Document downloaded from:

http://hdl.handle.net/10251/70229

This paper must be cited as:

Reaño González, C.; Silla Jiménez, F. (2015). Reducing the Costs of Teaching CUDA in Laboratories while Maintaining the Learning Experience Quality. En INTED2015 Proceedings. IATED. 3651-3660. http://hdl.handle.net/10251/70229.

The final publication is available at

https://library.iated.org/view/REANO2015RED

Copyright IATED

Additional Information

REDUCING THE COSTS OF TEACHING CUDA IN LABORATORIES WHILE MAINTAINING THE LEARNING EXPERIENCE QUALITY

Carlos Reaño[1], Federico Silla[1]

[1]*Departament d'Informàtica de Sistemes i Computadors (DISCA)*
Escola Tècnica Superior d'Enginyeria Informàtica (ETSINF)
Universitat Politècnica de València (UPV, SPAIN)

Abstract

Graphics Processing Units (GPUs) have become widely used to accelerate scientific applications; therefore, it is important that Computer Science and Computer Engineering curricula include the fundamentals of parallel computing with GPUs. Regarding the practical part of the training, one important concern is how to introduce GPUs into a laboratory: installing GPUs in all the computers of the lab may not be affordable, while sharing a remote GPU server among several students may result in a poor learning experience because of its associated overhead.

In this paper we propose a solution to address this problem: the use of the rCUDA (remote CUDA) middleware, which enables programs being executed in a computer to make concurrent use of GPUs located in remote servers. Hence, students would be able to concurrently and transparently share a single remote GPU from their local machines in the laboratory without having to log into the remote server. In order to demonstrate that our proposal is feasible, we present results of a real scenario. The results show that the cost of the laboratory is noticeably reduced while the learning experience quality is maintained.

Keywords: CUDA, reducing teaching costs, teaching labs.

1 INTRODUCTION

Parallel computing has been traditionally included in Computer Science and Computer Engineering curricula in order to teach students how to address the challenges imposed by complex problems, which demand large amounts of computing resources that must collaborate to achieve high performance computing.

During the last years, Graphics Processing Units (GPUs) have become widely used to accelerate applications from areas as diverse as data analysis [1], chemical physics [2], image analysis [3], finance [4], algebra [5], computational fluid dynamics [6], etc. Therefore, it is important that Computer Science and Computer Engineering curricula include the fundamentals of parallel computing with GPUs. In this regard, although OpenCL [7] is an open standard that may be used to program GPUs, CUDA [8] (Compute Unified Device Architecture), the parallel computing architecture proposed by NVIDIA –the largest GPU manufacturer for the last years– is currently the most used GPU programming environment in the professional field, also achieving higher performance. These reasons may influence the decision of professors for teaching CUDA instead of the open standard OpenCL.

As regards the practical part of CUDA training, one important concern is how to introduce CUDA GPUs into a laboratory in an efficient way, mainly from an economic perspective, but also considering the learning quality. On the one hand, when building a CUDA laboratory, installing CUDA GPUs in all the computers of the lab may not be affordable in terms of the economic cost of this approach, given the price of these cards. On the other hand, the opposite approach consists of requesting students to log into a remote GPU server, which may be located in the same lab or in a computer room somewhere in the university. However, this option may result in a poor learning experience quality because of its associated overhead in terms of time: all the students starting graphical sessions in the

server in order to use visual programming environments, all the students consuming the server main memory, additional CPU overhead when compiling and executing the test programs in the server, etc.

