Parallel Computing 40 (2014) 574–588

Contents lists available at ScienceDirect

Parallel Computing

journal homepage: www.elsevier.com/locate/parco

A complete and efficient CUDA-sharing solution for HPC clusters

Antonio J. Peña [a,*,1], Carlos Reaño [b], Federico Silla [b], Rafael Mayo [c], Enrique S. Quintana-Ortí [c], José Duato [b]

[a] MCS, Argonne National Laboratory, Argonne, IL 60439, USA
[b] DISCA, Universitat Politècnica de València, 46022 Valencia, Spain
[c] DICC, Universitat Jaume I, 12071 Castellón, Spain

ARTICLE INFO

Article history:
Received 10 July 2013
Received in revised form 3 September 2014
Accepted 30 September 2014
Available online 8 October 2014

Keywords:
Graphics processors
Virtualization
High performance computing
Clusters

ABSTRACT

In this paper we detail the key features, architectural design, and implementation of rCUDA, an advanced framework to enable remote and transparent GPGPU acceleration in HPC clusters. rCUDA allows decoupling GPUs from nodes, forming pools of shared accelerators, which brings enhanced flexibility to cluster configurations. This opens the door to configurations with fewer accelerators than nodes, as well as permits a single node to exploit the whole set of GPUs installed in the cluster. In our proposal, CUDA applications can seamlessly interact with any GPU in the cluster, independently of its physical location. Thus, GPUs can be either distributed among compute nodes or concentrated in dedicated GPGPU servers, depending on the cluster administrator's policy. This proposal leads to savings not only in space but also in energy, acquisition, and maintenance costs. The performance evaluation in this paper with a series of benchmarks and a production application clearly demonstrates the viability of this proposal. Concretely, experiments with the matrix–matrix product reveal excellent performance compared with regular executions on the local GPU; on a much more complex application, the GPU-accelerated LAMMPS, we attain up to 11x speedup employing 8 remote accelerators from a single node with respect to a 12-core CPU-only execution. GPGPU service interaction in compute nodes, remote acceleration in dedicated GPGPU servers, and data transfer performance of similar GPU virtualization frameworks are also evaluated.

1. Introduction

Recent advances in hardware and software for graphics processing units (GPUs) have unleashed the so-called GPGPU (general-purpose computing on GPU) era: namely, the use of graphics accelerators for solving general-purpose tasks, which are different from the graphical workloads these devices were initially designed for. This innovative GPU usage has been favored because of their extreme computational power (for certain applications) and affordable price, as well as the improvements in their programmability with the introduction of standard interfaces such as CUDA [1] or OpenCL [2].

* Corresponding author. Tel.: +1 630 252 7928.
E-mail address: apenya@mcs.anl.gov (A.J. Peña).
[1] The contributions of this author to this paper were performed while he was affiliated to Universitat Politècnica de València and Universitat Jaume I de Castellón.

http://dx.doi.org/10.1016/j.parco.2014.09.011
0167-8191/© 2014 Elsevier B.V. All rights reserved.

As a result, computing clusters are adopting these devices to accelerate compute-intensive parts for an increasing number of applications. For instance, two of the top ten supercomputers of the Top500 list (November 2013) [3] are equipped with GPU accelerators.

On the other hand, energy consumption has become a hot topic in the design of supercomputers and data centers [4–7]. In this regard, the use of GPUs exerts a great impact on both energy consumption and total cost of ownership (TCO) of the system, since high-end GPUs may well increase energy consumption of a high-performance computing (HPC) server over 30% and TCO around 50%.

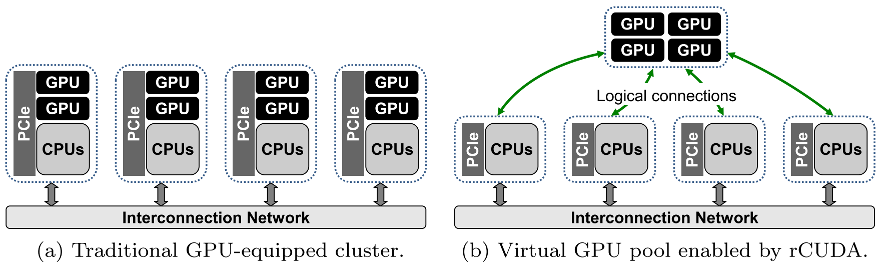
Strategies to reduce costs and increase flexibility without affecting performance are highly appealing. System virtualization solutions, such as Xen [8] or KVM [9], may yield significant energy savings, since they increase resource utilization by sharing a given computer among several users, reducing the required amount of instances of that system. Our approach pursues the virtualization of single GPU devices instead of complete computers, in order to increase flexibility and render energy and cost benefits. A GPU virtualization solution can enable seamless and concurrent usage of remote GPU devices. Our proposal, rCUDA (remote CUDA) permits a reduction in the number of GPUs of a cluster, thus yielding considerable energy and resource savings, while efficiently sharing the existing GPUs across all the nodes in the system. On the other hand, applications are not further limited by the number of GPUs physically attached to the node where they are running, since rCUDA enables applications running on a node to gain access to any amount of the GPUs available in the cluster.

To provide remote GPU acceleration, rCUDA creates virtual CUDA devices on those machines without a local GPU (see Fig. 1). These virtual devices represent physical GPUs located on a remote host offering GPGPU services. Thus, all the compute nodes are able to concurrently share the whole set of CUDA accelerators present in the cluster (referred to as a *globally shared GPU pool*), independently of which nodes the GPUs are physically attached to. Furthermore, since the number of processes concurrently sharing a GPU is limited by the amount of device memory (both in the traditional CUDA local solution and in rCUDA), no new scalability concerns within a GPGPU server are imposed.

rCUDA is not intended to bring any benefit to those supercomputers targeting the Top500 list, since this ranking tends to promote configurations with multiple GPUs per node. Similarly, the use of remote accelerators may be unfeasible for certain applications where high-bandwidth and/or low-latency GPU communications lie in the critical path to attain the desired performance, as opposed to others where the GPU computational power is the dominant factor. rCUDA is particularly useful in production clusters, where applications tend to alternate CPU- and GPU-compute periods, leading conventional configurations with one or more GPUs per node to experience low GPU utilization. Furthermore, the GPU-pool proposal adds flexibility to GPU-enabled clusters by decoupling GPUs from nodes, hence facilitating GPU upgrades, additions, and, in general, GPU maintenance. This is the case especially when GPGPU-dedicated servers are deployed as "GPU boxes" connected to the network (like disk servers). Actually, the idea of "GPU boxes" turns to be useful when these accelerators must be added to a GPU-less cluster. In this scenario, the new hardware components usually do not fit into the existing nodes. In these conditions, rCUDA provides a useful solution because it allows a "GPU box" to be attached instead of having to replace the existing hardware, thus lowering the cost of introducing GPGPU capabilities into the computing facility. In summary, these use cases are beneficial in many scenarios, ranging from research clusters to datacenters using GPUs to accelerate applications such as those dealing with datamining [10,11].

In previous work we either presented an early prototype of our rCUDA virtualization solution or simple performance results. However, basically no detail about the internals of rCUDA was introduced. In this paper we present a mature version of our GPU virtualization solution, with support for CUDA 5.5, with the following major properties:

1. An enhanced and up-to-date GPGPU remote virtualization solution that extends some of the features of CUDA without changing the original application programming interface (API).
2. An efficient architecture for data transfers involving remote GPUs.
3. An optimized implementation of the communications of the rCUDA remote GPU virtualization solution for InfiniBand (IB) interconnects.



(a) Traditional GPU-equipped cluster. (b) Virtual GPU pool enabled by rCUDA.

Fig. 1. rCUDA GPU pool compared with traditional view.

To illustrate these features, we provide an experimental evaluation of remote GPU acceleration including a correctness evaluation, several benchmarks, a production application evaluation (LAMMPS), GPGPU service interaction in compute nodes, dedicated GPGPU server (*GPU box*) evaluations, and a performance comparison with other existing GPU virtualization frameworks. To the best of our knowledge, this paper includes the most extensive remote GPGPU acceleration evaluation to date, demonstrating the feasibility of this proposal in HPC clusters using current technologies.

The rest of this paper is organized as follows. In Section 2 we review the related work. Section 3 describes the main features of rCUDA and its design and key implementation details. Section 4 focuses on the new InfiniBand communications module. In Section 5, our proposal is evaluated. Conclusions and future work directions are presented in Section 6.

2. Related work

GPU virtualization and sharing have been approached from both the hardware and software perspectives. In the first case, a range of commercial solutions allow multiple servers in a rack to share several GPUs, hence making it possible to use a cluster configuration with fewer GPUs than nodes. These solutions include, for example, the NextIO's vCORE product series [12] by PCI Express (PCIe) virtualization [13]. In this case, up to 8 servers can share a reduced number of GPUs within a two-meter distance. Unfortunately, these commercial solutions do not allow multiple nodes to concurrently access the same GPU, since the accelerators need to be assigned exclusively to a specific node at a time. Moreover, these proprietary hardware-supported solutions are often expensive for moderate to large-scale configurations.

