

Optimized Schwarz method without overlap for the gravitational potential equation on cluster of graphics processing unit

Frédéric Magoulès, Abal-Kassim Cheik Ahamed & Roman Putanowicz

To cite this article: Frédéric Magoulès, Abal-Kassim Cheik Ahamed & Roman Putanowicz (2015): Optimized Schwarz method without overlap for the gravitational potential equation on cluster of graphics processing unit, International Journal of Computer Mathematics, DOI: [10.1080/00207160.2015.1011628](https://doi.org/10.1080/00207160.2015.1011628)

To link to this article: <http://dx.doi.org/10.1080/00207160.2015.1011628>



Accepted author version posted online: 26 Jan 2015.
Published online: 24 Mar 2015.



Submit your article to this journal [↗](#)



Article views: 37



View related articles [↗](#)



View Crossmark data [↗](#)



Citing articles: 1 View citing articles [↗](#)

Optimized Schwarz method without overlap for the gravitational potential equation on cluster of graphics processing unit

Frédéric Magoulès^{a*}, Abal-Kassim Cheik Ahamed^a and Roman Putanowicz^b

^aEcole Centrale, Paris, France; ^bInstitute for Computational Civil Engineering (L-5), Cracow University of Technology, Cracow, Poland

(Received 27 May 2014; revised version received 26 September 2014; accepted 1 January 2015)

Many engineering and scientific problems need to solve boundary value problems for partial differential equations or systems of them. For most cases, to obtain the solution with desired precision and in acceptable time, the only practical way is to harness the power of parallel processing. In this paper, we present some effective applications of parallel processing based on hybrid CPU/GPU domain decomposition method. Within the family of domain decomposition methods, the so-called optimized Schwarz methods have proven to have good convergence behaviour compared to classical Schwarz methods. The price for this feature is the need to transfer more physical information between subdomain interfaces. For solving large systems of linear algebraic equations resulting from the finite element discretization of the sub-problem for each subdomain, Krylov method is often a good choice. Since the overall efficiency of such methods depends on effective calculation of sparse matrix–vector product, approaches that use graphics processing unit (GPU) instead of central processing unit (CPU) for such task look very promising. In this paper, we discuss effective implementation of algebraic operations for iterative Krylov methods on GPU. In order to ensure good performance for the non-overlapping Schwarz method, we propose to use optimized conditions obtained by a stochastic technique based on the covariance matrix adaptation evolution strategy. The performance, robustness, and accuracy of the proposed approach are demonstrated for the solution of the gravitational potential equation for the data acquired from the geological survey of Chicxulub crater.

Keywords: optimized Schwarz method; domain decomposition method; Krylov methods; GPU; CUDA; gravitational potential equation

1999 AMS Subject Classifications: 49M27; 65F10; 68W10; 97R60; 65Y05

1. Introduction

With constant need to solve larger and larger numerical problems, it is not possible to neglect the opportunity that comes from close adaptation of computational algorithms and their implementations to particular features of computing devices, that is, the characteristics and performance of available workstations and servers. In the recent decade, the advances in hardware manufacturing, decreasing cost and spread of graphics processing unit (GPUs) have attracted the attention of researchers for numerical simulation, as for some problems GPU-based simulations can significantly outperform the ones based on central processing units (CPUs). Although the field of

*Corresponding author. Email: frederic.magoules@hotmail.com

GPU-based computing matures, there are still some issues as communication, memory access patterns and load balancing, that are challenging and require further research. Until the beginning of this century, parallel numerical simulations were most often executed on clusters of CPUs. Emergence of the possibility of General-Purpose computations on GPU has opened the door for several new simulation approaches. This is due to the fact that GPU architecture differs significantly compared to the classical CPU. Because of their primary use for managing graphics-related calculations, GPU has less control units and much more arithmetical and logical unit (ALU). GPU design results in inherently parallel architecture well suited to many parallel simulations. A landmark in GPU-based computing was the appearance in 2006 of the *compute unified device architecture* (CUDA) prepared by NVIDIA. With it, the developers are given a standard tool, in a form of C/C++ compiler extension to build parallel programs to be executed on GPUs, as illustrated in Figure 3(b). On the other hand, open computing language (OpenCL) [33], a free and open language, offered in 2008 by Khronos group, is intended for use on all compatible graphics card of all manufacturers of graphics card market. OpenCL has also proven its efficiency [3,21,22,54,64,66] in numerical simulations. NVIDIA includes OpenCL API in CUDA toolkit since 2008. Today many engineering and scientific applications are transformed in order to utilize the advantages of GPU-based computing.

As an example of such transformation, we modify high-order finite element solver to be run on GPU. Geophysical exploration by seismic imaging can often benefit from complementary analysis of gravitational potential equation by finite element method. Our approach to the analysis of gravitational potential equation is based on partitioning the computational domain into a number of subdomains, and each subdomain is assigned to one processor. The next step is to carry on the iterations of the Schwarz method with optimal conditions obtained by CMA-ES algorithm. The solution of the independent subproblems in each subdomain required for each iteration step of the Schwarz algorithm is done in parallel. Central to our proposal is the application of CUDA/GPU computing [30] for solving local problems in each subdomain.

The outline of the paper is as follows: in Section 2, we shortly give the main idea behind the study of gravity potential equation for geophysical investigation. For the convenience and completeness, Section 3 gives an overview of GPU computing model for readers not familiar with graphics and CUDA programming. The GPU targeted implementation of algebraic operations and iterative Krylov methods is presented in Sections 4 and 5. Section 6 describes the optimized Schwarz method, followed in Section 7 by an overview of the stochastic-based technique to determine the optimized transmission conditions. Section 8 reports numerical results of our Schwarz methods with optimized stochastic conditions. Finally, Section 9 concludes the paper.

2. Gravitational potential equation

Earth's geological processes destroy most of the records of impact craters on Earth's surface; however, there are evidences that also Earth has experienced collisions with other bodies in the solar system. One such evidence is the Chicxulub ancient impact crater, located underneath the town of Chicxulub in Yucatán, at southwest of Mexico on the Yucatán Peninsula. Yucatán is bordered by the states of Campeche to the southwest and Quintana Roo to the east, and the Gulf of Mexico to the north. Geological and other investigations suggest that this impact crater dates back approximately more than 65 million years and was caused by a collision with a meteorite. This event coincides with the mass extinction at the late Cretaceous period, in which the last of the dinosaurs vanished. The Chicxulub impact crater is now widely accepted as the main footprint of the global mass extinction event that marked the Cretaceous/Paleogene (K/Pg) boundary. Because of its relevance, the area of the impact crater was subjected to several geophysical studies including land, marine and aerial surveys.

One of the techniques used to study geological formations of the crater area was gravity method. This method allows to quantify differences in the Earth's gravitational field at specific locations. Detected anomalies of the gravitational field allow to draw conclusion about the morphology and formation processes of geological structures. The gravity method allows to determine the depth, density and geometry of the gravitational anomaly source. For the case of Chicxulub impact crater, the collected survey is relevant to a physical domain covering the area $250 \text{ km} \times 250 \text{ km}$ and reaching 15 km in depth.

The other way for geological exploration is the analysis of seismic waves; however, it requires solution of inverse problem. By its nature, the problem is ill-posed and its solution might not sufficiently describe the subsurface, especially in the case of strong contrasts of seismic waves velocity appearing near the surface. Below a certain depth, it is important to define the structures of the basement that seismic method cannot resolve. Hence, other techniques such as potential method best describe the subsurface. It consists of determining mass density distribution correlated with seismic velocities. The study of gravitational potential equation gives additional information as to the mass density distribution. The gravity force is expressed as the sum of the gravitational force and the centrifugal force. The potential of gravity of a spherical density distribution is $\Phi(r) = Gm/r$, where m and r , respectively, represent the mass of the object and the distance to the object, and G is the universal gravity constant equal to $G = 6.672 \times 10^{-11} \text{ m}^3\text{kg}^{-1}\text{s}^{-2}$. For spatial position x , the gravitational potential is expressed as $\Phi(x) = G \int (\rho(x')/|x - x'|)dx'$ where ρ is an arbitrary density distribution and x' is point position within the considered density. The effect due to the rotation of the Earth, that is, the component of centrifugal force, is neglected. Hence, only the regional scale of the gravity potential equation is considered. The technique that has been used for the direct calculation of gravity anomalies is based on the finite element method approximation of Poisson equation [63]. The gravitational potential of a density anomaly distribution is thus given as the solution of the Poisson equation $\nabla\Phi = -4\pi\delta\rho$ where $\delta\rho$ is the variation of an arbitrary density distribution. This technique does not require defining discontinuities in the discrete formulation of the problem and enables to treat a larger number of discrete points than semi-analytic methods. The discretization of this problem leads to a linear system which can be very large when we consider high-order finite elements. In this paper, we have taken the advantage of the high performance of the graphical card processors to achieve considerable speed-up of our computations.

3. GPU programming paradigm

Putting GPU-based computations in a wider context can be helpful in understanding this technology. This is why in subsequent subsections we briefly discuss its history and evolution. Moreover, in order to master GPU programming one needs to be aware of hardware architecture issues, memory layout, programming language and management of threads of execution.

The GPU unit emerged as a specialized and highly parallel microprocessor designed in order to accelerate 2D or 3D rendering, and to make it independent on the load of the CPU. The appearance of add-on GPUs can be traced back to the middle 1990s, but it is the last 10 years when GPUs have become more and more programmable.

In recent years, we have witnessed that general-purpose graphics processing units (GPGPUs) were adopted as essential computational platform in all scientific fields. For high performance computing environments, the use of GPGPU is almost mandatory. The newest graphics card can achieve performance measured in teraflops, and the use of programming tools with API-based on high-level language such as C is probably one of the reasons for that. While offering considerable computational power, utilization of GPU architectures requires redesign of the most

of algorithms. Implementations and optimizations need to be done carefully, because if done carelessly can degrade the performance significantly. Programmers community and graphics card manufacturers are now working on making the power of GPGPU more accessible, with special programming tools and common standards.

3.1 History and evolution

As said above the first 3D add-in graphics card appeared in 1995. The main driving force for their evolution was demands from the gaming industry. The race to provide more processing power and better video quality to the end users caused the graphics card to reach such level of performance that their capabilities could no longer be ignored by computational science community. Special purpose graphics programming languages and platforms, such as RapidMind, Sh, Cg and Brook were another factor helping to spread GPU computing [56] over divers scientific fields [52,57]. By the beginning of 2000s, GPU units were able to outperform some of the supercomputers from previous decade. This was the time when the idea of GPGPU have gained wide acceptance, see [10,34,59,65]. Introduction by NVIDIA of its parallel computing platform and programming model (CUDA), sets the new direction of making GPU programming more accessible and user friendly for programmers of such high-level languages as C or C++ . That said, GPU programming still presents considerable endeavour, especially from the perspective of portability between devices and proving correctness of the calculations.

3.2 Architecture

Gaining the most from GPU programming paradigm requires understanding of the GPU architectures. In this section, we give a brief overview of them and compare their features to the architectures of CPU. As illustrated by Figure 1, one can note a big difference between the peak performance of CPUs and GPUs, partly due to the inherently different architectures of these processors. This is the reason why increasing number of researcher focuses their attention on GPU-based numerical simulations.

