An efficient multi-algorithms sparse linear solver for GPUs

Thomas JOST [a], Sylvain CONTASSOT-VIVIER [a,b] and Stéphane VIALLE [a,c]
[a] *AlGorille INRIA Project Team, Nancy, France*
[b] *Université Henri Poincaré, Nancy, France*
[c] *SUPELEC, Metz, France*

Abstract. We present a new sparse linear solver for GPUs. It is designed to work with structured sparse matrices where all the non-zeros are on a few diagonals. Several iterative algorithms are implemented, both on CPU and GPU. The GPU code is designed to be fast yet simple to read and understand. It aims to be as accurate as possible, even on chips that do not support double-precision floating-point arithmetic. Several benchmarks show that GPU algorithms are much faster than their CPU counterpart while their accuracy is satisfying.

Keywords. GPU, CUDA, linear solver, sparse matrix, Jacobi algorithm, Gauss-Seidel algorithm, biconjugate gradient algorithm

Motivations and objectives

Nowadays, many research areas rely on simulation. As a consequence, software simulations are expected to continuously become faster, more accurate, and to be able to handle new, bigger, more complex problems. A widely adopted solution to help building such software is to use supercomputers and grids where a large and scalable number of powerful processors are available. This can however be very expensive, both for the hardware and for the energy required to run a large number of processors.

Recent GPUs can be used for general purpose computing using dedicated libraries such as CUDA (for NVIDIA cards) or OpenCL. They are based on a SIMD architecture that makes it possible to handle a very large number of hardware threads concurrently, and can be used for various purposes, including scientific computing. Thanks to the video game industry, GPUs have become very powerful yet quite cheap. They are now able to handle massively parallel problems that require a huge amount of data, which makes them interesting for modern software simulations. Moreover, it is possible that a GPU cluster both computes faster and consumes less energy than a CPU cluster [1]. This makes GPU clusters very attractive for High Performance Computing.

Many physical problems can be modeled by a system of PDEs. A common solution to simulate the phenomenon is to discretize and linearize these equations, and then to solve the resulting linear system using a linear solver. However, solving a huge linear system is not an easy task and it often requires a lot of computational power. GPUs may therefore be of great interest for solving linear systems quickly and efficiently as soon as a SIMD algorithm can be deduced from the sequential scheme.

This article introduces a new linear solver for GPUs, specially designed to solve sparse structured systems. It aims to work on all CUDA-compatible cards, but it was developed using an entry-level board which does not support double-precision arithmetic.

Floating-point precision and memory accesses were therefore given a special attention in order to be as fast and accurate as possible. Finally, several iterative algorithms have been implemented to solve linear problems both on CPU and GPU.

This work takes place in a larger project concerning the adaptation of asynchronous iterative algorithms on a cluster of GPUs. Asynchronous algorithms are a very general class of parallel algorithms in which iterations and communications are asynchronous and overlap. This way, all idle times between iterations are suppressed, which often leads to faster execution times even if more iterations may be necessary. More details can be found in [2]. The two levels of communications present in a cluster of GPUs, the ones between a GPU and its hosting CPU, and the ones between CPUs, should yield representative communication times according to computation times. This represents a particularly good context of use of asynchronous algorithms and this is why we aim at developing asynchronous algorithms on clusters of GPUs.

1. Related works

Several different libraries already exist in the field of linear algebra on GPUs. Most of them use CUBLAS, an implementation of the BLAS on CUDA made by NVIDIA that is shipped with the CUDA SDK [4]. CUBLAS only provides with the most basic linear algebra operations, which are then used by other libraries to develop more or less complex algorithms.

