Influence of InfiniBand FDR on the Performance of Remote GPU Virtualization

C. Reaño, R. Mayo and E. S. Quintana-Ortí
Universitat Jaume I
Castellón, Spain 12071
carregon@gap.upv.es,{mayo,quintana}@icc.uji.es

F. Silla and J. Duato
Universitat Politècnica de València
València, Spain 46022
{fsilla, jduato}@disca.upv.es

A. J. Peña
Argonne National Laboratory
Argonne, IL, USA 60439
apenya@mcs.anl.gov

***Abstract*—**

The use of GPUs to accelerate general-purpose scientific and engineering applications is mainstream nowadays, but their adoption in current high performance computing clusters is primarily impaired by acquisition costs and power consumption. Therefore, the benefits of sharing a reduced number of GPUs among all the nodes of a cluster can be remarkable for many applications. This approach, usually referred to as remote GPU virtualization, aims at reducing the number of GPUs present in a cluster, while increasing their utilization rate.

The performance of the interconnection network is key to achieve reasonable performance results by means of remote GPU virtualization. In this line, several networking technologies with throughput comparable to that of PCI Express have appeared recently. In this paper we analyze the influence of InfiniBand FDR on the performance of remote GPU virtualization, comparing its impact on a variety of GPU-accelerated applications versus other networking technologies, such as InfiniBand QDR or Gigabit Ethernet. Given the severe limitations of freely available remote GPU virtualization solutions, the rCUDA framework is used as the case study for this analysis. Results show that the new FDR interconnect, featuring higher bandwidth than its predecessors, allows to reduce the overhead of using GPUs remotely, thus making this approach even more appealing.

I. Introduction

In the last years, high performance computing (HPC) clusters have become heterogeneous platforms that integrate both multi-core CPUs and special-purpose hardware accelerators, such as graphics processing units (GPUs). In this kind of clusters, one or more accelerators are usually attached to each node of the system. This system configuration has provided promising results by noticeably reducing application execution time in areas as diverse as finance [1], chemical physics [2], computational fluid dynamics [3], computational algebra [4], and image analysis [5], among others.

The trend of including accelerators in all cluster nodes presents several drawbacks. First, in addition to increasing the acquisition costs, the use of accelerators also increments maintenance, administration, and space costs [6]. Second, energy consumption is also increased, as GPUs are known to be power-hungry devices [7]. Third, GPUs in such a cluster may present a relatively low utilization rate, given that it is quite unlikely that all the accelerators in the cluster will be used all the time, as very few applications feature such an extreme data-concurrency degree. In consequence, virtualizing accelerators in the HPC context is revealed as an appealing strategy to deal with all these drawbacks simultaneously. By leveraging GPU virtualization, physical GPUs 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. GPU virtualization leads to the use of a lower number of GPUs across the cluster, thus reducing acquisition costs and power consumption, while increasing the accelerator utilization rate. In consequence, GPU virtualization enables a more efficient use of the available hardware. Several virtualization frameworks are currently available, like rCUDA [8], [9], GVirtuS [10], DS-CUDA [11], vCUDA [12], GViM [13], GridCuda [14], V-GPU [15], SnuCL [16], dOpenCL [17], VOCL [18], and VCL [19].

Obviously, using a remote GPU introduces some overhead, mainly due to the virtualization framework and the network. The GPU virtualization framework increases the latency to the real GPU, as requests must be forwarded to the remote GPU and responses delivered back to the application demanding GPU services. Furthermore, as GPUs are no longer located at the other end of a PCI Express (PCIe) link within the host, but in a remote node, data have to traverse at least two PCIe links and two network interfaces, as well as the entire network fabric between the node requesting GPU services and the node where the actual GPU resides. Therefore, in addition to latency, bandwidth also suffers, given that the PCIe bandwidth is usually noticeably larger than network bandwidth, thus increasing the performance gap between the local and remote uses of GPUs.

In order to minimize the impact of the network overhead, the latency and bandwidth of the interconnect should be comparable to those of PCIe. Although the performance gap between the internal PCIe interconnect and the external fabric was considerable in the past, recently there have appeared several networking technologies with throughput comparable to that of PCIe. As shown in Figure 1, the theoretical throughput of the last version of InfiniBand (IB) FDR is very close to that of PCIe 2 (current PCIe version supported by GPUs). This small performance difference motivates the present work, in which we analyze the influence of different networking technologies on the performance of GPU virtualization, clearly demonstrating that remote GPU virtualization is a feasible option if leveraging high performance networks, as it increases application execution time slightly but, in return, provides a noticeable flexibility for cluster configuration. Furthermore, the analysis in this paper also offers insights on the behavior of remote GPU virtualization once the performance gap between intra-node and inter-node interconnects is again noticeable (for example, when GPUs evolve to PCIe 3.0).

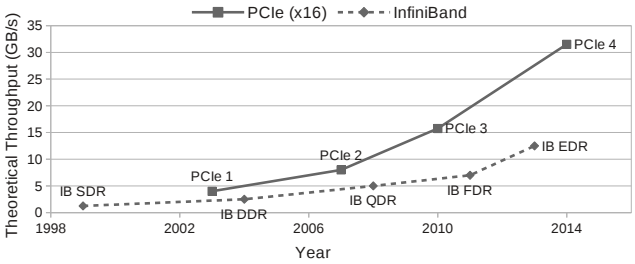

Fig. 1. Theoretical throughput in GB/s for several versions of PCI Express (16 lanes) compared with different InfiniBand connections.

The rest of the paper is organized as follows. In Section II we present some important software to virtualize GPUs, introducing in more detail rCUDA, the solution leveraged in this study. In Section III we compare the bandwidth and latency of PCIe with those of the networks considered in this work. In the next two sections we analyze the performance of several GPU applications using remote GPUs, again with different networks. We first use the NVIDIA CUDA Samples in Section IV; and then we evaluate some GPU-accelerated production applications in Section V. Finally, Section VI summarizes the main conclusions of our work.

