Progress In Electromagnetics Research, Vol. 128, 249–265, 2012

FINITE ELEMENT MATRIX GENERATION ON A GPU

A. Dziekonski*, P. Sypek, A. Lamecki, and M. Mrozowski

Wireless Communication Engineering (WiComm) Center of Excellence, Department of Microwave and Antenna Engineering, Faculty of Electronics, Telecommunications and Informatics, CUDA Research Center for Computational Electromagnetics, Gdansk University of Technology, Gdansk 80-233, Poland

Abstract—This paper presents an efficient technique for fast generation of sparse systems of linear equations arising in computational electromagnetics in a finite element method using higher order elements. The proposed approach employs a graphics processing unit (GPU) for both numerical integration and matrix assembly. The performance results obtained on a test platform consisting of a Fermi GPU (1x Tesla C2075) and a CPU (2x twelve-core Opterons), indicate that the GPU implementation of the matrix generation allows one to achieve speedups by a factor of 81 and 19 over the optimized single- and multi-threaded CPU-only implementations, respectively.

1. INTRODUCTION

The utilization of graphic processing units (GPUs) in order to reduce the simulation time is one of the most developing trends in computational electromagnetics (CEM) [1–5]. The finite element method (FEM) is one of the most advanced and powerful methods for solving Maxwell's equation and it is often used in computational electromagnetics (CEM) [6–11]. Numerical simulations of deterministic (driven) problems with the finite element method include the following steps [12–14]:

(i) Discretization of a domain by applying a grid of elements.

(ii) **Numerical integration** — computation of the local mass ($\mathbf{T}_e$) and stiffness ($\mathbf{S}_e$) matrices for each element.

(iii) **Matrix assembly** — construction of sparse global mass ($\mathbf{T}$) and stiffness ($\mathbf{S}$) matrices from local matrices.

Received 3 April 2012, Accepted 28 April 2012, Scheduled 31 May 2012

* Corresponding author: Adam Dziekonski (adam.dziekonski@gmail.com).

(iv) Solution of a large system of linear equations $(\mathbf{S} - k_0^2\mathbf{T})\mathbf{x} = \mathbf{b}$, where k_0 is the wavenumber, $\mathbf{b}$ represents the excitation and $\mathbf{x}$ is the vector of amplitudes of basis functions.

The simulation is usually performed for many frequencies, therefore the solution step (iv) is executed many times and exerts a significant influence on the overall performance of the simulation. As a result, many efforts have been directed towards acceleration the step of solving a large system of linear equations using direct and iterative methods on a CPU (Central Processing Unit) and a GPU (Graphics Processing Unit) [15–18]. However, if higher order curvilinear elements are used, steps ((ii)–(iii), finite element matrix generation) are also time-consuming. What is more, the performance of matrix generation is essential in situations where the mesh changes, like for example adaptive mesh refinement or full-wave design optimization. An obvious way to accelerate the finite element matrix generation process (**matgen**) is to parallelize numerical computations. Surprisingly, only few papers have been published on parallelization of steps ((ii)–(iii)) that occur in the finite element method on a CPU and a GPU [19–22].

In this paper, a GPU-accelerated implementation of the finite element matrix generation for a 9 pole microwave combline filter for the GSM band in double precision is presented. The proposed approach allows one to achieve 81- and 19-fold performance improvement over CPU implementations executed with 1 and 24[†] threads, respectively. This performance is obtained thanks to two levels of parallelization: application of optimized functions (kernels) dedicated for the execution on a GPU and the utilization of streams (GPU functions (kernels) executed concurrently).

2. RELATED WORK

In order to provide the necessary background, we shall start with a brief review of the efforts related to the GPU-accelerated computations of the finite-element matrices. While none of them concerned computational electromagnetics, they all involved similar operations and hence are relevant to this work.

2.1. Numerical Integration

A GPU implementation in the Open Computing Language (OpenCL) of the numerical integration in single precision has been presented in [19]. In this paper, quadrilateral finite elements with curved (second

[†] one thread assigned to one CPU core.

order) geometry have been considered. In the GPU implementation a work-item is responsible for calculating one or more matrix entries and each work-group is responsible for processing a single finite element[‡]. Comparing to a CPU (Intel Xeon E5520), a GPU (GTX 285) implementation of the numerical integration proposed in [19] allows one to obtain the speedups by a factor of 3–6.

Another GPU implementation of the numerical integration has been presented in [20]. In this paper, prismatic elements and shape functions being products of 2D shape functions for triangles and 1D shape functions in the vertical direction were considered. Authors have faced a problem of small resources available for a single thread on a GPU. Due to this fact the proposed GPU implementation of numerical integration was based on the assumption that a single finite element corresponds to a single threadblock and these individual threads calculate sets of element stiffness matrix entries. Depending on the approximation order, GPU (GeForce 8800 GTX) acceleration has resulted in 4–20 times performance gain compared to a two core CPU implementation (AMD X2).