There are many different libraries for dense linear algebra. Some of them seem to be quite interesting, like MAGMA [5], which "aims to develop a dense linear algebra library similar to LAPACK but for heterogeneous/hybrid architectures, starting with current 'Multicore+GPU' systems". Others have more restrictive goals, such as cudaztec [6] (GMRES solver) or GPUmatrix [7] (solver using LU, QR or Cholesky decomposition). However, none of these is interesting for us: they focus on dense linear algebra whereas we deal with sparse structured matrices. Besides, many of these libraries are ongoing research works: no source code is available at the moment for MAGMA, cudaztec seems to have very poor performances (GPU code slower than CPU code), and GPUmatrix does not compile on our Linux machines. Another interesting library is *Concurrent Number Cruncher*, a general sparse linear solver [8]. It is efficient and powerful, but restricted to *general* sparse matrices, while we need support for *structured* sparse matrices. Even though CNC would work fine for our problem, it would be very sub-optimal: for instance memory access patterns would be unaligned or uncoalesced, and would be slow.

So, implementing a new linear solver is the only way to get a code working at top efficiency on GPUs with our structured sparse matrices.

2. Choice of the linear method

For readers not familiar with linear solvers, a description of the ones mentioned here can be found in [3]. The most obvious constraint in the choice of the linear solver to develop is that it must contain a sufficient amount of potential parallelism. This discards intrinsically sequential methods such as Gauss-Seidel. Also, as we are in the context of sparse matrices and according to the limited memory available on GPU boards, itera-

tive methods will be preferred to direct ones, which are more memory consuming. So, methods like GMRES are also discarded. Finally, the linear solver we aim at developing on the GPU is to be used in general scientific problems, most of them being non-linear. Hence, that solver will be a key part of more general solvers. In that context, it must be able to handle as many kinds of matrices as possible. So, simple methods like Jacobi are interesting for matrices with absolute diagonal dominance but cannot be used with more complex ones. Also, the conjugate gradient is interesting for sparse systems but only works on symmetric positive-definite matrices. In the end, a more general method satisfying all the GPU constraints is the biconjugate gradient method. This method is a generalization of the conjugate gradient. However, for symmetric problems, it requires twice more computations than the simpler conjugate version. In the same way, the Jacobi method will be faster with the adequate matrices. So, an interesting approach is to develop several linear solvers on GPU and compare them to each other. This will allow us to decide whether it is sufficient to always use the same solver for any kind of matrices or if it is interesting to dynamically choose the most adapted solver among a small set according to the type of matrices to process.

3. Design and implementation

According to the limited length of the paper and the focus of our work, we only present here the design of the GPU version of the most complex method chosen in our study, i.e. the biconjugate gradient algorithm, which is known to have a very wide range of applications. Although the other methods discussed above are not detailed here, they have been implemented and their results in terms of accuracy and performance are presented in section 4.

3.1. Sequential algorithm

As said above, the biconjugate gradient, given on the right, is an extension of the conjugate gradient algorithm. It produces two mutually orthogonal sequences of vectors in place of the orthogonal sequence of residuals generated in the conjugate gradient algorithm. This implies a modification of the behavior of the iterative process and especially that the minimization is no more ensured. Moreover, it requires the use of the transpose of the matrix to compute one of the two sequences. However, although few theoretical results are known on this method, it has been proved that it is comparable to GMRES for non-symmetric matrices [9].

Biconjugate Gradient algorithm

Compute $r^0 \leftarrow b - A.x^0$
Compute $\widetilde{r}^0$ ($= r^0$ for example)
$i = 1$
repeat
  $\rho_{i-1} \leftarrow r^{i-1}.\widetilde{r}^{i-1}$
  if $\rho_{i-1} = 0$ **then**
    method fails
  end if
  if $i = 1$ **then**
    $p^i \leftarrow r^{i-1}$
    $\widetilde{p}^i \leftarrow \widetilde{r}^{i-1}$
  else
    $\beta \leftarrow \frac{\rho_{i-1}}{\rho_{i-2}}$
    $p^i \leftarrow r^{i-1} + \beta p^{i-1}$
    $\widetilde{p}^i \leftarrow \widetilde{r}^{i-1} + \beta \widetilde{p}^{i-1}$
  end if
  $q^i \leftarrow A.p^i$
  $\widetilde{q}^i \leftarrow A^t.\widetilde{p}^i$
  $\alpha \leftarrow \frac{\rho_{i-1}}{\widetilde{p}^i.q^i}$
  $x^i \leftarrow x^{i-1} + \alpha p^i$
  $r^i \leftarrow r^{i-1} - \alpha q^i$
  $\widetilde{r}^i \leftarrow \widetilde{r}^{i-1} - \alpha \widetilde{q}^i$
  $i \leftarrow i + 1$
until stopping criteria is reached