II. Remote GPU Virtualization Solutions

Currently, programmers are assisted by CUDA [20] or OpenCL [21] in order to use GPUs for general-purpose computing (GPGPU). Although CUDA is a proprietary technology from NVIDIA, it is currently more widely used than the open standard OpenCL. In the context of CUDA, there exist several frameworks which grant access to GPUs installed in remote nodes to CUDA-based applications, such as GVirtuS [10], DS-CUDA [11], vCUDA [12], GViM [13], GridCuda [14], V-GPU [15], and rCUDA [8], [9].

All these frameworks are usually structured following a client-server distributed architecture, as illustrated in Figure 2. In this manner, the client middleware runs in the same cluster node as the application demanding GPGPU services, while the server middleware runs in the cluster node where the physical GPU resides. Ideally, the middleware client should present to the application the very same interface as the regular NVIDIA CUDA Runtime API [22]. A common course of action could then be:

1) The middleware client receives a CUDA request from the application.
2) The request is processed and forwarded to the framework server.
3) The server interprets the request and performs the required processing by accessing the real GPU.
4) The GPU completes the execution of the request and the middleware server sends the results back to the client.
5) The client forwards the results to the demanding application.

Note that this sequence of events may occur concurrently with similar ones from other applications, as GPUs are shared among several nodes in the cluster. To support this concurrent scenario, the remote GPU virtualization frameworks should provide the required mechanisms, such as managing independent GPU contexts for each application.

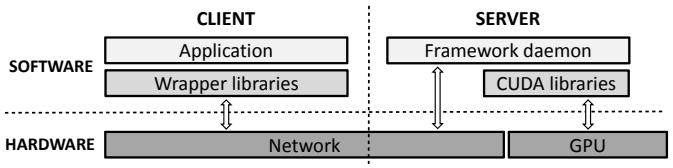

Fig. 2. Overview of the general architecture of remote GPU virtualization solutions.

Current virtualization frameworks present different characteristics. For example, rCUDA supports the last release of CUDA, version 5, and has specific communication modules for Ethernet and InfiniBand. On the other hand, GVirtuS supports CUDA 3.2 and is optimized for KVM virtual machines. DS-CUDA supports CUDA 4.1 and has specific communication libraries for InfiniBand. However, it has several limitations, such as not permitting copies with page-locked memory (also called *pinned* memory). V-GPU is a commercial tool which seems to support CUDA 4. Finally, vCUDA, GViM and GridCuda apparently support obsolete versions of CUDA (1.1 to 2.3). The only freely available virtualization solution that supports the last version of CUDA is the rCUDA framework, which motivates our adoption of this technology for our analysis.

A. rCUDA Communication Architecture

The internal architecture of rCUDA accommodates several underlying client-server communication technologies [23]. As illustrated in Figure 3, it consists of a modular, layered architecture which supports runtime-loadable network-specific communication libraries. rCUDA currently provides communication modules for Ethernet and InfiniBand.

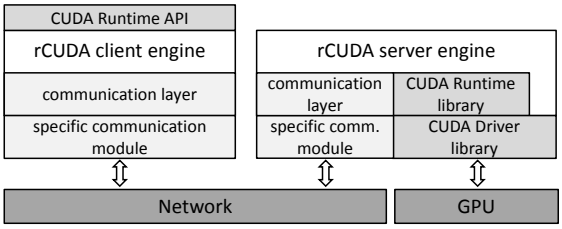

Fig. 3. rCUDA modular architecture.

Regardless of the specific communication technology used, data transfers between rCUDA clients and servers are pipelined in order to improve performance. For this purpose, rCUDA uses preallocated buffers of pinned memory [23], exploiting thus its higher throughput. As reported in [23], there is a relationship between the transfer size and the optimal pipeline block size. In general, small block sizes favor latency, as pipeline buffers are filled faster and data are moved earlier across the pipeline stages. However, they are inefficient for large data payloads, as large messages (i.e., large pipeline block sizes) are needed to exploit the peak throughput of PCIe and the network. Hence, the optimal block size should be chosen to be that as small as possible while still delivering the maximum PCIe and network throughput.

The optimal pipeline block size within rCUDA was already determined for TCP/IP-based communications over Ethernet (512KB) and also InfiniBand QDR (2MB) in [23]. In this paper we analyze GPU virtualization performance leveraging the new InfiniBand FDR version and, therefore, it is necessary to evaluate the optimal pipeline block size for this new technology.

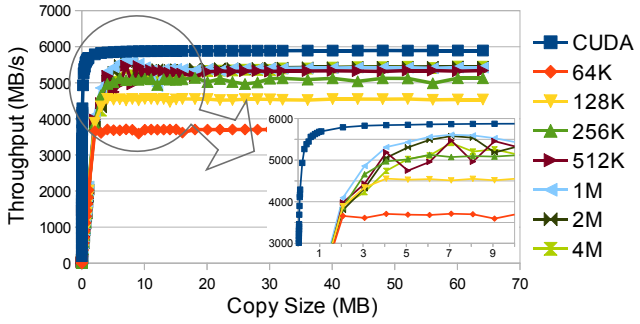

Fig. 4. Bandwidth test for copies from host to device with pinned host memory, using both native CUDA and the rCUDA framework over InfiniBand FDR, with different pipeline block sizes.

For that purpose, we have employed the `bandwidthTest` benchmark from the NVIDIA CUDA Samples [24]. Among other functionality, this test is capable of measuring CPU memory (also called *host* memory) to GPU memory (also referred to as *device* memory) copy bandwidth for pageable and pinned memory. In particular, we use the `shmoo` option, which measures bandwidth for a large range of values. As CUDA transfers between host memory and device memory are known to attain different peak transfer rates depending on the direction of the transfers (host to device or vice versa) and the type of host memory (pageable or pinned), for clarity, we will only present results for the host-to-device direction, using pinned host memory.

Figure 4 shows the results of this test using native CUDA and rCUDA over InfiniBand FDR for several pipeline block sizes. The results show that for copy sizes lower than 10MB, pipeline blocks of 1MB deliver the best bandwidths for rCUDA, while for transfers over 10MB, block sizes of 1MB, 2MB and 4MB obtain very similar results, very close to the maximum throughput. On average, rCUDA with a pipeline block size equal to 1MB yields the best performance, while being also the smallest block size. Consequently, hereafter we use this pipeline block size for our tests using rCUDA over InfiniBand FDR.

