ESAIM: PROCEEDINGS AND SURVEYS, September 2014, Vol. 45, p. 199-208
J.-S. Dhersin, Editor

MULTI-GPU NUMERICAL SIMULATION OF ELECTROMAGNETIC WAVES *

Philippe Helluy[1] and Thomas Strub[2]

Abstract. In this paper we present three-dimensional numerical simulations of electromagnetic waves. The Maxwell equations are solved by the Discontinuous Galerkin (DG) method. For achieving high performance, we exploit two levels of parallelism. The coarse grain parallelism is managed through MPI and a classical domain decomposition. The fine grain parallelism is managed with OpenCL in order to optimize the local computations on multicore processors or GPU's. We present several numerical experiments and performance comparisons.

Résumé. Dans cet article, nous présentons des simulations numériques tridimensionnelles d'ondes électromagnétiques. Les équations de Maxwell sont résolues par la méthode de Galerkin Discontinue (GD). Pour accélérer les calculs, nous exploitons deux niveaux de parallélisme. Le large grain est basé sur MPI. Le parallélisme à grain fin repose sur OpenCL afin d'exploiter les processeurs massivement multicoeur (GPU ou CPU) récents. Nous présentons plusieurs expériences numériques et des tests de performance.

Introduction

The Discontinuous Galerkin (DG) method is becoming more and more popular for the numerical simulation of electromagnetic phenomena [1, 2, 5]. Compared to the Finite Difference Method in the Time Domain (FDTD) of Yee [6], it allows high order unstructured meshes. It also simplifies local mesh refinements and local time-stepping. However, the high flexibility is paid by a higher numerical cost. It is thus very important to provide optimized parallel implementation of the DG method.

In this work we present the CLAC solver, which aims at solving the time-dependent Maxwell equations on GPU clusters. CLAC is an acronym for "Conservation Laws Approximation on many Cores". It is written in C++ and designed for solving linear or non-linear first order systems of conservation laws by the DG method, such that Maxwell's equation, MHD or Euler equations. It relies on an initial subdomain decomposition of the mesh. We manage the communications between the subdomains with the MPI library (coarse grain parallelism). In each domain, the computations of the local fluxes and source terms are accelerated through calls to OpenCL (fine grain parallelism). OpenCL is a recent software framework for programming multicore accelerators. Its main advantage on other tools, like CUDA, is its compatibility with GPU's of different brands. It is also able to address recent multicore processors.

* *This work is part of the project GREAT (Galerkin Resolution for Electromagnetic Applications in the Time domain) supported by DGA/DS/MRIS through a dual innovation project RAPID.*

[1] Inria Tonus and IRMA Université de Strasbourg; e-mail: `helluy@unistra.fr`

[2] Axessim, rue J. Sapidus Illkirch France; e-mail: `thomas.strub@axessim.fr`

Article published online by EDP Sciences and available at http://www.esaim-proc.org or http://dx.doi.org/10.1051/proc/201445020

We first describe the mathematical model. Then, we give some important details of our DG implementation. Finally, we present some numerical results and performance comparisons.

1. Mathematical model

The philosophy behind the CLAC solver is to separate the mathematical model from its physical description. In this way, it is possible to solve PDE's from different application fields. We thus consider a very general system of conservation laws

$$\partial_t W + \partial_i F^i(W) = S(W). \tag{1}$$

In this system, the unknown is a vector of conservative variables $W(X,t) \in \mathbb{R}^m$ that depends on the space variable $X = (x, y, z) \in \mathbb{R}^3$ and on the time t. We use the convention of sum on repeated indices: $\partial_i F^i$ means $\sum_{i=1}^3 \partial_i F^i$. The source term is denoted by S. The flux components F^i, $i = 1 \cdots 3$, are smooth applications from $\mathbb{R}^m$ to $\mathbb{R}^m$. For a vector $n = (n_1, n_2, n_3) \in \mathbb{R}^3$ we shall also define the flux function by

$$F(W, n) = F^i(W) n_i. \tag{2}$$

From the flux function, we can obviously recover the flux components. For instance, we can consider the Kronecker index

$$\delta_i^j = \begin{cases} 1 & \text{if} \quad i = j, \\ 0 & \text{if} \quad i \neq j. \end{cases}$$

