Iterative Solution of Linear Systems in Electromagnetics (And Not Only): Experiences with CUDA

Danilo De Donno, Alessandra Esposito,
Giuseppina Monti, and Luciano Tarricone

Department of Innovation Engineering,
University of Salento,
Via per Monteroni - 73100 - Lecce, Italy
{danilo.dedonno,alessandra.esposito,giuseppina.monti,
luciano.tarricone}@unisalento.it

Abstract. In this paper, we propose the use of graphics processing units as a low-cost and efficient solution of electromagnetic (and other) numerical problems. Based on the software platform CUDA (Compute Unified Device Architecture), a solver for unstructured sparse matrices with double precision complex data has been implemented and tested for several practical cases. Benchmark results confirm the validity of the proposed software in terms of speed-up, speed and GPU execution time.

Keywords: linear system, GPU, CUDA, biconjugate gradient, BiCG.

1 Introduction

Software simulations are continuously requested to become faster, more accurate, and able to handle new, bigger and more complex problems. Recently, thanks to the continuous impulse coming from video game industry, graphics processors (GPUs) are affirming as a valid solution to accelerate time-demanding scientific computations. This is facilitated by the publication of simple-to-use libraries such as CUDA [1] which greatly ease software implementation.

An important research effort is currently devoted to the implementation of GPU-enabled linear solvers, since modern software simulations often depend on the solution of a computationally demanding linear system. A wide range of linear solvers exist, the choice of which depends fundamentally on the properties of the system matrix. For instance, when the matrix is known to be sparse, real symmetric or complex Hermitian, an iterative solver, such as the conjugate gradient (CG) algorithm, is usually preferred [2]. The biconjugate gradient (BiCG) method is a generalization of the CG suited to handle real nonsymmetric and complex non-Hermitian matrices. This feature is a significant advantage in many areas, such as computational electromagnetics (CEM) for the analysis and design of EM structures. In this case, a key role is played by the Method of Moments (MoM) [4] which transforms the integral-differential Maxwell's equations into

M.R. Guarracino et al. (Eds.): Euro-Par 2010 Workshops, LNCS 6586, pp. 329–337, 2011.

a linear system of algebraic equations. MoM usually generates unstructured, sparse, symmetric and non-Hermitian matrices with complex coefficients. Moreover, double precision is needed in order to achieve a satisfying accuracy.

Starting from the requirements of such a category of problems, we implemented a new BiCG solver for GPUs. Indeed, available GPU-enabled iterative solvers, such as Iterative CUDA [5], CUSP [6], SpeedIT [7] and *"Concurrent Number Cruncher"* (CNC) [8], deal with real coefficient matrices, most concentrating on the CG algorithm.

Our solver is implemented in CUDA and tackles unstructured complex sparse matrices. It has been tested on matrices coming from concrete scientific problems, some being taken from well recognized matrix collections, others being generated during in-house experimentation in the area of EM circuit design.

2 Design and Implementation

2.1 CUDA Background

A NVIDIA GPU is built around an array of multiprocessors, each of which supporting up to 1024 threads. CUDA is a standard C language extension for thread-based application development on GPU. A CUDA application consists of a sequential *host* code that executes parallel programs (*kernels*) on a parallel *device* (the GPU). Kernels are SIMT (Single Instruction Multiple Thread) computations that are executed by a potentially large number of *threads* organized into a *grid* of thread *blocks*.

Great benefits are obtained when threads access a contiguous part of device memory: in this case the individual memory instructions are replaced by a single memory access (*memory coalescence*).

2.2 The BiCG Algorithm

BiCG is an extension of the CG algorithm. It produces two mutually orthogonal sequences of vectors in place of the orthogonal sequence of residuals generated in the CG algorithm.

