POLLACK PERIODICA
An International Journal for Engineering and Information Sciences
DOI: 10.1556/Pollack.10.2015.1.1
Vol. 10, No. 1, pp. 3–18 (2015)
www.akademiai.com

FAST ITERATIVE SOLVERS FOR LARGE COMPRESSED-SPARSE ROW LINEAR SYSTEMS ON GRAPHICS PROCESSING UNIT

[1]Frédéric MAGOULÈS, [1]Abal-Kassim CHEIK AHAMED
[2] Roman PUTANOWICZ

[1] Ecole Centrale Paris, France, e-mail: frederic.magoules@hotmail.com
[2] Cracow University of Technology, Poland

Received 29 April 2014; accepted 17 June 2014

Abstract: Engineering problems involve the solution of large sparse linear systems, and require therefore fast and high performance algorithms for algebra operations such as dot product, and matrix-vector multiplication. During the last decade, graphics processing units have been widely used. In this paper, linear algebra operations on graphics processing unit for single and double precision (with real and complex arithmetic) are analyzed in order to make iterative Krylov algorithms efficient compared to central processing units implementation. The performance of the proposed method is evaluated for the Laplace and the Helmholtz equations. Numerical experiments clearly show the robustness and effectiveness of the graphics processing unit tuned algorithms for compressed-sparse row data storage.

Keywords: Krylov methods, Iterative methods, Linear algebra, Dot product, Sparse matrix-vector product, Graphics processing unit, Single instruction multiple data, Compute unified device architecture, Dynamic tuning, Compressed-sparse row format

1. Introduction

Solving engineering problems require a computing effort to solve large sparse linear systems. These large sparse linear systems arise for instance from the discretization by finite element method and their solution requires fast and high performance algorithms, especially for the algebra operations; in particular for dot product, and matrix-vector multiplication. These operations could be very expensive in terms of computing time for classical Central Processing Units (CPU). In recent years, Graphics Processing Unit (GPU) has been commonly used to perform these operations due to their massively parallel architecture. Graphic processing units have become more and more used for

massively parallel computations even if they require specific algorithms design to match hardware constraints. The recent TOP 500 of most powerful supercomputers shows that General Purpose Graphic Processing Unit (GPGPU), a combination of general purpose processing unit with CPU architecture and graphic processing units with a Single Instruction Multiple Data (SIMD) programming model, provides the most powerful and efficient computational machines. Over the last decade scientists and researchers had paid more attention on GPU Computing to accelerate their programs. GPUs reach much greater performance than CPUs, and remain in the same price range. Nowadays graphics cards have become attractive tools to provide the high computing power in numerical simulations, especially for large sparse linear system solution [1], [2], [3], [4].

In this paper, an original computer implementation is proposed to solve the Laplace (real number) and the Helmholtz (complex number) equations using high order finite elements on Graphics Processing Units. Detailed comparison of both simple and double precision are presented, and showed the impact of accuracy when one deals with GPU computing. This paper focuses on two Krylov methods: Conjugate Gradient (CG), which is currently the most popular and probably the best method for systems with real symmetric positive definite matrices, and ORTHODIR for the solution of sparse linear systems with complex non-symmetric matrices. These two Krylov methods are implemented to operate on matrices stored in Compressed-Sparse Row (CSR) format.

The paper is organized as follows: section 2 briefly describes the GPU programming model with CUDA for readers not familiar with GPU programming. In section 3, key points of Conjugate Gradient (CG) and ORTHODIR iterative Krylov algorithms are presented. Section 4 gives an overview of linear algebra operations underlying these iterative methods. Section 5 collects numerical results of both linear algebra operations and iterative solvers for solving $\mathbf{Ax} = \mathbf{b}$ with matrices arising from finite element discretization of 2-dimensional (2D) and 3-dimensional (3D) Laplacian and Helmholtz equations respectively. Section 6 concludes the paper.

2. GPU programming model

GPU Computing or GPGPU model consists in offloading some time intensive computation of CPU to GPU device. In other word, GPGPU is based on host/CPU-device/GPU model where the CPU is in charge of handling memory data, copy it to and from the device and scheduling kernels of computations to the device. Driven by the demand of gaming, GPU have quickly evolved during the past decade, as it can be can seen in *Fig. 1*, which presents the evolution of the peak performance of CPUs and GPUs over the years. Their actual high peak computational power is gigantic: around the teraflops in simple precision and half in double precision for only one graphic card. The beginning of the GPU development coincides with the first steps of the GPU programming through the vertex shaders for 2D and 3D display. The graphic cards are specially projected to perform the renderization and manipulation of graphic objects. Nowadays, a remarkable feature of GPUs is their huge amount of processing elements, which may be applied in parallel programming. The first units designed to deal with

graphic objects appeared between the decades of 1970 and 1980, and were responsible for simple graphic functions including displaying characters, lines and arcs.

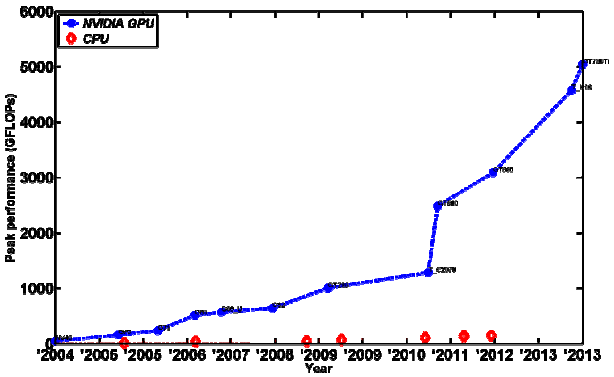

Fig. 1. Evolution of both CPUs and GPUs peak of performance

First GPGPU applications articles appear in the early 2000s with [5], [6], [7], signaling already huge computational capabilities. Vertex and pixel shaders programming progressively leads to computational-oriented languages (but still with graphics notion) as Cg, for the most common use, allowing GPGPU to spread over several scientific domains, see [8] and [9]. Finally, actual standard of programming languages are Compute Unified Device Architecture (CUDA) [10] and Open Computing Language (OpenCL) [11] are likely to generalize GPGPU at all the scientific application fields, one of the reasons being they are based on C/C++ language.

