Enabling CUDA Acceleration within Virtual Machines using rCUDA

José Duato, Antonio J. Peña, Federico Silla
D. Informática de Sistemas y Computadores
Universitat Politècnica de València
Camino de Vera s/n
46022 Valencia, Spain
{*jduato, fsilla*}*@disca.upv.es*
apenya@gap.upv.es

Juan C. Fernández, Rafael Mayo,
Enrique S. Quintana-Ortí
D. Ingeniería y Ciencia de los Computadores
Universitat Jaume I
Av. Vicente Sos Baynat s/n
12071 Castellón, Spain
{*jfernand, mayo, quintana*}*@icc.uji.es*

Abstract

The hardware and software advances of Graphics Processing Units (GPUs) have favored the development of GPGPU (General-Purpose Computation on GPUs) and its adoption in many scientific, engineering, and industrial areas. Thus, GPUs are increasingly being introduced in high-performance computing systems as well as in datacenters. On the other hand, virtualization technologies are also receiving rising interest in these domains, because of their many benefits on acquisition and maintenance savings. There are currently several works on GPU virtualization. However, there is no standard solution allowing access to GPGPU capabilities from virtual machine environments like, e.g., VMware, Xen, VirtualBox, or KVM. Such lack of a standard solution is delaying the integration of GPGPU into these domains.

In this paper, we propose a first step towards a general and open source approach for using GPGPU features within VMs. In particular, we describe the use of rCUDA, a GPGPU (General-Purpose Computation on GPUs) virtualization framework, to permit the execution of GPU-accelerated applications within virtual machines (VMs), thus enabling GPGPU capabilities on any virtualized environment. Our experiments with rCUDA in the context of KVM and VirtualBox on a system equipped with two NVIDIA GeForce 9800 GX2 cards illustrate the overhead introduced by the rCUDA middleware and prove the feasibility and scalability of this general virtualizing solution. Experimental results show that the overhead is proportional to the dataset size, while the scalability is similar to that of the native environment.

1. Introduction

Many-core specialized processors and accelerators and, in particular, graphics processors (GPUs), are experiencing increased adoption as an appealing way of reducing the time-to-solution in areas as diverse as finance [1], chemical physics [2], computational fluid dynamics [3], computational algebra [4], image analysis [5], and many others. These hardware accelerators offer a large amount of processing elements with reduced cache memories and high processor-to-memory bandwidth, so that applications featuring a high rate of computations per data item can attain high performance. In addition, as GPU technology targets the gaming market, these devices present a relatively high performance/cost ratio, resulting in an interesting option for HPC (high performance computing).

On the other hand, virtualization technologies are currently widely deployed, as their use yields important benefits such as resource sharing, process isolation, and reduced management costs. Thus, it is straightforward that the usage of virtual machines (VMs) in HPC is an active area of research [6]. VMs provide an improved approach to increase resource utilization in HPC clusters, as several different customers may share a single computing node with the illusion that they own their entire machine in an exclusive way. Security is ensured by the virtualization layer, which additionally may provide other services, such as migrating a VM from one node to another if required, for example.

Processes running in a VM may also require the services of a GPU in order to accelerate part of their computations. However, the real GPU does not directly belong to the VM, but it should be attached, in some way, to it. To do so, the GPGPU capabilities should

be exposed to the VMs in such a way that they can make use of the real GPU. However, although there is currently some work on the virtualization of the graphics application programming interfaces (APIs) of GPUs (e.g., [7]), those efforts are not directly useful to expose GPGPU features to virtualized environments. The main cause is that both uses of GPUs are completely different and, therefore, advances in one of them do not translate in progress in the other. The reason for this is that current GPUs lack a standard low-level interface —unlike other devices such as storage and network controllers— and, therefore, their use for graphics purposes is approached by employing high-level standard interfaces such as Direct3D [7] or OpenGL [8], while using them for GPGPU requires APIs like OpenCL [9] or NVIDIA CUDA (Compute Unified Device Architecture) [10]. GPGPU-oriented APIs significantly differ from their graphics-processing oriented counterparts (OpenGL and Direct3D), as the former have to deal with general-purpose computing issues, while the latter are mainly concerned with graphics-related features such as object interposition, flickering, or graphics redirection. On the other hand, the few efforts done up to now to expose CUDA capabilities to VMs [11], [12], [13] (1) are incomplete prototypes, (2) make use of obsolete versions of the GPGPU API, (3) are not general solutions, as they target a particular virtual machine monitor (VMM) environment, or (4) employ inefficient communication protocols between the front-end and back-end components of their middleware.

