Performance models for CUDA streams on NVIDIA GeForce series

Juan Gómez-Luna[◇]
José Ignacio Benavides[◇]
[◇]Dept. of Computer Architecture and Electronics
University of Córdoba

José María González-Linares[⋆]
Nicolás Guil[⋆]
[⋆]Dept. of Computer Architecture
University of Málaga

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. CUDA Application Programming Interface (API) provides asynchronous transfers and *streams*, which permit a staged execution, as a way to overlap communication and computation. Nevertheless, it does not exist a precise manner to estimate the possible improvement due to overlapping, 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 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 successfully checked the suitability of our performance models on several NVIDIA devices belonging to GeForce 8, 9, 200, 400 and 500 series.

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) [1] 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 [6, 11].

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) [7] and the Open Computing Language (OpenCL) [3]. 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) [4] bus and, after the computation, results are got to the CPU back. Since its first release, the CUDA API provides a function, called `cudaMemcpy()` [9], that transfers data between host and 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 `cudaMemcpyAsync()` [9], which permits asynchronous transfers, i.e. enables overlap of data transfers with computation, in devices with compute capability equal or higher than 1.1 [9]. 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.

Some research works have made use of streams, in order to improve applications performance [2, 5, 10]. However, finding optimal configurations, i.e. the best number of streams or stages in which transfers and computation are divided, required many attempts for tuning the application. Moreover, CUDA literature [8, 9] does not provides an accurate way to estimate the performance improvement due to the use of streams.

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 the 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 are able to derive two performance models, i.e. the one for devices with c.c. 1.x and the other for devices with c.c. 2.x. These models clearly fit the observed performance of the streams

on those NVIDIA GPUs that permit overlapping of communication and computation.

- 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 suitability of our models on several NVIDIA GPUs with c.c. 1.x and 2.x.

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 validated in Section 4 on several NVIDIA GPUs belonging to GeForce 8, 9, 200, 400 and 500 series. Finally, conclusions are stated in Section 5.

2 CUDA streams

In order to overlap communication and computation, CUDA permits to divide 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.

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, memory copy of one stream overlaps kernel execution of other stream, achieving a performance improvement. In [8], such a concurrency between communication and computation is depicted as in Figure 1 with $nStreams = 4$.

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. In other case the sequential execution could be faster than the streamed one.

The following code declares and creates 4 streams [9]:

```
cudaStream_t stream[4];
for (int i = 0; i < 4; ++i)
      cudaStreamCreate(&stream[i]);
```

Then, each stream transfers its portion of host input array, which should have been allocated as page-locked memory, to the device input array, processes this input on the device and transfers the result back to the host:

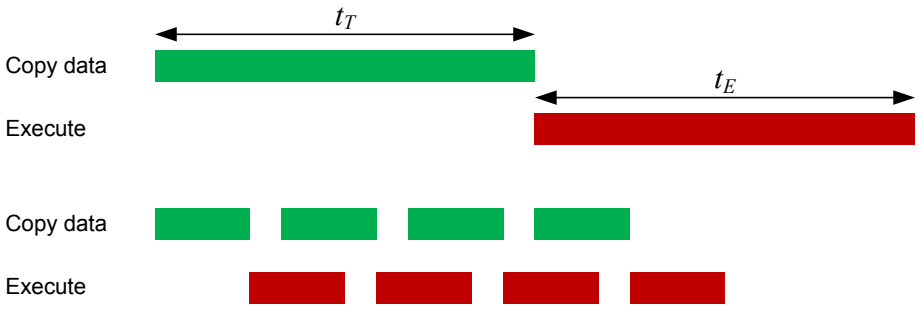

Figure 1: Comparison of timelines for sequential (top) and concurrent (bottom) copy and kernel execution, as presented in [8]. t_T means data transfer time and t_E kernel execution time.

```
for (int i = 0; i < 4; ++i)
      cudaMemcpyAsync(inputDevPtr + i * size,
            hostPtr + i * size, size,
            cudaMemcpyHostToDevice, stream[i]);
for (int i = 0; i < 4; ++i)
      MyKernel<<<num_blocks / 4, num_threads, 0,
            stream[i]>>> (outputDevPtr + i * size,
            inputDevPtr + i * size, size);
for (int i = 0; i < 4; ++i)
      cudaMemcpyAsync(hostPtr + i * size,
            outputDevPtr + i * size, size,
            cudaMemcpyDeviceToHost, stream[i]);
cudaThreadSynchronize();
```

As it can be seen, 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 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 frames transfers, as Figure 2 shows.

