ON THE DEPLOYMENT AND CHARACTERIZATION OF CUDA TEACHING LABORATORIES

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

When teaching CUDA in laboratories, an important issue is the economic cost of GPUs, which may prevent some universities from building large enough labs to teach CUDA. In this paper we propose an efficient solution to build CUDA labs reducing the number of GPUs. It is based on the use of the rCUDA (remote CUDA) middleware, which enables programs being executed in a computer to concurrently use GPUs located in remote servers. To study the viability of our proposal, we first characterize the use of GPUs in this kind of labs with statistics taken from real users, and then present results of sharing GPUs in a real teaching lab. The experiments validate the feasibility of our proposal, showing an overhead under 5% with respect to having a GPU at each of the students' computers. These results clearly improve alternative approaches, such as logging into remote GPU servers, which presents an overhead about 30%.

Keywords: Teaching, laboratories, CUDA.

1 INTRODUCTION

The remarkable increment in the use of Graphics Processing Units (GPUs) experienced during the last years has deeply changed the way in which high performance computing is addressed. In this regard, many supercomputers and data centers currently make use of these accelerators in order to reduce the execution time of their workloads.

The pervasive use of GPUs in current computing facilities makes necessary that their architecture and programming are included in modern Computer Engineering and Computer Science (CECS) curricula, so that students receive the proper training for their later job career. Thus, in the same way as parallel computing has been traditionally taught in CECS schools, now it is necessary to introduce theoretical lectures and lab sessions aimed to provide students the required knowledge about these accelerators.

An important concern about introducing GPU contents in the mentioned curricula is that there are currently two main trends to program GPUs: OpenCL [1], which is an open standard, and CUDA [2], the parallel computing architecture proposed by NVIDIA –the largest GPU manufacturer. Nevertheless, although OpenCL is an open standard and CUDA is proprietary by NVIDIA, the latter is currently the most used one in the professional field, also achieving higher performance. These reasons may influence the decision of professors for teaching CUDA instead of OpenCL.

Regarding the lab sessions that should be included in any course about CUDA, an important issue is the economic cost of GPUs, which may prevent some universities from building large enough labs to teach CUDA, so that the learning experience of students is satisfactory, thus improving their training and qualifying them for the best job opportunities, what later translates into a higher recognition of the university that trained them. Regarding the economic cost of a CUDA lab, the straightforward approach would be to install a CUDA GPU in each of the computers of the lab, what may not be affordable in terms of the economic cost of this approach. A cheaper approach would be to request students to log into a remote server containing a GPU. However, although this option is noticeably cheaper than the previous one, it may result in a poor learning experience due to the saturation that the remote server would experience because of several reasons, like all the graphical sessions started by students in order to use visual programming environments, the high CPU utilization when compiling and executing the test programs in the server, etc.

In this work we propose an efficient solution to build CUDA labs, which is based on the use of the rCUDA (remote CUDA) middleware [3, 4]. This framework enables programs being executed in a computer to concurrently use GPUs located in remote servers. Therefore, students would be able to

simultaneously share one remote GPU from their local computers in the lab without having to log into the remote server, thus avoiding the server saturation mentioned before at the same time that the cost of the lab is still noticeably reduced.

The rest of the paper is organized as follows. In Section 2 we present in more detail rCUDA. In Section 3 we characterize the use of GPUs in this kind of labs with statistics taken from real users. Section 4 explains the reasons for sharing GPUs in CUDA teaching laboratories and presents several ways for doing so. In Section 5 we present some experiments sharing GPUs in a real teaching lab, taking into account the results of the previous characterization. Finally, Section 6 summarizes the main conclusions of our work.

