Improving the User Experience of the rCUDA Remote GPU Virtualization Framework

Carlos Reaño[1]*, Federico Silla[1], Adrián Castelló[2], Antonio J. Peña[3], Rafael Mayo[2], Enrique S. Quintana-Ortí[2], and José Duato[1]

[1]*DISCA, Universitat Politècnica de València, 46022 València, Spain*
[2]*ICC, Universitat Jaume I, 12071 Castellón de la Plana, Spain*
[3]*MCS, Argonne National Laboratory, Argonne, IL 60439, USA*

SUMMARY

Graphics processing units (GPUs) are being increasingly embraced by the high-performance computing community as an effective way to reduce execution time by accelerating parts of their applications. rCUDA was recently introduced as a software solution to address the high acquisition costs and energy consumption of GPUs that constrain further adoption of this technology. Specifically, rCUDA is a middleware that allows a reduced number of GPUs to be transparently shared among the nodes in a cluster. While the initial prototype versions of rCUDA demonstrated its functionality, they also revealed concerns with respect to usability, performance, and support for new CUDA features. In response, in this paper we present a new rCUDA version that (1) improves usability by including a new component that allows an automatic transformation of any CUDA source code so that it conforms to the needs of the rCUDA framework, (2) consistently features low overhead when using remote GPUs thanks to an improved new communication architecture, and (3) supports multithreaded applications and CUDA libraries. As a result, for any CUDA-compatible program, rCUDA now allows the use of remote GPUs within a cluster with low overhead, so that a single application running in one node can use all GPUs available across the cluster, thereby extending the single-node capability of CUDA.

KEY WORDS: GPGPU; CUDA; Virtualization; HPC; Clusters

1. INTRODUCTION

The high computational cost of many complex scientific and engineering applications is leading researchers to adopt graphics processing units (GPUs) to reduce the execution time of their experiments in areas as diverse as medical imaging, bioinformatics, computational fluid dynamics, seismic exploration, and computational finance [14].

High-performance computing (HPC) facilities equipped with GPUs in general include one or more accelerators per cluster node. Although this configuration is appealing from the perspective of raw performance, it is not energy-efficient; even in an HPC cluster, all the GPUs are unlikely to be used 100% of the time, since few applications feature such an extreme degree of data parallelism.

*Correspondence to: Carlos Reaño, Universitat Politècnica de València, DISCA Edificio 1G, Camino de Vera s/n, 46022 València, Spain. Tel.: +34-96-3877007 Ext.75738. Fax: +34-96-3877579. E-mail: carregon@gap.upv.es

Contract/grant sponsor: Generalitat Valenciana under the PROMETEO program phase II.; contract/grant number: PROMETEOII/2013/009

Reducing the number of accelerators in a cluster, with the aim of increasing their utilization (fewer accelerators for the same workload), results in a less costly and more appealing solution, diminishing the contribution of the electricity bill to the operational cost (APEX), the capital cost (COPEX), and the environmental impact of HPC through a lower energy footprint. However, a configuration where only a limited number of the nodes in the cluster have a GPU introduces some difficulties, since it requires a global scheduler to map (distribute) jobs to GPU nodes according to their acceleration needs, thus making this new and more power-efficient configuration more difficult to manage efficiently. Moreover, this configuration will not really improve the GPU utilization rate unless the global scheduler can share GPUs among several applications, a detail that noticeably increases the complexity of the scheduler.

A better solution to deal with a cluster configuration featuring fewer GPUs than nodes is virtualization [5]. With GPU virtualization, GPUs are installed only in some of the nodes, which act as acceleration servers granting GPGPU (General-Purpose Computing on GPUs) services to the rest of the cluster. Furthermore, the scheduling difficulties mentioned above are avoided, since tasks can now be dispatched to any CPU node, thus increasing overall GPU utilization because they are shared among the rest of the cluster. This approach can be evolved by enhancing the schedulers so that GPU servers are put into low-power sleeping modes as long as their acceleration features are not required, thus further increasing energy efficiency. Moreover, instead of attaching a GPU to a large number of acceleration servers, GPUs can be consolidated in a reduced number of dedicated servers that can exhibit different numbers of accelerators, so that a certain level of granularity is provided to the scheduling algorithms in order to better adjust the powered resources to the workload present at any moment in the system. GPU consolidation would, however, require a careful analysis of network bandwidth in order to avoid congestion.

As a simple example showing the benefits of reducing the number of GPUs installed in the cluster and sharing the remaining ones, let us consider the case for a 64-node cluster with an NVIDIA GeForce GTX480 Fermi GPU attached to each node. According to our measurements, the power dissipated by this GPU may rise to 62.1 W when idle. Taking into account this power consumption, if the number of idle GPUs in the cluster is, on average, 50%, then up to 1,987.2 W are wasted. If the percentage of idle GPUs, on average, increases to 75%, the power consumed for doing nothing rises to 2,980.8 W. These numbers, although rough, provide an initial estimation of the reduction in power consumption that can be achieved by leveraging the remote GPU virtualization technique.

The rCUDA (remote CUDA) remote GPU virtualization framework [2, 3, 4] grants access to GPUs installed in remote nodes to CUDA-based applications. In this manner, applications remain unaware that they deal with virtualized GPUs instead of real ones. Previous work [2, 3, 4] focused on demonstrating that the exploitation of remote CUDA devices is feasible but revealed three major drawbacks:

- The usability of the rCUDA framework was limited by its lack of support for the CUDA C extensions. As shown in Section 4, the reason is the use of the CUDA runtime library, which includes several hidden and undocumented functions within these extensions. Therefore, in order to avoid the use of these undocumented functions, our framework offered support only for the plain CUDA C API, thus making it necessary to rewrite those lines of the application source files that employ the CUDA C extensions. For applications comprising large amounts of source code, performing this process manually was cumbersome.
- The use of remote GPUs in rCUDA resulted in lower performance compared with the regular local use made by CUDA [2, 3]. In our solution, the bandwidth available between the main memory of the node demanding GPU services and the remote GPU memory was constrained by that of the network connecting client and server (network bandwidth is usually lower than that of the PCI-Express bus connecting the GPU and the network interface in the server).
- Our rCUDA virtualization solution had to evolve to support the new versions of CUDA being periodically released. In this regard, the work presented in [2, 3, 4] supported the now-obsolete CUDA 2 and 3 versions. After those initial versions of rCUDA, NVIDIA introduced support for multithreaded applications in CUDA 4 and, more recently, a new way to internally manage CUDA libraries in CUDA 5, resulting in significant changes with respect to prior versions.

Because of these three drawbacks, the user's experience with the rCUDA remote GPU virtualization framework was far from ideal, where users can seamlessly access remote GPUs without noticing any significant performance loss or having to adapt their application codes to the requirements of the remote GPU virtualization framework. Thus, when using the rCUDA version presented in [2, 3, 4] (rCUDA v3), the user's experience was relatively unsatisfactory. On the one hand, given the reduced communication performance of the previous rCUDA versions, users experienced large increments of application execution time when using remote GPUs. On the other hand, rCUDA users suffered from a hard limitation about which CUDA programs they were able to remotely execute with rCUDA, given that the CUDA C extensions were not supported by the rCUDA framework and that new features introduced in version 4 and 5 of CUDA could not be leveraged. The result was that rCUDA users had to manually modify the source code of their programs, which was hardly acceptable.

In this paper we have enriched rCUDA (new rCUDA v4) by addressing the three concerns above, thus narrowing the distance between the actual user's experience and the ideal case. In this way, rCUDA has evolved from being just a proof of concept to becoming a mature technology that allows users to remotely execute their CUDA programs without having to manually modify them and also without experiencing a performance reduction. Thus, rCUDA has become a technology ready for production scenarios. In this evolution, we have improved rCUDA with the following additions:

- A complementary tool, `CU2rCU`, to automatically analyze and modify the application source code in order to find which parts, containing CUDA C extensions, must be modified and adapted to the requirements of rCUDA. This tool automatically performs the required changes, without the manual intervention of a programmer. Moreover, the `CU2rCU` tool has been integrated into the compilation flow, so that rCUDA users can effectively replace the call to NVIDIA's `nvcc` compiler with the `CU2rCU` command, which will internally make use of the required backend compilers after analyzing and adapting the source code files.
- An improved general communication architecture that can be tuned to a variety of network technologies. This new and modular communication architecture makes use of pipelined transfers in order to noticeably improve performance. In this paper we introduce the case for the InfiniBand communication module of rCUDA. Thanks to this new communication architecture, along with the use of high-performance interconnects such as InfiniBand, users do not suffer from the performance penalty present in previous versions of rCUDA.
- Support for multithreaded applications and for all CUDA libraries, thus allowing users to leverage all the new features included in this CUDA version. Additionally, the new version of rCUDA is now able to offer a single application, being executed in a single node, access to GPUs in many different nodes of the cluster (multinode configuration). This is an important improvement over CUDA, where the number of GPUs provided to an application running in a given node is limited to the GPUs that can be attached to that single node (usually 4 and never more than 8). In the new version, rCUDA enables an application to directly access all the GPUs in the cluster, thus boosting its performance. Therefore, the only limit is imposed by the characteristics of the applications and the ability of the programmer to extract parallelism.

In summary, this paper presents a CUDA 5-compatible (supporting all CUDA libraries) GPGPU virtualization solution that, in addition to promoting green computing, provides applications access to a virtually unlimited number of GPUs, thus making the use of GPU accelerators in the HPC context even more beneficial. Nevertheless, because of space constraints, we do not present an energy study in this paper, which is left for future publications.

The rest of the paper is organized as follows. In Section 2 we review previous work related to remote GPU virtualizattion. In Section 3 we introduce the rCUDA technology. The next three sections present how we have addressed the three concerns mentioned above. The `CU2rCU` tool is described and analyzed in Section 4; the new modular communication architecture is introduced in Section 5; and the evolution of rCUDA to support multithreaded applications and CUDA libraries is presented in Section 6. In Section 7 we briefly revisit the view from the rCUDA user. In Section 8 we summarize our conclusions and discuss future work.

Table I. Comparison of existing CUDA-based virtualization solutions

Feature (support)	Virtualization Solutions						
	vCUDA	GViM	gVirtuS	VGPU	GridCuda	rCUDA v3	rCUDA v4
CUDA version	3.2	1.1	2.3	4.0	3.2	3.2	5.0
Full CUDA API	no	no	no	N/A	N/A	yes	yes
Last activity year	2012	2009	2010	2013	2011	2012	2013
Multi-GPU	N/A	N/A	N/A	N/A	yes	yes	yes
Multi-thread	N/A	N/A	N/A	N/A	yes	no	yes
TCP/IP	yes	no	yes	yes	yes	yes	yes
HPC networks	no	no	no	yes	no	no	yes

2. RELATED WORK

GPU virtualization has been addressed in the past by using both hardware and software approaches. For brevity, we focus the following discussion on the use of GPUs in a GPGPU context, avoiding hardware approches as well as those solutions intended for graphics acceleration, which cannot be leveraged because they address completely different concerns.

vCUDA [19], GViM [8], gVirtuS [7], VGPU [20], and GridCUDA [9] pursue the virtualization of the CUDA Runtime API for GPGPU. In the case of OpenCL, the VCL [1], the VOCL [27], and the SnuCL [23] frameworks provide similar features. All these solutions present a similar middleware architecture composed of two parts: the front end, installed in the system requesting acceleration services, which becomes the interface to applications; and the back end, installed in the system with the accelerator, thus having direct access to it.