Then

$$F^j(W) = F(W, \delta^j) \tag{3}$$

The system (1) is written in a computational domain Ω. On the boundary $\partial\Omega$ we have to provide boundary conditions. We will detail them later on. Finally, the system is supplemented by an initial condition

$$W(X, 0) = W_0(X).$$

The Maxwell equations read

$$\partial_t E + \nabla \times H = -J, \tag{4}$$

$$\partial_t H - \nabla \times E = 0, \tag{5}$$

where E is the electric field, H the magnetic field and J the electric current. We set

$$W = (E^T, H^T)^T, \quad S = (-J^T, (0,0,0)^T)^T,$$

and

$$F(W, n) = \begin{bmatrix} 0 & n\times \\ -n\times & 0 \end{bmatrix} W = A(n) W,$$

in such a way that the Maxwell equations are a particular case of (1), with $m = 6$.
As in (3) we can define the following 6×6 symmetric matrices

$$A^i = A(\delta^i), \quad i = 1 \cdots 3.$$

Then, the Maxwell equations can also be written under the form of a symmetric hyperbolic system, also called a Friedrich's system

$$\partial_t W + A^i \partial_i W = S.$$

2. Discontinuous Galerkin (DG) approximation

2.1. General DG formalism

For defining the DG approximation, we have first to construct a mesh of the domain Ω. We consider a mesh made of a finite number of open sets $L_k \subset \Omega$, $k = 1 \cdots N$, called the cells or the elements and satisfying the two conditions:

(1) $\forall k, l \quad k \neq l \Rightarrow L_k \cap L_l = \emptyset$.
(2) $\overline{\cup_k L_k} = \overline{\Omega}$.

Let L and R be two neighbor cells. The face between L and R is denoted by

$$L/R = \overline{L} \cap \overline{R}.$$

We also denote by n_{LR} the unit normal vector on L/R oriented from L to R. Therefore, $n_{LR} = -n_{RL}$.
In each cell L, we consider a basis of scalar function φ_i^L, $i = 1 \cdots p_L$ with support in L. It is possible to take different approximation order p_L on different cells. In cell L, the solution of (1) is approximated by

$$W(X,t) = W_L(X,t) = W_L^j(t)\varphi_j^L(X) \quad X \in L. \tag{6}$$

The numerical solution satisfies the DG approximation scheme

$$\forall L, \forall i \quad \int_L \partial_t W \varphi_i^L - \int_L F(W, \nabla \varphi_i^L) + \int_{\partial L} F(W_L, W_R, n_{LR})\varphi_i^L = \int_L S\varphi_i^L. \tag{7}$$

For more clarity, let us give some remarks on the DG scheme (7):

(1) The DG formulation (7) is formally obtained by multiplying (1) by a basis function φ_i^L and integrating by part over the cell L.
(2) With some abuse, we use the same notation for the exact and the approximated solution. From now on, W denotes the DG approximation of the exact solution.
(3) By R we denote a generic cell that touches cell L on its boundary ∂L. This notation is justified by the fact that, up to a rotation, we can always assume that the normal vector n_{LR} is oriented from cell L at the Left to cell R at the Right.
(4) Because W is discontinuous at the cell boundaries, it is not possible to define $F(W, n_{LR})$ on ∂L. Therefore, as in the finite volume method, we have to introduce a numerical flux $F(W_L, W_R, n_{LR})$. Generally, the numerical flux satisfies the two conditions
   (a) Consistency: $F(W, W, n) = F(W, n)$.
   (b) Conservation: $F(W_L, W_R, n_{LR}) = -F(W_R, W_L, n_{RL})$.
(5) For Maxwell, we will use the standard upwind numerical flux

$$F(W_L, W_R, n) = A(n)^+ W_L + A(n)^- W_R.$$

(6) Finally, in (7) we have to be more precise when a cell L touches the boundary of the computational domain. Indeed, on $\partial L \cap \Omega$ the neighbor vector W_R is not available. We replace then the numerical flux $F(W_L, W_R, n_{LR})$ by a boundary flux

$$F_b(W_L, n_{LR}).$$

The form of this boundary term depends on the type of boundary condition (Silver-Müller, metal, *etc.*).