In this paper, we move a step forward in the virtualization of GPUs for their use as GPGPU accelerators by VMs. We propose using an open source, VMM-independent, and communication-efficient way of exposing GPGPU capabilities to VMs featuring a recent GPGPU API version. Our work addresses the virtualization of the CUDA Runtime API, a widely used API to perform general computations over the latest NVIDIA GPUs. By using our proposal, GPGPU features of CUDA compatible GPUs are available to processes running in any VM. In particular, we employ a framework initially intended to enable remote use of the CUDA Runtime API. This framework named *rCUDA* [14], was initially designed to use TCP sockets to communicate a GPU-accelerated process running in a computer not having a GPU with a remote host providing GPGPU services, thus providing the accelerated process with the illusion of directly using a GPU. Therefore, this framework is suitable for distributed-memory environments. Note however that although the primary goal of rCUDA was providing a way to reduce the number of GPU-based accelerators in a cluster, which leads to savings in acquisition, energy, maintenance, and cooling costs, in this paper we extend the applicability of the rCUDA framework to also cover a wide spectrum of GPGPU in virtualized environments, exposing CUDA capabilities to VMs running in a CUDA-enabled computer. More specifically, we explore the benefits of using rCUDA in VMs, ranging from a single VM instance to multiple VMs concurrently running in the same server, equipped with a small number of accelerators. To do this, we analyze the execution of a set of CUDA SDK examples on a platform composed of 8 general-purpose cores and two NVIDIA cards providing a total of 4 GPUs. The results obtained with the Kernel-based Virtual Machine (KVM) and Oracle's VirtualBox Open Source Edition (VB-OSE) VMMs using rCUDA are additionally compared with those of the native environment. Although our solution is also valid for the VMware Server solution, we cannot disclose the results due to licensing policies. To the best of our knowledge, this is the first paper analyzing rCUDA usage for VMs, as previous papers focused on cluster environments.

The rest of the paper is organized as follows: in Section 2 we introduce some general background related with our contribution. In Section 3, we describe the architecture of rCUDA, while Section 4 provides information about its usage in VM environments. Section 5 presents the results of the experimental evaluation. Finally, Section 6 summarizes the conclusions of this work.

2. Background on CUDA Virtualization

In addition to rCUDA, which will be described in the following section, there are other approaches that pursue the virtualization of the CUDA Runtime API for VMs: vCUDA [13], GViM [12], and gVirtuS [11]. All solutions feature a distributed middleware comprised of two parts: the front-end and the back-end. The frond-end middleware is installed on the VM, while the back-end counterpart, with direct access to the acceleration hardware, is run by the host operating system (OS) — the one executing the VMM.

vCUDA [13] implements an unspecified subset of the CUDA Runtime version 1.1. It employs XML-RPC for the application level communications, which makes the solution portable across different VMMs but, as the experiments in [13] show, the time spent in the encoding/decoding steps of the communication protocol causes a considerable negative impact on the overall performance of the solution.

On the other hand, GViM [12] uses Xen-specific mechanisms for the communication between both

middleware actors, including shared memory buffers, which enhance the communications between user and administrative domains, at the expense of losing VMM independence. In addition, this solution, based also in CUDA 1.1, does not seem to implement the whole Runtime API.

Finally, gVirtuS [11] (version 01-beta1) is a tool with a purpose similar to rCUDA. This version seems to only cover a subset of the Runtime API v2.3 (e.g.: it lacks 20 out of the 37 functions of the memory management module of this API version).

In this paper, we propose using rCUDA, a production-ready framework to run CUDA applications from VMs, based in a recent CUDA API version (currently 3.1). This middleware makes use of a customized communications protocol and is VMM-independent, thus addressing the main drawbacks of previous works.

3. The rCUDA Framework

rCUDA is intended to provide access to GPUs installed in remote nodes. Hence, this framework offers HPC clusters a way of reducing the total number of GPUs in the system or, alternatively, to significantly accelerate the computations of a traditional cluster by adding a reduced number of accelerators. In other words, in the former case, by slightly increasing the execution time of the applications that make use of GPUs to accelerate parts of their code, considerable savings can be achieved in energy consumption, maintenance, space, and cooling. On the other hand, when adding a few accelerators to a cluster, rCUDA brings the possibility of significantly reducing the execution time of suitable applications with a small impact on the system energy consumption.

As in the case of the software presented in the previous section, the rCUDA framework is split into two major software modules, as depicted in Fig. 1:

- **The client middleware** consists of a collection of wrappers which replace the NVIDIA CUDA Runtime (provided by NVIDIA as a shared library) in the client computer (not having a GPU). These wrappers are in charge of forwarding the API calls from the applications requesting acceleration services to the server middleware, and retrieving the results back, providing applications with the illusion of direct access to a physical GPU.
- **The server middleware** runs as a service on the computer owning the GPU. It receives, interprets, and executes the API calls from the clients, employing a different process to serve each remote execution over an independent GPU context, thus attaining GPU multiplexing.

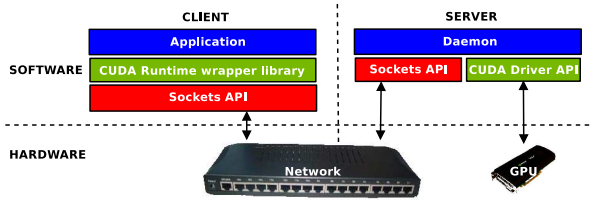

Figure 1. Outline of the rCUDA architecture.

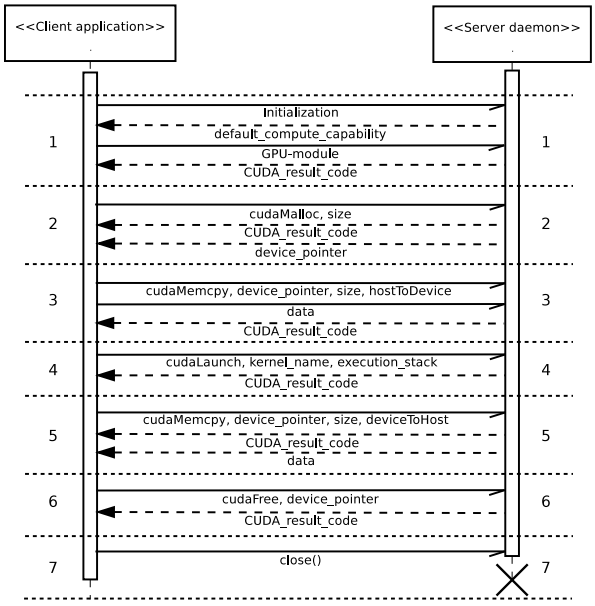

Figure 2. rCUDA communications protocol example — (1) initialization, (2) memory allocation on the GPU, (3) memory transfer of the input data from CPU to GPU, (4) kernel execution, (5) memory transfer of the results from GPU to CPU, (6) GPU memory release, and (7) socket closing and server process finalization.

