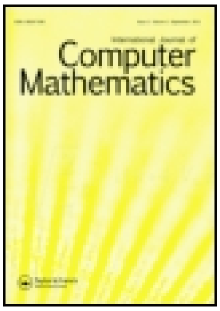


International Journal of Computer Mathematics
Publication details, including instructions for authors and subscription information:
http://www.tandfonline.com/loi/gcom20

Auto-tuned Krylov Methods on Cluster of Graphics Processing Unit

Frédéric Magoulès[a], Abal-Kassim Cheik Ahamed[b] & Roman Putanowicz[c]
[a] Ecole Centrale Paris, France
[b] Ecole Centrale Paris, France
[c] Cracow University of Technology, Poland
Accepted author version posted online: 02 Jun 2014.Published online: 02 Jun 2014.

Publisher: Taylor & Francis
Journal: *International Journal of Computer Mathematics*
DOI: 10.1080/00207160.2014.930137

Auto-tuned Krylov Methods on Cluster of Graphics Processing Unit

Frédéric Magoulès* , Abal-Kassim Cheik Ahamed** and Roman Putanowicz***

*,** *Ecole Centrale Paris, France*

*** *Cracow University of Technology, Poland*

(*Received 01 Mar 2014, revised 16 Apr 2014, accepted 26 May 2014*)

Exascale computers are expected to have highly hierarchical architectures with nodes composed by multiple core processors (CPU) and accelerators (GPU). The different programming levels generate new difficult algorithm issues. In particular when solving extremely large linear systems, new programming paradigms of Krylov methods should be defined and evaluated with respect to modern state-of-the-art of scientific methods. Iterative Krylov methods involve linear algebra operations such as dot product, norm, addition of vectors and sparse matrix-vector multiplication. These operations are computationally expensive for large size matrices. In this paper, we aim to focus on the best way to perform effectively these operations, in double precision, on Graphics Processing Unit (GPU) in order to make iterative Krylov methods more robust and therefore reduce the computing time. The performance of our algorithms are evaluated on several matrices arising from engineering problems. Numerical experiments illustrate the robustness and accuracy of our implementation compared to the existing libraries. We deal with different preconditioned Krylov methods: Conjugate Gradient (P-CG) for symmetric positive definite matrices, and Generalized Conjugate Residual (P-GCR), Bi-Conjugate Gradient Conjugate Residual (P-BiCGCR), transpose-free Quasi Minimal Residual (P-tfQMR), Stabilized BiConjugate Gradient (BiCGStab) and Stabilized BiConjugate Gradient (L) (BiCGStabl) for the solution of sparse linear systems with non symmetric matrices. We consider and compare several sparse compressed formats, and propose a way to implement effectively Krylov methods on GPU and on multicore CPU. Finally, we give strategies to faster algorithms by auto-tuning the threading design, upon the problem characteristics and the hardware changes. As a conclusion, we propose and analyze hybrid sub-structuring methods that should pave the way to exascale hybrid methods.

Keywords: Krylov methods; iterative methods; linear algebra; sparse matrix-vector product; GPU; CUDA; auto-tuning; Compressed-Sparse Row (CSR) format; ELLPACK (ELL) format; Hybrid (HYB) format; Coordinate (Coo) format; Cusp; CUSPARSE; CUBLAS

1. Introduction

Many consumer personal computers are equipped with powerful Graphics Processing Units (GPUs) which can be harnessed for general purpose computations. Distributed computing constantly gains in importance and becomes important tool in common scientific research work. Comparing the evolution of Central Processing Units (CPUs) to the one of GPUs, one can talk about the revolution in case of GPUs, looking at it from the perspective of the reached performance and advances in multicore designs. Currently, the computational efficiency of GPU reaches the Teraflops (Tflops) for single precision computations and roughly half of that for double precision computations, all this for one single graphics card. Recent developments resulting in easier programmable access to

*Email: frederic.magoules@hotmail.com

GPU devices, have made it possible to use them for general scientific numerical simulations. Among many tools there are software libraries, such as Cusp [5], that enable solving sparse linear systems with various iterative Krylov methods, both on CPU and GPU devices. Such libraries are important because they help to achieve portability of simulation codes, an important issue in nowadays heterogeneous, grid based, simulation environments. The other important aspect is automatic tuning of algorithm characteristics to different hardware architectures, key point to achieve good performance of simulations.

The factors that contribute to the overall performance of linear algebra algorithms are data structures design (i.e. sparse matrix formats), optimization of memory layout and access patterns, and implementation of GPU kernel core algebraic operations. The aim of this paper is to investigate such factors and to search for the best way to solve large sparse linear systems efficiently by taking into account hardware characteristics and problem features. In order to measure our contribution, we compare our results to measurements obtained for other libraries, such CUBLAS, CUSPARSE and Cusp, that set some sort of a standard in their category. *Alinea*, our research group library, is introduced and compared with the existing linear algebra libraries on GPU. For numerical experiments we use matrices taken form the matrix repository at University of Florida. We hope that the results presented in this paper are helpful in setting the direction for the further optimisation of linear algebraic solvers, that constitute important element of numerical simulation software.

The plan of this paper is as follows. First an introduction to the GPU programming model and hardware configuration issues is given in Section 2. The next section discusses the best way to efficiently perform linear algebra operations, in particular dot product and sparse matrix-vector multiplication (SpMV). In Section 3 we report our research on the issue of auto-tuning the gridification by taking into account the features of GPU architecture in order to optimize execution of linear algebra operations. Section 4 contains a general description of the parallel iterative Krylov methods, particularly the parallel Conjugate Gradient algorithm by sub-structuring method. In section 5 we describe how Krylov methods are implemented on GPU, and how the performance of the basic linear algebra operations influence the overall performance of these methods. Finally, Section 6 concludes this paper.

2. GPU programming model

The performance reached by add-on graphics cards has attracted the attention of many scientists, with the underlying idea of utilizing GPU devices for other tasks than graphics related computations. In the beginning of 2000s this gave the birth to the technology known as GPGPU (General Purpose computation on GPU) [6, 17, 40]. The real boost to this technology was the introduction by Nvidia of their programming paradigm and tools referred as CUDA (*Compute Unified Device Architecture*) [34]. CUDA and OpenCL [36] (*Open Computing Language*) provide tools based on extensions to the most popular programming languages (most notably C/C++/Fortran), that make computing resources of GPU devices easily accessible for utilisation for general tasks. Effective use of these tools requires understanding of implications of GPU hardware architecture, thus in this section we provide brief overview of GPU architecture and then discuss the issue of memory access and management of computing threads referred as *gridification*.

2.1 *Architecture*

The GPU device architecture stems from their role in speeding up graphic related data processing independently from the load of the central processing unit. Tasks such as texture processing or ray tracing imply doing similar calculations on a large amount of independent data (Single Instruction Multiple Data). Thus the main idea behind the GPU design is to have several simple floating point processors working on large amount of partitioned data in parallel. The key point of effective processing of such data lies in specially designed memory hierarchy that allows each processor to access requested data optimally. While nowadays a CPU has hardly more than 8 cores, a modern GPU has more than 400 as illustrated in Figure 1. Furthermore, several hundreds ALUs (Arithmetical and Logical Units) are integrated inside a single GPU card. All this contributes to the ability of GPU to perform massively parallel data processing.

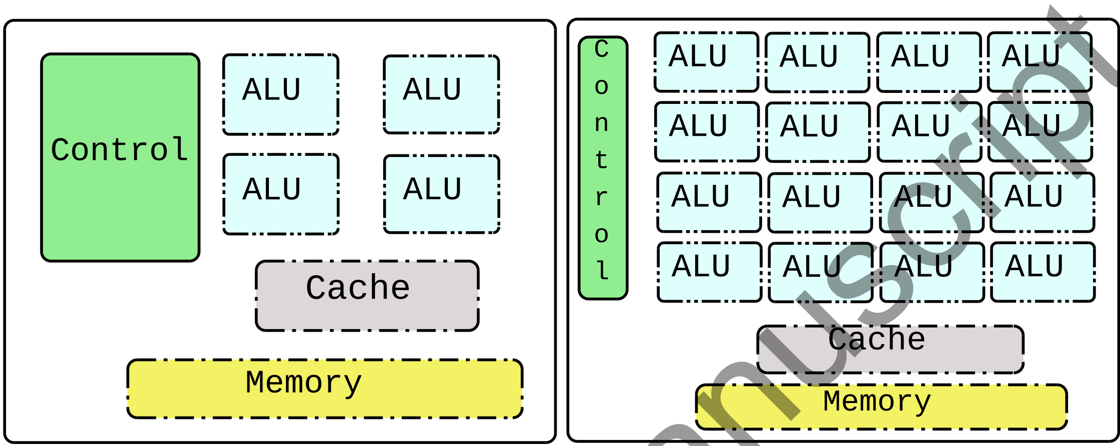


Figure 1. Difference of CPU and GPU architecture

2.2 *Overview of CUDA programming and hardware configuration*

In this section we will discuss important issues of CUDA programming paradigm. Understanding the issues of memory access, management of execution threads and necessity to account for the features of particular GPU device is important in order to provide effective implementation of the basic linear algebra operations that affect the performance of more complex algorithms.

2.2.1 *CUDA programming language*

CUDA propounded by Nvidia is a programming paradigm and the same time a set of tools that give easier access to computing power of Nvidia GPUs devices. Although being proprietary, CUDA technology sets de facto standard in GPGPU driven computing, being feature rich and delivering top performance. The CUDA technology is based on extending high level languages, that simplify and optimize handling of GPU memory and thread execution. Initially based on C language, nowadays it is also available for C++ and Fortran. For other languages, most notably scripting languages commonly used in programming numerical simulations, like Matlab, Python or Haskel, it is possible to find third party wrappers. In our presentation we will show examples based on C programming language.

2.2.2 *Memory*

The memory layout and access patterns are the features that strongly differentiate CPU and GPU devices. CPU memory exhibits very low latency for the price of reduced throughput. On contrary, GPU devices allow to push large portions of data but the access to their memory is rather slow.

We discuss here four main types of memory as illustrated in Figure 2: *global, local, constant and shared*. *Global memory* is available to all threads (also known as compute units) and is the slowest one. It is the memory that ensures the interaction with the host (CPU) and is large in size. *Local memory* is specific to each compute unit and cannot be used to communicate between compute units. It is much faster than the *global memory*. The local and global memory spaces are not cached what means that each memory accesses to local or global memory generates an explicit, true memory access. *Constant memory* is generally cached for fast access and is read-only from the device. It also provides interaction with the host. Compute units are divided into blocks of threads. *Shared memory* is much faster than global memory and is accessible by any thread of the block from which it was created. Each block has its own *shared memory*.

