UNIVERSITAT POLITÈCNICA DE VALÈNCIA

CU2rCU: A CUDA-to-rCUDA Converter

Master's Thesis in Software Engineering, Formal Methods and Information Systems
Department of Information System and Computation

Author: Carlos Reaño
Advisors: Federico Silla and Germán Vidal

June 26, 2012

Acknowledgements

This master's thesis would not have been possible without the help of several people who, in one way or another, have contributed to its preparation and completion.

First and foremost, my entire gratitude to my advisors, Federico Silla and Germán Vidal, whose guidance has been essential to carry out this work.

Antonio J. Peña, Rafael Mayo, and Enrique S. Quintana-Ortí, of Universitat Jaume I (UJI) at Castelló, for their valuable assistance.

My colleagues in the Parallel Architectures Group (GAP) of Universitat Politècnica de València (UPV), for their moral support.

Last but not the least, my family and friends, for their patience.

Thank you all so much.

Abstract

GPUs are being increasingly embraced by the high performance computing and computational communities as an effective way of considerably reducing application execution time by accelerating significant parts of their codes. For that purpose, NVIDIA is developing since 2006 a new technology called CUDA (Compute Unified Device Architecture) which leverages the parallel compute engine in NVIDIA GPUs to solve many complex computational problems in a more efficient way than it could be done on a regular CPU.

However, despite the extraordinary computing capabilities of GPUs, their adoption 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 configuration favors that GPUs may frequently remain idle, as applications usually do not fully utilize them. Furthermore, idle GPUs consume non-negligible amounts of energy, which translates into very poor energy efficiency during idle cycles.

rCUDA (remote CUDA) was recently developed as a software solution to address these concerns. Specifically, it is a middleware that allows transparently sharing a reduced number of CUDA-compatible GPUs among the nodes in a cluster. rCUDA thus increases the GPU-utilization rate, at the same time that allows to simplify 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, the rCUDA framework was limited by its lack of support for the CUDA extensions to the C language. Thus, it was necessary to manually convert the original CUDA source code into C plain code functionally identical but that does not include such extensions.

For such purpose, in this document we present a new component of the rCUDA suite that allows an automatic transformation of any CUDA source code into plain C code, so that it can be effectively accommodated within the rCUDA technology.

Contents

List of Tables

List of Figures

Chapter 1

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 surprising 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 [1] or OpenCL [2] 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 [3], finance [4], health-care equipment [5], computational fluid dynamics [6], chemical physics [7], or image analysis [8], to name only a few. Moreover, the GPU technology mainly targets the gaming market, leading to high GPU manufacturing volumes, thus presenting a very 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 therefore becoming an appealing and consolidated choice for the application of high performance computing (HPC) to computational sciences.

The approach currently used 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 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 concerns 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 a disruptive power-efficient proposal for HPC deployments, the rCUDA framework has been recently developed [10, 11, 12]. This 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 [10, 11, 12] 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 to the CUDA C extensions. As it will be thoroughly exposed in Chapter 3, 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, the rCUDA 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

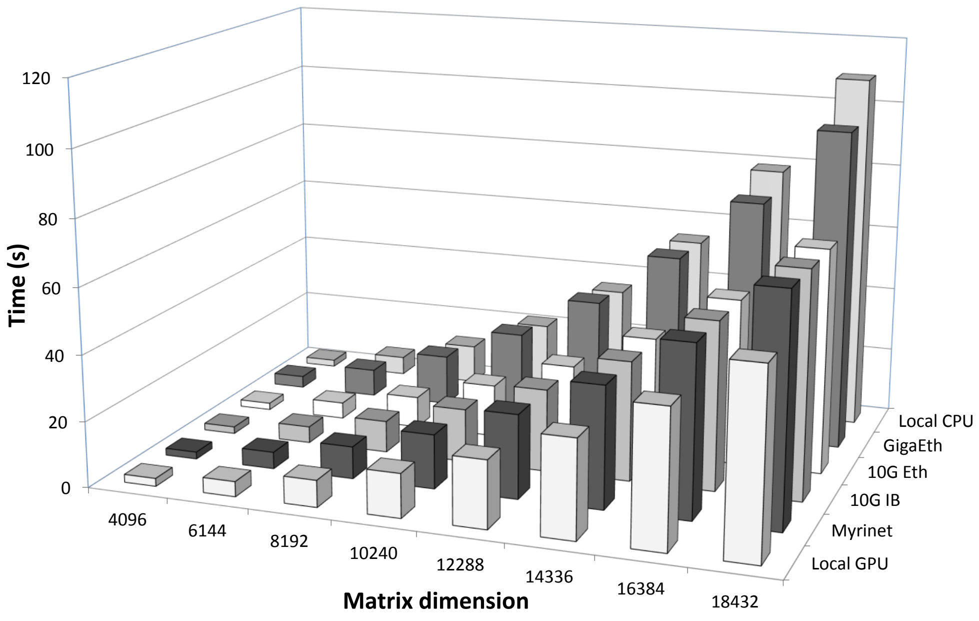


Figure 1.1: Execution time for a matrix-matrix product in several scenarios: a GPU local to the host executing the product; rCUDA on top of Myrinet, InfiniBand, and Ethernet networks; a general-purpose multi-core CPU local to the host executing the product. Nodes equipped with 2 x Quad-Core Intel(R) Xeon(R) E5520. The GPU is an NVIDIA Tesla C1060.

code had to be modified. For applications comprising large amounts of source code, manually performing this process may be painful and error-prone.

- The use of remote GPUs in rCUDA reduces performance. In this 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 1.1, 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 the 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 [10, 11, 12] 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.

In this document we present how we have addressed the first of these three concerns, enriching rCUDA with 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.

The rest of this document is organized as follows:

- Chapter 2 gives an overview of the CUDA architecture, introduces the rCUDA technology and presents several frameworks for source-to-source transformation.
- Chaper 3 describes and analyzes the `CU2rCU` tool.
- Chapter 4 shows the different experiments carried out to evaluate `CU2rCU`.
- Chapter 5 summarizes the conclusions of this work and also presents future developments.

Chapter 2

Background

This chapter provides an overview of the CUDA architecture (Section 2.1), introduces the rCUDA technology (Section 2.2) and presents several frameworks for source-to-source transformation (Section 2.3).

2.1 CUDA: Compute Unified Device Architecture

Nowadays, due to the increasing demand of computing resources required from GPUs (Graphics Processor Units), important progress in their development has been made [13], leading to devices with large computational horsepower and high memory bandwidth, as illustrated in Figures 2.1 and 2.2.

Figure 2.1 presents how using GPUs it is possible to achieve, in some cases, a theoretical rate of floating-point operations per second until 8 times greater than the one obtained using the most powerful CPUs in 2010. At the same time, Figure 2.2 shows that the theoretical memory bandwidth for the GPU is 6 times greater than the one corresponding to the CPU also in 2010, despite the increasing improvements made on the latter to ameliorate it.

The advances made in GPUs have allowed to use them in order to improve the throughput of High Performance Computing (HPC) clusters. For that matter, NVIDIA is developing since 2006 a new technology called CUDA (Compute Unified Device Architecture) which leverages the parallel compute engine in NVIDIA GPUs to solve many complex computational problems in a more efficient way than on a CPU.

CUDA introduces a new parallel programming model and instruction set

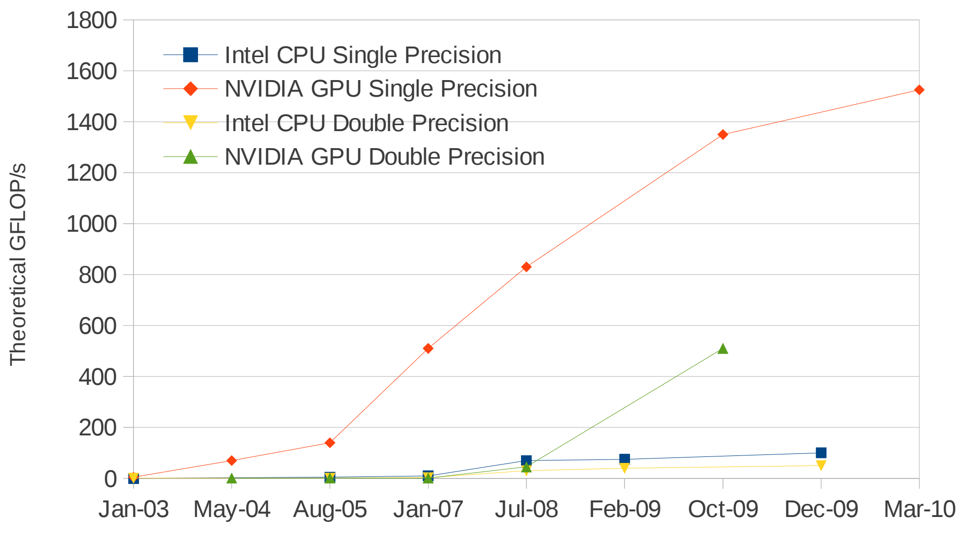


Figure 2.1: Floating-point operations per second for the CPU and GPU.

