UnConventional High Performance Computing (UCHPC) 2010 workshop at:

Euro-Par 2010

August 31st - September 3rd, 2010

Ischia - Naples, Italy

Hotel Continental Terme

Iterative Solution of Linear Systems in Electromagnetics (and not only): Experiences with CUDA

D. De Donno, A. Esposito, G. Monti, L. Tarricone

Innovation Engineering Department

University of Salento - Lecce - Italy

EMLab ELECTROMAGNETIC LAB LECCE

August 30st, 2010

UNIVERSITA' DEL SALENTO

Outline

- Background on CEM issues
- Motivations and objectives
- Design and implementation
- Experimental results
- Conclusions and ongoing work

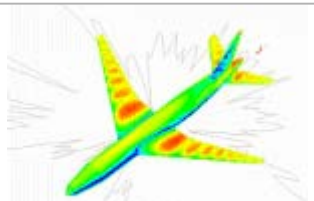
Background

One of the fundamental steps of numerical computing is the ability to solve linear systems:

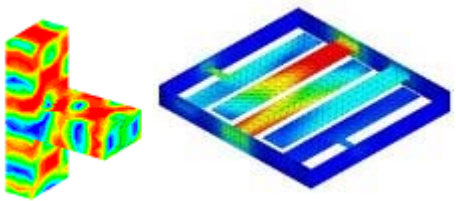
$$Ax = b$$

These systems arise very frequently in scientific computing, from finite difference or finite element approximations to partial differential equations.

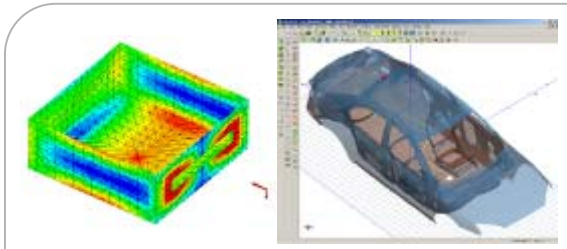
For example, in *Computational ElectroMagnetics* (CEM) the process of modeling the interaction of electromagnetic fields with physical objects and the environment give rise to linear systems with large number of unknowns.



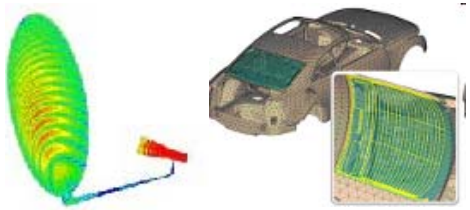
Scattering problems



RF components and 3D EM cicruits



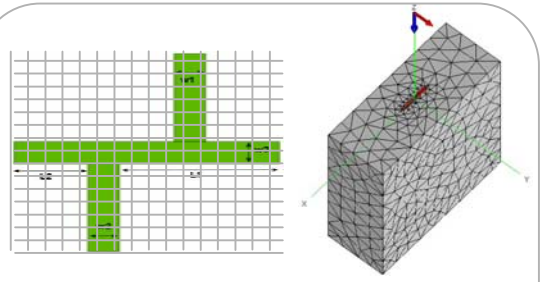
EMC analysis



Antenna design

Method of Moments (MoM) linear systems

In CEM a key role is played by the *Method of Moments* (MoM) which transforms the integral-differential Maxwell's equations into a linear system of algebraic equations.



Object discretization

MoM-based EM simulator

Solve linear system

$$Z \cdot I = V$$

Z is the MoM impedance matrix containing the ***complex*** reaction terms between basis functions.

In ***large*** EM problems Z can be reduced to an ***unstructured*** and significantly ***sparse*** matrix without affecting the numerical accuracy.

ITERATIVE SOLVERS (i.e. CG, BiCG) are preferred in these cases.

Objective

We implemented a ***Bi-Conjugate Gradient*** **(BiCG)** iterative solver for GPUs. It tackles ***unstructured sparse matrices*** with ***double precision complex data***.

Recently, cheap and powerful *graphics processors* (GPU) are emerging as a valide alternative to supercomputers and computational grids.

Our implementation takes advantage from CUDA, a standard C language extension for parallel application development on NVIDIA GPUs.

A CUDA application consists of SPMD computations (*kernels*) performed by *threads* running in parallel on the GPU *streaming multiprocessors* (SMs).

