2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing

On the Design of a Demo for Exhibiting rCUDA

Carlos Reaño, Ferrán Pérez, and Federico Silla
Universitat Politècnica de València
València, Spain
carregon@gap.upv.es, fsilla@disca.upv.es

***Abstract*— CUDA is a technology developed by NVIDIA which provides a parallel computing platform and programming model for NVIDIA GPUs and compatible ones. It takes benefit from the enormous parallel processing power of GPUs in order to accelerate a wide range of applications, thus reducing their execution time.**

rCUDA (remote CUDA) is a middleware which grants applications concurrent access to CUDA-compatible devices installed in other nodes of the cluster in a transparent way so that applications are not aware of accessing a remote device.

In this paper we present a demo which shows, in real time, the overhead introduced by rCUDA in comparison to CUDA when running image filtering applications. The approach followed in this work is to develop a graphical demo which contains both an appealing design and technical contents.

***Keywords*-GPGPU; CUDA; HPC; virtualization;**

I. Introduction

GPU-accelerated computing consists in using the massively parallel power of graphics processing units (GPUs) to boost the performance of a wide range of application in areas such as computational algebra, chemical physics, finance, or image analysis, to name only a few. Since 2006, NVIDIA response to this trend has been CUDA (Compute Unified Device Architecture) [1], a technology which provides a parallel computing platform and programming model for NVIDIA GPUs and compatible ones.

However, the use of GPUs in current high performance computing (HPC) clusters presents several disadvantages, such as high acquisition costs and power consumption. In addition, current computational science and HPC applications make, in general, a relatively low utilization of GPUs. Hence, sharing a reduced number of GPUs among the nodes of a cluster might be beneficial both to reduce acquisition costs and power consumption, and to increase GPU utilization rate.

rCUDA (remote CUDA) [2], [3] is a middleware which enables sharing remote CUDA-compatible devices concurrently and transparently. It grants applications concurrent access to GPUs installed in other nodes of the cluster in a manner that they are not aware of accessing a remote device. Furthermore, rCUDA does not require to modify the source code of applications and, additionally, introduces a small overhead with respect to CUDA.

In this paper we introduce a demonstrator for rCUDA consisting of a graphical demo which combines an appealing design and live applications along with technical contents.

The rest of the paper is organized as follows. In Section II we present rCUDA in more detail. Section III describes the applications later used in the demo. Finally, in Section IV we assemble all the components and describe the live demo to be presented.

II. rCUDA: remote CUDA

In the same way as CUDA uses local GPUs to accelerate certain parts of applications, rCUDA (remote CUDA) [2], [3] takes benefit from remote GPUs to do so. Figure 1 illustrates a sample scenario for the sake of clarity.

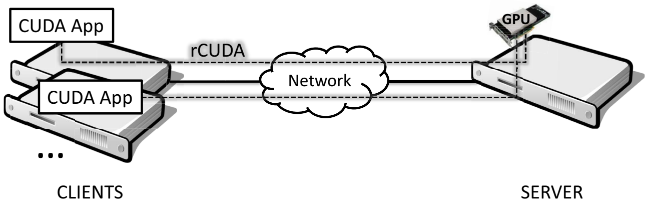


Figure 1. rCUDA sample scenario.

As commented previously, rCUDA enables sharing remote CUDA-compatible devices concurrently and transparently to applications. In this manner, a GPU installed in one node of a cluster (the server node) can be used by the rest of the nodes of the cluster (the client nodes) to accelerate applications using CUDA. In order to do so, the rCUDA middleware intercepts the application calls to the CUDA API and forwards them to the remote GPU. Notice that the application continues using the very same CUDA API and it does not require to be modified. Once a CUDA call arrives at the remote GPU thanks to rCUDA, it is executed using the real CUDA library and the real GPU. When the CUDA call completes, its results are returned by rCUDA to the application which made the initial call. Notice that this process is transparent to the application, which is not aware of accessing a remote GPU.