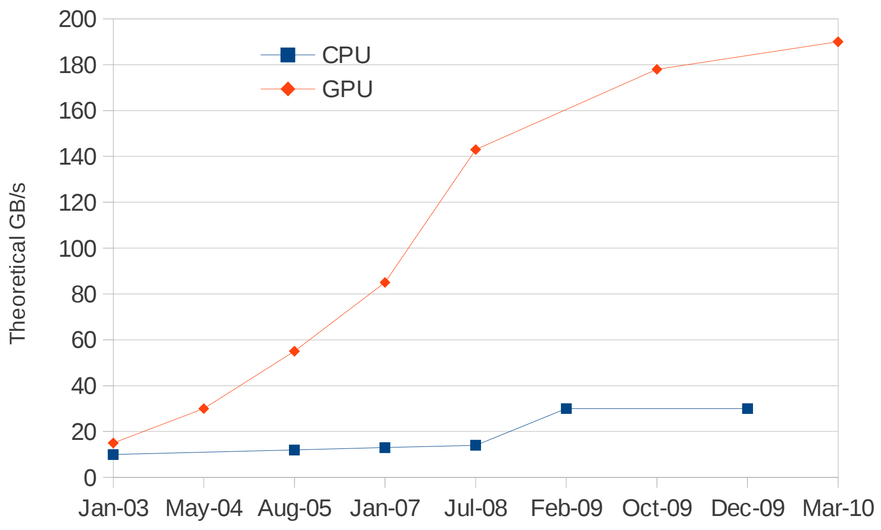


Figure 2.2: Memory bandwidth for the CPU and GPU.

architecture. It also provides a software environment that allows developers to use C and other languages as a high-level programming language. As illustrated by Figure 2.3, CUDA enables the user application to access its architecture via these high level programming languages. Other supported languages are C++, Fortran, Java, Python, and the Microsoft .NET Framework.

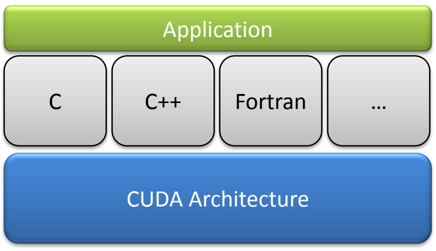


Figure 2.3: CUDA Architecture Programming Interface.

CUDA programs are compiled with the NVIDIA `nvcc` compiler [19], which looks for fragments of GPU code within the program and compiles them separately from the CPU code. The following piece of code shows an example of a simple program written in CUDA which adds two vectors, A and B, of size N and stores the result into vector C[1]:

```
// Device code (executed in the GPU)
// ``__global__'' functions are usually referred as
// ``kernels''.
__global__ void VecAdd(const float* A, const float* B,
                       float* C, int N)
{
    int i = blockDim.x * blockIdx.x + threadIdx.x;
    if (i < N)
        C[i] = A[i] + B[i];
}

// Host code (executed in the CPU)
int main(int argc, char** argv)
{
    int N = 50000;
```

[1]Full code for this example can be found in the `vectorAdd` code sample of the NVIDIA GPU Computing SDK [21].

```
    size_t size = N * sizeof(float);

    // Allocate vectors in host memory (CPU)
    float *h_A = (float*)malloc(size);
    float *h_B = (float*)malloc(size);
    float *h_C = (float*)malloc(size);

    // Initialize input vectors with random float
    // entries.
    RandomInit(h_A, N);
    RandomInit(h_B, N);

    // Allocate vectors in device memory (GPU)
    float* d_A, *d_B, *d_C;
    cudaMalloc((void**)&d_A, size);
    cudaMalloc((void**)&d_B, size);
    cudaMalloc((void**)&d_C, size);

    // Copy vectors from host memory (CPU) to device
    // memory (GPU)
    cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);

    // Invoke kernel VecAdd (will be executed in the
    // GPU).
    int threadsPerBlock = 256;
    int blocksPerGrid =
        (N + threadsPerBlock - 1) / threadsPerBlock;
    VecAdd<<<blocksPerGrid, threadsPerBlock>>>(d_A, d_B,
                                                d_C, N);

    // Copy result from device memory to host memory,
    // h_C contains the result in host memory
    cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);

    // Free device memory (GPU)
    cudaFree(d_A);
    cudaFree(d_B);
    cudaFree(d_C);

    // Free host memory (CPU)
    free(h_A);
    free(h_B);
```

```
59      free(h_C);
60  }
```

In the first place, it is defined the function `VecAdd`, referred to as *kernel* in the CUDA terminology. This function or kernel is executed in the GPU. Next, in the `main` function, data to be used by kernels running on the GPU must be properly allocated (lines 30-32) and copied into the GPU memory space (lines 36-37). Then, the kernel is executed or *launched* in the CUDA terminology (lines 44-45). Once the kernel has finished, the results must be copied again from the GPU memory space to the CPU memory space (line 49). Finally, the memory allocated in the GPU memory space must be free (lines 52-53).

Note that in the example above, the kernel `VecAdd`, as opposed to functions running in the CPU, is implicitly executed several times in parallel by different *CUDA threads*. Thus, the same function is applied to different data at the same time, taking benefit from the CUDA architecture.

2.2 rCUDA: Remote CUDA

As it has already been commented, the approach currently in use in HPC facilities to leverage GPUs consisting in including one or more accelerators per cluster node presents several disadvantages, mainly:

- High power consumption
- Low GPU utilization
- High acquisition cost

In order to overcome these concerns the rCUDA framework was developed. rCUDA is a software which 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 2.4.

The client middleware is contacted by the application demanding GPGPU services, both (the client middleware and the application) 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

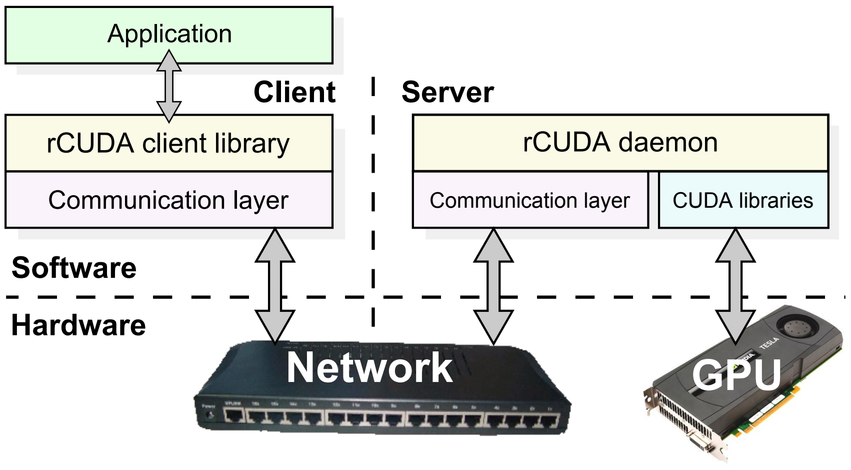


Figure 2.4: Overview of the rCUDA architecture.

of a request from the application, the client middleware processes it and forwards the corresponding requests to the rCUDA server middleware, which is running in a remote node. In turn, the server interprets the requests and performs the required processing by accessing the real GPU to execute the corresponding command. Once the GPU has completed the execution of the requested operation, 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. This feature is the one that allows us to achieve high GPU utilization.

