JCuda vectorized and parallelized computation strategy for solving integral equations in electromagnetism on a standard personal computer

C. Rubeck, B. Bannwarth, O. Chadebec, B. Delinchant, J-P. Yonnet and J-L. Coulomb
Grenoble Electrical Engineering Laboratory (G2Elab)
Grenoble INP – Université Jospeh Fourier - CNRS UMR 5269
38402 Saint Martin d'Hères Cedex, France
christophe.rubeck@g2elab.grenoble-inp.fr

***Abstract* — In this paper, we present a computation strategy in order to solve integral equations in electromagnetism. Nowadays, Graphics Processing Units (GPU) can be found in any standard and recent computers. This paper proposes to use these units in order to improve the computation speed of a problem solved thanks to integral method. The Java language and the JCuda library, not so common in our scientific community have been used and we report a speed-up to 3 times for the solving of an electrostatic problem.**

I. Introduction

Integral equation methods (IEM) are currently widely used in electromagnetic modeling. Unlike the finite element method (FEM), they do not require the meshing of non-active materials like air. However, they are based on the computation of electromagnetic interactions between all elements (i.e. full interaction). Therefore, they lead to fully dense systems of equations. They are well known to be easily parallelizable because of the independence of interactions. Moreover, the interest of IEM has considerably increased since the emergence of acceleration methods such as the fast multipole method (FMM). In this kind of algorithm, near and far field interactions are separated. While far fields computations are highly accelerated by FMM, the near field interaction is treated classically. In particular, full near field matrices have to be computed with a high performance strategy in order to keep the advantage of using FMM.

In this work, we focus on the implementation of a parallelized and vectorized full matrix interaction computation strategy, developed on a standard personal computer equipped with a CUDA capable GPU [1]. The main software is developed thanks to Java language so the use of the JCuda library enables GPU interfacing directly from Java [2]. The choice of using Java can seem to be surprising for intensive computations, but the use of this language enables robust and fast software developments and performance are not so bad in comparison with most commonly used language like C++ [3].

In a first step, a high-performance vectorized matrix library has been developed in our lab showing that the use of Java can be competitive for numerical matrix manipulation. In a second step, we use the graphic card of our computer to speed-up the computation time. To illustrate this work, we present numerical results dealing with the resolution of a classical electrostatic problem (i.e. the computation of charge distribution on the surface of a perfect conductor).

II. Charge density computation

We consider a perfect conductor in free space associated to a known potential V_0. To compute the charge density in electrostatics, we have to solve the following integral equation:

$$V_0 = \frac{1}{4\pi\varepsilon_0}\iint_S \frac{\sigma}{r} dS \qquad (1)$$

Where S is the surface of the conductor, σ is the charge density, ε_0 is the vacuum permittivity and r is the distance between the point where the potential is expressed (on the conductor) and the integration point.

The surface is meshed into a set of triangle patches. The system of equations is generated using a point matching approach with 0-order shape function. This method is very simple but has already shown its accuracy for solving such problem. Once the set of equations is obtained, the full interaction problem is solved by a LU decomposition.

III. Vectorized Java computation

In (1), if we consider that S is meshed into N cells, we have to compute N integrals on N cells. This is why time needed to do that increase in N^2. Moreover, integrals of (1) are sometime numerically singular (in particular for the computation of the interaction of an element on itself so when r is very small). The use of analytical formulae [4] to evaluate the kernel of (1) is then preferred but these computations can be time-consuming.

In our approach, we prefer to compute integrals thanks to a numerical Gauss integration technique. With such an approach, we have three overlapped loops in our algorithm:

```
// Loop 1: on all the N elements
For i=0,1,…N // can be parallelized if multi-CPU
   // Loop 2: interaction of one element with all the N elements
   For j=0,1,…N
      // Loop 3: Gauss integration
      For k=0,…number of Gauss points
         Integral(i,j) += 1/r(i,j,k) * weight(k) * jacobian(k)
      End
   End
End
```

Let us note that it is possible to change the order of the loops and let us remember that all the interactions are independent. To improve the computation speed, we are going to vectorize a loop with a high number of indexes:

```
// Loop 1: on all the N elements
For i=0,1,…N // can be parallelized if multi-CPU
   // Loop 2: Gauss integration
   For k=0,1,…N
      // Vectorized interaction of one element with all elements
      Integral(i,:) += 1/r(i,:,k) * weight(k:) * jacobian(k:)
   End
End
```

An optimized vectorized Java matrix package has been developed in our laboratory. It is based on contiguous memory storage of the matrix, adapted indexes and macro matrix manipulation commands.

Before computing the matrix, a pre-processing is needed. A table is generated containing all the coordinates of elements Gauss points in the main referential. The process is repeated for the Gauss weights and jacobian.

In a final step, let us remember that artificial singularities have been introduced with the numerical process. We fix it by correcting the diagonal coefficients by the analytical solution. More sophisticated and more precise correction techniques will be discussed in the full paper.