One key difficulty for GPU virtualization is that GPUs, unlike devices such as storage or network controllers, lack a standard low-level interface. Furthermore, the low-level protocols used to drive GPUs are proprietary and strictly closed by GPU vendors. Consequently, the virtualization boundary has been commonly placed on top of open high-level APIs, such as Microsoft's Direct3D [14] and OpenGL [15] for graphics acceleration or CUDA and OpenCL for GPGPU, leading to two markedly different types of virtualization solutions: those intended for graphics and those for GPGPU.

2.1. GPU virtualization for graphics processing

Although there are conceptual similarities between these works pursuing GPU virtualization for graphics processing and our approach (i.e., GPU virtualization by means of high-level API interception), even when dealing with the virtualization of the same device we found that the intrinsics of both graphics and GPGPU APIs differ significantly, since graphics APIs have to deal with graphics-related issues, such as flickering, object interposition (with the optimization opportunities this brings), or graphics redirection. Conversely, our research focuses mainly on general-purpose computing, discarding graphical representation issues.

2.2. GPU virtualization for GPGPU

Like rCUDA, a collection of approaches pursue the virtualization of the CUDA Runtime API. All these solutions feature a distributed middleware composed of two parts: the front-end, installed on client systems, and the back-end, executed by the hosts offering GPGPU services and with direct access to the physical GPUs.

2.2.1. Virtualization frameworks for cloud/virtual machines

Although the constraints that these solutions face differ from those of our target systems—such as those derived from the virtualized environment and the limited network performance—the principles of operation of these GPGPU virtualization solutions are markedly similar to those of rCUDA. We next review the related work in this field.

vCUDA [16] implements an unspecified subset of the obsolete CUDA Runtime version 1.1. It employs XML-RPC for the application-level communications, which, as the experiments in [16] show, causes a considerable negative impact on the overall performance of this solution as a result of the time spent in the encoding/decoding stages of the communication protocol.

GViM [17] is specifically designed to expose GPGPU capabilities to Xen VMs. It uses Xen-specific mechanisms for the communication between its middleware components, including shared-memory buffers, which enhance communications between user and administrative domains at the expense of targeting a highly specific environment. This solution, also based on CUDA 1.1, does not seem to implement the whole CUDA Runtime API.

gVirtuS [18] covers only a subset of the CUDA Runtime 3.2. For example, it lacks over 50% of the memory management functions of this version.

DS-CUDA [19] is a similar remote virtualization tool, limited to provide multiple remote virtualized GPUs to a single computing node. It also incorporates low-level IB-based communications, though the implementation intrinsics are not detailed. An interesting additional feature of this software for its specific target environment is that it implements fault-tolerance capabilities performing redundant computations in multiple GPUs. A subset of CUDA 4.1 is reported as being covered, but neither asynchronous operations nor memory copies larger than 32 MB are supported.

The work in [20] introduced GPU kernel consolidation into their GPGPU virtualization framework to optimize kernel workloads in cloud environments.

2.2.2. Other frameworks

VGPU [21] is a commercial tool with no detailed information available. Furthermore, no public version is released that can be used for testing and comparison. GridCuda [22] is a similar tool aimed at providing remote access to GPUs in a grid environment; although this work mentions interesting features, these are neither detailed nor evaluated in their experiments. On the other hand, VCL [23], VOCL [24], SnuCL [25], and dOpenCL [26] are virtualization tools targeting the OpenCL API.

2.2.3. rCUDA

Compared with other virtualization solutions, rCUDA is the only production ready framework targeting HPC cluster environments. The use of specific, highly tuned, low-level IB communications unleashes all the potential of this technology for our target environment, as will be shown later.

In previous work [27] we introduced the idea underlying rCUDA, along with an overview of a simple proof-of-concept implementation and early results. We explored the support for asynchronous operations along with the energy-saving potential [28], and we derived models to estimate the performance of remote GPU virtualization on different networks [29,30]. We explored the use of rCUDA to expose GPGPU capabilities to virtual machines [31]. An automatic converter from CUDA extensions to plain C (CU2rCU) was described in [32], and the influence of the new InfiniBand FDR technology on the performance of GPU remoting was assessed in [33]. In [34] we explore the usage of remote accelerators from clients featuring low-power processors.

This paper provides an extensive performance evaluation of remote GPU acceleration in computing clusters using rCUDA, exploring novel aspects such as GPU service interaction in compute nodes or GPGPU service from dedicated GPU boxes. Also presented is a performance comparison with other publicly available solutions, demonstrating our remote GPU acceleration proposal to be feasible for production computing clusters using current technologies. In addition, we detail the design and implementation choices leading rCUDA to become a highly efficient solution that attains the aforementioned objectives.

2.3. NVIDIA's GPU virtualization technology

The previous approaches are based on the interception of a particular API. Nevertheless, vendors can virtualize the entire GPU, as NVIDIA intends with the new VGX [35]. This technology, announced for the Kepler GPU architecture, includes hardware and software support to efficiently virtualize a whole GPU and share it among different virtual machines running on the same node. Although rCUDA is also suitable for VM environments, this type of intranode virtualization solution should not be confused with the remote virtualization solution that rCUDA and some of the other proposals provide, which involves dealing with distributed-memory scenarios. Furthermore, this new technology, intended mainly for desktop virtualization, requires specific hardware features, and thus it will be available only for specialized GPUs, limiting its applicability.

3. rCUDA: features, architectural design, and implementation

By leveraging rCUDA, applications can transparently access CUDA compatible accelerators installed in any node of the cluster as if those were directly connected to a local PCIe port.

In our solution, remote GPUs are virtualized devices made available by an interception library that replaces the CUDA Runtime (provided by NVIDIA as a shared object). Basically, the rCUDA library forwards the CUDA API calls to a remote server equipped with a GPU, where the corresponding requests are executed. From the user/programmer point of view, there is no difference in the behavior of the CUDA calls, apart from a possible increase of the execution time caused by the network transmission latency. On the other side, the rCUDA server middleware is configured as a daemon that runs in those nodes offering GPGPU services controlling the local GPU(s) and sending back the results from the requests of the calling applications. Fig. 2 depicts the different components that form rCUDA.

The virtual CUDA devices presented by rCUDA feature the same capabilities as the physical devices they represent. An exception is the "zero-copy," a mechanism to directly access host memory from GPU kernels, which is not currently supported by rCUDA devices (this feature is expected to be incorporated in future versions by tracking the `cudaHostAlloc` calls with the flag `cudaHostAllocMapped` set, and managing the corresponding data transfers between clients and servers on the different epochs). This should not pose a strong inconvenient, however, since applications are encouraged to check

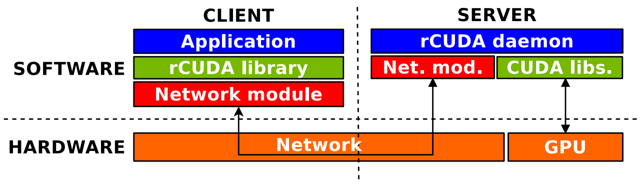


Fig. 2. rCUDA components.

device capabilities prior to using them, and consequently these are expected to provide alternative mechanisms for the different GPU capabilities they may find. Graphics interoperability is also not supported in this version of rCUDA and will be incorporated in future releases of the framework. Also, in our current implementation the output of a "`printf`" call made within GPU kernels is not forwarded to rCUDA clients but dumped into the server logs instead.

3.1. Client side

The rCUDA interception library is installed on those nodes requiring GPU-based acceleration capabilities. This library intercepts, processes, and forwards the CUDA Runtime calls from the CUDA-accelerated applications (clients) to the rCUDA server.

When loaded, the client library automatically tries to establish a connection with the server(s) demanded by the application. Hence, the asymptotic complexity of the communications of the startup phase on the client side is $O(S)$, where S is the number of GPU servers that the application will use. The servers will be specified by the job scheduler described in Section 3.6, although for the experiments presented in this document these have been set manually. When the library is unloaded, the connection is automatically closed, and the library resources are released.

For each CUDA call, the following tasks are performed by the rCUDA client: (1) conduct local checks (function dependent), for example, determining whether the requested amount of data to transfer is positive or the pointers are valid; (2) optionally, perform some mappings, for example assigning identifiers to pointers or to locally store additional information; (3) pack the arguments together with a function identifier; (4) send the execution request to the server; and (5) in synchronous functions, wait for the server's response.

3.2. Server side

An rCUDA server daemon, located at each node offering acceleration services, is in charge of receiving, interpreting, and executing remote CUDA call requests. For each client application, a new server process is created (using prefork for performance purposes) to execute all the requests from that individual remote execution into an independent GPU context. Spawning a different server process for each remote execution over a new GPU context facilitates GPU multiplexing, also ensuring the survival of the other GPU contexts in the event of a crash of one of the servers (e.g., caused by an improper CUDA call). A multithreaded server architecture solution would not provide this safety feature.