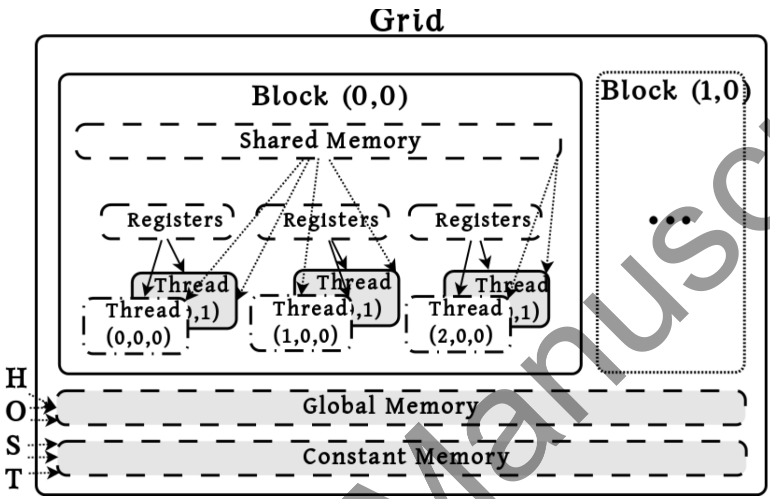


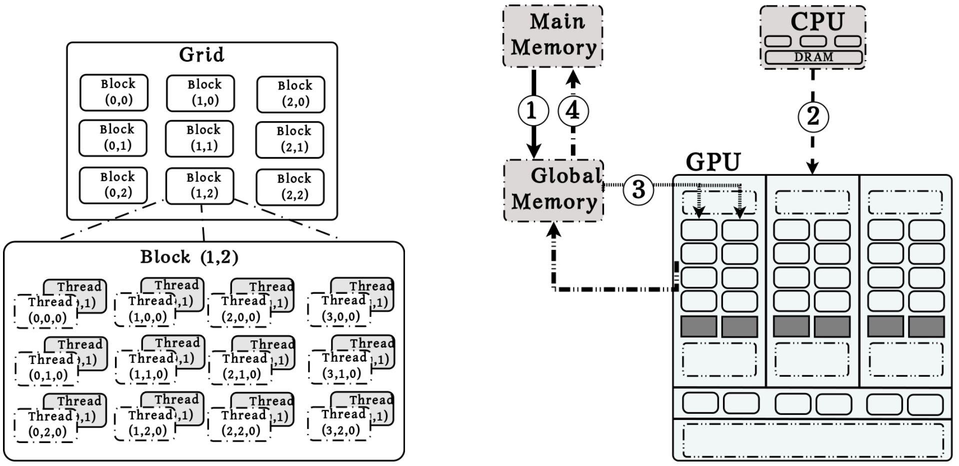
Figure 2. Memory type

An issue that is closely related to memory issues is the selected precision of computations, that affects the size of real number representation. For most of the cases, for graphic computations it is sufficient to rely on single precision (32-bits) [14], and original GPUs provided only this mode. Unfortunately for general purpose numerical simulations this assumption is to restrictive, thus in the latest GPU and CUDA versions the possibility to perform double precision computations has appeared. One should however keep in mind that double precision computation time is usually 4 to 8 times higher than for single precision case.

2.2.3 Gridification

Gridification is the concept of threads distribution over the GPU grid. A thread is the smallest unit of processing that can be scheduled by an operating system. CUDA threads are collected into blocks as shown in Figure 3(a) and executed simultaneously. A group of 32 *threads* executed together is called *warp*. An *ALU* is associated with the active *thread* and a *GPU* is associated with a *grid*. By the grid we understand all running or waiting blocks in the running queue, and a *kernel* that will run on many cores. There is one restriction to be kept in mind that the threads of a wrap are necessarily executed together. The notion of *warp* is close to the SIMD (Single Instruction, Multiple Data) execution idea, corresponding to an execution of a same program on multiple data.

The design of the grid, *i.e.* threading pattern, is not an automated process, the devel-



(a) thread, block, grid

(b) GPU Computing processing flow with CUDA

Figure 3. Gridification

oper chooses for each *kernel* the distribution of the threads. This procedure is named *"gridification"*.

The configuration hierarchy of a GPU grid looks as follows: (i) threads are grouped into blocks, (ii) each block has three dimensions to index the threads, (iii) blocks are grouped together in a grid of two dimensions. The threads are then distributed according to this hierarchy and become easily identifiable by their positions in the grid described by the block number of a block they belong to and their index within the block. As an illustration, the global index of current local thread is retrieved by the following expression:

```
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)
```

If we use a two-dimensional grid, the global index is given by

```
int x = blockIdx.x * blockDim.x + threadIdx.x;
int y = threadIdx.y + blockIdx.y * blockDim.y;
int pitch = blockDim.x * gridDim.x;
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
```

Figure 3(b) shows GPU Computing processing flow. As the initial step, data is copied from the main memory to the GPU memory (1). Then the host (CPU) orders the device (GPU) to perform calculations (2). CUDA *kernel* represents the CUDA function executed on the GPU. Finally, the device results are copied back from GPU memory to the main host memory (3) and (4). The state of art [7, 11] demonstrated that grid tuning, *i.e.* adjusting the properties of the grid, has a significant impact on the performances of the kernels.

In the next subsection we will discuss our tuning strategy in detail.

2.2.4 Auto-tuning

In order to build efficient implementations, programmers must take care of hardware configuration issues. The main problem lies in setting up an optimal threads hierarchy and grid configuration that will match the hardware configuration and the specific problem parameters, for instance the size of processed matrices.

The kernel function requires at least two additional arguments corresponding to the gridification: the number of threads per block, nThreadsPerBlock, and the dimension of the block, nBlocks. An example of a kernel function declaration for one-dimensional grid is given below:

```
MyKernel<<<nBlocks, nThreadsPerBlock>>>(arguments);
```

These values depend both on the total number of necessary threads related to the size of a problem to be solved, and both number of blocks and number of threads per block, that is the specificity of the GPU architecture. Choosing a number of threads higher than the amount supported natively will result in non-optimal performance. Accordingly, auto-tuning of the gridification is a way that helps the CUDA program to achieve near-optimal performance on a GPU architecture. The idea behind the tuning of the grid is to recognize device features and then adjust the gridification according to the problem size. We focus on finding the gridification, which gives the best performance.

To illustrate the concept of gridification let us consider the example shown in the listing below:

```
//Kernel definition
__global__ void MyKernel(float* parameter);
//Using the kernel
MyKernel <<< Dg, Db, Ns, S >>> (parameter);
```
Listing 1 Kernel parameters and call

where Dg, Db, Ns, S denote respectively: (i) Dg, the number of blocks of type dim3 is used to define the size and dimension of the grid (the product of these three components provides the number of units launched); (ii) Db, the number of threads per block of type dim3 is used to specify the size and dimension of each block (the product of these three components offers the number of threads per block); (iii) Ns, the number of bytes in shared memory of type size_t represents the number of bytes allocated dynamically in shared memory per block in addition to statically allocated memory (by default Ns=0); (iv) S corresponds to the CUDA stream of type cudaStream_t, and gives the associated stream (by default S=0). Each kernel has *read-only* implicit variables of type dim3: blockDim, the number of threads per block, *i.e.* value of nThreadsPerBlock of kernel's setting, blockIdx, the index of the block in the grid and threadIdx, the index of the thread in the block.

For the sake of clarity and without loss of generality, the gridification is illustrated for the Daxpy (Double-precision real Alpha X Plus Y) operation:

```
__global__ void Daxpy(double alpha, const double* d_x,
                      double* d_y, int size) {
  unsigned int idx=blockIdx.x*blockDim.x+threadIdx.x;
  if(idx<size) {
      d_y[idx] = alpha * d_x[idx] + d_y[idx];
  }
}
```
Listing 2 Daxpy on one-dimensional grid

As an example of a one-dimensional grid, consider one block of *size* number of threads:

```
int main(int argc, char** argv) {
  // -- variables declaration and initialization
  ...
  // -- kernel call
  Daxpy<<<1, size>>>(alpha,d_x,d_y,size);
  // a vector of size element added once
}
```
Listing 3 basic kernel call

A basic self-configuration of the number of blocks, by taking account the given number of threads, can be formulated as follows:

```
//call function
int main(int argc, char** argv) {
  //kernel call
  const int nThreadPerBlock = 50;
  const int nBlocks = 1 + (size-1)/nThreadPerBlock;
  Daxpy<<<nBlocks,nThreadPerBlock>>>(alpha,d_x,d_y,size);
}
```

Listing 4 kernel call with basic self-threading computation

where nBlocks and nThreadPerBlock of type dim3 are required for 2-d an 3-d. The following code shows the *Daxpy* kernel assuming two-dimensional grid.

```
__global__ void Daxpy(double alpha, const 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;
  int pitch = blockDim.x * gridDim.x;
  int idx = x + y * pitch;
  if(idx<size) {
      d_y[idx] = alpha * d_x[idx] + d_y[idx];
  }
}
```

Listing 5 Daxpy on two-dimensional grid

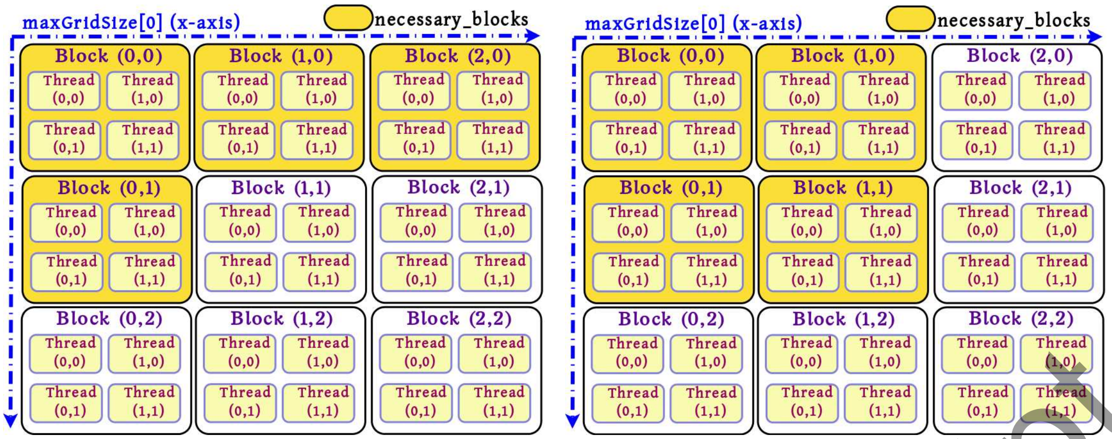
If the vector content exceeds the one-dimensional grid, the extra coefficients are stored on the second dimension. According to the kernel implementation, we may avoid it by building the kernel configuration upon the hardware characteristics such as:

```
dim3 ComputeGridBlock(int size,
                      int thread_per_block) {
  dim3 nblocks;
  // compute the necessary blocks
  int necessary_blocks=1+(size-1)/thread_per_block;
  //maximum 1D-grid size
  int max_grid_size_x=dev_properties.maxGridSize[0];
  // Required two-dimensional grid
  if(necessary_blocks > max_grid_size_x) {
    nblocks.x=max_grid_size_x;
    nblocks.y=(necessary_blocks-1)/max_grid_size_x+1;
    nblocks.z=1;
  } else { //Dimensional grid sufficient
    nblocks.x = necessary_blocks;
    nblocks.y = 1;
    nblocks.z = 1;
  }
  return nblocks;
}
```

Listing 6 One-dimensional grid fully used

```
dim3 ComputeGridBlock(int size, int thread_per_block) {
  dim3 nblocks;
  // compute the necessary blocks
  int necessary_blocks = 1 + (size-1) / thread_per_block;
  int max_grid_size_x  = dev_properties.maxGridSize[0];
  if(necessary_blocks <= max_grid_size_x) {
    // Dimensional grid sufficient
    nblocks = dim3(necessary_blocks);
  } else {
    // Required two-dimensional grid
    int each_block_dim = ceil(sqrt(necessary_blocks));
    nblocks = dim3(each_block_dim, each_block_dim);
  }
  return nblocks;
}
```

Listing 7 Two-dimensional grid



(a) Skeleton of “one-dimensional grid fully used” (b) Skeleton of “two-dimensional grid”

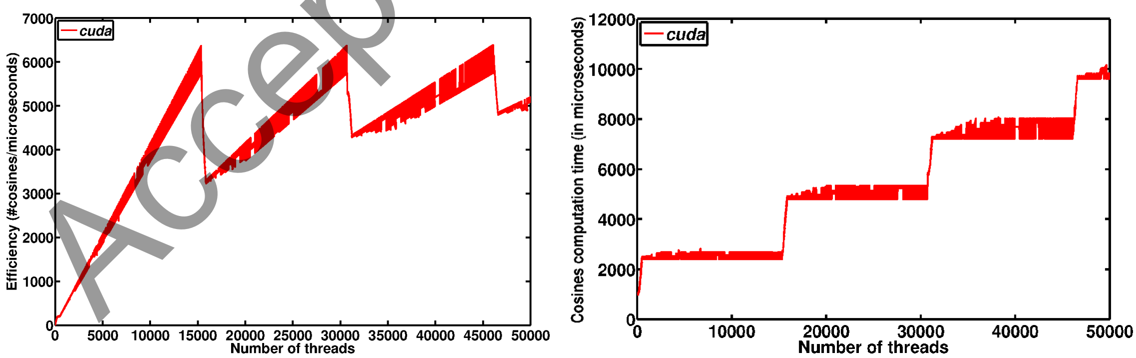
Figure 4. Grid computing strategies

Figure 4(a) and Figure 4(b) illustrate respectively the first strategy presented in Listing 6 and the second one exposed in Listing 7. Line 7 of the former listing and line 5 of the later one calculate the maximum size of the first dimension. The strategy implemented in Listing 6, consists in using all threads of the first dimension of the grid and then completing the remaining ones with the second dimension. The strategy implemented in Listing 7, uses a two-dimensional square grid. This strategy involves a greater proximity between the blocks of the first dimension with the blocks of the second dimension, leading to better efficiency as we will see in the next sections.