In this paper we propose a solution to efficiently address the introduction of GPUs into a teaching lab. Our proposal is based on the use of the rCUDA [9] [10] (remote CUDA) middleware. rCUDA enables programs being executed in a computer to make concurrent use of GPUs located in remote servers. Hence, students would be able to concurrently and transparently share one or more remote GPUs from their local machines in the laboratory without having to log into the remote server. In this way, students would use the computer at their workplace to load the visual programming environment and to develop and compile their programs. Thus, the remote server offering the GPU services will not be overloaded with these tasks. Moreover, the practical exercises coded during the lab session would also be executed at the workplace computer and rCUDA would transparently execute the part of the program not requiring the GPU (i.e., the CPU part) in the student's computer, while the part of the program actually demanding the intervention of the GPU would be run in the remote server owning the GPU (this will be transparently done by rCUDA without the students' intervention). In this manner, the remote server would not be overloaded by the CPU parts of students' programs. In addition, rCUDA is fully compatible with CUDA, so that CUDA programs do not need to be modified and thus students will still learn only CUDA without having to worry about rCUDA, which would be transparent to students.

The rest of the paper is organized as follows. In Section 2 we present in more detail rCUDA. In Section 3 we compare different ways for introducing GPUs in teaching laboratories, presenting results of a real scenario in order to demonstrate that our proposal is feasible. Finally, Section 4 summarizes the main conclusions of our work.

2 rCUDA: REMOTE CUDA

As mentioned before, CUDA is a technology created by NVIDIA which provides a parallel computing platform and programming model to be used along with NVIDIA GPUs or compatible ones. CUDA takes benefit from the great computational power of GPUs to accelerate certain parts of applications, thus reducing their execution time. Nevertheless, it is the programmer who decides which parts of the application are executed in the traditional CPU and which parts are offloaded to the GPU. This decision depends, basically, on the level of parallelization of different parts of the application.

rCUDA [9] [10] (remote CUDA) is a software framework which enables sharing remote CUDA compatible devices in a way that is transparent to the programmer. In this manner, a GPU installed in one computer of a network (the server providing GPU services) can be concurrently used by other computers of the network (the clients demanding acceleration services) to accelerate applications using CUDA, as depicted in Fig. 1. rCUDA grants applications transparent access to GPUs installed in remote computers, so that they are not aware of being accessing a remote device.

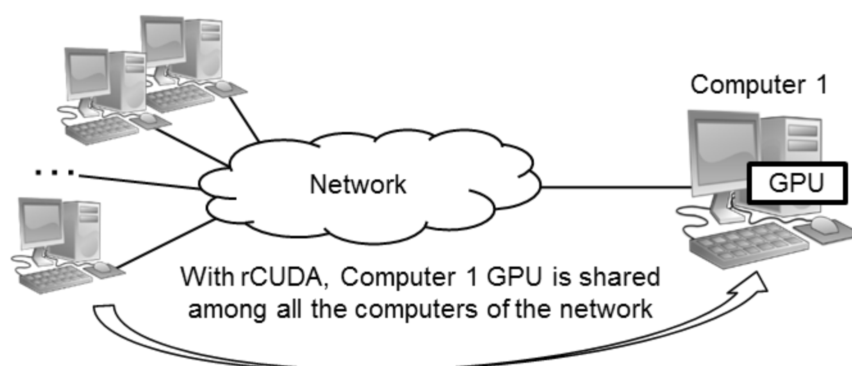

Fig. 1. rCUDA sample scenario.

Fig. 2 presents the architecture of the rCUDA framework. As we can see, it is organized following a client-server distributed architecture. When an application demands GPU services, the rCUDA client forwards the request to the server side by means of the network. Notice that the application continues using the same interface (i.e., the original CUDA API) as a regular CUDA program. Therefore, no modifications are required in the application. The way to achieve this is by dynamically replacing at

runtime the CUDA library by the rCUDA one, which presents the very same interface, as commented. Thus, when the application calls a CUDA function, it will actually execute the corresponding function within the rCUDA library, which will forward the call to the remote server.

Once the rCUDA server receives the client request, it is executed in the real GPU. Upon completion of the GPU task, the rCUDA server forwards the corresponding response to the client and, finally, the rCUDA client sends on the result to the application initially requesting acceleration services from CUDA. Notice that the application will not be aware of being accessing a remote GPU. All the process is automatic and transparent to the application.