We implemented the complex BiCG algorithm with Jacobi preconditioning, in the form presented by Jacobs in [9]. First, we define initial variables: the residual r and bi-residual $\overline{r}$, the direction vector p and the bi-direction vector $\overline{p}$, the preconditioned residual d and bi-residual $\overline{d}$, the residual error ρ. Then, assuming that b is the rigth-hand side (r.h.s) of the linear system $Ax = b$, M the Jacobi preconditioner and x_0 the initial guess of the solution, the following steps are repeated for each iteration (the asterisk denotes the complex conjugate):

1. calculate the step length parameter and form the new solution estimate

$$q_i = Ap_{i-1} \qquad \overline{q}_i = A^H \overline{p}_{i-1} \tag{1}$$

$$\alpha_i = \rho_{i-1} / p^*_{i-1} q_i \tag{2}$$

$$x_i = x_{i-1} + \alpha_i p_{i-1} \tag{3}$$

2. update residual and bi-residual, with and without preconditioning

$$r_i = r_{i-1} + \alpha_i q_i \qquad \overline{r}_i = \overline{r}_{i-1} + \alpha_i^* \overline{q}_i \tag{4}$$

$$d_i = M^{-1} r_i \qquad \overline{d}_i = M^{-1} \overline{r}_i \tag{5}$$

3. calculate the residual error ρ and bi-conjugacy coefficient β

$$\rho_i = d_i^T \overline{r}_i^* \tag{6}$$

$$\beta_i = \rho_i / \rho_{i-1} \tag{7}$$

4. update next direction and bi-direction vectors

$$p_i = d_{i-1} + \beta_i p_{i-1} \qquad \overline{p}_i = \overline{d}_i + \beta_i^* \overline{p}_{i-1} \tag{8}$$

Iteration is continued till a termination condition of the form:

$$\left\| r^i \right\|_2 / \left\| b \right\|_2 \leq \epsilon \tag{9}$$

is satisfied. Values of ϵ used in literature range from $10^{-6}/10^{-7}$ [3].

2.3 Matrix Format

There is a multitude of sparse matrix representations, each with different storage requirements, computational characteristics and methods of accessing and manipulating entries of the matrix. We focused on schemes which efficiently store matrices with arbitrary sparsity patterns. Moreover, we preferred formats suited for the GPU use, where the amount of available memory is strictly limited and memory accesses should be as regular as possible. Based on these considerations, we considered two matrix formats: Compressed Row Storage (CRS) and the hybrid (HYB) Ellpack-Coordinate format [10].

CRS is the most common data structure used to represent general sparse matrices. It makes no assumptions about the matrix sparsity and provides a compact representation. It uses three one-dimensional arrays, from where non-zero elements, column indices and pointers to the first element of each row are retrieved. HYB joins features from the so-called Ellpack (ELL) and Coordinate (COO) formats. ELL is suited for matrices whose maximum number of non-zeros per row does not substantially differ from the average; it stores the non-zero values in a dense bi-dimensional array and the corresponding column indices in a vector. COO, instead, is a very simple format which uses three vectors to store the row indices, column indices, and non-zero values of the matrix. The HYB format, proposed in [10], calculates the *typical* number of non-zeros per row and stores the majority of matrix entries in ELL and the remaining in COO.

2.4 Implementation

In the CUDA implementation of the BiCG algorithm, the main loop controlling the iterations is kept on the CPU, whilst the computations inside are performed

Table 1. Summary of BiCG functions and floating point operations (N is the matrix dimension, nnz is the number of non-zeros)

OPERATION	FORMULAS	DESCRIPTION	FLOPS
SpMV	Eq. 1	sparse matrix-vector product	$8nnz$
dot product	Eq. 2, 6	scalar product of two vectors	$8N$
e. w. product	Eq. 5	element-wise product of two vectors	$6N$
axpy	Eq. 3, 4, 8	$ax + y$ (a scalar, x and y vectors)	$8N$

on the GPU. Four kernels are in charge of the operations carried out in the BiCG main loop (see Table 1). Among them, the sparse matrix-vector product (SpMV) is the heaviest operation, even though each of the listed operations deserves being optimized on the GPU hardware. The implementation of the four kernels is shortly described below.