The BiCG algorithm (1/2)

BiCG is a generalization of CG (*Conjugate Gradient*) method.

CG method
- real symmetric matrices
- complex Hermitian matrices

BiCG method
- real non-symmetric matrices
- complex non-Hermitian matrices

In the initialization phase of BiCG, the following variables are defined:

$r_0 = b - Ax_0$	$\bar{r}_0 = r_0^*$	***Residual and bi-residual***
$p_0 = r_0$	$\bar{p}_0 = p_0^*$	***Direction and bi-direction***
$d_0 = M^{-1} \cdot r_0$	$\bar{d}_0 = M^{-1} \cdot \underline{r}_0^*$	***Pre-conditioned residual and bi-residual***
$\rho_0 = d^T \cdot \underline{r}_0^*$		***Initial residual error***

The BiCG algorithm (2a/2)

In the BiCG main loop, the following steps are repeated for each iteration:

1. *Calculate the step lenght parameter and form the new solution estimate.*

STEP 1

$$q_i = A \cdot p_{i-1}$$

$$\overline{q}_i = A^H \cdot \overline{p}_{i-1}$$

$$\alpha_i = \frac{\rho_{i-1}}{p_{i-1}^* \cdot q_i}$$

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

The BiCG algorithm (2b/2)

In the BiCG main loop, the following steps are repeated for each iteration:

2. *Update residual and bi-residual, with and without preconditioning.*

STEP 2

$$r_i = r_{i-1} + \alpha_i \cdot q_i$$

$$\overline{r}_i = \overline{r}_{i-1} + \alpha_i^* \cdot \overline{q}_i$$

$$d_i = M^{-1} \cdot r_i$$

$$\overline{d}_i = M^{-1} \cdot \overline{r}_i$$

The BiCG algorithm (2c/2)

In the BiCG main loop, the following steps are repeated for each iteration:

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

STEP 3

$$\rho_i = d_i^T \cdot \bar{r}_i^*$$

$$\beta_i = \frac{\rho_i}{\rho_{i-1}}$$

The BiCG algorithm (2d/2)

In the BiCG main loop, the following steps are repeated for each iteration:

4. *Update next direction and bi-direction vectors.*

STEP 4

$$p_i = d_i + \beta_i \cdot p_{i-1}$$

$$\overline{p}_i = \overline{d}_i + \beta_i^* \cdot \overline{p}_{i-1}$$

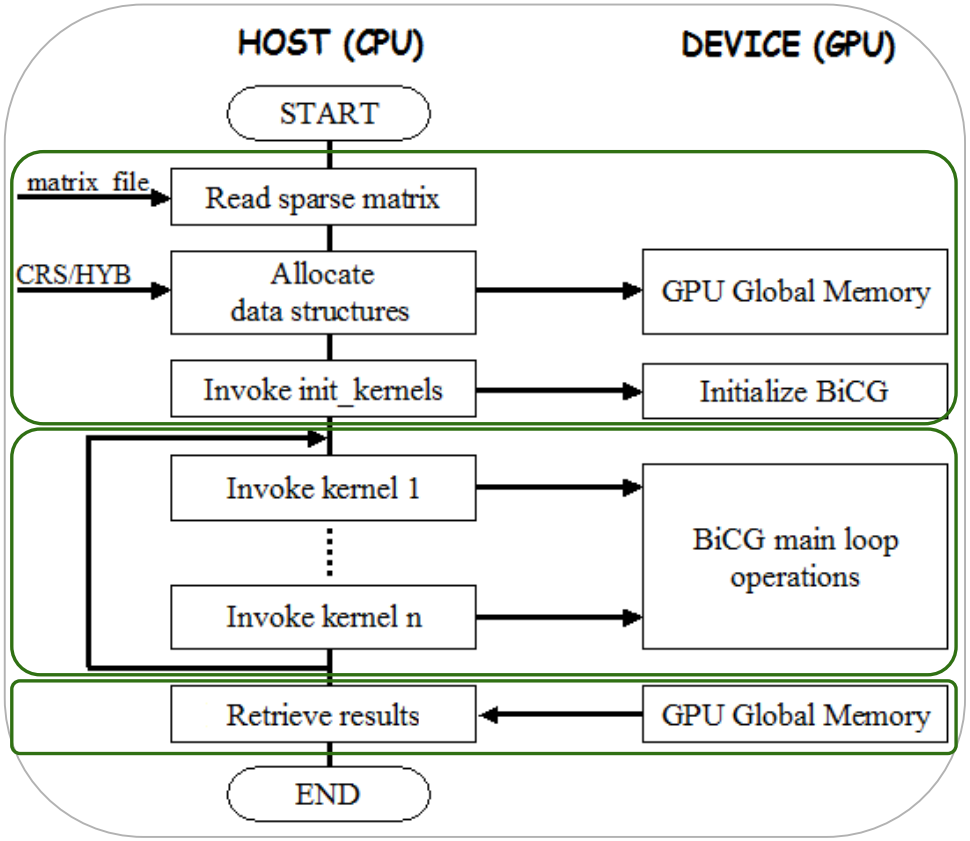
Iteration is continued till the convergence criterion is satisfied:

Values of ε commonly used are 10^{-6} / 10^{-7}.

$$\frac{\|r^i\|_2}{\|b\|_2} \leq \varepsilon$$

Design of the GPU-enabled BiCG

In the GPU-enabled BiCG algorithm, the main loop execution is controlled on the host side, whereas the computations inside are performed on the GPU.



INITIALIZATION consists in:

- reading and storing the system matrix in a given sparse format;
- allocating data structures on the GPU and calculating BiCG initial variables.

BiCG MAIN LOOP consists of the iterative invocation of parallel CUDA kernels performing the BiCG operations.

FINALIZATION consists in retrieving final results from GPU global memory.

Implementation (1/3)

Four basic CUDA kernels are enough to completely describe the BiCG main loop:

Operation	Description	FLOPS
SpMV	*Sparse matrix-vector product*	$8 \cdot nnz$
Dot product	*Scalar product of two vectors*	$8 \cdot N$
E-w product	*Element-wise product of two vectors*	$6 \cdot N$
axpy	*ax+y (a scalar, x and y vectors)*	$8 \cdot N$

Recall that we are considering ***non-symmetric*** and ***non-Hermitian sparse*** matrices with *N* rows and *nnz* non-zero ***complex*** coefficients.

Implementation (2a/3)

SpMV – this CUDA kernel implements the Bell and Garland algorithm (*) which is the best performing code currently avaible for solving sparse matrix-vector product.

$$q_i = A \cdot p_{i-1}$$
$$\bar{q}_i = A^H \cdot \bar{p}_{i-1}$$

Supported sparse matrix formats

- **CRS** (*Compressed Row Storage*)
- **HYB** (*hybrid ELLpack-COOrdinate format*)

Main modifications to the original code

- double precision complex matrix support
- CUDA grid, register number, shared and texture memory exploitation optimized for double precision complex data.

(*) N. Bell and M. Garland: "*Implementing sparse matrix-vector multiplication on throughput oriented processors*", In Supercomputing '09, Nov. 2009.

Implementation (2b/3)

(**) M. Harris, S. Sengupta, J.D. Owens: "*Parallel prex sum (scan) with CUDA*", in GPU Gems 3, Nguyen H., Ed. Addison Wesley, August 2007.

Dot product – cuBLAS dot function doesn't support double precision complex data, therefore we implemented it from scratch.
To maximize performance we also adapted Mark Harris' *parallel reduction* code (**) as the core of our code.