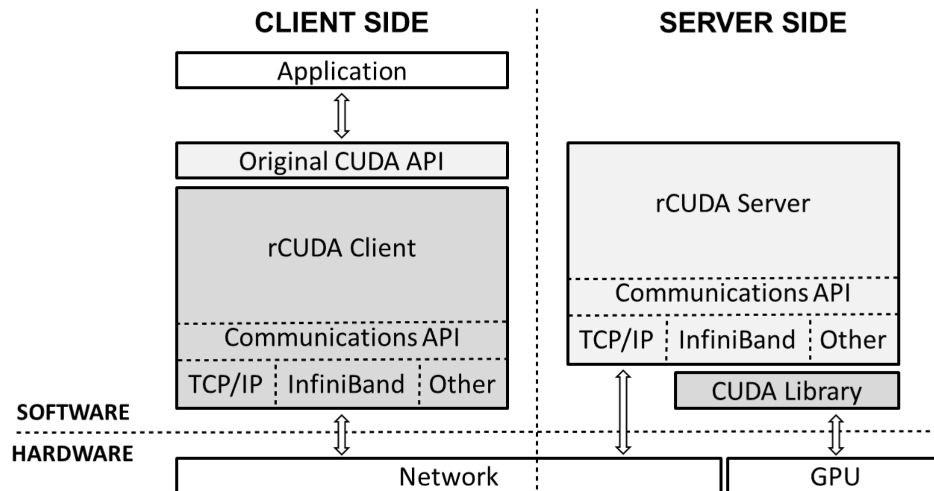

Fig. 2. rCUDA architecture.

Communication between the rCUDA client and the rCUDA server is done via a customized communication protocol that uses the network available between the computer where the application is running and the computer where the physical GPU is located. Currently, rCUDA provides two different implementations of the communications protocol: (1) one optimized for InfiniBand networks, which employs InfiniBand Verbs and is intended to be used in High Performance Computing (HPC) clusters; and (2) a generic version using TCP sockets, which is supported by almost every network, intended for those environments in which performance is not so crucial, such as, for example, when using rCUDA for teaching purposes. In these scenarios, the widely available Ethernet network would be used, thus incurring in no additional cost.

The last version of rCUDA, available at http://rcuda.net/, supports CUDA 6.5 Runtime API [8] and Driver API [11]. It also offers support for some routines of the most common CUDA Libraries, such as cuBLAS [12], cuFFT [13], cuRAND [14] and cuSPARSE [15]. Furthermore, the rCUDA middleware is distributed at no cost, thus allowing an inexpensive introduction of this technology.

In order to better understand how a CUDA program is executed when using the rCUDA framework, in Fig. 3 we present a sequence diagram of a typical CUDA program executed with rCUDA. First of all, when the program is loaded, the operating system will automatically replace the use of the CUDA library by the rCUDA one. This is easily done by properly setting an environment variable. The only requirement is that the application is compiled using dynamic libraries instead of static ones. Once the program starts its execution, rCUDA automatically creates a new connection to the remote server owning the GPU. In second place, the program wants to allocate memory in the GPU. rCUDA intercepts the CUDA call that initiates such memory allocation and forwards it to the remote server, which allocates memory in the remote GPU transparently to the program, which is not aware of being using a remote GPU. Third, the input data is copied from system memory in the node executing the application (the client) to the GPU memory in the remote server. This is transparently done by rCUDA, which copies the data to the remote GPU memory. Next, after some computations (i.e., kernel execution) are done in the remote GPU managed by rCUDA, the accelerated application executes a cudaMemcpy function to copy the results from GPU memory to main memory. This call is actually executed by rCUDA and the results are copied back from the remote GPU memory to the local system memory without the program intervention. Finally, the application is programmed to free the memory in

the GPU. Again, this call is attended by the rCUDA library, which forwards it to the remote server so that the remote GPU memory is freed and, when the program exits, the rCUDA server connection is closed.