In an HPC cluster environment where jobs are scheduled and assigned to the different general-purpose cores of the system, rCUDA allows all the GPU coprocessors to be safely shared by the different jobs, provided there is sufficient GPU memory to concurrently run all the requested applications. The NVIDIA-proprietary device driver running in the GPU servers will manage the concurrent execution of the different active contexts using its own scheduler, in the same way as it does in a local GPU context. Our multiple servers sharing local GPUs could benefit from techniques designed to improve this usage mode, such as some of those employed in [36–38]. Furthermore, the recent Kepler GPU architecture [39] features enhanced multitask support with the Hyper-Q technology, which directly benefits our virtualization technology without the need of any further development or customization. These enhancements are left to be explored in future work.

On the other hand, depending on the particular requirements of the target applications, GPGPU servers can be either regular cluster nodes, with one or more GPUs, or specialized nodes. In the first case, GPGPU servers can also be employed as compute nodes, if some overhead is tolerable, since the GPGPU service may use some CPU and memory bandwidth resources (see Section 5.4). In the second case, a specialized system design with low-end processors, several GPUs, and high I/O and network capabilities may be desirable when leveraging exclusive GPGPU servers. From a logical point of view, this latter alternative is analogous to storage servers, conceived as "boxes" of shared resources directly connected to the network fabric.

3.3. Client–server communications

The application-level protocol used to communicate rCUDA clients and servers has been designed to be simple, so that the interactions involve little computation and can efficiently use the network resources. For details, see [27].

The initial releases of rCUDA exclusively employed TCP sockets for communication between the different actors, and sockets API calls were directly placed within the rCUDA code. In HPC networks, however, such as in InfiniBand interconnects, TCP usually attains a reduced portion of the fabric throughput. Therefore, to optimize rCUDA for different underlying communication technologies, we have redesigned the rCUDA architecture to enable modularized communications. Specifically, we have devised a generic internal communications API and created internal communication libraries, thus decoupling rCUDA communications from the rest of the tasks. In this way, we can develop specific communication libraries for different technologies, enabling a higher exploitation of each particular interconnect by developing specialized communication modules. The current rCUDA version provides specific support for the InfiniBand fabric, by means of the InfiniBand communications module (described in Section 4). For the rest of the fabrics, a generic highly tuned TCP/IP module is currently provided.

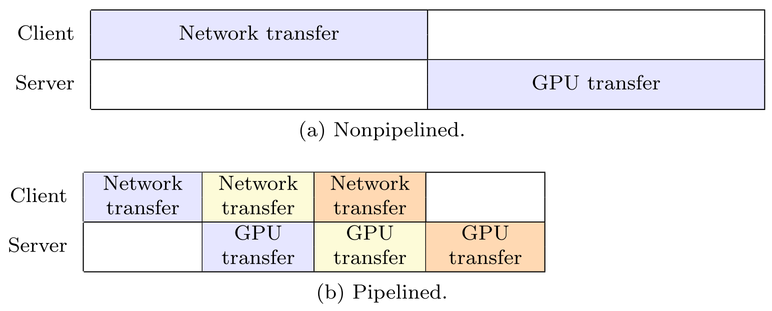


Fig. 3. Transfer to remote GPU in nonpipelined and pipelined modes.

3.4. Overlapped GPU and network transfers

The straightforward way of transferring data to a remote GPU consists of two main steps (see Fig. 3a): (1) on the client side, transfer the data to the server through the network, and (2) on the server, parse the headers to determine the buffer size, allocate a temporary buffer in main memory, receive the data, transfer them to GPU memory, and release the buffer.

We *pipelined* these steps to obtain a virtually constant data flow from the client's local main memory to the remote GPU memory. Fig. 3 shows the difference between the straightforward and pipelined modes when transferring data to a remote GPU. As shown there, the pipeline reduces the amount of time needed for the overall data transfer. Overlapping data transfers involves concurrent transfers at every stage, since sending a data chunk to the following stage must be performed concurrently with the reception of the next chunk from the former stage, which may stress memory bandwidth. The size of these chunks has to be carefully selected in order to balance the duration of the different pipeline stages and avoid unnecessary delays. Determining this size *a priori* is not an easy task, however, since the same pipeline stage may present different transfer rates in two similar systems when employing, for example, different revisions of the same hardware. Furthermore, those stages present variable transfer rates for different data sizes.

On the other hand, during the initialization stage, internal buffers are allocated and registered for both network transfers and CUDA use (pinned). This enables all user-requested GPU data transfers to be internally driven using CUDA asynchronous copies, regardless of their user-level type, which notably increases their throughput.

Notice that data transfers to remote GPUs can be more efficient if the intermediate storage on the server's main memory is avoided, thus providing a direct flow from the network into the GPU memory. This constraint was overcome in release 5.0 of CUDA with the introduction of GPUDirect RDMA[2] [40], being also supported by Mellanox with the ConnectX-3 InfiniBand network interfaces. rCUDA also supports this feature.

3.5. Implementing CUDA asynchronous memory copies

In the previous subsection we revisited how data transfers are implemented within rCUDA, by internally leveraging a pipelined approach based on the use of CUDA asynchronous memory copies. These efficient data transfers are involved in servicing both the synchronous and asynchronous memory copy operations that the CUDA API features. In the former case, the implementation based on asynchronous pipelined transfers is relatively simple, since the execution flow is stopped until all the data are transferred and the final synchronization is achieved.

When servicing a CUDA asynchronous memory copy, however, further work is required, since the CUDA asynchronous memory transfers, when driven by rCUDA, are not local to the computer but become distributed operations, noticeably increasing their implementation complexity. Providing support for CUDA asynchronous memory copy operations posed a major design challenge in our framework. Actually, most of the related virtualization technologies introduced in Section 2 do not support this feature.

These operations may be associated with a CUDA *stream*.[3] When an asynchronous transfer function is called, the program continues its execution once the memory transfer is programmed, that is, before the actual memory transfer is completed or even started. This is also the case with rCUDA, which provides the same API as does CUDA. Thus, our client middleware issues a memory transfer request and immediately returns the execution control to the caller, while the middleware still performs the required data transfers and synchronizations in the background. In our implementation, different CUDA streams are served following a round-robin policy, while operations in a stream are executed according to the first-come, first-served algorithm (see Fig. 4a).

[2] RDMA stands for Remote Direct Memory Access.

[3] A sequence of operations associated with a specific stream are executed in order, while different streams may be executed either out of order or concurrently.

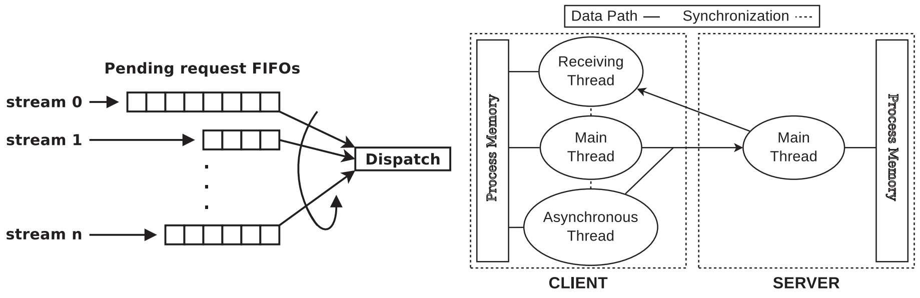


Fig. 4. rCUDA internal threads.

To effectively handle asynchronous operations and to enable the reception of both synchronous and asynchronous data via the same communication channel, the client middleware employs a dedicated POSIX thread, conveniently synchronized (by means of mutual exclusion regions and condition variables) with the main thread and the *receiving thread* (see Fig. 4b). These auxiliary threads are automatically created during the first CUDA call by each client thread and are finalized on library destruction routines. In the event of an asynchronous reception, data are concurrently stored into the memory region extracted from their associated destination information.

3.6. Other extended features

In order to fully cover the CUDA functionality, the initial rCUDA development was completely re-engineered to allow a thread-safe implementation, enabling multiple application threads to interact with the client library of rCUDA. Moreover, the fact that GPUs may be spread along the cluster nodes required moving a step beyond, enabling applications (either single- or multithreaded) to interact with GPUs possibly placed at different GPGPU servers. The main result from this effort is a robust, versatile, and production-ready rCUDA library.

Sharing GPU devices among cluster nodes poses an additional requirement to those of the original CUDA. In particular, it requires the *global job scheduler*—e.g., SLURM [41]—to be aware of the rCUDA operation mode, as current scheduling schemes for GPU-accelerated tasks [42,43] are not suitable for pools of resources decoupled from computing nodes. In order to integrate our new remote GPU virtualization framework with job schedulers, a different approach to allocate resources is needed. Since rCUDA allows a resource to be allocated independently from its physical location, global resource counters have to be leveraged instead. We propose two execution modes for remote GPU allocation, *exclusive* and *shared*, selected by the system administrator. The exclusive mode prioritizes performance, while the shared mode optimizes resource utilization at the expense of attaining lower performance. If GPUs are configured in exclusive mode, a counter for the available GPUs across the whole cluster must be taken into account. On the other hand, if GPUs are configured in shared mode, the available memory per GPU constraints GPU allocation. Notice that both modes may co-exist in a given deployment.