In order to understand the most out of GPU Computing, first the differences between CPU and GPU architectures, and their features are presented. The basic difference between the processing models in a CPU and a GPU is that the CPUs are optimized for single-threaded performance, whereas GPUs are designed to optimize parallel executions. The simplified architecture of a CPU processor consists of Arithmetical and Logical Unit (ALUs), which are units of computations, a complex unit control, and lots of memory with multiple levels of associated cache.

On the other hand, the simplified architecture of a GPU processor is very different; the ALUs are multiplied to reach several hundreds. They are divided into sub-processors, which include a simplified unit of control and some associated memory. Generally, the global GPU memory is smaller than CPU main memory. *Fig. 2* illustrates the inherent differences between CPU and GPU architectures. The GPU architecture promotes the usage of GPGPU in a massively parallel environment. The algorithms are designed to break down into a number of threads running in parallel in groups of hundreds of ALU graphics cards. Typically there are thousands or even millions of threads that are executed at the same time. However, it is not possible to use GPU without the support of a CPU. Indeed, the GPU architecture is limited to performing specific calculations but does not take over the entire code execution.

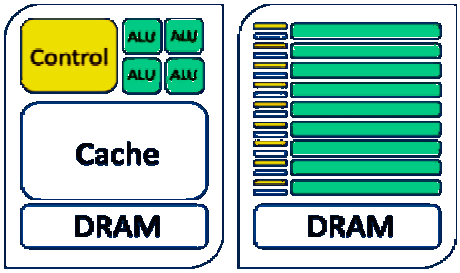

Fig. 2. Difference of CPU (left) and GPU (right) architectures

The CPU therefore requires the graphics card to the steps of rough calculations, but is the only one to manage the whole code, that is the concept of CPU-GPU hybridization. When the number of cores on the CPU hardly reaches 12 cores, the number of current GPU reaches a thousand cores. At the beginning of GPGPU, GPU computations have been available only in single precision, as single precision has been sufficient to compute display geometry. Now, motivated by the demand of engineering simulations, CUDA supports double precision computations as well.

CUDA, is a model of parallel programming that handles transparently the power of a graphics card composed of several multiprocessors. This programming model is an extension of C/C++ language as can be seen in the function named *kernel* described in *Algorithm 1*. However support for other programming languages like, FORTRAN, Python, Java is under active development. Official guide references of CUDA architecture are available in references [10], [12], [13].

***Algorithm 1**: Saxpy CUDA Kernel (CUDA/C++)*

```
// T is float (Saxpy) or double (Daxpy), and the complex equivalent, and U is int
template <class T, class U>
_ _global_ _ void Saxpy ( const T alpha , const T * d_x , const T * d_y , U size ) {
   unsigned int x = threadIdx.x  + blockIdx.x * blockDim.x;
   unsigned int y = threadIdx.y  + blockIdx.y * blockDim.y;
   unsigned int pitch = blockDim.x * gridDim.x;
   unsigned int idx = x + y * pitch;
   if ( idx < size ) {
     d_y[idx] = alpha * d_x[idx] + d_y[idx];
   }
}
```

CUDA architecture is composed of a set of *grids*, *blocks* and *threads*. Each grid has blocks of threads, which are executed on the device. One block is executed by one core of the GPU, and the threads within the blocks communicate using Shared Memory (SM). This decomposition preserves language expressivity by allowing threads to cooperate within the threads on each block. In [1], [14] authors proved that performance strongly depends on the gridification (arrangement of threads), and a best tuning of the grid upon the hardware features promotes good performance. *Fig. 3* describes the CUDA processing flow, which explains:

- data are first copied from the main CPU/host memory (DRAM) to the GPU/device global memory;
- host instructs the device to carry out computations;
- execute parallel computations in each core;
- device results are copied back from global memory to main host memory.

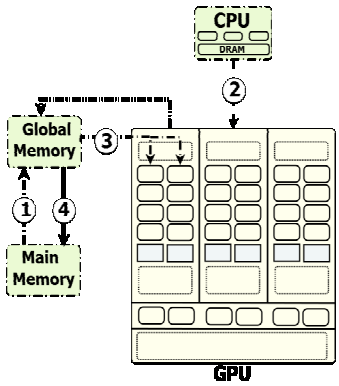

Fig. 3. GPGPU processing flow

The GPU performs a given function named *kernel,* an example of kernel is given in the *Application Programming Interface (API 1)* of *Alinea*, a new library developed by the authors, on large amount of data. This kind of architecture is called *SIMD*.

Data needs to be well organized in memory. In [14], authors demonstrated that memory management is a key issue when we deal with GPU Computing. They showed that performance also depends on the type of memory used to store data and the way one access to data. Contiguous access generally gives better results than non-contiguous one.