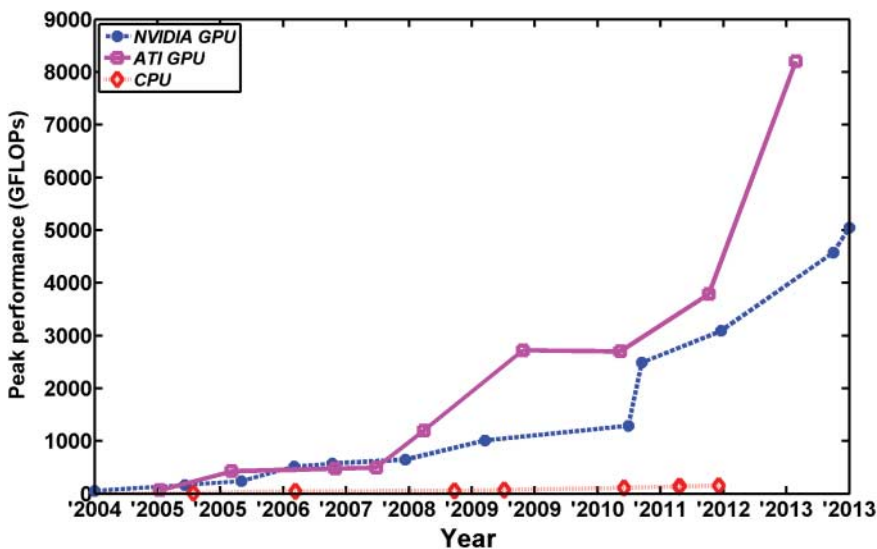


Figure 1. Evolution of the peak performance of GPUs.

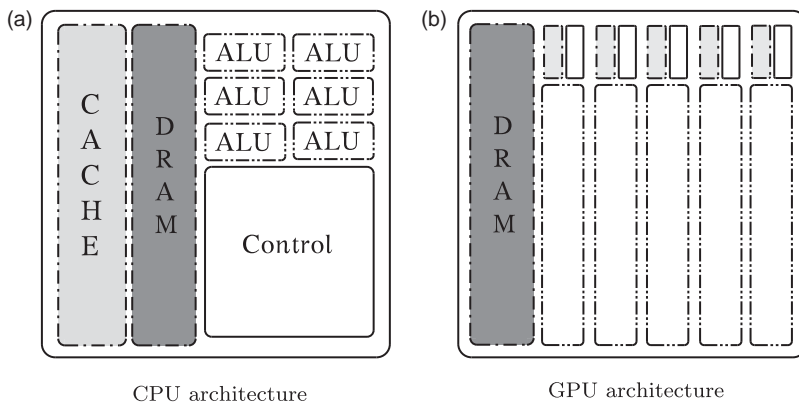


Figure 2. CPU architecture vs GPU architecture.

The simplified CPU architecture contains *ALU* (the basic unit of computations), several memory units with multiple levels of associated cache memory and a complex control unit. Figure 2 shows a simplified comparison between CPU and GPU architecture. On contrary, the main idea behind the architecture of GPU is to have many simpler floating point processors, processing a large amount of data in parallel. The reason for this is that the envisioned usage of GPU is to do the same type calculations on a large amount of independent data (single instruction multiple data model). The way this is achieved is, in general, through a memory hierarchy allowing each processor to access needed data in a somehow optimal manner. GPU has in fact become massively parallel computing device with hundred of cores (ALUs) that operate together to massively parallel process the computing data.

3.3 Overview of CUDA programming and hardware configuration

GPGPU-based programming has undergone substantial evolution over the past few years. The basic programming model of traditional GPGPU is *stream processing*, which is closely related to single instruction multiple data (SIMD).

NVIDIA has proposed CUDA for general-purpose computing on NVIDIA GPUs. CUDA programming is easier than legacy GPGPU, since it does not require knowledge specific to computer graphics. Instead, it is based on learning a couple of extensions to C/C++ language. Figure 3(b) shows the components of the CUDA that accelerate development of codes to be run on GPU.

In CUDA programming, the CPU is classified as a *host* and the GPU as a *device*. CUDA lowers the initial barrier for developers to benefit from NVIDIA GPGPUs technology for solving the most intensive computations for linear algebra operations such as matrix–vector multiplication. Thus, it enables programs to use the computation power of GPU, facilitated by the CUDA-SDK. Despite enormous improvement there are still some CUDA [6] issues that needs to be resolved. GPUs are originally designed for single precision (32 bits) computations [24]. In fact, single precision is generally sufficient for graphics rendering. Unfortunately, the numerical simulation generally requires more precision.

All implementations prior to CUDA 1.3 are performed in single precision, since double precision was introduced for CUDA 1.3. Nevertheless, double precision computation time is still usually 4 to 8 higher than single precision. Regarding the arithmetic, CUDA implementation of double precision floating point operations differ at some points from IEEE 754 standard. More research is needed to ensure the correctness, stability and portability of such operations.

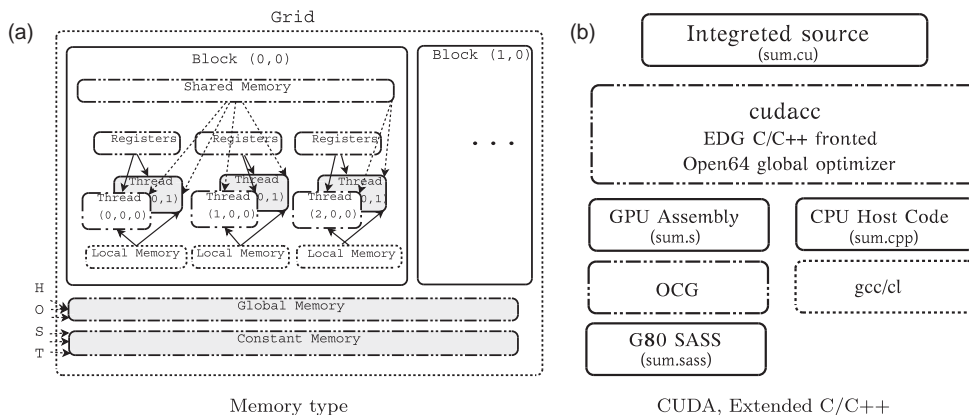


Figure 3. Gridification and CUDA/C++.

3.3.1 Memory

The GPU performs the same calculations on a large number of data instances (SIMD model). This alone, however, is not enough to get more performance, because the data must be well organized on memory to optimize access to it, the key point on which the algorithm performance depends. To compare, a CPU is constantly accessing the random access memory (RAM), with low latency but restricted throughput. On contrary, GPU memories have enable to transfer large amounts of data, but their access remains rather slow. CUDA architectures have a divers set of memories that can be used by developers in order to get high performance. We talk here about four main types of memory as illustrated in Figure 3(a): *global*, *local*, *constant* and *shared*. The slowest memory, *global memory*, is available to all threads. This memory, large in size, is responsible for the interaction with the host (CPU). A thread is the smallest unit of processing that can be scheduled by an operating system. Each thread has a private *local memory*, which is specific to each computing unit and cannot be used to communicate between them. It is much faster than the *global memory*. The *constant memory* is generally cached for fast access and is read-only from the device. It also provides interaction with the host. Computing units are divided into blocks of threads. Each block has its own *shared memory* that is available to block's threads. The key point in achieving good performance in GPU computing is the wise use of this memory hierarchy. It might require redesign of traditional implementation of some basic algorithms. What is also important, the algorithm design must not only account for the architecture but also be tunable to GPU device characteristics.

3.3.2 Gridification

CUDA threads are grouped by *blocks* as shown in Figure 4(a) and the GPU places the blocks over the multiprocessors. The execution of all threads is done simultaneously. On a multiprocessor, blocks are split in *warps*, which is a group of 32 *threads*. At each instruction step, a warp is chosen and its next instruction is executed. The best case means the same execution path for all 32 threads, which is close to the SIMD execution idea. The programmer must pay attention to build blocks with no execution dependencies and that make sure they do not write to the same memory. An *ALU* is associated with the *thread* which is currently processing and a *GPU* is associated with a *grid*, that is all running or waiting blocks in the running queue. This forms a *kernel* that will run on many cores. The configuration of the grid of threads for GPU grid is called *gridification*. Such arrangement of execution threads is not automatic, and it is the responsibility

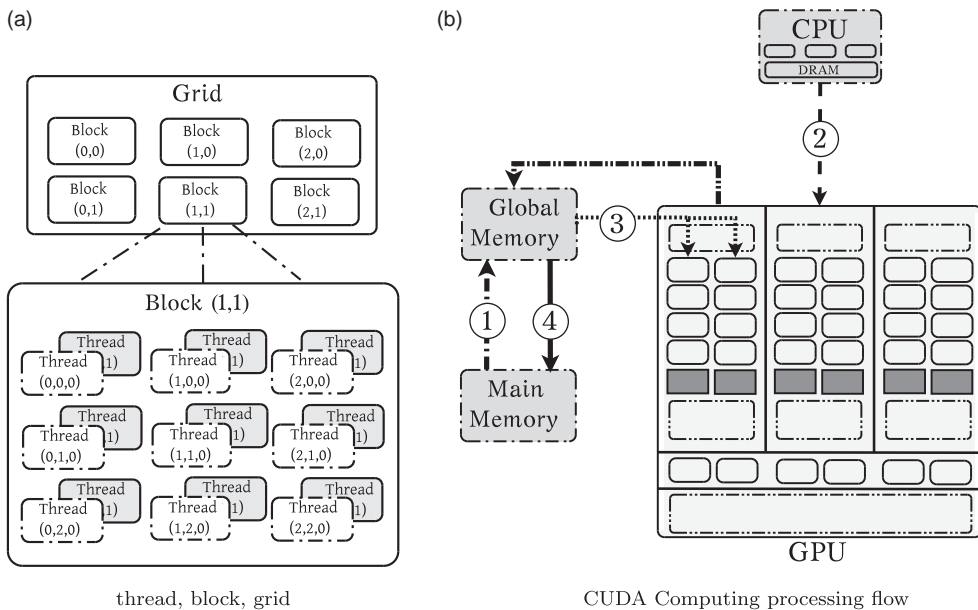


Figure 4. Gridification and GPU computing procedure.

of the programmer to design the threads distribution, that is the *kernel*. The configuration of the GPU grid is done as follows: (i) threads are grouped into blocks, (ii) each block has three dimensions to classify threads and (iii) blocks are grouped together in a grid of two dimensions. The threads are then spread to these levels and become easily identifiable by their positions in the grid according to the block they belong and their indices inside the block. Every thread has two pieces of information: in which block it is running and by which block internal number it is identified. The numbering of the threads for one-dimensional grid can be done as in Listing 1.

```
int idx = blockIdx.x * blockDim.x + threadIdx.x;
// for one dimension (x), where
// threadIdx.x: local index
// blockIdx.x: block index
// blockDim.x: number of threads per block (block size)
```

Listing 1 Thread indexing for one-dimensional grid

For a two-dimensional grid, the numbering of the threads is done as in Listing 2.

```
unsigned int x = blockIdx.x * blockDim.x + threadIdx.x;
unsigned int y = threadIdx.y + blockIdx.y * blockDim.y;
unsigned int pitch = blockDim.x * gridDim.x;
unsigned int idx = x + y * pitch;
// for one dimension (x,y), where
// threadIdx.d: local index
// blockIdx.d: block index
// blockDim.d: number of threads per block (block size)
// where d = x | y
```

Listing 2 Thread indexing for two-dimensional grid

GPU execution flow is drawn in Figure 4(b). First, data are copied from the main memory to the GPU memory (1). Then, the host (CPU) give instructions to the device (GPU) for performing calculations (2). At the end, the device results are copied back from GPU memory to main host memory, (4). Research results presented in [12,17] show clearly that choosing appropriate arrangement of the grid has significant impact on the performances of the kernels.