III. Impact of InfiniBand FDR on the Bandwidth and Latency of Remote GPU Usage

The performance of data transfers to/from the remote GPU is mainly influenced by the bandwidth and latency of the communication path. When transferring small amounts of data, latency is the most important factor while, when transferring large blocks, bandwidth is crucial. In this section we start our analysis of the influence of a high bandwidth network such as InfiniBand FDR on the performance of remote GPU virtualization. To do so, we compare the bandwidth and latency of PCIe when using CUDA, with those observed for rCUDA over three different network technologies, namely InfiniBand FDR, InfiniBand QDR and also Gigabit Ethernet.

A. Testbed System

The setup employed for the experiments carried out in this section, which will also be the same for the rest of the tests presented along this paper, consists of two servers, each with the following characteristics:

- Two Intel Xeon hexa-core processors E5-2620 (Sandy Bridge) operating at 2.00GHz
- 32 GB of DDR3 SDRAM memory at 1,333 MHz
- 1 Mellanox ConnectX-3 single-port InfiniBand Adapter
- CentOS Linux Distribution release 6.3, with Mellanox OFED 1.5.3 (InfiniBand drivers and administrative tools), CUDA 5.0 with NVIDIA driver 285.05, and rCUDA 4.0.1 (the latest stable release from February 2013)

Additionally, one of the nodes has an NVIDIA Tesla K20 GPU. On the other hand, both nodes are interconnected by a Gigabit Ethernet network with a Cisco SLM2014 switch, and also by an InfiniBand fabric. Two different Mellanox switches are leveraged for the InfiniBand fabric: an MTS3600 switch providing QDR compatibility, and an SX6025 for FDR compatibility. Depending on which of them is actually used, QDR or FDR features are leveraged.

B. Influence on Bandwidth

In this section we analyze the memory copy (memcopy) bandwidth between host memory and the device memory for several scenarios:

- Local GPU: memcopy bandwidth across PCIe (referred to as "CUDA" in the figures).
- Remote GPU: memcopy bandwidth for different networks: Gigabit Ethernet (rCUDA GbE), InfiniBand QDR (rCUDA QDR) and InfiniBand FDR (rCUDA FDR).
- Both the local and remote scenarios are evaluated using pageable host memory as well as pinned host memory.

In order to analyze the bandwidth, we employ the `bandwidthTest` benchmark from the NVIDIA CUDA Samples, with the `shmoo` option. Figure 5 presents the results of this test using pinned host memory, while Figure 6 illustrates the results for pageable host memory. As expected, rCUDA over GbE provides the worst results, with its bandwidth reaching a maximum of 113.1MB/s in both cases (pinned and pageable). When using pinned host memory, rCUDA over IB FDR achieves a substantial gain (46.01%) with respect to IB QDR. Furthermore, its throughput is very close to that obtained by regular CUDA over a local GPU, with a difference of 0.5GB/s (9.4%). Regarding the use of pageable host memory, rCUDA over IB renders a higher bandwidth than native CUDA with a local GPU. This is due to the fact that memory copies between rCUDA clients and remote GPUs are pipelined using preallocated buffers of pinned memory, as explained in Section II, exploiting thus the higher throughput of this type of memory.

In summary, the higher bandwidth of InfiniBand FDR allows remote GPU virtualization frameworks to experience a bandwidth similar to that of PCIe all across the entire path between the local application demanding GPGPU services and the remote GPU.

C. Influence on Latency

In order to analyze the impact of the improvements on interconnect latency, we employ a synthetic test similar to the

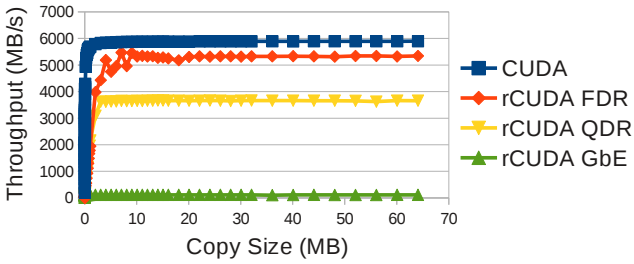

Fig. 5. Bandwidth test for copies from host to device with pinned host memory, using CUDA and the rCUDA framework over different networks.

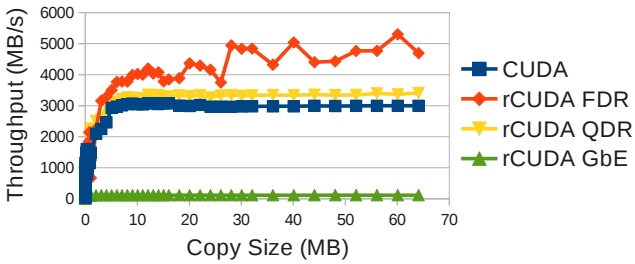

Fig. 6. Bandwidth test for copies from host to device with pageable host memory, using CUDA and the rCUDA framework over different networks.

previous bandwidth test, but using negligible volumes of data (concretely, from 1 to 64 bytes). Table I shows the results for this experiment, obtained from the average of 100 repetitions of each scenario. Again, GbE presents the worst performance. On the other hand, Table I also reveals that the use of rCUDA over IB FDR does not improve latency with respect to IB QDR. However, standard deviation values exhibit a more constant behavior for FDR, demonstrating a higher stability.

To determine how latency to remote GPU affects application performance, we have implemented one additional synthetic benchmark which also copies a small dataset (64 bytes). However, in this case, it performs a varying number of copies, from 100 to 102,400, doubling the number of copies in each iteration (i.e., 100, 200, 400, etc.). This resembles the behavior of applications, which typically will perform several sequential requests to the GPU along its execution. As Figure 7 shows, rCUDA over GbE performs very poorly starting from 25,000 iterations, whereas rCUDA over IB begins its degradation from 50,000 iterations. In general, the exact numbers show that IB QDR achieves slightly better results (an average of 5ms).

