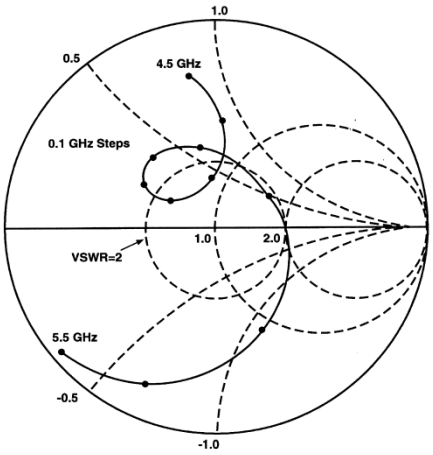
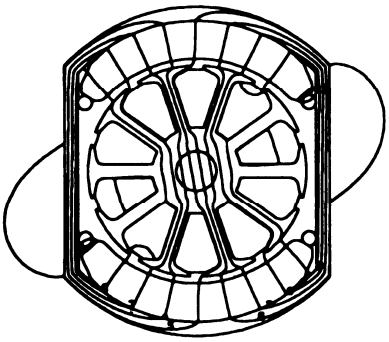
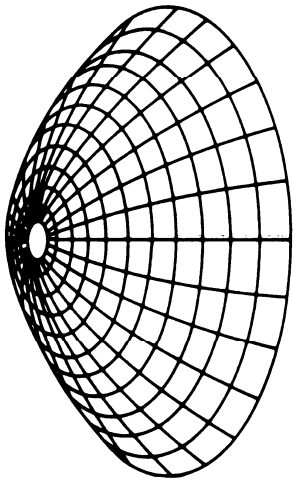
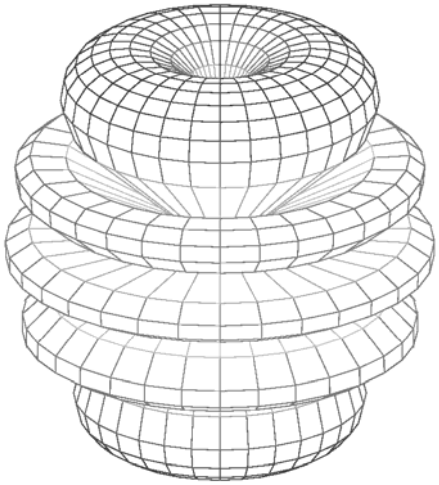








Applied Computational Electromagnetics Society Journal

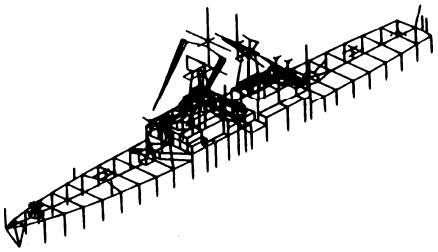
Special Issue on

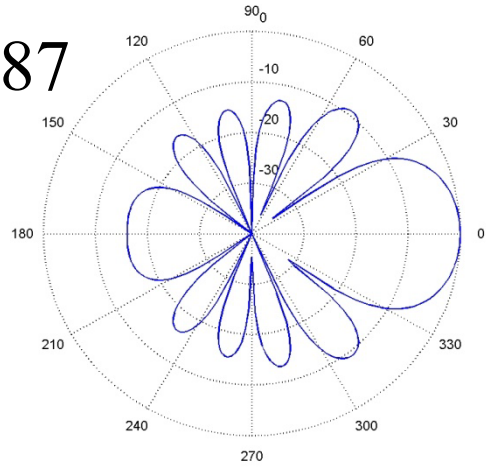
Hardware Accelerated Computational Techniques for Electromagnetic Simulations of Complex Problems

Guest Editors

Ozlem Kilic and Atef Z. Elsherbeni

April 2010
Vol. 25 No. 4
ISSN 1054-4887





GENERAL PURPOSE AND SCOPE: The Applied Computational Electromagnetics Society ***(ACES)*** Journal hereinafter known as the ***ACES Journal*** is devoted to the exchange of information in computational electromagnetics, to the advancement of the state-of-the art, and the promotion of related technical activities. A primary objective of the information exchange is the elimination of the need to "re-invent the wheel" to solve a previously-solved computational problem in electrical engineering, physics, or related fields of study. The technical activities promoted by this publication include code validation, performance analysis, and input/output standardization; code or technique optimization and error minimization; innovations in solution technique or in data input/output; identification of new applications for electromagnetics modeling codes and techniques; integration of computational electromagnetics techniques with new computer architectures; and correlation of computational parameters with physical mechanisms.

SUBMISSIONS: The ***ACES Journal*** welcomes original papers relating to applied computational electromagnetics. Typical papers will represent the computational electromagnetics aspects of research in electrical engineering, physics, or related disciplines. However, papers which represent research in applied computational electromagnetics itself are equally acceptable.

Manuscripts are to be submitted through the upload system of ***ACES*** web site http://aces.ee.olemiss.edu See "Information for Authors" on inside of back cover and at ***ACES*** web site. For additional information contact the Editor-in-Chief:

Dr. Atef Elsherbeni
Department of Electrical Engineering
The University of Mississippi
University, MS 386377 USA
Phone: 662-915-5382 Fax: 662-915-7231
Email: atef@olemiss.edu

SUBSCRIPTIONS: Members of the Applied Computational Electromagnetics Society who have paid their subscription fees are entitled to download any published journal article available at http://aces.ee.olemiss.edu, and have the option to receive the ***ACES Journal*** with a minimum of three issues per calendar year.

Back issues, when available, are $15 each. Subscriptions to ***ACES*** is available through the web site. Orders for back issues of the ***ACES Journal*** and changes of addresses should be sent directly to ***ACES***:

Dr. Allen W. Glisson
302 Anderson Hall
Dept. of Electrical Engineering
Fax: 662-915-7231
Email: aglisson@olemiss.edu

Allow four week's advance notice for change of address. Claims for missing issues will not be honored because of insufficient notice or address change or loss in mail unless the ACES Treasurer is notified within 60 days for USA and Canadian subscribers or 90 days for subscribers in other countries, from the last day of the month of publication. For information regarding reprints of individual papers or other materials, see "Information for Authors".

LIABILITY. Neither ***ACES***, nor the ***ACES Journal*** editors, are responsible for any consequence of misinformation or claims, express or implied, in any published material in an ***ACES Journal*** issue. This also applies to advertising, for which only camera-ready copies are accepted. Authors are responsible for information contained in their papers. If any material submitted for publication includes material which has already been published elsewhere, it is the author's responsibility to obtain written permission to reproduce such material.

APPLIED COMPUTATIONAL ELECTROMAGNETICS SOCIETY JOURNAL

Special Issue on
Hardware Accelerated Computational Techniques for Electromagnetic Simulations of Complex Problems

Guest Editors
Ozlem Kilic and Atef Z. Elsherbeni

April 2010
Vol. 25 No.42
ISSN 1054-4887

The ACES Journal is abstracted in INSPEC, in Engineering Index, DTIC, Science Citation Index Expanded, the Research Alert, and to Current Contents/Engineering, Computing & Technology.

The first, fourth, and sixth illustrations on the front cover have been obtained from the Department of Electrical Engineering at the University of Mississippi.

The third and fifth illustrations on the front cover have been obtained from Lawrence Livermore National Laboratory.

The second illustration on the front cover has been obtained from FLUX2D software, CEDRAT S.S. France, MAGSOFT Corporation, New York.

THE APPLIED COMPUTATIONAL ELECTROMAGNETICS SOCIETY

JOURNAL

Vol. 25 No. 4 April 2010

TABLE OF CONTENTS

Overview of Reconfigurable Computing Platforms and Their Applications in Electromagnetics Applications

Ozlem Kilic[1], Miaoqing Huang[2]
[1]Department of Electrical Engineering and Computer Science
The Catholic University of America, Washington, DC 20064, USA
kilic@cua.edu

[2]Department of Computer Science and Computer Engineering
University of Arkansas, Fayetteville, AR 72701, USA
mqhuang@uark.edu

***Abstract*—This paper investigates the utilization of field programmable gate arrays (FPGAs) in the acceleration of numerically intensive electromagnetics applications. We investigate the speed improvement by employing FPGAs for two different applications: (i) the optimization of a phased array antenna pattern by amplitude control using the ant colony optimization algorithm, (ii) implementation of the rigorous coupled wave (RCW) analysis technique for the design of engineered materials. The first application utilizes FPGAs as the only processor; i.e., all functionalities of the algorithm reside on the FPGA. The second one employs a hybrid hardware/software approach where the FPGA serves as a coprocessor to the CPU. The hybrid approach identifies the most numerically intensive part of the RCW algorithm and implements it on the FPGA. In both applications we demonstrate orders of magnitude of improvement in speed proving that FPGAs are highly flexible platforms suited well for the challenging electromagnetics problems. An overview of available FPGA platforms for scientific computing and how they compare are also presented in the paper.**

***Index Terms*—Field programmable gate array, Reconfigurable computing, Electromagnetics applications, Rigorous coupled wave analysis, Eigenvalue solver, Bio-inspired optimization, Ant colony optimization (ACO), Phased array.**

I. INTRODUCTION

The recent commercial and military applications for communications, imaging and remote sensing demand high mobility and multi-functionality. For instance, military applications require improved performance of their communication, radar and tracking systems while reducing size, cost and radar cross-section. Similarly, commercial communication devices are expected to perform seamlessly on the move for both voice and data exchange. In response, the research community has been investigating the use of advanced and engineered electromagnetic materials. We have witnessed the emerging of new classes of materials, such as meta-materials, photonic crystals, and plasmonics, etc [1]–[5]. These are typically complex heterogeneous mixtures of dielectric and metallic structures, which require rigorous electromagnetic simulation tools for an optimal design. Other applications involve smart antennas that can steer a beam electronically. The combined performance of the antenna with the beamformer can be a tedious task to simulate as the structure can consist of fine features with large overall dimensions, i.e., multiple wavelengths. However, the computations for such complex materials are often very cumbersome and time consuming. As a consequence, iterative design of advanced materials and simulations of antenna performance is often too slow to be of practical use.

There are many electromagnetic software packages that allow users to model complex 3-D structures. Many of these use one of the full-wave solutions such as Finite Difference Time Domain (FDTD) Method, Finite Element Method (FEM), Method of Moments (MoM), or asymptotic techniques like geometrical theory of diffraction (GTD), unified theory of diffraction (UTD), etc. The full wave solutions are limited to low-frequency applications or electrically small structures since they involve discretization of the geometry and the size of the problem becomes prohibitive for finer resolutions. The asymptotic methods are based on the assumption that the wavelength is much smaller than the finest part of the geometry. However, many of the practical applications involve modeling structures that possess fine details on large surfaces. The finer details suit well for full-wave solutions while the large surfaces are more appropriate for asymptotic approaches. Some typical applications are antennas on vehicle platforms, electrically large structures such as Rotman lens beam-formers [6], advanced RF material such as electronic band gap (EBG) and frequency selective surface (FSS) structures, and scattering properties of a medium with small and large features with respect to the wavelength.

This paper investigates the utilization of field programmable gate arrays (FPGAs) in the acceleration of numerically intensive electromagnetics applications as described above. FPGAs render themselves to parallel computing and can be customized to optimally fit the problem at hand, creating a highly efficient computing machine for the particular application. With the advancement of semiconductor technology, FPGAs have become mature enough to accommodate complicated computations. Due to the intrinsic parallelism of hardware

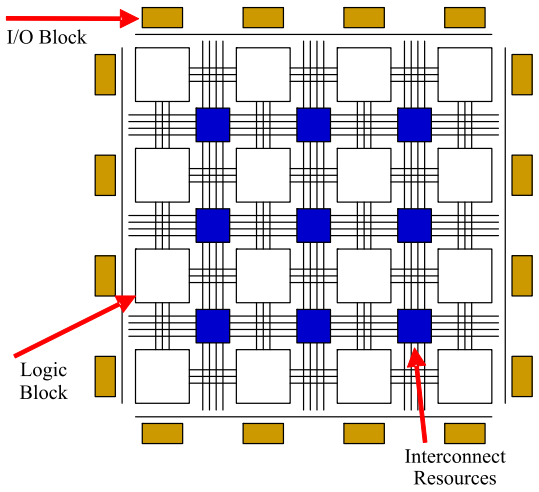


Fig. 1. The internal architecture of an FPGA device.

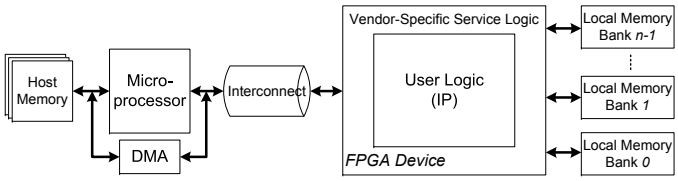


Fig. 2. The general architecture of a reconfigurable computer.

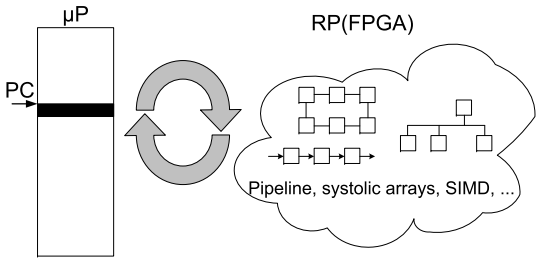


Fig. 3. The execution model of a reconfigurable computer.

implementation on FPGA devices, it is possible to achieve several orders of magnitude performance speedup compared with the corresponding implementation in software [7]. Therefore, FPGA devices have been integrated into the traditional workstations as co-processors. Generally, these workstations with addition of FPGA co-processors are called reconfigurable computers. As opposed to the specially designed ASICs, the functionality of the co-processor can be switched in milliseconds by downloading different configuration files (so the name "reconfigurable computing") into the FPGA device so that it can perform different types of operations.

The basic architecture of an FPGA device is shown in Fig. 1. FPGAs contain programmable logic components called "logic blocks", and a hierarchy of reconfigurable interconnects that allow the blocks to be "wired together". Logic blocks can be configured to perform complex combinational functions, or merely simple logic functions like AND and XOR. In most FPGAs, the logic blocks also include memory elements, which may be simple flip-flops or more complete blocks of memory. The I/O blocks surrounding the logic blocks provide the interface to communicate with the outside world. The two leading FPGA manufactures as of 2010 are Xilinx [8] and Altera [9]. FPGA devices from both companies are quite visible in reconfigurable computers as co-processors.

The rest of the paper is organized as follows. In Section II, we provide an overview of the available FPGA based platforms for scientific computing applications. Programming these devices involve understanding of parallelized and pipelined computing techniques. The details on how programming can be approached are provided in Section III, with a brief description of the differences between the currently available platforms. We discuss examples of electromagnetics applications and their implementation on FPGAs in Section IV. Finally, Section V concludes this work.

II. AVAILABLE RECONFIGURABLE COMPUTERS FOR SCIENTIFIC COMPUTING

Using FPGA devices as co-processors to microprocessors in reconfigurable computers (RC) has been an industrial interest and academic research topic for many years. Fig. 2 shows a simplified architecture of a reconfigurable computer including one FPGA and one microprocessor. The FPGA co-processor is equipped with several local memory banks, usually SRAM, acting as cache. An interconnect is used to transfer data between the FPGA and the microprocessor. An application implemented on reconfigurable computers is divided into two parts. The main flow is executed on the microprocessor. The computation intensive parts of the application can be implemented on the FPGA device by taking advantage of pipelining and parallelism, as shown in Fig. 3. FPGAs differ from a single-core microprocessor in their ability to execute thousands of operations concurrently. This is achieved by programming the logic blocks in the device. A single-core microprocessor, on the other hand, is only able to perform one operation at a time.

The reconfigurable (or hybrid) computers can be divided into two subcategories based on the different integration technology used. In the first subcategory, a PCI or PCI-Express based FPGA expansion card is inserted into a conventional workstation. In most cases, the FPGA card and the workstation are from different vendors. For the second subcategory, the same vendor will design both the FPGA board and the workstation, and integrate them together using a proprietary interconnect. Since these reconfigurable computers provide more computing capacity than those in the first subcategory, they are typically called high-performance reconfigurable computers (HPRCs). In this paper, we will focus on the use of HPRCs in the field of electromagnetics. Three example systems discussed in this paper are Cray XD1 [10], SRC-6 [11] and SGI RC100 [12].

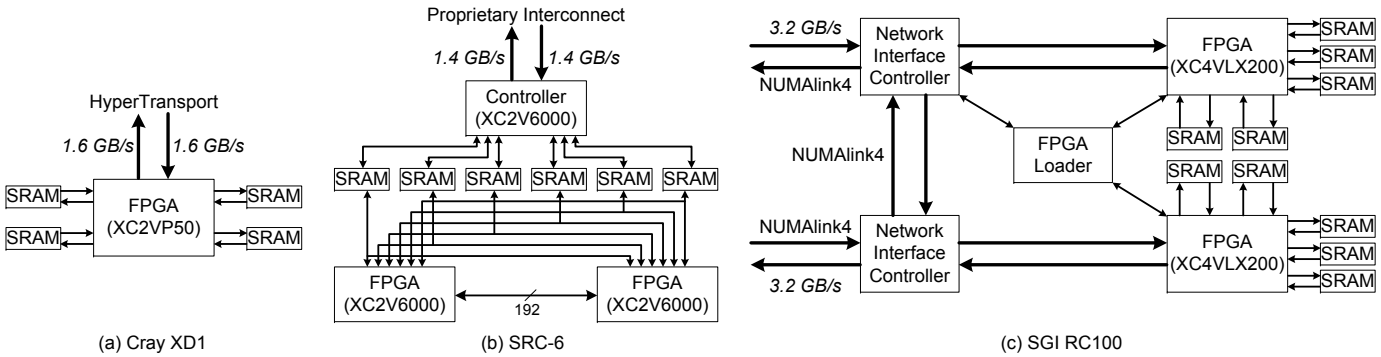


Fig. 4. The local architecture of the FPGA co-processor.

On the Cray XD1 platform, the FPGA co-processor resides on the same board as the microprocessor. This layout is different from the other two reconfigurable computers, which consist of separate FPGA board and microprocessor board. In spite of this difference, they share two similarities. (i) Multiple local SRAM modules are directly connected to the same FPGA device so that the hardware implementation can access and process multiple data blocks simultaneously, as shown in Fig. 4. On all three platforms, each memory access port is 64-bit wide. However, on both Cray XD1 and SGI RC100, the FPGA device has two separate read and write ports for each memory bank. In other words, the user logic on the FPGA device can read from and write to the same memory bank concurrently. On the other hand, the user logic on the SRC-6 platform has only one port for both reading from and writing to the same memory bank. This single-port access will degrade the performance for some applications. (ii) The FPGA co-processor is connected to the microprocessor and the host memory using high-speed interconnect in order to reduce the transportation overhead. These interconnects generally provide shorter latency and higher bandwidth for the data transfer between the FPGA and the microprocessor.

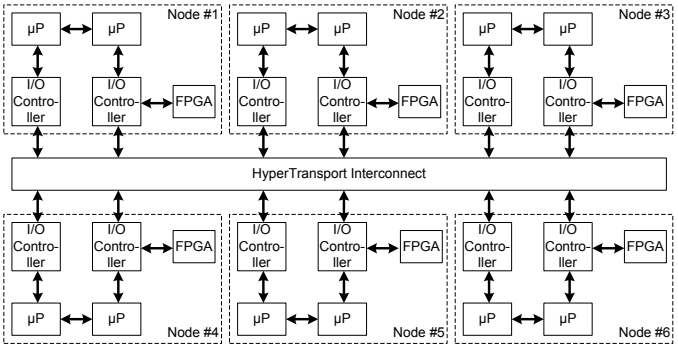
For an application that needs to use multiple FPGA devices at the same time in an RC system, different platforms deal with it differently.

- The Cray XD1 is a cluster-based reconfigurable computer. In other words, it may consist of dozens of FPGA co-processors, each of which belongs to a separate workstation. As shown in Fig. 5(a), 6 workstations (i.e., nodes) compose a chassis, which is the basic unit in a Cray XD1 system. If the user intends to use more than one FPGA co-processor, it has to cross the boundary of the operating system. One approach to use multiple FPGA devices in a single application is to use MPI (Message Passing Interface). Apparently, all the communication between any two FPGA co-processors has to be handled explicitly by software.
- The SRC-6 is a cluster-based platform as well. As shown in Fig. 4(b), there are two FPGA devices in one workstation. The user can program these two FPGA devices simultaneously in one application. These two FPGA devices can communicate to each other using the dedicated 192-bit channel. Once the communication is beyond the boundary of an operating system, MPI can be used for data transfer among different systems.
- The SGI RC100 is different from the other two platforms. On SGI RC100, different types of processing boards, i.e., microprocessor boards and FPGA boards, are connected to a same network and are visible in one single operating system, as shown in Fig. 5(c). However, as shown in Fig. 4(c), there is no communication channel between two FPGA devices on the same board. If the raw data can be divided into independent pieces, each of which is to be processed by one FPGA device, the user can allocate multiple FPGA co-processors in one software thread, and the co-processor driver will distribute the data evenly across different FPGAs. On the other hand, if the user wants to use multiple FPGAs and there is communication among these FPGAs during the data processing, a multiple-thread application is required to deal with this scenario.

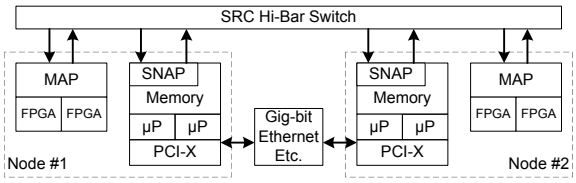
The implementation on the FPGA co-processor depends on the available resources, e.g., the available logic blocks in the FPGA device and the number of local memory banks. One example in the electromagnetics domain is the matrix multiplication, i.e., $C = AB$. Each element in C is the product of a row in A and a column in B, i.e., $c_{i,j} = \sum_{k=1}^{M} a_{ik} b_{kj}$. If matrix A and B are stored in two separate local memory banks and the result matrix C is saved in another separate memory bank, the user logic can read one pair of (a_{ik}, b_{kj}) every clock cycle assuming the multiplier and the accumulator are both fully pipelined. Therefore, it would take approximately M clock cycles to compute one element in matrix C no matter how complex the multiplication and the accumulation are.

III. PROGRAMMING RECONFIGURABLE COMPUTERS

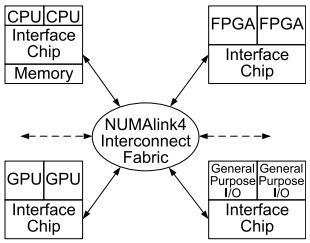
An application implemented on a reconfigurable computer consists of a hardware part and a software part as shown in Fig. 3. The user needs to program both parts and then integrate them together using vendor APIs (Application Programming Interfaces). The typical programming language for



(a) A Cray XD1 chassis.



(b) An SRC-6 consisting of 2 nodes.



(c) SGI RC100 architecture.

Fig. 5. The architecture of three representative reconfigurable computers.

```
// The block for source data reading
always @ (posedge clk) begin
    if (reset) begin
        mem_0_rd_cmd_vld     <= 0;
        mem_1_rd_cmd_vld     <= 0;
        mem_0_rd_addr        <= 0;
        mem_1_rd_addr        <= 0;
    end
    else begin
        if (mem_0_rd_addr == (MATRIX_RANK - 1)) begin
            mem_0_rd_cmd_vld <= 0;
            mem_1_rd_cmd_vld <= 0;
            mem_0_rd_addr    <= mem_0_rd_addr;
            mem_1_rd_addr    <= mem_1_rd_addr;
        end
        else begin
            mem_0_rd_cmd_vld <= 1;
            mem_1_rd_cmd_vld <= 1;
            mem_0_rd_addr    <= mem_0_rd_addr + 1;
            mem_1_rd_addr    <= mem_1_rd_addr + 1;
        end
    end
end

// The block for source data feeding
multiplier U1(.di1(mem_0_rd_data),.di2(mem_1_rd_data),
  .di_vld(mem_0_rd_data_vld),.do(product),.do_vld(pro_vld));
accumulator U2(.di(product),.di_vld(pro_vld),
  .do(ac),.do_vld(ac_vld));

// The block for result data writing
always @ (posedge clk) begin
    if (reset) begin
        mem_2_wr_cmd_vld     <= 0;
        mem_2_wr_addr        <= 0;
    end
    else begin
        mem_2_wr_cmd_vld     <= ac_vld;
        mem_2_wr_data        <= ac;
        if (mem_2_wr_cmd_vld) begin
            mem_2_wr_addr    <= mem_2_wr_addr + 1;
        end
        else begin
            mem_2_wr_addr    <= mem_2_wr_addr;
        end
    end
end
```

Fig. 6. Compute one element in matrix C using Verilog.

the software part is the C language in most cases. The more challenging part is the hardware part and it typically requires some hardware design expertise to gain the full benefit of using the FPGA co-processor.

It has been mentioned before that an FPGA co-processor is capable of performing thousands of operations concurrently. In order to achieve this concurrency, all the logic blocks in one FPGA device have to be programmed into a specific status by using a configuration file. Since the implementation depends on the available hardware resources on the FPGA device (e.g., memory, built-in multipliers, logic blocks), it might be necessary at times to distribute the hardware part into multiple FPGA configurations, each of which is called a *bitstream*. At runtime, different configurations are downloaded into the FPGA device following a pre-defined order in an application.

There are two different approaches for the user to implement the functions running on the FPGA co-processor, i.e., hardware description languages (HDLs) and high-level languages (HLLs). The default languages to program the FPGA device are HDLs, i.e., VHDL and Verilog HDL. In the meantime, there are several HLLs available, e.g., Carte-C [11], Impulse C [13], Handel-C [14], and Mitrion-C [15], bringing the ease of use at the expense of efficiency.

If HDLs are used to design the bitstream, it will require three parallel blocks to implement the matrix multiplication example, as shown in Fig. 6.

- One block is to control the source data reading from two memory banks saving matrix A and B. The functionality of this block involves the generation of reading addresses and reading commands.
- One block is to check the arrival of source data and feed them into the multiplier and the accumulator.
- One block is to control the result data writing to another memory bank for matrix C. The functionality of this block involves the generation of writing addresses and writing commands.

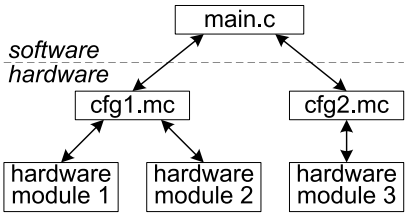


Fig. 7. Implement an application on SRC-6 using Carte-C.

```
/* Define three 2D arrays of 400x400 in three local *
 * memory banks                                      */
OBM_BANK_A_2D (A, double, 400, 400)
OBM_BANK_B_2D (B, double, 400, 400)
OBM_BANK_C_2D (C, double, 400, 400)
......
ac = 0;
for (k=0; k<400; k++) {
    Multiplier(A[i][k], B[k][j], &product);
    Accumulator(product, &ac);
}
C[i][j] = ac;
```

Fig. 8. Compute one element in matrix C using Carte-C.

On both Cray XD1 and SGI RC100 platforms, either VHDL or Verilog can be used to generate the bitstream. In order to reduce the complexity of communications with the local memory, the vendor generally provides the service logic (as shown in Fig. 2), which gives a simplified interface to access the local memory as well as the interconnect.

On SRC-6, the vendor provides a high-level language, i.e., Carte-C, to implement the hardware part. Carte-C is a rich subset of C, with non-standard extensions to control hardware instantiation and parallelism. Each FPGA bitstream is defined by a single Carte-C file, which is converted into HDL during the compilation. For instance, in the case shown in Fig. 7, the hardware part is distributed into two bitstreams, described in two Carte-C files, i.e., *cfg1.mc* and *cfg2.mc*. A single Carte-C file consists of multiple blocks, which are executed in a sequence during the runtime. The Carte-C compiler will maximize the parallelism within a single block to improve the performance. It is difficult for the compiler to achieve the maximum performance for complicated operations. In this case a hand-written HDL module can be integrated into the bitstream, leaving the main flow written in Carte-C. As demonstrated in Fig. 7, two hardware modules are integrated into the first bitstream file.

Fig. 8 shows a section of codes to compute one element in the resultant matrix, in which Multiplier and Accumulator are two pipelined HDL modules. If A, B and C are stored in three separate memory banks, the Carte-C compiler is capable of generating fully pipeline hardware codes for the maximum performance.

Carte-C is a proprietary language used on the SRC-6 platform; i.e., it does not extend to other platforms. Other HLLs can be used across different platforms. For example, both Impulse C and Mitrion-C can be used to program Cray XD1 and SGI RC100.

IV. DEVELOPING ELECTROMAGNETICS APPLICATIONS ON RECONFIGURABLE COMPUTERS

A. Background Information

Electromagnetics applications tend to be numerically intensive, with most problems requiring memory intensive implementations. Complex structures can be analyzed using numerical methods by segmenting the structure into small meshes. Often these meshes can be treated independently from the rest of the geometry with the use of appropriate boundary conditions. This allows analysis to be carried out in a parallel fashion.

In terms of utilizing hardware acceleration in electromagnetics applications, there is an increasing interest in the use of general purpose graphics processing units [16], [17] mostly due to their C-like implementation and relatively low cost. The use of VLSIs has also been suggested in [18] and [19]. However, the FPGA implementation of electromagnetics algorithms has been very scarce due to the hardware expertise required on these platforms. One area of numerical electromagnetics that has been investigated for the FPGA implementation is the FDTD algorithm [20]–[23]. FDTD expresses Maxwell's equations in difference form and renders itself to parallel implementation as each cell can be handled separately from the others. A three dimensional FDTD model on hardware has been reported in [24], where an FPGA-based accelerator has been used in conjunction with a host PC and a CAD interface. This algorithm has been applied to the analysis of a Rotman lens in [25].

In the following sections we demonstrate the FPGA implementation of two applications: (i) optimization of a linear array antenna pattern using the ant colony optimization technique, (ii) implementation of the rigorous coupled wave analysis algorithm. The first implementation utilizes the FPGA as the sole processor with the CPU functionality being to call the FPGA and retrieve the final result. The second implementation uses a hybrid hardware/software approach where only the most numerically intensive components of the algorithm resides on the FPGA.

B. Ant Colony Optimization (ACO) Implementation

The utilization of FPGAs in the field of electromagnetics was recently investigated by applying the ant colony optimization (ACO) method in the design of phased array antennas for multiple beam satellite communication systems [26]. In this application, the amplitudes of the array elements were optimized to reduce the co-channel interference in a multiple beam satellite communication system. Potential gains in the speed of the calculations in the order of 10,000 has been demonstrated for the particular application. A brief overview of the problem solved and how the FPGA was utilized is discussed in this section.

1) ACO Algorithm: The ACO is a nature inspired optimization algorithm that utilizes heuristic search principles carried out simultaneously by agents and their collective intelligence.

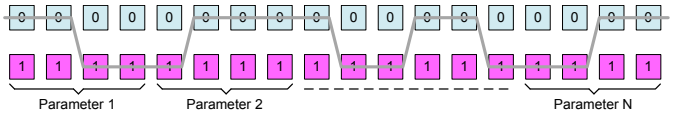


Fig. 9. Path Definition in ACO.

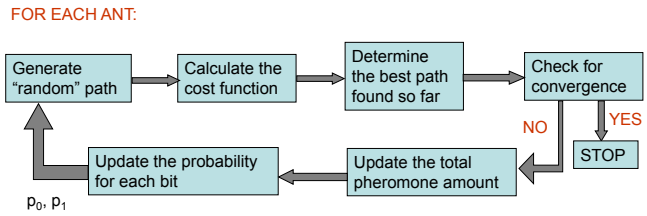


Fig. 10. Implementation of ACO on CPU - Recursive for each ant.

The ACO mimics the behavior of ants in their search for the shortest path between their nest and the food. Despite being nearly blind, ants demonstrate the capability to establish the shortest path between their nest and food. They achieve this by depositing a chemical substance called pheromone on their paths, which is used later on by other ants in their search process. During this process, the most traveled path is marked with the highest level of pheromone. This positive feedback behavior allows more ants to choose the path with the most pheromone amount [27]. The random search is iteratively applied by the ants until one of the chosen paths satisfies the required convergence criteria. The intelligence is introduced to the random search process via the cost function, which measures how far off an ant is from the desired solution. Since each solution is represented by a path in ACO, the optimization space is discretized into binary strings where a path is defined by the choice of 1 or 0 for the bit value at each bit position as shown in Fig. 9 [28].

Each path represents a possible solution and a number of ants sample the solution space at each iteration. Once all ants decide on their paths, the cost function is computed for each path. The cost is a measure of how satisfactory a solution is, with low cost values implying a "better" solution. The pheromone amount to be laid on each path is inversely proportional to the cost value associated with the path. The probability of a zero for each bit position is then calculated for all the ant paths as a function of the total pheromone levels on the path as follows:

$$p_0 = \frac{\tau_0}{\tau_0 + \tau_1} \tag{1}$$

where τ_0 and τ_1 correspond to the total pheromone levels accumulated at the bit position of interest for bit value of zero and one, respectively. The probability for bit value of one is then calculated as $1 - p_0$ for each bit. A block diagram of the algorithm is shown in Fig. 10, where each ant is processed iteratively on a typical software implementation.

2) Application - Linear Array Optimization: The ACO algorithm is used to optimize the radiation from a linear array.

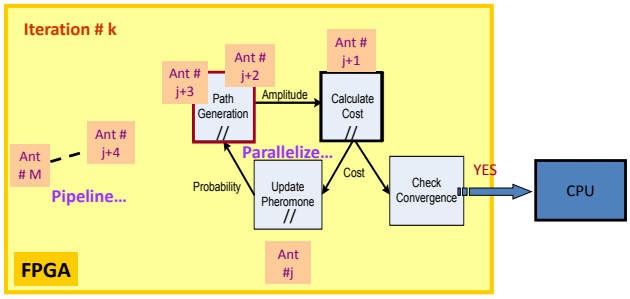


Fig. 11. Implementation of ACO on FPGA - parallelized and pipelined.

Nulls are placed at certain positions to reduce potential interference in a multiple beam satellite communication system. To achieve the desired radiation, the amplitudes of the array elements are optimized.

This implementation is unique in the sense that the FPGA has been utilized as the sole processer for the entire implementation. This avoids any overhead of communication between the microprocessor and the FPGA, and uses the FPGA to its full potential. The CPU is only used to call the FPGA and retrieve the results for processing. In short, an ACO machine has been implemented with this application. While this enables the ultimate parallelization and pipelining of the algorithm, there are limitations due to the problem size that can be handled by the particular FPGA at hand. The implementation was carried out on the SGI Altix 450 platform utilizing the Xilinx Virtex4LX200 FPGAs as demonstrated in Fig. 11.

Paths are produced using 8 bits for each optimization parameter (i.e., the amplitudes of the array elements), 40 parameters in each ant path (i.e., the number of array elements), 40 ant paths per iteration (i.e., 40 ants carry search for a solution simultaneously in each iteration), and as many iterations as it takes to converge, with an upper limit set by the user. With this implementation, increasing the number of bits per parameter will increase the FPGA resource requirement for this function, but will not increase the processing time. The number of nulls that can be achieved is also run in parallel, i.e., has no impact on the processing time as long as there are sufficient FPGA resources. We were able to carry out 8 bits and three nulls in parallel at a clock rate of 100 MHz for the optimization of a 40 element array on the Altix platform.

As observed in Fig. 11, there are three major sections in the algorithm: Path Generation, Cost Calculation and Pheromone Update. The details of the implementation of these sections on FPGA is given in [26].

As a test, only three null positions were required to be below -30 dB. When the algorithm was run on a standard PC (CPU: Intel Pentium M, 3 GHz and RAM: 1 GB) using Matlab, the time per a single iteration took about 0.47 seconds. The same algorithm when implemented on C and run on the same platform ran about 53.4 times faster than the Matlab version, roughly at 8.8 milliseconds per iteration. The VHDL implementation on the Altix 450 system performed at 31.3

microseconds for runs after the bit loading was completed, resulting in a factor of 15,160 in speed compared with the Matlab implementation.

C. Rigorous Coupled Wave Analysis Implementation

The previous application was small enough to be implemented fully on FPGA utilizing the platform to its full potential and achieving very promising acceleration. The ACO implementation was fully optimized to achieve this kind of acceleration. However, increasing the number of nulls or number of bits per variable are not feasible as the algorithm would cease to fit on the FPGA. The problems in electromagnetics are often complex, and require flexibility in the range of the parameters. The second application is one such example. The Rigorous Coupled Wave (RCW) algorithm applies to diffraction problems from multiple layers with periodic gratings. It is based on an extension of enhanced transmittance matrix approach [29] and adopts Lalanne's improved eigenvalue formalism [30]. A detailed discussion on the RCW algorithm can be found in these references. It has been used effectively in the design of engineered materials, such as antireflective surfaces [31]–[33]. We provide a brief overview in this section in order to describe our motivations for the hardware implementation.

The stacked multiple layer in RCW algorithm can consist of any number of gratings. However, all gratings must be periodic with the same periodicity along a given direction on the plane. The periodicity results in a spatially periodic permittivity (and inverse permittivity) within each layer and can be represented as a Fourier series expansion, as follows.

$$\varepsilon_l(x,y) = \sum_{g,h} \varepsilon_{l,gh} \exp\left(j\frac{2\pi g x}{\Lambda_x} + j\frac{2\pi h y}{\Lambda_y}\right) \tag{2a}$$

$$\varepsilon_l^{-1}(x,y) = \sum_{g,h} A_{l,gh} \exp\left(j\frac{2\pi g x}{\Lambda_x} + j\frac{2\pi h y}{\Lambda_y}\right) \tag{2b}$$

where $\varepsilon_{l,gh}$ and $A_{l,gh}$ are the Fourier coefficients for the lth layer in the stack for the permittivity and inverse permittivity respectively. The electric field inside the layers can similarly be expressed as a Fourier series in terms of spatial harmonics. Maxwell's equations for the layered structure can be written in terms of the tangential components of the electric and magnetic fields, resulting in a coupled equation set in (3), where S_l represents the amplitudes of the spatial harmonics of the electric field in the lth layer, with subscripts x and y denoting the directions of periodicity in the plane of the stack. The parameters B and D in (3b) are matrices given as $B = k_x \varepsilon_l^{-1} k_x - I$ and $D = k_y \varepsilon_l^{-1} k_y - I$.

Thus, the coupled wave equation can be solved by finding the eigenvalues of the matrix Ω_l, which is a function of the stack properties. The rank of this matrix is $M \times N$, where M and N are the number of spatial harmonics retained along the two dimensions of periodicity in the plane of stacked layers. Ideally an infinite number of them are needed for an exact solution but truncation with minimal error is possible. Despite this truncation, the rank can be in the order of magnitude of 400 or more for a typical application of AR surface design. Hence, the most numerically intensive component of the RCWA algorithm is this eigenvalue computation.

Algorithm 1: Hessenberg Reduction

Input: A square complex matrix A with rank n
Output: The reduced Hessenberg matrix H

1.1 **for** k=*0* **to** $n-3$ **do**
1.2 $v_k = \textbf{House}(A_{k+1:n-1,k})$; /*Step 1: See Alg. 2*/
1.3 $A_{k+1:n-1,k:n-1} = A_{k+1:n-1,k:n-1} - 2v_k(v_k^* A_{k+1:n-1,k:n-1})$; /*Step 2: $P_k A_{k+1:n-1,k:n-1}$, $P_k = I - 2v_k v_k^*$*/
1.4 $A_{0:n-1,k+1:n-1} = A_{0:n-1,k+1:n-1} - 2(A_{0:n-1,k+1:n-1} v_k)v_k^*$; /*Step 3: $A_{0:n-1,k+1:n-1} P_k$*/

Algorithm 2: House(x)

Input: A complex vector x
Output: The Householder vector v

2.1 $\alpha = -e^{i\varphi}\|x\|$; /*$\varphi$ is the argument of x_1*/
2.2 $u = x - \alpha e_1 = x + e^{i\varphi}\|x\|e_1$; /*$e_1 = [1, 0, ..., 0]^T$*/
2.3 $v = \frac{u}{\|u\|}$;

1) QR Eigenvalue Algorithm: Given a square matrix $A \in \mathbb{C}^{n\times n}$, an eigenvalue λ and its associated eigenvector $\mathbf{v}$ are, by definition, a pair obeying the relation $A\mathbf{v} = \lambda\mathbf{v}$. Equivalently, $(A - \lambda I)\mathbf{v} = 0$ (where I is the identity matrix), implying $\det(A - \lambda I) = 0$. This determinant can be expanded into a polynomial in λ, known as the *characteristic polynomial* of A. One common method for determining the eigenvalues of a small matrix is by finding the roots of its characteristic polynomial. However, a general polynomial of order $n > 4$ cannot be solved by a finite sequence of arithmetic operations and radicals. Therefore, many numerical iterative algorithms have been proposed [34] to solve the eigenvalue problem of high-rank square matrices, such as Power Method, Inverse Iteration, Jacobi Method, etc. Among these, the shifted Hessenberg QR algorithm [35]–[37] is accepted as a practical solution adopted in most applications to deal with general square matrices.

There are two phases in the practical QR algorithm, as described in (4). In the first phase, the original matrix A is reduced to the upper Hessenberg form H using the Householder transformation [38]. The second phase involves applying the implicit QR iteration with shifts on the unreduced Hessenberg matrix H until it converges to a triangular matrix, i.e., the Schur form S. The eigenvalues of a triangular matrix are listed on the diagonal, i.e., the $\otimes$s in (4), and the eigenvalue problem is solved once this form is achieved.

