CU2rCU: towards the Complete rCUDA Remote GPU Virtualization and Sharing Solution

C. Reaño, A. J. Peña, F. Silla and J. Duato
Universitat Politècnica de València
46.022–València, Spain
{carregon, apenya}@gap.upv.es,{fsilla, jduato}@disca.upv.es

R. Mayo and E. S. Quintana-Ortí
Universitat Jaume I
12.071–Castellón, Spain
{mayo,quintana}@icc.uji.es

***Abstract*—GPUs are being increasingly embraced by the high performance computing and computational communities as an effective way of considerably reducing execution time by accelerating significant parts of their application codes. However, despite their extraordinary computing capabilities, the adoption of GPUs in current HPC clusters may present certain negative side-effects. In particular, to ease job scheduling in these platforms, a GPU is usually attached to every node of the cluster. In addition to increasing acquisition costs this favors that GPUs may frequently remain idle, as applications usually do not fully utilize them. On the other hand, idle GPUs consume non-negligible amounts of energy, which translates into very poor energy efficiency during idle cycles.**

rCUDA was recently developed as a software solution to address these concerns. Specifically, it is a middleware that allows transparently sharing a reduced number of GPUs among the nodes in a cluster. rCUDA thus increases the GPU-utilization rate, taking care of job scheduling. While the initial prototype versions of rCUDA demonstrated its functionality, they also revealed several concerns related with usability and performance. With respect to usability, in this paper we present a new component of the rCUDA suite that allows an automatic transformation of any CUDA source code, so that it can be effectively accommodated within this technology. In response to performance, we briefly show some interesting results, which will be deeply analyzed in future publications. The net outcome is a new version of rCUDA that allows, for any CUDA-compatible program, to use remote GPUs in a cluster with minimum overhead.

I. Introduction

Due to the high computational cost of current compute-intensive applications, many scientists view graphic processing units (GPUs) as an efficient means of reducing the execution time of their applications. High-end GPUs include an extraordinary large amount of small computing units along with a high bandwidth to their private on-board memory. Therefore, it is no surprise that applications exhibiting a large ratio of arithmetic operations per data item can leverage the huge potential of these hardware accelerators.

In GPU-accelerated applications, high performance is usually attained by off-loading the computationally intensive parts of applications for their execution in these devices. To achieve this, programmers have to specify which portion of their codes will be executed on the CPU and which functions (or *kernels*) will be off-loaded to the GPU. Fortunately, there have been many attempts during the last years aimed at exploiting the massive parallelism of GPUs, leading to noticeable improvements in the programmability of these hybrid CPU-GPU environments. Programmers are now assisted by libraries and frameworks, like CUDA [28] or OpenCL [25] among many others, that tackle this separation process. As a result, the use of GPUs for general-purpose computing (or GPGPU, from General-Purpose computation on GPUs) has accelerated the deployment of these devices in areas as diverse as computational algebra [2], finance [12], health-care equipment [36], computational fluid dynamics [30], chemical physics [31], or image analysis [22], to name only a few. Moreover, the GPU technology mainly targets the gaming market, of high manufacturing volumes and with a largely favorable performance/cost ratio. The net result is that GPUs are being adopted as an effective way of reducing the time-to-solution for many different applications and are thus becoming as a wide-appeal and consolidated choice for the application of high performance computing (HPC) in computational sciences.

The approach currently in use in HPC facilities to leverage GPUs consists in including one or more accelerators per cluster node. Although this configuration is appealing from a raw performance perspective, it is not efficient from a power consumption point of view as a single GPU may well consume 25% of the total power required by an HPC node. Besides, in this class of systems, it is quite unlikely that all the GPUs in the cluster will be used 100% of the time, as very few applications feature such an extreme degree of data-parallelism. Nevertheless, even idle GPUs consume large amounts of energy. In summary, attaching a GPU to all the nodes in an HPC cluster is far away from the green computing spirit, instead being highly energy inefficient.

On the other hand, reducing the amount of accelerators present in a cluster so that their utilization is increased is a less costly and more appealing solution that would additionally reduce both the contribution of the electricity bill to the total cost of ownership (TCO) and the environmental impact of GPGPU through a lower power consumption. However, a configuration where only a limited number of the nodes in the cluster have a GPU presents some difficulties, as 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 harder to be efficiently managed. Moreover, this configuration does not really address the low GPU utilization unless the global scheduler can share GPUs among several applications, a detail that noticeably

978-1-4673-2371-0/12/$31.00 ©2012 IEEE

increases the complexity of these schedulers.

A better solution to deal with a cluster configuration having less GPUs than nodes is virtualization. Hardware virtualization has recently become a commonly accepted approach to improve TCO as it reduces acquisition, maintenance, administration, space, and energy costs of HPC and datacenter facilities [9]. With GPU virtualization, GPUs are installed only in some of the nodes, to be later shared across the cluster. In this manner, the nodes having GPUs become acceleration servers that grant GPGPU services to the rest of the cluster. With this approach, the scheduling difficulties mentioned above are avoided, as now tasks can be dispatched to any node independently of their hardware needs while, at the same time, accelerators are shared among applications, thus increasing GPU utilization. This approach can be further evolved by enhancing the global schedulers so that GPU servers are put into low-power sleeping modes as long as their acceleration features are not required, thus noticeably increasing energy efficiency. Furthermore, instead of attaching a GPU to each acceleration server, GPUs could be consolidated in dedicated servers that would additionally present different amounts of accelerators, so that some kind 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. If the global schedulers are further enhanced so that they accommodate GPU task migration, then this architecture would adhere to the green computing paradigm, as the amount of energy consumed at any moment by the accelerators would be the minimum required for the workload being served.

In order to enable our disruptive power-efficient proposal for HPC deployments, we have recently developed the rCUDA framework [5], [6], [7]. Our technology employs a client-server middleware. The rCUDA client is executed in every node of the cluster, whereas the rCUDA server is executed only in those nodes equipped with GPU(s). The client software resembles a real GPU to applications, although in practice it is only the front-end to a virtual one. In this rCUDA configuration, when an application requests acceleration services, it will contact the client software, which will forward the acceleration request to a cluster node owning the real GPU. There the rCUDA server will process the request and contact the GPU so that it performs the required action. Upon completion, the rCUDA server will usually send back the corresponding results, which will be delivered to the application by the rCUDA client. The application will not become aware that it is dealing with a virtualized GPU instead of its real instance.