To sum up, the latency results show that, although the new InfiniBand FDR version noticeably improves performance when the application can benefit from its superior bandwidth, this new interconnect does not enhance performance when the application is sensitive to latency.

TABLE I. LATENCY TEST USING CUDA AND THE RCUDA FRAMEWORK OVER DIFFERENT NETWORKS.

	Time (μs)			
Copy size (bytes)	CUDA	rCUDA: FDR	QDR	GbE
1 (100-copy average)	11.62	50.73	50.34	130.63
2 (100-copy average)	11.56	50.53	50.49	130.05
4 (100-copy average)	11.59	50.32	50.32	130.59
8 (100-copy average)	11.55	50.69	50.26	130.68
16 (100-copy average)	11.56	50.71	50.06	130.50
32 (100-copy average)	11.67	50.64	50.22	133.03
64 (100-copy average)	11.71	50.87	50.12	135.18
Max. standard deviation	0.06	0.18	1.55	2.15

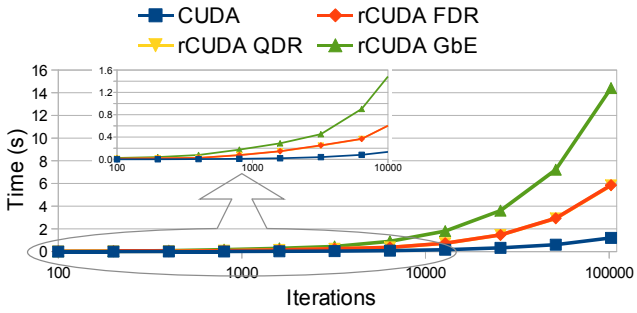

Fig. 7. Latency test varying the number of iterations for CUDA and the rCUDA framework over different networks. X-axis in logarithmic scale.

IV. NVIDIA CUDA SAMPLES

In this section we leverage the entire NVIDIA CUDA Samples suite, also referred to as CUDA SDK samples, to analyze remote GPU virtualization performance in a comprehensive way. The NVIDIA CUDA Samples contain simple code programs covering a wide range of applications and techniques using CUDA, which we consider useful for an initial study.

Figure 8a presents the normalized sample execution time for CUDA and also for rCUDA over IB FDR, IB QDR, and GbE. Times are normalized to those obtained with a local CUDA. The average of 10 repetitions is used, and the maximum Relative Standard Deviation (RSD) observed was 0.390 for sample `boxFilterNPP` (BN) when executed with CUDA, 0.147 for sample `transpose` (TA) in the case of rCUDA over FDR, 0.187 for sample `volumeRender` (VR) with rCUDA over QDR, and 0.563 for sample `simpleCUFFT` (FF) with rCUDA over GbE. In order to complete the data in Figure 8a, we have also measured the amount of bytes transferred between the rCUDA client and server (to take into account bandwidth), and the number of requests sent from the rCUDA client to the rCUDA server (to consider latency). The results for this experiment are displayed in Figure 8b.

Comparing rCUDA over GbE, we observe in these figures that most of the samples that exhibit a poor behavior (i.e., normalized time greater than or equal to 1.5) whether exceed 100MB of transfers (sent or received) or 1,000 requests to the rCUDA server. This is the case of 19 out of the 28 samples, namely `bandwidthTest` (BT), `simpleStreams` (IE), `alignedTypes` (AT), TA, FF, `smokeParticles` (SK), `simpleCUBLAS` (CB), `matrixMulCUBLAS` (MC), `cdpLUDecomposition` (LU), `Mandelbrot` (MB), `scan` (SC), `freeImageInteropNPP` (FI), `histEqualizationNPP` (HE), BN, `imageSegmentationNPP` (IS), `simpleDevLibCUBLAS` (LC), `conjugateGradient` (CG), `segmentationTreeThrust` (TT), and `interval` (IN).

Although sample `reduction` (RE) does not exceed these thresholds, it is close to both of them, 65MB of data and 749 requests sent to the rCUDA server, which explains the overhead. Samples `sortingNetworks` (SN) and `FDTD3d` (FD) transfer more than 100MB, but in these cases the overhead of using GbE is canceled by the long execution time of the samples, 9.93 and 17.47 seconds, respectively. For the same reason, sample `convolutionFFT2D` (F2), which takes almost 6 seconds, is not much affected by its large number of requests