SpMV - this kernel implements the Bell and Garland algorithm [10], which is, at our knowledge, the best performing code currently available for solving sparse matrix-vector product on GPUs. This algorithm is avaible in the CUSP library [6] only for single and double precision real coefficients. Therefore, we adapted it in order to tackle matrices and vectors having double precision complex values and replicated the sophisticated optimization strategies, such as loop unrolling and shared memory exploitation, described in [10]. Memory accesses were optimized according to the storage format: one warp was assigned to each matrix row in the CRS format, whilst one thread was assigned to each row of the matrix in the HYB format. In addition to the matrix-vector product Ap_{i-1}, BiCG requires computing a Hermitian product $A^H\overline{p}_{i-1}$ (see equation 1). In a distributed-memory context, there will be extra communication costs associated with one of the two matrix-vector products, depending upon the storage scheme for A. To alleviate this problem, in the initialization phase we precalculate A^H at the cost of doubling the storage requirements for the matrix.

Dot product - cuBLAS complex dot function provided with CUDA is available only for single precision coefficients. Therefore, we implemented such function from scratch. Our kernel is an adaptation and generalization of the well known parallel reduction algorithm proposed by Harris et al. in [11]. Such algorithm deals with the sum of vector elements and is appreciated for its efficiency due to the adoption of advanced optimization strategies such as shared memory exploitation, loop unrolling and algorithm cascading (combine parallel and sequential reduction). The result is a function which, given as input two complex double precision arrays, provides as output their dot product, with performances aligned with those obtained by Harris.

Element-wise product and axpy - also in these cases, the cuBLAS functions provided with CUDA don't support double precision complex coefficients. Therefore, we implemented such functions from scratch by taking advantage of the CUDA cuComplex library [1] and by adopting optimization strategies

finalized to the maximization of coalesced accesses. Moreover, in order to reduce the overhead due to communication between host and device we aggregated multiple calls in a single kernel wherever possible.

In addition to the optimization strategies briefly mentioned above, we also took advantage from CUDA texture memory, which provided relevant performance gains, as it caches data spatially closed together. Texture memory is normally read from kernels by using device functions called fetches. We implemented our own fetching functions since CUDA provides them only for real single precision data. Moreover, thanks to the exploitation of the so-called CUDA built-in arrays we efficiently minimized the cost of memory accesses.

3 Experiments and Results

The GPU-enabled solver has been tested on the CUDA-compatible nVidia GeForce GTX 260 GPGPU, featuring 24 streaming processors. The code was compiled by using CUDA 2.3 with driver version 190.53. For comparison we used a serial code implemented in C and all calculations were performed by a single core of an Intel Core2 Quad CPU Q9550 @ 2.83 GHz. The CPU code was compiled by GCC 4.3.4 with the "-O3" optimization option enabled on a PC equipped with 4 GB of DDR2 RAM, Ubuntu 9.10 as the 32-bit Linux operating system and ATLAS 3.6 [14] as BLAS library.

We tested our algorithms on sparse matrices, some of which were obtained from the application of the MoM to the design of EM circuits, the remaining were taken from the *"University of Florida Sparse Matrix Collection"* [12]. In both cases, based on GPU characteristics, we maximized multiprocessor occupancy and adopted equation (9) as convergence criterion with ϵ set to 10^{-7}.

3.1 EM Matrices

As to EM matrices derived from MoM, they concern the design of branch-line couplers in microstrip technology, which are four ports devices widely adopted in microwave and millimetre-wave applications like power dividers and combiners [13]. More specifically, the analyzed layout consists of two branch-line couplers connected by means of a 360° microstrip line and operating in the 2.5-3.5 GHz frequency band (see Fig. 1). The desired sparsity pattern was obtained by making a thresholding operation which determines the number of non-zero elements while maintaining a good accuracy of the final solution (error less than 2%).

The left side of Fig. 2 shows the convergence times of the host (CPU) and device (GPU) code for different matrix sizes (N) and formats. The percentage of non-zero elements is kept to 5% of the total number of entries by thresholding. The maximum matrix size was imposed by the memory limit of the available CPU, as before thresholding the entire dense matrix had to be loaded. Table 2 lists the achieved speed-ups. They are higher when matrix dimension allows for an optimum exploitation of hardware resources.

