

A New Approach to rCUDA

José Duato, Antonio J. Peña, and Federico Silla¹
Juan C. Fernández, Rafael Mayo, and Enrique S. Quintana-Ortí²

Abstract—In this paper we propose a first step towards a general and open source approach for using GPGPU (General-Purpose Computation on GPUs) features within virtual machines (VMs). In particular, we describe the use of rCUDA, a GPGPU virtualization framework, to permit the execution of GPU-accelerated applications within 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.

Keywords—CUDA, GPUs, GPGPU, high performance computing, virtual machines, virtualization.

I. INTRODUCTION

MANY-CORE specialized processors 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], image analysis [2], and many others. These hardware accelerators offer a large amount of processing elements and high processor-to-memory bandwidth, so that applications featuring a high rate of computations per data item can attain high performance. In addition, 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 straight-forward that the usage of virtual machines (VMs) in HPC is an active area of research [3]. 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 it in an exclusive way.

Processes running in a VM may also require the services of a GPU in order to accelerate part of their computations. To do so, the GPGPU capabilities should be exposed to VMs so 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., [4]), 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 using high-level standard interfaces such as Direct3D [4] or OpenGL [5], while using them for GPGPU requires APIs like OpenCL [6] or NVIDIA CUDA [7], which significantly differ from their graphics-processing oriented counterparts. On the other hand, the few efforts done up to now to expose CUDA capabilities to VMs [8], [9], [10] (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), or (4) employ inefficient communication protocols between their middleware sides.

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 GPGPU API supporting the latest NVIDIA GPUs. The framework we employ, named *rCUDA* [11], [12], 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. Note however that although the primary goal of rCUDA was providing a way to reduce the number of GPU-based accelerators in a cluster, in this paper we extend its applicability to also expose 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 environment, we cannot disclose the results due to licensing policies.

II. BACKGROUND ON CUDA VIRTUALIZATION

In addition to rCUDA, which will be described in the following section, there are other approaches that

¹DISCA, Universitat Politècnica de València (UPV), e-mail: {jduato, fsilla@disca.upv.es}, apenya@gap.upv.es.

²DICC, Universitat Jaume I (UJI), e-mail: {jfernand,mayo,quintana}@icc.uji.es.

pursue the virtualization of the CUDA Runtime API for VMs. All solutions feature a distributed middleware comprised of two parts: the front-end and the back-end. The former 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 [10] 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 [10] 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 [9] 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. This solution, based in CUDA 1.1, does not seem to implement the whole API.

Finally, gVirtuS [8] (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).

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.

III. 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, 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:

- **The client middleware** consists of a collection of wrappers which replace the NVIDIA CUDA Runtime (provided 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 re-

trieving the results back, providing applications with the illusion of direct access to a real 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 application over an independent GPU context, thus attaining GPU multiplexing.

Communication between rCUDA components is performed employing a highly-tuned, TCP-based application-level protocol.

The current release of rCUDA (2.0) 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 [10]¹.

Although there are other GPGPU APIs such as the CUDA Driver API or OpenCL, rCUDA focuses on the most widely used: the CUDA Runtime, as at the moment applications using CUDA seem to achieve higher performance than when employing OpenCL [13]. These alternatives might be explored in the future employing tools similar to rCUDA for these APIs, such as VCL [14] for OpenCL.

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. 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 in [11], and a more detailed description of the implementation with a discussion on energy consumption implications in [12]. Further details are available on the Web (www.gap.upv.es/rCUDA, www.hpca.uji.es/rCUDA).

IV. 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, 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.

¹gVirtuS software is supposed to support the 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”.

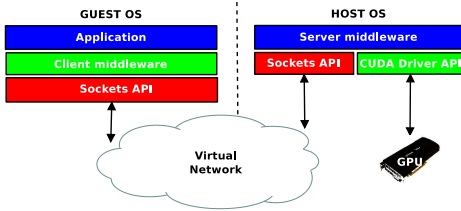


Fig. 1. rCUDA architecture on a VMM environment.

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. 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. 1 shows an rCUDA architecture diagram 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 a recent 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.

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.