4. rCUDA InfiniBand communications module

Although the IB software stack supports a wide range of communication APIs, including standard TCP/IP sockets, the lowest-level API, which directly exposes all its features, is the InfiniBand Verbs (IBV). Therefore the development of a specific IB communications module for rCUDA on top of IBV is clearly convenient, since TCP over InfiniBand attains only a relatively small fraction of the transfer rate for this fabric.

In spite of the IBV-based initialization latency being over three times larger than its TCP-based counterpart for the target IB interconnect, in our tests this time difference is under 20 ms, which is easily overcome for sufficiently long application executions, because of the higher throughput attained by this communications module.

The remainder of this section discusses the design of our IBV-based communications module for rCUDA, which is the key to enabling high-throughput as well as low-latency communications with remote GPUs.

4.1. Communication mechanism

rCUDA uses both channel (send/receive) and memory (RDMA) IBV communication semantics. The rCUDA main operation mode consists of the rCUDA server waiting for a client request and returning the operation result. This behavior matches the channel semantic, since synchronized intervention from both endpoints is required. On the other hand, CUDA asynchronous memory transfers are implemented by making use of the memory semantic: once the operation request has been sent via the regular mechanism based on the channel semantic, an RDMA transfer can be issued to be eventually performed with no further intervention from the software, and both rCUDA middleware actors do not expect further actions from each other. Note that the issuer side of the RDMA request may still synchronize with the operation finalization in order to initiate further actions, such as executing a CUDA memory copy operation to transfer the received data to the GPU memory.

Two communication channels per rCUDA client thread are created. As we discuss in Section 4.2, this mechanism allows preposting both receive operations for function call information and memory transfer data to the appropriate memory buffers, thus improving performance. We make use of GPUDirect capabilities to share buffers for both IB and CUDA transfers.

4.2. Efficient pipeline implementation

An efficient implementation of network and GPU overlapped transfers in the rCUDA IBV based communications module requires determining whether the transfers are synchronous or asynchronous. The support for CUDA synchronous transfers, which may involve pageable memory regions, is efficiently implemented in a three-stage pipeline involving a pair of preallocated and preregistered (with CUDA and IBV) buffers per direction and endpoint; see Fig. 5a. On the other hand, asynchronous transfers enable a highly efficient way of addressing their implementation. Since this type of primitives must involve pinned memory regions, their corresponding client and server memory buffers can be employed to eliminate a pipeline stage on the client side, as depicted in Fig. 5b, and hence reduce both main memory stress and latency. Furthermore, as introduced in Section 4.1, to service asynchronous transfers, rCUDA leverages IB RDMA operations, which present more favorable latency and CPU usage, since they do not involve the CPU in the remote endpoint.

5. Experimental evaluation

This section covers the experimental evaluation of rCUDA employing the IB communications module. Our testbed is introduced first, followed by an evaluation of the NVIDIA CUDA Samples. Performance results are presented next, initially using a set of benchmarks and then the LAMMPS production application, considering also the case for a networked GPU box attached to the cluster. We then evaluate the interaction between GPGPU service and CPU computation in both client and server sides, and present a basic performance comparison with other publicly available CUDA virtualization solutions.

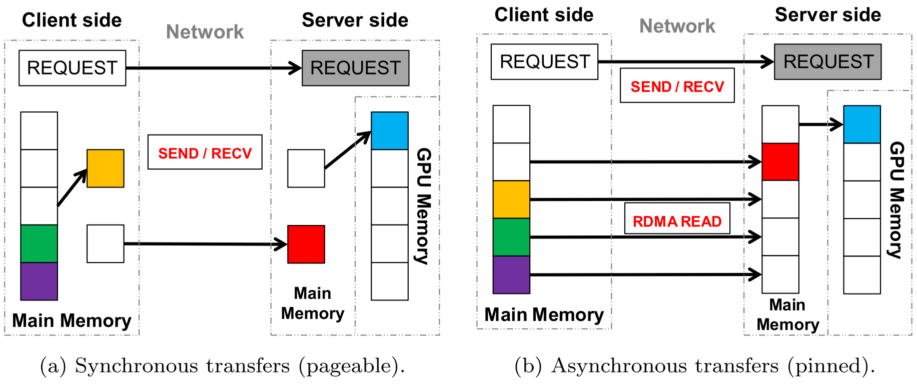
In order to reduce the effect of variability during the tests, the timings presented in the following correspond to the minimum obtained from five executions.

5.1. Testbed system

Our target platform is a cluster of nine 1027GR-TRF Supermicro servers, each equipped with two Intel Xeon ES-2620v2 six-core CPUs, 32 GB of DDR3 SDRAM memory at 1.60 GHz, and an NVIDIA Tesla K20 GPU. These are interconnected by an InfiniBand FDR fabric through Mellanox ConnectX-3 single-port adapters and a Mellanox SX6036 switch. The cluster nodes run CentOS release 6.4, with Mellanox OFED 2.1-1.0.0 (IB drivers and administrative tools), and CUDA 5.5 with NVIDIA driver 331.22. MVAPICH2 2.0b [44] is used for LAMMPS executions, while the latest release of rCUDA (version 4.1) is employed for remote GPGPU service.

5.2. Correctness evaluation

We used the CUDA Samples 5.5 (formerly SDK) to prove the correctness and completeness of rCUDA, since these benchmarks use a wide range of the CUDA API. From the 118 base samples contained in this package, 24 were not supported by rCUDA because of the use of the Driver API, graphics interoperability, zero-copy capabilities, or unsupported libraries. The 94 remaining samples were successfully executed and offered correct results. The total execution time was 245.35 s, whereas



(a) Synchronous transfers (pageable). (b) Asynchronous transfers (pinned).

Fig. 5. IBV pipeline support in IBV communications module.

their execution in a CUDA native environment (employing a local GPU) was 49% slower. This slower execution is due mainly to the initialization of the CUDA environment, which in our system requires about 1.3 s per execution, whereas rCUDA pre-initializes it at server start-up only, thus avoiding such a penalty. Arguably, the execution times obtained with these simple coding examples may not be considered significant for assessing the performance of our virtualizing solution on real use cases, since these are intended to demonstrate and test the CUDA functionality, comprising unoptimized codes in most cases, and tend to comprise short executions.

5.3. Performance evaluation

We selected two benchmarks to assess the performance of our solution. A synthetic microbenchmark based on CUDA memory transfers is evaluated first, followed by a single-precision matrix–matrix product employing the CUBLAS library shipped with the CUDA environment. We then include a performance evaluation that relies on LAMMPS, a complex production application, in several use cases.

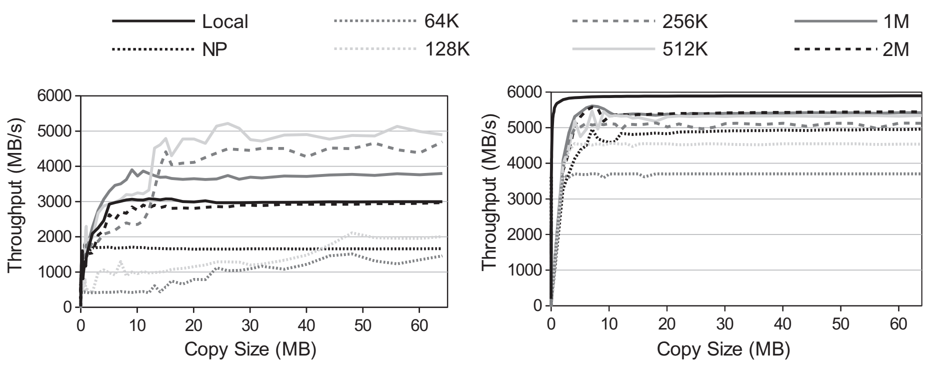
5.3.1. Remote GPU throughput

In this microbenchmark, the throughput between client's main memory and remote GPU's memory is measured with the `bandwidthTest` program from the CUDA Samples. This test performs 10 consecutive memory copy operations followed by a synchronization in case of asynchronous transfers. CUDA transfers are known to attain different peak transfer rates depending on the direction of the transfers (host to device or vice versa). For simplicity, we tackle only the host-to-device direction.

To present the results in this section in their context, we measured the basic throughput of our IB network, which attained 6054.83 MB/s, according to the `ibwritebw` tool from OFED. However, internal shared paths—such as main memory bandwidth—pose a limit on IB throughput in our system to 5903.20 MB/s when concurrent GPU and network transfers are carried out. We do not further pursue this issue, since the analysis of these hardware limitations is out of the scope for this paper (see [45]).

