J. Parallel Distrib. Comput. 72 (2012) 1117–1126

Contents lists available at SciVerse ScienceDirect

J. Parallel Distrib. Comput.

journal homepage: www.elsevier.com/locate/jpdc

Performance models for asynchronous data transfers on consumer Graphics Processing Units

Juan Gómez-Luna [a,*], José María González-Linares [b], José Ignacio Benavides [a], Nicolás Guil [b]

[a] Department of Computer Architecture and Electronics, University of Córdoba, Spain

[b] Department of Computer Architecture, University of Málaga, Spain

ARTICLE INFO

Article history:
Available online 12 August 2011

Keywords:
GPU
CUDA
Asynchronous transfers
Streams
Overlapping of communication and computation

ABSTRACT

Graphics Processing Units (GPU) have impressively arisen as general-purpose coprocessors in high performance computing applications, since the launch of the Compute Unified Device Architecture (CUDA). However, they present an inherent performance bottleneck in the fact that communication between two separate address spaces (the main memory of the CPU and the memory of the GPU) is unavoidable. The CUDA Application Programming Interface (API) provides asynchronous transfers and *streams*, which permit a staged execution, as a way to overlap communication and computation. Nevertheless, a precise manner to estimate the possible improvement due to overlapping does not exist, neither a rule to determine the optimal number of stages or streams in which computation should be divided. In this work, we present a methodology that is applied to model the performance of asynchronous data transfers of CUDA streams on different GPU architectures. Thus, we illustrate this methodology by deriving expressions of performance for two different consumer graphic architectures belonging to the more recent generations. These models permit programmers to estimate the optimal number of streams in which the computation on the GPU should be broken up, in order to obtain the highest performance improvements. Finally, we have checked the suitability of our performance models with three applications based on codes from the CUDA Software Development Kit (SDK) with successful results.

© 2011 Elsevier Inc. All rights reserved.

1. Introduction

Communication overhead is one of the main performance bottlenecks in high-performance computing systems. In distributed memory architectures, where the Message Passing Interface (MPI) [10] has the widest acceptance, this is a well-known limiting factor. MPI provides asynchronous communication primitives, in order to reduce the negative impact of communication, when processes with separate address spaces need to share data. Programmers are able to overlap communication and computation by using these asynchronous primitives [9,19].

Similar problems derived from communications are being found in Graphics Processing Units (GPU), which have spectacularly burst in the scene of high-performance computing, since the launch of Application Programming Interfaces (API) such as the Compute Unified Device Architecture (CUDA) [14] and the Open Computing Language (OpenCL) [7]. Their massively parallel architecture is making possible impressive performances at cheap prices, although there exists an inherent performance bottleneck due to data transfers between two separate address spaces, the main memory of the Central Processing Unit (CPU) and the memory of the GPU.

In a typical application, non-parallelizable parts are executed in the CPU or *host*, while massively parallel computations can be delegated to a GPU or *device*. With this aim, the CPU transfers input data to the GPU through the PCI Express (PCIe) [15] bus and, after the computation, results are brought back to the CPU. Since its first release, the CUDA API provides a function, called `cudaMemcpy()` [12], that transfers data between the host and the device. This is a blocking function in the sense that the GPU code, called *kernel*, can be launched only after the transfer is complete. Despite that the PCIe supports a throughput of several gigabytes per second, both transfers inevitably burden the performance of the GPU. In order to alleviate such a performance bottleneck, later releases of CUDA provide the non-blocking `cudaMemcpyAsync()` [12], which requires host pinned memory. It permits asynchronous transfers, i.e. enables overlap of data transfers with computation, in devices with compute capability equal or higher than 1.1 [12]. Such a concurrency is managed through *streams*, i.e. sequences of commands that are executed in order. Streams permit transfers and execution to be broken up into a number of stages, so that some overlapping of data transfer and computation is achieved.

* Corresponding author.
E-mail address: el1goluj@uco.es (J. Gómez-Luna).

0743-7315/$ – see front matter © 2011 Elsevier Inc. All rights reserved.
doi:10.1016/j.jpdc.2011.07.011

```
for (int i = 0; i < number_of_streams; ++i)
      cudaMemcpyAsync(inputDevPtr + i * size, hostPtr + i * size, size,
          cudaMemcpyHostToDevice, stream[i]);
for (int i = 0; i < number_of_streams; ++i)
       MyKernel<<<num_blocks / number_of_streams, num_threads, 0, stream[i]>>>
          (outputDevPtr + i * size, inputDevPtr + i * size, size);
for (int i = 0; i < number_of_streams; ++i)
      cudaMemcpyAsync(hostPtr + i * size, outputDevPtr + i * size, size,
          cudaMemcpyDeviceToHost, stream[i]);
cudaThreadSynchronize();
```

This paradigm is closely related to the streaming programming model, which has been used to facilitate code portability to GPU architectures [6] and cooperative application execution on multi-core processors and accelerators [18].

Some research works have made use of the CUDA stream model in order to improve application performance [2,4,16]. However, finding optimal configurations, i.e. the best number of streams or stages in which transfers and computation are divided, requires many attempts for tuning the application. Moreover, CUDA literature [11,12] does not provide an accurate way to estimate the performance improvement due to the use of streams. Such a lack of reliable analytical models limits the usefulness of asynchronous transfers and streams. In this way, we consider that this research work covers an empty space, because we have obtained performance models, which have been validated from both architectural and experimental points of view. They permit programmers to estimate the execution time of a streamed application and the optimal number of streams that is recommended for use.

GPU performance modeling has been tackled in some valuable research works [1,5,20], but none of them deals with data transfers between CPU and GPU and the use of streams. To the best of our knowledge, there is only one research work focused on CUDA streams performance [8]. It presents some theoretical models for asynchronous data transfers, but they are not empirically validated, neither related to architectural issues. The authors do not give any hint about the applicability of these models and assume that the optimal number of streams is 8 for any application.

Our work starts with a thorough observation of CUDA streams performance, in order to accurately characterize how transfers and computation are overlapped. We have carried out a huge number of experiments by changing the ratio between kernel execution time and transfers time, and the ratio between input and output data transfer times. Then, we have tried out several performance estimates, in order to check their suitability to the results of experiments. Thus, our main contributions are:

- We present a novel methodology that is applicable for modeling the performance of asynchronous data transfers when using CUDA streams.
- We have applied this methodology to devices with compute capabilities (c.c.) 1.*x* and 2.*x*. Thus, we have derived two performance models, i.e. the one for devices with c.c. 1.*x* and the other for devices with c.c. 2.*x*.
- Moreover, from the mathematical expressions obtained can be derived the optimal number of streams to reach the maximum computation time speed-up. The optimal number of streams to be used for a specific application only depends on the data transfer time and the kernel computation time of the non-streamed application.
- We have successfully checked the applicability of our models to several applications based on codes from the CUDA Software Development Kit (SDK). We also show that signal processing applications, where data are being continuously processed, can benefit from our approach as they can recalculate the optimal number of streams from previous calculations.

