# rCUDA: Reducing the Number of GPU-Based Accelerators in High Performance Clusters

José Duato, Antonio J. Peña, and Federico Silla
*Universidad Politécnica de Valencia (UPV)*
{*jduato, fsilla*}*@disca.upv.es, apenya@gap.upv.es*

Rafael Mayo and Enrique S. Quintana-Ortí
*Universidad Jaume I (UJI)*
{*mayo, quintana*}*@icc.uji.es*

## ABSTRACT

*The increasing computing requirements for GPUs (Graphics Processing Units) have favoured the design and marketing of commodity devices that nowadays can also be used to accelerate general purpose computing. Therefore, future high performance clusters intended for HPC (High Performance Computing) will likely include such devices. However, high-end GPU-based accelerators used in HPC feature a considerable energy consumption, so that attaching a GPU to every node of a cluster has a strong impact on its overall power consumption. In this paper we detail a framework that enables remote GPU acceleration in HPC clusters, thus allowing a reduction in the number of accelerators installed in the cluster. This leads to energy, acquisition, maintenance, and space savings.*

**KEYWORDS:** Energy saving, virtualization, high performance computing, clusters, CUDA.

## 1. INTRODUCTION

Current personal computers exhibit a nonnegligible energy consumption; e.g., a desktop computer for gaming may consume around 500 W. In HPC clusters and data processing centers, a single node may consume 50% more energy. Hence, minimizing energy consumption has become a hot topic in the design of large data centers [1, 2].

On the other hand, the advances in hardware and software for GPUs have initiated the so-called GPU Computing or GPGPU (General Purpose Computing on GPU); that is, the use of GPUs for solving general purpose tasks, which are different from those they were initially designed for. This trend has been favoured by the relatively low cost of GPUs as high performance processors, mostly caused by the gaming business volume. Thus, in the near future, large clusters will probably adopt the use of these relatively inexpensive and powerful devices as a way of accelerating parts of the code of the applications they are serving. However, current GPUs have a great impact on the power consumption of the system, as a high-end GPU may well increase the power consumption of an HPC cluster node up to 30%.

Virtualization techniques may provide significant energy savings, as they enable a larger resource usage by sharing a given hardware among several users, thus reducing the required amount of instances of that particular device. As a result, virtualization is being increasingly adopted in data centers. In this way, virtualizing GPUs may report power and cost benefits.

The most extended virtualization techniques are based on software solutions, such as the well-spread VMware (by VMware Inc.), or Xen [3], which is highly appreciated by the research community because it is Open Source. Other system virtualization solutions include Microsoft Virtual PC, Parallels Desktop, Oracle's VirtualBox, and the Kernel-based Virtual Machine (KVM). However, all these solutions virtualize an entire system, which is not useful for GPU-based accelerators, as only the GPU requires to be virtualized and be directly offered to the rest of the cluster. On the other hand, virtualization software mechanisms that emulate a given device are not useful for HPC applications, because of the unacceptable overhead they may introduce.

Therefore, it is desirable an alternative for virtualizing specific resources that avoids software emulation by multiplexing the real resource, so that it can be shared by several nodes in the cluster. This alternative would save energy and resources while delivering an acceptable performance.

In this paper we describe rCUDA. This framework enables the concurrent usage of CUDA-compatible GPUs remotely. To enable a remote GPU-based acceleration, our framework creates virtual CUDA-compatible devices on those machines without a local GPU. These virtual devices represent physical GPUs located in a remote host offering GPGPU services. Thus, all of the nodes are able to concurrently access the whole set of CUDA accelerators installed

978-1-4244-6830-0/10/$26.00 

in the cluster, independently of which nodes the GPUs are physically attached to. In another words, our solution aims at offering a noticeable reduction in execution time to computationally-intensive applications running in an HPC cluster equipped with only a few CUDA-compatible accelerators, enabling remote hardware acceleration. As far as we know, this is the first solution to enable a general CUDA remote acceleration in HPC clusters, as previous works made use of GPU acceleration over different nodes using a combination of MPI and CUDA [4]. Our experiments demonstrate that rCUDA leads to a reduction of the number of GPUs required in the system, thus attaining considerable energy savings.