On the other hand, there is a relationship between the transfer size and the optimal pipeline block size. In general, small block sizes favor latency, since pipeline buffers are filled faster and data are moved earlier through the pipeline stages; on the other hand, they are inefficient for large data payloads, since large messages are in general needed to attain close-to-peak throughput for both the PCIe and the network. Hence, the optimal block size should be chosen to be as small as possible while still delivering the maximum PCIe and network throughput. Fig. 6 shows the throughput for synchronous memory copies from pageable buffers and asynchronous copies from pinned memory buffers in our FDR network. The maximum relative standard deviation (RSD) observed in the five repetitions is 1.58 for 1 MB of data payload in the first case and 1.73 for 9 KB in the second, although these relatively large RSDs tend to decrease when larger data payloads are involved, and the eventual external noise becomes negligible, reaching a maximum of 0.35 for the largest size. The plots also include the GPU throughput with regular CUDA over a local GPU (`Local`) for reference; `NP` stands for "nonpipelined" transfers to a remote GPU with rCUDA, and the rest of the keys refer to the pipeline block size used for each particular case.

Fig. 6a reveals largely different behaviors when employing distinct pipeline block sizes in synchronous operations. Differences are much smaller for asynchronous memory transfers (Fig. 6b) because the successive asynchronous iterations performed in the benchmark allow the GPU transfers of a transaction to overlap with the network transfers of the following operation, hence improving network utilization and reducing pipeline block-size impact. In addition, the latter are able to benefit from direct RDMA IB transfers avoiding a pipeline stage, as described in Section 4. This results in asynchronous transfers attaining 95% of efficiency with respect to the 5903.20 MB/s limit imposed by our available IB throughput. Additionally, these experiments reveal an improvement of over 200% in remote GPU throughput when employing pipelined GPU and network transfers with respect to the straightforward nonpipelined method.



(a) Pageable buffers ("cudaMemcpy"). (b) Pinned buffers ("cudaMemcpyAsync").

Fig. 6. Throughput with remote GPU for different pipeline block sizes.

For comparison, Fig. 7a shows the asynchronous benchmark results for the IB network employing TCP communications (IPoIB), as well as the IBV-based communications over the IB fabric, all with optimal pipeline block sizes (1 MB for IBV and 2 MB in the IPoIB case), and the local GPU for reference. The RSD observed is smaller than 0.01 for the larger data payloads, scaling up to 1.73 for the smallest data transfers. These results clearly show the performance benefits that the IBV communications module yields: a throughput at most only 5% lower than a local GPU (8% sustained), and up to over three times the maximum transfer rate of the IP over IB (IPoIB) functionality.

For our subsequent experiments we set a pipeline block size of 512 KB, as it offers the best combined performance for a wide range of data payloads in the four possible combinations of transfer directions and memory user-buffer types (pageable and pinned). The study of the actual benefits that leveraging adaptive pipeline block sizes may provide is left for future work.

5.3.2. Benchmark: matrix–matrix product

In our next benchmark we consider the single-precision matrix–matrix product, $C = A \times B$, using the implementation of this BLAS operation (sgemm) in CUBLAS 5.5, on a remote GPU employing rCUDA leveraging the IB interconnect. Because of the large amount of data, pinned memory buffers are not recommended for matrix storage, and regular pageable buffers are employed instead. The results for different matrix sizes are depicted in Fig. 7b, where we also report the performance corresponding to the execution on the local GPU, as well as CPU computations employing Intel MKL 11.1.0.080 making use of the 12 processor cores of the compute node. GPU times include GPU memory allocation and deallocation, as well as GPU transfers. As the attained computation performance employing the remote GPU (often expressed in GFLOPS) does not differ from that of native executions—only GPU communications are impacted—in our experiments we consider the total execution time of our benchmark instead. In addition, as this is a simple benchmark and the matrix–matrix product is expected to be performed as part of other computations in both CPU and GPU sides, in this experiment we have removed the initialization overhead from the native executions. Square matrices are used in all the experiments, and the maximum RSD is below 0.18.

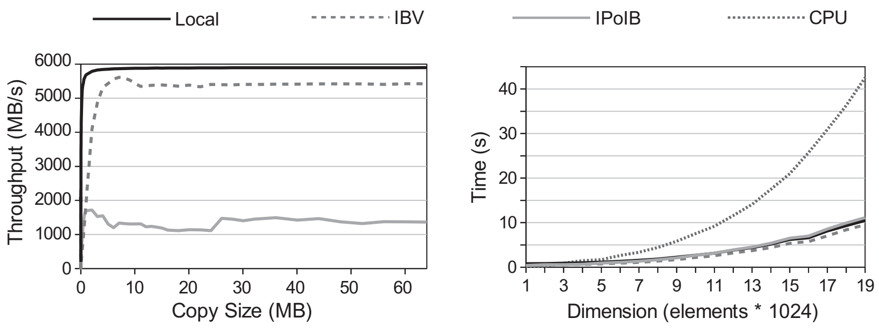
The experiments in Fig. 7b reveal that the rCUDA executions over IB are in fact from 10 to 85% faster than their local CUDA counterpart because of rCUDA's enhanced implementation of synchronous transfers involving pageable memory, as described in Section 3.4, whereas they are up to 4.5 times faster than CPU executions using a highly tuned library and the 12 cores of the computer. These results demonstrate the viability of remote GPU acceleration over HPC interconnects for suitable problems.

The results are, in some way, not surprising, since the matrix–matrix problem is a compute-bound operation, with $O(n^3)$ floating-point operations on $O(n^2)$ data, where n is the matrix dimension, and is highly appropriate for the GPU architecture, hence being a perfect candidate for remote GPU acceleration. The tests in next section show that remote GPU acceleration is also a general solution suitable for more challenging applications.

5.3.3. Production application: LAMMPS

The rCUDA functionality and performance have been tested with a number of benchmarks and production applications (such as OpenFOAM [46], HOOMD-Blue [47], WideLM [48], mCUDA-MEME [49], GPU-Blast [50], and CUDASW++ [51]). For this paper we selected LAMMPS [52] because (1) it features a complex code, which is useful for demonstrating the ability of rCUDA to handle large applications; and (2) this application permits us to extract representative results for remotely accelerated applications.

LAMMPS (Large-scale Atomic/ Molecular Massively Parallel Simulator) is an open source molecular dynamics simulator. The software is designed to run on distributed-memory systems employing MPI. Like most molecular dynamics simulations, LAMMPS executions are compute-bound. Two development efforts, the USER-CUDA and GPU packages, enable CUDA acceleration, each reporting to be more efficient than its counterpart depending on the particular features of the simulation. LAMMPS version dated February 1, 2014 with the USER-CUDA package [53,54] is used in our experiments. We use two different benchmarks provided in the standard LAMMPS distribution, the embedded atom method metallic solid benchmark (eam)



(a) Effective throughput with remote GPU. (b) Matrix multiply benchmark.

Fig. 7. Remote GPU throughput and matrix multiply benchmarks.

and the Lennard-Jones liquid benchmark (lj), both over a time span of 100 iterations. We scale the benchmarks to simulate $4 \cdot 10^6$ atoms. Loop time as reported by LAMMPS, which is the average time spent per iteration of the simulation, is reported in our results. For brevity, we omit the results with our TCP-based communications module.

5.3.3.1. GPU aggregation: providing many GPUs to an application. The USER-CUDA LAMMPS package performs most of the computations on the GPU. Thus, executing more than one MPI task per GPU does not usually lead to a performance improvement, and it may even exhibit some performance degradation caused by communication and synchronization overheads, as shown in Fig. 8a with the keys CUDA eam and CUDA lj. This figure reports the speedup with respect to CPU-only executions employing the 12 cores of a single computing node (an MPI task per core), obtaining Loop times of 113.03 s for the eam benchmark and 43.20 s in the case of lj.

rCUDA makes all cluster GPUs seamlessly available to all nodes. By leveraging rCUDA, we ran up to 8 MPI tasks in one compute node employing 8 remote GPUs, attaining speedups of up to 10.7× and 7.9× with respect to the CPU-only execution (see Fig. 8a). The highest RSD in these experiments is under 0.01. Note that in this single experiment we compare the performance of up to eight remote GPUs with that of the local resources, which include only one GPU. The remainder of the experiments include comparisons involving the same number of local and remote GPUs, enabling a fair discussion of the overhead incurred by our remote GPU acceleration solution.

5.3.3.2. GPU reduction: sharing applications. Since LAMMPS executions are tightly synchronized, data transfers from a single node to all remote GPUs concurrently compete for the single IB link connecting the compute node to the rest of the cluster where the GPUs are hosted, thus limiting scalability. Since a similar issue arises also when a few local GPUs share internal paths, applications caring about this potential limitation will seamlessly benefit from this feature of rCUDA. Desynchronized communication patterns among MPI tasks may further benefit from accessing multiple distributed GPUs from a single node, as explored in subsequent experiments.