The rest of the paper is organized as follows. Section 2 reviews the use of CUDA streams. In Section 3, we explain how the behavior of CUDA streams has been analyzed and we propose two performance models. Our models are checked in Section 4 using several applications. Finally, conclusions are stated in Section 5.

2. CUDA streams

In order to overlap communication and computation, CUDA permits division of memory copies and execution into several stages, called streams. CUDA defines a stream as a sequence of operations that are performed in order on the device. Typically, such a sequence contains one memory copy from host to device, which transfers input data; one kernel launch, which uses these input data; and one memory copy from device to host, which transfers results [12] (see code listing given in Box).

Given a certain application which uses D input data instances and defines B blocks of threads for kernel execution, a programmer could decide to break up them into *nStreams* streams. Thus, each of the streams works with $\frac{D}{nStreams}$ data instances and $\frac{B}{nStreams}$ blocks. In this regard, the memory copy of one stream overlaps kernel execution of other stream, achieving a performance improvement. An important requirement for ensuring the effectiveness of the streams is that $\frac{B}{nStreams}$ blocks are enough for maintaining all hardware resources of the GPU busy. Otherwise the sequential execution could be faster than the streamed one.

The use of streams can be very profitable in applications where input data instances are independent, so that computation can be divided into several stages. For instance, video processing applications satisfy this requirement, when computation on each frame is independent. A sequential execution should transfer a sequence of n frames to device memory, apply certain computation on each of the frames, and finally copy results back to the host. If we consider a number b of blocks used per frame, the device will schedule $n \times b$ blocks for the whole sequence. However, a staged execution of *nStreams* streams transfers chunks of $\frac{n}{nStreams}$ size. Thus, while the first chunk is being computed using $\frac{n \times b}{nStreams}$ blocks, the second chunk is being transferred. An important improvement will be obtained by hiding the frame transfers.

Estimating the performance improvement that is obtained through streams is crucial for programmers, when an application is to be streamed. Considering data transfer time t_T and kernel execution time t_E, the overall time for a sequential execution is $t_E + t_T$. In [11], the theoretical time for a streamed execution is estimated in two ways:

- Assuming that t_T and t_E are comparable, a rough estimate for the overall time is $t_E + \frac{t_T}{nStreams}$ for the staged version. Since it is assumed that kernel execution hides data transfer, in the following Sections, we call this estimate *dominant kernel*.
- If the transfer time exceeds the execution time, a rough estimate is $t_T + \frac{t_E}{nStreams}$. This estimate is called *dominant transfers*.

Table 1
NVIDIA GeForce series features related to data transfers and streams.

GeForce series	Features	Considerations related to streams
8 9 200	Compute capability 1.x ($x > 0$) PCIe × 16 (8 series) PCIe × 16 2.0 (9 and 200 series) 1 DMA channel Overlapping of data transfer and kernel execution	Host-to-device and device-to-host transfers cannot be overlapped (only one DMA channel) No implicit synchronization: Device-to-host data transfer of a stream just can start when that stream finishes its computation. Consequently, this transfer can be overlapped with the computation of the following stream
400 500	Compute capability 2.x PCIe × 16 2.0 1 DMA channel Overlapping of data transfer and kernel execution Concurrent kernel execution	Host-to-device and device-to-host transfers cannot be overlapped (only one DMA channel) Implicit synchronization: Device-to-host data transfer of the streams cannot start until all the streams have started executing

3. Characterizing the behavior of CUDA streams

The former expressions do not define the possible improvement in a precise manner or give any hint about the optimal number of streams. For this reason, in this Section, we apply a methodology which consists of testing and observing the streams, by using a sample code, included in the CUDA SDK. This methodology thoroughly examines the behavior of the streams through two different tests:

- First, the size of the input and output data is fixed, while the computation within the kernel is variable.
- After that, the size of the data transfers is asymmetrically changed. Along these tests, the number of bytes that are transferred from host to device is ascending, while the number of bytes from device to host is descending.

After applying our methodology, we are able to propose two performance models which fit the results of the tests.

3.1. *A thorough observation of CUDA streams*

The CUDA SDK includes the code `simpleStreams.cu`, which makes use of CUDA streams. It compares a non-streamed execution and a streamed execution. The kernel is a simple code in which a scalar is repeatedly added to an array, that represents a vector. One variable defines the number of times that the scalar is added to each element of the array, that is, the number of iterations within the kernel.

`simpleStreams.cu` declares streams that include the kernel and the data transfer from device to host, but not the data transfer from host to device. We have modified the code, so that transfers from host to device are also included in the streams. Thus, we observe the behavior of CUDA streams in the whole process of transferring from CPU to GPU, executing on the GPU and transferring from GPU to CPU. Testing this code gives us three parameters which define a huge number of cases: the size of the array, the number of iterations within the kernel and the number of streams. In this way, in the first part of our methodology, we use a fixed array size and change the number of iterations within the kernel and the number of streams. This permits us to compare dominant transfers and dominant kernel cases. Afterwards, in the second part, the sizes of data transfers are changed asymmetrically, in order to refine the performance estimates.

After observing the behavior of CUDA streams, one performance model for stream computation will be calculated for each of the two most recent NVIDIA architectures (compute capabilities 1.x and 2.x). In this paper, the applied methodology is illustrated on the Geforce GTX 280, as an example of c.c. 1.x, and on the Geforce GTX 480, as an example of c.c. 2.x. In [3], results for others GPUs can be found.

Details about NVIDIA devices are presented in Table 1. As stated in [12], devices with compute capability 1.x do not support concurrent kernel execution. In this way, streams are not subject to implicit synchronization. In devices with compute capability 2.x, concurrent kernel execution entails that those operations, which require a dependency check (such as data transfers from device to host), cannot start executing until all thread blocks of all prior kernel launches from any stream have started executing. These considerations should be ratified by the execution results, after applying our methodology.

3.1.1. *First observations: Fixed array size*

First tests carried out consist of adding a scalar to an array of size 15 MB, using the modified `simpleStreams.cu`. The number of iterations within the kernel takes 20 different values (from 8 to 27 in steps of 1, in GTX 280; and from 20 to 115 in steps of 5, in GTX 480). Thus, these tests change the ratio between kernel execution and data transfers times, in order to observe the behavior of the streams in a large number of cases. The number of streams is changed along the divisors of 15 M between 2 and 64.

