InfiniBand Verbs Optimizations for Remote GPU Virtualization

Carlos Reaño and Federico Silla
Universitat Politècnica de València, València, 46022, Spain
carregon@gap.upv.es, fsilla@disca.upv.es

***Abstract*—The use of InfiniBand networks to interconnect high performance computing clusters has considerably increased during the last years. So much so that the majority of the supercomputers included in the TOP500 list either use Ethernet or InfiniBand interconnects. Regarding the latter, due to the complexity of the InfiniBand programming API (i.e., InfiniBand Verbs) and the lack of documentation, there are not enough recent available studies explaining how to optimize applications to get the maximum performance from this fabric. In this paper we expose two different optimizations to be used when developing applications using InfiniBand Verbs, each providing an average bandwidth improvement of 3.68% and 217.14%, respectively. In addition, we show that when combining both optimizations, the average bandwidth gain is 43.29%. This bandwidth increment is key for remote GPU virtualization frameworks. Actually, this noticeable gain translates into a reduction of up to 35% in execution time of applications using remote GPU virtualization frameworks.**

***Keywords*-HPC; InfiniBand; CUDA; remote GPU virtualization; rCUDA; performance; optimizations.**

I. 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. Actually, as it can be seen in the figure, the presence of the InfiniBand technology in current supercomputers is even higher than that of Ethernet, having the former a share of 44.8% whereas the latter presents a share of 37.6%. 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.

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

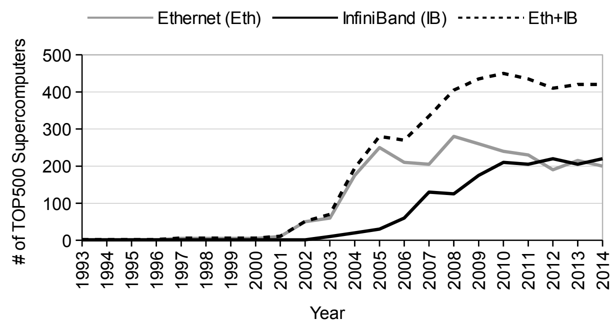


Figure 1. Presence of Ethernet and InfiniBand in the TOP500 list.

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 recent documentation makes it difficult to get all the benefits from this network fabric. In this paper we explore two general code optimizations that may be helpful when developing applications using InfiniBand Verbs. These optimizations achieve an average increase of up to 43% in the attained bandwidth, what is later translated into a reduction of up to 35% in execution time of applications using remote GPU virtualization frameworks.

The rest of the paper is organized as follows. In Section II, we discuss the work related to our study. Next, in Section III, we present and analyze the code optimizations proposed in this paper. Section IV analyzes the benefits that these optimizations bring to remote GPU virtualization frameworks. Finally, in Section V, the main conclusions of this work are presented.

II. 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 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 an optimized GPU to GPU communication design for InfiniBand clusters. Nevertheless, although these proposals improve performance, both of them are focused on optimizing MPI libraries, whose requirements differ from general applications.

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

III. Bandwidth Optimizations

In this section we introduce and analyze the optimizations proposed in this work. The setup used for the experiments reported in this paper consists of two 1027GR-TRF Supermicro servers connected by an SX6025 InfiniBand Switch (FDR), each of the 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.
- 1 Mellanox ConnectX-3 single-port InfiniBand adapter.
- 1 NVIDIA Tesla K20m 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 are attached to the same NUMA node (i.e., processor 0), and processes and memory buffers are bound to this processor in the experiments.

A. 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 in Figure 3(a)), and the memory semantics (i.e., RDMA read and write, `ib_read_bw` and `ib_write_bw` benchmarks, in Figure 3(b) and Figure 3(c), respectively).