2.2. Matrix Assembly

Issues related to matrix assembly using GPU and CPU have been discussed in [21]. In this paper, the authors have compared matrix assembly based on the *Add to* operation to a technique called *LMA — Local Matrix Approach*. In the former approach, one construct a matrix that maps the local node of the element to a global node number. Terms of the local matrix of each element are summed into particular terms in the global matrix using the *Add to* operation. The final sparse global matrix is stored in the CRS format (Compressed Row Storage [29]). The authors have concluded that the *Add to* approach is preferable for an implementation on a CPU but not suitable for GPUs. This is because the atomic operations have to be used on a GPU to ensure correctness. Also, since the global matrix is stored using a compressed format (CRS) finding the location of a particular term in memory requires a bisection search of the sparsity structure of the matrix. To remedy this problem, the *LMA — Local Matrix Approach* dedicated for an implementation on a GPU has been proposed. *LMA* omits the global assembly of stiffness matrix altogether and as a result avoids the usage of atomic operations and bisection searches. A GPU (NVIDIA GTX480) implementation based on the *Local Matrix Approach* has been found to provide a speedup of approximately an

[‡] A multiprocessor executes a CUDA thread for each OpenCL work-item and a thread block for each OpenCL work-group (see Subsection 5.1).

order of magnitude over the 4-core Intel Xeon E5620 CPU version based on the *Add to* approach.

Another two approaches to matrix assembly, dedicated for a GPU have been presented in [22]. In this paper, 3D problems in single precision have been considered. The first approach, called *Global NZ*, takes advantage of global memory and divides computations into two GPU functions (kernels). In the first kernel, one thread computes the element data and in order to avoid race conditions, writes the element data to global memory in coalesced memory transactions. In the second kernel (reduction stage), one thread is assigned to assemble one nonzero entire from global memory. This approach is limited by the global memory size and its performance is adversely affected by the fact that both kernels refer to global memory, and hence the number of those references is doubled.

The second approach, called *Shared NZ*, takes advantage of shared memory to store the element data and reduce the number of transactions with global memory. A thread block is responsible for assembling a set of nonzero entries of the system of equations. This approach is limited by the shared memory size. The performance results reported in [22] indicate that both proposed algorithms executed on a GPU (GTX 480) yield speedups by a factor of 20 over a CPU implementation (Intel Core Duo).

3. FEM FORMULATION AND CONSTRUCTION OF MASS AND STIFFNESS MATRICES

The standard finite element problem is to solve the electric field vector wave equation

$$\nabla \times \mu^{-1} \nabla \times \mathbf{E} - \omega^2 \epsilon \mathbf{E} = 0 \tag{1}$$

with suitable boundary conditions.

In FEM a computational domain is divided into small elements, inside of each element the electric field is described as a linear combination of B vector basis functions ($\mathbf{N}_i$) with coefficients (e_i)

$$\mathbf{E} = \sum_{i=1}^{B} e_i \mathbf{N}_i \tag{2}$$

For each element the local mass matrix $\mathbf{T}_e$ and stiffness matrix $\mathbf{S}_e$ are computed as the following volume integrals:

$$t_{e_{ij}} = \iiint_V \mathbf{N}_i \epsilon_r \mathbf{N}_j dV \tag{3}$$

$$s_{e_{ij}} = \iiint_V (\nabla \times \mathbf{N}_i)\mu_r^{-1}(\nabla \times \mathbf{N}_j)dV \tag{4}$$

The implementation presented in this paper employs curvilinear tetrahedral elements and vector basis functions of up-to the third order (QTCuN) [23]. This implies that $B = 50$ basis functions (5 of which are linear combinations of the remaining 45 and are later rejected) are defined for each element. High-order Gaussian quadrature was applied [24] to evaluate integrals (3) and (4). A Gaussian quadrature over a tetrahedron in simplex coordinates can be computed as

$$\iiint_V f(x,y,z)dV \approx |V| \cdot \sum_{i=1}^{Q} w_i \cdot f(p_i) \tag{5}$$

where Q denotes the number of quadrature points p_i in simplex coordinates, and w_i-s are the quadrature weights. In our work, we use $Q = 81$. For curvilinear elements and quadrature (5) Equations (3), (4) can be rewritten in a matrix notation as

$$\mathbf{T}_e \approx \frac{1}{6}\sum_{i=1}^{Q} w_i \mathbf{N}_{Di}\left(\mathbf{J}_i^T \epsilon_r \mathbf{J}_i\right)\mathbf{N}_{Di}^T \det(\mathbf{J}_i) \tag{6}$$

