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

Tuning Remote GPU Virtualization for InfiniBand Networks

Carlos Reaño · Federico Silla

Received: date / Accepted: date

Abstract In the past few years, a tendency towards using InfiniBand networks to interconnect high performance computing clusters can be observed. Thus, most of the supercomputers appearing in the TOP500 list either use Ethernet or InfiniBand interconnects. Regarding the latter, the complexity of the InfiniBand programming API (i.e., InfiniBand Verbs) makes it difficult for applications to get the maximum performance of these networks. In this paper we expose how we have tuned a remote GPU virtualization framework whose communications module is implemented using InfiniBand Verbs. The net result is a noticeable increase in the performance of this framework, significantly reducing the gap between remote and local GPUs.

Keywords HPC · InfiniBand · CUDA · remote GPU virtualization · networks · performance · tuning

1 Introduction

InfiniBand (IB) [1] is an interconnect providing high bandwidth and low latency, being commonly used in high performance computing (HPC). The high performance attained by InfiniBand makes that its use in supercomputers has considerably increased during the last years [2], as shown in Figure 1. This figure presents the amount of supercomputers in the TOP500 list [3] using the InfiniBand network as well as different versions of the Ethernet one. Although InfiniBand and Ethernet are far apart both in terms of programming models as well as performance, we are also showing Ethernet in this figure for the sake

C. Reaño
Universitat Politècnica de València, València, 46022, Spain
Tel.: +34 96 387 70 07 Ext.75738
E-mail: carregon@gap.upv.es

F. Silla
Universitat Politècnica de València, València, 46022, Spain

of completeness. As it can be seen, the presence of the InfiniBand technology in current supercomputers is even higher than that of Ethernet, having the former a share of 47.4% whereas the latter presents a share of 36.2%. Furthermore, we can observe that the total sum represented by systems based on any of these two interconnect technologies accounts for more than 80% of the systems in this list, what reveals that the InfiniBand technology is the most widely one used in the HPC domain.

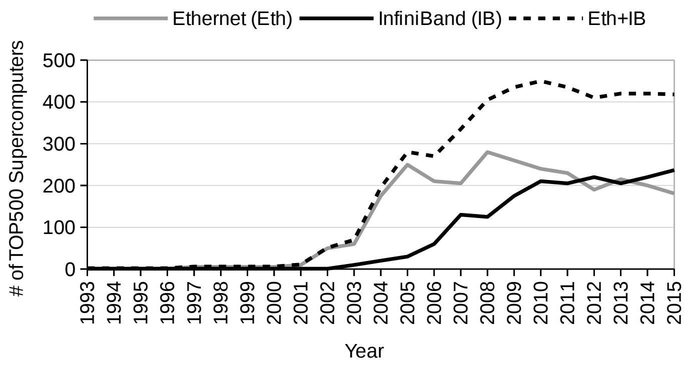


Fig. 1 Presence of Ethernet and InfiniBand in the TOP500 list.

However, a major disadvantage of the InfiniBand network lies in the fact that its specification [4] does not clearly define an API that can be easily learned without attending specific courses. Indeed, it only describes a set of functions, usually referred to as *verbs* (i.e., the InfiniBand Verbs–IBV), which must be available in any commercial product adhering the specification. As a consequence, the lack of such explicit API in conjunction with the complexity of the IBV semantics make it difficult to develop even a simple program. This is evidenced by the publication of papers with the sole purpose of clarifying how to interact with the InfiniBand Verbs, such as [5], [6] or [7], to name only a few.

The net result is that InfiniBand is the most widely used interconnect in the supercomputers included in the TOP500 list, but the lack of documentation makes it difficult to get all the benefits from this network fabric. In this paper we expose how we have tuned a remote GPU virtualization framework, whose communications module is implemented using InfiniBand Verbs.

The rest of the paper is organized as follows. In Section 2, we discuss the work related to our study. Next, in Section 3, we exposed the improvements presented in this paper. Section 4 analyzes the benefits that these enhancements bring to the remote GPU virtualization framework under study. Finally, in Section 5, the main conclusions of this work are presented.

2 Related Work

As commented before, the lack of an easy to understand programming API for InfiniBand is one of the major concerns when developing applications that use this network fabric. Actually, this was what motivated G. Kerr to dissect in [5] a simple *pingpong* program, in an attempt to make clear how to interact with the InfiniBand Verbs API.

In view of this, exploring optimizations for InfiniBand applications has typically remained a big challenge. Several researchers have attempted to present recommendations for achieving optimal performance. One such example is the work by Liu et al. in [8], which presents a recent in-depth analysis of the InfiniBand FDR network, proposing several interesting optimizations. However, the study is limited to the memory semantics (i.e., Remote Direct Memory Access–RDMA), not addressing the channel semantics (i.e., send/receive verbs no using RDMA). Additionally, the tests are done using Sandy Bridge processors, while results over later generation processors (i.e., Ivy Bridge) could lead to different results, and maybe to different conclusions.

Other researchers have also presented improvements in recent studies. For instance, Subramoni et al. in [9] study the benefits of using the new Dynamically Connected (DC) InfiniBand transport protocol, showing great improvements for both synthetic benchmarks and production applications. Another such example can be found in the work by Wang et al. in [10], where it is proposed a tuned GPU to GPU communication design for InfiniBand clusters. Nevertheless, although these proposals improve performance, both of them are focused on tuning MPI libraries, whose requirements differ from general applications.

Most recently, it has been introduced UCX [11] (Unified Communication X), a collaboration between industry, laboratories, and academia to create an open-source production grade communication framework for data centric and high-performance applications. When this technology becomes well-establish and mature, it would be interesting to study if the new UCX API could solve part of the difficulties to use the InfiniBand low level interface, while maintaining the performance.

From our point of view, all these previous efforts to tune InfiniBand environments will benefit from the work presented in this paper, as the improvements here exposed can be applied to all those fields.

3 Tuning InfiniBand Bandwidth

In this section we introduce and analyze the enhancements proposed in this work. The setup used for the experiments reported in this paper consists of two 1027GR-TRF Supermicro servers with the following characteristics:

– Two Intel Xeon hexa-core processors E5-2620 v2 (Ivy Bridge) operating at 2.1 GHz.
– 32 GB of DDR3 SDRAM memory at 1,600 MHz.

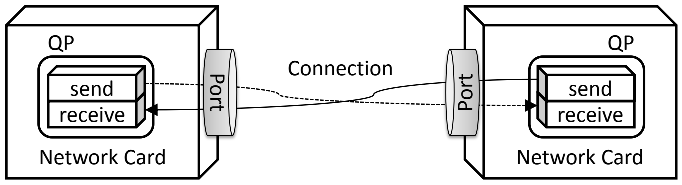


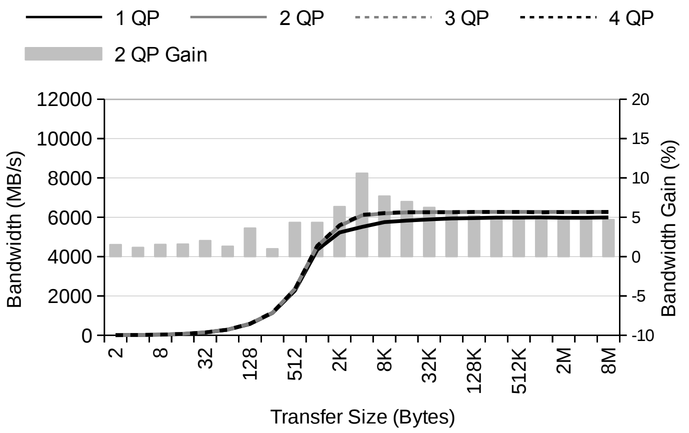
Fig. 2 InfiniBand Queue Pair (QP) scheme.

- 1 Mellanox ConnectX-3 single-port FDR InfiniBand adapter.
- 1 Mellanox ConnectX-4 single-port EDR InfiniBand adapter.
- 1 NVIDIA Tesla K20m GPU.
- 1 NVIDIA Tesla K40m GPU.
- CentOS 6.4 operating system with Mellanox OFED 2.3-2.0.0 (InfiniBand drivers and administrative tools) and CUDA 6.5 with NVIDIA driver 340.29.

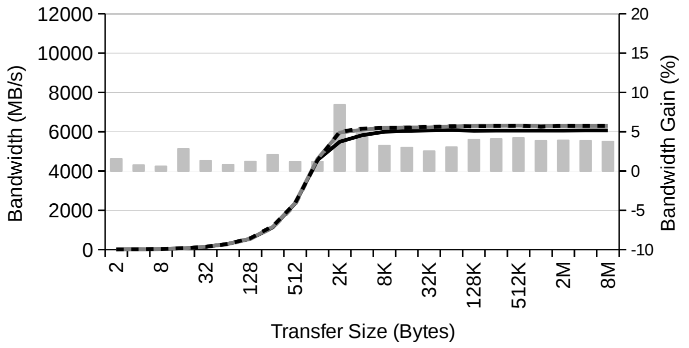
The testbed servers are NUMA machines and therefore NUMA effects matter for the experiments shown in this paper. For this reason, the InfiniBand adapter and the NVIDIA GPU used in each experiment are attached to the same NUMA node and processes and memory buffers are bound to that processor.

