International Conference on Mathematics, Computational Methods & Reactor Physics (M&C 2009)
Saratoga Springs, New York, May 3-7, 2009, on CD-ROM, American Nuclear Society, LaGrange Park, IL (2009)

Massively Parallel Solving of 3D Simplified P_N Equations on Graphics Processing Units

W. Kirschenmann, L. Plagne, S. Ploix, A. Ponçot
EDF R&D
1 av. du Gal de Gaulle F-92141 BP 408 Clamart, France
{wilfried.kirschenmann,laurent.plagne,stephane.ploix,angelique.poncot}@edf.fr

S. Vialle
SUPELEC - IMS group & AlGorille INRIA project team
2, rue Edouard Belin F-57070 Metz Cedex, France
stephane.vialle@supelec.fr

ABSTRACT

This paper presents an efficient parallel implementation on Graphics Processing Units (GPUs) for the Simplified P_N (SP_N) calculations in the 3D case. For a nuclear operator such as EDF, the time required to compute nuclear reactor core simulations is rather critical. The SP_N method provides a convenient trade-off between accuracy and numerical complexity and is used in several industrial simulations. The parallelization of the algorithm should allow to reduce the computation time required to solve the eigenvalue problem. To solve the problem on distributed memory machines such as PC clusters, Domain Decomposition Methods have been investigated. Complementary to this approach, this work aims at using emerging massively parallel processors such as the GPUs. Based on a fine grained parallelism, this solution offers the opportunity to achieve good performances at very low cost. Indeed, GPUs provide a large computational power and some specific optimizations allow to near the hardware limits. Our GPU implementation solves 3D SP_N problems 30 times faster than its sequential CPU counterpart.

Key Words: parallel SP_N, Graphics Processing Units (GPUs), CUDA, neutron transport

1. Introduction

As operator of nuclear power plants, EDF needs many nuclear reactor core simulations in the processes of either production or conception. Currently under development at EDF R&D, the COCAGNE nuclear reactor core simulation system aims to meet this need. As a convenient trade-off between accuracy and numerical complexity, Simplified P_N (SP_N) approximation is frequently used in neutron transport simulations [1]. The SP_N approximation has been implemented in several flavors by various authors [2, 3] and this paper focuses on the algorithm derived in [2] that solves the *mixed dual* formulation of the SP_N. A generic C++ implementation of this algorithm has been developed for the COCAGNE system [4]. This sequential SP_N code has been used as a reference starting point for the present work.

The time required to solve the SP_N equations is rather critical in several industrial simulations and its reduction by means of parallelization on distributed memory targets has been studied by several authors (e.g. [5, 6, 7]). It is an ongoing research topic in our laboratory where Domain Decomposition Methods (DDM) are investigated [8]. The DDM approach corresponds to a rather coarse

Table I. Processors Comparison

Processor	Type	Date of Release	Maximum Active Threads [*]	RAM Bandwidth (GB/s) [†]	Computational Power (GFlops) [†]
Intel Core i7	CPU	11/2008	8 (32 with SSE)	12.8	51.2
AMD 8386	CPU	11/2008	4 (16 with SSE)	12.8	55.0
Intel Xeon 5410	CPU	11/2007	4 (16 with SSE)	7.7	51.2
NVIDIA FX5800	GPU	11/2008	30 720	97.3	933.0
ATI FireGL V8650	GPU	01/2008	320	119.0	1 120.0
NVIDIA C870	GPU	06/2007	12 288	71.5	480.0

grained parallelism that typically runs a few hundreds of concurrent tasks on distributed memory target machines such as PC clusters. Complementary to the DDM approach, the present work focuses on a fine grained parallel implementation of the SP_N code.

Driven by the mass-market of video-games, the Graphics Processing Units (GPUs) have evolved over the recent years to become programmable massively parallel processors. They are now powerful coprocessors (see Table I), and their high efficiency makes them useful for a variety of applications. Compared to a PC cluster, a GPU can provide a very low cost computational power with respect to the material procurement, installation and energy consumption.

Very recently and independently, Kodama et al. have reported a successful GPU parallel implementation for SP_N calculations in the 2D case based on a response matrix formulation [3]. Compared to this anterior work, we applied the GPU parallelization for solving the *mixed dual* formulation of the SP_N equations in the 3D case. To our knowledge, this parallelization is the first successful implementation for this specific application. As a result, a substantial speed-up (up to $\times 30$) compared to a sequential implementation running on a CPU of the same year (2007) was achieved.

The rest of the paper is structured as follows. In Section 2 and 3, we introduce respectively the SP_N method and the programming of GPUs with some specific optimization rules. Section 4 shows how the code has been modified, adapted and optimized for the GPU to achieve the performances discussed in Section 5. Finally, the conclusion follows in Section 6.

