Optimizing a GPU Algorithm Through Hardware Profiling Analysis

Fernando G. Tinetti
III-LIDI, Facultad De Informática
Comisión de Inv. Científicas Pcia. de Bs. As.
Universidad Nacional de La Plata, La Plata, Argentina
fernando@info.unlp.edu.ar

Sergio M. Martin
Dto. de Ingeniería e Investigaciones Tecnológicas
Universidad Nacional de La Matanza
Buenos Aires, Argentina
smartin@ing.unlam.edu.ar

***Abstract*— Usage of GPU-based architectures for scientific computing has been steadily increasing in the last years. This new paradigm for both programming and execution has been applied to solve several classic problems much faster than using the conventional multiprocessor and/or multicomputer approach. These architectures allow an increase in performance – compared to conventional CPU processors – for specific types of algorithms that are particularly suitable for its greater number of simpler cores which execute one single instruction at a time, each one for different sets of data. Since this is still a relative new technology, GPU device manufacturers as well as independent researchers have published several experiences (*success stories*), best practices, and optimization guides to aid developers for obtaining the maximum program performance. However, there is still little information about the possible optimizations that can only be harnessed by analyzing the specific device's hardware performance counters. In this paper, we discuss several optimizations based on hardware profiling and share our learned lessons about how such data can be used to optimize a scientific algorithm on a GPU using CUDA.**

Keywords—GPU Computing, CUDA, Hardware Counters, Profiling, Scientific Computing.

I. INTRODUCTION

Nowadays, many scientists that rely on computational power to solve or simulate scientific problems are using the recent development of many-core technologies, such as GPGPU (General Purpose Computing on Graphics Processing Units) [1] and Intel Xeon Phi programming [2]. These architectures differ from the conventional complex multi-core processors in the use of the available transistors. While conventional CPUs dedicate more transistors to allow complex instructions, branch prediction, and instruction-level parallelism for each core, many-core devices minimize each core's complexity. The result is that GPU cores are much smaller, and many more of them can be allocated. The limitation is that all of its cores must execute exactly the same instruction. While the majority of conventional programs could not function properly under this architecture, several scientific algorithms provide an ideal scenario to harness the potential of this new technology. Easily divisible problems that require the repeated execution of a relatively simple procedure upon a large amount of data are particularly convenient to execute on a GPU. Therefore, adapting a parallel algorithm to be run in a GPU device can lead to important performance gains [3].

Since programming algorithms on a GPU device is a relatively new discipline, there is still much to learn about how to better harness their potential. Many useful general-purpose guides have been available [4] [5] [6] so that the programmer can have a better idea of how to better tweak the algorithms for performance improvement. Furthermore, there are many publications about how to optimize specific algorithms, from linear algebra [7] to physics simulations [8]. Top-down approaches –analyzing algorithms to improve hardware's performance– are extremely helpful during the first optimization stages of a many-core algorithm. However, very few of the existing material take bottom-up approach– analyzing hardware counters to improve the algorithm–. Using hardware profiling [14] tools or information that is accessible from the built-in hardware counters that are provided by the device could also be a useful way to discover new optimizations on particular algorithms. A hint of how much potential this may hold for general-purpose algorithms is given in [9]. Based on this potential, we set our aim on the possibility of analyzing optimization using hardware performance counters.

In order to determine which optimization opportunities could be identified by analyzing hardware counters, we used an actual CUDA implementation of the N-body algorithm [10]. This algorithm is a CUDA adaptation of our original parallel version for multi-core (CPU) clusters [11]. After applying all possible optimizations described by classic guidelines and bibliography, we used a profiling tool to inspect a series of runs and try to obtain useful information about how are the GPU's resources were being (under)utilized. Then, based on profiling results, we summarize which changes improved the overall performance of the algorithm, and analyze its possible application on other cases/applications.

The rest of this article is organized as follows: Section 2 introduces the architecture, the profiling tools, and the algorithm used; in Section 3, we provide an overview of the hardware counters we used during the experiments; the original algorithm optimizations applied are explained and measured in Section 4. The optimizations based on hardware counter analysis explained in in Section 5. Results for the applied optimization are shown in Section 6. Finally, Section 7 includes conclusions and further work.