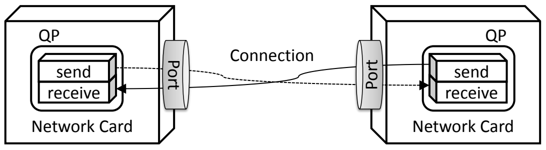
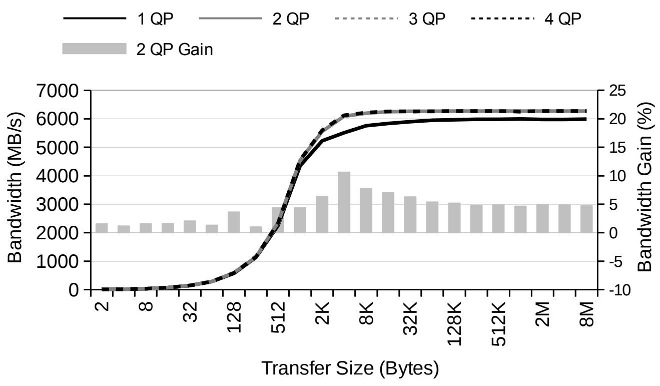


Figure 2. InfiniBand Queue Pair (QP) scheme.

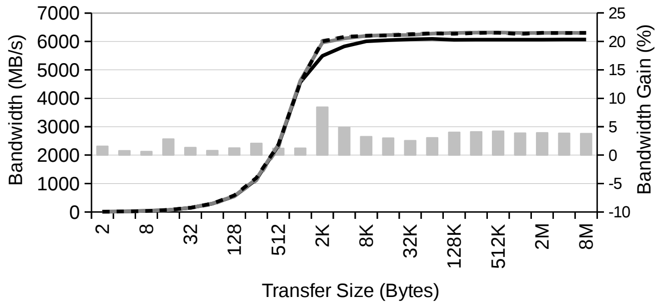
Figure 3 shows the results of the mentioned benchmarks, which were run varying the number of QPs per port. Notice that when using more than one QP, the transferred data are split among the available QPs. For instance, when transferring 2KB using 2 QPs, 1KB is sent using QP1 and 1KB is sent using QP2. 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 figure shows 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.

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:

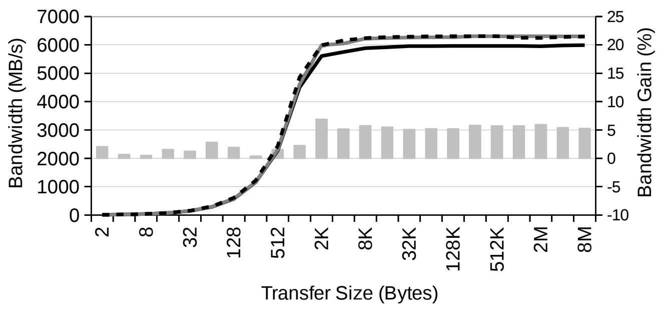
- Less than 2KB: using more than 1 QP translates into an average bandwidth gain of approximately 2%.
- 2KB: the maximum peak bandwidth is achieved because the maximum transfer unit (MTU) is 2KB, and



(a) InfiniBand send bandwidth (no RDMA).



(b) InfiniBand RDMA read bandwidth.



(c) InfiniBand RDMA write bandwidth.

Figure 3. InfiniBand 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.

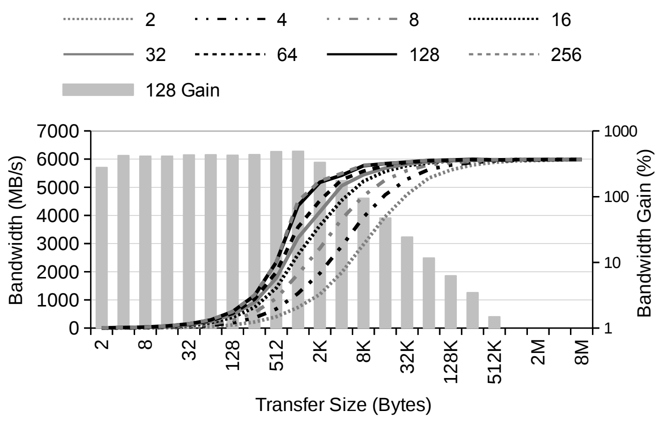
the benefits of using more than 1 QP are more evident.