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 2.5 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, 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, which are: Fedora 14, Redhat 5.5 and 6.0, Ubuntu 10.04 and 11.04, OpenSuse 11.2 and Suse Server 11 SP1. This last version of rCUDA supports

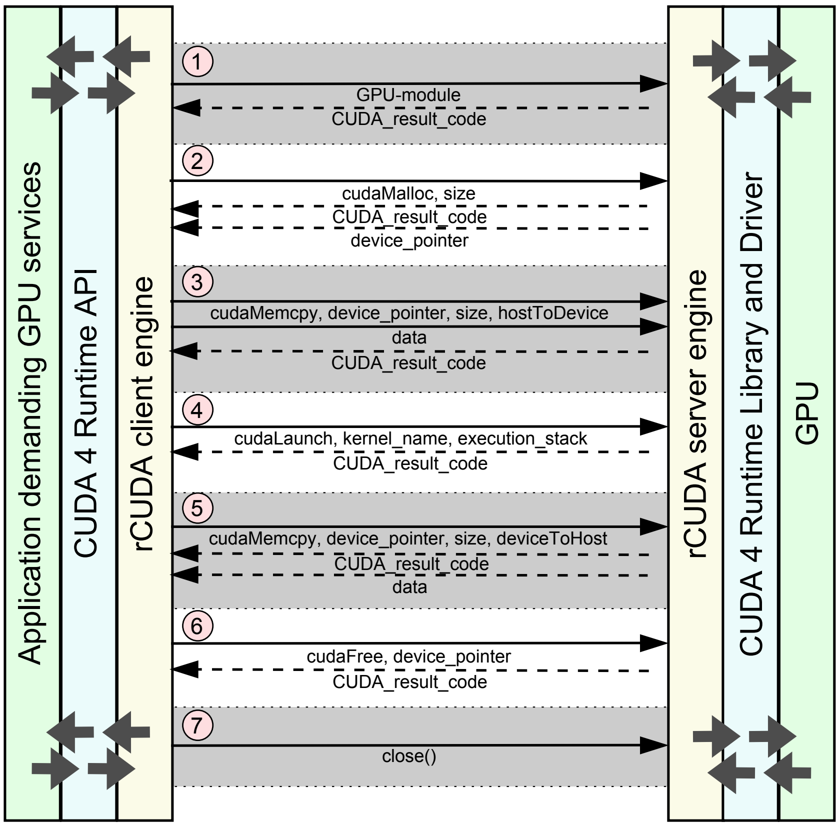


Figure 2.5: Example of the proprietary communications protocol used within rCUDA: (1) initialization, (2) memory allocation on the remote GPU, (3) CPU to GPU memory transfer of the input data, (4) kernel execution, (5) GPU to CPU memory transfer of the results, (6) GPU memory release, and (7) communication channel closing and server process finalization.

the CUDA Runtime API version 4, except for graphics-related CUDA capabilities, as this particular class of features are rarely of interest in the HPC environment. However, they are planned to be supported on future releases, as they could be useful in cloud computing, for example to virtualize desktops or for cloud gaming.

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. Figure 2.6 shows the effective bandwidth attained in memory copy operations to remote GPUs through different interconnects and communication modules. These results, translated into the execution of an application,

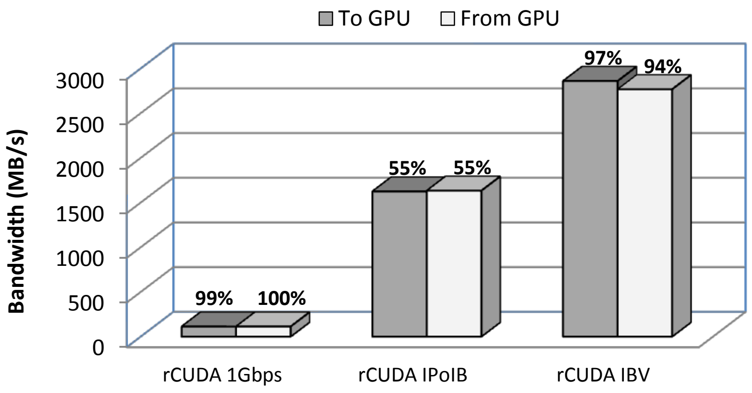


Figure 2.6: Bandwidth between CPU and remote GPU for several scenarios: NVIDIA GeForce 9800 and Mellanox ConnectX-2 cards.

lead to an efficient remote GPU usage with negligible overheads when compared to local GPU acceleration; see Figure 2.7 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.

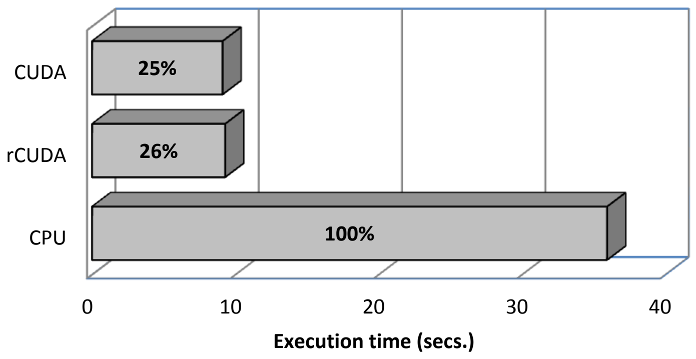


Figure 2.7: Execution time for a matrix product executed in an NVIDIA Tesla C2050 versus CPU computation on 2 x Quad-Core Intel Xeon E5520 employing GotoBlas 2. Matrices of 13,824x13,824 single-precision floating point elements.

In the same way as it happens in the example shown in Figure 2.7, 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 overcome the small overhead it introduces.

2.3 Source-to-source Transformation Tools

As noted in previous sections, the rCUDA framework is limited by its lack of support for the CUDA extensions to the C language. Thus, before executing a CUDA program in the rCUDA framework, it is necessary to manually convert the original CUDA source code into C plain code functionally identical but that does not include such extensions.

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 open source frameworks which leverage complex source transformations, below we detail some of the most popular:

- ROSE [14] is an open source compiler infrastructure to build source-to-source program transformation and analysis tools for large-scale Fortran 77/95/2003, C, C++, OpenMP, and UPC applications. It is particularly well suited for building custom tools for static analysis, program optimization, arbitrary program transformation, domain-specific optimizations, complex loop optimizations, performance analysis, and cyber-security.

- GCC [15], the GNU Compiler Collection, includes front-ends for C, C++, Objective-C, Fortran, Java, Ada, and Go, as well as libraries for these languages (libstdc++, libgcj,...). GCC could be used to perform a preprocessing of GNU C source code, but it would be necessary other tool to perform the automated transformation.

- Clang [16], one of the primary sub-projects of LLVM [18], 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. Below we detail both: the LLVM project and the Clang sub-project.

Between them, we have chosen Clang because, on the one hand, it is widely-used and, on the other hand, 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` [17], which converts source from CUDA to OpenCL [2].

2.3.1 The LLVM Compiler Infrastructure

The LLVM Project [18] is a collection of modular and reusable compiler and toolchain technologies. It began as a research project at the University of Illinois, with the goal of providing a modern, SSA-based (Static Single Assignment) compilation strategy capable of supporting both static and dynamic compilation of arbitrary programming languages. Since then, LLVM has grown to be an umbrella project consisting of a number of different subprojects, many of which are being used in production by a wide variety of commercial and open source projects as well as being widely used in academic research.

As it has been said, Clang is one of the primary sub-projects of LLVM. It is an "LLVM native" C/C++/Objective-C compiler, which aims to deliver fast compiles, useful error and warning messages and to provide a platform for building great source level tools such as source analysis or source-to-source transformation tools.

2.3.2 Clang: a C Language Family Frontend for LLVM

As it is explained in [16], the main goal of the Clang project is to create a new C, C++, Objective C and Objective C++ front-end for the LLVM compiler. Some additional goals for the project include the following (the most relevant for the project will be further detailed next):

- End-User Features:
  - Fast compiles and low memory use.
  - Expressive diagnostics.

  - GCC compatibility.
- Utility and Applications:
  - Modular library-based architecture.
  - Support diverse clients (refactoring, static analysis, code generation, etc.).
  - Allow tight integration with IDEs (Integrated Development Environments).
  - Use the LLVM BSD (Berkeley Software Distribution) License.
- Internal Design and Implementation:
  - A real-world, production quality compiler.
  - A simple and hackable code base.
  - A single unified parser for C, Objective C, C++, and Objective C++.
  - Conformance with C/C++/ObjC and their variants.

End-User Features

A major focus of Clang developers is to make it fast, light and scalable. Thus, the Clang front-end is significantly quicker than GCC and uses less memory. Moreover, Clang developers maintains that the clean framework-based design of Clang allows for many features to be possible that, otherwise, would be very difficult in other systems. For example, incremental compilation, multithreading, intelligent caching, etc.

In addition to being fast and functional, Clang developers say that it aims to be user friendly. They enforce it in several ways, e.g., by making the diagnostics information (error and warning messages) generated by the compiler as useful as possible.