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 [8], 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

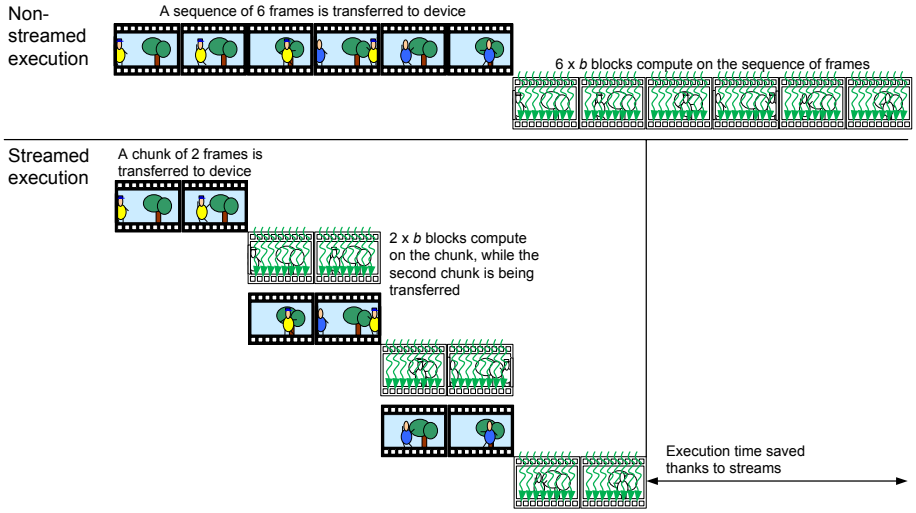

Figure 2: Computation on a sequence of 6 frames for non-streamed and streamed execution. In the streamed execution, frames are transferred and computed in chunks of size 2, what permits to hide part of the transfers

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*.

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 of the kernel presented in the following lines. This is a simple code in which a scalar `*factor` is repeatedly added to an array, that represents a vector. The variable `num_iter` defines the number of times that `*factor` is added to each element of the array.

```
__global__ void init_array (int *g_data, int *factor, int num_iter)
{
      int idx = blockIdx.x*blockDim.x+threadIdx.x;
      for(int i=0; i<num_iter; i++)
            g_data[idx] += *factor;
}
```

`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 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, what 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.

Details about NVIDIA devices are presented in Table 1. As stated in [9], 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 Mbytes, using the modified `simpleStreams.cu`. The number of iterations within the

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

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.

Figure 3 shows the execution results on the GeForce GTX 280. With the aim of facilitating the understanding of the results, this Figure only shows a blue line with diamond markers, which presents the non-streamed execution results, and an orange line with square markers, which 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. In Figure 4, together with the execution times for non-streamed and streamed configurations, two thick lines and two thin lines are depicted. 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 [8]. The red thin 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 green thin 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 (green thick 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 green thick line and the red thin line intersect. In this point, the dominant kernel estimate equals the transfers time. In this way, a reference for the optimal number of streams is $nStreams = \frac{t_T}{t_T - t_E}$.