4. Linear algebra operations

For many scientific and engineering applications, the overall computational performance is the direct derivative of the performance of linear algebraic equations solver, and this in turn can be related to the performance of basic linear algebra operations, like dot product, scaling or matrix–vector product. For sparse vectors and matrices, it is exactly the matrix–vector product that appears the most time consuming operation. The basic indicator for computational complexity of sparse linear algebra operations is the number of non-zero values (nnz) of the matrix. For the Krylov algorithm, the operations that are performed for each iteration step consist of several scaled vector additions, dot products and one or more sparse matrix–vector multiplications.

4.1 Testing environment and performance measurement

In terms of hardware, our testing environment for linear algebra operations and iterative Krylov solver, was a workstation with an Intel Core i7 920 2.67 GHz processor, which has four physical cores but due to the use of Hyper Threading technology also have four logical cores, 5.8 GB RAM memory and two NVIDIA GTX275 GPUs of 895MB memory each. This configuration allows to utilize CUDA 4.0 features. The numerical tests are performed for *double precision* implementation.

Because of the difference in clock frequency for host/CPU at the level of few milliseconds and device/GPU at the level of few nanoseconds, some precautions have to be taken to accurately measure the performance. To average the measurements, we propose to repeat the same operation until the total time measured is greater than 100 times the accuracy of the clock and to perform no less than 10 repetitions.

4.2 Basic linear algebra operations

In this section, we report the main idea of our algorithms and collect the numerical results of the experiments, we have performed to evaluate the speed of our linear algebra operations both on CPU and GPU.

The default gridification ($nBlocks, nThreadsPerBlock$) of ‘Scale’, ‘Addition of Vectors’, ‘Element wise product’ is set to the following values:

$$nBlocks = \frac{numb_rows + numb_th_block - 1}{numb_th_block},$$

$$nThreadsPerBlock = 256,$$

where $numb_rows$ and $numb_th_block$ represent, respectively, the number of rows of the matrix, and the thread block size. For dot product, we consider the following default gridification:

$$nBlocks = \frac{numb_rows + numb_th_block - 1}{numb_th_block},$$

$$nThreadsPerBlock = 128.$$

The cost of transferring data from CPU/GPU to GPU/CPU is not negligible. In all the algorithms presented here, the step before launching the kernel consists to send data from CPU to GPU and then copy back the result from GPU to CPU. In all our programs, we pay attention to minimize data transfers, that is, only necessary transfers are performed.

4.3 Scaling vectors

The ‘Scale’ operation (Dscal) is equivalent to the level one (vector) operation in the Basic Linear Algebra Subprograms (BLAS) package, which scale all elements of a vector by a scalar α . The first extension of this package to GPU has been done in [55]. This operation is done ‘in-place’ that is the result will overwrite elements of the input vector. Consider our CUDA kernel for scale operation:

```
__global__ void Dscal ( double alpha, double* d_x, int size ) {
    unsigned int x = blockIdx.x * blockDim.x + threadIdx.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_x[idx] = alpha * d_x[idx];
    }
}
```

Listing 3 Dscal on two-dimensional grid

In Listing 3, d_x is the vector, $size$ is the length (number of elements in d_x we wish to use), and α is the scaling factor. Table 1 collects the double precision execution time of our implementation for the *Scale* operation. The first column h gives the size of the vector. The second and third columns contain the execution time in milliseconds for CPU and GPU algorithms, respectively. The fourth and fifth columns give the floating point operations per second for CPU and GPU algorithms. The last column shows the ratio of CPU to GPU time. Table 1 clearly shows the better results of GPU compared to CPU. As this operation is inherently parallel, we observed higher speed-up for larger vectors.

4.3.1 Addition of vectors

Another operation that is an excellent candidate for parallel implementation on a GPU is another level one BLAS routine named Daxpy, that is scaled update of a vector: $y[i] = \alpha \times x[i] + y[i]$. This operation is also done in place. Our implementation for two-dimensional computing grid is illustrated in Listing 4.

Table 1. Double precision Scale (Dscal) operation.

h	CPU time (ms)	GPU time (ms)	CPU Gflops	GPU Gflops	CPU/GPU time ratio
49248	0.197	0.045	0.249	1.0	4
89056	0.364	0.058	0.244	1.5	6
101168	0.414	0.059	0.243	1.7	6
181888	0.729	0.066	0.249	2.7	11
270336	1.123	0.076	0.240	3.5	14
296208	1.176	0.083	0.251	3.5	14
416176	1.666	0.104	0.249	3.9	15
450528	1.886	0.096	0.238	4.6	19
544563	2.222	0.108	0.245	5.0	20
606816	2.702	0.109	0.224	5.5	24
650848	2.777	0.121	0.234	5.3	22
848256	3.571	0.149	0.237	5.6	23
1213488	5.555	0.194	0.218	6.2	28
1325848	5.555	0.208	0.238	6.3	26
1587808	6.666	0.232	0.238	6.8	28

Table 2. Double precision addition of vectors (Daxpy).

h	CPU time (ms)	GPU time (ms)	CPU Gflops	GPU Gflops	CPU/GPU time ratio
49248	0.401	0.052	0.367	2.7	7
89056	0.735	0.058	0.363	4.5	12
101168	0.833	0.067	0.364	4.5	12
181888	1.538	0.083	0.354	6.5	18
270336	2.325	0.094	0.348	8.6	24
296208	2.439	0.102	0.364	8.6	23
416176	3.448	0.139	0.362	8.9	24
450528	3.846	0.119	0.351	11.3	32
544563	4.761	0.155	0.343	10.5	30
606816	5.555	0.156	0.327	11.6	35
650848	5.555	0.175	0.351	11.1	31
848256	7.692	0.188	0.330	13.4	40
1213488	10.000	0.283	0.364	12.8	35
1325848	11.111	0.291	0.357	13.6	38
1587808	14.285	0.342	0.333	13.9	41

```

__global__ void Daxpy ( double alpha , double* d_x ,
double* d_y , int size ) {
unsigned int x = blockIdx.x * blockDim.x + threadIdx.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];
}
}

```

Listing 4 Daxpy on two-dimensional grid

The numerical results for our implementation of the *Daxpy* operation in double precision are reported in Table 2. Table 2 confirms expected better results for the GPU implementation compared to the CPU one. As we can see by comparing Tables 1 and 2, the speed-up for Daxpy is better than for Dscal.

4.3.2 Element-wise product

Element-wise product or *scalar-vector one-by-one multiplication* is generally used at preconditioning step, which consists in multiplying two vectors element by element. Again the operation is done in place: $y[i] = y[i] \times x[i]$, and is inherently parallel making it an excellent candidate for implementation on GPU, as shown in our implementation in Listing 5.

```

__global__ void ElementWiseProduct ( double alpha , double* d_x ,
double* d_y , int size ) {
unsigned int x = blockIdx.x * blockDim.x + threadIdx.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] = d_y[idx] * d_x[idx];
}
}

```

Listing 5 Element-wise product on two-dimensional grid

Table 3 presents the double precision execution time of our implementation for the *Element-wise product* operation.

Table 3. Double precision Element wise product.

h	CPU time (ms)	GPU time (ms)	CPU Gflops	GPU Gflops	CPU/GPU time ratio
49248	0.373	0.051	0.131	0.9	7
89056	0.680	0.060	0.130	1.4	11
101168	0.781	0.072	0.129	1.4	10
181888	1.408	0.083	0.129	2.1	16
270336	2.000	0.092	0.135	2.9	21
296208	2.439	0.116	0.121	2.5	20
416176	3.125	0.121	0.133	3.4	25
450528	3.571	0.123	0.126	3.6	28
544563	4.166	0.146	0.130	3.7	28
606816	5.000	0.155	0.121	3.8	32
650848	5.000	0.166	0.130	3.8	29
848256	6.666	0.209	0.127	4.0	31
1213488	9.090	0.294	0.133	4.1	30
1325848	10.000	0.289	0.132	4.5	34
1587808	11.111	0.341	0.142	4.6	32

4.3.3 Dot product and norm

Calculation of dot product for large size vectors, which most Krylov methods require, is rather expensive on CPU in terms of computing time. This is the reason, we prefer to use GPU for this task in order to boost the performance of the whole Krylov method implementation. The basic dot product algorithm, which is a simple loop with simultaneous summation, is not effective on GPU. For better efficiency, the calculation of dot product is done in two separate steps: the first one is element-wise product of two vectors, the second one is summing up the elements of the computed product. This second step is done in hierarchical way by summing product components in neighbouring threads as indicated by the gridification and illustrated in Figure 5. At the final level of the reduction, the partial sum of all elements of all threads of a block is stored into the first thread (thread[0]) of the current block. Finally, the dot product results as a sum of all partial sums of all the blocks. Obviously, the calculation of the euclidean vector norm is done in the similar way. Table 4 gives the comparison of our implementation on both CPU and GPU for the *dot product* double precision execution time. As illustrated in Table 4, GPU implementation outperforms the CPU with a speed-up of 27 times of the CPU, for the largest vector case.

The number of threads harnessed by the shared memory while performing dot product operation depends on the fixed number of threads per block. Obviously, we need to use a given number of threads that will provide good performance when treated in shared memory, and which also ensure optimal performance for the reduction of the global sum between all blocks; which depend on the number of threads per block. As a conclusion, the simulations performed demonstrate that

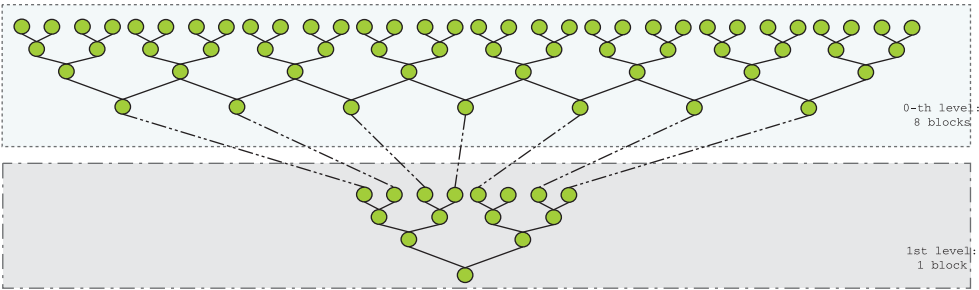


Figure 5. Dot product scheme for tree cutting passes sums.

Table 4. Double precision Dot product.

h	CPU time (ms)	GPU time (ms)	CPU Gflops	GPU Gflops	CPU/GPU time ratio
49248	0.275	0.098	0.357	1.0	2
89056	0.495	0.110	0.359	1.6	4
101168	0.564	0.103	0.358	1.9	5
181888	1.041	0.124	0.349	2.9	8
270336	1.587	0.126	0.340	4.2	12
296208	1.612	0.131	0.367	4.5	12
416176	2.272	0.150	0.366	5.5	15
450528	2.564	0.150	0.351	6.0	17
544563	3.125	0.192	0.348	5.6	16
606816	3.571	0.166	0.339	7.2	21
650848	3.571	0.196	0.364	6.6	18
848256	5.263	0.204	0.322	8.3	25
1213488	7.142	0.312	0.339	7.7	22
1325848	7.692	0.336	0.344	7.8	22
1587808	9.090	0.335	0.349	9.4	27

with 128 threads per block we have a good rate of performance and a compromise between the use of shared memory and the reduction of the global sums.