Utility and Applications

A major design concept for Clang is its use of a library-based architecture. In this design, various parts of the front-end can be cleanly divided into separate libraries which can then be mixed up for different needs and uses. In addition, the library-based approach encourages good interfaces and makes it easier for new developers to get involved (because they only need to understand small pieces of the big picture). Currently, clang is divided into the following libraries and tool:

- libsupport - Basic support library. It provides many underlying libraries and data-structures, including command line option processing, various containers and a system abstraction layer, which is used for file system access.
- libsystem - System abstraction library. The library's purpose is to shield LLVM from the differences between operating systems for the few services LLVM needs from the operating system.
- libbasic - This library contains a number of low-level utilities for tracking and manipulating source buffers, locations within the source buffers, diagnostics, tokens, target abstraction, and information about the subset of the language being compiled for.
- libast - Provides classes to represent the C AST, the C type system, builtin functions, and various helpers for analyzing and manipulating the AST (visitors, pretty printers, etc).
- liblex - Lexing and preprocessing, identifier hash table, pragma handling, tokens, and macro expansion.
- libparse - Parsing. This library invokes coarse-grained "Actions" provided by the client (e.g. libsema builds ASTs) but knows nothing about ASTs or other client-specific data structures.
- libsema - Semantic Analysis. This provides a set of parser actions to build a standardized AST for programs.
- libcodegen - Lower the AST to LLVM IR (Intermediate Representation) for optimization and code generation.
- librewrite - Editing of text buffers (important for code rewriting transformation, like refactoring).
- libanalysis - Static analysis support.
- clang - A driver program, client of the libraries at various levels.

Internal Design and Implementation

Clang is designed to be a real-world, production quality compiler, that means being high performance, solid, bug free, and being used by a broad range of people.

With regard to its implementation, Clang is the "C Language Family Front-end", which means it intends to support the most popular members of the C family (C, Objective C, C++, and Objective C++). Clang is based on the idea that the right parsing technology for this class of languages is a hand-built recursive-descent parser. Because it is plain C++ code, Clang developers hold that recursive descent makes it very easy for new developers to understand the code, easily supports ad-hoc rules and other strange hacks required by C/C++, and makes it straight-forward to implement diagnostics and error recovery.

In addition, Clang developers believe that implementing C/C++/ObjC in a single unified parser makes the end result easier to maintain and evolve than maintaining a separate C and C++ parser which must be bugfixed and maintained independently of each other.

Chapter 3

CU2rCU: A CUDA-to-rCUDA Converter

This chapter describes in detail the need for a CUDA-to-rCUDA source-to-source converter and presents the tool developed for that purpose, `CU2rCU`, describing also how to use it.

3.1 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). Nevertheless, once the constructs from the CUDA extensions to C are internally translated by the NVIDIA CUDA compiler to plain C, both codes are equivalent.

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:

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

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

// Host code
int main(int argc, char **argv) {
  char h_str[] = ``Hello World!'';
  // ...
  cudaMalloc((void**)&d_str, size);
  // copy the string to the device
  cudaMemcpy(d_str, h_str, size, cudaMemcpyHostToDevice);
  // launch the kernel
  helloWorld<<< BLOCKS, THREADS >>>(d_str);
  // retrieve the results from the device
  cudaMemcpy(h_str, d_str, size, cudaMemcpyDeviceToHost);
  // ...
  cudaFree(d_str);
  printf(``%s\n'', str);
  return 0;
}
```

As mentioned before, CUDA programs are compiled with the NVIDIA `nvcc` compiler [19], 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* [20], 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 of these undocumented functions is easily subject to change without prior notification from NVIDIA.

To overcome these problems, we have decided not supporting these undocumented functions in rCUDA, offering instead a compile-time work-around which avoids their use. 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, within the work presented in this report, 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 (this part will be further detailed later).

Coming back to the previous "hello world" CUDA example, the next code shows the transformation of the kernel call in line 17 of the previous code employing the extended syntax into plain C (lines 20-24 of the following code):

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

 4 #define ALIGN_UP(offset, align) (offset) = \
 5 ((offset) + (align) - 1) & ?((align) - 1)

 7 // Device code
 8 __global__ void helloWorld(char* str) {
 9   // GPU tasks.
10 }

12 // Host code
13 int main(int argc, char **argv) {
```

```
14    char h_str[] = ‘‘Hello World!’’;
15    // ...
16    cudaMalloc((void**)&d_str, size);
17    // copy the string to the device
18    cudaMemcpy(d_str, h_str, size, cudaMemcpyHostToDevice);
19    // launch the kernel
20    cudaConfigureCall(BLOCKS, THREADS);
21    int offset = 0;
22    ALIGN_UP(offset, __alignof(d_str));
23    cudaSetupArgument(&d_str, sizeof(d_str), offset);
24    cudaLaunch(‘‘helloWorld’’);
25    // retrieve the results from the device
26    cudaMemcpy(h_str, d_str, size, cudaMemcpyDeviceToHost);
27    // ...
28    cudaFree(d_str);
29    printf(‘‘%s\n’’, str);
30    return 0;
31 }
```

In the kernel call employing the extended syntax, firstly, grid and block dimensions are specified (line 20). Next, the only argument of the kernel is set (line 23). Finally, the kernel is launched (line 24). The rest of the code will be the same as in the first code shown for the example.

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. Notice also that `nvcc` does not need the CPU code to be removed from the original source code to generate the binary file with only the GPU code. The separation and transformation process is graphically illustrated in Figure 3.1.

3.2 Interaction with Clang

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. As explained in Section 2.3 of Chapter 2,

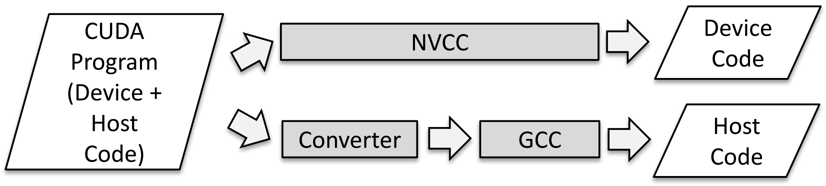


Figure 3.1: CUDA-to-rCUDA conversion process.

we have chosen Clang for this purpose.

Figure 3.2 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 section. 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 (sentence `#include <...>`), also converting them when required.

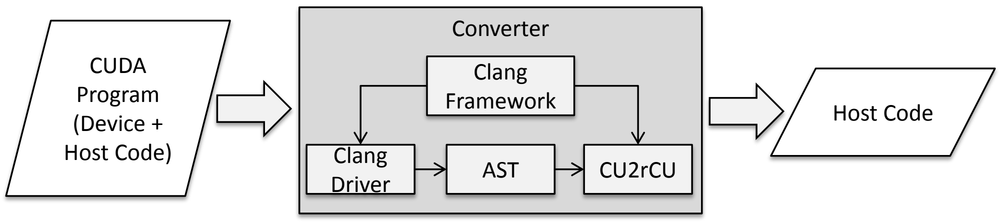


Figure 3.2: CUDA-to-rCUDA converter detailed view.

3.3 CU2rCU Source Transformations

As explained in the preceding sections, our converter transforms the original source code written in CUDA into code using only the plain C API, *unextend-*

ing CUDA C extensions, and removing device code. The transformations to be carried out, which will be detailed next, concerns to:

- Function and Variable Types Qualifiers
- Kernel Calls
- Kernel Names
- CUDA Symbols
- C++ API Routines
  - Texture and Surface Declarations
  - Texture and Surface Functions
  - CUDA Symbol Functions
  - Kernel Functions

3.3.1 Function and Variable Types Qualifiers

CUDA extends C language [13] with the following function type qualifiers:

- `__device__`
- `__global__`
- `__host__`
- `__noinline__`
- `__forceinline__`

And with the following variable type qualifiers:

- `__device__`
- `__constant__`
- `__shared__`
- `__restrict__`

All of them, except the `__host__` qualifier, refer to functions/variables which execute on the device. So the transformation consists on removing these functions and variables, excluding `__host__` functions, which are not removed.

3.3.2 Kernel Calls

As specified in [13], the functions declared with the `__global__` qualifier, also called kernels, are executed on the device (that is why we remove them, as explained in previous section). However, these kind of functions are only callable from the host and, therefore, just removing them would lead to errors in the source code. So, additional transformations are required.