2 rCUDA: REMOTE CUDA

During the last years, GPUs have become widely used to accelerate applications from areas as diverse as data analysis [5], chemical physics [6], image analysis [7], finance [8], algebra [9], computational fluid dynamics [10], etc. As mentioned before, there are currently two main trends to program GPUs: OpenCL, which is an open standard, and CUDA, the parallel computing architecture proposed by NVIDIA. Given that the latter is currently the most used one in the professional field, in this paper we will focus on CUDA.

CUDA is an extension of the well-known C programming language. In order to better understand how it works, we show next a simple CUDA program. As we can see, it is very similar to a regular C program. The main difference is that some parts of this program will be executed in the GPU (i.e. the code labelled as “GPU code”):

```
#include <stdio.h>
const int N = 8;

// Function which will be executed in the GPU
  __global__
void my_gpu_function(int *a, int *b)
{
  b[threadIdx.x] = a[threadIdx.x] * 2;
}

int main()
{
  int a[N] = {0, 1, 2, 3, 4, 5, 6, 7};
  int *ad, *bd;
  const int isize = N*sizeof(int);

  // Allocate GPU memory
  cudaMalloc( (void**)&ad, isize );

  // Copy data to GPU memory
  cudaMemcpy( ad, a, isize, cudaMemcpyHostToDevice );

  // Run function in the GPU
  my_gpu_function<<<1, N>>>(ad, bd);

  // Copy results from GPU memory
  cudaMemcpy( b, bd, isize, cudaMemcpyDeviceToHost );

  // Free GPU memory
  cudaFree( ad );
  cudaFree( bd );

  return 0;
}
```

GPU code

rCUDA (remote CUDA) is a middleware which enables programs being executed in a computer to use GPUs located in remote servers. Continuing the example program above, when using CUDA the GPU code will be executed in a local GPU attached to the computer, while the rest of the program will run in the computer CPU, as usual. On the other hand, when using rCUDA, the non-GPU code will run also in the computer CPU. However, the GPU code will be executed in a remote GPU which is not attached to the computer, but located in a different computer. For doing so, rCUDA utilizes the

network between both computers to transparently transfer and run the GPU code in the remote GPU computer. The user is not aware that the GPU code of his/her program is running in a remote GPU. rCUDA manages everything and, from the point of view of the end user, it is as if he/she were using regular CUDA with a local GPU, instead of rCUDA with a remote GPU. For the sake of clarity, the differences between CUDA and rCUDA explained here are also graphically illustrated in Fig. 1.

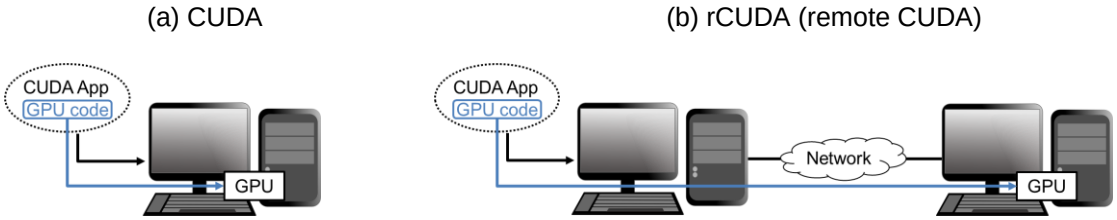

Fig. 1. Graphical illustration of a CUDA application using (a) CUDA or (b) rCUDA.

The last version of rCUDA is freely available at www.rcuda.net, supports CUDA 6.5 Runtime API [2] 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 into CUDA teaching laboratories.

3 CHARACTERIZING CUDA TEACHING LABORATORIES

In this section we characterize the use of GPUs in a CUDA teaching laboratory with statistics taken from a real scenario. The experiments were done in a laboratory composed of 20 computers, as shown in Fig. 2.