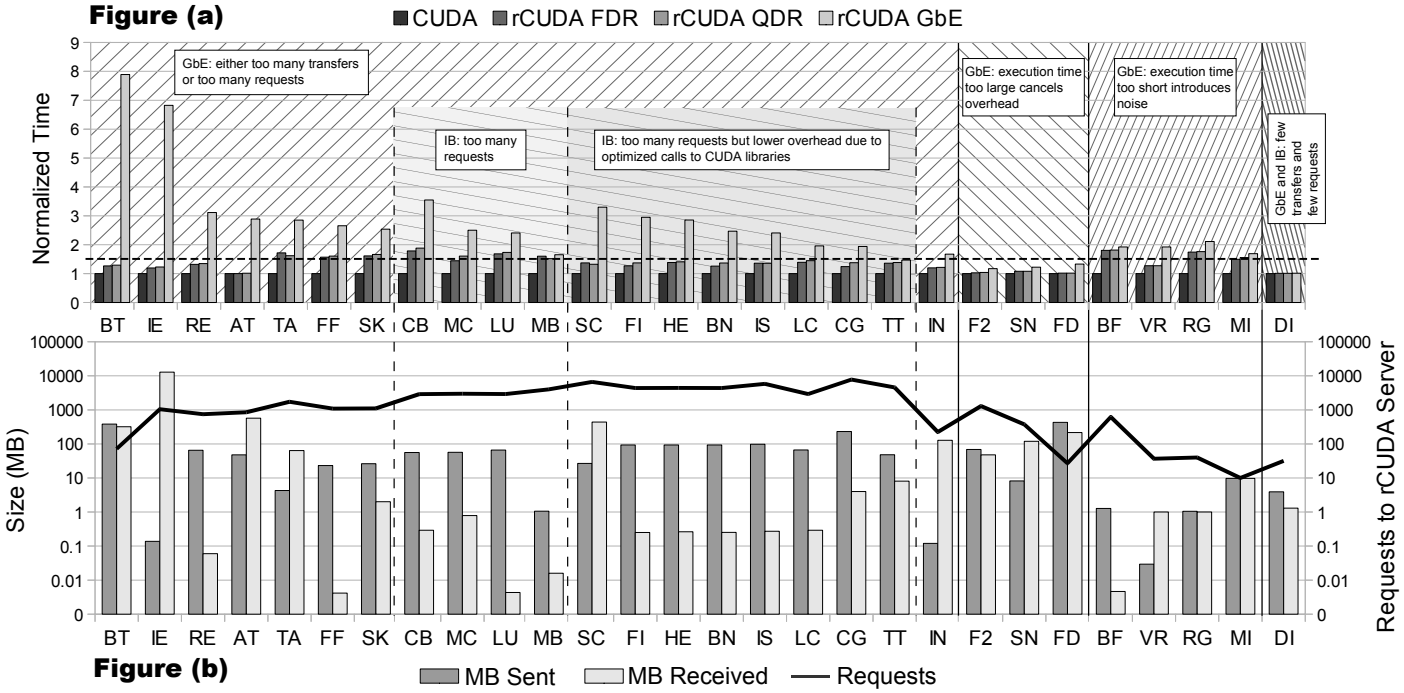

Fig. 8. (a) CUDA SDK samples normalized execution time using CUDA and rCUDA over different networks. (b) rCUDA profiling measurements for CUDA SDK samples. Primary Y-axis shows MB sent/received by samples to/from server when using the rCUDA framework. Secondary Y-axis presents requests sent to the rCUDA server. Both Y-axes in logarithmic scale.

(over 1,300). On the contrary, samples `bilateralFilter` (BF), VR, `recursiveGaussian` (RG), and `simpleMPI` (MI) are too short, less than 1 second, which distort their overheads, in spite of performing neither more than 100MB of transfers nor 1,000 requests.

Concerning rCUDA over IB, the figures reveal that the samples performing worst are TA, FF, SK, CB, MC, LU, MB, BF, RG, and MI. The reason for some of them (CB, MC, LU, and MB) is a large amount of requests to the rCUDA server (over 2,800). The execution time of the rest of samples is too short, less than 1 second, which distort their overheads, despite not going beyond this limit. Samples SC, FI, HE, BN, IS, LC, CG, and TT also have a large amount of requests, but these samples use CUDA libraries [25] and the vast majority of the requests are originated by the load of these libraries, which are optimized by rCUDA, resulting in a lower overhead. Considering the amount of bytes transferred in these samples, they are insufficient to penalize rCUDA over IB. Sample IE is the only one that transfers a considerable amount of data, over 12GB, but the network overhead is hidden by the long duration of the sample, nearly 20 seconds.

When comparing the different networking technologies, almost all the samples perform as expected: rCUDA over IB always runs faster than rCUDA over GbE. Sample `stereoDisparity` (DI) takes similar time to rCUDA over the different networks because it performs very few transfers and very few requests to the remote server. These two factors, added to its long duration, nearly 50 seconds, hide the network overhead. Regarding rCUDA over IB, the FDR executions are, in general, faster than the QDR runs, except for samples TA, MB, SC, and IS. These cases perform few transfers and a large number of requests to the rCUDA server, 1,729, 4,020, 6,669, and 6,048, respectively. This allows them to benefit from the slightly lower latency of QDR.

In conclusion, although the samples analyzed in this section are simple and usually involve few transfers and requests to the rCUDA server, they reveal the following insights:

- For rCUDA over GbE, transfers over 100MB and requests from 1,000 up drive to bad performance, because of the low bandwidth and high latency of this interconnect.
- For rCUDA over IB, these samples do not present enough transfers to arise IB throughput constraints. However, in terms of latency, samples with 2,800 or more requests start showing higher overheads.
- Compared to IB FDR, rCUDA over IB QDR benefits from its lower latency when samples imply thousands of requests to the rCUDA server. Nevertheless, in general, FDR takes advantage of its higher bandwidth, overcoming QDR in 87.5% of the studied samples.
- The actual execution time of the samples introduce a considerable noise in this study and modify the thresholds mentioned above concerning transfers and requests. Thus, longer samples minimize the impact of these limits, while shorter ones maximize it.

V. Influence of InfiniBand FDR on Production Applications

In order to study more thoroughly the influence of the network on remote GPU virtualization, in this section we analyze some production codes selected from the NVIDIA Popular GPU-Accelerated Applications [26].

A. CUDASW++

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

Figure 9 shows CUDASW++ execution time for queries of different sequence lengths using CUDA and rCUDA over different networks. The average of 10 repetitions is presented, and the maximum RSD observed was 0.019 for a sequence of length 222 when executed with CUDA, 0.010 for a sequence of length 464 in the case of rCUDA over FDR, 0.010 for a sequence of length 144 with rCUDA over QDR, and 0.009 for a sequence of length 657 with rCUDA over GbE. The figure also presents the overhead of using rCUDA: over IB the execution time is very close to that of CUDA. FDR and QDR introduce average overheads of 0.67% and 1.37%, respectively. For rCUDA over GbE, the average overhead is significantly higher though (21.88%).

The reason for rCUDA over IB performing only slightly worse than the native CUDA is the small number of transfers and the reduced number of requests done by the application to the rCUDA server (see Figure 10). For GbE, this small transfer size (around 160MB) is enough to penalize rCUDA because of the low bandwidth of this technology. With respect to the different IB networks, QDR presents an average overhead 0.7% higher than FDR.

We can also observe that longer query sequences reduce the overhead introduced by rCUDA. Figure 11 reveals that this is due to the fact that the time spent in transfers (i.e., time spent in memory copies between host memory and the device memory, also referred to as *CUDA memcopy*) remains always the same for all the query lengths, but the time employed by computations (i.e., time employed by CUDA kernels) increases with the query sequence length. Performing more computations helps 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 the transfers across the network.