Previous work in [5], [6], [7] mainly focused on demonstrating that using remote CUDA devices is feasible. Nevertheless, three main concerns quickly arose during the completion of those studies:

- The usability of the rCUDA framework was limited by its lack of support for the CUDA C extensions. As it will be thoroughly exposed in Section IV, this is due to the fact that the CUDA Runtime library includes several hidden and undocumented functions used by these extensions. One easy way to address this issue would require NVIDIA to provide the needed support by opening the full API (application programming interface) and the required documentation for those currently internal-use-only functions. Unfortunately, commercial reasons hinder disclosing the entire API. Therefore, in order to avoid the use of these undocumented functions, our framework only supports the plain CUDA C API, so far making necessary to rewrite those lines of the application source files that make use of the CUDA C extensions. In the case of the CUDA SDK examples, up to 12.7% of the lines of code had to be modified. For applications comprising large amounts of source code, performing this process manually may be painful.
- The use of remote GPUs in rCUDA reduces performance. In our solution, the available bandwidth between the main memory of the node demanding GPU services and the remote GPU memory is constrained by that of the network connecting client and server. (Note that network bandwidth is usually lower than that of the PCI-Express –PCIe– bus connecting the GPU and the network interface in the server node.) This limitation in bandwidth between client and server noticeably reduces rCUDA's performance, as clearly reported in Figure 1, that depicts the execution time for a matrix-matrix multiplication in different scenarios.
- Finally, the third concern is related with CUDA itself, which is an ongoing technology from NVIDIA with new versions being eventually released. As our rCUDA virtualization solution aims at being compatible with the latest release, it must evolve to support the new CUDA versions. In this regard, the work presented in [5], [6], [7] supported the now obsolete CUDA 2 and 3 versions. After those initial versions of rCUDA, NVIDIA released CUDA 4, with significant changes with respect to prior versions. Therefore, rCUDA had to be upgraded in order to support the new functionality introduced in the last version of CUDA.

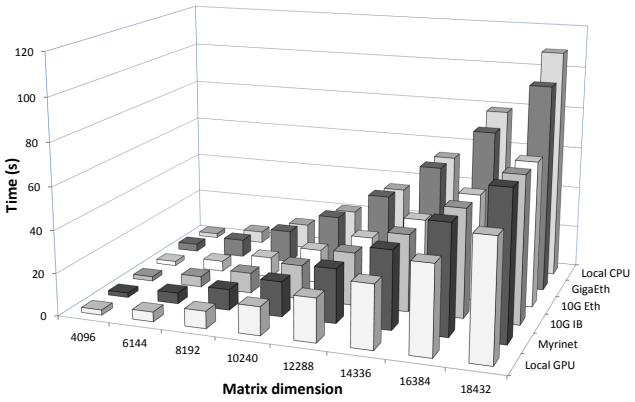


Fig. 1. Execution time for a matrix-matrix product in several scenarios: a GPU local to the host executing the product; rCUDA on top of Myrinet, InfiniBand, and Ethernet networks; a general-purpose multi-core CPU local to the host executing the product. Nodes equipped with 2 x Quad-Core Intel(R) Xeon(R) E5520. The GPU is an NVIDIA Tesla C1060.

In this paper we present how we have addressed these three concerns, focusing on the first one, and giving an overview for the rest. In this manner, we have enriched rCUDA with the following additions:

- A complementary tool, `CU2rCU`, to automatically analyze the application source code in order to find which lines of code must be modified so that the original code is adapted to the requirements of rCUDA. This tool automatically performs the required changes, without the 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 backend compilers after analyzing and adapting the source code files.
- An improved general communication architecture that can later be tuned to a given particular network technology. In this paper we introduce the case for the InfiniBand technology.
- Support for the new CUDA 4, including multi-threaded applications. Additionally, the new version of rCUDA is now able to provide a single application access to GPUs from many different nodes in the cluster (multi-node configuration). This is an important improvement over CUDA, where the amount of GPUs provided to an application is limited to the GPUs which could be attached to a single node (usually not more than 8). In the new version, rCUDA can provide an application direct access to all the GPUs in the cluster, thus boosting application performance. Therefore, the only limit is the ability of the programmer to extract parallelism.

In summary, this paper presents the only CUDA 4 compatible GPGPU virtualization solution existing nowadays that, in addition to enable green computing, also provides applications a virtually unlimited amount of GPUs, thus making the use of GPU accelerators in the HPC context even more appealing.

The rest of the paper is organized as follows. In Section II we present previous work related to our solution. In Section III we introduce the rCUDA technology. The next three sections present how we have addressed the three concerns mentioned above. We first thoroughly describe and analyze the `CU2rCU` tool in Section IV; the new communication architecture is briefly introduced in Section V; and the evolution of rCUDA to support CUDA 4 is concisely presented in Section VI. Although the last two topics well deserve a thorough analysis, for brevity we will focus on the first one. Finally, Section VII summarizes the conclusions of our work and also presents future developments.

II. Related Work

GPU virtualization has been addressed using both hardware and software approaches. From the hardware perspective, perhaps the most prominent commercial solution has been NextIO's N2800-ICA [26], based on PCIe virtualization [15]. This technology allows multiple computers in a rack to share a small number of GPUs, thus enabling cluster configurations with less GPUs than nodes. Nevertheless, the way this solution shares a given GPU considerably limits performance and GPU scheduling, as GPUs can be only assigned to a single node at a time and, therefore, it does not permit concurrent access to the same GPU from several cluster nodes. Additionally, this hardware solution may result substantially expensive.

From the software perspective, there have been many efforts to virtualize GPUs. In this case, as the protocols used at the lowest level in order to address GPUs are proprietary and GPU vendors do not disclose them, the virtualization boundary is usually placed at the high-level APIs, which are always public. In the case of graphics acceleration, these interfaces include Microsoft's Direct3D [3], OpenGL [35], [19], and Cg [11], [23], whereas in the case of GPGPU the most commonly used APIs are CUDA [28] and OpenCL [25].