To communicate client and server nodes, rCUDA provides two different communications modules: one using the general TCP/IP protocol stack, and another using the InfiniBand Verbs API.

The last available rCUDA version, release 5.0, supports CUDA Runtime and Driver API 6.5. It also supports the

978-1-4799-8006-2/15 $31.00 © 2015 IEEE
DOI 10.1109/CCGrid.2015.53

most important routines of the following CUDA specific libraries: cuBLAS (Basic Linear Algebra Subprograms), cuFFT (Fast Fourier Transform), cuRAND (generation of random numbers), and cuSPARSE (BLAS subroutines for handling sparse matrices).

Finally, rCUDA is free and can be obtained from the website www.rcuda.net.

III. Applications Used in the Demo

In this section we describe the applications used in the live demo. It is important to remark that one of the demo requirements was that it should attract the attention of the exhibition attendees. Therefore, the demo should be devised with a very appealing design in order to attract the interest of attendees. For this reason, the applications used in the demo are two image filters: color image to grayscale conversion (Subsection III-A), and image blurring (Subsection III-B). Additionally, those filters will be applied to a set of over 200 pictures especially selected to attract the attention of attendees.

A. Color Image to Grayscale Conversion

In computer graphics, each pixel of a color image is commonly represented by four parameters: RGBA [4]. 'R' indicates how much red is in the pixel, 'G' how much green and 'B' how much blue. 'A' stands for Alpha and specifies the opacity of the pixel. Each one of these parameters is represented by one byte, so there are 256 different possible values for each parameter.

On the other hand, each pixel of a grayscale image is represented by a single parameter which specifies the level of gray using one byte. Hence, each pixel has 256 possible values.

To convert an image from color to grayscale, given that the eye responds most strongly to green, followed by red and then blue, the NTSC (National Television System Committee) recommends using Equation 1. Notice that the parameter 'A' is ignored in this formula.

$$I = 0.299 * R + 0.587 * G + 0.114 * B \qquad (1)$$

Based on an initial program extracted from [4], we have developed a CUDA application which performs the image conversion in the GPU using the formula shown in Equation 1.

B. Image Blurring

Blurring an image [4] consists in applying to each pixel and its neighbors a filter which varies depending on the desired level of distortion. For instance, imagine that we have an image represented by the matrix shown in Figure 2, where 'B' represents the pixel to blur, 'N1..N8' refer to what we have called the neighbor pixels and 'X' are pixels which will not be modified when blurring pixel 'B'. To blur pixel 'B' of the image represented by the matrix in Figure 2, we apply Equation 2, where 'd' is an array which specifies the distortion level for each neighbor pixel and 'db' is the distortion level for pixel 'B'.

```
X   X   X   X   X   X
    ----------
X  |N1   N2   N3|  X
   |            |
X  |N4   B    N5|  X
   |            |
X  |N6   N7   N8|  X
    ----------
X   X   X   X   X   X
```

Figure 2. Matrix representing the image to blur: 'B' represents the pixel to blur, 'N1..N8' refer to the neighbor pixels and 'X' are pixels which will not be modified when blurring pixel 'B'.

$$blur(B) = B * db + \sum_{i=1}^{8} N[i] * d[i] \qquad (2)$$

Based on an initial program extracted from [4], we have developed a CUDA application which performs the image blurring in the GPU using the formula shown in Equation 2.

IV. rCUDA Demo Description

In this section we describe the demo. We first present the equipment used for the demo (Subsection IV-A) and then the demo itself (Subsection IV-B). Finally, we show performance results in Subsection IV-C.

