High locality and increased intra-node parallelism for solving finite element models on GPUs by novel element-by-element implementation

Imre KISS[1] Zsolt BADICS[2] Szabolcs GYIMÓTHY[1] József PÁVÓ[1]

e-mail: kiss@evt.bme.hu

[1]Budapest University of Technology and Economics,
Egry József utca 18, H-1111 Budapest, Hungary

[2]Tensor Research LLC,
100 Wildwood Road, Andover, MA 01810, USA

HPEC '2012

Outline

- Overview
  - Element-by-Element method
  - Disassembling MxV operation
  - Further properties of the EbE FEM
- Implementation
  - Storage technique
  - Breaking MxV
  - EbE BiCG solver
  - CUDA implementation
- Results
  - Problem statement
  - Results
- Conclusion, Outlook

Overview of the FEM method

In computational electromagnetics, Finite Element Method (FEM) is a one of the most commonly used technique to perform complex analysis.

The linear equation system resulting from the FEM approximation of a partial differential equation (PDE) is usually written in the form

$$\mathbf{Ax} = \mathbf{b},$$

where **A** is the so-called system matrix, **x** is the vector of unknowns and **b** on the right hand side (RHS) represents the excitation.

$\Rightarrow$ Since **A** is sparse, iterative solvers are usually preferred over direct ones

- treatable problem size is limited by available memory (even in sparse)
- the memory bound is even more restrictive when using GPUs

Overview of the FEM method

In computational electromagnetics, Finite Element Method (FEM) is a one of the most commonly used technique to perform complex analysis.

The linear equation system resulting from the FEM approximation of a partial differential equation (PDE) is usually written in the form

$$\mathbf{Ax} = \mathbf{b},$$

where **A** is the so-called system matrix, **x** is the vector of unknowns and **b** on the right hand side (RHS) represents the excitation.

$\Rightarrow$ Since **A** is sparse, iterative solvers are usually preferred over direct ones

- treatable problem size is limited by available memory (even in sparse)
- the memory bound is even more restrictive when using GPUs

GPU implementations for FEM

Several aspects implementing FEM on GPUs have been already investigated.

These usually tried to fit the "traditional" FEM formulation to this unique architecture (GPU) by "outsourcing" a specific part of the algorithm only

- accelerating the sparse matrix-vector (spmv) kernel which is the basis of every iterative type solver,
- executing only the assembly step on the GPU,
- utilizing some domain decomposition technique to fit the sliced (partial) problems into the limited GPU memory.

In general, these accelerator type designs could not utilize GPUs efficiently.

The underlying problem is the large memory footstep of the system matrix
$\Rightarrow$ usually can only be resolved by significant amount of data movement (slow).

GPU implementations for FEM

Several aspects implementing FEM on GPUs have been already investigated.

These usually tried to fit the "traditional" FEM formulation to this unique architecture (GPU) by "outsourcing" a specific part of the algorithm only

- accelerating the sparse matrix-vector (spmv) kernel which is the basis of every iterative type solver,
- executing only the assembly step on the GPU,
- utilizing some domain decomposition technique to fit the sliced (partial) problems into the limited GPU memory.

In general, these accelerator type designs could not utilize GPUs efficiently.

The underlying problem is the large memory footstep of the system matrix
$\Rightarrow$ usually can only be resolved by significant amount of data movement (slow).

The Element-by-Element FEM approach

The Element-by-Element (EbE) technique is a revised version of the FEM method to overcome on the memory bound problem.

