Parallel realisation of the Element-by-Element FEM technique by CUDA

Imre Kiss, Szabolcs Gyimóthy and József Pávó

Budapest University of Technology and Economics, Egry József utca 18, H-1111 Budapest, Hungary
E-mail: kiss@evt.bme.hu

***Abstract*—In this paper, the utilization of Graphical Processing Units (GPUs) for computations on unstructured meshes such as those in Finite Element Methods (FEM) will be shown. The Element-by-Element (EBE) technique is a long since known method that can be applied to perform the parallel execution of a Conjugate Gradient (CG) type iterative solver. Instead of assembling the global system matrix, NVIDIA's parallel computing solution, the Compute Unified Device Architecture (CUDA) will be used to perform the required element-wise computations parallelly. Since the element matrices will not be stored (which is unfeasible for linear problems, but quite normal for non-linear ones), the memory requirement of this technique is extremely low. It will be shown that this *low-store* and *high-computational* needed technique is better suited for GPUs than those facilitating the massive manipulation of large data sets.**

***Index Terms*—CUDA, EBE, GPGPU, parallel FEM**

I. Introduction

The Finite Element Method is nowadays one of the most frequently used technique for engineering analysis of complex, real-word applications of both linear and non-linear types. The basics of the method are very well known [1], hence here we only recall those properties that are important for our investigations.

Let us consider the well known linear equation system of the form

$$\mathbf{A}\mathbf{u} = \mathbf{b} \tag{1}$$

which is obtained by the FEM approximation of a Partial Differential Equation (PDE). The solution of (1) – especially for large size problems – is traditionally obtained using iterative solvers, like the variants of the gradient type methods (e.g. the Bi-Conjugate Gradient (BiCG) method).

Most iterative methods for the solution of (1) can be described by the general formula

$$\mathbf{u}_{k+1} = \mathbf{u}_k - \mathbf{Q}^{-1}(\mathbf{A}\mathbf{u}_k - \mathbf{b}) \tag{2}$$

where $\mathbf{Q}$ is a non-singular preconditioning matrix and $k = 1, 2, \ldots, N$ is the iteration number. Since the most computation intensive part in a CG iteration is the matrix-vector multiplication (MxV) of $\mathbf{A}\mathbf{u}_k$, due to the independent computations its parallel execution can be easily designed [2].

Among the many possibilities to carry out parallel execution of the entire CG iteration (2), one may find the Element-by-Element (EBE) technique [3]. Its main advantage is the extremely low memory consumption, since it does not assemble the global system matrix, but rather traces back the manipulation to the level of element matrices (see Section III-A).

Although many methods accelerating the FEM are already implemented on GPUs [4], [5], [6], these usually suffer from the strict limitation of available memory. As large scale FEM problems need large storage capacity, the several GB of available memory on GPUs must be continuously cached to the global system memory through the relatively slow bus system.

II. Aim of the work

As the gap between bus speed and computation density increases, codes which use the accelerator design (that is, only computation intensive parts of the program are executed on the GPU) will fall behind codes that take full advantage of it [7]. The latter perform all the necessary computations on the GPU, in contrary to the accelerator design, where e.g. only the MxV operation is performed on precomputed and transferred elements. Based on the precedings, the cost of data transfers is increasing relative to the amount of computation.

The aim of this paper is therefore to extend the scope of problems GPUs can effectively handle, by avoiding large scale data transfers. Relying on the fact it is cheaper to recompute element matrices than continuously cache the global matrix between GPU and system memory, the EBE technique will be revisited to solve (almost) arbitrarily big problems, which traditionally requires substantial data transfer, and hence is time consuming.

III. Element-by-element method

A. Element-wise computations

A typical finite element assembly program relies on given element subroutines to compute an element matrix $\mathbf{A^e}$. These subroutines vary on the type of the PDE that must be solved as well as on the type of the applied basis functions. The constructed element matrices are then expanded to the size of the full global equation system (with other entries being zero) corresponding to some *global numbering*, and are summarized as

$$\mathbf{A} = \sum_{e=1}^{E} \hat{\mathbf{A}}_\mathbf{e} \tag{3}$$

where E is the number of elements, and $\hat{\mathbf{A}}_\mathbf{e}$ is the contribution of the expansion of the element matrix $\mathbf{A}_\mathbf{e}$. In contrary to the sparse expanded element matrix, $\mathbf{A}_\mathbf{e}$ is dense with the size of $n_e \times n_e$ (n_e being the local degrees-of-freedom).

Substituting (3) into (2), one obtains

$$\mathbf{A}\mathbf{u}_k = \left(\sum_{e=1}^{E} \hat{\mathbf{A}}_\mathbf{e}\right)\mathbf{u}_k = \sum_{e=1}^{E}\left(\hat{\mathbf{A}}_\mathbf{e}\hat{\mathbf{u}}_k^e\right) = \sum_{e=1}^{E}\widehat{\mathbf{A}_\mathbf{e}\mathbf{u}_k^e} \quad (4)$$

where $\hat{\mathbf{u}}_k^e$ is an "expansion" with non-zeros only at positions corresponding to element e, $\mathbf{u}_k^e$ is the dense representation of $\hat{\mathbf{u}}_k^e$, and the "hat" operator corresponds to global expansion. Therefore individual products in (4) can be computed at element level as dense matrix-vector multiplications [3].