3.2. Sparse matrix storage

As in the scope of our study the matrix A is sparse, we had to choose a storage scheme providing a good compromise between memory saving and structure regularity. The first point is obvious in the context of the GPU use as the amount of available memory is strictly limited. The second point comes from the way the memory accesses are performed on the GPU board. In order to obtain efficient accesses to the memory from numerous concurrent threads, those accesses have to be as regular as possible.

So, the structure used is similar to the DIA scheme described in [10] and consists in storing only the diagonals containing non-zero values. Hence, the matrix is represented by a two-dimensional array of the non-zero diagonals in which each row contains one diagonal of A. The ordering of the diagonals in that array follows the horizontal order from left to right. An additional one-dimensional array is needed to get the link between a row number in the array and its corresponding diagonal in A.

3.3. GPU scheme

According to the hardware design and functioning scheme of the GPU, the parallel algorithm to produce must follow the SIMT programming model (single-instruction, multiple-thread), which is a slightly more flexible variant of the SIMD paradigm [4]. Our GPU version of the biconjugate gradient algorithm is not fully deported on the GPU. In fact, the main loop controlling the iterations is kept on the CPU but all the computations inside it are performed on the GPU, using either standard CUBLAS functions for classical operations such as dot products, or specific kernels for the more complicated operations. So, the algorithm is divided into three main steps:

- The first one is the initialization of the solver. It mainly consists in allocating the memory on the GPU and transferring the data, the matrix A and the vector b.
- The second step corresponds to the iterative computations of the algorithm. That part is fully detailed below.
- Once the algorithm has converged, the third and last step takes place, which performs the transfer of the result vector x from the GPU to the CPU and the memory deallocation on the GPU.

The middle part of that scheme is decomposed in order to be easily and efficiently implemented on the GPU. The `cublasSdot` function is used to compute the scalar products taking place in the computations of ρ and α. The copies of vectors r and $\widetilde{r}$ respectively in p and $\widetilde{p}$ at the first iteration are directly performed with the `cudaMemcpy` function. Finally, four specific kernels have been designed to implement the computations of:

- p and $\widetilde{p}$: kernel `update_p(Real` β`)`
  as the vectors involved in those computations are already in the GPU memory, the only parameter required by this function is the β value.
- q and $\widetilde{q}$: kernel `update_q()`
  for the same reason as above, that function takes no parameter.
- x, r and $\widetilde{r}$: kernel `update_xr(Real` α`)`
  that function takes the value of α involved in the three updates.
- residual error : kernel `delta(Real* newDelta)`
  the residual is the maximal absolute value in the vector $b - A.x$.

All those kernels are mapped in the same way on the GPU, that is in one-dimensional blocks of 256 threads. This is usually an optimal number of threads per block when scheduling a large number of blocks. In all those computations performed on the GPU, the only transfer between GPU and CPU is the residual error (only one scalar real) which is required on the CPU for the control of the main iterative loop.