II. Architecture, Tools, and Algorithm

Since most of the paper is based on experimentation, we detail a) the specific architecture (from which performance event counters is recollected and analyzed), b) the profiling tool (providing access to the hardware event counters), and c) the specific CUDA algorithm taken as departure point.

A. GPU Architecture

Table I describes the GPU device used in the experiments:

TABLE I. NVIDIA GPU Hardware Description

GPU Device	GeForce GTX 550Ti
CUDA Cores	192
Capability	CUDA 2.1
DRAM	1 GB GDDR5
Max Active Blocks	8
Max Active Warps	48
Max Active Threads	1536
Max Treads/Block	1024
Max Warps/Block	32
Max Registers/Thread	63
Max Registers/Block	32768
Max SharedMemory/Block	49152 bytes

The data provided in Table I about the architecture and its limitations are of vital importance during profiling. Knowing the maximum capacity for each particular aspect of the execution of a CUDA kernel allows knowing how much resources are being (under)used by the algorithm. In the next section, we will analyze the relation between these maximum capacities and the actual measures for the base algorithm.

B. Profiling Tool

Profiling tools provide measurements of actual resource usage at runtime. NVIDIA provides a CUDA Profiler [12] that returns specific values taken directly from the hardware counters. However, we chose to use NVIDIA Nsight Visual Studio Edition [13] because it is integrated automatically with the development environment and elaborates higher level information based on the hardware counters.

C. CUDA Algorithm

For our tests, we used a version of an N-body algorithm for CUDA [10]. This algorithm defines a set of bodies where each one of them is processed according to its relationship with all of the other bodies, therefore having $O(n^2)$ complexity. For its CUDA kernel execution we create as much threads as bodies in the simulation, and let each thread do the calculations for a single body. Then, several threads are grouped into blocks so that they are able use the available per-block shared memory as an explicit cache. This execution scheme is shown in Fig. 1:

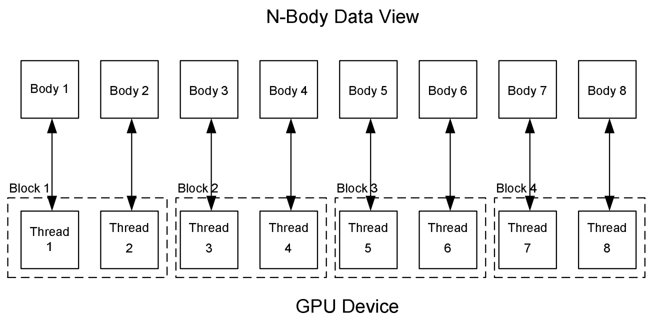


Fig. 1. Distribution of bodies, threads, and blocks in our N-body CUDA kernel, as an example for N=8

After the kernel is launched for execution, every thread will execute the following steps for its corresponding body:

- Load a single body's initial values from the device global memory. Each thread will load a different body based on its thread ID.
- For each other block of bodies in the simulation:
  - Load the bodies block values from the device global to shared memory.
  - Calculate the force that all other bodies (except itself) loaded into shared memory impose to the loaded body.
- Save the new values for acceleration in the body data back into the device global memory.

The need of excluding the calculation of a body with itself demands the inclusion of an *if*-clause that compares both source and destination bodies index. If both indexes are identical, then the calculation is omitted.

III. GPU Hardware Counters Overview

A. Achieved Occupancy

A set of counters is used by Nsight in order to measure how much the available processing capacity of the device is being used by a CUDA kernel. The result of the performance analysis is called *achieved occupancy*. A high (close to 100%) achieved occupancy means that all GPU cores were assigned a thread during the execution of the kernel. Several factors could lead to lower achieved occupancies:

- If the algorithm defines threads for the execution of its kernel with a count smaller than the maximum active threads (1536), then the occupancy will be limited to that count.
- If a small count (e.g. 32) of threads per block is defined, then the maximum active blocks (8) may sum fewer threads than the maximum (1536).
- Even if the maximum (1024) threads per block is used, the amount of maximum active blocks may be limited to the actual per-block memory usage limitations. This will be explained in the next item.

