Towards a more efficient use of GPUs

Pedro Valero[1] and Fernando L. Pelayo[2]

[1] University of Castilla-La Mancha, Albacete Research Institute of Informatics, Pedro.ValeroLara@uclm.es
[2] University of Castilla-La Mancha, Computing Systems Department, FernandoL.Pelayo@uclm.es, Avda. España s/n, 02071-Albacete (Spain).

Abstract. This paper presents a new proposal of use for GPUs in which more than one job can be executed simultaneously in a single GPU. The requirements for this are provided and a performance evaluation for such a new scenario is also presented.
Finally the results of these performance measurements are analyzed and the paper is concluded with some key guidelines for getting the most of these devices under the performance point of view.

Introduction

Current GPUs are an appropriate parallel platform in order to accelerate any application, due to that, these devices have the highest ratio performance/cost with a high number of fragment processors (e.g. 240-512) and high memory bandwidth.

GPUs can execute millions of threads at the same time however, they only can execute one job or application at the same time. For example, if one job would need 512 threads, and some GPU can execute 35,000,000 threads, the job will only use a 0.001% of the total capacity of the GPU, being a use of GPU very inefficient. For that reason, we have modified the traditional method of using GPUs in order to be able to execute more than one job at the same time. We have compared the traditional method and this new proposal, mainly from performance the point of view.

Nowadays, GPUs are the devices, with the best performance/cost ratio. In fact, current GPUs are being used in the most of the youngest HPC environments, e.g., there are three supercomputing systems in the top four of top 500[17] list which use GPUs , in the first, third and fourth places. Moreover, GPUs are also being introduced in both GRID[7] and CLOUD[6] environments in a massive way. In these environments the GPUs are used in a large different fields, like science, economy, medicine, astronomy, engineering, etc. [6,7,8,9,10,11,12,13,14,15,16].

The paper is structured as follows: Section 1 describes the main characteristic of the current GPUs. Section 2 presents a new proposal for a use more efficient of GPUs, moreover, section 3 shows experimental results and performance analysis over the proposed programming philosophy. Finally, Section 4 outlines the conclusions and future work.

1 GPU

GPUs are traditionally used for interactive applications, and are designed to achieve high rasterization performance however, their characteristics have allowed the opportunity to other more general applications to be accelerated in GPU-based platforms. This trend is now called General Purpose Computing on GPU (GPGPU) [1], or what is the same, the usage of GPUs for applications for which they were not originally designed. These general applications must have parallel characteristics and an intense computational load to obtain a good performance.

As mentioned, the main feature of these devices is a large number of processing elements integrated into a single chip at the expense of a significant reduction in cache memory. These processing elements are arranged on memory cards that have a local high-speed external DRAM and are connected to the computer through a high-speed I/O interface (PCI Express).

The era of single-threaded processor performance increasing has come to an end. Programs will only increase in performance if they utilize parallelism. However, there are different kinds of parallelism. For instance, multicore CPUs provide task-level parallelism, on other hand, Graphics Processing Units (GPUs) provide data-level parallelism.

1.1 GPU Architecture

Current GPUs consist of a high number (e.g. 8-480) of fragment processors with high memory bandwidth. They can offer 10x higher main memory bandwidth and they can use data parallelism to achieve up to 10x more floating point throughput than CPUs [2].

A GPU architecture consist of a number of multiprocessors (e.g. 1-30), each of then having a number cores between 8 and 32. All multiprocessors share the same memory, this memory is called “global memory”. On other hand, all cores of one multiprocessor can access to the same memory, this memory is called “shared memory”. This memory only is useful when many threads of the same multiprocessor have to access to the same data many times; this is because, in order to load a piece of information in shared memory it is necessary to access to global memory. The classical GPU architecture is shown in figure 1.