2. The SP_N Method

The P_N method represents one way to solve the Boltzmann equation [9]. It is based on the representation of the angular dependence of the angular flux by a truncated spherical harmonic series. Contrary to discrete ordinates method, the P_N method does not exhibit ray effect. But in multi-dimensional problems, P_N equations are quite complex making this approach very time-consuming. To overcome this difficulty, Gelbard proposed a simplification of the P_N equations

[*] A definition of *active thread* is given in subsection 3.1. Using this definition, SSE units have a MNAT equal to 4.
[†] Given bandwidths and computational power obtained from constructors datasheets.

called Simplified P$_N$ equations [1]. SP$_N$ equations are derived from mono-dimensional P$_N$ equations and extended to multi-dimensional problems by replacing odd moments of the angular flux with vector functions and even moments with scalar functions. Furthermore, the first order derivative is replaced either by a divergence operator for the vector functions or by a gradient operator for the scalar functions. This leads to a system of $N+1$ equations which can be seen as $(N+1)/2$ coupled diffusion equations where the vector functions $\vec{\psi}_i$ represent the current unknowns and the scalar functions ϕ_i represent the flux unknowns:

$$\left\{\begin{array}{l}\dfrac{2i+1}{4i+1}\,\mathrm{div}\;\vec{\psi}_i(\vec{r})+\Sigma_{a,2i}\phi_i(\vec{r})=S_i^{\phi}(\vec{r})-\dfrac{2i}{4i+1}\,\mathrm{div}\;\vec{\psi}_{i-1}(\vec{r}),\\ \dfrac{2i+1}{4i+3}\,\overrightarrow{\mathrm{grad}}\,\phi_i(\vec{r})+\Sigma_{a,2i+1}(\vec{r})\vec{\psi}_i(\vec{r})=\vec{S}_i(\vec{r})-\dfrac{2i+2}{4i+3}\,\overrightarrow{\mathrm{grad}}\,\phi_{i+1}(\vec{r}).\end{array}\right. \tag{1}$$

In our implementation, each diffusion system is solved spatially by using the *mixed dual* finite element RTN described in [2]. With this element, we get a continuous current approximation of order n and a discontinuous flux approximation, of order $n-1$, which is well suited to describe strong flux variations at interfaces. In this paper, computations are made with the RT0 element in a Cartesian mesh. This element has 7 degrees of freedom (DoFs) per cell: 1 DoF for the scalar unknown and 6 DoFs for the vector unknown as shown in Fig. 1.

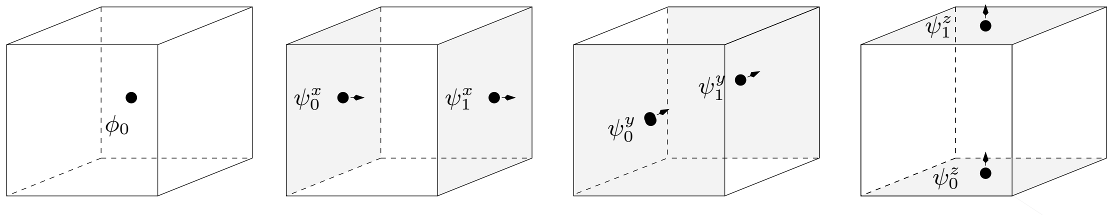


Figure 1. Degrees of freedom for the RT0 element.

Flux and current basis are chosen in order to obtain a very sparse matrix system without direction coupling term (Eq. 2).

$$\begin{pmatrix} A_x & 0 & 0 & -B_x \\ 0 & A_y & 0 & -B_y \\ 0 & 0 & A_z & -B_z \\ B_x^T & B_y^T & B_z^T & T \end{pmatrix}\begin{pmatrix}\psi_x\\ \psi_y\\ \psi_z\\ \phi\end{pmatrix}=\begin{pmatrix}S_x\\ S_y\\ S_z\\ S_\phi\end{pmatrix}. \tag{2}$$

With an appropriate DoFs numbering depending on the direction $d \in \{x,y,z\}$, the A_d matrices exhibit a block diagonal structure. Each block corresponds to a line of current DoFs and has a band structure (the matrix bandwidth is 3 for the RT0 element). To obtain a positive definite system, flux unknowns are eliminated. The resulting current system is solved using an alternating direction method. For each direction d, one has to solve a linear system of the form:

$$W_d\psi_d=\tilde{S}_d, \tag{3}$$

which is achieved with a Cholesky decomposition of each block of the matrix $W_d = A_d + B_d T^{-1} B_d^T$.

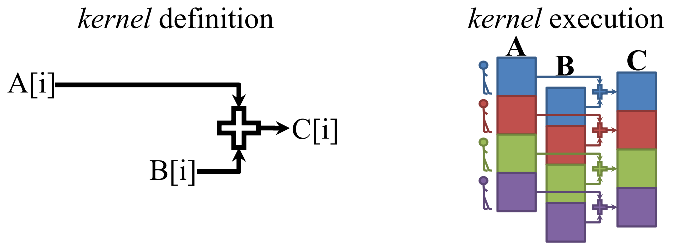


Figure 2. Addition of Two Vectors. The left scheme represents a *kernel* used to add two vectors. When the *kernel* is executed (right scheme), each *thread* (colored workers) apply the same *kernel* but on different data (each worker treats data with the same color).