As programming in CUDA is quite a recent exercise and since it is not yet an usual task to write a kernel from a high level description of an algorithm, we provide below the two most complex kernels codes.

```
__global__ void update_q() { // ------------------------- Kernel update_q
  // Compute q <- A.p and qt <- A.pt. Each thread: 1 element of q and qt.
  extern __shared__ int diags[];
  int i = blockIdx.x * BLOCKSIZE + threadIdx.x;
  Real qi = 0., qti = 0.;

  // Fetch the diagonals <-> rows array to shared memory (faster access)
  if (threadIdx.x < g.diags)
    diags[threadIdx.x] = g.la[threadIdx.x];
  __syncthreads();

  if (i < g.size) {       // Avoids out-ranging indices in the last block
    for (int d=0; d < g.diags; d++) {     // For each diagonal...
      int k = diags[d];                   // Row d is diagonal k in A
      int kt = g.size - k;                // and diag. kt in transpose(A)
      if (kt >= g.size) kt -= g.size;

      int j = i + k;                      // Column of the element in A
      if (j >= g.size) j -= g.size;
      int jt = i + kt;                    // Column in transpose(A)
      if (jt >= g.size) jt -= g.size;

      Real a = g.ad[d*g.row_size + i];    // Coalesced read
      Real at = g.ad[d*g.row_size + jt];  // Uncoalesced read
      Real pj = g.p[j];                   // Uncoalesced read
      Real ptj = g.pt[jt];                // Uncoalesced read

      qi += a * pj;       // Perform needed additions and multiplications
      qti += at * ptj;
    }
    g.q[i] = qi;      // Once computation is over, save qt and qti (fast
    g.qt[i] = qti;    // local registers) to the GPU global memory (slow)
  }
}

__global__ void delta(Real* dst) { // ----------------------- Kernel delta
  // Compute dst <- b-A.x. Each thread computes one element of dst.
  extern __shared__ int diags[];
  int i = blockIdx.x * BLOCKSIZE + threadIdx.x;

  // Fetch the diagonals <-> rows array to shared memory (faster access)
  if (threadIdx.x < g.diags)
    diags[threadIdx.x] = g.la[threadIdx.x];
  __syncthreads();

  if (i < g.size) {       // Avoids out-ranging indices in the last block
    Real di = g.b[i];
    for (int d=0; d < g.diags; d++) {       // For each diagonal...
      int k = diags[d];                     // Row d is diagonal k in A
      int j = i+k;                          // Column of the element in A
      if (j >= g.size) j -= g.size;

      Real x = g.x[j];                      // Uncoalesced read
      Real v = g.ad[d*g.row_size + i] * x;  // Coalesced read
      di -= v;
    }
    dst[i] = di;          // Save local register to the GPU global memory
  }
}
```

This code shows that our data structure is easy to use and well adapted to the GPU. In these kernels, the most common access scheme is implemented: we iterate over the

diagonals (i.e. the lines of the two-dimensional array called `g.ad`), and for each diagonal we iterate over all of its elements. It is then easy to deduce the corresponding (i, j) coordinates in the "actual" matrix using the diagonal identifier k which is defined as $k = j - i$. Coordinates in the transposed matrix are a bit more difficult to deduce, but this can easily be done too. Textures may then be used to read memory, thus avoiding uncoalesced memory accesses.

The actual code of our solver is a bit more complicated that this one in several ways:

- Textures are used to avoid uncoalesced memory reads. They allow faster access to the memory by hiding latency using a small local cache.
- FMAD (Fused Multiply-Add) opcodes are avoided using the `__fmul_rn()` intrinsic function. This has a small (non-measurable) performance cost, but it improves accuracy since the GPU FMAD performs aggressive rounding in the multiplication.
- Some additions are made using the Kahan summation algorithm [11] to improve precision.

These modifications make the code more difficult to read, but slightly enhance both accuracy (especially using single-precision arithmetic) and performances.

4. Experiments

According to our study, we have considered three pairs of algorithms for our benchmarks, running on GPU and CPU (mono-core only):

- Jacobi (GPU) and Gauss-Seidel (CPU). We use Gauss-Seidel on the CPU because it is faster and it requires less memory than Jacobi while having quite the same convergence domain. Therefore, it is relevant to compare these algorithms as they are respectively better suited to the tested platforms.
- Biconjugate gradient (BiCG) on CPU and GPU, as described in section 3.1.
- BiCG-Stab, a stabilized variant of the regular BiCG algorithm [12].

We first tested these algorithms on *artificial matrices*, built in a reproducible way, for sizes varying between 100 and 3,000,000 (maximum size before getting *out of memory* errors on the GPU cards); secondly we ran tests using *real matrices* from a collection of real-world sparse matrices [13]. The tests were made in single precision on a NVIDIA GeForce 8800GT card, and in double precision on a more recent NVIDIA G200 card which includes about 30 IEEE 754R double units over a total of 240 units. On GPU, the time measures include CPU-GPU data transfer time and GPU computation time. The compilers used were `nvcc 2.1` (CUDA compiler) and `gcc 4.1`.