The rest of this paper is organized as follows: Section 2 introduces some prior work related with our contribution. In Section 3 we review the NVIDIA CUDA framework. Sections 4 and 5 describe the details and limitations of our proposal. In Section 6 we provide some experimental results, while Section 7 discusses energy consumption implications. Concluding remarks close the paper in Section 8.

## 2. PRIOR WORK ON VIRTUALIZATION

In this section we introduce a virtualization taxonomy and then describe previous work related with our proposal. We review GPU virtualization for graphics processing and introduce prior work on GPU virtualization in the field of GPGPU. Note that although both virtualization areas use the same device, their related APIs significantly differ. Therefore, as there is no low-level standard interface to drive GPUs, the virtualization boundary has to be placed at the high-level standard APIs, thus leading to different strategies for virtualizing GPUs depending on the target usage.

### 2.1. Virtualization Taxonomy

We can divide virtualization into two major categories: front-end (application facing) virtualization and back-end (hardware facing) virtualization [5].

Back-end virtualization techniques run the device driver in the client machine, which has direct access to the physical hardware. This makes sense in a virtual machine monitor (VMM) environment, where the device is located in the same computer as the virtual machine (VM). However, in our environment the GPU we want to virtualize is located in a remote host. Thus, back-end virtualization is not a choice to implement virtualized remote GPUs across a network.

On the other hand, front-end techniques can be implemented in two flavours: *API remoting* and *device emulation*. As mentioned before, *device emulation* causes an overhead not feasible for an HPC environment. Instead, in *API remoting*, API calls are intercepted and forwarded to a remote host for its execution there. This technique is also referred to as *API interception*. This is the approach we follow to implement our virtualized remote GPUs.

### 2.2. GPU Virtualization for Graphics Processing

Unlike other devices such as storage or network controllers, GPUs lack of a standard low-level interface. In contrast, the common way to access the 3D acceleration facilities of the GPU is using one of the several standardized high-level APIs, such as Microsoft's Direct3D [6]; or OpenGL [7], an open and cross-platform specification.

There are several works from the research community on GPU virtualization for graphics acceleration: Chromium [8] makes use of API interception (employing wrapper libraries) to enable OpenGL parallel rendering on commodity clusters; Blink [9] and VMGL [10] follow a similar approach to provide hardware OpenGL-based rendering in UNIX-like VMs running on a VMM.

Other efforts are proprietary software: VMware's Virtual GPU [5] mimics a similar approach to virtualize the Direct3D API for Windows, and Parallels Desktop, which makes use of Intel's Virtualization Technology [11].

Although there are conceptual similarities between these works and our research, we found that even if dealing with the virtualization of the same device, the intrinsics of both OpenGL/Direct3D and CUDA APIs significantly differ. First, graphics APIs have to deal with graphic-related issues, such as flickering, object interposition (with the optimization opportunities this brings), graphics redirection, etc. Second, in a VMM environment, there are different features to take into account, such as portability, suspend and resume, and migration. Conversely, CUDA is mainly related with general purpose computing, not considering graphical representation issues.

### 2.3. GPU Virtualization for GPGPU

Shi et al. presented a first CUDA-oriented GPU virtualization solution, named vCUDA [12]. This package comprises an unspecified subset of 31 functions of the CUDA Runtime API version 1.1[1], explicitly excluding OpenGL and Direct3D interoperability support. It uses API interception to capture CUDA calls on the guest Operating System (OS) with a wrapper library, and to redirect them to the host OS where a stub service was running.

A major problem experienced by Shi et al. (as well as by us) is that the CUDA runtime library features some internal undocumented functions, with the aggravation that the CUDA Front End (*cudafe*), the first stage of the compilation trajectory managed by the CUDA compiler driver *nvcc*,

[1]The official documentation [13] reports a total of 61 API functions, including OpenGL and Direct3D interoperability, and excluding the high-level API for texture reference management.

automatically inserts calls to them. These hidden functions seem to provide support to easily use GPU kernels and variables from host[2] code. Shi et al. solved this issue by blindly redirecting the parameters of these functions to the host stub for its execution there.

In our approach, we avoid the use of these undocumented functions at the expense of loosing the support for the CUDA C extensions. In a distributed memory system, we cannot trust that these hidden functions are accessing the appropriate memory locations (consider, e.g., pointers to some data structure such as a kernel code). In addition, vCUDA tracks the GPU status to provide suspend and resume functionality, which is not needed in the HPC environment targeted by our solution. vCUDA also features a lazy mode to minimize world switches (switches between a guest and the VMM), which can cause a delay in error report, thus altering the behavior of a native execution.