Fig. 1 shows the execution results on GeForce GTX 280. A blue line with diamond markers presents the non-streamed execution results and an orange line with square markers stands for the streamed execution results. The graph is divided into several columns. Each of the columns represents one test using a certain number of iterations within the kernel. This number of iterations, between 8 and 27, which determines the computational complexity of the kernel, is shown in abscissas. Together with the execution times for non-streamed and streamed configurations, two thick lines and two thin lines have been included. Thick lines represent the data transfers and the kernel execution times. Thin lines correspond to possible performance models for the streamed execution, as stated in [11]. The thin red line considers a dominant kernel case and estimates the execution time as $t_E + \frac{t_T}{nStreams}$, where t_T is the copy time from CPU to GPU plus the copy time from GPU to CPU. The thin green line represents a dominant transfers case and the estimate is $t_T + \frac{t_E}{nStreams}$.

The dominant kernel hypothesis is reasonably suitable when the kernel execution time is clearly longer than the data transfers time. However, the dominant transfers hypothesis does not match the results of any test. In this way, we observe that the transfers time t_T (thick green line) is a more accurate reference when the data transfers are dominant.

In the dominant transfers cases (results on the left of the graph) on the GeForce GTX 280, we also observe that the best results for the streamed execution are around the point where the thick green line and the thin red line intersect. In this intersection point, the dominant kernel estimate equals the transfers time. In this way, a reference for the optimal number of streams is $nStreams = \frac{t_T}{t_T - t_E}$.

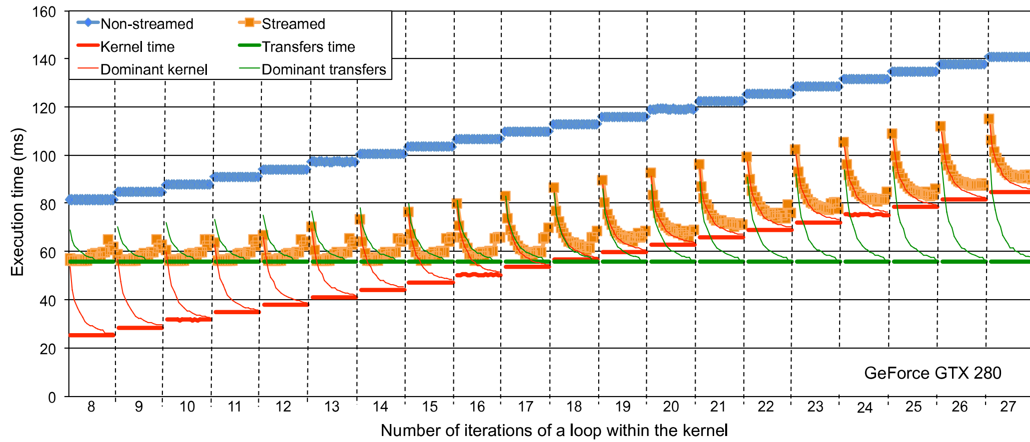

Fig. 1. Execution time (ms) for the addition of a scalar to an array of size 15 MB on GeForce GTX 280. The blue line represents the execution time for non-streamed executions and the orange line stands for the results of the streamed execution. Each column in the graph represents a test with a changing number of iterations between 8 and 27 in steps of 1, in abscissas. In each column, the number of streams has been changed along the divisors of 15 M between 2 and 64. Thick green and red lines represent respectively the transfers time and the kernel execution time in each column. Thin green and red lines represent possible performance models (dominant transfer or dominant kernel) as stated in [11]. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

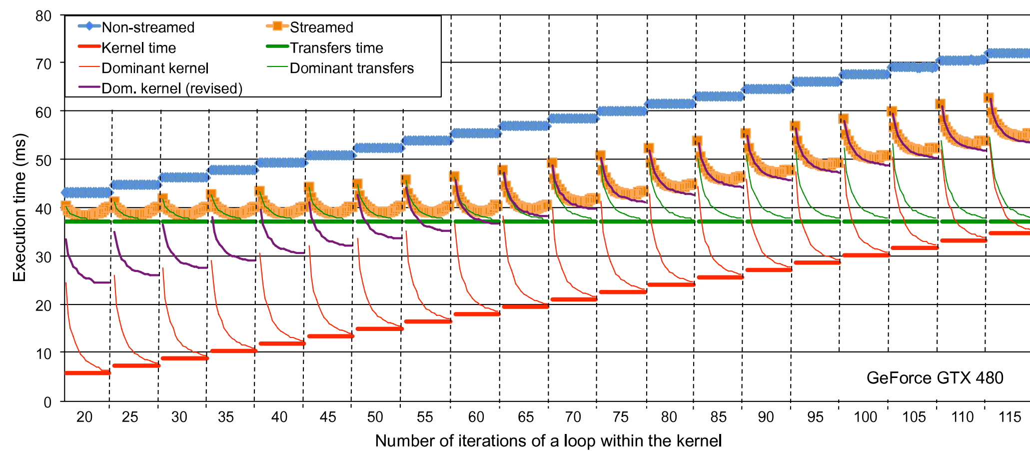

Fig. 2. Execution time (ms) for the addition of a scalar to an array of size 15 MB on GeForce GTX 480. Each column in the graph represents a test with a changing number of iterations between 20 and 115 in steps of 5. In each column, the number of streams has been changed along the divisors of 15 M between 2 and 64. Thick green and red lines represent respectively the data transfers time and the kernel execution time in each column. Thin green and red lines represent possible performance models (dominant transfer or dominant kernel) as stated in [11]. The thin purple line stands for a revised dominant kernel model, in which only one of the transfers is hidden. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

On the GeForce GTX 480, the dominant transfers hypothesis suits properly on the left of the graph. However, the dominant kernel hypothesis does not fit in any case. Fig. 2 shows that a revised dominant kernel hypothesis (thin purple line), in which the streams hide only one of the data transfers, is a better match. The revised estimate is $t_E + \frac{t_{T1}}{nStreams} + t_{T2}$, where $t_{T1} + t_{T2} = t_T$. At this point we are not able to assert which of the transfers, i.e. host to device or device to host, is hidden, since both copy times are similar.

Finally, it is remarkable that, in all tests on both GPUs, the streamed time gets worse from a certain number of streams. One can figure out that some overhead exists due to the generation of a stream. Thus, the higher the number of streams the longer the overhead time.