On the GeForce GTX 480, the dominant transfers hypothesis suits properly

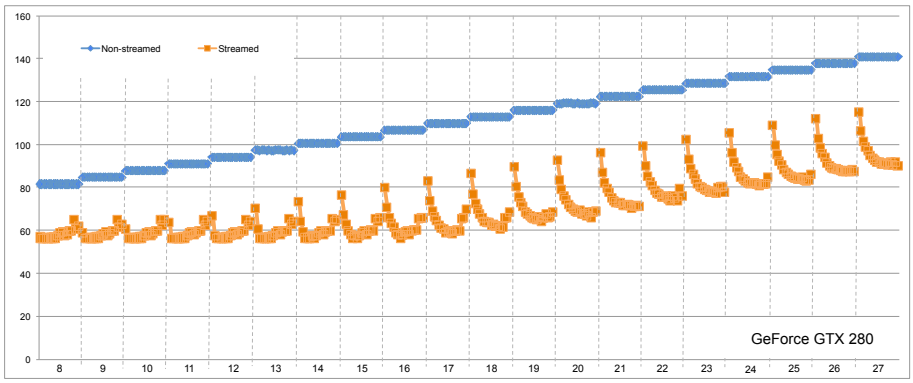

Figure 3: Execution time (ms) for the addition of a scalar to an array of size 15 Mbytes on GeForce GTX 280. The blue line represents the execution time for non-streamed executions. The orange line stands for the results of the streamed execution. Each column in the graph represents a test with a changing number of iterations between 8 and 27 in steps of 1. These numbers are shown in abscissas. The number of streams has been changed along the divisors of 15 M between 2 and 64. Thus, in each column, one marker of the orange line represents the execution time using a certain number of streams

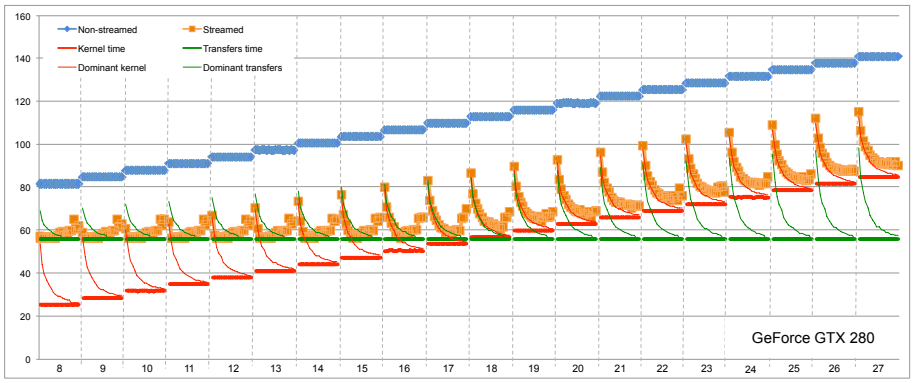

Figure 4: Execution time (ms) for the addition of a scalar to an array of size 15 Mbytes on GeForce GTX 280. Each column in the graph represents a test with a changing number of iterations between 8 and 27 in steps of 1. In each column, the number of streams has been changed along the divisors of 15 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 [8]

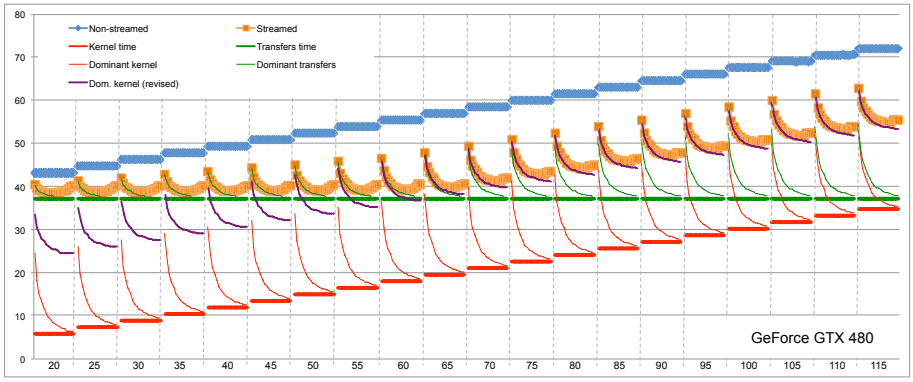

Figure 5: Execution time (ms) for the addition of a scalar to an array of size 15 Mbytes on GeForce GTX 480. Each column in the graph represents a test with a changing number of iterations between 20 and 115 in steps of 5. In each column, the number of streams has been changed along the divisors of 15 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 [8]. Thin purple line stands for a revised dominant kernel model, in which only one of the transfers is hidden

on the left of the graph. However, the dominant kernel hypothesis does not fit in any case. Figure 5 shows that a revised dominant kernel hypothesis (purple thin line), in which the streams hide only one of the data transfers, matches better. 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 both 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 Mbytes 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 Mbytes from host to device and 23 Mbytes from device to host, and in the last test 23 Mbytes are copied from host to device and 1 Mbytes from device to host. The number of streams has been established in 16 for every test.

