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

Christophe Rubeck, Bertrand Bannwarth, Olivier Chadebec, Benoît Delinchant, Jean-Paul Yonnet, Jean-Louis Coulomb

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, INP/UJF/CNRS UMR 5269, 38402 Saint Martin d'Hères Cedex, France

The paper presents a computation strategy for solving integral equations in electromagnetism. Nowadays, powerful programmable Graphic Processing Units (GPU) can be found in any standard computer. The paper investigates the benefits of the use of GPUs in addition to the CPU one in order to improve computation speed by using integral methods. Java language and the JCuda library, not often used in speed calculation by the computing community, has been used here. A 100 time speed-up is reported in matrix assembly between an optimized traditional CPU computation and a CPU+GPU one. FMM…

***Index Terms*—Fast Multipole Method on GPU, JCuda computing, Pure Java ...**

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 [1]:

$$V = V_{near} + V_{far} \tag{1}$$

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, a parallelized and vectorized full matrix interaction computation strategy is implemented, first using a classical CPU on a standard personal computer then a CUDA capable GPU [2]. The main software is developed thanks to Java language so the use of the JCuda library enables GPU interfacing directly from Java [3]. The choice of using Java can seem to be surprising for speed computations, but the use of this language enables easy portability, robust and fast software developments, and performances are respectable in comparison with most commonly used language like C++ [4].

II. Charge density computation

Let consider a perfect conductor be in free space associated to a known potential V_0. To compute the charge density in electrostatics, the following integral equation has to be solved:

$$V_0 = \frac{1}{4\pi\varepsilon_0}\iint_S \frac{\sigma}{r} dS \tag{2}$$

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.

III. Near field computation

The near potential (1) is computed as presented in (2). Let mesh the surface into a set of triangle patches. The system of linear equations is generated using a point matching approach with 0-order shape function. This method is very simple but has already shown its accuracy.

A. High performance matrix assembly

In (2), if S is meshed into N cells, N integrals on N cells have to be computed. This is why the computation time increases in N^2. Moreover, the numerical evaluation of the integrals of (2) is sometime numerically singular (in particular for the computation of the interaction of an element on itself so when r is close to zero). The use of analytical formulae [5] to evaluate the kernel of (2) is then preferred but these computations can be time-consuming. The chosen approach is a mix of numerical and analytical integral computation in order to get the best ratio between accuracy and speed.

The integrals are first computed thanks to a numerical Gauss integration technique. In this approach, there are three overlapped loops in the algorithm (fig. 1).

```
// Loop 1: on all the N elements
For i = 0,1,…N // can be parallelized if multi-CPU
  // Loop 2: interaction of element i with all 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
```

Fig. 1. Classical matrix assembly algorithm.

Manuscript received January 1, 2008 (date on which paper was submitted for review). Corresponding author: F. A. Author (e-mail: f.author@nist.gov).

Digital Object Identifier inserted by IEEE

```
// 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,… number of Gauss points
      // Vectorized interaction of element i with all elements
      Integral(i,:) += 1/r(i,:,k) *weight(k,:) * jacobian(k,:)
   End
End
```

Fig. 2. Vectorized matrix assembly algorithm.

Working on larger and continuous sets of data improves memory access speed and therefore the computation speed. That is why the two last loops are switched in order to have a high number of indexes in the last loop. It is allowed because all interactions are independent. Furthermore vectorized operators can be used as shown in the fig. 2. The benefit of the vectorized algorithm toward the classical one is a speed-up of almost 30 to 50 times for the matrix assembly.

B. *Fixing singularities*

After the matrix assembly, the artificial singularities which have been introduced with the numerical integration are fixed. The diagonal coefficients of the matrix are corrected by the analytical solution of the corresponding integrals.

C. *Pure Java computation*

An optimized vectorized Java matrix package has been developed in the G2Elab [6]. It is based on contiguous memory storage of the matrix, adapted indexes and macro matrix manipulation commands.

Before assembling the matrix, a pre-processing is needed. A table containing all the coordinates of elements Gauss points in the main referential is generated. The process is repeated for the Gauss weights, the Gauss jacobians and the matching points. Then the matrix is assembled line per line thanks to vectorized operators and the diagonal is fixed.