2) Implementation on SGI RC100 Reconfigurable Computer: The RCW algorithm in the most general sense creates a square matrix with complex values. Both real and imaginary parts of a matrix entry are represented in double precision (64-bit) floating-point format. In the hardware implementation of QR eigenvalue algorithm on FPGA device, we combine the two physical local memory banks into a 128-bit wide logical memory bank so that each memory entry can store one com-

$$\begin{bmatrix}\partial^2 S_{l,y}/\partial z'^2\\ \partial^2 S_{l,x}/\partial z'^2\end{bmatrix} = \Omega_l \begin{bmatrix}S_{l,y}\\ S_{l,x}\end{bmatrix} \tag{3a}$$

$$\Omega_l = \begin{bmatrix}k_x^2 + D[\alpha\varepsilon_l + (1-\alpha)A_l^{-1}] & k_y\{\varepsilon_l^{-1}k_x[\alpha A_l^{-1} + (1-\alpha)\varepsilon_l] - k_x\}\\ k_x\{\varepsilon_l^{-1}k_y[\alpha\varepsilon_l + (1-\alpha)A_l^{-1}] - k_y\} & k_y^2 + B[\alpha A_l^{-1} + (1-\alpha)\varepsilon_l]\end{bmatrix} \tag{3b}$$

$$\underset{A}{\begin{bmatrix}\times&\times&\times&\times&\times\\ \times&\times&\times&\times&\times\\ \times&\times&\times&\times&\times\\ \times&\times&\times&\times&\times\\ \times&\times&\times&\times&\times\end{bmatrix}} \xrightarrow{\text{Phase 1}} \underset{\text{Hessenberg } H}{\begin{bmatrix}\times&\times&\times&\times&\times\\ \times&\times&\times&\times&\times\\ 0&\times&\times&\times&\times\\ 0&0&\times&\times&\times\\ 0&0&0&\times&\times\end{bmatrix}} \xrightarrow{\text{Phase 2}} \underset{\text{Triangular } S}{\begin{bmatrix}\otimes&\times&\times&\times&\times\\ 0&\otimes&\times&\times&\times\\ 0&0&\otimes&\times&\times\\ 0&0&0&\otimes&\times\\ 0&0&0&0&\otimes\end{bmatrix}} \tag{4}$$

Table 1. Calculation breakdown of iteration k in Hessenberg reduction.

Step	Sub-step	Calculation	Number of clock cycles for computation*	
1	1.1	$\|x\|,\ \|x_1\|$	$n-k-1$	$3k^2-9nk+6n^2-3n-2$
	1.2	$x_{1\_r} + \|x\|\cos\varphi,\ x_{1\_i} + \|x\|\sin\varphi$	1	
	1.3	$\|u\|$	$n-k-1$	
	1.4	$u/\|u\|$	$n-k-1$	
2	2.1	$m = v_k^* A_{k+1:n-1,k:n-1}$	$(n-k)(n-k-1)$	
	2.2	$N = v_k m$	$(n-k)(n-k-1)$	
	2.3	$A_{k+1:n-1,k:n-1} - 2N$	$(n-k)(n-k-1)$	
3	3.1	$m' = A_{0:n-1,k+1:n-1} v_k$	$n(n-k-1)$	
	3.2	$N' = m' v_k^*$	$n(n-k-1)$	
	3.3	$A_{0:n-1,k+1:n-1} - 2N'$	$n(n-k-1)$	

*Ignoring all latencies.

plete matrix entry. Therefore, the real part and the imaginary part of a complex variable can be accessed simultaneously.

As described earlier, there are two phases in the QR algorithm. These phases are implemented in two separate FPGA configurations. The first phase, Hessenberg reduction, is carried out by applying the Householder reflection for $n-2$ iterations (see Alg. 1), where n is the rank of the original matrix A. Each iteration comprises three steps, as shown in Table 1. Each step further includes multiple sub-steps. In our hardware design, Steps 1, 2 and 3 comprise 4, 3 and 3 sub-steps, respectively. All iterations, the steps in each iteration, and the sub-steps within every step have to be carried out sequentially due to the data dependency among them. The advantage of hardware implementation comes from the parallel processing within each sub-step. For example, Sub-step 1.1 involves multiplication, addition, accumulation and square root operation to calculate the norm of a vector. If all the basic operators, e.g., multipliers and adders, are fully pipelined, it will take roughly $n-k-1$ clock cycles to finish this sub-step (if we ignore all potential latencies). By putting everything together, the total number of clock cycles required to reduce a matrix of rank n to its Hessenberg form can thus be computed as:

$$\sum_{k=0}^{n-3}(3k^2 - 9nk + 6n^2 - 3n - 2) = \frac{5}{2}n^3 - \frac{9}{2}n - 11. \tag{5}$$

The second phase of the QR algorithm is to convert the upper Hessenberg matrix to its upper triangular form, which is implemented as a hardware/software co-design (see pp. 359 in [39] for a detailed description of the algorithm). The main step of the second phase is the Francis QR Step, in which the most computation demanding part is implemented in hardware as a separate FPGA configuration. Its functionality is shown in Alg. 3. By comparing Alg. 1 and Alg. 3, it can be found that Alg. 3 is a shrunk version of Alg. 1. In other words, the implementation of both algorithms will share the majority of their logic such as the floating-point operators and the control flow. Some small modifications are required to reduce the scope of the computation in Alg. 1 to match the functionality of Alg. 3. In general, the computation in the Francis QR Step is significantly less than the computation in the Hessenberg reduction phase.

Algorithm 3: Francis QR Step (hardware part)

Input: A square complex matrix H with rank n
Output: Matrix H
3.1 **for** k=0:$n-5$ **do**
3.2 $v_k = \textbf{House}(H_{k+1:k+3,k})$;
3.3 $H_{k+1:k+3,k:n-1} = H_{k+1:k+3,k:n-1} - 2v_k(v_k^* H_{k+1:k+3,k:n-1})$;
3.4 $H_{0:k+4,k+1:k+3} = H_{0:k+4,k+1:k+3} - 2(H_{0:k+4,k+1:k+3}v_k)v_k^*$;

3) Results: The hardware implementation of Hessenberg reduction occupies 56,520 (63%) slices on the target FPGA device and runs at 100 MHz. The basic operators, i.e., mul-

Table 2. Performance improvement of Hessenberg reduction.

Matrix Rank	Computation Time (s)		Speedup	Matrix Rank	Computation Time (s)		Speedup	Matrix Rank	Computation Time (s)		Speedup
	Hardware*	Software			Hardware*	Software			Hardware*	Software	
20	0.007	0.062	**9.4**	180	0.161	428.911	**2663.3**	340	1.019	5476.617	**5375.0**
40	0.008	1.020	**123.4**	200	0.217	654.289	**3013.0**	360	1.206	6964.269	**5773.9**
60	0.013	5.209	**410.2**	220	0.285	957.911	**3355.5**	380	1.415	8696.029	**6144.3**
80	0.021	16.553	**789.6**	240	0.367	1358.445	**3696.9**	400	1.647	10717.100	**6505.7**
100	0.034	40.516	**1184.9**	260	0.464	1870.955	**4035.2**	420	1.904	13055.131	**6858.1**
120	0.054	84.318	**1574.2**	280	0.576	2516.849	**4371.2**	440	2.185	15750.099	**7207.7**
140	0.080	156.366	**1944.8**	300	0.705	3318.075	**4707.9**	460	2.493	18859.268	**7563.6**
160	0.116	267.548	**2309.5**	320	0.852	4293.784	**5038.9**	480	2.829	22393.864	**7914.8**

*Including data transportation time and data processing time.

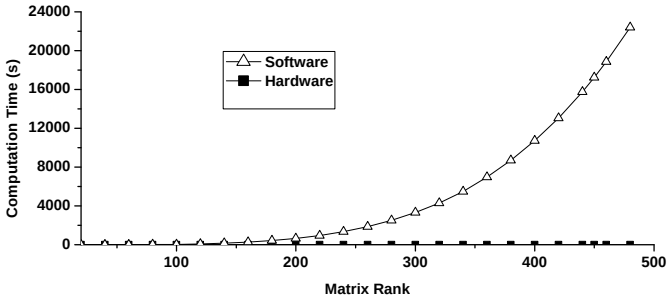


Fig. 12. Computation time of Hessenberg reduction.

tipliers, adders, subtractor, dividers and square rooters, are generated using CORE Generator, which is a tool included in the Xilinx ISE package. The rank of the object matrix is passed to hardware design as a parameter through a register. Before the FPGA starts processing, the original matrix as well as its rank are transferred from host memory to FPGA local memory. After the processing is finished, the upper Hessenberg matrix is transferred back to host memory. We tested matrices of different ranks and collected their corresponding hardware computation times, as listed in Table 2 and Fig. 12. The hardware computation time consists of both data transportation time and data processing time. It is found that the measured time matches the estimation using (5) in all cases.

For a comparison of acceleration over a pure software based implementation, we coded the Hessenberg reduction phase in C++ and ran it on a PC with Itanium 2 using a 1.6 GHz microprocessor. The speedup between is in the order of thousands (as shown in Fig. 12), which is mainly due to two factors. (i) The hardware implementation is fully pipelined, which means that multiple operations can be processed concurrently. On the other hand, the microprocessor has to process these operations in a sequential means. (ii) FPGA devices are equipped with large amount of directly accessible local memory, e.g., 40 MB on Altix RASC RC100. The local memory of FPGA devices can be compared to the L1/L2 cache of microprocessors, which are much smaller in terms of capacity. As we can see from Alg. 1, the Hessenberg reduction operation spans on all the matrix, along both columns and rows. Since the local memory of the FPGA device is quite large, it is able to accommodate the whole matrix. On the other hand, the Hessenberg reduction operation on the microprocessor is accompanied by frequent data swapping among the L1 cache, the L2 cache and the main memory, which contributes a lot of overhead in the software implementation.

The hardware implementation of Alg. 3 takes almost the same resources (i.e., 56,327 (63%) slices) on the FPGA device and runs at the same frequency. The interface to the second bitstream is the same as the first one. We applied the same type of comparison between the hardware implementation and the software version on the Francis QR Step. For a 480×480 matrix, the computation time is 0.450 s for software and 0.063 s for hardware, respectively. In other words, the hardware implementation is able to outperform the corresponding software version by 7.2 folds for those matrices we are interested. The comparatively small performance improvement is mainly due to the dramatic reduction of computation in Alg. 3. For example, the computation in line 3.4 in Alg. 3 only involves $3k + 15$ matrix elements. The corresponding line (i.e., line 1.4) in Alg. 1 involves $\mathcal{O}(n^2)$ matrix elements. Therefore, the advantage of a deep pipeline is more evident in the hardware implementation for Hessenberg reduction.

V. CONCLUSIONS

The potential use of FPGAs in electromagnetics has been demonstrated in the context of two applications: (i) the optimization of a linear array using the ant colony optimization (ii) implementation of rigorous coupled wave analysis method. The first application renders itself to parellel computing as the ant colony optimization is based on sampling of the optimization space simultaneously by a set of "ants". Like in many other heuristic search algorithms, the simultaneous search is independent of each other in each iteration while the agents gather collective intelligence. The problem investigated was small enough to fit fully on a single FPGA, enabling remarkable speed improvement (in the order of 15,000). This application demonstrated the ultimate power of FPGAs when the platform and problem are a perfect fit. The second application involved a more challenging task, where the FPGA was utilized as a co-processor to the CPU, mainly carrying out the most numerically intensive part of the algorithm. The task was the computation of eigenvalues of a complex matrix with rank of 400 or more. We have used the QR eigen value algorithm and implemented

the Hessenberg reduction and Francis QR methods on the FPGA. We have observed speed improvement in the order of thousands, with increased efficiency as the matrix rank increases. While FPGAs are finding their way slowly in the scientific computing area due to the challenges in being able to implement code using hardware description languages, their potential in providing reconfigurable parallelism make them an attractive platform.

ACKNOWLEDGMENT

The authors would like to thank Charles Conner for the software implementation of Hessenberg reduction and the integration of the FPGA bitstreams into the QR algorithm.

REFERENCES

[1] J. Witzens, M. Lončar, and A. Scherer, "Self-collimation in planar photonic crystals," *IEEE J. Sel. Topics Quantum Electron.*, vol. 8, no. 6, pp. 1246–1257, Nov. 2002.

[2] S. Y. Lin, E. Chow, S. G. Johnson, and J. D. Joannopoulos, "Demonstration of highly efficient waveguiding in a photonic crystal slab at the 1.5-μm wavelength," *Optics Letters*, vol. 25, no. 17, pp. 1297–1299, Sep. 2000.

[3] M. Notomi, K. Yamada, A. Shinya, J. Takahashi, C. Takahashi, and I. Yokohama, "Extremely large group-velocity dispersion of line-defect waveguides in photonic crystal slabs," *Physical Review Letters*, vol. 87, no. 25, pp. 253 902–1–253 902–4, Dec. 2001.

[4] M. Lončar, D. Nedeljkovic, T. Doll, J. Vučković, A. Scherer, and T. P. Pearsall, "Waveguiding in planar photonic crystals," *Applied Physics Letters*, vol. 77, no. 13, pp. 1937–1939, Sep. 2000.

[5] S. John, "Strong localization of photons in certain disordered dielectric superlattices," *Physical Review Letters*, vol. 58, no. 23, pp. 2486–2489, Jun. 1987.

[6] O. Kilic and R. Dahlstrom, "Rotman lens beam formers for army multifunction RF antenna applications," in *Proc. IEEE AP-S International Symposium and USNC/URSI National Radio Science Meeting,* vol. 2B, pp. 43–46, Jul. 2005.

[7] T. El-Ghazawi, E. El-Araby, M. Huang, K. Gaj, V. Kindratenko, and D. Buell, "The promise of high-performance reconfigurable computing," *IEEE Computer*, vol. 41, no. 2, pp. 78–85, Feb. 2008.

[8] *http://www.xilinx.com*.

[9] *http://www.altera.com*.

[10] *Cray XD1™FPGA Development (S-6400-14)*, Cray Inc., May 2006.

[11] *SRC Carte™C Programming Environment v2.2 Guide (SRC-007-18)*, SRC Computers, Inc., Aug. 2006.

[12] *Reconfigurable Application-Specific Computing User's Guide (007-4718-007)*, Silicon Graphics, Inc., Jan. 2008.

[13] *Impulse C – http://www.impulsec.com*, Impulse Accelerated Technologies, Inc., 2009.

[14] *Handel-C Language Reference Manual*, Agility Design Solutions Inc., 2007.

[15] *Mitrion C – http://www.mitrionics.com*, Mitrionics AB, 2009.

[16] M. J. Inman and A. Z. Elsherbeni, "Programming video cards for computational electromagnetics applications," *IEEE Antennas Propag. Mag.*, vol. 47, no. 6, pp. 71–78, Dec. 2005.

[17] N. Takada, T. Takizawa, Z. Gong, N. Masuda, T. Ito, and T. Shimobaba, "Fast computation of 2-D finite-difference time-domain method using graphics processing unit with unified shader," *IEICE Trans. Inf. Syst.*, vol. J91-D, no. 10, pp. 2562–2564, 2008.

[18] J. R. Marek, M. A. Mehalic, J. Andrew, and J. Terzuoli, "A dedicated VLSI architecture for Finite-Difference Time Domain calculations," in *Proc. 8th ACES Conference*, 1992.

[19] P. Placidi, L. Verducci, G. Matrella, L. Roselli, and P. Ciampolini, "A custom VLSI architecture for the solution of FDTD equations," *IEICE Transactions on Electronics*, vol. E85-C, no. 3, pp. 572–577, Mar. 2002.

[20] L. Verducci, P. Placidi, G. Matrella, L. Roselli, F. Alimenti, P. Ciampolini, and A. Scorzoni, "A feasibility study about a custom hardware iplementation of the FDTD algorithm," in *Proc. the 27th General Assembly of the URSI*, 2002.

[21] J. P. Durbano, *Hardware implementation of a 1-dimensional Finite-Difference Time-Domain algorithm for the analysis of electromagnetic propagation*. M.E.E.Thesis, Department of Electrical and Computer Engineering, University of Delaware, Newark, USA, 2002.

[22] J. P. Durbano, F. E. Ortiz, J. R. Humphrey, D. W. Prather, and M. S. Mirotznik, "Implementation of three-dimensional FPGA-based FDTD solvers: An architectural overview," in *Proc. 11th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM03)*, pp. 269–270, Apr. 2003.

[23] R. N. Schneider, L. E. Turner, and M. M. Okoniewski, "Application of FPGA technology to accelerate the Finite-Difference Time-Domain (FDTD) method," in *Proc. the 10th ACM International Symposium on Field-Programmable Gate Arrays*, pp. 97–105, 2002.

[24] J. P. Durbano, J. R. Humphrey, F. E. Ortiz, P. F. Curt, D. W. Prather, and M. S. Mirotznik, "Hardware acceleration of the 3D finite-difference time-domain method," in *Proc. IEEE AP-S International Symposium and USNC/URSI National Radio Science Meeting*, pp. 77–80, Jun. 2004.

[25] O. Kilic, M. S. Mirotznik, and J. P. Durbano, "Application of FPGA based FDTD simulators to Rotman lenses," in *Proc. 22nd ACES Conference*, 2006.

[26] O. Kilic, "FPGA accelerated phased array design using the ant colony optimization," *to appear in ACES Journal*.

[27] M. Dorigo, V. Maniezzo, and A. Colorni, "The ant system: Optimization by a colony of cooperating agents,"

IEEE Trans. Syst., Man, Cybern. B, vol. 26, no. 1, pp. 29–41, Feb. 1996.

[28] T. Hiroyasu, M. Miki, Y. Ono, and Y. Minami, "Ant colony for continuous functions," *The Science and Engineering Review of Doshisha University*, 2000.

[29] M. G. Moharam, D. A. Pommet, E. B. Grann, and T. K. Gaylord, "Stable implementation of the rigourous coupled-wave analysis for surface relief gratings: enhanced transmittance matrix approach," *Journal of the Optical Society of America A*, vol. 12, no. 5, pp. 1077–1086, 1995.

[30] P. Lalanne, "Improved formulation of the coupled-wave method for two-dimensional gratings," *Journal of the Optical Society of America A*, vol. 14, no. 7, pp. 1592–1598, 1997.

[31] M. G. Moharam and T. K. Gaylord, "Rigorous coupled-wave analysis of planar-grating diffraction," *Journal of the Optical Society of America*, vol. 71, no. 7, pp. 811–818, Jul. 1981.

[32] J. M. Jarem, "Rigorous coupled wave analysis of radially and azimuthally-inhomogeneous,elliptical, cylindrical systems," *Progress In Electromagnetics Research,* PIER 34, pp. 181–237, 2001.

[33] W. Lee and F. L. Degertekin, "Rigorous coupled-wave analysis of multilayered grating structures," *Journal of Lightwave Technology*, vol. 22, no. 10, pp. 2359–2363, Oct. 2004.

[34] J. W. Demmel, *Applied Numerical Linear Algebra*. Philadelphia, PA: Society for Industrical and Applied Mathematics (siam), 1997.

[35] J. G. F. Francis, "The QR transformation, I," *The Computer Journal*, vol. 4, no. 3, pp. 265–271, 1961.

[36] ——, "The QR transformation, II," *The Computer Journal*, vol. 4, no. 4, pp. 332–345, 1962.

[37] V. N. Kublanovskaya, "On some algorithms for the solution of the complete eigenvalue problem," *USSR Computational Mathematics and Mathematical Physics*, vol. 1, no. 3, pp. 637–657, 1963.

[38] A. S. Householder, "Unitary triangularization of a nonsymmetric matrix," *Journal of the ACM*, vol. 5, no. 4, pp. 339–342, Oct. 1958.

[39] G. H. Golub and C. F. V. Loan, *Matrix Computations (3rd edition)*. Baltimore, MD: The John Hopkins University Press, 1996.

Ozlem Kilic graduated from The George Washington University (1996) with a D.Sc. degree in Electrical Engineering. She is presently an Assistant Professor in the Department of Electrical Engineering and Computer Science at The Catholic University of America. Before joining CUA, she worked at the U.S. Army Research Laboratories, Adelphi, MD and COMSAT Laboratories, Clarksburg, MD. Her research areas include computational electromagnetics, hardware accelerated programming for scientific computing, antennas and propagation, and radiation and scattering problems from random media.

Miaoqing Huang is an Assistant Professor in the Department of Computer Science and Computer Engineering at University of Arkansas. His research interests include reconfigurable computing, high-performance computing architectures, cryptography, computer arithmetic, and cache design in Solid-State Drives. Huang received a B.S. degree in electronics and information systems from Fudan University, China in 1998, and a Ph.D. degree in computer engineering from The George Washington University in 2009, respectively. He is a member of IEEE.

Using GPUs for Accelerating Electromagnetic Simulations

Manuel Ujaldon

Department of Computer Architecture
University of Malaga, Malaga 29071, Spain
ujaldon@uma.es

Abstract- The computational power and memory bandwidth of graphics processing units (GPUs) have turned them into attractive platforms for general-purpose applications at significant speed gains versus their CPU counterparts [1]. In addition, an increasing number of today's state-of-the-art supercomputers include commodity GPUs to bring us unprecedented levels of performance in terms of raw GFLOPS and GFLOPS/cost. Inspired by the latest trends and developments in GPUs, we propose a new paradigm for implementing on GPUs some of the major aspects of electromagnetic simulations, a domain traditionally used as a benchmark to run codes in some of the most expensive and powerful supercomputers worldwide. After reviewing related achievements and ongoing projects, we provide a guideline to exploit SIMD parallelism and high memory bandwidth using the CUDA programming model and hardware architecture offered by Nvidia graphics cards at an affordable cost. As a result, performance gains of several orders of magnitude can be attained versus thread-level methods like pthreads used to run those simulations on emerging multicore architectures

Index Terms - Graphics processors, electro-magnetic simulations, CUDA, GPGPU.

I. INTRODUCTION

Graphics processors are usually characterized by parallelism, pipelining and bandwidth. After completing a steady transition from mainframes to workstations to PC cards, Graphics Processing Units (GPUs) emerge nowadays like a solid and compelling alternative to traditional computing, delivering extremely high floating point performance for those applications which can be arranged to fit and exploit the inherent parallelism and high memory bandwidth [2]. The newest versions of programmable graphics processing units (GPUs) have consistently demonstrated an outstanding performance in many applications beyond graphics, including data mining [3,4], computer vision [5], signal and image processing and segmentation [6,7,8], numerical methods [9], and assorted simulations [10,11,12].

This fact has attracted many other researchers and encouraged the use of GPUs in a broader range of applications, where developers will need to leverage this technology with new programming models which ease the developer's task of writing programs to run efficiently on GPUs. Nvidia and ATI/AMD, manufacturers of the popular GeForce and Radeon sagas of graphics cards, have released software components which provide simpler access to GPU computing power than that realized by treating the GPU as a traditional graphics processor. CUDA (Compute Unified Device Architecture) [13] is Nvidia's solution as a simple block-based API for programming; AMD's alternative is called Stream Computing and includes technologies such as the Brook+ compiler [14] and the Compute Abstraction Layer, both of which allow the developer to work in a high-level language which abstracts away GPUs' specifics. Those companies have also developed hardware products aimed specifically at the General Purpose GPU (GPGPU) computing market: The Tesla products [15] are from Nvidia, and Firestream [16] is AMD's product line.

Between Stream Computing and CUDA, we chose the latter to program the GPU for being more popular and providing more mechanisms to optimize general-purpose applications which do not entirely fit into the more traditional graphics processing paradigm. More recently, Apple's OpenCL framework [17] emerges as an attempt to

unify those two models with a superset of features, but since it is closer to CUDA and inherits most of its mechanisms, we are confident on an eventual portability for the methods described throughout this paper without loss of generality.

Novel scientific applications are good candidates to take the opportunity offered by CUDA and counterparts (see Fig. 1), and electromagnetic simulations is clearly one of them for three primary reasons:

1. This field has traditionally proven to be of great success for GPUs during its evolution towards high-performance general-purpose computing.
2. The increasing complexity of recent electromagnetic algorithms has made simulation part of the workflow in both academia and industry to be very computationally demanding.
3. Traditional architectures reveal themselves as inefficient solutions for this class of applications.

Electromagnetic simulations are memory intensive applications containing assorted access patterns where memory optimizations play a primary role. Fortunately, CUDA provides a set of powerful low-level mechanisms for controlling the use of memory and the behavior of its hierarchy. This affects performance severely at the expense of a considerable programming effort, which we describe throughout this paper.

The rest of the paper is organized as follows. Section II reviews the most recent results obtained by GPUs on electromagnetic simulations. Section III focuses on the specifics of the GPU programming with CUDA, and Section IV describes optimization strategies particularly oriented to simulation codes. Section V concludes.

II. THE GPU ON ELECTROMAGNETIC SIMULATIONS

A. Related Work

Over the past few decades, the increase of overall computing power coupled with the maturation of many electromagnetic algorithms has produced a blooming on the simulation side. Many explorations focused on 2D first, were later extended to 3D, and even were modeled as so-called 2.5D problems.

In response to that evolution, a number of approaches to hardware acceleration of electromagnetic simulations have been investigated in the past five years. Those approaches can be classified into two main categories:

1. Stand-alone computing devices like ASICS, which represent the highest achievable acceleration but quickly becomes too expensive due to the massive hardware required.
2. Co-processors with their own memory and connected to a host PC via an input/output bus or socket interface. Within this category, we may find Field Programmable Gate Arrays (FPGAs) [18] and Graphics Processing Units (GPUs) [19].

GPUs stand out in a unique way from all these innovative solutions because they are produced as commodity processors and their floating point performance has significantly outpaced that of any other processor. In addition, GPUs have become easier to program, which allows developers to effectively exploit their computational power.

Modern GPUs have been at the leading edge of increasing chip-level parallelism over the past five years. Scaling from 8 to 240 processors in the most popular saga of Nvidia GPUs, they have completed a steady transition from multi-core to many-core processors. The high degree of parallelism achieved, combined with their wide availability and affordable budget, has ultimately confirmed GPUs as a popular platform among universities and students to run computationally expensive simulations [1].

More recently, several companies that supply leading edge electromagnetic simulation software have joined this movement to ease code transition to the GPU for all kind of users belonging to this area regardless of their programming skills. Some illustrative examples are Acceleware and CST, which have announced a new GPU-based solution for accelerating lengthy electromagnetic design simulations, reporting performance gains of up to 40% compared to previous products [20,21]. This software uses CUDA, a programming interface particularly designed to solve complex computational general-purpose problems, which we describe later in Section III. Large corporations and research institutions have also been able to tap into clusters of GPUs for large scale simulations [22], enabling a step forward in performance while maintaining a limited budget. This way, the GPU technology aspires to have a tremendous impact on engineering electromagnetic education, as universities and research centers worldwide will be able to simulate realistic problems with

affordable GPU-based hardware platforms, which will also be available to students on their own personal computers.

Successful implementations of electromagnetic algorithms on GPUs can be seen as the key for the integration of simulators into design and optimization tools [23]. The GPU power may be combined here with the development of behavioral models and multi-grid, graded mesh and multi-resolution techniques for boosting the performance of electromagnetic simulations.

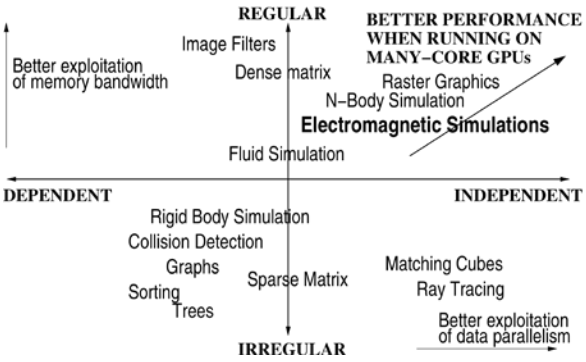


Fig. 1. An overview of general purpose applications evaluated by GPU performance according to two major features: Amount of parallelism extracted (on X axis) and memory bandwidth exploitation (on Y axis).

B. Characterization

The GPU has been extensively used in scientific computing over the past five years, but the degree of success has been different depending on algorithm features and how they meet GPU hardware idiosyncrasies. Nvidia [13,24] has reported a list of illustrative examples. Just to mention a few involving simulations, we have: molecular dynamics (36x), fluid dynamics (17x), multi-fluid (50x), astrophysics (100x), multi-body mechanical (13x), financial (149x), oil and gas (18x), DNA and liquids (18x), and interactive visualization of volumes (146x).

In general, expectations for a particular algorithm to reach certain levels of speedup factor when running on GPUs depend on a number of features which conform a list of requirements to be fulfilled. From less to more important, we have:

1. Small local data requirements (memory and registers).
2. Stream computing (non-recursive algorithms).
3. Arithmetic intensity (high data reuse).
4. Bandwidth (fast data movement).
5. Data parallelism (data independency).

The two key factors are analyzed in Fig. 1, where some of the most popular applications are placed in conjunction with electromagnetic simulations to quantify the memory bandwidth and data parallelism each algorithm can benefit from. This gives us an estimation about how successfully each code can run on GPU platforms.

C. Upsides

Simulations usually consists of a mixture of fundamentally serial control logic and inherently parallel computation. Furthermore, those computations are often data-parallel in nature, which matches the programming model that CUDA adopts (see Section III-B), basically a sequential control thread capable of launching a series of parallel kernels. This makes it relatively easy to parallelize an application's individual components as kernels, rather than requiring a wholesale rewriting of the entire application.

In our case of a typical electromagnetic simulation, the same executable is invoked multiple times on each parallel processor by a job-queuing algorithm and the results are then reassembled. This constitutes an embarrassingly parallel computing model, as it does not require much internode communication or global data sharing. Electromagnetic computations are in fact very close to graphics processing in this respect: Million of operations can be performed in parallel exhibiting a speed which can reach up to two orders of magnitude when compared to the computational power shown on typical quad-core CPUs.

On the other hand, simulations often deal with a large amount of data, which are responsible for the realism and accuracy of the simulated physics. GPUs reach data bandwidth with video memory around ten times higher than CPUs with main memory, and because of the way data is transferred, regular access patterns in the code behave better when running on GPUs.

A third issue is also worth mentioning: Arithmetic intensity. Electromagnetic simulations usually require the computation of complex mathematical formulas, which are efficiently mapped to the GPU platform due to the presence of

those units devoted to a typical graphics rendering. Moreover, newer generations of GPUs like GeForce Series 8 include internally a powerful co-processor devoted to the computational physics required in many realistic animation and effects. Such co-processor, called PhysX [25], was originally invented by Ageia, whose design was inspired in those we found in GPUs for building arithmetic units and massively parallelism.

Finally, we leave on the CPU those parts of our simulation that do not have high arithmetic intensity or do not expose substantial amounts of data or thread-level parallelism. This way, that tough part of our application remains unchanged and can benefit from overlapping computations on a bi-processor CPU-GPU platform.

D. Downsides

For the GPU to succeed as the favourite platform to run electromagnetic simulations in the future, we still envision two main challenges in the horizon: Accuracy and memory capacity.

Accuracy. The lack of 32-bit floating-point precision was a major drawback in many application areas during the first half of this decade. Starting in 2008 with the GT 200 series from Nvidia, the situation has reversed and all major GPU vendors now offer 64-bit massively parallel hardware which will further enhance modelling and simulation capabilities. For example, the Tesla T10P GPU from Nvidia provides full IEEE rounding, fused multiply-add, and denormalized number support for double precision.

The problem arises when you look at execution times, since in most cases performance drops from five to ten times when you migrate your algorithm from single to double precision. This is mainly due to the reduced degree of parallelism we can exploit in the architecture, as usually the ratio of single to double precision floating-point arithmetic units available in a typical GPU is four to one or even eight to one. In the past, the primary argument for not to overcome this lack was that classical rendering did not require such enhancement. With the recent movements towards general-purpose GPU-like architectures, double precision floating-point will be offered at a much lighter performance penalty as more applications demand it.

E. Memory size

Some of the large scale simulations are not necessary complicated in nature, but they require a large amount of memory space. For example, modelling of the near electromagnetic fields around antennas fall into this category, and more in general, field and signal analysis for high-speed electronic circuits and systems has become increasingly difficult due to the complexity of new electronic devices. GPU memory has progressed at a higher speed rate than the CPU counterpart over the last decade, and GDDR5, the video memory currently available, keeps consistently two generations ahead versus CPU DDR3 memory placed on the mainboard. But when it comes to capacity, the reduced form factor (size) of the graphics card in conjunction with its wider bus width versus the GPU, introduce serious routing problems which prevent video memory capacity from growing at the same rate. We believe that the solution to this problem lies more in the software layer, particularly in programmer's hands, who has to be able to partition data efficiently and ultimately perform computations through a blocking strategy to overcome memory constraints.

III. CUDA

The Compute Unified Device Architecture (CUDA) [13] is a programming interface and set of supported hardware to enable general-purpose computation on Nvidia GPUs.

The CUDA programming interface is ANSI C extended by several keywords and constructs which derive into a set of C language library functions as a specific compiler generates the executable code for the GPU in conjunction with the counterpart version running on the CPU acting as a host.

Since CUDA is particularly designed for generic computing, it can leverage special hardware features not visible to more traditional graphics-based GPU programming, such as small cache memories, explicit massive parallelism and lightweight context switch between threads.

A. Hardware Platforms

All the latest Nvidia developments on graphics hardware are compliant with CUDA: For low-end users and gamers, we have the GeForce series starting from its 8th generation; for high-end users

and professionals, the Quadro FX 5600/4600 series; for general-purpose computing, the Tesla boards. Focusing on Tesla, the C870 is an homogeneous CMP endowed with 128 cores and 1.5 GB of video memory to deliver a theoretical peak performance of 518 GFLOPS (single precision), a peak on-board memory bandwidth of 76.8 GB/s and a peak main memory bandwidth of 4 GB/s under its PCI-express x16 interface.

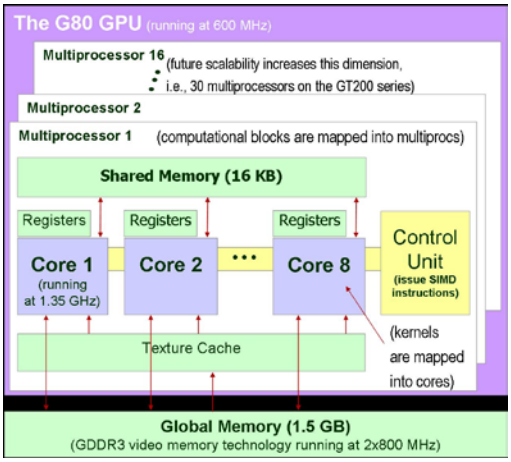


Fig. 2. The CUDA hardware interface.

B. Execution Modes

The G80 parallel architecture is a SIMD (Single Instruction Multiple Data) processor endowed with 128 cores. Cores are organized into 16 multiprocessors, each having a large set of 8192 registers, a 16 KB shared memory very close to registers in speed (both 32 bits wide), and constants and texture caches of a few kilobytes. Each multiprocessor can run a variable number of threads, and the local resources are divided among them. In any given cycle, each core in a multiprocessor executes the same instruction on different data based on its *threadID*, and communication between multiprocessors is performed through global memory (see Fig. 3).

Future architectures from Nvidia will support the same CUDA executables, but they will be run faster in order to include more multiprocessors per die, or more cores, registers or shared memory per multiprocessor. For example, the GT200 architecture contains 30 multiprocessors for a total of 240 cores, while registers and shared memory per multiprocessor remain the same.

The CUDA programming model guides the programmer to expose fine-grained parallelism as required by massively multi-threaded GPUs, while at the same time providing scalability across the broad spectrum of physical parallelism available in the range of GPU devices.

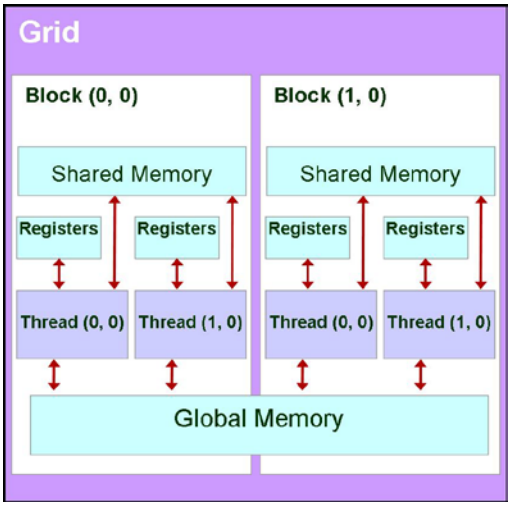


Fig. 3. The CUDA programming model.

C. Memory Spaces

The CPU host and the GPU device maintain their own DRAM and address space, referred to as host memory and device memory (on-board memory). The latter can be of three different types. From inner to outer, we have constant memory, texture memory and global memory. They all can be read from or written to by the host and are persistent through the life of the application. Texture memory is the more versatile one, offering different addressing modes as well as data filtering for some specific data formats. Global memory is the actual on-board video memory, usually exceeding 1 GB of capacity and embracing GDDR3/GDDR5 technology. Constant memory has regular size of 64 KB and latency time close to a register set. Texture memory is cached to a few kilobytes. Global and constant memories are not cached at all.

D. Programming Elements

There are some important elements involved in the conception of a CUDA program that are key for understanding the programming model as well as the optimizations we have carried out during the implementation phase. We describe them below and Fig. 3 summarizes their relations.

A program is decomposed into **blocks** running in parallel. Assembled by the developer, a block is

a group of threads that is mapped to a single multiprocessor, where they can share 16 KB of memory (see Fig. 2). All the threads in blocks concurrently assigned to a single multi-processor divide the multiprocessor's resources equally amongst themselves. The data is also divided amongst all of the threads in SIMD fashion explicitly managed by the developer.

A **warp** is a collection of 32 threads that can physically run concurrently on all of the multiprocessors. The size of the warp is less than the total number of cores due to memory access limitations. The developer has the freedom to determine the number of threads to be executed, but if there are more threads than the warp size, they are time-shared on the actual hardware resources. This can be advantageous, since time-sharing the ALU resources amongst multiple threads can overlap the memory latencies when fetching ALU operands.

A **kernel** is a code function compiled to the instruction set of the device, downloaded on it and executed by all of its threads. Threads run on different processors of the multiprocessors sharing the same executable and global address space, though they may not follow the same path of execution, since conditional execution of different operations on each multiprocessor can be achieved based on a unique *threadID*. Threads also work independently on different data according to the SIMD model described in Section III-B. A kernel is organized into a grid as a set of **thread blocks**.

A **grid** is a collection of all blocks in a single execution, explicitly defined by the application developer, which is assigned to a multiprocessor. The parameters invoking a kernel function call define the sizes and dimensions of the thread blocks in the grid thus generated, and the way hardware groups threads in warps affects performance, so it must be accounted for.

A **thread block** is a batch of threads executed on a single multiprocessor. They can cooperate together by efficiently sharing data through its shared memory, and synchronize their execution to coordinate memory accesses using the *__syncthreads()* primitive. Synchronization across thread blocks can only be safely accomplished by terminating a kernel. Each thread block has its own *threadID,* which is the number of the thread within a 1D, 2D or 3D array of arbitrary size. The use of multidimensional identifiers helps to simplify memory addressing when processing multidimensional data. Threads placed in different blocks from the same grid cannot communicate, and threads belonging to the same block must all share the 8K registers and 16 KB of shared memory on a given multiprocessor. This tradeoff between parallelism and thread resources must be wisely solved by the programmer to maximize performance on a certain architecture given its limitations.

At the highest level, a program is decomposed into kernels mapped to the hardware by a grid composed of blocks of threads scheduled in warps. No inter-block communication or specific schedule-ordering mechanism for blocks or threads is provided, which guarantees each thread block to run on any multiprocessor, even from different devices, at any time.

The number of blocks in a thread block is limited to 512. Therefore, blocks of equal dimension and size that execute the same kernel can be batched together into a grid of thread blocks. This comes at the expense of reduced thread cooperation, because threads in different thread blocks from the same grid cannot communicate and synchronize with each other. Again, each block is identified by its *blockID*, which is the number of the block within a 1D or 2D array of arbitrary size for the sake of a simpler addressing to memory.

Kernel threads are extremely lightweight, i.e. creation overhead and context switching between threads and/or kernels is negligible.

IV. OPTIMIZATIONS

Once that major hardware and software limitations have been introduced, it becomes clear that managing those limits is critical when optimizing applications. Programmers still have a great degree of freedom, though side effects may occur when deploying strategies to avoid one limit, causing other limits to be hit.

We consider two basic pillars when optimizing an application to run on CUDA GPUs: First, organize threads in blocks to maximize parallelism, enhance hardware occupancy and avoid memory banks conflicts. Second, access to shared memory wisely to maximize arithmetic intensity and reduce global memory usage. We address each of these issues separately now.

A. Threads Deployment

Each multiprocessor contains 8192 registers which will be split evenly among all the threads of the blocks assigned to that multiprocessor. Hence, the number of registers needed in the computation will affect the number of threads which can be executed simultaneously, and the management of registers becomes important as a limiting factor for the amount of parallelism we can exploit.

The CUDA documentation suggests a block to contain between 128 and 256 threads to maximize execution efficiency. A tool developed by Nvidia, the CUDA Occupancy Calculator, may also be used as guidance to attain this goal. For example, when a kernel instance consumes 16 registers, only 512 threads can be assigned to a single multiprocessor. This can be achieved by using one block with 512 threads, two blocks of 256 threads, and so on.

We followed an iterative process to achieve the lowest execution time: First, the initial implementation was compiled using the CUDA compiler and a special *-cubin* flag that outputs the hardware resources (memory and registers) consumed by the kernel. Using these values in conjunction with the CUDA Occupancy Calculator, we were able to analytically determine the number of threads and blocks that were needed to use a multiprocessor with maximum efficiency.

B. Memory Usage

Even though video memory delivers a magnificent bandwidth, it is still a frequent candidate to hold the bottleneck when running the application because of its poor latency (around 400 times slower compared to shared memory) and the high floating-point computation performance of the GPU. Attention must be paid to how the threads access the 16 banks of shared memory, since only when the data resides in different banks can all of the available ALU bandwidth truly be used.

Each bank only supports one memory access at a time; simultaneous memory bank accesses are serialized, stalling the rest of the multiprocessor's running threads until their operands arrive. The use of shared memory is explicit within a thread, which allows the developer to solve bank conflicts wisely. Although such optimization may represent a daunting effort, sometimes can be very rewarding: Execution times may decrease by as much as 10x for vector operations and latency hiding may increase by up to 2.5x.