It is important to note that a full occupancy does not imply the best performance. If, for example, it is necessary to reduce the amount of registers per thread to increase the thread per block count, a much more number of accesses to the device (slow) RAM will be needed. This means that much more time will be lost on memory latency, thus drastically slowing down the algorithm even if the occupancy reaches 100%. This is, in fact, the example that we will examine through the execution of our N-Body algorithm.

B. *Memory Usage*

Each CUDA device specifies the amount of intra-processor fast memory for thread-private registers. Each register is an instance of a variable defined within a CUDA kernel, for each thread. For example, if the kernel uses four private variables, and 1024 threads are launched simultaneously, then 4096 registers will be used. All registers are considered as chunks of 4-bytes memory. If the amount of total registers per thread exceeds the maximum, the extra registers are stored in the device RAM memory, which is much slower.

Shared memory works as an L2 cache that is shared among all the threads within a block. Since this memory is also placed inside the GPU processor, it has a limited per-block capacity. Exceeding this capacity will result in an execution error.

In practice, the only limitation with the memory usage to the achieved occupancy is the maximum amount of registers per block. This maximum defines how many registers can persist for all blocks. Therefore, if the amount of threads per block multiplied by the amount of registers per thread exceeds this number, the maximum active blocks will be reduced, therefore limiting occupancy.

C. *Single FLOP Count*

Nsight also provides the single FLOP count, which determines exactly how many single-precision floating-point operations are executed. This is very useful to determine the actual calculation demands of the algorithm, which is not necessarily the same as that provided by the algorithm analysis. Single FLOP count includes basic operations (addition, subtraction, multiplication, division), as well as complex operations (in our example, square roots).

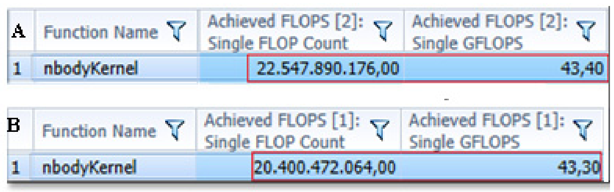
D. *Branches and Thread Divergence*

One of the measures that can be taken directly from the device's hardware counters is the amount of branches (conditional jumps) that each warp – the minimal group of threads executing the same instruction – has taken. It is important to reduce the amount of branches of a kernel executing in GPU architectures as they force all threads to execute as taking both paths, even if only one thread required a different one. Such phenomena can lead to much longer execution times. Therefore, it is important to measure not only the amount of branches reached, but also the relation of taken/not taken branches.

IV. Initial Optimizations and Measures

A. *Fast Math Instructions*

The CUDA compiler allows defining optimization flags that reduce the complexity of their arithmetic operations. In particular, our algorithm uses the *sqrt* (square root) function to calculate the overall distance between to objects. This function is, by default, defined for double precision. Since we used single precision floating-point variables for each body's position, velocity, and mass, such precision is not necessary. Therefore, we changed this instruction to the faster *fsqrtf* function. Besides operating in single-precision, this function also is an implementation of the CUDA fast-math library. This means that a little precision is sacrificed to allow for fewer operations. It can be seen in Fig. 2 that more than two single GFLOP were saved from using *fsqrtf*.


A

	Function Name	Achieved FLOPS [2]: Single FLOP Count	Achieved FLOPS [2]: Single GFLOPS
1	nbodyKernel	22.547.890.176,00	43,40

B

	Function Name	Achieved FLOPS [1]: Single FLOP Count	Achieved FLOPS [1]: Single GFLOPS
1	nbodyKernel	20.400.472.064,00	43,30

Fig. 2. A. Single FLOP Count for *sqrt*(); B. Single FLOP Count for *fsqrtf*()

B. *If-clause Within Innermost Loops*

The N-Body algorithm requires the programmer to avoid the calculation of forces between a body with itself. This evaluation requires the existence of an if-clause to be executed on a quadratic per-body basis. To accomplish this, our original code implementation placed such clause within the innermost for-loop. The problem with this approach is that, since GPU devices are not designed to optimize divergent branch execution, this would mean that, for each loop execution, one thread per block would diverge, duplicating the total required time. To solve this problem, we have defined a per-block if clause that will diverge only if the block index corresponds to the current one. Therefore, all threads within a block can always execute non-divergently. Fig. 3 shows the difference in branches executed and taken between both versions, showing a reduction of 25% in branches executed/taken for the per-block if compared to a per-thread basis.