We can now introduce expansion (6) into (7). This leads to a system of ordinary differential equations satisfied by the basis coefficients $W_L^j(t)$. We solve it with a standard second order Runge-Kutta integrator.

2.2. **Gauss-Legendre interpolation**

Theoretically, in a DG approximation, the basis function φ_L^i and the mesh elements L_k can be arbitrary. However for performance reasons a clever choice has to be provided. A good compromise between efficiency and generality is to choose linear hexahedral cells and basis functions of arbitrary order d associated to tensorized Gauss-Legendre points. The advantages of such a choice are the following:

(1) Our hexahedrons are defined by 8 points. Their faces are not necessarily plane. This allows to mesh any complex geometry with a reasonable precision.
(2) The basis functions are the Lagrange polynomials associated to the Gauss-Legendre points that we also use for the numerical integration of the volume integrals in (7). In this way the local mass matrices of the DG scheme are diagonal and thus easy to invert.
(3) Two tricks greatly accelerate the computations of the terms $\int_L F(W, \nabla\varphi_i^L)$. First, the values of the fields at the volume Gauss points are simply the basis coefficients W_L^i. Secondly, the gradient $\nabla\varphi_i^L$ cancels at many Gauss points due to the tensor nature of the basis function. For instance, when using polynomials of degree d, our choice leads to $3(d+1)$ nonzero gradients instead of $(d+1)^3$.
(4) For extrapolating W at one Gauss point of a face, it is sufficient to consider only one line of Gauss interpolation points in the volume. In this step we thus use only $(d+1)$ points instead of the all $(d+1)^3$ interpolation points in the volume.

Being fair, we admit a few drawbacks:

(1) In practice, it is much easier to mesh an arbitrary domain with tetrahedrons. It is then possible to cut every tetrahedron into four hexahedron, but some of them may be to much stretched.
(2) We could have used Gauss-Lobatto points for avoiding the extrapolation at the Gauss points of the faces. The loss of precision of the numerical integration would certainly be compensated by the faster flux integration.

2.2.1. *Geometric transformation*

Let us now define precisely our interpolation method. The reference element $\hat{L}$ is the unit cube

$$\hat{L} = [0,1]^3.$$

We denote by $\hat{X} = (\hat{x}, \hat{y}, \hat{z})$ the reference coordinates. The reference nodes $\hat{X}^i$, $i = 1 \cdots 8$ and the geometric functions $\hat{\psi}_i$ of the reference elements are given by

i	$\hat{X}^i$	$\hat{\psi}_i$
1	$(0,0,0)$	$(1-\hat{x})(1-\hat{y})(1-\hat{z})$
2	$(1,0,0)$	$\hat{x}(1-\hat{y})(1-\hat{z})$
3	$(1,1,0)$	$\hat{x}\hat{y}(1-\hat{z})$
4	$(0,1,0)$	$(1-\hat{x})\hat{y}(1-\hat{z})$
5	$(0,0,1)$	$(1-\hat{x})(1-\hat{y})\hat{z}$
6	$(1,0,1)$	$\hat{x}(1-\hat{y})\hat{z}$
7	$(1,1,1)$	$\hat{x}\hat{y}\hat{z}$
8	$(0,1,1)$	$(1-\hat{x})\hat{y}\hat{z}$

An arbitrary cell is then defined by eight nodes X_L^i. The geometric transformation that maps $\hat{L}$ on L is then given by

$$\tau_L(\hat{X}) = \hat{\psi}_i(\hat{X}) X_L^i.$$

We assume that the nodes X_L^i are chosen in such a way that τ_L is a direct and invertible transformation. Because the reference basis functions satisfy

$$\hat{\psi}_i(\hat{X}^j) = \delta_{ij}$$

we deduce that the geometric transformation maps the reference nodes on the nodes of element L