B. GPU-BLAST

GPU-BLAST [28] has been designed to accelerate the gapped and ungapped protein sequence alignment algorithms of the NCBI-BLAST (http://www.ncbi.nlm.nih.gov) implementation using GPUs. It is integrated into the NCBI-BLAST code and produces identical results. We utilize release 1.1 in the next experiments, where we have followed the installation instructions for (1) sorting a database, and (2) creating a GPU database. We then use the query sequences which come with the application package to search the database.

Figure 12 depicts the GPU-BLAST execution time for queries of different sequence lengths using CUDA and rCUDA over the three networks. The average of 10 repetitions is presented, and the maximum RSD observed was 0.207 for a sequence of length 100 when executed with CUDA, 0.031 for a sequence of length 200 in the case of rCUDA over FDR, 0.051 for a sequence of length 700 with rCUDA over QDR, and 0.012 for a sequence of length 1400 with rCUDA over GbE.

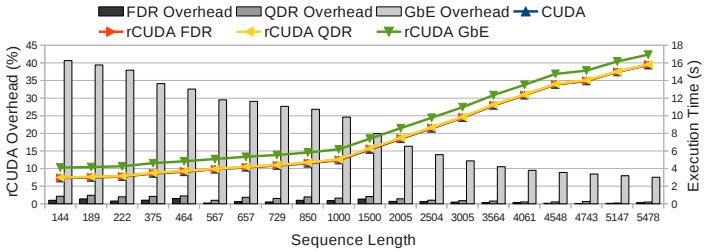

Fig. 9. CUDASW++ execution time for queries of different sequence lengths, using CUDA and rCUDA over different networks. Primary Y-axis shows rCUDA's overhead and secondary Y-axis execution time.

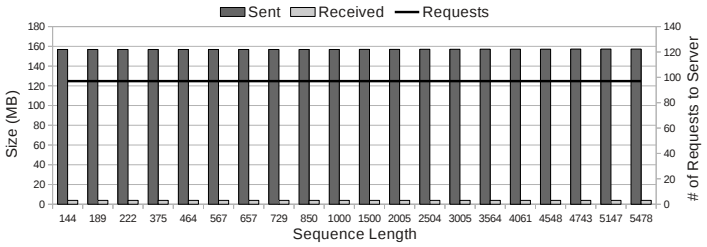

Fig. 10. rCUDA profiling measurements for CUDASW++ executing queries of different sequence lengths. Primary Y-axis shows MB sent/received by CUDASW++ to/from server when using the rCUDA framework. Secondary Y-axis presents requests sent to the rCUDA server.

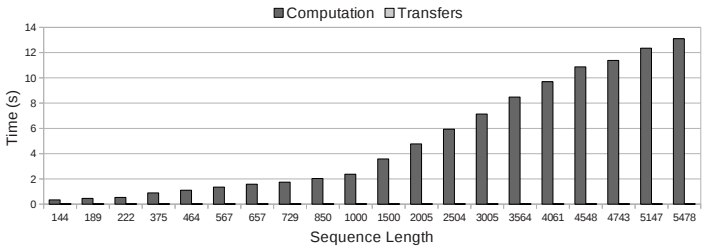

Fig. 11. NVIDIA profiling result for CUDASW++ executing queries of different sequence lengths. In particular, time employed by computations (i.e., CUDA kernels) and memory transfers (i.e., CUDA memcopy).

From our results we also extract that the average overhead of using rCUDA is 7.07%, 8.63%, and 113.71% for IB FDR, IB QDR, and GbE, respectively. Data transfers over 1.2GB (see Figure 13) hurt performance for rCUDA over GbE. Concerning rCUDA over IB, QDR presents an average overhead 1.56% higher than FDR.

As it was the case for CUDASW++, Figure 14 illustrates that the time spent in transfers is constant for all the queries with GPU-BLAST and only the time required by computations varies. Again, we can observe that rCUDA's overhead decreases as computation time increases. Figure 14 also reveals a peak in time spent in GPU computations when running GPU-BLAST with a sequence of length 600, which explains a similar peak in Figure 12 for this sequence length.

C. LAMMPS

LAMMPS [29] is a classic molecular dynamics simulator which can be used to model atoms or, more generically, as a parallel particle simulator at the atomic, mesoscopic, or continuum scale. For the tests below, we use the release from Feb. 19, 2013, and benchmarks `in.eam` and `in.lj` installed with the application. We run the benchmarks with one

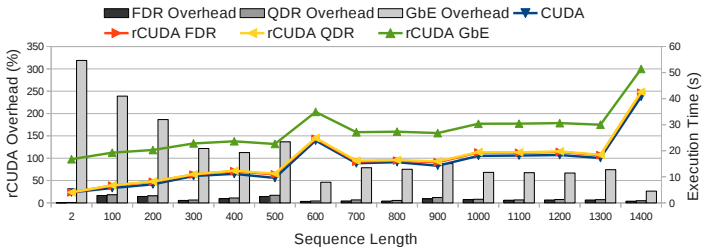

Fig. 12. GPU-BLAST execution time for queries of different sequence lengths, using CUDA and rCUDA over different networks. Primary Y-axis shows rCUDA's overhead, while secondary Y-axis represents execution time.

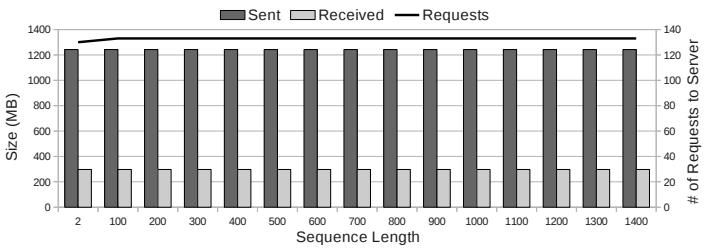

Fig. 13. rCUDA profiling measurements for GPU-BLAST executing queries of different sequence lengths. Primary Y-axis shows MB sent/received by GPU-BLAST to/from server when using the rCUDA framework. Secondary Y-axis presents requests sent to the rCUDA server.