This is first evaluated by running noncollaborative LAMMPS tasks simulating $1.16 \cdot 10^6$ atoms each, sharing a different number of GPUs. The rCUDA experiments consist of four tasks running in different nodes targeting a different number of remote GPGPU servers, whereas native CUDA executions were performed by 1 to 4 tasks within a single node. As shown in Fig. 8b—where averaged Loop times are normalized with respect to single-process native CUDA executions—the overhead introduced by the remote acceleration with respect to a native execution is up to 25% when the eam simulation employs a single process per remote GPU. On the other hand, some rCUDA-assisted executions were faster than native simulations (up to 5% in the case of lj with four processes sharing a GPU). The reason is that, in addition to the faster synchronous transfers rCUDA provides, CUDA and rCUDA implement different mechanisms to determine the finalization of asynchronous operations, based on distinct polling intervals, which can lead to considerably different performance. The maximum RSD in these experiments is 0.03.

In production environments, where largely different applications may concurrently employ the same GPU, the performance obtained by the distinct nonsynchronized processes is likely to be even higher than those observed in the previous case.

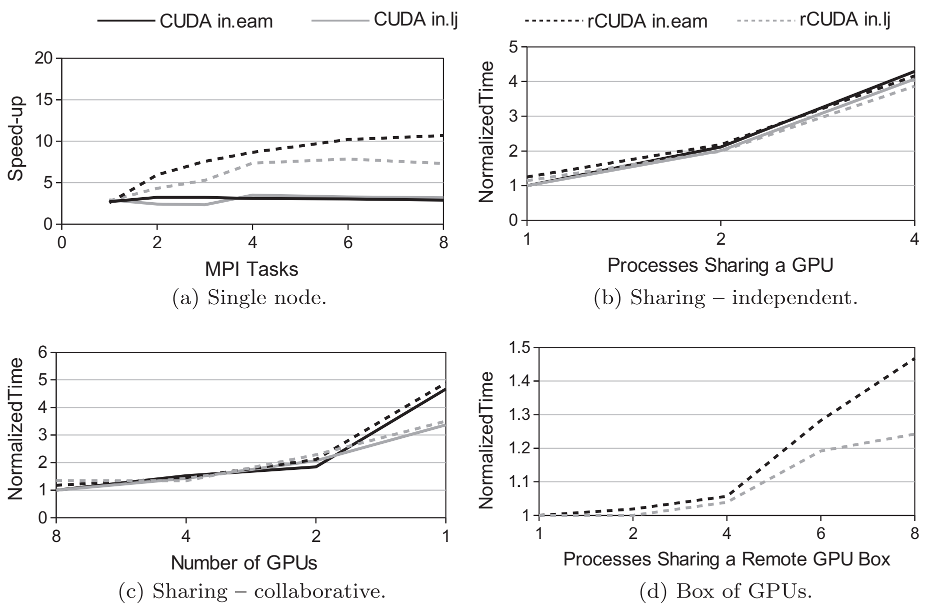


Fig. 8. GPU-accelerated LAMMPS performance evaluation with rCUDA.

Since the behavior shown in Fig. 8b can be considered a favorable case for rCUDA, in order to illustrate a worst case scenario involving collaborative tasks that tend to be tightly synchronized and hence perform communications and computations at the same time, we performed a second experiment running a LAMMPS simulation of $4 \cdot 10^6$ atoms deploying 8 MPI tasks sharing a different number of GPUs. In this evaluation, rCUDA-assisted executions are run from a single node targeting a different number of remote GPUs. For reference, the behavior is compared with different MPI tasks (running on different nodes when needed) sharing local GPUs with regular CUDA. Times in Fig. 8c are normalized with respect to the 8-node native executions. The highest RSD is 0.01. In this case, rCUDA executions pay an inherent overhead (up to 35% in the case of `lj` with 8 GPUs) introduced by the network path, while the scalability behavior mimics that of native executions. Again, some rCUDA executions revealed a higher performance than their native counterpart, being up to 6% faster for `lj` when employing four GPUs.

5.3.3.3. GPGPU server. To illustrate the scenario where GPUs are consolidated in GPGPU servers, as it could be the case of deploying the "GPU boxes," we used a Supermicro 7047GR-TPRF server, equipped with four K20 GPUs and a ConnectX-3 Dualport IB card. This specialized GPU server features enough PCIe lanes to fully serve the 4 GPUs plus the IB adapter, and hence there is no competition for PCIe resources. However, shared internal paths (as main memory bandwidth) and IB links become shared resources in this configuration and introduce some impact on the performance of the solution.

In this experiment we run up to 8 processes in separate nodes, using up to the 4 remote GPUs through the two different IB ports. Two rCUDA parent servers own two GPUs each are accessed through different IB ports. The remote processes make a balanced use of the different accelerator and network resources. For instance, when we execute two remote processes, these employ different IB ports and 2 separate GPUs (the other 2 remain idle), whereas in the case of 8 remotely-accelerated processes, every 4 share an IB port in the server, while every pair share a remote GPU. This distribution will be handled by the enhanced global job scheduler capabilities as discussed in Section 3.6, currently under development. In these experiments, $4 \cdot 10^5$ atoms per process are simulated.

Fig. 8d shows our results, where LAMMPS `Loop` times are normalized with respect to single remotely accelerated executions. The maximum RSD of these experiments is 0.06. As shown in the figure, we obtain a maximum slowdown of 47% when the GPGPU server is servicing the 8 remote executions and oversubscribing the available GPUs, while it falls below 6% when the GPUs are not oversubscribed. This poses a maximum overhead with respect to dedicated local accelerations of over 100%, although the maximum timings are still up to 46% and 58% below those of local CPU-only executions for the `eam` and `lj` benchmarks, respectively. In addition, there is still room for improvement: beyond GPU oversubscription, this overhead is largely introduced by the IB card, which, in spite of featuring two IB ports, is not capable of offering an aggregate throughput above that of the single-port version because of saturating the 8 PCIe lanes it features. Recent Mellanox Connect-IB cards leveraging 16 PCIe lanes are expected to overcome this limitation and provide around double throughput. Similarly, a larger amount of separate network adapters with dedicated PCIe lanes would benefit remote GPU service by providing higher network throughput.

Our results reveal that remote acceleration in shared GPGPU servers is highly beneficial compared to CPU-only executions. Although as expected it introduces some overhead with respect to dedicated local acceleration mainly due to the shared resources, a higher network throughput brought by recently-emerged technologies will further improve the performance of this proposal. In conclusion, *GPU boxes* combined with GPU service have the potential to bring enhanced flexibility to cluster configurations, while providing the energy and resource savings of the *shared GPU pool* proposal.

5.4. Remote GPGPU service impact on concurrent CPU-only executions

We next analyze the impact on performance of GPGPU service in compute nodes by running a highly CPU-demanding benchmark in the rCUDA server and measuring its performance variation during GPGPU service. The benchmark selected to be run in the rCUDA server is a BLAS single-precision matrix–matrix product, $C = A \times B$ with operands of 4096 rows/columns. The highly tuned MKL BLAS implementation and the 12 cores of our compute nodes were used. For the applications demanding remote GPU services we evaluate `sgemm` CUBLAS executions with square matrices of dimension 13,848, as well as LAMMPS `lj` and `eam` executions sized as in the previous experiments.

Fig. 9 shows the impact on CPU computations of servicing the remote CUBLAS `sgemm`, as well as that of LAMMPS `lj` and `eam` executions. Times are normalized with respect to stand-alone average. In the case of `sgemm` service (Fig. 9a), the CPU execution on the server is increased by up to 106% (52% on average), while over 50% of the executions in the server remained within 25% overhead. LAMMPS GPGPU service (see Fig. 9b) presented similar results once the proper simulation started, whereas server CPU executions were not noticeably impacted while the remote application was preparing the simulation, and therefore the rCUDA daemon was not intensively servicing requests. Hence, the first 56 CPU executions at the rCUDA server show mostly no impact, whereas some of the last 19 executions are clearly affected.

These results reveal a potentially high impact of the GPGPU service on concurrent CPU executions on the server, due to main memory bandwidth stress, along with CPU requirements of the rCUDA server, since some CUDA calls show a high CPU usage. Therefore, for those cases where this performance impact is unbearable, the "GPU boxes" approach may become a nice alternative to avoid CPU computing on GPGPU servers.