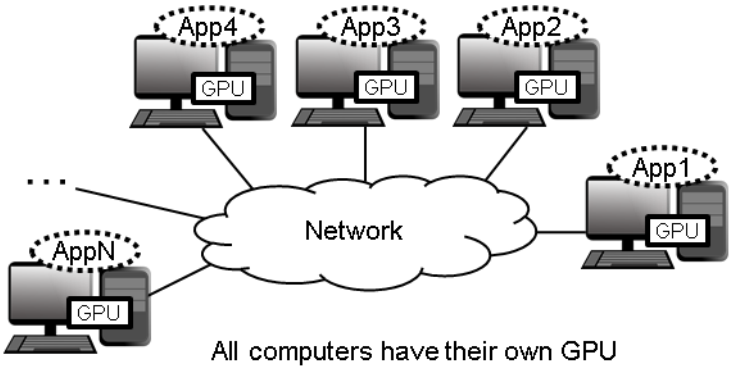

Fig. 2. CUDA teaching laboratory composed of 20 computers, each one with a CUDA GPU.

The metrics used for characterizing the utilization of the GPUs were:

- Number of compilations: compiling a program does not require the use of the GPU
- Number of executions: executing a program requires the use of the GPU

Fig. 3 presents the results of this experiment. The figure shows the number of compilations and the number of executions in a CUDA teaching laboratory composed of 20 computers, similar to the one

depicted in Fig. 2. The session was 90 minutes long and the measurements show the number of compilations/executions grouped in intervals of 5 minutes.

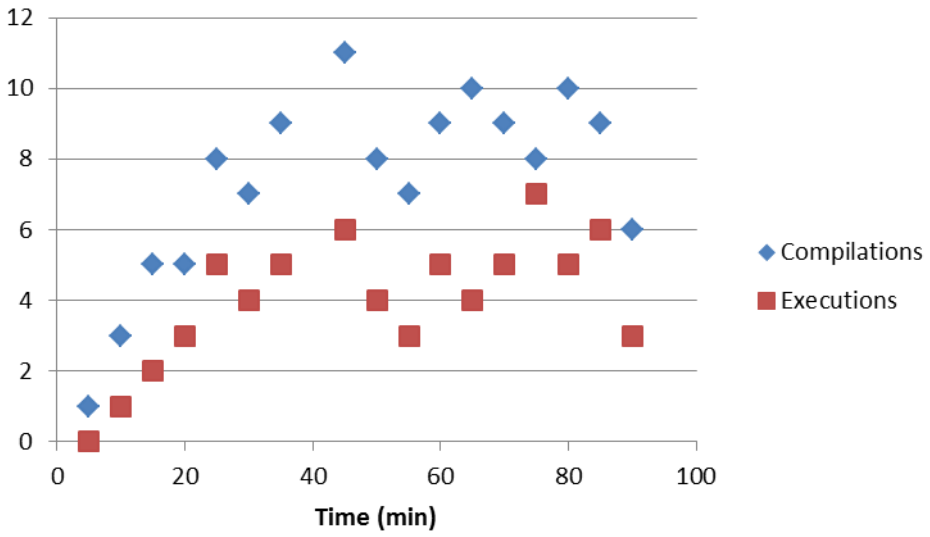

Fig. 3. Number of compilations and number of executions in a CUDA teaching laboratory of 20 students using 20 different computers. The course was 90 minutes long and the measurements show the number of compilations/executions grouped in intervals of 5 minutes.

As we can see, at the beginning of the session the number of compilations and executions is very low. The reason is that students are usually reading the exercises to be done. As the session progresses, the number of compilations and executions increase, as expected. Finally, at the end of the session it decreases again, because some of the students have already finished the exercises.

The conclusion that we can extract from this experiment is that the average number of executions for each 5 minute interval is approximately 5. Therefore, we could say that a maximum of 5 GPUs are, in theory, required at any given time. We will refer to this result in next sections.

4 SHARING GPUs IN CUDA TEACHING LABORATORIES

4.1.1 Why sharing GPUs

The main reason for sharing GPUs in CUDA teaching laboratories is the cost of CUDA GPUs. In this manner, installing CUDA GPUs in all the computers of a laboratory may not be affordable in terms of the economic cost of this approach. For instance, the cost of a laboratory as the one shown in Fig. 2. could be similar to the one presented in Table I.