- 4KB or more: the gain of using more than 1 QP stabilizes, resulting in an average bandwidth improvement of approximately 5%.

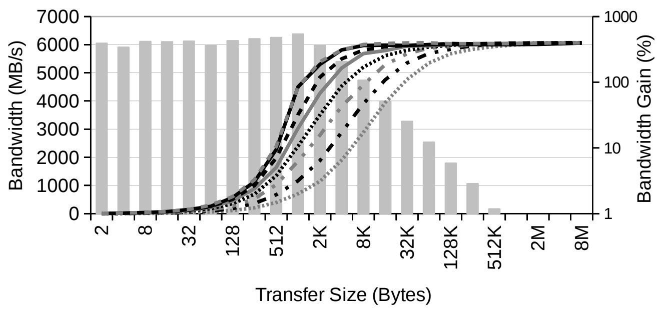
Therefore, based on these results, we can conclude that using more than one queue pair per port turns into an average gain of 3.68%. Given that using 2 or more QPs per port provides the same performance, we consider that 2 QPs per port is the optimal value, because the more QPs per port we use, the more the programming complexity increases.

B. Capacity of Send/Receive Queues

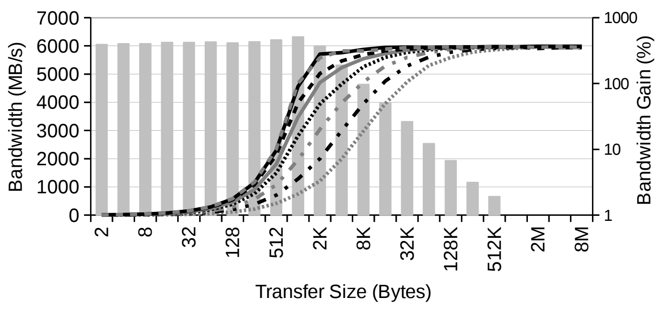
As commented previously, applications communicating over IB must create queue pairs for sending and receiving



(a) InfiniBand send bandwidth (no RDMA).



(b) InfiniBand RDMA read bandwidth.



(c) InfiniBand RDMA write bandwidth.

Figure 4. InfiniBand 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 queues over using only 2 queues. Notice the logarithmic scale of the secondary Y-axis.

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 4 shows 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, Figure 4(a)), the `ib_read_bw` benchmark (RDMA read, Figure 4(b)), and the `ib_write_bw` benchmark (RDMA write, Figure 4(a)). 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. As can be observed in Figure 4, 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.

In summary, averaging the results in Figure 4 for all transfer sizes, using a send/receive queue capacity of 128 requests provides a bandwidth gain of 217.14% when compared to a 2-request queue capacity.

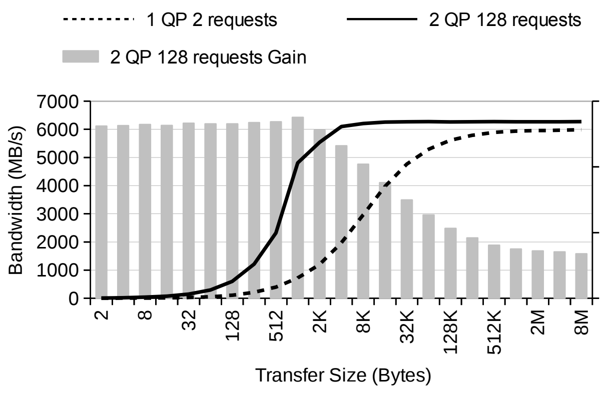
C. Combining both Optimizations

The optimizations presented in the previous subsections complement each other: the first optimization increases performance for large data transfers starting from 2KB, whereas the second optimization boosts performance for small 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 optimizations are combined and applied at the same time?