$$\mathbf{S}_e \approx \frac{1}{6}\sum_{i=1}^{Q} w_i \mathbf{N}_{Pi}\left(\mathbf{J}_i^T \mu_r^{-1} \mathbf{J}_i\right)\mathbf{N}_{Pi}^T \frac{1}{\det(\mathbf{J}_i)} \tag{7}$$

where $\mathbf{J}_i$ is a Jacobian matrix computed at point p_i and $\mathbf{N}_{Di}$, $\mathbf{N}_{Pi}$ are matrices of the size $B \times 3$ storing the values of basis functions $\mathbf{N}_j$ and their curls $\nabla \times \mathbf{N}_j$ computed at point p_i (the integration is performed on reference tetrahedron). If an isotropic medium is taken into account, both ϵ_r and μ_r^{-1} are scalars, so they can be taken outside the integral. Similarly, if the element is rectilinear, the Jacobian $\mathbf{J}_i$ is constant inside the element and both $(\mathbf{J}_i^T \epsilon_r \mathbf{J}_i)$ and $(\mathbf{J}_i^T \mu_r^{-1} \mathbf{J}_i)$ can be put outside the integral to improve the efficiency. Once the local matrices have been computed they can be assembled into a global system of equations, resulting in large and sparse matrices $\mathbf{S}$ and $\mathbf{T}$.

4. GSM BAND FILTER

FEM formulations presented in previous section are applied in order to construct the global mass and stiffness matrices for a complex electromagnetics structure of a 9 pole microwave combline filter. The filter is designed for the GSM band and has three coaxial cross couplings (Fig. 1). Combline resonators consist of posts with

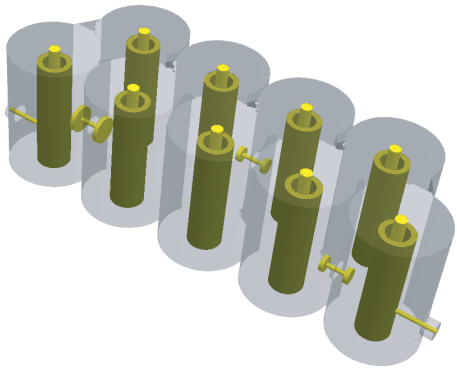

Figure 1. Test structure — 9th order combline filter for the GSM band with three coaxial cross couplings.

drilled holes and screws. The tetrahedral mesh of the structure was generated with the Netgen mesher [25]. In this paper the finite element matrix generation is performed for two meshes with 13613 and 28806 tetrahedra (see details in Table 1).

5. GPU IMPLEMENTATION OF FINITE ELEMENT MATRIX GENERATION

In this section, a GPU-based implementation of finite element matrix generation is described. Firstly, the most important features of GPU programming are presented. Later, GPU implementations of the fundamental phases of matrix generation are described.

5.1. Programming Fermi GPUs

In our research, we used NVIDIA's Fermi Graphics Processing Units. In this section a few concepts essential for understanding the efficiency of computations using Fermi architecture are recalled [26].

A single GPU has many processors that execute in parallel the same code on different data. In the Fermi architecture[§] [27], 32 processors are gathered into multiprocessors. A function that is called from a CPU (host) and executed on a GPU (device) is named a *kernel*. A thread is the smallest unit of parallelization in kernels. Threads are gathered into blocks of threads, while blocks are gathered into grids of blocks. Threads may access a few kinds of GPU memory: global memory (big latency, read-write), shared memory (on-chip, low

[§] Fermi is the code name for the generation of a CUDA architecture introduced in 2010.

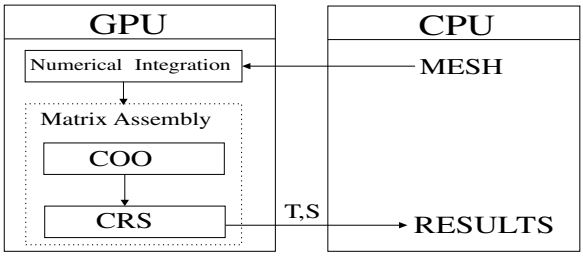

Figure 2. A basic variant of matrix generation on a GPU.

latency, limited to 16/48 kB per block), texture memory (low latency, read-only), and registers (low latency).

Compared to previous generations of GPUs the Fermi architecture has the following advantages [27]:

- computations in double precision are performed four times faster.
- atomic operations are performed remarkably faster (20x).
- concurrent kernel execution is possible (up to 16 different GPU functions can be executed independently).

These features can be used to improve the performance of FEM matrix generation either at the numerical integration or matrix assembly stage, or both. To obtain a high efficiency of the code execution on a GPU one should minimize the transfer between a GPU and a CPU, use shared memory as much as possible, replace global memory accesses with shared memory accesses and guarantee a coalesced access to global memory [26].