$$\alpha_i = \frac{\rho_{i-1}}{p_{i-1}^{*} \cdot q_i}$$

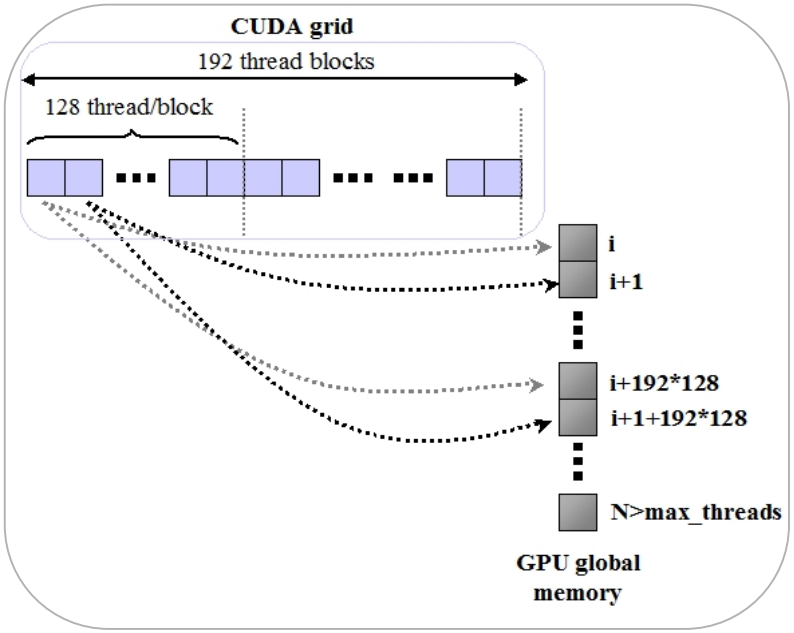
$$\rho_i = d_i^{T} \cdot \overline{r}_i^{*}$$

Implementation (3/3)

E.w. product and axpy – also in this case, the cuBLAS function provided by CUDA doesn't support double precision complex data, therefore we implemented it from scratch.

$$d_i = M^{-1} \cdot r_i$$
$$\bar{d}_i = M^{-1} \cdot \bar{r}_i$$

- We defined a CUDA grid of 192 blocks, each with 128 threads, to fully exploit GPU's resources.



$$x_i = x_{i-1} + \alpha_i \cdot p_{i-1}$$
$$r_i = r_{i-1} + \alpha_i \cdot q_i$$
$$\bar{r}_i = \bar{r}_{i-1} + \alpha_i^* \cdot \bar{q}_i$$
$$p_i = d_i + \beta_i \cdot p_{i-1}$$
$$\bar{p}_i = \bar{d}_i + \beta_i^* \cdot \bar{p}_{i-1}$$

- Threads load data from GPU global memory and perform calculations in parallel.
- We optimized global memory access pattern to obtain completely coalesced loads and stores, thus minimizing latency.

Optimization strategies

In design and implementation of CUDA kernels we adopted the following optimization strategies:

- CUDA on-chip ***shared memory exploitation*** for fast memory accesses;
- ***register usage*** optimization;
- ***loop unrolling***;
- ***texture memory exploitation*** for caching data that are spatially closed together;
- ***built-in arrays*** to store complex data thus maximizing aligned memory spaces;
- optimization of ***thread block dimension*** to maximize multiprocessor occupancy.

Experiments and results

We tested our GPU-enabled BiCG solver on linear systems whose matrices were obtained:

1. from the application of the MoM to the design of EM circuits;
2. from the "*University of Florida Sparse Matrix Collection*".

The experimentation process was carried out on the following platform:

HARDWARE configuration

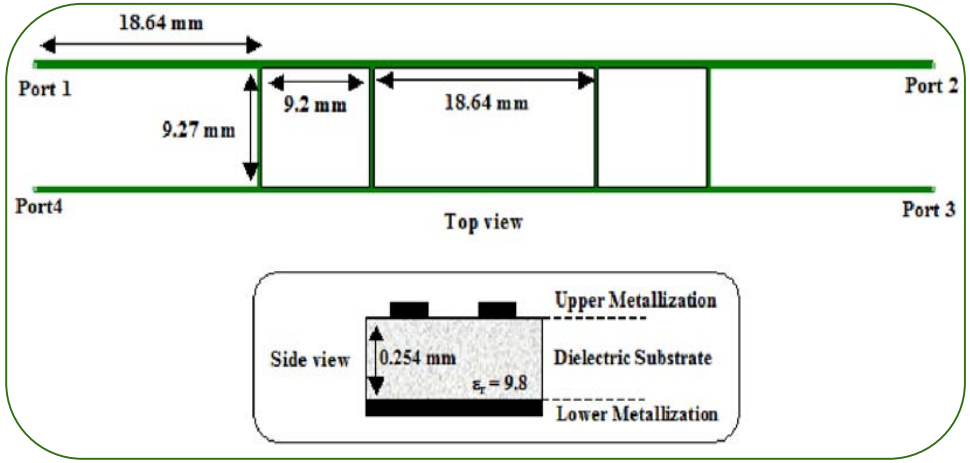
GPU: nVIDIA GeForce GTX 260, 24 SMs (192 cores), 896 MB of GDDR3 SDRAM
CPU: Intel Core2 Quad Q9550 @ 2.83 GHz, 4 GB of RAM