2.2.5 Data transfers and memory access

We focus on the CPU and GPU interaction and communication with respect to minimize CPU and GPU idling, and maximize the total throughput.

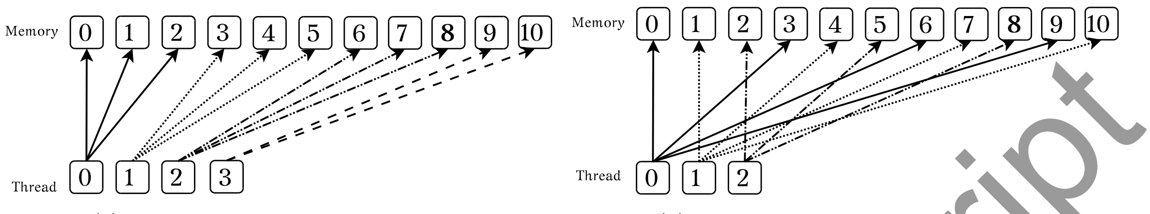
Figure 5(a) shows the effectiveness of cosine function upon the number of threads, where the efficiency calculation is based on the number of transactions per second. Figure 5(a) clearly illustrates that the GPU is more effective when the required number



(a) Cosines raw efficiency (b) Cosines computing time in microseconds

Figure 5. Cosines raw efficiency (a) and computing time in microseconds (b)

of operations increases. The higher is the number of operations, the smaller is the influence of particular gridification. The shape of the curve in Figure 5(b) that represents the cosine computation time in microseconds, confirms this phenomena and the importance of the gridification for small data sizes. A gap in the curve is observed when load remains constant while block is not fully utilized, which involve an overloading when an extra warp is required. More requests are required if each thread accesses a distant memory location, contrary to the case when threads access localised memory locations. The performance bottleneck may occur when memory access is poorly managed. To illustrate this phenomenon, we carry out two tests where each thread accesses a memory location to indices $i + k * step$ first, as described in Figure 6(a) and second to indices $i * step + k$, as described in Figure 6(b), for $k = 0, \ldots, 1000$ where i denotes the index of the thread. Figure 7 shows that when we vary $step$ for the indices $i * step + k$ the threads are access-



(a) Gather memory access, $i + k * step$ (b) Scatter memory access, $i * step + k$

Figure 6. Memory access

ing memory locations more far from each other and this results in degraded performance because a given memory request can satisfy only one thread. On other hand, it also illustrates that for the indices $i + k * step$ the threads are always accessing contiguous memory locations, and thus it results in good performance because of localised memory access.

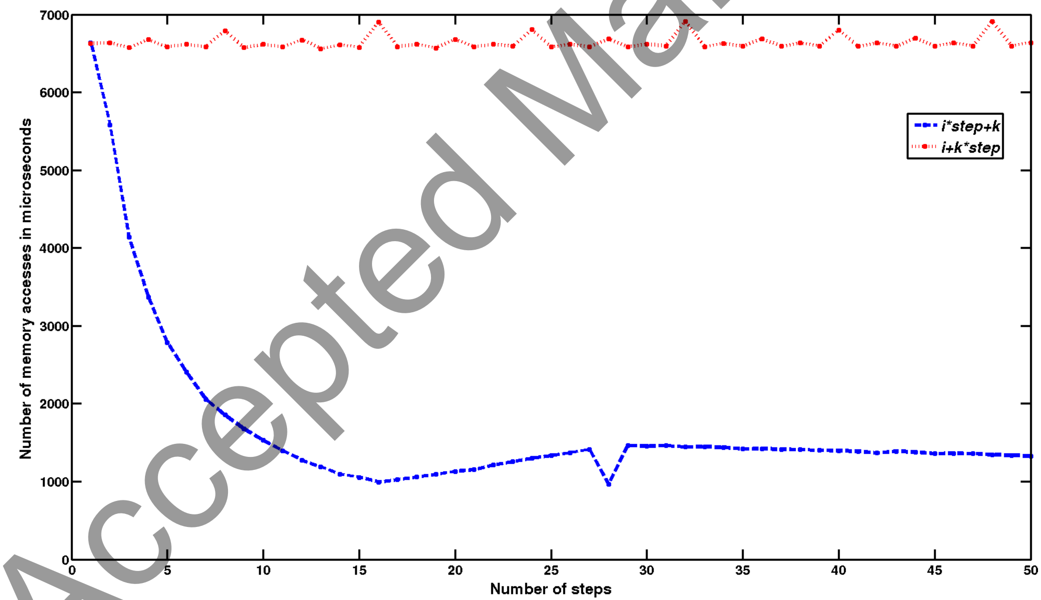


Figure 7. Number of memory access per microseconds

3. Linear algebra operations

Linear algebra operations are at the very core of scientific computations. Among them, the primary place is taken by solvers of linear algebraic systems. For most of the time, in practical problems, one has to deal with large sparse matrices, with the matrix sparsity inherent to the nature of the problem or resulting from the design of computational methods. In the set of various operations, the sparse matrix vector multiplication is commonly considered as the most time consuming operation. Being the basic operation for iterative algorithms, sparse matrix vector multiplication is the best candidate for optimisation and efficient parallelisation with respect to GPU use.

3.1 *Overview of existing scientific libraries*

In our comparative study of scientific libraries targeted for GPU, we concentrate exclusively on libraries implemented with CUDA provided languages. The primary target of the study is our Alinea library. Alinea is designed to provide efficient linear algebra operations on hybrid multi-CPU and GPU clusters. The framework offers direct and iterative solvers for solving large, both dense and sparse, linear systems. Our primary aim is to ease the development of engineering and scientific applications on both CPU and GPU by hiding most of the intricacies encountered when dealing with these architectures. In Alinea we pay particular attention to auto-tuning implementation of GPU with accordance to the present and future hardware on which the library is likely to be used. For maximum flexibility, the library provides a whole set of matrix data storage formats. In the subsequent sections Alinea will be compared with the following libraries: *CUBLAS*, *CUSPARSE*, *Cusp*, *CUDA_ITSOL*. The *CUBLAS* [32] library, which is a CUDA version of the Basic Linear Algebra Subprograms (BLAS) library, gives the basic set of tools to access to the computational resources of Nvidia GPUs. The *CUSPARSE* [33] library is an Nvidia CUDA library for basic linear algebra used for sparse matrix operations. The *Cusp* [5] library provides flexible, high-level interface for manipulating sparse matrices and solving sparse linear systems on GPU. Cusp appears as the popular library that offers GPU iterative solvers. Cusp library includes various sparse matrix formats such as *COO, CSR, ELL and HYB*. The *CUDA_ITSOL* [18] library is developed under CUDA language by Ruipeng Li [PhD Student supervised by Yousef Saad, Univ. of Minnesota]. It supports several sparse matrix operations and, more importantly, provides a variety of GPU linear systems iterative solvers.

3.2 *Environment for benchmarks*

The numerical experiments and benchmarks were performed on a workstation equipped with an Intel Core i7 920 2.67GHz processor, which has 8 cores composed of 4 physical cores and 4 logical cores, 5.8 GB RAM memory and two NVIDIA GTX275 GPUs fitted with 895MB memory. This configuration is adequate for carrying out all linear algebra operations for the selection of matrices used in the tests. The workstation hardware and software configuration allows to use CUDA 4.0 features. All presented test are performed with *double precision* data.

3.3 *Measurement of execution time*

The resolution and accuracy of the clock depends on the hardware platform: host/CPU or device/GPU. The original clock of a graphic card has an accuracy of a few nanoseconds while the host has an accuracy of a few milliseconds. This difference might negatively influence the objective comparison of the measured times. To overcome this problem, the proposed solution is to perform the same operation several times, at least 10 times and until the total time measured is greater than 100 times the accuracy of the clock.

3.4 *Addition of vectors*

Daxpy is a vector scaled update operation defined in the level one (vector) Basic Linear Algebra Subprograms (BLAS) specification. The first implementation of BLAS on GPU has been reported in [32]. Vector addition and scalar-vector multiplication are inherently parallel, what makes them excellent candidates for implementation on a GPU. Alinea library propose a basic *Daxpy* algorithm described in Listing 2. The update, which is done in place i.e. $y[i] = \alpha \times x[i] + y[i]$ is done with simple GPU kernel. The same strategy is applied for scalar-vector multiplication, also done in-place, i.e. $a[i] = a[i] \times b[i]$.

3.5 *Parallel dot product*

The computation of dot product and Euclidean norm are prevailing linear algebra operations, which could be very expensive both in terms of computing time and memory storage for large size vectors. The dot product operation is characterized by a simple loop with simultaneous summation, which is computationally expensive in a sequential implementation on CPU. The optimized dot product operation given here is computed in two steps. The first task is to multiply the elements of both vectors one by one and the second task consists in calculating the sum of each of these products to get the final result. On a sequential processor, the summation operation is implemented by writing a single loop with a single accumulator variable to construct the sum of all elements in the sequence. In case of parallel processing each element of the input data is handled by one thread. The first thread (thread 0) of each block stores the sum result of all elements of all threads of the block at the end of the reduction operation. The scalar product is acquired by adding all partial sums of all blocks.

Table 1. Kernel execution time in milliseconds

Size	Daxpy			Dot product			Norm		
	Alinea	CUBLAS	Cusp	Alinea	CUBLAS	Cusp	Alinea	CUBLAS	Cusp
10^3	0.045	0.030	0.025	0.045	0.060	0.202	0.053	0.151	0.200
10^5	0.070	0.033	0.051	0.082	0.077	0.234	0.114	0.059	0.317
10^7	2.490	0.037	2.607	1.693	0.082	1.958	5.301	0.077	1.146

The experiments have been performed on vectors of different size changing from 10^3 to 10^7. Table 1 shows the computation time in milliseconds for the Daxpy, dot product and norm operations, by considering 256 threads per block for Alinea Daxpy experiment and, 128 threads per block for Alinea dot product and norm. Alinea dot product outperforms the Cusp dot product as we can see in Table 1. Indeed CUBLAS gives better results that Cusp and Alinea for basic linear operations (BLAS 1). Since CUSPARSE for basic linear operations (BLAS 2 and BLAS 3), in section 3.6, is based on CUBLAS implementation, it is important to outline the efficiency of CUBLAS, in section 3.5, This ensure an exhaustive comparison of Alinea v.s. CUSPARSE, in section 4, when solving large scale linear systems. Explanations have been added.

3.6 *Parallel Sparse matrix-vector product (SpMV)*

In this section we discuss the data structures for matrix storage involved in this paper. In the following are presented some popular and more specific storage formats.