GPU virtualization intended for graphics acceleration has mainly been addressed in the context of virtual machines, as the applications running in the virtualized computers demand graphics acceleration which is usually provided by the real GPU attached to the host computer. In this context, VMware's Virtual GPU [4] is a GPU virtualization architecture specifically intended for VMware virtual machines. A more general solution is VMGL [16], an OpenGL-based virtualization solution which is independent of the particular virtual machine monitor used, as well as of the GPU architecture deployed.

When GPUs are used in the GPGPU context, the previous solutions, intended for graphics acceleration, cannot be leveraged because they address completely different concerns (e.g., on GPGPU there is no need to take care of the visual output and a bunch of related issues). Additionally, the specifics of virtualizing GPUs drastically change. Therefore, new solutions have been developed, like vCUDA [34], GViM [14], and gVirtuS [13], which pursue the virtualization of the CUDA Runtime API for GPGPU purposes. In the case of OpenCL, the VCL [1] and the SnuCL [39] frameworks provide similar features. All these solutions present a very 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 owning the accelerator, thus having direct access to it.

Unfortunately, the CUDA-based solutions mentioned above present different concerns. In the case of vCUDA, this technology only supports a very old CUDA version (version 1.1) and, additionally, it implements an unspecified subset of the CUDA runtime. Moreover, its communication protocol presents a considerable overhead because of the time devoted to the encoding and decoding stages. This overhead causes a noticeable reduction in the overall performance. GViM is also based on the old CUDA version 1.1 and does not seem to implement the entire runtime API. Regarding gVirtuS, it presents similar problems, as it is based on the old CUDA version 2.3 and only implements a small fraction of the runtime API. For example, in the case of the memory management module, it only implements 17 out of 37 functions.

VGPU [37] is a recent tool that claims to provide very sim-

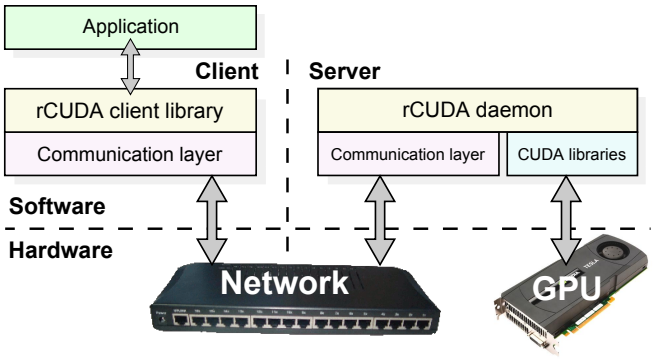


Fig. 2. Overview of the rCUDA architecture.

ilar features to initial public open-source versions of rCUDA. Unfortunately, the information provided by the VGPU authors is very fuzzy and up to now there is no publicly available version that can be used for testing and comparison purposes. In a very similar way, GridCuda [17] 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 only detail similar capabilities to those included in the initial public open-source releases of rCUDA, not providing any insight about the supposedly enhanced features. There is currently no publicly available version of GridCuda that can be downloaded for testing purposes.

III. Review of rCUDA

The rCUDA framework grants applications transparent access to GPUs installed in remote nodes, so that they are not aware of being accessing an external device. This framework is organized following a client-server distributed architecture, as shown in Figure 2.

The client middleware is contacted by the application demanding GPGPU services, both running in the same cluster node. The rCUDA client presents to the application the very same interface as the regular NVIDIA CUDA Runtime API. Upon reception of a request from the application, the client middleware processes it and forwards the corresponding requests to the rCUDA server middleware. In turn, the server interprets the requests and performs the required processing by accessing 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 finally forwarded to the demanding application. Notice that in this approach GPUs are concurrently shared among several demanding applications by using different rCUDA server processes to support different remote executions over independent GPU contexts.

The communication between rCUDA clients and (GPU) servers is carried out via a customized application-level protocol that leverages the network available in the cluster. Figure 3 shows an example of the protocol implemented in the rCUDA framework for a generic request. This example illustrates how a kernel execution request is forwarded from client to server,

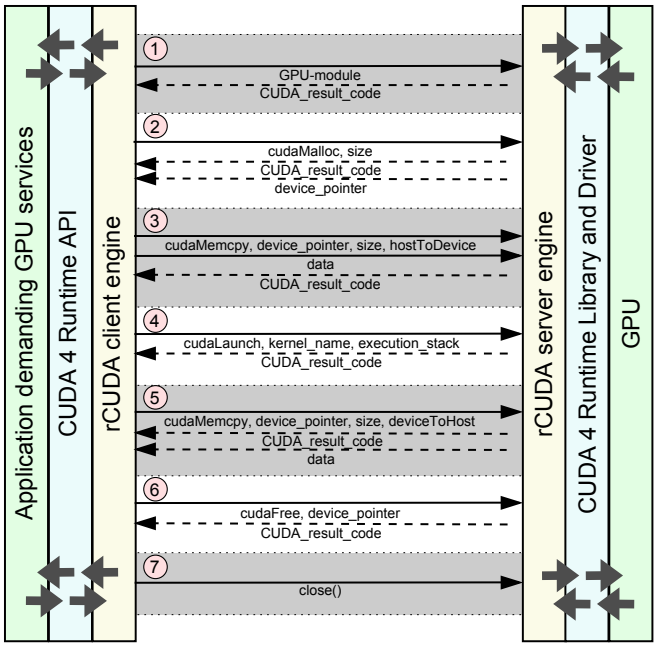


Fig. 3. Example of the proprietary communications protocol used within rCUDA: (1) initialization, (2) memory allocation on the remote GPU, (3) CPU to GPU memory transfer of the input data, (4) kernel execution, (5) GPU to CPU memory transfer of the results, (6) GPU memory release, and (7) communication channel closing and server process finalization.

as well as the dataset used as its input. The retrieval of the output dataset is also displayed.

The most recent version of the rCUDA framework targets the Linux operating system, supporting the same Linux distributions as NVIDIA CUDA. This last version of rCUDA supports the CUDA Runtime API version 4, except for graphics-related CUDA capabilities, as this particular class of features are rarely of interest in our HPC environment.