V. 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 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

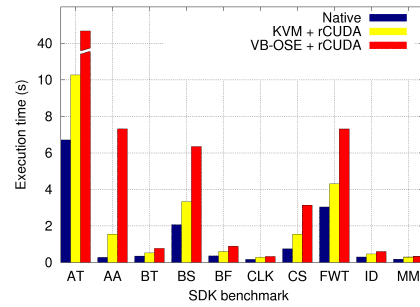


Fig. 2. Native vs. KVM and VB-OSE.

loads and data sizes, which use different CUDA features: `alignedTypes` (AT), `asyncAPI` (AA), `bicubicTexture` (BT), `BlackScholes` (BS), `boxFilter` (BF), `clock` (CLK), `convolutionSeparable` (CS), `fastWalshTransform` (FWT), `imageDenoising` (ID), and `matrixMul` (MM). A description of each benchmark can be found in the documentation of the SDK package [15]. 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. 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).

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, from the start of the execution of the application till the end of it. The experiments are presented in this section in two groups. First, those concerning one VM are presented. Later, we introduce experiments involving several VMs being concurrently executed.

A. 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. 2. It would also be interesting including data for a version of the benchmarks that makes only use of the CPU. 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 [11].

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 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, 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.

Figure 2 shows that 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 rCUDA uses, and (3) both used the Xen virtual platform². On the other hand, we already mentioned that 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 [10], up to 1.25 in the case of GViM [9], and up to 7.12 and 2.98 for gVirtuS [8] 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. 3 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 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 know if they can be improved.

Fig. 4 relates the dataset size of each benchmark with the network transfer time. 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

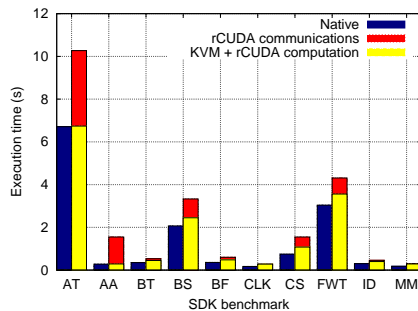


Fig. 3. Breakdown of KVM + rCUDA.

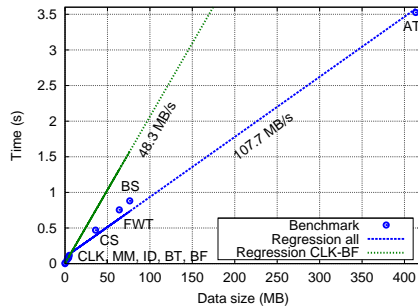


Fig. 4. Dataset size and total network transfer time on KVM.

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. 4 are so close to the axis origins that they cannot be clearly distinguished, Fig. 5 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. In order to analyze the reason for this lower throughput, we determined the degree of utilization of the bandwidth of the virtual network examining 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 I, 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 I may be far from the theoretical peak of the network due to the intrinsic of the TCP protocol. That is, the cause for these low transfer rates may be related with the size of the memory transfer operations 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 I 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

²NVIDIA 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.

TABLE I
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

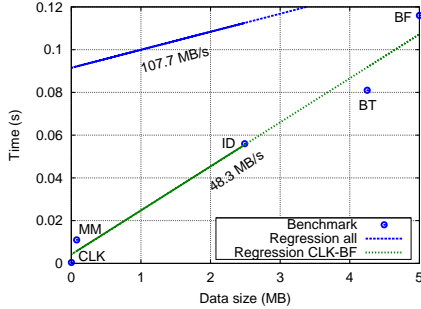


Fig. 5. Area closest to the coordinate origin in Fig. 4.

and the intrinsic 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. 6 shows that the transfer rates obtained by the ping-pong test vary from 16 to a maximum of 119 MB/s. However, the average throughputs for the benchmarks 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. 7 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 I are 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 in Fig. 3.

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 upon the TCP connection establishment. However, as the maximum effective transfer rate of the virtual network is 126 MB/s, when compared to native

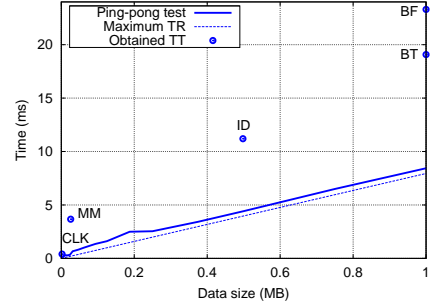


Fig. 6. Ping-pong test, peak transfer rate (TR) of the virtual network, and minimum obtained transfer times (TT).

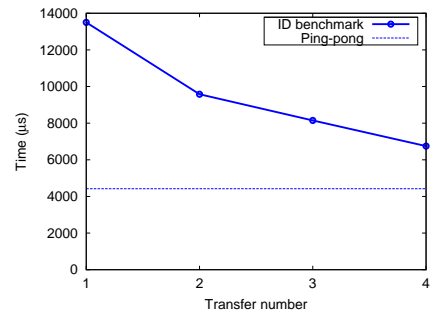


Fig. 7. Four consecutive transfers in ID vs. ping-pong test.