3.6.1 Data structures

Several computer methods, for instance finite element method, result in algebraic problems for large size sparse matrices. Diverse data structures can be considered to store matrices more efficiently in memory. The basic idea for sparse matrix storage is to store only the non-zero matrix elements. Such data structures offer several advantages in terms of memory usage and algorithm execution strategy, compared to naive matrix data structure implementation. Unfortunately, each special data format also exhibits disadvantages and trade-offs when compared to other special purpose formats. The purpose of this part is to present the data structures, as well as their advantages and disadvantages. The purpose of each of these formats is to gain efficiency both in terms of number of arithmetic operations and memory usage. We describe in detail the storage schemes that are handled in this paper: COOrdinate format (*COO*), Compressed-Sparse Row format (*CSR*), ELLPACK format (*ELL*) and HYBrid format (*HYB*). The sparse matrix A drawn in Figure 8 will be used to illustrate each case.

$$A = \begin{pmatrix} -5 & 14 & 0 & 0 & 0 \\ 0 & 8 & 1 & 0 & 0 \\ 2 & 0 & 10 & 0 & 0 \\ 0 & 4 & 0 & 2 & 9 \\ 0 & 0 & 15 & 0 & 7 \end{pmatrix}$$

Figure 8. Example of basic sparse matrix

The structure of the *COO*rdinate (COO (1)) format is trivial [4]. It consists of three one-dimensional arrays of size nnz, equal to the number of non-zero matrix elements. An array AA holds the non-zero coefficients in such a way that $AA(i) = A(IA(i), JA(i))$, where IA and JA store respectively the indices of rows and columns of the i^{th} value of A.

$$\begin{aligned} IA &= 1 \;/\; 1 \;/\; 2 \;/\; 2 \;/\; 3 \;/\; 3 \;/\; 4 \;/\; 4 \;/\; 4 \;/\; 5 \;/\; 5, \\ JA &= 1 \;/\; 2 \;/\; 2 \;/\; 3 \;/\; 1 \;/\; 3 \;/\; 2 \;/\; 4 \;/\; 5 \;/\; 3 \;/\; 5, \\ AA &= -5 \;/\; 14 \;/\; 8 \;/\; 1 \;/\; 2 \;/\; 10 \;/\; 4 \;/\; 2 \;/\; 9 \;/\; 15 \;/\; 7 \end{aligned} \tag{1}$$

The Compressed-Sparse Row storage format (CSR (2)) [4], [35] is probably the most commonly used format due to the great savings it terms of computer memory. The sparse matrix $A(n \times m)$ is stored in three one-dimensional arrays. Two arrays of size nnz, AA and JA, store respectively the non-zero values, stored by major row storage (row by row) and the column indices, thus $JA(k)$ is the column index in A matrix of $AA(i)$. Finally, IA, an array of size $n+1$ that stores the list of indices at which each row starts. $IA(i)$ and $IA(i+1)-1$ correspond respectively to the beginning and the end of the i^{th} row in arrays AA and JA, *i.e.* $IA(n+1) = nnz + 1$.

$$\begin{aligned} AA &= -5 \;/\; 14 \;/\; 8 \;/\; 1 \;/\; 2 \;/\; 10 \;/\; 4 \;/\; 2 \;/\; 9 \;/\; 15 \;/\; 7, \\ JA &= 1 \;/\; 2 \;/\; 2 \;/\; 3 \;/\; 1 \;/\; 3 \;/\; 2 \;/\; 4 \;/\; 5 \;/\; 3 \;/\; 5, \\ IA &= 1 \;/\; 3 \;/\; 5 \;/\; 7 \;/\; 10 \;/\; 12 \end{aligned} \tag{2}$$

The *ELL*PACK (ELL (3)) format [16] for a $n \times m$ sparse matrix consists in storing a $m \times k$ (where k is the maximum number of non-zero values by row) dense matrix,

$COEF$, and a matrix, $JCOEF$, which stores the column indices of non-zero elements.

$$COEFF = \begin{pmatrix} -5 & 14 & 0 \\ 8 & 1 & 0 \\ 2 & 10 & 0 \\ 4 & 2 & 9 \\ 15 & 7 & 0 \end{pmatrix} ; \ JCOEFF = \begin{pmatrix} 1 & 2 & 0 \\ 2 & 3 & 0 \\ 1 & 3 & 0 \\ 2 & 4 & 5 \\ 3 & 5 & 0 \end{pmatrix} \quad (3)$$

COO format allows rows of different sizes without wasting memory space in contrary to ELL storage.

The *HYB*rid (HYB (4)) format introduced by Bell and Garland [3] is based on the ELL and COO formats. The basic idea of this format lies in choosing the number of columns of the ELL format, depending on both the context and the structure of the matrix (Bell and Garland give experimental threshold), and then to store as many coefficients of A as possible in the ELL format. All non-zero matrix elements that would result in decreasing the benefits of ELL format are stored in COO format.

ELL part:

COO part:

$$COEFF = \begin{pmatrix} -5 & 14 \\ 8 & 1 \\ 2 & 10 \\ 4 & 2 \\ 15 & 7 \end{pmatrix} ; JCOEFF = \begin{pmatrix} 1 & 2 \\ 2 & 3 \\ 1 & 3 \\ 2 & 4 \\ 3 & 5 \end{pmatrix} ; \qquad AA = [9] \ ; \ JA = [5] \ ; \ IA = [4] \quad (4)$$

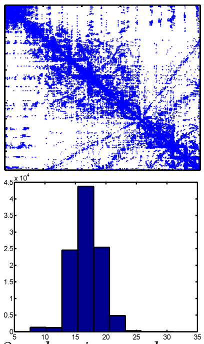
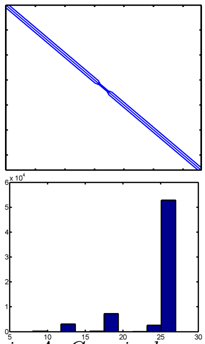
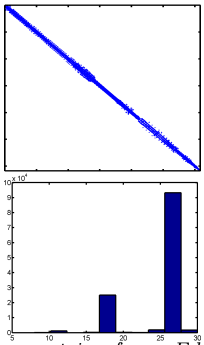
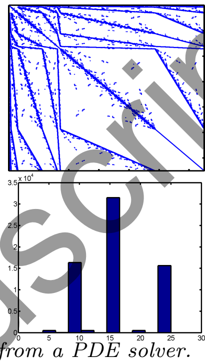
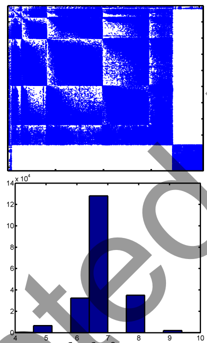
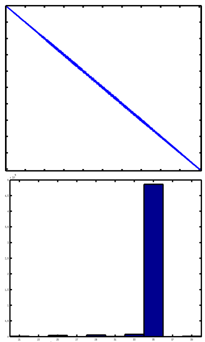
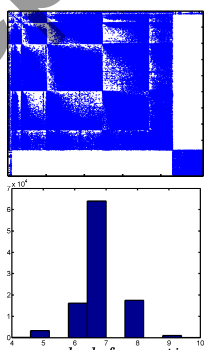
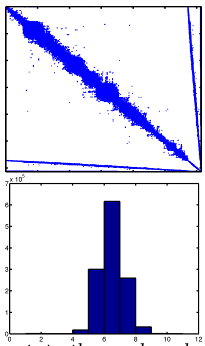
3.6.2 Visualisation of matrices for the test cases

We have decided to use a set of realistic engineering matrices from University of Florida Sparse Matrix Collection [12] in order to illustrate the performance, robustness and accuracy of our implementation in comparison to some other libraries. In the repository of the University of Florida, one can find matrices arising from different scientific fields such as mechanics, finance, electromagnetism, meteorology, etc., and from various computer methods such as finite difference, finite volume, finite element, etc. Table 2 gives the detailed characteristics of these matrices. For each matrix, the quantities h, nz, $density$, $max\,row$, $bandwidth$, nz/h and $nz/hstddev$ denote respectively the size of the matrix, the number of non-zero elements, the density, *i.e.* the number of non-zero elements divided by the total number of elements, the maximum row density, the upper bandwidth equal to the lower bandwidth for a symmetric matrix, the mean row density and the standard deviation of nz/h. For each matrix in Table 2, the upper figure shows sparse matrix pattern and the lower one gives the histogram of the distribution of the non-zero elements.

3.6.3 Matrix vector multiplication

The parallel algorithm for computing the sparse matrix-vector product is based on the algorithm described in references [3] and [4]. The very basic, intuitive approach called CSR scalar version, consists in simply assigning a thread of execution to each matrix row. This results in a straightforward CUDA implementation. However the consequence of such design is that the access into storage arrays of the matrix is not contiguous [3], which is not efficient. In fact, the manner in which threads within a warp accesses to the coefficients results with serious drawbacks.

Table 2. Characteristics of matrices for the test cases

2cubes_sphere h = 101492 nz = 1647264 density = 0.016 max row = 100407 bandwidth = 31 nz/h = 16.230 nz/h stddev = 2.654  *FEM, electromagnetism, 2 cubes in a sphere. Evan Um, Geophysics, Stanford.*	**qa8fm** h = 66127 nz = 1660579 density = 0.038 max row = 1048 bandwidth = 27 nz/h = 25.112 nz/h stddev = 4.183  *3D acoustic FE mass matrix. A. Cunningham, Vibro-Acoustic Sciences Inc.*
cfd2 h = 123440 nz = 3085406 density = 0.020 max row = 4332 bandwidth = 30 nz/h = 24.995 nz/h stddev = 3.888  *CFD, symmetric pressure matrix, from Ed Rothberg, Silicon Graphics, Inc.*	**Dubcova2** h = 65025 nz = 1030225 density = 0.024 max row = 64820 bandwidth = 25 nz/h = 15.844 nz/h stddev = 5.762  *Univ. Texas at El Paso, from a PDE solver.*
thermomech_dM h = 204316 nz = 1423116 density = 0.003 max row = 204276 bandwidth = 10 nz/h = 6.965 nz/h stddev = 0.715  *FEM problem, temperature and deformation of a steel cylinder.*	**af_shell8** h = 504855 nz = 17579155 density = 0.007 max row = 4909 bandwidth = 40 nz/h = 34.820 nz/h stddev = 1.285  *Olaf Schenk, Univ. Basel: AutoForm Eng. GmbH, Zurich. sheet metal forming.*
thermomech_TK h = 102158 nz = 711558 density = 0.007 max row = 102138 bandwidth = 10 nz/h = 6.965 nz/h stddev = 0.715  *FEM problem, temperature and deformation of a steel cylinder.*	**thermal2** h = 1228045 nz = 8580313 density = 0.001 max row = 1226000 bandwidth = 11 nz/h = 6.987 nz/h stddev = 0.811  *Unstructured FEM, steady state thermal problem. Dani Schmid, Univ. Oslo.*

The algorithm used in this paper is close to the CSR vector version presented in [3]. Threads are grouped into warps of size 32 (or 16, *i.e.* a half warp, or 8, *i.e.* a half half warp). We assume a warp with 32 threads, so on each cycle, every thread in the warp executes the same instruction (*SIMD*). A row of the matrix is thus worked on by a warp performing 32 (number of threads per warp) computations per iteration. The main step of the computation is to sum, in an array of shared memory (more faster than global memory), the values computed by the threads of the warp. This allow us to avoid the main constraint of the CSR scalar approach, due to the recovered contiguous memory access into storage array and column indices. Although having optimised the access to matrix elements, the sparse matrix-vector computation remains handicapped, because of the non contiguous access to the values of the dense vector. The access pattern to the elements of this vector is closely related to the distribution of the non-zero values in the matrix presented in Table 2. The computation times exhibited in Table 3, clearly corroborate this dependency.