5.2. Phases of Matrix Generation

In this subsection, GPU implementations of three fundamental phases of the finite element matrix generation: numerical integration, matrix assembly in the coordinate (COO) format, conversion from the COO to the CRS (Compressed Rows Storage) format are described (Fig. 2).

5.2.1. Numerical Integration

In the numerical integration phase (Eqs. (6)–(7)) computations are performed in two loops (listing 1). The outer one is over M tetrahedra, while the inner loop involves Q iterations where Q is the number of points in the Gauss quadrature (in our case $Q = 81$). In [19, 20] a single thread block (work-group) is responsible for processing a single

finite element, which means that the outer loop is parallelized[‖]. In our implementation we also parallelize the outer loop. To achieve this, 32 finite elements are assigned to be processed in parallel. This number was selected based on the results of numerical tests which have shown that processing more than 32 tetrahedra at one time does not improve efficiency[¶].

Listening 1: A pseudocode of the numerical integration.

```
// M — no. of finite elements
for (int i = 0; i < M; i++){
  // Q — no. of points in Gauss quadrature
  for (int q = 0; q < Q; q++){
    //computation of mass matrix (Te), steps:
    T1 = ND * J_T;
    T2 = T1 * J;
    Te += T2 * ND_T;
    //computations of stiffness matrix (Se), steps:
    S1 = NP * J;
    S2 = S1 * J_T;
    Se += S2 * NP_T; }}
```

Inside the inner loop there are six dense matrix-matrix products. These products can be performed on a GPU with a function *cublasDgemm* from the CUBLAS library [28]. However, the CUBLAS library does not perform well enough for matrices the size of which is smaller than 100 and is not a multiple of 16. Such a situation occurs in the proposed formulations (Eqs. (2)–(7)) where the maximal size of dense matrices is equal to 50. To overcome this problem, we developed our own functions (kernels) to perform the dense matrix-matrix product on a GPU for dedicated matrix dimensions. What is more, the flexibility of the kernels was extended in order to parallelize a Gauss quadrature. In this case $Q = 81$ matrix-matrix products are performed in parallel, which is not directly possible with a *cublasDgemm* function. As a result, the numerical integration, which is performed by the inner loop in (listing 1), was replaced in our implementation by the parallel execution of 81 thread blocks.

The next optimization step was to replace disjoint computations of matrices T_1 and T_2 into one operation and do a similar thing related to

[‖] The number of active blocks processed on GPU multiprocessors depends on the GPU resources (f.e. up to 8 active blocks per multiprocessor in [19]).

[¶] Particularly if a Gauss quadrature is parallelized (see. next paragraph).

matrix $\mathbf{S}_e$. To achieve this, matrices T_1 (S_1) are stored in low latency on-chip shared memory instead of global memory (Subsection 5.1). Such a modification allows one to avoid the time-consuming writing and reading of sub-results (matrices T_1, S_1) from and to the global memory.

Finally, concurrent kernel execution was applied in such a manner that computations related to the generation of local matrices $\mathbf{T}_e$ and $\mathbf{S}_e$ were assigned to two separate streams. This Fermi architecture feature provides a twofold parallelization — not only threads inside kernels work in parallel, but also kernels are executed independently. What is worth mentioning, a concurrent kernel execution has not been utilized in any previous papers [19–22].

All in all, the code optimization described above allowed us to achieve a significantly better performance than a CUBLAS-based implementation (see the discussion in Section 6).

5.2.2. Matrix Generation in a Coordinate Format

In this phase, dense local matrices ($\mathbf{T}_e$, $\mathbf{S}_e$) arisen from the numerical integration are converted to sparse global matrices ($\mathbf{T}$, $\mathbf{S}$). At this stage the boundary conditions are also applied. Since the whole process is performed on a GPU, no additional transfer (GPU $\leftrightarrows$ CPU) between the numerical integration and this phase is required (see Fig. 2).

One should keep in mind that matrices ($\mathbf{T}, \mathbf{S}$) are sparse and in order to optimize the usage of memory a compression format should be applied. One of the most commonly used formats is CRS (Compressed Row Storage [29]). However, due to the fact that a direct matrix assembly into the CRS format has been proven to be inefficient on a GPU (see Subsection 2.2, [21]), we decided to divide the matrix assembly process into two steps. Firstly, sparse matrices ($\mathbf{T}$, $\mathbf{S}$) are generated and stored in the Coordinate format (COO)[+] and later on they are converted to the CRS format (Subsubsection 5.2.3). Such an approach is different from implementations reported in [21, 22] and to the best of the authors knowledge has not been implemented on a GPU yet.