Since the global system matrix is never assembled, there is no need for the clever, hence computation intensive global numbering of the nodes (and edges), which is critical to have a low bandwidth matrix. The lack of global numbering will also ease up handling of adaptively refined unstructured meshes, because the summation in (4) can be computed in any order.

B. Parallel processing of EBE

Because GPUs are designed for total computational throughput rather than fast execution of serial calculations, they have the potential to dramatically speed-up scientific computing applications over multi-core CPUs. To achieve high computational throughput, GPUs have hundreds of lightweight cores and execute tens of thousands of threads simultaneously.

The GPU parallel processing of (4) can be done efficiently, since the assembly of local element matrices is a computationally intensive task with a low amount of required information. As the number of element matrices can be arbitrarily large, and they are independent from each other, their assembly can take full advantage from the given architecture.

On shared memory architectures like the GPU (similar to distributed memory architectures), an important question is how the partial products are summarized. If the different threads have significantly different number of elements to deal with, the ill-balanced partitioning will result in the underutilization of the device.

The key challenge in the global update is to ensure that contributions from two local nodes, associated with an identical global node, do not update some global value from different threads. This leads to the concept of coloring, which has successfully been used previously in the context of finite elements on supercomputers [8], [9] and through which the dependencies between mesh points can be suppressed.

C. Matrix-storage free approach

The lack of assembling the global system matrix makes the method applicable in GPU computing environments, where the amount of stored data is critical, but raises several problems, too. The first one is related to preconditioners, which traditionally need the system matrix in assembled form. To overcome this problem, one may use some sophisticated element-by-element preconditioner [9], [10].

The second problem is related to the required computational demand. Since element matrices are not stored, they must be recomputed in each iteration. Note that it is not needed for linear problems, and seems to be a waste of time. On the other hand, for nonlinear problems the re-computation of element matrices is required anyway, and the same holds if we use some mesh refinement/reduction techniques during the iteration [11]. Therefore CUDA parallel EBE may perform best for non-linear problems and dynamic meshing, significant acceleration of linear problems is also expected though.

IV. Floating point precision

In most cases the assembling of local element matrices and the computations on them require double precision representation of floating numbers. Although many GPUs are capable nowadays to perform such computations, their throughput is much more moderate than for single precision computations. To fully utilize GPUs, and also to fulfill the requirements of double precision computation, the so-called *mixed precision* iterative refinement [12] will be applied.

Although the GPU implementation of the mixed precision iterative refinement is already carried out [6] and proved to be more efficient than full-double precision computations, the main drawback of the presented method is its need for the global system matrix.

V. Numerical examples

To demonstrate the capabilities of EBE parallel computing of FEM on GPUs, numerical examples will be given for different sizes and types for both linear and non-linear properties. Pure single and double precision computations will be carried out, and the effect of using the mixed precision iterative refinement compared with them will be shown.

References

[1] J.-M. Jin, *The Finite Element Method in Electromagnetics*, 2nd ed. Wiley-IEEE Press, Jun 2002.

[2] G. F. Carey, E. Barragy, R. McLay, and M. Sharma, "Element-by-element vector and parallel computations," *Commun. appl. numer. methods*, vol. 4, no. 3, pp. 299–307, 1988.

[3] G. F. Carey and B.-N. Jiang, "Element-by-element linear and nonlinear solution schemes," *appl. num. meth.*, vol. 2 (2), pp. 145–153, 1986.

[4] J. Bolz, I. Farmer, E. Grinspun, and P. Schröoder, "Sparse matrix solvers on the GPU: conjugate gradients and multigrid," *ACM Trans. Graph.*, vol. 22, pp. 917–924, July 2003.

[5] C. Cecka, A. Lew, and E. Darve, "Introduction to assembly of finite element methods on graphics processors," *IOP Conference Series: Materials Science and Engineering*, vol. 10, no. 1, p. 012009, 2010.

[6] A. Cevahir, A. Nukada, and S. Matsuoka, "Fast conjugate gradients with multiple GPUs," in *ICCS 2009*, G. G. van Albada, J. Dongarra, and P. Sloot, Eds., 2009, vol. 5544, pp. 893–903.

[7] I. Kiss, J. Pávó, and S. Gyimóthy, "Acceleration of moment method using CUDA," in *Proceedings, IGTE 2010*, 2010, (to apper soon).

[8] C. Farhat and L. Crivelli, "A general approach to nonlinear FE computations on shared-memory multiprocessors," *Computer Methods in Applied Mechanics and Engineering*, vol. 72, no. 2, pp. 153–171, Feb. 1989.

[9] A. J. Wathen, "An analysis of some element-by-element techniques," *Computer Methods in Applied Mechanics and Engineering*, vol. 74, no. 3, pp. 271–287, Sep. 1989.

[10] G. Golub and Q. Ye, "Inexact preconditioned conjugate gradient method with inner-outer iterations," *SIAM J. on Scientific Computing*, vol. 21(4), pp. 1305–1320, 2000.

[11] S. Gyimothy and I. Sebestyen, "Symbolic description of field calculation problems," *Magnetics, IEEE Transactions on*, vol. 34, no. 5, pp. 3427 –3430, 1998.

[12] A. Buttari, J. Dongarra, J. Kurzak, P. Luszczek, and S. Tomov, "Using mixed precision for sparse matrix computations to enhance the performance while achieving 64-bit accuracy," *ACM Trans. Math. Softw.*, vol. 34, pp. 17:1–17:22, July 2008.