Another critical issue related to memory performance is data coalescing. A coalesced access involves a contiguous region of global memory where the starting address must be a multiple of region size and the k^{th} thread in a half-warp must access the k^{th} element in a block being read. This way, the hardware can serve completely two coalesced accesses per clock cycle, maximizing memory bandwidth, bus usage and throughput. It is programmer's responsibility to organize memory accesses in such a way, though CUDA has relaxed the conditions to be fulfilled for coalescing in their latest versions (from Compute Capabilities 1.2 on).

V. CONCLUDING REMARKS

We have presented the CUDA programming model and hardware interface as a very compelling alternative for high-performance computing when applied to electromagnetic simulations. Particular features of these simulations are identified and a number of techniques and optimizations are introduced to wrench the full performance out of the GPU resources for a large class of important scientific applications, even unveiling opportunities for further innovation.

GPUs are highly scalable and become more valuable for general-purpose computing. We envision electromagnetic simulations as one of the most exciting fields able to benefit from GPUs in the future of this emerging architecture. Additionally, new tools like CUDA and OpenCL may assist non-computer scientists with a friendlier interface for adapting these applications to GPUs. This computational power may then be multiplied on a cluster of GPUs to enhance parallelism and provide even faster responses to electromagnetic simulations at a very low cost.

Alternatively, we may think of a CPU-GPU hybrid system where an application can be decomposed into two parts to take advantage of the benefits of this bi-processor platform, and the programming models must evolve to include programming heterogeneous manycore systems including both CPUs and GPUs.

GPUs will continue to adapt to the usage patterns of both graphics and general-purpose programmers, with a focus on additional processor

cores, number of threads and memory bandwidth available for electromagnetic simulations. In addition, the programming models must evolve to include programming heterogeneous manycore systems including both CPUs and GPUs.

REFERENCES

[1] GPGPU, "General-purpose computation using graphics hardware", http://www.gpgpu.org, 2009.

[2] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Kruger, A. E. Lefohn, and T. J. Purcell, "A survey of general-purpose computation on graphics hardware," *Journal of Computer Graphics Forum*, vol. 26, pp. 21–51, 2007.

[3] S. Guha, S. Krisnan, and S. Venkatasubramanian, "Data visualization and mining using the GPU," *Tutorial at 11th ACM Intl. Conference on Knowledge Discovery and Data Mining,* 2005.

[4] N. K. Govindaraju, B. LLoyd, W. Wang, M. Lin, and D. Manocha, "Fast Computation of Database Operations Using Graphics Processors," *ACM SIGMOD International Conference on Management of Data,* pp. 215–226, 2004.

[5] R. Yang and M. Pollefeys, "A Versatile Stereo Implementation on Commodity Graphics Hardware", *Real Time Imaging,* vol. 11, no. 1, pp. 7–18, February 2005.

[6] T. Sumanaweera and D. Liu, "Medical Image Reconstruction with the FFT," *GPU Gems,* March 2004.

[7] I. Viola, A. Kanitsar, and M. E. Groller, "Hardware Based Nonlinear Filtering and Segmentation Using High-Level Shading Languages," *IEEE Visualization,* pp. 309–316, October 2003.

[8] M. Hadwiger, C. Langer, H. Scharsach, and K. Buhler, "State of the art report on GPU-based segmentation," *VRVis Research Center,* Tech. Rep. TR-VRVIS-2004-17, 2004.

[9] W. Wu and P. Heng, "A hybrid condensed finite element model with GPU acceleration for interactive 3D soft tissue cutting: Research articles", *Computer Animation and Virtual Worlds,* vol. 15, no. 3-4, pp. 219–227, 2004.

[10] M. Harris, "Fast Fluid Dynamics Simulation on the GPU," *GPU Gems,* 2004.

[11] P. Sander, N. Tartachuk, and J. L. Mitchell, "Explicit Early-Z Culling for Efficient Fluid Flow Simulation and Rendering", *ATI Research Journal Technical Report,* August 2004.

[12] Y. Zhao, Y. Han, Z. Fan, F. Qiu, Y. Kuo, Kaufman, and K. A., Mueller, "Visual simulation of heat shimmering and mirage," *IEEE Trans. on Visualization and Computer Graphics,* vol. 13, no. 1, pp. 179–189, 2007.

[13] CUDA, "Home page maintained by Nvidia" http://developer.nvidia.com/object/cuda.html.

[14] Brook+, "Web Page maintained by AMD", http://ati.amd.com/technology/streamcomputing/AMD-Brookplus.pdf, 2009.

[15] "Nvidia Tesla GPU computing solutions for HPC" http://www.nvidia.com/object/tesla_ computing_ solutions.html, 2009.

[16] Firestream, "AMD Stream Computing", http://ati.amd.com/technology/streamcomputing.

[17] T. K. Group, "The OpenCL Core API Specification, Headers and Documentation," http://www.khronos.org/registry/cl, 2009.

[18] E. Kelmelis, J. Durbano, P. Curt, and J. Zhang, "Field-programmable gate array accelerates FDTD calculations," *Laser Focus World,* September 2006.

[19] S.E. Krakiwsky, L.E. Turner, M.M. Okoniewski", Acceleration of finite-difference time-domain (FDTD) using graphics processing units (GPU)," *IEEE MTT- S Int. Conference,* June 2004.

[20] http://www.acceleware.com/em

[21] http://www.cst.com/

[22] T. Hartley, U. Catalyurek, A. Ruiz, M. Ujaldon, F. Igual, and R. Mayo", "Biomedical Image Analysis on a Cooperative Cluster of GPUs and Multicores," *22nd ACM Intl. Conf. on Supercomputing,* 2008.

[23] P. So, "EM-based simulation tools for signal and systems analysis", *International Symposium on Signals, Systems and Electronics,* August 2007.

[24] M. Harris, "Manycore parallel computing with CUDA", *Keynote Session at the 22nd ACM Intl. Conference on Supercomputing,* June 2008.

[25] Ageia, "The PhysX co-processor", http://www.nvidia.com/object/nvidia_physx.html.
[26] T. R. Halffill, "Parallel Processing with CUDA", *MicroProcessor Report Online*, January 2008.
[27] Nvidia Compute Unified Device Architecture (CUDA) Programming Guide v. 1.1, Nov. 2007.
[28] Nvidia CUDA CUBLAS Library v. 1.1, Sep. 2007.
[29] Nvidia CUDA CUFFT Library v. 1.1, Oct. 2007.

Manuel Ujaldon received his B.S. degree in Computer Science from the Univ. of Granada (Spain, 1991) and his M.S. and Ph.D. degrees in Computer Science from the Univ. of Malaga (Spain, 1993 and 1996). During 1994 and 1995 he was a Research Assistant in the Computer Architecture Dept. at the University of Malaga, where he became Assistant Professor in 1996 and Associate Professor in 1999.

Dr. Ujaldon was a predoctoral and postdoctoral researcher at the Computer Science Dept. of the University of Maryland (USA, 1994, 1996/97) and Biomedical Informatics Department of the Ohio State University (USA, 2003-08).

He has published 8 books on computer architecture and more than 50 papers in international peer-reviewed journals and conferences. His research interest includes streaming architectures as well as compiler and software development for running general-purpose scientific applications on GPUs.

Compute Unified Device Architecture (CUDA) Based Finite-Difference Time-Domain (FDTD) Implementation

Veysel Demir[1] and Atef Z. Elsherbeni[2]

[1]Department of Electrical Engineering
Northern Illinois University, DeKalb, IL 60115, USA
demir@ceet.niu.edu

[2]Department of Electrical Engineering
The University of Mississippi, University, MS 38677, USA
atef@olemiss.edu

Abstract— Recent developments in the design of graphics processing units (GPUs) have made it possible to use these devices as alternatives to central processor units (CPUs) and perform high performance scientific computing on them. Though several implementations of finite-difference time-domain (FDTD) method have been reported, the unavailability of high level languages to program graphics cards had been a major obstacle for scientists and engineers who would want to develop codes for graphics cards. Relatively recently, compute unified device architecture (CUDA) development environment has been introduced by NVIDIA and made GPU computing much easier.

This paper presents an implementation of FDTD method based on CUDA. Two thread-to-cell mapping algorithms are presented. The details of the implementation are provided and strategies to improve the performance of the FDTD simulations are discussed.

Index Terms—FDTD methods, parallel architectures, graphics processing unit (GPU) programming, Compute Unified Device Architecture (CUDA), hardware accelerated computing.

I. INTRODUCTION

Recent developments in the design of graphics processing units (GPUs) have been occurring at a much greater pace than with central processor units (CPUs) and very powerful processing units have been designed solely for the processing of computer graphics. For instance, the current generation of GPU based NVIDIA® Tesla™ C1060 Computing Processors are running at approximately 1.3 GHz with a 512 bit data and memory bandwidth of 102 GB/sec. While GPU clock speed seems slow compared to modern 3.8 GHz Pentium CPU's or 3.0 GHz Core Duo's, parallelism provided by the graphics cards enables better efficiency in computations. Due to this potential in faster computations, the GPUs have received the attention of the scientific computing community. Initially these cards were designed for computer graphics and floating precision arithmetic has been sufficient for such applications. Due to the demand of higher precision arithmetic from the scientific community, the vendors have started to develop graphics cards that support double precision arithmetic as well, introducing a new generation of graphical computation cards.

The computational electromagnetics community as well has started to utilize the computational power of graphics cards, and in particular, several implementations of finite-difference time-domain (FDTD) [1]-[3] method have been reported [4]-[24]. Initially the GPUs were not designed for general purpose programming and high level programming languages were not conveniently available; programmers were required to learn the intricacies of specialized low-level hardware languages. For instance, the FDTD implementations in [4], [5] and [11] are based on OpenGL. As a result of the need for high level languages a new subset language for C titled "Brook" has been introduced for general

programming environments [25]. This subset negates the need for detailed low-level programming knowledge by introducing a few, relatively simple, commands in the C language. Brook is used as the programming language in [7]-[10], [14]-[15] and [24]. Moreover, use of High Level Shader Language (HLSL) is reported in [16].

Relatively recently, the introduction of the Compute Unified Device Architecture (CUDA) [26] development environment from NVIDIA made GPU computing much easier. CUDA is a general purpose parallel computing architecture. To program the CUDA architecture, developers can use C, which can then be run at great performance on a CUDA enabled processor. The CUDA architecture and its associated software provide a small set of extensions to standard programming languages, like C, that enable a straightforward implementation of parallel algorithms. With CUDA and C for CUDA, programmers can focus on the task of parallelization of the algorithms rather than spending time on their implementation. The CPU and GPU are treated as separate devices that have their own memory spaces. This configuration also allows simultaneous computation on both the CPU and GPU without contention for memory resources. CUDA-enabled GPUs have hundreds of cores that can collectively run thousands of computing threads [27].

CUDA has been reported as the programming environment for implementation of FDTD in [17]-[18] and [20]-[22]. In [21] the use of CUDA for two-dimensional FDTD is presented, and its use for three-dimensional FDTD implementations is proposed. The importance of coalesced memory access and efficient use of shared memory is addressed without sufficient details. Another two-dimensional FDTD implementation using CUDA has been reported in [22] and use of convolution perfectly matched layer (CPML) [28] boundaries is discussed, however no implementation details are provided. Some methods to improve the efficiency of FDTD using CUDA are presented in [20], which can be used as guidelines while programming FDTD using CUDA. The discussions are based on FDTD updating equations in its simplest form: updating equations consider only dielectric objects in the computation domain, the cell sizes are equal in x, y, and z directions, thus the updating equations include a single updating coefficient. The efficient use of shared memory is discussed; however the presented methods limit the number of threads per thread block to a fixed size. The coalesced memory access, which is a necessary condition for efficiency on CUDA, is inherently satisfied with the given examples; however its importance has never been mentioned.

In this current contribution a more comprehensive discussion of CUDA implementation of FDTD is provided. The FDTD updating equations assume more general material media and different cell sizes. Strategies to improve the efficiency are discussed, and their application to unified FDTD updating equations, as presented in [3], is presented.

Section II summarizes an overview of concepts in CUDA. Section III presents the FDTD equations that are considered for CUDA implementation, while Section IV introduces two algorithms of implementation. Section V reports the performances achieved in computation speed by these implementations.

II. COMPUTE UNIFIED DEVICE ARCHITECTURE

In this section, a brief description of some concepts in CUDA is summarized from [29] in order to prepare the reader for the discussions that follow. Then, general guidelines to improve the efficiency of CUDA programs, as they apply to FDTD method, are summarized based on [29] and [30]. Application of these guidelines to improve the efficiency of an FDTD implementation is discussed in the subsequent sections.

A. CUDA Concepts

A programmable graphics processor unit is essentially a highly parallel, multithreaded, many core processor. The GPU is especially well-suited to address problems that can be expressed as data-parallel computations – the same program is executed on many data elements in parallel. FDTD is such an algorithm in which the same computation is performed on all field components in the cells of a computation domain.

CUDA is a general purpose parallel computing architecture with a new parallel programming model and instruction set architecture. C for CUDA extends C by allowing the programmer to define C functions, called *kernels*, that, when called, are executed *N* times in parallel by *N* different CUDA *threads*, as opposed to only once like regular C functions. Each of the threads that execute a kernel is given a unique *thread ID* that is accessible within the kernel through the built-in `threadIdx` variable. For convenience, `threadIdx` is a 3-component vector, so that threads can be identified using a one-dimensional, two-dimensional, or three-dimensional *thread index*, forming a one-dimensional, two-dimensional, or three-dimensional *thread block*. A kernel function can be executed by multiple equally-shaped thread blocks, so that the total number of threads is equal to the number of threads per block times the number of blocks. These multiple blocks are organized into a one-dimensional or two-dimensional *grid* of thread blocks. Each block within the grid can be identified by a one-dimensional or two-dimensional index accessible within the kernel through the built-in `blockIdx` variable. The dimension of the thread block is accessible within the kernel through the built-in `blockDim` variable.

CUDA threads may access data from multiple memory spaces during their execution. Each thread has a private *local memory* and a *shared memory* visible to all threads of the block and with the same lifetime as the block. Finally, all threads have access to the same *global memory*. Global memory is the main memory space on the device to store the application data. However, data access to global memory is very small and that inefficiency becomes the main bottleneck in the execution of a kernel. On the other hand the shared memory is much faster to access but the size of the shared memory is very limited. However, though very limited in size, the shared memory can provide the means for data reuse and improve the efficiency of a kernel. *Constant* and *texture memory* spaces are two additional read-only memory spaces, limited in size, accessible by all threads during the lifetime of the application. The kernels execute on a GPU that is referred to as *device* and the rest of the C program executes on a CPU that is referred to as *host*.

B. Performance Optimization Strategies

Recommendations for optimization and the list of best practices for programming with CUDA are explained in [30]. While not all of these recommendations are applicable to the case of FDTD; the following list of recommendations is used to optimize our FDTD implementation:

R1) structure the algorithm in a way that exposes as much data parallelism as possible. Once the parallelism of the algorithm has been exposed, it needs to be mapped to the hardware as efficiently as possible.

R2) ensure global memory accesses are coalesced whenever possible.

R3) minimize the use of global memory. Prefer shared memory access where possible.

R4) use shared memory to avoid redundant transfers from global memory.

R5) hide latency arising from register dependencies, maintain at least 25 percent occupancy on devices with CUDA compute capability 1.1 and lower, and 18.75 percent occupancy on later devices.

R6) use a multiple of 32 threads for the number of threads per block as this provides optimal computing efficiency and facilitates coalescing.

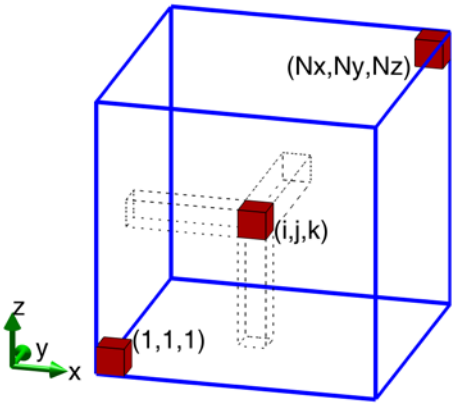


Fig. 1. An FDTD problem space composed of cells [3].

III. THE FDTD FORMULATION

The FDTD formulation considered for CUDA implementation is based on updating equations for general anisotropic material properties including arbitrary permittivity, permeability and electric and magnetic conductivity parameter values [3]. The FDTD problem domain is a rectangular domain composed of cells, referred to as Yee cells [1], as illustrated in Fig. 1. The problem space size is $Nx \times Ny \times Nz$, where Nx, Ny, and Nz are number of cells in x, y, and z directions, respectively. Field components are defined at discrete positions on a Yee cell as shown in Fig. 2. The formulation in consideration assumes different cell sizes in x, y, and z directions in a rectangular grid. Thus, for instance, the equation that updates x-component of the magnetic field is given in [3] as

$$H_x^{n+\frac{1}{2}}(i,j,k) = C_{hxh}(i,j,k)\,H_x^{n-\frac{1}{2}}(i,j,k) + C_{hxey}(i,j,k)\left(E_y^n(i,j,k+1) - E_y^n(i,j,k)\right) + C_{hxez}(i,j,k)\left(E_z^n(i,j+1,k) - E_z^n(i,j,k)\right), \quad (1)$$

where $H_x^{n+\frac{1}{2}}(i,j,k)$ is the x component of magnetic field in a Yee cell, shown in Fig. 2, indexed with (i,j,k), and E_y^n and E_z^n are the electric field components. The superscripts indicate the time instants at which the fields are evaluated: i.e. superscript n indicates the field at time $n\Delta t$, where Δt is the duration of time step. C_{hxh}, C_{hxey}, C_{hxez} are the coefficients used to update H_x. Similarly, there are two other updating equations that update H_y and H_z, and moreover, there are three other updating equations that update electric field components E_x, E_y, and E_z. A reference example for the update of magnetic field components when using the FORTRAN programming language is shown in Listing 1. As shown, all field and coefficient parameters in this listing are three-dimensional arrays.

```
subroutine update_magnetic_fields
! nx, ny, nz: number of cells in x, y, z
directions

Hx =  Chxh   *  Hx &
    + Chxey * (Ey(:,:,2:nz+1) - Ey(:,:,1:nz)) &
    + Chxez * (Ez(:,2:ny+1,:) - Ez(:,1:ny,:));

Hy =  Chyh   *  Hy &
    + Chyez * (Ez(2:nx+1,:,:) - Ez(1:nx,:,:)) &
    + Chyex * (Ex(:,:,2:nz+1) - Ex(:,:,1:nz));

Hz =  Chzh   *  Hz &
    + Chzex * (Ex(:,2:ny+1,:) - Ex(:,1:ny,:)) &
    + Chzey * (Ey(2:nx+1,:,:) - Ey(1:nx,:,:));

end subroutine update_magnetic_fields
```

Listing 1. Fortran code to update magnetic field components.

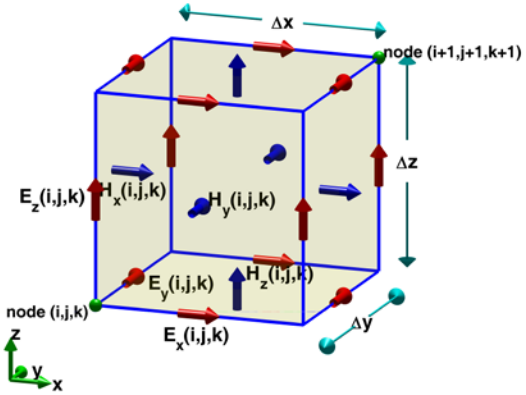


Fig. 2. Yee cell: the basic building block of an FDTD problem space [3].

IV. FDTD USING CUDA

In our implementation, the allocation of all field components and the initialization of coefficient arrays for the FDTD problem space are coded in FORTAN and executed on the CPU (host). Then these arrays are transferred to the global memory of GPU and they are ready to use by the kernels coded in CUDA and run on GPU (device). It should be noted that while the arrays in FORTRAN are three-dimensional, these same arrays are stored in device (GPU) global memory as one-dimensional arrays and elements of these arrays are accessed in kernel functions in a linear fashion. Thus, as will be shown later, a three-dimensional to one-dimensional index mapping is employed.

This section describes our procedure for developing CUDA kernels.

A. Achieving Parallelism

At every time iteration of the FDTD loop new values of three magnetic field components are

recalculated at every cell simultaneously using the past values of electric field components. Similarly, electric field components can be updated simultaneously in a separate function. Since the calculations for each cell can be performed independent from the other cells, a CUDA algorithm can be developed by assigning each cell calculation to a separate thread, and the highest level of parallelism can be achieved to satisfy the recommendation R1 that is discussed in Section II.

In CUDA, a number of threads form a thread block, and a number of thread blocks form a grid. The maximum number of threads in a block can be 512, where these threads can be arranged to form a one-dimensional, two-dimensional or three-dimensional block. Thus a subsection of three-dimensional problem space can be naturally mapped to a three-dimensional thread block. However, a grid (of thread blocks) can be composed of blocks arranged in a one-dimensional fashion or a two-dimensional fashion. Hence, the entire three-dimensional FDTD domain cannot be naturally mapped to a one-dimensional or two-dimensional grid. Therefore, an alternative mapping between threads and FDTD domain shall be considered.

In this contribution, two different approaches between cells and threads are presented and their performance comparisons are provided.

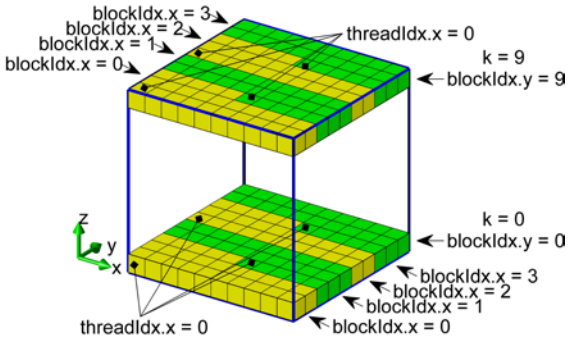


Fig. 3. Mapping of threads to cells of an FDTD domain using the xyz-mapping.

In the first mapping, a thread block is constructed as a one-dimensional array, as shown on the first two lines in Listing 2, which is a piece of code that defines the grid and block sizes. The threads in this array are mapped to cells in an *x*-*y* plane cut of the FDTD domain. The grid of the thread blocks is constructed as two-dimensional as shown on the third and fourth lines in Listing 2. Then, the *x* dimension of the grid is mapped to *x*-*y* plane, and *y* dimension of the grid is mapped to *z*-dimension of the FDTD domain. Figure 3 illustrates the mapping of threads to an FDTD domain. This mapping approach ensures one-to-one mapping between threads and cells, thus the highest level of parallelization is achieved. This mapping will be referred to as xyz-mapping in the following sections.

```
block_dim_x = number_of_threads;
block_dim_y = 1;
n_blocks_y = nz;
n_blocks_x = (nx*ny)/number_of_threads
  + ((nx*ny)%number_of_threads == 0 ? 0 : 1);
```

Listing 2. CUDA code to define block and grid sizes.

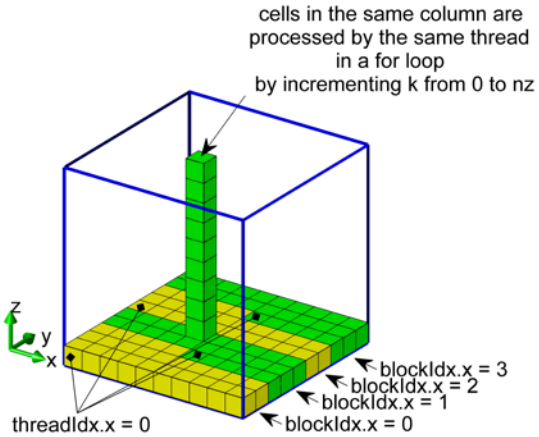


Fig. 4. Mapping of threads to cells of an FDTD domain using the xy-mapping.

The second mapping is partly the same as the first one: a thread block is constructed as a one-dimensional array, as shown on the first two lines in Listing 2, and the threads in this array are mapped to cells in an *x*-*y* plane cut of the FDTD domain as illustrated in Fig. 4. In the kernel function, each thread is mapped to a cell; *thread index* is mapped to *i* and *j*. Then, each thread traverses in the *z* direction in a *for* loop by incrementing *k* index of the cells. Field values are updated for each *k*, thus the entire FDTD domain is covered. As will be illustrated later, this

algorithm helps for global memory reuse, which improves efficiency. For the second mapping the above Listing 2 code will be modified for one line as

```
n_blocks_y = 1;
```

This mapping will be referred to as xy-mapping in the following sections.

B. Coalesced Global Memory Access

Memory instructions include any instruction that reads from or writes to shared, local or global memory. When accessing local or global memory, there are, 400 to 600 clock cycles of memory latency. Much of this global memory latency can be hidden by the thread scheduler if there are sufficient independent arithmetic instructions that can be issued while waiting for the global memory access to complete [29]. Unfortunately in FDTD updates the operations are dominated by memory accesses rather than arithmetic instruction. Hence, the memory access inefficiency is the bottle neck for the efficiency of FDTD on GPU. Global memory bandwidth is used most efficiently when the simultaneous memory accesses by threads in a half-warp (during the execution of a single read or write instruction) can be *coalesced* into a single memory transaction of 32, 64, or 128 bytes [29].

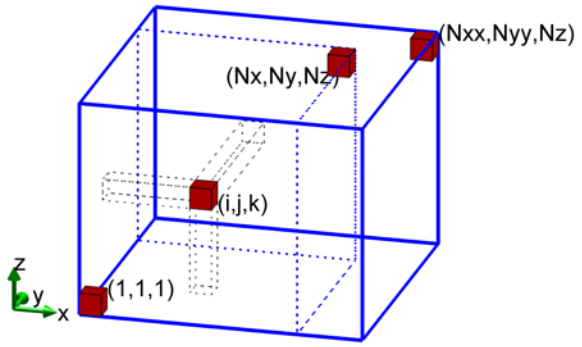


Fig. 5. An FDTD problem space padded with additional cells to ensure coalesced memory operations.

The three-dimensional field and coefficient arrays in FORTRAN are treated as one-dimensional arrays in kernel functions. It should be noted that the first array index varies most rapidly in FORTRAN multi-dimensional arrays. As shown in Listing 1, *i* index varies most rapidly, and then *j*. This ordering is retained after the arrays are transferred to GPU. If the size of the three-dimensional arrays, thus the size of the FDTD domain in number of cells, in the *x* and *y* directions is a multiple of 16, then the coalesced memory access is ensured. In general an FDTD domain size would be an arbitrary number. In order to achieve coalesced memory access, the FDTD domain is extended by padded cells such that the number of cells in *x* and *y* directions is an integer multiple of 16 as in Fig 5. Although, these padded cells increase the amount of memory need to be used to store array, it improves the efficiency of the kernel function tremendously. Thus the recommendation R2 is satisfied. The modified size of the FDTD domain becomes $Nxx \times Nyy \times Nz$, where Nxx, Nyy, and Nz are number of cells in *x*, *y*, and *z* directions, respectively.

Since the size of the FDTD domain has changed, calculation of the number of blocks in Listing 2 need to be slightly modified as

```
n_blocks_x = (nxx*nyy)/ number_of_threads))
  + ((nxx*nyy)%number_of_threads == 0 ? 0 : 1);
```

C. Use of Shared Memory

Because it is on-chip, the access to shared memory is much faster than the local and global memory. Parameters that reside in the shared memory space of a thread block have the lifetime of the block, and are accessible from all the threads within the block [29]. Therefore if a data block on global memory is going to be used frequently in a kernel, it is better to load the data to shared memory and reuse the data from the shared memory.

Shared memory is especially useful when threads need to access to unaligned data. For instance, examining Listing 1 reveals that in order to calculate $H_y(i,j,k)$, a thread mapped to the cell (i,j,k) needs E_x and E_z in (i,j,k) as well as E_x in $(i,j,k+1)$ and E_z in $(i+1,j,k)$. In the kernel code the index of a thread is calculated as

```
ci = blockIdx.x * blockDim.x + threadIdx.x;
```

This thread is mapped to a cell with *i* and *j* indices as

```
j  = ci/nxx;
i  = ci - j*nxx;
```

A cell with indices (i+1, j, k) can be accessed by `ci+1`, a cell with indices (i, j+1, k) can be accessed by `ci+nxx`, and a cell with indices (i, j, k+1) can be accessed by `ci+nxx*nyy`. Access to (i, j+1, k) and (i, j, k+1) are coalesced, however (i+1, j, k) is not. If an access to a field component at a neighboring cell in the x direction is needed, i.e. $E_z(i+1,j,k)$ while calculating $H_y(i,j,k)$ and $E_y(i+1,j,k)$ while calculating $H_z(i,j,k)$, then shared memory can be used to load the data block mapped by the thread block, and then the neighboring field value is accessed from the shared memory. At this point one needs to use the CUDA function `__syncthreads()` to ensure that all threads in the block are synchronized; thus all necessary data is loaded to the shared memory before it is used by the neighboring threads.

As discussed above, uncoalesced memory accesses can be eliminated by using shared memory. However, a problem arises when accessing the neighboring cells' data through shared memory. While loading the shared memory, each thread copies one element from the global memory to the shared memory. If the thread on the boundary of the thread block needs to access the data in the neighboring cell, this data will not be available since it has not been loaded to the shared memory. One way to overcome this problem is to load another set of data, which includes the neighboring cell's data, to shared memory. In the presented implementation the size of the data allocation in the shared memory is extended by 16, and some of the threads in the thread block are used only to copy data from global memory to this extended section in the shared memory. Then, for instance, the piece of code that calls the kernel function to update magnetic field components would be as in Listing 3.

The kernel function that updates magnetic field components based on xyz-mapping is shown in Listing 4.

```
threads = dim3(block_dim_x, block_dim_y, 1);
grid    = dim3( n_blocks_x,  n_blocks_y, 1);

shared_mem_size =
2*sizeof(float)*number_of_threads;

update_magnetic_fields_on_kernel
    <<<grid, threads, shared_memory_size>>>
  (nxx, nyy, nx, ny, nz,
   Ex,  Ey,  Ez,    Hx, Hy, Hz,
   Chxh,Chyh,Chzh,  Chxey,
   Chxez,    Chyez, Chyex,  Chzex, Chzey);
```

Listing 3. CUDA code to call kernel function for magnetic field updates.

```
__global__ void
update_magnetic_fields_on_kernel(int nxx, int
nyy, int nz, float *Ex, float *Ey, float *Ez,
float *Hx, float *Hy, float *Hz, float *Chxh,
float *Chyh, float *Chzh, float *Chxey, float
*Chxez, float *Chyez, float *Chyex, float
*Chzex, float *Chzey)
{
  extern __shared__ float sEyz[];
  float *sEy = (float*) sEyz;
  float *sEz = (float*) &sEy[blockDim.x+16];

  // ci: cell index
  // si: index in shared memory array

  int ci = blockIdx.x * blockDim.x +
threadIdx.x;
  int j  = ci/nxx;
  int i  = ci - j*nxx;
  int si = threadIdx.x;
  int sip1 = si+1;
  int nxxyy = nxx*nyy;
  int cizp;
  int ciyp;
  float ex;

  ci = ci + blockIdx.y*nxxyy;

  if (j < ny)
    {
      cizp = ci+nxxyy;
      ciyp = ci+nxx;
      ex = Ex[ci];
      sEz[si] = Ez[ci];
      sEy[si] = Ey[ci];
      if (threadIdx.x<16)
      {
      sEz[blockDim.x+threadIdx.x] =
      Ez[ci+blockDim.x];
      sEy[blockDim.x+threadIdx.x] =
      Ey[ci+blockDim.x];
      }
      __syncthreads();

      Hx[ci] = Chxh[ci]  *  Hx[ci]
             + Chxey[ci]  * (Ey[cizp]-Ey[ci])
             + Chxez[ci]  * (Ez[ciyp]-sEz[si]);

      Hy[ci] = Chyh[ci] *  Hy[ci]
             + Chyez[ci] * (sEz[sip1]-sEz[si])
             + Chyex[ci] * ( Ex[cizp]-ex);

      Hz[ci] = Chzh[ci] *  Hz[ci]
             + Chzex[ci] * (Ex[ciyp]-ex)
             + Chzey[ci] * (sEy[sip1]-sEy[si]);
  }
}
```

Listing 4. CUDA code to update magnetic field components based on xyz-mapping.

D. Data Reuse

As discussed above, the global memory access affects the performance of a CUDA program significantly. Therefore, data transfers from and to the global memory should be avoided as much as possible. It may even be better to recalculate some data instead of recalling the data from global memory. If some data is already transferred from the global memory and it is available, it is better to use it as many times as possible. As can be observed from Listing 1, such data reuse is possible in an FDTD algorithm: while calculating $H_x(i,j,k)$ and $H_y(i,j,k)$, $E_y(i,j,k+1)$ and $E_x(i,j,k+1)$ are used and the values of these components are ready in the registers of the thread. If one increments the *k* index by one, these values will be reused to calculate $H_x(i,j,k+1)$ and $H_y(i,j,k+1)$. Therefore, a kernel function can be constructed based on the xy-mapping in which each thread traverses in the *z* direction in a *for* loop by incrementing *k* index of the cells. A kernel function based on xy-mapping can be coded as shown in Listing 5.

```
__global__ void
update_magnetic_fields_on_kernel(int nxx, int
nyy, int nx, int ny, int nz,  float *Ex, float
*Ey, float *Ez, float *Hx, float *Hy, float
*Hz, float *Chxh, float *Chyh, float *Chzh,
float *Chxey, float *Chxez, float *Chyez, float
*Chyex, float *Chzex, float *Chzey)
{
  extern __shared__ float sEyz[];
  float *sEy = (float*) sEyz;
  float *sEz = (float*) &sEy[blockDim.x+16];

  int ci = blockIdx.x * blockDim.x +
threadIdx.x;
  int j  = ci/nxx;
  int i  = ci - j*nxx;
  int si = threadIdx.x;
  int sip1 = si+1;
  int nxxyy = nxx*nyy;
  int cizp;
  int cipnxx;
  float ey, eyzp;
  float ex, exzp;

  if (j < ny)
  {
    ey = Ey[ci];
    ex = Ex[ci];
    for (int k=0;k<nz;k++)
    {
      cizp    = ci + nxxyy;
      exzp    = Ex[cizp];
      eyzp    = Ey[cizp];
      sEz[si] = Ez[ci];
      if (threadIdx.x<16)
      {
        sEz[blockDim.x+threadIdx.x] =
        Ez[ci+blockDim.x];
      }
      __syncthreads();

      Hx[ci] = Chxh[ci]*Hx[ci]
             + Chxey[ci]*(eyzp-ey)
             + Chxez[ci]*(Ez[ci+nxx]-sEz[si]);

      Hy[ci] = Chyh[ci] * Hy[ci]
             + Chyez[ci] * (sEz[sip1]-sEz[si])
             + Chyex[ci] * (exzp-ex);

      sEy[si] = ey;
      if (threadIdx.x<16)
      {
        sEy[blockDim.x+threadIdx.x] =
        Ey[ci+blockDim.x];
      }
      __syncthreads();
      Hz[ci] = Chzh[ci] * Hz[ci]
             + Chzex[ci] * (Ex[ci+nxx]-ex)
             + Chzey[ci] * (sEy[sip1]-sEy[si]);

      ci = cizp;
      ey = eyzp;
      ex = exzp;
    }
  }
}
```

Listing 5. CUDA code to update magnetic field components based on xy-mapping.

At this point it should be noted that although the electric field updating equations are the same in form as the magnetic field updating equations, the implementation of kernels for electric field updates will be slightly different than those shown in Listings 4 and 5. The indices of the electric and magnetic field components adjacent to the FDTD domain boundaries and need to be updated are different as discussed in [3], and this difference need to be accounted for in the kernel implementations. Thus the implementations and also the performances of these kernels are slightly different.

E. Optimization of Number of Threads

As pointed out in recommendations R5 and R6, occupancy of the microprocessors and number of threads in a block are two other important parameters that affect the performance of a CUDA program. Number of threads and occupancy are tightly connected. It is possible to set the number of threads as a desired value while it may not be possible to control the occupancy; it is a function of number of threads, number of registers used in the kernel, amount of shared memory used by the

kernel, compute capability of the device, etc. A good practice is to optimize the number of threads while keeping the occupancy a reasonable value.

In order to determine optimum number of threads CUDA Visual Profiler is used: the kernel functions that update the electric and magnetic field components are run using different values of number of threads per block for both the xyz-mapping and xy-mapping algorithms, and the cpu times are recorded as they are captured by the CUDA Visual Profiler. For this test, an FDTD domain with size of 8 million cells $(200\times200\times200)$ is used. The result of the parameter sweep is shown in Fig. 6. It is found that for the magnetic field updates using xy-mapping algorithm performs the best with 512 threads per block, while electric field updates performs best with 128 threads per block. For the xyz-mapping both electric and magnetic field updates perform the best with 64 threads per block. These numbers are used in the subsequent performance analysis tests. From the figure it can be noticed that xy-mapping algorithm is faster than the xyz-mapping algorithm.

One can notice in Fig. 6 that, the cpu time is not shown for 448 and 512 number of threads for the electric field kernel using the xy-mapping. The number of registers for this kernel is 37 and occupancy becomes zero for large number of threads. Hence, the kernel cannot be run with 448 or 512 threads per block.

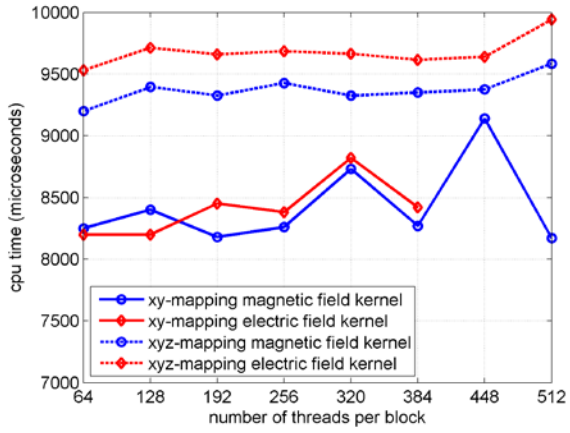


Fig. 6. CPU time versus number of threads per block.

V. PERFORMANCE ANALYSIS

The performance of the developed CUDA code for a general FDTD method as described before is examined as a function of problem size for both the xy-mapping and xyz-mapping algorithms. The analysis is performed on an NVIDIA® Tesla™ C1060 Computing Processor installed on a 64 bit Windows XP computer. This card has 240 streaming processor cores operating at 1.3 GHz. Size of a cubic FDTD problem domain has been swept and the number of million cells per second (*NMCPS*) processed is calculated as a measure of the performance of the CUDA program. Number of million cells is calculated as [20]

$$NMCPS = \frac{n_{steps} \times Nx \times Ny \times Nz}{t_s} \times 10^{-6}, \qquad (2)$$

where n_{steps} is the number of time steps the program has been run and t_s is the total time of program run in seconds. The result of the analysis is shown in Fig. 7. It can be observed that the xy-mapping algorithm processes about 450 million cells per second on the average while xyz-mapping algorithm processes 400 million cells per second.

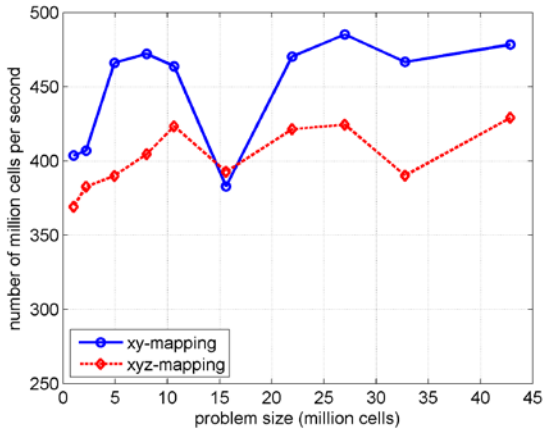


Fig. 7. Algorithm speed versus problem size.

VI. CONCLUSION

A CUDA implementation of FDTD method is presented in this contribution. The FDTD formulation considered is for general dielectric media and conductive media and does not assume the same cell sizes in *x*, *y*, and *z* directions. Two thread-to-cell mapping algorithms are discussed and it is shown that the so referred to as xy-

mapping algorithm is better in terms of performance.

It should also be noted that each cell in the FDTD problem space can have a different material. If a limited number of materials are considered, the presented codes can be revised based on material indexed FDTD formulation, thus GPU constant memory space, which is faster than the global memory, can be utilized and a faster CUDA implementation for these FDTD formulations can be achieved.

REFERENCES

[1] K. S. Yee, "Numerical Solution of Inital Boundary Value Problems Involving Maxwell's Equations in Isotropic Media," *IEEE Transactions on Antennas and Propagation*, vol. 14, pp. 302–307, May 1966.

[2] A. Taflove and S. C. Hagness, *Computational Electrodynamics: The Finite-Difference Time-Domain Method,* 3rd edition, Artech House, 2005.

[3] A. Elsherbeni and V. Demir, "The Finite Difference Time Domain Method for Electromagnetics: With MATLAB Simulations," *SciTech Publishing,* 2009.

[4] S. E. Krakiwsky, L. E. Turner, and M. M. Okoniewski, "Graphics Processor Unit (GPU) Acceleration of Finite-Difference Time-Domain (FDTD) Algorithm," *Proc. 2004 International Symposium on Circuits and Systems*, vol. 5, pp. V-265–V-268, May 2004.

[5] S. E. Krakiwsky, L. E. Turner, and M. M. Okoniewski, "Acceleration of Finite-Difference Time-Domain (FDTD) Using Graphics Processor Units (GPU)," *2004 IEEE MTT-S International Microwave Symposium Digest*, vol. 2, pp. 1033–1036, June 2004.

[6] R. Schneider, S. Krakiwsky, L. Turner, and M. Okoniewski, "Advances in Hardware Acceleration for FDTD," *Ch. 20 in Computational Electrodynamics: The Finite-Difference Time-Domain Method, 3rd edition,* Artech House, 2005.

[7] M. J. Inman, A. Z. Elsherbeni, and C. E. Smith "GPU Programming for FDTD Calculations," *The Applied Computational Electromagnetics Society (ACES) Conference*, 2005.