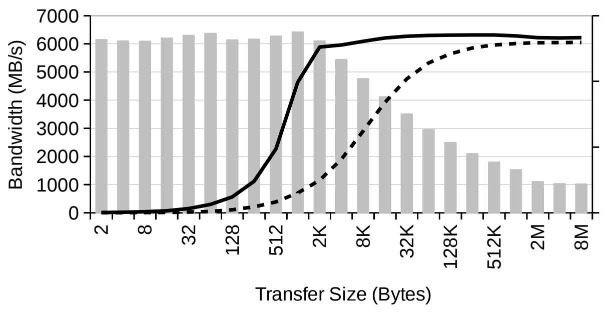
Figure 5 presents results for the combination of both optimizations. The results shown are the average bandwidth of 100 repetitions, the maximum RSD being 0.423 for 2B of transfer size when running the `ib_send_bw` benchmark. It can be seen in this figure that bandwidth for transfer sizes up to 2KB is increased, in 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, in average, 37.68%, whereas for larger transfer sizes starting from 512KB bandwidth only increases an average of 4.73%. In average, considering all the transfer sizes analyzed, bandwidth is increased 43.29%.

IV. Experiments

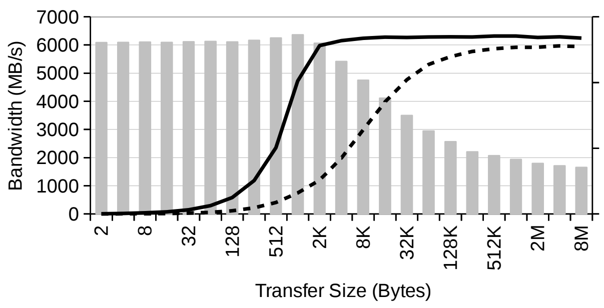
In this section we analyze how the optimizations presented in Section III influence the performance of upper software layers. For doing so we use a two level approach: first we



(a) InfiniBand send bandwidth (no RDMA).



(b) InfiniBand RDMA read bandwidth.



(c) InfiniBand RDMA write bandwidth.

Figure 5. InfiniBand 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.

analyze these optimizations in the context of a remote GPU virtualization framework and later we study the benefits provided to applications that use this framework.

A. rCUDA: Remote CUDA

CUDA [11] 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 [12] (remote

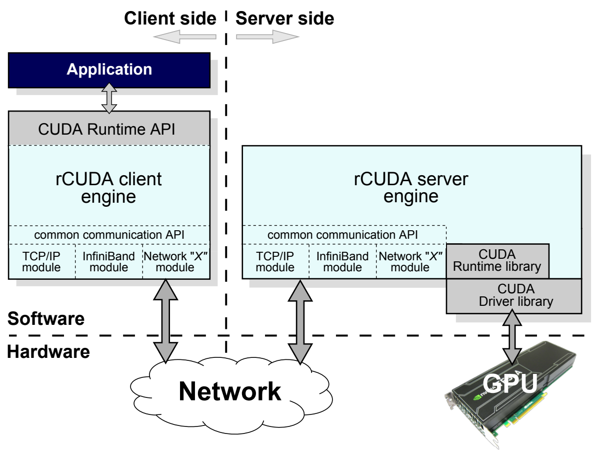


Figure 6. Overview of the rCUDA client-server architecture.

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; see Figure 6. 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 [12]. rCUDA has an specific communication protocol implemented using InfiniBand Verbs, which has been optimized with the results of the analysis shown in the previous section.

B. Impact of the Optimizations on rCUDA

Figure 7 presents the results for a CUDA bandwidth test, available in the NVIDIA CUDA Samples [13]. This test measures the bandwidth when copying data from page-locked system memory to GPU memory. Figure 7 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 [12]. 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 optimizations.

