Progress In Electromagnetics Research, Vol. 116, 49–63, 2011

A MEMORY EFFICIENT AND FAST SPARSE MATRIX VECTOR PRODUCT ON A GPU

A. Dziekonski, A. Lamecki, and M. Mrozowski

WiComm Center of Excellence, Faculty of Electronics
Telecommunications and Informatics (ETI)
Gdansk University of Technology (GUT), Gdansk 80-233, Poland

Abstract—This paper proposes a new sparse matrix storage format which allows an efficient implementation of a sparse matrix vector product on a Fermi Graphics Processing Unit (GPU). Unlike previous formats it has both low memory footprint and good throughput. The new format, which we call Sliced ELLR-T has been designed specifically for accelerating the iterative solution of a large sparse and complex-valued system of linear equations arising in computational electromagnetics. Numerical tests have shown that the performance of the new implementation reaches 69 GFLOPS in complex single precision arithmetic. Compared to the optimized six core Central Processing Unit (CPU) (Intel Xeon 5680) this performance implies a speedup by a factor of six. In terms of speed the new format is as fast as the best format published so far and at the same time it does not introduce redundant zero elements which have to be stored to ensure fast memory access. Compared to previously published solutions, significantly larger problems can be handled using low cost commodity GPUs with limited amount of on-board memory.

1. INTRODUCTION

Solving electromagnetic problems involves intensive and time consuming computations. In order to reduce the processing time, the computational electromagnetics community has been exploring the possibility to use GPUs for accelerating various numerical techniques including the Finite Difference Time Domain (FDTD) method [1–4], Alternating Direction Implicit (ADI) method [5], Transmission Line Modeling

Received 16 March 2011, Accepted 16 April 2011, Scheduled 20 April 2011

Corresponding author: Adam Dziekonski (adziek@eti.pg.gda.pl).

(TLM) method [6, 7] and also for applications such as Radar Cross Section Prediction (RCS) [8–10]. Relative little attention has been so far given to the Finite Element Method (FEM). This method, often used for analyzing complex resonators or waveguides or solving electromagnetic radiation and scattering problems [11–16], produces a large sparse system of equations. The resulting eigenvalue or a driven problem is then frequently solved using Krylov space iterative techniques [17–19]. Such techniques involve many computations of a sparse matrix times vector product (SpMV) [20]. Several groups have demonstrated that SpMV operation can be executed much faster on GPUs than on multicore Central Processing Units (CPUs) [21–24]. The speed of the SpMV product on a GPU is important because this operation exerts a significant impact on the overall solution time of the Krylov space solvers especially when polynomial [17] or multilevel preconditioners with the Jacobi smoother [25] are used. One of the factors that affects the efficiency of the matrix-vector product is the way the sparse matrix is stored in the GPU memory [21]. For matrices with irregular non-zero entry patterns the best results on a GPU are obtained using variants of the storage scheme known as Ellpack [21]. The Ellpack format can be modified in order to best exploit the computational features of a particular GPU architecture [22–24].

Unfortunately, the Ellpack format has a serious drawback, namely it stores a matrix row by row with each row containing an identical number of elements. If a row contains fewer non-zero elements, it is padded with zeros. For matrices in which the number of non-zeros per row varies significantly this implies storing a large number of redundant elements. This becomes a problem for computations involving commodity GPUs which carry a very limited amount of fast memory (typically up to 1.5 GB). The problem of finding a memory efficient matrix storage format which is also suited for fast execution of SpMV on GPUs is even more pronounced in computational electromagentics where one often deals with complex matrices.

In this paper we propose a new memory efficient and fast storage format which is well suited for implementing sparse matrix times vector multiplications on the CUDA (Compute Unified Device Architecture) developed by NVIDIA. We also show how to tune the SpMV operation for the latest NVIDIa's Fermi family of GPUs by using a configurable cache and concurrent kernel execution. We demonstrate the efficiency of the new format and the SpMV operation on a number of complex valued sparse matrices obtained from FEM discretization of a dielectric antenna problem. Not only does the new approach remarkably reduce redundant zero-padding but it also allows one to achieve a performance of SpMV which is on a par with best results reported so far.