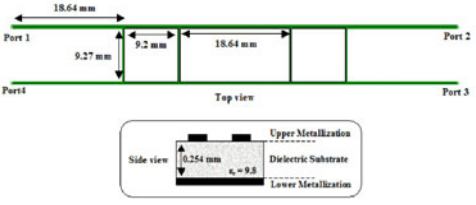


Fig. 1. Layout of the EM circuit used for testing

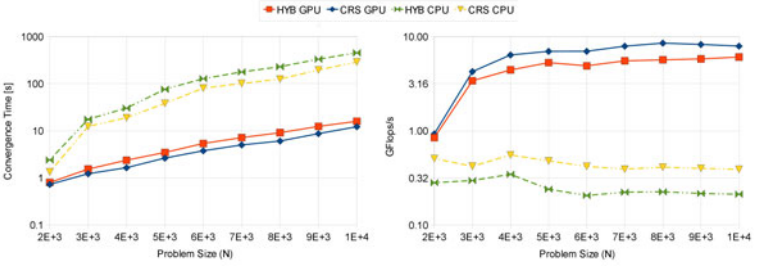


Fig. 2. BiCG convergence time (left) and performance in GFlops/s (right) for EM matrices

The right side of Fig. 2 shows the BiCG performance in terms of number of floating point operations (FLOPs) per second. For clarity, we report the equation used for calculating performance:

$$GFlops/s = \frac{C_{init} + n_{it} \cdot (2C_{spmv} + 2C_{ewp} + 2C_{dot} + 5C_{axpy})}{10^9 \cdot T_e} \tag{10}$$

where C_{init} is the number of FLOPs in the initialization phase of the algorithm, C_{spmv}, C_{ewp}, C_{dot} and C_{axpy} respectively represent FLOPs required for SpMV, element-wise product, dot product and axpy functions (see last column of Table 1), n_{it} is the number of iterations of BiCG main loop and T_e is the total execution time of the algorithm. The multiplying factors of each C term in the numerator indicates the number of corresponding operations in the BiCG main loop.

In all EM matrices we analyzed, CRS format always produces faster results because of the high variability of the non-zero number per row (see Fig. 3). It is

Table 2. Achieved speed-ups for EM matrices

Problem Size (N)	Speed-Up CRS	Speed-Up HYB
2E+3	3.01	1.84
4E+3	12.8	11.5
6E+3	23.79	21.36
8E+3	25.17	20.74
10E+3	28.57	23.69

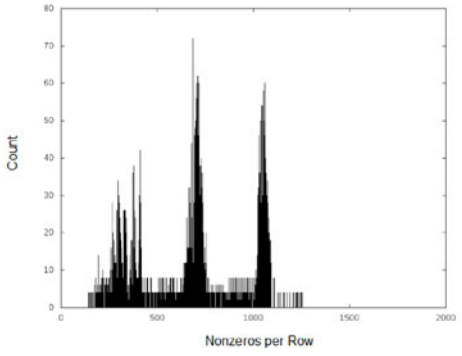


Fig. 3. Distribution of number of non-zeros per row for EM-MoM matrices

well known that convergence behavior of BiCG may be quite irregular, and the method may even break down, but in practical cases of EM circuit analysis we never observed this phenomenon in agreement with what observed in [3].

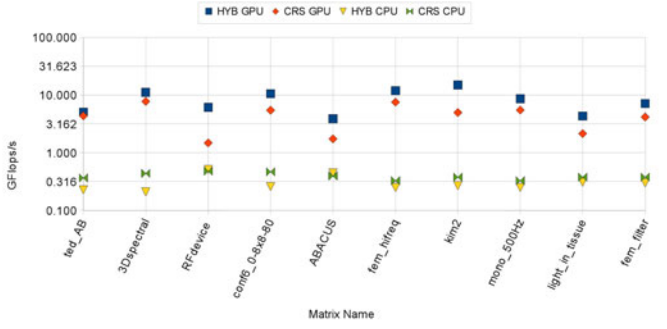
3.2 Florida Matrices

As to the *"University of Florida Collection"* [11], we identified ten complex sparse matrices (see table 3), belonging to different research areas and exhibiting different size, sparsity pattern and number of non-zeros.