[8] M. J. Inman and A. Z. Elsherbeni, "Programming Video Cards for Computational Electromagnetics Applications," *IEEE Antennas and Propagation Magazine*, vol. 47, no. 6, pp. 71–78, Dec. 2005.

[9] M. J. Inman and A. Z. Elsherbeni, "Acceleration of Field Computations Using Graphical Processing Units," *The Twelfth Biennial IEEE Conference on Electromagnetic Field Computation CEFC 2006*, April 30 - May 3, 2006.

[10] M. J. Inman, A. Z. Elsherbeni, J. G. Maloney, and B. N. Baker, "Practical Implementation of a CPML Absorbing Boundary for GPU Accelerated FDTD Technique," *The 23rd Annual Review of Progress in Applied Computational Electromagnetics Society,* 19-23 March 2007.

[11] S. Adams, J. Payne, and R. Boppana, "Finite Difference Time Domain (FDTD) Simulations Using Graphics Processors," *Proceedings of the 2007 DoD High Performance Computing Modernization Program Users Group (HPCMP) Conference,* pp. 334–338, 2007.

[12] D. K. Price, J. R. Humphrey, and E. J. Kelmelis, "GPU-based Accelerated 2D and 3D FDTD Solvers," in *Physics and Simulation of Optoelectronic Devices XV*, of *Proceedings of SPIE*, vol. 6468, Jan. 2007.

[13] D. K. Price, J. R. Humphrey, and E. J. Kelmelis, "Accelerated Simulators for Nano-Photonic Devices," *International Conference on Numerical Simulation of Optoelectronic Devices 2007,* pp. 103–104, Sept. 2007.

[14] M. Inman, A. Elsherbeni, J. Maloney, and B. Baker, "Practical Implementation of a CPML Absorbing Boundary for GPU Accelerated FDTD Technique," *Applied Computational Electromagnetics Society Journal*, vol. 23, no. 1, pp. 16–22, 2008.

[15] M. J. Inman and A. Z. Elsherbeni, "Optimization and parameter exploration using GPU based FDTD solvers," *IEEE MTT-S International Microwave Symposium Digest*, pp. 149-152, June 2008.

[16] N. Takada, N. Masuda, T. Tanaka, Y. Abe, and T. Ito, "A GPU Implementation of the 2-D Finite-Difference Time-Domain Code Using High Level Shader Language," *Applied Computational Electromagnetics Society Journal*, vol. 23, no. 4, pp. 309–316, 2008.

[17] A. Valcarce, G. de la Roche, and J. Zhang, "A GPU Approach to FDTD for Radio Coverage Prediction," *Proceedings of the 11th IEEE Singapore International Conference on Communication Systems (ICCS '08)*, pp. 1585–1590, Nov. 2008.

[18] P. Sypek and M. Michal, "Optimization of a FDTD Code for Graphical Processing Units," *17th International Conference on Microwaves, Radar and Wireless Communications (MIKON)*, pp. 1–3, May 2008.

[19] A. Balevic, L. Rockstroh, A. Tausendfreund, S. Patzelt, G. Goch, and S. Simon, "Accelerating Simulations of Light Scattering Based on Finite-Difference Time-Domain Method with General Purpose GPUs," *Proceedings of the 2008 11th IEEE International Conference on Computational Science and Engineering*, pp. 327–334, 2008.

[20] P. Sypek, A. Dziekonski, and M. Mrozowski, "How to Render FDTD Computations More Effective Using a Graphics Accelerator," *IEEE Transactions on Magnetics*, vol. 45, no. 3, pp. 1324–1327, 2009.

[21] N. Takada, T. Shimobaba, N. Masuda, and T. Ito, "High-speed FDTD Simulation Algorithm for GPU with Compute Unified Device Architecture," *IEEE International Symposium on Antennas & Propagation & USNC/URSI National Radio Science Meeting*,. 4, 2009.

[22] A. Valcarce, G. De La Roche, A. Jüttner, D. López-Pérez, and J. Zhang, "Applying FDTD to the coverage prediction of WiMAX femtocells," *EURASIP Journal on Wireless Communications and Networking*, Feb. 2009.

[23] C. Ong, M. Weldon, D. Cyca, and M. Okoniewski, "Acceleration of Large-Scale FDTD Simulations on High Performance GPU Clusters," *2009 IEEE International Symposium on Antennas & Propagation & USNC/URSI National Radio Science Meeting*, 2009.

[24] M. J. Inman, A. Elsherbeni, and V. Demir, "Graphics Processing Unit Acceleration of Finite Difference Time Domain", *Ch. 12 in The Finite Difference Time Domain Method for Electromagnetics (with MATLAB Simulations),* SciTech Publishing, 2009.

[25] I. Buck, *Brook Spec v0.2*, Stanford Univ. Press, 2003.

[26] NVIDIA CUDA ZONE, http://www.nvidia.com/object/cuda_home.html.

[27] CUDA 2.1 Quickstart Guide, http://www.nvidia.com/object/cuda_develop.html.

[28] J. A. Roden and S. Gedney, "Convolution PML (CPML): An Efficient FDTD Implementation of the CFS-PML for Arbitrary Media," *Microwave and Optical Technology Letters,* vol. 27, no. 5, pp. 334–339, 2000.

[29] CUDA 2.1 Programming Guide, http://www.nvidia.com/object/cuda_develop.html.

[30] CUDA Best Practices Guide, http://www.nvidia.com/object/cuda_develop.html.

Veysel Demir is an Assistant Professor at The Department of Electrical Engineering, Northern Illinois University. He received his B.Sc. degree in electrical engineering from Middle East Technical University, Ankara, Turkey, in 1997. He studied at Syracuse University, New York, where he received both a M.Sc. and Ph.D. in electrical engineering in 2002 and 2004, respectively. During his graduate studies, he worked as research assistant for Sonnet Software, Inc., Liverpool, New York. He worked as a visiting research scholar in the Department of Electrical Engineering at the University of Mississippi from 2004 to 2007. He joined Northern Illinois University in August 2007. His research interests include numerical analysis techniques as well as microwave and radiofrequency (RF) circuit analysis and design.

Dr. Demir is a member of IEEE and ACES and has coauthored more than 20 technical journal and conference papers. He is the coauthor of the books *Electromagnetic Scattering Using the Iterative Multiregion Technique* (Morgan & Claypool, 2007) and *The Finite Difference Time Domain Method for Electromagnetics with MATLAB Simulations* (Scitech 2009).

Atef Z. Elsherbeni is a Professor of Electrical Engineering and Associate Dean for Research and Graduate Programs, the Director of The School of Engineering CAD Lab, and the Associate Director of The Center for Applied Electromagnetic Systems Research (CAESR) at The University of Mississippi. In 2004 he was appointed as an adjunct Professor, at The Department of Electrical Engineering and Computer Science of the L.C. Smith College of Engineering and Computer Science at Syracuse University. On 2009 he was selected as Finland Distinguished Professor by the Academy of Finland and Tekes.

Dr. Elsherbeni has conducted research dealing with scattering and diffraction by dielectric and metal objects, finite difference time domain analysis of passive and active microwave devices including planar transmission lines, field visualization and software development for EM education, interactions of electromagnetic waves with human body, sensors development for monitoring soil moisture, airports noise levels, air quality including haze and humidity, reflector and printed antennas and antenna arrays for radars, UAV, and personal communication systems, antennas for wideband applications, antenna and material properties measurements, and hardware and software acceleration of computational techniques for electromagentics.

Dr. Elsherbeni is the co-author of the book "*The Finite Difference Time Domain Method for Electromagnetics With MATLAB Simulations*", SciTech 2009, the book "*Antenna Design and Visualization Using Matlab*", SciTech, 2006, the book "*MATLAB Simulations for Radar Systems Design*", CRC Press, 2003, the book *"Electromagnetic Scattering Using the Iterative Multiregion Technique"*, Morgan & Claypool, 2007, the book *"Electromagnetics and Antenna Optimization using Taguchi's Method"*, Morgan & Claypool, 2007, and the main author of the chapters "*Handheld Antennas"* and "*The Finite Difference Time Domain Technique for Microstrip Antennas"* in Handbook of Antennas in Wireless Communications, CRC Press, 2001.

Dr. Elsherbeni is a Fellow member of the Institute of Electrical and Electronics Engineers (IEEE) and a Fellow member of The Applied Computational Electromagnetics Society (ACES). He is the Editor-in-Chief for ACES Journal and an Associate Editor to the Radio Science Journal.

A Practical Look at GPU-Accelerated FDTD Performance

Mike Weldon[1], Logan Maxwell[1], Dan Cyca[1], Matt Hughes[1], Conrad Whelan[1], Michal Okoniewski[1,2]

[1] Acceleware Corp. 1600 – 37th St. SW, Calgary, AB T3C 3P1, Canada
logan.maxwell@acceleware.com, mike.weldon@acceleware.com

[2] Department of Electrical and Computer Engineering
University of Calgary, Calgary, Alberta T2N 1N4, Canada

***Abstract*—** This paper outlines several key features and conditions that impact the performance of FDTD on GPUs. It includes relevant performance measurements as well as practical suggestions on how to mitigate their impact. Among these factors are: PML depth, the number of unique materials, dispersive materials, the impact of field reads/observations, simulation orientation, and domain decomposition using multiple GPUs. The paper shows that the performance of FDTD on GPUs can be limited in certain extreme cases, but with proper care on the part of the designer these cases can be managed and maximum performance guaranteed.

***Index Terms*—** GPU, acceleration, FDTD, CPML, dispersive materials.

I. INTRODUCTION

For several years, running FDTD (Finite Difference Time Domain) [1] on graphical processing units, or GPUs, accelerators has been shown as a successful technique to reduce run times [2-4]. The fine-grained parallelism of FDTD maps well to the several hundreds of computational streaming processor cores available on modern GPU hardware. The faster memory bandwidth from GPU RAM to the GPU processing elements is also largely responsible for the observed performance gain versus traditional CPU (central processing unit) architectures.

The complexity involved when writing GPU-enabled FDTD codes involves making sure that the processing elements are not data starved. This is done through effective memory, cache and memory bandwidth management. This complexity must be addressed for every feature of FDTD, not just the basic [1] Yee updates.

Section II of this paper will introduce and explain the basics of FDTD performance on GPUs from an end user perspective. It will also detail the general limiting cases of this performance caused by the GPU architecture mentioned above.

The body of the paper, Sections III through IX, will build on this overview and introduce more advanced features and their impact on performance. These features are: PML (Perfectly Matched Layer) boundary conditions, the number of unique materials, dispersive materials, simulation orientation, observation/modification of field data during the simulation, and domain decomposition across multiple GPUs. In each case, the performance of the feature or concept is illustrated with a graph and explained in words. In addition, practical suggestions are offered for ensuring maximum performance and mitigating any adverse effects.

To illustrate these effects, Acceleware's FDTD library, version 9.x, implemented in CUDA and running NVIDIA Tesla C1060 are the software and GPU platforms of choice.

II. FUNDAMENTALS OF FDTD PERFORMANCE ON GPU

The graph below is an overview that illustrates the performance of Acceleware's FDTD library on both multi-core CPU and GPU hardware. The CPU hardware used throughout the paper is AMD Opteron™ 2214 2.2GHz, and Intel® Xeon® CPU X5550 @ 2.67GHz code-named 'Nehalem'. The GPU hardware is NVIDIA Tesla C1060 GPU

cards connected via PCI Express x16 to the host system.

Memory bandwidth is perhaps the key determinant of FDTD performance from a hardware perspective. The memory bandwidth of the C1060 architecture is 102GB/S, the Xeon X5550 architecture is 32GB/s, and the Opteron 2214 architecture is 11.8GB/s. These numbers roughly correlate with the peak performance observed for FDTD in Figure 1 below. Newer generations of both CPU and GPU hardware will continue to increase memory bandwidth.

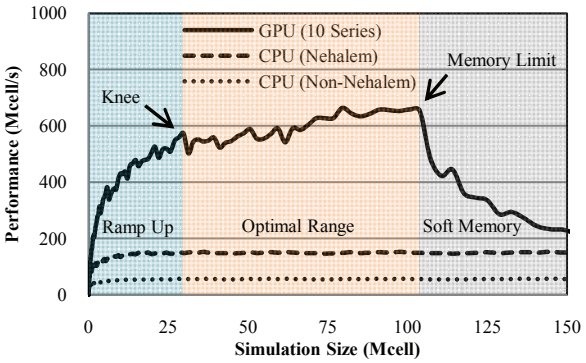


Fig. 1. FDTD performance – CPU vs. GPU.

CPU performance versus simulation size ramps up relatively quickly and reaches a constant steady state limited by the finite memory bandwidth, and provided the CPU memory is not exceeded. The GPU performance curve is more dramatic, and shows several key operating regimes which are noted in Fig. 1 and explained below.

Ramp Up - In this range the GPU is not using all of its compute resources and memory bandwidth efficiently. PML may also take up a large portion of the total simulation size and acts to slow the total simulation throughput.

Knee - The knee is the point at which the performance levels off and the GPU is running optimally.

Optimal Range - This is the optimal range for the GPU as processing resources are being fully utilized and the impact of data transfer minimized. The goal of any GPU FDTD code is to maximize the breadth and magnitude of this region.

GPU Memory Limit - This is the point at which the GPU runs out of memory. There is a clear and dramatic performance impact beyond this point due to FDTD updates shifting to the CPU. While the amount of GPU RAM is fixed, the memory limit as measured by the number of cells will change depending on the materials and features of the simulation.

'Soft Memory' - In this area the CPU is solving the remaining calculations that the GPU does not have memory for. As simulation size goes further into soft memory, the performance will approach that of the CPU.

The performance in MCells/s of throughout the paper is calculated as follows:

$$Performance\left(\frac{Mcells}{s}\right) = \frac{Simulation\ Size\ (Mcells) x\ Time\ Steps}{Simulation\ Time\ (s)} \quad (1)$$

In the above calculations, the 'simulation size' does *not* include any PML cells used in the simulation, while the 'simulation time' is the time elapsed from the beginning of the time stepping.

Unless being treated as an independent variable or otherwise noted, the performance results in this document have 16 dielectric materials, four-layer CPML (convolutional perfectly matched layer) [5], are cubic, and have field observations disabled. This includes the results in Fig. 1. While 16 materials and four-layer CPML may be a simplistic case compared with more advanced simulations which use more of each, the effect on performance is enough to be representative of a broad range of simulations. The precise dependence on more CPML and more materials are both examined in subsequent sections.

Finally, results in this paper are all reported for single precision, floating-point numerical representation of field and material data. FDTD has the advantage that the numerical dispersion is usually more significant than any single precision error. This is fortuitous since the double precision performance of GPUs has, until recently, been an order of magnitude slower than single precision.

III. PERFECTLY MATCHED LAYERS (PML)

Adding absorbing CPML (convolutional perfectly matched layer) boundary layers is done to truncate the overall simulation volume. Computation of these cells is made more intensive due to the recursive convolution performed. The number of layers required depends on the desired reflection coefficient from the boundary and is done at the discretion of the designer. Typically five to ten layers are used, with five providing -30dB or better reflection. [5]

While reducing reflections, these layers can also reduce simulation performance by as much as 50% [6], especially for small simulation volumes. The maximum simulation size the GPU is capable of running will also be partially reduced, as CPML cells require more memory than non-CPML cells. They also are more expensive to compute, which is why the performance is reduced.

Figure 2 below shows GPU-accelerated FDTD performance for simulations with different amounts of CPML. Note that both the performance and maximum GPU-accelerated simulation size are affected. Also notice that the point at which the GPU enters the soft-memory limit is reduced. This is a reflection of the increased memory usage of the CPML cells.

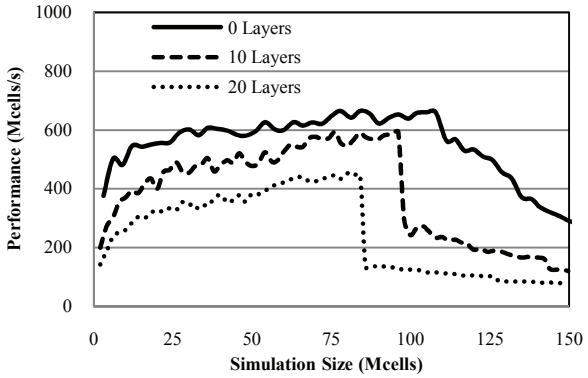


Fig. 2. GPU-accelerated FDTD performance with CPML layers.

Small simulations are impacted more than larger simulations because CPML cells represent a greater majority of the computational load.

Minimizing the use of CPML as a technique to preserve performance should be evident, and is already well known for CPU implementations. How much CPML a given simulation requires can only be understood by the designer, who will balance reflections, accuracy and performance.

IV. NUMBER OF UNIQUE MATERIALS

The number of materials, defined as materials with unique permeability, permittivity, electric and magnetic conductivity, can have a large impact on performance - up to a 20% decrease. In the case of Acceleware's implementation, this is a result of the way these properties are stored for use on the GPU.

Acceleware's library employs the well known look-up table method to save memory and avoid unnecessary copies of material data. In this technique, the look-up table index is passed to the computational kernel which in turn fetches the material parameters before completing the update. This works well (uses the least memory) for simulations where the number of unique materials is much less than the number of cells. For simulations with very large numbers of materials, it becomes more advantageous from a memory perspective to send the material parameters directly to the kernel. While there can be a performance impact to doing so, it may be more memory efficient. This so called direct technique can also support unique E and H materials up to the number of cells in the simulation.

The performance is illustrated below in Fig. 3 and shows that there is indeed a performance impact for the C1060 GPU hardware when moving from the look-up table method to the direct method. The number of unique E and H materials is also shown to have an independent and additive effect on simulation performance.

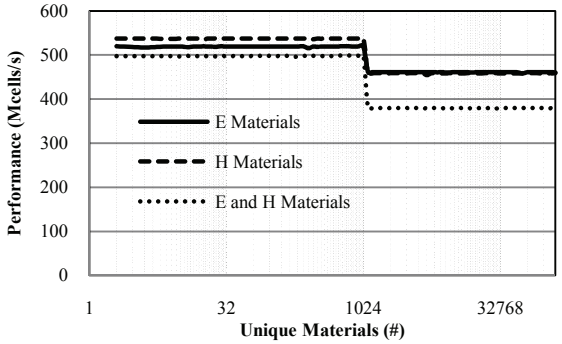


Fig. 3. GPU-accelerated FDTD performance versus the number of materials.

The choice of 1024 is somewhat arbitrary and unique to the Acceleware library. Other implementations may use a different break point or use the look-up table method or the direct method exclusively. On different GPU hardware, the direct and look-up table methods perform differently, which adds further complexity to understanding the performance.

The general end user recommendations should be to simply keep an awareness of the number of unique materials in your simulation and ensure they are not an unnecessary cause of simulation slow down. Many applications add arbitrary complexity by allowing for continuous variation of the material parameters.

For FDTD developers, additional intelligence in the library itself may also more automatically optimize the material storage and access *a priori* depending on the number of materials, hardware, and kernel implementations available. This would ensure an optimal performance where the simulation is not memory limited and maximum simulation size where it is.

V. DISPERSIVE MATERIALS

Simulating materials with dispersive properties can have an even more significant impact on simulation performance and maximum simulation size. Both the order of the dispersive materials (number of poles) and the total volume of dispersive material need to be considered. For a given cell, adding a single dispersive pole to describe its behavior will increase the memory requirement of the cell and increase the required number of flops for additional material current calculation. This increases proportionally with the number of poles. Simulations with larger volumes and more higher-order poles will hence show more pronounced degradation of performance and more reduced simulation size. Several relevant cases are shown in Fig. 4 below.

The above effect applies to all dispersive materials types: Drude, Debye, Lorentz, Drude-Lorentz, etc. Simulations with dispersive materials also run slower on the CPU, so the 'speed up factor' when using GPUs is roughly the same as for non-dispersive simulations.

Managing the effect of dispersive materials involves using only the minimum volume and order required to achieve your desired result. As is illustrated comparing cases four with five, and two with three, the distribution of dispersive materials does not significantly affect the performance; it is the overall volume and order that counts.

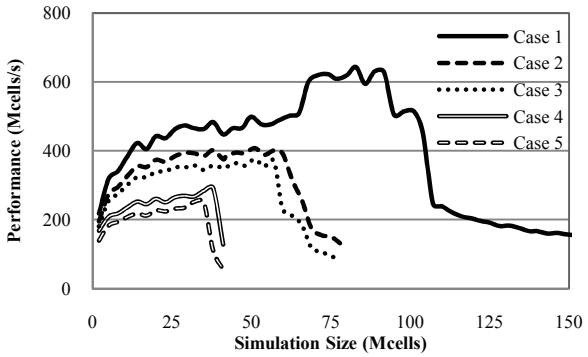


Fig. 4. GPU-accelerated FDTD performance for several cases of dispersive material usage.

Case 1 *– 1600 non-dispersive materials distributed evenly thought the entire simulation space.*

Case 2 *– 1 single-pole dispersive material occupies 40% of the total volume contiguously.*

Case 3 *– 1 single-pole dispersive distributed evenly throughout the entire volume, 40% of the total volume is made up of dispersive materials.*

Case 4 *– 1600 Multi-pole dispersive materials distributed contiguously throughout 40% of the total volume.*

Case 5 *– 1600 Multi-pole dispersive materials distributed evenly throughout the entire volume, 40% of the total volume is made up of dispersive materials.*

VI. READS AND READ REGIONS (WRITES AND WRITES REGIONS)

Moving field data between GPU and CPU system memory during a simulation can dramatically impact performance and has been discussed as a limitation of GPU FDTD implementations. For the purpose of this discussion we will refer to these data moves as *reads* and *writes*. Field data is read when calculating relevant outputs like SAR, far field patterns, optical generation, special updates for assessing convergence, and in other regards. Field data is

written in cases like soft or customized excitations or active materials.

The two critical factors affecting performance for reads and writes are: how much of the volume is read/written and how frequently. Figure 5 below shows performance for different read volumes based on a percentage of the total volume. All six fields are read for each cell in the volume. The number of time steps between each volume read is swept and shown on the horizontal axis.

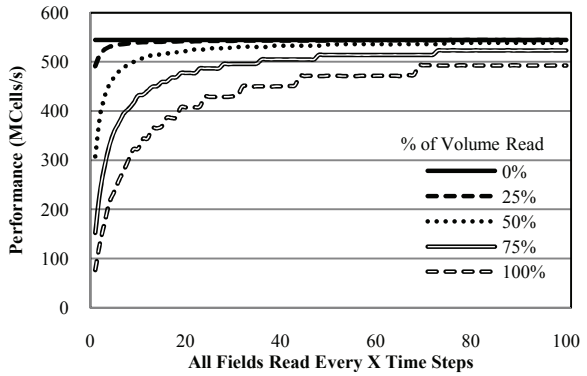


Fig. 5. GPU-accelerated FDTD performance for different field observations.

The above volume reads are made for contiguous volumes within the simulation space which is a simplified though still realistic case. The other extreme would be a large number of individual point reads dispersed evenly through a volume or plane. Individually reading these points one by one would further reduce the performance given the overhead attached with each read and their disparate locations in physical memory. Acceleware has implemented a strided region function as one technique to eliminate this overhead. An exhaustive study of all possible read patterns and techniques is a significant effort in itself and beyond the scope of this paper.

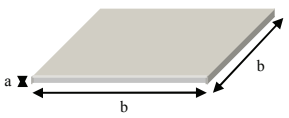
The general suggestions to manage the impact of field observations are relevant in all cases. They are: keep the read volume to a minimum, only observe the region (volume) that is of direct interest, and read only as frequently as is necessary to achieve accurate power, DFT, SAR, optical generation or other results. For steady state measurements like optical generation, far field etc. only start to read after a simulation has converged.

VII. SIMULATION ORIENTATION

Single-GPU simulations where the number of cells on one particular axis is significantly smaller than the others will experience a decrease in simulation performance and maximum simulation size. 'Significant' in this case is defined as a dimension that is 20% or less of the size of the other dimensions. For the Acceleware library, the particular dimension is Z, but this is implementation dependent.

This behavior is related to the way in which memory is optimally accessed for a given 3D layout in memory. This problem is not unique to GPU FDTD solutions; it is also present in vectorized CPU-only FDTD solvers.

The example illustrated in Fig. 6 and plotted in Fig. 7 shows an extreme case, 10:10:1, of smallest dimension. For less extreme cases the decrease in performance and max simulation size is proportionally smaller.



	X	Y	Z
X Smallest	(a,	b,	b)
Y Smallest	(b,	a,	b)
Z Smallest	(b,	b,	a)

Fig. 6. Illustration of an extreme simulation orientation case.

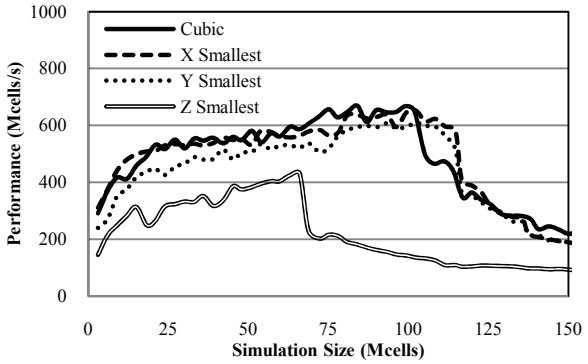


Fig. 7. GPU-accelerated FDTD performance for various extreme simulation orientations.

An important note is that partitioning across multiple GPUs will change the effective simulation dimensions on each GPU, and hence the performance, which is an important fact to consider.

To manage the effects of simulation orientation, simply rotate the simulation so that the Z (or critical) axis is not the smallest dimension. Avoid extreme differences in dimensions between the axis, if possible. Cubic simulation sizes will show the best performance.

VIII. MULTI-GPU SYSTEMS

Using multiple GPUs in concert on a single problem will increase both performance as well the maximum simulation size that can be run in full accelerated mode. Doing this requires significant additional complexity in the code, as it is not handled automatically at the hardware or driver level. The performance curve shown in Figure 1 for one GPU is now extended to show two, four and eight GPUs and plotted in Figure 8.

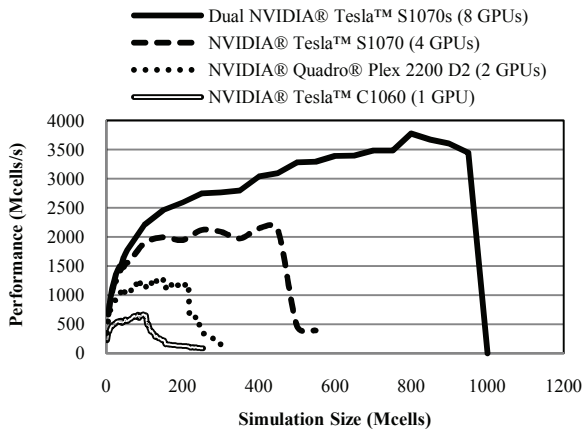


Fig. 8. GPU-Accelerated FDTD performance on multiple GPUs.

In the Acceleware implementation, the scaling with the number of GPUs depends on the simulation size. Small simulations in the ramp up range will experience a smaller scaling factor than simulations in the optimal range. For a simulation of 100 MCells, scaling is on the order of 80-90 percent up to four GPUs with diminishing returns going above four. However, if one considers the maximum throughput of each configuration, the scaling remains over 70 percent all the way up to eight GPUs. The scaling is plotted in Figure 9 below.

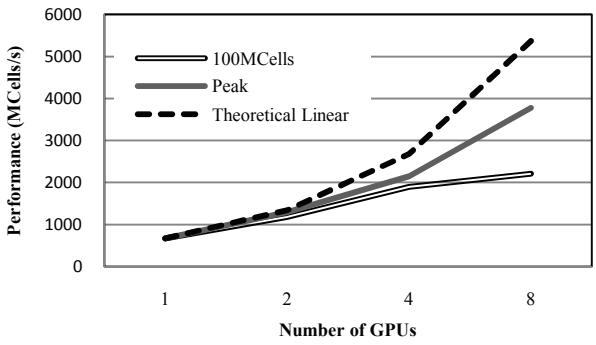


Fig. 9. GPU-Accelerated FDTD performance scaling across multiple GPUs. Peak performance observed.

Scaling beyond eight GPUs is also possible, but necessitates the use of an MPI or cluster layer above the GPU code. Clusters of up to 64 GPUs have been demonstrated at Acceleware and more details can be found in [8]. In general, GPU cluster scaling remains above 60% and well above that of CPU performance for large numbers of cores.

IX. OTHER CONSIDERATIONS

There are several other practical considerations to be aware of when running GPU accelerated FDTD. It is common to see GPUs used for FDTD computation also used to drive a display device either directly or indirectly. Display and computational work contesting for the same GPU resources can negatively impact performance. The two most common ways this can happen are with graphically intensive applications or screen savers, and with remote desktop applications.

Screen savers are not that impactful to simulation performance, typically less than 5%, but running a blank or non-3D screen saver will ensure maximum simulation performance.

Remote desktop applications on the other hand can have a severe, >50%, impact on performance and also prevent simulations from running. The best way to avoid this problem is to use a KVM, Keyboard-Video-Mouse, as it does not require any additional GPU resources. Next to that, IP-based remote desktop applications such as UltraVNC are another solution but can still reduce performance by 10-30%.

X. CONCLUSION

As GPU-accelerated FDTD has become an accepted and advantageous computational technique, its use is becoming more and more widespread. With increased usage, several practical limitations have been exposed. Most of these are in extreme cases and depend heavily on the particular implementation of accelerated FDTD functions as well as the hardware itself.

This paper looked at several of the most common practical limitations and suggested techniques to prevent them from excessively impacting performance. These included the number of materials, dispersive materials, PML absorbing boundaries, the extent of field observations, simulation orientation, and the use of multiple GPUs. Performance reductions vary from 5% to 50% versus a similar simulation in a less extreme case.

It is demonstrated that with proper care on the part of the end user, any performance degradation can be mitigated or eliminated to achieve the maximum benefit of running on GPUs. From both a SW and HW development perspective it also exposes potential architectural limitations of GPUs and should be a call to developers and designers to examine their code and hardware to further improve performance in these extreme cases.

REFERENCES

[1] K. S. Yee, "Numerical Solution of Initial Boundary Value Problems Maxwell's Equation in Isotropic Media", *IEEE Trans. Antennas and Prop*., Vol. 14, No. 3, pp. 302-307, 1966.

[2] S. E. Krakiwsky, L. E. Turner, M. Okoniewski, "Acceleration of Finite-Difference Time-Domain (FDTD) Using Graphics Processor Units (GPU)", *IEEE MTT-S Int. Symposium Digest*, Vol. 2, pp. 1033-1036, 2004.

[3] P. F. Curt, J. P. Durbano, M. R. Bodnar, S. Shi, M. S. Mirotznik "Enhanced Functionality for Hardware-Based FDTD Accelerators," *ACES Journal,* Vol. 22 No. 1, 2007.

[4] P. Sypek, M. Mrozowski, "Optimization of a FDTD code for graphical processing units," *The 17th. Int. Conf. on Microwaves, Radar and Wireless Communications,* MIKON, May 2008.

[5] A. Taflove, S. Hagness. *Computational Electrodynamics: The Finite-Difference Time-Domain Method*, 3rd ed., Artech House, 2005.

[6] M. J. Inman, A. Z. Elsherbeni, J. G. Maloney, and B.N. Baker, "GPU Based FDTD Solver with CPML Boundaries," *IEEE Antennas and Propagation Society International Symposium*, pp. 5255- 5258, 2007.

[7] J. A. Roden and S. D. Gedney, "Convolutional PML (CPML): An efficient FDTD implementation of the CFS-ML for arbitrary media", *IEEE Transactions on Antennas and Propagation,* Vol. 50, 2002, pp. 258-265.

[8] C. Ong, M. Weldon, D. Cyca, and M. Okoniewski, "Acceleration of large-scale FDTD simulations on high performance GPU clusters," *IEEE Antennas and Propagation Society International Symposium, APSURSI '09,* pp. 1 – 4, 2009.

Mike Weldon has an MSEE from the University of Calgary / TRLabs where he researched RF to optical conversion techniques for wireless network infrastructure. He also spent five years as an RF development engineer working on broadband Doherty power amplifiers at Nortel. He is currently a product manager at Acceleware responsible for the GPU-accelerated, RF/optical FDTD product.

Logan Maxwell is presently an intern at Acceleware and will graduate next year with a BSc in Electrical Engineering from the University of Calgary.

Dan Cyca has an M. Sc. in Electrical Engineering. He has spent six years developing GPU computing products at Acceleware.

Matt Hughes is currently a senior software developer with Acceleware, working on various Professional Services projects. He was the team lead for Acceleware's FDTD product prior to heading up the linear algebra team. Prior to joining Acceleware in 2005, Matt completed a M.A.Sc in Electrical Engineering at the University of Victoria, where he studied optical transmission through thin metal films using FDTD simulations on shared memory and distributed computers. He obtained a B.Sc in Electrical Engineering from the University of Calgary in 2003.

Conrad Whelan studied Electrical and Computer Engineering at the University of Calgary where he was awarded BSc and MSc degrees. During his time with the applied electromagnetics group, he investigated conformal methods for reducing the run time of patch antenna simulations. This segued right into his work with Acceleware where he has been a member of the FDTD team for four years, seeing the full range of transition from CPU to OpenGL GPU computing to the arrival of CUDA and the integration of Multi-node MPI for cluster operations.

Michal Okoniewski, P.Eng., is a Professor at the Department of Electrical and Computer Engineering, University of Calgary. He holds Libin/Ingenuity Chair in biomdeical-engineering and Canada Research Chair in applied electromagnetics. His interests range from computational electrodynamics, to tunable reflectarrays, RF MEMS and RF micro-machined devices, as well as hardware acceleration of computational methods. He is also involved in bio-electromagnetics, where he works on tissue spectroscopy and medical imaging. Dr Okoniewski is a fellow of IEEE and member of IEEE AP-S AdCom. In 2004 he co-founded Acceleware Corp.

A Stacking Scheme to Improve the Efficiency of Finite-Difference Time-Domain Solutions on Graphics Processing Units

Veysel Demir

Department of Electrical Engineering
Northern Illinois University, DeKalb, IL 60115, USA
demir@ceet.niu.edu

Abstract—Advances in computer hardware technologies accompanied by easy-to-use parallel programming software platforms have led to the wide spread use of parallel processing architectures, such as multi-core central processor units (CPUs) and graphic processing units (GPUs), in technical and scientific computing. Among electromagnetic numerical analysis methods, the finite-difference time-domain (FDTD) method is very well suited for parallel programming, and several implementations of FDTD have been developed and reported to solve electromagnetics problems orders of magnitude faster. Examination of performances of these implementations reveals that, in general, it is more efficient to solve larger FDTD domains than smaller domains. In this paper it is demonstrated that one can exploit the higher efficiency inherent to the solution of larger problem sizes to solve parameter sweep and optimization problems faster: instead of solving multiple smaller FDTD domains separately, these domains can be combined or stacked to form a larger problem and the large problem can be solved more efficiently. It has been shown that up to 40% faster solution can be achieved on GPUs with this method.

Index Terms—FDTD methods, parallel architectures, graphics processing unit (GPU) programming, Compute Unified Device Architecture (CUDA), hardware accelerated computing.

I. INTRODUCTION

The finite-difference time-domain (FDTD) method [1]-[3] has been the most popular numerical analysis technique throughout the past decades to solve a variety of electromagnetics problems. In FDTD, the problem space is composed of cells, in which electric and magnetic field components are located at discrete positions. These field components are recalculated at every time-step of a time-marching algorithm. The calculations for each cell can be performed independent from other cells at each time step; thus FDTD is very suitable for parallel programming. Until recently central processor units (CPUs) have been the main hardware architecture to perform high performance scientific and technical computing, and several implementations of FDTD have been developed for high performance CPU clusters and multi-core CPUs.

Recently, graphic processing units (GPUs), equipped with hundreds of processing cores, have evolved rapidly and outmatched CPUs in terms of computation power. Accompanied by advances in parallel programming software technologies, the advances in GPUs enabled widespread use of these devices, which had been initially designed for processing computer graphics, for general purpose computing. Initially GPUs were designed to support only single-precision floating-point arithmetic operations, which is sufficient for graphics processing. To further aid general purpose computing, latest generation GPUs support double-precision floating-point arithmetic operations as well. Thus graphics cards have evolved into computation cards.

Implementations of various numerical analysis methods have been developed on GPU platforms to solve electromagnetics problems faster. In particular, several implementations of FDTD method have been developed and reported [4]-[24]. These implementations are based on various programming platforms. For instance, [4]-[7] are

based on OpenGL; [8]-[14] are based on Brook; [15] uses High Level Shader Language (HLSL); and [16]-[20] are based on compute unified device architecture (CUDA) [25]. Independent of the different programming languages used in these contributions, it has been shown that FDTD problems can be solved orders of magnitude faster on graphics cards.

Parameter sweep and optimization are two commonly used techniques in the design of circuits. Electromagnetic calculations are computationally expensive and usually it takes a long time to run a simulation. Since, parameter sweeps and most optimization techniques need a large number of runs to achieve their target; long execution times can seriously hinder the adoption of these techniques. With the significant speed gains available with the GPU based FDTD solvers; optimization becomes a viable option in electromagnetic design [13]. The use of GPU based FDTD solvers for optimization and parameter sweep has been presented in [13]. Similarly, GPU based FDTD is used in [16] and [20] in optimization for radio coverage prediction.

One reasonable way to measure the efficiency of an FDTD implementation is to calculate the number of cells processed per second, in other terms the *throughput*, such as [26]

$$NMCPS = \frac{n_{steps} \times Nx \times Ny \times Nz}{t_s} \times 10^{-6}, \qquad (1)$$

where *NMCPS* is the number of million cells processed per second, n_{steps} is the total number of time steps the program has been run, and t_s is the total computation time in seconds. Here, Nx, Ny, and Nz are the number of cells in an FDTD problem space in x, y, and z directions, respectively. Such throughput data as a function of the FDTD domain size have been provided in [18], [24], and [26]. Generally the trend of throughput as a function of problem size in these data is similar to that illustrated in Fig. 1. The data shows that the computation efficiency is directly proportional to the problem size; i.e. the efficiency is higher for larger problem sizes. This trend is expectable since it is more efficient to load larger amounts of data to the GPU memory at once than loading smaller chunks at multiple times. Furthermore, the multiprocessors can more efficiently schedule the threads for larger number of threads, thus problem sizes. The higher efficiency inherent to the solution of a larger domain implies that if solutions of multiple smaller problems are required, it will be more efficient to combine their spatial domains as a single larger domain and solve the larger problem. This scenario fits to optimizations or parameter sweeps very well, since solutions of multiple similar size problems are sought in such cases: Similar size FDTD spatial domains can be stacked, where each domain is electromagnetically isolated from the others, and a large domain can be obtained. The entire combined domain can be solved in a single run. Using this method, a significantly faster solution can be achieved compared to the case where all the individual domains are solved separately.

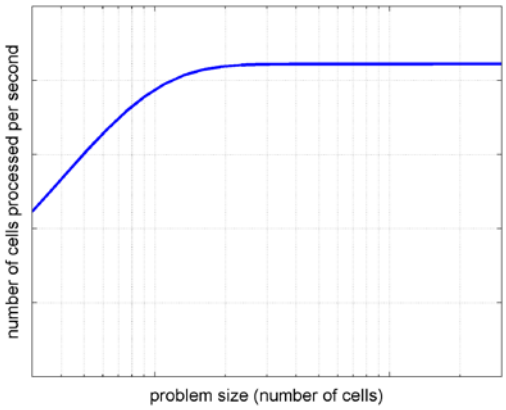


Fig. 1. Throughput versus problem size.

This paper demonstrates, via examples, that overall simulation time can be significantly reduced by a CUDA based FDTD code using the presented stacking method. The paper is organized as follows. Section II presents an FDTD implementation based on CUDA for GPU platforms. Section III discusses various schemes to stack FDTD spatial domains and shows time reductions in calculation times achieved by these stacking schemes in an example case. Section IV presents an analysis to examine the effect of orientation of the problem geometry on the efficiency of the solutions. Section V discusses some other benefits the stacking method can offer to the FDTD solutions of problems.

II. FDTD USING CUDA

New software development platforms and languages have accompanied and aided the general purpose GPU (GPGPU) computing as it has evolved and become widespread. OpenGL, Brook, and HLSL are among those languages that are commonly used, as discussed earlier, to program FDTD. However, steep learning curves of these languages prevent their widespread use.

Recently, Compute Unified Device Architecture (CUDA) is introduced by NVIDIA as a software development platform. CUDA is a general purpose parallel computing architecture. To program the CUDA architecture, developers can use C, which can then be run at great performance on a CUDA enabled processor [27]. Compared to other programming languages, some advantages and disadvantages of CUDA can be listed [28]. One of the main limitations of CUDA is that it can be used only with CUDA-enabled GPUs, which are manufactured only by NVIDIA. Some of the main advantages of CUDA are listed as availability of scattered reads from arbitrary addresses in memory, and shared memory that can be simultaneously accessed by the threads in the same thread block. Besides these advantages, two major factors led to its widespread use in many applications including FDTD: NVIDIA provides extensive support to programmers who would like to develop codes using CUDA, and programming for GPU computing is easier with CUDA.

In this current contribution a CUDA implementation of FDTD is used to prove the performance improvement achieved by the proposed stacking method. The details of this implementation are presented in [29], where the algorithm of the implementation is referred to as *xy*-mapping. In order to aid the understanding of discussions in the subsequent sections, some details of this implementation are summarized here as a reference.

In CUDA a number of threads work in parallel and form a thread block, while a number of thread blocks form a grid. In the *xy*-mapping algorithm, a grid of threads is mapped to the cells in the *xy*-plane cut of an FDTD problem space. Each thread is mapped to a cell and the field components in that cell are updated by the thread. Each thread then traverses the cells in the same column in the *z* direction and updates the field components in the same column, as illustrated in the pseudocode in Listing 1. This algorithm implies that there is anisotropy in the computations in the sense that computation in the *z* direction is treated differently.