To sum up, SPN equations discretization leads to solve one spatial subsystem per independent line of current DoFs. As discussed in Section 3, this massive degree of parallelism is well suited for GPU computing.

3. Introduction to the Principles of GPU Programming with CUDA

In this section, a first part will introduce some basic vocabulary and concepts relative to GPU programming. A second part will introduce three general optimization rules that have been applied to our code. The interested reader may refer to [10] for further details about GPU programming.

3.1. Definitions

Let us define a generic *thread* as an execution stream. It consists of a set of data and a stream of instructions that are to be applied to this set. When a program can be split into several subprograms that can be executed concurrently, each of these subprograms corresponds to the stream of instructions of one or several *threads*. Thus, a *thread* runs a subprogram and possibly cooperates with other concurrent *threads* in order to complete the program functionalities. When all threads share the same subprogram, the latter is defined as a computational *kernel*. Fig. 2 represents such a *kernel* used to add two vectors (left) and its execution by four *threads* (four colored workers on the right). In this example, each colored worker applies the *kernel* to different data sets corresponding to vector indexes (with the same color). On *Single Instruction Multiple Data* (SIMD) processors [11], several threads execute synchronously the same instruction and thereby the same kernel. Usually *kernels* designed for SIMD processors are simpler than those designed for other type of processors. The *kernel* of Fig. 2 is typically designed for a SIMD processor.

Most modern processors are built upon several smaller processors called cores on which different programs can be executed concurrently. Depending on the number of cores they contain, these processors are said to be either multicore (from two to some tens of cores) or manycore (tens or hundreds of cores). Fig. 3 shows a CPU with two cores and a GPU with 16 SIMD cores. Some types of processors expose more cores than they contain. Such processors are said to be simultaneously multithreaded and embed a hardware *thread* scheduler (e.g. Intel's hyperthreading [12]). This means that the operating system schedules the programs and their threads as if there were more cores available. For instance, when running on an Intel Core i7 quad-core processor, the

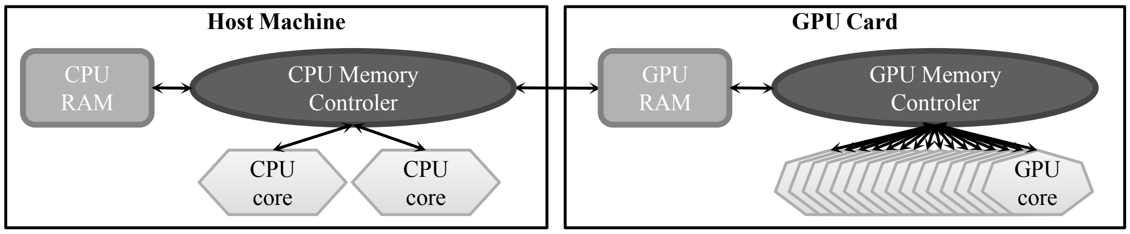


Figure 3. Simplified Architecture of a Compute Node with a GPU. While current CPUs have two or four cores and have a MNAT up to eight (32 when considering SSE units), current GPUs have up to 30 SIMD cores for a MNAT up to 30 720.

operating system sees eight cores although these processors only have four. On each core, a hardware *Thread* scheduler runs the two *Threads* to maximize the utilization of the different compute units. Let us define the Maximum Number of Active *Threads* (MNAT) as the maximum number of *threads* that can be scheduled simultaneously by a processor. Table I gives the MNAT for different processors. On GPU, the MNAT is higher since it is easier to combine many basic (comparatively simple) SIMD units on one chip than more complex processors.

In the rest of this paper, GPUs will be considered as SIMD processors without further considerations for the architecture details. As shown in Fig. 3, GPUs are usually embedded on graphics cards with very fast dedicated memory: on our test machine (presented in subsection 3.2), the *best throughput measured* is $62\,$GB/s on the GPU which is more than 15 times the $4\,$GB/s measured on our CPU. Actually, the performance of a code on a given hardware does not depend only on the available computational power: a processor can only compute with data in register. Hence, the performance of an algorithm may be limited either by the available computational power (*computation bound* algorithms) or the memory access speed (*memory bound* algorithms). For instance, a large dense matrix product requires far more operations than memory accesses. Hence, an optimized implementation is *computation bound*. On the contrary, a *saxpy* function ($Y = \alpha X + Y$ where Y and X are vectors of single precision floating numbers and α is a coefficient) contains $2N$ operations and $3N$ memory accesses and is usually *memory bound* on current processors. Since 1995, more and more algorithms have become *memory bound* since computational power increases faster than memory bandwidth [13, 14].

NVIDIA is a GPU constructor who provides CUDA, an extension to C/C++, to program its GPUs. NVIDIA considers GPUs as coprocessors and CUDA allows to write GPU *kernels* that will be called from a C/C++ program running on CPU. Hence, the adaptation of an algorithm for the GPU consists in developping a set of GPU *kernels* corresponding to this algorithm. This requires the algorithm to be written with loops having identical and independent iterations (regular loops). The constraints relative to memory accesses on GPU are similar to those that existed on vector machines (see subsection 3.2.1) and performance optimizations on GPU may require changes in the data structure. Our sequential $\mathrm{SP_N}$ implementation [4] is relatively easy to adapt for the GPU since it relies on extremely regular loops that iterate on independent lines of coupled degrees of freedom (DoFs) as explained in Section 4.

