Towards real-time finite element simulation on GPU

V-Q Dinh[1], Yves Marechal[1, 2], Gerard Meunier[1, 2]

[1] Univ. Grenoble Alpes, G2Elab, F-38000 Grenoble, France,
[2] CNRS, G2Elab, F-38000 Grenoble, France

In this paper, we introduce a parallel assembly technique on NVIDIA CUDA GPUs for finite element method (FEM) applied in the magnetic field computation. Basically, each thread calculates the integration associated with an element. To avoid memory conflicts, we introduced a fast procedure based on sorting and rearrangement of elementary non-zero (NZ) entries. Finally, a reducing process is executed to assemble NZ in the stiffness matrix. This algorithm does not require any preprocessing on mesh but also take advantage of parallel computing power of GPU. In our tests, using this parallel assembly improved the speed assembling up to 20x times faster.

***Index Terms*—CUDA, GPU, FEM, assembly, sorting**

I. Introduction

The finite element method is a well-used numerical technique for finding approximate solutions in the electromagnetic computation field. We consider the problem of finding a function u: $\Omega \rightarrow R$ that satisfies:

$$\mathcal{L}(u) = f \qquad (1)$$

in the domain Ω with some boundary conditions on the boundary $\partial\Omega$. In which, Ω is the spatial n-dimension domain R^n, $\mathcal{L}$ is a general linear differential operator, f is a source term. The FEM begins by the subdivision of domain Ω into a set of finite elements and defines an approximate function, note $\underline{u}$, based on the nodal variables $\underline{u_i}$ associated with N_i geometric nodes: $u \approx \underline{u} = \sum_{i=1}^{N_i} h_i(x)\underline{u}_i$

$\underline{h_i}(x)$ denoting spatial distributions associated with the approximation value $\underline{u_i}$ on i^{th} node.

Equation (1) is transformed by FEM into a linear system of equations such as:

$$A\underline{u} = F \qquad (2)$$

A is the stiffness matrix, **F** is the forcing vector, $\underline{\mathbf{u}}$ is the approximation vector of unknown nodal variables.

The objective of this work is to push computation on GPU as far as possible in the pursuit of "pseudo real-time" simulation goal. In this paper, we will concentrate on the assembly process. In the full paper, solving process will be equally presented.

II. General Consideration On Assembly

In practice, each entry A_{ij} of the matrix **A** is assembled from contributions of all elements that contain both nodes $\underline{u_i}$ and $\underline{u_j}$ and similarly each entry f_i of the vector **F** is assembled from all elements that contain $\underline{u_i}$.

$$A_{[i,j]} = \sum_{\substack{e,k,l \\ E(e,k)=i \\ E(e,l)=j}} A^e{}_{[k,l]}; F_{[i]} = \sum_{\substack{e,k \\ E(e,k)=i}} F^e{}_{[k]} \qquad (3)$$

$\mathbf{A^e}$ and $\mathbf{F^e}$ are the elementary matrix and vector based on the integration of operator $\mathcal{L}$ and function f in the space of element. **E** is the element connectivity data, for instance, **E**(e,k) corresponds to i^{th} global degree of freedom (DOF) that associates with the k^{th} node of e^{th} element. The matrix A is usually sparse with a few non-zero entries on each row.

There are a lot of proposed strategies for GPU assembly: mesh coloring to partition elements into non-overlapping sets [4], graph partitioning and reduction list, local matrices [3], assembly by NZ using global, local, shared memory [1], patching mesh [2]. These approaches generally use a step of pre-computing or reorganization of the nodes and elements of the mesh. Consequently, this affects the overall performance. Our alternative method does not require any preprocessing on mesh but uses a sorting method by NZ's index of rows for separating parallel threads.