1.2 CUDA

In previous years, there were many attempts to use graphics-oriented languages, see [3,4], in order to accelerate specific parts of code using GPUs. More recently, the GPU manufacturers, like NVIDIA or ATI, have proposed new languages or even extensions for the most common used high level programming languages. As example, NVIDIA proposes CUDA [5], which is a software platform for massively parallel high-performance computing on NVIDIA’s powerful GPUs.

In CUDA, the calculations are distributed in a mesh or grid of thread blocks, all thread blocks have the same size (number of threads). These threads run

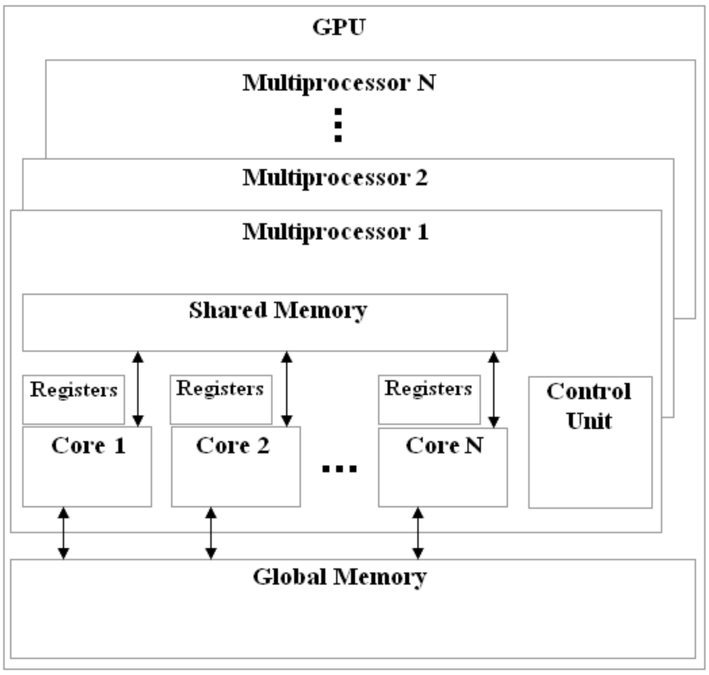

Fig. 1. GPU Architecture.

the GPU code, known as kernel; we'd like notice that although this kernel is originally called by the CPU, finally it is executed in the GPU, as seen in figure 2. The dimensions of both the mesh and the threads blocks, should be carefully chosen in order to achieve the maximum performance depending on the specific problem being treated.

The *NVIDIA*'s *CUDA* Programming Model considers the GPU as a computational device able to execute a high number of parallel threads. CUDA includes *C/C++* software development tools, function libraries, and a hardware abstraction mechanism that hides the GPU hardware from developers such as an Application Programming Interface (API). Among the main tasks to be done in CUDA, they can be founded, allocating data on the GPU, transferring data between the GPU and the CPU and vice versa, and launching kernels.

A CUDA kernel executes a sequential code in a large number of threads in parallel. The threads are arranged in a grid of blocks CUDA (figure 2). The threads within a block can work together efficiently exchanging data via a local shared memory and synchronize low-latency execution through synchronization barriers (where threads in a block are suspended until they all reach the synchronization point). By contrast, the threads of different blocks in the same network can only coordinate their implementation through a high-latency accesses to global memory (the memory graphic board). Each block is executed in a different multiprocessor. Within limits, the programmer specifies the number of blocks and the number of threads per block to be allocated, in the implementation of a given kernel.