4.4 Advanced linear algebra operations

In this section, we present advanced linear algebra operations, which are equivalent of level two procedures of the BLAS package. For the start, we characterize the set of matrices used for numerical experiments. One should note that the sizes of matrices correspond to those used in testing basic linear algebra operations. Then, we present the results for the sparse matrix–vector multiplication.

4.4.1 Data sets matrices

Table 5 shows characteristics of matrices from the finite element discretization of the gravitational potential equation based on real data from the Chicxulub in the Yucatán Peninsula in Mexico. The illustration of sparsity structure of a selected matrix is given in Table 6. To characterize the matrices we provide quantity h , the size of the matrix, nz , the number of non-zero elements, *density*, the density that corresponds to the number of nonzero elements divided by the total number of elements, *bandwidth*, the upper bandwidth equals to the lower bandwidth for a symmetric matrix, *max_row*, the maximum row density, nz/h , the mean row density, and $nz/h \text{ stddev}$, the standard deviation of the quantity nz/h . The sizes of the problems are ranged from 89056 up to 1587808 rows for 2 010 320 up to 33 321 792 of non-zeros values. Figures in Table 6 show the sparse matrix pattern and the distribution of the non-zero elements, respectively.

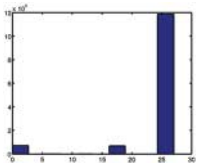
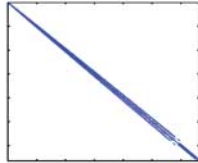
4.4.2 Sparse matrix–vector product

Sparse matrix–vector product (SpMV) is the basic operation not only for Krylov algorithms. It can be found at the core of many scientific numerical programs. In [12,13,44], authors proved that the sparse matrix–vector multiplication is the crucial operation to reach a good performance. Many numerical methods, among them finite element method, involve extremely large size sparse matrices. Crucial to the effective implementation of algebraic operations for such matrices is the matrix storage format. Sparse data structures such as Skyline or compressed-sparse row

Table 5. Sketches of the matrices.

h	nz	density	bandwidth	max_row	nz/h	nz/h stddev
49249	1067632	0.044	3779	27	21.678	9.425
89056	2010320	0.025	5675	27	22.573	8.805
101168	2310912	0.022	6209	27	22.752	8.666
181888	4268192	0.013	9239	27	23.466	8.051
270336	6456960	0.009	12095	27	23.885	7.638
296208	7101472	0.008	12869	27	23.975	7.544
416176	10107680	0.006	16205	27	24.287	7.199
450528	10972752	0.005	17099	27	24.355	7.119
544563	13348267	0.005	19439	27	24.512	6.932
606816	14926000	0.004	20915	27	24.597	6.826
650848	16044032	0.004	21929	27	24.651	6.758
848256	21073920	0.003	26225	27	24.844	6.504
1213488	30434592	0.002	33389	27	25.080	6.171
1325848	33321792	0.002	33389	27	25.132	6.094
1587808	40077872	0.002	40019	27	25.241	5.930

Table 6. Example matrix sparsity pattern for discretized gravitational potential equation.

gravi_1325848	h = 89056 nz = 2010320 density = 0.025 max row = 33389 bandwidth = 27 nz/h = 25.132 nz/h stddev = 6.094		
Finite element discretization of gravitational potential equation on ($3D - mesh$ 250 km \times 250 km \times 15 km)			

(CSR) [8] are also essential with respect to storing effectively the matrices on GPU memory. The papers [12,13] and [17] have shown that computation time of GPU kernels strongly depends on both gridification, i.e. threads distribution upon the grid, and the performance the machine (CPU and GPU). In this work, we use advanced gridification tuning techniques described in [12,13].

Our parallel algorithm for computing the SpMV in CSR format is a modified version of the algorithm introduced in reference [7]. The main idea of this algorithm is to change dynamically the number of threads handled by a warp. In our simulations, we fix three sizes of warps: warp of size 32 threads, a half warp that contains 16 used threads and a half warp with 8 used threads. All threads of a warp, i.e. its 32 threads, executes the same instruction simultaneously for each cycle. In this way, thirty two computations are performed at each cycle where a warp manages a row of the matrix. The algorithm has two levels: first the sum of the values calculated by the threads in an array of shared memory and second a segmented sum is performed. When a thread requires access to shared memory outside the range of cached data, global memory access is needed for these data. The idea of our approach consists to seek for the optimal number of threads to use together within warp upon the pattern of the matrix, that is, the distribution and the density of non-zero values per row. According to the row non-zero values of the $row \times vector$ multiplication, we aim to find the optimal warp size to use. Denote that a $row \times vector$ product can be handled by several warps. We exploit the power of warp and manage a kind of balance on the density of the distribution of the number of non-zero element per row. By the use of CSR format, which ensure contiguous access for column indices, we enhance the efficiency of the algorithm. Nevertheless, due to the sparsity of the matrix, access to the values of the dense vector stays non-contiguous during the sparse matrix–vector product computation, and therefore the algorithm becomes inefficient in terms of computing time. According to the structure of the matrices, i.e. distribution of the non-zero coefficients, the more far memory access is required, the

more algorithm latency is observed. This phenomena also depends on the gridification strategy. Knowing the total number of threads per block and the number of threads used per warp, we compute the total number of blocks for the grid with a given size of problem by the following expression:

```
//N_THREAD: number of threads per block
//WARP: number of threads per warp
//Number of block per grid for SpMV
#define GRID_SpMV(n) ((WARP*n)+N_THREAD-1)/N_THREAD
```

Listing 6 Gridification strategy

The default gridification ($nBlocks, nThreadsPerBlock$) of “sparse matrix–vector product” is given by

$$nBlocks = \frac{numb_rows \times n_th_warp + numb_th_block - 1}{numb_th_block}$$

$$nThreadsPerBlock = 256$$

where $numb_rows$, n_th_warp and $numb_th_block$ represent, respectively, the number of rows of the matrix, the number of threads per warp and the thread block size. The number of threads per warp and number of threads per block are given upon a dynamic tuning of the gridification according to the properties of the graphics card and size of the problem. The idea behind our optimization of our SpMV implementation is based on tuning the performance of warps in order to speed the computation. Table 7 compares the double precision execution time of our CSR SpMV implementation for CPU and GPU. As we can see in Table 7, the speed-ups become more than thirty three in favour for GPU. Numerical experiments clearly illustrate the gains from GPU computations for large size problems and especially for linear algebra operations.

5. Krylov methods

The common feature of the computer methods for boundary value problems is that they require solving system of linear algebraic equations. In the whole arsenal of methods for solving linear systems [11], we distinguish two groups – direct solvers and iterative solvers. The choice

Table 7. Double precision CSR matrix–vector multiplication.

h	CPU time (ms)	GPU time (ms)	CPU Gflops	GPU Gflops	CPU/GPU time ratio
49248	7.142	0.302	0.298	7.0	23
89056	13.750	0.523	0.292	7.6	26
101168	16.666	0.671	0.277	6.8	24
181888	27.500	1.123	0.310	7.5	24
270336	40.000	1.694	0.322	7.6	23
296208	46.666	1.923	0.304	7.3	24
416176	65.000	2.500	0.311	8.0	26
450528	70.000	2.857	0.313	7.6	24
544563	85.000	3.333	0.314	8.0	25
606816	100.000	3.571	0.298	8.3	27
650848	100.000	4.000	0.320	8.0	25
848256	130.000	4.166	0.324	10.1	31
1213488	200.000	5.882	0.304	10.3	33
1325848	220.000	7.142	0.302	9.3	30
1587808	260.000	7.692	0.308	10.4	33

of concrete algebraic solver is governed by several factors, but the two main ones are the size of the system and the parallelization issues. Direct solvers are tuned almost to perfection but above certain size of the linear system they are inferior to iterative solvers due to the available memory constraints. Also iterative solvers are more amenable for parallel implementation. Among many available iterative methods, preconditioned Krylov methods [2,36] are the ones used most often, especially for very large systems. Effective implementation of Krylov solves depends mostly on effective sparse matrix–vector multiplication. Having investigated the different strategies to optimize linear algebra operations, in this section we propose to explore and compare different iterative Krylov methods, which require these operations. We focus on various preconditioned Krylov methods: Conjugate Gradient (P-CG) for symmetric positive-definite matrices, Bi-Conjugate Gradient Conjugate Residual (P-BiCGCR) and Stabilized BiConjugate Gradient (BiCGStab) for the solution of sparse linear systems with non-symmetric matrices. We propose numerical experiments for the matrices resulting from application of the finite element method to the gravitational potential equation presented in Section 2. In subsequent subsections, we give an overview of mathematical aspects of Krylov subspace methods, then we investigate their performance referring to the collected numerical results.

5.1 Mathematical overview of Krylov subspace methods

Consider the following linear system with an initial guess x_0 :

$$Ax = b. \quad (1)$$

A general *projection method* looks for an approximate solution x_i from an affine subspace $x_0 + K_i$ of dimension i with Petrov–Galerkin condition expressed as follows:

$$b - Ax_i \perp \mathcal{L}_i \quad (2)$$

where \mathcal{L}_i is another subspace of dimension i . In the Krylov subspace method, the subspace $K_i(A, r_0)$ is generated from r_0 by A :

$$K_i(A, r_0) = \text{span} \{r_0, Ar_0, A^2r_0, \dots, A^{i-1}r_0\}. \quad (3)$$

where $r_0 = b - Ax_0$.

Instances of the Krylov method differ in the choice of \mathcal{L}_i and K_i .

5.2 Numerical aspects of Krylov methods

Iterative Krylov methods are very sensitive to matrix conditioning. In order to ensure rapid convergence efficient preconditioning techniques must be used, such as domain decomposition methods [46,58,62], hierarchical preconditioner, ILU factorization [1,31]. Particular attention will be paid to domain decomposition methods in the next section. Relying on the work of [5,36,51,69], we will discuss the way to implement iterative Krylov methods effectively on GPU device. Exchanging data between CPU/host and GPU/device is generally expensive and should be considered carefully. In our Krylov algorithms, all input data are sent once from CPU to GPU device before starting the iterative routine. Nevertheless, on each iteration of iterative Krylov algorithm the computation of dot product and norm involves communication (copy-transfer) from device to host. The solution is to copy back outside the main loop. The proposed iterative Krylov algorithms consists in offloading all linear algebra computations on the device, when the main processes are still orchestrated by the host. The CPU and GPU codes are similar [15], except

that for GPU code the algebra operations are performed by the graphics card. A residual tolerance threshold 1×10^{-6} and an initial guess equal to zero are considered for numerical experiments of iterative Krylov methods. The diagonal preconditioned conjugate gradient method (PCG) is considered for solving subproblems and the coefficients of the submatrices are stored in CSR format. Designing the optimal preconditioner is research problem in itself and is outside the scope of this paper. Therefore, we choose for experiments to use a diagonal preconditioner, which provides a pretty good preconditioning, whether the matrices are not too ill-conditioned.