Finally, vCUDA's authors mention that their solution is focused on portability across VMMs and OSs and, as a consequence, their transport protocol might be inefficient.

To address these issues, we have developed a cluster-oriented solution, specifically designed to run in an HPC cluster environment. Although applications using a virtualized remote GPU may experience an increase in their execution time, in comparison with that obtained with the usage of a local accelerator, the overhead due to the need to transfer the data to/from the remote resource can be insignificant compared with a much slower execution on the local CPU.

We previously presented an early overview and some performance results of our cluster-oriented solution [14, 15]. In this paper, we provide a more detailed description of our framework, as well as some discussion on its usability and limitations, energy saving implications, and further performance results based on the CUDA SDK examples.

## 3. BACKGROUND ON NVIDIA CUDA

This section offers a brief review of the terminology and main features of the NVIDIA CUDA framework. The reader can find a more detailed description in [16]. In addition, those who are already familiar with CUDA may skip the section.

The main feature of GPUs is their large amount of processing elements or cores, as well as the reduced size of their cache memory. The architecture is deeply pipelined and suited to an SIMD-like (Single-Instruction, Multiple-Data) programming model. GPUs are located in cards with high-speed DRAM (Dynamic Random Access Memory), and connected to the computer via a high-speed input/output interface, typically a PCI-Express bus (PCIe).

CUDA enables the use of NVIDIA GPUs as coprocessors in order to accelerate certain parts of a program, generally those with a high computing load per data.

Two APIs are currently provided: the low-level *Driver API* and the high-level *Runtime API*. The latter is implemented on top of the former, and offers a set of extensions to the C language, known as *C for CUDA*. These extensions introduce some new syntax providing, for example, a simpler manner of specifying the kernel execution parameters.

The high-level API is easier to use than its low-level counterpart. In addition to support for C extensions, which make the code more concise, it provides implicit functionalities (initialization, context and module management) and supports device emulation. In contrast, the driver API offers a broader range of functionalities (such as explicit module management), and therefore a higher degree of control.

The portion of the code to be executed on the accelerator is written as a *kernel*. At run time, kernels are transferred to and executed on the GPU by the device driver.

Finally, a compiler driver (*nvcc*) is also included as part of the CUDA framework. It is in charge of driving the CUDA compilation trajectory. It uses a set of utilities, including the host native C/C++ compiler, to separate host and device code, and compile them. Any source file that makes use of the C extensions has to be compiled with this utility [16].

## 4. CUDA REMOTING FRAMEWORK

In our proposal, each node in the cluster can "see" any of the CUDA-compatible accelerators installed in the cluster nodes. Actually, they can access GPUs as if they were directly connected to its local PCIe port. Remote GPUs are virtualized devices made available by a wrapper library replacing the CUDA Runtime. Basically, this library forwards the API calls to a remote server, and retrieves the results from those remote executions to offer them to the calling application. Hence, there is no difference on the behavior of the API calls, apart from the slightly increased execution time caused by network transmission delays.

Our framework is composed of a client middleware, which is a library of wrappers replacing the CUDA Runtime (provided by NVIDIA as a dynamic library), and a server middleware, configured as a daemon which runs in those nodes offering GPGPU acceleration services.

In the rest of this section we provide further details on the implementation of our solution. Some complementary details were published in preliminary work [14, 15].

[2] In CUDA terminology, *host* stands for the computer running the regular C/C++-written program, while the *device* is the coprocessor in charge of executing the kernels.

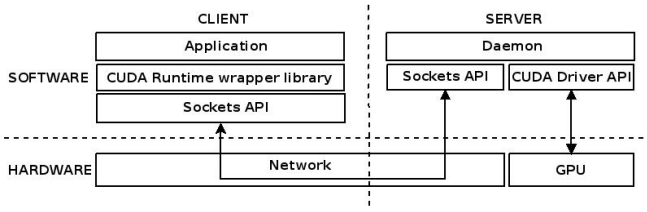


**Figure 1. Outline of the rCUDA Architecture**

### 4.1. Architecture

Our solution is organized as a client-server distributed architecture, depicted in Figure 1. Clients use a library of wrappers to the CUDA Runtime API to access virtualized devices, while nodes hosting the physical accelerators run a daemon servicing API execution requests. On the server, CUDA Runtime API functionalities are implemented employing the low-level Driver API, as the high-level interface lacks of the module management support needed to handle remote kernel loading and execution; moreover, according to official documentation, both APIs are mutually exclusive.