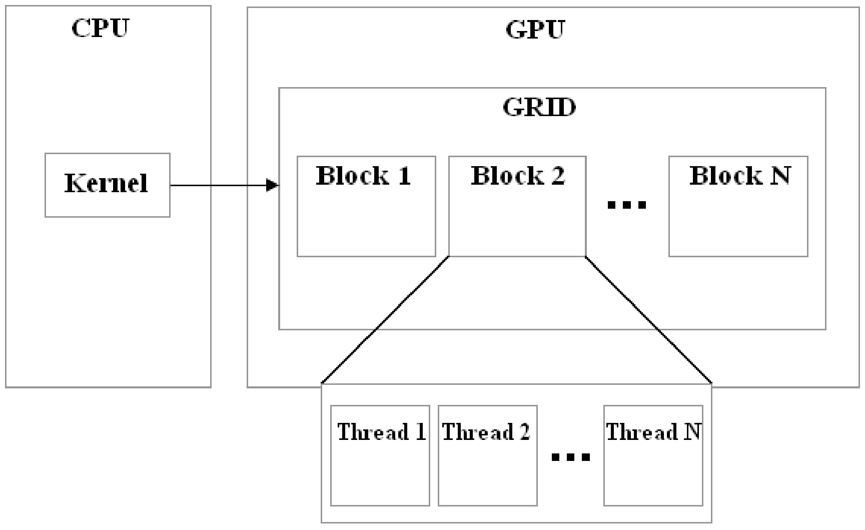

Fig. 2. Grid of CUDA blocks.

All CUDA code is divided in two different parts. The first is the CPU code, this code provides the instructions to be performed by the CPU, e.g. allocating data on the CPU and GPU, transferring data between the GPU and CPU (and vice versa) and launching kernels. Moreover, the GPU code (kernel) provides the instructions to be executed in the GPU, by all threads of the kernel.

All CPU codes have the same steps, so ordered:

1. Allocating data on CPU
2. Allocating data on GPU
3. Transferring data form CPU to GPU
4. Launching kernels
5. Transferring data between from GPU to CPU

2 A new proposal for a more efficient use of GPUs

The new proposal consists in executing more than one job at the same time. From the characteristics of the current GPUs and new software tools like CUDA, allows this proposal to be implemented, because each multiprocessor has its own instructions control unit, show all CUDA codes have the same steps and each threads block is executed in a single multiprocessor. These characteristics allow us to execute so many jobs as blocks can be executed by the GPU.

Algorithm 1 CUDA pseudocode example 1.

```
   Function CPUJob1                                         ▷ CPU Code
1: CPUAllocate(A-CPU)                                       ▷ Allocate data on CPU
2: CPUAllocate(B-CPU)
3: GPUAllocate(A-GPU)                                       ▷ Allocate data on GPU
4: GPUAllocate(B-GPU)
5: CPU-GPUTransfer(A-CPU,A-GPU)                             ▷ Tranfer data from CPU to GPU
   (Parameters)
6: KernelJob1(A-GPU,B-GPU)                                  ▷ Launch kernel
7: GPU-CPUTransfer(B-GPU,B-CPU)                             ▷ Transfer data from GPU and CPU
   (Results)
   Function kernelJob1(A,B)                                 ▷ GPU Code
8: i = index of thread
9: B[i] = A[i] + 100
```

Executing more than one job at the same time force to join all kernels codes, thus each kernel code will be indexed in one or a set of blocks. Therefore, each job is executed independently and simultaneously with others.

Several conditions are necessary to execute more than one job at the same time:

– Each job must have its own steps (1-3) and 5 (subsection 1.2).
– A single kernel must contain all kernel of the jobs indexed by blocks.
– The jobs must be independent, so that, each job must have its own parameters and results.

In order to indicate the changes made to execute more than one job at the same time, we show two different examples (algorithms 1 and 2) which describe the traditional way of using a GPU. Finally, algorithm 3 is formed of the union of algorithms 1 and 2 for the sake of executing the two kernels of these algorithms at the same time. The three algorithms show the two parts of the code, the part executed by CPU (CPUJob), and that executed by GPU (KernelJob).

3 Performance evaluation

Once described in previous section our proposed for improving the performances of GPUs, it is time to run out these proposals and to analyze the real, or not, advantages of them.