3.1.2. Second observations: Asymmetric transfers

Second tests use the same kernel with a variable number of iterations, but data transfers are asymmetric. For each kernel using a certain number of iterations, we perform 13 tests in which 24 MB are transferred from host to device or from device to host. Along the 13 tests, the number of bytes copied from host to device is ascending, while the number of bytes from device to host is descending. In this way, the first test transfers 1 MB from host to device and 23 MB from device to host, and in the last test 23 MB are copied from host to device and 1 MB from device to host. The number of streams has been established as 16 for every test.

Fig. 3 (top) shows the results on the GeForce GTX 280. It can be observed that the streamed results match the transfers times, when data transfers are dominant (tests with 1, 2 and 4 iterations). When the kernel execution is longer (test with 16 iterations), the dominant kernel estimate fits properly.

Moreover, one can notice that the execution time decreases along the 13 tests in each column, despite the whole amount of data transferred from or to the device being constant. We have observed that on GTX 280 data transfers from device to host take around 36% more time than transfers from host to device. For this reason, the left part of the test with 8 iterations follows the transfers time, while the right part fits the dominant kernel hypothesis.

In Section 3.1.1, we observed that on the GeForce GTX 480 only one of the data transfers was hidden by the kernel execution, when the kernel was dominant. In these tests with asymmetric transfers, we conclude that the transfer from host to device is the one being hidden, as can be observed in Fig. 3 (bottom). It depicts two revised

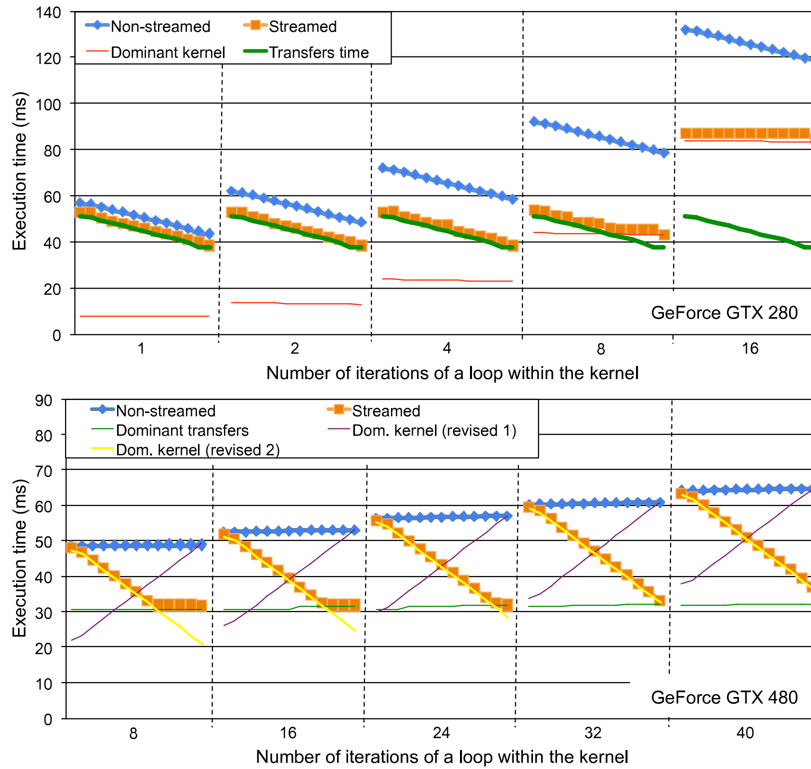

Fig. 3. Execution time (ms) on GeForce GTX 280 (top) and GTX 480 (bottom) for tests with asymmetric transfers. 24 MB are copied from host to device or from device to host. Abscissas represent the number of iterations within the kernel. In each column, 13 tests are represented with an ascending number of bytes from host to device and a descending number of bytes from device to host. In all cases, the number of streams is 16. In the graph on top, the thin red line stands for a dominant kernel hypothesis and the thick green line is the transfers time. In the graph on the bottom, the thin green line stands for the dominant transfers hypothesis, and thin purple and yellow lines represent two revisions of the dominant kernel estimate. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

dominant kernel estimates, purple and yellow thin lines. The first revised estimate assumes that the transfer from device to host is hidden, while the second one considers the transfer from host to device to be overlapped with execution. It is noticeable that the latter estimate matches perfectly when kernel execution is clearly dominant (32 and 40 iterations).

The former observation agrees with the fact that dependent operations in GTX 480 do not start until all prior kernels have been launched. Thus, data transfers from device to host are not able to overlap with computation, since all kernels from any stream are launched before data transfers from device to host, as it can be seen in the code at the beginning of Section 2.

When the data transfer from host to device takes more time than the kernel execution, the streamed execution follows the dominant transfers hypothesis. For this reason, the right part of the columns with 8, 16 and 24 iterations follows the thin green line.

On the GTX 480, data transfers from device to host are slightly faster (around 2%) than transfers from host to device. This fact explains the weak increase of the execution time along the 13 tests in each column.

3.2. CUDA streams performance models

Considering the observations in the previous subsections, we are able to formulate two performance models which fit the behavior of CUDA streams on devices with c.c. 1.*x* and 2.*x*. In the following equations, t_E represents the kernel execution time, t_{Thd} stands for the data transfer time from host to device and t_{Tdh} the data transfer time from device to host. Transfer times satisfy $t_T = t_{\mathrm{Thd}} + t_{\mathrm{Tdh}}$, and it depends on the number of data to be transmitted and the characteristics of the PCIe bus. Moreover, we define an overhead time t_{oh} derived from the creation of the streams. We consider that this overhead time increases linearly with the number of streams, i.e. $t_{\mathrm{oh}} = t_{\mathrm{sc}} \times nStreams$. The value of t_{sc} should be estimated for each GPU. In [3], we show values of t_{sc} for NVIDIA devices belonging to the GeForce 8, 9, 200, 400 and 500 series. In the particular case of GTX 280 and GTX 480, t_{sc} takes a value of 0.10 and 0.03, respectively.

3.2.1. Performance on devices with compute capability 1.x

When data transfers time is dominant, we realized that the streamed execution time t_{streamed} tends to the data transfers time t_T. Since the performance of CUDA streams on these devices is not subject to implicit synchronization, the data transfers time is able to completely hide the execution time. Thus, we propose the following model for *nStreams* streams:

$$\text{If}\left(t_T > t_E + \frac{t_T}{nStreams}\right), \quad t_{\mathrm{streamed}} = t_T + t_{\mathrm{oh}}. \tag{1}$$

In Section 3.1.1, we noticed that the optimal number of streams $nStreams_{op}$, with dominant transfers times, is around:

$$nStreams_{op} = \frac{t_T}{t_T - t_E}. \tag{2}$$

In a dominant kernel scenario, the most suitable estimate counts the kernel execution time and the data transfers time divided by *nStreams*:

$$\text{If}\left(t_T < t_E + \frac{t_T}{nStreams}\right), t_{\mathrm{streamed}} = t_E + \frac{t_T}{nStreams} + t_{\mathrm{oh}}. \tag{3}$$

Deriving Eq. (3) permits obtaining the optimal number of streams in a dominant kernel case:

$$nStreams_{op} = \sqrt{\frac{t_T}{t_{sc}}}. \quad (4)$$

Fig. 4 (top) shows the suitability of our performance model to the execution time results presented in Section 3.1.1.