Unfortunately, the CUDA-based solutions mentioned above exhibit different drawbacks. The vCUDA technology supports only an old CUDA version (v3.2) and implements an unspecified subset of the CUDA runtime. Moreover, its communication protocol presents a considerable overhead, because of the costs of the encoding and decoding stages, that cause a noticeable drop in overall performance. GViM is based on the old CUDA version 1.1 and, in principle, does not implement the entire runtime API. Furthermore, GViM is designed to be used in virtual machines so that applications being executed in such virtual scenarios can access GPUs located in the real host; GViM does not support remote GPU virtualization. The gVirtuS approach is based on the old CUDA version 2.3 and implements only a small portion of the runtime API. For example, in the case of the memory management module, it implements only 17 out of 37 functions. Nevertheless, though it is intended mainly to be used in virtual machines, granting them access to the real GPU located in the same node, it does provide TCP/IP communications for remote GPU virtualization, thus allowing access to GPUs located in other nodes.

VGPU is a recent tool that in principle provides similar features to the initial public open-source versions of rCUDA. Unfortunately, the information provided by the VGPU authors is fuzzy, and so far there is no publicly available version that can be used for testing and comparison. GridCuda also provides access to remote GPUs in a cluster, as rCUDA does. Although the authors of GridCuda mention how their proposal overcomes some of the limitations of the early versions of rCUDA, they later detail only capabilities similar to those included in the initial public open-source releases of rCUDA, and do not provide any insight about the supposedly enhanced features. There is currently no publicly available version of GridCuda that can be used for testing.

In summary, all the cited CUDA-based virtualization solutions present the same fundamental characteristics, with the main difference among them being the number of supported CUDA functions that each framework provides to users. As mentioned in Section 1, the ideal remote GPU virtualization solution should provide the same performance as the local usage of GPUs, in addition to not requiring any modification to the regular user workflow, including any modifications of the application source code. In Table I the features of the CUDA-based virtualization solutions

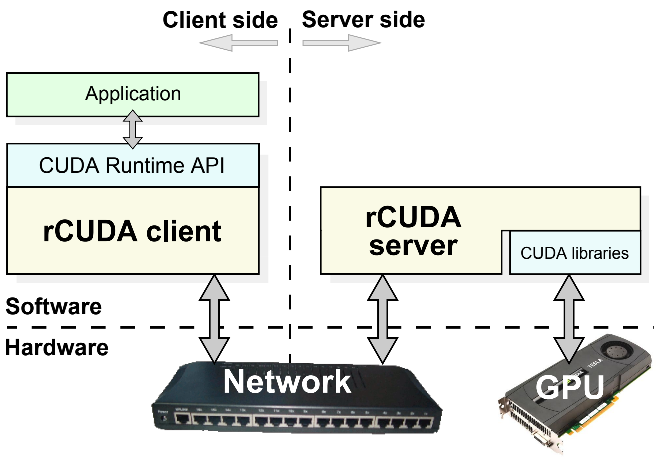


Figure 1. Overview of the rCUDA client-server architecture.

enumerated above are compared from an ideal point of view. The comparison between rCUDA v3 and the new rCUDA v4 has also been included for completeness. In the rest of the paper we describe the new features of rCUDA.

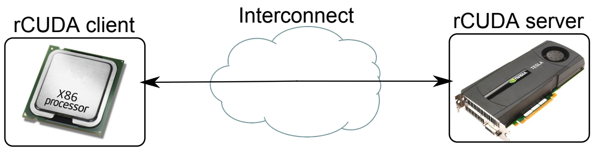
3. REVIEW OF rCUDA

The rCUDA framework grants applications transparent access to GPUs installed on remote nodes, so that they are not aware of the use of an external device. This framework is organized following a client-server architecture; see Figure 1. The client middleware is used by the application demanding GPGPU services and presents to the application the same interface as the regular NVIDIA CUDA Runtime API does. Upon receiving a request from the application, the client middleware processes it and forwards the corresponding requests to the rCUDA server middleware, running on a remote node. The server interprets the requests and performs the required processing by instructing the real GPU to execute the corresponding request. Once the GPU has completed the execution of the requested command, the results are gathered by the rCUDA server, which sends them back to the client middleware. There, the output is forwarded to the demanding application.

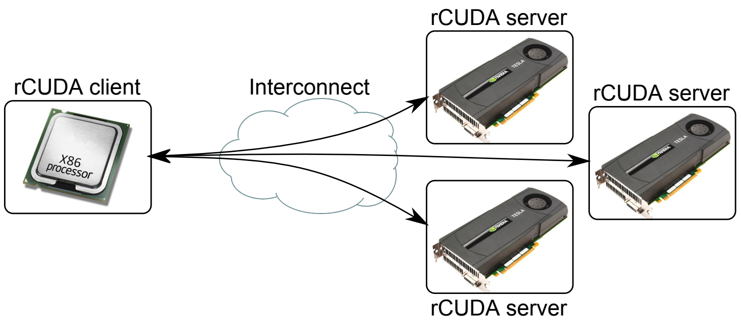
The rCUDA framework does not modify the behavior of CUDA programs. If the programmer writes a code with multiple invocations to the same kernel, all of them reusing the same data, and the CUDA program copies data back-and-forth between GPU and CPU memory for every invocation, then rCUDA will forward the server the same `cudaMemcpy` calls, offering the same behavior. On the contrary, if the programmer writes a more efficient program where data is copied to a GPU only once before the first kernel invocation and, after executing the last kernel, the data is copied back to CPU memory, then rCUDA will forward the same functions to the remote GPU, replicating the behavior of the CUDA program. This approach is also true with GPUs assigned to a given program: in the same way that a given GPU is not preempted over the lifetime of an application with regular CUDA, with rCUDA GPUs remain assigned to the same application until it ends.

The communication between rCUDA clients and remote GPU servers is carried out via a customized application-level protocol tailored for the underlying network; see [2, 3, 4].

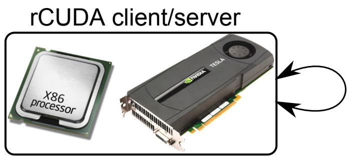
In general, the performance offered by rCUDA is lower than that of the original CUDA, since with rCUDA the GPU is farther away from the invoking application than it is with CUDA, thus introducing some overhead. With the version of rCUDA presented in this paper, however, this penalty is very low for most applications. Moreover, the performance of applications using the new version of rCUDA is still noticeably higher than that provided by computations on regular CPUs. Taking into account the flexibility provided by this new version of rCUDA, in addition to the reduction in energy and acquisition costs it enables, the benefits of rCUDA outweigh the overhead



(a) Typical configuration with one client and one server.



(b) Configuration where one rCUDA client makes use of three servers.



(c) A single computer acting as both client and server. The local GPU is accessed through the rCUDA middleware.

Figure 2. Examples of possible rCUDA client-server combinations.

it introduces, as will be shown later in the paper. Figure 2 depicts different possibilities that the use of rCUDA provides, showing the flexibility if introduces.

4. CU2rCU: A CUDA-TO-rCUDA CONVERTER

This section motivates the need for a CUDA-to-rCUDA source-to-source converter; presents `CU2rCU`, a tool developed for that purpose; and describes the experiments carried out to evaluate the tool.

4.1. Need for a CUDA-to-rCUDA Converter

A CUDA program can be viewed as a regular C program where some of its functions have to be executed by the GPU (also referred to as *device*) instead of the traditional CPU (also known as *host*). Programmers usually control the CPU-GPU interaction via the CUDA Runtime API for GPGPU programming. This API includes CUDA extensions to the C language, which are constructs that follow a specific syntax designed to make CUDA programming more accessible, usually leading to fewer lines of source code than its plain C equivalent (though both codes tend to look similar). The code in Listing 1 shows an example of a “hello world” program in CUDA. In this example, the functions *cudaMalloc* (line 13), *cudaMemcpy* (lines 15 and 19), and *cudaFree* (line 21) belong to the plain C API of CUDA, whereas the kernel launch sentence in line 17 uses the syntax provided by the CUDA extensions.

CUDA programs are compiled with the NVIDIA `nvcc` compiler [13], which detects fragments of GPU code within the program and compiles them separately from the CPU code. During this compilation process, references to structures and functions not made public in the CUDA documentation are automatically inserted into the CPU code. These undocumented functions impair

```
 1 #include <cuda.h>
 2 #include <stdio.h>

 4 // Device code
 5 __global__ void helloWorld(char* str) {
 6   // GPU tasks.
 7 }

 9 // Host code
10 int main(int argc, char **argv) {
11   const char h_str[] = "Hello World!";
12   // ...
13   cudaMalloc((void**)&d_str, size);
14   // copy the string to the device
15   cudaMemcpy(d_str, h_str, size, cudaMemcpyHostToDevice);
16   // launch the kernel
17   helloWorld<<< BLOCKS, THREADS >>>(d_str);
18   // retrieve the results from the device
19   cudaMemcpy(h_str, d_str, size, cudaMemcpyDeviceToHost);
20   // ...
21   cudaFree(d_str);
22   printf("%s\n", h_str);
23   return 0;
24 }
```

Listing 1: "Hello world" program

the creation of tools that need to replace the original CUDA Runtime Library from NVIDIA. To overcome this problem, we have decided not to support these undocumented functions in rCUDA; instead, we offer a compile-time work-around that avoids their use. Notice that doing so requires bypassing `nvcc` for CPU code generation, since this compiler automatically inserts references to them into this code. Therefore, the CPU code in a CUDA program should be directly managed by a regular C compiler (e.g., GNU `gcc`). On the other hand, since a plain C compiler cannot deal with the CUDA extensions to C, they should be *unextended* back into plain C. Since manually performing these changes for large programs is a tedious, sometimes error-prone task, we have developed an automatic tool that modifies a CUDA source code with CUDA extensions and transforms it into its plain C equivalent. This automatic tool is a major contributor to improving the experience of those using the rCUDA framework. With this new tool, when a given CUDA source code is to be compiled for execution within the rCUDA framework, it is split into two parts:

- Host code: processed with a backend compiler such as GNU `gcc` (for either C or C++ languages), after being unextended, and executed on the host
- Device code: compiled with the `nvcc` compiler and executed on the device.

Revisiting the previous "hello world" CUDA example, the code snippet in Listing 2 shows the transformation into plain C of the kernel call in line 17 employing the extended syntax.

To separately generate CPU and GPU code, we leverage a feature of `nvcc` that allows one to extract and compile only the device code from a CUDA program and generate a binary file containing only the GPU code. For the host code, once the CUDA extensions to C have been transformed into code using only the plain C CUDA API, we generate the corresponding binary file with a backend C compiler. Notice that prior to using a regular C compiler, the GPU code should additionally be removed from the program code, since regular C compilers cannot cope with such code. The separation and transformation process is illustrated in Figure 3.