Communication between rCUDA clients and servers is performed using TCP sockets through the network available in the cluster. It also implements a customized application-level protocol to efficiently communicate client and server middleware. Fig. 2 shows an example of the protocol implemented by rCUDA, based on a generic kernel call which uses one data set as input and produces a dataset output, such as the 1D FFT (1-dimensional Fast Fourier Transform).

rCUDA 2.0 —the most recent version at the moment of the writing of this paper— targets the Linux OS. It implements the CUDA Runtime API version 3.1, excluding OpenGL and Direct3D interoperability, as graphics-oriented capabilities are not of interest in this

environment. One drawback of rCUDA is that it lacks support for the *C for CUDA* extensions, as the CUDA Runtime library comprises some hidden and undocumented support functions, as reported by the vCUDA developers in [13][1]. As the rCUDA architecture features separate memory maps for the client and server middleware, the calls to these hidden functions cannot be blindly forwarded to the server (e.g., consider that pointers to the local CPU memory map might be involved in those functions; in this case, the memory pointed at the server would not contain the right data). We are currently working on solving this limitation.

Although there are other GPGPU APIs such as the CUDA Driver API or OpenCL (with the advantage that the latter is a completely open specification), rCUDA focuses on the most widely used: the CUDA Runtime. These alternatives might be explored in the future employing tools similar to rCUDA for these APIs, such as VCL [15] for OpenCL.

Another interesting use of rCUDA is for educational purposes, as it can also be used in academic environments, by providing a way of sharing a remote access to a few GPUs to dozens of students. This usage would report significant economic savings to educational institutions, while the slightly increased execution time is not a concern in this context.

Note that the NVIDIA CUDA Runtime Library also allows CUDA executions on computers with no CUDA-compatible devices by means of the Device Emulation Mode, as GPU kernels are executed by the CPU emulating the many-core architecture of the GPU. Therefore, this facility could also be used for educational purposes. However, the resulting overhead is often unbearable for complete executions (indeed, this feature is intended for debugging purposes instead of a replacement of a physical accelerator).

Readers can find additional details about the rCUDA architecture and the possibilities it offers in [14], a more detailed description of the implementation with a discussion on energy consumption implications in [16], and a network-dependent performance prediction model in [17].

4. rCUDA on Virtual Machines

rCUDA was initially designed to provide access to GPGPU features to computers not owning a GPU by accessing remote computers equipped with that hardware, as explained in the previous section. However, we propose to additionally use this framework to access GPUs from VMs. In this case, from the point of view of the rCUDA architecture, the VMs are considered nodes without a physical GPU, and the host OS is that acting as a GPGPU server. Hence, the client middleware of rCUDA is installed on the guest OS (that executed by a VM), as a replacement of the NVIDIA Runtime library, while the rCUDA server is executed on the host OS.

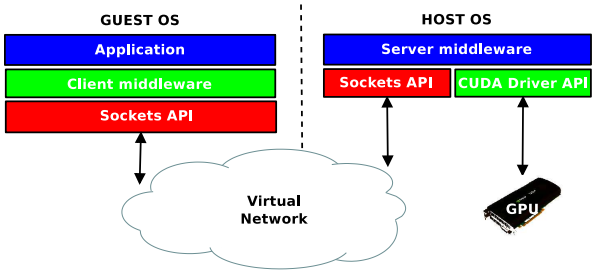

Figure 3. rCUDA on a VMM environment.

When used with VMs, the communication protocol in rCUDA will make use of the virtual network device to communicate the front-end and back-end middleware shown in Fig. 1. Therefore, the network has to be configured in a way that both the VM and the host OS can address IP packets to each other. Fig. 3 shows the diagram of Fig. 1 modified to reflect its usage in VM environments. We were able to successfully test the current implementation of rCUDA in KVM, VB-OSE, and VMware Server virtualization solutions. However, we were unable to run it in the current release of the Xen Hypervisor (3.4.3), as we could not gain access to a recent NVIDIA GPU driver that worked properly under the modified kernel for the administrative domain, with the ultimate reason being that this driver is not designed to support the Xen environment.

With rCUDA, multiple VMs running in the same physical computer can make concurrent use of all CUDA-compatible devices installed on the computer (as long as there is enough memory on the devices to be allocated by the different applications). Furthermore, although not addressed in this paper, rCUDA also allows the usage of a GPU located in a different physical computer. Effectively, rCUDA also features complete independence of the physical location of the GPU, as it may be located in a remote node instead of the local computer. In this case, the client middleware in the VM would forward the API call coming from the application through the virtual network to the host OS, which would route the IP datagrams containing

1. The gVirtuS software is supposed to support these undocumented functions. However, when looking at the corresponding source code, the following advise is found: “Routines not found in the cuda’s header files. KEEP THEM WITH CARE”.

the CUDA request through the real network interface towards the remote computer owning the GPU, where the rCUDA server middleware would be reached. That server middleware would forward the request to the real GPU. Obviously, the latency of the CUDA calls would increase due to the longer path between client and server. Nevertheless, as stated in [17], an increased latency is not expected to critically impact the performance of applications using rCUDA, as usually only a few relatively small-sized CUDA remote calls are needed, while those involving memory transfers are expected to be sufficiently large to hide latency. Without such a middleware enabling GPGPU capabilities, applications running inside VMs have the only choice to make use of the CPU to perform their computations.