The code snippet in Listing 8 illustrates the very basic strategy for computing the gridification for CSR, ELL and HYB sparse matrix-vector implementation in order to make possible the auto-tuning of the gridification. On GPU, the required number of blocks in a kernel execution is calculated as follows:

$$\frac{(numb\_rows \times n\_th\_warp) + numb\_th\_block - 1}{numb\_th\_block}$$

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.

```
//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 8 Gridification strategy

This dynamic control strategy consists in calculating the required number of blocks used in a grid for computing the SpMV, with a given number of threads per block, a number of threads to be used in a warp and the size of the matrix. A row of the matrix is associated with a warp. The proposed optimization for accelerating the calculation is based on the full utilisation of the *performance of warps*. Therefore, the number of warps used depends on the number of threads in a block. Moreover, depending on the structure and size of the matrix, a row can be handled by several warps, which can be located in different blocks. By this way, we focus on looking up a grid layout that provides contiguous memory access, because the performance is strongly impacted by the memory accesses patterns as discussed in Section 2.

The idea behind the Alinea SpMV ELL algorithm consists in assigning to each thread the treatment of a row of the matrix. The matrix is stored in an array arranged in column-increasing order that ensure continuous and linear access to memory. If the number of non-zero values per row of a matrix alters a lot, *i.e.* if the variance of the density of non-zero values per row is high, this data storage format turns of to be very suitable. Bell and Garland [4] show that this leads to fewer problems when the matrices are well structured, but in cases where the distribution is non balanced, the algorithm loses its effectiveness.

The tests have been performed on large sparse matrices with different size and properties of the structure. Unless otherwise stated, we define the *default gridification* with 256 threads per block and 8 used threads per warp.

The sparse matrix-vector multiplication total running time (in milliseconds) for different implementation, that is Alinea, CUSPARSE and Cusp library for CSR storage, is detailed in Table 3 in columns two to five. Results for CSR format in Table 3 clearly

Table 3. Running time of SpMV (ms)

Matrix	**Alinea CPU CSR**	**Alinea GPU CSR**	**CUSPARSE GPU CSR**	**Cusp GPU CSR**	**Alinea GPU ELL**	**Alinea GPU HYB**
`2cubes_sphere`	14.717	0.943	0.988	0.939	0.904	1.044
`cfd2`	24.303	1.187	2.433	1.465	0.958	1.189
`thermomech_dM`	15.906	0.757	1.009	0.813	0.685	0.866
`thermomech_TK`	7.827	0.465	0.509	0.484	0.445	0.668
`qa8fm`	12.852	0.633	1.319	0.788	0.529	0.395
`Dubcova2`	8.326	0.488	0.670	0.498	0.477	0.663
`af_shell8`	131.043	5.798	9.302	7.103	4.512	3.838
`finan512`	5.555	0.360	0.452	0.362	0.485	0.517

show the good performance of the GPU compared to the CPU. We can also see in this table that our implementation Alinea outperforms CUSPARSE and Cusp libraries for double precision computations.

Now we focus on the question how to improve our library by taking into account the features of the GPU and the specification of the problem. We propose to vary the parameters used in the gridification strategy given in Listing 8 for CSR. The results with the best gridification are reported in Table 4. Comparison between the computational

Table 4. GPU execution time of CSR SpMV (ms) with auto-tuning

Matrix	**Alinea**	**Grid Alinea** $< ntb, tw >$	**Alinea T.**	**Grid Alinea T.** $< ntb, tw >$
`2cubes_sphere`	0.943	$< 256, 8 >$	0.928	$< 64, 8 >$
`cfd2`	1.187	$< 256, 8 >$	1.141	$< 64, 8 >$
`thermomech_dM`	0.757	$< 256, 8 >$	0.751	$< 128, 8 >$
`thermomech_TK`	0.465	$< 256, 8 >$	0.460	$< 64, 8 >$
`qa8fm`	0.633	$< 256, 8 >$	0.609	$< 64, 8 >$
`thermal2`	4.158	$< 256, 8 >$	4.158	$< 256, 8 >$

times of the Alinea library with the default gridification and the Alinea library with auto-tuning of the gridification are given in Table 4. The second column presents the SpMV GPU time in milliseconds by considering the *default gridification* indicated in the third column. The fourth column reports the GPU execution time in milliseconds of Alinea with the tuned grid ($< ntb, tw >$) given in the fifth column, where ntb and tw describe respectively the number of threads per block and the number of threads used per warp.

Our experiments gave the results that are encouraging to continue investigation of the best gridification approach. The best results are obtained when we consider a gridification with a half half-warp, *i.e.* 8 threads in a warp. In some rare cases, when our implementation is not the best (Table 3), by tuning the gridification, we obtain better results. For example, CUSPARSE is better than Alinea for the `2cubes_sphere` matrix by considering 256 threads per block and a *half-half warp* used, but Alinea becomes better than CUSPARSE for 64 threads per block and 8 threads used per warp. The gridification tuning, by optimizing the number of threads per block and the number of thread per warp depending on the matrix size, definitely increases the performance.

Knowing that the gridification strongly impacts the performance of our algorithms, as demonstrated with CSR matrices, we now focus on the question which storage format results in the best performances for Alinea library. The results are pointed out in Table 3 in the two last columns. As we can see in column six and seven of Table 3, the ELL format is very effective when the matrix has nearly uniform distribution of non-zero values in the rows, see nz/h in Table 2. This is the case when the standard deviation of the number of non-zero values per row is low. However, if a sparse matrix has a full line, this ELL format may take more space than the matrix itself. In this case the ELL format loses all its advantages. When we deal with a matrix with high standard deviation of the non-zero values per row, *i.e.* the matrix is poorly structured, COO format is better almost in every case. If the matrix is moderately structured, ELL format treats efficiently the "healthy" part and COO supports the rest. When the number of zeros per row is almost constant then the matrix is well structured, and the ELL format shows its full advantages. Furthermore, it is worthwhile to choose the ELL format rather than the CSR format even if the standard deviation of the number of values per row is not uniform, because it its memory-efficient when the average row density is close to the greatest row density.

According to the obtained results, the format to use for computing sparse matrix vector product should be carefully selected to fit the matrix sparse pattern characteristics, especially the row density of non-zero entries. The results clearly show that the linear algebra operations on the GPU device are faster than on the CPU host.

4. Iterative Krylov methods

Solving linear systems represents an indispensable step in numerical methods such as finite difference method, finite element method, etc. Generally, these methods require the solution of large size sparse linear systems. When the linear systems are so large that we hit memory constraint for direct solvers, iterative Krylov methods [39], [2] are preferred for such cases. In addition, it is easier to design parallel implementation for these iterative methods. To do it effectively one needs the efficient implementation of basic linear algebra operations as the ones presented in the previous section.

The chosen procedure for solving linear equations strongly impacts the efficiency of numerical methods [8], [9]. In this section we explain the implementation of Krylov algorithms on GPU clusters environments [19], [31], [42], and collect different results in order to compare both CPU and GPU solvers, and the impact of data structures chosen to store matrices for CPU and GPU cases.

Data transfers (sending and receiving) between host and device are not negligible when dealing with GPU computing. It is essential to minimize data dependencies between CPU and GPU. In our implementation we propose to sent all input data from host to device just once, before starting the iterative routine. Nevertheless, each iteration of iterative Krylov algorithm requires more than one calculation of dot product (or norm) that implies data copy from device to the host.

To illustrate the implementation of the solvers algorithms, we will first present the conjugate gradient method (CG), probably the most widely used Krylov algorithms for symetric positive definite matrix.

4.1 *Krylov subspace methods*

Given an initial guess x_0 to the linear system

$$Ax = b, \tag{5}$$

a general *projection method* seeks an approximate solution x_i from an affine subspace $x_0 + K_i$ of dimension i by imposing the Petrov-Galerkin condition:

$$b - Ax_i \perp \mathcal{L}_i \tag{6}$$

where $\mathcal{L}_i$ is another subspace of dimension i. A Krylov subspace method is to finding out which the subspace $\mathcal{K}_i$ is the Krylov subspace

$$\mathcal{K}_i(A, r_0) = \text{span}\,\{r_0, Ar_0, A^2r_0, \ldots, A^{i-1}r_0\}. \tag{7}$$

in which $r_0 = b - Ax_0$. Krylov methods vary in the choice of $\mathcal{L}_i$ and $\mathcal{K}_i$.

4.2 *Conjugate gradient (CG)*

The conjugate gradient (CG) algorithm is an iterative method for solving linear systems of type $Ax = b$ with A symmetric positive definite matrix. The conjugate gradient method is a particular instance of the Krylov subspace method with $\mathcal{L}_i = \mathcal{K}_i = \mathcal{K}_i(A, r_0)$ when the matrix is symmetric positive definite.

The proposed conjugate gradient algorithm optimised implementation is based on the minimization of unnecessary data transfers between the CPU and the GPU, and it also takes care of CPU management (copy, etc.).

In Algorithm 1, we can see that typical operations of CG are those of scalar-vector multiplications, Daxpy operations, sparse matrix-vector products (SpMV), dot products and Euclidean norms. Iterative Krylov methods performance depends on the matrix conditioning, and efficient preconditioning techniques must be used to ensure fast convergence such as ILU factorization [1], [15], domain decomposition methods [37], [41], [26], which involves various interface conditions [20], either based on a continuous optimization [24], [25], [13], [23], [21] using approximation of Steklov-Poincaré operators or an algebraic optimization [27], [28], [29], [30], [22] using approximation of Schur complement matrix [38], etc. All the mentioned preconditioning techniques are very efficient and are mandatory to achieve good performance. Despite these preconditioning strategies are implemented in our Alinea library [10], it is not easy to integrate them with the native Cusp library, what is necessary for objective comparison of both libraries!

As a consequence, in this work we decided to use the basic diagonal preconditionner, which is available in all libraries (native Cusp and Alinea). For the sake of simplicity, we chose a preconditioning matrix M easy to compute and to invert. In this way, we adopted M as the A diagonal which provides a relatively good preconditioning in most cases. Our implementation, computes the inverse of the diagonal on CPU and sends it to the GPU outside the iterative routine. Then preconditioning step is done on GPU with scalar-vector multiplication operation.

The iterative Krylov algorithm has been carried out on the host but all computing steps (dot, norm, Daxpy, matrix-vector product) are performed on the device. At the end of the algorithm, the vector result is copied from the device to the host. In our library, we implement the generic algorithm template and then only specialize operations according to architecture (CPU or GPU). As a consequence, both CPU and GPU codes

Algorithm 1 Conjugate gradient algorithm

Require: A CSR symmetric positive definite matrix, rhs the right-hand side, it_max the maximum number of iteration, eps the residual threshold and x the solution.