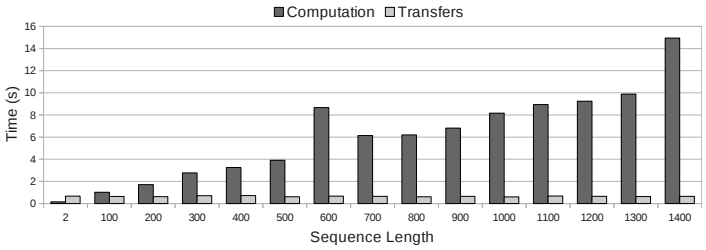

Fig. 14. NVIDIA profiling result for GPU-BLAST executing queries of different sequence lengths. In particular, time employed by computations (i.e., CUDA kernels) and memory transfers (i.e., CUDA memcopy).

processor and scaling by a factor of 5 in all three dimensions (i.e., a problem size of 4 million atoms).

Table II reports the execution time for these benchmarks using CUDA and rCUDA over the three networks, and Table III shows rCUDA's overhead for the same tests. The average of 10 executions is presented, and the maximum RSD observed was 0.013 for benchmark `in.lj`, when executed with rCUDA over GbE. 0.005 for benchmark `in.eam` in the case of rCUDA over FDR, 0.009 for benchmark `in.eam` with rCUDA over QDR, and 0.004 for benchmark `in.lj` with CUDA. Once again, rCUDA over GbE exhibits a poor performance due to the large transfers involved and the huge number of requests sent to the rCUDA server, as shown in Figure 15. For rCUDA over IB, QDR has an average overhead 2.39% higher than FDR. Despite the large amount of requests, which could help QDR in terms of latency, these experiments reveal that FDR's better bandwidth has more influence than its worse latency when data transfers are of a considerable size, as also pointed out in previous sections.

In this application, the benchmark presenting greater overhead, `in.eam`, spends also significantly more time in computations (see Figure 16) than benchmark `in.lj`, which shows a lower overhead. Apparently, this behavior does not obey

TABLE II. LAMMPS EXECUTION TIME FOR BENCHMARKS IN.EAM AND IN.LJ, SCALED BY A FACTOR OF 5 IN ALL THREE DIMENSIONS

LAMMPS benchmark	Execution time (s) CUDA	rCUDA FDR	rCUDA QDR	rCUDA GbE
in.eam	52.33	56.36	57.60	102.09
in.lj	36.39	38.02	38.90	79.37

TABLE III. RCUDA OVERHEAD FOR THE BENCHMARKS IN.EAM AND IN.LJ, SCALED BY A FACTOR OF 5 IN ALL THREE DIMENSIONS

LAMMPS benchmark	Overhead (%) rCUDA FDR	rCUDA QDR	rCUDA GbE
in.eam	7.71	10.07	95.10
in.lj	4.50	6.90	118.12

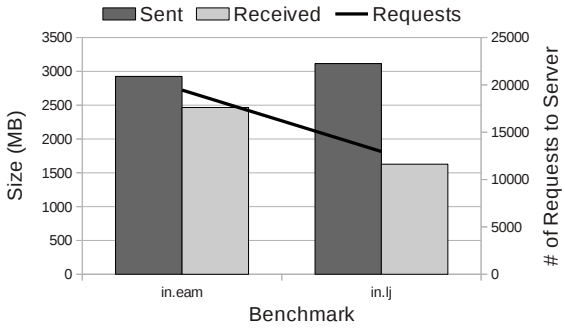

Fig. 15. rCUDA profiling measurements for LAMMPS executing benchmarks in.eam and in.lj, scaled by a factor of 5 in all three dimensions. Primary Y-axis shows MB sent/received by LAMMPS to/from server when using the rCUDA framework. Secondary Y-axis presents requests sent to the rCUDA server.

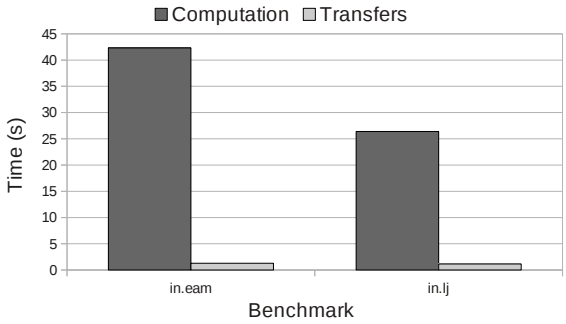

Fig. 16. NVIDIA profiling result for LAMMPS executing benchmarks in.eam and in.lj, scaled by a factor of 5 in all three dimensions. Time employed by computations (i.e., CUDA kernels) and by memory transfers (i.e., CUDA memcopy) is shown.

the conclusions obtained in previous sections with respect to the time employed by computations and rCUDA's overhead. The explanation lies in the fact that here, unlike in preceding experiments, the number of bytes sent is almost the same for both benchmarks, but the number of bytes received and requests to the server are significantly higher for `in.eam`, thus making a higher use of the network fabric than `in.lj`.

D. Summary

Table IV summarizes the most important results for the applications under study. In this case, we have analyzed applications with low, medium, and high volumes of data transfers. This analysis reveals that rCUDA over GbE has a high overhead independently of the amount of transfers. In contrast, rCUDA over IB exploits the much higher throughput

than GbE, exposing a very small overhead for applications with a low amount of transfers. For applications that involve a moderate to high volume of transfers, the overhead of using rCUDA over IB depends on the time spent in computations. Thus, if the amount of computations is enough to compensate for the extra time spent transferring data across the network, then the overhead of rCUDA over IB is very low. Otherwise, the overhead becomes significant.

TABLE IV. SUMMARY OF RCUDA OVERHEAD USING DIFFERENT NETWORKS, RELATED WITH AVERAGE NUMBER OF RCUDA TRANSFERS AND REQUESTS TO RCUDA SERVER, FOR THE STUDIED APPLICATIONS.