2. PROGRAMMING GPUS

We shall briefly recall a few concepts that are essential for understanding the efficiency of GPU computations [26, 27]. GPUs have many processors that execute in parallel the same code, called kernel, on different data. In the CUDA architecture [28], processors on a GPU are gathered into multiprocessors[†]. Kernels are called from a CPU. A thread is the smallest unit of parallelization in kernels. Threads are gathered into blocks of threads, which share memory on a single multiprocessor. Blocks are gathered into grids of blocks that logically are executed in parallel during a kernel's execution. Threads may access a few kinds of GPU memory: global memory (big latency, read-write), shared memory (on-chip, low latency, limited to 16 kB per block), texture memory (low latency, read-only), and registers (low latency). To obtain a high efficiency of code execution, it is important to pay attention to the following rules:

- guarantee coalesced access to global memory
- if coalesced access to global memory is impossible - use texture memory instead
- use shared memory as much as possible
- replace global memory accesses with shared memory accesses (if possible)
- minimize transfer between GPU and CPU

Fermi, which is the code name for the latest generation of CUDA architecture [29], adds features that can be exploited for increasing performance. In particular, the Fermi architecture:

- supports configurable cache, can be allocated 16 kB for shared memory and 48 kB of L1 extra cache; or 48 kB for can be allocated for shared memory and 16 kB of L1 extra cache
- allows concurrent kernel execution (up to 16 different GPU functions executed in parallel)
- performs double precision computations remarkably faster than previous generations of GPUs

3. COMPRESSION STORAGES FOR EFFICIENT SPMV

As explained in the introduction the format in which the matrix is stored affects both the performance of SpMV operation and the amount

[†] 8 processors = 1 multiprocessor for the CUDA architecture, 32 processors = 1 multiprocessor for the Fermi architecture.

of memory used. The second factor that affects speed is a judicious use of various features of the CUDA architecture.

In this section we shall discuss the features of a few matrix storage formats that can be used in the SpMV product and present their pros and cons in context of GPU computing.

3.1. CRS

In a CRS (Compressed Row Storage) format, a sparse matrix is compressed into three vectors: a vector of non-zero entries, a vector of column indices of non-zero entries, a vector of column indices of the first non-zero entry in a row [17].

This format is used in the Intel MKL (Math Kernel Library) and it is efficient for sparse matrix times vector operation implemented on CPUs with multiple cores. While GPU implementations of SpMV based on CRS [21, 30] perform better than MKL, due to the lack of coalesced access to global memory they show worse throughput than SpMV based on Ellpack-like formats. Nevertheless, from the memory point of view, this format is very efficient because no zero-padding is needed. The number of bytes required for storage in the CRS format for double precision and complex valued matrices is:

$$\mathrm{CRS} = (2 \times NNZ \times 8\,\mathrm{Bytes} + (NNZ + N) \times 4\,\mathrm{Bytes}) \tag{1}$$

where: NNZ — length of vector of non-zero entries, N — number of rows.

3.2. ELL-R

To ensure coalesced memory access while executing SpMV on a GPU Ellpack (ELL) format was proposed in [21]. In the ELL format, each row of a compressed matrix is stored in two vectors: a vector of non-zero entries and vector of column indices of non-zero entries (both with some extra zeros). This format allows one to achieve better performance on a GPU than on a CPU, but introduces significant redundancy in terms of memory[‡], since rows are zero-padded in order to have the length of the longest non-zero entry row ($N_{\max}$). In a GPU implementation of the SpMV product based on the ELL-R format, one thread works on one row. However, the performance of SpMV can be even further improved by adding an extra vector which provides the information about the number of non-zeros in each row. This modified format is called ELL-R and enhances the performance since only non-zeros are involved in computations [22]. For better performance on a

[‡] Redundancy understood as the percentage of extra elements required for storing the matrix in a sparse format.