Assembly of element matrices to the system matrix is a linear operation, hence

- $\Rightarrow$ certain operations with the system matrix (e.g. MxV, VxV, Vxc) can be traced back to the level of finite elements
- $\Rightarrow$ thus can be converted to calculations with the individual element matrices $\mathbf{A}_e$, appearing in the elementary equations

$$\mathbf{A}_e \mathbf{x}_e = \mathbf{b}_e.$$

Iterative solvers can be decomposed into a sequence of matrix-vector products and inner products of vectors (both suitable to the EbE concept)

- $\Rightarrow$ do not store the element matrices as traditional methods do with the system matrix, rather recompute them in each iteration
- $\Rightarrow$ transforms a highly memory dependent problem to a massively computational dependent one, which in turn can be efficiently parallelized

Disassembling matrix manipulations I

- The FEM assembling procedure relies on some functions to generate the element matrix $\mathbf{A}_e$ and the RHS $\mathbf{b}_e$.
- Computed element matrices and RHS vectors are assembled to form the global system matrix $\mathbf{A}$ and RHS $\mathbf{b}$.
- Let this assembly step be represented by an operator $\mathcal{M}$ as

$$\mathcal{M} : \mathbf{A}_e\mathbf{x}_e = \mathbf{b}_e, \;\; e = 1, \ldots, m \qquad \Longrightarrow \qquad \mathbf{A}\mathbf{x} = \mathbf{b}$$

$$\begin{aligned} \mathbf{A} &= \mathcal{M}(\mathbf{A}_e) = \sum_{e \in \mathcal{E}} \mathbf{C}_e^T \mathbf{A}_e \mathbf{C}_e \\ \mathbf{b} &= \mathcal{M}(\mathbf{b}_e) = \sum_{e \in \mathcal{E}} \mathbf{C}_e \mathbf{b}_e \end{aligned}$$

where $\mathcal{E}$ is the set of elements, and matrix $\mathbf{C_e}$ represents the transition between *local* and *global* numbering of the unknown variables for the e-th element. Contrary to the sparse global system matrix $\mathbf{A}$, the element matrix $\mathbf{A_e}$ is usually dense.

Disassembling matrix manipulations II

Using this concept, the matrix-vector product, which is the basis of iterative solvers, can be reformulated in terms of element-wise computations as

$$\mathbf{A}\mathbf{x} = \sum_{e \in \mathcal{E}} \mathbf{C}_e^T \mathbf{A}_e \mathbf{C}_e \mathbf{x} = \sum_{e \in \mathcal{E}} \mathbf{C}_e^T \mathbf{A}_e \mathbf{x}_e = \mathcal{M}(\mathbf{A}_e \mathbf{x}_e).$$

- $\Rightarrow$ product of an assembled global matrix and a vector is equivalent with the assembled vector of the elementary matrix-vector products.
- $\Rightarrow$ the elementary contributions can be accumulated in a vector having the size of the global degrees-of-freedom (DoF)
- $\Rightarrow$ only vectors have to be stored during the computations
- $\Rightarrow$ elementary matrix-vector products can be computed for each element separately, which enables parallel realization.

The inner product of two DoF-sized vectors is also an elementary operation

- $\Rightarrow$ this operation is obviously independent of the mesh structure and connectivity, hence its parallel execution is straightforward.

Further properties

Using the EbE-FEM concept, there are several further advantages compared to the traditional (assembly based) FEM techniques

- global numbering of unknowns and finite elements is not required at all
- due to the lack of assembly, there is no need for an optimized global numbering to obtain a low bandwidth system matrix
- when using some mesh refinement/reduction technique during the iterations, locality for the necessary modifications is also ensured

However it is also notable that there are several disadvantages as well

a) since preconditioning techniques are usually requires the assembled global system matrix, special pre-conditioning methods are required

b) since element matrices are not stored, they must be recomputed in each iteration, which is obviously redundant when dealing with linear problems $\Rightarrow$ this extra computation becomes necessary for non-linear problems