A. Equipment used

The equipment necessary for this demo consists of two 1027GR-TRF Supermicro servers, each with the following characteristics:

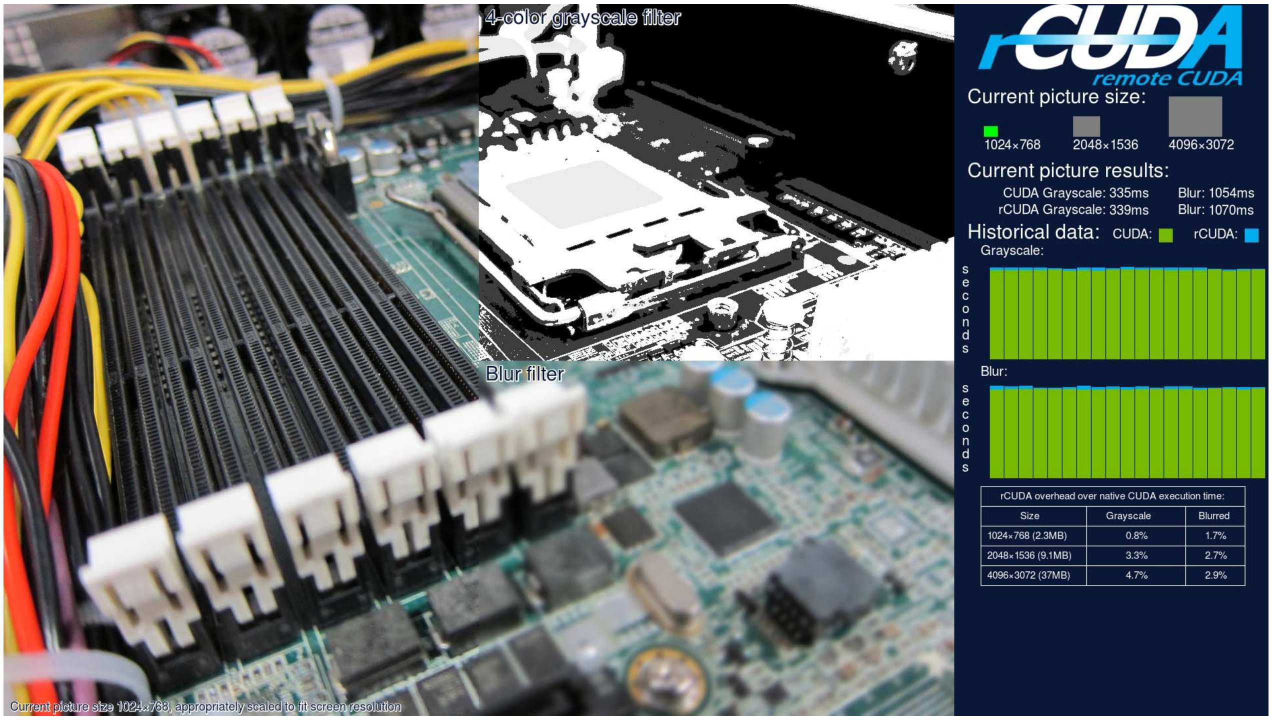
- Two Intel Xeon hexa-core processors E5-2680 v2 (Ivy Bridge) operating at 2.8 GHz.
- 32 GB of DDR3 SDRAM memory at 1,600 MHz.
- 1 Mellanox Connect-IB (FDR) dual-port InfiniBand adapter.
- Red Hat Enterprise Linux Server release 6.4 with Mellanox OFED 2.1-1.0.0 (InfiniBand drivers and administrative tools) and CUDA 6.5 with NVIDIA driver 340.29.
- 1 NVIDIA Tesla K80

In addition, one monitor is necessary to display the graphical part of the demo. Figure 4 shows how the equipment is interconnected. The demo runs in node A, whereas node B hosts an rCUDA server.

B. Description of the Demo

Figure 3 presents a screen shot of the demo. A video of the demo can also be seen at http://youtu.be/qblh6wW3DHA.