III. Assembly Parallel On GPU

In this paragraph, we illustrate the assembly algorithm of stiffness matrix A in the proposed diagram below:

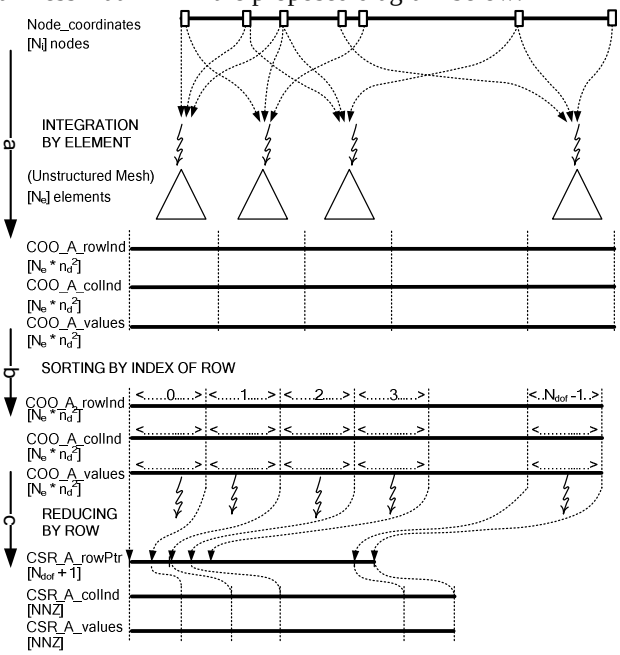

Fig.1. The global assembly algorithm uses 3 kernels CUDA: integration by element, sorting by the index of rows, reducing.

First, in step "a", we assign each parallel thread to compute the integration of each element. The results are saved into a temporary matrix **A** described in the coordinate (COO) format [8] with 3 arrays: its row index, column index and value. The integration by element is straightforward and ensures load balancing between parallel threads. So we kept the original code and only changed the way to access database that are handled on global memory of GPU. Since

the mesh is unstructured, the coalesced reading on global memory [9] is impossible if not arrangement the database. However, the matrix **A** is saved on the global memory by a coalesced way. With a mesh of N_e elements and n_d DOFs associated per element, so the dimension of an array of **A** is $(N_e * n_d^2)$.

Next, in step "b", we use a parallel sorting method for rearranging NZ entries by its row index. In general, the sorting implementation is suitable on GPU and there are a lot of effective sorting methods proposed in the literature. In our test, we used the radix sorting algorithm in the Cudpp library [6] coded by jCuda [7]. As a result, the NZ entries with same index of row are saved adjacently on the memory that facilitates the coalescent access in the next.

In step "c", each parallel thread is assigned to a memory segment that corresponds to a row and performs a reducing process if NZ entries of that row have the same index of column. In this step, the share memory on GPU facilitates the fast accessing on the data and improves the reduction. Finally, we obtain the stiffness matrix **A** described in the Compressed Row Storage (CRS) format [9] by 3 arrays: row pointer, column index and values. The vector F is computed by a similar way.

IV. Test

We perform some tests in the linear static magnetic fields [5] in which the equations are written as:

$$\nabla \times H = J_0$$
$$\nabla . B = 0 \qquad (4)$$
$$B = \begin{cases} \mu_0 H & \text{in air} \\ \mu_r \mu_0 H & \text{in magnetically linear material} \end{cases}$$

The geometries and physical properties are described in the Fig.2 and Fig.3 below:

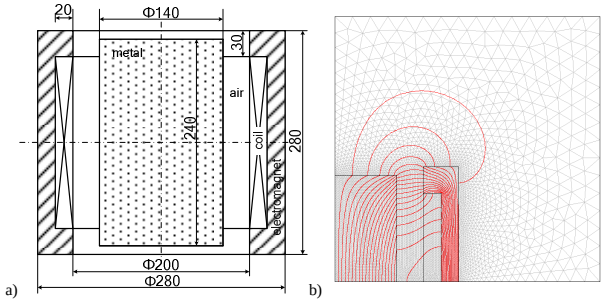

Fig.2. A electrical cylindrical oven's geometry (a): metal $\mu_r = 100$; coil with a continuous current density 1A/mm^2, a electromagnet $\mu_r = 1000$; Potential iso-value in a quarter of domain (b)