$$\tau_L(\hat{X}^i) = X_L^i.$$

2.2.2. *Gauss-Legendre interpolation*

For the interpolation, we first fix in the cell L a degree d. We consider the $(d+1)$ zeros $(\xi_i)_{i=0\cdots d}$ of the $(d+1)^{\text{th}}$ Legendre polynomial on $[0,1]$, and the corresponding integration weights ω_i. We also denote by I_k the k^{th} Lagrange interpolation polynomial associated to the ξ_i's. Recall that I_k is a polynomial of degree d and that

$$I_j(\xi_i) = \delta_{ij}.$$

We construct the reference Gauss points $\hat{Y}_q$, weights $\hat{\lambda}_q$ and interpolation functions $\hat{\varphi}^q(\hat{X})$ from tensor products of the one-dimensional quantities. More precisely, let i, j and k be three integers in $\{0\cdots d\}$ and let $q = (d+1)^2k + (d+1)j + i$ then

$$\hat{Y}_q = (\xi_i, \xi_j, \xi_k), \quad \hat{\lambda}_q = \omega_i\omega_j\omega_k, \quad \hat{\varphi}^q(\hat{X}) = I_i(\hat{x})I_j(\hat{y})I_k(\hat{z}).$$

Finally, we obtain the basis functions of element L by transporting the reference interpolation functions with the geometric transformation.

$$\varphi_L^i(X) = \hat{\varphi}^i(\hat{X}) \text{ with } X = \tau_L(\hat{X}).$$

3. Fine grain parallelism : OpenCL implementation

3.1. **OpenCL library**

OpenCL library provides a model of parallel computing which can be used on CPU and GPU.

In this model, the memory is divided into global memory of some gigabytes with slow access and local (cache) memory of a few tens of kilobytes with quick access. Calculations are performed by "work-items" having access to all the global memory and which number can be arbitrarily large. The work-items are grouped into "work-groups" with common local memory.

All work-items execute the same program, called "kernel". Each work-item is identified by an index for parallelization.

To make the best use of OpenCL parallelization, certain rules must be followed:

- work-items must perform similar operations, conditional statements are costly;
- access the global memory must be done in a linear way;
- several work-items cannot write at the same memory at the same time;
- intermediate results can be temporarily stored in local memory for faster access.

Based on these rules, we defined kernels and data structures adapted to GPU parallel computations.

3.2. **Parallelization**

The algorithm is divided into stages, each carried out by a specific kernel. These steps are (1) the calculation of the volume integral $\int_L F(W, \nabla\varphi_i^L)$ for each basis function φ_i^L, (2) the calculation of the surface integral $\int_{\partial L} F(W_L, W_R, n_{LR})\varphi_i^L$ for each basis function φ_i^L and (3) the Runge-Kutta step.

In each of these kernels, a work-group is associated with a mesh element and the work-items are distributed on the surface or volume element integration points.

Let us describe more precisely the calculation of the surface integral made by kernel (1). We have

$$\begin{aligned}\int_{\partial L} F(W_L, W_R, n_{LR})\varphi_i^L &= \int_{\partial \hat{L}} F(W_L \circ \tau_L, W_R \circ \tau_L, D\tau_L^* \hat{n}_{LR})\hat{\varphi}_i \\ &= \sum_{k=0}^{5} \hat{\lambda}_{p_k(i)} \cdot \\ &\quad F(W_L \circ \tau_L(\hat{X}_{p_k(i)}), W_R \circ \tau_L(\hat{X}_{p_k(i)}), D\tau_L^* \hat{n}_{LR}(\hat{X}_{p_k(i)})) \end{aligned} \quad (8)$$

The point $\hat{X}_{p_k(i)}$ is the projection of the volume Gauss-Legendre point $\hat{X}_i$ on the face k. $D\tau_L^*$ is the adjugate of the jacobian matrix of τ_L. The vector $\hat{n}_{LR}$ is the normal vector on the corresponding face of the reference element. The computation are simplified because the basis function $\hat{\varphi}_i$ cancels at all surface points except at the projection of Gauss point i on the face.

In kernel (1), the number of work-items is equal to $\max((d+1)^3, 6(d+1)^2)$, i.e. the maximal number of surface or volume interpolation points. The kernel is itself split into two steps (a) and (b). In one of the two steps, a part of the work-item will be idling.