Notice that the code transformations we propose, along with the specific usage of the compilers mentioned above, modify the compilation flow. In the original CUDA compilation flow, the input program was separated into host code and device code. During that process, the device code was

```
#define ALIGN_UP(offset, align) (offset) = \
((offset) + (align) - 1) & ~((align) - 1)

int main() {
  // ...
  // The following code lines replace line 17 of the previous
  // piece of code, where kernel "helloWorld" was launched:
  cudaConfigureCall(BLOCKS, THREADS);
  int offset = 0;
  ALIGN_UP(offset, __alignof(d_str));
  cudaSetupArgument(&d_str, sizeof(d_str), offset);
  cudaLaunch("helloWorld");
  // ...
}
```

Listing 2: Unextending line 17 of “Hello world” program

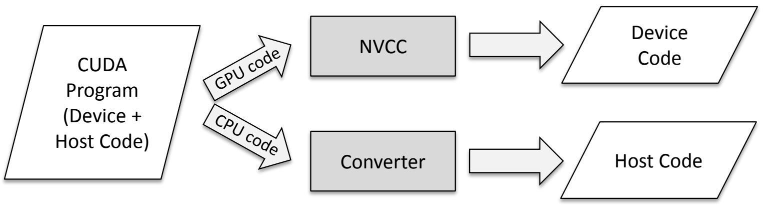


Figure 3. CUDA-to-rCUDA conversion process.

transformed into binary code and embedded into the previously separated host code. The host code with the binary device code embedded was then compiled with `gcc`, generating an executable that had the binary device code embedded. On the contrary, in the rCUDA compilation flow, the `CU2rCU` converter is initially applied to the input program in order to obtain its equivalent source code using only plain C and without device code. From this converted code, the tool produces an executable that has references to device code stored in an external repository generated with `nvcc` by using the appropriate option.

4.2. Implementing the `CU2rCU` *Converter*

A source-to-source transformation framework was leveraged in order to implement the automatic tool that transforms source code employing CUDA extensions into plain C code. Different options for this class of source transformations are available, from simple pattern string replacement tools to frameworks that parse the source code into an abstract syntax tree (AST) and transform the code using that information. Since our tool needs to do complex transformations involving semantic C++ code information, we have selected the latter approach.

Several frameworks are available to tackling complex source transformations, for example ROSE [16], GCC [6], and Clang [10]. We have chosen Clang because it is widely used and explicitly supports programs written in CUDA. Moreover, some converters of CUDA source code exist that are also based on Clang, such as `CU2CL` [12], which transforms code from CUDA to OpenCL.

Clang, one of the primary subprojects of LLVM [11], is a C language family compiler that aims at providing a platform for building source code level tools, including source-to-source transformation frameworks.

Figure 4 shows how the developed converter interacts with Clang. The inputs to the converter are CUDA source files containing device and host code with CUDA extensions, as explained in the previous subsection. The Clang driver (a compiler driver providing access to the Clang compiler and tools) parses those files generating an AST. The Clang plugin that we have developed, `CU2rCU`, then uses the information provided by the AST and the libraries contained in the Clang framework

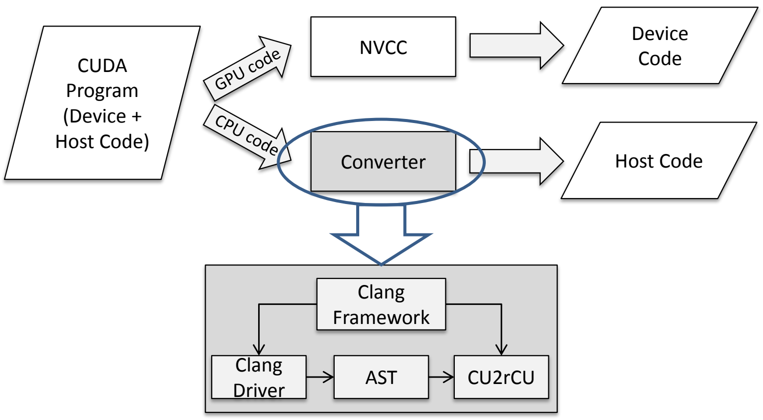


Figure 4. CUDA-to-rCUDA converter detailed view.

to perform the needed transformations, generating new source files that contain only host code with the plain C syntax. During the conversion `CU2rCU` automatically analyzes user source files included by the input files, converting them when necessary. The most important transforms carried out by our `CU2rCU` tool are described in [17].

Finally, it is worth to mention that we have validated the correctness of the implemented converter on several CUDA programs, as explained in next section, confirming that the functionality of the original programs has been fully preserved.

4.3. Evaluation of the `CU2rCU` *converter*

To test the new `CU2rCU` tool, we have used sample codes from the NVIDIA GPU Computing SDK [15] and the production code of the LAMMPS Molecular Dynamics Simulator [18].

Our first experiments dealt with a representative set of examples from the NVIDIA GPU Computing SDK. Table II shows the time[†] required for their conversion in seconds. In the experiments we employed a desktop platform equipped with an Intel(R) Core(TM) 2 DUO E6750 processor (2.66 GHz, 2 GB RAM) and a GeForce GTX 590 GPU, running Linux OS (Ubuntu 10.04). We used `nvcc` version 5, as well as Clang version 3.0. Table II also reports the number of lines of the original application and the modified sources obtained by our tool. The total time required for the automatic conversion of all these examples, 10.26 seconds, compared with the time spent on a manual conversion by an expert from the rCUDA team, 30.5 hours, clearly shows the benefits of using the converter. We note that the tasks of manually converting the code, on the one hand, and those for creating the converter, on the other hand, were fully disconnected among them, being carried out by different people, in order to avoid a bias in the comparison. We note also that an automatic source code conversion leads to a slightly larger amount of modified lines (though some of them correspond to sentences split into two lines). Nevertheless, the code automatically generated is similar to the one obtained from a manual conversion.

In addition to testing the converter with NVIDIA SDK codes, we have evaluated it on a real-world production code: the LAMMPS molecular dynamics simulator. This is a classic molecular dynamics code that can be used to model the interactions among atoms or, more generically, as a parallel particle simulator at the atomic, meso, or continuum scale. The entire application comprises more than 300,000 lines of code distributed over 30 packages. Some of those packages are written for CUDA, such as *GPU* or *USER-CUDA*, which are mutually exclusive. We have evaluated our tool against the USER-CUDA package, with over 14,000 lines of code. The bottom part of Table II

[†]Conversion and compilation time shown in this section have been gathered in an iterative way so that a given compilation (or conversion) has been repeated until the standard deviation of the measured time was lower than 5%.

Table II. CU2rCU Conversion Statistics

CUDA SDK Sample		Time (s)	Lines of Code		
			CUDA	Modified/Added	
Name	Acronym		Code	Num.	%
alignedTypes	AT	0.259	186	32	17.20
asyncAPI	AP	0.191	78	6	7.69
bandwidthTest	BT	0.407	708	0	0.00
BlackScholes	BS	0.364	281	13	4.63
clock	CL	0.196	75	8	10.67
concurrentKernels	CK	0.196	100	11	11.00
convolutionSeparable	CS	0.591	319	18	5.64
cppIntegration	CI	0.685	129	12	9.30
dwtHaar1D	DH	0.221	266	11	4.14
fastWalshTransform	FW	0.360	241	20	8.30
FDTD3d	FD	1.082	860	13	1.52
matrixMul	MM	0.394	272	34	12.50
mergeSort	MS	0.917	1124	105	9.34
scalarProd	SP	0.358	138	10	7.25
scan	SC	0.548	359	26	7.24
simpleAtomicIntrinsics	SA	0.367	211	6	2.84
simpleMultiCopy	SM	0.202	211	22	10.43
simpleTemplates	ST	0.211	241	13	5.39
simpleVoteIntrinsics	SV	0.196	222	19	8.56
SobolQRNG	SQ	1.278	10586	8	0.08
sortingNetworks	SN	0.761	571	70	12.26
template	TE	0.357	97	7	7.22
vectorAdd	VA	0.192	88	8	9.09
LAMMPS Molecular Dynamics Simulator					
Package USER-CUDA	LA	6.910	14742	1409	9.56

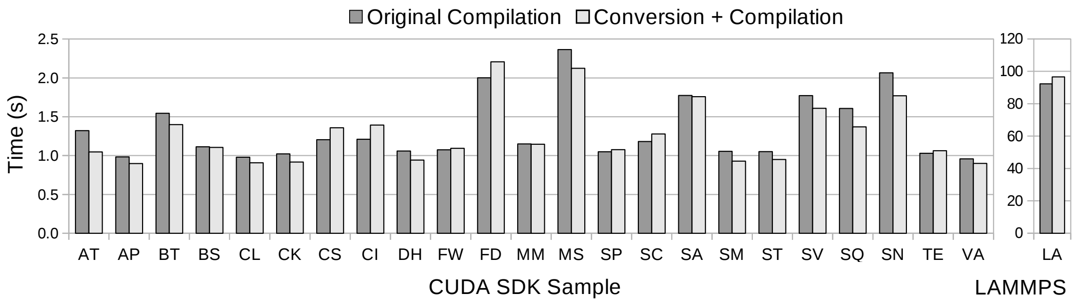


Figure 5. `nvcc` compilation time compared with `CU2rCU` conversion plus compilation time for CUDA SDK samples and LAMMPS.

shows the results of the conversion. The time spent by a CUDA expert to adapt the original code was two weeks with full-time dedication.

Table III. Comparison of CUDA and rCUDA Compilation Phases

	Times Each Phase is Executed	
Compilation Phase	CUDA	rCUDA
cu2rcu	**0**	**1**
gcc	**6**	**5**
cudafe	2	2
cudafe++	1	1
filehash	**1**	**0**
nvopencc	1	1
ptxas	1	1
fatbin	1	1

Moreover, we have compared the time spent in the compilation of the original CUDA source code of the SDK samples and LAMMPS with the period spent in their conversion and subsequent compilation of the converted code by our tool. The results, shown in Figure 5, demonstrate that the time of converting the original code and later compiling it is similar to the compilation time of the original sources. To explain why both times are similar, in Table III we present a comparison of the phases in the CUDA and rCUDA compilation flows, in terms of how many times each phase of `nvcc` is invoked[‡]. The `CU2rCU` compilation phase can be regarded as a `gcc` one because we are really calling Clang, which is also a C compiler. Therefore, we could state that, in both flows, the `gcc` compilation phase is actually executed six times. In the `CU2rCU` compilation phase we must also take into account that, apart from compilation time, we are doing source transforms, which also take time. In the rCUDA compilation flow, this time is compensated for, since (1) it does not require the `filehash` phase [13], present only in the CUDA flow, and (2) Clang is faster than `gcc`. Hence, the total time is similar in both cases.

4.4. Summary on the CU2rCU Converter

In this section we have shown the feasibility of creating a tool that automatically unextends CUDA source code, which leverages the CUDA extensions to the C language, to its plain C equivalent code. This tool overcomes the limitations imposed by the undocumented functions used by the NVIDIA `nvcc` compiler regarding remote GPU virtualization.

A thorough analysis of the performance of this tool shows that the time it requires to convert and compile a given CUDA code is very similar to the time required by the `nvcc` compiler, thus making it feasible to substitute the regular `nvcc`-based compilation flow by a `CU2rCU`-based one.