In this section, we develop the performance evaluation. We have used two different GPU-based platforms (9600 GT and GTX 285), with different characteristics (table 1), in order to obtain the factor or factors to guess the main reasons or causes of the results achieved. We would like to emphasize that we haven't used shared memory because, as we explained in the section 1, the use of shared memory is efficient just in certain cases, as those in which more than one thread of the same threads block have to access to the same positions of

Algorithm 2 CUDA pseudocode example 2.

```
    Function CPUJob2                                      ▷ CPU Code
 1: CPUAllocate(C-CPU)                                    ▷ Allocate data on CPU
 2: CPUAllocate(D-CPU)
 3: GPUAllocate(C-GPU)                                    ▷ Allocate data on GPU
 4: GPUAllocate(D-GPU)
 5: CPU-GPUTransfer(C-CPU,C-GPU)                          ▷ Tranfer data from CPU to GPU
    (Parametres)
 6: CPU-GPUTransfer(D-CPU,D-GPU)
 7: KernelJob2(C-GPU,D-GPU)                               ▷ Launch kernel
 8: GPU-CPUTransfer(D-GPU,D-CPU)  ▷ Tranfer data from GPU to CPU (Results)
    Function kernelJob2(C,D)                              ▷ GPU Code
 9: i = index of thread
10: D[i] = C[i] × D[i]
```

Table 1. Characteristics of GPUs 9600_M GT and GTX 285.

Characteristic	9600_M GT	GTX 285
Number of multiprocessors	4	30
Number of cores	32	240
Core clock	625 Mhz	648 Mhz
Memory clock	800 Mhz	1242 Mhz
Memory capacity	512 MB	1 GB
Memory Bus Size	128 bits	512 bits
Memory Bandwidth	25.6 GB/s	159 GB/s
Gigaflops	120	1062.72

memory many times. This is not our case since, there aren't any data which are shared by more than one thread.

We have proposed a set of test scenarios to evaluate the performance of both platforms. Each scenario is formed of a set of jobs. In scenarios 1-4 the jobs are identical, as a consequence of being interested in evaluating the GPU performance with the same requirements. Nevertheless, in the last scenario (Scenario 5) the jobs are different in order to evaluate the GPU performance with different requirements. The number of threads of each job is 512, due to that this number is an upper limit of the number of threads per block in each platform.

In all the scenarios, we show one table with the execution time of the following two alternatives; the first alternative consists in executing the different jobs sequentially, while the second one consists in executing the different jobs at the same time. We define factor 1 for indicating the number of jobs related with the number of multiprocessors. For example, the 9600 GT has 4 multiprocessors, so a factor equal to 0.5 indicates 2 jobs.

$$Factor = \frac{\#jobs}{\#multiprocessors} \tag{1}$$

Algorithm 3 modified CUDA pseudocode example.

```
    Function CPUJobUnited                                        ▷ CPU Code
 1: CPUAllocate(A-CPU)                              ▷ Allocate data on CPU Job1
 2: CPUAllocate(B-CPU)
 3: CPUAllocate(C-CPU)                              ▷ Allocate data on CPU Job2
 4: CPUAllocate(D-CPU)
 5: GPUAllocate(A-GPU)                              ▷ Allocate data on GPU Job1
 6: GPUAllocate(B-GPU)
 7: GPUAllocate(C-GPU)                              ▷ Allocate data on GPU Job2
 8: GPUAllocate(D-GPU)
 9: CPU-GPUTransfer(A-CPU,A-GPU)                  ▷ Tranfer data from CPU to GPU
    (Parametres) Job1
10: CPU-GPUTransfer(C-CPU,C-GPU)                  ▷ Tranfer data from CPU to GPU
    (Parametres) Job2
11: CPU-GPUTransfer(D-CPU,D-GPU)
12: KernelUnited(A-GPU,B-GPUC-GPU,D-GPU)                         ▷ Launch kernel
13: GPU-CPUTransfer(B-GPU,B-CPU)  ▷ Tranfer data from GPU to CPU (Results)
    Job1
14: GPU-CPUTransfer(D-GPU,D-CPU)                 ▷ Tranfer data from GPU and CPU
    (Results) Job2
    Function kernelJobUnited(A,B,C,D)                            ▷ GPU Code
15: i = index of thread
16: j = index of block
17: if j = 0 then                                                ▷ kernelJob1
18:     B[i] = A[i] + 100
19: else if j = 1 then                                           ▷ kernelJob2
20:     D[i] = C[i] × D[i]
21: end if
```