In general, the performance achieved by rCUDA is expected to be lower than that of the original CUDA, as with rCUDA the GPU is farther away from the invoking application than with CUDA, thus introducing some overhead. Nevertheless, this penalty is currently very low for most applications. Moreover, the performance of applications using rCUDA is often noticeably higher than that provided by computations on regular CPUs. Taking into account the flexibility provided by rCUDA, in addition to the reduction in energy and acquisition costs it enables, rCUDA's benefits definitely overcome the overhead it introduces.

IV. CU2rCU: A CUDA-to-rCUDA Converter

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

A. The Need of 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 control the CPU-GPU interaction via the CUDA API, which aims at easing GPGPU programming. This API includes CUDA extensions to the C language which are constructs following a specific syntax designed to make CUDA programming more accessible, usually leading to fewer lines of source code than its plain C equivalent (although both codes tend to look quite similar). The following piece of code 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:

```
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   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", str);
23   return 0;
24 }
```

CUDA programs are compiled with NVIDIA `nvcc` compiler [27], which looks for fragments of GPU code within the program and compiles them separately from the CPU code. Moreover, during the compilation of a CUDA program, references to structures and functions not made public in the CUDA documentation are automatically inserted into the CPU code. These undocumented functions impair the creation of tools which need to replace the original CUDA Runtime Library from NVIDIA. There exist a few solutions, e.g. *GPU Ocelot* [8], which overcome this limitation by implementing their own versions of these internals, inferring the original functionality. However, this may easily render a behaviour that is not fully compliant with the original library. Furthermore, the stability of these approaches is hampered as the specification of the internals is easily subject to change without prior notification from NVIDIA. To overcome these problems, we have decided to not support these undocumented functions in rCUDA, offering a compile-time work-around which avoids their use instead. Notice that avoiding the use of these undocumented functions requires bypassing `nvcc` for CPU code generation, as this compiler automatically inserts references to them into the host code. Therefore, the CPU code in a CUDA program should be directly derived to a regular C compiler (e.g., GNU `gcc`). On the other side, since a plain C compiler cannot deal with the CUDA extensions to C, they should be *unextended* back to plain C. As manually performing these changes for large programs is a tedious, sometimes error-prone task, we have developed an automatic tool which modifies a CUDA source code employing CUDA extensions and transforms it into its plain C equivalent. In this way, in order to be compiled for execution within the rCUDA framework, a given CUDA source code is split into the following two parts:

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

Coming back to the previous "hello world" CUDA example, the next code snippet shows the transformation of the kernel call in line 17 employing the extended syntax into plain C:

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

4  int main() {
5    // ...
6    cudaConfigureCall(BLOCKS, THREADS);
7    int offset = 0;
8    ALIGN_UP(offset, __alignof(d_str));
9    cudaSetupArgument(&d_str, sizeof(d_str), offset);
10   cudaLaunch("helloWorld");
11   // ...
12 }
```

In order to separately generate CPU and GPU code, we leverage an `nvcc` feature which allows to extract and compile only the device code from a CUDA program and generate a binary file containing only the GPU code. In 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. The separation and transformation process is graphically illustrated in Figure 4.

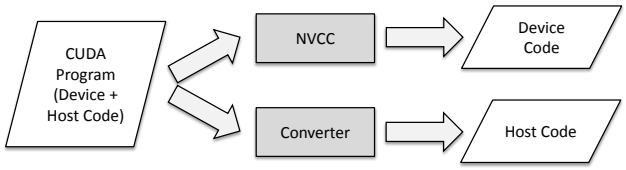


Fig. 4. CUDA-to-rCUDA conversion process.

B. Source-to-Source Transformation Tools

In order to implement the automatic tool that transforms source code employing CUDA extensions into plain C code, a source-to-source transformation framework has been leveraged. Different options for this class of source transformations are available nowadays, from simple pattern string replacement tools to frameworks which parse the source code into an Abstract Syntax Tree (AST) and transform the code using that information. Given that our tool needs to do complex transformations involving semantic C++ code information, we have selected the latter.

There are several frameworks which leverage complex source transformations, as for example ROSE [32], GCC [10], and Clang [20]. We have chosen Clang because, on one hand, it is widely-used and, on the other, it explicitly supports programs written in CUDA. Moreover there are some converters of CUDA source code that are also based on Clang, such as `CU2CL` [24].

Clang, one of the primary sub-projects of LLVM [21], is a C language family compiler which aims, among others, at providing a platform for building source code level tools, including source-to-source transformation frameworks.

Figure 5 shows how the developed converter interacts with Clang. The input 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. After that, the Clang plugin that we have developed, `CU2rCU`, uses the information provided by the AST and the libraries contained in the Clang framework to perform the needed transformations, generating new source files which only contain host code employing the plain C syntax. Notice that during the conversion process our `CU2rCU` tool is able to automatically analyze user source files included by the input files to be converted, also converting them when necessary.

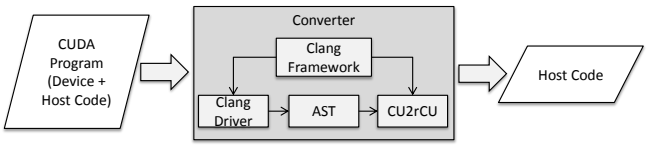


Fig. 5. CUDA-to-rCUDA converter detailed view.

C. `CU2rCU` *Source Transformations*

As explained in the preceding subsections, our converter transforms the original source code written in CUDA into code using only the plain C API, *unextending* CUDA C extensions, and removing device code. Some of the most important transformations are detailed next.

1) Kernel Calls: A kernel call employing CUDA C extensions, as for example:

```
kernelName <<< Dg, Db >>>(param_1 ,... , param_n);
```

must be transformed in order to use the plain C API as follows:

```
cudaConfigureCall(Dg, Db);
int offset = 0;
setupArgument(param_1, &offset);
setupArgument(..., &offset);
setupArgument(param_n, &offset);
cudaLaunch("MangledkernelName");
```

The function `setupArgument()` is provided by the rCUDA framework. It is a wrapper of the plain C API function `cudaSetupArgument()` and, therefore, it just simplifies the inserted code by avoiding the need to explicitly handle argument offsets.

2) Kernel Names: In the `cudaLaunch()` call inserted in the previous transformation, the mangled kernel name must be used if it is not a function with external C linkage. Otherwise, the kernel name as written must be used. For instance, if we have the following kernel declaration:

```
__global__ void increment_kernel(int* x, int y);
```

its mangled name may be used when launching this kernel:

```
cudaLaunch("_Z16increment_kernelPii");
```

However, if the kernel is declared with external C linkage:

```
extern "C"
__global__ void increment_kernel(int* x, int y);
```

the original kernel name has to be used instead.

Determining the mangled kernel name becomes a complex task when there are kernel template declarations with type dependent arguments. For example, for the kernel template declaration:

```
template<class TData> __global__ void testKernel(
  TData *d_odata, TData *d_idata, int numElements);
```

the mangled kernel name used to launch it depends on the type of `TData`:

```
if((typeid(TData) == typeid(unsigned char))) {
  cudaLaunch("_Z10testKernelIhEvPT_S1_i");
} else if((typeid(TData) == typeid(unsigned short))) {
  cudaLaunch("_Z10testKernelItEvPT_S1_i");
} else if((typeid(TData) == typeid(unsigned int))) {
  cudaLaunch("_Z10testKernelIjEvPT_S1_i");
}
```

3) CUDA Symbols: When using CUDA symbols as function arguments, they can be either a variable declared in device code or a character string naming a variable that was declared in device code. As the device code has been removed, only the second option becomes feasible. For this reason, those occurrences that fall into the first category have to be transformed. For instance, in the following function call:

```
__constant__ float symbol[256];
float src[256];
cudaMemcpyToSymbol(symbol, src, sizeof(float)*256);
```

the argument `symbol` has to be surrounded by quotation marks to transform it into a character string:

```
cudaMemcpyToSymbol("symbol", src, sizeof(float)*256);
```

4) Textures and Surfaces: Similarly to CUDA C extensions, in order to use the C++ high level API functions from the CUDA Runtime API, an application needs to be compiled with the `nvcc` compiler. However, as within the rCUDA framework application source code needs to be compiled with a GNU compiler, we need to transform these functions. This is the case of CUDA textures and surfaces. Thus, textures declared using this API, like:

```
texture<float, 2> textureName;
```

are transformed as follows:

```
textureReference *textureName;
cudaGetTextureReference((const textureReference **)
                &textureName, "textureName");
```

A consequence of this transformation is that texture variables become pointers, and access to their attributes such as:

```
textureName.attribute = value;
```

will now result in:

```
textureName->attribute = value;
```

The same transformations explained for CUDA textures apply to CUDA surfaces.

D. Evaluation

In order to test the new `CU2rCU` tool, we have used sample codes from the NVIDIA GPU Computing SDK [29], and production codes from the LAMMPS Molecular Dynamics Simulator [33].

Our first experiments dealt with a number of examples from the NVIDIA GPU Computing SDK. Table I shows the time[1] required for their conversion. In the experiment we employed a desktop platform equipped with an Intel(R) Core(TM) 2 DUO E6750 processor (2.66GHz, 2GB RAM) and a GeForce GTX 590 GPU, running the Linux OS (Ubuntu 10.04). Table I also reports the amount of lines of the original application and the modified sources obtained by our tool. The total amount of time required for the automatic conversion of all these examples, 10.68 seconds, compared with the time spent on a manual conversion by an expert from the rCUDA team, 31.5 hours, clearly shows the benefits of using the converter. Notice 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 obtained is very similar to the one obtained from a manual conversion.

Moreover, we have compared the time spent in the compilation of the original CUDA source code of the SDK samples with the period spent in their conversion and subsequent compilation of the converted code by our tool. Results are shown in Figure 6, demonstrating that the time of converting the original code and later compiling it is similar to the compilation time of the original sources.

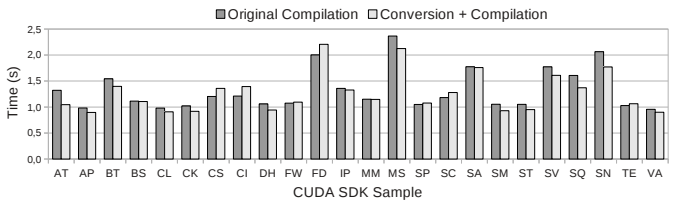


Fig. 6. CUDA SDK compilation time compared with `CU2rCU` conversion plus compilation time.

After having successfully tested the converter with NVIDIA SDK codes, we have evaluated it on a real-world production code: the LAMMPS Molecular Dynamics Simulator.

[1]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 I
NVIDIA GPU COMPUTING SDK CONVERSION STATISTICS

CUDA SDK Sample	Time (s)	Lines		
		CUDA Code	Modified/Added	
			Num.	%
alignedTypes	0.259	186	32	17.20
asyncAPI	0.191	78	6	7.69
bandwidthTest	0.407	708	0	0.00
BlackScholes	0.364	281	13	4.63
clock	0.196	75	8	10.67
concurrentKernels	0.196	100	11	11.00
convolutionSeparable	0.591	319	18	5.64
cppIntegration	0.685	129	12	9.30
dwtHaar1D	0.221	266	11	4.14
fastWalshTransform	0.360	241	20	8.30
FDTD3d	1.082	860	13	1.52
inlinePTX	0.351	91	6	6.60
matrixMul	0.394	272	34	12.50
mergeSort	0.917	1124	105	9.34
scalarProd	0.358	138	10	7.25
scan	0.548	359	26	7.24
simpleAtomicIntrinsics	0.367	211	6	2.84
simpleMultiCopy	0.202	211	22	10.43
simpleTemplates	0.211	241	13	5.39
simpleVoteIntrinsics	0.196	222	19	8.56
SobolQRNG	1.278	10586	8	0.08
sortingNetworks	0.761	571	70	12.26
template	0.357	97	7	7.22
vectorAdd	0.192	88	8	9.09

TABLE II
LAMMPS CONVERSION STATISTICS

LAMMPS Package	Time (s)	Lines		
		CUDA Code	Modified/Added	
			Num.	%
USER-CUDA	6.910	14742	1409	9.56

LAMMPS is a classic molecular dynamics code which can be used to model 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. Table II shows the results of the conversion. The time spent by an expert from the rCUDA team to adapt the original code was two weeks with full time dedication. Again, the benefits of using the converter are clearly proved.

Compilation time of the original LAMMPS source code is compared with conversion plus compilation time in Figure 7, showing that they are close. In this case, the time spent in conversion and compilation is separately shown in order to point out that the conversion process produces a smaller compilation time of the converted code, thus compensating each other.

V. AN IMPROVED COMMUNICATION ARCHITECTURE

The rCUDA internal architecture has been enhanced in order to provide efficient support for several underlying client-server

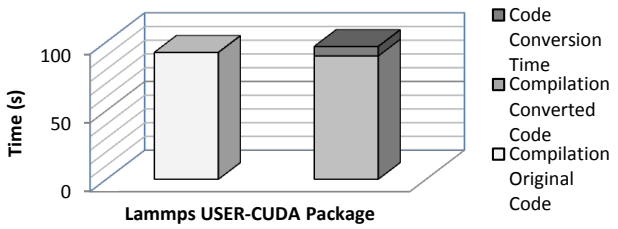


Fig. 7. LAMMPS USER-CUDA `nvcc` compilation time compared with `CU2rCU` conversion plus compilation time.

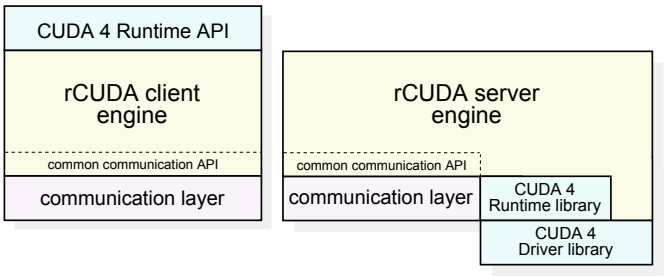


Fig. 8. New modular rCUDA architecture. **Left:** rCUDA client. **Right:** rCUDA server.

communication technologies. In the initial versions of rCUDA —supporting only the TCP/IP protocol— communication between rCUDA clients and servers was directly implemented within the code deploying general rCUDA functionality, in a monothilic design. The new modular communication architecture (see Figure 8) supports runtime-loadable specific communication libraries. This functionality 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, and (2) to do it efficiently, as the communication functionality can be specifically implemented and tuned up for each different communication technology.

The new modular rCUDA architecture currently supports efficient communication over Ethernet and InfiniBand, and opens the door to other interesting technologies like EXTOLL [18] and virtual machine environments like Xen [38]. Due to space constraints, in the following we merely present some performance results, as this is the most interesting feature from a usage viewpoint. A complete analysis of the new architecture would require substantially more space.

Figure 9 illustrates the performance benefits that the new architecture brings to applications leveraging rCUDA. The plot in that figure shows the effective bandwidth attained in synchronous memory copy operations (i.e., `cudaMemcpy` calls) to remote GPUs through different interconnects and communication modules. The highly-tuned Ethernet module enables rCUDA to attain up to 99.9% of the effective bandwidth of a Gigabit Ethernet network, but a mere 55.5% of the 40 Gbps InfiniBand QDR fabric when employing its IP over InfiniBand (IPoIB) functionality. However, the specific InfiniBand module, directly employing the InfiniBand Verbs