Consequently, a kernel call from the host employing CUDA C extension, that is, using an expression of the form[1]:

```
functionName <tparameter_1,...,tparameter_n>
             <<< Dg, Db[, Ns[, S]] >>>
              (parameter_1,...,parameter_n);
```

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

```
cudaConfigureCall(Dg, Db[, Ns[, S]]);
int offset = 0;
setupArgument(parameter_1, &offset);
setupArgument(..., &offset);
setupArgument(parameter_n, &offset);
cudaLaunch(``MangledKernelName'');
```

The function `setupArgument()` is provided by the rCUDA framework (the header `rCUDA_util.h` is automatically included in the source code by our `CU2rCU` tool). It is a wrapper of the plain C API function `cudaSetupArgument()` and, hence, it just simplifies the inserted code by avoiding the need to explicitly handle argument offsets.

3.3.3 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:

[1]From now on, arguments between square brackets "[argument]" means that they are optional.

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

its mangled name should 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:

```
cudaLaunch(``increment_kernel'');
```

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'');
}
```

The task is even more complex if the call to a kernel template is inside a function template, as for instance:

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

template<class TData> void runTest(void){
  ...
  testKernel<TData><<<64, 256>>>(
      (TData *)d_odata,
      (TData *)d_idata,
      numElements
  );
  ...
}
```

In both cases, the complexity lies in finding out in what types `TData` will be specialized. For this, it is necessary to parse all the code and, after collecting all possible specializations, return back to the kernel call in order to rewrite it properly.

3.3.4 CUDA Symbols

When using CUDA symbols as function arguments, they can be either a variable declared in the device code or a character string naming a variable that was declared in the 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.

The following functions of the CUDA Runtime API [1] use CUDA symbols:

```
cudaMemcpyToSymbol(symbol, src, count[, offset[, kind]]);

cudaMemcpyToSymbolAsync(symbol, src, count, offset,
          kind[, stream]);

cudaMemcpyFromSymbol(dst, symbol, count[, offset[,
          kind]]);

cudaMemcpyFromSymbolAsync(dst, symbol, count, offset,
          kind[, stream]);
```

If the argument `symbol` is not a character string, it must be converted to such variable type. In general, this transformation just consists in putting it in quotation marks as follows:

```
cudaMemcpyToSymbol(``symbol'', src, count[, offset[,
        kind]]);

cudaMemcpyToSymbolAsync(``symbol'', src, count, offset,
        kind[, stream]);

cudaMemcpyFromSymbol(dst, ``symbol'', count[, offset[,
        kind]]);

cudaMemcpyFromSymbolAsync(dst, ``symbol'', count,
        offset, kind[, stream]);
```

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);
```

It must be taken into account that the argument `symbol` could also be a macro definition; in that case, the result of this macro must be used. For example, in the following function call:

```
#define MY_PREFIX prx
#define MY_VAR_TO_STR(var) #var
#define MY_VAR_TO_STR2(var) MY_VAR_TO_STR(var)
#define MY_CONST(var) (MY_VAR_TO_STR2(MY_PREFIX) ``_''
        MY_VAR_TO_STR2(var))

__constant__ float symbol[256];
float src[256];
```

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

is the result of the macro `MY_CONST` what has to be surrounded by quotation marks:

```
cudaMemcpyToSymbol(``prx_symbol'', src,
            sizeof(float)*256);
```

3.3.5 C++ API Routines

Similarly to the CUDA C extensions, in order to use the C++ high level API functions from the CUDA Runtime API [1], an application needs to be compiled with the `nvcc` compiler. However, as within the rCUDA framework the application source code needs to be compiled with a GNU compiler, we need to transform these functions.

C++ API routines are just wrappers of the CUDA Runtime API functions which just simplify the code by avoiding the need to explicitly informing or casting some arguments. Thus, C++ API routines could be seen as a high-level API and the CUDA Runtime API as a low-level API. For example, the following code using the low-level API [13]:

```
texture<float, cudaTextureType2D,
        cudaReadModeElementType> texRef;
textureReference* texRefPtr;
cudaGetTextureReference(&texRefPtr, ``texRef'');
cudaChannelFormatDesc channelDesc;
cudaGetChannelDesc(&channelDesc, cuArray);
cudaBindTextureToArray(texRefPtr, cuArray, &channelDesc);
```

it is equivalent to the following one using the high-level API:

```
texture<float, cudaTextureType2D,
        cudaReadModeElementType> texRef;
cudaBindTextureToArray(texRef, cuArray);
```

The difference is that in the first version the function call to `cudaBindTextureToArray()` refers to a function from the CUDA Runtime API and the second version to one of the C++ API routines.

In Sections 3.3.6, 3.3.7, 3.3.8 and 3.3.9 are detailed the transformations related with C++ API routines.

3.3.6 Texture and Surface Declarations

CUDA [13] supports a subset of the texturing hardware that GPUs use for graphics to access texture and surface memory. In this section, and the following ones, we refer to textures and surfaces as variables which reference texture and surface memory.

Textures declared using the C++ API, that is:

```
texture<T, [texType[, mode]]> textureName;
```

must be 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. Thus:

```
surface<T[, dim]> surfaceName;
```

changes into:

```
surfaceReference *surfaceName;
cudaGetSurfaceReference((const surfaceReference
        **)&surfaceName,
                        ‘‘surfaceName’’);
```

And access to their attributes:

```
surfaceName.attribute = value;
```

turns into:

```
surfaceName->attribute = value;
```

Note that the calls to `cudaGetTextureReference()` and `cudaGetSurface-Reference()` are only necessary before using the texture/surface for the first time. Furthermore, textures and surfaces can only be declared as global variables. Thus, the previous function calls could not be inserted after declaring the texture/surface. For instance:

```
texture<float, 2> tex;

int main(int argc, char    argv   )
{
    ...
    // set texture attributes
    tex.normalized = true;
    tex.filterMode = cudaFilterModeLinear;
    tex.addressMode[0] = cudaAddressModeClamp;
    tex.addressMode[1] = cudaAddressModeClamp;
    ...
}
```

will be transformed into:

```
1  textureReference *tex;

3  int main(int argc, char      argv      )
4  {
5      ...
6      // set texture attributes
7      cudaGetTextureReference((const textureReference**)
              &tex, ``tex'');
8      tex->normalized = true;
9      tex->filterMode = cudaFilterModeLinear;
10     tex->addressMode[0] = cudaAddressModeClamp;
11     tex->addressMode[1] = cudaAddressModeClamp;
12     ...
13 }
```

In addition to have changed the access to the texture attributes, it has been necessary to call `cudaGetTextureReference()` before using the texture (line 7).

3.3.7 Texture and Surface Functions

In this section are detailed the conversions done to function calls that manage textures and surfaces.

cudaBindTexture

When binding a memory area to a texture using the C++ API routine:

```
cudaBindTexture(offset, tex, devPtr [, desc] [, size]);
```

if argument `desc` is not present, we must inform it:

```
cudaBindTexture(offset, tex, devPtr, &tex->channelDesc[,
        size]);
```

Moreover, if `desc` is present and its type is `const struct cudaChannel-FormatDesc`, it must be transformed into type `const struct cudaChannel-FormatDesc *`, that is:

```
cudaBindTexture(offset, tex, devPtr, desc[, size]);
```

becomes:

```
cudaBindTexture(offset, tex, devPtr, &desc[, size]);
```

cudaBindTexture2D

Similar transformations must be applied when binding a 2D memory area to a texture:

```
cudaBindTexture2D(offset, tex, devPtr [, desc], width,
            height, pitch);
```

If the argument `desc` is not informed, the transformed code will be the following:

```
cudaBindTexture2D(offset, tex, devPtr ,
            tex->channelDesc, width, height, pitch);
```

If `desc` is informed and its type is `const struct cudaChannelFormatDesc`, then the new code will be:

```
cudaBindTexture2D(offset, tex, devPtr , &desc, width,
            height, pitch);
```

cudaBindTextureToArray

In a similar way, when binding an array to texture:

```
cudaBindTextureToArray(tex, array[, desc]);
```

If the argument `desc` is not present, it must be informed:

```
cudaChannelFormatDesc desc;
cudaGetChannelDesc(&desc, array);
cudaBindTextureToArray(tex, array, &desc);
```

If `desc` is present and its type is `const struct cudaChannelFormatDesc`, it must be transformed into `const struct cudaChannelFormatDesc *`:

```
cudaBindTextureToArray(tex, array, &desc);
```

cudaBindSurfaceToArray

In the same way, when binding an array to a surface:

```
cudaBindSurfaceToArray(surf, array[, desc]);
```

If the argument `desc` is not present, it must be informed:

```
cudaChannelFormatDesc desc;
cudaGetChannelDesc(&desc, array);
cudaBindSurfaceToArray(surf, array, &desc);
```

If `desc` is present and its type is `const struct cudaChannelFormatDesc`, it must be transformed into `const struct cudaChannelFormatDesc *`:

```
cudaBindSurfaceToArray(surf, array, &desc);
```

3.3.8 CUDA Symbol Functions

The same transformations explained in Section 3.3.4 apply for functions:

```
cudaGetSymbolAddress(devPtr, symbol);
```

and:

```
cudaGetSymbolSize(size, symbol);
```

This means that, in a general way, if the argument `symbol` is not a character string, it must be transformed as follows:

```
cudaGetSymbolAddress(devPtr, ``symbol'');
```

and:

```
cudaGetSymbolSize(size, ``symbol'');
```

As explained in Section 3.3.4, note that if `symbol` is a macro definition, what must be quoted is the result of this macro.

3.3.9 Kernel Functions

When referring to a kernel in the host code, we must use a character string with the kernel name to avoid the use of the CUDA C extensions. For this reason, when using one of the following C++ API routines:

```
cudaFuncSetCacheConfig(function, cacheConfig);

cudaLaunch(function);

cudaFuncGetAttributes(attr, function);
```

if the argument `function` is not a character string, we must transform them as follows:

```
cudaFuncSetCacheConfig(``functionName'', cacheConfig);

cudaLaunch(``functionName'');

cudaFuncGetAttributes(attr, ``functionName'');
```

As in kernel calls explained in Section 3.3.3, note that the argument `functionName` must be the mangled function name if it is not a function with external, C linkage.

3.3.10 Included Files

In general, included files must be parsed in order to have all the necessary information to properly apply all the transformations described in previous sections. In particular, every time a CUDA user included file is found, as for example:

```
#include ''user_cuda_file.cu''
#include ''user_cuda_header.cuh''
```

these CUDA files, in addition to be parsed, must also be transformed. However, if the CUDA files have already been transformed during the conversion of other file(s) in which they where also included, the job must not be repeated. This leads to save time during the conversion process.

3.4 Compilation Flow

This section presents the resulting rCUDA compilation flow after integrating the `CU2rCU` converter within the rCUDA framework, comparing it with the original CUDA compilation flow.

On the one hand, in Figure 3.3 we can see the original CUDA compilation flow with the NVIDIA `nvcc` compiler [19]. The states named `cudafe`, `cudafe++`, `filehash`, `nvopencc`, `ptxas` and `fatbin` refer to calls to NVIDIA internal compilation tools, while the ones named `gcc` refer to calls to the GNU compiler. Notice that all these calls are automatically performed by the `nvcc` compiler and, therefore, they are transparent to users. As can be seen, the input program, `input.cu`, is separated into host code and device code. During the process, the device code is transformed into binary code and embedded into the previously separated host code. Finally, the host code with the binary device code embedded is compiled with `gcc`, generating an executable which also has the binary device code embedded.

On the other hand, Figure 3.4 shows the rCUDA compilation flow. Initially, we apply to the input program the `CU2rCU` converter to obtain its equivalent source code using only plain C and without device code. From this converted code, we get an executable which have references to device code stored in an external repository. This external repository of device code is generated by `nvcc`, which has an option for specifying a concrete compilation phase, instead of executing the whole process. In this case, we use the compilation phase called "fatbin", which only compiles the device code from the input program, generating an external repository with it.

3.5 CU2rCU Installation and Deployment

In this section we will show how to install and use the `CU2rCU` converter on Unix-like systems. There is no support for other platforms.

As it has been explained in previous sections, `CU2rCU` uses Clang's facility for providing a plugin. To install `CU2rCU` properly we must follow the following steps.

Step 1: Installing LLVM and Clang

In order to install `CU2rCU`, first we have to install the LLVM Compiler Infrastructure and its Clang sub-project. There are two ways of doing this:

- Download Clang binaries, version 3.0, from `http://llvm.org/releases/download.html#3.0` and extract them in the desired directory. These binaries also include LLVM.
- Download Clang source code and build it following the instructions given in Section *Get Started* from Clang webpage `http://clang.llvm.org/get_started.html`.

Step 2: Installing CU2RCU

Once we have installed Clang, we get the `CU2rCU` software, which is distributed with the rCUDA framework, place it in the chosen directory and edit the file named *Makefile*. In this file, we modify the variable called `LLVM_PATH` indicating the path where Clang has been installed. For instance, if we have installed Clang in the following path `''home/user/clang+llvm-3.0''`, we will set as follows the `LLVM_PATH` variable:

```
LLVM_PATH := home/user/clang+llvm-3.0
```

Then, we run the *Makefile* from the path where it is placed to build the plugin. For example, if we have extracted `CU2rCU` software in the path `''home/user/CU2rCU''`, then we open a terminal, change to that directory and type command `make`.

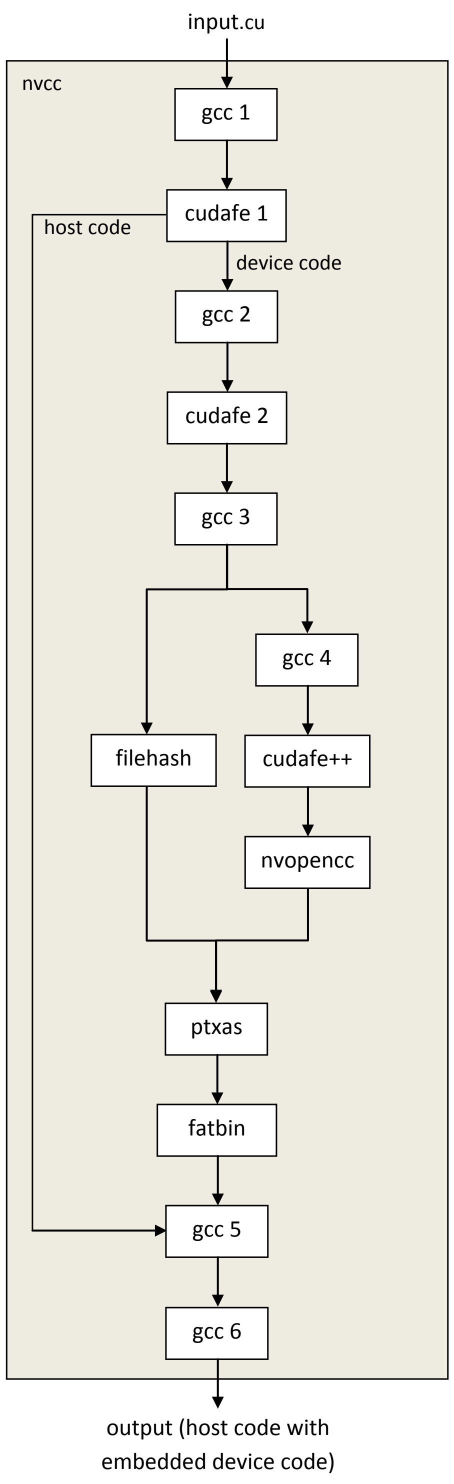


Figure 3.3: CUDA compilation flow.

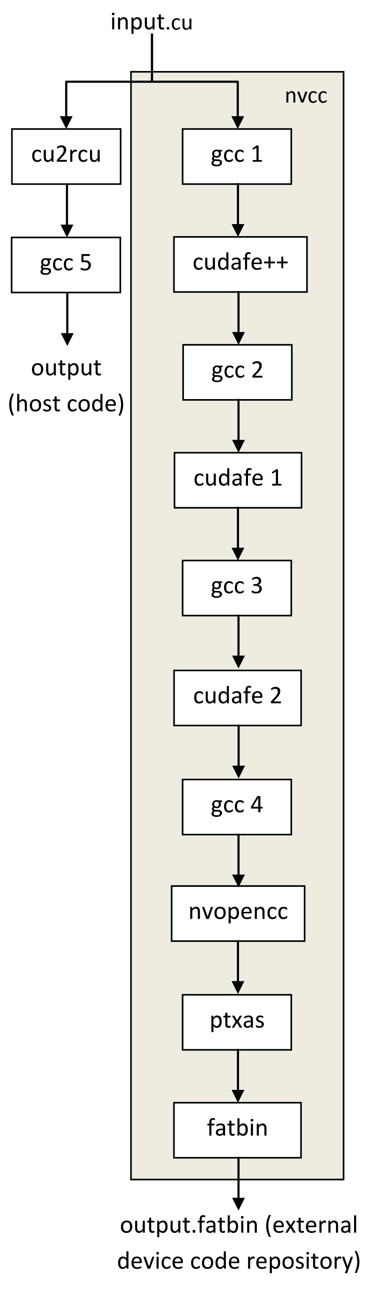


Figure 3.4: rCUDA compilation flow.

Step 3: Using CU2RCU

Once the plugin has been installed, we can start using it. We can run it from a terminal using the following command:

```
1 $ clang -cc1 \
2   -load $(LLVM_PATH)/Debug+Asserts/lib/libCU2RCU.so \
3   -plugin-arg-plain-CUDA <arguments> \
4   -plugin plain-CUDA \
5   -I /$(CUDA_INSTALL_PATH)/include/ \
6   -D __CUDACC__ <files_to_be_converted>
```

Where valid `<arguments>` in line 3 are:

- `‘help’`: prints help about the plugin.
- `‘verbose’`: prints modifications made by the plugin to the original source file.

The files to which apply the converter are specified in `<files_to_be_converted>`, line 6. `LLVM_PATH` (line 2) and `CUDA_INSTALL_PATH` (line 5) are respectively the paths to LLVM and CUDA installation directories.

Apart from the `CU2rCU` converter, an script has also been developed to simplify its deployment: `CU2RCU.bash`. It is also distributed with the rest of the converter software. As it will be shown next, it is only necessary to execute this script with the same files that we would compile with the NVIDIA `nvcc` compiler as parameters. Thus, the use of the converter becomes into invoking this script with files to be converting as arguments, that is:

```
$ bash CU2RCU.bash <files_to_be_converted>
```

Example Of Use

To illustrate the entire process in order to convert and compile a CUDA program, we are going to use a simple example. Initially, we have a CUDA program called `sample.cu` which is located at the following path: `‘‘home/Carpeta personal/sample’’`. In Figure 3.5 we can see the initial state.

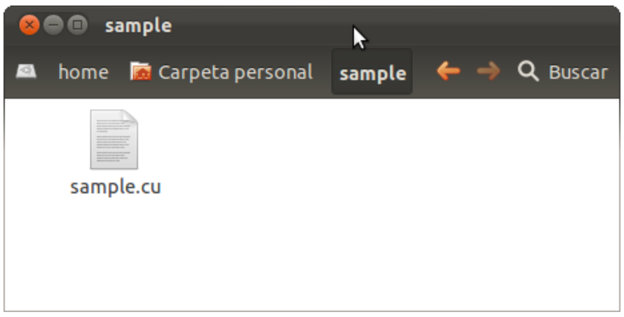


Figure 3.5: Initial state of file to be converted.

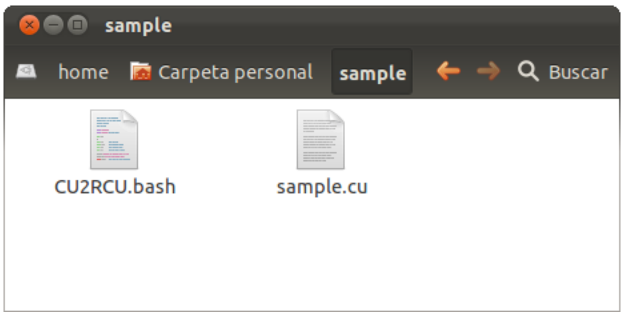
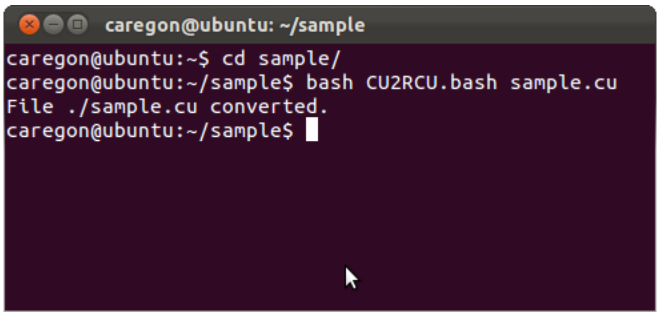


Figure 3.6: State after having copied the `CU2RCU.bash` script.

Next, we copy the script `CU2RCU.bash` which is distributed with the converter in that path, as shown in Figure 3.6.

Then, we open a terminal, move to the directory where our file is located, and execute the `CU2RCU.bash` script, which will print a message informing of the result of the conversion. In Figure 3.7 it can be seen the script output after a successful conversion.

Now, in the directory where our source file is located, it has been created a new directory called `‘‘CU2RCU’’`, as illustrated in Figure 3.8. Inside this new directory the new converted file has been created, which is called in the same way as the original file but with a new extension if needed (`‘‘.cu.cpp’’` for `‘‘.cu’’` files, and `‘‘.cuh.h’’` for the `‘‘.cuh’’` ones, the rest of file extensions do not change). Figure 3.9 shows the created file in our sample.