We also measured the opposite effect: the impact on GPGPU service of a server performing CPU executions concurrently. In this case, we execute CPU-only BLAS `sgemm` from MKL and LAMMPS `in` and `eam` benchmarks, and assess the impact of

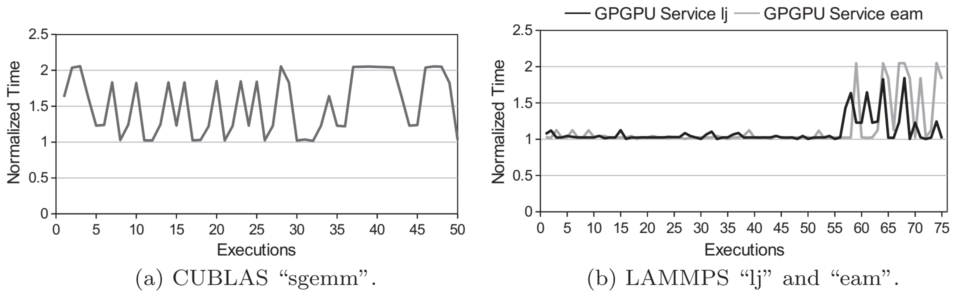


Fig. 9. Impact of GPGPU service on CPU computing.

these executions on remotely accelerated matrix–matrix products on the client side with respect to a dedicated GPGPU server execution. The executions were sized as in the previous experiment. As shown in Fig. 10, where LAMMPS times refer to `Loop` time, the impact of external CPU workloads on GPGPU service is small, remaining under 7% overhead.

5.5. Comparison with other CUDA virtualization solutions

To place the performance of our virtualization solution into the right context, in addition to the above-presented performance comparison with native CUDA and GPU-less environments, in this section we compare the performance of rCUDA with that of other CUDA virtualization frameworks.

From the remote virtualization solutions listed in Section 2.2, apart from rCUDA, currently the only publicly available frameworks are gVirtuS (01-beta3) and DS-CUDA (version 1.2.7). Since the latest CUDA revisions supported by them differ largely, offering different capabilities and forcing the usage of different CUDA library versions in the server, we limit ourselves to report latency and throughput for memory transfers, which are the leading factors affecting the performance obtained by remote acceleration.

Our results are summarized in Table 1. We show the latency for short and transfer rate for long memory copies (64 B and 30 MB, respectively) involving remote GPUs. For reference, in the table we include numbers for the native CUDA environment using a local GPU. In this experiment we target pinned memory transfers, since these are representative of the way applications perform transfers involving GPUs for performance purposes. However, we fall back to pageable memory regions in the case of DS-CUDA because of its lack of support for their more efficient counterpart.

As shown in the table, rCUDA is the remote virtualization solution featuring the lowest latency and highest transfer rates, offering a performance close to that of local GPU operations in most cases. We are investigating the reasons behind the different latency gaps with native CUDA in both transfer directions. These results demonstrate that the techniques described in

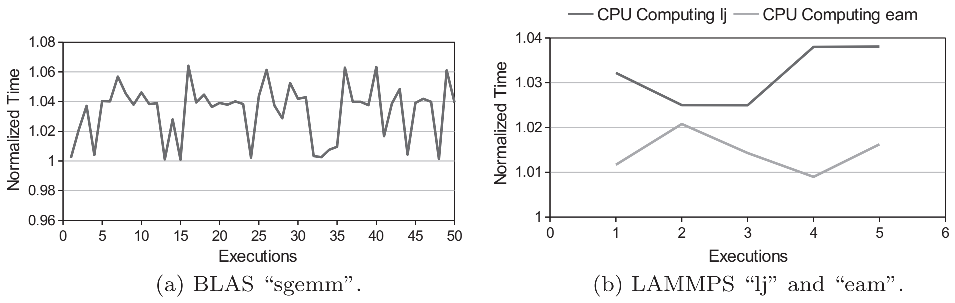


Fig. 10. Impact of CPU computing on remote GPGPU service.

Table 1
Comparison of different virtualization solutions and native CUDA.

Framework	Latency (μs)		Throughput (GB/s)	
	To GPU	From GPU	To GPU	From GPU
CUDA	4.3	5.1	5.8	6.2
rCUDA	23.1	6.0	5.3	5.6
GVirtuS	200.3	182.8	0.3	0.3
DS-CUDA	45.9	26.5	1.7	1.0

this paper enable rCUDA to attain a high degree of efficiency compared with other solutions as well as with native environments, effectively enabling the feasibility of remote GPU computing for a wide range of applications. Readers may refer to [55] for a performance comparison of different virtualization technologies on a virtual machine environment, including an early version of rCUDA.

6. Conclusions and future work

In this paper we have detailed rCUDA, an advanced framework offering remote CUDA acceleration that enables a reduction on the number of accelerators in a cluster, providing enhanced flexibility.

We have described the key features, main architectural design, and implementation details of rCUDA, including the low-level IB communication module. Our evaluation results, including a set of benchmarks and a production application, demonstrate the viability of our shared-GPU pool proposal, revealing an excellent performance for remote acceleration of the matrix–matrix product. The GPU-accelerated version of a complex application, LAMMPS, attained over $10\times$ speedup on a single node employing a remote GPU pool.

In summary, our results demonstrate that remote GPU acceleration leads to considerable savings at the expense of slightly increased execution time with respect to local GPU execution, maintaining large performance gains compared with those of traditional CPU-only execution.

Future work directions include analyzing the performance benefits that the recent Hyper-Q capabilities bring to our proposal with enhanced multitask support, studying potential benefits of using enhanced CUDA schedulers for remote GPU acceleration, reviewing specific hardware configurations for dedicated GPGPU servers, and assessing the actual benefits that adaptive pipeline block sizes may bring to production applications.

Acknowledgments

This work was supported by the Spanish Ministerio de Economía y Competitividad (MINECO) and by FEDER funds under Grant TIN2012-38341-C04-01. It was also supported by MINECO, FEDER funds, under Grant TIN2011-23283, and by the Fundación Caixa-Castelló Bancaixa, Grant P11B2013-21. This work was also supported in part by the U.S. Department of Energy, Office of Science, under contract DE-AC02-06CH11357. Authors are grateful for the generous support provided by Mellanox Technologies to the rCUDA Project. The authors would also like to thank Adrián Castelló, member of The rCUDA Development Team, for his hard work on rCUDA.

References

[1] NVIDIA Corporation, CUDA API Reference Manual Version 5.5 2013.
[2] A. Munshi (Ed.), The OpenCL Specification Version 1.2, Khronos OpenCL Working Group, 2011.
[3] H. Meuer, E. Strohmaier, J. Dongarra, H. Simon, TOP500 supercomputing sites. <http://www.top500.org/lists/2013/11>, 2013.
[4] S. Ashby, P. Beckman, J. Chen, P. Colella, B. Collins, D. Crawford, J. Dongarra, D. Kothe, R. Lusk, P. Messina, et al., The opportunities and challenges of Exascale computing, 2010.
[5] J. Dongarra, P. Beckman, T. Moore, P. Aerts, G. Aloisio, J.-C. Andre, D. Barkai, J.-Y. Berthou, T. Boku, B. Braunschweig, et al, The international exascale software project roadmap, Int. J. High Perform. Comput. Appl. 25 (1) (2011) 3–60.
[6] M. Duranton, D. Black-Schaffer, K. De Bosschere, J. Maebe, The HiPEAC vision for advanced computing in horizon 2020, 2013.
[7] P. Kogge, K. Bergman, S. Borkar, D. Campbell, W. Carson, W. Dally, M. Denneau, P. Franzon, W. Harrod, K. Hill, et al., ExaScale computing study: Technology challenges in achieving ExaScale systems, 2008.
[8] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, A. Warfield, Xen and the art of virtualization, in: Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles (SOSP), New York, NY, USA, 2003, pp. 164–177.
[9] A. Kivity, Y. Kamay, D. Laor, U. Lublin, A. Liguori, kvm: the Linux virtual machine monitor, in: Proceedings of the Linux Symposium, vol. one, 2007, pp. 225–230.
[10] H. Wu, G. Diamos, S. Cadambi, S. Yalamanchili, Kernel weaver: automatically fusing database primitives for efficient GPU computation, in: Proceedings of the 45th International Symposium on Microarchitecture (MICRO), IEEE Computer Society, 2012, pp. 107–118.
[11] J. Young, H. Wu, S. Yalamanchili, Satisfying data-intensive queries using GPU clusters, in: High Performance Computing, Networking, Storage and Analysis (SCC), IEEE, 2012, p. 1314.
[12] NextIO Inc, vCORE. <http://www.nextio.com/products/vcore>, 2012.
[13] V. Krishnan, Towards an integrated IO and clustering solution using PCI Express, in IEEE International Conference on Cluster Computing 2007, 2007, pp. 259–266.
[14] D. Blythe, The Direct3D 10 system, ACM Trans. Graph. 25 (3) (2006) 724–734.
[15] D. Shreiner, OpenGL, OpenGL(R) Programming Guide: The Official Guide to Learning OpenGL(R), Versions 3.0 and 3.1, 7th ed., Addison-Wesley Professional, 2009.
[16] L. Shi, H. Chen, J. Sun, K. Li, vCUDA: GPU-accelerated high-performance computing in virtual machines, IEEE Trans. Comput. 61 (6) (2012) 804–816.
[17] V. Gupta, A. Gavrilovska, K. Schwan, H. Kharche, N. Tolia, V. Talwar, P. Ranganathan, GViM: GPU-accelerated virtual machines, in: 3rd Workshop on System-level Virtualization for High Performance Computing, ACM, NY, USA, 2009, pp. 17–24.
[18] G. Giunta, R. Montella, G. Agrillo, G. Coviello, A GPGPU transparent virtualization component for high performance computing clouds, in: Euro-Par 2010 – Parallel Processing, LNCS, vol. 6271, Springer, Berlin/Heidelberg, 2010, pp. 379–391.
[19] M. Oikawa, A. Kawai, K. Nomura, K. Yasuoka, K. Yoshikawa, T. Narumi, DS-CUDA: a middleware to use many GPUs in the cloud environment, in: High Performance Computing, Networking, Storage and Analysis (SCC), 2012, pp. 1207–1214.
[20] V. Ravi, M. Becchi, G. Agrawal, S. Chakradhar, Supporting GPU sharing in cloud environments with a transparent runtime consolidation framework, in: Proceedings of the 20th International Symposium on High-Performance Parallel and Distributed Computing, ACM, 2011, pp. 217–228.
[21] Zillians Inc., VGPU. <http://www.zillians.com/vgpu>, 2012.
[22] T.-Y. Liang, Y.-W. Chang, GridCuda: A grid-enabled CUDA programming toolkit, in: Proceedings of the 25th IEEE International Conference on Advanced Information Networking and Applications Workshops (WAINA), 2011, pp. 141–146.