D. *JCuda implementation*

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

1) *GPU computing accuracy*

It is well known that GPU are much faster in single precision computing than in double precision. Furthermore, only latest GPU are double precision capable. Therefore the good computation accuracy of (2) in single precision must be checked before developing a CUDA method. Computations of (2) on a triangle in single and double precision are compared for analytical and numerical method. Errors are noticed in the order of magnitude of 1e-7. So the quality of the evaluation of (2) is only a few influenced by the single precision computation. For the same calculations, results between CPU and GPU are compared. A difference on the 1-3 last digits is noticed. The explanation is that in CUDA most functions are implemented in non-standard-compliant way [2].

2) *JCuda matrix assembly*

Performances in CUDA programming are better if the algorithm is massively parallelized, therefore the matrix has to be computed with a high number of threads. The chosen approach allocates one thread to each interaction. So, N^2 threads are defined. Each thread contains only the Gauss integration loop. A simplified version of the matrix assembly CUDA kernel is presented in fig. 3.

```
__global__ void Assembling_kernel
      (int N, float* matrix, ..., int nb_gauss_pt)
{
   // get the thread coordinates
   int j = blockDim.x * blockIdx.x + threadIdx.x;
   int i = blockDim.y * blockIdx.y + threadIdx.y;

   if ( i < N && j < N )
   {
      - get the matching point from global memory
      // loop on Gauss points
      float integral = 0.0;
      for ( int k=0; k < nb_gauss_pt; k++ )
      {
         - get gauss point coordinates from global memory
         - get gauss weight and jacobian from global memory
         - compute r
         // compute the integral
         integral += 1.0 / r * gauss_weight * gauss_jac;
      }
      // return the potential to the global memory
      matrix[i*N+j] = 1.0 / (4*Pi*Epsilon) * integral;
   }
}
```

Fig. 3. Simplified CUDA kernel for matrix assembly using Gauss integration technique.

Only the mesh is sent to the GPU. Gauss points, Gauss jacobians, Gauss weights and matching points tables are generated on the GPU thanks to CUDA kernels. The tables stay in the GPU global memory. The access to the data from the assembling CUDA kernel is done continuously to preserve high speed computation (i and j indexes are inverted). Finally the diagonal of the matrix is fixed with analytical solutions.

3) *JCuda linear system solving*

Once the set of linear equations is obtained and stored in the global memory of GPU, the full interaction problem is solved by a GMRES iterative method [7]. The iterative method developed here is not a fully GPU algorithm because it is hardly parallelizable, only the matrix product for the computation of the residual is done using the JCublas library.

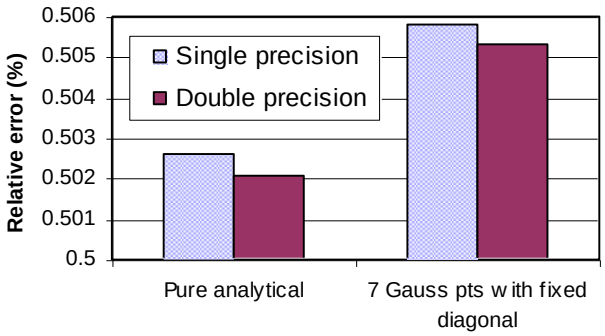

Fig. 4. Relative error of the charge distribution depending of integration technique and variables precision (lower is better).

Therefore at each iteration a vector is transmitted between the host computer and the CUDA device. A simple Jacobi preconditioning method is used.

4) Hardware adaptability

The main limiting factor of GPU computing is the graphic memory amount. That is why two strategies have been developed. First if the GPU can contain the matrix then the problem is solved with the iterative method presented before. If there is no memory enough, then the matrix is assembled by blocks and transferred to the host computer. The system is solved by a classical CPU method.

E. Results

1) Benchmark problem

In our example, a spherical iso-potential conductor (radius of 10 cm) is modeled. The potential is set to 1 mV. This example is simple but the theoretical charge density is known, so the accuracy of computing can be precisely evaluated.

2) Hardware specifications

The CPU results are obtained with an Intel Xeon 2.67 GHz in single core. The CUDA devices are a Geforce 320M (48 cores, 0.95 GHz, 250 MB of shared memory) and a Tesla C1060 (240 cores, 1.30 GHz, 4 GB of RAM).