3.1 Number of Queue Pairs per Port

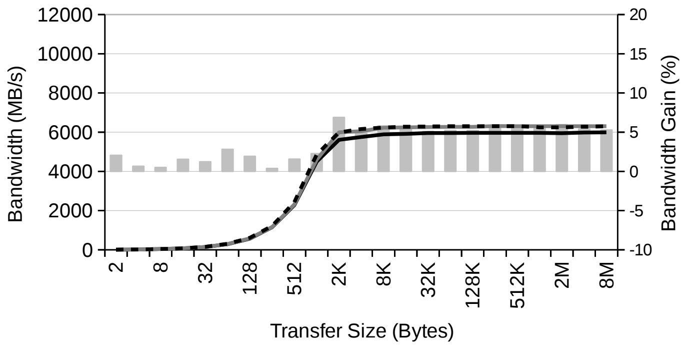
As depicted in Figure 2, in order for an application in one computer to communicate over an InfiniBand network with another application in a different cluster node, it must first create a connection that consists of a queue pair (QP) at each end: one queue for sending data and another queue for receiving them. Interestingly, a QP does not store data but work requests submitted by the application. A work request can be seen as a descriptor of the transfer operation to be performed. A given QP is assigned to one port and a process may create one or more QPs associated to the same network adapter port for communicating purposes with an application in another computer. Obviously, the use of several QPs increases the complexity of maintaining all of them coordinated and synchronized. In this subsection we analyze the impact on performance of using several QPs per port, trying to determine the optimal number of QPs per port that a programmer should use. To do so, we base our analysis in the maximum bandwidth achieved when varying the number of QPs associated to a network adapter port. We make use of the bandwidth benchmarks included in the Mellanox OFED software distribution. These tests measure the bandwidth when copying different data sizes using the channel semantics (i.e., send/receive verbs no using RDMA, `ib_send_bw` benchmark) and the memory semantics (i.e., RDMA read and write, `ib_read_bw` and `ib_write_bw` benchmarks).



(a) InfiniBand FDR send bandwidth (no RDMA).

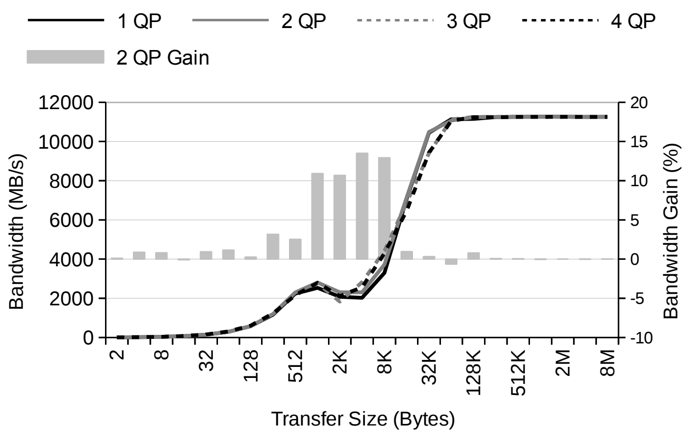


(b) InfiniBand FDR RDMA read bandwidth.

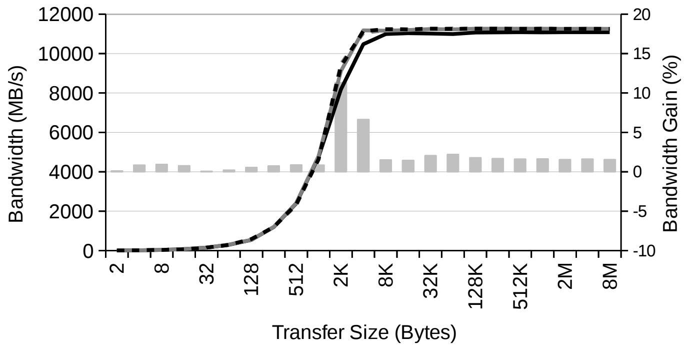


(c) InfiniBand FDR RDMA write bandwidth.

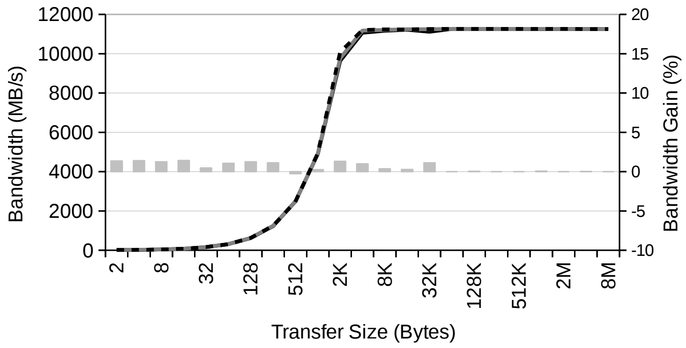
Fig. 3 InfiniBand FDR bandwidth tests varying the number of queue pairs (QP) per port. Primary Y-axis shows attained bandwidth, while secondary Y-axis presents the bandwidth gain of using 2 QPs over using only 1 QP.



(a) InfiniBand EDR send bandwidth (no RDMA).



(b) InfiniBand EDR RDMA read bandwidth.



(c) InfiniBand EDR RDMA write bandwidth.

Fig. 4 InfiniBand EDR bandwidth tests varying the number of queue pairs (QP) per port. Primary Y-axis shows attained bandwidth, while secondary Y-axis presents the bandwidth gain of using 2 QPs over using only 1 QP.

Figure 3 and Figure 4 show the results of the mentioned benchmarks, which were run varying the number of QPs per port, when using FDR InfiniBand and EDR InfiniBand, respectively. Notice that when using more than one QP, the transferred data are split into the available QPs. For instance, when transferring 2KB using 2 QPs, 1KB is sent using QP1 and 1KB is sent using QP2. Notice also that this division of labor between the several QPs is not automatically performed but the programmer must take care of distributing the work at the same time that all the QPs remain synchronized and balanced. The figures show the average bandwidth of 100 repetitions for each test. The maximum Relative Standard Deviation (RSD) observed was 0.391 for 16B of transfer size when using 3 QPs in the `ib_write_bw` benchmark of Figure 3.

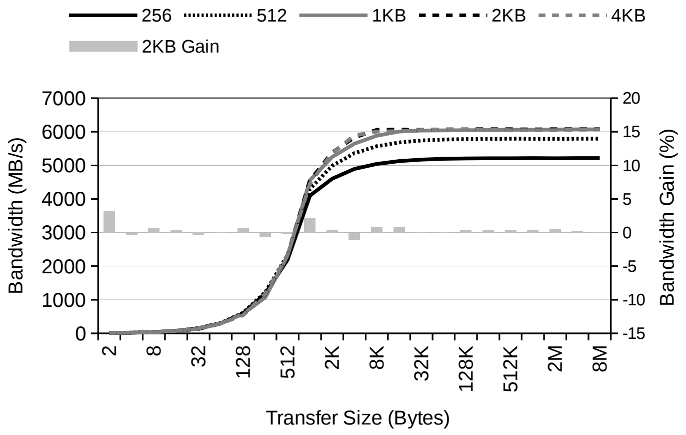
From the results in Figure 3 two main conclusions can be derived. First, there exists a performance difference between using one or several QPs. However, when more than one QP are used, performance remains the same independently of the amount of QPs. Second, results in Figure 3 can be divided, from the point of view of performance, into three groups, depending on the size of the transferred data:

– Less than 2KB: using more than 1 QP translates into an average bandwidth gain of approximately 2%.
– 2KB: the maximum peak bandwidth is achieved and the benefits of using more than 1 QP are more evident.
– 4KB or more: the gain of using more than 1 QP stabilizes, resulting in an average bandwidth improvement of approximately 5%.

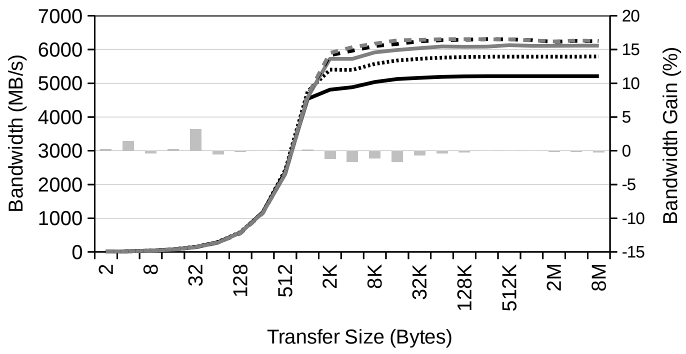
However, when using EDR InfiniBand in Figure 4, the gain of using more than one QP is not as evident as in the case of FDR InfiniBand previously shown in Figure 3. In this manner, the improvement for the RDMA write test, Figure 4(c), is negligible. Regarding the send test, Figure 4(a), using more than 1 QP increases the bandwidth over 12% on average when the size of the transferred data is between 1KB and 8KB. For other data sizes, the increase is very low. Finally, in the case of the read test, Figure 4(b), the improvement depends on the size of the transferred data:

– Less than 2KB: the improvement of using more than 1 QP is negligible.
– 2KB: the maximum peak bandwidth is achieved and the benefits of using more than 1 QP are more evident.
– 4KB or more: the gain of using more than 1 QP stabilizes, resulting in an average bandwidth improvement of over 2%.

Therefore, based on these results, we can conclude that, in general, using more than one queue pair per port improves the performance. Given that using 2 or more QPs per port provides the same performance, we consider that 2 QPs per port is the optimal value, because the more QPs per port we use, the more the programming complexity increases.



(a) InfiniBand FDR RDMA read bandwidth using 1 QP.



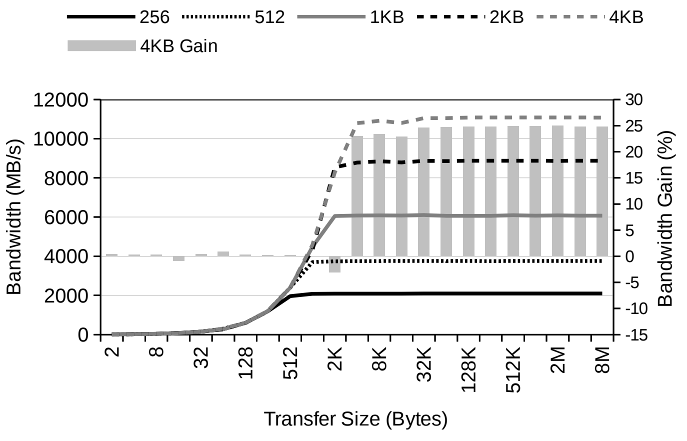
(b) InfiniBand FDR RDMA read bandwidth using 2 QP.