Tests made with artificial matrices in single precision are presented in Figures 1 and 2. Here are the main results:

- GPU algorithms run faster than their CPU counterparts by a factor 20 to 25.
- The absolute error ($\|b - Ax\|_\infty$) is usually higher on GPU than on CPU, but the difference is quite low and both of them are sufficient (around 10^{-6}).

Tests made with artificial matrices in double precision have shown that double precision does not change the factor between CPU and GPU convergence times, and that error

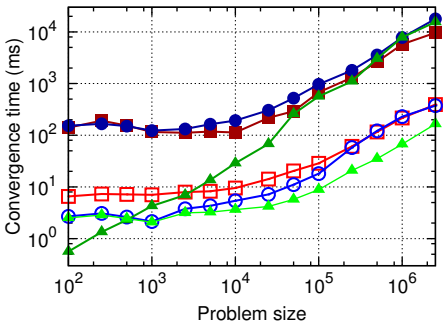


Figure 1. Convergence time in single precision

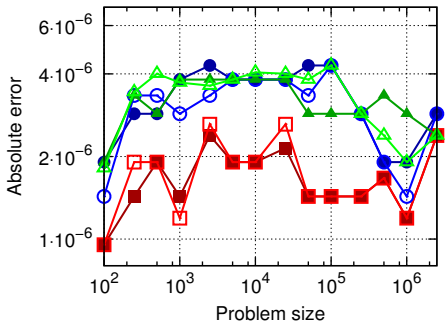
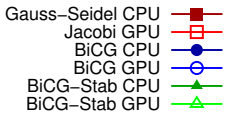


Figure 2. Absolute error in single precision



Legend for Figures 1, 2 and 3.

Measures obtained with our set of *artificial matrices*.

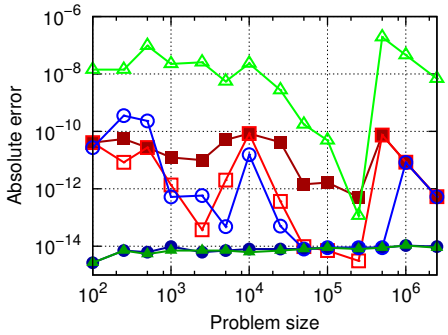


Figure 3. Absolute error in double precision

decreases compared to single precision tests. Figure 3 details the error obtained with double precision; we can observe BiCG and BiCG-Stab errors on CPU are very low (10^{-14}) and close, while corresponding GPU errors are larger (between 10^{-14} and 10^{-6}) and vary a lot. With single precision, BiCG and BiCG-Stab errors were close both on CPU and GPU. Errors of the Gauss-Seidel algorithm on CPU and of the Jacobi one on GPU vary a lot in single precision and still vary in double precision, but are not the lowest errors in double precision. Globally, we observe all errors are lower in double precision, but CPU and GPU algorithms of the same pair have really different behaviors in double precision while they have close behaviors in single precision.

Tests with real matrices, both in single precision (Table 1) and double precision (Table 2), show that BiCG and BiCG-Stab have a wider convergence domain than Jacobi/Gauss-Seidel algorithms (empty and italics cells indicate cases where the algorithm did not converge). They are therefore suitable for a greater variety of problems, which is why we chose them in the first place.

5. Conclusion

Several sparse linear solvers have been designed and implemented on CPU and GPU. The different tested methods have different complexities and applicability ranges.