```
r ← A * x // r := Ax_0 (initial guess)
r ← rhs − q // r := b - Ax_0 (initial residual)
norm_r0 ← ||r||
if norm_r0 = 0 then
    norm_r0 ← 1
end if
first ← true
is_converged ← false
iter ← 0
while iter < it_max && !is_converged do
    if preconditionner then
        z ← M^{-1} * r // M the preconditionner
    else
        z ← r // copy r to z
    end if
    rho ← < r, z >
    if first then
        first ← false
    else
        β ← rho/rho_1
        z ← z + β * p
    end if
    Move(z, p) // mark z dead and "reallocate" p
    Ap ← A * p
    sigma ← < p, Ap >
    α ← rho/sigma
    x ← x + α * p
    r ← r − α * Ap
    rho_1 ← rho
    norm_r ← rho/norm_r0
    if norm_r <= eps then
        is_converged ← true
    end if
    iter ← iter + 1
end while
```

are similar except that the GPU one performs operations on the graphics card by calling the associated kernel. At every conjugate gradient iteration, the preconditioning step is applied in order to improve the convergence rate of the method. The process is similar for the all other Krylov methods presented in this paper.

All experimental results of iterative Krylov methods presented here are obtained for the residual tolerance threshold 1×10^{-6}, an initial guess equal to zero and the right-hand side vector filled with ones. The maximum number of iterations is fixed to 30000 for all considered Krylov algorithms. The execution times in seconds, of the preconditioned conjugate gradient (P-CG) algorithm with Cusp and Alinea libraries respectively for CSR format, are reported in columns three to five in Table 5. The times given in the table corresponds to the global running time, which includes the communication time between the CPU and the GPU. As illustrated by this table, with respect to the native Cusp

Table 5. Execution time of P-CG (seconds)

Matrix	#iter.	Alinea CPU CSR	Alinea GPU CSR	Cusp GPU CSR	Alinea GPU ELL	Alinea GPU HYB
2cubes_sphere	24	0.13	0.04	0.04	0.04	0.04
cfd2	2818	38.62	5.23	11.79	10.75	8.98
thermomech_dM	12	0.08	0.02	0.02	0.02	0.02
thermomech_TK	13226	42.55	14.32	18.7	18.32	15.32
qa8fm	29	0.11	0.04	0.04	0.04	0.03
Dubcova2	168	0.41	0.15	0.18	0.22	0.17
af_shell8	2815	93.3	21.2	18.00	13.84	13.56
finan512	15	0.03	0.01	0.02	0.02	0.02
thermal2	4453	198.49	32.16	26.45	30.8	27.46

iterative algorithm (column five) Alinea still outperforms Cusp library for CSR format for GPU case (column four). The performance of preconditioned Conjugate Gradient on GPU for ELL and HYB formats is illustrated in columns six and seven in Table 5. The comparison proves better performance for HYB format. In Table 6 we report the

Table 6. Running time of P-CG (seconds) with auto-tuning

Matrix	#iter	Alinea T.	Grid Alinea T. $< ntb, tw >$
qa8fm	29	0.04	$< 64, 8 >$
2cubes_sphere	24	0.04	$< 64, 8 >$
thermomech_TK	17039	21.72	$< 64, 8 >$
cfd2	5938	12.05	$< 64, 8 >$
thermomech_dM	12	0.03	$< 128, 8 >$
thermal2	4552	32.27	$< 256, 8 >$

running time for preconditioned Conjugate Gradient on GPU by considering the best gridification given in Table 4 that ensure effective sparse matrix-vector multiplication. Other Krylov methods of practical interest, such as P-tfQMR, P-BiCGCR, P-GCR(50), and P-BiCGStab, are presented in the next section.

4.3 *Numerical results for a selection of Krylov methods*

In this part we report numerical experiments of presented Krylov method, performed on GPU device.

Table 7 gives respectively the execution time in seconds for CSR format of preconditioned transpose-free Quasi Minimal Residual (P-tfQMR), preconditioned Bi-Conjugate Gradient Conjugate Residual method (P-BiCGCR) and preconditioned Generalized Conjugate Residual method (P-GCR) with restart parameter equal to 50. Table 8 presents the execution time in seconds for the preconditioned Bi-Conjugate Gradient Stabilized (P-BiCGStab) for CSR, ELL and HYB formats. Table 9 illustrates the execution time in seconds for CSR format for the preconditioned Bi-Conjugate Gradient Stabilized (L) (P-BiCGStabl) algorithm, where L denotes the stablized parameter.

The collected numerical results show that solving linear systems is less efficient than computing a simple SpMV, because in Krylov methods several other operations takes place. Anyway, as we can see in Table 9 when the parameter l varies for the P-BiCGStabl method, the number of iterations decreases and so decreases the computation time. It should be also noted that Alinea gives encouraging results for solving linear systems as seen from these tables.

Table 7. Iterations number and execution time (in seconds) for P-GCR, P-BiCGCR and P-tfQMR algorithms for CSR matrix format

	Matrix	#iter.	Time
P-tfQMR	`2cubes_sphere`	15	0.05
	`cfd2`	5284	19.53
	`thermomech_dM`	6	0.03
	`thermomech_TK`	90	0.19
	`qa8fm`	25	0.06
	`Dubcova2`	191	0.38
	`af_shell8`	30000	448.78
	`finan512`	9	0.02
	`thermal2`	112	1.68
P-BiCGCR	`2cubes_sphere`	23	0.04
	`cfd2`	4664	8.56
	`thermomech_dM`	12	0.02
	`thermomech_TK`	14373	15.92
	`qa8fm`	28	0.03
	`Dubcova2`	165	0.17
	`af_shell8`	2298	16.8
	`finan512`	15	0.02
	`thermal2`	4151	31.62
P-GCR, restart=50	`2cubes_sphere`	13	0.04
	`cfd2`	30000	176.38
	`thermomech_dM`	23	0.09
	`thermomech_TK`	30000	123.75
	`qa8fm`	50	0.22
	`Dubcova2`	703	2.74
	`af_shell8`	723	13.96
	`finan512`	15	0.03
	`thermal2`	30000	441.37

Table 8. Execution time of P-BiCGStab (seconds)

Matrix	CSR		ELL		HYB	
	#iter.	Time	#iter.	Time	#iter.	Time
`2cubes_sphere`	15	0.05	15	0.05	15	0.06
`cfd2`	6285	24.98	5466	15.65	5026	17.69
`thermomech_dM`	6	0.02	6	0.02	6	0.02
`thermomech_TK`	30000	64.8	30000	61.73	30000	88.06
`qa8fm`	30000	80.37	30000	61.25	30000	61.41
`Dubcova2`	114	0.25	114	0.23	114	0.31
`af_shell8`	2540	41.32	3014	32.86	2842	31.09
`finan512`	10	0.02	15	0.05	10	0.03
`thermal2`	4006	54.29	30000	380.7	30000	346.16

5. Parallel iterative Krylov methods

In this part of the paper, we propose to analyse parallel iterative Krylov methods with specific techniques such as sub-structuring. The influence of application of these techniques on the computation time of parallel processing is illustrated for the case of parallel conjugate gradient methods.

5.1 *Iterative solution*

Among all iterative methods that can be used to solve a symmetric positive definite linear system, the conjugate gradient (CG) algorithm has the advantage of being efficient and very simple to implement [39]. This algorithm consists in the minimization of the distance of the iterative solution to the exact solution on the Krylov subspaces

$$V_m = Vect\{v_1, Kv_1, \cdots, K^{m-1}v_1\}$$

Table 9. Execution time of P-BiCGStabl (seconds) for CSR format

Matrix	#iter.	Time	#iter.	Time	#iter.	Time
	L = 1		**L = 2**		**L = 3**	
2cubes_sphere	16	0.07	7	0.04	5	0.04
cfd2	5950	19.44	3287	22.05	1705	17.68
thermomech_dM	7	0.02	4	0.02	2	0.03
thermomech_TK	20	0.04	18	0.07	36	0.21
qa8fm	37	0.1	6	0.02	12	0.08
Dubcova2	115	0.22	57	0.22	39	0.21
af_shell8	2795	37.88	987	27.14	678	28.42
finan512	10	0.02	5	0.01	3	0.02
thermal2	4535	59.25	1969	53.03	1171	49.05
	L = 4		**L = 5**		**L = 6**	
2cubes_sphere	3	0.04	3	0.05	2	0.04
cfd2	1251	17.81	692	12.85	590	13.4
thermomech_dM	2	0.03	2	0.04	1	0.02
thermomech_TK	21	0.17	6	0.06	8	0.1
qa8fm	2	0.02	7	0.07	5	0.07
Dubcova2	29	0.23	25	0.25	20	0.26
af_shell8	421	24.2	418	30.32	330	29.26
finan512	2	0.02	2	0.03	2	0.02
thermal2	1070	62.23	775	58.22	671	62.56
	L = 7		**L = 8**		**L = 9**	
2cubes_sphere	2	0.05	2	0.06	2	0.07
cfd2	587	16.07	450	14.5	457	17.08
thermomech_dM	1	0.02	1	0.03	1	0.04
thermomech_TK	5	0.08	3	0.06	8	0.17
qa8fm	1	0.02	1	0.02	1	0.02
Dubcova2	17	0.27	15	0.28	13	0.28
af_shell8	254	26.7	244	29.85	233	32.71
finan512	2	0.03	1	0.02	1	0.02
thermal2	553	62.31	483	64.44	384	59.51

of increasing dimension, for the scalar product associated to the matrix K. The initial vector v_1 is equal to g^0, with g^0 the initial residual $g^0 = Kx^0 - b$.

Knowing at iteration p the approximate solution x^p, the residual $g^p = Kx^p - b$ and the descent direction vector w^p, the calculations at $p+1$ iteration step of the CG algorithm go as follows:

- compute the matrix vector product

$$Kw^p$$

- compute the optimal descent coefficient

$$\rho^p = -\frac{(g^p, Kw^p)}{(Kw^p, w^p)}$$

- update the solution and the residual

$$x^{p+1} = x^p + \rho^p w^p$$
$$g^{p+1} = g^p + \rho^p K w^p$$

- determine the new descent direction by orthogonalization for the scalar product associated to the matrix K, of the new gradient to the previous descent direction

$$\gamma^p = -\frac{(g^{p+1}, Kw^p)}{(Kw^p, w^p)}$$
$$w^{p+1} = g^{p+1} + \gamma^p w^p$$

where the brackets $(\circ, \circ)$ represent the Euclidean scalar product. Using the properties of orthogonality of the direction vectors, it can be shown that in exact arithmetic this algorithm converges with a number of iterations that is smaller or equal to the dimension of the matrix. At each iteration, a product by the matrix K has to be performed : this is the most time consuming operation ; the other operations are only scalar products and linear combinations of vectors.

If the CG algorithm is parallelized with a simple algebraic approach, the rows of the matrix can be split on the different processors. Each processor computes the matrix-vector product for its own rows and then calculates the linear combination for the part of the resulting vector corresponding to these rows; the same methodology is used for their contributions to the global scalar product.

Because of the repartition of the non zero coefficients in the matrix (especially if no renumbering method has been used), each processor will need the complete vector for the matrix-vector product. This implies the gathering of all parts of the vector at all processors before the computation of the matrix-vector product which involves a lot of communication.

In order to reduce the amount of transferred data, some rows' number associated with degree of freedom localized on neighboring nodes of the mesh can be attributed to each processor. In such a way the main amount of the non zero coefficients of the local sub-matrices are located near the diagonal block. Hence the matrix is quasi-diagonal per block and the communication between the processors is reduced.

This approach leads to a natural partitioning of the matrix associated with the topology of the mesh. Each matrix block associated with a processor corresponds to a set of nodes of the mesh and composes a subdomain. The matrix-vector product performed on each subdomain requires the knowledge of the coefficients of the vector in the associated subdomain as well as the ones associated to the boundary in the neighboring subdomains ; this boundary is called interface. It means that the smaller the borders of the subdomains are, the less data transfer is involved.

5.2 *Iterative sub-structuring methods*

Generally, finite element method is based on the mesh elements partitioning rather than on the degrees of freedom (dof's), specially because the global matrix is computed as an assembly of elementary matrices. The initial domain Ω is split into two non-overlapping subdomains Ω_1 and Ω_2, with a common interface Γ as shown in Figure 9. Having in mind

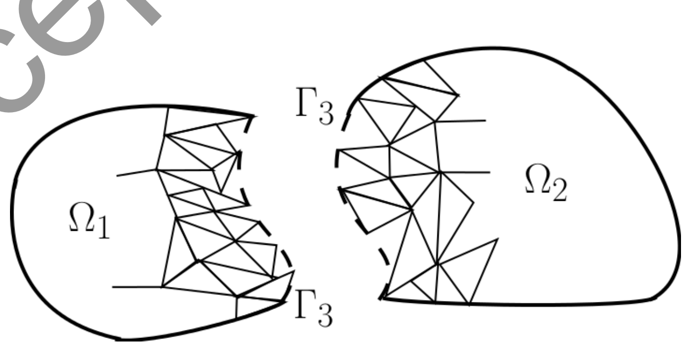