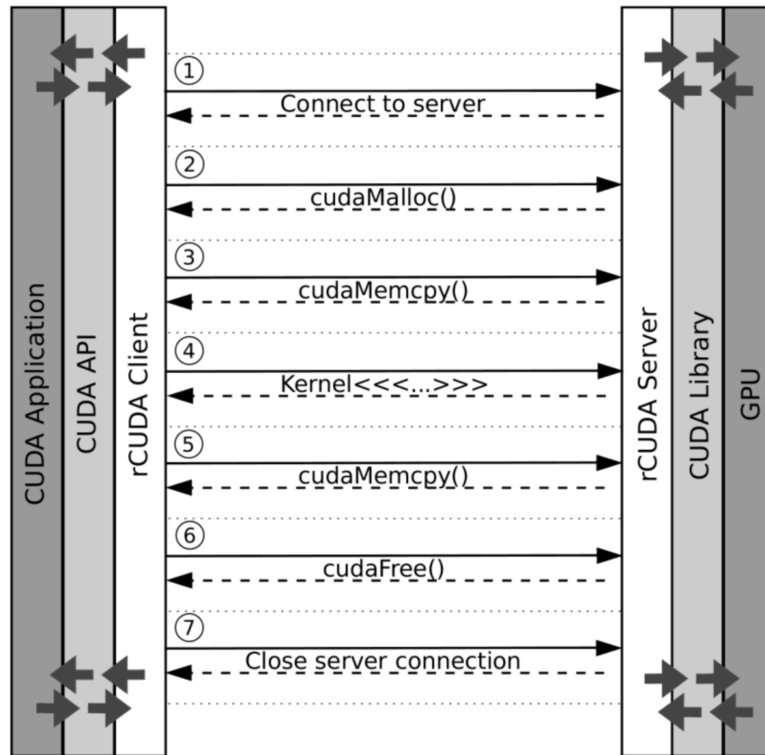

Fig. 3. Sequence diagram of a typical CUDA program executed with rCUDA: (1) connect to rCUDA server, (2) allocate memory in the remote GPU, (3) copy input data from local CPU memory to remote GPU memory, (4) kernel execution in the remote GPU, (5) copy output data from remote GPU memory to local CPU memory, (6) free memory in the remote GPU, (7) close connection.

3 INTRODUCING GPUS IN TEACHING LABORATORIES

As commented in the introduction, installing CUDA GPUs in all the computers of a laboratory may not be affordable in terms of the economic cost of this approach. In this section, we expose different ways of introducing GPUs in teaching laboratories, comparing their cost and commenting on the advantages and disadvantages that they present.

3.1 The importance of the GPU to achieve a good learning experience quality

From our point of view, it is very important that the students realize of the benefits of parallel computing with GPUs, such as the important reduction of execution time, in order to motivate them making the effort of learning a new programming paradigm. Consequently, the GPUs used in the laboratory must clearly outperform the CPUs to really appreciate the performance benefits of GPUs. If not, the students will not be motivated and they will not take as much profit as possible from the time spent in the lab. In this section, we show the importance of the GPU to encourage students and have, therefore, a good learning experience quality.

In Table I we show the price of three different NVIDIA desktop GPUs currently available in the market. It can be seen that the cost of GPUs varies in a very wide range, which mainly depends on the computing capabilities of the GPUs as well as on the amount of GPU memory. We will use these GPUs in next experiments.

Although Table I compares the three GPUs from an economical point of view, it is also necessary to consider their computing power to have the full picture. In order to do so, in Fig. 4 we present results of the matrixMul program from the NVIDIA CUDA Samples 6.5 [19], which performs a matrix

multiplication in the GPU. This program has been selected because it illustrates various CUDA programming principles and, from our point of view, it is a good starting point for encouraging the attendees of a CUDA course. The program was run with the 3 different GPUs previously exposed in Table I. In addition, results for a similar matrix multiplication performed without GPU, using the well-known GotoBLAS2 [20] library, is included for reference.

Table I. NVIDIA Desktop GPUs comparison.