5. IMPROVING THE COMMUNICATION ARCHITECTURE IN rCUDA

The rCUDA internal architecture has been enhanced in order to provide efficient support for several underlying client-server communication technologies. In the initial versions of rCUDA (v1, v2, and v3)—with support only for the TCP/IP protocol—communication between rCUDA clients and servers was directly implemented within the code that deploys the general rCUDA functionality, in a monolithic design, as shown in Figure 1. The new modular communication architecture (see Figure 6) included in the new rCUDA v4 supports runtime-loadable specific communication modules. Furthermore, communication between clients and servers has been

[‡] The states `cudafe`, `cudafe++`, `filehash`, `nvopencc`, `ptxas`, and `fatbin` refer to calls to NVIDIA internal compilation tools, while `gcc` refers to calls to the GNU compiler. All these calls are automatically performed by the NVIDIA `nvcc` compiler and, therefore, are transparent to users. For details about why each of the tools are called and about what happens at each step, see [13].

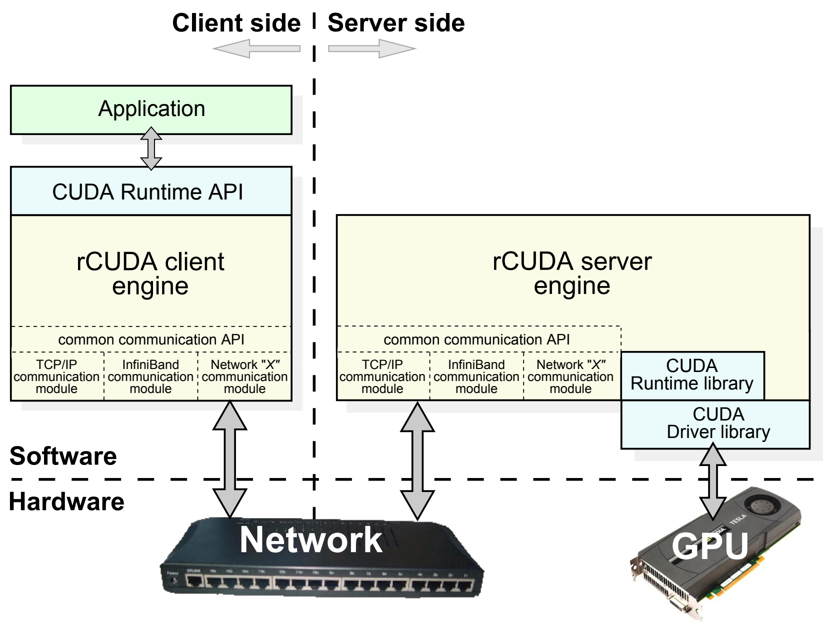


Figure 6. New modular rCUDA architecture, showing also the runtime-loadable specific communication modules.

improved by accommodating a pipelined approach, in order to increase effective throughput and, therefore, squeeze as much bandwidth as possible from the underlying interconnect. All this new functionality, which is transparent to rCUDA users, has been made possible by a carefully designed proprietary common API for rCUDA communications. That API enables rCUDA clients and servers to (1) communicate through different underlying communication technologies (Ethernet, InfiniBand, etc.) and (2) to do so efficiently, since the communication functionality can be specifically implemented and tuned up for each different communication technology (in addition to leveraging the pipelined approach).

The new modular rCUDA architecture currently supports efficient communication over Ethernet and InfiniBand and opens the door to other interesting future network technologies as well as to virtual machine environments such as Xen [22]. Furthermore, regardless of the specific communication technology used, data transfers between rCUDA clients and servers are pipelined in order to improve performance. For this purpose, rCUDA uses preallocated buffers of pinned memory, exploiting the higher throughput that this kind of storage provides. In the following we focus on presenting performance results, since these are the most interesting feature from a usage viewpoint. A complete analysis of the new architecture is beyond the scope of this paper and would require substantially more space. The reader can refer to [24] for a detailed analysis.

Figure 7 illustrates the performance benefits that the new architecture brings to applications leveraging rCUDA. The plot shows the effective bandwidth attained in synchronous memory copy operations (i.e., `cudaMemcpy` calls) to remote GPUs through different interconnects, for the new release of rCUDA (version 4), and for the previous one (version 3). The new highly tuned Ethernet module enables rCUDA v4 to attain up to 99.9% of the effective bandwidth of a Gigabit Ethernet fabric, over 10% more than rCUDA v3 does. The main reason for this improvement is the integration of a pipelined communication in the new communication module. Concerning the 40 Gbps InfiniBand QDR fabric, when employing IP over InfiniBand (IPoIB functionality), rCUDA v4 attains 55.5% of the raw available bandwidth, over 15% more performance than rCUDA v3 does. Again, the pipelined communication mechanism exploited in the new communication module is the largest contributor to this improvement. Furthermore, the specific InfiniBand module, directly employing the InfiniBand Verbs (IBV) API, enables rCUDA v4 applications to reach 97.7% of the available bandwidth. Notice that rCUDA v3 did not provide support for specific InfiniBand

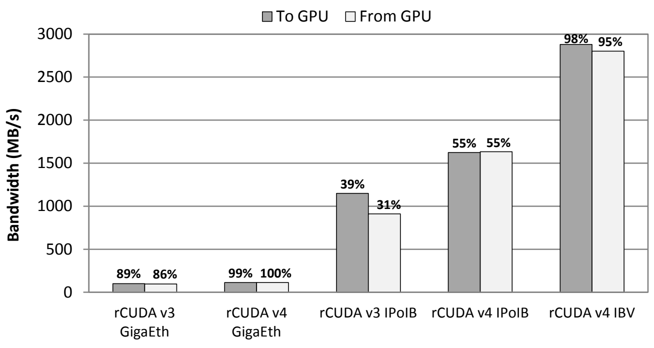


Figure 7. Bandwidth between CPU and remote GPU for several scenarios, using an NVIDIA Tesla C2050 PCIe 2.0-based GPU and Mellanox ConnectX-2 cards. Percentages above the bars represent what fraction of the raw bandwidth of that particular interconnect is attained by rCUDA.

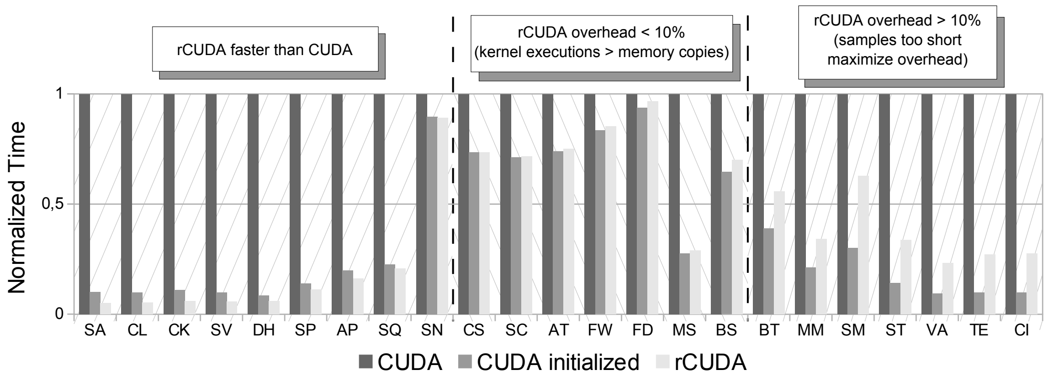
communications. Furthermore, the monolithic design of the previous rCUDA version made it extremely hard to encode such support, while the new modular version approach eases the task of targeting different network fabrics.

The previous results, translated into the execution of an application, lead to an improved user experience, given that overall application execution times are noticeably reduced with respect to the use of the previous rCUDA versions, also featuring negligible overheads when compared with local GPU acceleration. Below, we describe some experiments that demonstrate this.

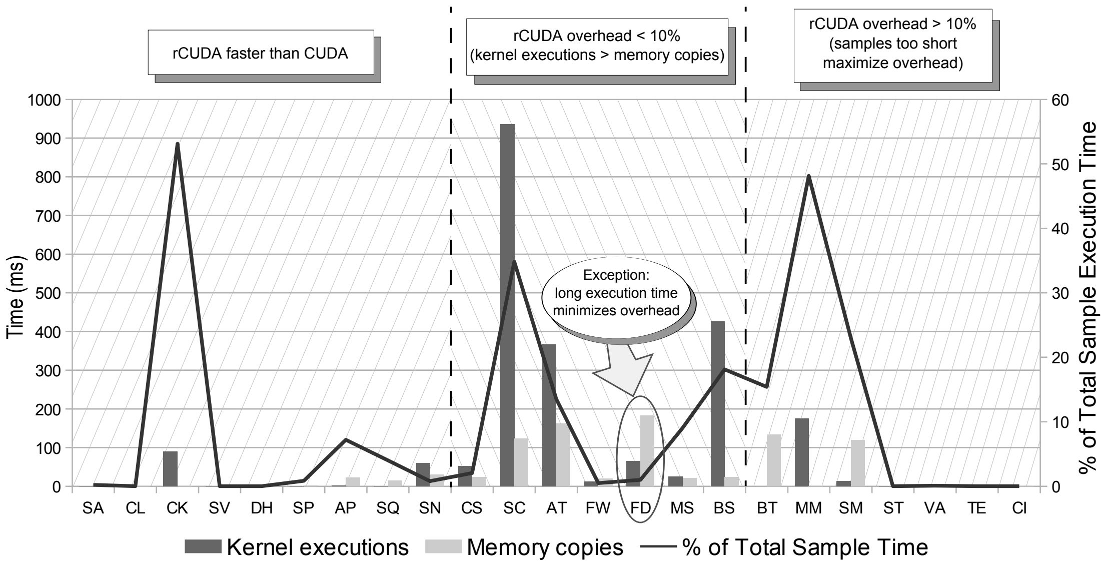
In Figure 8(a) we introduce the performance of the SDK samples converted in the previous section. This figure presents normalized execution time for CUDA and rCUDA over an InfiniBand QDR fabric. Notice that a raw comparison between rCUDA and CUDA is intrinsically unfair, since the rCUDA framework initializes the CUDA environment in the remote GPU server only once, during the rCUDA daemon start-up, whereas CUDA does the initialization each time an application is executed. Hence, when an application leverages CUDA, it has to wait for the initialization of the CUDA environment. On the contrary, applications making use of rCUDA do not have to wait for such initialization because it is already done in the rCUDA server. Therefore, to provide a complete view of the rCUDA performance, we have also included in Figure 8(a) the execution time for the local CUDA but with the CUDA environment already initialized (i.e., the GPU environment previously initialized by other process), labeled as *CUDA initialized* in the figure.

As shown in Figure 8(a), the execution time for rCUDA is, in general, noticeably smaller than that of CUDA, despite the access to a remote GPU. The reason is the initialization overhead. This can be clearly observed when introducing into the comparison the CUDA-initialized execution time, which is also noticeably smaller than that of CUDA and, at the same time, similar to the rCUDA execution time in most of the samples. Table IV shows that the CUDA initialization time is almost constant for all these samples. Since the execution time for the major part of the samples is short (less than 2 seconds), this initialization time (over 1.3 seconds) impairs the aim of these experiments, which is showing rCUDA's overhead. In this regard, we can also observe that, for long samples, the difference between CUDA and rCUDA is smaller. For example, in the FD and SN samples (over 26 and 12 seconds, respectively), the influence of the initialization time is blurred. Hence, to unmask the actual delay introduced by the network, as well as the overhead of the rCUDA framework, from now on we will compare rCUDA execution time versus that of CUDA initialized. In this regard, we can classify the relative performance between CUDA initialized and rCUDA into three main cases: one including those samples for which rCUDA presents an overhead higher than 10% (right side of the plot), one including the samples presenting an overhead lower than 10% (middle part of the plot), and one including the samples that execute faster within the rCUDA framework than with regular CUDA initialized (left side of the plot). Next, we analyze these three cases in more detail.