The demo consists of 245 different color pictures, each of them available in three different sizes: 1024x768 (2.4MB), 2048x1536 (9.4MB), and 4096x3072 (37.7MB). The current



Current picture size 1024×768, appropriately scaled to fit screen resolution

Current picture size:

🟩 1024×768 — 2048×1536 — 4096×3072

Current picture results:

CUDA Grayscale: 335ms Blur: 1054ms
rCUDA Grayscale: 339ms Blur: 1070ms

Historical data: CUDA: 🟩 rCUDA: 🟦

Grayscale:

Blur:

rCUDA overhead over native CUDA execution time:

Size	Grayscale	Blurred
1024×768 (2.3MB)	0.8%	1.7%
2048×1536 (9.1MB)	3.3%	2.7%
4096×3072 (37MB)	4.7%	2.9%

Figure 3. Screen shot of the demo.

image size being computed during the live demo is shown at the top right part of the screen under the label "Current picture size". For each image and size, the next steps are followed:

1) The original image is displayed on the screen.
2) The image is converted to grayscale first using CUDA (the calculations are done in a local GPU), then using rCUDA (the calculations are done in a remote GPU). For so, we employ the filter exposed in Subsection III-A. The image conversion times with CUDA and with rCUDA are stored separately.
3) Although the complete image is converted to grayscale, only the top right part of the image displayed on the screen is changed to grayscale due to aesthetic reasons.
4) The blur filter explained in Subsection III-B is then applied to the image using again CUDA and rCUDA. Both conversion times are also stored.
5) Though the whole image is blurred, only the bottom right part of the blurred image is displayed on the screen for aesthetic reasons.
6) The conversion time of CUDA and rCUDA for both filters (grayscale and blur) is numerically displayed at the right side of the screen. It is also represented in the form of a bar chart: the green part of the bars is the CUDA conversion time, while the blue part of the bars refers to the overhead of doing the same conversion with rCUDA. The bar chart keeps track of the results for the last 20 images.
7) The bottom right part of the screen is then updated showing the average rCUDA overhead over CUDA for the different image sizes and filters, taking into account all the executions since the demo started.

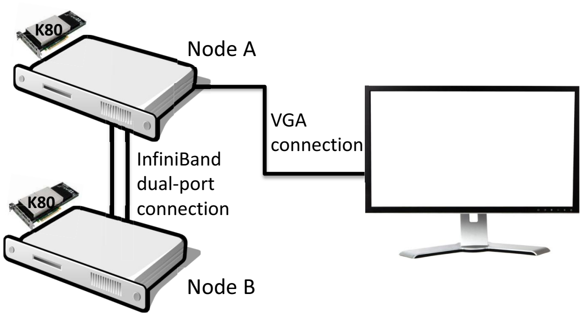


Figure 4. Scheme of the equipment used for the demo.

Once the previous sequence is completed, a new image is displayed on the screen, and the process starts again. It is repeated for all the images and all the sizes.

C. *Performance Results*

The applications used in the demo have different behaviors in order to show the performance of rCUDA under distinct scenarios. In general, three factors influence rCUDA:

- Transfers: CUDA memory copies translate into network transfers when using rCUDA, what introduces an overhead which depends on network bandwidth.
- Computations: the time employed by CUDA kernels in the GPU is the same for CUDA and rCUDA. Therefore, performing a large amount of computations helps rCUDA to compensate the overhead caused by transferring data across the network.
- CUDA calls: when using rCUDA, calls to the CUDA API turn into small size network transfers, which increment rCUDA overhead depending on network latency.

Figure 5 presents the rCUDA overhead over CUDA when running the applications explained in Section III using the three different image sizes commented in Subsection IV-B. The results are the average of ten executions, and the maximum Relative Standard Deviation (RSD) observed was 0.077. This RSD was achieved when using CUDA and the grayscale filter over an image of size 1024x768. To ease the interpretation of the results, we also show in Figure 6 and Figure 7 profiling information obtained by using the NVIDIA profiling tools.

Regarding the application which converts the images from color to grayscale, referred to as "grayscale" in the figures, we can observe that the overhead experienced by rCUDA noticeably increases with image size (see Figure 5). This is due to the fact that the time spent in transfers (Figure 7), is growing much faster than the time spent in computations (Figure 6) when increasing the image size, thus making more notorious the overhead introduced by rCUDA because of network transfers.