In each scenario we show the pseudocode of the jobs, and one table which shows the execution time of the two alternatives, first alternative (F.A.) and second alternative (S.A.), the speed up (S.) and the total number of memory accesses (M.A.) according to the factor for both platforms. The speed up is the quotient between the time of execution of the first and second alternatives. Therefore, the speed up refers to how much the second alternative is faster than the first alternative. Thus, $S = 3$ means that time has been divided by 3.

$$S = \frac{first\ alternative\ time}{second\ alternative\ time} \tag{2}$$

Table 2 shows the execution time (ms) and number of memory accesses of one job for every scenario.

Table 2. Execution time of one job for each scenario.

Scenario	9600_M GT(ms)	GTX 285(ms)	M.A.
1	0.042	0.059	0
2	0.047	0.062	1,536
3	1.27	0.33	132,096
4	2.95	0.61	263,168

3.1 Scenario 1

The only proposed under this scenario is to evaluate the handling of threads blocks. For this reason, in this scenario there are not any memory accesses (Algorithm 4). As we can see in table 3, the speed up obtained is almost ideal (speed up equal to the number of jobs). Therefore, it is possible to execute a set of jobs at the same time in one GPU.

In order to evaluate memory management, different numbers of memory accesses are performed in the rest of scenarios.

Algorithm 4 Scenario1.

```
   Job Scenario1
1: int r1,r2,r3
2: r3 = r1 + r2
```

Table 3. Execution time for the scenario 1.

				9600_M GT				GTX 285
Factor	F.A.	S.A.	S.	M.A.	F.A.	S.A.	S.	M.A.
0.5	0.084	0.044	1.9	0	0.885	0.062	14.27	0
1	0.168	0.046	3.65	0	1.77	0.064	27.65	0
1.5	0.252	0.047	5.36	0	2.65	0.065	40.76	0
2	0.336	0.048	7	0	3.54	0.066	53.63	0

3.2 Scenario 2

In this scenario there are 3 memory accesses for each thread and therefore $3 \times 512 = 1,536$ memory accesses for each job. The speed up obtained (Table 4) is smaller than in scenario 1 due to the time required for memory accesses. As expected, the speed up decreases when the factor increases, this is because the number of memory accesses increases too.

Algorithm 5 Scenario2.

```
   Job Scenario2($A, B, C$)
   Inputs:
     $A, B$
   Output:
     $C$
1: i = index of thread
2: C[i] = A[i] + B[i]
```

Table 4. Execution time for the scenario 2.

				9600_M GT				GTX 285
Factor	F.A.	S.A.	S.	M.A.	F.A.	S.A.	S.	M.A.
0.5	0.094	0.051	1,84	3,072	0.93	0.067	13.88	23,040
1	0.188	0.054	3.48	6,144	1.86	0.070	26.57	46,080
1.5	0.282	0.055	5.12	9,216	2.79	0.076	36.71	69,120
2	0.376	0.056	6.71	12,288	3.72	0.079	47.08	92,160

3.3 Scenario 3

In this scenario, each thread has $256 + 2$ memory accesses thus each job has $258 \times 512 = 132,096$ memory accesses, which is $130,560$ memory accesses more than scenario 2 for each job. When memory accesses are substantially increased the speed up falls down even more than before.