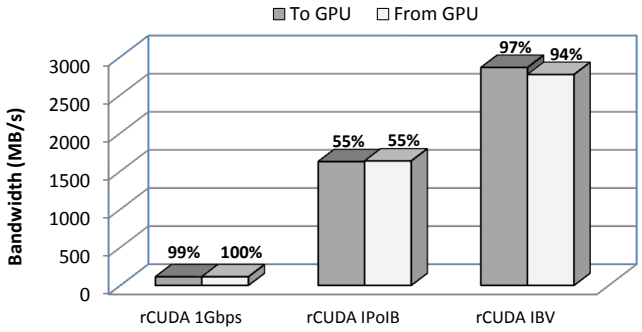


Fig. 9. Bandwidth between CPU and remote GPU for several scenarios: NVIDIA GeForce 9800 and Mellanox ConnectX-2 cards.

(IBV) API, enables rCUDA applications to reach 97.1% of the available bandwidth.

The previous results, translated into the execution of an application, lead to an efficient remote GPU usage with negligible overheads when compared to local GPU acceleration; see Figure 10 for the particular example of a matrix-matrix product. Compared with traditional CPU computing, the figure also shows that computing the product on a remote GPU is noticeably faster than its computation using the 8 general-purpose CPU cores of a computing node employing a highly-tuned HPC library.

Similar performance results can be obtained from the execution of a more complex application. In Figure 11, the execution time of a LAMMPS simulation of the `in.eam` input script included in the standard distribution package under the `bench` directory scaled by a factor of 5 in the three dimensions is compared in the following 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 plain CUDA); employing rCUDA over an InfiniBand QDR fabric to access a remote GPU. Once again, remote GPU acceleration brings a faster execution than its CPU-equivalent counterpart, while just introducing a small overhead when compared to the locally-accelerated execution.

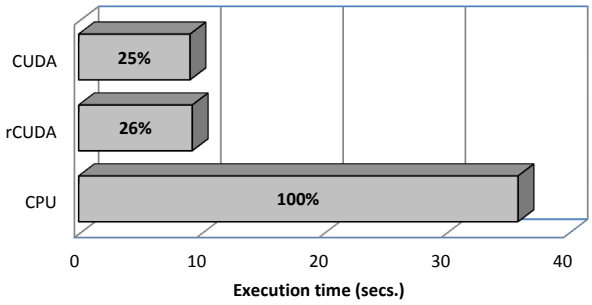


Fig. 10. Execution time for a matrix product executed in an NVIDIA Tesla C2050 versus CPU computation on 2 x Quad-Core Intel Xeon E5520 employing GotoBlas 2. Matrices of 13,824x13,824 single-precision floating point elements.

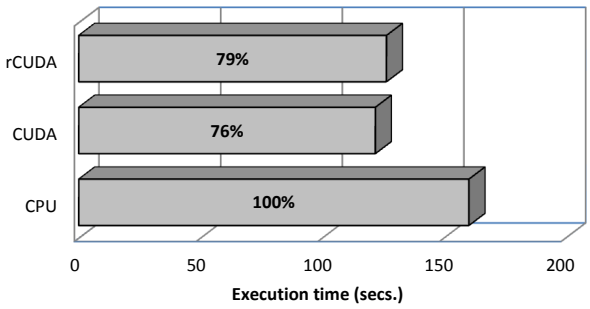


Fig. 11. Execution time for the LAMMPS application, *in.eam* input script scaled by a factor of 5 in the three dimensions. Computing node equipped with 2 x Quad-Core Intel Xeon E5520, NVIDIA Tesla C2050, and InfiniBand QDR fabric. The CPU case employs the OPT package and 8 MPI tasks, whereas the USER-CUDA package with one MPI task is used for CUDA (local GPU) and rCUDA (remote GPU).

VI. CUDA 4 Support

In order to support the new features in CUDA 4, rCUDA has evolved and now supports multi-threaded applications. Figure 12(a) illustrates one possible scenario, where all the threads of a multi-threaded application access the same remote GPU, which is shared among them. The improvements to rCUDA go farther than just supporting new CUDA features; indeed, it also allows an application to access many remote GPUs located in different nodes. We refer to this new feature as multi-node support, as shown in Figure 12(b).

The combination of the new capabilities of rCUDA enables the scenario represented in Figure 12(c), where each thread of a multi-threaded application can access remote GPUs located in different nodes.

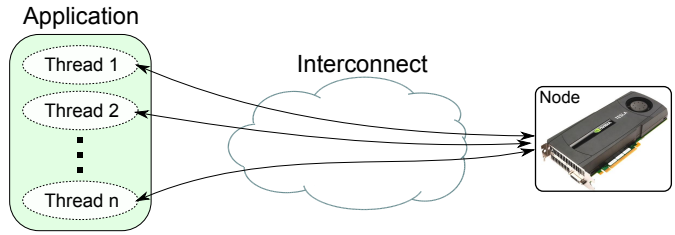
Since a depth analysis of these new features is not possible due to space limitations, we just present results for a single experiment in Figure 13. There we report the normalized execution time for a matrix-matrix product using CUDA and rCUDA, employing several GPUs, where each GPU computes its own product. All the four GPUs used in the experiments with CUDA were in the same node, while the six GPUs used in the execution with the rCUDA framework were located in six different nodes. Notice that the experiments with CUDA for 5 and 6 GPUs were not feasible because of the lack of a node which such an equipment.

As shown in this figure, rCUDA not only mimics the behavior of the original CUDA in terms of scalability, but also allows an application to use a larger amount of GPUs.

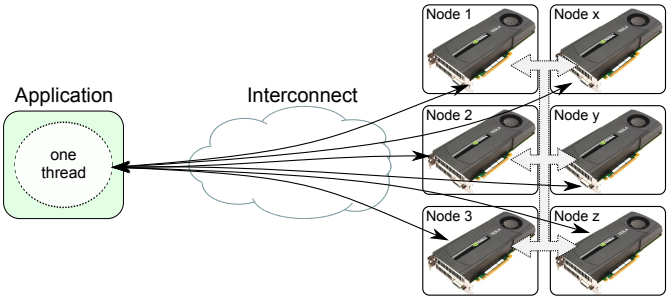
VII. Conclusion

In this paper we have presented the new version of rCUDA, the first complete remote GPU virtualization solution with support for CUDA 4. This virtualization technology allows to fulfill our disruptive approach to GPGPU green computing, providing the flexibility necessary to adjust the number of powered resources to the exact cluster workload.

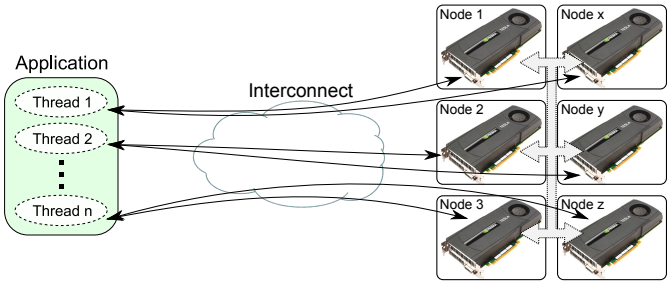
This new rCUDA version comprises a CUDA-to-rCUDA converter, which enables leveraging rCUDA from any CUDA application, as well as an enhanced communication architecture that features a minimum overhead when accessing remote



(a) Multi-thread scenario.



(b) Multi-node scenario.



(c) Multi-node and a multi-thread scenario.

Fig. 12. Using rCUDA in different scenarios.

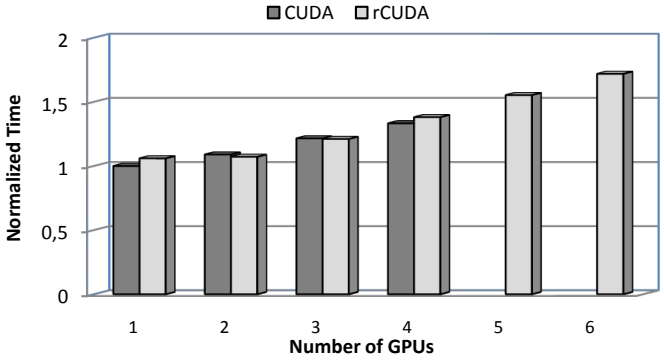


Fig. 13. Matrix-matrix product normalized execution time for a matrix dimension 14,336. Each GPU executes its own product associated with a different thread. CUDA experiments were carried out in a node equipped with 2 Quad-Core Intel Xeon E5440 processors and a Tesla S2050 computing system (4 Tesla GPUs). rCUDA executions were done using 6 nodes, each one equipped with 4 Quad-Core Intel Xeon E5520 processors and one NVIDIA Tesla C2050.

devices and which will be detailed in future work. Moreover, support for the last CUDA 4 version is also provided, at the same time that applications can now exploit a larger amount of GPUs than is possible with CUDA. A deep analysis of these new supported features will be presented in next publications.

Finally, it is planned that future releases of rCUDA will support efficient job scheduling within clusters, as a result of an ongoing integration of rCUDA with schedulers like SLURM.

Further Information

For further details on rCUDA, please visit its web site at http://www.rcuda.net. In particular, instructions can be found in the website about how to obtain a free copy of rCUDA.

Acknowledgment