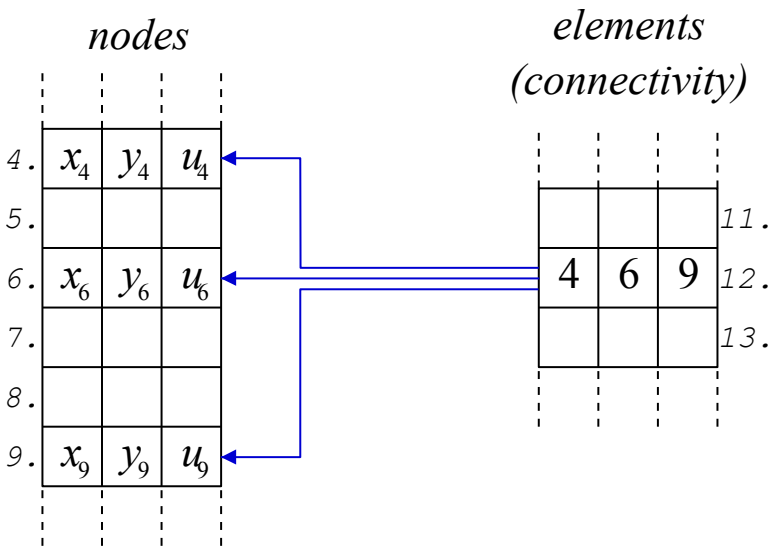
Dynamic storage structure

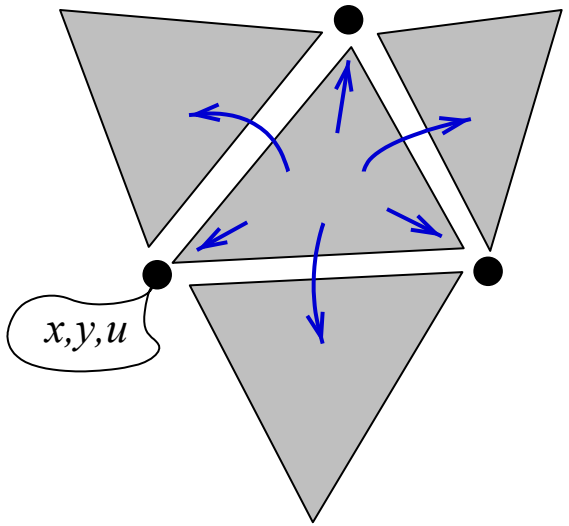
The way the variables are stored gives the real modularity of the EbE method

⇒ contrary to traditional FEM methods using global numbering, here a dynamic storage structure is used instead

⇒ it can be thought as an index array (pointers in the actual implementation) keeping the information how local unknowns correspond to global ones

- such storage pattern is often referred as "spatial data structure"





A global unknown contains all the (self) associated elements of not only the solution vector, but also the global variables required by the iterative solver.

Breaking MxV into element-wise computations

In an EbE implemented BiCG solver computations can be grouped into so-called "EbE" steps and "DoF" steps:

"EbE iteration" refers to the matrix-vector product $\mathbf{Ax} = \mathcal{M}(\mathbf{A}_e\mathbf{x}_e)$

- indicates the computation of the element matrices,
- to avoid race conditions during global updates, the elements are colored with respect to the global unknown variable,
- the iteration goes through all colors serially, and performs the computations on the elements having the actual color in parallel.

"DoF iteration" indicates the computation of the vector-vector products.

- this iteration is performed simultaneously on all $\mathcal{U}$ global unknown.
- "global update" means that the value of a single global variable (related to the BiCG algorithm) is affected. To avoid race conditions, atomic updates are used to access global variables.

BiCG solver in terms of EbE and DoF iterations

EbE BiCG initialization

```
foreach u ∈ U do                                   // DoF iteration
    r^(u) ← 0
    D^(u) ← 0
end
for c = 1 to |colors| do                           // serial loop
    foreach e ∈ E^(c) do                           // EbE iteration
        r_e ← r_e + b_e − A_e u_e
        D_e ← D_e + diag(e)
    end
end
foreach u ∈ U do                                   // DoF iteration
    D^(u) ← 1/D^(u)
    r̃^(u) ← r^(u)
    d^(u) ← D^(u) · r^(u)
    d̃^(u) ← D^(u) · r̃^(u)
    q^(u) ← 0
    q̃^(u) ← 0
end
δ ← 0                                              // global update
foreach u ∈ U do                                   // DoF iteration
    δ ← δ + r^(u) d^(u)                            // global update
end
```