Table I. 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 [16]	20 x 600€
TOTAL	18,000€

Table II. 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 [16]	1 x 600€
TOTAL	7.600€

4.1.2 How sharing GPUs

There are mainly two ways [17] of sharing a GPU among the computers of a CUDA teaching laboratory:

- Logging into a remote GPU server (see Fig. 4). This approach consists in requesting students to log into a remote server containing a GPU. However, although this option noticeably reduces the cost of installing one GPU in each computer, it may result in a poor learning experience due to the saturation that the remote server would experience because of several reasons [17], like all the graphical sessions started by students in order to use visual programming environments, the high CPU utilization when compiling and executing the test programs in the server, etc.
- Using the rCUDA middleware (see Fig. 5). This framework enables programs being executed in a computer to concurrently use GPUs located in remote servers. Therefore, students would be able to simultaneously share one remote GPU from their local computers in the lab without having to log into the remote server, thus avoiding the server saturation mentioned before at the same time that the cost of the lab is still noticeably reduced. Notice that rCUDA is completely compatible with CUDA, so that is not necessary to modify CUDA programs and therefore students will only learn CUDA without having to bother with rCUDA, which would be transparent to them.

The cost of a laboratory using both approaches could be similar to the one presented in Table II.

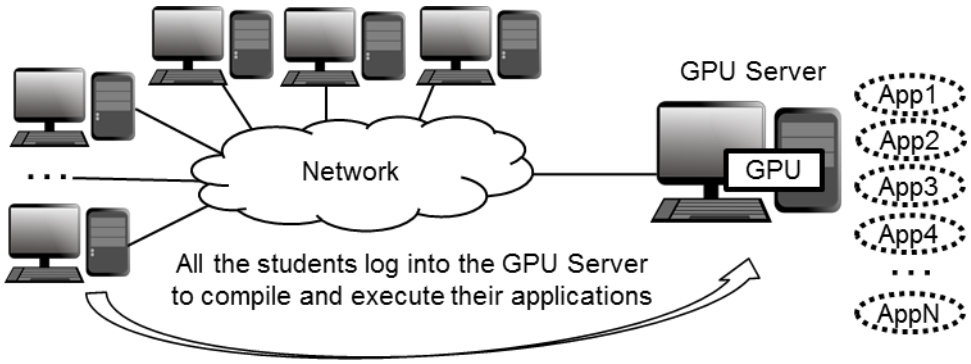

Fig. 4. CUDA teaching laboratory composed of regular computers and one server with a CUDA GPU. All the students log into the remote server, and compile and run their applications there.

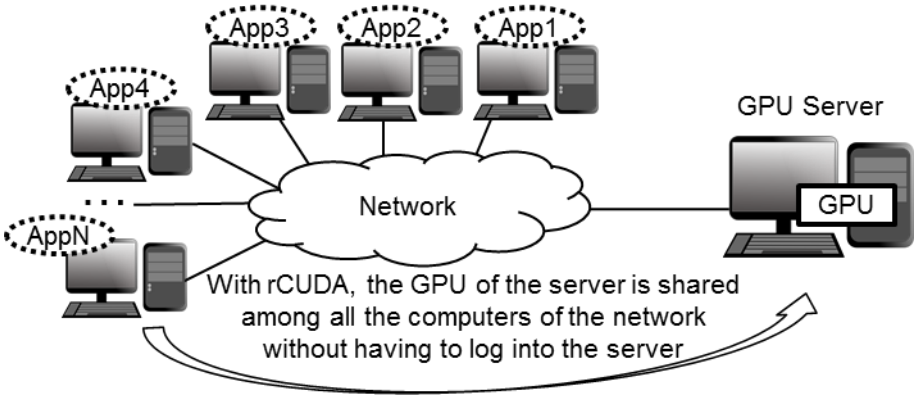

Fig. 5. CUDA teaching laboratory composed of regular computers and one server with a CUDA GPU.