To explain the results in Figure 8(a), we have used the NVIDIA profiling tools [25] to measure the time spent in transfers (i.e., time spent in memory copies between host memory and the device



(a) CUDA SDK normalized execution time compared with CUDA leveraging the GPU previously initialized by other process (CUDA initialized) and rCUDA. Executions for CUDA were carried out using one node equipped with 4 Quad-Core Intel Xeon E5520 processors and one NVIDIA Tesla C2050. For the rCUDA experiments, two nodes with the same specifications as the node employed with CUDA were used, both connected by an InfiniBand QDR network.



(b) Bars represent the time employed in kernel executions (i.e., CUDA kernels) and memory transfers (i.e., CUDA memcpy) for each one of the CUDA SDK samples, whereas the line reports the percentage that the sum of them (CUDA kernels + CUDA memcpy) represent with regard to the total execution time of the CUDA initialized samples.

Figure 8. Performance of the SDK samples converted in the previous section.

memory, also referred to as *CUDA memcopy*) and the time employed by computations (i.e., time employed by CUDA kernels). Figure 8(b) details these measurements, as well as the cost that this time (CUDA memcopy + CUDA kernels) represents with respect to the total execution time of the samples (plotted with the continuous black line). As shown in Figure 8(a), for samples CS, SC, AT, FW, FD, MS, and BS, the overhead introduced by rCUDA is bearable (i.e., lower than 10%). The reason is that all these samples spend, in general, more time in computations than in transfers. This property benefits rCUDA in the sense that the time spent in computations in the GPU is the same for CUDA and rCUDA, thus compensating for the overhead of rCUDA due to the transfers across the network. The only exception is sample FD, in which the time spent transferring data is greater than the computations time. The explanation for the low overhead of this sample is that it has a long duration (over 24 seconds with CUDA initialized), so that the overhead of transferring data across the network is minimized.

Table IV. Initialization time of the CUDA environment in the analyzed CUDA SDK Samples

CUDA SDK Sample	Time (s)			
	CUDA	CUDA Initialization	CUDA Initialized	rCUDA
Samples that run faster with rCUDA than with CUDA				
simpleAtomicIntrinsics	1.490	1.340	0.150	**0.076**
clock	1.488	1.341	0.147	**0.079**
concurrentKernels	1.519	1.351	0.168	**0.093**
simpleVoteIntrinsics	1.488	1.341	0.147	**0.084**
dwtHaar1D	1.491	1.363	0.128	**0.089**
scalarProd	1.559	1.340	0.219	**0.175**
asyncAPI	1.675	1.341	0.334	**0.273**
SobolQRNG	1.732	1.340	0.392	**0.359**
sortingNetworks	12.735	1.325	11.410	**11.369**
Samples that present an overhead with rCUDA <10%				
convolutionSeparable	5.057	1.342	**3.715**	3.717
scan	4.269	1.229	**3.040**	3.059
alignedTypes	5.274	1.367	**3.907**	3.926
fastWalshTransform	8.062	1.335	**6.727**	6.887
FDTD3d	26.506	1.648	**24.858**	25.646
mergeSort	1.854	1.340	**0.514**	0.538
BlackScholes	3.839	1.361	**2.478**	2.696
Samples that present an overhead with rCUDA >10%				
bandwidthTest	2.215	1.351	**0.864**	1.237
matrixMul	1.725	1.360	**0.365**	0.589
simpleMultiCopy	1.920	1.342	**0.578**	1.205
simpleTemplates	1.564	1.341	**0.223**	0.528
vectorAdd	1.490	1.350	**0.140**	0.346
template	1.488	1.341	**0.147**	0.404
cppIntegration	1.488	1.340	**0.148**	0.412

In Figure 8(a), there are also samples for which the overhead introduced by rCUDA is considerable (i.e., higher than 10%). This is the case of BT, MM, SM, ST, VA, TE, and CI. The explanation for samples BT and SM is that they spend more time in transfers than in computations (see Figure 8(b)). The execution time for the rest of the samples is too short, less than 0.5 seconds, which maximizes their overheads just opposite the way long samples minimize it.

Surprisingly, some samples run faster with rCUDA than with CUDA initialized. This is the case of the SA, CL, CK, SV, DH, SP, AP, SQ, and SN. Even if we assume that the overhead of the rCUDA framework is negligible, they still should present worse results than those of CUDA initialized, given that the additional delay introduced by the network will still be present. Figure 8(b) reveals that these samples present few memory transfers, so that the network overhead is negligible, but this fact still does not explain why rCUDA outperforms CUDA initialized.

A deeper profiling revealed that the analyzed samples have synchronization points, such as calls to `cudaDeviceSynchronize` or `cudaStreamWaitEvent`, that take more time when using CUDA than when using our framework. For instance, the unique call to `cudaDeviceSynchronize` in sample AP takes 529 microseconds in CUDA, whereas it takes only 41 microseconds in rCUDA. The reason for this lies in the internal algorithm used in the rCUDA framework used to determine the finalization of the CUDA tasks, which favors rCUDA in these simple samples. Briefly, this algorithm performs a nonblocking wait during a small period

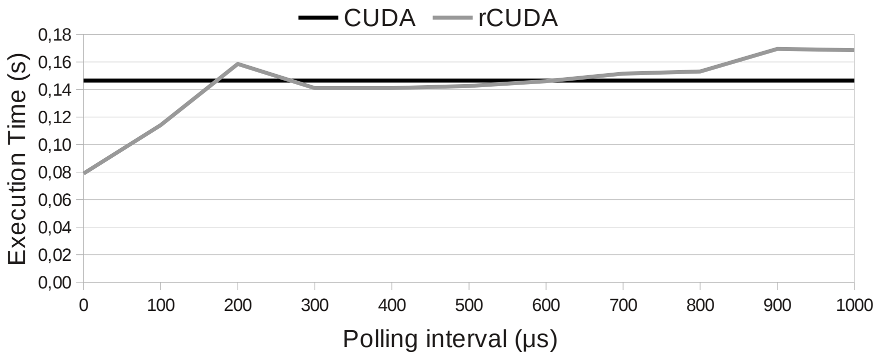


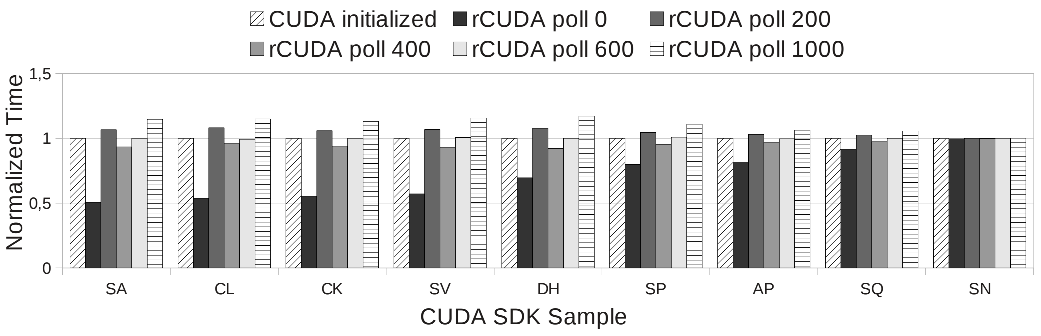
Figure 9. Execution time of the CL sample in CUDA compared with rCUDA using different intervals for polling the network devices. The value for the polling interval within CUDA is the default one used by the CUDA driver. Executions for CUDA were done using one node equipped with 4 Quad-Core Intel Xeon E5520 processors and one NVIDIA Tesla C2050. For the rCUDA experiments, two nodes with the same specifications as the node employed with CUDA were used, both connected by an InfiniBand QDR network.

of time before calling the synchronization function. If this nonblocking wait is successful, the synchronization function is not called, and control is immediately returned to the program. Thus, less time is used at synchronization points.

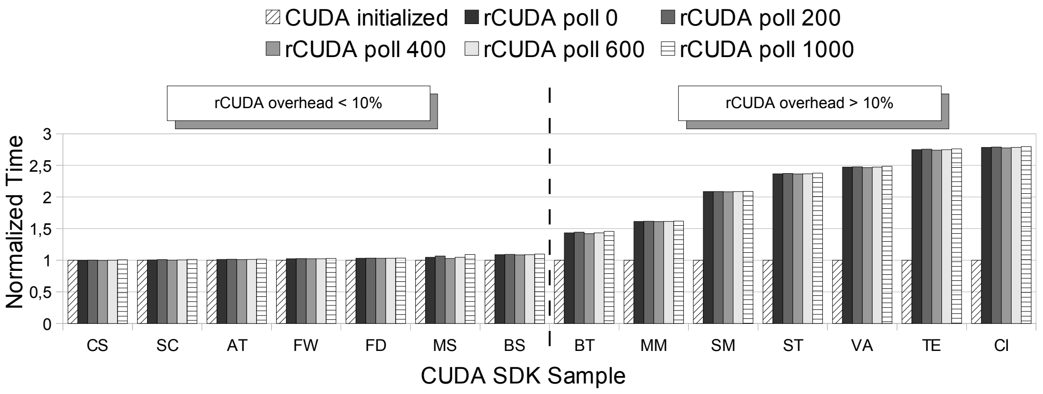
Nevertheless, the time saved in synchronization points in rCUDA does not fully explain the shorter execution time of these samples. For instance, for sample AP, the execution time with rCUDA is 61 ms faster than with CUDA initialized (see Table IV), while the time saved (for the reasons explained above) represents only 0.488 ms. After further research, we found an additional factor that affects execution time: the network polling interval (i.e., the frequency used to poll the network for work completions). This interval in the rCUDA implementation is lower than the default polling interval to the PCIe made by CUDA. To demonstrate this, in Figure 9 we report the execution time of the CL sample when incrementing the rCUDA network polling interval. As we can observe, this factor clearly affects the final execution time, making rCUDA perform better or worse than CUDA initialized depending on its value. Figure 10(a) presents the normalized execution time of all samples for which rCUDA is faster than CUDA initialized, but using different polling intervals, confirming the relevance of this interval. Figure 10(b) presents similar results for the rest of SDK samples analyzed in Table IV.

The previous results have shown the performance achieved by the new rCUDA architecture in the context of the CUDA SDK samples, which are, in general, small codes. Therefore, it is also important to show the overhead of using rCUDA in production codes. For that purpose, in Figure 11 we analyze a more complex case, the matrix-matrix product. The figure shows that, compared with traditional CPU computing, computing the product on a remote GPU is noticeably faster than just using the 8 general-purpose CPU cores of a computing node employing a highly tuned HPC library. Furthermore, the figure shows that rCUDA v3 introduced an overhead of 5% with respect to the local use of CUDA, whereas the new rCUDA v4 introduces a negligible overhead (1%) for this application. This low overhead agrees with the discussion of Figure 8, since the matrix product is a compute-intensive task, thus compensating for the overhead introduced by the transfers across the network.