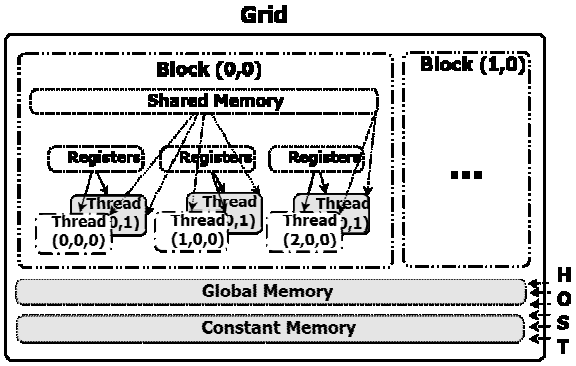

Fig. 4. CUDA memory hierarchy

CUDA architecture proposes four main memory types as presented in *Fig. 4*: *global*, *local*, *shared* and *constant*. *Global memory* and *constant memory* are accessible by all

threads of the grid. *Local memory* is accessible only by each thread. *Shared memory* is shared by the threads of the same block. It is faster than global memory. The *global* and *constant* memories are the ones able to exchange data with the host CPU solving each sub-problem, and at the same time enable automatic scalability. An example of 2D grid is given in Fig. 5 where each thread and block are identified by *x, y* coordinates, which can be accessed by the programmer through CUDA's built-in variables. The distribution of threads (gridification) is not an automated process. For each kernel, the number of blocks is set that it will be used in the grid, which depends on the graphics card properties, and the memory. All threads have read and write access to global memory, but read-only for constant memory. Local memory is used to share data between threads of the same block.

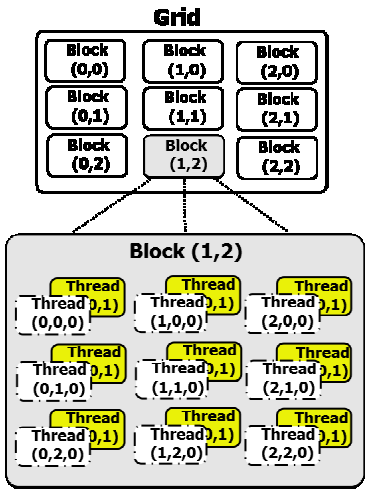

Fig. 5. Gridification: thread, block, grid

The kernel function needs at least two more arguments for defining the gridification: the number of threads per block, *nThreadsPerBlock*, and the dimension of the block, *nBlocks*. These values depend on the characteristics of the graphic card and the amount of data. A basic example with a one-dimensional grid is given as follows:

API 1: *Saxpy (float) CUDA Kernel calling (CUDA/C++)*

```
Saxpy<float, int><<<nBlocks, nThreadsPerBlock >>>( alpha, d_x, d_y, size );
```

3. Principle of the Conjugate Gradient and ORTHODIR method

Conjugate Gradient method is a member of Krylov iterative methods [15]. CG method is probably the best iterative Krylov method for systems with symmetric positive definite matrices. Krylov methods are based on the linear subspace spanned by the first p powers of $\mathbf{A}$, i.e. $\mathbf{K}_p(\mathbf{A},\mathbf{r}_0) = span\{\mathbf{r}_0, \mathbf{A}\mathbf{r}_0, \mathbf{A}^2\mathbf{r}_0, \cdots, \mathbf{A}^{p-1}\mathbf{r}_0\}$. The main

principle relies on a short recurrence for the approximate solution update step: $\mathbf{x}_k = \mathbf{x}_{k-1} + \rho \cdot \boldsymbol{\omega}_k$, where $\boldsymbol{\omega}_k$ represents the descent-director vector and ρ_k is the descent coefficient. Next consists in the minimization of the distance (gradient vector) of the iterative solution to the exact solution: $\mathbf{r}_k = \mathbf{b} - \mathbf{A} \cdot \mathbf{x}_k$. By using the successive $\mathbf{A}$ *-orthogonal* gradient vectors:

$$\boldsymbol{\omega}_k = \sum_{i=0}^{p} \alpha_i \mathbf{r}_i \,, \tag{1}$$

the basis $\mathbf{W}_k = \{\boldsymbol{\omega}_k\}_k$ becomes $\mathbf{A}$ *-orthogonal* and the updates can be performed in short recurrences with one single coefficient. The symmetric feature of the real matrix is the key point of this method. The goal is to have in n iterations the residual $\mathbf{r}_k \to 0$.

Algorithm 2 and *Fig. 6* respectively present the CG algorithm and the processing flow of GPU CG algorithm. Both the CPU and GPU codes are similar except that linear algebra operations are performed on graphics card for the GPU version. Communication and transfer between host and device form the crucial task in GPU computing. According to the amount of data, communication requires special attention for avoiding escalation of the associated cost, as showed in [13]. In this way, in the proposed GPU algorithms more attention about communication is carried our. As described in *Fig. 6* the input data $(\mathbf{b}, \mathbf{A}, \mathbf{x}_0)$ are copied before the start of the Conjugate Gradient algorithm. At the initialization step, all required data are located on GPU memory for the GPU version code. When the convergence is ensured, the final solution is copied back to CPU. The most computational time is spent in the matrix-vector product step.

The memory on GPU cards is generally smaller than on CPU, so it is important to wisely manage the vectors on GPU, which explains the importance of the function *Move* used in CG algorithm. ORTHODIR method requires a basis $\mathbf{W}_k = \{\boldsymbol{\omega}_k\}_k$ $\mathbf{A}^t\mathbf{A}$ *-orthogonal* of $\mathbf{K}_p$ in order that $\mathbf{W}_k^t \mathbf{A}^t \mathbf{W}_k$ is diagonal. To do that, an ortho-normalization process of Gram-Schmidt modified to vectors obtained by successive product by the matrix for the product associated with $\mathbf{A}^t\mathbf{A}$ is applied. This approach results in the algorithm given in *Algorithm 3*. The implementation of this algorithm has the same process as in the Conjugate Gradient one, with similar code for CPU and GPU except that all operations are performed on GPU cards for GPU version. As in CG code, required input data have been copied from CPU to GPU before the start of the algorithm, and one copy is done to copy back the solution, when algorithm converges. Nevertheless, at each computation of dot product, the result is copied back to CPU for both algorithms.

4. Linear algebra operations

In this part of the paper, the main linear algebra algorithms are introduced briefly and numerical results are reported and analyzed.