In step (a), the work-items *are distributed to the surface integration points* and calculate the values of the numerical flux at these points i.e. $F(W_L, W_R, D\tau_L^* \hat{n}_{LR})$. The fields W_L and W_R are evaluated at the face Gauss point using the trick described in 2.2 (only compute the interpolation points aligned with the surface point). The flux is then stored into local memory in order to be efficiently available to all the work-items in step (b).

In the second step (b), the work-items *are distributed to the volume interpolation points*. Work-item i calculates the surface integral (8). Only the values at one point of each face of the element comes in, because on the other points the basis functions vanish. These values are simply collected from local memory thanks to the previous step (a). In step (b), because generally the degree $d < 6$, some work-items are idling. The small loss of computational power is compensated by the fact that we have kept the fluxes into the cache memory. We point out that this strategy is different from the one described in [4] and can be generalized to non-linear hyperbolic systems.

Finally in kernel (3), the vector representing the time derivative of the field is updated.

3.3. **Division into homogeneous zones**

In the different OpenCL kernels participating to the element computations, the number of work-items depends on the degree of interpolation of the element.

A strategy for grouping elements of the same order into homogeneous zones was implemented. In this way the work-items of a same zone perform exactly the same operations, which improve the GPU efficiency.

4. Coarse grain parallelism : MPI implementation

MPI allows us to parallelize computations on a distributed memory architecture by launching multiple process which communicate between each others. We perform a subdivision of the mesh into sub-domains. During the simulation, the boundary cells data are exchanged between sub-domains thanks to standard MPI send and receive operations. Before the MPI exchanges, the boundary data have also to be copied between the GPU and the CPU, which increases the simulation cost compared to other MPI parallel algorithms.

5. Numerical results

In this section, we present the results obtained with three test cases. In the first test case, we compute the order of the numerical scheme. The second test case compares the Silver-Müller boundary condition and the cPML absorbing layers. The last test case highlights the scalability of the method with MPI on a mesh of large size.

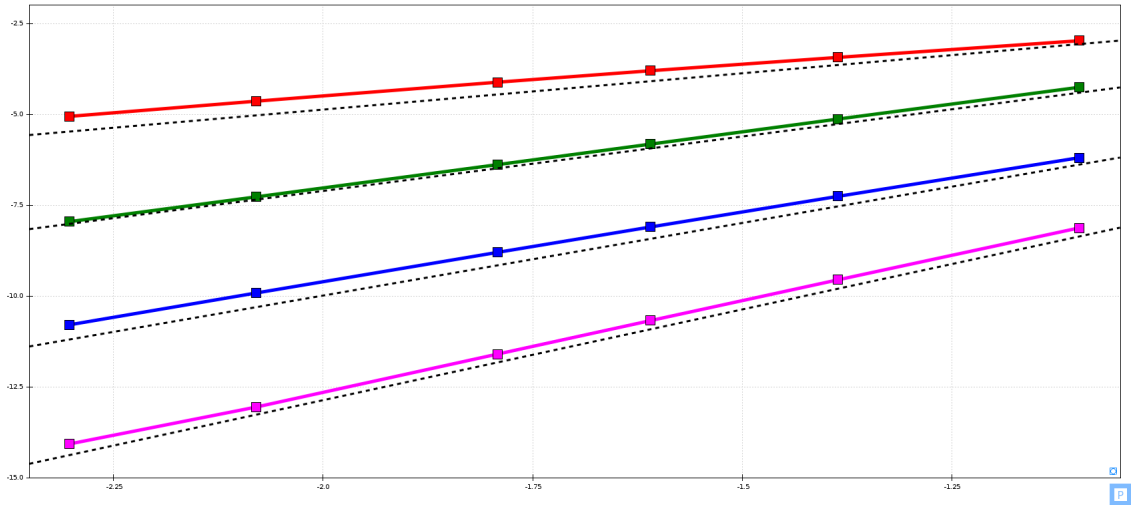

FIGURE 1. Convergence orders ($d = 1$: red, $d = 2$: green, $d = 3$: blue, $d = 4$: purple)

5.1. **Convergence**

In this section we compute the convergence order of the scheme with different interpolation order d. Theoretically the order (in the L^2 norm) of the scheme should be $d + 1$.