Figure 9. Two subdomains mesh with common interface

the analysis from the previous sections, if an adequate numbering of degrees of freedom (dof's) is used, the stiffness matrix of the global problem takes the following block form:

$$K = \begin{pmatrix} K_{11} & 0 & K_{13} \\ 0 & K_{22} & K_{23} \\ K_{31} & K_{32} & K_{33} \end{pmatrix} \tag{8}$$

This corresponds to the case where the set of nodes numbered 1 (resp. 2) is associated to subdomain Ω_1 (resp. Ω_2). The set of nodes numbered 3 is associated to the interface nodes between the subdomains. The unknown vector x (resp. the right hand side b) can be written as $x = (x_1, x_2, x_3)^t$ (resp. $b = (b_1, b_2, b_3)^t$) and the linear system for matrix (8) now takes the following form :

$$\begin{pmatrix} K_{11} & 0 & K_{13} \\ 0 & K_{22} & K_{23} \\ K_{31} & K_{32} & K_{33} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ b_3 \end{pmatrix} \tag{9}$$

The blocks K_{13} and K_{31} (resp. K_{23} and K_{32}) are transpose of each other and the diagonal block K_{11} and K_{22} are symmetric positive definite if the matrix K was symmetric positive definite.

If the local stiffness matrices in each subdomain are computed this leads to :

$$K_1 = \begin{pmatrix} K_{11} & K_{13} \\ K_{31} & K_{33}^{(1)} \end{pmatrix} \ , \ K_2 = \begin{pmatrix} K_{22} & K_{23} \\ K_{32} & K_{33}^{(2)} \end{pmatrix}$$

The matrices $K_{33}^{(1)}$ and $K_{33}^{(2)}$ represent the interaction matrices between the nodes on the interface obtained by integration on Ω_1 and on Ω_2. The block K_{33} is the summation of the two local contributions ($K_{33} = K_{33}^{(1)} + K_{33}^{(2)}$). This approach implies of course that the interface nodes are known by each subdomain. This means that the subdomain Ω_1 (resp. Ω_2) knows the set of nodes 1 and 3 (resp. 2 and 3).

In order to solve the problem (9) by the CG detailed in sub-section 5.1 a product of the matrix K by a descent direction vector $w = (w_1, w_2, w_3)^t$ needs to be performed at each iteration. This product can be computed in two steps :

- compute the local matrix-vector products in each subdomain :

$$\begin{pmatrix} v_1 \\ v_3^{(1)} \end{pmatrix} = \begin{pmatrix} K_{11} & K_{13} \\ K_{31} & K_{33}^{(1)} \end{pmatrix} \begin{pmatrix} w_1 \\ w_3 \end{pmatrix} \ , \ \begin{pmatrix} v_2 \\ v_3^{(2)} \end{pmatrix} = \begin{pmatrix} K_{22} & K_{23} \\ K_{32} & K_{33}^{(2)} \end{pmatrix} \begin{pmatrix} w_2 \\ w_3 \end{pmatrix}$$

- assemble the vector values along the interfaces :

$$v_3 = v_3^{(1)} + v_3^{(2)}$$

which gives the vector $v = (v_1, v_2, v_3)^t$. Note that the last step requires that each processor sends the local result restricted to the interface to all other neighboring processors, receives the contribution from the neighboring subdomains and assembles the internal and external results.

At sparse matrix-vector product step, in order to assemble the contributions of different subdomains, each process in charge of a subdomain must have the description of its interfaces. If a subdomain Ω_j has several neighboring subdomains, we denote Γ_{ji} the interface between Ω_j and Ω_i such as described in Figure 10. An interface is both described

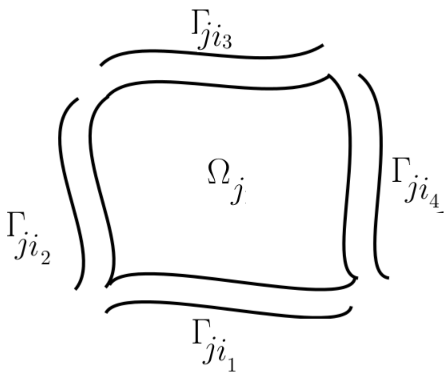


Figure 10. Interface description

by the number of its neighboring subdomain and the list of equations associated with the nodes of the interface. The contributions of different subdomains at the interface are performed in sparse matrix-vector product in two steps:

(1) For each neighboring subdomain, *receive* the values of the local vector $y = Ax$ for all interface equations in a vector and *send* it to process in charge of the neighboring subdomain.

  for $s = 1$ **to** *number_of_neighboring* **do**
    for $j = 1$ **to** n_s **do**
      $temp_s(j) = y(list_s(j))$
    end for
    Send $temp_s$ to neighbour(s)
  end for

(2) For each neighbour subdomain, *receive* the contributions of the vector which stored the matrix-vector product at the interface and add them to the components of vector y corresponding to the interface equations.

  for $s = 1$ **to** *number_of_neighboring* **do**
    Receive $temp_s$ from neighbour(s)
    for $j = 1$ **to** n_s **do**
      $y(list_s(j)) = y(list_s(j)) + temp_s(j)$
    end for
  end for

If an equation belongs to multiple interfaces, the corresponding coefficient of the local vector $y = Ax$ is sent to all neighboring subdomains interfaces to which it belongs.

The procedure for any number of subdomains consists in applying for each interface the same procedure as for two subdomains.

The sub-structuring approach leads to a natural parallelization of the iterative CG algorithm. The matrix-vector product involves local independent matrix-vector products which can be computed in parallel on different processors. Once the local contributions are obtained, they are assembled along the interface. This method is well suited for distributed memory where one processor is associated to one subdomain for example: the processor 1 can be associated to the subdomain Ω_1 and the processor 2 to the subdomain Ω_2. The most important part of the data is located in the local matrix K_1 (resp. K_2) which arise from the finite element discretization in the subdomain Ω_1 (resp. Ω_2). At each iteration, the processor 1 (resp. 2) computes the local matrix-vector product by the descent direction vector and add the local result to the contribution from the processor 2 (resp. 1). the new descent direction can then be determined. This last step involves local linear combinations and scalar products which require in addition to the local computation the assembly of the local contributions between the network. Of course a weight of the interface nodes must be determined in order to avoid multiple contributions in the scalar product.

The only difference between the sequential mono-domain code and the parallel multi-domain code lies in the two steps of data exchange. The first one helps to assemble the result of the matrix-vector product on the interface. For that, each processor needs only to know the list of the nodes of their interfaces and the number of the neighboring subdomains. The second one consists in exchanging data on the network to assemble the local scalar product. For that, the partitioning has no influence, each subdomain does the same work. These operations are usually realized with some message passing library functions. Finally, the generalization of this method in the case of more than two subdomains is immediate.

5.3 *Numerical results for parallel sub-structuring CG method*

In this section, we report numerical results of parallel sub-structuring Conjugate Gradient method. For each simulation case, we ran 10 trials with residual tolerance threshold 1×10^{-6}, an initial guess equals to zero and a right-hand side vector filled with ones.

Table 10. Degree of freedom and non-zero values of the matrix of each subdomain

#subdoms	#subdom's number	2cubes_sphere dof	2cubes_sphere nnz	af_shell8 dof	af_shell8 nnz	cfd2 dof	cfd2 nnz	Dubcova2 dof	Dubcova2 nnz
1	(0)	101492	874378	504855	9046865	123440	1605669	65025	547625
2	(0)	52016	826126	253221	8810853	62601	1548093	32919	519515
2	(1)	51491	836151	252674	8794432	62156	1553740	32884	518240
4	(0)	26617	422713	126597	4398547	32707	787341	16716	262504
4	(1)	26563	423375	126767	4406873	31179	775703	16641	260889
4	(2)	26998	426534	126876	4406834	31744	782066	16647	261091
4	(3)	26966	416932	127080	4414982	31433	783833	16603	261111
8	(0)	13759	215379	63467	2201715	16601	397151	8435	131319
8	(1)	14038	219588	63723	2206843	16037	395369	8601	133729
8	(2)	13768	214322	63665	2209225	16150	396526	8530	132948
8	(3)	13731	213449	63668	2205260	15992	394800	8451	131319
8	(4)	13928	211472	64104	2219056	16299	380831	8558	132986
8	(5)	13869	210473	63602	2206164	16660	408534	8442	131728
8	(6)	14164	217980	63974	2215750	15724	387660	8461	131671
8	(7)	14168	218740	63422	2198874	15966	394114	8590	133900

#subdoms	#subdom's number	finan512 dof	finan512 nnz	qa8fm dof	qa8fm nnz	thermomech_dM dof	thermomech_dM nnz	thermomech_TK dof	thermomech_TK nnz
1	(0)	74752	335872	66127	863353	204316	813716	102158	406858
2	(0)	37436	298814	33248	831710	102158	711558	51134	356060
2	(1)	37395	298479	33268	832512	102158	711558	51179	355965
4	(0)	18713	149273	17092	419714	51116	355948	25630	177754
4	(1)	18713	149273	17328	423986	51200	356086	25702	178598
4	(2)	18713	149273	17226	421230	51173	356313	25672	178354
4	(3)	18713	149273	17217	422921	51150	355746	25660	178392
8	(0)	9369	74649	9067	218487	25693	178523	12871	89213
8	(1)	9369	74649	9032	217326	25650	178332	12903	89315
8	(2)	9369	74649	8995	214621	25664	177978	12907	89383
8	(3)	9369	74649	8997	214955	25643	178209	13027	90187
8	(4)	9369	74649	8949	214675	25677	178115	12842	88912
8	(5)	9369	74649	8899	214329	25680	178428	12917	89491
8	(6)	9369	74649	9080	217594	25673	178385	12870	89184
8	(7)	9369	74649	9146	219042	25612	178094	12895	89097

Table 10 reports the degree of freedom (dof) and non-zero values (nnz) of the matrix of each subdomain. The first column gives the number of subdomains, followed by the subdomain number in brackets. Comparison of the results obtained for CPU clusters and GPU clusters is given in Table 11 for sparse CSR preconditioned Conjugate Gradient (P-CG), and Table 12 represents the corresponding speed-up. To obtain these results,

Table 11. Execution time for parallel sub-structuring CG (seconds) for CSR format

Matrix	#iter.	1CPU	2CPUs	4CPUs	8CPUs	GPU	2GPUs	4GPUs	8GPUs
2cubes_sphere	24	0.386	0.209	0.124	0.125	0.026	0.047	0.065	0.113
af_shell8	2815	374.356	198.998	110.662	107.668	14.549	23.422	18.814	21.385
cfd2	2818	71.078	38.114	21.657	22.111	3.730	5.904	8.280	12.186
Dubcova2	168	1.776	0.926	0.540	0.850	0.128	0.364	0.405	0.640
finan512	15	0.117	0.121	0.067	0.145	0.017	0.063	0.071	0.120
qa8fm	29	0.418	0.235	0.198	0.168	0.023	0.099	0.137	0.115
thermomech_dM	12	0.313	0.176	0.100	0.091	0.016	0.037	0.072	0.106
thermomech_TK	13226	141.359	74.423	41.979	40.888	13.214	22.877	32.587	51.528

Table 12. Speed-Up of parallel sub-structuring CG (seconds) for CSR format

Matrix	#iter.	1CPU	2CPUs	4CPUs	8CPUs	GPU	2GPUs	4GPUs	8GPUs
2cubes_sphere	24	1.0	1.8	3.1	3.1	15.1	8.2	5.9	3.4
af_shell8	2815	1.0	1.9	3.4	3.5	25.7	16.0	19.9	17.5
cfd2	2818	1.0	1.9	3.3	3.2	19.1	12.0	8.6	5.8
Dubcova2	168	1.0	1.9	3.3	2.1	13.8	4.9	4.4	2.8
finan512	15	1.0	1.0	1.8	0.8	7.0	1.9	1.6	1.0
qa8fm	29	1.0	1.8	2.1	2.5	18.0	4.2	3.0	3.6
thermomech_dM	12	1.0	1.8	3.1	3.5	19.0	8.5	4.4	3.0
thermomech_TK	13226	1.0	1.9	3.4	3.5	10.7	6.2	4.3	2.7