3.2.2. Performance on devices with compute capability 2.x

In Section 3.1.1, we observed that on GTX 480 a dominant transfers scenario was properly defined as in [11]. Moreover, from Section 3.1.2 we infer that on GTX 480 only the data transfer from host to device is overlapped with kernel execution. In this way, when data transfer is dominant, we propose:

$$\text{If } (t_{\text{Thd}} > t_E),\ t_{\text{streamed}} = t_{\text{Thd}} + \frac{t_E}{nStreams} + t_{\text{Tdh}} + t_{\text{oh}}. \quad (5)$$

The first derivative of the former equation gives an optimal number of streams:

$$nStreams_{op} = \sqrt{\frac{t_E}{t_{sc}}}. \quad (6)$$

In a dominant kernel scenario, we propose the last revised estimate presented in Section 3.1.2:

$$\text{If } (t_{\text{Thd}} < t_E),\ t_{\text{streamed}} = \frac{t_{\text{Thd}}}{nStreams} + t_E + t_{\text{Tdh}} + t_{\text{oh}}. \quad (7)$$

The optimal number of streams, when the kernel is dominant, is obtained with:

$$nStreams_{op} = \sqrt{\frac{t_{\text{Thd}}}{t_{sc}}}. \quad (8)$$

This performance model fits perfectly on the behavior of CUDA streams on GeForce GTX 480, as it is shown in Fig. 4 (bottom). It also considers the limitations derived from the implicit synchronization that exists in devices with compute capability 2.x.

4. Testing the streams with SDK-based applications

We have tested our performance models with three applications based on codes belonging to the CUDA SDK. We have compared performances of non-streamed and streamed executions. Applying a streamed execution consists of dividing kernel execution into several stages. In this way, if a number B of thread blocks is defined in the non-streamed execution, an execution with *nStreams* streams will use $\frac{B}{nStreams}$ thread blocks in each stage.

In the last subsection, we deal with dynamically recalculating the optimal number of streams. This is applicable in those cases where the computational complexity of the kernels is dependent on the characteristics of the frames, as in histogram calculation.

4.1. Matrix multiplication

CUDA SDK includes a sample code of matrix multiplication [13]. This code performs the product of a $m \times p$ matrix A with a $p \times n$ matrix B. The result is an $m \times n$ matrix C. The code divides matrix C into 16×16 tiles and defines 16×16 blocks, so that each thread computes one element of C. The streamed configuration splits computation into *nStreams* stages. Each stream consists of copying part of matrix A to the device, computing and copying the resulting part of matrix C to the host. Matrix B has been previously transferred to the device. We have carried out five tests with $m = 512, p = 256, n = 256$; $m = 1024, p = 512, n = 512$; $m = 2048, p = 1024, n = 1024$; $m = 4096, p = 2048, n = 2048$; and $m = 8192, p = 4096, n = 4096$. Fig. 5 shows the results on GTX 280 (left) and GTX 480 (right). The suitability of our performance model is ratified in both GPUs.

In the optimal cases, the performance improvement thanks to the streams ranges between 8% and 19% for the GTX 280, and between 5% and 14% for the GTX 480. Optimal values of the number of streams can be estimated through the equations in Section 3.2. Table 2 compares the estimated optimal number of streams with the experimental optimal number of streams. It can be observed that our estimations are very close to the experimental results. There is only one anomalous estimation, which is due to the fact that applying streams reduces excessively the number of blocks that are used in each kernel launch. As we indicated in Section 2, if the number of blocks $\frac{B}{nStreams}$ is not high enough to make an extensive use of the hardware resources available on the GPU, the performance will be burdened.

4.2. 256-bins histogram

We have adapted the 256-bins histogram code in CUDA SDK [17], so that it computes the histogram of each frame belonging to a video sequence of n frames. In this way, a thread block votes in the histogram of the corresponding frame.

Three tests with different frame sizes have been carried out: 176×144, 352×288 and 704×576. The number of frames of the video sequence is $n = 64$. We proceed as it was explained in Section 2 for video processing applications. In the non-streamed execution, the histogram of each of the 64 frames is computed in one kernel invocation. The 64 frames are transferred to the GPU; then, the histograms are computed; and, finally, the 64 histograms are copied to the CPU. However, in the streamed execution, computation is divided into a number of streams. In this way, each kernel call computes the histograms of $\frac{64}{nStreams}$ frames.

Fig. 6 shows the execution results. The improvement due to the streams is between 25% and 44% for the GTX 280, and between 6% and 21% for the GTX 480. Our performance model fits the behavior of CUDA streams almost perfectly. The comparison between the estimated and the experimental optima is presented in Table 2. As it can be observed, our estimations are in the order of magnitude of the experimental optima.

Thanks to our performance models, the computation of the histograms of a video can be carried out optimally, hiding the latencies of frame transfers to the GPU and histogram transfers to the CPU. Nevertheless, the execution time is dependent on the distribution of luminance values of the pixels. In Section 4.4, we explain how a dynamical calculation of the optimal number of streams can be performed.

4.3. RGB to grayscale conversion

This application is also based on the 256-bins histogram code. It consists of converting a sequence of RGB frames to grayscale and then generating their histograms. With respect to the 256-bins histogram code, it includes more computation that will increase the kernel execution time. We have used sequences of 32 frames. Execution results are presented in Fig. 7. It can be observed that our models match the results properly. In the best cases, the improvement obtained with streams is between 52% and 63% for the GTX 280, and between 6% and 18% for the GTX 480. The estimation of the optimal number of streams is clearly correct, if we compare them to the experimental optima, as Table 2 shows.