EbE BiCG looping

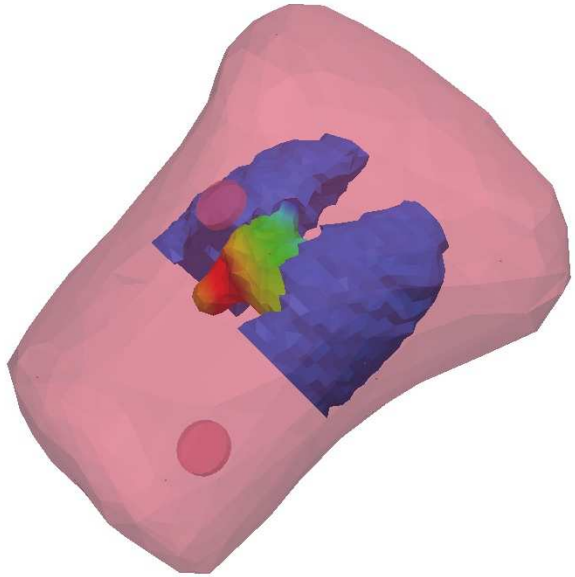
```
while δ > prescribed accuracy do                   // host loop
    for c = 1 to |colors| do                       // serial loop
        foreach e ∈ E^(c) do                       // EbE iteration
            q_e ← q_e + A_e d_e
            q̃_e ← q̃_e + A_e^T d̃_e
        end
    end
    α ← 0                                          // global update
    foreach u ∈ U do                               // DoF iteration
        α ← α + d̃^(u) · q^(u)                      // global update
    end
    foreach u ∈ U do                               // DoF iteration
        x^(u) ← x^(u) + δ/α · d^(u)
        r^(u) ← r^(u) − δ/α · q
        r̃^(u) ← r̃^(u) − δ/α · q̃
    end
    δ̃ ← δ; δ ← 0                                   // global update
    foreach u ∈ U do                               // DoF iteration
        δ ← δ + r^(u) D^(u) r̃^(u)                  // global update
    end
    α ← δ/δ̃                                        // global update
    foreach u ∈ U do                               // DoF iteration
        d^(u) ← D^(u) · r^(u) + α · d^(u)
        d̃^(u) ← D^(u) · r̃^(u) + α · d̃^(u)
        q^(u) ← 0
        q̃^(u) ← 0
    end
end
```

CUDA implementation of the BiCG solver

- In "EbE" iterations, kernels are started serially for each set of elements (colors) with as many threads as elements in the set.
- In "DoF" iterations, a single kernel with a dedicated thread for each vector element operation is started.
- During "global updates", atomic operations are used to avoid race conditions.
- BiCG looping is implemented as a host iteration.
- Only the δ variable is transferred to the host side at every iteration.
- The Jacobi preconditioner is stored associated with the global unknowns locally.
- If multiple GPUs are used, both EbE and DoF iterations can be split, but certain barriers are necessary to propagate all sub-results to all cards.

Test problem: the UTAH torso model

The test problem is a static conduction problem (an ECG forward problem):



$\Rightarrow$ the Laplace equation with spatially varying conductivity is to be solved