SOFTWARE configuration

- CUDA v. 2.3 with 190.53 driver optimized for Ubuntu 9.10 32-bit O.S.
- ATLAS v. 3.6 BLAS library

EM-MoM matrices: case study

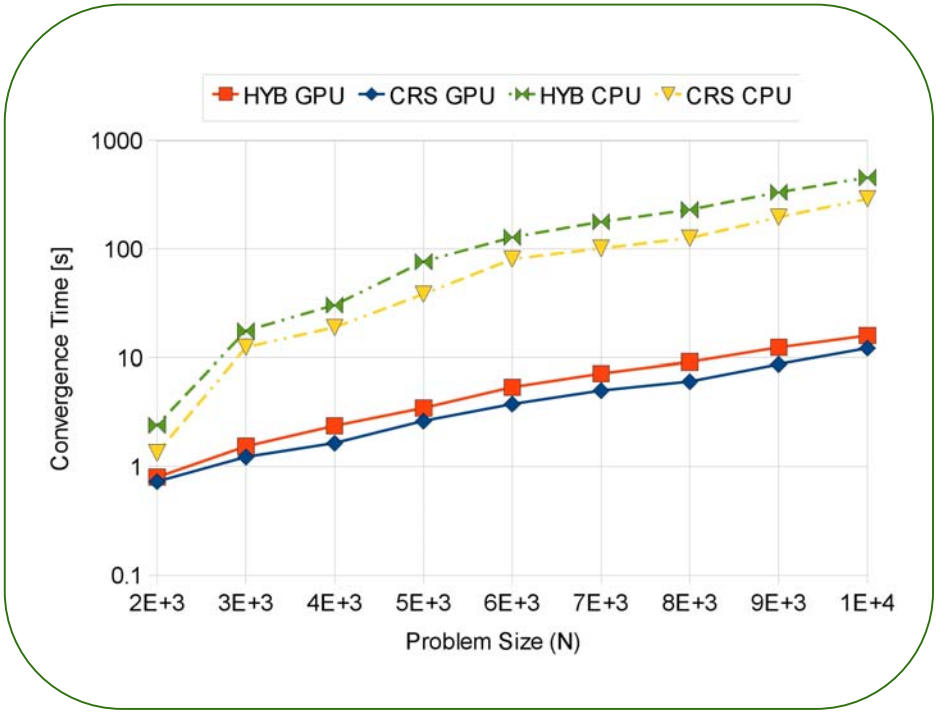
As to EM matrices derived from MoM, they concern the design of branch-line couplers in microstrip technology, which are four ports devices widely adopted in microwave and millimetre-wave applications like power dividers and combiners.



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.

EM-MoM matrices: results (1/3)

The figure below shows the convergence times of the sequential (on CPU) and parallel (on GPU) BiCG algorithm when varying the number of system unknowns, for two different sparse matrix storage formats (HYB and CRS).