With the concepts of *thread* and *kernel* now introduced, we will present some rules that enable an

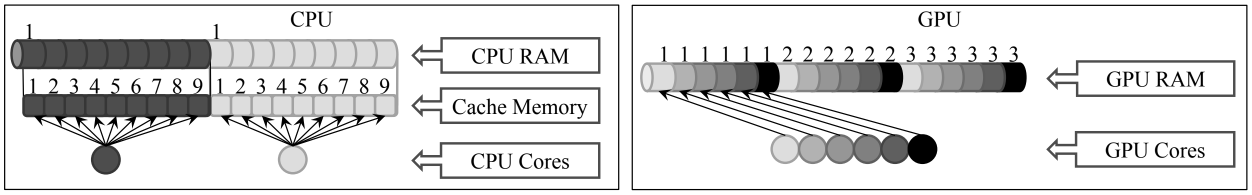


Figure 4. Differences Between Memory Access Patterns on CPU and on GPU. This example is based on a *kernel* in which a loop iterates over several elements of an array. On CPU (left), a contiguous range of data must be accessed by a *thread* in order to fit in the cache memory. In this example, all dark grey data are loaded in the dark grey cache memory in the first loop iteration. In other iterations, data are already in the cache memory and accesses are very fast. On GPU (right), two contiguous *threads* must access two contiguous data to minimize the total number of access instructions. In each iteration, all *threads* access to a contiguous range of data in the RAM.

efficient use of GPUs. In the context of this paper, since our SP_N is *memory bound* we focus only on rules adapted to *memory bound* algorithms. These rules allow to reduce the number of memory accesses and to optimize the *kernels* so that the whole memory bandwidth can be used.

3.2. Optimization Rules for *Memory Bound* Algorithms

Our test machine was a HP XW6600 with 2 processors Intel Xeon 5410 (quad-core 2.3 GHz) and 8 GB RAM. It was connected to a NVIDIA TESLA D870 Deskside GPU Computing containing 2 C870 graphics cards with 1.5 GB RAM each. Unless specified otherwise, all measurements were made with this configuration. CPU measurements used sequential code whereas GPU measurements were made with one C870. Both the CPU and the GPU were released at the end of 2007. Hence one of the best Intel's CPUs and its equivalent in NVIDIA's GPUs at that time are compared.

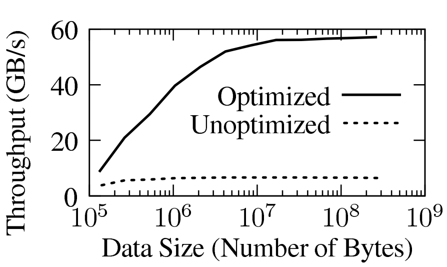
The three optimization rules to be introduced deal respectively with the memory accesses, the algorithm implementation and the degree of parallelism to be expressed. These rules must be applied in order to increase *memory bound* algorithms performances close to the hardware limits.

3.2.1. Optimization Rule #1: Respect the Memory Access Pattern

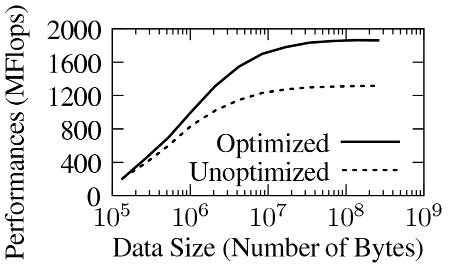
Due to the presence of a cache memory, CPU *threads* must access data continuously to have efficient memory accesses as shown in Fig. 4. On the contrary, GPUs do not have any cache memory. Therefore all *threads* have to collectively access a contiguous range of data so that it can be accessed in one common instruction. Differences of *saxpy* performances depending on the access pattern are shown in Fig. 5(a). When the access pattern is optimized, the measured throughput is close to 60 GB/s while it reaches only 6.5 GB/s otherwise.

3.2.2. Optimization Rule #2: Replace Slow Memory Accesses with Fast Memory Accesses

Some operations require to access the same data over several consecutive iterations of a loop. A generic example of such an operation is given in the following example code:



(a) *Saxpy* Performances With and Without Optimization Rule #1.



(b) `y[i]=x[i]-x[i+1]` Performances With and Without Optimization Rule #2

Figure 5. Performances Achieved with the First Two Optimization Rules.

```
float * x; /* Input */  float * y; /* Output */
for (int i = 0 ; i < N-1 ; i++) y[i] = x[i] - x[i+1];
```

In this example, the data accessed at the iteration `i` in `x[i+1]` will be accessed again at the iteration `i+1` in `x[i]`. On CPU, as the cache memory will store `x[i+1]` between two successive iterations, there is no overhead (see Fig. 4). However, since there is no memory cache on GPU, these redundant accesses need to be removed by the developer. The solution is to store the value in a faster memory between each iteration. We choose to use registers to store these values:

```
float x_i = x[0];        // Local variable in register
for (int i = 0; i<N-1 ; i++){
        const float x_ip1 = x[i+1];
        y[i] = x_i-x_ip1;
        x_i = x_ip1;     // Updating the local variable
}
```

As this modification allows to decrease the number of accesses from $3N$ to $2N$ and as this function is *memory bound*, a speed-up around $3/2$ was expected and actually measured as shown on Fig. 5(b).