In CUDA, it is very important to have global memory accesses coalesced to achieve faster computations. In the *xy*-maping algorithm, to make sure that the global memory accesses are coalesced, the FDTD problem space is enlarged by padding extra cells to the problem space, such that the number of cells in *x* and *y* directions are integer multiples of 16. For instance, if a problem space is composed of $Nx \times Ny \times Nz$ cells, the problem size becomes $Nxx \times Nyy \times Nz$ after the padding, where Nxx and Nyy are integer multiples of 16.

It should be noted that in the *xy*-mapping algorithm, the fields in the cells padded in the *x* direction are computed by the associated threads, while the cells padded in the *y* direction are not processed, as shown in Listing 1. This algorithm implies another anisotropy in the *x* and *y* directions throughout the computations. As a result, the FDTD algorithm in consideration is anisotropic in the sense of computations in *x*, *y*, and *z* directions. Therefore, if a non-cubic problem space will be computed, different computation performances should be expected if the problem geometry is rotated to align in different directions.

```
Function update_magnetic_fields

  Calculate thread index ti
  Calculate cell index i and j using ti

  If j < ny
    For k from 1 to nz
      Update Hx, Hy, and Hz
    End for
  End if

End function
```

Listing 1. Pseudocode of CUDA kernel to update magnetic field components based on *xy*-mapping.

III. STACKING SCHEMES

Many different scenarios can be envisioned to stack smaller FDTD spatial domains to obtain a larger domain. For instance, domains can be stacked in a linear sequence in one-dimension, a planar sequence in two-dimensions, or a cuboidal sequence in three-dimensions. In this contribution, linear stacking is considered for the analyses. The linear stacking, as well, can be achieved in three-different scenarios: stacking in the *x* direction, stacking in the *y* direction, or stacking in the *z* direction, as illustrated in Fig. 2 for three domains. These three schemes will be referred to as *x*-stacking, *y*-stacking, and *z*-stacking, respectively, in the following discussions.

As discussed in the previous section, the algorithm acts differently in different directions. This algorithmic anisotropy would cause a different calculation time every time the problem spaces are stacked in different directions.

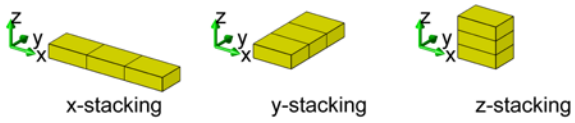


Fig. 2. FDTD problem spaces stacking schemes.

A performance analysis test is performed to find which direction is the best for stacking. An NVIDIA® Tesla™ C1060 Computing Processor running at 1.3 GHz is used for the tests presented in this paper. A problem space composed of $100\times100\times25$ cells is used as the base FDTD spatial domain, where a smaller number of cells is used in the *z* direction to model a microstrip type structure. Then this domain is stacked in *x*, *y*, and *z* directions. Each time the number of stacked domains is increased, simulation is performed, and throughput is calculated. The results of this test are plotted in Fig. 3. It is found that as the problem size increases the efficiency increases, as expected. Furthermore, the *x*-stacking is found to be the best performing scheme, while the *z*-stacking is the worst. One reason for why *x*-stacking performs better is that as the problem domain is enlarged in the *x* direction, the number of unnecessarily processed padding cells becomes negligible compared to the number of cells in the main domain. The reason for why the *y*-padding performs better than the *z*-padding is that as the domain size increases, the number of thread blocks also increases with the *y*-padding, and the thread blocks are scheduled more efficiently by the GPU multiprocessor.

In this given example, the base problem domain size is 250,000 cells, and this number of cells is processed with a throughput of 340 million cells per second. When 128 of this domain are stacked in the *x* direction, the problem size becomes 32 million cells, while the throughput becomes 497 million cells per second. These numbers show that, for instance, if 128 runs of the base domain are required for an optimization problem, it will be more than 40% faster to complete the optimization using the proposed stacking method compared to the case where all the base domains are solved separately.

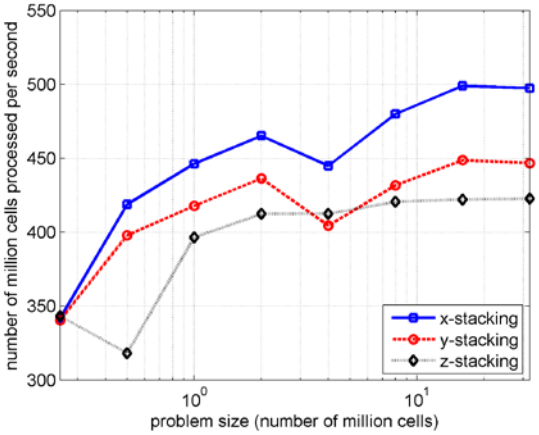


Fig. 3. Throughput of different stacking schemes.

It should be noted that these findings are valid only for the code of the presented *xy*-mapping algorithm and for different codes based on different algorithms the efficiencies due to stacking directions may be different. In any case, an increase in efficiency should be expected if the problem size is increased by stacking.

IV. ALIGNMENT OF GEOMETRIES

The analysis presented in the previous section revealed that it is better to stack the FDTD spatial domains in the *x* direction to achieve the best performance out the presented algorithm. In general a problem space can be in arbitrary size in

different directions; i.e. number of cells in a direction may be different from the number of cells in other directions. It is easy to rotate the geometry such that a side is aligned in any desired direction. For instance, one can align the longest side of a domain in *x*, *y*, or *z* direction by simple transformations.

Since there is a flexibility to align sides in desired directions, one can expect different performances from stacking for different alignments. In order to find which alignment is the best, an alignment test is performed as described below. A base FDTD domain with size of 64 cells on the short side, 128 cells on the medium side, and 192 cells on the long side, as shown in Fig. 4, is prepared. Then this domain is rotated and stacked in the *x* direction for six different alignment scenarios. For instance, Fig. 5 illustrates one of these cases, in which the three copies of the base domain are stacked in the *x* direction such that the short side is aligned in the *x* direction, and the long side is aligned in the *z* direction.

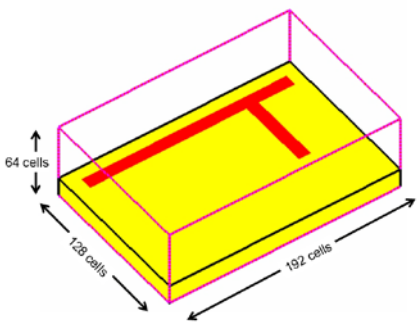


Fig. 4. A problem space with different sizes in different directions.

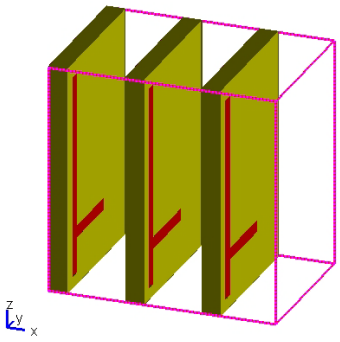


Fig. 5. Base FDTD domain stacked in the x direction such that short side is aligned in the x direction and the long side is aligned in the z direction.

For each of these six cases, first, the base domain is simulated alone and then 20 copies of the base domain are stacked and simulations are repeated. The throughput is calculated for each simulation, and results are tabulated as shown in Table 1. The results reveal that for all of these six cases the efficiencies of the stacked domains simulations are better than that of individual domains simulation. When the efficiencies of the stacked domains simulations are compared, no significant difference has been observed between the six alignments. However, the case in which the shortest side is aligned in the *z* direction, the last row in Table 1, has slightly higher throughput, thus efficiency. It should be reminded that this alignment efficiency analysis is valid for the code of the *xy*-mapping in consideration, and for a different code the results might be different. Nonetheless, alignment directions of geometries shall be taken into consideration to reach the best performance out of the proposed stacking algorithm.

V. OTHER ADVANTAGES OF STACKING

The tests presented in the previous section are performed using a simple FDTD domain with PEC boundaries and a microstrip structure excited by a single voltage source. For some other classes of problems, stacking can provide some other means to achieve better performance in speed as well as memory usage.

One class of problems that can benefit from stacking is scattering due to an incident field. For scattering calculations, the problem space is illuminated by an incident field that has to be recalculated at every time step of time-marching. In an optimization problem, all problem spaces will be excited with the same incident field. Therefore, if incident field is calculated and stored for the base domain, it can be used to excite the other domains in the stack as well. This way, recalculation and storage of fields for separate domains can be avoided and efficiency can be significantly improved both in terms of simulation time and memory.

Another class of problems is the calculation of scattering parameters in a multi-port circuit. For the solution of such problems, in each simulation,

one port is active as a source and scattering parameters are calculated with respect to the active port. Thus for an *N*-port problem, the calculation shall be repeated *N* times. These *N* problem spaces can be stacked and simulated at one run; thus a faster solution can be achieved. Another advantage is that, all FDTD updating coefficients are essentially the same in all of these individual problems. Therefore, it is sufficient to calculate and store the updating coefficients only for a base domain and reuse these coefficients in other domains. Thus efficiency can be achieved in terms of memory use as well.

Table 1. Efficiency of stacking with respect to alignment.

direction of alignment (x, y or z)	short side	medium side	long side	number of stacked domains	stacked domain size (million cells)	number of million cells processed per second
	x	y	z	1	1.6	324
	x	y	z	20	31.5	499
	x	z	y	1	1.6	375
	x	z	y	20	31.5	496
	y	x	z	1	1.6	344
	y	x	z	20	31.5	480
	y	z	x	1	1.6	414
	y	z	x	20	31.5	495
	z	x	y	1	1.6	436
	z	x	y	20	31.5	499
	z	y	x	1	1.6	448
	z	y	x	20	31.5	503

VI. CONCLUSION

The concept of stacking FDTD problem spaces to achieve computation efficiency in terms of solution speed is introduced for optimization and parameter sweep problems on graphics processing platforms. In particular, an FDTD implementation based on CUDA is discussed for GPU platforms and it has been shown that significantly shorter solution times can be achieved if problem spaces are stacked and solved at one run compared to the case where all these problems are solved separately. It has also been shown that, for some classes of problems, stacking can achieve memory efficiency as well.

REFERENCES

[1] K. S. Yee, "Numerical Solution of Initial Boundary Value Problems Involving Maxwell's Equations in Isotropic Media," *IEEE Transactions on Antennas and Propagation*, vol. 14, pp. 302–307, May 1966.

[2] A. Taflove and S. C. Hagness, *Computational Electrodynamics: The Finite-Difference Time-Domain Method*, 3rd edition, Artech House, 2005.

[3] A. Elsherbeni and V. Demir, *The Finite Difference Time Domain Method for Electromagnetics: With MATLAB Simulations*, SciTech Publishing, 2009.

[4] S. E. Krakiwsky, L. E. Turner, and M. M. Okoniewski, "Graphics Processor Unit (GPU) Acceleration of Finite-Difference Time-Domain (FDTD) Algorithm," *Proc. 2004 International Symposium on Circuits and Systems*, vol. 5, pp. V-265–V-268, May 2004.

[5] S. E. Krakiwsky, L. E. Turner, and M. M. Okoniewski, "Acceleration of Finite-Difference Time-Domain (FDTD) Using Graphics Processor Units (GPU)," *2004 IEEE MTT-S International Microwave Symposium Digest*, vol. 2, pp. 1033–1036, Jun. 2004.

[6] R. Schneider, S. Krakiwsky, L. Turner, and M. Okoniewski, "Advances in Hardware Acceleration for FDTD," Chapter 20 in *Computational Electrodynamics: The Finite-Difference Time-Domain Method*, 3rd edition, Artech House, 2005.

[7] S. Adams, J. Payne, and R. Boppana, "Finite Difference Time Domain (FDTD) Simulations Using Graphics Processors," *Proceedings of the 2007 DoD High Performance Computing Modernization Program Users Group (HPCMP) Conference*, pp. 334–338, 2007.

[8] M. J. Inman, A. Z. Elsherbeni, and C. E. Smith, "GPU Programming for FDTD Calculations," *The Applied Computational*

Electromagnetics Society (ACES) Conference, 2005.

[9] M. J. Inman and A. Z. Elsherbeni, "Programming Video Cards for Computational Electromagnetics Applications," *IEEE Antennas and Propagation Magazine*, vol. 47, no. 6, pp. 71–78, December 2005.

[10] M. J. Inman and A. Z. Elsherbeni, "Acceleration of Field Computations Using Graphical Processing Units," *The Twelfth Biennial IEEE Conference on Electromagnetic Field Computation CEFC 2006*, April 30 - May 3, 2006.

[11] M. J. Inman, A. Z. Elsherbeni, J. G. Maloney, and B. N. Baker, "Practical Implementation of a CPML Absorbing Boundary for GPU Accelerated FDTD Technique," *The 23rd Annual Review of Progress in Applied Computational Electromagnetics Society,* 19-23 March 2007.

[12] M. Inman, A. Elsherbeni, J. Maloney, and B. Baker, "Practical Implementation of a CPML Absorbing Boundary for GPU Accelerated FDTD Technique," *Applied Computational Electromagnetics Society Journal*, vol. 23, no. 1, pp. 16–22, 2008.

[13] M. J. Inman and A. Z. Elsherbeni, "Optimization and parameter exploration using GPU based FDTD solvers," *IEEE MTT-S International Microwave Symposium Digest*, pp. 149–152, June 2008.

[14] M. J. Inman, A. Elsherbeni, and V. Demir, "Graphics Processing Unit Acceleration of Finite Difference Time Domain", Chapter 12 in *The Finite Difference Time Domain Method for Electromagnetics (with MATLAB Simulations),* SciTech Publishing, 2009.

[15] N. Takada, N. Masuda, T. Tanaka, Y. Abe, and T. Ito, "A GPU Implementation of the 2-D Finite-Difference Time-Domain Code Using High Level Shader Language," *Applied Computational Electromagnetics Society Journal*, vol. 23, no. 4, pp. 309–316, 2008.

[16] A. Valcarce, G. de la Roche, and J. Zhang, "A GPU Approach to FDTD for Radio Coverage Prediction," *Proceedings of the 11th IEEE Singapore International Conference on Communication Systems (ICCS '08)*, pp. 1585–1590, November 2008.

[17] P. Sypek and M. Michal, "Optimization of an FDTD Code for Graphical Processing Units," *17th International Conference on Microwaves, Radar and Wireless Communications*, *MIKON 2008*, pp. 1–3, 19-21 May 2008.

[18] P. Sypek, A. Dziekonski, and M. Mrozowski, "How to Render FDTD Computations More Effective Using a Graphics Accelerator," *IEEE Transactions on Magnetics*, vol. 45, no. 3, pp. 1324–1327, 2009.

[19] N. Takada, T. Shimobaba, N. Masuda, and T. Ito, "High-speed FDTD Simulation Algorithm for GPU with Compute Unified Device Architecture," *IEEE International Symposium on Antennas & Propagation & USNC/URSI National Radio Science Meeting*, p. 4, 2009.

[20] A. Valcarce, G. De La Roche, A. Jüttner, D. López-Pérez, and J. Zhang,"Applying FDTD to the coverage prediction of WiMAX femtocells," *EURASIP Journal on Wireless Communications and Networking*, February 2009.

[21] D. K. Price, J. R. Humphrey, and E. J. Kelmelis, "GPU-based Accelerated 2D and 3D FDTD Solvers," in *Physics and Simulation of Optoelectronic Devices XV*, *Proceedings of SPIE*, vol. 6468, 2007.

[22] D. K. Price, J. R. Humphrey, and E. J. Kelmelis, "Accelerated Simulators for Nano-Photonic Devices," *International Conference on Numerical Simulation of Optoelectronic Devices 2007, NUSOD '07*, pp. 103–104, September 2007.

[23] A. Balevic, L. Rockstroh, A. Tausendfreund, S. Patzelt, G. Goch, and S. Simon, "Accelerating Simulations of Light Scattering Based on Finite-Difference Time-Domain Method with General Purpose GPUs," *Proceedings of the 2008 11th IEEE International Conference on Computational Science and Engineering*, pp. 327–334, 2008.

[24] C. Ong, M. Weldon, D. Cyca, and M. Okoniewski, "Acceleration of Large-Scale FDTD Simulations on High Performance GPU Clusters," *2009 IEEE International Symposium on Antennas & Propagation & USNC/URSI National Radio Science Meeting*, 2009.

[25] NVIDIA CUDA ZONE: www.nvidia.com/object/cuda_home.html.

[26] Acceleware: www.acceleware.com.
[27] CUDA_Getting_Started_2.3_Windows.pdf: http://www.nvidia.com/object/cuda_develop.html.
[28] http://en.wikipedia.org/wiki/CUDA.
[29] V. Demir and A. Z. Elsherbeni, "Compute Unified Device Architecture (CUDA) Based Finite-Difference Time-Domain (FDTD) Implementation," *Journal of the Applied Computational Electromagnetics Society (ACES)*, vol. 25, no. 4, 2010.

Veysel Demir is an assistant professor at the Department of Electrical Engineering at Northern Illinois University. He received his Bachelor of Science degree in electrical engineering from Middle East Technical University, Ankara, Turkey, in 1997. He studied at Syracuse University, New York, where he received both a Master of Science and Doctor of Philosophy degrees in electrical engineering in 2002 and 2004, respectively. During his graduate studies, he worked as a research assistant for Sonnet Software, Inc., Liverpool, New York. He worked as a visiting research scholar in the Department of Electrical Engineering at the University of Mississippi from 2004 to 2007. He joined Northern Illinois University in August 2007.

Dr. Demir's main field of research is electromagnetics and microwaves. He is especially experienced in applied computational electromagnetics. He heavily participated in the development of time domain and frequency domain numerical analysis tools for new applications and contributed to research on improving the accuracy and speed of algorithms being developed. He is experienced in designing RF/microwave circuits and antennas for the related technologies, and performing experimental characterizations of these devices.

Dr. Demir is a member of IEEE and ACES and has coauthored more than 20 technical journal and conference papers. He is the coauthor of the books *Electromagnetic Scattering Using the Iterative Multiregion Technique* (Morgan & Claypool, 2007) and *The Finite Difference Time Domain Method for Electromagnetics with MATLAB Simulations* (Scitech 2009).

Accelerating Multi GPU Based Discontinuous Galerkin FEM Computations for Electromagnetic Radio Frequency Problems

Nico Gödel [1], Nigel Nunn, Tim Warburton [2], and Markus Clemens [3]

[1] Faculty of Electrical Engineering
Helmut-Schmidt-University, University of the Federal Armed Forces Hamburg,
P.O. Box 700822, D-22008 Hamburg, Germany
Nico.Goedel@hsu-hh.de

[2] Computational and Applied Mathematics
Rice University, 6100 Main Street MS-134, Houston, TX, USA

[3] Bergische Universität Wuppertal, FB E, Chair for Electromagnetic Theory
Rainer-Gruenter-Str. 21, D-42119 Wuppertal, Germany

***Abstract*—**A Graphics Processing Unit (GPU) accelerated simulation of Maxwell's equations in the time domain is presented. The Discontinuous Galerkin Finite Element Method (DG-FEM) is used for discretization since the elementwise structure fits the parallelization design aspects of the GPU architecture and the NVIDIA Compute Unified Device Architecture (CUDA), a GPU programming model. The parallelization strategy for a multi-GPU setup using CUDA is focused. Several performance improvements are analyzed and investigated with the help of a realistic 3D electromagnetic scattering example.

***Index Terms*—**GPU-Computing, GPGPU, DG-FEM, electromagnetics, CUDA, TESLA.

I. INTRODUCTION

Numerical simulation of electromagnetic devices during the development and certification process can significantly reduce time, efforts and costs. Efficiency and costs of numerical simulations depend on hardware and software investments as well as on personnel expenses, which directly evolve from code performance and simulation time. The presented hardware accelerated approach is able to significantly reduce both, simulation time and hardware costs using consumer based GPUs instead of highly expensive large scale computing clusters.

Hardware accelerated computation is not a new research domain, but recently gained attention due to the availability of high-level compute abstractions such as CUDA [1], BROOK+ [2] and OPENCL [3]. Furthermore, floating-point performance and device memory bandwidth of current consumer based GPUs exceed their CPU counterparts by more than one order of magnitude at approximately the same price per unit.

The combination of these GPU based co-processing units and the evolving programming models provide a significant potential for high-performance related computations.

This potential has been investigated for different volume based discretization methods, such as the Finite Difference Method [4, 5] and the Finite Integration Technique. Recently, the Discontinuous Galerkin Finite Element Method (DG-FEM) has gained attention in connection with GPU computations [6, 7].

In this paper, the parallelization model and several optimization techniques for DG-FEM computations on GPU-clusters will be investigated. The focus will be on the scalability of multi-GPU systems.

The paper is organized as follows: In Section II, the model is defined stating both the governing differential equations for the electric and the magnetic field and the spatial discretization using DG-FEM. Subsequently, in Section III, the DG-FEM discretization is investigated with respect to the suitability of a parallel implementation. In

Section IV, the parallelization model for three different abstraction layers as well as the hardware in use is presented and Section V provides numerical results for the different types of implementations with the use of a complex example.

II. DESCRIPTION OF THE MODEL

A. Maxwell's Equations

Electromagnetic wave propagation in lossless medium can be described using Ampère's law together with Faraday's law of induction

$$\frac{\partial}{\partial t}\varepsilon\vec{E} = \nabla \times \vec{H} \quad (1)$$

$$\frac{\partial}{\partial t}\mu\vec{H} = -\nabla \times \vec{E} \quad . \quad (2)$$

Here, $\vec{E}$ and $\vec{H}$ denote the electric and magnetic field strength, respectively, whereas ε and μ identify the electric permittivity and the magnetic permeability. The right-hand sides (RHS) of (1) and (2) describing the spatial dependencies can be discretized with help of the Nodal Discontinuous Galerkin method.

B. Discontinuous Galerkin Discretization

DG-FEM was first introduced by Reed and Hill in 1973 [8] for neutron transport simulation and during the last decade, this method was intensively investigated for solving Maxwell's equations. Relevant results have been published especially by Cockburn et. al. [9], Cohen et. al. [10] and by Hesthaven and Warburton [11, 12].

The DG method was chosen due to some important characteristics, i.e.
(1) the treatment of complex geometry through unstructured tetrahedral meshes,
(2) explicit time stepping schemes, especially multirate time stepping schemes,
(3) the use of high order basis functions and
(4) a domain decomposition approach, which is intrinsically included in the DG formulation.

Along with these features, the caveats are restricted to geometry-dependent time steps and an overhead in degrees of freedom at the element faces compared to continuous FEM as well as not providing a strictly conservative model of electric charges.

A Nodal DG discretization of eqn. (1) and (2) is derived in [12] using Lagrange polynomials as basis functions. The main characteristic of DG-FEM is that it allows for an elementwise computation of the elements, i.e. each element is computed separately. The semi-discrete scheme, still continuous in time, for each element reads

$$\frac{\mathrm{d}}{\mathrm{d}t}\varepsilon\mathbf{E} = \mathbf{M}^{-1}\mathbf{S}\,\mathbf{H} - \mathbf{M}^{-1}\mathbf{F}\cdot[\hat{\mathbf{n}}\cdot(\mathbf{f}_{\mathsf{E}} - \mathbf{f}^{*}_{\mathsf{E}})] \quad (3)$$

$$\frac{\mathrm{d}}{\mathrm{d}t}\mu\mathbf{H} = -\mathbf{M}^{-1}\mathbf{S}\,\mathbf{E} + \mathbf{M}^{-1}\mathbf{F}\cdot[\hat{\mathbf{n}}\cdot(\mathbf{f}_{\mathsf{H}} - \mathbf{f}^{*}_{\mathsf{H}})] \quad . (4)$$

Here, $\mathbf{M}^{-1}$ is the inverse of a local mass-matrix, **S** a local stiffness-matrix and **F** a local flux-matrix. The size of the matrices depends on the number of nodes inside each element. The size of the symmetric matrices **M** and **S** is listed in Table 1 for different polynomial orders. The term $\mathbf{M}^{-1}\mathbf{S}$ refers to a local differentiation in the element without the need for using a custom elementwise mass-matrix for every single element.

The flux-matrix **F** refers to the integration over every triangular face of each tetrahedron. The size of the flux matrix **F** depends on the number of nodes inside the element and the number of nodes on all surfaces of the element. The flux terms f_E and f_H in eqn. (3) and (4) refer to flux density terms of adjacent face values on each face.

III. PARALLEL STRUCTURE OF THE MODEL

To allow for efficient parallelization on GPUs as well as on CPUs, the computation has to be split up into small pieces of work. Ideally, each piece of work has a completely independent data structure, thus requires no memory interaction with other parts. However, treating hyperbolic problems, it seems clear that there has to be communication, at least for neighboring elements, to resolve the propagation of electromagnetic waves. The idea of the DG-FEM and the proposed implementation is to minimize and encapsulate all dependencies and to efficiently handle communication with help of special GPU features.

As described in section II.B., the first operation of the RHS, referring to the curl computation on Maxwell's equation, is computed inside each element. Here, every element can be computed completely independent from each other, providing the opportunity of a highly parallel implementation. The second term in the

RHS has to be treated more carefully. In the flux term evaluations, the jumps in the electric and magnetic fields are integrated over the elements faces. The integration itself, being numerically expensive, is computed for each element separately. The jump computation however needs data from adjacent elements. This computation has to be treated in a special way regarding elementwise, highly parallel implementations and will be presented in Section IV.C.

IV. HARDWARE, PROGRAMMING MODEL AND PARALLELIZATION STRATEGY

A. TESLA S1070 GPU Server

For the GPU computations, a NVIDIA TESLA S1070 with four GPUs and 4 GB GDDR3 RAM each is used. Each GPUs consists of 30 Multiprocessors (MP) with 8 Streamingprocessors (SP) each. The GPU server is attached to the host server using two PCIe x16 Gen2 connections.

B. Coarse Grained Parallelism

Regarding GPU-cluster computations, the most coarse grained parallelization can be realized using a METIS [13] domain decomposition of the computational domain as presented in Fig. 1.

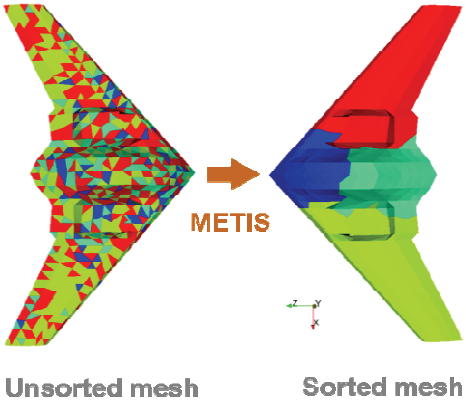


Fig. 1. Domain decomposition using METIS graph partitioning.

Each color maps the tetrahedral elements to one GPU. The METIS domain decomposition can be executed using two constraints: load balancing and minimization of communication between the subdomains. In [7], an algorithm using KMETIS was presented, providing good minimization of the surface, but not optimal load balancing for this small amount of subdomains. In this work, PMETIS being more suitable for a small amount of subdomains is used. The resulting performance improvements are presented in Section V.

Each METIS subdomain is computed on one GPU. For the flux computation presented in Section III, field data of neighboring elements across METIS boundaries have to be provided to the corresponding GPU. Since this communication has to be carried out as a CPU task via the PCIe bus, the data exchange is supposed to be a potential bottleneck of the presented approach. To minimize the effects of the latency and the bandwidth of the PCIe bus, two aspects should be ensured:

1. Only the data which are needed by another GPU should be transferred, and only to this specific device.
2. The data transfer should be hidden behind other computations with help of asynchronous file transfer.

The second part is one of the key aspects regarding highly scalable code on multi-GPU systems. On NVIDIA TESLA GPU devices with compute capability 1.1, the possibility of simultaneous execution of kernel functions and host-device/device-host memory transfer is introduced. A kernel can be executed concurrently to a data transfer to or from the host. This is applicable as long as the kernel does not depend on the transferred data. Regarding the presented DG-FEM implementation, the kernel function evaluating the curl operator inside each element can be executed while field data of adjacent METIS subdomain boundaries is transferred. With this feature, the scalability bottleneck can be expanded.

C. GPU Based Block Parallelism

On each GPU, one METIS subdomain is computed. All data (fields, geometry and operators) related to this subdomain are stored in the device memory and stays in the device memory for the entire simulation. Using CUDA as a programming model, the computational work is arranged in a CUDA GRID. This GRID consists of CUDA BLOCKS, each having the same amount of computational work. This data management is

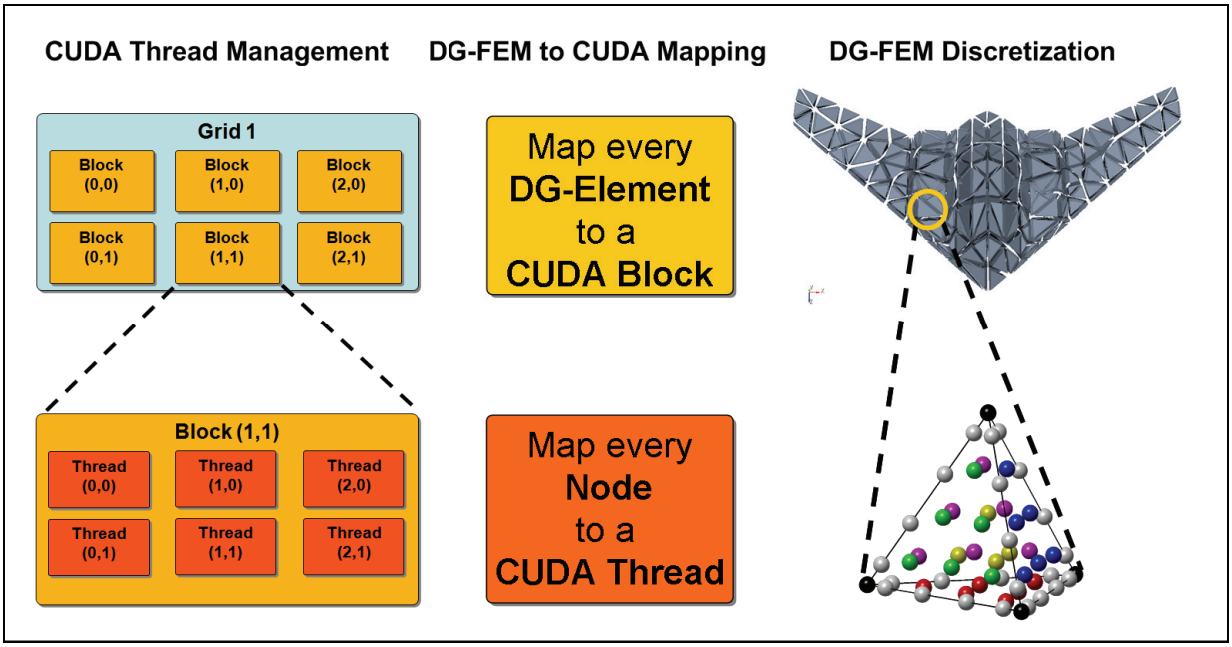


Fig. 2. Correlation between DG-FEM discretization and CUDA data management.

pictured in Fig. 2, highlighting the proposed strategy of implementing DG simulations in CUDA. Instead of conventional vector based implementations, branching is possible within this Single Instruction Multiple Thread (SIMT) architecture. CUDA BLOCKS cannot communicate with each other during their execution. One BLOCK is executed on a MP, where it profits from the high-speed, low-latency shared memory space within each MP. Global memory fetches have to be executed in a coalesced way, avoiding multiple read operations on a single address. In the case that coalesced reads cannot be ensured, read conflicts are serialized with the effect of several hundreds of cycles latency for each conflict. In this case, the use of texture memory is beneficial, providing buffered memory access at almost the same bandwidth as coalesced global memory fetches.

As long as all operations are done locally within each element, coalesced reads can be ensured for the fields. However, when calculating the curl operator in the RHS, spatial derivatives in the reference element have to be provided for all the elements. Here, coalesced reads cannot be ensured and the use of texture buffered memory access provides higher performance as presented in section V.

Regarding flux computations, for the evaluation of field differences of the triangular faces, each field value at the faces will be read more than one time, leading to serialized memory fetches. This read conflict is less severe than the one earlier presented since the number of read conflicts depends on the maximum number of elements, a vertex is connected to. However, also in this case, the use of texture memory is preferable.

Figure 2 highlights that the GRID – BLOCK decomposition within the CUDA model is reflecting the geometric discretization of each METIS subdomain with help of finite elements, here tetrahedral elements. The idea of the proposed approach is to map every finite element to a CUDA BLOCK.

D. Fine Grained GPU Thread Based Parallelism

The lowest level of abstraction within the parallelization strategy is formed by the threads managed by each MP for the computation of each BLOCK. Each MP is able to manage several hundreds of threads at the same time to feed the 8 SP. The instruction unit of each MP can process one instruction every 4 cycles. Therefore a set of

32 CUDA THREADS form a WARP, the smallest scheduling unit.

Since the solution in each DG element will be approximated with nodal basis functions, the nodes inside each element reflect the degrees of freedom for each field component. Fig. 2 highlights the proposed strategy, which maps the nodes, i.e. the degrees of freedom inside each element to the CUDA THREADS. The number of nodes Np depends on the order N of the polynomial basis functions with Np = (N+1)(N+2)(N+3)/6 (see Table 1). Since the number of nodes does not match a multiple of a single WARP, some THREADS do not contribute to the computation. A strategy for improving the efficiency using these "padding" threads can be found in [6], where several element are grouped together for a low polynomial order N.

Table 1: Polynomial order and number of nodes.

N	Np
1	4
2	10
3	20
4	35
5	56
6	84
7	120
8	165

In the code presented in this work, the number of threads per CUDA BLOCK is defined as the number of nodes in one element.

V. NUMERICAL INVESTIGATIONS

In this section, the effects of several CUDA related optimization techniques are investigated. The accuracy of the presented approach is presented in [7]. In this work, more challenging geometries are considered. As application, an electromagnetic scattering object as presented in Figs. 1 and 2 is used.

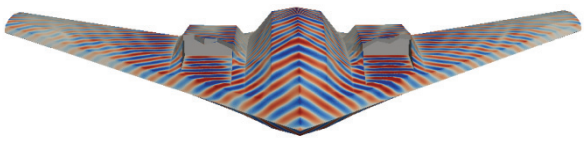


Fig. 3. Ey surface field component of a scatterer hit by a TEM wave.

A TEM wave with 0.5 GHz is used as excitation function. The electrical surface field component in y-direction is presented in Fig. 3. The object is treated as perfect electric conductor and is surrounded by vacuum medium which is enclosed by an absorbing layer. The domain is discretized with 143936 tetrahedra. Unless otherwise noted, the simulations were carried out using 6^{th} order polynomials leading to $7.2 \cdot 10^7$ unknowns and 5362 timesteps using the Low Storage Explicit Runge-Kutta (LSERK) scheme. On a single GPU, 2.02 GB memory is needed for this configuration.

A. Texture Buffered Memory Fetches

In this subsection, the effects of texture buffered memory fetches are investigated. As described in Section IV.C, the use of texture memory can be beneficial whenever coalesced reads cannot be ensured. The most drastic performance increase has been encountered within the curl computation of (3) and (4). Here, local derivatives in the reference element have to be provided for all elements / CUDA BLOCKS. The presented approach uses texture memory to buffer the operator and shared memory to buffer the field data.

Within the flux computation, the evaluation of the field differences at the faces cannot be realized coalesced, however, the number of conflicts is small. Here, the caching of field data through texture memory is analyzed.

On a single GPU, using texture fetches for the derivatives in the curl computation yields a performance improvement of a factor of 2.69, as presented in Table 2.

Table 2: Performance gain with help of texture buffered memory fetches on a single GPU.

Implementation	Performance [GFlops]	Speedup
Without TEXTURE usage	79.3	1.0
TEXTURE usage for curl computation	213.2	2.69
TEXTURE usage for flux computation	78.9	0.995
TEXTURE usage for curl and flux computation	209.3	2.64

For the flux computation, the expected performance improvement cannot be produced. In contrast, texture buffered memory fetches are slightly less efficient than their global memory counterparts. In comparison to the former G92 core, where performance increases of 6% were encountered, NVIDIA seems to have improved the global memory access. In [1], the constraints for using coalesced memory access are weakened compared to earlier documentations, which might be one reason for the observed changes in different architectures.

To summarize, the use of the texture units for buffered memory access can speed up the implementation whenever a multiplicity of read conflicts occur. In case that the number of read conflicts is small compared to the whole data volume transferred, global memory fetches can profit from their larger bandwidth.

B. Performance and Scalability of Multi-GPU Computations

In this subsection, performance and scalability of multi-GPU computations using the four GPUs of a TESLA S1070 server are investigated. CPU computations were carried out on four AMD quad-core Opteron CPUs with 2.3 GHz. The price for the S1070 is about 2900€ (academic pricing) compared to 16.759€ of the latest HP Proliant DL785G CPU server.

In Fig. 4, a comparison of GPU and CPU performance for different polynomial orders is presented.

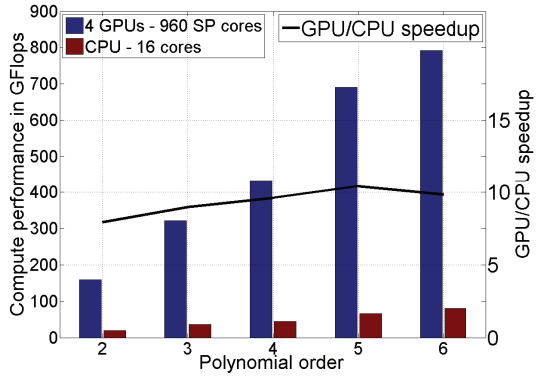


Fig. 4. Performance of GPU and CPU computations for different polynomial orders.

All computations were carried out using the IEEE-754 single precision floating point standard. GPU performance is about ten times higher than the corresponding CPU implementation.

In Fig. 5, the scalability of multi-GPU computations is presented using different METIS distributions. Furthermore, the effect of asynchronous file transfer is highlighted. For the scalability evaluation, a polynomial order of 5 has been chosen. With help of the perfectly load balanced PMETIS distribution and asynchronous file transfer, a strong scalability of 98.8 % is achieved. Due to the asynchronous file transfer, almost the complete communication overhead could be hidden behind the arithmetic curl computation which needed 15ms time in average in contrast to the communication which required 3ms in average. Except for a numerically cheap packaging of the transferred data, the presented solution incorporated zero communication overhead. The difference in parallel communication and curl computation time yields potential for further parallelization with eight GPUs at the same high degree of scalability.

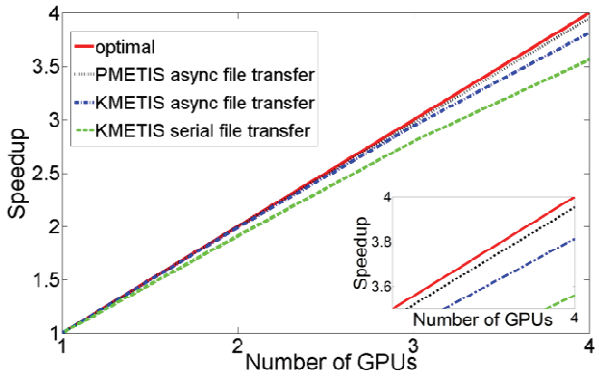


Fig. 5. Scalability of multi-GPU DG computations.

Strong scalability in this case means that the global problem size does not change when increasing the number of GPUs. However, according to [6], a minimum work of approximatively 10000 elements per GPU should be provided to get the full floating point performance. With the 143936 elements of the scattering example, the problem could be distributed on more GPUs without losing much of the efficiency.

The METIS distributions are presented in Fig. 6 highlighting the difference in load balancing for the four subdomains.

The PMETIS distribution is perfectly balanced whereas the KMETIS distribution leads to unbalanced workload, where thread number 2 has to do more work than the other threads.

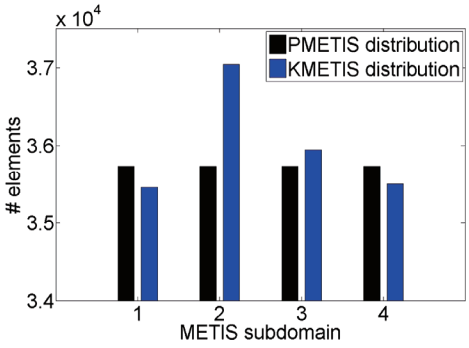


Fig. 6. Partitioning of PMETIS and KMETIS algorithms for four subdomains.

VI. CONCLUSION

A CUDA based GPU implementation of Maxwell's equations in the time domain was presented. The matching modeling principles of DG-FEM and CUDA regarding parallel implementation were highlighted and several optimization techniques have been investigated. As key point to high performing implementation, a detailed memory access concept is necessary to profit from the high floating point performance of the GPUs.

Almost perfect scalability up to four GPUs was presented using the asynchronous file transfer feature to hide all inter-GPU communication behind the curl kernel execution which does not depend on the data.

Further work will include scalability tests for up to 8 GPUs, as well as hybrid CPU/GPU implementations and porting to upcoming architectures like GT300 and new compute abstractions like OpenCL.

ACKNOWLEDGMENTS

T. Warburton is partly supported by AFOSR FA9550-05-1-0473, NSF CNS-0514002 and NSF DMS-0512673. N. Gödel is partly supported by DFG travel grant CL 143/8-1.

The authors would like to thank Andreas Klöckner and Jeff Bridge for discussions and support related to DG implementation on GPUs.

REFERENCES

[1] Nvidia Corporation. "NVIDIA CUDA 2.2 Compute Unified Device Architecture Programming Guide", USA, 2009.

[2] Advanced Micro Devices, Inc., "ATI Stream Computing", Sunnyvale, USA, 2009.

[3] KHRONOS GROUP, "The OpenCL Specification Version 1.0", 2008.

[4] S. Krakiwsky, L. Turner, and M. Okoniewski, "Acceleration of Finite-Difference Time-Domain (FDTD) Using Graphics Processor Units(GPU)", *IEEE MTT-S International Microwave Symposium*, pp. 1033-1036, 2004.

[5] M. J. Inman, A. Z. Elsherbeni, J. G. Maloney and B. N. Baker, "Practical Implementation of a CPML Absorbing Boundary for GPU Accelerated FDTD Technique", *ACES Journal*, vol. 23, 2008.