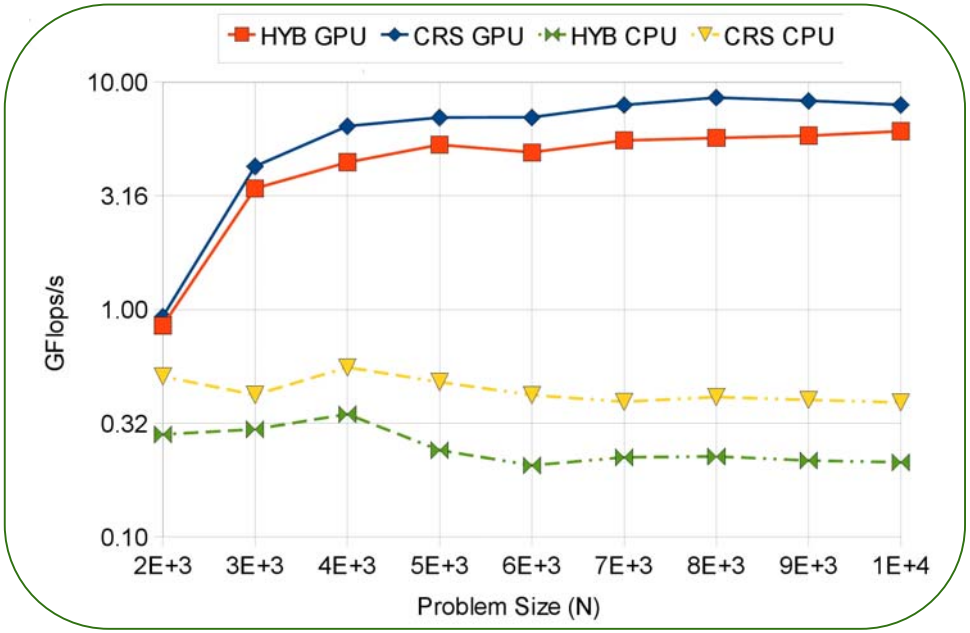
The desired matrix sparsity pattern was obtained by mean of a thresholding operation. We kept the percentage of non-zero elements to about 5% of the total number of entries while maintaining a good accuracy of the final solution.

The adopted BiCG stopping criterion:

$$\frac{\|r^i\|_2}{\|b\|_2} \leq 10^{-7}$$

EM-MoM matrices: results (2/3)

In the figure below we show BiCG performance in terms of number of floating point operations (FLOPs) per second.

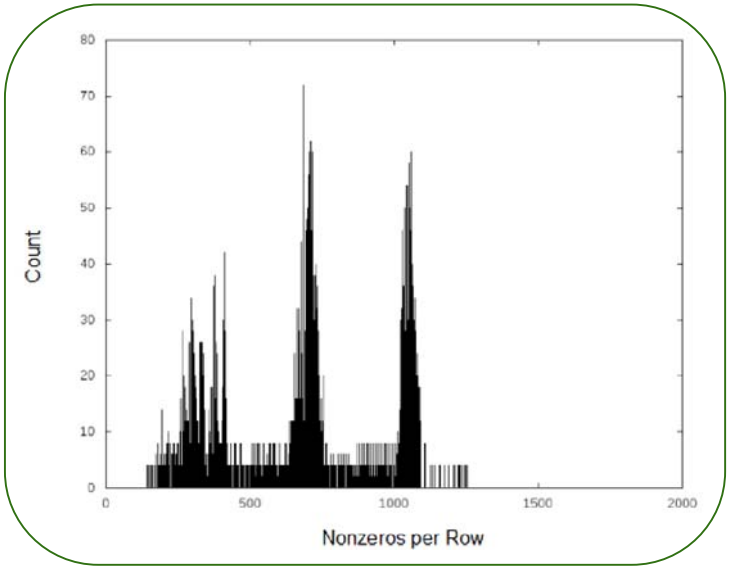


Problem Size (N)	Speed-Up CRS	Speed-Up HYB
2E+3	3.01	1.84
4E+3	12.8	11.5
6E+3	23.79	21.36
8E+3	24.76	20.74
10E+3	**28.57**	**23.69**

Achieved speed-ups are higher when matrix dimension allows for an optimum exploitation of hardware resources. CRS has the maximum benefits from GPU parallelization, achieving a speed-up of almost 30.

EM-MoM matrices: results (3/3)

In all EM matrices we analyzed, CRS format always produces faster results because of the high variability of the non-zero number per row.



As figure shows, the number of non-zeros per row varies widely in typical EM-MoM matrices, so CRS performs better than HYB. Indeed, HYB storage format is suitable when the non-zero distribution is quite compact.