IV. PARALELLISED JCUDA IMPLEMENTATION

Thanks to JCuda library, it is possible to call CUDA functions from Java. A Java GPU matrix library has been developed. It enables the management of the GPU memory allocation, the data transfer between it and the host CPU, matrix manipulations and the call of CUDA kernels.

Performances in CUDA programming are better if the algorithm is massively parallel, therefore we have to compute the matrix with a high number of threads. The chosen approach allocates one thread to each interaction. So, there are N×N threads defined. Each thread contains only the Gauss integration loop. This approach is very simple but has shown a good efficiency. Some optimizations on GPU architecture dealing with the use of shared memory will be discussed in the full paper.

Like previously, Gauss point, jacobian and weight tables are generated on the CPU, then they are sent to the graphic card which returns the integration matrix. Currently, the diagonal correction is still operated by the CPU.

V. PRELIMINARY RESULTS

We have tested our approach on personal computer equipped with a GeForce 320M that is CUDA capable. It contains 48 graphical cores and 250MB of shared memory. The CPU is an Intel C2D 2.4 GHz with 4GB of RAM. The operating system (MacOS 10.6) is full 64 bits. Let us notice that this kind of computer is very classical and low cost.

In our example, a square plate iso-potential conductor (1 by 1 cm) is modeled. This example is very simple but the goal here is to evaluate the computation performances. The host computer computation is performed in double precision with 2 CPUs whereas the GPU device maximum supports single precision. The numerical integration on triangles is provided with 7 Gauss points.

The time of the interaction matrix computation for only numerical integration is given for problems with different mesh sizes (fig. 1). Les us notice that the quality of the solution is only few influenced by the single precision conversion (less than 6e-5 % of error on charge density).

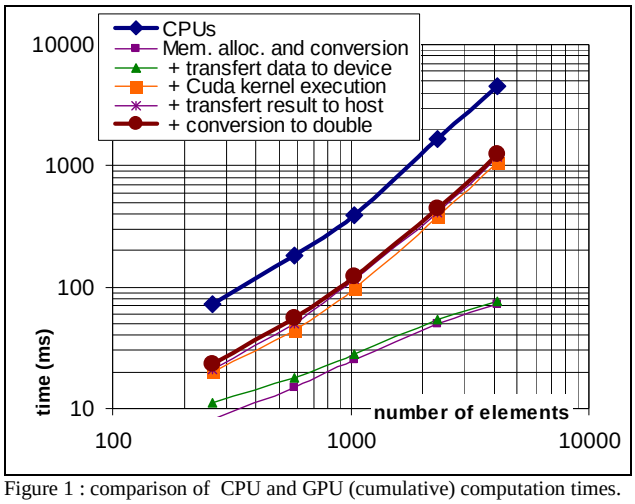

Figure 1 : comparison of CPU and GPU (cumulative) computation times.

First of all, the Java vectorized strategy seems to be quite efficient and fast (less than 5s to compute a 4000×4000 fully dense matrix). As it was expected, the GPU computation is faster than the CPU one. We note a speed-up of around 3 times in spite of the fact that the graphic card used here is a very low cost one.

As expected, the time needed to transfer and convert the data from the graphic card to the computer decrease the performance of the algorithm. However, several ways to do it more efficiently have still to be studied.

The portability of the Java code has been tested on a win32 computer. After the setting of the JCuda library, only a recompilation of the Cuda kernel is needed to make the experiments. We do not report a speed-up because the graphic card (Quadro NVS 160M) only owns 8 cores.

VI. CONCLUSION AND PERSPECTIVES

We have reported in this paper a strategy for computing electromagnetic fields on Java platform with the JCuda library on a standard computer. These first results are maybe not so impressive, but these techniques are still not optimized and can be applied without buying additional hardware device. A far fields GPU-based FMM technique is currently under development to improve the speed of the computation like in the presented near field computation.

VII. REFERENCES

[1] NVIDIA. (2010, Oct.) “NVIDIA CUDA programming guide”, http://developer.download.nvidia.com/compute/cuda/3_2/toolkit/docs/CUDA_C_Programming_Guide.pdf

[2] M. Hutter, JCuda (Java bindings for CUDA), http://www.jcuda.de/

[3] V. Reinauer, T. Wendland, C. Scheiblich, R. Banucu, “Object-Oriented Development and Runtime Investigation of 3-D electrostatic FEM problems in Pure Java”, Proceeding of CEFC 2010 Confrenece, to be published in *IEEE Trans. Mag*.,2011.

[4] S. Rao, A. Glisson, D. Wilton, and B. Vidula, “A simple numerical solution procedure for statics problems involving arbitrary-shaped surfaces,” *IEEE Trans. Antennas Propagat.* vol. 27, no. 5, pp. 604–608, Sep 1979.