In the following section, we provide an in-depth analysis of the use of rCUDA to enable GPGPU capabilities within VMs. We believe our proposal is the first work describing a VMM-independent production-ready CUDA solution for VMs.

5. Experimental Evaluation

In this section, we conduct a collection of experiments in order to evaluate the performance of the rCUDA framework on a VMM environment. The target system consists of two Quad-core Intel Xeon E5410 processors running at 2.33 GHz with 8 GB of main memory. An OpenSuse Linux distribution with kernel version 2.6.31 is run at both host and guest sides. The GPGPU capabilities are provided by two NVIDIA GeForce 9800 GX2 cards featuring a total of 4 NVIDIA G92 GPUs; the driver version is 190.53.

We selected two Open Source VMMs for the performance analysis: KVM (*userspace* `qemu-kvm` v0.12.3) and VB-OSE 3.1.6, with their respective VMs configured to make use of para-virtualized network devices. In addition, for load isolation purposes, each VM was configured to make use of only one processor core.

All benchmarks employed in our evaluation are part of the CUDA SDK. From the 67 benchmarks in the suite, we selected 10 representative SDK benchmarks of varying computation loads and data sizes, which use different CUDA features (see Table 1). A description of each benchmark can be found in the documentation of the SDK package [18]. The benchmarks were executed with the default options, other than setting the target device. In addition, benchmarks requiring OpenGL capabilities for their default executions (BT, BF, and ID) were executed with the `-qatest` argument, in order to perform a "quick auto test", which does not make use of the graphics-oriented API. In this case, computations are made without graphically displaying the data. To make the original benchmark code compatible with rCUDA, which does not support the *C for CUDA* extensions, the pieces of code using these extensions were rewritten using the plain C API (only a 7% of the total effective source lines of code required being modified).

Table 1. SDK Benchmarks in our experiments

SDK name	Acronym	Data transfers
`alignedTypes`	AT	413.26 MB
`asyncAPI`	AA	128.00 MB
`bicubicTexture`	BT	4.25 MB
`BlackScholes`	BS	76.29 MB
`boxFilter`	BF	5.00 MB
`clock`	CLK	2.50 KB
`convolutionSeparable`	CS	36.00 MB
`fastWalshTransform`	FWT	64.00 MB
`imageDenoising`	ID	2.49 MB
`matrixMul`	MM	79.00 KB

The execution times reported in the next experiments are the minimum from 5 executions, in order to avoid eventual network and CPU noise. They reflect the elapsed time experienced by the users, that is, from the start of the execution of the application till the end of it, including operations such as standard I/O. The experiments are presented in this section in two different groups. First, those concerning one VM are presented. Later, we introduce experiments involving several VMs being concurrently executed in our testbed.

5.1. Single Virtual Machine

We first analyze the performance of the CUDA SDK benchmarks running in a VM using rCUDA, and compare their execution times with those of a native environment —i.e., using the regular CUDA Runtime library in a non-virtualized environment. The results of this experiment are reported in Fig. 4. In addition, Table 2 compares the results of the native and virtualized environments using the real accelerator with those obtained using the NVIDIA Device Emulation Mode (see Section 3). It would also be interesting including in Table 2 data on execution timings for a version of the benchmarks that makes only use of the CPU, apart from those of the Device Emulation Mode. However, as it is difficult to find optimized algorithms for CPUs performing the same operations as all of our benchmarks, and those included in the SDK package are often naive versions, we cannot present such a comparison. Nevertheless, it is not strictly required for understanding the experiments presented and, additionally, the convenience of using virtualized remote GPUs instead of the local CPU was previously discussed [14].

Table 2. Execution times (in seconds and normalized) on the different environments

Bench.	Native	rCUDA				CUDA emulation			
		KVM		VB-OSE		KVM		VB-OSE	
	Time	Time	Norm.	Time	Norm.	Time	Norm.	Time	Norm.
AT	6.71	10.27	1.53	40.68	6.06	477.61	71.18	580.45	86.50
AA	0.28	1.55	5.54	7.32	26.14	33.08	118.14	42.37	151.32
BT	0.35	0.53	1.51	0.77	2.20	11.06	31.60	14.87	42.49
BS	2.07	3.33	1.61	6.35	3.07	731.96	353.60	876.16	423.27
BF	0.36	0.60	1.67	0.89	2.47	0.83	2.30	0.97	2.69
CLK	0.17	0.28	1.65	0.33	1.94	0.63	3.71	1.04	6.12
CS	0.75	1.55	2.07	3.14	4.19	308.13	410.84	388.25	517.67
FWT	3.04	4.31	1.42	7.31	2.40	780.20	256.64	1019.61	335.40
ID	0.30	0.46	1.53	0.60	2.00	93.77	325.90	102.84	342.80
MM	0.18	0.30	1.67	0.34	1.89	0.62	3.44	0.86	4.78

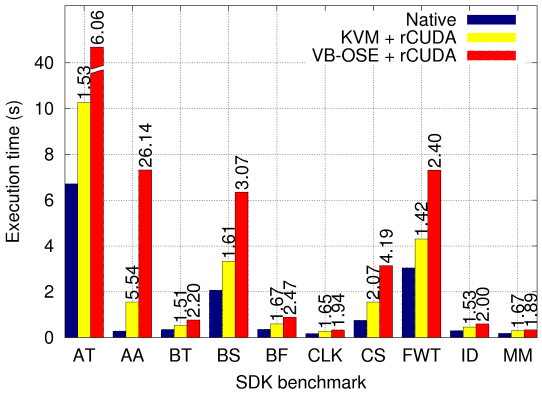

Figure 4. Native vs. KVM and VB-OSE. Normalized times annotated.