Tables 8–10 report the execution times in seconds for the preconditioned (by the diagonal) conjugate gradient, bi-conjugate gradient stabilized, and generalized conjugate residual for CSR format, respectively. The CPU/GPU speed-ups obtained for the three Krylov solvers reach 26, 30 and 27, respectively. The numerical results clearly confirm the robustness, effectiveness and performance of GPU for double precision computation for matrices arising from gravitational potential equation.

Previously, we investigate the best way to carry out linear algebra operations, and their use together within iterative linear algebra operations. We sought for tuning both hardware parameters and problem features in order to get optimal performance of GPU Computing. The experiments clearly show the interest of GPU Computing compared to CPU, and demonstrate

Table 8. CSR double precision, Conjugate Gradient (CG).

h	#iter	cpu time (s)	gpu time (s)	ratio
49248	155	1.450	0.130	11
89056	194	3.200	0.241	13
101168	204	3.850	0.279	13
181888	252	8.720	0.483	18
270336	290	15.130	0.747	20
296208	299	17.130	0.817	20
416176	336	27.360	1.211	22
450528	351	30.990	1.366	22
544563	376	40.690	1.716	23
606816	394	47.240	1.949	24
650848	403	51.970	2.139	24
848256	450	76.030	3.043	24
1213488	516	125.700	4.869	25
1325848	591	157.520	6.030	26
1587808	727	232.980	8.814	26

Table 9. CSR double precision, (BiCGStab).

h	#iter	cpu time (s)	gpu time (s)	ratio
49248	98	1.760	0.163	10
89056	133	4.390	0.283	15
101168	128	4.830	0.296	16
181888	163	11.290	0.565	19
270336	186	19.420	0.872	22
296208	195	22.380	0.990	22
416176	211	34.350	1.425	24
450528	215	37.980	1.550	24
544563	223	47.880	1.926	24
606816	271	65.090	2.572	25
650848	257	66.230	2.601	25
848256	305	105.390	3.951	26
1213488	325	180.390	5.944	30
1325848	382	203.780	7.591	26
1587808	465	297.970	11.013	27

Table 10. CSR double precision, BiConjugate Gradient Conjugate Residual (P-BICGCR).

h	#iter	cpu time (s)	gpu time (s)	ratio
49248	143	1.400	0.151	9
89056	179	3.270	0.235	13
101168	187	3.910	0.259	15
181888	231	8.810	0.462	19
270336	264	15.180	0.699	21
296208	273	17.230	0.787	21
416176	307	27.480	1.153	23
450528	319	30.970	1.288	24
544563	342	40.700	1.619	25
606816	357	47.030	1.860	25
650848	365	51.590	2.025	25
848256	409	75.760	2.889	26
1213488	464	123.760	4.580	27
1325848	529	154.440	5.669	27
1587808	650	227.590	8.260	27

that the tuning of the parameters such as gridification and the type of memory used, help GPU to improve the performance of initial algorithms. The experiments also exhibit that the tuning of the parameters depends on the objective of the algorithm, in fact the dot product features will differ to those of SpMV.

6. Optimized Schwarz method

In order to solve gravity problems arising from finite element analysis, domain decomposition methods [14,18,27] are considered. As said in the previous section, preconditioning techniques such as domain decomposition methods guarantee fast convergence. Domain decomposition methods [35,46,58,62,67] rely on splitting the global domain into several subdomains, and in solving independently in parallel the subproblems formulated for each subdomain. In order to set up boundary conditions for the subproblems, the neighbouring subsystems must exchange information about the solution along the interfaces. The performance of algorithms strongly depends on the interface conditions [41], which can be adjusted either with a discrete approach [26,47,50,60] or a continuous approach [16,19,25,29,40,42,43,45]. Similar approaches like the Aitken–Schwarz methods have shown a strong efficiency too, as described in references [23,28]. These transmission conditions [41] between adjacent subdomains must be carefully defined to assure the convergence of the solution. The solution of independent subproblems in parallel is required at each iteration. The interface problem is often solved with an iterative method, while the subproblems can be solved either by direct method or iterative one. In the presented case, we chose the iterative Krylov methods to resolve the subproblems.

The origins of classical Schwarz algorithms are dated more than 100 years ago [61]. At that time Schwarz has proposed a method, later called his name, to demonstrate the existence and uniqueness of solutions to Laplace equation on irregular domains. At the core of his method is the decomposition of non-regular domain into regular subdomains with some overlap. The solution for the whole domain was obtained by special iterative scheme, special in the sense that each iteration step requires solving problem only for the regular subdomain. The suitably selected boundary conditions are called transmission conditions, as they propagate solution in one subdomain onto the boundary of the overlapping subdomain. The transmission conditions between subdomains are crucial to obtain satisfactory speed of convergence of Schwarz domain

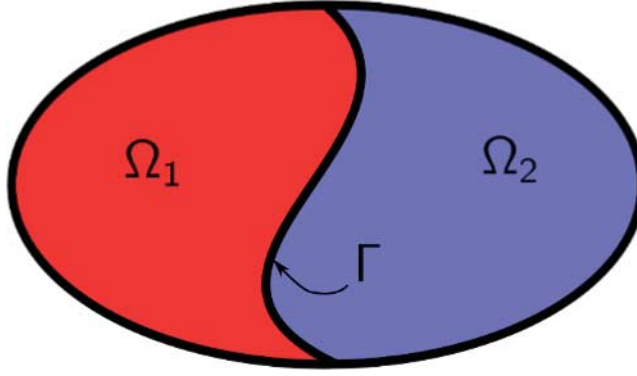


Figure 6. Non-overlapping domain decomposition consists of $\Omega = \Omega_1 \cup \Omega_2$.

decomposition methods [68]. The rate of convergence of the classical Schwarz method depends on both the size of the overlapping layers and the spatial frequency k of the Fourier transform of the solution.

In the case of Schwarz domain decomposition methods without overlap between subdomains [37–39], the algorithm can be formulated by changing the transmission conditions from Dirichlet to Robin [19,20]. The absorbing boundary transmission conditions located on the interface between the non-overlapping subdomains are the crucial factors to ensure a rapid convergence of the iterative Schwarz algorithm.

The difficulty in parallel implementation of an algorithm that produces the *optimal* interface conditions is related to the non-local properties of the operators by which the problem is formulated. One of the approaches to deal with this problem is based on approximating the associated discrete non-local operators with algebraic ones [26,47–50,60] or on approximating these continuous non-local operators with partial differential operators [16,25,53]. In this paper we study an approximation based on a new stochastic optimization process.

Without loss of generality and for the sake of clarity, the gravitational potential equation is defined in the domain Ω with homogeneous Dirichlet conditions. We consider this domain partitioned into two non-overlapping subdomains Ω_1 and Ω_2 with an interface Γ as shown in Figure 6. We want to solve the *gravitational potential equation*, a particular case of *Poisson equation*:

$$\begin{aligned} -\Delta \Phi &= 4\pi G \delta \rho = f \quad \text{on } \Omega \\ \mathcal{C}(\Phi) &= b \quad \text{on } \partial \Omega \end{aligned}$$

with $\delta \rho$ the density anomaly and G the gravitational constant, equal to $6.67428 \times 10^{-11} \text{ m}^3 \text{ kg}^{-1} \text{ s}^{-2}$. Let (Φ_1^n, Φ_2^n) be an approximation to $(\Phi|_{\Omega_1}, \Phi|_{\Omega_2})$ at iteration n of the Schwarz algorithm, $(\Phi_1^{n+1}, \Phi_2^{n+1})$ is expressed as

$$\begin{aligned} -\Delta \Phi_1^{n+1} &= f \quad \text{in } \Omega_1, \\ (\partial_\nu \Phi_1^{n+1} + \mathcal{A}^{(1)} \Phi_1^{n+1}) &= (\partial_\nu \Phi_2^n + \mathcal{A}^{(1)} \Phi_2^n) \quad \text{on } \Gamma, \\ -\Delta \Phi_2^{n+1} &= f \quad \text{in } \Omega_2, \\ (\partial_\nu \Phi_2^{n+1} - \mathcal{A}^{(2)} \Phi_2^{n+1}) &= (\partial_\nu \Phi_1^n - \mathcal{A}^{(2)} \Phi_1^n) \quad \text{on } \Gamma, \end{aligned}$$

where ν is the unit normal vector along Γ . To achieve the best performance of the algorithm the operators $\mathcal{A}^{(1)}$ and $\mathcal{A}^{(2)}$ have to be suitably selected. Application of Fourier transform in

$\Omega = \mathbb{R}^2$ for the homogeneous problem $f = 0$, leads to the expression of the Fourier convergence rate that depends on the quantities $\Lambda^{(1)}$ and $\Lambda^{(2)}$, which are the Fourier transforms of operators $\mathcal{A}^{(1)}$ and $\mathcal{A}^{(2)}$. Over the years researches have proposed several ways to approximate these non-local operators with partial differential operators. One of such methods relies on looking for a partial differential operators enforcing a tangential derivative on the interface such as: $\mathcal{A}^{(s)} := p^{(s)} + q^{(s)} \partial_{\tau_2}^2$, where s denotes the subdomain number, $p^{(s)}$, $q^{(s)}$ are two coefficients, and τ is the unit tangent vector. From references [42,43,45], one can clearly see that both the coefficients $p^{(s)}$, $q^{(s)}$ of the subdomain s participate largely to speed-up the convergence when they are chosen optimally. The results shown in [9,19] use a zero order Taylor expansion of the non-local operators to find $p^{(s)}$ and $q^{(s)}$. A minimization procedure is considered in [25] for the Helmholtz equation, in [16] for Maxwell equation, in [32] for convection diffusion equations, and in [40] for heterogeneous media. The minimization function or cost function is defined as the maximum of the Fourier convergence rate for the considered frequency ranges. This approach consists in calculating the free parameters $p^{(s)}$ and $q^{(s)}$ through an optimization problem. The functions to minimize for zeroth- and second-order approximation for the one-sided formulation are, respectively formulated as follows:

$$\max_{k_{\min} < k < k_{\max}} \frac{(|k| - p)^2}{(|k| + p)^2} e^{-2|k|L}, \quad \max_{k_{\min} < k < k_{\max}} \frac{(|k| - p - qk^2)^2}{(|k| + p + qk^2)^2} e^{-2|k|L}, \quad (4)$$

where $p > 0$ and $q > 0$ and L is the size of the overlap. For the two-sided formulation:

$$\max_{k_{\min} < k < k_{\max}} \frac{(-|k| + p_1)(-|k| + p_2)}{(|k| + p_1)(|k| + p_2)} e^{-2|k|L}, \quad \max_{k_{\min} < k < k_{\max}} \frac{(|k| - p_1 - q_1 k^2)(|k| - p_2 - q_2 k^2)}{(|k| + p_1 + q_1 k^2)(|k| + p_2 + q_2 k^2)} e^{-2|k|L}, \quad (5)$$

where $p_1 > 0$, $q_1 > 0$, $p_2 > 0$ and $q_2 > 0$ and L is the overlap size. Since the evaluation of the minimization function is enough quick and the dimension of the search space are reasonable, a more effective and solid minimization method could be envisaged as introduced in the next section.