Figure 6 shows the results on the GeForce GTX 280. It can be observed that the streamed results match the transfers time, 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 is 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 subsection 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 Figure 7. It depicts two revised 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 later 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 green thin 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 $t_T = t_{Thd} + t_{Tdh}$, 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. $t_{oh} = t_{sc} \times nStreams$. The value of t_{sc} should be estimated for each GPU. In Section 4, the value of t_{sc} for several GPUs are given.

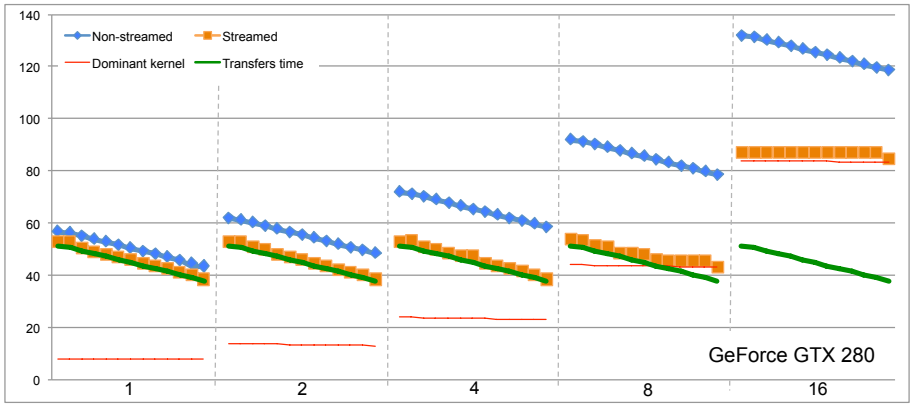

Figure 6: Execution time (ms) on GeForce GTX 280 for tests with asymmetric transfers. 24 Mbytes are copied from host to device or from device to host. Abscissas represent the number of iterations within the kernel, which is a power of two between 1 and 16. In each column, 13 tests are represented with 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. The red thin line stands for a dominant kernel hypothesis and the green thick line is the transfers time

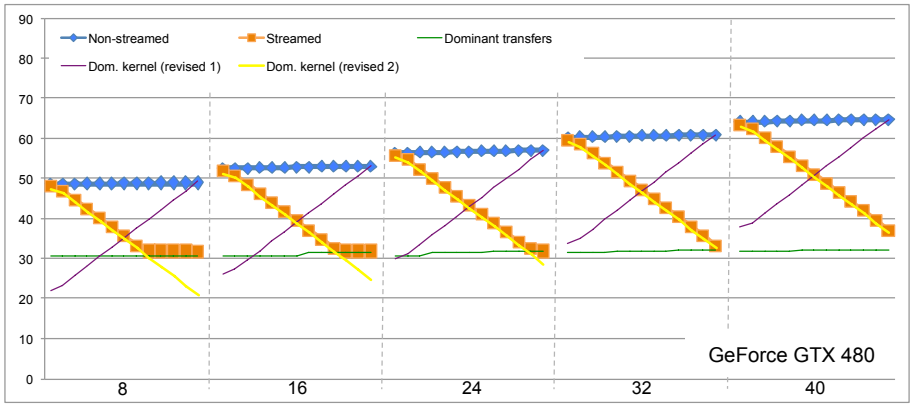

Figure 7: Execution time (ms) on GeForce GTX 480 for tests with asymmetric transfers. 24 Mbytes are copied from host to device or from device to host. Abscissas represent the number of iterations within the kernel, which changes between 8 and 40 in steps of 8. In each column, 13 tests are represented 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. The green thin line stands for the dominant transfers hypothesis. Purple and yellow thin lines represent two revisions of the dominant kernel estimate

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:

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

In subsection 3.1.1, we noticed that the optimal number of streams $nStreams_{op}$, with dominant transfers time, 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 $nStreams$:

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

Deriving equation 3 permits to obtain the optimal number of streams in a dominant kernel case:

$$nStreams_{op} = \sqrt{\frac{t_T}{t_{sc}}} \tag{4}$$

3.2.2 Performance on devices with compute capability 2.x

In subsection 3.1.1, we observed that on GTX 480 a dominant transfers scenario was properly defined as in [8]. Moreover, from subsection 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:

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

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

$$nStreams_{op} = \sqrt{\frac{t_E}{t_{sc}}} \tag{6}$$

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

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

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

$$nStreams_{op} = \sqrt{\frac{t_{Thd}}{t_{sc}}} \tag{8}$$

As it can be observed, this performance model considers the limitations derived from the implicit synchronization that exists in devices with compute capability 2.x.