3.2.3. Optimization Rule #3: Achieve Massive Parallelism

In order to determine the optimal degree of parallelism for our experimental GPU, the performances of a *saxpy kernel* are measured as a function of the number of GPUs *threads*. Fig. 6(a) shows *saxpy* execution times corresponding to a vector size of 6.7×10^7 elements as a function of the number of GPU *threads* launched. As expected, the time required to perform this operation decreases while the number of *threads* increases. 160 GPU *threads* are needed to match the performance of this particular kernel on one CPU. Fig. 6(b) shows the acceleration compared to the execution with one GPU *thread*. This acceleration is linear up to 4×10^3 *threads* and becomes almost constant after that. The best acceleration ($\times 2410$) is achieved with 8.3×10^6 GPU *threads*. In our case, this implies that each *thread* iterates over eight indices of vectors. This acceleration saturation is explained by Fig. 6(c) which shows the data throughput achieved by this *kernel*. The saturation appears when the throughput is $52\,$GB/s which corresponds to $81\,\%$ of the best throughput measured on our GPU ($62\,$GB/s as seen in Section 3.1). Finally more than $96\,\%$ of this throughput ($60\,$GB/s) is used when we have 8.3×10^6 GPU *threads*. This confirms that the *saxpy* is a *memory bound* function on this device. Finally, this experimentation showed that GPUs require a very fine grained parallelism since 4×10^3 GPU *threads* are necessary to achieve good performances and several millions of GPU *threads* have to be launched to reach optimal performance.

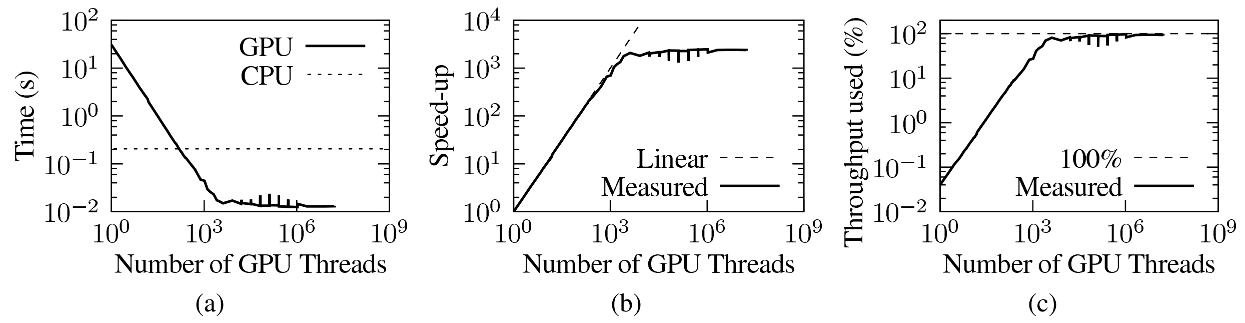


Figure 6. *Saxpy* Performances as a Function of the Number of GPU *threads* for a Vector of 6.7×10^7 elements.

To summarize, GPUs are massively SIMD parallel processors on which best performances are achieved when many *threads* make a few operations and when the memory access constraints are well respected.

4. SP$_\mathrm{N}$ GPU Implementation and Development Strategy

As the SP$_\mathrm{N}$ method and the basics of GPU programming have successively been presented, we may now describe how the power method was implemented on GPU. The developed SP$_\mathrm{N}$ code solves an eigenvalue problem, $A\psi = \frac{1}{k_\mathrm{eff}} F\psi$, using the power method with Chebyshev acceleration. Each power iteration consists in several nested loops:

- a loop on energy groups;
  - a loop on harmonics;
    - a loop on the three Cartesian directions ($d \in X, Y, Z$);
      - directional kernels.

Thanks to the abstraction techniques used to write our CPU implementation, the identification of these loops and the modification of the data structures were relatively facile. Indeed, these techniques, based on C++ templates and exposed in [4], allow a separation between the different algorithms and between the algorithm and the data structures. Consequently, very few places in the code had to be modified in order to launch GPU kernels instead of their CPU counterpart.

Although NVIDIA provides a GPU emulator and a debugger with CUDA 2.1, debugging CUDA code remains a complex exercise. Indeed, the emulator currently distributed does not behave exactly like the hardware while the debugger is presently only available for 32 bit platforms. Therefore, we have adopted a progressive and conservative strategy for the parallelization of our SP$_\mathrm{N}$ code for GPUs. The basic idea was to write the CUDA *kernels* and to integrate them independently. To achieve this, copies of input data from the CPU RAM to the GPU RAM are made before each *kernel* execution. Both the CPU and the GPU versions of the *kernels* are run and GPU results are copied back. Therefore, the two results can be compared and one can ensure that next kernel will get right inputs. Thus, if any problem occurs, the defective kernel is relatively facile to identify. Once all GPU *kernels* are working properly, redundant transfers may be removed as shown on the last line of Fig. 7.