[6] A. Kloeckner, T.Warburton, J. Bridge, and J. Hesthaven, "Nodal discontinuous Galerkin methods on graphics processors," *Journal of Computational Physics,* vol. 228, pp. 7863 – 7882, 2009.

[7] N. Gödel, T. Warburton, M. Clemens, "GPU Accelerated Discontinuous Galerkin FEM for Electromagnetic Radio Frequency Problems", *IEEE APS Conference Charleston,* 2009.

[8] W. Reed and T. Hill, "Triangular mesh methods for the neutron transport equation," *Los Alamos Scientific Laboratory, vol. Tech. Report*, no. LAUR-73-479, 1973.

[9] B. Cockburn, G. Karniadakis, and C.-W. Shu, "Discontinuous Galerkin Methods: Theory, Computation and Applications", *Springer,* 2000.

[10] G. Cohen, X. Ferrieres and S. Pernet, "A spatial high-order hexahedral discontinuous Galerkin method to solve Maxwell's equations in time domain", *Journal of Computational Physics*, vol. 217, pp. 340-363, 2006.

[11] J. S. Hesthaven and T. Warburton, "Discontinuous Galerkin methods for the time-domain Maxwell's equations," *ACES Journal,* vol. 19, pp. 10–29, 2004.

[12] J. S. Hesthaven and T. Warburton, Nodal Discontinuous Galerkin Methods. Springer, 2008.

[13] G. Karypis and V. Kumar, "A fast and high quality multilevel scheme for partitioning irregular graphs." Conference on Parallel Processing, pp. 113-122, 1995.

Nico Gödel, born 1978 in Minden, Germany, received his diploma in Electrical Engineering from the Helmut-Schmidt University, University of the Federal Armed Forces in Hamburg in 2006. From 2007 till 2010, he worked as a Research Engineer at the Chair for Theory in Electrical Engineering and Computational Electromagnetics at the Helmut-Schmidt-University, University of the Federal Armed Forces Hamburg.

Tim Warburton received a PhD in Applied Mathematics from Brown University in 1999. He is currently an Associate Professor of Computational and Applied Mathematics, at Rice University, Houston, Tx.
He co-authored the first major text on discontinuous Galerkin methods, published by Springer in 2008.

Markus Clemens, born 1968 in Wittlich, received his diploma in Mathematical Engineering ("Diplom Technomathematik") with a minor in Mechanical Engineering from the University of Kaiserslautern in 1995. In 1998 he finished his Phd at the Institute for Theory of Electromagnetic Fields at the Technische Universität Darmstadt in the field of Computational Electromagnetics. Working as postdoc at the same institute he became team leader of an interdisziplinary team of phd and postdoc researchers. In December 2003 he received his venia legendi in "Electromagnetic Theory" and "Scientific Computing". From 2004 to 2009 he was working as head of the Chair for Theory in Electrical Engineering and Computational Electromagnetics at the Helmut-Schmidt University, University of the Federal Armed Forces Hamburg. In October 2009 he took on the position as head of the Chair of Electromagnetic Theory at the Bergische Universität Wuppertal, Germany. His teaching activities involve courses in Electromagnetic Theory, Advanced Engineering Mathematics, and Computational Electromagnetics. His research activities are in the field of Computational Engineering and Mathematical Engineering. His research specifically involves the development and application of numerical simulation methods for Computational Electromagnetics and Computational Multiphysics.

CUDA Based LU Decomposition Solvers for CEM Applications

Matthew J. Inman[1], Atef Z. Elsherbeni[1], and C. J. Reddy[2]

[1] Department of Electrical Engineering
University of Mississippi, University, MS 38677-1848, USA
atef@olemiss.edu , mjinman@olemiss.edu

[2] Applied EM
Hampton, VA 23666, USA
cjreddy@emssusa.com

Abstract ─ The use of graphical processing units to perform numerical computations required by electromagnetic analyses have been shown over the past several years significant increase in the computational speed. Most of the previous work concentrated on electromagnetic analyses that do not require matrix inversion. This paper uses the NVIDIA's compute unified device architecture (CUDA) language to develop and modify routines for matrix solution based on the LU decomposition procedure to enhance and speed up a class of electromagnetic simulations. This implementation is utilizing the CPU and GPU for the inversion procedure. Various implementations for real, complex, single precision and double precision will be examined. The performance details of the developed LU decomposition routines especially for complex and double precision arithmetic are presented.

Index Terms ─ CUDA, GPU, CEM, LU Decomposition, Matrix Solvers.

I. INTRODUCTION

As computational power has increased exponentially over the past few decades, the need for solving complex systems of equations has grown equally in tandem. Even simple geometries can often lead to complex matrices whose size can easily be in the order of thousands. In order to accurately and quickly provide results from these simulations an appropriate solution method must be chosen. This paper will address the use of graphical processing units (GPU's) based LU decomposition solvers for matrix solutions.

The LU decomposition offers many advantages for solving dense matrices. Full inversion methods (such as Gaussian elimination) can allow for the solving of many right hand sides easily once the inversion is complete. However, full inversions often require large computational runtimes compared with LU decomposition. Many parts of LU decomposition lend itself well to implementation on the GPU due to its past widespread use on other various parallel computing systems [2-6].

Using the NVIDIA compute unified device architecture (CUDA) interface, many of the computations required for LU decomposition can be offloaded to the GPU. While LU decomposition on the GPU has previously been demonstrated to outperform the CPU [3-4], past published work has been mainly limited to real matrices in single precision. In order for LU decomposition to be of widespread use in computational electromagnetics (CEM), any GPU implementation must be able to support complex values. Large matrixes will also require double precision support in order to maintain stability.

In this paper the construction of LU decomposition solver on the GPU is performed using existing and newly developed routines. While many of the subroutines used in LU

decomposition can run on the GPU faster than the CPU, some portions of the code are still more appropriate to run on the CPU [2-3]. Maintaining data integrity between CPU and GPU for complex double precision numbers must be established. The inclusion of double-precision calculations will also be examined from a memory standpoint in optimizing the local cache memory in the GPU to achieve the fastest execution possible.

II. Data Types and Computations Efficiency

It is widely known that different data types can have a large effect on the computational runtime for any algorithm. For instance, going from any real data type to a complex one not only doubles the amount of memory required to move and store, but the complexity of even simple arithmetic operation increases by a significant amount. Complex addition and subtraction requires two separate additions or subtractions. Multiplication requires one addition, one subtraction, and four multiplications. Division requires eight multiplications, three additions, one subtraction, and two divisions. This increase in complexity for complex numbers can have major effects on the runtime of any algorithm.

In addition, the change from single to double precision calculations can have a likewise effect on performance. The double precision performance of the NVIDIA Tesla C1060 is almost 12 times slower than single precision. An Intel Core i7 CPU has double precision speed only 1.4 times slower. This major discrepancy is due to both the maturity of the arithmetic hardware and how this hardware is implemented. Double precision support on NVIDIA GPU's are only a single generation old and are implemented by combining multiple single precision units together to create a double precision unit. Future generation of NVIDIA Fermi GPU's are expected to have better double precision support according to the vendor information that are about to be released.

In this paper we will consider various aspects in the comparison between solvers utilizing different data types. The amount of data to be transferred and stored in system, the increase in computations required for complex number, and the efficiency of double precision calculations will be taken into account. Comparisons will address all these issues within the results.

III. LU Decomposition Solvers in CUDA

The LU decomposition has been previously demonstrated on the GPU using CUDA and other programming techniques for single precision real matrices [3-4]. Published result produced speed gains approaching an order of magnitude over common CPU's. These solvers mixed a combination of CPU Basic Linear Algebra Subprograms (BLAS) calls, CUDA CUBLAS (NVIDIA's GPU based BLAS libraries) calls, and CUDA kernel. The BLAS libraries contain highly tuned functions commonly used in many programs to perform basic linear algebra. The published LU solvers were facilitated by the complete and mature development of CUBLAS libraries for single precision real data types. These solvers showed a speed increase of 6 to 12 times (relative to various hardware). However, the restriction of single precision real data types limits its usefulness for CEM simulations. Many common CEM problems require the solver to be available for any combination of single precision, double precision, real, and complex data.

The development of solvers that support data other than a real single precision on the CUDA/GPU platform presents several unique challenges to be addressed. These challenges occur from the status of the CUBLAS libraries. The CUBLAS libraries (previous to release 3.0) only supported complete BLAS routines in single precision real and only very limited support for single and double precision complex. In the utilized version 2.0 of CUBLAS for this paper, only 2 out of 13 level 1 BLAS routines, 1 out of 16 level 2 BLAS routines, and 2 out of 6 level 3 BLAS routines were supported. The CUBLAS version 3.0, recently released, claims full support for all BLAS routines in all data types.

With the release of CUBLAS 3.0 it is now possible to perform the LU decomposition directly on the GPU without the aid of any CPU calls.

However, this does not mean that the CUBLAS functions outperform their CPU based counterparts. Certain linear algebra functions still perform significantly faster (such as factorization) on the CPU compared to the GPU's as utilized in this paper. The algorithm presented here was carefully profiled to determine when and which parts of the LU decomposition routine can be solved on the GPU with maximum efficiency.

The real single precision solver presented here follows the published methodology of utilizing both the CPU and the GPU as in [3-4] and the established algorithms for parallel computing systems [5]. The code has been programmed and tuned by the authors using these methods. In order to extend this solver for other data types, some of the CUBLAS calls have been replaced with custom developed kernels (GPU functions).

In the solvers presented here, the "*trsm" function which is a standard BLAS routine used to solve a triangular matrix, has been offloaded to the CPU. The transpose functions have been developed in CUDA to support all types of data (complex and real in single and double precisions). With this added support for the various data types, the developed GPU code was tuned for various block sizes which determines how much data gets transferred, at a time, between the GPU and CPU. Offloading the "*trsm" function back to the CPU also presents problems in maintaining data consistency. The transfer of data between CUBLAS on GPU and Intel MKL BLAS on CPU is simple when working with single (float) or double precision real numbers. However, for complex data, MKL BLAS and CUBLAS have different data types and data structures to represent the numbers. In order to accomplish consistent data transfer, the MKL BLAS has been modified so that its data structure is compatible with CUBLAS data types. This modification allowed the free exchange of data between CUBLAS on the GPU and MKL BLAS on the CPU for complex numbers.

The custom routines in CUDA for transposition and pivoting were developed to support all combinations of data types. Depending on the data type needed, the additional data overhead requires smaller blocks of the matrix to be transferred at a single time (as a double precision complex matrix has 4 times the data as a single precision real matrix). The transpose routines make use of local cache memory inside the GPU in order to make this process as efficient as possible.

Table 1 details the various functions used for the developed CPU+GPU based LU decomposition and where they are performed. The basic algorithm iterates through the various block columns of the matrix and performs the decomposition as detailed in [5]. Each block is first transposed and the L/U matrices are updated on the GPU. The block is transferred to the computer system and factorization takes place on the CPU. The block then streams through the GPU for pivoting and back to the CPU. The block is then inverted and the L matrix is solved. The update for the U matrix is performed on the GPU, then the data is transferred back and the final U solve is done on the CPU.

Table 1: Functions required for LU decomposition

Transpose Block	GPU (CUDA Kernel)
Matrix Multiply	GPU (CUBLAS)
Factorization	CPU (MKL BLAS)
Pivot	GPU (CUDA Kernel)
Triangular Matrix Solve	CPU (MKL BLAS)

Each of the functions listed in Table 1 can be implemented on either the CPU or the GPU. For the factorization and the matrix solve routines, the CPU was more efficient in processing even with the added overhead of transferring the data. Both of these functions are not easily parallelized which explains why they are more efficiently performed on the CPU. The transpose and pivoting functions were written in CUDA and optimized for each data type and block size. This necessitated writing separate CUDA functions for each separate data type in order to maintain the highest processing speed possible.

IV. LU Solver Results

The developed CUDA based LU solver was implemented on different systems for various data types. Similarly a pure CPU solver based on the Intel MKL library was used for all comparisons. The solvers were run on various CPU and CPU+GPU based configurations as detailed in Table 2. In all cases, the Intel MKL library uses all available cores on a CPU (2 cores on Core Duo, and 4 cores on i7).

Table 2: System configurations

System	
System 1	2.66 GHz Intel Core i7 6GB DDR3 PC12800 NVIDIA 280GTX 1GB NVIDIA Tesla C1060 4GB
System 2	2.4 GHz Intel Core Duo 4GB DDR2 PC4700 NVIDIA 8800GTX 768 MB

Figure 1 shows the runtime results for the first case of single precision real data for CPU and CPU+GPU implementations on various systems. This baseline case matches other published results [3] in runtimes and speed gain. The CPU+GPU implementations outperformed the CPU only implementation anywhere from 3 to 12 times based on the configuration of the CPU and GPU.

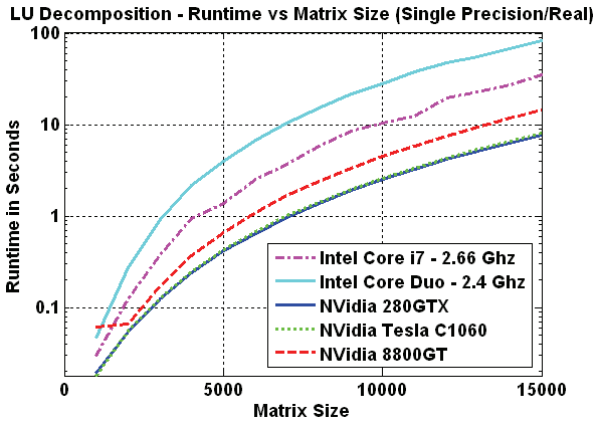


Fig. 1. Runtimes for real single precision LU decomposition.

In the real single precision case, the implementation is quite simple and the best speed gain can be realized. When the solver is expanded to double precision, the results show a moderate decrease in speed for all the available cases as seen in Figure 2. Only the NVIDIA 280 and Tesla C1060 support GPU based double precision and thus are shown here. The Intel Core i7 is the CPU for both the CPU and CPU+GPU cases in this figure. For this real double precision case, the CPU only implementation increased the runtime speed by roughly double across all the various matrix sizes, while the CPU+GPU implementation increased runtime by only around 90% over the single precision case.

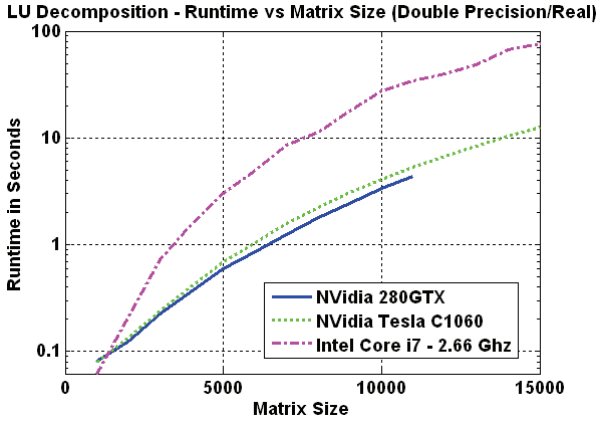


Fig. 2. Runtimes for real double precision LU decomposition.

In the real double precision cases, the CPU+GPU implementation achieved a speed gain of seven times over the CPU only based counterpart. Interestingly, even though twice the amount of data is required to be moved for a double precision case and known inefficiencies of the GPU processing double precision data, the CPU+GPU case only increased runtime by 90%. This can be explained by examining the memory access patterns in processing double precision data. In algorithms such as LU decomposition, data access to the memory of the CPU and GPU are not optimal for the fastest transfer. The addition of double precision data in these cases actually increase the efficiency of memory access since larger blocks of linear memory is being read at a single time. The addition of double precision arithmetic for these cases did not account for any

noticeable increase in processing time. This is due to the fact that in these cases the arithmetic is fairly simple. The calculations were completed before the next block of data has arrived from memory even with the overhead of double precision calculations.

The last implementation presented is the complex double precision case. Figure 3 shows the runtimes for various configurations. With the addition of complex numbers, the runtimes have slowed significantly over the real single precision cases. The CPU only implementation runs approximately 9 times slower while the CPU+GPU implementation runs approximately twenty times slower. It can still be seen that in all cases the CPU+GPU implementation still outperforms the CPU only implementation by approximately two (2.66 Intel Core i7) to four (2.4 GHz Core Duo).

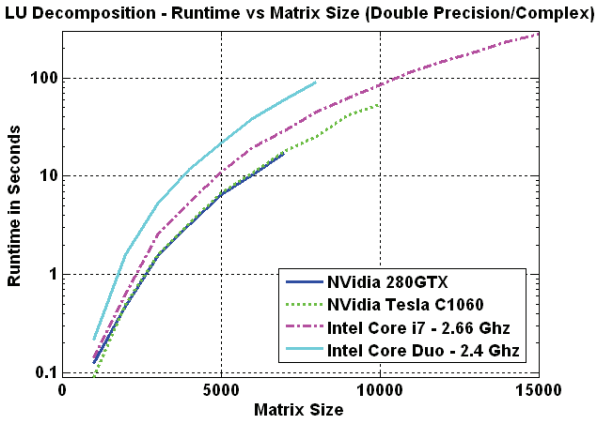


Fig. 3. Runtimes for complex double precision LU decomposition.

The addition of complex data to the solver showed a drastic effect on the runtimes of the developed LU solvers. In order to understand how the various implementations performed, it is necessary to examine how the CPU only and the CPU+GPU implementations compared against themselves. Figure 4 shows the runtimes on the Intel Core i7 for the CPU only implementations.

The addition of double precision to the implementation increased the runtime by only double. Since twice the data is being transferred in this case, it can be concluded that for the real single and double precision cases, the runtimes are simply a matter of the memory transfer rates. In the complex double precision case, the runtimes lagged the real single precision case by a factor of approximately 7. Since four times the data is required to be transferred it can be seen that the arithmetic itself becomes the limiting factor in performance.

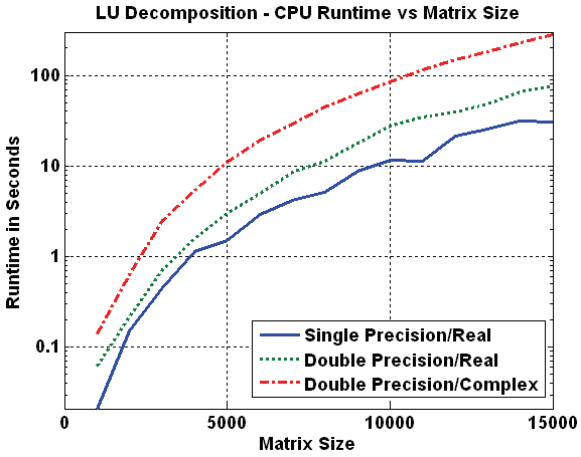


Fig. 4. Runtimes for CPU only LU decomposition.

Figure 5 shows the comparison for the CPU+GPU cases running on the Intel Core i7 with the NVIDIA Tesla C1060 GPU. As shown before, the double precision increased the runtime relative to the single precision by only around 90%. While the data transferred did double, the GPU was able to handle the data more efficiently and thus did not require twice the time to make the transfer. Likewise from the CPU only cases, the memory transfer rates appear to be the limiting factor in the runtimes for these cases. However, in the complex double precision case, the slowdown is more pronounced. The runtime for the complex double precision is approximately twenty times slower over the real single precision case. Just as with the CPU only case, the complex double precision implementation becomes limited not by the memory access rate, but by the speed the system can perform the computations. Since the current CUBLAS on GPU is nowhere near as efficient as the MKL BLAS on CPU in performing double precision calculations, the GPU performance suffers a larger runtime penalty.

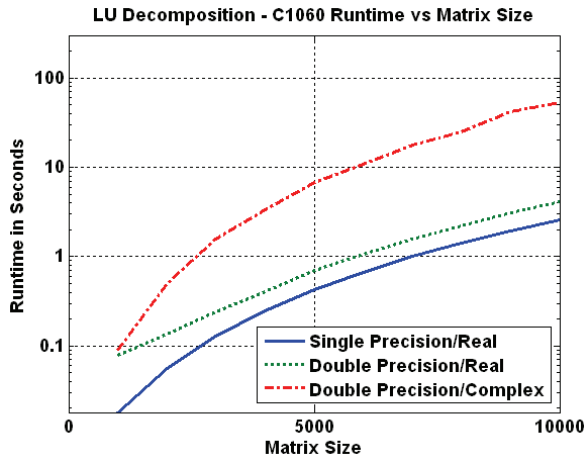


Fig. 5. CPU+GPU LU decomposition runtimes.

VI. Verification and Examples

To show the advantage of the CPU+GPU based solver, few examples were tested. These examples are based on a method of moments (MoM) solution whose results are well documented. Each of these examples will be used to compare both the speed and the accuracy of the CPU+GPU based solutions relative to the CPU only solution. For simplicity, all examples will be discretized with 4096 segments and run on an Intel Core i7 with a NVIDIA 280GTX. The 4096 segments were choosen to show the performance for a simulation of decent size. The CPU only code utilizes all 4 cores of the Core i7 and the CPU+GPU code utilizes the same with the addition of the graphics card. All solutions were computed with double precision complex solvers.

The first example is a simple wire dipole antenna. This example will calculate the current along a wire antenna of length ***L*** (0.1m) and diameter ***A*** (0.2mm) that is excited by a magnetic frill model as shown in Fig. 6. Sinusoidal basis functions and mid-point integration procedure are used for the solution of the resulting integral equation.

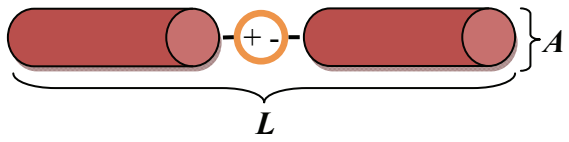


Fig. 6. Dipole wire antenna configuration.

The CPU+GPU code was run against the reference codes to ensure proper operation. Figure 7 shows the current along the wire in both codes. The results show very good agreement with only very minor differences in the magnitude of the current. These differences which are less than 0.1% can be attributed to minor differences in how the numbers were stored and calculated in the various programs and the use of the GPU in the simulation.

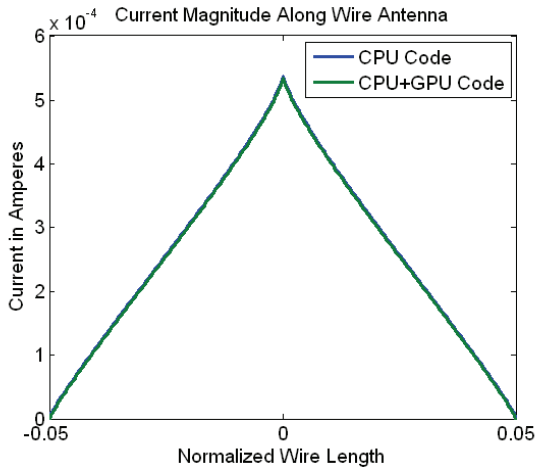


Fig. 7. Current distribution along the dipole wire antenna.

The second example shows the calculation of the current distribution along a PEC plate illuminated by a TM_z plane wave. Figure 8 shows the configuration of this setup. In this setup the width of the PEC plate is one wavelength and the TM_z plane wave incident to the face of the plate at a 45 degree angle.

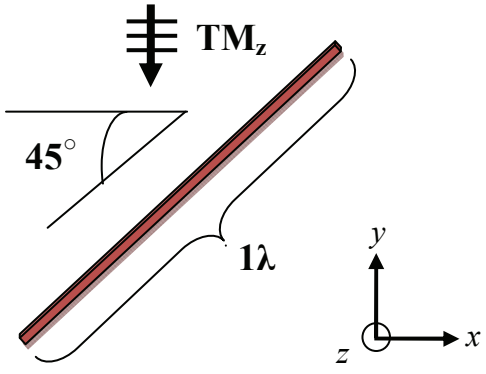


Fig. 8. PEC plate and excitation configuration.

This example was run and compared against the reference code as seen in Fig. 9. The CPU+GPU code again show excellent agreement in calculating the surface current of the PEC plate. The maximum error observed between the two solvers is 0.08%.

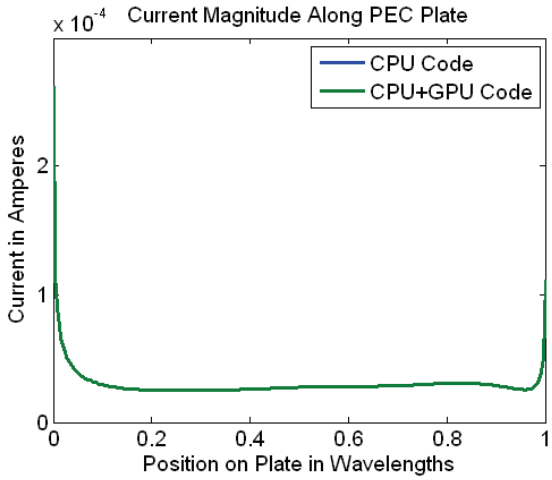


Fig. 9. Current distribution along the PEC plate.

The last example shows the calculation of the current distribution along a PEC cylinder illuminated by a TM_z plane wave. Figure 10 shows the configuration of this setup with the diameter of the PEC cylinder being one wavelength.

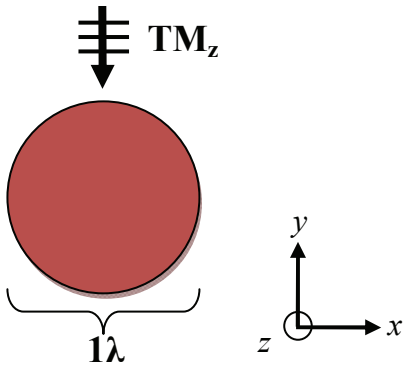


Fig. 10. PEC cylinder and excitation configuration.

Figure 11 shows the current magnitude along the PEC cylinder for both cases. As shown, the agreement between the two codes is excellent. In this case, the maximum error between the CPU and CPU+GPU codes was less than 0.02%

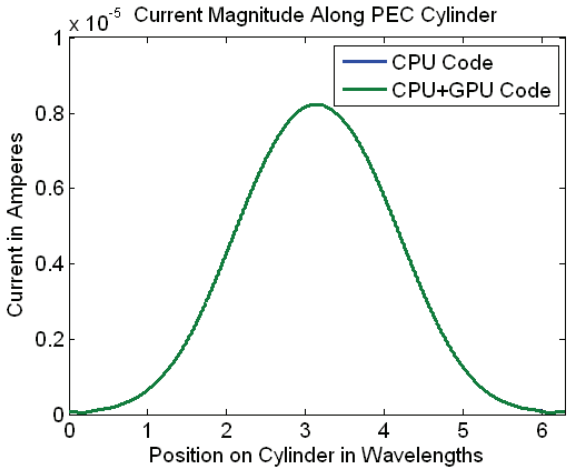


Fig. 11. Current distribution along the PEC cylinder.

All three of the sample cases show excellent agreement with the CPU only solver and successfully solved the problems utilizing the GPU. For these cases a single solve time on the CPU required approximately 6.3 seconds while the CPU+GPU only required 3.2 seconds. Many cases in computational electromagnetics, such as computing the monostatic RCS of an object, require solving for hundreds or more of right hand sides. The speed increase shown for even a moderate matrix of rank 4096 can halve the solution time compared against a high end CPU. If double precision is not required, the time savings can be even greater.

VI. Conclusions

It has been shown that an LU decomposition solver can be effectively implemented utilizing the GPU for various data types from real single precision to complex double precision. Due to the nature of certain functions required for LU decomposition, the use of the CPU to perform various operations is necessitated.

While the complex double precision LU decomposition solver did not maintain increase in speed as for the real precision cases did, the increase of two-fold can have a drastic effect on CEM simulation times, especially for problems of multiple right-hand sides. The decrease in speed gain from the CPU+GPU implementation in the

complex double precision cases can be easily attributed to the immature state of double precision arithmetic on this generation of GPU's. Future generations of GPU's have been promised to dramatically increase double precision arithmetic computations speed which should allow for greater utilization of the developed GPU routines for faster solutions to a variety of CEM and other applications.

REFERENCES

[1] M. J. Inman and A. Z. Elsherbeni, "Programming video cards for computational electromagnetics applications," *IEEE Antennas Propagation Mag.*, Vol. 47, Issue 6, pp. 71-78, 2005.

[2] K. Fatahalian, et. al., "Understanding the Efficiency of GPU Algorithms for Matrix-Matrix Multiplication", Stanford University, 2004.

[3] V. Volkov and J. W. Demmel, *Benchmarking GPUs to tune dense linear algebra*, SC08, 2008

[4] N. Galoppo, N. Govindaraju, M. Henson, and D. Manocha, *LU-GPU: Efficient Algorithms for Solving Dense Linear Systems on Graphics Hardware*, Proceedings of the ACM/IEEE conference on Supercomputing, 2005.

[5] E. Anderson, Z. Bai, J. Dongarra, A. Greenbaum, A. Mckenney, J. Du Croz, S. Hammerling, J. Demmel, C. Bischof, And D. Sorensen, LAPACK: a portable linear algebra library for high-performance computers, *Supercomputing '90*, 1990.

[6] M. Baboulin, J. Dongarra, and S. Tomov. Some Issues in Dense Linear Algebra for Multicore and Special Purpose Architectures, LAPACK Working Note 200, 1993.

[7] CUDA User Forums, http://forums.nvidia.com

Matthew Joseph Inman received his B.S. in Electrical Engineering in 2000 and his Masters in Electromagnetics in 2003 from the University of Mississippi. He is currently pursuing Ph. D. studies in electromagnetics there. He is employed by the University of Mississippi as a research assistant and graduate instructor teaching a number of undergraduate courses. His interests involve electromagnetic theories, numerical techniques, antenna design and visualization.

Atef Z. Elsherbeni is a Professor of Electrical Engineering and Associate Dean for Research and Graduate Programs, the Director of The School of Engineering CAD Lab, and the Associate Director of The Center for Applied Electromagnetic Systems Research (CAESR) at The University of Mississippi. In 2004 he was appointed as an adjunct Professor, at The Department of Electrical Engineering and Computer Science of the L.C. Smith College of Engineering and Computer Science at Syracuse University. On 2009 he was selected as Finland Distinguished Professor by the Academy of Finland and Tekes. Dr. Elsherbeni has conducted research dealing with scattering and diffraction by dielectric and metal objects, finite difference time domain analysis of passive and active microwave devices including planar transmission lines, field visualization and software development for EM education, interactions of electromagnetic waves with human body, sensors development for monitoring soil moisture, airports noise levels, air quality including haze and humidity, reflector and printed antennas and antenna arrays for radars, UAV, and personal communication systems, antennas for wideband applications, antenna and material properties measurements, and hardware and software acceleration of computational techniques for electromagentics. Dr. Elsherbeni is the co-author of the book "*The Finite Difference Time Domain Method for Electromagnetics With MATLAB Simulations"*, SciTech 2009, the book "*Antenna Design and Visualization Using Matlab*", SciTech, 2006, the book "*MATLAB Simulations for Radar Systems Design*", CRC Press, 2003, the book *"Electromagnetic Scattering Using the Iterative Multiregion Technique"*, Morgan & Claypool, 2007, the book *"Electromagnetics and Antenna Optimization using Taguchi's Method"*, Morgan & Claypool, 2007, and the main author of the chapters "*Handheld Antennas"* and "*The Finite Difference Time Domain Technique for Microstrip Antennas"* in Handbook of Antennas in Wireless Communications, CRC Press, 2001. Dr. Elsherbeni is a Fellow member of the Institute of Electrical and Electronics Engineers (IEEE) and a Fellow member of The Applied Computational

Electromagnetics Society (ACES). He is the Editor-in-Chief for ACES Journal and an Associate Editor to the Radio Science Journal.

C. J. Reddy received B. Tech. degree in Electronics and Communications Engineering from Regional Engineering College (now National Institute of Technology), Warangal, India in 1983. He received his M.Tech. degree in Microwave and Optical Communication Engineering and Ph.D. degree in Electrical Engineering, both from Indian Institute of Technology, Kharagpur, India, in 1986 and 1988 respectively. From 1987 to 1991, he worked as a Scientific Officer at SAMEER (India) and participated in radar system design and development. In 1991, he was awarded NSERC Visiting Fellowship to conduct research at Communications Research Center, Ottawa, Canada. Later in 1993, he was awarded a National Research Council (USA)'s Research Associateship to conduct research in computational electromagnetics at NASA Langley Research Center, Hampton, Virginia. Dr. Reddy worked as a Research Professor at Hampton University from 1995 to 2000, while conducting research at NASA Langley Research Center. During this time, he developed various FEM codes for electromagnetics. He also worked on design and simulation of antennas for automobiles and aircraft structures. Particularly development of his hybrid Finite Element Method/Method of Moments/Geometrical Theory of Diffraction code for cavity backed aperture antenna analysis received Certificate of Recognition from NASA.

Currently, Dr. Reddy is the President and Chief Technical Officer of Applied EM Inc, a small company specializing in computational electromagnetics, antenna design and development. At Applied EM, Dr. Reddy successfully led many Small Business Innovative Research (SBIR) projects from the US Department of Defense (DoD). Some of the technologies developed under these projects are being considered for transition to the DoD. Dr. Reddy also serves as the President of EM Software & Systems (USA) Inc. At EMSS (USA), he is leading the marketing and support of commercial 3D electromagnetic software, FEKO in the US, Canada, Mexico and Central America.

Dr. Reddy is a Senior Member of the IEEE. He is also a member of Applied Computational Electromagetic Society (ACES) and serves as a member of Board of Directors. He has published more than 60 referred journal articles and conference papers.

GPU Based TLM Algorithms in CUDA and OpenCL

Filippo Rossi, Colter McQuay, and Poman So

Computational Electromagnetics Research Laboratory
Department of Electrical and Computer Engineering
University of Victoria, Victoria, BC, V8W 3P6, Canada

***Abstract*—** Recent advancements in graphics computing technology has brought highly parallel processing power to desktop computers. As multi-core multi-processor computing technology becomes mature, a new front in parallel computing technology based on graphics processing units has emerged. This paper reports a highly parallel symmetrical condensed node TLM procedure for the NVIDIA graphics processing units. The algorithm has been tested on three NVIDIA processors, from low-end laptop graphics card to high-end workstation graphics processors.

***Index Terms*—** TLM, FDTD, GPU, SIMD, time-domain, parallel computing, stream computing.

I. INTRODUCTION

Graphics processing unit (GPU) based parallel computing has been an important topic for the computing industry for over a decade. Macedonia addressed this topic in a computing magazine article in 2003 [1]. Most of the papers on GPU computing were related to signal and image processing [2–6]. Krakiwsky *et al.* and Inman *et al*. applied the technique to accelerate the FDTD algorithm [7, 8]. Takizawa *et al.* applied GPU computing to heat transfer simulation [9]. Z. Luo *et al.* and Harding *et al.* applied the paradigm to artificial neural network [10] and genetic algorithm [11], respectively. Furthermore, a cluster of GPU based computers can be created to execute grand challenge problems [12]. Researchers at Stanford [13] have been using this technique for years in protein folding computation.

Developing general purpose numerical modules for GPU was made easy by NVIDIA. The company released its Compute Unified Device Architecture (CUDA) Software Development Kit (SDK) in early 2007. The SDK enables programmers to develop GPU code in a high level language, *C-for-CUDA*. Rossi *et al*. reported the first implementations of a two- and a three-dimensional transmission line matrix (TLM) [14-17] program using the CUDA SDK [18]. This highly parallel TLM code has been ported to the new released OpenCL [19] environment. This makes it possible to run the program on non-NVIDIA GPUs and on heterogeneous computing hardware (for instance, GPU based computers with multiple multi-core CPUs). This paper addresses the algorithm design, programming techniques, and performance issues for implementing GPU based programs; in particular, the pros and cons of choosing CUDA and OpenCL will be discussed.

II. GPU COMPUTING

Modern GPU designs architectures are based on the Single Instruction Multiple Data (SIMD) computing paradigm. This hardware architecture utilizes multiple processors to perform similar tasks on vast quantities of data. The appeal for GPUs exists not only because of their computational ability, but also given that they are relatively inexpensive and can be installed on existing workstations. The NVIDIA GPUs used in this project are GeForce 8800 Ultra, Quadro FX 570M and Quadro FX5600 graphics cards [20]; these GPUs have 4 to 16 multi-processors with 8 processors each for a total of 32 to 128 processors. The GPUs have a maximum of 1.5 GB of GDDR3 global memory. A schematic that depicts the computing model of the NVIDIA GPU using a layer of a TLM mesh is shown in Fig. 1. The figure illustrates a typical iteration cycle. The data structure to be processed (called a mesh) is defined in both the CPU and the GPU. After seeding a data structure with initial conditions, the host transfers the data to the GPU's global memory and constant memory. A GPU function (called a kernel) would then be invoked which would execute on all multi-processors (4 to 16). This computing paradigm is scalable by utilizing GPU clusters internal or even external to a workstation [21]. Adaptation to GPUs is suitable for many science and engineering applications. However, the parallelization of existing algorithms may require intricate and complex adaption efforts.

The driving forces behind the computing framework depicted in Fig. 1 are the thread-blocks that control the GPU executions, Fig. 2. A thread block is defined as a grouping of threads that executes concurrently on the GPU multi-processors. Multiple data elements could be

assigned per thread (data block). A maximum of 512 threads per thread block are available for the GPUs, therefore it is then necessary to partition the mesh into data blocks. Each multi-processor would, in turn, execute on a data block utilizing its thread-block and then transfer the results back to the global memory. After which the multiprocessor would download the next available data block. The 16 multi-processors on the graphics card used in this project worked concurrently and are coordinated to process all data blocks in the global memory.

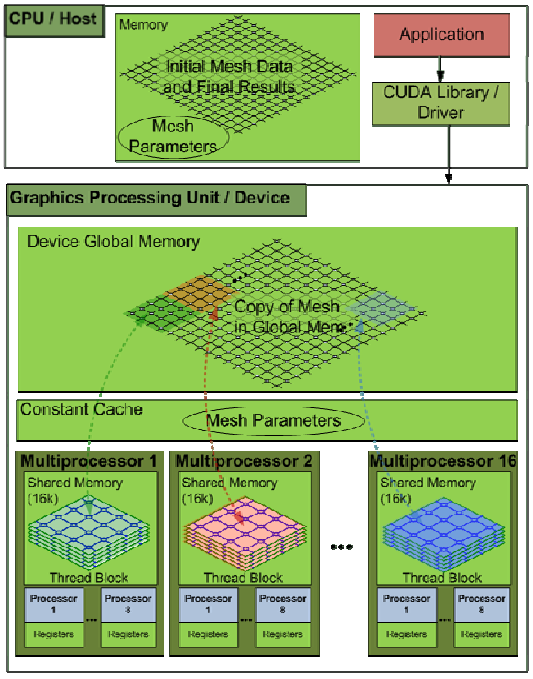


Fig. 1. NVIDIA CUDA based GPU computing framework.

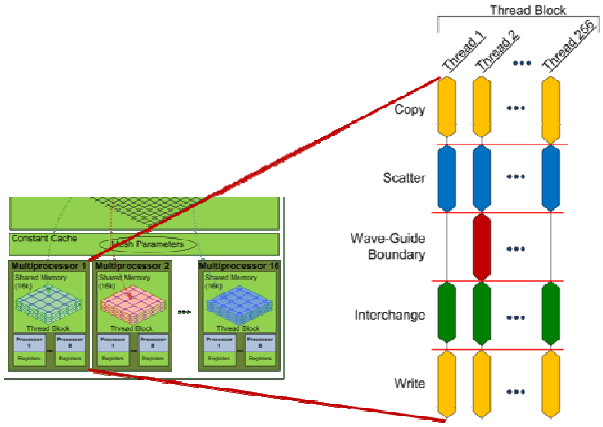
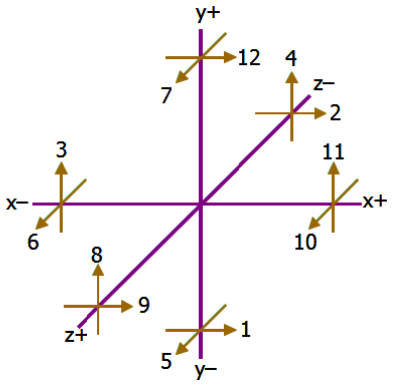


Fig. 2. NVIDIA CUDA based GPU computing framework.

III. SYMMETRIC CONDENSED NODE TLM

The three-dimensional symmetrical condensed node (SCN) is depicted in Fig. 3. The fundamental procedures in a TLM algorithm are the scattering, transfer and reflection of voltage impulses, [17]. There are two orthogonal transmission link lines in each port of the TLM node. Voltage impulses travelling along the *x*-axis are polarized in the *y*- or *z*-direction; similarly voltage impulses on the *y*- and *z*-axis are polarized in the other two orthogonal directions on the plane transverse to the direction of propagation. Hence, there are a total of 12 voltage impulses in each symmetrical condensed node. The scattering matrix for the symmetrical condensed node TLM method is a 12×12 matrix [17]. Therefore, a matrix multiplication operation can be used to obtain the reflected voltage vector from the incident voltage vector. The other two TLM operations are transfer of impulses to the neighboring link lines and reflection of impulses at material boundaries, Fig. 4. These two operations are applied to the two orthogonally polarized voltage impulses. All three TLM operations — scattering, transfer and reflection of voltage impulses — are localized operations which may be executed in parallel to reduce computing time. A quad-core processor may execute each operation concurrently for four TLM



$$[V]_{k+1}^{r} = [S] \times [V]_{k}^{i}$$

$$[S] = \begin{bmatrix} a & a & & & & & a & -a & & & & \\ a & & & & a & & & -a & a & & & \\ a & & a & & & & a & & & & & -a \\ & & a & & a & -a & & & a & & & \\ & & & a & & a & -a & a & & & & \\ a & & & & a & & a & & -a & & & \\ & & & -a & a & a & & a & & & & \\ & & a & & -a & a & & & a & & & \\ a & & & & & -a & & & a & a & & \\ & -a & & & a & a & & a & & & & \\ -a & & a & & & & a & & & & & a \\ & a & -a & & & & a & & a & & & \end{bmatrix}$$

Fig. 3. SCN scattering algorithm.