Algorithm 6 Scenario3.

```
   Job Scenario3($A, B, C$)
   Inputs:
     $A, B$
   Output:
     $C$
1: i = index of thread
2: int r1
3: r1 = A[i]
4: for $j = 0$ to 255 do
5:     r1 += B[j]
6: end for
7: C[i] = r1
```

Table 5. Execution time for the scenario 3.

			9600$_M$ GT				GTX 285	
Factor	F.A.	S.A.	S.	M.A.	F.A.	S.A.	S.	M.A.
0.5	2.54	2.25	1.12	264,192	4.95	1.51	3.27	1,981,440
1	5.08	4.47	1.13	528,384	9.9	4.92	2.01	3,962,880
1.5	7.62	6.58	1.15	792,576	14.85	4.96	2.99	5,944,320
2	10.16	8.80	1.15	1,056,768	19.8	9.98	1.98	7,925,760

3.4 Scenario 4

There are 512+2 memory accesses for each thread in this scenario, and therefore each job has $512 \times 514 = 263,168$ memory accesses, which is $261,632$ memory accesses more than scenario 1 and $129,536$ more than scenario 2 for each job. Thus, the speed up of this scenario (Table 6) is the smallest due to the fact that the number of memory accesses is the greatest, even more, in some executions the time taken by both alternatives have been almost equal.

Algorithm 7 Scenario4.

```
   **Job** Scenario4($A, B, C$)
   **Inputs:**
     $A, B$
   **Output:**
     $C$
1: i = index of thread
2: int r1
3: r1 = A[i]
4: **for** $j = 0$ to 511 **do**
5:     r1 += B[j]
6: **end for**
7: C[i] = r1
```

Table 6. Execution time for the scenario 4.

			9600$_M$ GT				GTX 285	
Factor	F.A.	S.A.	S.	M.A.	F.A.	S.A.	S.	M.A.
0.5	5.9	5.42	1.08	526,336	9.15	3.15	2.90	3,947,520
1	11.8	11.20	1.05	1,052,672	18.3	10.97	1.66	7,895,040
1.5	17.7	16.60	1.06	1,579,008	27.45	11.53	2.38	11,842,560
2	23.6	21.90	1.07	2,105,344	36.6	34.93	1.04	15,790,080

3.5 Scenario 5

In this scenario the jobs executed are different for the sake of evaluating GPU performance with different requirements. Besides, there are 2 different test cases; in the first test case, we execute jobs of the 4 scenarios, for each scenario, we execute a different number of jobs for each platform (factor equals to 0.5), for the 9600_M GT, we executed 2 jobs for each scenario, on the other hand, for the GTX 285, we executed 15 jobs for each scenario. In the second test case, we execute jobs of scenarios 3 and 4, which are the more computationally expensive, because this is the reason of having 2 different test cases (one specially focused on heavy computations), for the 9600 $_M$ GT we execute 4 jobs for each scenario, and for the GTX 285 we execute 30 jobs for each scenario.

Algorithm 8 shows the structure of the kernel for both cases and platforms.

Algorithm 8 Scenario5.

```
    **Case1** Scenario5($Set\ of\ Parameters$)
 1: j = index of block
 2: if ((j < 2 ($9600_M$ GT)) OR (j < 15 (GTX 285))) then
 3:     kernelScenario1
 4: else if ((j < 4 ($9600_M$ GT)) OR (j < 30 (GTX 285))) then
 5:     kernelScenario2
 6: else if ((j < 6 ($9600_M$ GT)) OR (j < 45 (GTX 285))) then
 7:     kernelScenario3
 8: else
 9:     kernelScenario4
10: end if
    Case2 Scenario5($Set\ of\ Parameters$)
11: j = index of block
12: if ((j < 4 ($9600_M$ GT)) OR (j < 30 (GTX 285))) then
13:     kernelScenario3
14: else
15:     kernelScenario4
16: end if
```

Again, from table 7, the more memory accesses, the smaller speed up.