[23] A. Barak, T. Ben-Nun, E. Levy, A. Shiloh, A package for OpenCL based heterogeneous computing on clusters with many GPU devices, in: Workshop on Parallel Programming and Applications on Accelerator Clusters (PPAAC), 2010, pp. 1–7.
[24] S. Xiao, P. Balaji, Q. Zhu, R. Thakur, S. Coghlan, H. Lin, G. Wen, J. Hong, W. Feng, VOCL: an optimized environment for transparent virtualization of graphics processing units, in: Proceedings of the 1st Innovative Parallel Computing (InPar), 2012, pp. 1–12.
[25] J. Kim, S. Seo, J. Lee, J. Nah, G. Jo, J. Lee, SnuCL: an OpenCL framework for heterogeneous CPU/GPU clusters, in: Proceedings of the 26th ACM International Conference on Supercomputing, ACM, 2012, pp. 341–352.
[26] P. Kegel, M. Steuwer, S. Gorlatch, dOpenCL: towards a uniform programming approach for distributed heterogeneous multi-/many-core systems, in: 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), IEEE, 2012, pp. 174–186.
[27] J. Duato, F.D. Igual, R. Mayo, A.J. Peña, E.S. Quintana-Ortí, F. Silla, An efficient implementation of GPU virtualization in high performance clusters, Euro-Par 2009, Parallel Processing – Workshops 6043, 2010, pp. 385–394.
[28] J. Duato, A.J. Peña, F. Silla, R. Mayo, E.S. Quintana-Ortí, rCUDA: reducing the number of GPU-based accelerators in high performance clusters, in: Proceedings of the 2010 International Conference on High Performance Computing and Simulation (HPCS), 2010, pp. 224–231.
[29] J. Duato, A.J. Peña, F. Silla, R. Mayo, E.S. Quintana-Ortí, Modeling the CUDA remoting virtualization behaviour in high performance networks, in: First Workshop on Language, Compiler, and Architecture Support for GPGPU (LCA-GPGPU-I), 2010.
[30] J. Duato, A.J. Peña, F. Silla, R. Mayo, E.S. Quintana-Ortí, Performance of CUDA virtualized remote GPUs in high performance clusters, in: Proceedings of the 2011 International Conference on Parallel Processing (ICPP), 2011, pp. 365–374.
[31] J. Duato, A.J. Peña, F. Silla, J.C. Fernández, R. Mayo, E.S. Quintana-Ortí, Enabling CUDA acceleration within virtual machines using rCUDA, in: Proceedings of the 2011 International Conference on High Performance Computing (HiPC), 2011, pp. 1–10.
[32] C. Reaño, A.J. Peña, F. Silla, J. Duato, R. Mayo, E.S. Quintana-Ortí, CU2rCU: Towards the complete rCUDA remote GPU virtualization and sharing solution, in: 19th International Conference on High Performance Computing (HiPC), IEEE, 2012, pp. 1–10.
[33] C. Reaño, R. Mayo, E.S. Quintana-Ortí, F. Silla, J. Duato, A.J. Peña, Influence of InfiniBand FDR on the performance of remote GPU virtualization, in: Proceedings of the IEEE Cluster 2013 Conference, 2013, pp. 1–8.
[34] A. Castelló, J. Duato, R. Mayo, A.J. Peña, E.S. Quintana-Ortí, V. Roca, F. Silla, On the use of remote GPUs and low-power processors for the acceleration of scientific applications, in: The Fourth International Conference on Smart Grids, Green Communications and IT Energy-aware Technologies (ENERGY), 2014, pp. 57–62.
[35] NVIDIA Corporation, NVIDIA VGX. <http://www.nvidia.com/object/vdi-desktop-virtualization.html>, 2012.
[36] T. Li, V.K. Narayana, E. El-Araby, T.A. El-Ghazawi, GPU resource sharing and virtualization on high performance computing systems, in: 40th International Conference on Parallel Processing (ICPP), 2011, pp. 733–742.
[37] T. Li, V.K. Narayana, T.A. El-Ghazawi, Accelerated high-performance computing through efficient multi-process GPU resource sharing, in: ACM International Conference on Computing Frontiers, 2012, pp. 269–272.
[38] T. Li, V.K. Narayana, T. El-Ghazawi, Exploring graphics processing unit (GPU) resource sharing efficiency for high performance computing, Computers 2 (4) (2013) 176–214.
[39] NVIDIA Corporation, NVIDIA's next generation CUDATM compute architecture: KeplerTM GK110 V1.0, White paper, NVIDIA Corporation, 2012. <http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf>.
[40] NVIDIA Corporation, NVIDIA GPUDirectTM. <http://developer.nvidia.com/gpudirect>, 2012.
[41] A.B. Yoo, M.A. Jette, M. Grondona, SLURM: Simple linux utility for resource management, in: Job Scheduling Strategies for Parallel Processing, Springer, 2003, pp. 44–60.
[42] Lawrence Livermore National Laboratory, SLURM Generic Resource (GRES) Scheduling. <https://computing.llnl.gov/linux/slurm/gres.html>, 2012.
[43] PBS WorksTM, Scheduling jobs onto NVIDIA Tesla GPU computing processors using PBS Professional, technical paper, PBS WorksTM, Oct. 2010.
[44] H. Wang, S. Potluri, M. Luo, A. Singh, S. Sur, D. Panda, MVAPICH2-GPU: optimized GPU to GPU communication for InfiniBand clusters, Comput. Sci. – Res. Dev. 26 (2011) 257–266.
[45] A.J. Peña, S.R. Alam, Evaluation of inter- and intra-node data transfer efficiencies between GPU devices and their impact on scalable applications, in: The 13th International Symposium on Cluster, Cloud and Grid Computing (CCGrid), 2013, pp. 144–151.
[46] H. Jasak, A. Jemcov, Z. Tukovic, OpenFOAM: A C++ library for complex physics simulations, in: International Workshop on Coupled Methods in Numerical Dynamics, 2007, pp. 1–20.
[47] J. Anderson, A. Keys, C. Phillips, T. Dac Nguyen, S. Glotzer, HOOMD-blue, general-purpose many-body dynamics on the GPU, Bulletin of the American Physical Society 55.
[48] M. Seligman, WideLM. <http://cran.r-project.org/package=WideLM>, 2013.
[49] Y. Liu, B. Schmidt, D. Maskell, An ultrafast scalable many-core motif discovery algorithm for multiple GPUs, in, IEEE International Symposium on Parallel and Distributed Processing Workshops and PhD Forum (IPDPSW) 2011 (2011) 428–434.
[50] P.D. Vouzis, N.V. Sahinidis, GPU-BLAST: using graphics processors to accelerate protein sequence alignment, Bioinformatics 27 (2) (2011) 182–188.
[51] Y. Liu, A. Wirawan, B. Schmidt, CUDASW++ 3.0: accelerating Smith–Waterman protein database search by coupling CPU and GPU SIMD instructions, BMC Bioinformatics 14 (1) (2013) 1–10.
[52] S. Plimpton, Fast parallel algorithms for short-range molecular dynamics, J. Comput. Phys. 117 (1995) 1–19.
[53] C.R. Trott, LAMMPSCUDA – a new GPU accelerated molecular dynamics simulations package and its application to ion-conducting glasses (Ph.D. thesis), Technische Universitt Ilmenau, 2011.
[54] C.R. Trott, L. Winterfeld, P.S. Crozier, General-purpose molecular dynamics simulations on GPU-based clusters, arXiv Mathematics e-prints, 2011. arXiv:1009.4330v2 [cond-mat.mtrl-sci].
[55] M.S. Vinaya, N. Vydyanathan, M. Gajjar, An evaluation of CUDA-enabled virtualization solutions, in: Parallel Distributed and Grid Computing (PDGC), 2012, pp. 621–626.