Graphic Card	Cost
NVIDIA GeForce GT 520 [16]	100€
NVIDIA GeForce GTX 590 [17]	300€
NVIDIA GeForce GTX 780 Ti [18]	600€

As it can be seen in Fig. 4, using a regular desktop GPU, like a GeForce GT 520, may not be enough to encourage students to learn a new programming paradigm, because better results can be obtained using a CPU. Actually, using such a GPU would completely demotivate the students, thus making the rest of the CUDA course harder to attend. For that reason we believe that, when teaching parallel programming with GPUs, it is important that results with GPUs clearly show the gain of this model when compared to the traditional one based on the use of CPUs. For instance, using more advanced GPUs, like the GeForce GTX 590 or GTX 780 Ti showed in Table I, the profit of parallel computing with GPUs is obvious. Therefore, this technology will be more appealing to the students, and the learning experience will be, in consequence, better. In any case, from our point of view, the higher the performance of the GPU is, the better would be the learning experience. This is why we use the GTX 780 Ti in next sections.

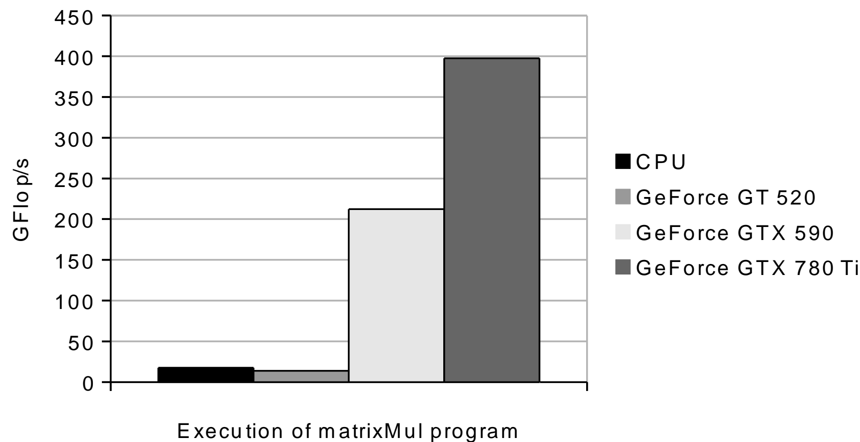

Fig. 4. Execution of matrixMul program from NVIDIA CUDA Samples with 3 different GPUs compared to a matrix multiplication executed in the CPU using GotoBLAS2 library.

3.2 Installing CUDA GPUs in all the computers of the laboratory

Obviously, having a CUDA GPU installed in each computer of the laboratory should be the most desirable configuration in terms of both performance and learning experience quality, especially when compared to a configuration where the students log into a shared GPU server and all the students use this single server during the entire lab session. For that reason, we will use this approach as the reference case.

In Table II we present the cost of a laboratory composed of 20 computers, each one with an NVIDIA GeForce GTX 780 Ti. Although this GPU is initially intended for gaming, it offers a very good performance for a teaching lab, as shown in the previous section, at less price than other GPUs specifically manufactured for parallel computation, such as for example the NVIDIA Tesla K20, which costs more than 2,000€. The cost shown in Table II includes a commodity computer with a current typical configuration.

Table II. Cost of a laboratory composed of 20 computers with GPUs.

Component	Cost
Intel Core I3-3220 3,30GHz with 4GB RAM	20 x 300€
NVIDIA GeForce GTX 780 Ti	20 x 600€
TOTAL	18,000€

Notice that we have excluded in Table II the network expenses because we assume that current laboratories are compulsory connected to an Ethernet network, and therefore the cost of the network will represent a constant value, not therefore modifying the cost of the different proposals described in this paper.

3.3 Logging into a remote GPU server

In a first attempt to reduce the cost of building a CUDA teaching laboratory, one possible solution is having one single GPU server so that the students log into this server remotely from their different workplaces in the lab. With this approach, we will avoid installing a GPU in all the computers, and the total cost of the laboratory will be similar to the one presented in Table III. In this table, we present the cost of a laboratory composed of 20 computers, and one additional GPU server with an NVIDIA GeForce GTX 780 Ti. Notice that the hardware configuration of the GPU server is noticeably better than that of the computers of the students' workplaces, since this server will have to host all the graphical environments of the different students, as well as their programming tools.