Let $W_e : \mathbb{R}^3 \times \mathbb{R} : \mathbb{R}^6$ be a solution of the Maxwell's equations given by

$$W_e(X, t) = \cos\left(2\pi(V \cdot X - t)\right) \begin{pmatrix} E_0 \\ H_0 \end{pmatrix}$$

where E_0, H_0 and V are unit vectors such that (E_0, H_0, V) is an orthonormal basis of $\mathbb{R}^3$.

We consider the domain $\Omega = [0, 1]^3$ made of $N \times N \times N$ cubic cells. The spatial step h equals N^{-1}.

The fields are initialized with the exact solution and the exact solution is applied on the boundaries of the domain.

We compute the L^2 norm of the difference between the computed field and the exact solution after a fixed time of 1.0s for $N \in \{3, 4, 5, 6, 8, 10\}$ and $d \in \{1, 2, 3, 4\}$.

The figure 1 represents the logarithm of the error in function of the logarithm of the spatial step.

We obtain the following numerical convergence orders :

- $d = 1$: 1.7
- $d = 2$: 3.1
- $d = 3$: 3.8
- $d = 4$: 4.9

5.2. **CPML test**

This test case allows us to compare the reflexions induced by the Silver-Müller boundary condition and the cPML absorbing layers, see [7, 8].

We consider the domain $\Omega = [0, 1]^3$. We apply on the faces $y = 0$, $y = 1$, $z = 0$ and $z = 1$ a metallic boundary condition defined by

$$E \times n = 0$$

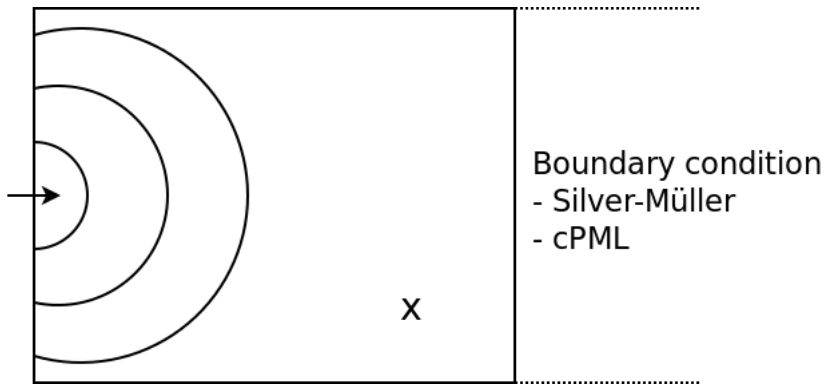

FIGURE 2. cPML test case

where n is a normal vector to the face.

The final time of the simulation is 20ns and we use an approximation of order $d = 2$.

On the face x=0, we apply a metallic boundary condition excepted on a square $(x, z) \in [0.3, 0.7]^2$ where we inject an incident field. The last condition is not physical but only intended at generating an incident field in the domain Ω such that the electric field is not tangent to the face x=1.

On the face x=1, we apply the two boundary conditions that we want to compare. See figure 2.

First, we generate a reference solution. We enlarge the domain of computation in the direction x=1. The domain becomes $\Omega_{ref} = \Omega \cup [1, 5] \times [0, 1] \times [0, 1]$. We apply a metallic boundary condition on the face x=1. Now, in Ω, the reflexion of the field on the faces x=1 will not appear during the simulation. We denote by W_{ref} the solution obtained in this case.

Then, we compute the field in Ω with a Silver-Müller boundary condition on the faces x=1. We denote by W_s the solution in this case.

Finally, we add to the domain Ω five cPML layers on the side $x = 1$. We denote by W_{cpml} the solution in this case.

The cPML profile describing the increase of the conductivity is defined by

$$p(x) = 20x^3 \left(6x^2 - 15x + 10\right)$$

where $x \in [0, n_{layer}]$ describe the depth in the cPML. This profile had been choosen in accordance with numerical experiments comparing differents profiles. We choose the profile which induces the lowest reflexion.