The matrix assembly in the COO format is performed by two kernels. One kernel prepares information about the indices of nonzero entries in the global matrices. The other one takes advantage of the massive parallelization in such a manner that one thread is assigned to assembling 8 nonzero entries in the global matrix stored in the COO format. Moreover, a significant reduction of matrix assembly time is

[+] In the COO format a sparse matrix is compressed into three vectors: I — row indices, J — column indices, V — nonzero entries.

obtained thanks to the usage of registers and an efficient transfer to global memory. It has to be born in mind, that while the assembly in the COO format is much simpler and better suited to GPU than the assembly in the CRS format, this approach has one drawback. Global matrices ($\mathbf{T}$, $\mathbf{S}$) stored in the COO format include duplicates (entries of the same coordinates (I, J) but different values (V) that appear for finite elements of the same edge). These duplicates have to be detected and added and we do it while converting the matrix from the COO storage format to CRS. This conversion is required anyway, as the CRS format is commonly used by sparse matrix solver libraries both on CPU (Intel Pardiso [30]) and GPU (CUSPARSE Library [31]). Moreover, for sparse matrices stored in double precision the CRS format is about 30% more efficient in terms of memory usage than the COO format.

5.2.3. COO to CRS

As mentioned in the previous subsection, the conversion has also to guarantee the elimination of duplicates that exist in global matrices ($\mathbf{T}$, $\mathbf{S}$) stored in the COO format. On a CPU this can readily be done using the function *umfpack_triplet_to_col* from the UMFPACK library [♯] [32]. To the best of our knowledge similar GPU procedures for the COO to CRS format conversion with summation of duplicates to one matrix entry have not been reported. Therefore we have developed a multithreaded version of the UMFPACK *umfpack_triplet_to_col* function which has been optimized for execution on a GPU.

In our implementation, atomic operations are used during the elimination of duplicates. One should be aware of the fact that duplicates occupy about 20% of all nonzero entries (NNZ) of sparse global matrices stored in the COO format so the number of atomic operation is large. In previous CUDA architectures this would imply a large performance penalty (see Subsection 5.1, and a discussion in [21]). Fortunately, compared to previous CUDA architectures, Fermi compatible GPUs [27] allow one to achieve 20x better performance for atomic operations. What is more, at the beginning of the conversion step we sort global matrices ($\mathbf{T}$, $\mathbf{S}$) stored in the COO format, which leads to a significant reduction of the number of atomic operations — from 20% of nonzero entries (NNZ) to $2N$ — twice the sparse matrix size (N).

Secondly, the proposed converter overcomes the limitations of the parallel prefix sum (scan) proposed by NVIDIA in the GPU Computing SDK. The NVIDIA's implementation of scan algorithm realizes a

[♯] A function umfpack_triplet_to_col converts the COO to the CCS format (Compressed Column Storage [29]). However, CCS equals CRS for symmetric matrices.

commonly known algorithm for short vectors whereas for large vectors all data is divided into smaller parts for which scan is performed independently. Our implementation modifies the parallel prefix sum (scan) proposed by NVIDIA in such a way that the parallel prefix sum is always performed on the whole vector.

In order to optimize the assembly procedures even further, we assumed that matrices **T** and **S** have the same "pattern" — they have the same row and column indices, different nonzero values[††]. Thanks to this assumption, operations on indices are performed only for matrix **T** during the format conversion and are mapped for matrix **S**. It reduces redundant memory operations and eliminates the need for storing vectors of row and column indices of matrix **S** on a GPU.

6. RESULTS

The proposed approach to the finite element matrix generation was verified for the combline filter described in Section 4 on a test platform consisting of:

- 1x NVIDIA's Tesla C2075 GPU (448 cores, 6 GB).
- 2x Opteron 6174 (12 cores, 2.2 GHz) plus 64 GB RAM.

In order to provide a fair comparison, the GPU implementation of the finite element matrix generation is compared with CPU implementations that take advantage of the Intel MKL BLAS 1-3 functions performed in a parallel mode, OpenMP API and the *umfpack_triplet_to_col* function from the UMFPACK as a core of the conversion from the COO to the CRS format.

The details of test problems in double precision are given in Table 1. With 6 GB of GPU memory, the largest problem that can be processed in double precision involves 30000 tetrahedra. Larger problems can be handled by sequentially processing chunks smaller than 30000 finite elements. This mode of operation can be optimized but this is beyond the scope of this paper and will not be discussed.

The comparison between GPU and CPU implementations of three phases of the matrix generation is presented in Fig. 3 and Table 2. The data presented in Fig. 3 prove that the numerical integration is the most time-consuming phase of the finite element matrix generation on a CPU (73%–83%). The matrix generation in the COO format and a conversion from the COO to the CRS format occupy about 9%–13% and 4%–18% of the whole matrix generation time on a CPU,

[††] In general, matrices **T** and **S** have different patterns. We have decided to store extra zero entries in matrix **S** in order to have the same pattern for both matrices. Thanks to this modification, the conversion from the COO to the CRS format is performed much faster.