GPU the number of rows (N) has to be a divisor of the block size, otherwise there is no coalesced access to global memory. The only way to fulfill this condition is to add some extra rows with zero elements (N'). From the above description it is evident that from the memory point of view, the ELL-R format is less effective than CRS. The number of bytes required for storage in the ELL-R format for double precision and complex valued matrices is:

$$\text{ELL-R} = (2 \times (N_{\max} \times N') \times 8\,\text{Bytes} + (N_{\max} \times N' + N) \times 4\,\text{Bytes}) \quad (2)$$

where: $N_{\max}$ — number non-zero entries in the longest row.

3.3. Sliced ELLPACK

To eliminate the redundancy inherent in the ELL format, Monakov et al. [23] proposed slicing the matrix prior to compression. In this format, called Sliced ELL, there is a specific preprocessing applied in which a sparse matrix is first divided into submatrices (slices) consisting of S adjacent rows ($S = 1, \ldots, N$) and each slice is then stored in the ELL format. As a result, the number of extra zeros is determined by the distances between the shortest and the longest rows in slices, rather than in a whole matrix.

3.4. ELLR-T

The best performance in terms of GFLOPS in SpMV operation on a GPU has so far been reported for a storage format presented in [24]. The format is called ELLR-T and it is an extension of ELL-R in which a preprocessing needs to be applied [24]. In this preprocessing non-zero elements and their column indices are permuted and zero-padding occurs and each row is a multiple of 16. Thanks to this modification, coalesced and aligned access to global memory occurs. In contrast to ELL-R, many threads ($T = 1, 2, 4, 8, 16, 32$) operate on a single row while executing the SpMV operation[§]. Both in Sliced ELL and ELLR-T there are many threads working on a single row, but there are completely different addressing schemes that result from the different preprocessings.

Specific addressing and thread parallelism combined with usage of shared memory yields on average 80% and 15% performance improvement over ELL-R and Sliced ELL, respectively. Unfortunately, in ELLR-T there is a similar problem with zero-padding as in ELL-R. The total memory that needs to be stored in ELLR-T depends on T

[§] For the previous NVIDIA architecture, the best performance was achieved for $T = 1, 2, 4, 8$ [24]. However, for the Fermi architecture where blocks may contain more threads (up to 1024), better performance is achieved also for $T = 16, 32$.

(the number of threads which operate on a single row), but all in all this amount does not differ from ELL-R a lot.

3.5. Sliced ELLR-T

The format that is proposed in this paper takes advantage of the above mentioned formats. In fact it is a modification of the Sliced ELL and ELLR-T formats. As in Sliced ELL-R, the matrix is divided into slices with S rows each. Additionally, as in ELLR-T, many threads ($T = 1, 2, 4, 8, 16, 32$) operate on single row while executing SpMV[‖]. For each slice permutation of non-zero entries and zero-padding occurs in order to achieve rows that are a multiple of 16. As a result coalesced and aligned access to global memory occurs in slices. Since the new format combines the features of ELLR-T and Sliced ELL we shall call it Sliced ELLR-T (Fig. 1). It is evident that this format should enable one to achieve sufficient performance due to the coalesced access to global memory, multiple treads working on a single row, and shared memory's being used in execution.

At the same time, thanks to cutting the matrix into slices, this format significantly reduces the amount of memory required for storage.

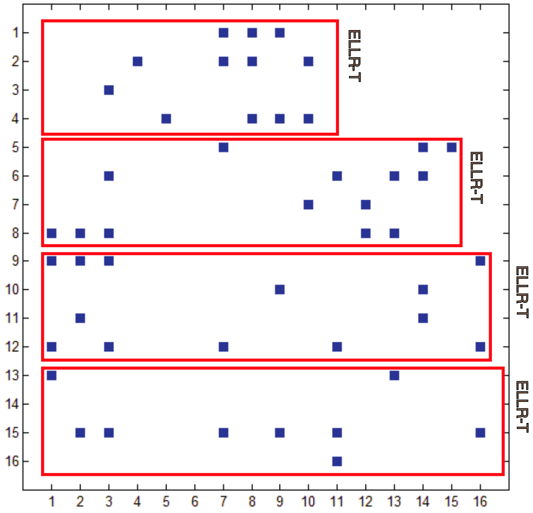

Figure 1. Sparse matrices divided into slices ($S = 4$), and each slice stored with the ELLR-T scheme.