Table 2: Features of NVIDIA GeForce GPUs used in this work

Parameter	8800GTS512	9800GX2	GTX260	GTX280	GTX480	GTX580
Series	8	9	200	200	400	500
Codename	G92-400	G92	GT200	GT200	GF100	GF110
Compute capability	1.1	1.1	1.3	1.3	2.0	2.0
PCIe	2.0 ×16	2.0 ×16	2.0 ×16	2.0 ×16	2.0 ×16	2.0 ×16
Overlapping of data transfer and kernel execution	✓	✓	✓	✓	✓	✓
Concurrent kernel execution	×	×	×	×	✓	✓

Table 3: Values of t_{sc} for devices in Table 2

	8800GTS512	9800GX2	GTX260	GTX280	GTX480	GTX580
t_{sc}	0.30	0.10	0.10	0.10	0.03	0.01

4 Validation of our performance models

In this Section, we validate the performance models presented in subsection 3.2 on several devices with compute capabilities 1.x and 2.x, belonging to NVIDIA GeForce 8, 9, 200, 400 and 500 series. Characteristics of these devices are shown in Table 2.

Figures 8 to 13 show the suitability of our performance models. Streams on those devices with compute capability 1.x fit the equations in subsection 3.2.1 and devices with compute capability 2.x, the equations in subsection 3.2.2.

In subsection 3.2, we indicated that the overhead time (t_{oh}) is obtained as a linear function of the number of streams. We consider the constant t_{sc} as the time needed to create one stream. Table 3 lists the values of t_{sc} that we have estimated for each GPU.

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

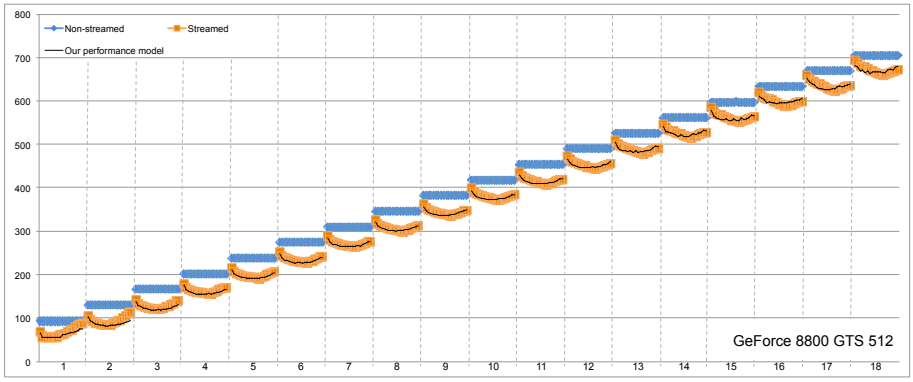

Figure 8: Execution time (ms) for the addition of a scalar to an array of size 15 Mbytes on GeForce 8800 GTS 512. The number of iterations changes between 1 and 18 in steps of 1. The number of streams takes the divisors of 15 M between 2 and 64. Black thin line stands for our performance model. Overhead time is obtained with $t_{sc} = 0.30$

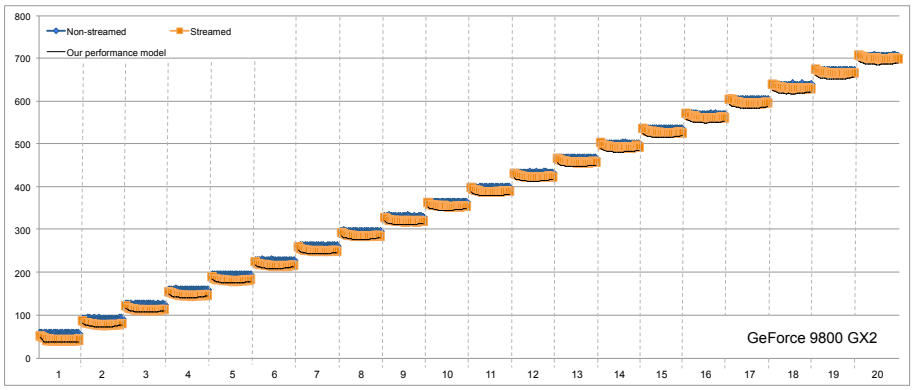

Figure 9: Execution time (ms) for the addition of a scalar to an array of size 15 Mbytes on GeForce 9800 GX2. The number of iterations changes between 1 and 20 in steps of 1. The number of streams takes the divisors of 15 M between 2 and 64. Black thin line stands for our performance model. Overhead time is obtained with $t_{sc} = 0.10$