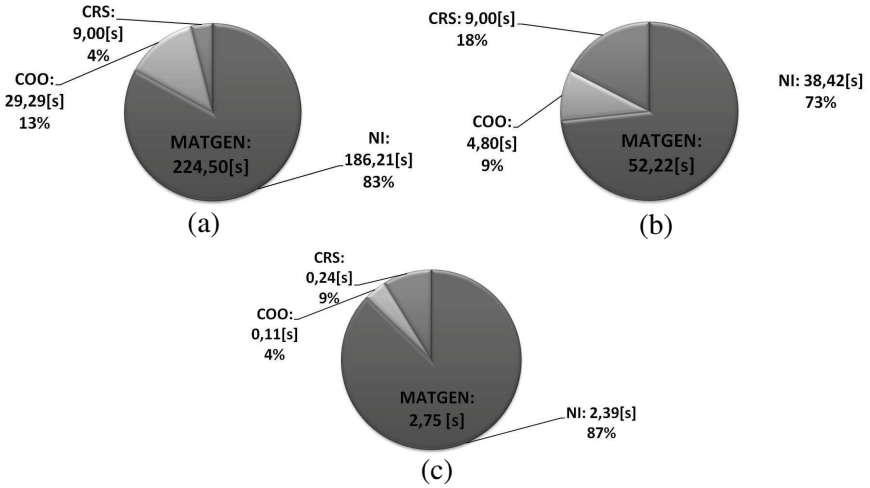

Figure 3. The finite matrix generation (TEST2) — time share required for the numerical integration (NI), matrix assembly into the COO format (COO) and conversion from the COO to CRS format (CRS). Test platforms GPU: (Tesla C2075) and CPU (2x twelve-core Opterons 6174). (a) Opteron — 1 thread. (b) Opteron — 24 thread. (c) Tesla C2075.

Table 1. Problems generated with the finite element method. NNZ — no. of nonzero entries, N — size of sparse system of linear equations, **T** — a mass matrix, **S** — a stiffness matrix.

Test problem	TEST1	TEST2
Elements (el.)	13613	28806
S, T_{COO} NNZ	17791074	38749374
S, T_{COO} N	213984	455916
S, T_{CRS} NNZ	14010426	30447378
S, T_{CRS} N	213984	455916

Table 2. Average speedup of the matrix generation phases between a GPU (Tesla C2075) and CPU (2x twelve-core Opteron 6174) implementation, *th.* — number of threads in a CPU implementation.

GPU vs. CPU	Tesla vs. Opteron-1th.	Tesla vs. Opteron-24th.
NI	77.88	15.56
COO	261.82	53.04
CRS	35.79	35.80

Table 3. Optimization of the numerical integration on a GPU (Tesla C2075). Test problem with 13613 tetrahedral elements.

GPU implementation	Description	[sec]	Speedup vs. CUBLAS
(A)	CUBLAS (*cublasDgemm*)	33.98	–
(B)	In house $C = A * B$	36.26	0.96
(C)	(B) for dedicated dimensions	28.5	1.19
(D)	(C) with parallel quadrature	2.26	16.50
(E)	(D) + joining steps $(T_1, T_2 \rightarrow T_{12})$	1.51	22.50
(F)	(E) + streams	1.37	24.68
(G)	(F) + many tetrahedra	1.13	30.02

respectively. This picture changes when the matrix is generated on a GPU. The numerical integration still takes up most of the matrix generation time (87%). However, the matrix assembly into the COO format achieves a remarkable performance — only 4% of the time of the matrix generation. The GPU-based conversion from the COO to CRS format that also eliminates duplicates takes about 9%.

One can notice that for each phase of the matrix generation the GPU implementation is more effective than a CPU one (Table 2). Comparing the performance obtained for the test platform (one Tesla vs. two twelve-core Opterons), one may observe that the numerical integration has been accelerated about 77.88 and 15.56 times on a Tesla C2075 versus CPU implementations that employ 1 and 24 threads, respectively. Moreover, for a matrix assembly into the COO format we have achieved even better results (speedups by a factor of 261.82 and 53.04 over CPU implementations that employ 1 and 24 threads). Finally, the GPU conversion from the COO to the CRS format that also eliminates duplicates has allowed us to achieve about 35 fold performance improvement over CPU implementations. Comparing a GPU implementation to CPU implementations that employ 1 and 24 threads, the cumulative performance improvement of the matrix assembly phase (COO+CRS) is about 105 and 41 fold, respectively.

Table 4. Performance of the finite matrix generation (th. — number of OpenMP threads). Test platforms GPU (Tesla C2075) and CPU (2x Opteron 6174).