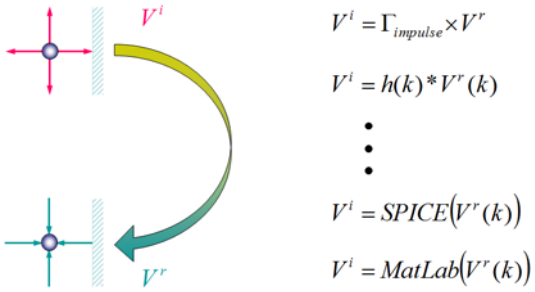


Fig. 4. TLM boundary operation.

nodes. A traditional serial TLM program can be easily parallelized by using OpenMP [22] compiler directives. However, the number of cores on a single CPU is small and the gain in performance by using OpenMP is therefore still limited. With GPUs, numerical procedures such those described above can be executed in parallel on a much larger scale.

IV. IMPLEMENTATION

Efficient use of multiprocessor resources, especially global memory transfer strategies, can help to achieve close to the maximum theoretical operating speeds of GPUs. Memory transfer rates between the global memory and multiprocessors can be used as a benchmark for GPU performance since much of the kernel execution time (70% to 80%) may be spent in accessing global memory. In the case of the Quadro FX 5600, the maximum theoretical memory bandwidth to global memory is 76.8 GB/sec [21] or expressed as read+write round trip: 38.4 GB/sec.

Memory coalescing is a performance enhancement technique whereby access to global memory by multiprocessors can be accelerated [20]. Global memory (GDDR3 memory) consists of physical banks of memory. Access to global memory by any of the multiprocessors results in 400-600 clock cycles of latency. In other words, each four byte float or integer copied from or written to global memory takes 400-600 clock cycles. Since the GDDR3 global memory exists physically as banks of memory, reads/writes can be organized such that [20]:

1. The starting address of each half-warp (16 threads) falls on a 64 byte interval
2. Each thread of a half-warp reads/writes 4, 8 or 16 bytes consecutively
3. The threads of each half-warp must be spaced at 4, 8 or 16 byte intervals.

Figure 5 illustrates the differences in memory access speed (GB/sec read-write round trip) between coalesced code (~25 GB/sec) and non-coalesced code (~3 GB/sec). A speed-up of over 8 times for coalesced kernel code warranted developing TLM kernel that adhered to coalescing coding strategies.

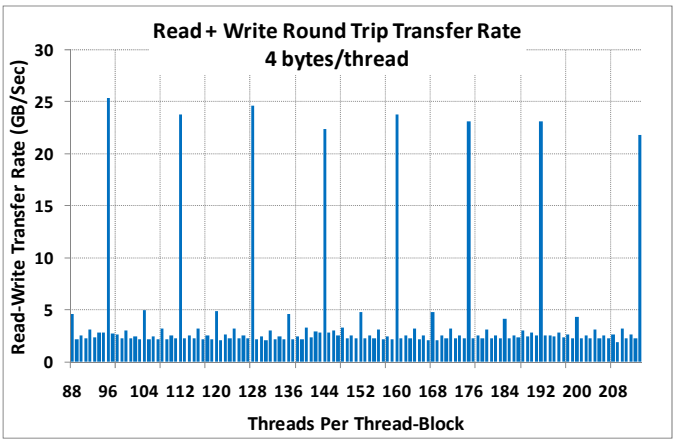


Fig. 5. GPU performance differences between coalesced a non-coalesced memory configurations.

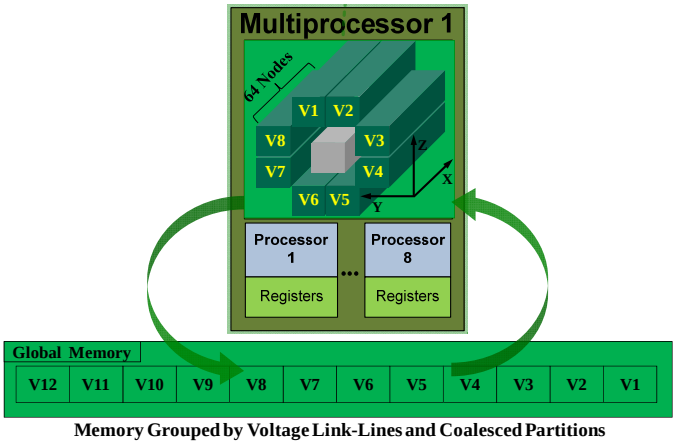


Fig. 6. Coalesced global memory access by thread-blocks of multiprocessors.

The TLM kernel is designed such that global memory is grouped by voltage link lines (12 per TLM node), and accessed by the multiprocessors in a coalesced manner to take maximum advantage of the GPUs speed performance, Fig. 6. The resolution of the Y and Z dimensions of a mesh is each one node wide. However, the X dimension is partitioned into 64 node segments. The addressing is thus contiguous, first in the x-direction, then the y-direction and finally the z-direction. The thread-block dimension is defined at 64 threads, where the kernel would read a voltage link line for 64 nodes at a time in the x-direction. For example, 64 values of V1 would be read for 64 nodes, then for V2 would be read for the same 64 nodes and so on until all 12 voltagess have been read so that the scattering

procedure may commence on 64 nodes. Writing results back to global memory is done in a similar manner. Fig 7 showcases the performance enhancement of configuring the TLM kernel in a coalesced fashion.

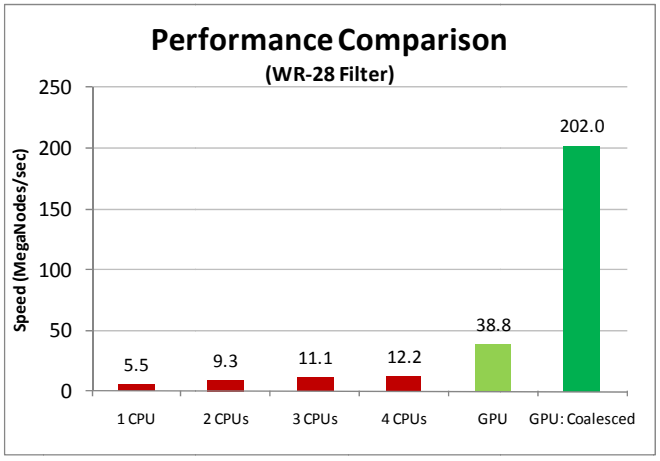


Fig. 7. TLM codes performance on 1 to 4 CPUs. The codes were used to analyze a WR-28 filter (150,000 nodes). The GPU codes have a non-coalesced GPU kernel and a coalesced GPU Kernel.

V. OPENCL IMPLEMENTATION

The newly released OpenCL application programming interface (API) has many features found in the CUDA paradigm. In fact, once a program has been developed using the CUDA framework, it is not difficult to implement an equivalent version in OpenCL. This is especially true if the NVIDIA's "Driver API" method is used instead of "C for CUDA" [19]. Both OpenCL and the Driver API standards are handle-based hence objects to memory and functions are required. However, OpenCL differs in its ability to utilize a heterogeneous mixture of parallel hardware. An OpenCL application can utilize a mixture of GPUs, CPUs, and other processors. Therefore OpenCL is a powerful API for integrating multiple parallel platforms under one cohesive programming paradigm.

Figure 8 depicts some equivalent programming constructs in the two APIs. As shown in the figure most of the equivalent constructs have meaningful mapping from CUDA to OpenCL; the only exception in the table is the "Local Memory" construct.

In addition to systemically replacing the programming constructs, it is also necessary to substitute CUDA kernel code attribute modifiers with equivalent OpenCL methods, Fig. 9. A short code segment that illustrates the code transformation concept is shown in Fig. 10. When compared to CUDA, OpenCL is a relatively new API hence it is less efficient than CUDA especially when the problem size is small. Figure 11 compares the performance of our CUDA and OpenCL TLM code. When the structure size reaches about 10 million nodes, the two programs have similar computation speed.

CUDA	OpenCL
Thread	Work-item
Thread Block	Work Group
Shared Memory	Local Memory
Local Memory	Private Memory

Fig. 8. Equivalent programming constructs in CUDA and OpenCL.

CUDA Kernel Code	OpenCL Kernel Code Replacement
gridDim	**get_num_groups(*n*)**
blockDim	**get_local_size(*n*)**
blockIdx	**get_group_id(*n*)**
threadIdx	**get_local_id(*n*)**
__global__ function (callable from host, not callable from device)	**__kernel** function (callable from host or device)
__device__ function (not callable from host)	
__constant__ variable declaration	**__constant** variable declaration
__device__ variable declaration	**__global** variable declaration
__shared__ variable declaration	**__local** variable declaration
__syncthreads()	**barrier()**

Fig. 9. Attribute transformation table.

```
__global__ void
meshTLM_simulate_kernel(
  __device__ nodeTLM* nodeMeshIn,
  __constant__ float* boundaryTable,
  unsigned int meshWidth,
  unsigned int meshHeight,
  ){
__shared__ nodeTLM* subMesh;
}
```

CUDA Kernel Code

```
__kernel void
meshTLM_simulate_kernel(
  __global nodeTLM* nodeMeshIn,
  __constant float* boundaryTable,
  unsigned int meshWidth,
  unsigned int meshHeight,
  __local nodeTLM* subMesh
  ){
}
```

OpenCL Kernel Code

Fig. 10. Kernel code transformation.

VI. CONCLUSION

We have successfully designed and implemented a highly parallel SCN TLM algorithm for the CUDA/OpenCL enabled NVIDIA GPU. It is found that both CUDA and OpenCL are good programming APIs for implementing computational intensive applications such as TLM on GPU based hardware. Our latest CUDA implementation of the 3D SCN TLM routine has achieved a 293 MNodes/sec performance of an empty

structure and 288 MNodes/sec with the filter implemented. Ongoing research activities are focusing on improving the speed of execution and adapting the

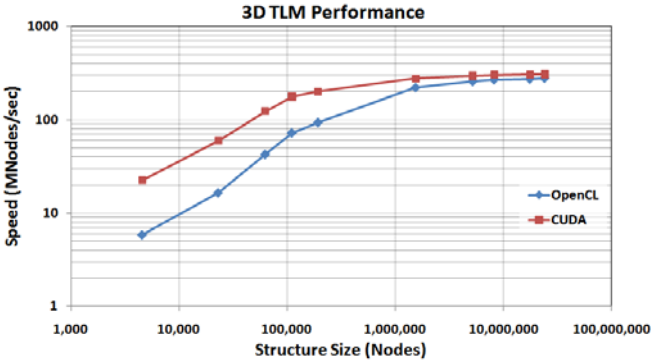


Fig. 11. Performance comparison — CUDA versus OpenCL.

algorithms to solve various structures and configurations. An investigation in utilizing a cluster of 4 NVIDIA GPUs on an Acceleware ClusterInABox™ Quad Q30 workstation is being conducted. The implementations described above can be modified to handle the generalized symmetrical condensed node (GSCN) TLM algorithm developed by Trenkic *et al.* [23, 24]. The total number of voltage impulses to be stored per node would thus increases from 12 to 18. This would reduce the number of nodes each multi-processor thread-block can handle. However the GSCN scattering procedure would not cause any significant reduction on the overall performance as the bottleneck is in the data transfer, not in number of floating point operations. Hence, the performance results depicted in figures 8 and 12 are still valid but the code would reach the maximum acceleration at a smaller structure size.

ACKNOWLEDGMENT

The authors wish to acknowledge the financial supports from the Canada Foundation for Innovation (CFI) and the Natural Science and Engineering Research Council (NSERC) of Canada.

REFERENCES

[1] M. Macedonia, "The GPU Enters Computing's Mainstream", *IEEE Computer*, vol. 36, no. 10, pp. 106–108, October 2003.

[2] G. Shen, G. P. Gao, S. Li, H. Y. Shum and Y. Q. Zhang, "Accelerating Video Decoding Using GPU", *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 15, no. 5, pp. 685–693, May 2005.

[3] J. Y. Hong and M. D. Wang, "High speed processing of biomedical images using programmable GPU", *International Conference on Image Processing*, vol. 4, pp. 2455–2458, October 2004.

[4] Y. Heng and L. Gu, "GPU-based Volume Rendering for Medical Image Visualization", *27th Annual International Conference on Engineering in Medicine and Biology*, pp. 5145–5148, 2005.

[5] O. Fialka and M. Cadik, "FFT and Convolution Performance in Image Filtering on GPU", *IEEE Proceedings of the Information Visualization*, pp. 609–614, July 2006.

[6] J. S. Meredith, S. R. Alam and J. S. Vetter, "Analysis of a Computational Biology Simulation Technique on Emerging Processing Architectures", *IEEE International Symposium on Parallel and Distributed Processing*, pp. 1–8, March 2007.

[7] S. E. Krakiwsky, L. E. Turner and M. M. Okoniewski, "Graphics Processor Unit Acceleration of Finite-Difference Time-Domain Algorithm", *Proceedings of IEEE International Symposium on Circuits and Systems*, vol. 5, pp. V265 – V268, May 2004.

[8] M. J. Inman, and A. Z. Elsherbeni, "Programming video cards for computational electromagnetics applications", *IEEE Antennas and Propagation Magazine,* vol. 47, no. 6, pp. 71–78, December 2005.

[9] H. Takizawa, N. Yamada, S. Sakai, and H. Kobayashi, "Radiative Heat Transfer Simulation Using Programmable Graphics Hardware", *5th IEEE/ACIS International Conference on Computer and Information Science*, pp. 29–37, July 2006.

[10] Z. Luo, H. Liu, and X. Wu, "Artificial Neural Network Computation on Graphic Process Unit", *Proceedings of IEEE International Joint Conference on Neural Networks*, vol. 1, pp. 622–626, August 2005.

[11] S. Harding, W. Banzhaf, "Fast Genetic Programming and Artificial Developmental Systems on GPUs", *21st International Symposium on High Performance Computing Systems and Applications*, p. 2, May 2007.

[12] F. Zhe, Q. Feng, A. Kaufman and S. Yoakum-Stover, "GPU Cluster for High Performance Computing", *Proceedings of the ACM/IEEE Conference on Supercomputing*, pp. 47, 2004.

[13] folding.stanford.edu/FAQ-ATI.html

[14] F. V. Rossi, "Massively Parallel Two-Dimensional TLM Algorithm on Graphics Processing Units," *IEEE International Microwave Symposium,* June 2008.

[15] F. Rossi and P. P. M. So, "Parallelized three-dimensional TLM algorithms on a graphics processing unit", *25th International Review of Progress in Applied Computational Electromagnetics Symposium,* pp. 110–114, March 2009.

[16] W. J. R. Hoefer, "The Transmission-Line Matrix Method – Theory and Applications", *IEEE Transactions on Microwave Theory and Techniques,* vol. MTT-33. No. 10, pp.882-893, October 1995.

[17] P. B. Johns, "A symmetrical condensed node for the TLM method," *IEEE Transactions on Microwave Theory and Technique,* vol-35, no. 4, pp. 370–377, April 1987.

[18] http://www.nvidia.com/object/cuda_home_new.html, April 2010.

[19] http://www.khronos.org/opencl/

[20] http://www.nvidia.com

[21] ClusterInABox Quad (Q30) Product Info, http://www.acceleware.com/default/index.cfm/our-products/clusterinabox-quad, November 2008.

[22] http://OpenMP.org/wp/

[23] V. Trenkic, C. Christopoulos, and T. M. Benson, "Development of a general symmetrical condensed node for the TLM method", *IEEE Trans. on Microwave Theory and Techniques,* vol. MTT-44, no. 12, pp. 2129–2135, December 1996.

[24] V. Trenkic, C. Christopoulos, and T. M. Benson, "Advanced node formulations in TLM — the adaptable symmetrical condensed node", *IEEE Trans. on Microwave Theory and Techniques,* vol. MTT-44, no. 12, pp. 2473–2478, December 1996.

Filippo Rossi received the B.Eng. degree in Electrical Engineering in 2008 from the University of Victoria, Victoria, British Columbia, Canada. Currently he is completing a Master of Applied Science at the University of Victoria. He is working at the Computational Electromagnetics Research Laboratory (CERL) at the University of Victoria in GPU computing, as well as working with the Millimeter Instrumentation team at the Herzberg Institute of Astrophysics, Victoria, B.C., Canada.

Colter McQuay is an undergraduate student at the University of Victoria, B.C. in Electrical Engineering with a specialization in Signal Processing and Computer Music. Colter was born in Kamloops B.C. in 1987. In 2009, his research focused on implementing TLM algorithms on GPU hardware using OpenCL , presenting a paper at the USRI Conference in Boulder Colorado in Jan 2010. Currently Colter is involved in writing an open source electromagnetic simulation application using the code developed in previous research.

Poman So is an Assistant Professor at the University of Victoria. He received the B.Sc. degree in Computer Science and Physics from the University of Toronto, Toronto, Ontario, Canada, in 1985; the B.A.Sc. and M.A.Sc. degrees in Electrical Engineering from the University of Ottawa, Ottawa, Ontario, Canada, in 1985 and 1987, respectively; and the Ph.D. degree in Electrical Engineering from the University of Victoria, Victoria, BC, Canada, in 1996.

Dr. So possesses twenty years of hands-on object-oriented software engineering experience in time-domain computational electromagnetics. He developed a number of electromagnetic wave simulators based on the Transmission Line Matrix (TLM) method. Dr. So is a co-founder of the Faustus Scientific Corporation and is the creator and chief software architect of MEFiSTo, a general purpose time-domain electromagnetic field solver based on the Transmission Line Matrix method. From July 1998 to June 2005, Dr. So was the Principal Software Engineer at Faustus Scientific Corporation. In July 2005, He joined the Department of Electrical Engineering at the University of Victoria. His research interests include object-oriented computational electromagnetics, graphics processing unit (GPU) based massively parallel TLM algorithms, time domain modeling of advanced electromagnetic structures, and modeling of bio-electromagnetic systems.

Dr. So is a Registered Professional Engineer in the Province of British Columbia, Canada. He is a senior member of The Institute of Electrical and Electronics Engineers (IEEE), a member of Applied Computational Electromagnetics Society (ACES), and a member of the Canadian Medical and Biological Engineering Society (CMBES). He has published over 100 refereed journal and conference papers. Dr. So serves regularly a reviewer for the IEEE Transactions on Microwave Theory and Techniques, the IEEE Microwave and Wireless Components Letters, the Applied Computational Electromagnetics Society Journal, International Journal of RF and Microwave Computer Aided Engineering, the International Journal of Numerical Modeling – Electronic Networks, Devices and Fields by John Wiley and Sons Ltd. He is a member of the Editorial Advisory Board for the International Journal of Numerical Modeling – Electronic Networks, Devices and Fields by John Wiley and Sons.

Fast CPU/GPU Pattern Evaluation of Irregular Arrays

A. Capozzoli, C. Curcio, G. D'Elia, A. Liseno, and P. Vinetti

Dipartimento di Ingegneria Biomedica, Elettronica e delle Telecomunicazioni
Università di Napoli Federico II, Via Claudio, 21 Naples, Italy
a.capozzoli@unina.it

Abstract- An approach for the fast analysis of "irregular", i.e., of conformal, periodic or aperiodic, 2D arrays, based on the use of the *p*-series approach and Non-Uniform FFT (NUFFT) routines is proposed. The approach allows for modulating the computational burden depending on the array curvature and, thanks to the use of the NUFFT, the asymptotic growth of the computing time reduces to that of a few, standard FFTs. A sub-array partition strategy is also sketched and shown to further unburden the procedure and control the accuracy. The approach has been implemented in both sequential and parallel codes enabling its execution on CPUs and on cost-effective, massively parallel computing platforms like Graphic Processing Units (GPUs). Its performance in terms of computational efficiency and accuracy has been assessed by an extensive numerical analysis and also against benchmarks provided by algorithms based on fast Matrix-Vector Multiplication routines.

Index Terms- Aperiodic array antennas, conformal array antennas, fast antenna analysis, GPU computing, CUDA.

I. INTRODUCTION

Array pattern synthesis is a computationally challenging problem since it requires demanding iterative algorithms for the (local or global) optimization of a properly defined objective functional [1-3]. The computational bottleneck of such algorithms is essentially related to the repeated calculation of the far field pattern (FFP) and possibly of the functional gradient (FG) (as long as gradient-based optimization approaches are adopted).

Different kinds of arrays have been subject in the literature of synthesis procedures. Many of the developed synthesis algorithms refer to "regular arrays" (RAs), for which the elements are arranged on a periodic grid of a portion of a line or plane (see Fig. 1). In the last decade, "irregular arrays" (IAs), namely, arrays for which the elements lay on an "aperiodic" grid and/or on conformal lines or surfaces have been proposed (see Figs. 2-4) to overcome the typical issues of RAs [4-8]. Indeed, "aperiodic" structures allow, as compared to "periodic" ones, a more efficient power handling, if uniformly excited in amplitude [5], and permit improving the bandwidth performance [9], while also reducing the overall number of elements and mitigating the effects of the grating lobes [10]. Furthermore, "conformal" structures, as compared to linear or planar ones, satisfy aerodynamic and low-scattering requirements in aircraft antennas [4], permit space deployability [11] and considerably reduce the feed path length, thus improving the bandwidth behavior, of reflectarray antennas [7]. However, IA synthesis appears computationally more demanding than RAs synthesis, since the FFP or FG evaluations become more burdened.

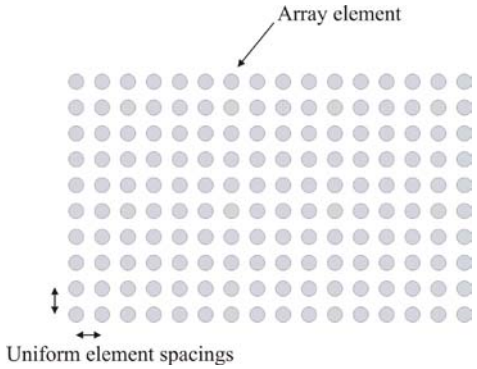


Fig. 1. Example of planar, periodic array.

For RAs, when the array factor can be employed [12] and the far field pattern is evaluated on a regular spectral grid, the excitation coefficients and the array factor are related by a "standard" Discrete Fourier Transform (DFT) link, i.e., a DFT defined on Cartesian, regular grids, as

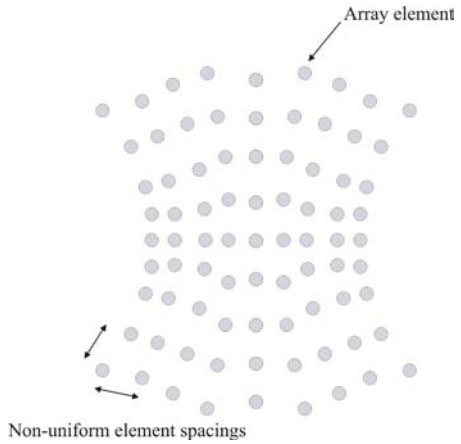


Fig. 2. Example of planar, aperiodic array.

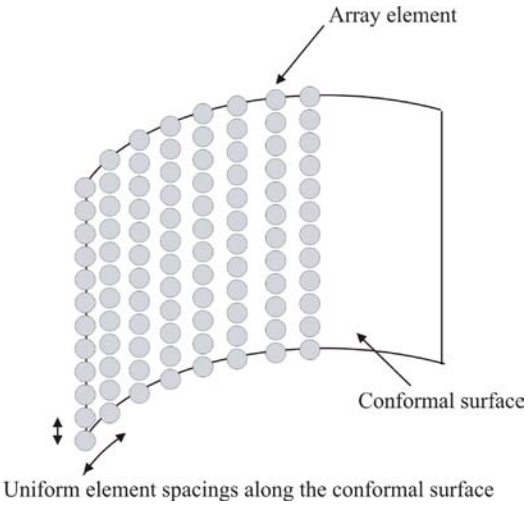


Fig. 3. Example of conformal, periodic array.

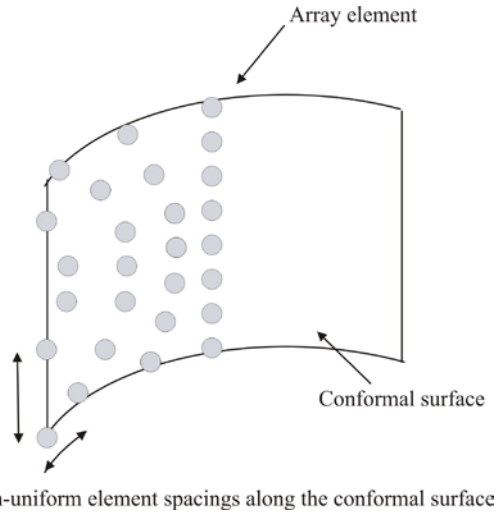


Fig. 4. Example of conformal, aperiodic array.

discussed in [13]. In this case, the speedup is achieved by means of standard Fast Fourier Transform (FFT) routines [14]. Indeed, taking for example arrays of M elements, the use of FFTs changes the $O(M^2)$ computational complexity of each DFT to $O(M\log M)$.

On the other hand, for IAs, the possibility of a direct use of FFTs [6-8] breaks down. Indeed, for planar IAs, evaluating the FFP as well as the FG requires the DFT to be computed on irregular grids (Cartesian non-uniform or non-Cartesian), so that the requirement of standard FFTs is not met anymore. Moreover, for IAs whose elements are arranged on non-linear or non-planar domains, a standard DFT link between array factor and excitation coefficients is lost [7].

The aim of the paper is to show how the asymptotic growth of the computational burden when dealing with IAs can be reduced to $O(M\log M)$ as long as the computation of the FFP can be recast as that of a few FFTs. To this end, three different tools are exploited in the following, namely the *p*-series approach [7,15,16], the Non-Uniform FFT (NUFFT) algorithms [17,18] and a sub-array partitioning strategy. In particular, the *p*-series approach enables, for conformal surfaces with mild curvature, recasting the link between the array excitation coefficients and the FFP as the sum of a few, possibly non-standard, DFTs. On the other hand, NUFFT algorithms quickly evaluate non-standard DFTs as the sum of a few FFTs. And so, the two approaches together are able to restore the yearned $O(M\log M)$ computational complexity. Finally, the sub-array partitioning strategy is capable to additionally improve the method in terms of computational burden and accuracy. It is also shown that the computational approach herein proposed can be even more fruitfully exploited if implemented on innovative, intrinsically parallel, off-the-shelf hardware provided by Graphical Processor Units (GPUs) [19]. GPUs represent, in fact, inexpensive, highly-parallel hardware, significantly mitigating the requirements in terms of space, management, cost and user access, when compared to more complex CPU grid/cluster systems [20]. In addition, while programming on GPUs remains more involved than standard sequential programming, the recent interest in GPUs for scientific computing has promoted the development of effective programming

frameworks [21,22], which in return simplified implementations on these platforms [23]. Finally, it is further shown how the proposed strategy fruitfully modulates, depending on the array curvature, the computational burden without impairing the accuracy of the FFP evaluation.

The performance of the approach is tested by an implementation in C language on a standard, single-core (sequential) CPU and by an implementation in NVIDIA CUDA (Compute Device Unified Architecture) language [24] on a (multithread) NVIDIA GTX 260 GPU. More in detail, C language implementations of the proposed strategy and of an approach based on sequential Optimized Matrix-Vector Multiplication (OMVM) [25,26], generally having an $O(M^2)$ asymptotic complexity, but performing better than a brute-force (i.e., a matrix vector multiplication based on the use of "for loops" [27]) one, have been setup. The OMVM approach has been purposely developed in this paper to be used as a reference for assessing the performance of the proposed strategy. Speedups of more than 10 times for arrays of 10^4 elements obtained by our method as compared to OMVM, FFP evaluation are highlighted. Similarly, CUDA language implementations of the proposed strategy and of an approach based on parallel OMVM routines have been realized. Speedups of more than 8 times for arrays of 10^4 elements, when comparing the GPU version of our approach against that of employing parallel OMVMs are pointed out. Moreover, speedups of more than 40 times, when comparing the GPU and CPU versions of the developed algorithm, are indicated.

Finally, the accuracy of the procedure is discussed.

The paper is organized as follows. In Section II, the problem of radiation is formulated and the strategy exploited for the NUFFT-based evaluation of the FFP, relying on the use of the *p*-series approach, is presented. The benchmarking, OMVM-based method is also sketched. Section III briefly enlightens some details of the sequential (CPU) and parallel (GPU) implementations for both considered approaches (i.e. NUFFT and OMVM). Sections IV and V illustrate and compare the computational performance and accuracy of the NUFFT-based method, as compared to the OMVM-based one. Finally, in Section VI, conclusions are drawn and future developments are foreseen. In the Appendices, the NUFFT algorithm is shortly recalled, C-like and CUDA-like listings of the developed NUFFT routines are reported and ancillary calculations concerning the convenience of adopting a sub-array partitioning are presented.

II. RADIATION BY 2D IRREGULAR ARRAYS

In the following, the approach to the fast analysis of IAs is presented by referring to a general 2D geometry.

Let us consider an antenna array made of M elements, non-uniformly distributed on a 2D arbitrary surface, S, of equation $z=f(x,y)$, $(x,y)\in D$, with D a planar, auxiliary domain, so that the radiating elements are located at the points (x_m,y_m,z_m) with $z_m=f(x_m,y_m)$ and $m = 0, 1, \ldots, M-1$ (see Fig. 5). The complex excitation coefficients are denoted with a_m, $m = 0, 1, \ldots, M-1$.

Generally speaking, the FFP of an IA can be written as [4,6-8]

$$\underline{F}_r(u,v) = \sum_{m=0}^{M-1} a_m \underline{h}_r(u,v,x_m,y_m) e^{j[ux_m+vy_m]} \quad (1)$$

where

$$\begin{cases} u = \beta \sin\theta \cos\varphi \\ v = \beta \sin\theta \sin\varphi \end{cases}, \quad (2)$$

$\beta=2\pi/\lambda$, λ being the wavelength, and $\underline{h}_r(u,v,x_m,y_m)$ accounts for the radiation characteristics and position of the m-th element (see also Subsection C).

Henceforth, the vector aspects of the problem are dismissed. In other words, we assume that $\underline{h}$ can be factored out as

$$\underline{h}_r(u,v,x_m,y_m) = h(u,v,x_m,y_m)\underline{p}(u,v), \quad (3)$$

that is, all the array elements share a common polarization behavior described by $\underline{p}$. Accordingly, $\underline{F}_r(u,v)=F(u,v)\underline{p}(u,v)$, where

$$F(u,v) = \sum_{m=0}^{M-1} a_m h(u,v,x_m,y_m) e^{j[ux_m+vy_m]}. \quad (4)$$

We notice that

- for mild conformal geometries (the elements have approximately the same orientation), vector correction terms to eq. (3) are often negligible;
- for non-mild conformal geometries, the sub-array partitioning strategy helps to mitigate the assumptions needed for the validity of eq. (3) (see Subsection II.e).

In practice, the FFP is required at a number H of spectral positions (u_h,v_h), so that the corresponding discrete values F_h of F can be written, following eq. (4), as

$$F_h = \sum_{m=0}^{M-1} a_m h(u_h,v_h,x_m,y_m) e^{j[u_h x_m + v_h y_m]}. \qquad (5)$$

Thus, even in the case when the spatial and spectral points (x_m,y_m) and (u_h,v_h), respectively, form a Cartesian grids, the samples of the FFP cannot be evaluated by a standard FFT since eq. (5) is not in the form of a DFT [28].

A. Far Field Pattern Computation by Optimized Matrix-Vector Multiplications

Whenever it is not possible to conveniently express the function $h(u,v,x_m,y_m)$, an effective way to evaluate the samples F_h of the FFP is employing OMVM routines.

Indeed, the kernel $h(u_h,v_h,x_m,y_m)\exp j[u_h x_m + v_h y_m]$ of eq. (5) can be arranged as a matrix $\underline{\underline{B}}$ whose generic element is

$$B_{hm} = h(u_h,v_h,x_m,y_m) e^{j[u_h x_m + v_h y_m]} \qquad (6)$$

so that eq. (5) can be recast as a matrix-vector multiplication

$$F_h = \sum_{m=0}^{M-1} B_{hm} a_m. \qquad (7)$$

Eq. (7) is amenable to be evaluated by OMVM routines, which in general perform as $O(M^2)$ or, in the case of particular symmetries of $\underline{\underline{B}}$, as $O(M\log^5 M)$ [25,26].

In the following, we illustrate a strategy capable of reducing the computational complexity needed to calculate F_h's in cases when the function $h(x_m,y_m,u,v)$ can be factored out.

B. Factorization of the Function $h(u,v,x_m,y_m)$

As long as $h(u,v,x_m,y_m)$ can be written (in an exact or approximate way) as

$$h(x_m,y_m,u,v) = \sum_{p=0}^{P-1} \varphi_p(u,v)\psi_p(x_m,y_m), \qquad (8)$$

then the FFP samples F_h can be calculated as

$$F_h = \sum_{p=0}^{P-1} \varphi_p(u_h,v_h) \left[\sum_{m=0}^{M-1} a_m \psi_p(x_m,y_m) e^{j[u_h x_m + v_h y_m]} \right]. \qquad (9)$$

Now, each inner summation of eq. (9) is in the form of a (possibly non-uniform, depending on the values of x_m, y_m, u_h, and v_h) DFT which can be computed, in the general case, by a NUFFT routine call, performing, as already stressed in the Introduction, as O(Mlog$M$). Consequently, the FFP samples F_h can be evaluated by P NUFFT calls, for an overall O(PMlog$M$) complexity.

Generally speaking, a simple way to obtain a factorization of h is to regard it as the kernel of a linear operator $\mathcal{A}$ so that the singular functions of $\mathcal{A}$, can be employed [29] as functions φ_p and ψ_p which then provide, when the summation in eq. (9) involves infinite terms, an exact representation of h. However, when truncating, such summation requires a high number of terms for an accurate representation, then the expansion of h can be obtained by selecting proper basis functions φ_p and ψ_p, depending on the features of h. In the following, we present a simple example, of relevant practical interest, concerning the factorization (8), for a proper choice of φ_p and ψ_p.

C. *p*-Series Factorization

In order to focus the attention on a case of practical interest, we consider an IA for which

$$h(u,v,x_m,y_m) = f(u,v) e^{jwz_m}, \qquad (10)$$

where $w = \sqrt{\beta^2 - (u^2+v^2)}$ and $f(u,v)$ is the element factor [12]. As prefigured at the beginning of Section II, h depends on the radiation characteristics of the m-th element through the

element factor p, and on its position through the quota z_m. For the sake of simplicity, in the following formulas we will skip the element factor, unessential in the discussion, and we will nevertheless deal with it throughout the numerical analysis.

Under this hypothesis, following the approach in [7, 15, 16] and on denoting by w_0 the value of w related to the main beam direction, eq. (4) can be rewritten as

$$F(u,v) = \sum_{m=0}^{M-1} a'_m \, e^{j[ux_m + vy_m + w'z_m]} \qquad (11)$$

with $w'=w-w_0$ and $a'_m = a_m \, exp[jw_0 z_m]$. For mild curvatures of S, the exponential $exp[jw'z_m]$ can be expanded by a truncated Taylor series up to the (P-1)-th order (p-series), so that

$$F(u,v) \cong \sum_{p=0}^{P-1} \frac{(jw')^p}{p!} \sum_{m=0}^{M-1} z_m^p a'_m e^{j[ux_m + vy_m]} \qquad (12)$$

and the discrete values F_h of F can be expressed as

$$F_h = \sum_{p=0}^{P-1} \frac{(jw'_h)^p}{p!} \sum_{m=0}^{M-1} z_m^p a'_m e^{j(u_h x_m + v_h y_m)}. \qquad (13)$$

Obviously, the smaller the curvature of S, the smaller the value of P required for a given accuracy. In practice, a proper value for P can be chosen to trade off the computational burden and the accuracy of the approach, as it will be clearer in Subsection II.e and in the numerical analysis presented in Section IV. In general, the number of p-series terms is chosen in a way to ensure that the argument of exp[$jw'z_m$] is less than a "small" value all over the spectral (u,v) region of interest. Such a value is typically assumed equal to $\pi/8$, or even lower if more accurate results are desired. Moreover, the chosen number of p-series terms depends also on the coverage, so that, once the array and the coverage are given, the number of p-series terms can be consequently assigned. We stress that the inner summation in eq. (13) is not in the form of a standard DFT [28], so that, to recover the desired computational complexity, a NUFFT structure should be employed.

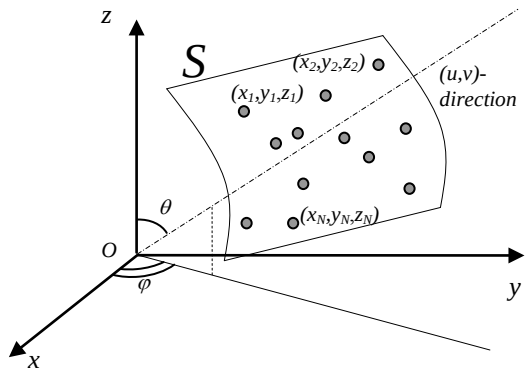


Fig. 5. Geometry of the problem.

D. Use of the NUFFT

Concerning the fast evaluation of each inner summation in eq. (13), a different NUFFT algorithm should be considered depending on the spatial and spectral grids at hand. More in detail:

a) for an arbitrary spatial grid (x_m,y_m) (aperiodic conformal array) and for a regular, Cartesian spectral lattice (u_h,v_h), a Non-Equispaced Data (NED), or "type-1", NUFFT is of interest [17];
b) for a regular, Cartesian spatial grid (periodic conformal array) and for an arbitrary spectral one, a Non-Equispaced Result (NER), or "type-2", NUFFT should be employed [17];
c) for arbitrary spatial and spectral grids, a "type-3" NUFFT should be adopted [18].

Since cases b) and c) are extensions of case a) which do not add any conceptual difficulty, in this paper we assume to evaluate the FFP on a regular, Cartesian spectral lattice, so that NED-NUFFTs are of interest (case a)). This is the most frequently occurring case, since, in antenna synthesis, the design specifications are usually given on a regular, Cartesian spectral lattice (u_h,v_h), leading indeed to the use of NED NUFFTs. Accordingly, for the sake of brevity, cases b) and c) will not be dealt with in the details.

Different approaches have been proposed in the literature for evaluating NUFFTs [17,18,30-33].

The main idea underlying many NUFFT algorithms is to approximate the non-uniform exponential function exp($jp\Delta ux_m$) (having assumed that the h-th spectral point (u_h,v_h) corresponds to the ($p\Delta u$,$q\Delta v$) uniform spectral grid point), by interpolating L, "oversampled", properly chosen and windowed uniform exponentials exp($jp\Delta ul\Delta x$), l=1,..,L. Accordingly, non-

uniformly sampled exponentials can be approximated by properly weighted sums of uniform exponentials, enabling to exploit a finite number of standard FFT routine calls. It is worth noting that this strategy is not equivalent to a "brute-force thinning" of an array which, on the contrary, requires significantly denser element grids [34]. In this paper, we use the approach in [17], based on an "exact" representation of the exponentials $\exp(jp\Delta u x_m)$. For the reader's convenience, we quote Appendix A for a brief mathematical description of the employed NUFFT algorithm.

E. Sub-Array Partitioning

It should be mentioned that the array can be also partitioned into N sub-arrays, each one made up of m_n elements, such that $m_0+ m_1+... m_{N-1}=M$. Accordingly, eq. (13) can be rewritten by explicitly describing the radiation by each array portion, thus leading to

$$F_h = \sum_{n=0}^{N-1}\sum_{p=0}^{P-1} \frac{(jw_h^{\cdot})^p}{p!} \sum_{m=m_n}^{m_{n+1}} z_m^p a_m^{\cdot} e^{j(u_h x_m + v_h y_m)}. \quad (14)$$

As an advantage, the number P of terms involved to represent the exponential in (10) by a Taylor series expansion associated to each subarray is expected to reduce, for a fixed accuracy. Accordingly, the strategy can be applied to non-mild shapes, also to reduce model errors related to the vector aspects (i.e., model errors in the assumption (3)). To better enlighten this advantage, we mention the borderline case of a faceted array (or a faceted reflectarray [35-37]). In this case, across the junctions between the facets, the curvature is singular. Nevertheless, a partitioning into subarrays enables accurate computations with a number of P=1 terms for each facet (see also Subsection IV.c). Moreover, the sub-array partitioning strategy is further facilitated by the use of type-2 (for non-aperiodic arrays) or type-3 (for aperiodic arrays) NUFFT routines. Indeed, even when dealing with a uniform array, the field radiated by the various facets should be computed onto the common (u,v) grid associated to the overall antenna, a procedure requiring in general time-consuming interpolation stages. From this point of view, the opportunity of employing (type-2 or type-3) NUFFT routines, enabling arbitrary (u,v) output grids, offers the possibility of performing such an interpolation with $O(M\log M)$ complexity. In Section IV, we discuss how much convenient such a strategy can be.

Finally, the sub-array approach is amenable to a multi-level implementation [38], but, for the sake of simplicity, in this paper we will deal with a single-level one.

III. IMPLEMENTATION OF THE ALGORITHMS

The approach proposed in Subsections B-E has been implemented in both, a sequential code, running on conventional computing architectures (single-core CPU), and in a parallel code, taking advantage of GPU acceleration. Moreover, sequential and parallel implementations of an approach based on OMVM routines according to eq. (7) have been also setup to serve as a benchmark for the performance of the proposed approach.

For both, the sequential and parallel codes, particular care has been devoted to

- selecting high performance FFT routines, as required by the proposed, NUFFT-based approach;
- choosing high performance Matrix-Vector multiplication routines, as required by the OMVM-based scheme.

In the following, some implementation details concerning the developed sequential and parallel codes will be discussed. We remark that symbols in the following are defined in the Appendices A, B and C.

A. Sequential Implementations

All the sequential codes have been developed in ANSI C language. Such a choice is due to the use of the CUDA environment to develop the parallel counterpart. Indeed, a CUDA program consists of "phases" that are executed on the host (CPU) or the device (GPU) and of data structures that can be allocated on the host or the device, as well (see [24]). The host code is straight ANSI C code. The device code is ANSI C code, extended with special keywords for calling data-parallel functions (*kernels*), and managing the associated data structures. Accordingly, the development of a parallel code can be performed by starting with the sequential ANSI C code, spotting the phases that should be parallelized, and extending the corresponding instruction and data structures with

the special keywords for parallel executions provided by CUDA.