Results shown in Figure 7 are the average bandwidth of 100 repetitions, and the maximum RSD observed was 1.319 for 14KB of transfer size when using the initial rCUDA version. However, this high RSD tends to decrease for larger sizes, reaching a maximum of 0.461 for the biggest ones. It can be seen in the figure that when increasing the capacity of send/receive queues to 128 requests there is a noticeable increase in bandwidth (over 4 times more than the bandwidth obtained by the original rCUDA software) for small/medium transfer sizes (up to 4MB). Furthermore, when using two QPs, bandwidth is only increased by 1% with respect to the original rCUDA version. However, in Section 3 we have determined an average gain of 3.68% when using multiple QPs. Thus, rCUDA internal paths are limiting the gain. We will investigate further during future work on this matter.

C. Impact of the Optimizations on Applications using rCUDA

Next we evaluate the benefits that the optimized 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 HOOMD-Blue, MAGMA, and GROMACS production codes:

- HOOMD-Blue [14], [15]: it is a general-purpose particle simulation toolkit. In particular, we have used its version 1.0.1 for our study, running a classic MD simulation, the Lennard-Jones liquid, with 10 random particles and 10 time steps.
- MAGMA [16], [17]: 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 for different matrix sizes (from 1K to 10K elements per dimension, in 1K increments).
- GROMACS [18], [19]: 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.

Figure 8 presents the results of this evaluation. Figure 8(a), shows the normalized execution time when running these applications with regular CUDA, original rCUDA, and

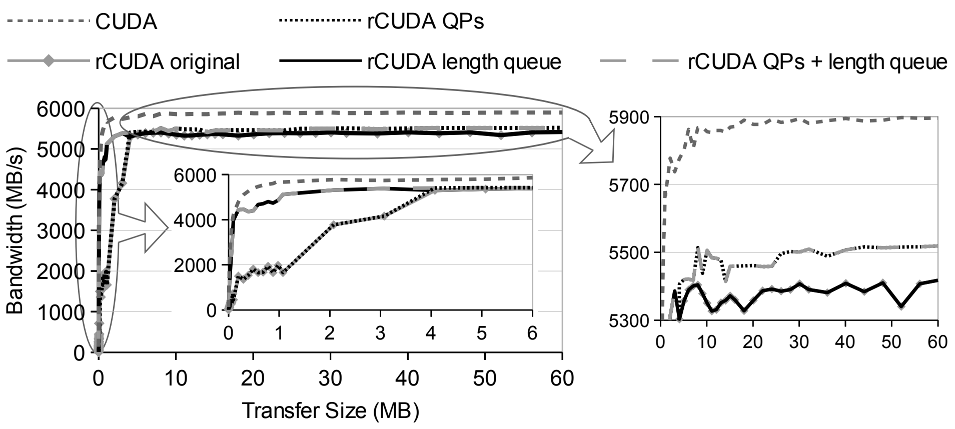
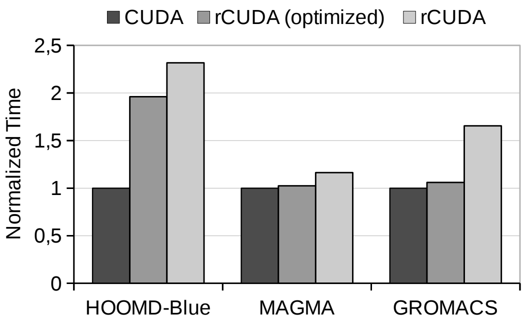


Figure 7. Bandwidth test for regular CUDA (using the GPU within the host executing the benchmark) and also for rCUDA (using a remote GPU). Four different versions of rCUDA are considered: the original rCUDA, rCUDA optimized by increasing the capacity of send/receive queues, rCUDA optimized by using two QPs, and rCUDA optimized combining both optimizations.

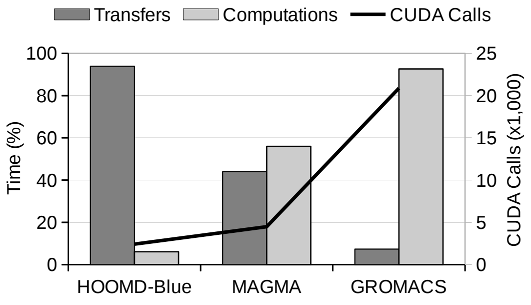
optimized rCUDA (the latter is referred to as *rCUDA (optimized)*, whereas the use of the original version of rCUDA is denoted by the *rCUDA* label). In order to better analyze these results, Figure 8(b) shows some profiling results: time spent in transfers (i.e., copies to/from GPU memory, also referred to as CUDA memcopy), 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 0.671 when running the HOOMD-Blue simulation with the original version of rCUDA.