we execute 100 times the same algorithm with the same input data. The performance of our parallel P-CG solver for various sparse matrices are reported in columns five to six for CPU and in columns eight to ten. Both sequential CPU and GPU are given in columns three and seven, respectively. The first column lists the test case name and the second column gives the number of of iterations for the P-CG solver. For the presented set of test case matrices the parallel sub-structuring P-CG gives satisfactory results. Nevertheless, we can remark that relative gains of some kinds of matrices increase when matrix bandwidths decrease and matrix sizes increase. As already mentioned the gains of using GPU are the most visible for large size matrices. Moreover, when the number of subdomains increases, the GPU code lost performance. This is mainly due to the decreasing number of degrees of freedom within each subdomain and more particularly by the transfert between the GPU and the host when data exchanges are performed at the interfaces between the subdomains. This degrades the overall execution time, as we can see in Table 12; in fact MPI calls cannot be performed inside the GPU.

6. Conclusions

In this paper, we have investigated the best way to compute efficiently linear algebra operations in order to implement effective iterative Krylov methods for solving large and sparse linear systems on Graphics Processing Unit for double precision computation. We briefly mentioned in the introduction the challenge of GPU computing in the field of computational science.

We give an overview of GPU programming model and look closer at the hardware specifications in order to understand the specific aspects of graphics card computations. We have taken care to optimize CPU operations, data transfers and memory management. We have compared the performances of *Alinea*, our implementation against existing scientific linear algebra libraries for GPU.

The experiments have been performed on a set of large size sparse matrices for several engineering and scientific problems from University of Florida Sparse Matrix Collection. Numerical experiments clearly illustrate that the GPU operations are significantly more efficient than CPU. The presented results also show that data matrix storage format has a strong impact on the execution time obtained on GPU. After presenting a comparison of the most classical formats, we analyze the behaviour of linear algebra operations upon these formats.

In order to ensure even better efficiency, after a brief presentation of the gridification principles, auto-tuning of the gridification upon the GPU architecture is performed in order to obtain faster implementation. Auto-tuning proves that gridification strongly impacts the performance of algorithms.

In this way, we describe how to implement efficient iterative Krylov methods on GPU by using gathered experience. Different Krylov methods are then developed and compared for different data matrix storage formats. The experiments, performed for several matrices, confirm the performance of our proposed implementation. The results demonstrate the robustness, competitiveness and efficiency of our own implementation compared to the existing libraries.

Acknowledgements

The authors acknowledge partial financial support from OpenGPU project (2010-2012), and CRESTA project (2011-2014), and the Cuda Research Center at Ecole Centrale Paris, France for the computer time used during this long-term trend.

References

[1] J.I. Aliaga, M. Bollhofer, A.F. Martien, and E.S. Quintana-Orti, *Parallelization of multilevel ILU preconditioners on distributed-memory multiprocessors*, in *PARA (1)*, *Lecture Notes in Computer Science*, vol. 7133, Springer, 2010, pp. 162–172.

[2] H. Anzt, V. Heuveline, and B. Rocker, *Mixed Precision Iterative Refinement Methods for Linear Systems: Convergence Analysis Based on Krylov Subspace Methods.*, in *PARA (2)*, *Lecture Notes in Computer Science*, vol. 7134, Springer, 2010, pp. 237–247.

[3] N. Bell and M. Garland, *Efficient sparse matrix-vector multiplication on CUDA*, Nvidia Technical Report NVR-2008-004, Nvidia Corporation, 2008.

[4] N. Bell and M. Garland, *Implementing sparse matrix-vector multiplication on throughput-oriented processors*, in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis (SC'09)*, Portland, Oregon, ACM, New York, NY, USA, 2009, pp. 1–11.

[5] N. Bell and M. Garland, *Cusp: Generic parallel algorithms for sparse matrix and graph computations* (2012), available on line at: http://cusplibrary.github.io/ (accessed on May 30, 2014).

[6] J. Bolz, I. Farmer, E. Grinspun, and P. Schröoder, *Sparse matrix solvers on the gpu: conjugate gradients and multigrid*, ACM Trans. Graph. 22 (2003), pp. 917–924.

[7] A.K. Cheik Ahamed and F. Magoulès, *Fast sparse matrix-vector multiplication on graphics processing unit for finite element analysis*, in *High Performance Computing and Communication 2012 IEEE 9th International Conference on Embedded Software and Systems (HPCC-ICESS), 2012 IEEE 14th International Conference on*, IEEE Computer Society, 2012, pp. 1307–1314.

[8] A.K. Cheik Ahamed and F. Magoulès, *Iterative Methods for Sparse Linear Systems on Graphics Processing Unit*, in *High Performance Computing and Communication 2012 IEEE 9th International Conference on Embedded Software and Systems (HPCC-ICESS), 2012 IEEE 14th International Conference on*, june, IEEE Computer Society, 2012, pp. 836 –842.

[9] A.K. Cheik Ahamed and F. Magoulès, *Iterative Krylov Methods for Gravity Problems on Graphics Processing Unit*, in *Distributed Computing and Applications to Business, Engineering Science (DCABES), 2013 12th International Symposium on*, IEEE Computer Society, 2013, pp. 16–20.

[10] A.K. Cheik Ahamed and F. Magoulès, *Schwarz Method with Two-Sided Transmission Conditions for the Gravity Equations on Graphics Processing Unit*, in *Proceedings of the 12th International Symposium on Distributed Computing and Applications to Business, Engineering and Science (DCABES), Kingston, London, UK, September 2nd-4th, 2013*, **IEEE Computer Society**, 2013.

[11] A. Davidson, Y. Zhang, and J.D. Owens, *An Auto-tuned Method for Solving Large Tridiagonal Systems on the GPU*, in *Proceedings of the 25th IEEE International Parallel and Distributed Processing Symposium*, Anchorage, Alaska, May, IEEE, 2011.

[12] T.A. Davis and Y. Hu, *The University of Florida sparse matrix collection*, ACM Trans. Math. Softw. 38 (2011), pp. 1–25.
[13] M.J. Gander, *Optimized schwarz methods*, SIAM 44 (2006), pp. 699–731.
[14] IEEE 754: Standard for Binary Floating-Point Arithmetic (2008), available on line at: http://grouper.ieee.org/groups/754/ (accessed on May 30, 2014).
[15] C. Janna, M. Ferronato, and G. Gambolati, *A block FSAI-ILU parallel preconditioner for symmetric positive definite linear systems.*, SIAM J. Scientific Computing 32 (2010), pp. 2468–2484.
[16] D.R. Kincaid, T.C. Oppe, and D.M. Young, *ITPACKV 2D user's guide*, Report CNA-232, University of Texas at Austin Department of Mathematics Austin, TX USA, 1989.
[17] J. Krüger and R. Westermann, *Linear algebra operators for GPU implementation of numerical algorithms*, ACM Transactions on Graphics 22 (2003), pp. 908–916.
[18] N. Li, B. Suchomel, D. Osei-Kuffuor, R. Li, and Y. Saad (2010), available on line at: www-users.cs.umn.edu/ saad/software/ITSOL/index.html (accessed on May 30, 2014).
[19] R. Li and Y. Saad, *GPU-accelerated preconditioned iterative linear solvers* (2010).
[20] Y. Maday and F. Magoulès, *Absorbing interface conditions for domain decomposition methods: a general presentation*, Computer Methods in Applied Mechanics and Engineering 195 (2006), pp. 3880–3900.
[21] Y. Maday and F. Magoulès, *Improved ad hoc interface conditions for Schwarz solution procedure tuned to highly heterogeneous media*, Applied Mathematical Modelling 30 (2006), pp. 731–743.
[22] Y. Maday and F. Magoulès, *Optimal convergence properties of the FETI domain decomposition method*, International Journal for Numerical Methods in Fluids 55 (2007), pp. 1–14.
[23] Y. Maday and F. Magoulès, *Optimized schwarz methods without overlap for highly heterogeneous media*, Computer Methods in Applied Mechanics and Engineering 196 (2007), pp. 1541–1553.
[24] F. Magoulès, P. Iványi, and B. Topping, *Convergence analysis of Schwarz methods without overlap for the Helmholtz equation*, Computers and Structures 82 (2004), pp. 1835–1847.
[25] F. Magoulès, P. Iványi, and B. Topping, *Non-overlapping Schwarz methods with optimized transmission conditions for the Helmholtz equation*, Computer Methods in Applied Mechanics and Engineering 193 (2004), pp. 4797–4818.
[26] F. Magoulès and F.X. Roux, *Lagrangian formulation of domain decomposition methods: a unified theory*, Applied Mathematical Modelling 30 (2006), pp. 593–615.
[27] F. Magoulès, F.X. Roux, and L. Series, *Algebraic way to derive absorbing boundary conditions for the Helmholtz equation*, Journal of Computational Acoustics 13 (2005), pp. 433–454.
[28] F. Magoulès, F.X. Roux, and L. Series, *Algebraic approximation of Dirichlet-to-Neumann maps for the equations of linear elasticity*, Computer Methods in Applied Mechanics and Engineering 195 (2006), pp. 3742–3759.
[29] F. Magoulès, F.X. Roux, and L. Series, *Algebraic Dirichlet-to-Neumann mapping for linear elasticity problems with extreme contrasts in the coefficients*, Applied Mathematical Modelling 30 (2006), pp. 702–713.
[30] F. Magoulès, F.X. Roux, and L. Series, *Algebraic approach to absorbing boundary conditions for the Helmholtz equation*, International Journal of Computer Mathematics 84 (2007), pp. 231–240.
[31] K.K. Matam and K. Kothapalli, *Accelerating Sparse Matrix Vector Multiplication in Iterative Methods Using GPU*, in *ICPP*, IEEE, 2011, pp. 612–621.
[32] Nvidia Corporation, *CUDA toolkit 4.0, CUBLAS Library* (2011), available on line at: http://developer.nvidia.com/cuda-toolkit-40 (accessed on May 30, 2014).
[33] Nvidia Corporation, *CUDA Toolkit 4.0, CUSPARSE Library* (2011), available on line at: http://developer.nvidia.com/cuda-toolkit-40 (accessed on May 30, 2014).
[34] Nvidia Corporation, *CUDA Toolkit Reference Manual*, 4th ed. (2011), available on line at: http://developer.nvidia.com/cuda-toolkit-40 (accessed on May 30, 2014).
[35] T. Oberhuber, A. Suzuki, and J. Vacata, *New row-grouped csr format for storing the sparse matrices on gpu with implementation in cuda*, CoRR abs/1012.2270 (2010).
[36] OpenCL (2010), available on line at: http://www.khronos.org/opencl/ (accessed on May 30, 2014).
[37] A. Quarteroni and A. Valli, *Domain Decomposition Methods for Partial Differential Equations*, Oxford University Press, Oxford, UK (1999).
[38] 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*, Springer-Verlag, Haidelberg, Lecture Notes in Computational Science and Engineering (LNCSE), edited by R. Kornhuber, R. Hoppe, J. Périaux, O. Pironneau, O. Widlund and J. Xu, 2005.
[39] Y. Saad, *Iterative methods for sparse linear systems, 2nd ed.*, SIAM (2003).
[40] C.J. Thompson, S. Hahn, and M. Oskin, *Using modern graphics architectures for general-purpose computing: a framework and analysis*, in *Proceedings of the 35th annual ACM/IEEE international*

symposium on Microarchitecture, MICRO 35, Istanbul, Turkey, IEEE Computer Society Press, Los Alamitos, CA, USA, 2002, pp. 306–317.
[41] A. Toselli and O. Widlund, *Domain decomposition methods*, Computational Mathematics 34 (2004).
[42] A.H.E. Zein and A.P. Rendell, *Generating optimal CUDA sparse matrix-vector product implementations for evolving GPU hardware*, Concurrency and Computation: Practice and Experience 24 (2012), pp. 3–13.