5 EXPERIMENTS

In this section we compare the performance of five common applications which are usually used in CUDA teaching laboratories. All of them were extracted from the NVIDIA CUDA Samples [18]:

- *deviceQuery*: it is a simple program which only queries the properties of the GPU (compute capability, memory, number of cores…). It can be very useful to choose the most powerful GPU in systems with multiple available GPUs.
- *clock*: this sample shows how to use the *clock* function in order to measure the performance of a given function running in the GPU. Moreover, it shows the terminology used for running functions in the GPU. These functions are referred to as *kernels* in the CUDA jargon.
- *vectorAdd*: a program which implements an element by element vector addition. Apart from showing a kernel with functionality, it also introduces CUDA error checking.
- *template*: it is a simple template which can be later used by the students as starting point to create new CUDA programs.
- *cppIntegration*: this program shows how to introduce CUDA in a C++ existing application.

The performance of the applications previosly detailed is measured in three different scenarios:

- Using CUDA: the scenario is the one shown in Fig. 2, a laboratory composed of 20 computers with GPUs. The specifications of the computers are the ones presented in Table I.
- Logging into a remote GPU server: the scenario is the one shown in Fig. 4, a laboratory composed of 20 computers and 1 GPU server. The specifications of the computers are the ones presented in Table II.
- Using rCUDA: the scenario is the one shown in Fig. 5, a laboratory composed of 20 computers and 1 GPU server. The specifications of the computers are the ones presented in Table II.

Fig. 6 shows the results of this performance comparison. Regarding the scenario using CUDA, we measure the application execution time in one computer (i.e. one user) and we then use this result as the baseline performance. The reason for using only one computer is that in this scenario (see Fig. 2) all the computers have its own GPU, therefore the fact that other students in other computers are running applications does not influence the results in this computer.

In previous sections we have characterized the use of GPUs in a CUDA teaching laboratory and we have concluded that a maximum of 5 GPUs are simultaneously used at any given time in a laboratory composed of 20 computers. For this reason, in the scenarios sharing resources we just consider 5 concurrent users. This is the case of the scenario in which the students log into a remote GPU server

(see Fig. 4), and also the case of the scenario in which the students share the GPU of a remote server using rCUDA (see Fig. 5).

Regarding the scenario in which the students log into a remote GPU server, results in Fig. 6 show the average execution time of concurrently running the same application by 5 different students, who have previously logged into the GPU server from 5 differents computers. Notice that this average execution time is normalized to the baseline time previously commented.

With respect to the scenario in which the students share the GPU of a remote server using rCUDA, results in Fig. 6 present the average execution time of concurrently running the same application by 5 different students, who run the application in their own computers but, thanks to rCUDA, use the GPU at the remote server for executing the GPU parts of the applications. Notice again that this average execution time is normalized to the baseline time previously commented.

As it can be seen in Fig. 6, the overhead of the scenario using rCUDA with respect to the one using CUDA is under 5%, while the scenario in which the students log into a remote GPU server introduces, in general, an overhead about 30%. The only exception is the *deviceQuery* application, in this case logging into a remote GPU server adds hardly any overhead. The reason for this behaviour lies in the fact that this application is the only one of the applications under analysis which does not compute anything in the GPU (i.e., it does not run any *kernel* in the GPU). Given that CUDA applications are expected to make computations in the GPU, this will not be the common case. Therefore, the experiments presented in this section show that using rCUDA in teaching laboratories is a feasible option. rCUDA clearly improves the performance of logging into a remote GPU server, whereas the laboratory cost is not incremented.