With this profile, the conductivity and its derivative vanishes at the interface between the real domain and the cPML and equals 20 at the end of the cPML.

The figure 3 represents the component E_x of the electric field at the point $(x, y, z) = (0.71, 0.83, 0.5)$.

The Silver-Müller boundary condition reflects more than 10% of the incident field and the the cPML layers reflects about 0.03% of the incident field.

This test case was perform on a GPU GeForce GTX 295. 1200 time iterations have been computed in 14.8s. This represent about 25 GFlops.

5.3. **MPI scalability**

The objective of this test case is to evaluate the MPI scalability of the computation.

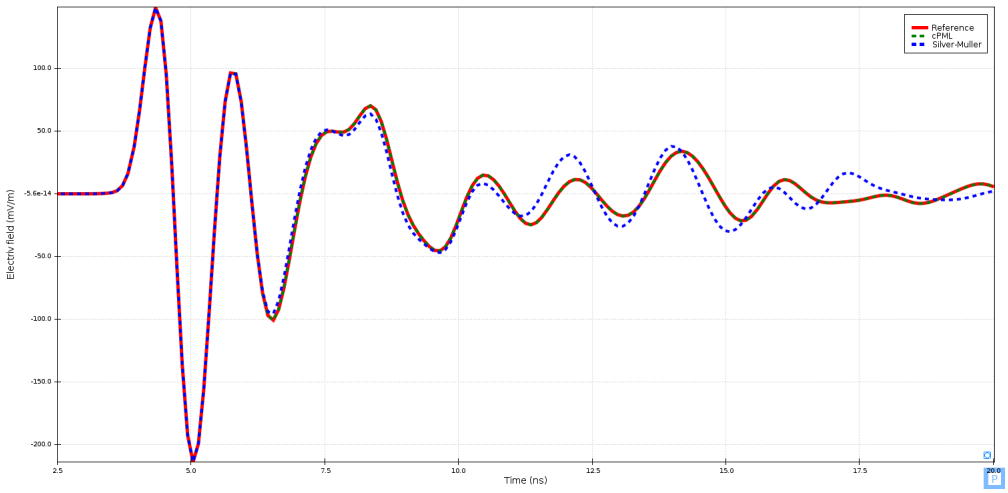

FIGURE 3. Comparison of the component E_x at $(0.71, 0.83, 0.5)$

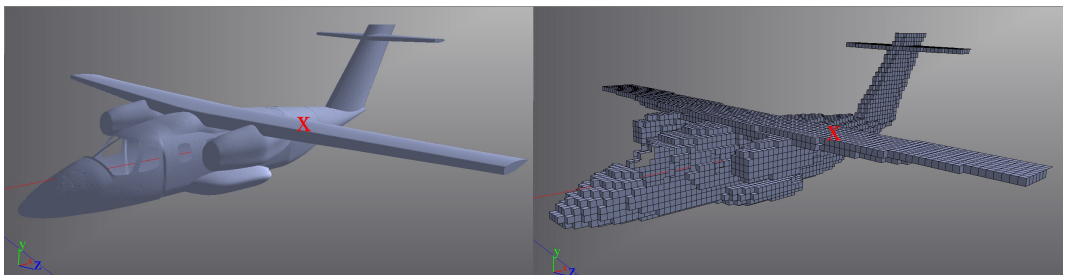

FIGURE 4. Generic ariplane.
Left : triangular mesh. Rigth : hexaedral mesh obtain from srtuctured mesh.

We consider a mesh made of $528,840$ elements with an order 2 approximation. This represents $14,278,680$ degree of freedom. The mesh represent a generic airplane, see figure 4, illuminated by a plane wave with Gaussian profile polarized in E_y.

Currently, we have not yet developed a tools which allow us to directly generate a mesh made of hexahedrons from a triangular surface mesh. So, we first generate a structured mesh from a triangular surface mesh. Then, we approximate this mesh by a hexahedrons mesh. We obtain a mesh which is not perfectly suited to CLAC software. But, here, we only want to evaluate the MPI scalability.