Each loop of our SP$_\mathrm{N}$ code contains some vector operations such as dot products, *saxpys* and scal-

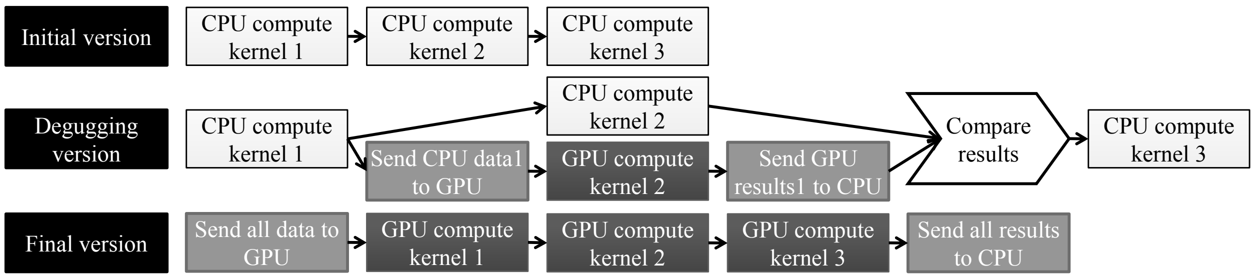


Figure 7. Integration Strategy for GPU *kernels* in an Existing GPU Code. In the initial code, all computation are made on CPU (top). When a GPU *kernel* is written, it is plugged redundantly with the CPU kernel in order to compare the results after each call (middle). Finally, when all *kernels* run properly, all computations are launched on GPU only (bottom).

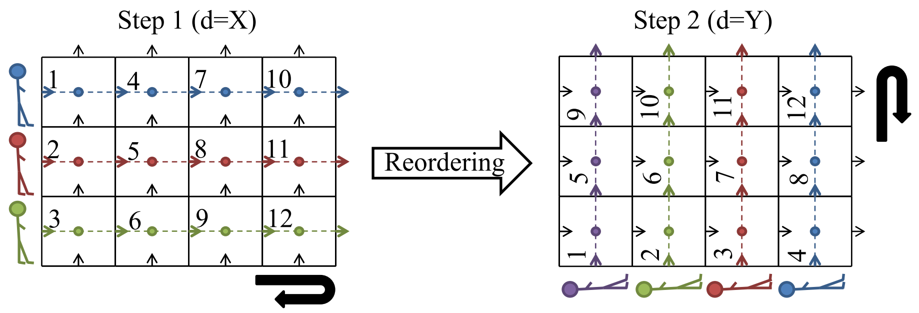


Figure 8. Flux (Circles) and Current (Arrows) SP_N DoFs. The horizontal (left) and vertical (right) lines symbolize the DoFs coupling lines. Colored workers symbolize GPU *threads*.

ing operations. Furthermore, in the loop on directions, several matrix operations (corresponding to Eq. 3 in Section 2) are applied to the vector and scalar DoFs represented respectively by arrows and circles in Fig. 8. An essential property of these matrices is to become block diagonal if the DoFs are numbered along the considered direction d. In other words, these DoFs are only coupled inside current lines that are represented by colored lines on Fig. 8. Consequently, each of these lines can be treated independently. This offers a huge potential of parallel executions. In an industrial pin by pin case with a $289 \times 289 \times 38$ spatial mesh, one has a minimum of 289×38 independent tasks at each step. In Fig. 8 the GPU *threads* are symbolized by small colored workers who have to treat simultaneously the coupled lines of DoFs of the corresponding color. As explained in subsection 3.2.1 contiguous GPU *threads* must simultaneously access contiguous array elements to achieve good performances. In the example of Fig. 8 at step 1 stage 1, workers 1, 2 and 3 address simultaneously the corresponding 1-3 array elements. Then at stage 2, they address simultaneously the 4-6 array elements and continue with 7-9, 10-12 and return to 1-3. When the workers have finished to treat the coupling lines in a given direction, one has to reshuffle the scalar DoFs (circles) in order to treat the next direction in the same way (see Fig. 8 step 2). All operations made here are merged (loop fusions) and then optimized in accordance with the principles presented in subsection 3.2.

The *kernels* used in this implementation have been gathered in 4 categories in accordance with their performances observed on different sizes and their nature:

- `reductions` (dot products and norms);
- `transpositions` (reshuffles);
- `vector expressions` (*saxpy*-like functions);
- `line_computations` (matrix operations made on each line of DoFs).

To summarize, *kernels* have been extracted from our sequential CPU code and have been parallelized and optimized for the GPU. In order to insert these *kernels* safely in our code, a strategy of development facilitating diagnostics has been applied.