Problem	TEST1	TEST2
GPU — Tesla [s]	1.29	2.75
CPU — Opteron (1th.) [s]	104.92	224.50
CPU — Opteron (24th.) [s]	24.03	52.22
Speedup — Tesla vs. Opteron (1th.)	**81.1**	**81.8**
Speedup — Tesla vs. Opteron(24th.)	**18.6**	**19.0**

Since the numerical integration is the most time-consuming phase of the finite element matrix generation, the effect of various optimization techniques used in our GPU implementation at this phase shall be described in details. The impact of each optimization technique proposed in our implementation (see Subsubsection 5.2.1 for details) on the efficiency of numerical integration can be seen from the data collected in Table 3 (variants A-G). Let us recall, that the baseline version implements the algorithm shown in listing 1 using CUBLAS procedures for matrix-matrix products. This version is denoted by letter (A). Our enhancements consisted of replacement of CUBLAS procedure by our own implementation of matrix-matrix product (B) and reformulation of this product for a specific matrix dimension occurring in our operations (C), parallelization of the inner loop (D), calculating local matrices $\mathbf{T}_e$ (and $\mathbf{S}_e$) in two steps, rather than three and application of shared memory at this stage (E), applying streams to compute $\mathbf{T}_e$ and $\mathbf{S}_e$ concurrently (F), and finally, processing many tetrahedra in parallel (G). One may notice that the in-house implementation of matrix-matrix product achieves slightly worse performance than functions from CUBLAS library (B). However, the modification of the code that takes the dedicated dimensions into consideration allows one to achieve 19% performance improvement (C). Nevertheless, the most significant improvement is obtained thanks to the parallelization of the Gauss quadrature (D). Moreover, the usage of the shared memory instead of the global memory reduced the time of numerical integration (E). A twofold parallelization obtained thanks to the concurrent kernel execution used in (F) allowed us to achieve further performance improvement. With many tetrahedra processed in parallel (G), the cumulative speedup by a factor of 30.02 over the CUBLAS based implementation has been reached.

Porting both numerical integration and the matrix assembly bore fruits in a significant overall performance improvement of the finite

element matrix generation. As shown in Table 4, the proposed GPU-based approach outperforms the optimized CPU implementations. With a Tesla C2075 performing computations, the sparse linear system with 455916 of unknowns is generated in just 2.75 seconds, while the best of CPU implementations allows obtaining the same system in 52.22 seconds (Opteron, 24 threads). All in all, the GPU-accelerated finite element matrix generation allowed us to achieve the speedups by a factor of 81 and 19 over CPU implementations executed on 1 and 24 threads, respectively.

7. CONCLUSION

A new effective and parallel approach for finite element matrix generation has been described in this paper. The obtained performance results indicate that the proposed GPU-accelerated implementation allows one to obtain a significant reduction of the matrix generation time in double precision. In the future, the authors will focus on the implementation which will allow for generation of even bigger sparse linear systems on a single GPU.

ACKNOWLEDGMENT

This work has been supported by the Polish Ministry of Science and Higher Education and carried out within the framework of The National Centre for Research and Development under agreement LIDER/21/148/L-1/09/NCBiR/2010. A Tesla C2075 GPU donation from Nvidia is also gratefully acknowledged.

REFERENCES

1. Shahmansouri, A. and B. Rashidian, "GPU implementation of split-field finite-difference time-domain method for Drude-Lorentz dispersive media," *Progress In Electromagnetics Research*, Vol. 125, 55–77, 2012.
2. Gao, P. C., Y. B. Tao, Z. H. Bai, and H. Lin, "Mapping the SBR and TW-ILDCs to heterogeneous CPU-GPU architecture for fast computation of electromagnetic scattering," *Progress In Electromagnetics Research*, Vol. 122, 137–154, 2012.
3. Gao, P. C., Y. B. Tao, and H. Lin, "Fast RCS prediction using multiresolution shooting and bouncing ray method on the GPU," *Progress In Electromagnetics Research*, Vol. 107, 187–202, 2010.
4. Banasiaka, R., Z. Yeb, and M. Soleimanic, "Improving three-dimensional electrical capacitance tomography imaging using approximation error model theory," *Journal of Electromagnetic Waves and Applications*, Vol. 26, Nos. 2–3, 411–421, 2012.