7. Stochastic-based optimization

We consider the covariance matrix adaptation evolution strategy (CMA-ES) in order to obtain good results rapidly. The CMA-ES algorithm investigates the whole space of solutions and selects the absolute minima. This recent algorithm clearly demonstrates its robustness in [4] with good global search ability. Its advantage is that it does not need the calculation of the derivatives of the cost function. The main idea behind the CMA-ES strategy consists in finding the minimum of the cost function by refining a search distribution iteratively. The distribution is expressed as a general multivariate normal distribution $d(m, C)$. The initial distribution is taken to be the uniform one. The population size parameter reflects the trade-off between algorithm speed and its ability to find the global minimum. The algorithm is faster for smaller populations but we have a greater chance of finding a local minimum. When the size of the populations is large, the algorithm is able to avoid local minima but it needs larger number of cost function evaluations. In this paper, a population size of 26 has been enough to find the global minimum in few seconds (< 4 sec). Knowing the distribution, λ samples are randomly chosen in this distribution at each iteration and the evaluation of the cost function at those points is performed in order to compute a new distribution. When the variance of the distribution is small enough, the centre of the distribution m is considered to be the sought solution. After analysing the cost function for the new population, the samples are sorted and we select the best sample μ . Then, we recombine

them to find a new distribution centre by considering a weighted mean. The step size σ is an ingredient which serves to scale the standard deviation of the distribution. The variance of the search distribution is proportional to the square of the step size. The most complex task of the algorithm is to update the covariance matrix. While this could be done using only the current population, it would be unreliable especially with a small population size; thus, the population of the previous iteration should also be taken into account. According to the theory of CMA-ES, we have to fix a size of population that is a trade-off between speed and global search of the minimum. In our case, a population size of 25 has been enough to find the global minimum in a few seconds or less. The maximum number of iterations, depending on the population size, is computed as follows: $1000 \times (N + 5)^2 / \sqrt{p}$, where N and p present, respectively, the problem dimension and the population size. In this paper, with $N = 1$ and $p = 25$, the following stopping criteria for the CMA-ES algorithm is considered: a maximum number of iterations equals to 7200 and a residual threshold equals to 5×10^{-11} .

Figure 7 collects the isolines of the convergence rate in the Fourier space of the Schwarz algorithm for the symmetric zeroth order (top-left), non-symmetric zeroth order (bottom-left), the symmetric second order (top-right) and non-symmetric second order (bottom-right) arising from CMA-ES strategy. Figure 8 gives in the same position as Figure 7 the convergence rate of the Schwarz algorithm in the Fourier space. Table 11 reports the obtained exact values of the coefficients obtained by CMA-ES minimization procedure. These values are used to define the local operators previously mentioned in order to optimize our Schwarz domain decomposition method.

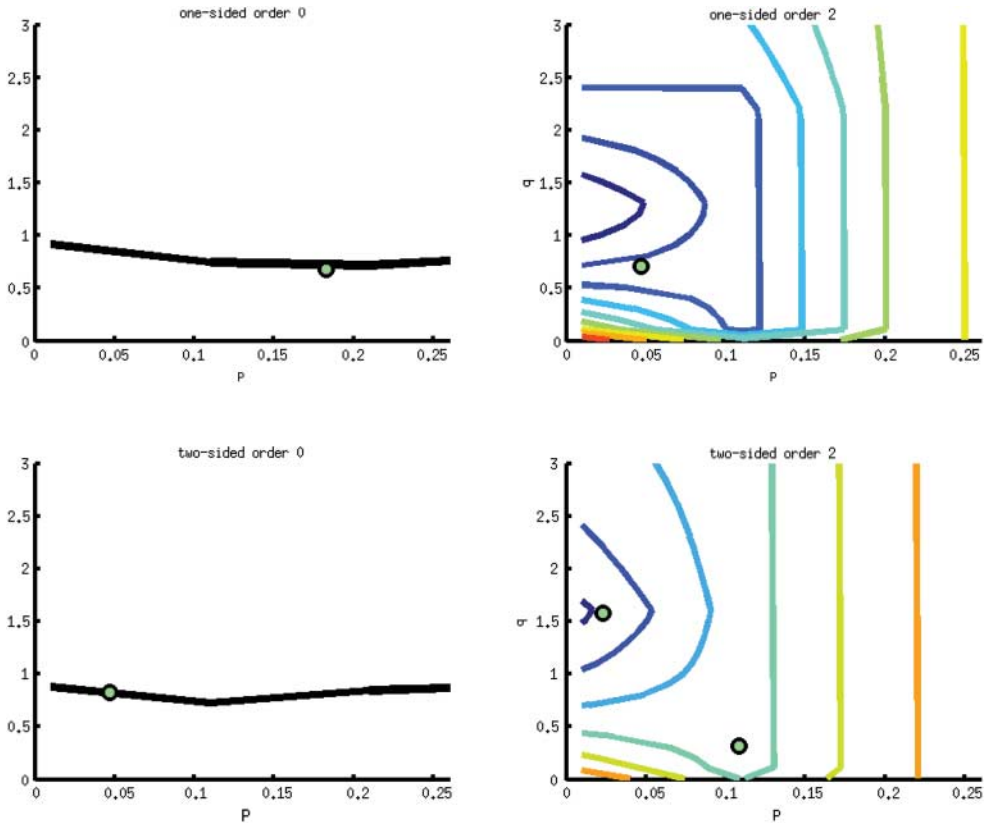


Figure 7. Convergence rate isolines of the Schwarz algorithm with optimized transmission conditions.

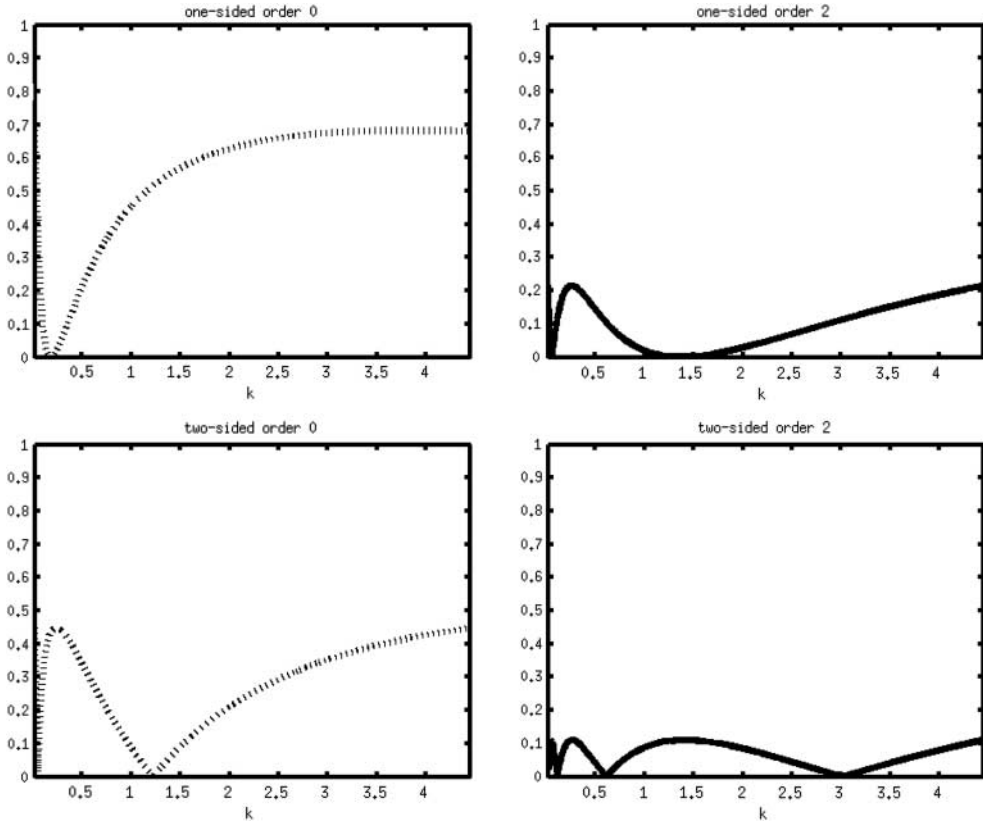


Figure 8. Fourier convergence rate of the Schwarz algorithm with optimized transmission conditions.

Table 11. Optimized coefficients obtained from *CMA-ES* algorithm.

	$p^{(1)}$	$q^{(1)}$	$p^{(2)}$	$q^{(2)}$	ρ_{\max}
oo0_symmetric	0.185	0.000	0.183	0.000	0.682
oo0_unsymmetric	1.219	0.000	0.047	0.000	0.446
oo2_symmetric	0.047	0.705	0.047	0.705	0.214
oo2_unsymmetric	0.095	1.386	0.024	0.329	0.110

8. Numerical experiments

In this section, we summarize the results of experiments performed to evaluate the speed-up of our optimized Schwarz implementation with the optimized coefficients obtained from *CMA-ES* algorithm. All test cases are related to the study of the Chicxulub impact crater presented in Section 2. The solution of the gravitational potential equation $\nabla\Phi = -4\pi\delta\rho$ where $\delta\rho$ is the variation of an arbitrary density distribution is evaluated with a finite element method. Q_1 Lagrange finite elements are considered for the discretization. The numerical solution of the gravitational potential equation required for this study was done for parallelepiped geometric domain of dimensions $250\text{ km} \times 250\text{ km} \times 15\text{ km}$. The finite element discretization of this domain shown in Figure 9 resulted with a total of 19,933,056 degrees of freedom. This mesh is split in the x -direction. Each subdomain is assigned to one single processor and each iteration of the optimized Schwarz method implying the solution of the equations inside each subdomain is

allocated to one single accelerator (GPU). Figure 10(a) and (b) shows the perspective view of the main structural feature and the gravity measure of the Chicxulub impact crater, respectively, obtained from our simulation. The workstation used for all the experiments of domain decomposition methods consists of 1596 servers Bull Novascale R422 Intel Nehalem-based node, where each node is composed of 2 processors Intel Xeon 5570 quad-cores (2.93 GHz) and 24 GB of memory (3Go per cores). Ninety-six CPU servers are interconnected with 48 compute Tesla S1070 servers NVIDIA (4 Tesla cards with 4GB of memory by server) and 960 processing units are available for each server. Each GPU has 4 GPUs of 240 cores.

For each iteration step GPU is involved only in solving the subproblems inside subdomains. The algebraic solver selected for this task is the preconditioned conjugate gradient (PCG) with a diagonal preconditioner. We fix a residual tolerance threshold of $\epsilon_{\text{PCG}} = 10^{-10}$ for PCG for subdomains solution and a residual tolerance threshold of $\epsilon_{\text{Schwarz}} = 10^{-6}$ for Schwarz algorithm. The distribution of processors on our experiment workstation is computed as follows: number of processors = $2 \times$ number of nodes, where 2 corresponds to the number of GPU per node as available on our workstation. As a consequence, only two processors will share the bandwidth, which strongly enhance the communication, especially the inter-subdomain communications. The number of computing units both depends on the size of the subdomain problem and the gridification that use 256 threads per block and 8 threads per warp as introduced in Section 4. Table 12 presents the numerical results obtained for double precision arithmetic

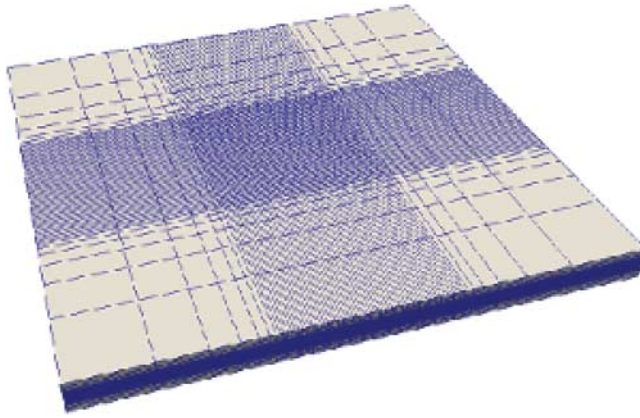


Figure 9. Mesh of study case: volume of $250 \text{ km} \times 250 \text{ km} \times 15 \text{ km}$.