Results show that, for the evaluated benchmarks, the combination of KVM + rCUDA performs much better than VB-OSE + rCUDA. The reason for these differences will probably mostly lay on the way each VMM manages the virtual I/O. However, as the focus of the paper is analyzing the feasibility of using GPUs in VMs and not the discussion of a performance comparison between VMMs, (in other words, the analysis of the specifics of different VMMs is out of the scope of this work), we will not pursue further the causes of this difference. Hereafter, for simplicity and brevity, the results for VB-OSE are not further discussed, as compared with those of KVM they report similar behavior but at a higher scale.

Table 2 shows that the execution times of the benchmarks running in the VM employing the physical accelerator by means of rCUDA are up to two orders of magnitude lower than those of the benchmarks in the emulation mode. Indeed, the performance of KVM + rCUDA is close to that of the native executions. Therefore, even though the combination KVM + rCUDA pays the penalty for a non-optimized host-guest communication, using this general approach is feasible. Unfortunately, we cannot compare the overhead of the rCUDA-based solution with that of solutions based in prior middleware such as GViM or vCUDA because (1) GViM and vCUDA software are not publicly available, (2) in their associated papers, CUDA 1.1 was used instead of the more recent version 3.1 rCUDA uses, and (3) both used the Xen virtual platform[2]. On the other hand, we already mentioned that, unfortunately, we could not get the public version of gVirtuS working in our test-bed. For reference purposes, executions up to 5.28 times slower using vCUDA with respect to those on a native environment can be extracted from [13], up to 1.25 in the case of GViM in [12], and up to 7.12 and 2.98 for gVirtuS [11] when using TCP-based and VMM-dependent communications, respectively. Nevertheless, as previously stated, rCUDA aims at attaining a performance somewhere between those prior prototypes with the advantage of featuring VMM independence.

Fig. 5 specifies the time required by network transfers, thus illustrating that the overhead of KVM + rCUDA mostly originates in the network. Network transfer times have been measured as the addition of the times spent in data sending in both server and client middleware sides of rCUDA. A conclusion from this experiment is that a shared-memory scheme for communications may improve the performance at the expense of losing VMM independence. However, losing VMM independence would lead to a significant reduction in the flexibility provided by rCUDA, which is the main benefit of this package. Therefore, in the rest of this section we will analyze the exact causes for the overhead introduced by virtual network transfers in order to assess if they can be improved while maintaining VMM independence. As mentioned

2. NVIDIA GPU drivers supporting up to version 1.1 of CUDA worked on the Xen dom0, but this is no longer the case for more recent versions like those used by us.

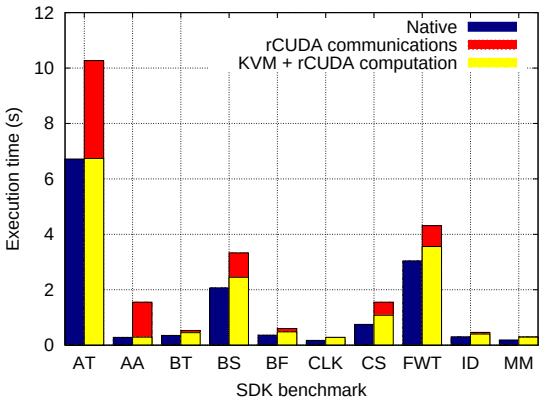

Figure 5. Breakdown of KVM + rCUDA.

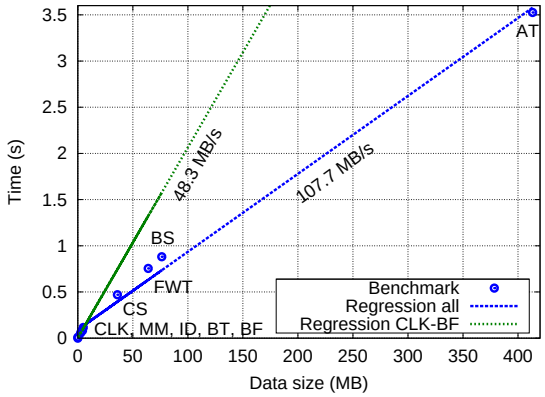

Figure 6. Correlation between dataset size and total network transfer time on KVM. For reference purposes, linear regression lines for all points as well as for the 5 points closest to the origin of coordinates are drawn.