Table III. Cost of a laboratory composed of 20 computers and 1 GPU server.

Component	Cost
Intel Core i3-3220 3.30GHz with 4GB RAM	20 x 300€
Intel Core i7-4790 3.6Ghz with 32GB RAM	1 x 1000€
NVIDIA GeForce GTX 780 Ti	1 x 600€
TOTAL	7.600€

As it can be seen, the total cost of the laboratory is noticeably reduced, but this approach may result in a poor learning experience because of its associated overhead: all the students starting graphical sessions with the server in order to use visual programming environments, all the students consuming the server main memory, additional CPU overhead when compiling and executing the test programs in

the server, etc. Next we present some experiments using a similar laboratory to the one exposed in Table III.

For instance, in Fig. 5 we show how the compilation time of the matrixMul program from the NVIDIA CUDA Samples, used in previous sections, increases with the amount of concurrent users compiling it at the same time. Of course, having all the students compiling a program at the same time will be the worst case. Usually, there is not such level of concurrency in the laboratories, while having 5-10 users out of 20 may be the common scenario.

Results in Fig. 5 show the average compilation time for each user. As we can see, the time is increased from 10 seconds when there is only one user compiling the program, to 40 seconds when the 20 students are compiling the program at the same time. Taking into account that when students are learning compilations are very frequent, this waiting time will definitely worsen the learning experience. Notice also that the processors of the computers in the students' workplaces remain basically unused, thus not amortizing their cost.

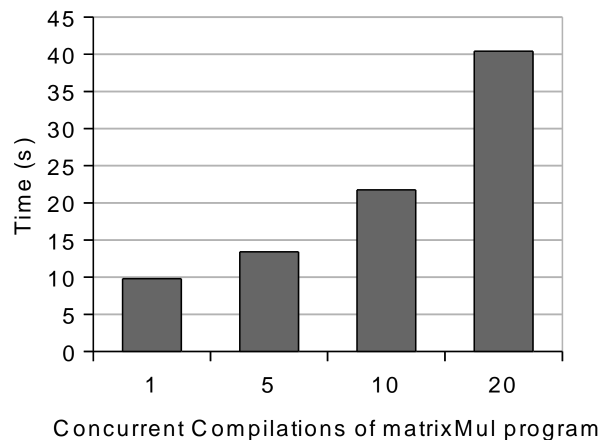

Fig. 5. Average compilation time of matrixMul program when one or more users are logged into the GPU server compiling this program at the same time.

Another disadvantage of this approach is the performance decrease when several students are running programs that use the GPU. For example, in Fig. 6 we present the average throughput of the matrixMul program when several users are running it at the same time. Again, the results are considerably worsened when compared to the execution of just one instance of the program. In this manner, GPU performance is reduced by a factor of over 10 when 20 students are concurrently using it with respect to the performance of a single student making use of the GPU. This low performance will also deteriorate the learning experience.

3.4 Using rCUDA

The approach presented in the previous subsection considerably reduces the cost of the teaching laboratory, but also affects the learning experience. In this subsection we propose a solution with the same cost as the one showed in previous subsection 3.3, but maintaining the learning experience of the approach shown in subsection 3.2.

Our proposal is based on the use of the rCUDA (remote CUDA) middleware. As explained in Section 2, rCUDA enables programs being executed in a computer to make concurrent use of GPUs located in remote servers. Hence, students would be able to concurrently share a single remote GPU from their local machines in the laboratory without having to log into the remote server. In this way, students would use the computer at their workplace to load the visual programming environment and to develop and compile their programs. Thus, the remote server offering the GPU services will not be overloaded with these tasks. Moreover, the exercises coded during the lab session would also be executed at the workplace computer and rCUDA would transparently execute the part of the program not requiring the GPU (i.e., the CPU part) in the student's computer, while the part of the program actually demanding the intervention of the GPU would be run in the remote server owning the GPU. In this manner, the remote server would not be overloaded by the CPU parts of students' programs.