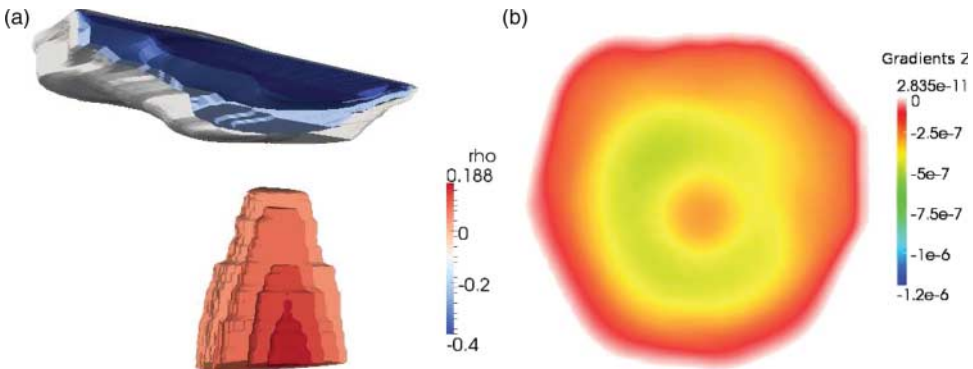


Figure 10. Perspective view of the main structural features and gravity anomaly map of the Chicxulub impact crater: (a) Perspective view of the main structural features. (b) Gravity anomaly map over the Chicxulub crater (Yucatán basin).

Table 12. Comparison of the stochastic-based optimized Schwarz domain decomposition method on CPU and GPU.

#subdomains	#iter	CPU time (sec)	GPU time (sec)	Speed-up
32	41	11240	1600	7.03
64	45	5360	860	6.23
128	92	6535	960	6.81

with a residual threshold, that is the stopping criterion, equal to 10^{-6} , for various number of subdomains. Each subdomain is handled by one classical processor (CPU) and one graphic processor (GPU). Increasing the number of subdomains without increasing the number of degrees of freedom of the submeshes results in increasing the number of communications between the neighbouring subdomains, and this handicaps the GPU version. Each PCG involves a copy from host to device for the right-hand side and a copy back from device to host to update the sub-domain solutions. However, the GPU-based implementation remains clearly superior than pure CPU implementation.

9. Conclusion

In this paper, we presented an efficient double precision implementation of iterative Krylov methods for GPU, that leads to effective implementation of the optimized Schwarz domain decomposition method. The main objective of the presented research was to find the best way to implement linear algebra operations and iterative Krylov solvers using an NVIDIA GPU graphics card and the CUDA/C++ programming language. With our implementation of the vector scaling, vector addition, element-wise product, dot product and matrix–vector product for CSR format we were able to accelerate the computations about the factors 28, 41, 34, 27, 33, when comparing GPU-based implementations with the CPU ones. These algorithmic building blocks were used in implementation of the following solvers: preconditioned (by the diagonal) conjugate gradient, bi-conjugate gradient stabilized and generalized conjugate residual for CSR format. The calculations speed-up measured on matrices arising from application of finite element method to the gravitational potential equation were, respectively, 26, 30 and 27, for the above algebraic solvers. Dynamic tuning of the gridification upon the properties of the GPU architecture is performed in order to obtain faster execution.

In order to test the performance of implemented solvers on real data, we have solved the gravitational potential equation for the input data collected from geological surveys of the Chicxulub crater in Yucatán, at southwest of Mexico. The approach taken to solve that equation is the optimized Schwarz domain decomposition method. Numerical optimization by stochastic-based algorithm (CMA-ES) is used to find the optimal coefficients of the approximate transmission conditions. This algorithm is robust and has good global search property. It is faster and more precise than a brute force approach.

In our experiments, the number of (mesh) subdomains ranged from 32 up to 128. Those experiments have clearly showed benefits of application of the GPU-based implementation of the domain decomposition method for large size problems and confirmed the robustness, performance and effectiveness of the Schwarz method with stochastic-based optimized transmission conditions. With carefully tuned CPU/GPU code and sufficient balance between the number of subdomains and the size of the subproblems, we were able to calculate the solution 7 times faster, with respect to calculation times for equivalent, only CPU based, reference implementation.

Acknowledgements

The authors are grateful to the CUDA Research Center at Ecole Centrale Paris for the computer facilities and computer time used during this long-term research.

Disclosure statement

No potential conflict of interest was reported by the authors.

Funding

The authors acknowledge partial financial support from the OpenGPU project (2012-2014), and CRESTA (Collaborative Research into Exascale Systemware, Tools and Applications).

References

- [1] J.I. Aliaga, M. Bollhofer, A.F. Martien, and E.S. Quintana-Orti, *Parallelization of multilevel ILU preconditioners on distributed-memory multiprocessors*, Proceedings of the 10th International Conference PARA, Reykjavik, Iceland, June 6–9, Vol. 7133, Revised Selected Papers, Part I, Lecture Notes in Computer Science, Springer, Berlin and Heidelberg, 2010, pp. 162–172.
- [2] H. Anzt, V. Heuveline, and B. Rucker, *Mixed precision iterative refinement methods for linear systems: Convergence analysis based on Krylov subspace methods*, Proceedings of the 10th International Conference PARA, Reykjavik, Iceland, June 6–9, Vol. 7134, Revised Selected Papers, Part II, Lecture Notes in Computer Science, Springer, Berlin and Heidelberg, 2010, pp. 237–247.
- [3] J.P. Arun, M. Mishra, and S.V. Subramaniam, *Parallel implementation of MOPSO on GPU using Open CL and CUDA*, Proceedings of the 2011 18th International Conference on High Performance Computing, Washington, DC, USA, 2011, pp. 1–10.
- [4] A. Auger and N. Hansen, *Tutorial CMA-ES: Evolution strategies and covariance matrix adaptation*, Proceedings of the 14th Annual Conference Companion on Genetic and Evolutionary Computation (GECCO'12), New York, NY, USA, 2012, pp. 827–848.
- [5] J.M. Bahi, R. Couturier, and L.Z. Khodja, *Parallel GMRES implementation for solving sparse linear systems on GPU clusters*, in Proceedings of the 19th High Performance Computing Symposia, Boston, MA, Society for Computer Simulation International, San Diego, CA, 2011, pp. 12–19.
- [6] A. Bakhoda, G. Yuan, W. Fung, H. Wong, and T. Aamodt, *Analyzing CUDA workloads using a detailed GPU simulator*, IEEE International symposium on performance analysis of systems and software, Boston, MA, USA, April 26–28, 2009, pp. 163–174.
- [7] N. Bell and M. Garland, *Efficient sparse matrix–vector multiplication on CUDA*, Nvidia Technical Report NVR-2008-004, Nvidia Corporation, 2008. Available at http://www.nvidia.com/object/nvidia_research_pub_001.html (Accessed March 7, 2015).
- [8] N. Bell and M. Garland, *Implementing sparse matrix–vector multiplication on throughput-oriented processors*, Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis (SC'09), Portland, OR, ACM, New York, 2009, pp. 1–11.
- [9] J.-D. Benamou and B. Després, *A domain decomposition method for the Helmholtz equation and related optimal control problems*, J. Comput. Phys. 136 (1997), pp. 68–82.
- [10] J. Bolz, I. Farmer, E. Grinspun, and P. Schröder, *Sparse Matrix Solvers on the GPU: Conjugate Gradients and Multigrid*, ACM SIGGRAPH 2003 Papers, New York, 2003, pp. 917–924.
- [11] A.F. Camargos, V.C. Silva, J.M. Guichon, and G. Meunier, *Iterative solution on GPU of linear systems arising from the A-V edge-FEA of time-harmonic electromagnetic phenomena*, Proceedings of the 2014 22nd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, Washington, DC, USA, IEEE Computer Society, 2014, pp. 365–371.
- [12] A.K. Cheik Ahamed and F. Magoulès, *Fast sparse matrix–vector multiplication on graphics processing unit for finite element analysis*, 2012 IEEE 14th International Conference on High Performance Computing and Communication 2012 IEEE 9th International Conference on Embedded Software and Systems (HPCC-ICCESS), IEEE Computer Society, 2012, pp. 1307–1314.
- [13] A.K. Cheik Ahamed and F. Magoulès, *Iterative methods for sparse linear systems on graphics processing unit*, High Performance Computing and Communication 2012 IEEE 9th International Conference on Embedded Software and Systems (HPCC-ICCESS), 2012 IEEE 14th International Conference on, 25–27 June, Liverpool, UK, IEEE Computer Society, 2012, pp. 836–842.
- [14] A.K. Cheik Ahamed and F. Magoulès, *Schwarz method with two-sided transmission conditions for the gravity equations on graphics processing unit*, Proceedings of the 12th International Symposium on Distributed Computing and Applications to Business, Engineering and Science (DCABES), Kingston, September 2–4, London, UK, IEEE Computer Society, 2013, pp. 105–109.