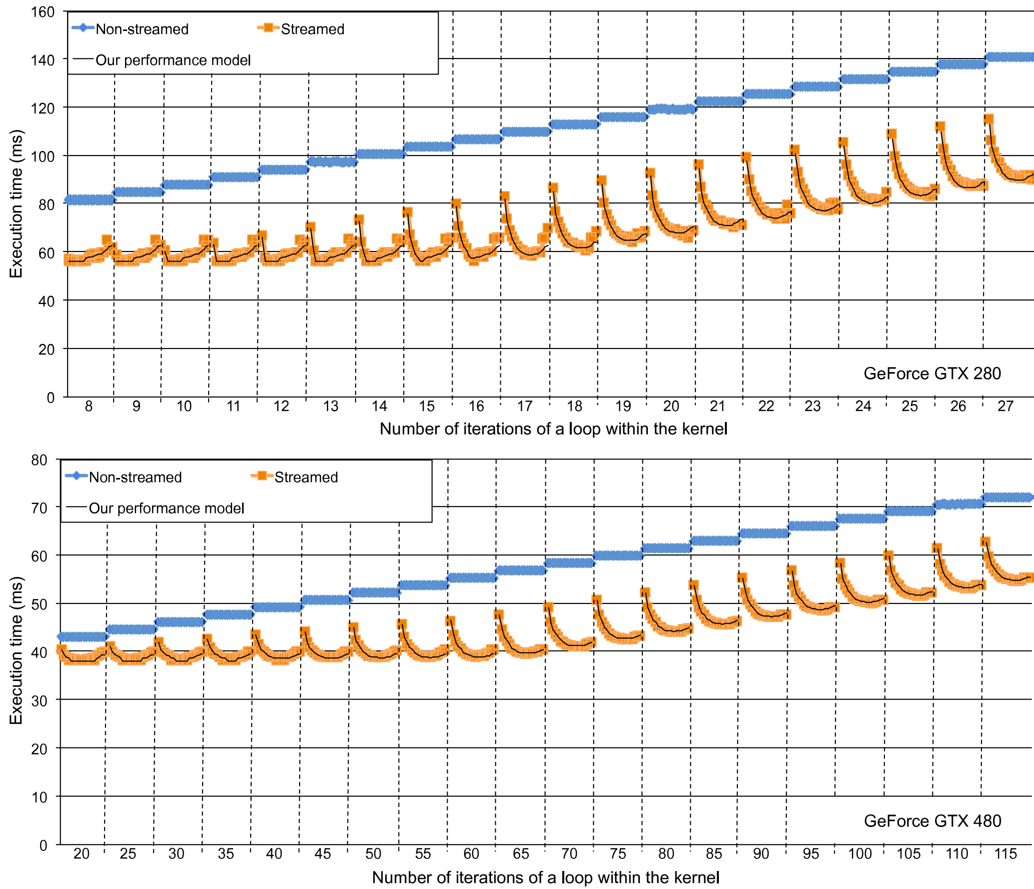

Fig. 4. Execution time (ms) for the addition of a scalar to an array of size 15 MB on GeForce GTX 280 (top) and GTX 480 (bottom). Abscissas represent the number of iterations within the kernel. The number of streams takes the divisors of 15 M between 2 and 64. The thin black line stands for our performance model. Overhead time is obtained with $t_{sc} = 0.10$ for GTX 280 and $t_{sc} = 0.03$ for GTX 480.

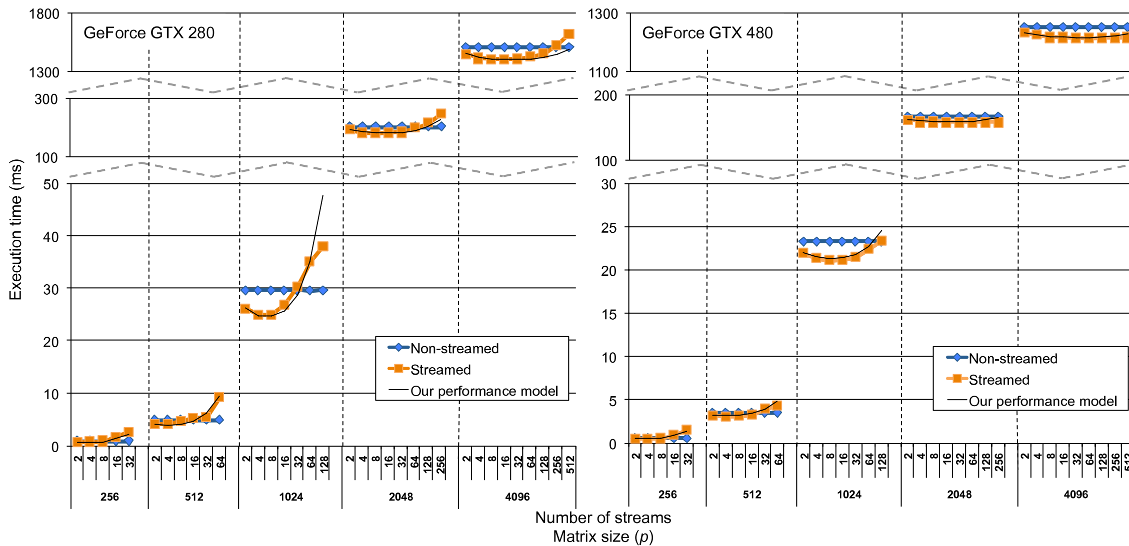

Fig. 5. Execution time (ms) for matrix multiplication on GeForce GTX 280 (left) and GeForce GTX 480 (right). Abscissas present the number of streams and the value of p. On GTX 280, overhead time is obtained with $t_{sc} = 0.10$. On GTX 480, overhead time takes $t_{sc} = 0.03$.

4.4. *Adaptation to variable kernel computation time*

A class of application that clearly can benefit from streams is signal processing, since they process long or even endless input data to generate new output data. Their computational complexity can be dependent on the input data, as the histogram computation of a video frame (see Section 4.2). Our method can be employed in these circumstances to recalculate the optimal number of streams at any moment. We have carried out an experiment in which histograms of video frames are calculated on GeForce GTX 280 and

Table 2
Estimated and experimental optimal number of streams for streamed matrix multiplication, 256-bins histogram calculation and RGB to grayscale conversion. Two values are presented when the difference between the experimental results is less than 1%.

Application	GPU	Matrix (p) or frame size	Estimated optimum	Experimental optimum
Matrix multiplication	GTX 280	256	5.4	2[a]
		512	4.3	4
		1024	6.1	4–8
		2048	12.2	8–16
		4096	24.5	16–32
	GTX 480	256	3.1	2–4
		512	6.4	4–8
		1024	12.8	8–16
		2048	25.8	16–32
		4096	51.7	32–64
256-bins histogram	GTX 280	176 × 144	2.6	2
		352 × 288	5.1	4–8
		704 × 576	9.9	8–16
	GTX 480	176 × 144	2.3	2
		352 × 288	4.5	4
		704 × 576	9.1	8–16
RGB to grayscale	GTX 280	176 × 144	3.5	4
		352 × 288	7.0	8
		704 × 576	13.9	16
	GTX 480	176 × 144	2.8	2–4
		352 × 288	5.6	4–8
		704 × 576	11.3	8–16

[a] Represents an anomalous result.