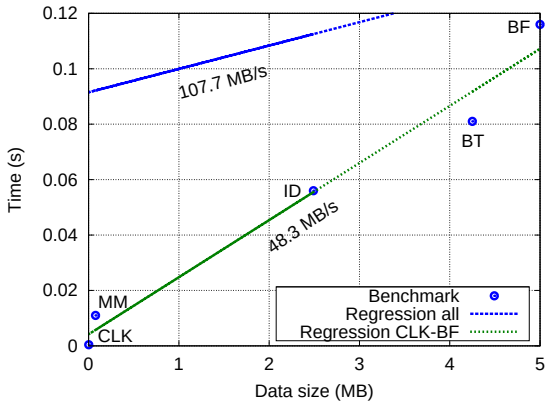

Figure 7. Zoom of the area closest to the origin of coordinates of the plot in Fig. 6.

before, we will focus this analysis for the KVM VMM.

Fig. 6 relates the dataset size of each benchmark with the network transfer time, identifying the different overheads caused by network communications shown in Fig. 5. Note, however, that as the AA benchmark performs asynchronous memory transfers, which might be overlapped with CPU and GPU computations, the time spent in network communications might not be proportional to the global overhead introduced by those; therefore, the data corresponding to this benchmark is not included in the figure and skipped during the rest of this section. As shown in the figure, the time spent in network communications seems to be proportional to the data size of the problem, as other transfers related with the application-level communication protocol become negligible for "large" datasets. Nevertheless, as the points for CLK, MM, ID, BT, and BF in Fig. 6 are so close to the axis origins that they cannot be clearly distinguished, Fig. 7 provides a zoom of the plot area for those points. As can be seen, the points in that figure evidence a significantly lower network throughput than AT, CS, FWT, or BS (i.e., those farther from the origin of coordinates). In order to analyze the reason for this lower throughput, we determined the degree of utilization of the bandwidth of the virtual network. To do so, we analyzed the average transfer rates of the memory transfer operations for each of the benchmarks. Additionally, a simple ping-pong test revealed a peak transfer rate between KVM VMs and the host OS of 126 MB/s. The results of this analysis are shown in Table 3, illustrating that in some cases the experienced average transfer rate (TR) was much lower than this value.

The bandwidth for the smallest data sizes shown in Table 3 may be far from the theoretical peak of the network due to the intrinsics of the TCP protocol. That is, the cause for these low transfer rates may be related with the size of the memory transfer operations of each benchmark and the configuration of the TCP transmission window. Inspecting the source code we determined that data transfers are performed in chunks of sizes between 2 KB and 32 MB. Numbers in Table 3 reveal that the benchmarks that transfer data in chunks smaller than 1 MB yield specially low average transfer rates (below 50% of the peak). To confirm the relationship between the low bandwidth and the intrinsics of TCP, we next compare the results of the ping-pong test for data payload sizes up to 1 MB with the average transfer rates obtained in our executions. Fig. 8 shows that the transfer rates obtained by the ping-pong test vary from 16 to a maximum of 119 MB/s. However, as illustrated in the figure, the average throughputs for

Table 3. Average transfer rate (TR) obtained for each benchmark

	AT	BT	BS	BF	CLK	CS	FWT	ID	MM
Data (MB)	413.26	4.25	76.29	5.00	$2 \cdot 10^{-3}$	36.00	64.00	2.49	0.08
Time (s)	3.53	0.08	0.88	0.12	$4 \cdot 10^{-4}$	0.47	0.75	0.06	0.01
TR (MB/s)	117.19	52.44	86.61	42.93	6.17	76.55	84.82	44.21	6.91
Transfers	13	4	5	5	1	2	2	5	3
Chunk	32 MB	1 MB	15 MB	1 MB	2 KB	18 MB	32 MB	512 KB	15-40 KB

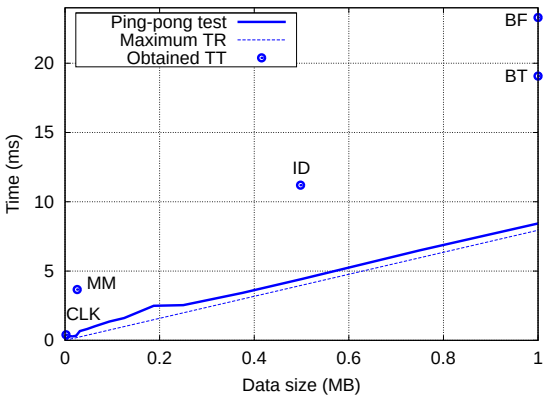

Figure 8. Peak transfer rate (TR) of the virtual network, ping-pong test results, and minimum obtained transfer times (TT).

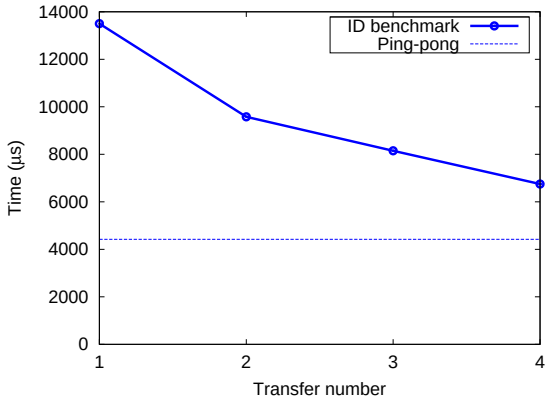

Figure 9. Experienced times in 4 consecutive transfers for the ID benchmark vs. those obtained from the ping-pong test for that data size.

the benchmarks measured in our experiments are still below those obtained with this simple test. Therefore, the reason may be that the TCP layer protocol is based on a transmission window size which is progressively —but not immediately— adapted to the amount of transferred data. To assess the impact of this phenomenon in our experiments, we performed a careful analysis of the time employed by each memory transfer operation. In Fig. 9, we show the transfer times for 4 consecutive and identical memory transfers of one of the benchmarks. As expected, the figure reveals that the transfer of the first large packet takes significantly longer time than the following transfers of the same size, which require times close to those of the ping-pong test, as the TCP transfer window is progressively being increased to reach the appropriate size for that data payload. Therefore, the low average transfer rates shown in Table 3 are quite confidently explained by the transport layer protocol particularities regarding how the window in the transmitter side is managed by TCP. This window management also explains the network overhead shown in Fig. 5.