Clients and servers communicate using the sockets API. In order to optimize network data exchange, we developed a customized application-level communication protocol [15].

### 4.2. Client Side

On those nodes requiring remote GPU-based code acceleration capabilities, we install a CUDA Runtime API wrapper library. This library intercepts, processes, and forwards API calls from client to server. Although we introduced the virtualization boundary at the high level CUDA API (because it is the most widely used), we do not anticipate any problem to virtualize the Driver API.

Once the wrapper library is loaded by the system dynamic linking loader, it automatically tries to establish a connection with the server(s) specified in an environment variable. Also, when the library is unloaded, the connection is automatically closed and the library resources are released. Thus, we avoid extending the original CUDA Runtime API with explicit initialization and destruction functions.

For each CUDA call, the following tasks are performed: (1) local checks (function dependent); (2) optionally performs some mappings, e.g., to assign identifiers to pointers or to locally store additional information to be retrieved later; (3) packs the arguments together with a function identifier; (4) sends the execution request to the server; and (5) in synchronous functions, waits for a server response (blocks).

### 4.3. Server Side

The server daemons located on nodes offering acceleration services are in charge of receiving, interpreting, and executing remote API call requests. For each remote execution, a new process is created (using a pre-fork technique for performance purposes) to execute all the requests from a single remote application into an independent GPU context.

Therefore, GPU multiplexing is accomplished by spawning a different server process for each remote execution over a new GPU context. This also ensures the survival of the rest of the servers in the event of a crash of one of them (e.g., caused by an improper CUDA call), as opposed to a multi-threaded solution. Thus, in an HPC cluster environment where the jobs are scheduled and assigned to the different general-purpose cores of the system, all the GPU co-processors can be safely shared by the different jobs, provided that there is sufficient device memory to run all the requested applications concurrently, as the device proprietary driver will manage the concurrent execution of the different active contexts using its own scheduler.

### 4.4. Asynchronous Memory Transfers

Asynchronous memory copy operations posed a major design challenge in our framework. First, CUDA asynchronous memory copies between host and device must involve a *page-locked* (or *pinned*) memory region on the former. Second, these operations can be associated to a *stream*[3].

When a remote 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. Thus, our client middleware issues a memory transfer request and immediately returns the execution control to the caller. Then, depending on the direction of the transfer, it has to subsequently perform the following tasks:

1. **From host to host.** Special case which does not involve the remote device. Concurrently perform a local memory copy if there is no prior device-to-host operation pending to complete on the stream.
2. **From host to device.** Once all previous device-to-host operations on the stream are completed, request a memory transfer to the server and send the data.
3. **From device to host.** Directly request a device to host memory transfer, as the remote driver will care about other pending operations on the stream. Later on, the requested data will asynchronously be sent back.
4. **From device to device.** The same as in the prior case, but do not expect any data to arrive.

Different streams are served in a round-robin fashion, while operations in a stream are executed following a first-come,

[3] A sequence of operations associated to a specific stream execute in order, while different streams execute either out of order or concurrently.

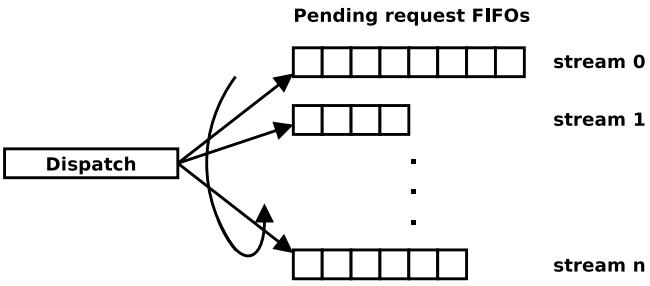


**Figure 2. Operation Scheme for the Client Asynchronous Thread**

first-served policy —see Figure 2—. Thus, the only tasks that remain pending after an asynchronous operation request are the reception of data resulting from device-to-host operations. For this reason, those transfers with a host pointer as the source of the copy have to wait for pending operations to complete. To asynchronously handle these operations, and thus enable the reception of both synchronous and asynchronous data via the same socket, the client middleware employs a dedicated Posix thread, conveniently synchronized with the main thread and the *receiving thread*. Thus, in the event of an asynchronous reception (which has a special identifier), data is concurrently stored into the memory region extracted from its associated destination data.