C. *Loop Unrolling*

Another way of reducing the amount of branches per kernel execution is to perform compiler-assisted loop unrolling. Duplicating the contents of a loop before considering a conditional jump, halves the amount of branches executed. In our example, since we are using a large amount of bodies for our N-Body simulations, both loops (internal and external) were unrolled as shown in Fig. 4.

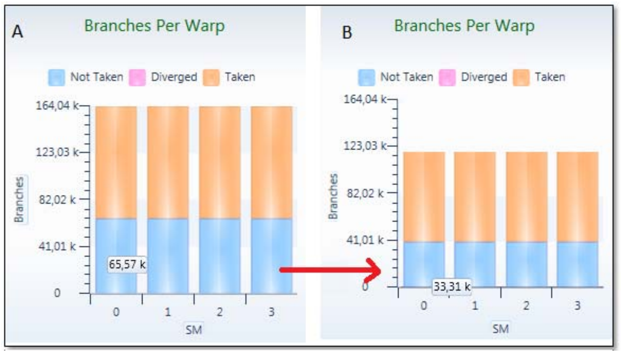


Fig. 3. A. Branches executed and taken for the per-thread *if* version B. Branches executed and taken for the per-block *if* version

```
#pragma unroll 2 // Assumes an even amount of blocks
for (i = 0; i < block_count; i++)
{
  ...
      #pragma unroll 2
       for (j = 0; j < thread_count; j++)
            {
              ...
```

Fig. 4. Unrolled *for* clauses for both internal and external loops

As a result, the total amount of branches executed was once again reduced, this time by a 33% as in Fig. 5.

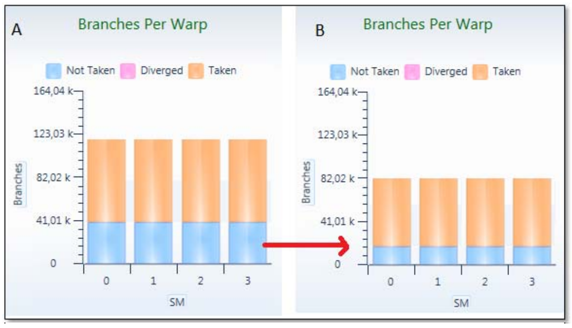


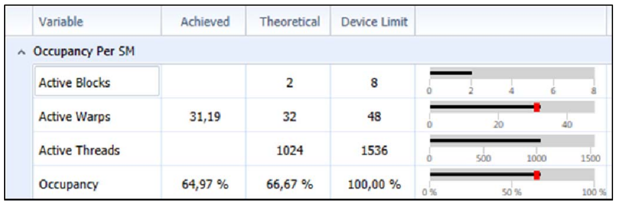
Fig. 5. A. Original branch executions B. Branch executions using loop unroll

V. Hardware Counter-Based Optimizations

After applying all the possible *classical* optimizations, we looked for indications of additional optimization opportunities that could be discovered by analyzing the hardware counters. The actual execution of the CUDA kernel has been used in order to get the corresponding hardware performance counters. Following is a list of our most relevant findings.

A. *Occupancy vs. Memory Efficiency*

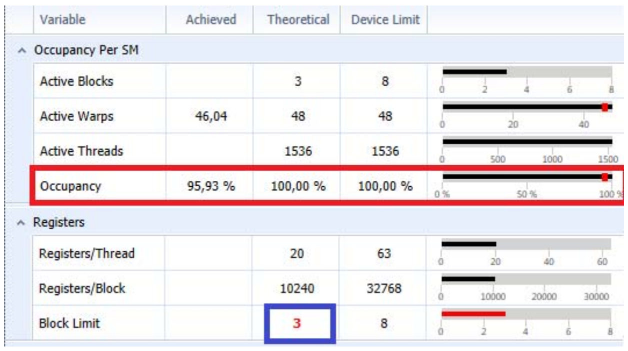
As Fig. 6 shows, the occupancy counters of our optimized algorithm indicate that not all the device's parallel capability is being used.