3) Integration method accuracy

The sphere is meshed into 9068 triangular patches. The GMRES convergence break is set to 1e-9 and the Krylov subspace is extended to 30. Therefore the errors are mainly due to the meshing and the integration technique. The fig. 4 shows the relative error of the charge distribution between the theoretical values and the computations, depending of integration technique and variable precision. As expected, a double precision computation is better than a single one, and a pure analytical computation [8] is better than a numerical one. However, the differences are really poor (0.004%) and it is reasonable to use the numerical method in single precision.

4) Matrix assembly speed

The mesh is refined to 20602 patches and the matrix is computed in single precision. As shown on fig. 5, numerical technique is faster than the pure analytical one (3-4 times). There is one order of magnitude in computation times between the CPU computation and the low cost Geforce 230M one, for which the matrix is assembled in 9 parts and transferred to the host computer. Finally, two orders of magnitude are noticed between the CPU and the Tesla C1060 computations. The speed up in using GPU is not the number of GPU cores because a GPU core is less powerful than an Intel CPU one.

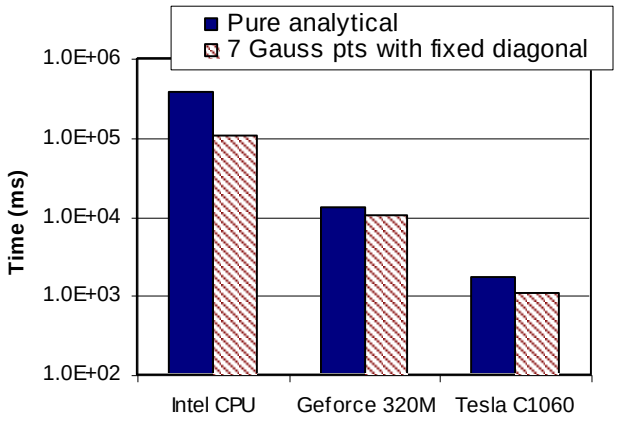

Fig. 5. Comparison of matrix assembly times depending of architecture and integration technique (lower is better).

5) Iterative solver speed

A gain of an order of magnitude between CPU and GPU in the solving of the linear system is noticed in spite of the data transfers between the CUDA device and the host computer. The computation is done in single precision and the meshing and solving parameters are the same as previously.

6) JCuda limitations

Some commonly optimizations, like the use of pinned memory in order to speed up data transfers, are not attractive. Indeed the data transfer is faster but the access to these data from Java is slower. In fact JCuda is based on C library and performances decrease in using C memory scheme from Java.

F. Conclusion

A very efficient integration method has been developed, mixing analytical and numerical approach. Matrix assembly is faster without real losing of precision. Furthermore, the use of GPU thanks to JCuda library speeds up the computation of two orders of magnitude. The linear solving is also speeded up of one order of magnitude.

IV. Far Field Computation

A. Fast Multipole Method

B. Single precision computation

C. JCuda implementation

D. Results

E. Conclusion

V. 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. ….

Wavelets compression [9], [10] which seems more parallelizable than the FMM will be investigated.

VI. References

[1] L. Greengard and V. Rokhlin, “A fast algorithm for particle simulations, Journal of Computational Physics”, vol. 73, Issue 2, pp. 325-348, Dec 1987.

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

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

[4] 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.

[5] E. Lezar and D.B. Davidson, “GPU acceleration of method of moments matrix assembly using Rao-Wilton-Glisson basis functions,” Proceeding of ICEIE 2010 conference, vol.1, no., pp.V1-56-V1-60, 1-3 Aug. 2010.

[6] J-L. Coulomb, “Numerical Design Of Experiments and Optimization” , http://forge-mage.g2elab.grenoble-inp.fr/project/got

[7] Y. Saad and M. H. Schultz, “GMRES: A generalized minimal residual algorithm for solving nonsymmetric linear systems”, *SIAM J. Sci. Stat. Comput.* vol. 7, no. 3, pp. 856, July 1986.

[8] 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.

[9] C. Scheiblich, V. Kolitsas and W. M. Rucker, “Compression of the Radiative Heat Transfer BEM Matrix of an Inductive Heating System Using a Block-Oriented Wavelet Transform,” *IEEE Trans. Magn.*, vol. 47, no. 3, pp. 1712-1715, Mar. 2009.

[10] C. Scheiblich, R. Banucu, V. Reinauer, J. Albert and W. M. Rucker, “Parallel Hierarchical Block Wavelet Compression for an Optimal Compression for 3-D BEM Problems,” *IEEE Trans. Magn.*, vol. 47, no. 5, pp. 1386-1389, May 2011.