The network analysis presented above reveals that, in order to obtain faster network transmissions, on the one hand memory transfer operations should involve as much data as possible and, on the other hand, the initial TCP window size should be increased. One proposal for future work in rCUDA is to try to artificially open the transmission window when the TCP connection between the client and the server is established. However, as the maximum effective transfer rate of the virtual network is 126 MB/s, when compared to native solutions, where memory transfers directly use the PCI Express (PCIe) bus (with an effective transfer rate around 5.5 GB/s in our tests over a PCIe v2.0 x16), the overhead when performing GPGPU over a VM using a virtual network will never be reduced below a minimum value. In this regard, the experiments show that despite the increase of the throughput with large data transfers from VMs, the overhead of transferring large amounts of data is higher than the benefits obtained with the higher throughput, and proportional to the dataset size. In general, this overhead could be reduced with improved support for the virtual network device provided by VMM developers.

5.2. Multiple Virtual Machines

To measure the usability of a highly loaded system using rCUDA, we performed some scalability tests running up to eight VMs on the target platform, making use of the 4 GPUs of the computer via rCUDA.

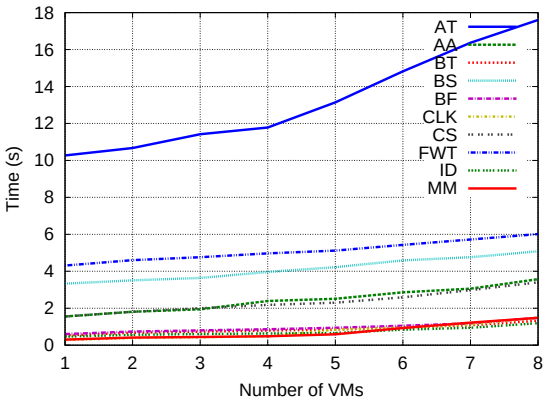

Figure 10. Concurrently running the benchmarks on multiple KVM VMs and GPUs.

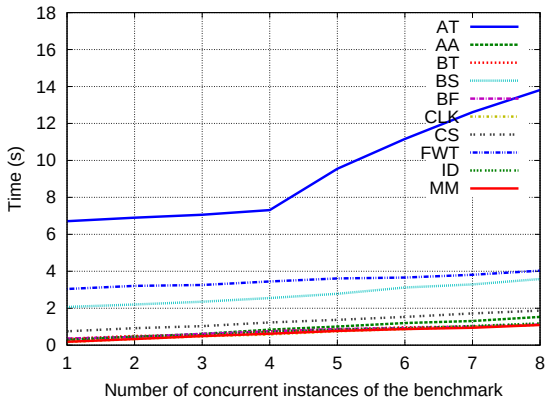

Figure 11. Concurrently running several benchmarks on the native environment.

The results were compared with those corresponding to concurrent executions in a native environment.

Fig. 10 shows the results employing from one to eight KVM VMs. The GPUs used by each VM are distributed in a round-robin fashion as the required number increases. Thus, as soon as 5 or more concurrent VMs are run, the GPUs become a shared resource. The results show a smooth degradation in performance up to 4 VMs, as the different instances are only competing for the network channel and the PCIe bus; from five to eight VMs, the overhead introduced is more evident, as the GPUs also become a shared resource. For instance, for the AT sample, the most time-consuming benchmark in the set, the overhead when executed in four VMs is 14.7%, raising to 71.4% when the eight VMs are used.

On the other hand, the native concurrent tests, shown in Fig. 11 —where the different GPUs of the system are used following the same policy as in the prior case— present scalability results similar to those obtained in the VM environment. For reference, the AT sample overhead when running 4 concurrent instances is 8.9%, reaching 105.9% for 8 instances as, similarly to the VM environment, there is a competition for the PCIe bus up to 4 instances, while there is an additional competition for the GPU resources starting from 5 concurrent instances.

Interestingly, in our case studies we noticed better scalability in rCUDA than in the native environment. As extracted from Fig. 5, CPU and GPU computation time, in addition to that of the data transfers across the PCIe bus, present no major differences in native and KVM tests. Instead, the difference in time between both environments is mostly caused by the network transfers.

6. Conclusions

The rCUDA framework enables remote CUDA Runtime API calls, thus enabling an application making use of a CUDA-compatible accelerator to be run on a host without a physical GPU. Thus, this framework can offer GPGPU acceleration to applications running either in a remote host, or similarly in a VM, where no direct access to the hardware of the computer is provided.

In this paper, we have reported a variety of experimental performance results based on a set of CUDA SDK benchmarks, showing that the rCUDA framework employing TCP/IP sockets for communications between its front-end and back-end components can deliver CUDA-based acceleration support to multiple VMs running in the same physical server equipped with several GPUs. In addition, the experiments reported an acceptable overhead —compared with native executions— for most applications ready to be run in a virtualized environment. Our tests revealed a good level of scalability, thus demonstrating that this solution can be run in a productive system with concurrent VMs in execution. In summary, our results state that it is possible to provide GPGPU capabilities with reasonable overheads to processes running in a VM, while keeping VMM independence.