Similar performance results can be observed for the execution of a much more complex application. In Figure 12, the execution time of a LAMMPS simulation is evaluated in three scenarios: using the OPT package (an optimized CPU package typically attaining 5–20% performance improvement) on the 8 cores of the CPU, employing the USER-CUDA package on a local NVIDIA Tesla C2050 (use of regular local CUDA), and employing rCUDA over an InfiniBand QDR fabric to access a remote GPU. For this experiment we employed the `in.eam` input script included in the standard distribution package under the `bench` directory scaled by a factor of 5 in



(a) SDK samples that execute faster with rCUDA than with CUDA



(b) SDK samples that execute slower with rCUDA than with CUDA

Figure 10. CUDA SDK normalized execution time with the GPU previously initialized by other process (CUDA initialized) compared with rCUDA using different intervals for polling the network devices (0, 200 μs, 400 μs, 600 μs, and 1000 μs). Executions for CUDA were done using one node equipped with 4 Quad-Core Intel Xeon E5520 processors and one NVIDIA Tesla C2050. For the rCUDA experiments, two nodes with the same specifications as the node employed with CUDA were used, both connected by an InfiniBand QDR network.

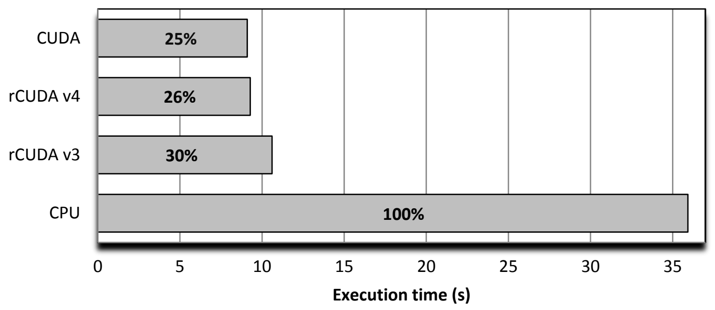


Figure 11. Execution time for a matrix product executed in an NVIDIA Tesla C2050 versus CPU computation on 2 x Quad-Core Intel Xeon E5520 employing GotoBlas 2. Matrices of 13,824x13,824 single-precision floating-point elements. The InfiniBand QDR fabric was used in the rCUDA scenario.

the three dimensions. Here the remote GPU acceleration also yields a faster execution than its CPU-equivalent counterpart, while just introducing a small overhead when compared with the locally accelerated execution. Notice again that the new version of rCUDA introduces less overhead than its predecessor.

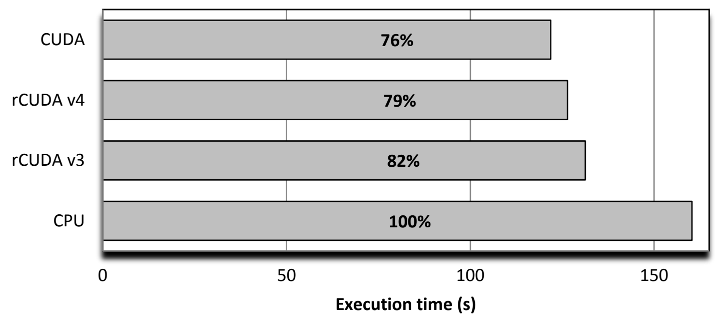


Figure 12. Execution time for the LAMMPS application, *in.eam* input script scaled by a factor of 5 in the three dimensions. Computing node equipped with 2 x Quad-Core Intel Xeon E5520, NVIDIA Tesla C2050, and InfiniBand QDR fabric. The CPU case employs the OPT package and 8 MPI tasks, whereas the USER-CUDA package with one MPI task is used for CUDA (local GPU) and rCUDA (remote GPU).

6. SUPPORT FOR MULTITHREADED APPLICATIONS AND CUDA LIBRARIES

Typically, providing support within rCUDA for the new features introduced in a given CUDA version just requires the implementation of new functions in the rCUDA framework. The exact effort needed to address each new version of CUDA depends on the number of new functions to be introduced into rCUDA and the impact of each new CUDA feature on the rCUDA framework. In this regard, introducing the new functionalities provided by CUDA 4 and 5 into rCUDA basically meant, from a simplistic point of view, increasing the rCUDA code. However, in practice, some of the new features introduced in CUDA 4 and 5 affected rCUDA far more deeply. More specifically, supporting CUDA 4 implicitly implied dealing with multithreaded applications, whereas the appearance of CUDA 5 brought a new way to deal with CUDA libraries. In this section we highlight how rCUDA was evolved in order to integrate both new features.

6.1. Support for Multithreaded Applications

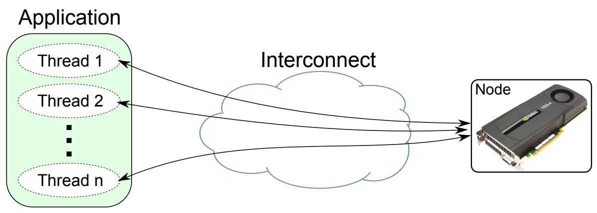
The support for multithreaded applications (first introduced in CUDA 4) has made rCUDA evolve from a non-thread-safe framework to a thread-safe one. This evolution required deep changes in the internal structure of rCUDA. The way rCUDA provides support for multithreaded applications will be thoroughly analyzed next.

Figure 13(a) illustrates one possible scenario, where all the threads of a multithreaded application access the same remote GPU, which is shared among them. The improvements to rCUDA reach further than just supporting new CUDA features; indeed, the new release also allows an application to access all remote GPUs located in different nodes. We refer to this new feature as multinode support, as shown in Figure 13(b).

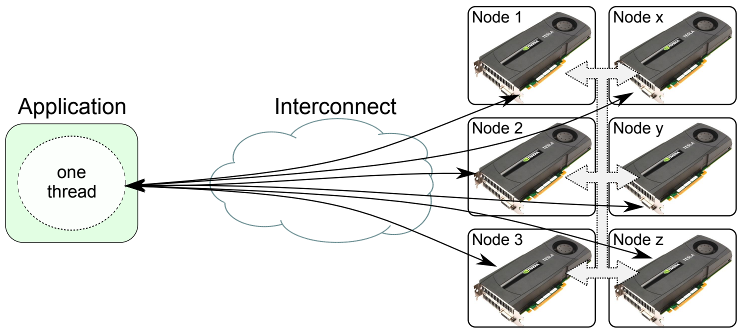
The combination of the new capabilities of rCUDA enables the scenario represented in Figure 13(c), where each thread of a multithreaded application can access remote GPUs located in different nodes.

In the following we present some experiments to analyze these new features. In all the results shown in this section, CUDA experiments were carried out in a node equipped with 2 Quad-Core Intel Xeon E5440 processors and a Tesla S2050 computing system (4 Tesla GPUs); and rCUDA executions were done using 8 nodes, each equipped with 4 Quad-Core Intel Xeon E5520 processors and one NVIDIA Tesla C2050. The InfiniBand QDR fabric was leveraged for rCUDA.

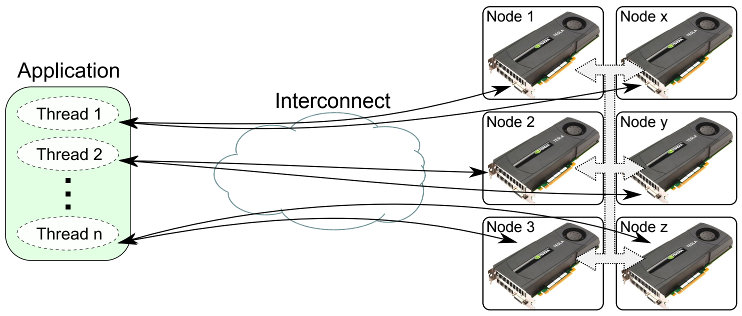
In the first experiment, we used the MonteCarloMultiGPU sample from the NVIDIA GPU Computing SDK, which evaluates fair call price for a given set of European stock options using the Monte Carlo method. We have modified this sample in order to operate on a bigger problem size (a set of 2,048 stock options) and to allow us to specify the number of GPUs to use in the evaluations.



(a) Multithread scenario.



(b) Multinode scenario.



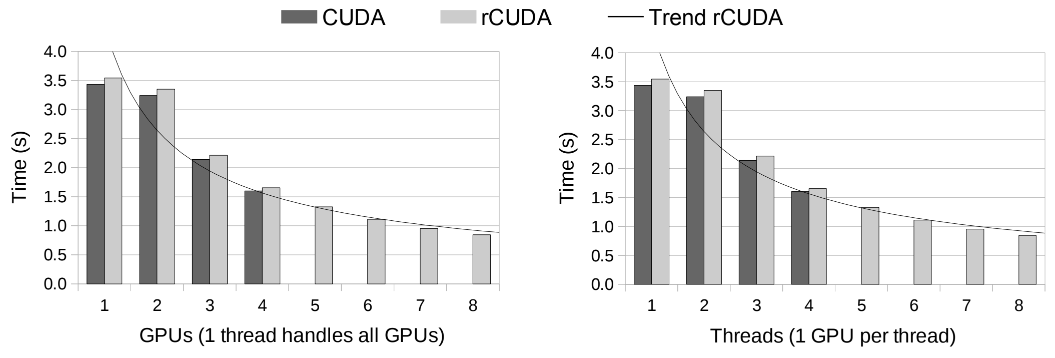
(c) Multinode and a multithread scenario.

Figure 13. Using rCUDA in different scenarios.

Figures 14(a) and 14(b) display the total computation time for this SDK sample. In the experiments, the problem size is constant for all the executions (2,048 stock options), while the number of GPUs and threads involved in the computation varies. This approach allows us to compare the results from different executions in order to assess whether the use of more GPUs and/or threads really reduces the total computation time. It also allows us to compare the scalability features of regular CUDA with those of rCUDA.

Figure 14(a) shows the results of the streamed version of the MonteCarloMultiGPU sample, where one CPU thread handles all GPUs. The problem size is divided among the employed GPUs. Hence, each GPU computes a part of the problem. In the single-GPU case, all the computation is done by that accelerator. Figure 14(b) shows the results for the multithreaded version of the MonteCarloMultiGPU sample in which there is one CPU thread for each GPU. The problem size is also divided among the employed GPUs, but this time each GPU is handled by a different thread.

As shown in Figures 14(a) and 14(b), both versions of the MonteCarloMultiGPU sample, the streamed one and the multithreaded one, provide similar results. Additionally, the total computation time in both versions improves when increasing the number of GPUs. Furthermore, rCUDA not only mimics the behavior of the original CUDA in terms of scalability but also allows an application to use a larger number of GPUs. In this regard, remember that the four GPUs used in these experiments



(a) Streamed version: each GPU executes its part of computation associated with the sole thread.

(b) Threaded version: each GPU executes its part of computation associated with a different thread.