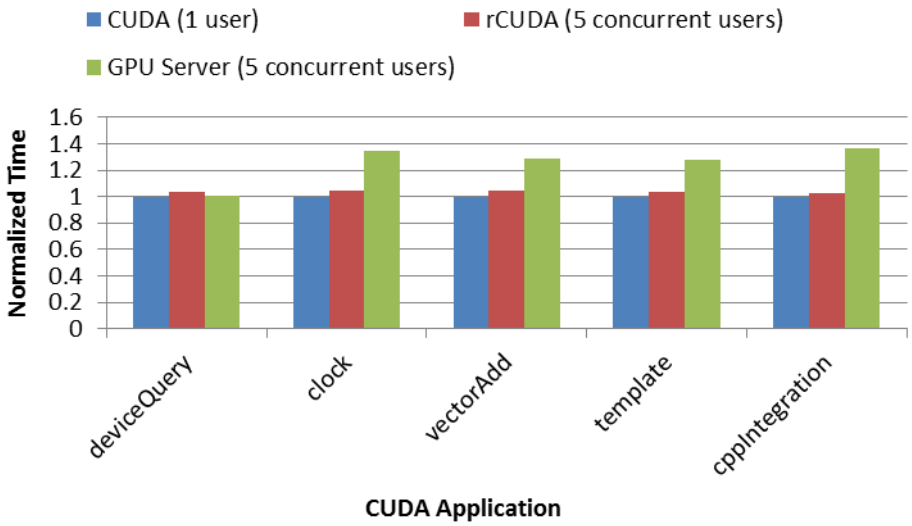

Fig. 6. Performance comparison of several CUDA applications running in three different scenarios: (1) a laboratory with one GPU in each computer, (2) a laboratory in which all the students log into one GPU server, and (3) a laboratory with one GPU shared among the students using rCUDA.

6 CONCLUSIONS

When teaching CUDA in laboratories, an important issue is the economic cost of GPUs, which may prevent some universities from building large enough labs to teach CUDA. In this work we have proposed an efficient solution to build CUDA labs reducing the number of GPUs. It is based on the use of the rCUDA (remote CUDA) middleware, which enables programs being executed in a computer to concurrently use GPUs located in remote servers. Therefore, students are able to simultaneously share one remote GPU from their local computers in the lab without having to log into the remote server, thus avoiding the saturation of the server at the same time that the cost of the lab is still noticeably reduced.

To study the viability of our proposal, we have first characterized the use of GPUs in this kind of labs with statistics taken from real users. Then, we have presented the results of sharing GPUs in a real teaching lab, taking into account the results from the previous laboratory characterization.

The results presented validate the feasibility of our proposal. With respect to CUDA, our solution using rCUDA shows an overhead under 5%. These results clearly improve alternative approaches, such as logging into remote GPU servers, which presents an overhead about 30% in our experiments.

ACKNOWLEDGEMENT

This work was partially funded by *Escola Tècnica Superior d'Enginyeria Informàtica de la Universitat Politècnica de València* and by *Departament d'Informàtica de Sistemes i Computadors de la Universitat Politècnica de València*.

REFERENCES [Arial, 12-point, bold, left alignment]

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

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

[3] 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

[4] 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

[5] 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

[6] 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.

[7] 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

[8] 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

[9] Ichitaro Yamazaki, Tingxing Dong, Raffaele Solcà, Stanimire Tomov, Jack Dongarra, Thomas Schulthess. 2014. Tridiagonalization of a dense symmetric matrix on multiple GPUs and its application to symmetric eigenvalue problems. Concurrency and Computation: Practice and Experience Volume 26, Issue 16 (November 2014), pages 2652–2666. DOI: http://dx.doi.org/10.1002/cpe.3152

[10] 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

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

[17] Carlos Reaño and Federico Silla. 2015. Reducing the Costs of Teaching CUDA in Laboratories while Maintaining the Learning Experience Quality. International Technology, Education and Development Conference (INTED).

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