```
caregon@ubuntu:~$ cd sample/
caregon@ubuntu:~/sample$ bash CU2RCU.bash sample.cu
File ./sample.cu converted.
caregon@ubuntu:~/sample$ 
```

Figure 3.7: Execution of the `CU2RCU.bash` script for converting file `sample.cu`.

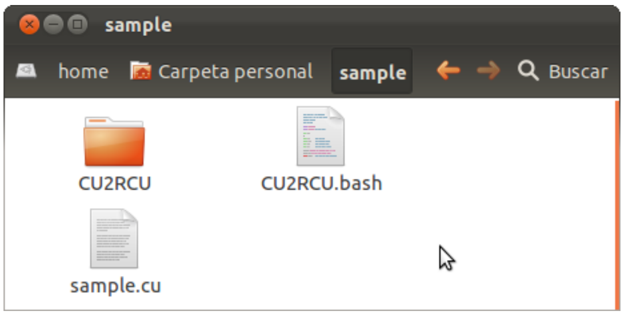


Figure 3.8: State after having run the `CU2RCU.bash` script.

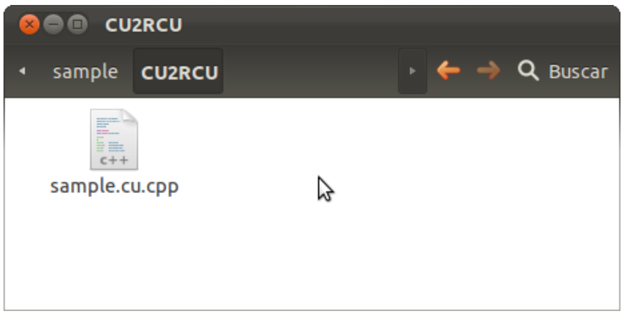


Figure 3.9: File created after having run the `CU2RCU.bash` script.

Chapter 4

Evaluation

This chapter describes the experiments carried out to evaluate the new `CU2rCU` tool. In order to test it, we have used sample codes from the NVIDIA GPU Computing SDK [21] (experiments shown next in Section 4.1) and production codes from the LAMMPS Molecular Dynamics Simulator [22] (tests presented below in Section 4.2).

4.1 NVIDIA GPU Computing SDK

Our first experiments dealt with a number of examples from the NVIDIA GPU Computing SDK, which contains simple code samples which covers a wide range of applications and techniques using CUDA. Table 4.1 shows the time required for their conversion. Notice that conversion 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%. This will also apply for compilation time shown later in this section.

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 4.1 also reports the amount of lines of the original application code, as well as the number of lines modified 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

Table 4.1: NVIDIA GPU Computing SDK Conversion Statistics

CUDA SDK Sample	Time (s)	Lines		
		CUDA Code	Modified/Added	
			No.	%
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

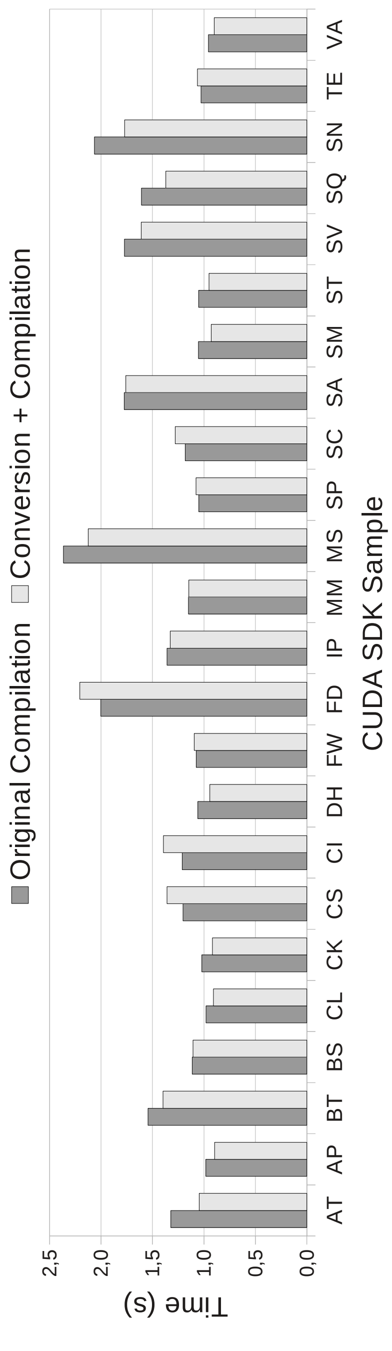


Figure 4.1: CUDA SDK compilation time compared with `CU2rCU` conversion plus compilation time.

benefits of using the converter.

Moreover, we have compared the time spent in the compilation of the original CUDA source code of the SDK samples with the time spent in their conversion and subsequent compilation of the converted code by our tool. Results are shown in Figure 4.1, demonstrating that the time of converting the original code and later compiling it is similar to the compilation time of the original sources. In Section 4.3 we will discuss these results.

4.2 LAMMPS Molecular Dynamics Simulator

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. 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 4.2 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 4.2, showing that they are similar. 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. In the next section we explain these results.

4.3 Compilation Time

After having tested the converter, we have noticed that the time of converting the original code with `CU2rCU` and later compiling it with `nvcc` and `gcc` is similar to the compilation time of the original sources with `nvcc`. This

Table 4.2: LAMMPS Conversion Statistics

LAMMPS Package	Time (s)	Lines		
		CUDA Code	Modified/Added	
			No.	%
USER-CUDA	6.910	14742	1409	9.56

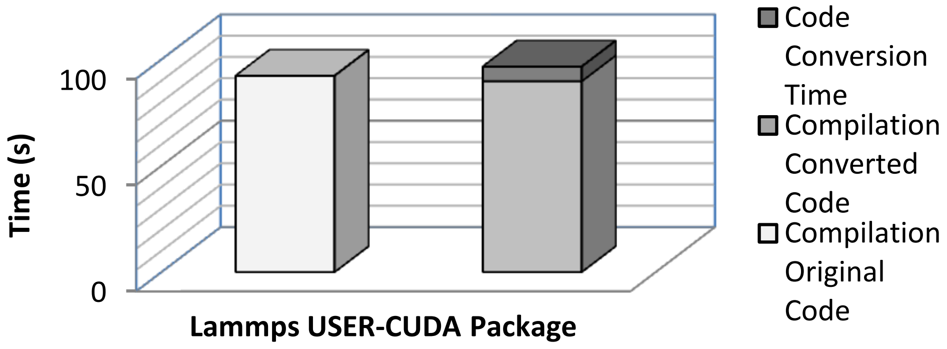


Figure 4.2: LAMMPS USER-CUDA `nvcc` compilation time compared with `CU2rCU` conversion plus compilation time.

section is intended to further analyze this fact.

In Section 3.4 we have seen the CUDA and the rCUDA compilation flows in Figures 3.3 and 3.4. Now, in Table 4.3 we present a comparison of the compilation phases in both compilation flows in terms of how many times each tool is used in each phase. The `CU2rCU` compilation phase could be seen as the `gcc` one because we are really calling to Clang, which is also a C compiler. Therefore, we could say 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 transformations, which also take time. In the rCUDA compilation flow, this time is compensated by (1) not requiring the `filehash` phase, present only in the CUDA flow, and by (2) the fact that Clang is quicker than GCC. That explains why the whole time spent is similar in both cases.

In order to clarify this, we have formulated it using mathematical ex-

Table 4.3: 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

pressions. Thus, Expression 4.1 represents the CUDA compilation flow and Expression 4.2 the rCUDA one.

$$\begin{aligned}\text{CUDA compilation} = {} & (6 \times \text{gcc}) + (2 \times \text{cudafe}) + \\ & (1 \times \text{cudafe++}) + (1 \times \text{filehash}) + \\ & (1 \times \text{nvopencc}) + (1 \times \text{ptxas}) + \\ & (1 \times \text{fatbin})\end{aligned} \tag{4.1}$$

$$\begin{aligned}\text{rCUDA compilation} &= (1 \times \text{cu2rcu}) + (5 \times \text{gcc}) + \\ & \quad (2 \times \text{cudafe}) + (1 \times \text{cudafe++}) + \\ & \quad (1 \times \text{nvopencc}) + (1 \times \text{ptxas}) + \\ & \quad (1 \times \text{fatbin}) \\ &=^* (transformations) + (6 \times \text{gcc}) + \\ & \quad (2 \times \text{cudafe}) + (1 \times \text{cudafe++}) + \\ & \quad (1 \times \text{nvopencc}) + (1 \times \text{ptxas}) + \\ & \quad (1 \times \text{fatbin}) \\ &\simeq \text{CUDA compilation} \\ \\ {}^*\text{cu2rcu} &= (transformations) + (1 \times \text{clang}) + \\ &\simeq (transformations) + (1 \times \text{gcc})\end{aligned} \tag{4.2}$$

Chapter 5

Conclusions

In this document we have provided an overview of the rCUDA framework, which enables the concurrent use of CUDA-compatible devices remotely. During this review, we have pointed out the following three main concerns of rCUDA:

- The usability of the rCUDA framework was limited.
- The use of remote GPUs in rCUDA reduces performance.
- rCUDA must evolve to support new CUDA versions.

As it was explained at the beginning of this document, the aim of this report was addressing the first one of these three concerns, that is, the usability of the rCUDA framework.

In the first place, we have explained that the usability of the rCUDA framework was initially limited by its lack of support for the CUDA C extensions. Thus, our framework only supported the plain CUDA C API, making necessary to rewrite those lines of the original application source files that make use of the CUDA C extensions.

To solve this limitation we have developed `CU2rCU`, a CUDA-to-rCUDA converter. `CU2rCU` is a complementary tool to rCUDA which enables leveraging rCUDA for any CUDA application. It automatically analyzes 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.

For performing the source code conversions we have used Clang, one of the primary sub-projects of LLVM. Clang 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.

Moreover, an script has also been developed to simplify the task of applying the `CU2rCU` converter, being only necessary to execute this script with the same files that we would compile with NVIDIA `nvcc` compiler as parameters.

In order to evaluate the new `CU2rCU` tool, we have used sample codes from the NVIDIA GPU Computing SDK and production codes from the LAMMPS Molecular Dynamics Simulator, obtaining in both cases successful results. The NVIDIA SDK contains simple code samples which cover a wide range of applications and techniques, while LAMMPS is a real-world production code with a considerable length. The large size of the LAMMPS codes proves the feasibility of our tool.

In addition to test the correct operation of our tool, we have also measured the time employed using it in combination with the rCUDA framework in order to achieve an executable, and compared this time with the one spent using the CUDA environment for the same purpose. We have seen that in both cases the time was similar and we have also analyzed the reasons.

As future work, it is planned that the `CU2rCU` tool is 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.

Bibliography

[1] NVIDIA, “The NVIDIA CUDA API Reference Manual”, NVIDIA, 2011.

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

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

[4] 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.

[5] 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.

[6] 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.

[7] 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.

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

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

[10] 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. 385–394.

[11] 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.

[12] 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.

[13] NVIDIA, "The NVIDIA CUDA C Programming Guide", NVIDIA, 2011.

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

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

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

[17] 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: June 2012.

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

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

[20] 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.

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

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

[23] Semantic Designs, Incorporated, "The DMS Software Reengineering Toolkit", online: http://www.semanticdesigns.com/Products/DMS/DMSToolkit.html, last access: June 2012.

[24] Q. Yi, "POET: a scripting language for applying parameterized source-to-source program transformations", in Software: Practice and Experience, vol. 42, pp. 675-706, 2012.

[25] G. Rudy, M. Khan, M. Hall, C. Chen, and J. Chame, "A Programming Language Interface to Describe Transformations and Code Generation", in Lecture Notes in Computer Science, vol. 6548, pp. 136-150, 2011.

[26] E. Visser, "Program Transformation with Stratego/XT", in Lecture Notes in Computer Science, vol. 3016, pp. 315-349, 2004.

[27] G. Necula, S. McPeak, S. Rahul, and W. Weimer, "CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs", in Lecture Notes in Computer Science, vol. 2304, pp. 209-265, 2002.

[28] LLVM, "The Clang Compiler User's Manual", online: http://clang.llvm.org/docs/UsersManual.html, last access: June 2012.

[29] LLVM, "The LLVM Programmer's Manual", online: http://llvm.org/docs/ProgrammersManual.html, last access: June 2012.