Figure 14. Total computation time for the MonteCarloMultiGPU sample from the NVIDIA GPU Computing SDK. All the executions operate with the same problem size (2,048 stock options). "Trend rCUDA" refers to the trend line with power regression type for rCUDA results.

with CUDA were in the same node, while the eight GPUs used in the executions with the rCUDA framework were located in eight different nodes. The experiments with CUDA for 5, 6, 7, and 8 GPUs were not feasible because of the lack of a node which such an extraordinary equipment. Thus, Figures 14(a) and 14(b) clearly show how rCUDA allows the aggregation of all the GPUs in a cluster and makes them available to a single application. In summary, the limit is no longer the number of available GPUs, but the capability of the application programmer to exploit all of them.

Table V presents the overhead introduced by rCUDA in these experiments. In the worst case, a mere 3.4% of overhead is introduced. Notice that these tables consider only the cases up to four GPUs; data for regular CUDA beyond that number is not available because of the equipment limitations.

Table V. Overhead introduced by rCUDA in MonteCarloMultiGPU sample

	Streamed Version		Multi-threaded Version	
Num. of GPUs	Num. of Threads	rCUDA overhead (%)	Num. of Threads	rCUDA overhead (%)
1	1	3,162	1	3,159
2	1	3,424	2	3,361
3	1	3,384	3	3,421
4	1	3,381	4	3,373

To further illustrate the features of the new rCUDA version, we consider a more complex application, which extends the `libflame` library [26] to perform dense matrix computations taking advantage of multiple GPUs. As in the previous tests, CUDA experiments leveraging more than four GPUs were not feasible because of the lack of that class of equipment. Performance results for this application are depicted in Figure 15. As this figure shows, using more GPUs does not translate into higher performance in this case. The best results are achieved by using CUDA with 3 GPUs; the results were slightly worse with 4 GPUs. With rCUDA, timings improve up to 4 GPUs and remain almost constant when using more GPUs. Once again, rCUDA's behavior is similar to that of the original CUDA in terms of scalability. In this case, however, using a larger number of GPUs is not beneficial because of the nature of the application.

With respect to the overhead introduced by rCUDA in these experiments, Table VI shows that it is higher than in the MonteCarloMultiGPU case. To explain this, we have measured separately the

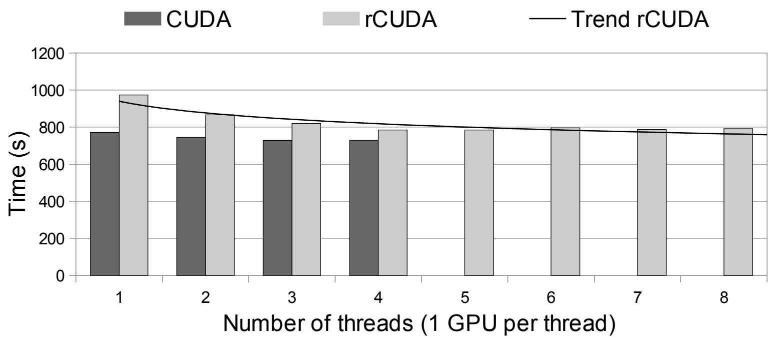


Figure 15. Total execution time for a matrix-matrix multiplication using the `libflame` library. All the executions operate with the same problem size (matrix of 18 million of single-precision elements). Each GPU executes the part of the computation associated with a different thread. “Trend rCUDA” refers to the trend line with power regression type for rCUDA results.

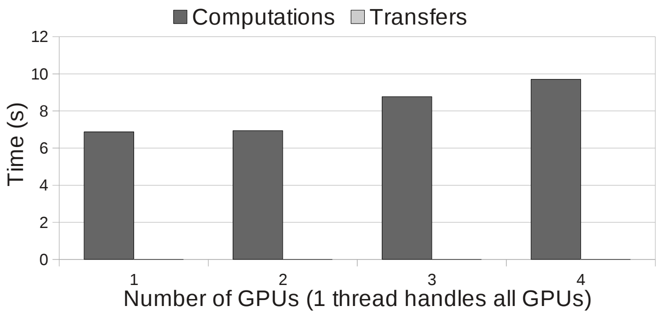
Table VI. Overhead introduced by rCUDA in the matrix-matrix multiplication using the `libflame` library

Num. of Threads	Num. of GPUs	rCUDA overhead (%)
1	1	26,245
2	2	16,284
3	3	12,597
4	4	7,668

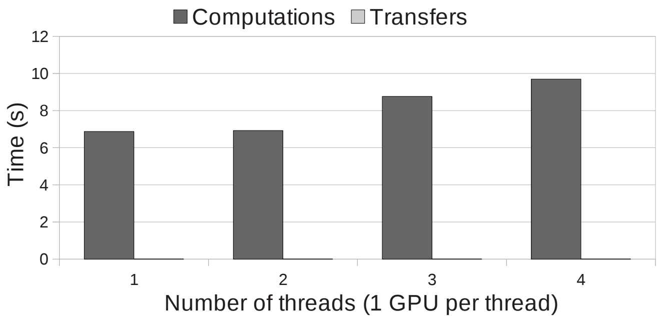
time spent in transfers and in computations for both applications (MonteCarloMultiGPU and the `libflame` matrix-matrix multiplication). The results are shown in Figure 16. Collecting data for more than four GPUs was not possible because we leveraged the NVIDIA profiling tools for the measurements, which are not supported by rCUDA yet. As shown in this figure, both versions of MonteCarloMultiGPU spend almost all their time performing computations. In contrast, the matrix-matrix multiplication spends more time in transfers than with computations. Furthermore, the time spent in transfers increases because the number of GPUs involved in the execution is larger, thus reducing the scalability of the application. On the other hand, performing many more computations than transfers helps rCUDA in the sense that the time spent in computations in the GPU is the same for CUDA and rCUDA, thus compensating the overhead of rCUDA due to the transfers across the network. This explains the low overhead introduced by rCUDA in the MonteCarloMultiGPU example, as well as the higher overhead introduced in the matrix multiplication.

6.2. Support for CUDA Libraries

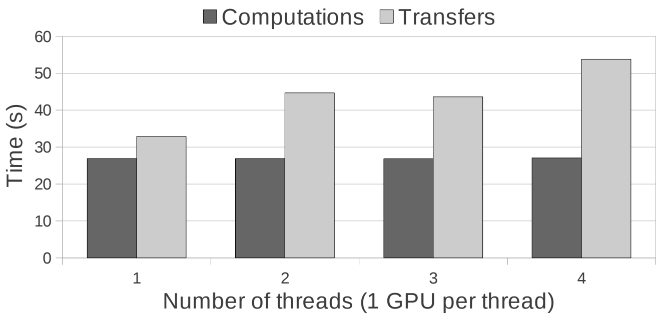
CUDA 5 introduced a significant change in the implementation of the CUDA libraries (CUBLAS or CUFFT, for example) and the way in which these libraries use the APIs offered by CUDA (Runtime and Driver APIs). These new features deeply affected rCUDA in terms of compatibility with CUDA. Before CUDA 5, the CUDA libraries used both the CUDA Runtime API and the CUDA Driver API. The latter was used indirectly by calling a function of the CUDA Runtime API (i.e., cudaGetExportTable) that simply bypassed the call to the CUDA Driver API function (i.e., cuGetExportTable). Since rCUDA does not offer support for the CUDA Driver API, the CUDA libraries were not supported. Nevertheless, in CUDA 5 these libraries have been modified, and now they use only the CUDA Runtime API. Since rCUDA already supports the CUDA Runtime API, the support for any library using the CUDA Runtime API is implicit. Figure 17 illustrates this aspect. The experiments presented in the previous section that used the `libflame` library also validate the



(a) MonteCarloMultiGPU sample (streamed version).

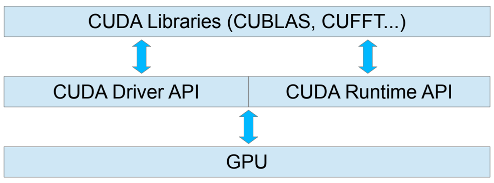


(b) MonteCarloMultiGPU sample (threaded version).

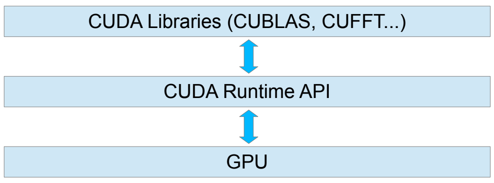


(c) Matrix-matrix multiplication using the `libflame` library.

Figure 16. NVIDIA profiling results for the different applications tested. In particular, time employed by computations (i.e., CUDA kernels) and by memory transfers (i.e., CUDA memcpy). Notice that the time depicted in the plots is the aggregation of the time used by each of the GPUs leveraged in the experiments.



(a) Before CUDA 5.



(b) With CUDA 5.

Figure 17. Integration of CUDA libraries in CUDA 5.

support for CUDA libraries presented in this section, given that the functions of libflame internally make use of the CUBLAS library.

7. USER VIEW OF rCUDA

Having discussed in the previous sections how rCUDA has been enriched, in this section we briefly revisit rCUDA from the user's point of view and provide hints about how to use this middleware successfully in a real cluster.

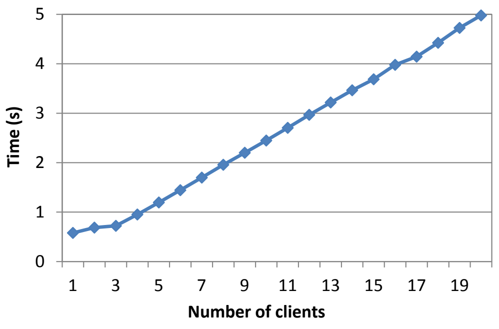
The new version of the rCUDA framework presented in this paper targets the Linux operating system (for 32- and 64-bit architectures), supporting the same Linux distributions as NVIDIA CUDA. This new version of rCUDA accommodates the CUDA Runtime API version 5 except for graphics-related CUDA capabilities, since this particular class of features is rarely of interest in the HPC environment.

As mentioned in Section 3, the rCUDA framework is organized according to a client-server architecture. The client side middleware is distributed in a single file: libcudart.so.5.0. This shared library has to be placed in the machine accessing remote GPGPU services. For this purpose, the LD_LIBRARY_PATH environment variable should point to the location of the shared library. On the server side, the rCUDA daemon (rCUDAd) must be run in the machine(s) offering remote GPGPU services. The RCUDAPROTO environment variable should be appropriately set in the server computer(s), in order to specify the network protocol to be used (remember that this new rCUDA version features a modular communication architecture supporting runtime-loadable specific communication modules). All the rCUDA servers and clients must use the same network protocol.

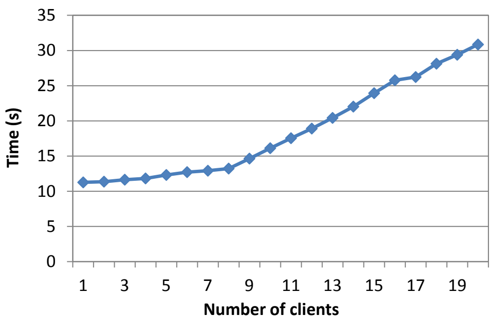
Once the rCUDA middleware is installed, the number and location of the remote GPUs can be set in the client node by setting the environment variables RCUDA_DEVICE_COUNT and RCUDA_DEVICE_n. For example, if two different rCUDA servers are in the cluster (multinode scenario depicted in Figure 13(b)), one with a single GPU and the other with two GPUs, then the variable RCUDA_DEVICE_COUNT should be set to 3, and the locations of the three GPUs can be specified by setting the variables RCUDA_DEVICE_0=192.168.0.1, RCUDA_DEVICE_1=192.168.0.2:0, and RCUDA_DEVICE_2=192.168.0.2:1. In this case, the server with two GPUs is pointed to by the IP address 192.168.0.2. The number after the colon just behind the IP addresses is used to specify the GPU number 0 and the GPU number 1 in the remote server. If the colon is not used, the first GPU will be accessed (by default). In summary, putting the rCUDA remote GPU virtualization framework to work is simple.