- [15] A.K. Cheik Ahamed and F. Magoulès, *Iterative Krylov methods for gravity problems on graphics processing unit*, Proceedings of the 12th International Symposium on Distributed Computing and Applications to Business, Engineering and Science (DCABES), September 2–4, Kingston, London, UK, IEEE Computer Society, 2013, pp. 16–20.
- [16] P. Chevalier and F. Nataf, *Symmetrized method with optimized second-order conditions for the Helmholtz equation, Domain Decomposition Methods, 10 (Boulder, CO, 1997)*, Amercian Mathematical Society, Providence, RI, 1998, pp. 400–407.
- [17] A. Davidson, Y. Zhang, and J.D. Owens, *An auto-tuned method for solving large tridiagonal systems on the GPU*, Proceedings of the 2011 IEEE International Parallel & Distributed Processing Symposium, Washington, DC, USA, IEEE Computer Society, 2011, pp. 956–965.
- [18] A. de La Bourdonnaye, C. Farhat, A. Macedo, F. Magoulès, and F.X. Roux, *A non overlapping domain decomposition method for the exterior Helmholtz problem*, Contemp. Math. 218 (1998), pp. 42–66.
- [19] B. Després, *Domain decomposition method and the Helmholtz problem. II*, Second International Conference on Mathematical and Numerical Aspects of Wave Propagation (Newark, DE, 1993), SIAM, Philadelphia, PA, 1993, pp. 197–206.
- [20] B. Després, P. Joly, and J.E. Roberts, *A domain decomposition method for the harmonic Maxwell equations*, in *Iterative Methods in Linear Algebra (Brussels, 1991)*, R. Beauwens and P. de Groen, eds., Elsevier Science Publishers B. V., North-Holland, Amsterdam, 1992, pp. 475–484.
- [21] P. Du, R. Weber, P. Luszczek, S. Tomov, G. Peterson, and J. Dongarra, *From CUDA to OpenCL: Towards a performance-portable solution for multi-platform GPU programming*, Parallel Comput. 38(8) (2012), pp. 391–407.
- [22] P. Du, P. Luszczek, and J. Dongarra, *OpenCL Evaluation for Numerical Linear Algebra Library Development*, Symposium Application Accelerators in High Performance Computing (SAAHPC10), 2010.
- [23] T. Dufaud and D. Tromeur-Dervout, *Efficient parallel implementation of the fully algebraic multiplicative Aitken-RAS preconditioning technique*, Adv. Eng. Softw. 53 (2012), pp. 33–44.
- [24] IEEE 754: Standard for Binary Floating-Point Arithmetic, 2008. Available at <http://grouper.ieee.org/groups/754> (Accessed September 24, 2014).
- [25] M.J. Gander, L. Halpern, and F. Magoulès, *An optimized Schwarz method with two-sided Robin transmission conditions for the Helmholtz equation*, Int. J. Numer. Methods Fluids 55 (2007), pp. 163–175.
- [26] M. Gander, L. Halpern, F. Magoulès, and F.X. Roux, *Analysis of patch substructuring methods*, Int. J. Appl. Math. Comput. Sci. 17 (2007), pp. 395–402.
- [27] M.J. Gander, F. Magoulès, and F. Nataf, *Optimized Schwarz methods without overlap for the Helmholtz equation*, SIAM 24 (2002), pp. 38–60.
- [28] M. Garbey and D. Tromeur-Dervout, *On some Aitken-like acceleration of the Schwarz method*, Int. J. Numer. Methods Fluids 40(2) (2002), pp. 1493–1513.
- [29] S. Ghanemi, *A domain decomposition method for Helmholtz scattering problems*, Ninth International Conference on Domain Decomposition Methods, ddm.org, 1997, pp. 105–112.
- [30] T.D. Han and T.S. Abdelrahman, *hicuda: High-level GPGPU programming*, IEEE Trans. Parallel Distrib. Syst. 22 (2011), pp. 78–90.
- [31] C. Janna, M. Ferronato, and G. Gambolati, *A block FSAI-ILU parallel preconditioner for symmetric positive definite linear systems*, SIAM J. Sci. Comput. 32 (2010), pp. 2468–2484.
- [32] C. Japhet and F. Nataf, *The best interface conditions for domain decomposition methods: Absorbing boundary conditions*, in *Absorbing Boundaries and Layers, Domain Decomposition Methods. Applications to Large Scale Computations*, L. Tourrette and L. Halpern, eds., Nova Science Publishers Inc., New York, 2001, pp. 348–373.
- [33] Khronos Group, *The OpenCL Specification*, 2010, Available at <http://www.khronos.org> (Accessed September 24, 2014).
- [34] J. Krüger and R. Westermann, *Linear algebra operators for GPU implementation of numerical algorithms*, ACM Trans. Graph. 22 (2003), pp. 908–916.
- [35] J. Kruis, *Domain Decomposition Methods for Distributed Computing*, Saxe-Coburg Publications, Stirling, Scotland, 2007.
- [36] R. Li and Y. Saad, *GPU-accelerated preconditioned iterative linear solvers*, J. Supercomput. 63(2) (2013), pp. 443–466.
- [37] P.L. Lions, *On the Schwarz alternating method. I*, First International Symposium on Domain Decomposition Methods for Partial Differential Equations, SIAM, Philadelphia, PA, 1988, pp. 1–42.
- [38] P.L. Lions, *On the Schwarz alternating method. II*, in *Domain Decomposition Methods*, T.F. Chan, R. Glowinski, J. Périaux, and O. Widlund, eds., SIAM, Philadelphia, PA, 1989, pp. 47–70.
- [39] P.L. Lions, *On the Schwarz alternating method. III: A variant for nonoverlapping subdomains*, Third International Symposium on Domain Decomposition Methods for Partial Differential Equations, Houston, TX, March 20–22, 1989, SIAM, Philadelphia, PA, 1990, pp. 202–223.
- [40] Y. Maday and F. Magoulès, *Non-overlapping additive Schwarz methods tuned to highly heterogeneous media*, Comptes Rendus à l'Académie des Sci. 341 (2005), pp. 701–705.
- [41] Y. Maday and F. Magoulès, *Absorbing interface conditions for domain decomposition methods: A general presentation*, Comput. Methods Appl. Mech. Eng. 195 (2006), pp. 3880–3900.
- [42] Y. Maday and F. Magoulès, *Improved ad hoc interface conditions for Schwarz solution procedure tuned to highly heterogeneous media*, Appl. Math. Model. 30 (2006), pp. 731–743.

- [43] Y. Maday and F. Magoulès, *Optimized Schwarz methods without overlap for highly heterogeneous media*, Comput. Methods Appl. Mech. Eng. 196 (2007), pp. 1541–1553.
- [44] F. Magoulès, A.-K. Cheik Ahamed, and R. Putanowicz, *Auto-tuned Krylov methods on cluster of graphics processing unit*, Int. J. Comput. Math. 92(6) (2015), pp. 1222–1250.
- [45] F. Magoulès, P. Iványi, and B.H.V. Topping, *Convergence analysis of Schwarz methods without overlap for the Helmholtz equation*, Comput. Struct. 82 (2004), pp. 1835–1847.
- [46] F. Magoulès and F.-X. Roux, *Lagrangian formulation of domain decomposition methods: A unified theory*, Appl. Math. Model. 30 (2006), pp. 593–615.
- [47] F. Magoulès, F.-X. Roux, and L. Series, *Algebraic way to derive absorbing boundary conditions for the Helmholtz equation*, J. Comput. Acoust. 13 (2005), pp. 433–454.
- [48] F. Magoulès, F.-X. Roux, and L. Series, *Algebraic approximation of Dirichlet-to-Neumann maps for the equations of linear elasticity*, Comput. Methods Appl. Mech. Eng. 195 (2006), pp. 3742–3759.
- [49] F. Magoulès, F.-X. Roux, and L. Series, *Algebraic Dirichlet-to-Neumann mapping for linear elasticity problems with extreme contrasts in the coefficients*, Appl. Math. Model. 30 (2006), pp. 702–713.
- [50] F. Magoulès, F.-X. Roux, and L. Series, *Algebraic approach to absorbing boundary conditions for the Helmholtz equation*, Int. J. Comput. Math. 84 (2007), pp. 231–240.
- [51] K.K. Matam and K. Kothapalli, *Accelerating sparse matrix Vector multiplication in iterative methods using GPU*, Proceedings of the 2011 International Conference on Parallel Processing (ICPP '11), Washington, DC, USA, IEEE Computer Society, 2011, pp. 612–621.
- [52] J. Meredith, D. Bremer, L. Flath, J. Johnson, H. Jones, S. Vaidya, and R. Frank, *The GAIA project: Evaluation of GPU-based programming environments for knowledge discovery*, Tech. rep., Lawrence Livermore National Labs, Livermore, 2004.
- [53] F. Nataf, F. Rogier, and E. de Sturler, *Optimal Interface Conditions for Domain Decomposition Methods*, CMAP (Ecole Polytechnique) 301 (1994), pp. 1–18.
- [54] S. Noury, S. Boivin, and O. L. Maître, *A Fast Poisson Solver for OpenCL using Multigrid Methods*, GPU Pro 2, W. Engel, Ed. A.K. Peters, 2011, pp. 445–471. ISBN 978-1-56881-718-7.
- [55] Nvidia Corporation, *CUDA toolkit 4.0, CUBLAS Library*, 2011. Available at <http://developer.nvidia.com/cuda-toolkit-40> (Accessed September 24, 2014).
- [56] J.D. Owens, M. Houston, D. Luebke, S. Green, J.E. Stone, and J.C. Phillips, *GPU computing*, Proc. IEEE 96 (2008), pp. 879–899.
- [57] J.D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A.E. Lefohn, and T.J. Purcell, *A survey of general-purpose computation on graphics hardware*, EUROGRAPHICS (2005). Available at <http://www.blackwell-synergy.com/doi/pdf/10.1111/j.1467-8659.2007.01012.x> (Accessed March 7, 2015).
- [58] A. Quarteroni and A. Valli, *Domain Decomposition Methods for Partial Differential Equations*, Oxford University Press, Oxford, 1999.
- [59] S. Ristov, M. Gusev, L. Djinevski, and S. Arsenovski, *Performance impact of reconfigurable L1 cache on GPU devices*, Federated Conference on Computer Science and Information Systems (FedCSIS 2013), IEEE Conference Proceedings, Krakow, Poland, September 9–11, 2013, pp. 507–510.
- [60] F.X. Roux, F. Magoulès, L. Series, and Y. Boubendir, *Approximation of optimal interface boundary conditions for two-Lagrange multiplier FETI method*, Proceedings of the 15th International Conference on Domain Decomposition Methods, Berlin, Germany, July 21–15, 2003, R. Kornhuber, R. Hoppe, J. Périaux, O. Pironneau, O. Widlund, and J. Xu, eds., Lecture Notes in Computational Science and Engineering, Springer-Verlag, Heidelberg, 2005, pp. 283–290.
- [61] H. Schwarz, *Über einen Grenzübergang durch alternierendes Verfahren*, Vierteljahrsschrift der Naturforschenden Gesellschaft in Zürich 15 (1870), pp. 272–286.
- [62] B. Smith, P. Bjorstad, and W. Gropp, *Domain Decomposition: Parallel Multilevel Methods for Elliptic Partial Differential Equations*, Cambridge University Press, New York, NY, 1996.
- [63] J. Sulaiman, M. Othman, and M.K. Hasan, *Nine Point-EDGSOR Iterative method for the finite element solution of 2D Poisson equations*, in *International Conference Computational Science and Its Applications (ICCSA 2009)*, Seoul, Korea, June 29–July 2, Lecture Notes in Computer Science, Vol. 5592, O. Gervasi, D. Taniar, B. Murgante, A. Lagan, Y. Mun, and M. Gavrilova, eds., Springer, Berlin and Heidelberg, 2009, pp. 764–774.
- [64] J.E. Stone, D. Gohara, and G. Shi, *OpenCL ccc A Parallel Programming Standard for Heterogeneous Computing Systems*, IEEE Des. Test 12(3) (2010), pp. 66–73.
- [65] C.J. Thompson, S. Hahn, and M. Oskin, *Using modern graphics architectures for general-purpose computing: a framework and analysis*, Proceedings of the 35th annual ACM/IEEE international symposium on Microarchitecture, MICRO 35, Istanbul, Turkey, IEEE Computer Society Press, Los Alamitos, CA, 2002, pp. 306–317.
- [66] P. Tillet, K. Rupp, and S. Selberherr, *An Automatic OpenCL Compute Kernel Generator for Basic Linear Algebra Operations*, Proceedings of the 2012 Symposium on High Performance Computing, Orlando, FL, 2012, pp. 4:1–4:2.
- [67] A. Toselli and O. Widlund, *Domain Decomposition methods: Algorithms and Theory*, Springer-Verlag, Berlin, 2005.
- [68] L. Tournette and L. Halpern, *Absorbing Boundaries and Layers, Domain Decomposition Methods: Applications to Large Scale Computers*, Nova Science Publishers, 2001. Available at <http://books.google.fr/books?id=KrCsq6WStwC>.
- [69] A.H.E. Zein and A.P. Rendell, *Generating optimal CUDA sparse matrix–vector product implementations for evolving GPU hardware*, Concurrency Comput. Pract. Exp. 24 (2012), pp. 3–13.