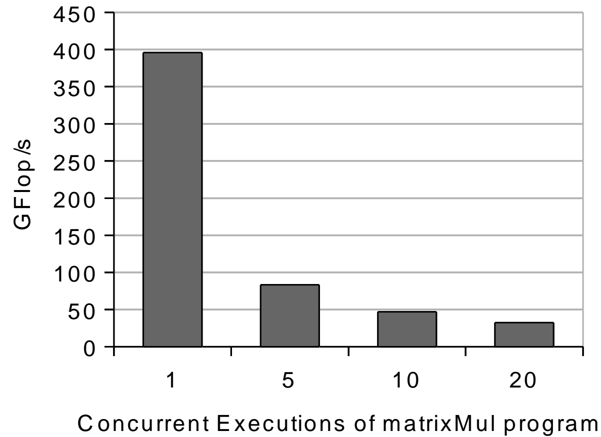

Fig. 6. Average execution time of matrixMul program when one or more users are logged into the GPU server executing this program at the same time.

Fig. 7 shows the same experiment performed in Fig. 6 (with the same equipment exposed in Table III), but using rCUDA instead of logging into a remote server. As we can see, in this case the performance of using a remote GPU (bars labeled as rCUDA in the chart) is slightly reduced in comparison to the performance of a local GPU (black line labeled as CUDA). However, it still clearly outperforms the CPU results shown in Fig. 4. Furthermore, the performance is maintained independently of the amount of concurrent executions. In this manner, the cost of the teaching laboratory has been reduced over half of the initial cost exposed in Table II, while the learning experience quality is maintained.

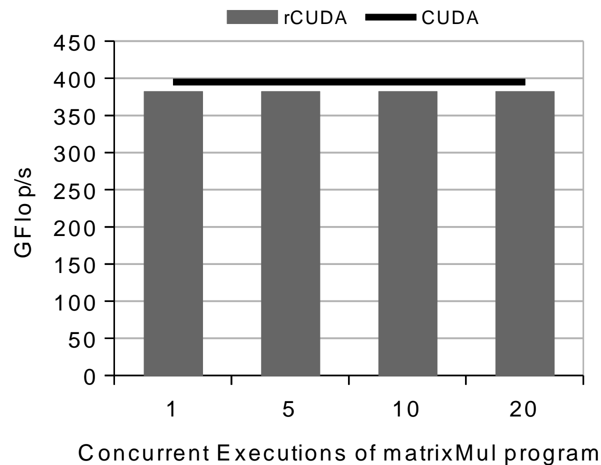

Fig. 7. Average execution time of matrixMul program when one or more users are using rCUDA and the GPU server executing this program at the same time.

4 CONCLUSIONS

In this paper we have proposed a solution to efficiently address the introduction of GPUs into a teaching laboratory. Our proposal is based on the use of the rCUDA (remote CUDA) middleware. rCUDA enables programs being executed in a computer to make concurrent use of GPUs located in remote servers. Hence, students are able to concurrently and transparently share a single remote GPU from their local machines in the laboratory without having to log into a remote server.

We have exposed three different ways to introduce GPUs in the teaching lab: (1) installing GPUs in all the computers, (2) logging into a remote GPU server, and (3) sharing the GPU of a remote server using rCUDA. In order to compare the three different approaches, we have presented results of a real

scenario: the experiments were carried out in a teaching laboratory with 20 computers. In addition, one GPU server was used for approaches 2 and 3.

Our study shows that the approach consisting of installing GPUs in all the computers provides the best learning experience quality, but its cost may not be affordable to some institutions. For the purpose of reducing the cost, the strategy of not having GPUs in the computers but logging instead into a remote GPU server reduces the expenses by more than half. However, this methodology also deteriorates the learning experience quality. Finally, the proposal based on the use of rCUDA achieves both goals: the cost of the laboratory is noticeably reduced while the learning experience quality is maintained.