Fig. 5 InfiniBand FDR bandwidth tests varying the MTU. Primary Y-axis shows attained bandwidth, while secondary Y-axis presents the bandwidth gain of using an MTU of size 2KB over using an MTU of size 4KB.

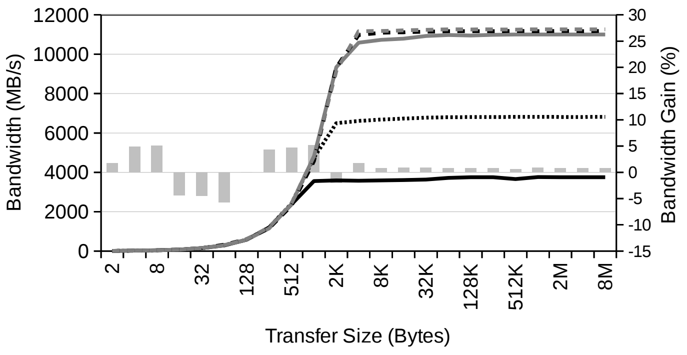
3.2 Influence of the Maximum Transfer Unit (MTU)

Results in Figure 3 and Figure 4 show that performance values change around transfer sizes of 2KB and 4KB. Both values are quite similar to the Maximum Transfer Unit (MTU) often used in InfiniBand devices. In the light of this, we have checked the MTU used by default in the InfiniBand cards used in our tests. In this manner, ConnectX-3 (FDR) uses an MTU of size 2KB by default, while ConnectX-4 (EDR) uses an MTU of size 4KB by default. In this section we analyze how the MTU is influencing the results of the experiments in the previous section, and how the results vary with respect to the MTU.

Figure 5 and Figure 6 show the results of the bandwidth benchmarks from the Mellanox OFED when InfiniBand FDR and InfiniBand EDR are used. The benchmarks were run using 1 QP and 2 QPs, and also varying the MTU



(a) InfiniBand EDR RDMA read bandwidth using 1 QP.



(b) InfiniBand EDR RDMA read bandwidth using 2 QP.

Fig. 6 InfiniBand EDR bandwidth tests varying the MTU. Primary Y-axis shows attained bandwidth, while secondary Y-axis presents the bandwidth gain of using an MTU of size 4KB over using an MTU of size 2KB.

with all the possible values: 256 bytes, 512 bytes, 1KB, 2KB and 4KB. As in the previous section, we use the `ib_send_bw` benchmark (no RDMA), the `ib_read_bw` benchmark (RDMA read), and the `ib_write_bw` benchmark, although only results for the RDMA read case are shown graphically for brevity (results for the other benchmarks will be provided in the text). The results shown are the average bandwidth of 100 repetitions, the maximum RSD being 0.451 for 4B of transfer size when using 1 QP and an MTU of 512 bytes in the `ib_read_bw` benchmark of Figure 5.

Regarding FDR results in Figure 5, we can observe that the best results are the ones obtained when using MTUs of 2KB or 4KB, being the former slightly better in general. The figure also shows the gain of using an MTU size of 2KB (the default value) with respect to using an MTU of 4KB. Thus, when using 1 QP, the average gain of using an MTU of 2KB with respect to using an MTU of 4KB for the different tests is: 2.99% for the `ib_send_bw` one, 0.39% for `ib_read_bw`, and 1% for `ib_write_bw`. When using 2 QP, the average gain is 0.91%, 0.44%, and 1.35%, respectively.

Regarding EDR results shown in Figure 6, we see that the best results are again the ones obtained when using MTUs of 2KB or 4KB, being the later much better, on average, when using 1 QP, and only slightly better when using 2 QPs. The figure also shows the gain of using an MTU of 4KB (the default value) with respect to using an MTU of 2KB. Thus, when using 1 QP, the average gain of using an MTU of 4KB with respect to using an MTU of 2KB for the different tests is: 0.22% for the `ib_send_bw` one, 13.02% for `ib_read_bw`, and 15.14% for `ib_write_bw`. When using 2 QP, the average gain is 0.07%, 0.64%, and 0.23%, respectively.

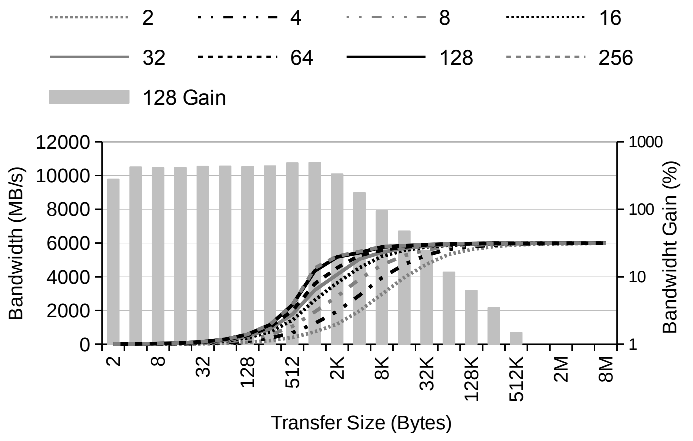
Consequently, using the default MTUs (i.e., 2KB for ConnectX-3 devices, and 4KB for ConnectX-4 devices) presents the best performance, being the improvement more noticeable when using 1QP than when using 2 QPs.

3.3 Capacity of Send/Receive Queues

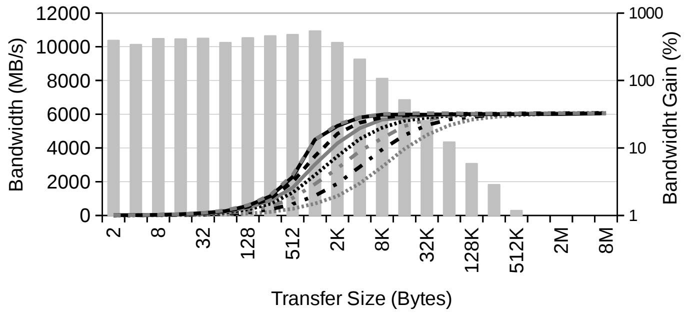
As commented previously, applications communicating over IB must create queue pairs for sending and receiving data. Choosing the length of these queues (i.e., number of work requests they can store) is not a trivial task: the queue should have enough space to allocate all incoming requests from the application in order to not lose performance, but larger queue sizes imply also higher resource consumption. This is especially noticeable in the case of work requests involving RDMA operations, which have associated page-aligned memory regions that must be allocated before submitting the work request to the QP. In this subsection we study the influence of the length of these queues in performance using the attained bandwidth as the metric.

Figure 7 and Figure 8 show the results of the bandwidth benchmarks from the Mellanox OFED mentioned before. As in the previous section, we use the `ib_send_bw` benchmark (no RDMA), the `ib_read_bw` benchmark (RDMA read), and the `ib_write_bw` benchmark (RDMA write). The benchmarks were run varying the length of the send/receive queues. The results shown are the average bandwidth of 100 repetitions, the maximum RSD being 0.587 for 8B of transfer size when using a queue capacity of 128 requests in the `ib_read_bw` benchmark of Figure 7.

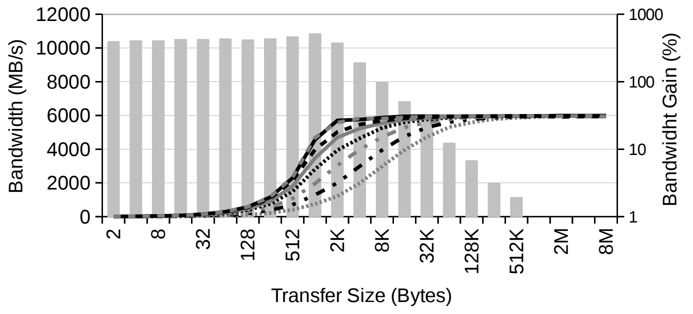
As can be observed in Figure 7, when using FDR InfiniBand, the queue length is particularly important for small transfer sizes (up to 2KB), where the use of a buffer with space for 128 requests increases the bandwidth an average of 418.69% in comparison to a buffer with capacity for 2 requests. For transfer sizes over 2KB, the bandwidth improvement decreases in the range of 4KB to 512KB, with an average gain of 48.46%. With regard to sizes over 512KB, the gain of increasing the number of queues is almost null (0.22%, on average). Additionally, from these experiments we also extract than using a queue length of more than 128 requests results in no gain.



(a) InfiniBand FDR send bandwidth (no RDMA).

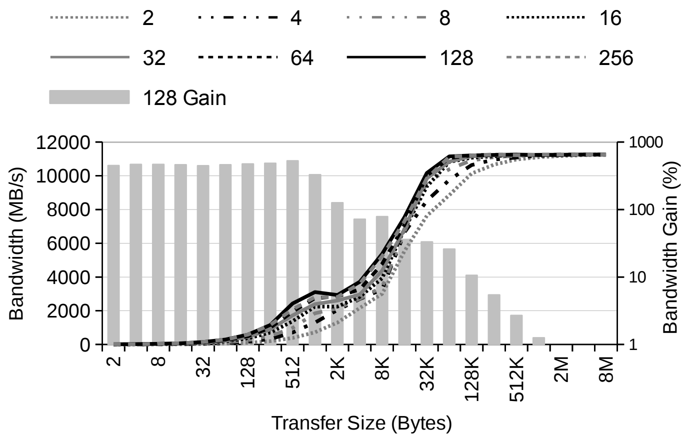


(b) InfiniBand FDR RDMA read bandwidth.

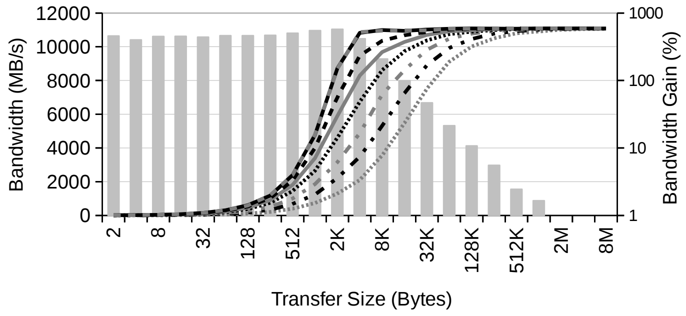


(c) InfiniBand FDR RDMA write bandwidth.