Algorithm 2: *Conjugate Gradient method*

Data:
- **A** (symmetric positive definite matrix
- **x** (solution)
- **b** (right hand side)
- ε (residual threshold

Result: **x**

{*Initialization*}

$\mathbf{r}_0 \leftarrow \mathbf{b} - \mathbf{A} \cdot \mathbf{x}_0$ // $\mathbf{x}_0$ is the initial guess

$\boldsymbol{\omega}_0 \leftarrow \mathbf{r}_0$ // $\mathbf{r}_0$ is the initial residual

{*Iterations*}

// k: number of iteration

for $k \leftarrow 0$ to $k \leftarrow it_{\max}$ ***do***

// Update descente coefficient

$$\alpha \leftarrow -\frac{(\mathbf{r}_k, \boldsymbol{\omega}_k)}{(\mathbf{A} \cdot \boldsymbol{\omega}_k, \boldsymbol{\omega}_k)}$$

// Update solution and residual

$\mathbf{x}_{k+1} \leftarrow \mathbf{x}_k + \alpha \cdot \boldsymbol{\omega}_k$

$\mathbf{r}_{k+1} \leftarrow \mathbf{r}_k - \alpha \cdot \boldsymbol{\omega}_k$

if $\|\mathbf{r}_k\| < \varepsilon$ ***then***

break;

end if

// Update descent-director vector

$$\beta \leftarrow -\frac{(\mathbf{r}_k, \mathbf{A} \cdot \boldsymbol{\omega}_k)}{(\mathbf{A} \cdot \boldsymbol{\omega}_k, \boldsymbol{\omega}_k)}$$

$\boldsymbol{\omega}_{k+1} \leftarrow \mathbf{r}_k + \beta \cdot \boldsymbol{\omega}_k$

end for

Algorithm 3: *ORTHODIR method*

Data:
- **A** (general matrix)
- **x** (solution)
- **b** (right hand side)
- ε (residual threshold

Result: **x**

{*Initialization*}

$\mathbf{r}_0 \leftarrow \mathbf{b} - \mathbf{A} \cdot \mathbf{x}_0$

$\boldsymbol{\omega}_0 \leftarrow \mathbf{r}_0$

$\mathbf{A}\boldsymbol{\omega}_0 \leftarrow \mathbf{A} \cdot \boldsymbol{\omega}_0$

$$\boldsymbol{\omega}_0 \leftarrow \frac{\boldsymbol{\omega}_0}{\|\mathbf{A}\boldsymbol{\omega}_0\|}$$

$$\mathbf{A}\boldsymbol{\omega}_0 \leftarrow \frac{\mathbf{A}\boldsymbol{\omega}_0}{\|\mathbf{A}\boldsymbol{\omega}_0\|}$$

{*Iterations*}

for $k \leftarrow 0$ to $k \leftarrow it_{\max}$ ***do***

$\alpha \leftarrow -(\mathbf{r}_k, \mathbf{A}\boldsymbol{\omega}_k)$

// Update solution and residual

$\mathbf{x}_{k+1} \leftarrow \mathbf{x}_k - \alpha \cdot \boldsymbol{\omega}_k$

$\mathbf{r}_{k+1} \leftarrow \mathbf{r}_k - \alpha \cdot \boldsymbol{\omega}_k$

if $\|\mathbf{r}_k\| < \varepsilon$ ***then***

break;

end if

for $i \leftarrow 0$ to $i \leftarrow k$ ***do***

$\alpha_i \leftarrow -(\mathbf{A} \cdot \mathbf{A}\boldsymbol{\omega}_i, \mathbf{A}\boldsymbol{\omega}_i)$

end for

$\boldsymbol{\omega}_{k+1} \leftarrow \mathbf{A}\boldsymbol{\omega}_k + \sum_i \alpha_i \cdot \boldsymbol{\omega}_i$

$\mathbf{A}\boldsymbol{\omega}_{k+1} \leftarrow \mathbf{A} \cdot \mathbf{A}\boldsymbol{\omega}_k + \sum_i \alpha_i \cdot \mathbf{A}\boldsymbol{\omega}_i$

$$\boldsymbol{\omega}_k \leftarrow \frac{\boldsymbol{\omega}_k}{\|\mathbf{A}\boldsymbol{\omega}_k\|}$$

$$\mathbf{A}\boldsymbol{\omega}_k \leftarrow \frac{\mathbf{A}\boldsymbol{\omega}_k}{\|\mathbf{A}\boldsymbol{\omega}_k\|}$$

end for

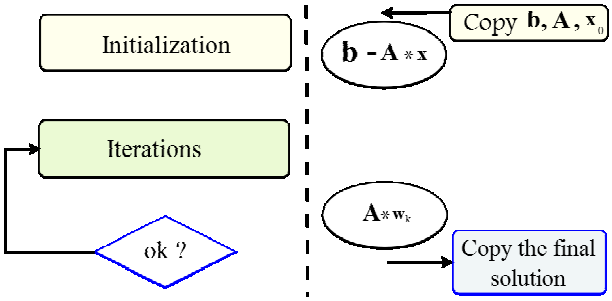

Fig. 6. Iterative Krylov algorithm, with GPU statement

On one hand basic linear algebra operations are presented. This corresponds to the *Basic Linear Algebra Subprograms* (BLAS) level one (vector) operations: addition of vectors (Saxpy for single precision, Daxpy for double precision), dot product and norm. On the other hand, sparse matrix-vector products are described as advanced operation. This corresponds to the BLAS level two. These BLAS operations are implemented in CUDA C++ by the authors and are denoted by $\text{Alinea}_{\text{BLAS}}$ in the following. Generally the matrix arising from finite element analysis of a large size problem is large and is sparse, which means that it contains many zero values. In order to optimize memory storage and allow easy memory access, we propose to compress the matrix in Compressed-Sparse Row (CSR) format, which is probably the most popular format used in sparse matrix computations. Let's take the following matrix to illustrate the format. An example of CSR matrix of the dense matrix (6×6) given in equation (2) is illustrated in equation (3).