[‖] Size of the block of threads = $S * T$.

4. PERFORMANCE TESTS

This section presents a comparison of the performance of the SpMV operation implemented on a CPU using a CRS matrix storage format and on a GPU for the various formats described in Section 3. The operating system is Windows 7 64-bits and the test platforms are: the GPU-NVIDIA's GTX 480 (480 cores, 1.5 GB memory, CUDA v3.2) and the CPU-Xeon 5680 (6 cores). In order to obtain a fair comparison, all GPU implementations are compared with CPU computations involving optimized Intel MKL functions (Intel(R) Compiler Pro 11.1 build 065, Intel MKL 10.2.5). The best results when performing SpMV on a CPU where obtained for the CRS format [31] applied to the matrix with the RCM (Reverse Cuthill McKee) ordering [17], all six cores enabled and hyper-threading disabled (Intel's recommends disabling hyper-threading when using MKL as in this particular type of computations the threaded portions of the library execute at high efficiencies using most of the available resources and perform identical operations on each thread [32]). On the other hand, on a GPU we permute rows in each matrix from the shortest to the longest one, which not only guarantees balanced thread blocks but also minimizes the number of redundant zero elements in each slice for the Sliced ELLR-T format.

Our test problem is a dielectric resonator antenna (DRA), which is fed from a rectangular cavity with an SMA connector [33]. The test problem was discretized by the finite element method with PML (Perfectly Matched Layer) and vector elements of up to the third order. As a result, we obtained a sparse complex matrix A with 146517 rows and over 21 M non-zero entries. The matrix was further divided into nine submatrices with each submatrix A_{ij} ($i = \{0,1,2\}$, $j = \{0,1,2\}$) corresponding to the order of the vector elements used in the evaluation of the inner products while assembling the FEM matrix. This resulted in test matrices arranged in the following way:

$$\mathbf{A} = \begin{pmatrix} A_{00} & A_{01} & A_{02} \\ A_{10} & A_{11} & A_{12} \\ A_{20} & A_{21} & A_{22} \end{pmatrix} \quad (3)$$

4.1. Results and Comments

The test problem yields 10 complex valued matrices with different sizes and non-zero patterns. The details of these matrices are given in Tab. 1. The last column in Tab. 1 shows the amount of memory which is required for storing the non-zero elements only. As explained in Section 3 extra space is needed for various storage schemes used in

Table 1. Description of a test problem.

Matrix	Rows	$nnz\,(complex)$	$\frac{nnz}{Rows}$	Double [MB]
A	146517	21697606	149.09	173.58
A_{00}	8091	218722	27.03	1.75
A_{01}	8091	681548	84.24	5.45
A_{02}	8091	1391158	171.84	11.13
A_{10}	40129	681548	16.98	5.45
A_{11}	40129	1973198	49.17	15.79
A_{12}	40129	3895180	97.07	31.16
A_{20}	98297	1391158	14.15	11.13
A_{21}	98297	3895180	39.63	31.16
A_{22}	98297	7569914	77.01	60.56

Table 2. Comparison of amount of memory [MB] required for storage for different formats in double precision. A — basic matrix, $A_{00\text{–}22}$ — nine submatrices from Tab. 1.

Matrix	CRS	ELL-R	ELLR-T	Sliced ELLR-T
A	217.56	783.90	785.98	226.76
$A_{00\text{–}22}$	218.74	786.10	795.05	236.24

SpMV computations. A comparison of the memory required for storing the entire complex matrix A and all the component matrices $A_{00\text{–}22}$ is given in Tab. 2. It is evident that compared to the ELLR-T (and also ELL-R) scheme, the new Sliced ELLR-T format reduces significantly (down to about one third) the amount of memory required for matrix storage and it is nearly as effective as the CRS format. Assuming that compaction of the CRS format is the reference, ELLR-T needs up to 350% more amount of the memory. With the Sliced ELLR-T format the redundancy is only about 8%.

