Gregory, Sheard Tim, et al. Red Fox: An Execution Environment for Relational Query Processing on GPUs. In: Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization:44:44–44:54ACM; 2014.

8. Phillips Everett H., Zhang Yao, Davis Roger L., Owens John D.. Rapid Aerodynamic Performance Prediction on a Cluster of Graphics Processing Units. In: no. AIAA 2009-565 in Proceedings of the 47th AIAA Aerospace Sciences Meeting; 2009.

9. Playne Daniel P, Hawick Kenneth A. Data Parallel Three-Dimensional Cahn-Hilliard Field Equation Simulation on GPUs with CUDA. In: PDPTA:104–110; 2009.

10. Yamazaki Ichitaro, Dong Tingxing, Solca Raffaele, Tomov Stanimire, Dongarra Jack, Schulthess Thomas. Tridiagonalization of a dense symmetric matrix on multiple GPUs and its application to symmetric eigenvalue problems. *Concurrency and Computation: Practice and Experience*. 2014;26(16):2652–2666.

11. Yuancheng Luo Duraiswami. Canny edge detection on NVIDIA CUDA. In: Computer Vision and Pattern Recognition Workshops, 2008. CVPRW '08. IEEE Computer Society Conference on:1–8IEEE; 2008.

12. Surkov Vladimir. Parallel option pricing with Fourier space time-stepping method on graphics processing units. *Parallel Computing*. 2010;36(7):372 - 380. Parallel and Distributed Computing in Finance.

13. Agarwal Pratul K., Hampton Scott, Poznanovic Jeffrey, Ramanthan Arvind, Alam Sadaf R., Crozier Paul S.. Performance modeling of microsecond scale biological molecular dynamics simulations on heterogeneous architectures. *Concurrency and Computation: Practice and Experience*. 2013;25(10):1356–1375.

14. Luo Guo-Heng, Huang Sheng-Kai, Chang Yue-Shan, Yuan Shyan-Ming. A parallel Bees Algorithm implementation on GPU. *Journal of Systems Architecture*. 2014;60(3):271 - 279.

15. Giunta Giulio, Montella Raffaele, Agrillo Giuseppe, Coviello Giuseppe. A GPGPU Transparent Virtualization Component for High Performance Computing Clouds. In: Proc. of the Euro-Par Parallel Processing, Euro-Par:379–391; 2010.

16. Oikawa Minoru, Kawai Atsushi, Nomura Kentaro, Yasuoka Kenji, Yoshikawa Kazuyuki, Narumi Tetsu. 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:1207–1214; 2012.

17. Reaño Carlos, Silla Federico, Shainer Gilad, Schultz Scot. Local and Remote GPUs Perform Similar with EDR 100G InfiniBand. In: Proceedings of the Industrial Track of the 16th International Middleware Conference; 2015.

18. Reaño Carlos, Silla Federico. A Comparative Performance Analysis of Remote GPU Virtualization over Three Generations of GPUs. In: 46th International Conference on Parallel Processing Workshops, ICPP Workshops:121–128; 2017.

19. NVIDIA . NVIDIA Grid Accelerated Virtual Desktops and Apps http://images.nvidia.com/content/grid/pdf/188270-NVIDIA-GRID-Datasheet-NV-US-FNL-Web.pdfAccessed 26 March 2017; 2016.

20. Shi Lin, Chen Hao, Sun Jianhua. vCUDA: GPU accelerated high performance computing in virtual machines. In: Proc. of the IEEE Parallel and Distributed Processing Symposium, IPDPS:1–11; 2009.

21. Liang Tyng Yeu, Chang Yu Wei. GridCuda: A Grid-Enabled CUDA Programming Toolkit. In: Proc. of the IEEE Advanced Information Networking and Applications Workshops, WAINA:141–146; 2011.

22. Gupta Vishakha, Gavrilovska Ada, Schwan Karsten, et al. GViM: GPU-accelerated virtual machines. In: Proc. of the ACM Workshop on System-level Virtualization for High Performance Computing, HPCVirt:17–24; 2009.

23. Merritt Alexander M., Gupta Vishakha, Verma Abhishek, Gavrilovska Ada, Schwan Karsten. Shadowfax: Scaling in Heterogeneous Cluster Systems via GPGPU Assemblies. In: Proc. of the International Workshop on Virtualization Technologies in Distributed Computing, VTDC:3–10; 2011.