The first aspect of that work was to compare the performances and accuracies of the GPU and CPU versions of those linear methods. Our experimental results clearly show that problems that only require single-precision arithmetic can be treated using a GPU

Table 1. Absolute error obtained with a set of *real matrices* in single precision

Matrix name	GS CPU	Jacobi GPU	BiCG CPU	BiCG GPU	BiCG-S CPU	BiCG-S GPU
minsurfo	7.15e-7	1.19e-6	8.34e-6	6.32e-6	2.15e-6	2.04e-5
obstclae	7.15e-7	1.19e-6	1.86e-5	7.15e-6	1.91e-6	2.96e-6
wathen100	1.91e-6		1.42e-5	1.60e-5	7.39e-6	7.74e-6
ex1			5.25e-6	4.77e-6	4.77e-6	*1.60e-1*
mcca			6.20e-6	7.63e-6	4.77e-6	*2.16e+0*
dw4096			*1.41e+2*		*4.00e-2*	

Table 2. Absolute error obtained with a set of *real matrices* in double precision

Matrix name	GS CPU	Jacobi GPU	BiCG CPU	BiCG GPU	BiCG-S CPU	BiCG-S GPU
minsurfo	1.33e-15	5.76e-11	1.24e-14	1.05e-7	6.07e-11	6.98e-9
obstclae	1.33e-15	1.06e-7	6.66e-15	4.59e-8	3.42e-9	4.61e-9
wathen100	3.55e-15	*3.02e-2*	3.11e-14	1.79e-7	1.42e-14	8.72e-8
ex1			9.33e-15	6.48e-8	9.10e-9	*6.27e-3*
mcca			1.15e-14	3.74e-8	1.29e-9	*2.55e+1*
dw4096			*3.88e+0*	*6.81e+0*	*3.07e-2*	*1.23e+5*

algorithm, which computes 20 to 25 times faster than its mono-core CPU equivalent. When double precision is required, depending on the really needed accuracy and the acceptable computation time, we have to choose between a CPU and a GPU algorithm.

Finally, comparing every GPU algorithm to each other has allowed us to point out that it would not be efficient to use only one method for all the possible matrices; building a contextual solver should thus be useful.

References

[1] L.A. Abbas-Turki, S. Vialle, B. Lapeyre and P. Mercier. *High Dimensional Pricing of Exotic European Contracts on a GPU Cluster, and Comparison to a CPU Cluster*, Second Workshop on Parallel and Distributed Computing in Finance (PDCoF 2009). 8 pages. May 29, 2009, Rome, Italy.

[2] J. Bahi, R. Couturier, K. Mazouzi and M. Salomon, *Synchronous and asynchronous solution of a 3D transport model in a grid computing environment*, Applied Mathematical Modelling, 30(7), 2006.

[3] J. M. Bahi, S. Contassot-Vivier and R. Couturier, *Parallel Iterative Algorithms: from sequential to grid computing*, Chapman & Hall/CRC, 2007. Numerical Analysis & Scientific Computing Series.

[4] NVIDIA, *CUDA SDK documentation*, 2009.

[5] M. Baboulin, J. Demmel, J. Dongarra, S. Tomov and V. Volkov, *Enhancing the performance of dense linear algebra solvers on GPUs*, Poster at Supercomputing 2008, November 18, 2008.

[6] D. Neckels, *cudaztec*, http://code.google.com/p/cudaztec/.

[7] N. Bonneel, *GPUmatrix*, http://sourceforge.net/projects/gpumatrix/.

[8] L. Buatois, G. Caumon and B. Lévy, *Concurrent number cruncher - A GPU implementation of a general sparse linear solver*, International Journal of Parallel, Emergent and Distributed Systems, to appear.

[9] R. W. Freund and N. M. Nachtigal, *QMR: a Quasi-Minimal Residual Method for Non-Hermitian Linear Systems*, Iterative Methods in Linear Algebra, 151-154. Elsevier Science Publishers. 1992.

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

[11] W. Kahan, *Further remarks on reducing truncation errors*, Comm. of the ACM, 8 (1965), p. 40.

[12] H. A. van der Vorst, *Bi-CGSTAB: A fast and smoothly converging variant of Bi-CG for the solution of nonsymmetric linear systems*, SIAM J. Sci. Comp. 13: pp. 631-644, 1992.

[13] T. A. Davis, *The University of Florida Sparse Matrix Collection*, Technical Report of the University of Florida, http://www.cise.ufl.edu/research/sparse/matrices