The memory economy clearly speaks in favour of the Sliced ELLR-T format. However, it is essential to verify if it does not lead to performance degradation. To this end, we have tested the performance of the SpMV product using the formats described in Section 3 for each submatrix A_{ij} in order to verify the usefulness of the implementation of SpMV based on the Sliced ELLR-T format. The results for single and double precision are shown in Figs. 2 and 3. As opposed to a CPU for which performance decreases for bigger problems, in both cases (single and double precision) the bigger the problem, the better the performance on a GPU. Figs. 2, 3 reveal the limitation of the ELL-

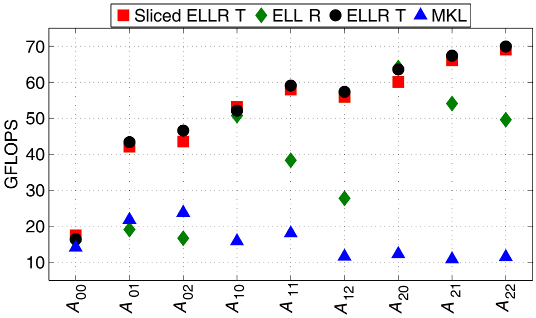

Figure 2. Performance of SpMV in GFLOPS for complex matrices $A_{00\text{–}22}$ in single precision. (GTX 480 vs. Intel Xeon 5680).

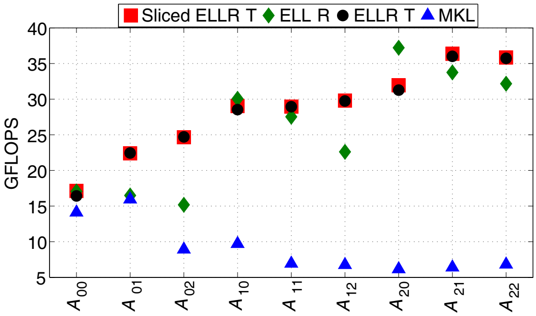

Figure 3. Performance of SpMV in GFLOPS for complex matrices $A_{00\text{–}22}$ in double precision. (GTX 480 vs. Intel Xeon 5680).

R format in which only one thread operates on a single row. When the number of non-zero entries per row grows, a single thread takes a longer time to complete the computations so the performance of SpMV drops. ELLR-T and Sliced ELLR-T formats do not suffer from this dependance, because more threads are allowed to operate per row. The number of concurrent threads (T) per row is adjusted for each of matrices A_{00}–A_{22}. Our tests have shown that the optimal performance was achieved for $T = 2$ per row for matrices A_{00}, A_{10}, A_{20}, A_{21} that have fewer than 40 non-zero elements per row, $T = 4$ for matrices A_{11}, A_{22} (more than 40 and fewer than 80 non-zero elements per row), $T = 8$ for matrices A_{01}, A_{12}, A_{02} (more than 80 non-zero elements per row).

Thanks to faster double precision available on the Fermi architecture, the speed difference between a GPU and a CPU is comparable for both precisions and for bigger test problems it reaches

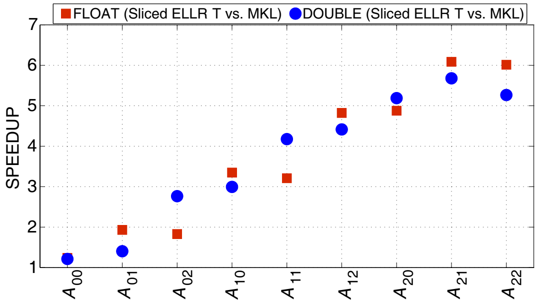

Figure 4. Comparison of speedup between a GPU (Sliced ELLR-T) and a CPU (Intel MKL) for single and double precision. GTX 480 vs. Intel Xeon 5680.

a factor of 6 (Fig. 4).

As mentioned in Section 2, the Fermi architecture is equipped with configurable cache memory and allows concurrent execution of kernels. As far as cache memory is concerned we found that allocating 16 kB of shared memory, and 48 kB of extra L1 cache, on a GPU gives 20% better results than the other way round (48 kB of shared memory and 16 kB of extra L1 cache).