$$\mathbf{A}=\begin{bmatrix} 0 & \alpha & 0 & 4 & 0 & 2 \\ \beta & 0 & 0 & 0 & 0 & 9 \\ 0 & 0 & \gamma & 9 & 0 & 5 \\ 0 & \delta & 1 & 0 & 8 & 0 \\ \theta & 0 & 0 & 0 & 7 & 0 \\ 0 & 0 & \mu & 0 & 0 & 6 \end{bmatrix}, \tag{2}$$

$$\text{where} \quad \begin{cases} \alpha \text{ and } \beta \ 1^{st} \text{ values of the } 1^{st} \text{ and } 2^{nd} \text{ row}, \\ \gamma \text{ and } \delta \ 1^{st} \text{ values of the } 3^{rd} \text{ and } 4^{th} \text{ row}, \\ \theta \text{ and } \mu \ 1^{st} \text{ values of the } 5^{th} \text{ and } 6^{th} \text{ row}. \end{cases}$$

CSR storage consists in two arrays **JA** and **AA** of size *nnz* storing respectively the nonzero values and the column numbers of nonzero coefficients in major row storage, *i.e.* row by row. A third array, **IA**, of size $n+1$ where n is the size of the

matrix that stores the list of indices (references in the *nnz* arrays) of the first nonzero values of each row. The last case of **IA** is set to $n+1$

$$\begin{aligned}
\mathbf{JA}: &\ [2\ \ 4\ \ 6\ \ 1\ \ 6\ \ 3\ \ 4\ \ 6\ \ 2\ \ 3\ \ 5\ \ 1\ \ 5\ \ 3\ \ 6], \\
\mathbf{AA}: &\ [\alpha\ \ 4\ \ 2\ \ \beta\ \ 9\ \ \gamma\ \ 9\ \ 5\ \ \delta\ \ 1\ \ 8\ \ \theta\ \ 7\ \ \mu\ \ 6], \\
\mathbf{IA}: &\ [1\ \ 4\ \ 6\ \ 9\ \ 12\ \ 14\ \ 16\ \ 0\ \ 0\ \ 0\ \ 0\ \ 0\ \ 0\ \ 0\ \ 0]
\end{aligned} \tag{3}$$

Numerical experiments have been performed on platform composed of four CPUs Intel Core i7 920. Each CPU is a Dual-core processor running at 2.66 GHz. It supplies 8GB RAM memory. GPU architecture is based on two NVIDIA GeForce GTX570 cards equipped with 1279MB memory. The workstation is compatible with CUDA model and supports both single and double precision arithmetic. In this paper, CUDA 4.0 [10] is used. The benchmark presented here consists in performing 10 times the same operation in order to overcome the problem caused by the finite precision of the clock of graphics cards, which corresponds to a few nanoseconds, while the host has an accuracy of a few milliseconds.

Real/Complex *Alpha X Plus Y*, or addition of vectors, is an operation between two vectors. This operation is inherently parallel making it an excellent candidate for implementation on GPU. An example of *Axpy* GPU kernel (*Saxpy*) is given in *Algorithm 1*. The dot product and norm algorithms used in this part are the one described by the authors in [14], [2].

When engineering and applied science problems are solved by numerical methods, for instance the finite element, the computation of sparse matrix-vector product (*SpMV)* is probably the key point to optimize to gain performance. This operation is presented as the most important and most expensive operation in matrix computation. The kernel of matrix-vector multiplication consists in handling a thread for the calculation of each row. However, the calculation of a row may require a lot of operations if the row contains several nonzero elements. Bell and Garland showed in [16] and [17] that one has better performance when uses multiple threads for the computation of a row. Moreover, the use of shared memory for each block is also fundamental for the performance gain. Authors demonstrated in [14], [1], [2] that the distribution of nonzero elements has a strong impact on the CSR matrix-vector product performance on GPU. The implementation performed by the authors of this paper takes care to use CUDA shared memory in order to reduce direct global memory access. As NVIDIA graphics cards use coalesced memory transfers, then the memory calls are re-ordered to fit with and take advantage of the coalescing. According to the size of the problem and the capacity of different memories of the graphics card, the problem is cut in grids and blocks, but must be paid to share the task in respect with the load-balancing (auto-tuning [14]). References [18], [19], [20] confirm the efficiency of the auto-tuning of the grid in an environment of distributed parallel computing such as domain decomposition methods. Formats of matrices impact the performances of algorithms, see [21]. References [22] and [23] also showed that classical iterative methods, for instance the Jacobi method are also efficient on GPU with CUDA.

5. Numerical analysis

In this section first numerical results of linear algebra operations for real and complex arithmetic are reported. These complex vector consist of a $2 \times n$ one-dimensional array, where n is the size of the problem. Two successive values a, b of the array correspond to one complex number $a + ib$. CPU and GPU algorithms are compared in simple and double precision for two test cases: 2D and 3D. The first test case arises from the discretization of Laplacian problems [24] [25], [26] applied to image reconstruction (for 2D and 3D image rendering). The second test case arises from the finite element discretization acoustics problems, similar to those presented in [27] for an open roof-car compartment (in 2D and 3D), but with a discretization with stabilized finite element [28] within the domain and with infinite element on the artificial boundary conditions defined on the truncation boundary of the domain [29] [30] [31]. These linear algebraic operations implemented in CUDA C++ by the authors are denoted in the following by $\text{Alinea}_{\text{BLAS(1)}}$ for vector-vector operations (Saxpy, Dot and Norm) and $\text{Alinea}_{\text{BLAS(2)}}$ for SpMV operations. Reference [32] gives an analysis of SpMV using iterative method for real number arithmetic. Numerical methods, including domain decomposition methods (DDM), require the solution of linear systems. So, optimal performance of linear algebra operations encourage better results of DDM, as described in [33] for real number arithmetic. *Fig. 7* and *Fig. 8* respectively present the time ratio of CPU/GPU of real $\text{Alinea}_{\text{BLAS}}$ 2D and 3D in single and double precision. Complex comparisons are presented in *Fig. 9* and *Fig. 10*.

e The results show a speed-up as high as 12 and 9 for 2D and 3D single precision $\text{Alina}_{\text{BLAS(1)}}$. A ratio up to 6 is obtained for double precision $\text{Alina}_{\text{BLAS(1)}}$. $\text{Alinea}_{\text{BLAS(2)}}$ single and double precision presents respectively a speed-up of 7 and 4 for 2D and a speed-up of 5 for 3D.