Variable	Achieved	Theoretical	Device Limit
Occupancy Per SM			
Active Blocks		2	8
Active Warps	31,19	32	48
Active Threads		1024	1536
Occupancy	64,97 %	66,67 %	100,00 %

Fig. 6. Only 2 of possible 3 blocks of 512 threads were executed.

The underlying reason for such underuse of the possible 1536 active threads that the architecture provided was the Max Registers/Block limit defined by the architecture. If the actual value approaches this maximum, only one block can run at a time. Trying to allow for 3 blocks of 512 threads each, we had to reduce the amount of registers per thread used. Since the algorithm could not function with less variable declarations, we had to use the *maxrregcount=20* (80 bytes per thread on registers memory) compiler flag to simulate register memory using private DRAM memory. Fig. 7 shows that, by reducing the amount of registers per thread, a bigger occupancy was reached:



Variable	Achieved	Theoretical	Device Limit
Occupancy Per SM			
Active Blocks		3	8
Active Warps	46,04	48	48
Active Threads		1536	1536
Occupancy	95,93 %	100,00 %	100,00 %
Registers			
Registers/Thread		20	63
Registers/Block		10240	32768
Block Limit		3	8

Fig. 7. A 95% occupancy is reached by limiting the use of registers.

Although the use of computational resources of the device was increased, this optimization yielded little gains in processing time. This is because the impact that using private DRAM instead of local registers memory has on the memory access latency. In fact, for some cases of N bodies (number of bodies), performance dropped due to excessive memory stalling. Therefore, we discarded this as an effective optimization for this algorithm.

B. *Maximizing Thread Use of Register Memory*

Another approach to optimize the use of the device resources is to take advantage of register memory. Since we discarded the maximum occupancy optimization method described above, there is more register memory available that we are not using for the only two 512-thread blocks in execution. As from Fig. 7, we have a 32768 registers limit; therefore, we could use 16384 registers per block (32 per thread). This extra register space allowed us to process two local bodies per thread, instead of just one, as shown in Fig. 8.

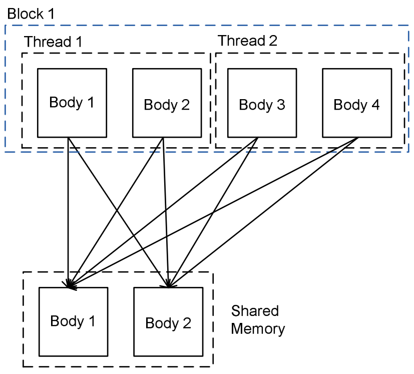
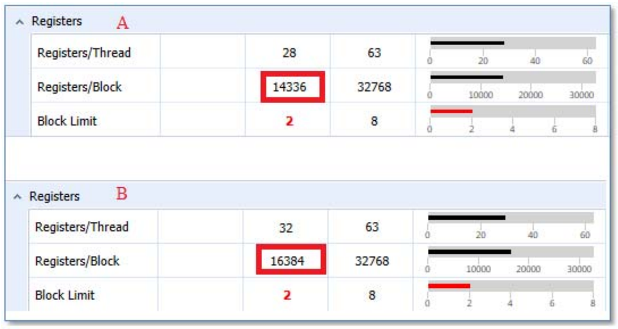


Fig. 8. Calculating two bodies per thread to maximize register memory usage

The result of this approach, compared to the original optimized algorithm, for the use of register memory can be seen in Fig. 9.



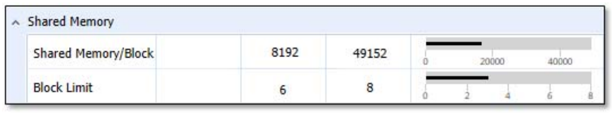
Registers A		
Registers/Thread	28	63
Registers/Block	14336	32768
Block Limit	2	8

Registers B		
Registers/Thread	32	63
Registers/Block	16384	32768
Block Limit	2	8

Fig. 9. A. Underuse of registers memory with single body per thread. B. Full use of registers memory by using two bodies per thread instead.