4.1.1. Impact of Concurrent Kernels

As mentioned in Section 2, the Fermi architecture is capable of executing of up to 16 GPU functions (kernels). To investigate the impact of concurrent kernel execution we performed three tests using matrix A.

It has to be pointed out that in this section we evaluate the performance of the SpMV operation executed on the entire matrix A, which is big enough for the concurrent kernels to have a noticeable impact. Let us recall that the matrix is composed of all matrices used in benchmarks presented in the previous section (see Eq. (3)). For the CPU execution (Intel MKL, CRS) and single precision the result is 10.32 GFLOPS when all 6 cores are engaged. This result will serve as a reference for evaluation the speedup offered by a GPU. For test involving a GPU we have implemented the SpMV operation in three different ways:

(i) The entire matrix A is stored in one of the storage formats suitable f or a GPU and the SpMV product is executed using one kernel only.

(ii) The matrix A is stored as nine separate matrices each and the

SpMV operation on the entire matrix A ($y = Ax$) is implemented by launching nine concurrent kernels with each kernel operating on one submatrix and a part of vector $x = [x_0, x_1, x_2]^T$, and each kernel contributing to the final vector $y = [y_0, y_1, y_2]^T$:

$$\begin{aligned} y_0 &= A_{00}x_0 + A_{01}x_1 + A_{02}x_2; \\ y_1 &= A_{10}x_0 + A_{11}x_1 + A_{12}x_2; \\ y_2 &= A_{20}x_0 + A_{21}x_1 + A_{22}x_2; \end{aligned}$$

(iii) The matrix A is stored as eleven separate matrices. This arrangement was obtained by dividing submatrices A_{11} and A_{22} horizontally. The SpMV operation on the entire matrix A ($y = Ax$) is implemented by launching eleven concurrent kernels with each kernel operating on one submatrix and a part of vector $x = [x_0, x_1, x_2]^T$ and each kernel contributing to the final vector $y = [y_0, y_1, y_2]^T$:

$$\begin{aligned} y_0 &= A_{00}x_0 + A_{01}x_1 + A_{02}x_2; \\ y_1 &= A_{10}x_0 + [A_{11a}x_1; A_{11b}x_1] + A_{12}x_2; \\ y_2 &= A_{20}x_0 + A_{21}x_1 + [A_{22a}x_2; A_{22b}x_2]; \end{aligned}$$

For a single kernel, the results of the SpMV operation for single precision on a GPU were as follows: (ELL-R) — 38.38 GFLOPS, GPU (ELLR-T) — 62.51 GFLOPS, GPU (Sliced ELLR-T) — 59.34 GFLOPS. In terms of speedup relative to a CPU we achieved for a single kernel execution of a GPU the speedup factors of 3.72, 6.06, and 5.75 for ELL-R, ELLR-T and Sliced ELLR-T, respectively.

Tests using concurrent kernels were carried out for the most effective formats from the performance point of view (ELLR-T and Sliced ELLR-T). For ELLR-T we achieved 65.45 GFLOPS and 69.02 GFLOPS, for 9 and 11 kernels, respectively. For Sliced ELLR-T the results were 62.76 GFLOPS (nine kernels) and 67.52 GFLOPS (11 kernels). These results imply that the concurrent kernels feature of the FERMI architecture allowed us to increase the speedup of the SpMV operation[¶] for ELLR-T from 6.06 (single kernel) to 6.69 (eleven kernels), which means the performance increase of 10.42%, and for Sliced ELLR-T from 5.75 (single kernel) to 6.54 (eleven kernels), which is equivalent to the performance increase of 13.78%.

The result of our tests have shown that the Sliced ELLR-T format is superior to the ELLR-T format. Both formats have been designed to maximize the throughput on a GPU but the memory economy offered by the Sliced ELLR-T scheme is significant (see Tab. 2) while the deterioration a of the SpMV performance in single and double precision is marginal.