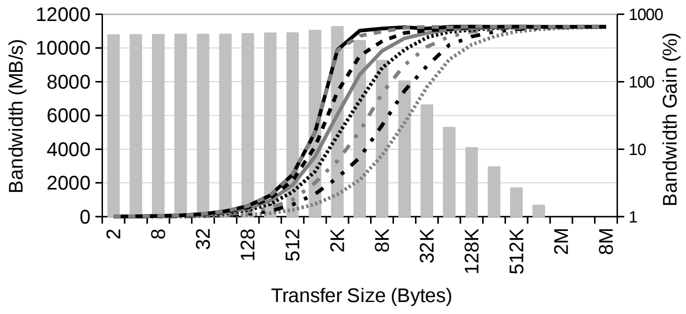
Fig. 7 InfiniBand FDR bandwidth tests varying the capacity of the send/receive queues (i.e., number of work requests that can be allocated) from 2 requests to 256. Primary Y-axis shows attained bandwidth, while secondary Y-axis presents the bandwidth gain of using 128 requests over using only 2 requests. Notice the logarithmic scale of the secondary Y-axis.



(a) InfiniBand EDR send bandwidth (no RDMA).



(b) InfiniBand EDR RDMA read bandwidth.



(c) InfiniBand EDR RDMA write bandwidth.

Fig. 8 InfiniBand EDR bandwidth tests varying the capacity of the send/receive queues (i.e., number of work requests that can be allocated) from 2 requests to 256. Primary Y-axis shows attained bandwidth, while secondary Y-axis presents the bandwidth gain of using 128 requests over using only 2 requests. Notice the logarithmic scale of the secondary Y-axis.

Similar conclusions can be made when using EDR InfiniBand, as shown in Figure 8. Thus, the queue length is particularly important for small transfer sizes (up to 2KB), where the use of a buffer with space for 128 requests increases the bandwidth an average of 475.62% in comparison to a buffer with capacity for 2 requests. For transfer sizes over 2KB, the bandwidth improvement decreases in the range of 4KB to 512KB, with an average gain of 78.15%. With regard to sizes over 512KB, the gain of increasing the number of queues is almost null (0.66%, on average). Additionally, from these experiments we also extract than using a queue length of more than 128 requests results in no gain.

In summary, averaging the results in Figure 7 and Figure 8 for all transfer sizes, using a send/receive queue capacity of 128 requests provides a bandwidth gain (compared to a 2-request queue capacity) of 217.14% and 254.77%, when using FDR and EDR InfiniBand, respectively.

3.4 Combining both Improvements

The improvements presented in the previous subsections complement each other: the first one boosts performance for medium/large data transfers starting from 2KB, whereas the second one increases performance for small/medium message transfers up to 2KB, point where the increment in performance starts diminishing. Therefore, the obvious question arises: which would be the performance when both of them are combined and applied at the same time?

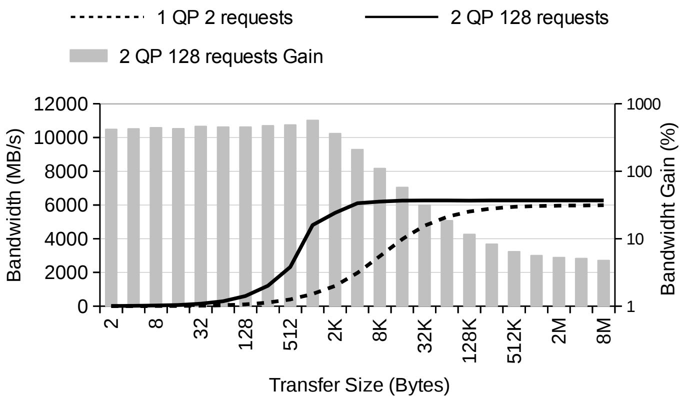
Figure 9 and Figure 10 present results for the combination of both improvements. The results shown are the average bandwidth of 100 repetitions, the maximum RSD being 0.423 for 2B of transfer size when running the `ib_send_bw` benchmark of Figure 9.

Regarding FDR InfiniBand, shown in Figure 9, it can be seen that bandwidth for transfer sizes up to 2KB is increased, on average, more than 450%. From this point, more modest improvements are achieved, although they are still significant. In this regard, from 4KB up to 512KB, bandwidth is increased, on average, 37.68%, whereas for larger transfer sizes starting from 512KB bandwidth only increases 4.73% on average. Considering all the transfer sizes analyzed, bandwidth is increased 43.29% on average.

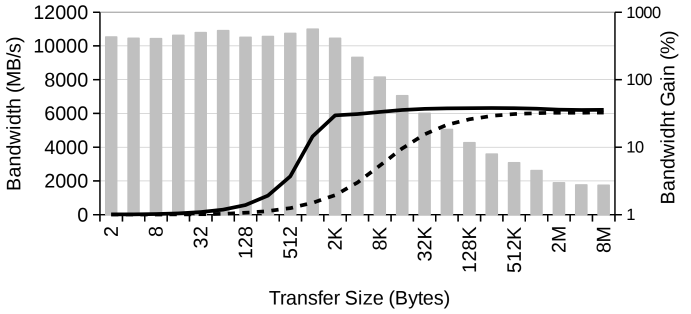
With respect to EDR InfiniBand, presented in Figure 10, similar conclusions can be observed. Thus, the bandwidth for transfer sizes up to 2KB is increased, on average, over 470%. From 4KB up to 512KB, the average increment is 47.32%, while for larger transfer sizes bandwidth only increases 1.58% on average. Taking into account all the transfer sizes analyzed, bandwidth is increased 34.84% on average.

4 Experiments

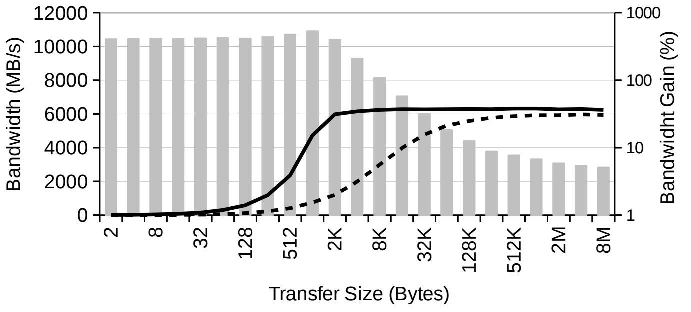
In this section we analyze how the improvements presented in Section 3 influence the performance of upper software layers. For doing so we use a two



(a) InfiniBand FDR send bandwidth (no RDMA).

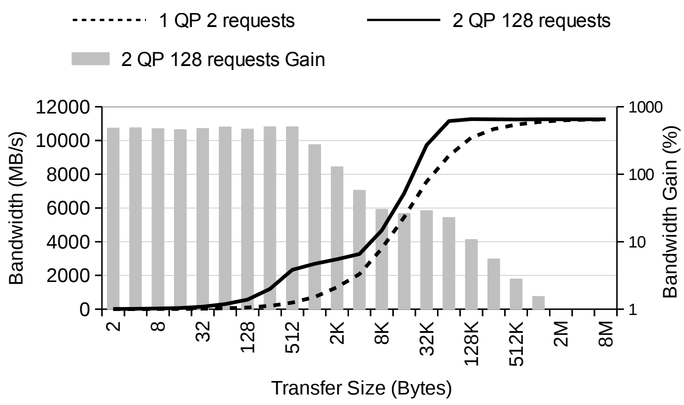


(b) InfiniBand FDR RDMA read bandwidth.

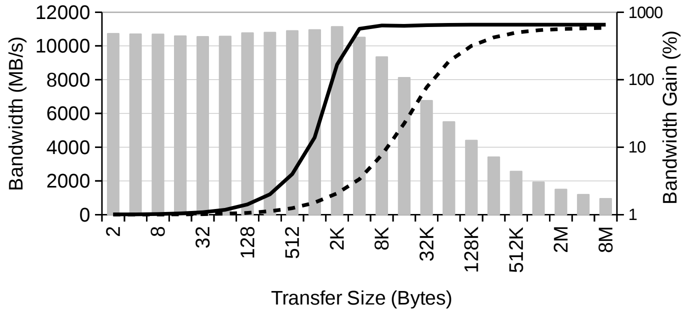


(c) InfiniBand FDR RDMA write bandwidth.

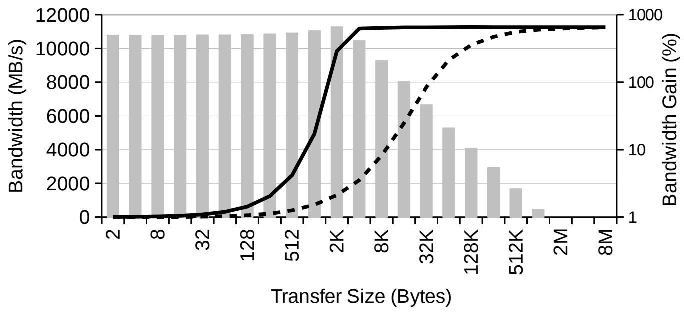
Fig. 9 InfiniBand FDR bandwidth tests varying the capacity of the send/receive queues from 2 requests to 128, and number of queue pairs per port from 1 QP to 2. Primary Y-axis shows the benchmark bandwidth, while secondary Y-axis presents the bandwidth gain of using 2 QPs and 128 queues over using only 1 QP and 2 queues. Notice that secondary Y-axis is in logarithmic scale.



(a) InfiniBand EDR send bandwidth (no RDMA).



(b) InfiniBand EDR RDMA read bandwidth.



(c) InfiniBand EDR RDMA write bandwidth.