We compute this case with one and four NVIDIA Tesla M2090 GPU and compare the average time of simulation per iteration.

With one GPU we obtain a computation time per iteration of 1.25s which represents approximately 79 Gflops. With 4 GPUs we obtain 0.378s which represents 261 Gflops. We obtain a speedup of 3.3.

We consider now the same case with a geometry described with $3,337,875$ hexaedrons. We use 8 GPUs to perform the computation. The simulation does not fit into a single GPU memory.

The figure 5 represents the norm of the surface current$|n \times H|$ on the body of the airplane at time 51 ns.

We obtain a time per iteration of 1.73s which represents 362 Gflops.The CLAC project is motivated by the acceleration of an existing already well-optimized sequential software using the same algorithm. With our current GPU otimizations we have accelerated the computations by of factor of 5 to 10.

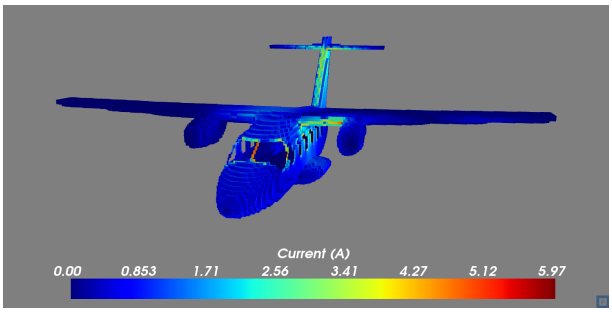

FIGURE 5. Norm of the current on the fuselage at time 51ns.

In this test case we spend about 30% of the computation time in the memory transfers between the CPU and the GPU and about 20% in the MPI communications.

CONCLUSION

We have presented CLAC, an electromagnetic simulation tool. This tool is intended for industrial practical computations. It is accelerated by two parallelism technologies: MPI and OpenCL, which allows to run CLAC on GPU clusters.

Current works are devoted to several improvements of CLAC. We intend to:

- hide the communication latency by performing computations inside the sub-domains while boundary data are exchanged;
- create new zone structures with well-aligned data in order to achieve higher memory bandwidth, where it is possible;
- create new structures for zone interfaces, in order to implement more general transfers between zones.

REFERENCES

[1] F. Bourdel, P. Mazet, and P. Helluy. Resolution of the non-stationary or harmonic Maxwell equations by a discontinuous finite element method. application to an e.m.i. (electromagnetic impulse) case. In 10th international conference on computing methods in applied sciences and engineering, Paris, february 11-14, pages 1–18. Nova Science Publishers, Inc., New York, 1992. http://www-irma.u-strasbg.fr/~helluy/ADMIN/CV/inria92.pdf

[2] G. Cohen, X. Ferrières and S. Pernet. A spatial high order hexahedral discontinuous Galerkin method to solve Maxwell's equations in time-domain. J. Comput. Phys., vol. 217, no. 2, pages 340–363, 2006.

[3] A. Crestetto and P. Helluy. Resolution of the Vlasov-Maxwell system by PIC Discontinuous Galerkin method on GPU with OpenCL.. ESAIM Proc. Vol. 38, pp 257 – 274. 2012. http://dx.doi.org/10.1051/proc/201238014

[4] A. Klockner, T. Warburton, J. Bridge, J.S. Hesthaven, Nodal discontinuous Galerkin methods on graphics processors, Journal of Computational Physics, Volume 228, Issue 21, 20 November 2009, Pages 7863-7882

[5] P. Lesaint and P.-A. Raviart. On a finite element method for solving the neutron transport equation. 1974

[6] K. Yee. Numerical solution of initial boundary value problems involving Maxwell's equations in isotropic media. IEEE Transactions on Antennas and Propagation, 14(3) :302–307, 1966.

[7] J.P. Berenger. A Perfectly Matched Layer for the Absorption of Electromagnetic Waves. Journal of Computational Physics, Vol. 114, pp 185-200, 1994.

[8] M. Lassas, J. Liukkonen, and E. Somersalo. Complex Riemannian metric and absorbing boundary conditions. Journal de mathématiques pures et appliquées, 80(7), pp 739-768, 2001.