For future work, we are planning to explore new communication protocols for rCUDA in order to reduce overhead in a VM scenario. For instance, reliable datagram sockets (RDS) could offer faster communications while maintaining VMM independence. Once having equipped rCUDA with a fast and reliable VMM-independent communications library, additional VMM-dependent mechanisms will be evaluated.

Further Information

For further details on rCUDA, visit its web pages:

- http://www.gap.upv.es/rCUDA
- http://www.hpca.uji.es/rCUDA

Information about how to obtain a copy of rCUDA can be found there.

Acknowledgments

The researchers at Universitat Politècnica de València were supported by PROMETEO from Generalitat Valenciana (GVA) under Grant PROMETEO/2008/060.

The researchers at Universitat Jaume I were supported by the Spanish Ministry of Science and FEDER (contract no. TIN2008-06570-C04), and by the Fundación Caixa-Castelló/Bancaixa (contract no. P1-1B2009-35).

References

[1] A. Gaikwad and I. M. Toke, "GPU based sparse grid technique for solving multidimensional options pricing PDEs," in *Proceedings of the 2nd Workshop on High Performance Computational Finance*, D. Daly, M. Eleftheriou, J. E. Moreira, and K. D. Ryu, Eds. ACM, Nov. 2009.

[2] D. P. Playne and K. A. Hawick, "Data parallel three-dimensional Cahn-Hilliard field equation simulation on GPUs with CUDA," in *International Conference on Parallel and Distributed Processing Techniques and Applications*, H. R. Arabnia, Ed., 2009, pp. 104–110.

[3] E. H. Phillips, Y. Zhang, R. L. Davis, and J. D. Owens, "Rapid aerodynamic performance prediction on a cluster of graphics processing units," in *Proceedings of the 47th AIAA Aerospace Sciences Meeting*, no. AIAA 2009-565, Jan. 2009.

[4] S. Barrachina, M. Castillo, F. D. Igual, R. Mayo, E. S. Quintana-Ortí, and G. Quintana-Ortí, "Exploiting the capabilities of modern GPUs for dense matrix computations," *Concurr. Comput. : Pract. Exper.*, vol. 21, no. 18, pp. 2457–2477, 2009.

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

[6] W. Huang, J. Liu, B. Abali, D. K. Panda, and Y. Muraoka, "A case for high performance computing with virtual machines," in *20th Annual International Conference on Supercomputing*, G. K. Egan, Ed., Cairns, Queensland, Australia, Jun. 2006, pp. 125–134.

[7] M. Dowty and J. Sugerman, "GPU virtualization on VMware's hosted I/O architecture," in *First Workshop on I/O Virtualization*, M. Ben-Yehuda, A. L. Cox, and S. Rixner, Eds. USENIX Association, December 2008.

[8] H. A. Lagar-Cavilla, N. Tolia, M. Satyanarayanan, and E. de Lara, "VMM-independent graphics acceleration," in *VEE '07: Proceedings of the 3rd international conference on Virtual execution environments*. New York, NY, USA: ACM, 2007, pp. 33–43.

[9] A. Munshi, Ed., *OpenCL 1.0 Specification*. Khronos OpenCL WG, 2009.

[10] NVIDIA, *NVIDIA CUDA Programming Guide Version 3.1*. NVIDIA, 2010.

[11] G. Giunta, R. Montella, G. Agrillo, and G. Coviello, "A GPGPU transparent virtualization component for high performance computing clouds," in *Euro-Par 2010 - Parallel Processing*, ser. LNCS, P. D Ambra, M. Guarracino, and D. Talia, Eds. Springer Berlin / Heidelberg, 2010, vol. 6271, pp. 379–391.

[12] V. Gupta, A. Gavrilovska, K. Schwan, H. Kharche, N. Tolia, V. Talwar, and P. Ranganathan, "GViM: GPU-accelerated virtual machines," in *3rd Workshop on System-level Virtualization for High Performance Computing*. NY, USA: ACM, 2009, pp. 17–24.

[13] L. Shi, H. Chen, and J. Sun, "vCUDA: GPU accelerated high performance computing in virtual machines," in *IEEE International Symposium on Parallel & Distributed Processing (IPDPS'09)*, 2009.

[14] J. Duato, F. D. Igual, R. Mayo, A. J. Peña, E. S. Quintana-Ortí, and F. Silla, "An efficient implementation of GPU virtualization in high performance clusters," in *Euro-Par 2009, Parallel Processing — Workshops*, ser. LNCS, vol. 6043. Springer-Verlag, 2010, pp. 385–394.

[15] A. Barak, T. Ben-Nun, E. Levy, and 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)*, Sep. 2010.

[16] J. Duato, A. J. Peña, F. Silla, R. Mayo, and 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 & Simulation (HPCS 2010)*, Jun. 2010, pp. 224–231.

[17] J. Duato, A. J. Peña, F. Silla, R. Mayo, and E. S. Quintana-Orti, "Performance of CUDA virtualized remote GPUs in high performance clusters," in *International Conference on Parallel Processing (ICPP)*, Sep. 2011.

[18] NVIDIA, "NVIDIA CUDA SDK code samples," http://developer.download.nvidia.com/compute/cuda/3_1/sdk/gpucomputingsdk_3.1_linux.run, 2010.