For double precision curves, the slope is constant but does not decrease. The difference shown in these results between single and double precision treatment is not due only to the number of ALU but also depends on the memory access, therefore on the width of the bandwidth.

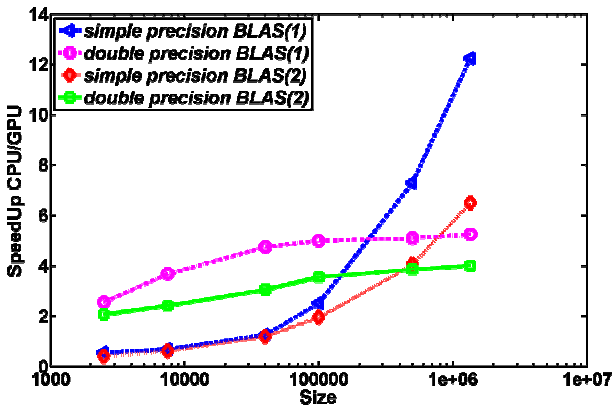

Fig. 7. Real $\text{Alinea}_{\text{BLAS}}$ (2D)

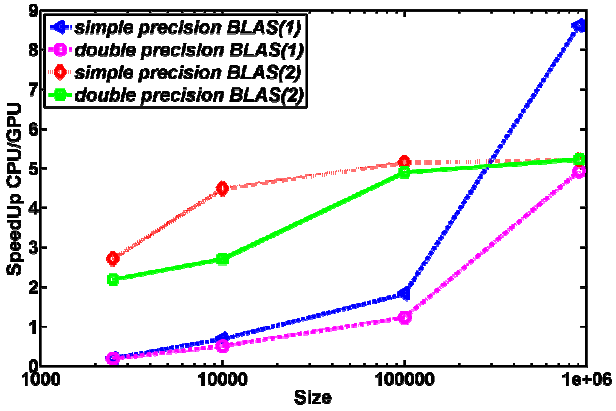

Fig. 8. Real Alinea$_{BLAS}$ (3D)

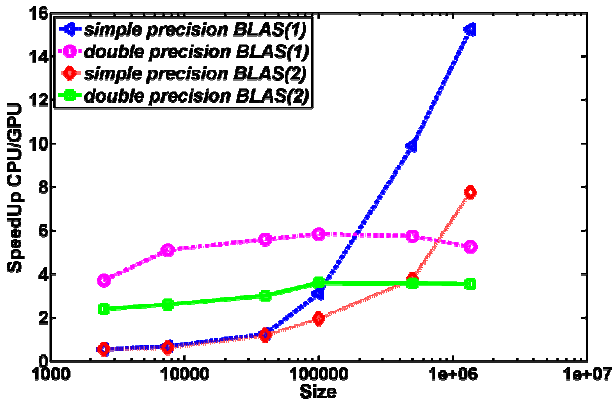

Fig. 9. Complex Alinea$_{BLAS}$ (2D)

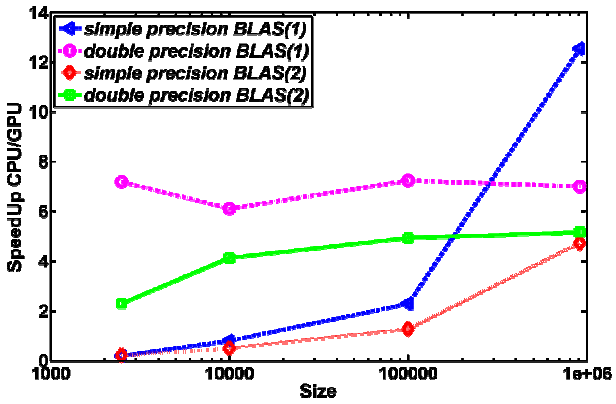

Fig. 10. Complex Alinea$_{BLAS}$ (3D)

Numerical results are now collected for both algorithms: CG and ORTHODIR. The matrices used for CG (real) and ORTHODIR (complex) arise respectively from the finite element discretization of the Laplace and the Helmholtz equations. In *Fig. 11* compares the speed-ups CPU/GPU of the conjugate gradient in simple and double precision for 2D case, and *Fig. 12* presents the ratios CPU/GPU for the 3D case. Complex value problems are solved with ORTHODIR and results of speed-ups are reported in *Fig. 13* and *Fig. 14* for 2D and 3D test cases respectively. *Fig. 11* and *Fig. 12* show CG method for 2D case and outline ratios up to 9 for single precision and up to 5 for double precision, and 3D case shows a speed up of 7 for single precision and up to 5 for double precision. The memory of the graphics card forces to restart the ORTHODIR algorithm for a few tens iterations. Nevertheless, the speedup of ORTHODIR 3D reaches a speed-up of 9 for single precision and a speed-up of 7 for double precision.