5. Jiang, W.-Q., M. Zhang, and Y. Wang, "CUDA-based radiative transfer method with application to the EM scattering from a two-layer canopy model," *Journal of Electromagnetic Waves and Applications*, Vol. 24, Nos. 17–18, 2509–2521, 2010.
6. Sadiku, M. N. O., *Numerical Techniques in Electromagnetics*, 2nd Edition, CRC, 2000.
7. Fotyga, G., K. Nyka, and M. Mrozowski, "Efficient model order reduction for FEM analysis of waveguide structures and resonators," *Progress In Electromagnetics Research*, Vol. 127, 277–295, 2012.
8. Klopf, E. M., S. B. Manic, M. M. Ilic, and B. M. Notaros, "Efficient time-domain analysis of waveguide discontinuities using higher order FEM in frequency domain," *Progress In Electromagnetics Research*, Vol. 120, 215–234, 2011.
9. Trujillo-Romero, C. J., L. Leija, and A. Vera, "FEM modeling for performance evaluation of an electromagnetic oncology deep hyperthermia applicator when using monopole, inverted T, and plate antennas," *Progress In Electromagnetics Research*, Vol. 120, 99–125, 2011.
10. Sun, H., Y. Wu, and Z. Ruan, "Edge-Based finite element method analysis of the transmission characteristics in antipodal finline," *Journal of Electromagnetic Waves and Applications*, Vol. 25, No. 4, 565–575, 2011.
11. Sun, H., Y. Wu, and Z. Ruan, "A study of transmission characteristics in elliptic-shaped microshield lines," *Journal of Electromagnetic Waves and Applications*, Vol. 25, Nos. 17–18, 2353–2364, 2011.
12. Jin, J., *The Finite Element Method in Electromagnetics*, John Wiley and Sons Inc., New York, 2002.
13. Volakis, J. L., A. Chatterjee, and L. C. Kempel, *Finite Element Method for Electromagnetics. Antennas, Microwave Circuits and Scattering Applications*, IEEE Series on Electromagnetic Wave Theory, IEEE Press, NJ, 1998.
14. Pelosi, G., R. Coccioli, and S. Selleri, *Quick Finite Elements for Electromagnetic Waves*, Artech House Inc., 2009.
15. Dehnavi, M. M., D. M. Fernandez, and D. Giannacopoulos, "Finite-element sparse matrix vector multiplication on graphic processing units," *IEEE Transactions on Magnetics*, Vol. 46, No. 8, 2982–2985, Aug. 2010.
16. Dziekonski, A., A. Lamecki, and M. Mrozowski, "A memory efficient and fast sparse matrix vector product on a GPU," *Progress In Electromagnetics Research*, Vol. 116, 49–63, 2011.

17. Dziekonski, A., A. Lamecki, and M. Mrozowski, "GPU acceleration of multilevel solvers for analysis of microwave components with finite element method," *IEEE Microwave and Wireless Components Letters*, Vol. 21, No. 1, 1–3, Jan. 2011.
18. Dziekonski, A., A. Lamecki, and M. Mrozowski, "Tuning a hybrid GPU-CPU V-Cycle multilevel preconditioner for solving large real and complex systems of FEM equations," *IEEE Antennas and Wireless Propagation Letters*, Vol. 10, 619–622, 2011.
19. Plaszewski, P., K. Banas, and P. Maciol, "Higher order FEM numerical integration on GPUs with OpenCL," *Proceedings of the International Multiconference on Computer Science and Information Technology (IMCSIT)*, 337–34, Oct. 18–20, 2010.
20. Maciol, P., P. Plaszewski, and K. Banas, "3D finite element numerical integration on GPUs," *Procedia Computer Science*, Vol. 1, No. 1, 1093–1100, 2010.
21. Markall, G., A. Slemmer, D. Ham, P. Kelly, C. Cantwell, and S. Sherwin, "Finite element assembly strategies on multi-core and many-core architectures," *International Journal for Numerical Methods in Fluids*, 2012.
22. Cecka, C., A. Lew, and E. Darve, "Application of assembly of finite element methods on graphics processors for real-time elastodynamics," *GPU Gems 3*, Jul. 2011.
23. Ingelstrom, P., "A new set of H(curl)-conforming hierarchical basis functions for tetrahedral meshes," *IEEE Trans. on Microwave Theory and Techniques*, Vol. 54, 106–114, Jan. 2006.
24. Zhang, L., T. Cui, and H. Liu, "A set of symmetric quadrature rules on triangles and tetrahedra," *Journal of Computational Mathematics*, Vol. 26, No. 3, 1–16, 2008.
25. Schöberl, J., "NETGEN an advancing front 2D/3D-mesh generator based on abstract rules," *Computing and Visualization in Science*, Vol. 1, No. 1, 41–52, Jul. 1997.
26. Sanders, J. and E. Kandrot, *CUDA by Example: An Introduction to General-Purpose GPU Programming*, NVIDIA Co., 2011.
27. *http://www.nvidia.com/content/PDF/fermi.*
28. CUBLAS Library Nvidia Co., 2011.
29. Saad, Y., Iterative Methods for Sparse Linear Systems, SIAM, 2004.
30. *http://software.intel.com/en-us/articles/intel-mkl/* .
31. CUDA CUSPARSE Library, NVIDIA Co., 2011.
32. *http://www.cise.ufl.edu/research/sparse/umfpack.*