Similarly, there is a *sending thread* on the server side for multiplexing the socket to allow sending both synchronous and asynchronous data.

## 5. DISCUSSION

In this section we further expose the limitations of our solution, and provide some information on how they can be addressed.

The rCUDA framework targets the Linux OS (for 32- and 64-bit architectures) on both client and server sides. It is currently in its first release stage [4].

Our middleware implements all functions in the CUDA Runtime API (version 2.3), excluding OpenGL and Direct3D interoperability. This should not pose a major inconvenient, as our framework is intended to be used in an HPC environment. However, it may be possible to extend the middleware with the integration of existing 3D remoting acceleration works, such as those described in Section 2.2.

One major drawback of our implementation is the lack of support for the CUDA C extensions. The reason is that the CUDA Runtime library features some hidden and undocumented functions related to these extensions. During the first stage of the compilation process of ".cu" files, the CUDA Front End utility automatically introduces calls to these hidden functions. Note that although it is possible to directly call these functions while still employing our wrappers for the documented API, by using the library preloading functionality of the Linux OS, and therefore overriding the documented functions, as the internals of the CUDA library are not revealed, these hidden functions should not be used. Therefore, in order to avoid the usage of the undocumented functions, our framework does not support the CUDA C extensions, forcing the use of the plain C API instead. As a consequence, our library is not able to locate the GPU code if it is embedded within the executable (which is the default action of the compiler), as the CUDA APIs lack of documented functions to find it. In order to overcome this, CPU and GPU codes have to be kept in separate source files. GPU code is compiled with the NVIDIA compiler driver, using its "device code repositories" feature [17]. On the other hand, CPU code is compiled with a native C/C++ compiler. To deal with this, support from NVIDIA will be required, e.g., opening the full API and thus providing documentation for the currently internal-only functions. In addition, we foresee no problem on porting our solution to a full open framework such as OpenCL [18].

One more limitation of our framework is that virtualized devices do not offer *zero copy* capabilities. Zero copy enables GPUs to directly access host memory. It was introduced in version 2.2 of the CUDA Toolkit, and avoids the need of prior copies to the device self memory. At the moment, our framework is only able to perform memory transfers between host and remote devices upon explicit requests through the corresponding API calls. We do not anticipate future support for this functionality, as it seems to be incompatible with a distributed system. However, this should be no major problem, as programmers are encouraged to check for this functionality prior to using it, because not all GPUs are supposed to support zero copy.

Finally, our future work will focus on implementing load balancing functionalities, so that the client nodes will be able to automatically discover lightly-loaded servers.

## 6. EXPERIMENTS

This section presents a set of experiments that evaluate the usability as well as the performance of our framework. In our experiments, we selected 6 examples from the CUDA SDK: `bandwidthTest` (BT), `binomialOptions` (BO), `dct8x8` (DCT), `simplePitchLinearTexture` (PLT), `simpleStreams` (SS), and `simpleTexture` (ST).

[4] For up-to-date details on the current release, visit the following web pages at UPV: `www.gap.upv.es/~apenya`, or UJI: `www.hpca.uji.es/?q=node/36`. Information about how to obtain a copy of rCUDA can be found there.

**Table 1. Number of Code Lines Modified (or Added) for each CUDA SDK Example**

| Example | Original | Modified | Rate |
|---|---|---|---|
| BT | 877 | 0 | 0.0% |
| BO | 629 | 4 | 0.6% |
| DCT | 3174 | 55 | 1.7% |
| PLT | 274 | 32 | 11.7% |
| SS | 216 | 20 | 9.3% |
| ST | 228 | 22 | 9.6% |

### 6.1. Usability

In order to run the examples in our framework, we first had to extract the host code from the CUDA source files (".cu") and write it into separate C++ files. Next, the pieces of host code using the CUDA C extensions had to be rewritten using the plain C API. For instance, the kernel call:

```
kernel<<<blocks, threads>>>(a, b);
```

could be rewritten as:

```
#define ALIGN_UP(offset, align) (offset) = \
 ((offset) + (align) - 1) & ~((align) - 1)
cudaConfigureCall(blocks, threads);
int offset = 0;
ALIGN_UP(offset, __alignof(a));
cudaSetupArgument(&a, sizeof(a), offset);
offset += sizeof(a);
ALIGN_UP(offset, __alignof(b));
cudaSetupArgument(&b, sizeof(b), offset);
cudaLaunch("kernel");
```