The left side of Fig. 4 shows the performance obtained in the ten cases. Results are shown in terms of number of floating point operations and calculated according to equation (10). As the number of non-zeros per row was substantially constant for all the chosen matrices, the HYB format performed better than the CRS in all cases. Therefore we reported in the table on the right side of Fig. 4 only the speed-up obtained for the HYB format. As shown, the obtained speed-ups are even higher than those reached with EM matrices since we could better exploit GPU hardware resources as much bigger matrices were elaborated.

Table 3. Florida matrices used for performance testing

ID.	GROUP	NAME	SIZE	Non-zeros	Kind of problem
1	Bindel	ted_AB	10605^2	522387	thermal
2	Sinclair	3Dspectralwave2	292008^2	12935272	materials
3	Rost	RFdevice	74104^2	365580	semiconductor device
4	QCD	conf6_0-8x8-80	49152^2	1926928	chemistry
5	Puri	ABACUS_shell_md	23412^2	218484	model reduction
6	Lee	fem_hifreq_circuit	491100^2	20239237	electromagnetic
7	Kim	kim2	456976^2	11330020	2D/3D mesh
8	FreeFieldTech.	mono_500Hz	169410^2	5033796	acoustic
9	Dehghani	light_in_tissue	29282^2	406084	electromagnetic
10	Lee	fem_filter	74062^2	1731206	electromagnetic



Matrix Name	Speed-Up HYB
ted_AB	21.000
3Dspectral	52.208
RFdevice	11.545
conf6_0-8x8	40.286
ABACUS	8.733
fem_hifreq	47.625
kim2	54.563
mono_500Hz	34.000
light_in_tissue	14.091
fem_filter	23.529

Fig. 4. Performance results (left) and speed-ups (right) for Florida matrices

4 Conclusions

In this paper, the achievement of peak-performance for EM solvers through the use of the inexpensive and powerful GPUs has been investigated. Based on the requirements coming from CEM, a sparse iterative solver for GPUs has been proposed. Taking advantage from CUDA library, we implemented a BiCG algorithm which tackles unstructured sparse matrices with two different storaging schemes. It has been tested on several matrices, both from well recognized matrix collections and from in-house experimentation in the area of EM circuit design. Results in terms of speed-up, execution time and GPU speed have been provided as a validation and assessment of solver efficiency.

References

1. NVIDIA, CUDA Zone, `http://www.nvidia.com/cuda`
2. Saad, Y.: Iterative methods for sparse linear systems, 3rd edn. SIAM, Philadelphia (2000)
3. Smith, C.F., et al.: The biconjugate gradient method for electromagnetic scattering. IEEE Trans. on Antennas and Propagation 38, 938–940 (1990)
4. Harrington, R.F.: Field Computation by Moment Methods. Krieger, Melbourne (1982)
5. Iterative CUDA, `http://mathema.tician.de/software/iterative-cuda`
6. CUSP Library v.0.1, `http://code.google.com/p/cusp-library`
7. SpeedIT Tools, `http://vratis.com/speedITblog`
8. Buatois, L., et al.: Concurrent number cruncher - A GPU implementation of a general sparse linear solver. International Journal of Parallel, Emergent and Distributed Systems 24(3), 205–223 (2009)
9. Jacobs, D.A.H.: A generalization of the conjugate gradient method to solve complex systems. IMA J. Numerical Analysis. 6, 447–452 (1986)
10. Bell, N., Garland, M.: Efficient sparse matrix-vector multiplication on CUDA. NVIDIA Tech. Rep. NVR-2008-004, NVIDIA Corporation (December 2008)

11. Harris, M., Sengupta, S., Owens, J.D.: Parallel prefix sum (scan) with CUDA. In: Nguyen, H. (ed.) GPU Gems, vol. 3. Addison Wesley, Reading (August 2007)
12. Davis, T.A.: The University of Florida Sparse Matrix Collection. Tech. Report of the University of Florida, http://www.cise.ufl.edu/research/sparse/matrices
13. Mongia, R.K., et al.: RF and Microwave Coupled-Line Circuits. Artech House, Boston (1999)
14. ATLAS Library, http://math-atlas.sourceforge.net