24. Shadowfax II - scalable implementation of GPGPU assemblies http://keeneland.gatech.edu/software/keeneland/kidron.htmlAccessed 9 September 2015; 2015.

25. NVIDIA . CUDA C Programming Guide. Design Guide http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdfAccessed 26 March 2017; 2017.

26. NVIDIA . CUDA Runtime API. API Reference Manual http://docs.nvidia.com/cuda/pdf/CUDA_Runtime_API.pdfAccessed 26 March 2017; 2016.

27. Reaño Carlos, Silla F.. A Performance Comparison of CUDA Remote GPU Virtualization Frameworks. In: 2015 IEEE International Conference on Cluster Computing; 2015.

28. NVIDIA . CUDA Samples Reference Manual 8.0 http://docs.nvidia.com/cuda/cuda-samples/Accessed 2 April 2017; 2017.

29. Bosma Wieb, Cannon John, Playoust Catherine. The MAGMA Algebra System I: The User Language. *J. Symb. Comput.*. 1997;24(3-4):235–265.

30. University of Tennessee . MAGMA: Matrix Algebra on GPU and Multicore Architectures http://icl.cs.utk.edu/magmaAccessed 2 April 2017; 2017.

31. Peña Antonio J., Reaño Carlos, Silla Federico, Mayo Rafael, Quintana-Ortí Enrique S., Duato José. A complete and efficient CUDA-sharing solution for HPC clusters. *Parallel Computing*. 2014;40(10):574–588.

32. Reaño Carlos, Silla Federico. Extending rCUDA with Support for P2P Memory Copies between Remote GPUs. In: 18th IEEE International Conference on High Performance Computing and Communications, HPCC:789–796; 2016.

33. NVIDIA . CUDA Basic Linear Algebra Subroutines (cuBLAS) https://developer.nvidia.com/cublasAccessed 23 March 2018; 2018.

34. University of Tennessee . Basic Linear Algebra Subprograms (BLAS) http://www.netlib.org/blas/Accessed 23 March 2018; 2018.

35. NVIDIA . CUDA Fast Fourier Transform library (cuFFT) https://developer.nvidia.com/cufftAccessed 23 March December 2018; 2018.

36. Frigo Matteo, Johnson Steven G.. The Design and Implementation of FFTW3. *Proceedings of the IEEE*. 2005;93(2):216–231. Special issue on "Program Generation, Optimization, and Platform Adaptation".

AUTHOR BIOGRAPHY

Carlos Reaño received a BS degree in Computer Engineering from University of Valencia, Spain, in 2008. He also holds a MS degree in Software Engineering, Formal Methods and Information Systems from Technical University of Valencia, Spain, since 2012, and a PhD in Computer Engineering from the same university since 2017. He is currently a postdoctoral researcher at the Department of Computer Engineering (DISCA) of Technical University of Valencia, where he is working in the rCUDA project (www.rcuda.net) since 2011. His research is mainly focused on the virtualization of remote GPUs. He has published several papers in peer-reviewed conferences and journals, and has also participated as reviewer in some conferences and journals. More information is available from http://mural.uv.es/caregon.

Federico Silla received the MS and Ph.D. degrees in Computer Engineering from Universitat Politenica de Valencia, Spain, in 1995 and 1999, respectively. He is currently an associate professor at the Department of Computer Engineering at that university, although he is credited by the Spanish Government as Full Professor since January 2016. Furthermore, he worked for two years at Intel Corporation, developing on-chip networks. His research addresses high performance on-chip and off-chip interconnection networks as well as distributed memory systems and remote GPU virtualization mechanisms. In this regard, he is the coordinator of the rCUDA remote GPU virtualization project since it began in 2008. He has published more than 100 papers in peer-reviewed conferences and journals, as well as 10 book chapters, what currently translates into an H-index impact factor equal to 26 according to Google Scholar. He has been member of Program Committees in many of the most prestigious conferences in his area, including CCGRID, SC, PACT, ISCA, ICS, MICRO, etc. He is also an Associate Editor of Journal of Parallel and Distributed Computing (JPDC) since 2015. In addition to organize several workshops, he has also been Workshop Chair in the International Conference on Parallel Processing 2017 and 2018. For more information about Federico Silla, please visit http://www.disca.upv.es/fsilla.

How to cite this article: C. Reaño and F. Silla, (2018), Redesigning the rCUDA Communication Layer for a Better Adaptation to the Underlying Hardware, *Concurrency Computat: Pract Exper., 2018;00:1–15*.