However, the 3 lines of code introduced for each argument setup operation can be replaced by a single line calling the following function:

```
template<class T>
inline void setupArg(T arg, int *off) {
 ALIGN_UP(*off, __alignof(arg));
 cudaSetupArgument(&arg, sizeof(arg), *off);
 *off += sizeof(arg);
}
```

Table 1 summarizes the changes in the SDK examples to make them compliant with our framework (using the `setupArg` function defined above from an included C header file). The numbers in the table illustrate that the changes introduced in the original source code were minor, from 0.0% to 11.7%, thus attaining a high level of usability.

Next, the device source code files (".cu") were combined into one file in order to generate a single GPU module containing all the device code. Then, device files were compiled using the NVIDIA CUDA compiler, while for host files the GNU C++ compiler was used.

Lastly, once the rCUDA server was running, and the client environment variables pointing to the rCUDA server and the rCUDA library were set, all examples were successfully run providing correct results.

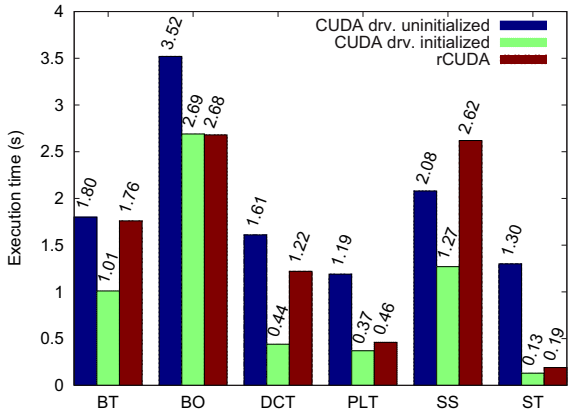


**Figure 3. Performance of rCUDA**

### 6.2. Performance

In this section we analyze the performance of our framework in comparison with the locally-accelerated solution. For performance evaluation focusing on the network interconnect, see [15].

The experiments were performed using two computing nodes equipped with two Quad-Core Intel® Xeon® E5520 processors running at 2.27 GHz and 24 GB of main memory. These nodes were running the Linux OS (kernel 2.6.18). The node offering remote acceleration services was equipped with an NVIDIA Tesla C1060 (driver 190.18) attached to a PCIe 2.0 x16 port. The nodes were connected via a 40 Gbps InfiniBand QDR network, employing a Mellanox MTS3600 switch.

The timing results resulting from our experiments are shown in Figure 3. The examples were executed with no arguments, so the default options were used. The average of 5 executions is presented, and the the maximum standard deviation obtained was 0.09 seconds.

In our experiments, we found that when the OS driver module for the GPU is not being used, the initialization of the CUDA environment is notoriously longer than when it is already in use[5]. In an HPC cluster node, the former case is common because it is not running a graphical interface. Thus, we performed experiments with/without the module driver previously in use. This phenomenon is not noticed when using the rCUDA framework, as the driver is initialized upon the remote daemon start-up.

The figure shows that in 5 out of the 6 examples, the locally accelerated execution having the GPU driver not previously in use was slower than that employing a remote accelerator —up to a 85% in the case of the ST. In the SS example,

[5]The use count of a loaded module can be checked in the `/proc/modules` file or by means of the `lsmod` utility.

the delay introduced by the network and the overhead of the middleware were higher than the driver initialization time.

When the driver was previously initialized, 5 out of 6 examples run in a smaller time (up to 1.35 secs. in SS) when employing the local accelerator than when employing a remote GPU using rCUDA. This is the expected behavior, as explained before. However, the BO example run slightly slower in the local accelerator than when using the remote solution. Detailed time measures of individual CUDA calls revealed that one of the calls to the `cudaThreadSynchronize` function was taking two orders of magnitude longer when using regular CUDA than when employing our solution. This is probably due to the internal algorithm used to determine the finalization of the streams, as it might have some polling period involved which may be different in both implementations.