5. Experimental Performance Measurements

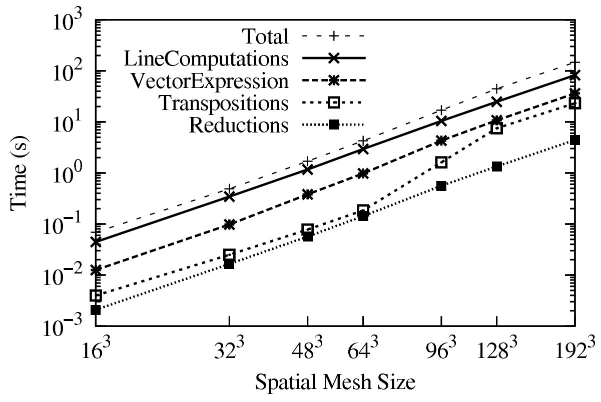
In order to understand the influence of size on performance results, different cubic test cases will be first considered. Then results obtained on a benchmark provided by the IAEA [15] will be discussed with a large rectangular spatial mesh that corresponds to a typical industrial problem size.

5.1. Results on Cubic Test Cases

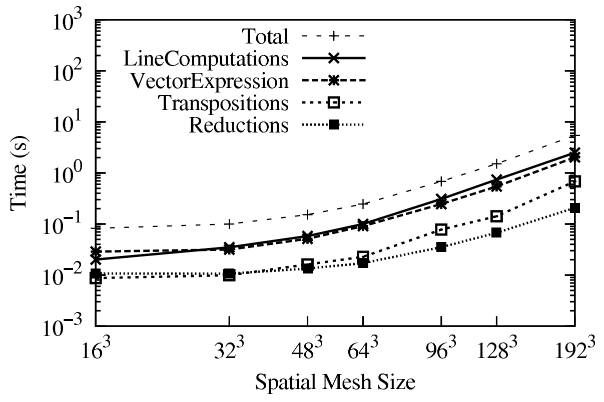
Both the performances of our CPU and GPU implementations are presented in Fig. 9. Note that a parallel version of the code for multicore CPU targets is currently under development. The sequential implementation has been shown in [4] to be competitive with other implementations and that its generic programming style does not imply any overhead. Fig. 9(a) shows the time required to compute k_{eff} on one CPU whereas Fig. 9(b) shows the GPU results. We can observe on Fig. 9(a) that for all *kernels* except the `transpositions` ones, time increases linearly with the number of cells. The performance of the `transpositions` *kernels* depends on whether data fit in cache or not. On the GPU, we observe two regions of performances: below 64^3 cells, time to execute the *kernels* is almost constant whereas it grows linearly above this mesh size. This is due to the fact that under 64^3 cells, the problem does not exhibit sufficient parallelism to exploit the full potential of the GPU. On the contrary, above this limit, `vector expressions` and `line_computations` *kernels* have a measured data throughput above $80\,\%$ of the best throughput measured. `transpositions` and `reductions` *kernels* use about $70\,\%$ of it. Hence this implementation is close to the hardware limits with this algorithm. Consequently no more significant performance improvements can be achieved with this algorithm on GPU. Finally, on our test machine, k_{eff} computation has been accelerated by a factor close to 30 in the 128^3 and 192^3 cases (respectively 29.5 and 26.9).

5.2. Results on Large Rectangular Meshes Corresponding to Industrial Pin by Pin Calculus

On current graphics cards (RAM $\geq 1.5\,\text{GB}$), this implementation allows to handle typical industrial calculations like a 2-group, pin by pin (spatial mesh: $289 \times 289 \times 38$) SP$_3$ (2-harmonics) problem with RT0 elements that leads to 5.1×10^7 single precision degrees of freedom. The performances achieved in k_{eff} computation are measured on an IAEA benchmark case [15] with an industrial spatial mesh ($289 \times 289 \times 38$). Corresponding results are presented in Table II. In these specific cases, the GPU was found to perform between 17 and 21 times faster than the existing CPU version. This is less than on the previous cubic cases ($\times 30$). Indeed, to achieve good performances with our implementation, the spatial mesh size must be multiple of 16. Hence the problem considered has an actual spatial mesh size of $304 \times 304 \times 48$. This is the first explanation for the decrease



(a) Results on a Xeon 5410



(b) Results on a Tesla C870

Figure 9. Performances on Cubic Test Cases with 2 Groups and 1 Harmonic (SP_1).

Table II. Times And Speed-ups for Eigenvalue Computations using a IAEA Benchmark [15] with 2 groups and a $289 \times 289 \times 38$ spatial mesh size using RT0 finite elements.

Processor type	Processor	SP_N	Number of iterations	k_{eff}	Time (s)	Speed-Up ($T_{\text{CPU\_seq}}/T_{\text{GPU}}$)
CPU	Xeon 5410	SP_1	64	1.02898	65.4	-
GPU	Tesla C870	SP_1	64	1.02898	3.1	20.8
GPU	FX5800	SP_1	64	1.02898	2.2	30.2
CPU	Xeon 5410	SP_3	64	1.02946	151.4	-
GPU	Tesla C870	SP_3	64	1.02946	8.8	17.1
GPU	FX5800	SP_3	64	1.02946	5.7	26.7

in the performances compared to previously introduced cubic cases. The second reason is the `transpositions` *kernels* whose performances are almost divided by 2. This issue is currently being addressed in our laboratory.