[¶] Relative to a CPU implementation using the Intel MKL and 6 cores.

5. CONCLUSIONS

In this paper we propose a new Sliced ELLR-T format to process SpMV on a GPU. The results of the tests indicate that the Sliced ELLR-T format is efficient from both memory and performance points of view. We also discuss the impact of several new GPU features of the Fermi architecture such as configurable cache and concurrent kernel execution on sparse matrix times vector operation in complex arithmetic. The new format and Fermi architecture allows one to achieve up to 69 GFLOPS and 36 GFLOPS on a GTX 480 in complex single and double precision respectively. The small memory footprint of the new format makes it especially attractive for implementing fast GPU accelerated iterative solvers for larger sparse and complex systems of equations which occur in the FEM analysis of electromagnetic radiation and scattering problems.

ACKNOWLEDGMENT

This work has been supported by the Polish Ministry of Science and Higher Education and carried out within the framework of COST IC 0603 ASSIST programme and The National Centre for Research and Development under agreement LIDER/21/148/L-1/09/NCBiR/2010.

REFERENCES

1. Krakiwsky, S. E., L. E. Turner, and M. Okoniewski, "Acceleration of finite difference time-domain (FDTD) using graphics processor units (GPU)," *IEEE MTT-S International Microwave Symposium Digest 2004*, 1033–1036, June 2004.
2. Adams, S., J. Payne, and R. Boppana, "Finite difference time domain (FDTD) simulations using graphics processors," *High Performance Computing Modernization Program Users Group Conference*, 2007.
3. Sypek, P., A. Dziekonski, and M. Mrozowski, "How to render FDTD computations more effective using a graphics accelerator," *IEEE Transactions on Magnetics*, Vol. 45, No. 3, 1324–1327, March 2009.
4. Xu, K., Z. Fan, D.-Z. Ding, and R.-S. Chen, "GPU accelerated unconditionally stable crank-nicolson FDTD method for the analysis of three-dimensional microwave circuits," *Progress In Electromagnetics Research*, Vol. 102, 381–395, 2010.
5. Stefanski, T. P. and T. D. Drysdale, "Acceleration of the 3D

ADIFDTD method using graphics processor units," *IEEE MTT-S International Microwave Symposium Digest 2009*, 241–244, June 2009.

6. Rossi, F. V., P. P. M. So, N. Fichtner, and P. Russer, "Massively parallel two-dimensional TLM algorithm on graphics processing units," *IEEE MTT-S International Microwave Symposium Digest 2008*, 153–156, June 2008.
7. Rossi, F. and P. P. M. So, "Hardware accelerated symmetric condensed node TLM procedure for NVIDIA graphics processing units," *IEEE APSURSI Antennas and Propagation Society International Symposium 2009*, 1–4, June 2009.
8. Tao, Y. B., H. Lin, and H. J. Bao, "From CPU to GPU: GPU-based electromagnetic computing (GPUECO)," *Progress In Electromagnetics Research*, Vol. 81, 1–19, 2008.
9. 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.
10. Lezar, E. and D. B. Davidson, "GPU-accelerated method of moments by example: Monostatic scattering," *IEEE Antennas and Propagation Magazine*, Vol. 52, 120–135, 2010.
11. Garcia-Castillo, L. E., I. Gomez-Revuelto, F. Saez de Adana, and M. Salazar-Palma, "A finite element method for the analysis of radiation and scattering of electromagnetic waves on complex environments," *Computer Methods in Applied Mechanics and Engineering*, Vol. 194, Nos. 2–5, 637–655, February 2005.
12. Gomez-Revuelto, I., L. E. Garcia-Castillo, D. Pardo, and L. Demkowicz, "A two-dimensional self-adaptive finite element method for the analysis of open region problems in electromagnetics," *IEEE Transactions on Magnetics*, Vol. 43, No. 4, 1337–1340, April 2007.
13. Lezar, E. and D. B. Davidson, "GPU-based arnoldi factorisation for accelerating finite element eigenanalysis," *Proceedings of the 11th International Conference on Electromagnetics in Advanced Applications — ICEAA'09*, 380–383, September 2009.
14. Jian, L. and K. T. Chau, "Design and analysis of a magnetic-geared electronic-continuously variable transmission system using finite element method," *Progress In Electromagnetics Research*, Vol. 107, 47–61, 2010.
15. Ping, X. W. and T. J. Cui, "The factorized sparse approximate inverse preconditioned conjugate gradient algorithm for finite element analysis of scattering problems," *Progress In Electromagnetics Research*, Vol. 98, 15–31, 2009.