## 7. POWER SAVINGS

This section illustrates the potential power savings that our proposal brings to a cluster. Although this section is focused on the power consumption of the entire system, readers should note that thanks to the speedup generated by GPUs, the energetic cost per execution might be further reduced.

Although the distributed acceleration architecture we propose obviously increases network traffic in comparison with the GPU-per-node solution, this higher traffic does not lead to a significant variation in the power consumption of the network equipment that interconnects the nodes of the cluster. The reason is that, in current HPC networks, links are always fully active to keep synchronization and thus attain low-latency transmissions. Several proposals from academia have been made in this field trying to lower the voltage and frequency of unused links [19, 20], or even turning them off [21]. Unfortunately, as energy savings always lead to performance loses, these mechanisms have not been incorporated into commercial switches yet, thus reinforcing the fact that our architecture does not increase network power consumption. If these techniques become fully mature in the future, they would probably be included in mainstream switches and, therefore, our architecture should be refined in order to be compatible with the afore-mentioned power saving mechanisms. We are currently devising approaches to achieve such compatibility, like channeling traffic from several clients into a small set of network links.

To determine the total power rating of a cluster, we have to take into account the rating of its nodes, as well as that of the network equipment. In order to compare cluster configurations with different number of accelerators, we separate the power rating of GPUs from node consumption.

For instance, consider a 100-node cluster equipped with NVIDIA Tesla C1060 GPUs, and the nodes connected via an InfiniBand network with 4 Mellanox MTS3600 switches configured in a star topology. According to the accelerator specifications [22], the total board power (including the cooling solution) is 193.58 W. The power rating of the MTS3600 is 316 W [23]. We will assume a 600 W power rating per node.

**Table 2. Sample of Power Savings on a Cluster**

| GPUs | Consumption | Savings | Rate |
|---|---|---|---|
| 100 | 80622.0 W | 0.0 W | 0.0% |
| 50 | 70943.0 W | 9679.0 W | 12.0% |
| 25 | 66103.5 W | 14518.5 W | 18.0% |
| 10 | 63199.8 W | 17422.2 W | 21.6% |

Table 2 summarizes the power savings for a range of accelerators installed in the cluster. As shown in the table, in this example, when the number of GPUs is decreased by 90%, the total power rating of the cluster is reduced by more than a 20%. Moreover, this also leads to savings in acquisition, maintenance, space, and cooling. On the other hand, the actual achievable reduction in the number of GPUs strongly depends on the particular applications running on the cluster, as well as on its hardware and network configuration. Thus, from the study of the characteristics of the environment of application, it is possible to reduce the number of GPUs while causing a negligible performance reduction.

## 8. CONCLUSIONS

Current GPUs feature a significant energy consumption. This becomes specially relevant in HPC clusters with a large number of nodes, where equipping each of them with GPU-accelerating capabilities could be prohibitive from the energy consumption perspective.

In this paper we have detailed a framework to enable remote GPU-based code acceleration, thus permitting the reduction on the number of accelerators in a cluster, and consequently its global energy consumption. Although our proposal has a few drawbacks mostly caused by the fact that the NVIDIA CUDA Runtime API is not fully open, those are easily addressed avoiding the use of the CUDA C extensions.

We have also discussed the usability of our solution on the basis of a set of CUDA SDK examples, and provided a brief performance analysis. Finally, we have covered the power saving potential of the use of remote GPUs. We showed that remote acceleration can lead to considerable savings at the expense of slightly increasing execution time.

## ACKNOWLEDGEMENTS

The researchers at UPV were supported by PROMETEO from Generalitat Valenciana (GVA) under Grant PROM-

ETEO/2008/060. The researchers at UJI were supported by the Spanish Ministry of Science and FEDER (contract no. TIN2008-06570-C04-01), and by the Fundación Caixa-Castelló/Bancaixa (contract no. P1-1B2009-35).

## REFERENCES

[1] A. Vasan, A. Sivasubramaniam, V. Shimpi, T. Sivabalan, and R. Subbiah, "Worth their watts? An empirical study of datacenter servers," The 16th IEEE International Symposium on High-Performance Computer Architecture (HPCA-16), Bangalore, India, Jan. 2010.

[2] M. Dolz, J. C. Fernández, R. Mayo, and E. S. Quintana, "EnergySaving Cluster Roll: Power saving system for clusters," Architecture of Computing Systems (ARCS 2010), Hannover, Germany, 2010.