Installing the rCUDA remoting framework in a cluster node does not prevent users from still using GPUs locally in that node, either by leveraging the traditional CUDA approach or by using rCUDA to access the local GPU. In the former case, users can still use the NVIDIA CUDA libraries by setting the LD_LIBRARY_PATH environment variable so that it points to the original NVIDIA library. In the latter case, users can use local GPUs through rCUDA by setting the mentioned rCUDA environment variable (i.e., RCUDA_DEVICE_0=localhost). In this case, accessing the local GPU will undergo the overhead introduced by the rCUDA framework, although this overhead is lower than 0.2% according to our measurements.

Furthermore, the RCUDA_DEVICE_n variables in several rCUDA clients may be configured to point to the same target GPUs. This feature implies that the rCUDA servers in the target nodes may concurrently provide GPGPU services to more than one client. To allow this approach, where GPUs are concurrently shared among several demanding applications, rCUDA has been designed so that it uses different rCUDA server processes to support different remote executions over independent GPU contexts. However, sharing a given GPU among several applications presents two drawbacks. In first place, execution time is increased, given that the computational resources of the GPU must be shared among the applications competing for them. Figure 18 shows two examples of such execution time increment for the matrixMult and sortingNetworks CUDA SDK samples. In the second place, given that GPU memory resources are limited, the number of applications that may share a single GPU is constrained by their memory needs. Actually, requesting more memory than available may produce execution errors (the same as for regular CUDA). In order to organize how GPUs across a cluster are used, job schedulers such as SLURM [21] may be extended, making them aware of virtual GPUs. In this way, in addition to requesting the use of GPUs locally, leveraging the regular



(a) matrixMult.



(b) sortingNetworks.

Figure 18. Total execution time for the matrixMult and sortingNetworks SDK samples when the remote GPU is concurrently shared among several applications. Tests are executed using one rCUDA server and up to 20 rCUDA clients, which share one remote GPU (NVIDIA Tesla C2050). Nodes leveraged in the experiment are equipped with 4 Quad-Core Intel Xeon E5520 processors.

CUDA libraries, users may request SLURM exclusive use of remote GPUs for some of their works, whereas they may allow a shared use of remote GPUs for other ones just by specifying the maximum amount of GPU memory that their jobs require. This per-job policy may bring new opportunities to datacentres administrators, who may bill differently exclusive GPU usage from the shared one. In both cases, once users specify their job requirements in the SLURM submission command, the entire management of which exact GPUs will be granted to a given application and whether those GPUs are shared with other jobs or not will be done transparently to applications.

Job schedulers such as SLURM will also be helpful when the overall load of the system exceeds the capability of the cluster. This situation is common during peak loads in current clusters and may occur more often when the number of GPUs is smaller than the number of nodes. When these excessive workloads appear, job schedulers will simply enqueue jobs requesting the busy resources, as they already do with current heterogeneous CPU-GPU cluster configurations. Nevertheless, remote GPU virtualization frameworks allow sharing of remote GPUs, thus making a more efficient use of them and diminishing the frequency of these peak loads (an analysis of the benefits that rCUDA reports to job scheduling in clusters is out of the scope of this paper).

The software and hardware requirements to introduce the rCUDA technology into a cluster are small. As commented, rCUDA supports any of the Linux distributions supported by NVIDIA CUDA (thus not presenting any additional requirement in this regard). It works with any NVIDIA GPU that supports CUDA as well as with any interconnect that provides a TCP/IP interface. Nevertheless, given the overhead that the TCP and IP protocols introduce (shown in Figure 7), the use of high-performance network protocols such as InfiniBand Verbs is preferred. Currently, rCUDA supports only this high-performance network protocol, although this does not represent a real limitation because the InfiniBand interconnect is widely used in HPC clusters, with a quota larger than 40% in the TOP500 list. We are currently working to provide support for other high-performance interconnects.

Regarding the program codes that can be used with the rCUDA framework, thanks to the enrichments presented in this paper (especially the `CU2rCU` source-to-source converter and the multithread support), it is now possible to execute any CUDA program. As shown in the performance evaluation figures in the previous sections, CUDA programs that present high computation/transfer ratios will experience, in general, a much lower overhead than do programs that transfer large amounts of data to/from the GPUs (recall Figure 16). Notice, however, that even when using a local GPU with CUDA, a large amount of transfers to/from the GPU is not a good practice, given that such transfers also result into a performance reduction in the traditional local CUDA usage.

8. CONCLUSIONS

In this paper we have presented the new version of rCUDA, the first complete remote GPU virtualization solution with support for CUDA 5. This virtualization technology is key to our innovative approach to GPGPU green computing because it decouples, from a logical point of view, GPUs from the exact nodes of the cluster where they are installed, thus making the use of GPUs more flexible and paving the way toward efficiently adjusting the number of powered resources to the exact cluster workload.

The new version presented in this paper makes three major contributions to the rCUDA framework. First, a CUDA-to-rCUDA converter has been developed to automatically analyze the application source code in order to find which parts must be modified and adapted to the requirements of rCUDA. This enables leveraging rCUDA for any CUDA application. Second, an enhanced communication architecture has been introduced that features a minimum overhead when accessing remote devices. The new architecture employs runtime-loadable specific communication modules and a pipelined approach for improving performance. The new InfiniBand module has been intensively demonstrated in this paper. Third, support for multithreaded applications and all the CUDA libraries is also provided, at the same time that applications can now exploit all the GPUs available in the cluster. Experiments show that, in general, the overhead of using the new version of rCUDA is very low.

We plan to have future releases of rCUDA that support efficient job scheduling within clusters, as a result of an ongoing integration of rCUDA with the SLURM scheduler.

ACKNOWLEDGMENT

This work was funded by the Generalitat Valenciana under Grant PROMETEOII/2013/009 of the PROMETEO program phase II. This work was partially supported by the U. S. Department of Energy, Office of Science, under Contract No. DE-AC02-06CH11357. The authors are also grateful for the generous support provided by Mellanox Technologies.

REFERENCES

1. A. Barak, T. Ben-Nun, E. Levy, and A. Shiloh. A package for OpenCL based heterogeneous computing on clusters with many GPU devices, *Workshop on Parallel Programming and Applications on Accelerator Clusters* 2010, pp. 1–7
2. 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, pp. 385–394
3. J. Duato, A. J. Peña, F. Silla, R. Mayo, and E. S. Quintana-Ortí. Performance of CUDA virtualized remote GPUs in high performance clusters, *International Conference on Parallel Processing* 2011, pp. 365–374
4. J. Duato, A. J. Peña, F. Silla, J. C. Fernández, R. Mayo and E. S. Quintana-Ortí. Enabling CUDA acceleration within virtual machines using rCUDA, *International Conference on High Performance Computing* 2011, pp. 1–10
5. R. Figueiredo, P. A. Dinda, J. Fortes. Guest editors introduction: Resource virtualization renaissance. *Computer* 2005, 38(5), pp. 28–31
6. Free Software Foundation, Inc. GCC, the GNU Compiler Collection, http://gcc.gnu.org/ [3 June 2013]
7. G. Giunta, R. Montella, G. Agrillo, and G. Coviello. A GPGPU transparent virtualization component for high performance computing clouds, *Euro-Par 2010 - Parallel Processing*, ser. LNCS, vol. 6271, pp. 379–391
8. V. Gupta, A. Gavrilovska, K. Schwan, H. Kharche, N. Tolia, V. Talwar, and P. Ranganathan. GViM: GPU-accelerated virtual machines, *3rd Workshop on System-Level Virtualization for High Performance Computing* 2009, pp. 17–24
9. T. Y. Liang and Y. W. Chang. GridCuda: A Grid-enabled CUDA programming toolkit, *Proceedings of the 25th IEEE International Conference on Advanced Information Networking and Applications Workshops (WAINA)* 2011, pp. 141–146
10. LLVM. Clang: a C language family frontend for LLVM, http://clang.llvm.org/ [3 June 2013]
11. LLVM. The LLVM Compiler Infrastructure, http://llvm.org/ [3 June 2013]
12. G. Martinez, W. Feng, and M. Gardner. CU2CL: A CUDA-to-OpenCL Translator for Multi- and Many-core Architectures, http://eprints.cs.vt.edu/archive/00001161/01/CU2CL.pdf [3 June 2013]
13. NVIDIA. *The NVIDIA CUDA Compiler Driver NVCC Version 5*, NVIDIA 2012
14. NVIDIA. *NVIDIA Industry Cases*, http://www.nvidia.es/object/tesla-case-studies [14 January 2014]
15. NVIDIA. *The NVIDIA GPU Computing SDK Version 4*, NVIDIA 2011

16. D. Quinlan and T. Panas and C. Liao. ROSE, http://rosecompiler.org/ [3 June 2013]
17. C. Reaño, A.J. Peña, F. Silla, J. Duato, R. Mayo, and E.S. Quintana-Orti, CU2rCU: Towards the complete rCUDA remote GPU virtualization and sharing solution *Proceedings of the 19th International Conference on High Performance Computing (HiPC)*, 2012
18. Sandia National Labs. LAMMPS Molecular Dynamics Simulator, http://lammps.sandia.gov/ [3 June 2013]
19. L. Shi, H. Chen, J. Sun, and K. Li. vCUDA: GPU-accelerated high-performance computing in virtual machines, *IEEE Transactions on Computers* June 2012, vol. 61, no. 6, pp. 804–816
20. Zillians. VGPU, http://www.zillians.com/vgpu [3 June 2013]
21. Slurm Workload Manager, http://slurm.schedmd.com [3 June 2013]
22. Citrix Systems, Inc. Xen, http://xen.org/ [3 June 2013]
23. J. Kim, S. Seo, J. Lee, J. Nah, G. Jo, and J. Lee. SnuCL: An OpenCL Framework for Heterogeneous CPU/GPU Clusters, *Proceedings of the 26th International Conference on Supercomputing* 2012, pp. 341–352
24. A. J. Peña. *Virtualization of Accelerators in High Performance Clusters*, Ph.D. thesis, University Jaume I of Castellón 2013
25. NVIDIA. *CUDA Profiler User's Guide Version 5*, NVIDIA 2012
26. F. D. Igual, E. Chan, E. S. Quintana-Ort, G. Quintana-Ort, R. A. van de Geijn, and F. G. Van Zee. The FLAME approach: From dense linear algebra algorithms to high-performance multi-accelerator implementations, *Journal of Parallel and Distributed Computing* 2012, vol. 7, no. 9, pp. 1134–1143
27. S. Xiao et al. VOCL: An optimized environment for transparent virtualization of graphics processing units, *Proceedings of InPar* 2012.