We finally ran our code on a more recent and powerfull graphics card: the NVIDIA FX5800 which embeds 4 GB RAM (other characteristics are given in Table I). In both SP_1 and SP_3 cases, the performances on the FX5800 are about 1.5 times better than on the C870 (speed-up: between 26 and 31) which is more than the bandwith ratio between these devices (1.37). The `transpositions` *kernels* have indeed a better behavior on this more recent device (performances are not divided by 2 compared with cubic cases).

6. Conclusions et Perspectives

Our sequential SP_N algorithm [4] has been parallelized and implemented for GPU computing. After applying GPU specific optimization, our implementation approaches the hardware limits with respect to measured data throughputs. Only modifications on the algorithm to reduce bandwidth consumption could improve significantly the performances. Speed-up factors up to 30 compared to its sequential CPU counterpart were finally achieved. However, this acceleration requires that all the SP_N data fit in the GPU RAM. On current graphics cards (RAM $\geq 1.5\,$GB), this implementation allows to handle typical industrial calculations like a 2-group, pin by pin (spatial mesh: $289 \times 289 \times 38$) SP_3 (2-harmonics) problem with RT0 elements (5.1×10^7 single precision degrees of freedom). This should be sufficient for a large fraction of core simulations but it is clear that the graphics card RAM size exclude very large SP_N calculations. This limitation could be eliminated by using a multi-GPU implementation. The coupling of the proposed implementation with the domain decomposition method described in [7, 8] should lead to an efficient multi-GPU implementation. We intend to report advances on this issue.

REFERENCES

[1] E. Gelbard. "Simplified Spherical Harmonics Equations and Their Use in Shielding Problems." Tech. rep., WAPD-T-1182 (Rev. 1), Westinghouse Electric Corp. Bettis Atomic Power Lab., Pittsburgh (1961)

[2] J. Lautard, D. Schneider, and A. Baudron. "Mixed Dual Methods for Neutronic Reactor Core Calculations in the CRONOS System." *Proc. Int. Conf. Mathematics and Computation, Reactor Physics and Environmental Analysis of Nuclear Systems, Madrid, Spain, Senda International, SA, Madrid*, pp. 814–826 (1999)

[3] Y. Kodama, A. Yamamoto, Y. Yamane, Y. Ohoka, and M. Tatsumi. "Fast computation of the neutron transport calculation with a graphic processor unit (GPU)." ANS Winter Meeting (2008)

[4] L. Plagne and A. Ponçot. "Generic programming for deterministic neutron transport codes." *Proceedings of Mathematics and Computation, Supercomputing, Reactor Physics and Nuclear and Biological Applications*. Palais des Papes, Avignon, France (2005)

[5] K. Pinchedez. *Calcul parallèle pour les équations de diffusion et de transport homogènes en neutronique (Parallel computation for diffusion and homogeneous transport equations applied to neutron physics)*. Ph.D. thesis, University of Paris XI. In French (1999)

[6] P. Guérin, A. Baudron, and J. Lautard. "Domain Decomposition Methods For Core Calculations Using The Minos Solver." *Joint International Topical Meeting on Mathematics & Computation and super Computing in Nuclear Applications* (2007)

[7] M. Barrault, B. Lathuilière, P. Ramet, and J. Roman. "A domain decomposition method applied to the simplified transport equations." *IEEE 11th International Conference on Computational Science and Engineering, Sao Paulo, Brazil* (2008)

[8] M. Barrault, B. Lathuilière, P. Ramet, and J. Roman. "A non overlapping parallel domain decomposition method applied to the simplified transport equations." *Proceedings of Mathematics, Computational Methods & Reactor Physics (M&C 2009)*. Saratoga Springs, New York, USA (2009)

[9] E. E. Lewis and W. F. Miller, Jr. *Computational Methods of Neutron Transport*. John Wiley & Sons (1984)

[10] NVIDIA. *CUDA* Programming Guide *2.0* (2008)

[11] M. J. Flynn. "Very high-speed computing systems." *Proc. IEEE*, **vol. 54(12)**, pp. 1901–1909 (1966)

[12] D. T. Marr, D. P. Group, and I. Corp. "Hyper-threading technology architecture and microarchitecture." *Intel Technology Journal*, **vol. 6**, p. 2002 (2002)

[13] W. A. Wulf and S. A. McKee. "Hitting the memory wall: implications of the obvious." *SIGARCH Comput. Archit. News*, **vol. 23(1)**, pp. 20–24. ISSN 0163-5964 (1995)

[14] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick. "The landscape of parallel computing research: A view from berkeley." Tech. rep., EECS Department, University of California, Berkeley (2006)

[15] B. Micheelsen. *2D-3D IAEA Benchmark Problem*, vol. ANL-7416 sup.2. Argonne National Laboratory (1977)