Application	rCUDA transfers	rCUDA requests	rCUDA overhead (%) FDR	QDR	GbE
CUDASW++	∼160MB	∼100	0.67	1.37	21.88
GPU-BLAST	∼1200MB	∼130	7.07	8.63	113.71
LAMMPS	∼3000MB	∼16000	6.10	8.49	106.61

Concerning the two different IB versions examined in these tests, QDR and FDR, it appears that as the size of transfers increases, FDR's higher throughput is more important. In this way, for the applications considered in this study, QDR average overhead in comparison to FDR grows together with the level of transfers: 0.7%, 1.56%, and 2.39%, for low, medium, and high volumes of transfers, respectively.

VI. CONCLUSIONS

Remote GPU virtualization is rising interest in the HPC and datacenter community given the flexibility it provides to cluster administrators regarding acquisition, space, and maintenance costs as well as the corresponding energy savings. However, despite the many benefits it provides, remote GPU virtualization also introduces some overheads due to the virtualization framework and also the network fabric connecting the cluster node running the accelerated application and the node owning the actual GPU.

In this paper we have analyzed how the current bandwidth matching between PCIe 2.0 and InfiniBand FDR influences the performance of remote GPU virtualization, using both synthetic tests and real GPU-accelerated applications. Synthetic tests have revealed that FDR achieves a substantial gain (over 40% with respect to the previous InfiniBand QDR version) in terms of bandwidth to/from the remote GPU. This bandwidth gain, when incorporated into the context of a GPU-accelerated application, which performs computations in addition to transfers between main memory and GPU memory, reduces overhead up to 2.39% with respect to InfiniBand QDR, clearly showing that the new interconnect not only serves traditional applications running in the cluster but also remotely accelerated ones.

Nevertheless, it is also important to consider that GPUs will support PCIe 3.0 soon, probably doubling their actual maximum bandwidth to the local GPU. This upgrade to PCIe 3.0 will increase again the performance gap between the intra-node and the inter-node interconnects, leading to higher overheads when using GPUs remotely. This overhead, according to the analysis carried out in this paper, will mainly depend on the amount of information transferred between main memory and GPU memory and the requests sent to the GPU server. However, other factors apart from bandwidth, such as the time spent in GPU computations or how efficiently is the application designed, will be key to achieve good performance.

ACKNOWLEDGMENTS

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

REFERENCES

[1] A. Gaikwad *et al.*, "GPU based sparse grid technique for solving multidimensional options pricing PDEs," in *WHPCF*, 2009.

[2] D. P. Playne *et al.*, "Data parallel three-dimensional Cahn-Hilliard field equation simulation on GPUs with CUDA," in *PDPTA*, 2009.

[3] E. H. Phillips *et al.*, "Rapid aerodynamic performance prediction on a Cluster of graphics processing units," in *AIAA*, 2009.

[4] S. Barrachina *et al.*, "Exploiting the capabilities of modern GPUs for dense matrix computations," *CPE*, 2009.

[5] Y. C. Luo and R. Duraiswami, "Canny edge detection on NVIDIA CUDA," in *CVPRW*, 2008.

[6] R. J. O. Figueiredo *et al.*, "Guest editors' introduction: resource virtualization renaissance," *IEEE Computer*, 2005.

[7] J. Enos *et al.*, "Quantifying the impact of GPUs on performance and energy efficiency in HPC clusters," in *IGCC*, 2010.

[8] A. J. Peña *et al.*, "An efficient implementation of GPU virtualization in high performance clusters," in *Euro-Par Workshops*, 2009.

[9] ——, "Performance of CUDA virtualized remote GPUs in high performance clusters," in *ICPP*, 2011.

[10] G. Giunta *et al.*, "A GPGPU transparent virtualization component for high performance computing clouds," in *Euro-Par*, 2010.

[11] M. Oikawa *et al.*, "DS-CUDA: a middleware to use many GPUs in the cloud environment," in *SC*, 2012.

[12] L. Shi *et al.*, "vCUDA: GPU accelerated high performance computing in virtual machines," in *IPDPS*, 2009.

[13] V. Gupta *et al.*, "GViM: GPU-accelerated virtual machines," in *HPCVirt*, 2009.

[14] T.-Y. Liang *et al.*, "GridCuda: a grid-enabled CUDA programming toolkit," in *WAINA*, 2011.

[15] Zillians, Inc. (2013) V-GPU: GPU virtualization. [Online]. Available: http://www.zillians.com/vgpu.

[16] J. Kim *et al.*, "SnuCL: an OpenCL framework for heterogeneous CPU/GPU clusters," in *ICS*, 2012.

[17] P. Kegel *et al.*, "dOpenCL: towards a uniform programming approach for distributed heterogeneous multi-/many-core systems," in *IPDPSW*, 2012.

[18] S. Xiao *et al.*, "VOCL: an optimized environment for transparent virtualization of graphics processing units," in *InPar*, 2012.

[19] A. Barak *et al.*, "A package for OpenCL based heterogeneous computing on clusters with many GPU devices," in *CLUSTER*, 2010.

[20] NVIDIA, *NVIDIA CUDA C Programming Guide 5.0*, 2012.

[21] Khronos OpenCL Working Group, *OpenCL 1.2 Specification*, 2011.

[22] NVIDIA, *NVIDIA CUDA API Reference Manual 5.0*, 2012.

[23] A. J. Peña, "Virtualization of accelerators in high performance clusters," Ph.D. dissertation, University Jaume I, Castellón, Spain, 2013.

[24] NVIDIA, *NVIDIA CUDA Samples 5.0*, 2012.

[25] ——, *NVIDIA CUDA Libraries*, 2012.

[26] ——, *Popular GPU-Accelerated Applications Catalog*, 2012.

[27] Y. Liu *et al.*, "CUDASW++ 3.0: accelerating Smith-Waterman protein database search by coupling CPU and GPU SIMD instructions," *BMC Bioinformatics*, 2013.

[28] P. D. Vouzis *et al.*, "GPU-BLAST: using graphics processors to accelerate protein sequence alignment," *Bioinformatics*, 2011.

[29] Sandia National Labs. (2013) LAMMPS Molecular Dynamics Simulator. [Online]. Available: http://lammps.sandia.gov/