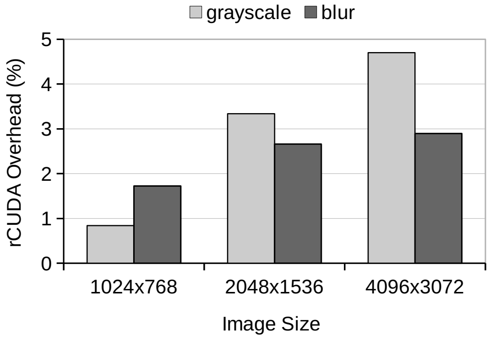


Figure 5. rCUDA overhead over CUDA when running grayscale and blur filters.

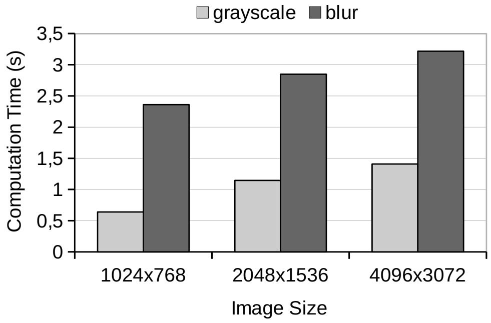


Figure 6. Time spent in computations (i.e., CUDA kernels) by grayscale and blur filters.

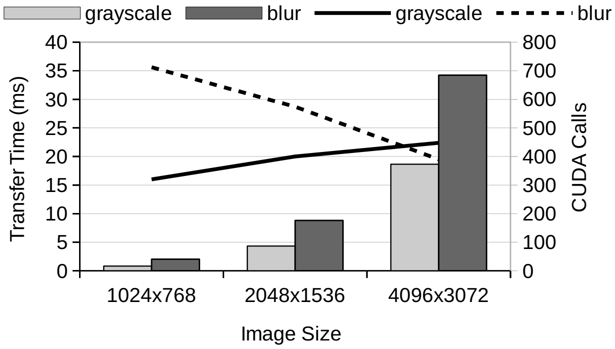


Figure 7. Time spent in transfers (i.e., CUDA memcopies) and calls made to the CUDA API by grayscale and blur filters. Bars represent transfers, whereas lines depict calls.

With respect to the application which blurs the images, labeled as "blur" in the figures, it can be seen that the overhead presented by rCUDA for an image size of 1024x768 is higher than in the grayscale application (see Figure 5). This is because the number of calls to the CUDA API for this filter is larger than for the grayscale one (Figure 7), what introduces an overhead due to the network latency which is not compensated by the time spent in computations (Figure 6).

In contrast, rCUDA overhead for image sizes of 2048x1536 and 4096x3072 is lower when running the blur filter than the grayscale one (Figure 5). The reason lies in the fact that the time spent in computations (Figure 6) in this case is enough to counterbalance the overhead due to network transfers (Figure 7).

ACKNOWLEDGMENT

This work was funded by the Spanish MINECO and FEDER funds under Grant TIN2012-38341-C04-01. Authors are also grateful for the generous support provided by Mellanox Technologies and the equipment donated by NVIDIA Corporation.

REFERENCES

[1] NVIDIA, *NVIDIA CUDA C Programming Guide 6.5*, 2014.

[2] A. J. Peña, C. Reaño, F. Silla, R. Mayo, E. S. Quintana-Ortí, and J. Duato, "A complete and efficient cuda-sharing solution for HPC clusters," *Parallel Computing (PARCO)*, vol. 40, no. 10, pp. 574–588, 2014.

[3] C. Reaño, R. Mayo, E. S. Quintana-Ortí, F. Silla, J. Duato, and A. J. Peña, "Influence of infiniband FDR on the performance of remote GPU virtualization," in *IEEE International Conference on Cluster Computing (CLUSTER)*, 2013, pp. 1–8.

[4] Udacity, *Intro to Parallel Programming*, 2015.