As we can observe in Figure 8(b), each application presents a different behavior. Firstly, the HOOMD-Blue test has been selected because it represents a scenario where there are much more transfers than computations: over 90% of the test execution time is devoted to transfer data to/from the GPU memory. As expected, this is the worst possible scenario for rCUDA, because the overhead due to the transfers across the network is more evident. Thus, rCUDA needs over 2 times more than CUDA to complete the test. However, it is also a good scenario to show the benefits of the analyzed optimizations in terms of bandwidth gain. In this manner, the optimized version of rCUDA presents an improvement of over the 15% with regard to the initial version of rCUDA.

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 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 transfers of small size (a header of 12 bytes



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



(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.

Figure 8. Performance evaluation of HOOMD-Blue, MAGMA, and GROMACS.

+ a variable size of data depending on the arguments of each CUDA call). As previously shown in Figure 7, the optimization consisting in increasing the send/receive queue capacity improved performance for small/medium transfer sizes (up to 4MB). This explains why the rCUDA optimized version needs 35% less time to complete this test, as shown in Figure 8(a), 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 when using rCUDA optimized (an 11% when compared to the initial version of rCUDA) 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.

V. 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 optimize applications to get the maximum performance of this fabric.

In this paper we have exposed two general optimizations to be used when developing applications using InfiniBand Verbs. Based on our experiments we can conclude that (1) using more than one queue pair per port clearly improves bandwidth (an average gain of 3.68% in our experiments), (2) increasing the capacity of the send/receive queues turns into an average bandwidth improvement of over 200%, being especially noteworthy for small/medium message sizes (over 400% more bandwidth in our experiments), and (3) both optimizations complement each other. In this regard, when combining both optimizations, the average bandwidth gain is 43.29%. This bandwidth increment is key for remote GPU virtualization frameworks. Actually, this noticeable gain translates into a reduction of up to 35% in execution time of applications using remote GPU virtualization frameworks.

Future work includes the analysis of further optimizations at the InfiniBand Verbs API layer as well as improving the integration of these optimizations within the upper software layers.

Acknowledgment

This work was funded by the Generalitat Valenciana under Grant PROMETEOII/2013/009 of the PROMETEO program phase II. Authors are also grateful for the generous support provided by Mellanox Technologies.

References

[1] InfiniBand Trade Association (IBTA), http://www.infinibandta.org, 2015. [Online]. Available: http://www.infinibandta.org

[2] J. DAmbrosia, “Ethernet in the TOP500,” http://www.scientificcomputing.com/blogs/2014/07/ethernet-top500, 2014. [Online]. Available: http://www.scientificcomputing.com/blogs/2014/07/ethernet-top500

[3] “TOP500 Supercomputer Sites,” http://www.top500.org/, 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] NVIDIA, *CUDA C Programming Guide 6.5*, 2014.

[12] 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

[13] NVIDIA, *NVIDIA CUDA Samples 6.5*, 2014.

[14] University of Michigan, “HOOMD-blue web page.” http://codeblue.umich.edu/hoomd-blue, 2014. [Online]. Available: http://codeblue.umich.edu/hoomd-blue

[15] J. A. Anderson, C. D. Lorenz, and A. Travesset, “General purpose molecular dynamics simulations fully implemented on graphics processing units,” *Journal of Computational Physics*, vol. 227, no. 10, pp. 5342 – 5359, 2008. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0021999108 000818

[16] 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

[17] 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

[18] “GROMACS web page,” http://www.gromacs.org/, 2014. [Online]. Available: http://www.gromacs.org/

[19] 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