Fig. 10 InfiniBand EDR bandwidth tests varying the capacity of the send/receive queues from 2 requests to 128, and number of queue pairs per port from 1 QP to 2. Primary Y-axis shows the benchmark bandwidth, while secondary Y-axis presents the bandwidth gain of using 2 QPs and 128 queues over using only 1 QP and 2 queues. Notice that secondary Y-axis is in logarithmic scale.

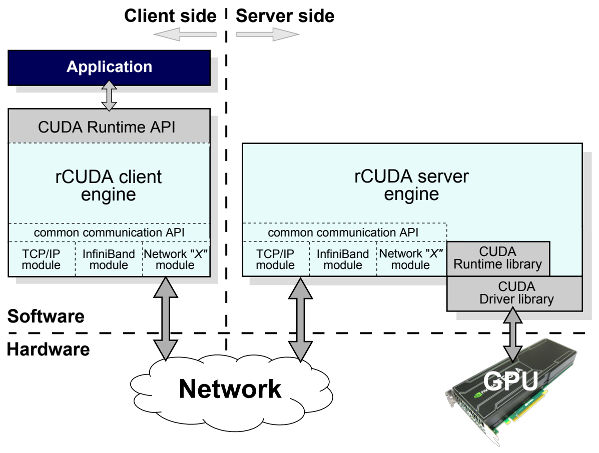


Fig. 11 Overview of the rCUDA client-server architecture.

level approach: first we analyze these improvements in the context of a remote GPU virtualization framework and later we study the benefits provided to applications that use this framework.

4.1 rCUDA: Remote CUDA

CUDA [12] is a technology created by NVIDIA which provides a parallel computing platform and programming model to be used along with NVIDIA GPUs or compatible ones. CUDA takes benefit from the great computational power of GPUs to accelerate certain parts of applications, thus reducing their execution time. rCUDA [13] (remote CUDA) is a middleware which enables CUDA applications being executed in a node of a cluster to make use of GPUs located in remote nodes of the cluster (unlike original CUDA, which is intended for local GPUs). In this manner, by using rCUDA, all the GPUs of the cluster are concurrently and transparently shared among all the nodes of the cluster.

rCUDA is organized following a client-server architecture, as shown Figure 11. The client middleware is used by the application demanding GPU services and presents to the application the same interface as CUDA. Upon receiving a GPU request from the application, the client middleware processes it and forwards the corresponding requests to the rCUDA server middleware, running on a remote node. The server interprets the requests and performs the required processing by instructing the real GPU to execute the corresponding request. Once the GPU has completed the execution of the requested command, the results are gathered by the rCUDA server, which sends them back to the client middleware. There, the output is forwarded to the demanding application.

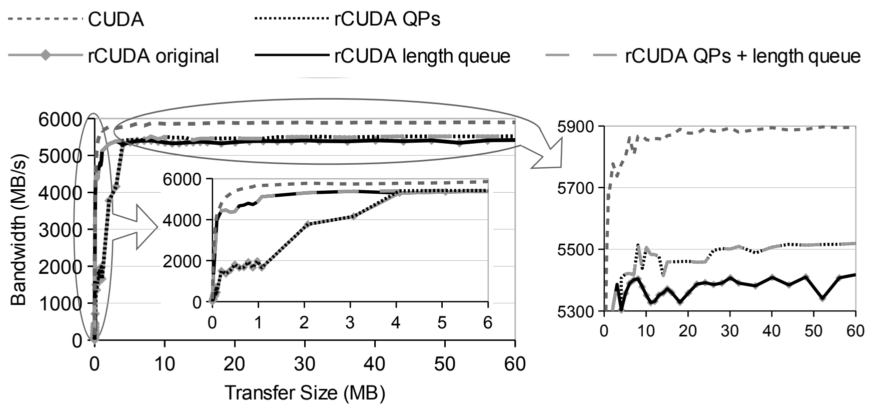
The communication between rCUDA clients and remote GPU servers is carried out via a customized application-level protocol tailored for the underlying network [13]. rCUDA has an specific communication protocol implemented using InfiniBand Verbs, which has been tuned with the results of the analysis shown in the previous section.

4.2 Why Using Remote GPU-accelerators in HPC Clusters

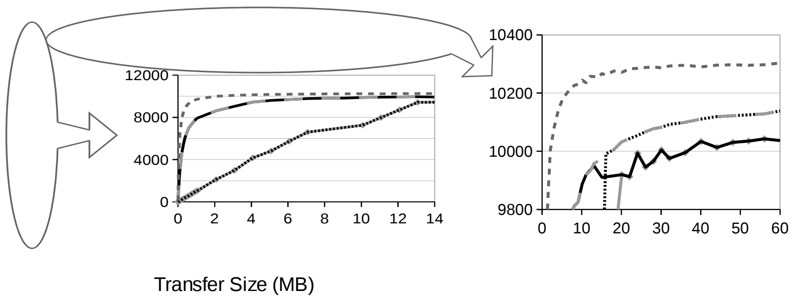
In general, the performance offered by remote GPUs is lower than that of local GPUs because the access to the remote GPU implies moving data through the network, that introducing some overhead. Thus, an important question is why to use remote GPUs when local ones give better performance. In this section we try to motivate the use of remote GPU-accelerators in HPC clusters.

The use of GPUs to accelerate general-purpose scientific and engineering applications is mainstream today, but their adoption in current HPC clusters is impaired primarily by acquisition costs and power consumption. Furthermore, GPU utilization is in general low, causing that the investment on GPU hardware cannot be quickly amortized. Using remote GPU-accelerators is an appealing strategy to deal with all these drawbacks simultaneously. By leveraging remote GPUs, physical accelerators are installed only in some nodes of the cluster, and they are transparently shared among all the nodes. Hence, those nodes equipped with GPUs become servers that provide GPU services to all the nodes in the cluster. Some of the benefits of using remote GPU-accelerators in HPC clusters are:

- By making use of remote GPU-accelerators, a single application being executed in one of the nodes of the cluster can access to all the GPUs installed in the cluster. This amount of GPUs is usually much larger than the number of GPUs that can fit in a single box. Consequently, applications can be further accelerated [14].
- HPC clusters can be easily updated. On the one hand, legacy HPC clusters can be converted in GPU-accelerated clusters just by connecting a box with GPUs to the legacy cluster. In this manner, all the nodes of the legacy cluster can use the remote accelerators of the GPU box. On the other hand, HPC cluster nodes without space for GPUs or without free PCIe 3.0 slots for installing the latest GPUs, can also benefit from a GPU box with the latest technology.
- Virtual machines can easily access to remote GPUs for acceleration purposes [15]. GPU manufacturers have already tried to tackle the problem with virtualization enabled accelerators (i.e., NVIDIA GRID), but this GPUs are oriented towards graphics acceleration, not compute.
- Busy CPU cores do not block GPUs: free GPU-accelerators can be virtually attached/detached to/from other remote cluster nodes. Thus, the free GPU of a node with all its CPU cores being use, can be remotely utilized by other node of the cluster [16].
- GPU task migration: from the point of view of cluster management and administration, server consolidation is a desired cluster capability in order to turn off nodes with low utilization levels, migrating their tasks to other



(a) Using a Tesla K20m GPU with both CUDA and the rCUDA over FDR InfiniBand.



(b) Using a Tesla K40m GPU with both CUDA and the rCUDA over EDR InfiniBand.

Fig. 12 Bandwidth test for regular CUDA (using the GPU within the host executing the benchmark) and also for rCUDA (using a remote GPU over an InfiniBand network). Four different versions of rCUDA are considered: the original rCUDA, rCUDA tuned increasing the capacity of send/receive queues, rCUDA tuned using two QPs, and rCUDA tuned combining both enhancements.

servers. By using remote GPU-accelerators, GPU task migration becomes possible, thus making this approach also possible for GPU-accelerated HPC clusters.

– Sharing GPUs could lead to the installation of less accelerators in HPC clusters, thus reducing acquisition costs and power consumption, while increasing the accelerator utilization rate. Therefore enabling a more efficient use of the available hardware [16].

4.3 Impact of the Improvements on rCUDA bandwidth

Figure 12 shows the results of a CUDA bandwidth test, available in the NVIDIA CUDA Samples [17]. This test measures the bandwidth when copying data from page-locked system memory to GPU memory. In the experiments shown in this and following sections, we will compare the performance of CUDA and rCUDA using two sets of hardware configurations: (1) CUDA using a local Tesla K20m GPU and rCUDA using a remote Tesla K20m over an FDR InfiniBand network, and (2) CUDA using a local Tesla K40m GPU and rCUDA using a remote Tesla K40m over EDR InfiniBand. The reason for this selection is that the bandwidth of the Tesla K20m is comparable to the bandwidth obtained by an FDR InfiniBand network, while the bandwidth of the Tesla K40m is similar to the bandwidth of EDR InfiniBand.

Figure 12 presents results when using CUDA and different versions of rCUDA:

- rCUDA original: this is the current version of rCUDA, which already implements an efficient communication layer based on the use of pipelined transfers [13]. We have included these results for reference.
- rCUDA length queue: this is an enhanced version of rCUDA where, in addition to the already existing pipelined communications, the capacity of send/receive queues has been increased to 128 requests.
- rCUDA QPs: this version of rCUDA uses two QPs in addition to the initial pipelined communication data transfer.
- rCUDA QPs + length queue: this version of rCUDA combines all the improvements.

Results shown in Figure 12 are the average bandwidth of 100 repetitions, and the maximum RSD observed was 1.319 for 14KB[1] of transfer size when using the initial rCUDA version in Figure 12(a). However, this high RSD tends to decrease for larger sizes, reaching a maximum of 0.461 for the biggest ones.