FLORIDA matrices (1/2)

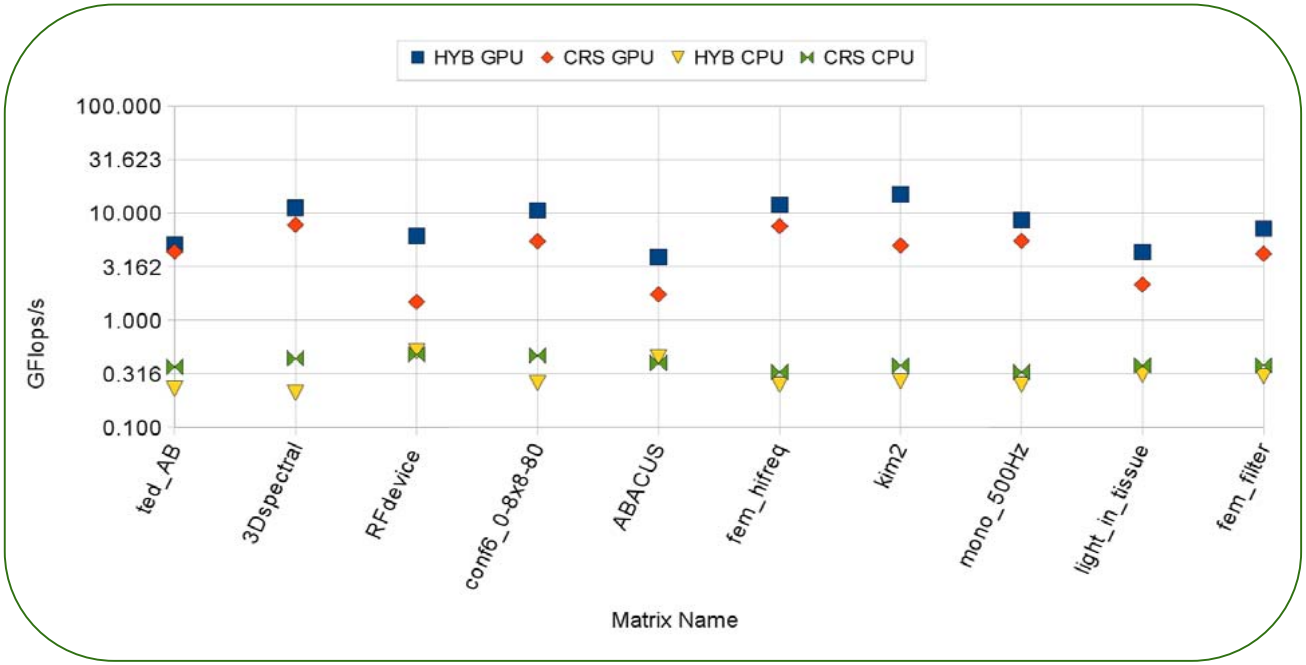
In order to demonstrate the validity of our GPU-enabled BiCG implementation, we conducted some tests on sparse matrices taken from "*The University of Florida Sparse Matrix Collection*".

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

We identified ten complex sparse matrices, belonging to different research areas and exhibiting different size, sparsity pattern and number of non-zeros.

FLORIDA matrices (2/2)

The figure below shows the performance obtained in terms of number of floating point operations per second. In the worst case the achieved speed-up is about 10, while at best we obtained 55 with 15 GFlops/s for GPU-enabled BiCG.



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

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.

Conclusions and ongoing work

- In this work, the achievement of peak-performance for EM solvers through the use of the inexpensive and powerful GPUs has been investigated.
- Taking advantage from CUDA library, we implemented a BiCG algorithm which tackles unstructured sparse matrices with double precision complex data and manages two sparse matrix formats.
- It has been tested on several research area problems. Results in terms of convergence behaviour and GPU vs. CPU performance have been provided as a validation and assessment of solver efficiency.

As further improvements to our work we plan to:

1. develop and test other sparse matrix formats suitable for EM-MoM problems;
2. integrate complex and well known pre-conditioners in our BiCG algorithm;
3. compare our code with efficient BiCG solvers adopting Intel MKL BLAS library on multi-core architectures;
4. hybridize our BiCG code to support together multi-GPU and multi-core CPUs.

Contacts

Danilo De Donno
danilo.dedonno@unisalento.it

Alessandra Esposito
alessandra.esposito@unisalento.it

Giuseppina Monti
giuseppina.monti@unisalento.it

Luciano Tarricone
luciano.tarricone@unisalento.it

ELECTROMAGNETIC LAB LECCE
UNIVERSITA' DEL SALENTO
www.electromagnetics.unile.it