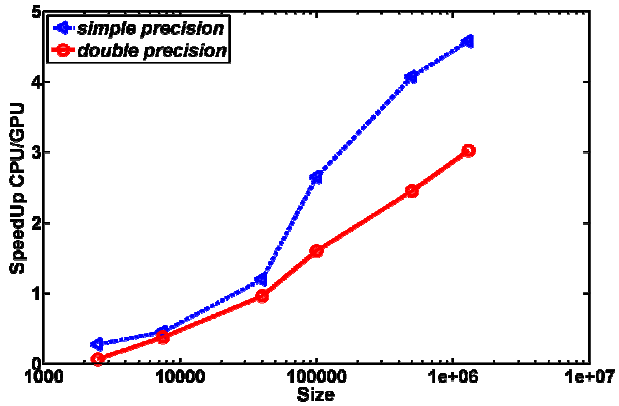

Fig. 11. Conjugate Gradient (2D)

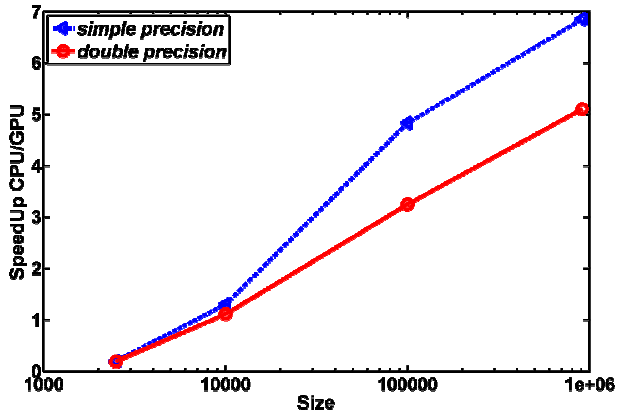

Fig. 12. Conjugate Gradient (3D)

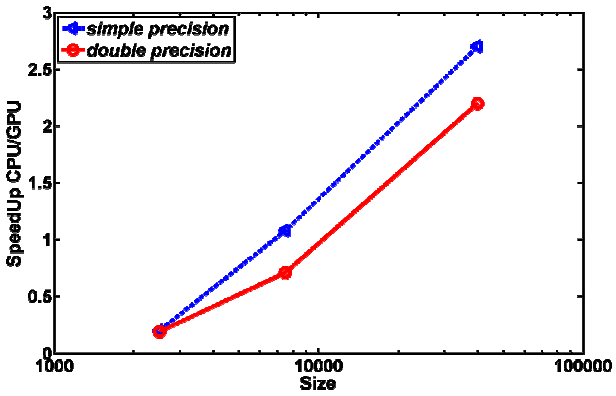

Fig. 13. ORTHODIR (2D)

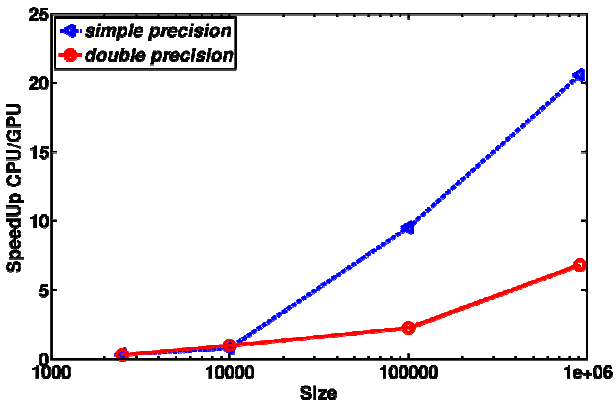

Fig. 14. ORTHODIR (3D)

6. Conclusion

This paper analyzes linear algebra operations on GPU and shows the performance evaluation of the Conjugate Gradient and the ORTHODIR algorithms tuned on GPU to solve large size linear systems. These linear systems arise from the finite element discretization of the Laplacian (real number) and the Helmholtz equations (complex number). Results on GPU are compared with those on CPU for single and double precision. Numerical experiments clearly showed that Conjugate Gradient method achieves a speed-up of 10 times for single and 6 times for double against 10 times for ORTHODIR with sufficient trade-off between the size of the problem and the restart parameter.

Acknowledgements

The Authors acknowledge partial financial support from the OpenGPU project (2010-2012), and the CUDA Research Center at Ecole Centrale Paris, France.

References

[1] Cheik Ahamed A. K., Magoules F. Iterative methods for sparse linear systems on graphics processing unit, *14th IEEE International Conference on High Performance Computing and Communication*, Liverpool, UK, 25-27 June 2012, pp. 836–842.

[2] Cheik Ahamed, A. K., Magoules F. Iterative Krylov methods for gravity problems on graphics processing unit, *12th International Symposium on Distributed Computing and Applications to Business, Engineering Science* (*DCABES*), Kingston upon Thames, Surrey, UK, 2-4 September 2013, pp. 16–20.

[3] Bahi J. M., Couturier R., Khodja L. Z. Parallel GMRES implementation for solving sparse linear systems on GPU clusters, *19th High Performance Computing Symposia*, Boston, MA, USA, 03-07 April 2011, pp. 12–19.

[4] Matam K. K., Kothapalli K. Accelerating sparse matrix vector multiplication in iterative methods using GPU, *2011 International Conference on Parallel Processing (ICPP'11)*, Taipei, Taiwan, 13-16 September 2011, pp. 612–621.

[5] Thompson C. J., Hahn S., Oskin M. Using modern graphics architectures for general-purpose computing: a framework and analysis, *35th Annual ACM/IEEE International Symposium on Microarchitecture*, Los Alamitos, CA, USA, 3-7 April 2002, pp. 306–317.

[6] Bolz J., Farmer I., Grinspun E., Schröoder P. Sparse matrix solvers on the GPU: conjugate gradients and multigrid, *ACM Trans. Graph.* Vol. 22, No. 3, 2003, pp. 917–924.

[7] Krüger J., Westermann R. Linear algebra operators for GPU implementation of numerical algorithms. *ACM Trans. Graph*, Vol. 22, 2003, pp. 908–916.

[8] Meredith J., Bremer D., Flath L., Johnson J., Jones H., Vaidya S., Frank R. The GAIA Project: Evaluation of GPU-based programming environments for knowledge discovery, Bericht, *Lawrence Livermore National Labs*, 2004.