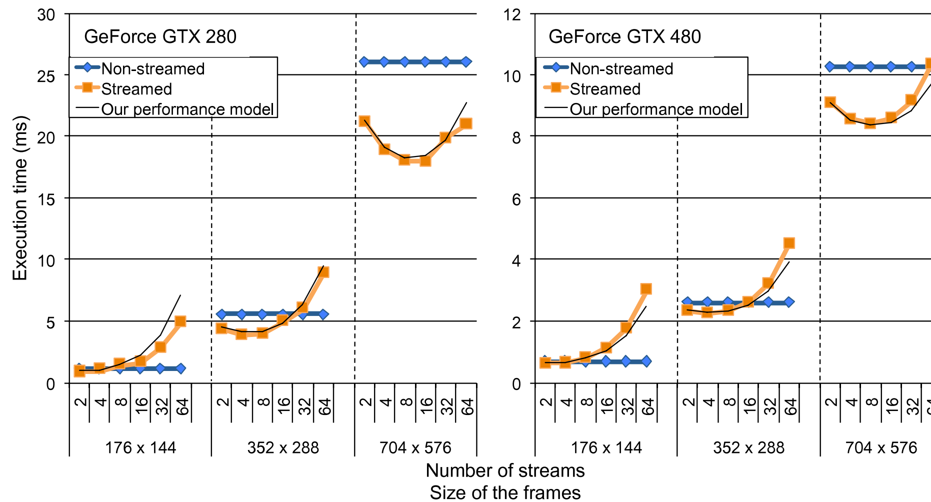

Fig. 6. Execution time (ms) for 256-bins histogram computation of 64 frames, on GeForce GTX 280 (left) and GeForce GTX 480 (right). Abscissas present the number of streams and the size of the frames. On GTX 280, overhead time is obtained with $t_{sc} = 0.10$. On GTX 480, overhead time takes $t_{sc} = 0.03$.

Table 3
Number of frames per second for histogram calculation of a video sequence, on GTX 280 and GTX 480. Frames are size 352 × 288 or 704 × 576.

GPU	Frame size	Frames per second	
		Non-streamed execution	Optimally streamed execution
GTX 280	352 × 288	5770	6502
	704 × 576	1401	1656
GTX 480	352 × 288	8469	9149
	704 × 576	2153	2429

GTX 480. We take advantage of the distribution of color pixels, and consequently the computation time, is normally very similar in consecutive frames. Only in shot transitions (cuts, dissolves and so on) this distribution can change abruptly.

A sequence of 4096 frames is divided into chunks of 32 frames. In the first half of the sequence, frames have uniform distribution of the luminance values. Frames of the second half present a degenerate distribution. In this way, histogram calculation of

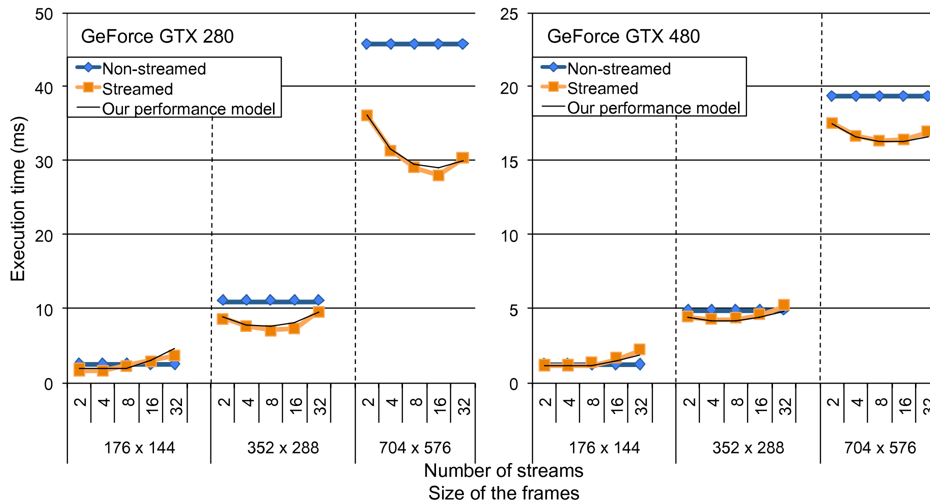

Fig. 7. Execution time (ms) for RGB to grayscale conversion of 32 frames, on GeForce GTX 280 (left) and GeForce GTX 480 (right). Abscissas present the number of streams and the size of the frames. On GTX 280, overhead time is obtained with $t_{sc} = 0.10$. On GTX 480, overhead time takes $t_{sc} = 0.03$.

the frames of the second half presents more collisions between threads, so that the execution time in this half is expected to be much longer. The first chunk is processed in a non-streamed way, in order to obtain the estimated time and the optimal number of streams. The estimated time is continuously compared to an on-the-fly measurement of the streamed execution time for each upcoming chunk. If both diverge over a certain threshold, the optimum is readily recalculated by executing again only one chunk in a non-streamed way. Table 3 summarizes the execution results for non-streamed and optimally streamed histogram calculation for the whole sequence. As it can be seen, the number of frames per second is clearly increased by using an automatically calculated optimal number of streams for each half of the video sequence.

5. Conclusions

Despite that GPUs are nowadays being successfully used as massively parallel coprocessors in high performance computing applications, the fact that data must be transferred between two separate address spaces (memories of CPU and GPU) constitutes a communication overhead. This can be reduced by using asynchronous transfers, if computation is properly divided into stages. CUDA provides streams for performing a staged execution, which allows programmers to overlap communication and computation. Although exploiting such a concurrency can achieve an important performance improvement, CUDA literature barely gives rough estimates, which do not steer towards the optimal manner to break up computation.

In this work, we have exhaustively analyzed the behavior of CUDA streams through a novel methodology, in order to define precise estimates for streamed executions. In this way, we have found two mathematical models which accurately characterize the performance of CUDA streams on consumer NVIDIA GPUs with compute capabilities 1.*x* and 2.*x*. Through these models, we have found specific equations for determining the optimal number of streams, once kernel execution and data transfers times are known. Although results in this paper have been presented for GeForce GTX 280 and GTX 480, our performance models have also been validated on other NVIDIA GPUs from the GeForce 8, 9, 200, 400 and 500 series [3].