The researchers at UPV were supported by the Spanish MICINN, Plan E funds, under Grant TIN2009-14475-C04-01 and also by PROMETEO from Generalitat Valenciana (GVA) under Grant PROMETEO/2008/060. Researchers at UJI were supported by the Spanish Ministry of Science and FEDER (contract no. TIN2011-23283), and by the Fundación Caixa-Castelló/Bancaixa (no. P1-1B2009-35).

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," in Workshop on Parallel Programming and Applications on Accelerator Clusters, Sep. 2010.

[2] S. Barrachina, M. Castillo, F. D. Igual, R. Mayo, E. S. Quintana-Ortí, and G. Quintana-Ortí, "Exploiting the capabilities of modern GPUs for dense matrix computations," Concurr. Comput. : Pract. Exper., vol. 21, no. 18, pp. 24572477, 2009.

[3] D. Blythe, "The Direct3D 10 system," ACM Trans. Graph, vol. 25, no. 3, pp. 724734, 2006.

[4] M. Dowty and J. Sugerman, "GPU virtualization on VMware's hosted I/O architecture," in First Workshop on I/O Virtualization. USENIX Association, Dec. 2008.

[5] J. Duato, F. D. Igual, R. Mayo, A. J. Peña, E. S. Quintana-Ortí, and F. Silla, "An efficient implementation of GPU virtualization in high performance clusters," in Euro-Par 2009 Workshops, ser. LNCS, vol. 6043, 2010, pp. 385394.

[6] J. Duato, A. J. Peña, F. Silla, R. Mayo and E. S. Quintana-Ortí, "Performance of CUDA Virtualized Remote GPUs in High Performance Clusters," in International Conference on Parallel Processing 2011, pp. 365–374, 2011.

[7] 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," in International Conference on High Performance Computing 2011, Bangalore, 2011.

[8] N. Farooqui, A. Kerr, G. Diamos, and Y. Gregory, "A framework for dynamically instrumenting GPU compute applications within GPU Ocelot", in Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units, ACM, New York, 2011.

[9] R. Figueiredo, P. A. Dinda, J. Fortes: Guest editors introduction: "Resource virtualization renaissance". Computer 38(5), pp. 28–31, 2005

[10] Free Software Foundation, Inc. , "GCC, the GNU Compiler Collection", online: http://gcc.gnu.org/, last access: March 2012.

[11] F. D. Igual, R. Mayo, and E. S. Quintana-Ortí, "Attaining high performance in general-purpose computations on current graphics processors," in High Performance Computing for Computational Science VECPAR 2008, ser. LNCS, vol. 5336, pp. 406–419.

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

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

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

[15] V. Krishnan, "Towards an integrated IO and clustering solution using PCI Express," in 2007 IEEE International Conference on Cluster Computing, 2007, pp. 259–266.

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

[17] T. Y. Liang and Y. W. Chang, "GridCuda: A Grid-enabled CUDA Programming Toolkit", in Proceedings of the 25th IEEE International Conference on Advanced Information Networking and Applications Workshops (WAINA), pp. 141–146, Singapore, 2011

[18] H. Litz, H. Froening, M. Nuessle, U. Bruening, "VELO: A Novel Communication Engine for Ultra-low Latency Message Transfers", in Proceedings of the 37th International Conference on Parallel Processing (ICPP-08), Portland, 2008.

[19] W. Liu, B. Schmidt, G. Voss, A. Schroeder, and W. Muller- Wittig, "Bio-sequence database scanning on a GPU," in 20th IEEE International Parallel & Distributed Processing Symposium, Rhodes Island, Apr. 2006.

[20] LLVM, "Clang: a C language family frontend for LLVM", online: http://clang.llvm.org/, last access: March 2012.

[21] LLVM, "The LLVM Compiler Infrastructure", online: http://llvm.org/, last access: March 2012.

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

[23] W. R. Mark, S. R. Glanville, K. Akeley, and M. J. Kilgard, "Cg: a system for programming graphics hardware in a C-like language," in SIGGRAPH 03, New York, NY, USA, 2003.

[24] G. Martinez and W. Feng and M. Gardner, "CU2CL: A CUDA-to-OpenCL Translator for Multi- and Many-core Architectures", online: http://eprints.cs.vt.edu/archive/00001161/01/CU2CL.pdf, last access: March 2012.

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

[26] NextIO, "N2800-ICA Flexible and manageable I/O expansion and virtualization." , online: http://www.nextio.com/docs/NextIO%20N2800-ICA%20IO%20Consolidation%20Appliance%20Product%20Brief%20v 0.18.pdf, last access: March 2012.

[27] NVIDIA, "The NVIDIA CUDA Compiler Driver NVCC", NVIDIA, 2011.

[28] NVIDIA, "The NVIDIA CUDA API Reference Manual", NVIDIA, 2011.

[29] NVIDIA, "The NVIDIA GPU Computing SDK", NVIDIA, 2011.

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

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

[32] D. Quinlan and T. Panas and C. Liao, "ROSE", online: http://rosecompiler.org/, last access: March 2012.

[33] Sandia National Labs, "LAMMPS Molecular Dynamics Simulator", online: http://lammps.sandia.gov/, last access: March 2012.

[34] L. Shi, H. Chen, and J. Sun, "vCUDA: GPU accelerated high performance computing in virtual machines," in IEEE International Symposium on Parallel & Distributed Processing (IPDPS09), 2009.

[35] D. Shreiner and OpenGL, OpenGL(R) Programming Guide : The Official Guide to Learning OpenGL(R), Versions 3.0 and 3.1 (7th Edition). Addison-Wesley Professional, Aug. 2009.

[36] S. S. Stone, J. P. Haldar, S. C. Tsao, W. Hwu, Z.P . Liang, and B. P. Sutton, "Accelerating advanced MRI reconstructions on GPUs", in Proceedings of the 2008 conference on Computing Frontiers (CF'08), pp. 261–272. ACM, New York, 2008.

[37] Zillians, "VGPU", online: http://www.zillians.com/vgpu, last access: March 2012.

[38] Citrix Systems, Inc. , "Xen", online: http://xen.org/, last access: March 2012.

[39] J. Kim, S. Seo, J. Lee, J. Nah, G. Jo, and J. Lee, "SnuCL: an OpenCL Framework for Heterogeneous CPU/GPU Clusters", in Proceedings of the 26th International Conference on Supercomputing, June 2012.