Table 7. Execution time for the scenario 5.

				9600_M GT				GTX 285
Case	F.A.	S.A.	S.	M.A.	F.A.	S.A.	S.	M.A.
1	8.61	2.85	3.02	793,600	15.915	5.95	2.67	5,952,000
2	16.88	5.85	2.88	1,581,056	28.2	21.47	1.31	11,857,920

4 Conclusions and future work

We have proven the capacity of executing more than one job at the same time on a GPU as the results obtained in this study shows.

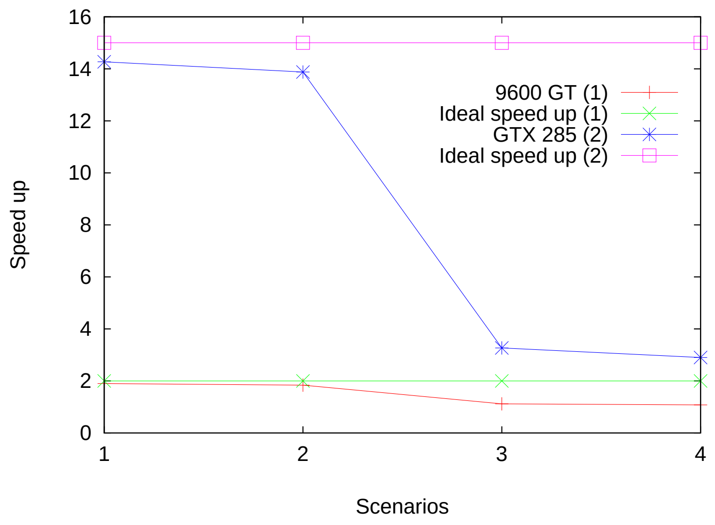

Fig. 3. Speed up obtained in all scenarios for a factor equal to 0.5.

When the number of memory accesses increases, the speed up falls down, this is mainly due to two reasons, firstly, because the memory accesses are slow; the memory conflicts are the second reason. One memory conflict occurs when two or more threads have to access the same memory bank at the very same time (simultaneously). In these cases the memory accesses are forced to be sequential. Thus, as we show in section 3, the more memory accesses, the more memory conflicts, so the speed up falls down, since, the global memory is shared by all multiprocessors, and therefore all threads of every multiprocessor could access to the same memory space. In the case of 9600 GT GPU, the speed up is smaller, due to the fact that this GPU has not only smaller memory capacity but also smaller bus size and smaller memory bandwidth than GTX 850 GPU. Figure 3 shows the tendency of speed up for all scenarios under a factor equal to 0.5 and the ideal speed up for both platforms. On other hand, figure 4 shows the number of memory accesses for all scenarios under the same factor. Both figures 3 and 4 show the behaviours of both platforms for each scenario.

It is obvious the increasing use of GPUs in a lot of HPC environments and in this line is focused our future research interest in this topic.

In our modest opinion these GPUs devices should have as many “private” memory spaces as number of multiprocessors, since the bottleneck of all these

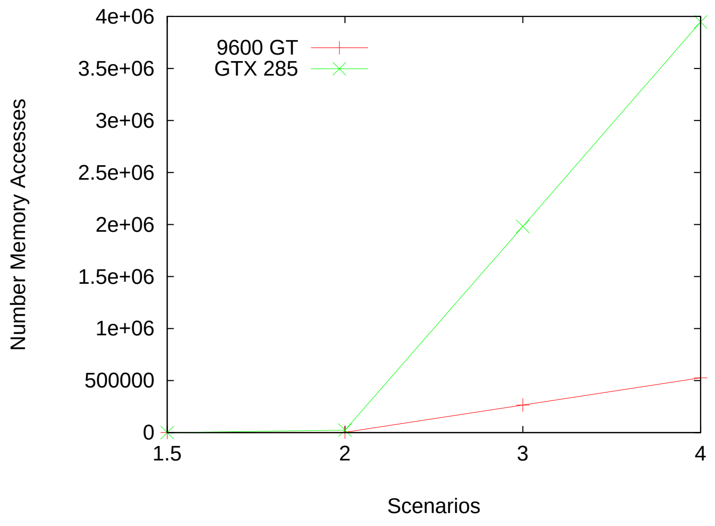

Fig. 4. Number of memory accesses of all scenarios for a factor equal to 0.5.