Similar conclusions can be extracted from Figure 12(a) and Figure 12(b). On the one hand, when increasing the capacity of send/receive queues to 128 requests (line labeled as "rCUDA length queue"), there is a noticeable increase in bandwidth for small/medium transfer sizes. On the other hand, when using two QPs (line labeled as "rCUDA QPs"), bandwidth is increased for large transfer sizes. Finally, when increasing the capacity of send/receive queues to 128 requests and using two QPs (line labeled as "rCUDA QPs + length queue"), the bandwidth is increased for all transfer sizes.

Regarding the improvement obtained in Figure 12(b) for large copies when using two QPs with EDR InfiniBand, note that in Section 3.3 we have shown that the bandwidth only improved when using the RDMA read benchmark. rCUDA uses this operation for data transfers in this test, and that is the reason why the bandwidth also improves for large transfer sizes in this figure.

In Section 3 we have concluded that, in general, using 2 queue pairs per port and a send/receive queue capacity of 128 requests improves the performance. However, the improvement seems to have a larger impact on the bandwidth benchmarks from the OFED distribution than on rCUDA bandwidth. The reason appears to be that bandwidth tests from the OFED distribution use

[1] Although the X-axis is shown in MB/s for clarity, notice that the test has been made using different transfer sizes from 1KB to 60MB.

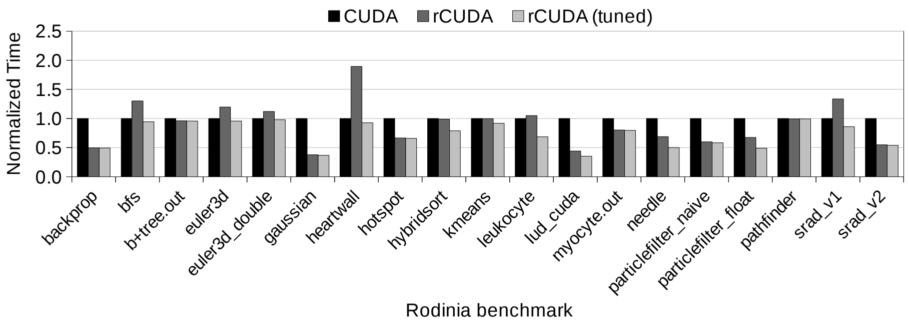


Fig. 13 Normalized execution time of several Rodinia benchmarks using a Tesla K20m GPU with both CUDA and the rCUDA over FDR InfiniBand.

the network in a manner that the vast majority of applications do not. Thus, these tests send the exact same message size over and over again in a loop (using the same buffer as well), which allows for cache effects that are not commonly replicated in real code. As such, that is probably the basis behind why applying the improvements did not see as large an impact as expected when using rCUDA. Nevertheless, the bandwidth of rCUDA has clearly been improved thanks to the conclusions extracted from Section 3 which, from our point of view, validates our study. In addition, next we present results using a GPU benchmark and also real applications, which also confirm that applying the optimizations obtained in previous sections turns into better performance.

4.4 Impact of the Improvements on Rodinia benchmark

Rodinia [18] is a popular benchmark suite for heterogeneous computing aimed to help architects study platforms such as GPUs. It includes applications and kernels which target multi-core CPU and GPU platforms. The Rodinia benchmarks cover a wide range of parallel communication patterns and synchronization techniques, which we consider useful for an initial study. For the experiments shown in this section, we have used the version 3.0 of the Rodinia benchmark, following the instructions inside each benchmark for running them

Figure 13 and Figure 14 present the normalized execution time of several of the benchmarks included in this suite when using CUDA, the original version of rCUDA and the version of rCUDA tuned using the improvements presented in Section 3.

As expected, the tuned version of rCUDA performs equal or better than the original version of rCUDA. The differences in performance are more noticeable for benchmarks in which data transfers are more important. This is the case, for instance, of the benchmarks `bfs`, `euler3d`, `euler3d_double`, `heartwall`, and `srad_v1`.

It is remarkable, however, that some benchmarks shown in these figures runs faster with rCUDA, using a remote GPU, than with CUDA, using a local

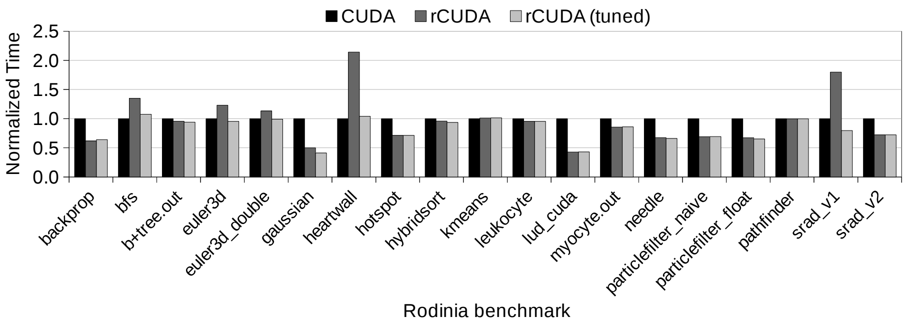


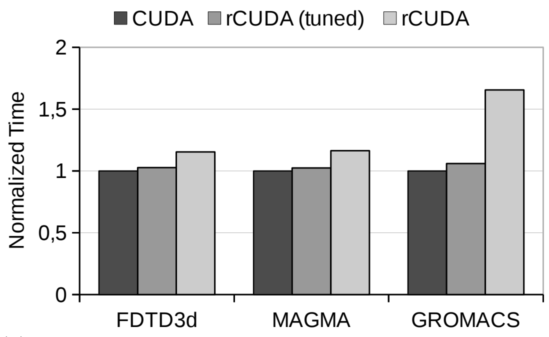
Fig. 14 Normalized execution time of several Rodinia benchmarks using a Tesla K40m GPU with both CUDA and the rCUDA over EDR InfiniBand.

GPU. A deeper profiling reveals that some of these benchmarks have synchronization points (i.e., calls to `cudaDeviceSynchronize` or `cudaStreamWaitEvent`), that run faster with rCUDA. For example, a single call to `cudaDeviceSynchronize` takes approximately 40 microseconds in rCUDA, and about 530 microseconds in CUDA. The cause of this variance resides in the internal algorithm used in rCUDA to determine the end of the call, which benefits rCUDA in these short tests [14]. Still, the time saved in these calls does not completely explain the better performance of rCUDA in these tests. Another aspect which influences the execution time is the frequency used to poll the network for work completions, usually referred to as the network polling interval. This period seems to be lower in rCUDA than the one used by CUDA to poll the PCIe link, as demonstrated in [14]. In this manner, for short tests where the sum of these small waits becomes an important part of the total execution time, rCUDA runs faster than CUDA. In short, there are several elements which contribute to increase and reduce the overhead of rCUDA with respect to CUDA. Hence, for each of the benchmarks executed, the exact combination of these factors results in a better or worse execution time.

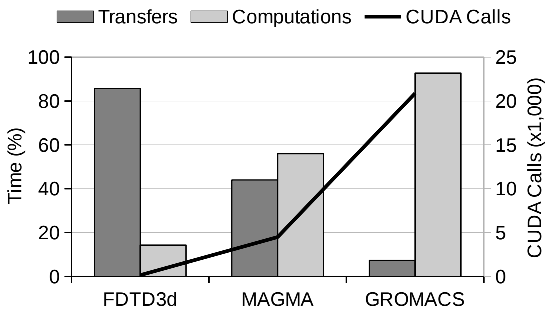
4.5 Impact of the Improvements on Applications using rCUDA

Next we evaluate the benefits that the tuned version of rCUDA (using 2 QPs and a queue capacity of 128 requests) provides to applications (the software layer immediately on top of it). For that purpose, we use the FDTD3d, MAGMA, and GROMACS production codes:

– FDTD3d [17]: applies a finite-difference time-domain (FDTD) progression stencil on a 3D surface. In particular, we have used the version distributed along with CUDA 6.5, running the default test: a FDTD on 376 x 376 x 376 volume with symmetric filter radius 4 for 5 time-steps.
– MAGMA [19,20]: it is a dense linear algebra library similar to LAPACK but for heterogeneous architectures. We utilize release 1.6.0 along with the `dpotrf_gpu` benchmark, which computes the Cholesky factorization



(a) Normalized execution time of the applications when using regular CUDA, rCUDA and rCUDA tuned.



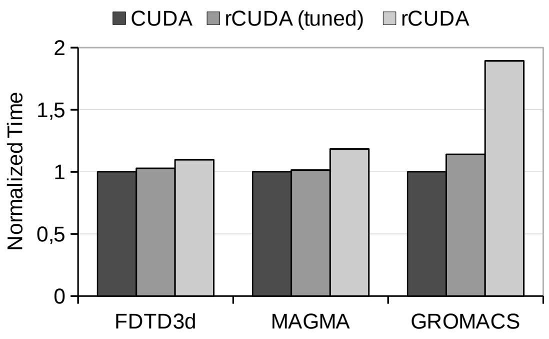
(b) NVIDIA profiling results: time spent in transfers (i.e., copies to/from GPU memory), time employed by computations (i.e., CUDA kernels), and total number of calls to the CUDA API.

Fig. 15 Performance evaluation of FDTD3d, MAGMA, and GROMACS using a Tesla K20m GPU and FDR InfiniBand.

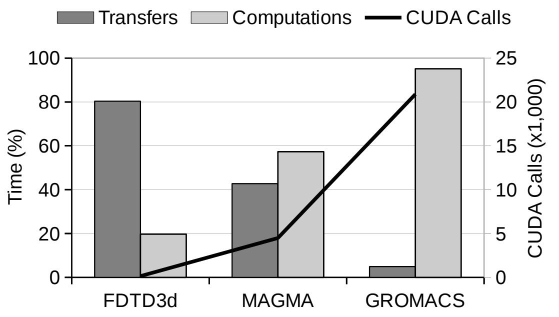
for different matrix sizes (from 1K to 10K elements per dimension, in 1K increments).

– GROMACS [21,22]: it is a versatile package to perform molecular dynamics, i.e., simulate the Newtonian equations of motion for systems with hundreds to millions of particles. We use version 4.6.5 and the ion channel system benchmark with 1K steps.