[9] Owens J. D., Luebke D., Govindaraju N., Krüger M., Harris J., Lefohn A. E., Purcell T. J. A survey of general-purpose computation on graphics hardware, *EUROGRAPHICS Conference*, ACM, Dublin, Ireland, 29 August-2 September 2005, pp. 21–51.

[10] NVIDIA: CUDA toolkit reference manual, 4th ed, *http://developer.nVidia.com/cuda-toolkit-40* (last visited 19 March 2012).

[11] Khronos Group Inc., OpenCL Specification (1.0.29), *OpenCL Working Group*, 12/8/2008.

[12] NVIDIA: CUDA Programming Guide, 4th ed., *http://developer.nVidia.com/cuda-toolkit-40*, (last visited 07 March 2014).

[13] NVIDIA: CUDA 4.0 Best Practices Guide, 4th ed., *http://developer.nVidia.com*, (last visited 07 March 2014).

[14] Cheik Ahamed A. K., Magoules F. Fast sparse matrix-vector multiplication on graphics processing unit for finite element analysis, *14th IEEE International High Performance Computing and Communication*, Liverpool, UK, 25-27 June 2012, pp. 1307–1314.

[15] Saad Y. Iterative methods for sparse linear systems (2nd ed.), *Society for Industrial and Applied Mathematics*, Philadelphia, PA, USA, 2003.

[16] Bell N., Garland M. Efficient sparse matrix-vector multiplication on CUDA, Bericht, *NVidia Corporation*, Nr. NVR-2008-004, 2008.

[17] Bell N., Garland M. Implementing sparse matrix-vector multiplication on throughput-oriented processors, *Conference on High Performance Computing Networking, Storage and Analysis* (*SC'09*), Portland, Oregon, 14-20 November 2009, pp. 1–11.

[18] Cheik Ahamed A. K., Magoules F. A stochastic-based optimized Schwarz method for the gravimetry equations on GPU clusters. *21st International Conference on Domain Decomposition Methods*, Rennes, France, June 25-29, 2012, Vol. 98, pp. 577-584, *Lecture Notes in Computational Science and Engineering (LNCSE)*, Springer-Verlag, 2014.

[19] Cheik Ahamed A. K., Magoules F. Schwarz method with two-sided transmission conditions for the gravity equations on graphics processing unit, *12th International Symposium on Distributed Computing and Applications to Business, Engineering Science* (*DCABES*), Kingston upon Thames, Surrey, UK, 2-4 September 2013, pp. 105–109.

[20] Guo, P., Wang, L. Auto-tuning CUDA parameters for sparse matrix-vector multiplication on GPUs, *International Conference on Computational and Information Sciences (ICCIS)'*, Chengdu, 17-19 December 2010, pp. 1154–1157.

[21] Cao W., Yao L., Li Z., Wang Y., Wang Z. Implementing sparse matrix-vector multiplication using CUDA based on a hybrid sparse matrix format, *International Conference on Computer Application and System Modeling (ICCASM 10')*, Taiyuan, 22-24 October 2010, pp. V11–161-V11-165.

[22] Wang T., Yao Y., Han L., Zhang D., Zhang Y. Implementation of Jacobi iterative method on graphics processor unit, *IEEE International Conference on Intelligent Computing and Intelligent Systems (*ICIS 09'), Shanghai, 20-22 November 2009, pp. 324–327.

[23] Zhang Z., Miao Q., Wang Y. CUDA-based Jacobi's iterative method, *International Forum on Computer Science-Technology and Applications* (*IFCSTA 09'*), Chongqing, 25-27 December 2009, pp. 259–262.

[24] Magoules F., Diago L. A., Hagivara I. Efficient preconditioning for image reconstruction with radial basis function, *Advances in Engineering Software*, Vol. 38, No. 5, 2007, pp. 320–327.

[25] Magoules F., Gbikpi-Benissan G. Coarse space construction based on Chebyshev polynomials for graphic analysis, *Pollack Periodica*, 2014, In Press

[26] Magoules F., Diago L. A., Hagiwara I. A two-level iterative method for image reconstruction with radial basis functions. *JSME International Journal*, Vol. 48, No. 2, 2005, pp. 149–159.

[27] Magoules F., Meerbergen K., Coyette J. P. Application of a domain decomposition method with Lagrange multipliers to acoustic problems arising from the automotive industry, *Journal of Computational Acoustics*, Vol. 8, No. 3, 2000, pp. 503–521.

[28] Harari I., Magoules F. Numerical investigations of stabilized finite element computations for acoustics, *Wave Motion*, Vol. 39, No. 4, 2004, pp. 339–349.

[29] Autrique J. C., Magoules F. Analysis of a conjugated infinite element method for acoustic scattering, *Computers and Structures*, Vol. 85, No. 9, 2007, pp. 518–525.

[30] Autrique J. C., Magoules F. Studies of an infinite element method for acoustical radiation, *Applied Mathematical Modeling*, Vol. 30, No. 7, 2006, pp. 641–655.

[31] Autrique J. C., Magoules F. Numerical analysis of a coupled finite-infinite element method for exterior Helmholtz problems, *Journal of Computational Acoustics*, Vol. 14, No. 1, 2006, pp. 21–43.

[32] Hassani R., Fazely A., Choudhury R., Luksch P. Analysis of sparse matrix-vector multiplication using iterative method in CUDA, *IEEE 8th International Conference on Networking, Architecture and Storage* (*NAS*), Xi'an, 17-19 July 2013, pp. 262–266.

[33] Cheik Ahamed A. K., Magoules F. Schwarz method with two-sided transmission conditions for the gravity equations on graphics processing unit, *12th IEEE International High Performance Computing and Communication*, Liverpool, UK, 25-27 June 2012, pp. 1307–1314.