REFERENCES

[1] Haicheng Wu, Gregory Diamos, Tim Sheard, Molham Aref, Sean Baxter, Michael Garland, and Sudhakar Yalamanchili. 2014. Red Fox: An Execution Environment for Relational Query Processing on GPUs. In Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO '14). ACM, New York, NY, USA, Article 44, 11 pages. DOI:http://dx.doi.org/10.1145/2544137.2544166

[2] D.P. Playne and K.A. Hawick. 2009. Data Parallel Three-Dimensional Cahn-Hilliard Field Equation Simulation on GPUs with CUDA. In Proc. 2009 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'09). WorldComp, Las Vegas, USA, 104–110.

[3] Yuancheng Luo and R. Duraiswami. 2008. Canny edge detection on NVIDIA CUDA. In Computer Vision and Pattern Recognition Workshops, 2008. CVPRW '08. IEEE Computer Society Conference on. 1–8. DOI:http://dx.doi.org/10.1109/CVPRW.2008.4563088

[4] Abhijeet Gaikwad and Ioane Muni Toke. 2009. GPU Based Sparse Grid Technique for Solving Multidimensional Options Pricing PDEs. In Proceedings of the 2Nd Workshop on High Performance Computational Finance (WHPCF '09). ACM, New York, NY, USA, Article 6, 9 pages. DOI:http://dx.doi.org/10.1145/1645413.1645419

[5] Sergio Barrachina, Maribel Castillo, Francisco D. Igual, Rafael Mayo, Enrique S. Quintana-Ortí, and Gregorio Quintana-Ortí. 2009. Exploiting the capabilities of modern GPUs for dense matrix computations. Concurr. Comput. : Pract. Exper. 21, 18 (December 2009), 2457-2477. DOI=10.1002/cpe.v21:18 http://dx.doi.org/10.1002/cpe.v21:18

[6] Kyle E. Niemeyer and Chih-Jen Sung. 2014. Recent progress and challenges in exploiting graphics processors in computational fluid dynamics. J. Supercomput. 67, 2 (February 2014), 528-564. DOI=10.1007/s11227-013-1015-7 http://dx.doi.org/10.1007/s11227-013-1015-7

[7] Khronos OpenCL Working Group. 2013. OpenCL 2.0 Specification.

[8] NVIDIA. 2014. CUDA API Reference Manual 6.5.

[9] Carlos Reaño, Rafael Mayo, Enrique S. Quintana-Ortí, Federico Silla, José Duato, and Antonio J. Peña. 2013. Influence of InfiniBand FDR on the performance of remote GPU virtualization. Cluster Computing (CLUSTER), 2013 IEEE International Conference on , vol., no., pp.1–8. DOI:10.1109/CLUSTER.2013.6702662

[10] Antonio J. Peña, Carlos Reaño, Federico Silla, Rafael Mayo, Enrique S. Quintana-Ortí, and José Duato. 2014. A complete and efficient CUDA-sharing solution for HPC clusters. Parallel Comput. 40, 10 (2014), 574 – 588. DOI:http://dx.doi.org/10.1016/j.parco.2014.09.011

[11] NVIDIA. 2014. CUDA Driver API 6.5.

[12] NVIDIA. 2014. CUBLAS Library 6.5.

[13] NVIDIA. 2014. CUFFT Library 6.5.

[14] NVIDIA. 2014. CURAND Library 6.5.

[15] NVIDIA. 2014. CUSPARSE Library 6.5.

[16] NVIDIA. GeForce GT 520. Available online: http://www.geforce.com/hardware/desktop-gpus/geforce-gt-520/specifications. Last accessed: January, 2015.

[17] NVIDIA. GeForce GTX 590. Available online: http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-590/specifications. Last accessed: January, 2015.

[18] NVIDIA. GeForce GTX 780 Ti. Available online: http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-780-ti/specifications. Last accessed: January, 2015.

[19] NVIDIA. 2014. CUDA Samples Reference Manual 6.5.

[20] Texas Advanced Computing Center (TACC), The University of Texas at Austin. GotoBLAS2 Library 1.13. Available online: https://www.tacc.utexas.edu/research-development/tacc-software/gotoblas2. Last accessed: January, 2015.