C. *Maximizing Thread Use of Shared Memory*

Since it was possible to maximize the use of registers memory, we sought to analyze the possibility of taking the same approach for shared memory. After analyzing the shared memory usage from the hardware counters, we obtained the values shown in Fig. 10. It can be seen that we are only using 16384 bytes (8192 bytes per block, using 2 blocks) of shared memory from the 49152 available.



Shared Memory		
Shared Memory/Block	8192	49152
Block Limit	6	8

Fig. 10. A full use of registers memory can be achieved by calculating two bodies per thread.

The shared memory underuse is also made evident by the profiler as it also shows that the maximum active blocks could be up to six, as for the shared memory limitations. Therefore, we tried to find a way to increment the usage of shared memory to reduce accesses to DRAM memory. To achieve this, we duplicated the amount of bodies to be loaded into shared memory as Fig. 11 shows.

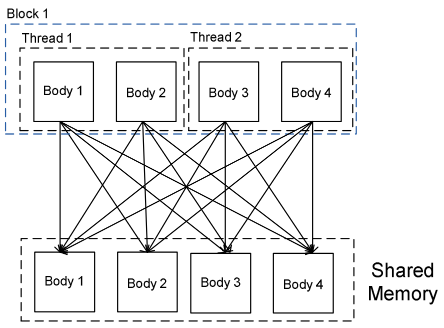
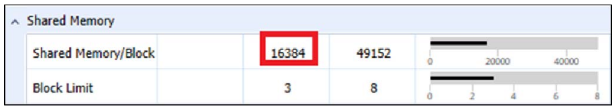


Fig. 11. An increase in shared memory use can be achieved by loading four bodies per thread.

Fig. 12 shows the result of this approach, doubling the shared memory usage (compared to that shown in Fig. 10).



Shared Memory			
Shared Memory/Block	16384	49152	
Block Limit	3	8	

Fig. 12. The use of shared memory doubled by loading additional bodies

VI. TESTS & RESULTS

In order to determine how much performance gains provide these optimizations, we ran three different versions of our N-Body algorithm:

Version1 - N-Body without optimizations

Version2 - N-Body with the *classical* optimizations

Version3 - N-Body with all the described optimizations (including those identified by the hardware performance counters)

For each version, we ran tests with different values of N = {131072, 262144, 524288, 1048576, 2097152}. In addition, for each version/N combination, we ran a set of tests, taking the average performance value in order to rule out possible dispersion due to unexpected interference. The results obtained are shown in Fig. 13.

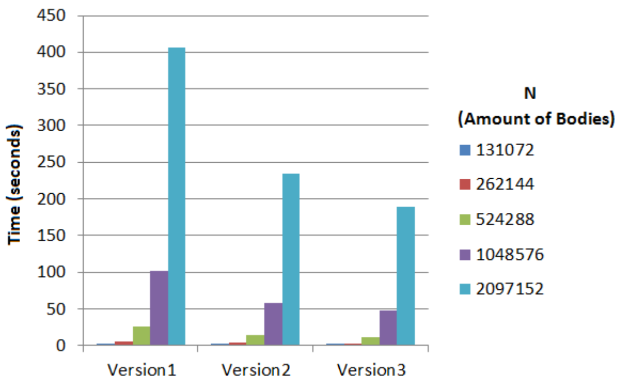


Fig. 13. Time taken for each version for different values of N

It can be seen from comparing the original algorithm (Version1) to the fully optimized algorithm (Version3) that the average time taken to complete a single step of simulation was reduced about 53%. However, this comparison does not provide any information as to how much the applied hardware counter-based optimizations helped to increase performance. Therefore, it is necessary to determine the ratio between the initial classical optimizations and the hardware counter analysis-based optimizations within the achieved 53% increase in performance. To obtain this relation, we compared how time decreases from Version1 to Version2 and from Version2 to Version3:

- Version1 to Version2: 171 seconds decrease (42%)
- Version2 to Version3: 45 seconds decrease (11%)

The results show that a potential 11% average performance gain could have been missing for our algorithm in case that we did not analyze the hardware counters in search of possible optimizations. Although this gain is almost four times smaller than the one obtained from best practices optimizations, it can be significant enough for large experiments, including larger values of N.