Notice that in these experiments the exact test for each application has been selected trying to cover the different possible behaviors of applications: (1) transfer-bound application (this is the case of the FDTD3d test), (2) same level of computations and transfers (for this we have used the MAGMA experiment), and (3) compute-bound application (the GROMACS test). Thus, this is the reason why we run the FDTD3d test with 376 x 376 x 376 volume with symmetric filter radius 4 for 5 time-steps: to evaluate the impact of our improvements in applications which have much more transfers than computations. To analyze the opposite scenario, we use the GROMACS ion channel system benchmark with 1K steps, because it performs much more compu-



(a) Normalized execution time of the applications when using regular CUDA, rCUDA and rCUDA tuned.



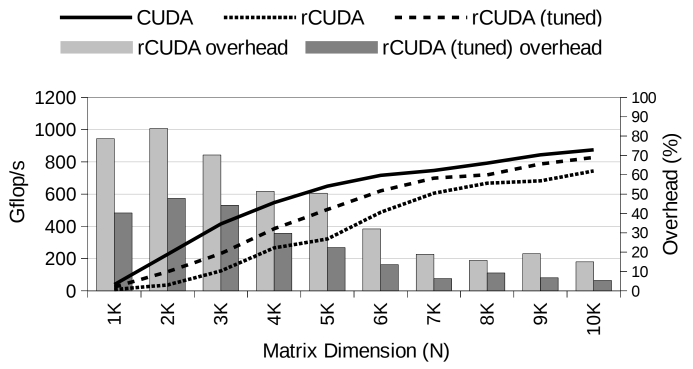
(b) NVIDIA profiling results: time spent in transfers (i.e., copies to/from GPU memory), time employed by computations (i.e., CUDA kernels), and total number of calls to the CUDA API.

Fig. 16 Performance evaluation of FDTD3d, MAGMA, and GROMACS using a Tesla K40m GPU and EDR InfiniBand.

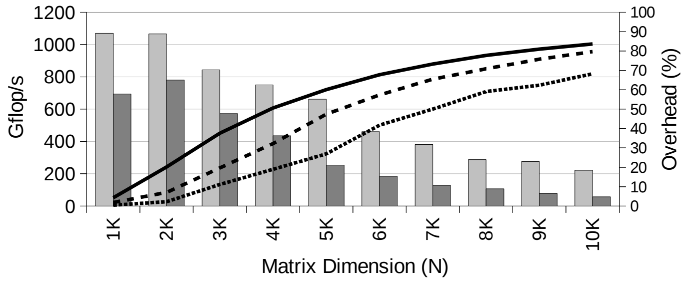
tations than transfers. Finally, to study the impact on applications with an intermediate behavior, we run the MAGMA `dpotrf_gpu` benchmark, which presents a similar level of computations and transfers.

Figure 15 and Figure 16 present the results of the experiments previously commented. Figure 15(a) and Figure 16(a) show the normalized execution time when running these applications with regular CUDA, original rCUDA, and rCUDA tuned. In order to better analyze these results, Figure 15(b) and Figure 16(b) show some profiling results: time spent in transfers (i.e., copies to/from GPU memory, also referred to as CUDA memcpy), time employed by computations (i.e., time employed by CUDA kernels), and total number of calls to the CUDA API. The results shown are the average of 10 repetitions, and the maximum RSD observed was 0.562 when running the FDTD3d simulation with the original version of rCUDA in Figure 15(a).

As we can observe in Figure 15(b) and Figure 16(b), each application presents a different behavior. Firstly, the FDTD3d test has been selected because it represents a scenario where there are much more transfers than



(a) Using a Tesla K20m GPU with both CUDA and the rCUDA over FDR InfiniBand.

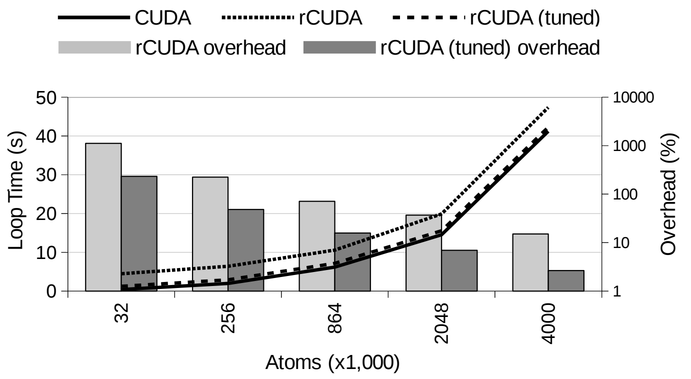


(b) Using a Tesla K40m GPU with both CUDA and the rCUDA over EDR InfiniBand.

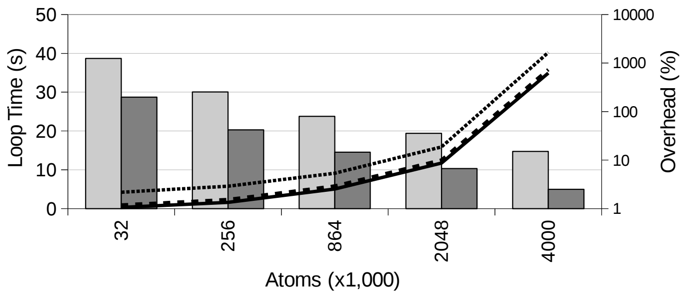
Fig. 17 Performance comparison when running MAGMA `dpotrf_gpu` test with CUDA and rCUDA. Primary Y-axis shows Gflop/s and secondary Y-axis rCUDA overhead with respect to CUDA. Lines refer to values in the primary Y-axis (i.e., Gflop/s), while bars to values in the secondary Y-axis (i.e., overhead).

computations: over 80% of the test execution time is devoted to transfer data to/from the GPU memory. This is the worst possible scenario for rCUDA because the overhead due to the transfers across the network is more evident. However, it is also a good scenario to show the benefits of the enhancements under analysis in terms of bandwidth gain. In this manner, the tuned version of rCUDA presents an improvement of over 10% and 6% with regard to the initial version of rCUDA, as shown in Figure 15(a) and Figure 16(a), respectively.

Next, GROMACS shows the opposite scenario: over 90% of the execution time is devoted to computations in the GPU. Performing much more computations than transfers benefits rCUDA in the sense that the time spent in computations in the GPU is the same for CUDA and rCUDA, thus compensating the overhead of rCUDA due to transfers across the network. Notice also that this application presents a huge number of calls to the CUDA API. Each CUDA call is forwarded by rCUDA over the network to the remote node owning the real GPU. From the InfiniBand perspective, CUDA calls can be seen as



(a) Using a Tesla K20m GPU with both CUDA and the rCUDA over FDR InfiniBand.



(b) Using a Tesla K40m GPU with both CUDA and the rCUDA over EDR InfiniBand.

Fig. 18 Performance comparison when running LAMMPS `in.eam` test with CUDA and rCUDA. Primary Y-axis shows loop time in seconds and secondary Y-axis rCUDA overhead with respect to CUDA in logarithmic scale. Lines refer to values in the primary Y-axis (i.e., loop time), while bars to values in the secondary Y-axis (i.e., overhead).

transfers of small size (a header of 12 bytes + a variable size of data depending on the arguments of each CUDA call). As previously shown in Figure 12, the improvement consisting in increasing the send/receive queue capacity improved performance for small/medium transfer sizes. This explains why the rCUDA tuned version needs 35% and 39% less time to complete this test, as shown in Figure 15(a) and Figure 16(a), respectively.

Finally, we have used the MAGMA application experiment to show an scenario where the time spent in transfers and computations is equilibrated (44% of time spent in transfers, 56% in computations). The number of CUDA calls is also an intermediate amount with respect to the previous experiments. In this case, we can attribute the gain of using rCUDA tuned (11% and 14% when compared to the initial version of rCUDA, as shown in Figure 15(a) and Figure 16(a), respectively) to both the increment of the maximum bandwidth because of using two QPs, and the reduction of the time spent in sending small/medium messages due to the increased send/receive queue capacity.

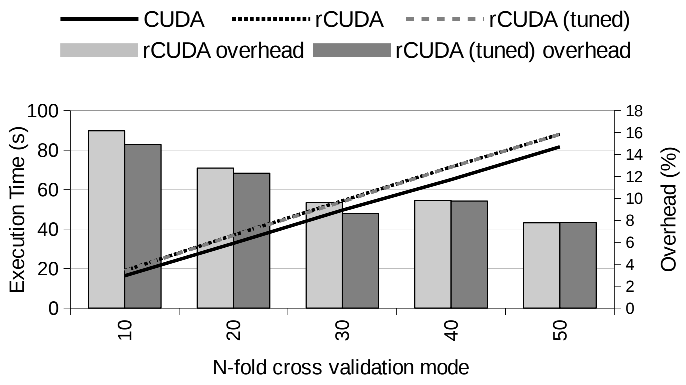
Figure 17 presents again results of the MAGMA Cholesky factorization for different matrix sizes. But now we show the results from the usual perspective they are presented when running this application: in terms of Gflop/s. As it can be seen, the improvement of using the tuned version of rCUDA, labeled as "rCUDA (tuned)", is clear. Thus, the original version of rCUDA, labeled as "rCUDA", presents an average overhead of 31% and 37% in Figure 17(a) and Figure 17(b), respectively, with respect to the tuned version. This evidences the improvements achieved with the enhancements exposed in this work.

Next we use another application to evaluate the impact of the improvements presented in this work: LAMMPS [23]. It is a classic molecular dynamics simulator that can be used to model atoms or, more generically, as a parallel particle simulator at the atomic, mesoscopic, or continuum scale. For the test in this work, we use the release from May 15, 2015, and the benchmark `in.eam` installed with the application. We run the benchmark with one processor and one GPU, scaling by a factor of 1, 2, 3, 4 and 5 in all three dimensions (i.e., problem sizes of 32,000, 256,000, 864,000, 2,048 and 4,000,000 atoms, respectively).