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 illustrated on GeForce GTX 280 and GTX 480, our performance models have also been validated on other NVIDIA GPUs from GeForce 8, 9, 200, 400 and 500 series.

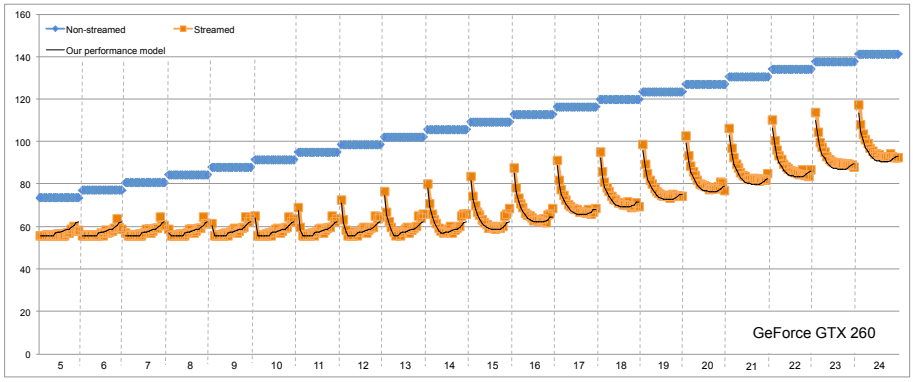

Figure 10: Execution time (ms) for the addition of a scalar to an array of size 15 Mbytes on GeForce GTX 260. The number of iterations changes between 5 and 24 in steps of 1. The number of streams takes the divisors of 15 M between 2 and 64. Black thin line stands for our performance model. Overhead time is obtained with $t_{sc} = 0.10$

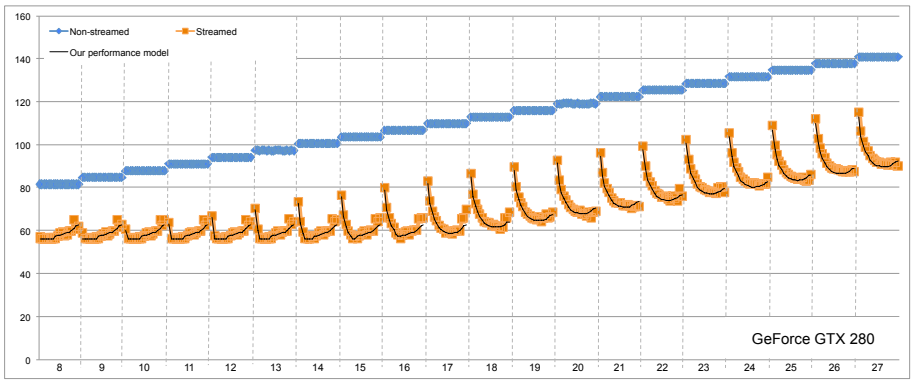

Figure 11: Execution time (ms) for the addition of a scalar to an array of size 15 Mbytes on GeForce GTX 280. The number of iterations changes between 8 and 27 in steps of 1. The number of streams takes the divisors of 15 M between 2 and 64. Black thin line stands for our performance model. Overhead time is obtained with $t_{sc} = 0.10$

References

[1] MPI Forum. The Message Passing Interface standard. http://www.mpi-forum.org/.