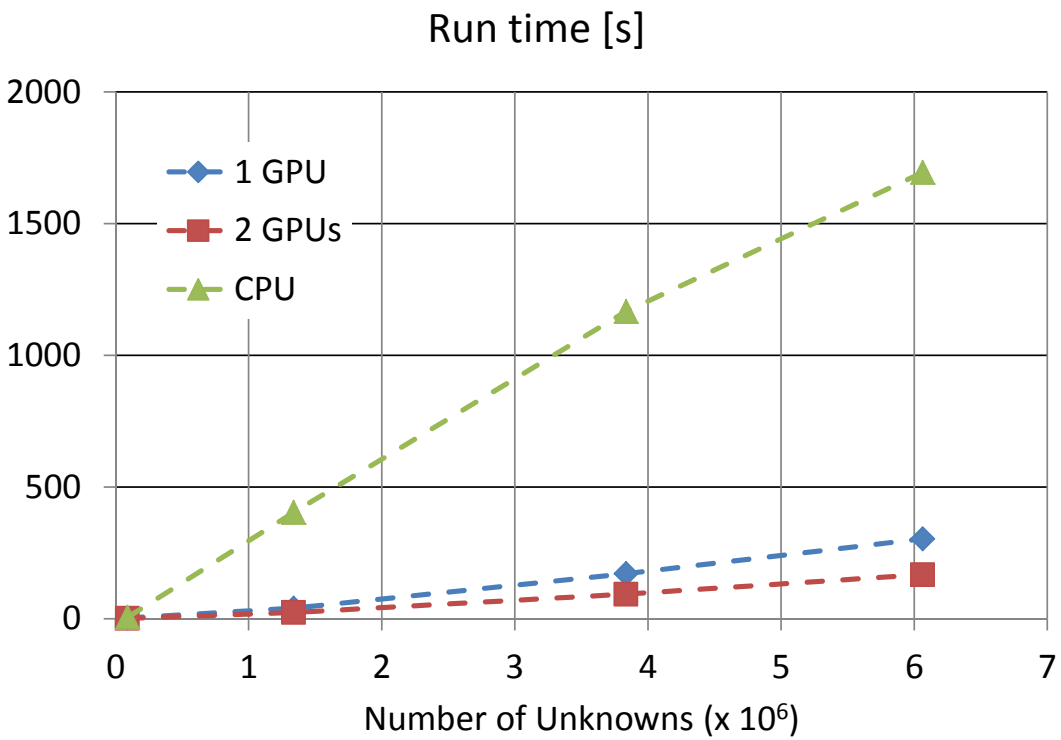
$$\nabla \cdot \sigma \nabla \varphi = 0.$$

- the domain is discretized by tetrahedral elements and linear nodal shape functions are used
- element matrices are computed using analytical expressions
- the global unknowns (DoF) are the φ potential values at mesh nodes

Results I

The computations have been carried out on a HP-XW8600 workstation

- having 128 GB memory
- an NVIDIA GTX 590 GPU card (with two GPU processors on a single board)
- a quad-core Intel Xeon X3440 CPU
- CPU implementation is based on the Intel®Math Kernel Library (MKL).



Results II (continued, detailed)

Test case	**#1**	**#2**	**#3**	**#4**
No. of tetrahedra	$560K$	$6,559K$	$18,884K$	$29,772K$
No. ov unknowns	$91K$	$1,339K$	$3,836K$	$6,064K$
GPU implementation				
No. of iterations	322	671	978	1 111
Colors used	41	45	53	56
Memory [MByte]	12	213	613	968
Runtime (1 GPU) [s]	2.8	40.3	171.3	303.8
Runtime (2 GPUs) [s]	1.7	23.6	93.4	167.6
CPU implementation				
No. of iterations	79	248	338	391
Memory [MByte]	1,564	5,251	13,645	19,371
Runtime [s]	5.9	403.4	1,166.6	1,694.6

Conclusion, Outlook

Conclusion

- Since the EbE-FEM requires no communication on the level of processing cores (highly localization) the hardware utilization can be maximized.
- This design was found to utilize GPUs more effectively than accelerator designs why also competitive with modern multi-CPU implementations.
- Excellent memory utilization since does not depend on the traditional "assemble-and-solve" style.

Outlook

- Preliminary multi-GPU results indicate good scalability of the method, which could easily be extended to GPU clusters.
- Real advantage of the method is exploited when treating non-linear problems or when the mesh structure changes during the computation.