[3] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the art of virtualization," The nineteenth ACM symposium on Operating systems principles, New York, NY, USA: ACM, 2003, pp. 164–177.

[4] L. A. Abbas-Turki, S. Vialle, B. Lapeyre, and P. Mercier, "High dimensional pricing of exotic European contracts on a GPU cluster, and comparison to a CPU cluster," International Symposium on Parallel Distributed Processing, IEEE, 2009, pp. 1–8.

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

[6] D. Blythe, "The Direct3D 10 system," *ACM Trans. Graph*, vol. 25, no. 3, pp. 724–734, 2006.

[7] D. Shreiner and OpenGL, OPENGL PROGRAMMING GUIDE, $7^{th}$ ed., Addison-Wesley Professional, Aug. 2009.

[8] G. Humphreys, M. Houston, R. Ng, R. Frank, S. Ahern, P. Kirchner, and J. Klosowski, "Chromium: A stream-processing framework for interactive rendering on clusters," SIGGRAPH 2002 Conference, ser. Annual Conference Series, J. Hughes, Ed. ACM Press, 2002, pp. 693–702.

[9] J. G. Hansen, "Blink: Advanced display multiplexing for virtualized applications," Feb. 07 2008, Technical Report, Available: http://www.diku.dk/~jacobg/pubs/blink_nossdav.pdf

[10] H. A. Lagar-Cavilla, N. Tolia, M. Satyanarayanan, and E. de Lara, "VMM-independent graphics acceleration," The 3rd international conference on Virtual execution environments, New York, NY, ACM, 2007, pp. 33–43.

[11] D. Abramson, J. Jackson, S. Muthrasanallur, G. Neiger, G. Regnier, R. Sankaran, I. Schoinas, R. Uhlig, B. Vembu, and J. Weigert, "Intel Virtualization Technology for directed I/O," *Intel Technology Journal*, vol. 10, no. 3, pp. 179–192, Aug. 2006.

[12] L. Shi, H. Chen, and J. Sun, "vCUDA: GPU accelerated high performance computing in virtual machines," IEEE International Symposium on Parallel & Distributed Processing (IPDPS'09), 2009, pp. 1–11.

[13] NVIDIA, NVIDIA CUDA Programming Guide Version 1.1, NVIDIA, 2007.

[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," EURO-PAR 2009 WORKSHOPS, ser. LNCS, vol. 6043. Springer-Verlag, 2010, pp. 385–394.

[15] J. Duato, A. J. Peña, F. Silla, R. Mayo, and E. S. Quintana-Ortí, "Modeling the CUDA remoting virtualization behaviour in high performance networks," First Workshop on Language, Compiler, and Architecture Support for GPGPU, Jan. 2010, Available: http://www.cse.iitk.ac.in/users/lca-gpgpu-I/duato.pdf

[16] NVIDIA, *NVIDIA CUDA Programming Guide Version 2.3*, NVIDIA, 2009.

[17] NVIDIA, *The CUDA Compiler Driver NVCC*, NVIDIA, 2009.

[18] A. Munshi, Ed., *OpenCL 1.0 Specification*, Khronos OpenCL Working Group, 2009.

[19] L. Shang, L.-S. Peh, and N. K. Jha, "Dynamic voltage scaling with links for power optimization of interconnection networks," The Ninth International Symposium on High-Performance Computer Architecture, Anaheim, CA, IEEE, Feb. 2003, pp. 91–102.

[20] J. Kim and M. Horowitz, "Adaptive supply serial links with sub-1-V operation and per-pin clock recovery," *IEEE Journal of Solid-State Circuits*, vol. 37, no. 11, pp. 1403–1413, Nov. 2002.

[21] V. Soteriou and L.-S. Peh, "Dynamic power management for power optimization of interconnection networks using on/off links," 11th Symposium on High Performance Interconnects, 2003, pp. 15–20.

[22] NVIDIA, *Tesla C1060 Computing Processor Board - Board Specification*, Jan. 2009, Available: http://www.nvidia.es/content/PDF/Tesla_product_literature/Tesla_C1060_boardSpec_v03.pdf

[23] Mellanox, *MTS3600 36-port 20 and 40 Gb/s InfiniBand Switch System*, 2009, Available: http://www.mellanox.com/related-docs/prod_ib_switch_systems/PB_MTS3600.pdf