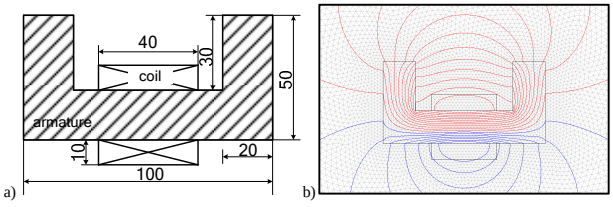

Fig.3. An actuator's geometry (a) armature $\mu_r = 1000$; coil 5A/mm^2; Potential iso-value in the domain (b)

The tests are applied on the meshes with variable density of nodes based on these domains. The information of meshes is shown in the Table I. About the hardware, all experiences are executed in a platform equipped a CPU Intel Xeon 2.67 GHz tested with a single core and a GPU NVIDIA Tesla C1060 240 cores CUDA 1.3 GHz. The original code of FEM on CPU is developed in JAVA v7.4 environment. Therefore, the parallel code of the assembly by CUDA GPUs is ported into JAVA by jCuda. The databases on the GPUs memory are saved in single precision (32 bit) to adapt to maximal performance of GPUs [9]. The results of stiffness matrix **A** and vector **F** are compared between CPU and GPU program for checking the correctness with the relative norm of difference about 10^{-6}.

Table I
The Information of Meshes With Number of DOFs, Number of Elements and Number of NNZ Entries

Mesh	N_{DOF}	N_e	NNZ
Oven_1	6,792	13,608	47,132
Oven_2	23,373	46,762	162,917
Actuator_1	11,211	22,580	78,151
Actuator_2	32,945	65,984	230,425
Actuator_3	61,617	123,392	430,993

The comparison of running times on CPU and GPU is shown in the Table II:

Table II
Assembly Performance, In Millisecond

Mesh	GPU	CPU	Speed up
Oven_1	15	172	11.47
Oven_2	45	1,021	22.69
Actuator_1	31	483	15.58
Actuator_2	47	1,014	21.57
Actuator_3	109	1,872	17.17

V. Conclusion

Our method improves the speed assembly FEM method and is suitable not only for the electromagnetic field, but also for many other fields. The advantages of this assembly algorithm are the load balancing, the suitable implementation of sorting and the scalability. One drawback is the large requirement of GPU global memory. On perspective, the mesh can be divided into sub-mesh to partially perform the assembly in the shared memory on GPU. Considering the solving process, the work in the full paper shows equivalent speed up result.

References

[1] C. Cecka, A. J. Lew, and E. Darve, "Assembly of finite element methods on graphics processors," International journal for numerical methods in engineering, vol. 85, no. 5, pp. 640–669, 2011.

[2] Z. Fu, T. James Lewis, R. M. Kirby, and R. T. Whitaker, "Architecting the finite element method pipeline for the GPU," Journal of Computational and Applied Mathematics, vol. 257, pp. 195–211, Feb. 2014.

[3] I. Kiss, S. Gyimothy, Z. Badics, and J. Pavo, "Parallel Realization of the Element-by-Element FEM Technique by CUDA," IEEE Transactions on Magnetics, vol. 48, no. 2, pp. 507–510, Feb. 2012.

[4] J. Zhang and D. Shen, "GPU-Based Implementation of Finite Element Method for Elasticity Using CUDA," 2013, pp. 1003–1008.

[5] M. Kuczmann, "Potential Formulations in Magnetics–App lying the Finite Element Method," Lecture notes, Laboratory of Electromagnetic Fields," Szechenyi Istvan", University, Gyor, Hungary, 2009.

[6] [Online] www.gpgpu.org

[7] [Online] www.jcuda.com

[8] A. Dziekonski, P. Sypek, A. Lamecki, and M. Mrozowski, "Finite element matrix generation on a GPU," Progress In Electromagnetics Research, vol. 128, pp. 249–265, 2012.

[9] [Online] http://docs.nvidia.com/cuda/cuda-c-programming-guide