16. Tian, J., Z. Q. Lv, X. W. Shi, L. Xu, and F. Wei, "An efficient approach for multifrontal algorithm to solve non-positive-definite finite element equations in electromagnetic problems," *Progress In Electromagnetics Research*, Vol. 95, 121–133, 2009.
17. Saad, Y., *Iterative Methods for Sparse Linear Systems*, SIAM, 2004.
18. Velamparambil, S., S. MacKinnon-Cormier, J. Perry, R. Lemos, M. Okoniewski, and J. Leon, "GPU accelerated krylov subspace methods for computational electromagnetics," *38th European Microwave Conference EuMC 2008*, 1312–1314, October 27–31, 2008.
19. Cwikla, A., M. Mrozowski, and M. Rewienski, "Finite-difference analysis of a loaded hemispherical resonator," *IEEE Transactions on Microwave Theory and Techniques*, Vol. 51, No. 5, 1506–1511, May 2003.
20. Yang, X., "A survey of various conjugate gradient algorithms for iterative solution of the largest/smallest eigenvalue and eigenvector of a symmetric matrix," *Progress In Electromagnetics Research*, Vol. 5, 567–588, 1991.
21. Bell, N. and M. Garland, "Efficient sparse matrix-vector multiplication on CUDA," *NVIDIA Technical Report NVR-2008-004*, NVIDIA Corporation, December 2008.
22. Vazquez, F., E. M. Garzon, J. A. Martinez, and J. J. Fernandez, "The sparse matrix vector product on GPUs," *Proceedings of the 2009 International Conference on Computational and Mathematical Methods in Science and Engineering*, Vol. 2, 1081–1092, July 2009.
23. Monakov, A., A. Lokhmotov, and A. Avetisyan, "Automatically tuning sparse matrix-vector multiplication for GPU architectures," *High Performance Embedded Architectures and Compilers, Lecture Notes in Computer Science*, Vol. 5952, 111–125, 2010.
24. Vazquez, F., G. Ortega, J. J. Fernandez, and E. M. Garzon, "Improving the performance of the sparse matrix vector product with GPUs," *IEEE 10th International Conference on Computer and Information Technology (CIT)*, 1146–1151, 2010.
25. 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, January 1–3, 2011.
26. Kirk, D. B. and W. W. Hwu, *Programming Massively Parallel Processors: A Hands-on Approach*, Elsevier Inc., 2010.

27. Sanders, J. and E. Kandrot, *CUDA by Example: An Introduction to General-Purpose GPU Programming*, Nvidia Corporation, 2011.

28. Programming Guide Version 3.2, Nvidia Corporation, 2011.

29. http://www.nvidia.com/object/fermi_architecture.html.

30. CUDA CUSPARSE Library, Nvidia Corporation, 2011.

31. Lee, V. W., C. Kim, J. Chhugani, M. Deisher, D. Kim, A. D. Nguyen, N. Satish, M. Smelyanskiy, S. Chennupaty, P. Hammarlund, R. Singhal, and P. Dubey, "Debunking the 100 X GPU vs. CPU myth: An evaluation of throughput computing on CPU and GPU," *ACM SIGARCH Computer Architecture News — ISCA'10*, Vol. 38, June 2010.

32. http://software.intel.com/en-us/articles/intel-math-kernel-library-intel-mkl-intel-mkl-100-threading/#5.

33. Kucharski, A. and P. Slobodzian, "The application of macromodels to the analysis of a dielectric resonator antenna excited by a cavity backed slot," *38th European Microwave Conference, EuMC 2008*, 519–522, October 27–31, 2008.