experiments seems to be there, for the sake of better being used under the requirements demanded by these HPC environments. Unfortunately, the future developments of GPU's are focused mainly on increasing the number of cores per multiprocessor and somehow, on improving the hierarchy of memory used. In order to improve these devices even more, we think that having an scheduling policy which allows to manage the executions of the different jobs (preventing to be forced to execute all jobs at the same time), therefore these jobs could be executed according their needs or priorities, perhaps defined by optimizing general/particular performances, could be very profitable for these devices. Figure 5, roughly captures these two proposals.

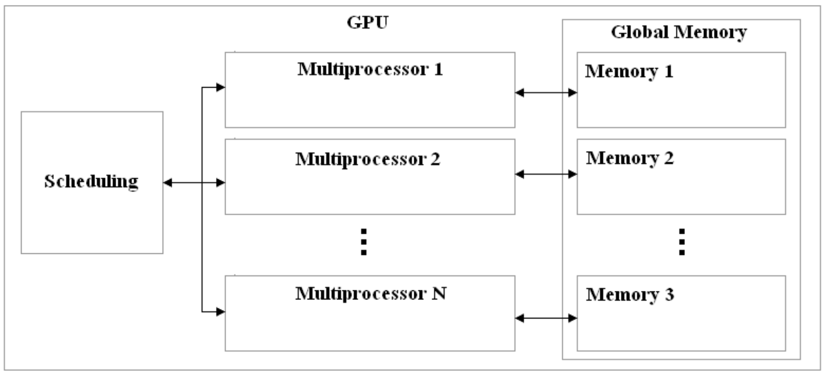

Fig. 5. Changes on GPU Architecture

References

1. GPGPU. *General-purpose computation using graphics hardware.* `http://www.gpgpu.org`.
2. W.-C. Feng, D. Manocha. High-performance computing using accelerators, Parallel Computing, Elsevier, 33 (2007), 645-647.
3. R.J. Rost. OpenGL Shading Language, Addison-Wesley, 2005.
4. W.R. Mark, S.R. Glanville, K. Akeley, M.J. Kilgard. Cg: a system for programming graphics hardware in a C-like language. SIGGRAPH'03: ACM SIGGRAPH 2003 Papers, pages 896-907, New York, NY, USA, 2003. ACM Press.
5. NVIDIA. *NVIDIA CUDA Compute Unified Device Architecture-Programming Guide, Version 2.3* 2009, `http://www.nvidia.com/object/cuda_home.html`.
6. Speeding up pricing complex instruments in the cloud with scifinance. SciComp Inc. `http://www.scicomp.com/`.
7. GPUGRID. `http://www.gpugrid.net`.
8. BOINC. `http://boinc.berkeley.edu/gpu.php`.
9. SETI. `http://setiathome.berkeley.edu/cuda.php`.
10. Milkyway. `http://milkyway.cs.rpi.edu/milkyway_gpu/`.
11. AQUA. `http://aqua.dwavesys.com/`.
12. The Lattice Project. `http://boinc.umiacs.umd.edu/`.
13. Einstein. `http://einstein.phys.uwm.edu/`.
14. Collatz. `http://boinc.thesonntags.com/collatz/`.
15. Primegrid. `http://www.primegrid.com/`.
16. DNETC. `http://dnetc.net/`.
17. TOP500. `http://www.top500.org/`.