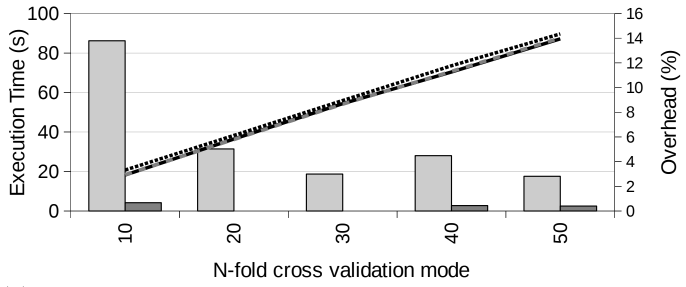
Figure 18 presents the loop time of running this benchmark with the different problem sizes. Lines show that loop time increases as the problem size increases. In contrast, bars show that the overhead of rCUDA with respect to CUDA decreases as the problem size increases. As we can observe, the version of rCUDA tuned with the improvements analyzed in this work, labeled as "rCUDA (tuned)", clearly outperforms the original version of rCUDA, referred to as "rCUDA" in the figures. On average, the tuned version runs 94% and 119% faster in Figure 18(a) and Figure 18(b), respectively. These results show, again, the influence the of improvements exposed in this work.

Finally, we use GPU-LIBSVM [24] application to show again the influence of the tuning exposed in this work. It is an integrated software that supports vector classification, (C-SVC, nu-SVC), regression (epsilon-SVR, nu-SVR) and distribution estimation (one-class SVM). In addition, it supports multi-class classification. For our experiments, we have used version 3.18, and the input data included in the package. More specifically, we have scaled the available training data, without the use of the shrinking heuristics and utilizing a N-fold cross validation mode, varying N from 10 to 50 in increments of 10.

Figure 19 presents the execution time of running the commented tests. When using rCUDA over EDR InfiniBand, in Figure 19(b), the results are the expected ones, and the original version of rCUDA presents an average overhead of 5.7% with respect to the tuned version of rCUDA. In contrast, when using rCUDA over FDR InfiniBand, in Figure 19(a), this overhead is only 0.5%. After profiling the application, we have found that the reason for this seems to lie in the size of data transfers which is between 5MB and 10MB. Thus, in Figure 12(a) we have seen that both the original version of rCUDA and the tuned one, achieved similar bandwidth starting from 4MB when using FDR. However, in Figure 12(b) we have seen that both rCUDA versions reached similar bandwidth starting from 13MB when using EDR, being the



(a) Using a Tesla K20m GPU with both CUDA and the rCUDA over FDR InfiniBand.



(b) Using a Tesla K40m GPU with both CUDA and the rCUDA over EDR InfiniBand.

Fig. 19 Performance comparison when running GPU-LIBSVM test with CUDA and rCUDA. Primary Y-axis shows execution time in seconds and secondary Y-axis rCUDA overhead with respect to CUDA. Lines refer to values in the primary Y-axis (i.e., execution time), while bars to values in the secondary Y-axis (i.e., overhead).

convergence curve much slower, and this is why the improvements are more noticeable here.

Another factor to consider is that in this experiment the overhead of rCUDA with respect CUDA is higher with FDR than with EDR. The reason can be found again in Figure 12, where we can observe that rCUDA bandwidth is closer to CUDA one when using EDR than when using FDR.

5 Conclusions

The use of InfiniBand networks to interconnect high performance computing clusters has considerably increased during the last years. However, due to the programming complexity of the InfiniBand API and the lack of documentation, there are not enough recent available studies explaining how to tune applications to get the maximum performance of this fabric.

In this paper we have exposed several improvements which can be applied when developing applications using InfiniBand Verbs. Based on our experiments we can conclude that (1) using more than one queue pair per port improves bandwidth for medium/large message sizes, (2) increasing the capacity of the send/receive queues also turns into a bandwidth increase, being especially noteworthy for small/medium message sizes, and (3) both improvements complement each other and can be combined. This bandwidth increment is key for remote GPU virtualization frameworks. Actually, the experiments presented in this paper show that this noticeable bandwidth gain translates into a great reduction in execution time of applications using remote GPU virtualization frameworks, significantly reducing the performance gap between remote and local GPUs.

Acknowledgements This work was funded by the Spanish MINECO and FEDER funds under Grant TIN2012-38341-C04-01. Authors are also grateful for the generous support provided by Mellanox Technologies.

References

1. InfiniBand Trade Association (IBTA), 2015. [Online]. Available: http://www.infinibandta.org
2. J. DAmbrosia, "Ethernet in the TOP500," 2014. [Online]. Available: http://www.scientificcomputing.com/blogs/2014/07/ethernet-top500
3. "TOP500 Supercomputer Sites," 2014. [Online]. Available: http://www.top500.org/
4. InfiniBand Trade Association (IBTA), *The InfiniBand Trade Association Specification*, 2007.
5. G. Kerr, "Dissecting a small infiniband application using the verbs API," *CoRR*, vol. abs/1105.1827, 2011. [Online]. Available: http://arxiv.org/abs/1105.1827
6. B. Woodruff, S. Hefty, R. Dreier, and H. Rosenstock, "Introduction to the infiniband core software," in *Linux Symposium*, vol. 2, 2005.
7. T. Bedeir, "Building an rdma-capable application with ib verbs," Technical report, HPC Advisory Council, 2010. Available from: http://www. hpcadvisorycouncil. com/pdf/building-an-rdma-capable-application-with-ib-verbs. pdf, Tech. Rep., 2010.
8. Q. Liu and R. D. Russell, "A performance study of infiniband fourteen data rate (fdr)," in *Proceedings of the High Performance Computing Symposium*, ser. HPC '14. San Diego, CA, USA: Society for Computer Simulation International, 2014, pp. 16:1–16:10. [Online]. Available: http://dl.acm.org/citation.cfm?id=2663510.2663526
9. N. Hjelm, "Optimizing one-sided operations in open mpi," in *Proceedings of the 21st European MPI Users' Group Meeting*, ser. EuroMPI/ASIA '14. New York, NY, USA: ACM, 2014, pp. 123:123–123:124. [Online]. Available: http://doi.acm.org/10.1145/2642769.2642792
10. H. Subramoni, K. Hamidouche, A. Venkatesh, S. Chakraborty, and D. Panda, "Designing mpi library with dynamic connected transport (dct) of infiniband: Early experiences," in *Supercomputing*, ser. Lecture Notes in Computer Science, J. Kunkel, T. Ludwig, and H. Meuer, Eds. Springer International Publishing, 2014, vol. 8488, pp. 278–295. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-07518-1_18
11. Unified Communication X (UCX), 2015. [Online]. Available: http://www.openucx.org
12. NVIDIA, *CUDA C Programming Guide 6.5*, 2014.
13. A. J. Peña, C. Reaño, F. Silla, R. Mayo, E. S. Quintana-Ortí, and J. Duato, "A complete and efficient cuda-sharing solution for hpc clusters," *Parallel Computing*, vol. 40, no. 10, pp. 574 – 588, 2014. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S016781911 4001227

14. C. Reaño, F. Silla, A. C. Gimeno, A. J. Peña, R. Mayo, E. S. Quintana-Ortí, and J. Duato, "Improving the user experience of the rcuda remote GPU virtualization framework," *Concurrency and Computation: Practice and Experience*, vol. 27, no. 14, pp. 3746–3770, 2015. [Online]. Available: http://dx.doi.org/10.1002/cpe.3409
15. J. Prades, C. Reaño, and F. Silla, "Flexible Access to CUDA Accelerators from Xen Virtual Machines in InfiniBand Clusters using rCUDA," in *21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP*, 2016.
16. S. Iserte, A. C. Gimeno, R. Mayo, E. S. Quintana-Ortí, F. Silla, J. Duato, C. Reaño, and J. Prades, "SLURM support for remote GPU virtualization: Implementation and performance study," in *26th IEEE International Symposium on Computer Architecture and High Performance Computing, SBAC-PAD*, 2014, pp. 318–325. [Online]. Available: http://dx.doi.org/10.1109/SBAC-PAD.2014.49
17. NVIDIA, *NVIDIA CUDA Samples 6.5*, 2014.
18. S. Che, M. Boyer, J. Meng, D. Tarjan, J. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*, 2009, pp. 44–54.
19. University of Tennessee, "MAGMA: Matrix Algebra on GPU and Multicore Architectures," http://icl.cs.utk.edu/magma, 2014. [Online]. Available: http://icl.cs.utk.edu/magma
20. W. Bosma, J. Cannon, and C. Playoust, "The Magma algebra system. I. The user language," *J. Symbolic Comput.*, vol. 24, no. 3-4, pp. 235–265, 1997, computational algebra and number theory (London, 1993). [Online]. Available: http://dx.doi.org/10.1006/jsco.1996.0125
21. "GROMACS web page," 2014. [Online]. Available: http://www.gromacs.org/
22. S. Pronk, S. Pll, R. Schulz, P. Larsson, P. Bjelkmar, R. Apostolov, M. R. Shirts, J. C. Smith, P. M. Kasson, D. van der Spoel, B. Hess, and E. Lindahl, "Gromacs 4.5: a high-throughput and highly parallel open source molecular simulation toolkit," *Bioinformatics*, vol. 29, no. 7, pp. 845–854, 2013. [Online]. Available: http://bioinformatics.oxfordjournals.org/content/29/7/845.abstract
23. W. M. Brown, A. Kohlmeyer, S. J. Plimpton, and A. N. Tharrington, "Implementing molecular dynamics on hybrid high performance computers: Particle-particle particle-mesh," *Computer Physics Communications*, vol. 183, no. 3, pp. 449–459, 2012.
24. A. Athanasopoulos, A. Dimou, V. Mezaris, and I. Kompatsiaris, "GPU Acceleration for Support Vector Machines," in *12th International Workshop on Image Analysis for Multimedia Interactive Services (WIAMIS)*, 2011.