VII. CONCLUSIONS

For very large simulations and experiments for physics or engineering scenarios -where the execution time and numerical precision are generally important factors–, every possible optimization, although minimal, can be significant. For scientists developing algorithms for such experiments to be run in GPU processors, every possible optimization matters.

The common and most effective practice towards optimizing GPU algorithms will always be analyzing the code taking into account the best-practices classical optimization guidelines. As verified in Section IV, those guidelines usually contain proven, thoroughly examined, effective techniques to improve the execution of the algorithm's kernel in the device. However, we have also seen in Section V, that even more optimizations can be gathered from analyzing the device's hardware counters from the algorithm's execution. Analyzing those counters can provide an insight as to how a specific algorithm can be further optimized.

In our case, the optimizations based on hardware counters allowed an additional time reduction of 11%. This reduction in time taken is small compared to the much more effective best-practices guided optimizations (42%). Moreover, hardware counter-based optimizations took much more analysis effort than the first ones.

Hardware counter-based optimizations could be, nonetheless, worth the effort. For large experiments and/or simulations that could take from hours to days, and make use of large hardware requirements, every optimization that could be done on the algorithm is bound to save hours and resources for the scientists running it. Therefore, in those cases, a thorough analysis of the algorithm execution looking for possible optimizations based on hardware counters can provide an additional value that overweighs the analysis' effort.

REFERENCES

[1] D. Luebke, M. Harris, N. Govindaraju, A. Lefohn, M. Houston, J. Owens, M. Segal, M. Papakipos, I. Buck, "GPGPU: general-purpose computation on graphics hardware". ACM/IEEE Supercomputing 2006. ACM, New York, USA, Art. 208.

[2] C. George, "Intel® Xeon Phi™ coprocessor (codename Knights Corner)." Proceedings of the 24th Hot Chips Symposium, HC. 2012.

[3] Z. Fan, F. Qiu, A. Kaufman, S. Yoakum-Stover, "GPU Cluster for High Performance Computing". ACM/IEEE Supercomputing 2004. ACM, Washington DC, USA, Art. 47.

[4] S. Ryoo, C. I. Rodrigues, S. S. Baghsorkhi, S. S. Stone, D. B. Kirk, W. W. Hwu, "Optimization principles and application performance evaluation of a multithreaded GPU using CUDA". ACM SIGPLAN Symposium on Principles and practice of parallel programming 2008. ACM, New York, USA, pp. 73-82.

[5] NVIDIA CUDA™ Programming Guide Version 5.5, NVIDIA Corporation, 2013.

[6] NVIDIA CUDA™ Best Practices Guide Version 5.5, NVIDIA Corporation. 2013.

[7] Ruetsch, Greg, and Paulius Micikevicius. "Optimizing matrix transpose in CUDA." Nvidia CUDA SDK Application.

[8] F. Molnár Jr., T. Szakály, R. Mészáros, I. Lagzi, “Air pollution modelling using a Graphics Processing Unit with CUDA”, Computer Physics Communications, vol. 181 (1), 2010, pp. 105-112.

[9] G. Teodoro, R. Oliveira, D. Neto, R.. Ferreira, “Profiling General Purpose GPU Applications”. Computer Architecture and High Performance Computing. October, 2009. Sao Paulo, Brazil.

[10] F. G. Tinetti, S. Martin, “Sequential optimization and shared and distributed memory parallelization in clusters: N-Body/Particle Simulation.” Parallel and Distributed Computing and Systems 2012. Las Vegas, USA.

[11] S. Martin, F. G. Tinetti, N. Casas, G. De Luca, D. Giulianelli, “N-Body Simulation Using GP-GPU: Evaluating Host/Device Memory Transference Overhead” XIX Congreso Argentino de Ciencia de la Computación 2013. Buenos Aires, Argentina.

[12] NVIDIA Profiler User’s Guide Version 5.5, NVIDIA Corporation, 2013.

[13] NVIDIA Nsight™ Visual Studio Edition 3.0 User Guide. NVIDIA Corporation. 2013.

[14] F. G. Tinetti, S. Martin, F. Frati, M. Méndez, “Optimization and parallelization experiences using hardware performance counters”. International Supercomputing Conference 2013. Colima, Mexico.