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 from VMM developers.

B. Multiple Virtual Machines

To measure the usability of a highly loaded system using rCUDA, we performed some scalability tests running up to 8 VMs on the target platform, making use of the 4 GPUs of the computer via rCUDA. The results were compared with those corresponding to concurrent executions in a native environment.

Fig. 8 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

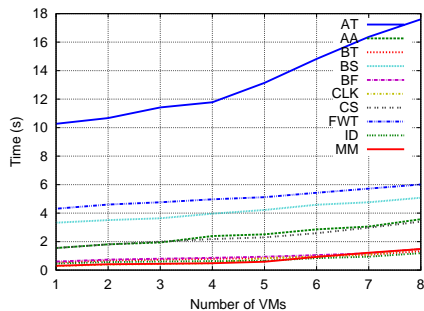


Fig. 8. Concurrency tests on multiple KVM VMs and GPUs.

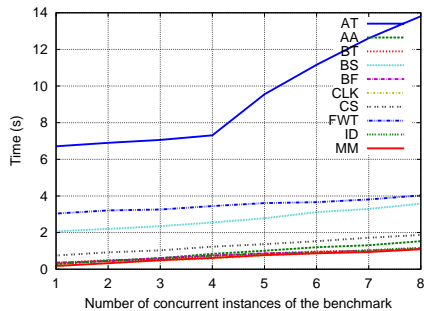


Fig. 9. Concurrency tests on the native environment.

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%, but it raises to 71.4% when the eight VMs are used.

On the other hand, the native concurrent tests shown in Fig. 9—where the different GPUs of the system are used following the same policy as in the prior case—present scalability results close 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 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 (see Fig. 3), the difference in time between both environments is mostly caused by network transfers.

VI. CONCLUSIONS

rCUDA 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 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 can deliver CUDA-based acceleration support to multiple VMs running in the same physical server equipped with several GPUs. The experiments reported an acceptable overhead—if 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.

ACKNOWLEDGEMENTS

The researchers at UPV were supported by PROMETEO from GVA (Generalitat Valenciana) under Grant PROMETEO/2008/060, while those at UJI 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*, 2009.
- [2] Y. C. Luo and R. Duraiswami, “Canny edge detection on NVIDIA CUDA,” in *Computer Vision on GPU*, 2008.
- [3] W. Huang, J. Liu, B. Abali, D. K. Panda, and Y. Muraoka, “A case for high performance computing with virtual machines,” in *ICS*, 2006.
- [4] M. Dowty and J. Sugerma, “GPU virtualization on VMware’s hosted I/O architecture,” in *First Workshop on I/O Virtualization*, December 2008.
- [5] H. A. Lagar-Cavilla, N. Tolia, M. Satyanarayanan, and E. de Lara, “VMM-independent graphics acceleration,” in *VEE*, 2007, pp. 33–43.
- [6] *OpenCL 1.0 Specification*, Khronos OpenCL WG, 2009.
- [7] *NVIDIA CUDA Programming Guide Version 3.1*, 2010.
- [8] G. Giunta, R. Montella, G. Agrillo, and G. Coviello, “A GPGPU transparent virtualization component for high performance computing clouds,” in *Euro-Par*. 2010.
- [9] V. Gupta, A. Gavrilovska, K. Schwan, H. Khariche, 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, 2009, pp. 17–24.
- [10] L. Shi, H. Chen, and J. Sun, “vCUDA: GPU accelerated high performance computing in virtual machines,” in *IPDPS*, 2009.
- [11] 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*, 2010.
- [12] 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 *HPCS*, June 2010, pp. 224–231.
- [13] K. Karimi, N. G. Dickson, and F. Hamze, “A Performance Comparison of CUDA and OpenCL,” *ArXiv e-prints*, 2010, Online: <http://arxiv.org/pdf/1005.2581v2>.
- [14] A. Barak, T. Ben-Nun, E. Levy, and A. Shiloh, “A package for OpenCL based heterogeneous computing on clusters with many GPU devices,” in *PPAAC*, 2010.
- [15] NVIDIA, “NVIDIA CUDA SDK code samples,” http://developer.download.nvidia.com/compute/cuda/3_1/sdk/gpucomputingsdk_3.1_linux.run, 2010.