We have successfully tested our approaches with three applications based on codes from CUDA SDK. Our performance models have matched the experimental results, as well as the estimated optima have resulted in the order of magnitude of the experimental ones.

Since some applications, such as histogram calculation, are workload-dependent, our method can be used for a dynamical calculation of the optimal number of streams. An on-the-fly analysis of the streamed execution time, checking if it diverges from the estimate over a certain threshold, will permit recalculating the optimum.

References

[1] Sara S. Baghsorkhi, Matthieu Delahaye, Sanjay J. Patel, William D. Gropp, Wen-mei W. Hwu, An adaptive performance modeling tool for GPU architectures, in: Proceedings of the 15th ACM SIGPLAN Smposium on Principles and Practice of Parallel Programming, PPoPP'10, ACM, New York, NY, USA, 2010, pp. 105–114.

[2] Juan Gómez-Luna, José María González-Linares, José Ignacio Benavides, Nicolás Guil, Parallelization of a video segmentation algorithm on CUDA-enabled graphics processing units, in: Proc. of the Int'l Euro-Par Conference on Parallel Processing, EuroPar'09, 2009, pp. 924–935.

[3] Juan Gómez-Luna, José María González-Linares, José Ignacio Benavides, Nicolás Guil, Performance models for CUDA streams on NVIDIA GeForce series, Technical Report, University of Málaga, 2011. http://www.ac.uma.es/~vip/publications/UMA-DAC-11-02.pdf.

[4] Wan Han, Gao Xiaopeng, Wang Zhiqiang, Li Yi, Using GPU to accelerate cache simulation, in: IEEE International Symposium on Parallel and Distributed Processing with Applications, 2009, pp. 565–570.

[5] Sunpyo Hong, Hyesoon Kim, An analytical model for a GPU architecture with memory-level and thread-level parallelism awareness, in: Proceedings of the 36th Annual International Symposium on Computer Architecture, ISCA'09, ACM, New York, NY, USA, 2009, pp. 152–163.

[6] Amir H. Hormati, Mehrzad Samadi, Mark Woh, Trevor Mudge, Scott Mahlke, Sponge: portable stream programming on graphics engines, in: Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS'11, ACM, New York, NY, USA, 2011, pp. 381–392.

[7] Khronos Group, OpenCL. http://www.khronos.org/opencl/.

[8] Supada Laosooksathit, Chokchai B. Leangsuksun, Abdelkader Baggag, Clayton F. Chandler, Stream experiments: toward latency hiding in GPGPU, in: Proceedings of the 9th IASTED International Conference on Parallel and Distributed Computing and Networks, PDCN'10, 2010, pp. 240–248.

[9] Vladimir Marjanović, Jesús Labarta, Eduard Ayguadé, Mateo Valero, Overlapping communication and computation by using a hybrid MPI/SMPSS approach, in: Proceedings of the 24th ACM International Conference on Supercomputing, ICS'10, ACM, New York, NY, USA, 2010, pp. 5–16.

[10] MPI Forum, The Message Passing Interface Standard. http://www.mpi-forum.org/.

[11] NVIDIA, CUDA C best practices guide 3.2, August 2010. http://developer.download.nvidia.com/compute/cuda/3_2/toolkit/docs/CUDA_C_Best_Practices_Guide.pdf.

[12] NVIDIA, CUDA C programming guide 3.2, September 2010. http://developer.download.nvidia.com/compute/cuda/3_2/toolkit/docs/CUDA_C_Programming_Guide.pdf.

[13] NVIDIA, CUDA SDK code samples: matrix multiplication. http://developer.download.nvidia.com/compute/cuda/sdk/website/samples.html#matrixMul.
[14] NVIDIA, CUDA Zone. http://www.nvidia.com/object/cuda_home_new.html.
[15] Peripheral Component Interconnect Special Interest Group, PCI Express. http://www.pcisig.com/.
[16] James C. Phillips, John E. Stone, Klaus Schulten, Adapting a message-driven parallel application to GPU-accelerated clusters, in: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing, SC'08, IEEE Press, Piscataway, NJ, USA, 2008, pp. 8:1–8:9.
[17] V. Podlozhnyuk, Histogram calculation in CUDA, White Paper, 2007. http://developer.download.nvidia.com/compute/cuda/1_1/Website/projects/histogram256/doc/histogram.pdf.
[18] Abhishek Udupa, R. Govindarajan, Matthew J. Thazhuthaveetil, Synergistic execution of stream programs on multicores with accelerators, in: Proceedings of the 2009 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems, LCTES'09, ACM, New York, NY, USA, 2009, pp. 99–108.
[19] Ta Quoc Viet, Tsutomu Yoshinaga, Improving linpack performance on SMP clusters with asynchronous MPI programming, IPSJ Digital Courier 2 (2006) 598–606.
[20] Yao Zhang, John D. Owens, A quantitative performance analysis model for GPU architectures, in: Proceedings of the 17th IEEE International Symposium on High-Performance Computer Architecture, HPCA 17, February 2011.

Juan Gómez-Luna received his B.S. degree in Telecommunication Engineering from the University of Sevilla, Spain, in 2001. He is currently pursuing his Ph.D. degree in Computer Science at the University of Córdoba, Spain. Since 2005, he is assistant professor at the University of Córdoba. His research interests focus on parallelization of image and video processing applications.

Jose María González-Linares received his B.S. degree in Telecommunication Engineering from the University of Málaga, Spain, in 1995, and his Ph.D. in Telecommunication Engineering from the University of Málaga, Spain, in 2000. During 1998–2002 and 2002–2010 he was assistant and associate professor at the University of Málaga. He has published more than 20 papers in international journals and conferences. His research interests are in the areas of parallel computing and video and image processing.

Jose Ignacio Benavides received his bachelor's degree in Physics from the University of Granada, Spain, in 1980 and his Ph.D. degree in Physics from the University of Santiago of Compostela, Spain, in 1990. From 1980 to 1983 he was an assistant professor at the University of Granada. He joined, as a full professor, the University of Córdoba in 1983. He is currently the head of Department of Computer Architecture and Electronic of the University of Córdoba. He has published more than 50 papers in international journals and conferences. His research interests are in the areas of parallel computing and video and image processing.

Nicolás Guil received his B.S. in Physics from the University of Sevilla, Spain, in 1986 and his Ph.D. in Computer Science from the University of Málaga in 1995. During 1990–1997 and 1998–2006 he was assistant and associate professor at the University of Málaga. Currently, he is full professor with the Department of Computer Architecture in the University of Málaga. He has published more than 50 papers in international journals and conferences. His research interests are in the areas of parallel computing and video and image processing.