[2] Juan Gómez-Luna, José María González-Linares, José Ignacio Benavides, and Nicolás Guil. Parallelization of a video segmentation algorithm on

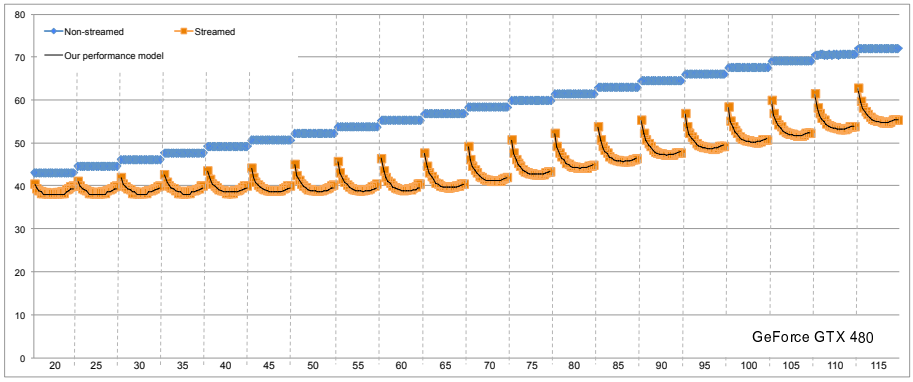

Figure 12: Execution time (ms) for the addition of a scalar to an array of size 15 Mbytes on GeForce GTX 480. The number of iterations changes between 20 and 115 in steps of 5. The number of streams takes the divisors of 15 M between 2 and 64. Black thin line stands for our performance model. Overhead time is obtained with $t_{sc} = 0.03$

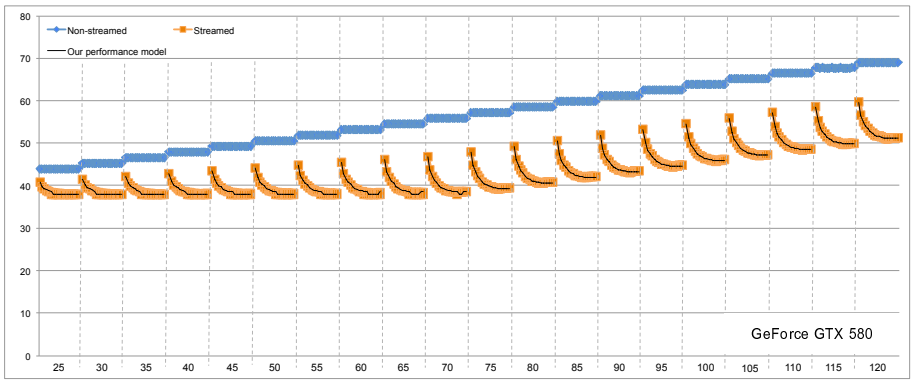

Figure 13: Execution time (ms) for the addition of a scalar to an array of size 15 Mbytes on GeForce GTX 580. The number of iterations changes between 25 and 120 in steps of 5. The number of streams takes the divisors of 15 M between 2 and 64. Black thin line stands for our performance model. Overhead time is obtained with $t_{sc} = 0.01$

CUDA–enabled graphics processing units. In *Proc. of the Int'l Euro-Par Conference on Parallel Processing (EuroPar'09)*, pages 924–935, 2009.

[3] Khronos group. OpenCL. http://www.khronos.org/opencl/.

[4] Peripheral Component Interconnect Special Interest Group. PCI Express. http://www.pcisig.com/.

[5] Wan Han, Gao Xiaopeng, Wang Zhiqiang, and Li Yi. Using gpu to accelerate cache simulation. In *IEEE International Symposium on Parallel and Distributed Processing with Applications*, pages 565–570, 2009.

[6] Vladimir Marjanović, Jesús Labarta, Eduard Ayguadé, and 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, pages 5–16, New York, NY, USA, 2010. ACM.

[7] NVIDIA. CUDA Zone. http://www.nvidia.com/object/cuda_home_new.html.

[8] NVIDIA. CUDA C Best Practices Guide 3.2. http://developer.download.nvidia.com/compute/cuda/3_2/toolkit/docs/CUDA_C_Best_Practices_Guide.pdf, August 2010.

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

[10] James C. Phillips, John E. Stone, and Klaus Schulten. Adapting a message-driven parallel application to gpu-accelerated clusters. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, SC '08, pages 8:1–8:9, Piscataway, NJ, USA, 2008. IEEE Press.

[11] Ta Quoc Viet and Tsutomu Yoshinaga. Improving linpack performance on smp clusters with asynchronous mpi programming. *IPSJ Digital Courier*, 2:598–606, 2006.