In particular, concerning the sequential code:

- the NUFFT algorithm has been implemented according to [17] (see also Appendix A); a particularly fast implementation of the FFT, based on the same philosophy of FFTW [39], and contained in the Intel Math Kernel Library (MKL) [40], has been exploited to speedup the required FFT calculations; a C-style listing of the algorithm is reported in Appendix B; the critical point of the algorithm is represented by the U matrix filling operations, performed within three, nested (m, l_1 and l_2) *for loops*; such a filling is "pseudo-random" (i.e., it does not obey to a "row-major" filling criterion [41]) since the indices i_x and i_y "jump" between non-consecutive values as long as m, l_1 and l_2 are swept; as known, this severely affects the memory latency when accessing the elements of U [41];
- the implementation of the OMVM-based approach relies on BLAS routines;

For both the cases, Intel Math Kernel Libraries (MKL) (v.1.0.02), including BLAS and FFT routines, have been exploited.

B. Parallel Implementations

As already stressed in the Introduction, all the parallel codes have been developed by means of the CUDA language [24].

For both, the NUFFT-based and OMVM-based algorithms, the GPU is exploited as an accelerating device, executing portions of the code in parallel [19]. More in detail:

- for the proposed approach, in correspondence to each NUFFT call, the execution is delivered to the GPU and the evaluation of each NUFFT performed by a proper, parallel implementation of the scheme detailed in Appendix A;
- for the OMVM-based approach, the evaluation of the matrix multiplication required by eq. (7) is performed, again through a parallel implementation on the GPU;

In the sequel, some details concerning the parallel implementations of the two considered approaches will be reported.

1. NUFFT-based approach

All the stages of Appendix A have been carefully examined and parallelized, according to the key rules of GPU programming [24]. A CUDA-style listing of the algorithm is reported in Appendix C. More in detail:

Stage 1

The calculations of the samples of the spatial and spectral windows Φ and $\hat{\Phi}$, respectively, as well as of the indices $\mu_{x,m}$ and $\mu_{y,m}$ (see Appendix B) are fully independent from each other and are evaluated in parallel, rather than by *for loops* as in the sequential case.

The computation of the U matrix is also parallel, but requires some more care, since different approaches could be envisaged to this end and the best performing one should be selected. Indeed, due to the already remarked "pseudo-random" access to U required by the sequential implementation, devising an efficient filling procedure in the parallel case is not straightforward and represents the main difficulty to be solved throughout the parallelization of the whole code described in Appendix B.

A first possible parallelization strategy would be to commit a thread to compute a single matrix element of U. However, in this way, the generic thread should perform, due to the "pseudo-random" filling, a time-consuming browsing of the input elements to establish whether they contribute to the committed element of U or not.

As an alternative, the implemented parallel code employs a 1D block grid of length M, each block allocating $(2K+1)\times(2K+1)$ threads. In this way, the above mentioned browsing is avoided since each thread is assigned to a different input element and updates the corresponding element of the U matrix.

Generally speaking, the number of allocable blocks in a 1D grid depends on the computing capability of the employed GPU [24]. For the GPU employed in this paper, the number of allocatable blocks is 65535, which is large enough for all the considered numerical tests. For arrays with M>65535, the algorithm should foresee a sequential allocation of 1D block grids. Since the maximum number of allocatable blocks depends on the employed GPU, the actual performance of

the algorithm depends on the hardware performance of the available graphic card.

However, it should be noticed that, by this solution and regardless to M, more than one thread may need to simultaneously update (namely, read, compute and store a new value) the same element $U(i_x,i_y)$. Unfortunately, when this happens, a conflict such as Writing After Writing (WAW) and Writing After Reading (RAW) [41] can occur, affecting the results. To preserve the integrity of the data, atomic operations have been exploited [41,42], which basically ensure the semantic correctness of the algorithm through a serialization of the updating operations. We finally observe that, the parallel implementation is such that two threads belonging to the same block never update the same element $U(i_x,i_y)$ (although threads belonging to different blocks can do). Moreover, since each block is $(2K+1)\times(2K+1)$ sized and, generally speaking, K is usually "small" (typical values range from 6, for single precision arithmetic, to 12, for double precision [17]), in order to speed-up the memory access, the updating operations are performed first on a temporary $(2K+1)\times(2K+1)$ matrix, allocated in the shared memory (and then shared by threads belonging to the same block), and subsequently on the global memory by the mentioned atomic operations.

Stage 2

The computation of the required FFT has been parallelized by means of the latest release of the cuFFT library (cuFFT v2.3) [43], implementing several, optimized parallel FFT algorithms, and choosing the one to be used depending on the shared memory occupation of the input array and on the possibility of reducing its size to a power of an integer factor.

Stage 3

Extracting the elements of the $\hat{U}$ matrix, their scaling with the elements Φ_{pq} and the subsequent memory updates are independent, and then easily parallelizable, operations.

We finally remark that, throughout the parallel implementation of the NUFFT routine, the typical suggested guidelines in programming GPUs [24] and concerning, for example,

- avoiding divergencies due, f.i., to conditional statements or non-coalesced memory accesses;
- balancing the computational load among the available resources;

have been applied.

Furthermore, data padding [24] has been adopted to manage a generic input data size. It should be noticed that, for the considered case, data padding does not significantly affect the algorithm performance since the amount of employed padding is always less or equal to the block size (which, as above discussed, contains $(2K+1)\times(2K+1)$ only threads) and, as such, negligible for a large input data size M.

2. OMVM-based approach

The implementation of the OMVM-based approach relies on the latest release of the cuBLAS (cuBLAS v.2.3) routines [44].

Also for this case, data padding has been applied.

3. Multilevel parallelization

It is worth noting that, the particular expression in eq. (13) is amenable to a further level of parallelization, since the different terms of the p-series summation can be simultaneously computed and, in turn, each NUFFT can be parallelized according to the guidelines above. Unfortunately, a single GPU cannot handle more kernels simultaneously and hence cannot effectively manage a multi-level, parallel computation.

Nevertheless, with a multi-GPU system [45], the computation of each term of the p-series can be executed by a different GPU accomplishing, in turn, the computation of a parallel NUFFT. Afterwards, all the terms can be added together by means of a reduction operation on a "master" processing unit. This strategy allows operating a two-level parallelism, one to compute the p-series and one to compute the NUFFTs.

Similar considerations apply also to the sub-array partitioning approach (see eq. (14)). Indeed, also in this case the computations for each sub-array are independent from all the others and a two-level parallelism can be obtained. Obviously, in this case, a three-level parallelism can even be achieved, by exploiting the independence of the sub-arrays and of the p-terms.

In this paper, only results concerning a single-level parallelization are shown. The multi-level case is left to future developments since it does not introduce any conceptual difficulty, but for communication protocols between the GPUs [45].

It should be finally observed that, with reference to the parallel implementation of the proposed NUFFT-based approach, the order of computation among the different *p*-series terms, or, in other words, the order of computation of the different required NUFFT routine calls, could be rearranged to simultaneously execute more than one NUFFT on the same GPU. In this way, apparently a multi-level parallelism could be achieved on a single GPU. However, if ever such a solution would be more effective, it should not be considered of practical interest since it would not comply with the typically required *transparent scalability* on a multi-scale architecture [24].

IV. COMPUTATIONAL PERFORMANCE

In this Section, we present a numerical analysis showing the computational performance of all the developed algorithms. More in detail, after having illustrated the hardware setup employed for the tests, a comparison between the computational performance of the CPU and the GPU implementations of the NUFFT-based and OMVM-based algorithms are reported. Finally, the trade-off between computational performance and accuracy of the *p*-series and sub-array partitioning approaches is discussed.

A. Hardware Setup

Sequential implementations have been run on a personal computer with a single-core, Intel Pentium IV processor, with 3 GHz of clock frequency, and equipped with a RAM, 2.0 GBytes sized.

Parallel CUDA codes have been executed on the same personal computer used for the sequential tests, but powered by a GeForce GTX 260 GPU, having 24 multi-processors working at 800 MHz and equipped with a memory, 872 MByte sized.

B. Computational Performance of the Implemented Algorithms

Fig. 6 reports a comparison of computing times for the FFP evaluation by all the implementations discussed in Section III versus the size M of the IA. More in detail, the computing time has been normalized, for all the algorithms, to the corresponding one concerning the case M=80. As it can be seen, the two GPU implementations outperform the corresponding CPU ones, and, in particular, the NUFFT-based implementation on GPU ensures the lowest growth rate. It is worth noting that, the considered GPU computing times (here and in the following) include transfers from/to host (PC memory) to/from device (GPU global memory), so that they represent the effective speed-ups that the GPU can provide against the CPU architecture. In Fig. 6, the M^2 and $M\log M$ trends, agreeing with those for the two considered CPU-based algorithms, are also depicted for higher values of M. Finally, some relevant speed-ups are summarized in Table 1.

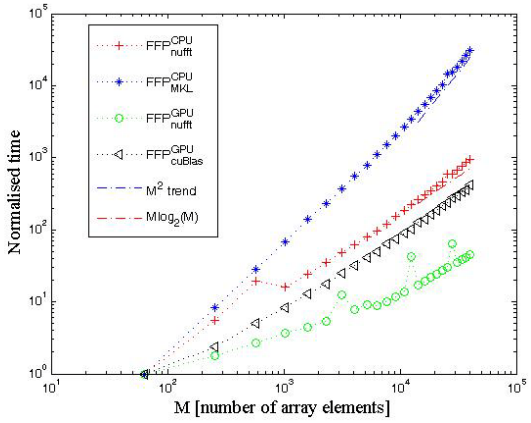


Fig. 6. Growth rates of the computing times for the FFP evaluation by all the implementations illustrated in Section III.

Table 1: Speed-ups among different implementations for an array with M=10^4.

Implementations	Speed-up
CPU NUFFT vs. OMVM	>10
GPU NUFFT vs. OMVM	8
NUFFT GPU vs. CPU	>40

In Fig. 7, the speed-up of the GPU implementation as compared to the CPU one (CPU computing time / GPU computing time) for the NUFFT-based approach against the size *M* of the input data is depicted. As it can be seen, the improvement in the performance for the GPU computation is significant already for small sized arrays (less than 100 elements) and the speed-up factor grows dramatically with the increasing IA dimension *M*.

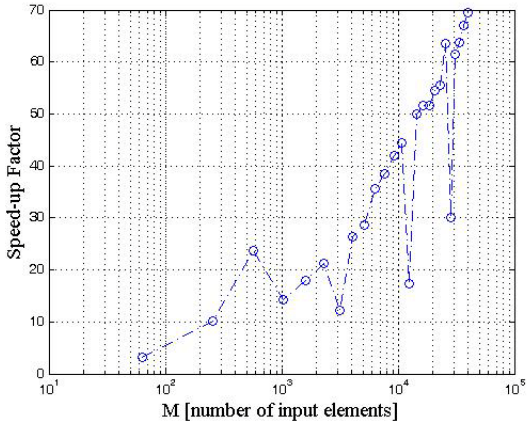


Fig. 7. Speed-up of the GPU vs. CPU implementations for the NUFFT-based approach.

In order to explain such a speed-up, a code profiling has been performed, enlightening that the improved performance is essentially due to the critical filling of the matrix *U* of the sequential case and to the employed effective solution in its parallelization.

We finally note that, the relative drops of GPU/CPU performance are due to particular sizes of the input elements whose data structure does not effectively fit the characteristics (number of shared registers, constant memory size, number of allocatable blocks, number of processors) of the employed hardware. As a consequence, for such particular input dimensions, the code execution is not as massively parallel as it occurs for the others. Nevertheless, the GPU still guarantees a significant speedup as compared to the CPU.

C. Computational Burden of the *p*-Series and of the Sub-Array Partitioning Strategy

We now aim at briefly clarifying, for the sequential implementation, the conditions under which the sub-array partitioning strategy (eq. (14)) becomes computationally convenient with respect to the only *p*-series approach (eq. (13)).

The computational burdens of eqs. (11) and (14) are reported and compared in Appendix D. As it can be expected, it turns out that (eq. (D3)), for sequential implementations, the sub-array partitioning becomes convenient as long as it "favorably" exchanges *p*-series terms with sub-arrays.

Obviously, these conclusions do not hold true when a multi-level parallelism is employed since, in this case, the computation time can be reduced by a *p*-series/sub-array partitioning approach despite the higher number of operations.

A remarkable case when the sub-array partitioning becomes convenient is that (already mentioned) of faceted arrays [35-37], i.e., when the surface *S* is made up by contiguous planar portions (facets) with different relative inclinations. In this case, each facet can be associated to a sub-array which, being flat, requires just 1 *p*-series term. To enlighten this point, we have considered the case of a faceted array having 3 facets and *M*=19600 elements. Tab. 2 summarizes the speed-ups obtained by the sub-array partitioning strategy as compared to *p*-series only evaluations of the FFP, as a function of *P*.

Table 2: Comparing the computational performance of sub-array partitioning vs. *p*-series.

# *p*-series terms	Speed-up
3	1
4	1.34
5	1.67
6	2
7	2.34

V. ACCURACY

In this Section, we present a numerical analysis illustrating the accuracy of the proposed, NUFFT-based strategy, by focusing the attention on two examples: a linear, aperiodic and parabolic, aperiodic arrays.

A. Linear, Aperiodic Array

Let us begin with a linear (non-conformal), aperiodic array. More in detail, we consider an equivalently tapered Chebyshev, 1D array [6,8], made of 2048 elements having uniform

excitations. The elements positions (see Fig. 8) have been properly determined, according to the approach in [6,8], in order to synthesize the same pattern of a Chebyshev array of 1024 uniformly spaced elements having a side lobe level of -26dB. The resulting inter-element spacing of the array elements varies from a minimum of 0.33λ to a maximum of 1.8λ, while the overall array size is 1000λ.

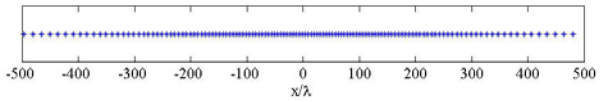


Fig. 8. Elements positions for the Chebyshev array. For the sake of clarity, only the position of one element every 16 are shown.

The adopted synthesis algorithm is based on the optimization of a proper objective functional and requires direct evaluations of the FFP, which have been performed by means of the proposed approach. Obviously, the array being linear, only 1 *p*-series term is required.

Figure 9 shows the synthesized FFP of the equivalently tapered Chebyshev array. The computations have been sequential and the reported "exact evaluation" has been performed by the OMVM approach. The computing times for the proposed and OMVM approaches have been 35ms and 62ms, respectively.

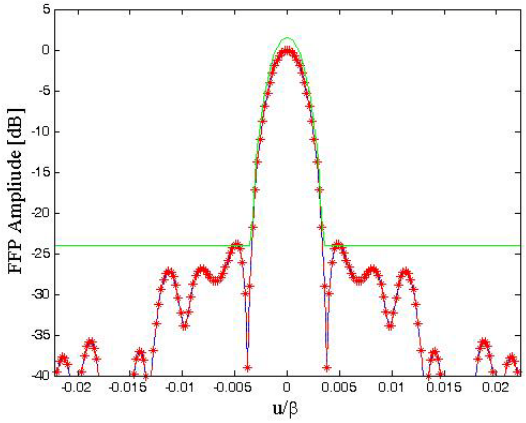


Fig. 9. *u* cut of the FFP of the synthesized equivalently tapered, 1D Chebyshev array. Red stars: proposed approach. Blue dashed line: exact evaluation.

B. Parabolic, Aperiodic Array: Accuracy of the *p*-Series and Sub-Array Partitioning Strategies

In this Subsection, we highlight, with reference to the case of a parabolic, aperiodic array, the accuracy of the *p*-series approach versus the array surface curvature, and the improvements in the accuracy provided by the sub-array partitioning strategy.

To this end, we consider two IAs having the same number (i.e., 65388) of elements lying on two parabolic surfaces having the same diameter (D) but different focal length (f). In particular, the first IA, say IA_1, has a focal/diameter ratio (f/D) equal to 1, while the other, say IA_2, has f/D equal to 1.5. Under these hypotheses, the curvature of IA_2 is smoother than that of IA_1 (see Fig. 10).

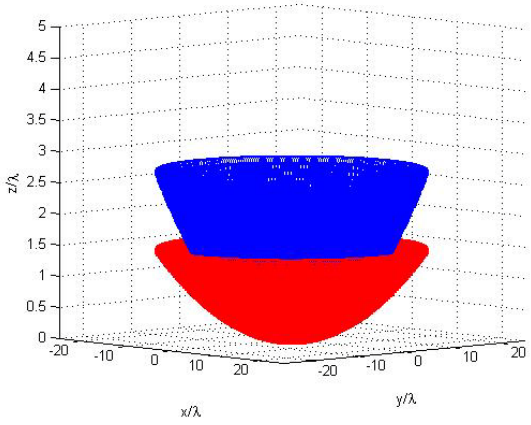


Fig. 10. The two considered parabolic IAs for the analysis of the *p*-series and sub-array partitioning accuracies. Blue: IA_1. Red: IA_2.

The histogram in Fig. 11 indicates the Root Means Square (RMS) error between the exact evaluations of the FFPs for IA_1 and IA_2 (eq. (7)) and their computations with the proposed approach (eq. (14)), against the number of considered *p*-series terms and sub-arrays. Assuming, as acceptable accuracy, the one corresponding to a RMS equal to 1%, Fig. 11 shows that if no partitioning is adopted for IA_1, even six *p*-series terms are not enough to attain the desired precision. On the contrary, partitioning IA_1 into 16 sub-arrays ensures the desired accuracy already with 5 terms and splitting up further the array into 64 or 256 sub-arrays reduces the number of required *p*-series terms to 4 or 3, respectively. Concerning now IA_2, Fig. 11 shows that its milder

curvature gives rise to a faster *p*-series convergence with respect to IA_1. Indeed, 4 *p*-series terms now ensure 1% of RMS error even without sub-array partitioning. Introducing the partitioning in this case allows reducing the *p*-series terms to 3 or 2 with 16 or 256 sub-arrays, respectively.

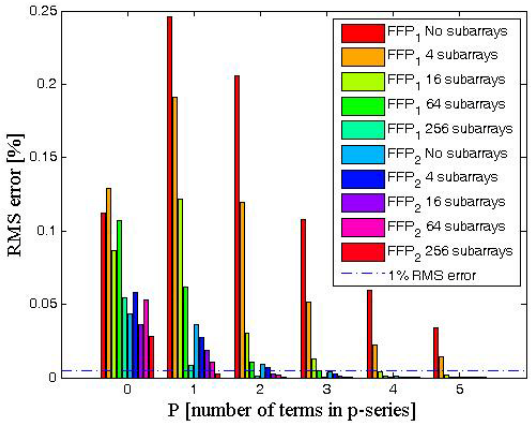


Fig. 11. RMS errors when computing the FFPs of IA_1 and IA_2 against number of *p*-series terms for different numbers of sub-arrays. FFP_1 refers to IA_1, while FFP_2 refers to IA_2.

C. Accuracy of the *p*-Series Approach Versus the Degree of Aperiodicity

We finally consider the case of a 2D IA, whose 16384 elements are aperiodically distributed on a paraboloid with focal length/diameter ratio equal to 0.8, a typical value in the applications. The inter-element spacing varies from 0.35λ to 0.9λ, while the excitation coefficients have been chosen according to:

$$a_m = e^{\alpha\cdot(F/d_m)-1} e^{-i*\beta d_m}, \quad m = 0,1,...,M-1 \qquad (15)$$

where d_m is the distance of the *m*-th array element from the foci, • is the wave-number and α has been properly determined to obtain an amplitude tapering of -4dB at the edge. 3 *p*-series terms have been considered, ensuring a negligible RMS error (~10^{-5}) in the visible region [*0.4*,*0.4*]x[*0.4*,*0.4*] $\bullet^2$ of the (*u*,*v*) plane.

Figure 12 compares the FFP evaluation of the considered IA by means of the proposed approach to the exact evaluation (eq. (7)).

The robustness of the proposed approach versus the "degree of aperiodicity" of the array is illustrated in Table 3 which reports the RMS error of the FFP evaluation when an increasing random fraction of array elements is erased, as compared to the setting of Fig. 12, thus increasing the degree of aperiodicity. It should be mentioned that, as long as an increasing random fraction of array elements is erased, the sidelobe intensity rises up, which leads to the higher RMS in Tab. 3. This could be however mitigated by an increasing number of *p*-series terms.

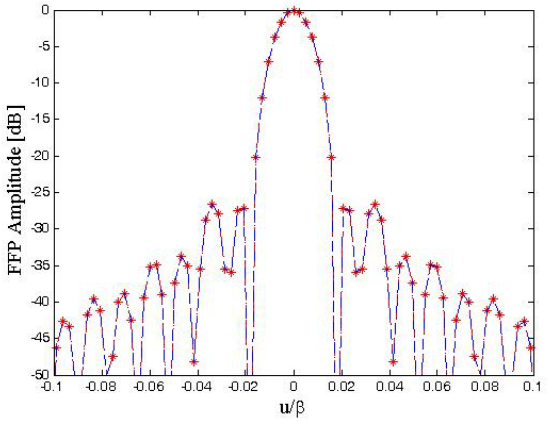


Fig. 12. *u* cut of the FFP of the parabolic, aperiodic array. Red stars: proposed approach with 3 *p*-series terms. Blue dashed line: exact evaluation.

Table 3: RMS vs degree of aperiodicity.

RMS error	Erased elements [%]
10^{-8}	0
2.9^{-3}	1
1.4^{-1}	5
0.27	10

VI. CONCLUSIONS

An approach for the fast analysis of IAs based on the use of the *p*-series expansion and NUFFT routines has been proposed and implemented in both, sequential (CPU) and parallel (GPU) codes.

The performance of the algorithms has been analyzed both in terms of computational efficiency and of achievable accuracy.

In particular:

- both, the sequential and parallel, NUFFT-based approaches are capable of improved performance as compared to (sequential and parallel) algorithms based on OMVMs;

- a sub-array partitioning approach can further reduce the computational burden by speeding-up the convergence of the *p*-series;
- a proper parallel code implementation enables GPU computing to significantly speed-up the execution as compared to that on CPU.

We finally remark that, some of these results can be extended to the FG computation in synthesis algorithm and to the case of volumetric (3D) IAs. Concerning array synthesis, it should be mentioned that often multi-stage synthesis approaches are employed as in [46] and that the most computationally demanding stages can strongly benefit of calculating FFP and FG by the approach here above proposed, committing the computation according to more sophisticated vector models just to the last synthesis steps.

APPENDIX A: THE NUFFT ALGORITHM

According to [17], the "exact" representation of the exponential function $\exp(jp\Delta u x_m)$ is the following:

$$e^{jp\Delta u x_m} = \frac{(2\pi)^{-1/2}}{\Phi(p\Delta u/c)} \sum_{l_1 \in Z} \hat{\Phi}(cx_m - l_1) e^{jp\Delta u l_1/c} \quad \text{(A1)}$$

where $c > 1$ is an "oversampling factor", Φ is a C_0^∞ function with support in $[-\pi,\pi]$ and strictly positive in $[-\pi/c,\pi/c]$, and $\hat{\Phi}$ is its Fourier transform.

Following eq. (A1), any of the NUFFTs in eqs. (13) or (14) can be rewritten as

$$\tilde{b}_{pq} = \sum_{m=0}^{M-1} b_m e^{j(x_m p\Delta u + y_m q\Delta v)} = \underbrace{\frac{(2\pi)^{-1}}{\Phi(p\Delta u/c)\Phi(q\Delta v/c)}}_{\text{Step 3}} \underbrace{\underbrace{\sum_{l_1,l_2 \in Z} \underbrace{\left\{ \sum_{m=0}^{M-1} b_m \hat{\Phi}(cx_m - l_1)\hat{\Phi}(cy_m - l_2) \right\}}_{U(l_1,l_2)\,(\text{Step 1})} e^{jp\Delta u l_1/c} e^{jq\Delta u l_2/c}}_{\text{Evaluate by a standard 2D FFT (Step 2)}}}$$

(A2)

where $b_m = z_m^p a_m'$. Henceforth, we assume to be interested in the values of $\tilde{b}_{pq}$ for p=-N_1/2,…, N_1/2 and q=-N_2/2,…, N_2/2.

The sums over l_1 and l_2 in eq. (A2) require the computation of standard 2D, FFTs. Furthermore, they can be effectively evaluated provided that $\hat{\Phi}$ is small outside some interval [-K,K], so that it is required that Φ has compact support in [-α,α] and $\hat{\Phi}$ is concentrated, as much as possible, in [-K,K]. To this end, a Kaiser-Bessel window Φ is used [17].

The computation of eq. (A2) can be divided into three stages (see also [17]).

Stage 1

For each (x_m,y_m), the nearest equispaced spatial frequencies l_1/c and l_2/c are determined. The samples of the windowing/interpolating functions Φ and $\hat{\Phi}$, respectively, are computed. The inner m summation in eq. (A2), that is, the $U(l_1,l_2)$ function, is calculated.

Stage 2

A standard, 2D FFT routine is performed on U. The output matrix $\hat{U}$ has size $cM \times cM$.

Stage 3

$\hat{U}$ is reduced to an $N_1 \times N_2$ matrix and then scaled with the windowing function $(2\pi)^{-1}/\Phi(p\Delta u/c)\Phi(q\Delta v/c)$.

APPENDIX B: C-STYLE LISTING OF THE SEQUENTIAL NUFFT ALGORITHM

```
//*********
//* STEP 1 *
//*********

for(p=-N1/2;p<N1/2;p++) {
   for(q=-N2/2;q<N2/2;q++) {

   Φpq=Φ(2πp/(cN1))Φ(2πq/(cN2));
                              }
                           }

   for (m=0;m<M;m++) {

      μx,m=round(c*xm);
      μy,m=round(c*ym);

      for (l1=-K;l1<=K;l1++) {

         ix=mod(μx,m +l1+c*N1/2,c*N1);
```

```
            w_xm=Φ̂(c*x_m-(μ_x,m+l_1));

            for(l_2=-K;l_2<=K;l_2++){

               i_y=mod(μ_y,m+l_2+c*N_2/2,c*N_2);
               w_ym=Φ̂(c* y_m-(μ_y,m+l_2));
               U[i_x,i_y]=U[i_x,i_y]+w_xm*w_ym*b_m;
                                    }
                                 }
                           }

//*********
//* STEP 2 *
//*********

Û=fft(U); // 2D fft routine provided by MKL


//*********
//* STEP 3 *
//*********

for (p=0;p<N_1;p++){
   for(q=0;q<N_2;q++) {

      nufft[p,q]=div(u[(c-
1)*N_1/2*c*N_2+p*c*N_2+(c-1)*N_2/2+q],Φ_pq);
                        }
                     }
```

APPENDIX C: CUDA-STYLE LISTING OF THE PARALLEL NUFFT ALGORITHM

```
//*********
//* STEP 1 *
//*********

/* Generates a 1D grid of threads to evaluate
μ_x,m and μ_y,m. NUM_THREADS = # threads per block
*/
dim3 dimGrid_mu(M/NUM_THREADS,1);
dim3 dimBlock_mu(NUM_THREADS,1);

// Parallel evaluation of μ_x,m and μ_y,m
data_round<<<dimGrid_mu,dimBlock_mu>>>(x_m,y_m,μ_x,
m,μ_y,m);

// Generates a 2D grid of threads to evaluate
Φ_pq
dim3 dimGrid_phi(N_1/BLOCK_SIZE, N_2/BLOCK_SIZE);

// Parallel evaluation of Φ_pq

dim3 dimBlock_phi(BLOCK_SIZE,BLOCK_SIZE);
Φ<<<dimGrid_phi,dimBlock_phi>>>(Φ_pq,N_1,N_2);

/* Generates a 2D grid of threads to evaluate
w_xm and w_ym and evaluates those quantities */

dim3 dimGrid_phi_hat(M,1);
dim3 dimBlock_phi_hat(2*K+1,1);
Φ̂<<<dimGrid_phi_hat,dimBlock_phi_hat>>>(w_xm,x_m
,μ_x,m,M);
Φ̂<<<dimGrid_phi_hat,dimBlock_phi_hat>>>(w_ym,y_m
,μ_y,m,M);

// Generates a 1D grid of threads and
evaluates U
dim3 dimBlock_u(2*K+1,2*K+1);
dim3 dimGrid_u(M,1);
U_matrix_evaluation<<<dimGrid_u,
dimBlock_u>>>(b_m,M,μ_x,m,μ_y,m,U,w_xm,w_ym,N_1,N_2);

//*********
//* STEP 2 *
//*********
Û=cuFFT(U);

//*********
//* STEP 3 *
//*********

dim3           dimGrid_scaling(N_1/BLOCK_SIZE,N_2
/BLOCK_SIZE);
dim3 dimBlock_scaling(BLOCK_SIZE,BLOCK_SIZE);
scaling<<<dimGrid_scaling,dimBlock_scaling>>>(
Φ_pq);
```

APPENDIX D: COMPUTATIONAL BURDENS OF THE *p*-SERIES AND SUB-ARRAY PARTITIONING APPROACHES

In the un-partitioned case, according to eq. (13), the number of operations needed to determine the FFP, say N_o, is (neglecting summation operations as compared to multiplications)

$$N_o = M\left\{P\left[4.5c^2\log_2(c^2M)+20K^2+3\right]+2\right\}, \quad \text{(D1)}$$

where K and c are the NUFFT oversampling factor and interpolation length, respectively, and the complexity for the evaluation of a single NUFFT has been determined according to [17].

When the surface is partitioned into N_{sub}• M sub-arrays, the computational complexity becomes (neglecting again summation operations as compared to multiplications)

$$N_o^{sub} = M\cdot\left\{P'\cdot\left[N_{sub}\cdot(4.5c^2\log_2(c^2M)+20K^2+2)+1\right]+N_{sub}+1\right\} \quad \text{(D2)}$$

where P'• P is the number of p-series terms needed in eq. (17) to achieve the same accuracy as for the un-partitioned case. Dividing (D2) by (D1) and enforcing that the ratio is less than one, we have:

$$N_o^{sub} \Big/ N_o \le 1 \Rightarrow$$
$$\Rightarrow N_{sub} \le \frac{9c^2 P \log_2 c + 20K^2 P + \Delta P + 1}{1 + (20K^2 + 9c^2 \log_2 c)(P - \Delta P)} \quad \text{(D3)}$$

where $P'=P-\bullet P$. Eq. (D3) provides a necessary (but not sufficient) condition, in terms of number of sub-arrays N_{sub}, for the sub-array partitioning approach to be computationally convenient as compared to the un-partitioned case, for a fixed accuracy. Obviously, in eq. (D3), $\bullet\theta \quad P<P$ since the partitioning can reduce the number of p-series terms at most to $P'=1$. Generally speaking, $\bullet P$ is a function of N_{sub} and it increases with the number of sub-array partitions.

To be more specific we observe that, typically, the values of the NUFFT oversampling factor and interpolation length are 2 and 6, respectively. Substituting these values in eq. (D3), we get:

$$N_{sub} \le \frac{756P + \Delta P + 1}{1 + 756(P - \Delta P)} \cong \frac{P}{(P - \Delta P)} \le P \,. \quad \text{(D4)}$$

REFERENCES

[1] O.M. Bucci, G. D'Elia, G. Mazzarella, and G. Panariello, "Antenna pattern synthesis: a new general approach", *Proc. of the IEEE*, vol. 82, no. 3, pp. 358-371, Mar. 1994.

[2] A. Capozzoli and G. D'Elia, "Global optimization and antennas synthesis and diagnostics, part one: concepts, tools, strategies and performances", *Progr. Electromagn. Res. PIER*, vol. 56, pp. 195-232, 2006.

[3] A. Capozzoli and G. D'Elia, "Global optimization and antennas synthesis and diagnosis, part two: applications to advanced reflector antennas synthesis and diagnosis techniques", *Progr. Electromagn. Res. PIER*, vol. 56, pp. 233-261, 2006.

[4] L. Josefsson and P. Persson, *Conformal array antenna theory and design*, J. Wiley & Sons., New York, 2006.

[5] G. Caille, I. Lager, L.P. Ligthart, C. Mangenot, A.G. Roederer, G. Toso, and M.C. Viganò, "Aperiodic arrays for multiple beam satellite applications," *Proc. of the 11th Int. Symp. on Microw. Opt. Tech.*, pp. 419-422, Dec. 2007.

[6] A. Capozzoli, C. Curcio, G. D'Elia, A. Liseno, and P. Vinetti, "FFT & aperiodic arrays with phase-only control and constraints due to super-directivity, mutual coupling and overall size", *Proc. of the 30th ESA Antenna Workshop on Antennas for Earth Observ., Science, Telecomm. and Navig. Space Missions*, May 2008.

[7] A. Capozzoli, C. Curcio, G. D'Elia, and A. Liseno, "Fast power pattern synthesis of conformal reflectarrays", *Proc. of the IEEE Antennas Prop. Symp.*, pp. 1-4, July 2008.

[8] A. Capozzoli, C. Curcio, G. D'Elia, A. Liseno, and P. Vinetti, "FFT & equivalently tapered arrays", *Proc. of the XXIX URSI General Assembly*, Aug. 2008.

[9] J.H. Doles III and F.D. Benedict, "Broad-band array design using the asymptotic theory of unequally spaced arrays," *IEEE Trans. Antennas Prop.*, vol. 36, no. 1, pp. 27-33, Jan. 1988.

[10] A. Akdagli and K. Guney, "Shaped-beam pattern synthesis of equally and unequally spaced linear antenna arrays using a modified tabu search algorithm," *Microw. Optical Tech. Lett.*, vol. 36, no. 1, pp. 16-20, Jan. 2003.

[11] J. Huang, M. Lou, A. Feria, and Y. Kim, "An inflatable L-band microstrip SAR array", *Proc. of the IEEE Antennas Prop. Int. Symp.*, pp. 2100-2103, Jun. 1998.

[12] R.E. Collin, *Antennas and radiowave propagation*, McGraw-Hill, New York, 1985.

[13] A.W. Rudge, K. Milne, A.D. Olver, and P. Knight, *The Handbook of Antenna Design Vol. 2*, London, Peter Peregrinus, 1983.

[14] J.W. Cooley and J.W. Tukey, "An algorithm for the machine calculation of complex Fourier series", *Math. Comput.*, vol. 19, no. 90, pp. 297-301, Apr. 1965.

[15] V. Galindo-Israel and R. Mittra, "A new series representation of the radiation integral with application to reflector antennas", *IEEE Trans. Antennas Prop.*, vol. AP-25, no. 5, pp. 631-641, Sept. 1977.

[16] O.M. Bucci, G. Franceschetti, and G. D'Elia, "Fast analysis of large antennas – a new computational philosophy", *IEEE Trans. Antennas Prop.*, vol. AP-28, no. 3, pp. 306-310, May 1980.

[17] K. Fourmont, "Non-equispaced fast Fourier transforms with applications to tomography,"

J. Fourier Anal. Appl., vol. 9, no. 5, pp. 431-450, Sept. 2003.

[18] J. Y. Lee and L. Greengard, "The type 3 nonuniform FFT and its applications", *J. Comput. Phys.*, vol. 206, n. 1, pp. 1-5, Jun. 2005.

[19] J.D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips, "GPU computing", *Proc. of the IEEE*, vol. 96, no. 5, pp. 879-899, May 2008.

[20] www.top500.org .

[21] T. R. Halfhill, "Parallel processing with CUDA", *Microproc. Rep.*, http://www.nvidia.com/docs/IO/55972/220401_Reprint.pdf, Jan. 28, 2008.

[22] S. S. Stone, J. P. Haldar, S. C. Tsao, W. M. Hwu, B.P. Sutton, and Z.P. Liang, "Accelerating advanced MRI reconstructions on GPUs", *J. Parallel Distr. Comp.*, vol. 68, no. 10, pp. 1307-1318, Oct. 2008.

[23] M. J. Inman and A. Z. Elsherbeni, "Programming video cards for computational electromagnetics applications", *IEEE Antennas Prop. Mag.*, vol. 47, no. 6, pp. 71-78, Dec. 2005.

[24] D. Kirk and H. M. Hwu, *CUDA Textbook*, in press.

[25] M. Bläser, "Lower bounds for the multiplicative complexity of matrix multiplication", *Comput. Complex.*, vol. 8, no. 3, pp. 203-226, Dec. 1999.

[26] K. Atkinson and D. D. K. Chien, "A fast matrix-vector multiplication method for solving the radiosity equation", *Adv. in Comput. Math.*, vol. 12, no. 2-3, Feb. 2000, pp. 151-174.

[27] http://matrixprogramming.com/MatrixMultiply/

[28] D. Sundararajan, *The Discrete Fourier Transform: Theory, Algorithms and Applications*, Singapore, Word Scientific, 2001.

[29] F. Smithies, *Integral equations*, Cambridge, Cambridge University Press, 1958.

[30] J. P. Boyd, "A fast algorithm for Chebyshev, Fourier and sinc interpolation onto an irregular grid", *J. Comput. Phys.*, vol. 103, no. 2, pp. 243-257, Dec. 1992.

[31] A. Dutt and V. Rokhlin, "Fast Fourier transforms for nonequispaced data", *SIAM J. Sci. Comput.*, vol. 14, no. 6, pp. 1368-1393, Nov. 1993.

[32] Q. H. Liu and N. Nguyen, "An accurate algorithm for nonuniform fast Fourier transforms (NUFFT's)", *IEEE Microw. Guided Wave Lett.*, vol. 8, no. 1, pp. 18-20, Jan. 1998.

[33] J. A. Fessler, B. P. Sutton, "Nonuniform fast Fourier transforms using min-max interpolation", *IEEE Trans. Signal Proc.*, vol. 51, no. 2, pp. 560-574, Feb. 2003.

[34] W. P. M. N. Keizer, "Large planar array thinning using iterative FFT techniques", *IEEE Trans. Antennas Prop.*, vol. 57, no. 10, pp. 3359-3362, Oct. 2009.

[35] H. Legay, B. Salome, E. Labiole, M. A. Milon, D. Cadoret, R. Gillard, R. Chaharmir,and J. Shaker, "Reflectarrays for satellite telecommunication antennas", *Proc. of the 2nd Europ. Conf. on Antennas Prop.*, Nov. 11-16, 2007.

[36] A. G. Roederer, "Reflectarray antennas", *Proc. of the 3rd Europ. Conf. on Antennas Prof.*, pp. 18-22, Mar. 2009.

[37] A. Capozzoli, C. Curcio, G. D'Elia, A. Liseno, D. Bresciani, and H. Legay, "Fast phase-only synthesis of faceted reflectarrays", *Proc. of the 3rd Europ. Conf. on Antennas Prop.*, pp. 1329-1333, Mar. 23-27, 2009,

[38] N. Yuan, T. S. Yeo, X. C. Nie, and L. W. Li, "A fast analysis of scattering and radiation of large microstrip antenna arrays", *IEEE Trans. Antennas Prop.*, vol. 51, no .9, pp. 2218-2226, Sept. 2003.

[39] M. Frigo and S.G. Johnson, "The design and implementation of FFTW3", *Proc. of the IEEE*, vol. 93, no. 2, pp. 216-231, Feb. 2005.

[40] The Numerical Algorithms Group, *Intel Math Kernel Library Reference Manual*, Intel Corporation, 2001.

[41] J. L. Hennessy and D. A. Patterson, *Computer Architecture: a quantitative approach*, San Francisco, USA, Morgan Kauffman Publisher, 2007.

[42] NVIDIA CUDA Reference Manual, v. 2.0, Jun. 2008.

[43] CUDA cuFFT Library, Oct. 2007.

[44] CUDA cuBLAS Library, Sept. 2007.

[45] D. Göddeke, R. Strzodka, J. Mohd-Yusof, P. McCormick, S.H.M. Buijssen, M. Grajewski, and S. Turek, "Exploring weak scalability for

FEM calculations on a GPU-enhanced cluster", *Parallel Comp.*, vol. 33, no. 10-11, pp. 685-699, Nov. 2007.

[46] A. Capozzoli, C. Curcio, G. D'Elia, and A. Liseno, "Power pattern synthesis of multifeed reconfigurable reflectarrays", *Proc. of the 29th ESA Antenna Workshop on Multiple Beams Reconfig. Antennas*, Apr. 2007.

Amedeo Capozzoli graduated (summa cum laude) in Electronic Engineering and received the PhD degree in Electronic Engineering and Computer Science, both from the University of Naples Federico II in 1994 and 2000 respectively.
In 1996**,** he was awarded the Telecom Italia Prize for the best degree thesis in Electronic Engineering discussed at the University of Naples Federico II in the Academic Year 1994-1995**.** In November 1999**,** he won the open competition for the post of Researcher at the University of Naples Federico II. In September 2002, he was awarded the Barzilai Prize for young scientists at the XIV Riunione Nazionale di Elettromagnetismo. In April 2003**,** he won the open competition for the post of Associate Professor at Politecnico di Milano. Since January 2005**,** he has been Associate Professor of Electromagnetic Fields at the University of Naples Federico II. His research interests include synthesis and diagnosis of radiating systems, inverse-scattering techniques, advanced measurement techniques, and the restoring of aberrations due to propagation through random media.

Claudio Curcio received the Laurea degree (summa cum laude) in Electronic Engineering and the PhD degree in Electronic and Telecommunication Engineering, both from the Università di Napoli Federico II, Naples, Italy, in 2002 and 2005**,** respectively. In 2006-2007**,** he held a post-doctoral position at the University of Naples Federico II. He is currently a Researcher at Università di Napoli Federico II**.** His main fields of interest are antenna measurements, phaseless near-field/far-field transformation techniques, optical beamforming techniques for array antennas, and reflectarray synthesis.
In February 2002, he was the recipient of the Optimus Award at the SIMAGINE 2002 "Worldwide GSM & Java Card Developer Contest."

Giuseppe D'Elia was born in Italy in 1950**.** He received the EE degree (summa cum laude) from the Università di Napoli, Naples, Italy. From 1983 to 1987, he was with the IRECE Institute of the National Research Council (CNR). From 1987 to 1990, he was an Associate Professor of Antennas and Propagation at the Università di Salerno, Salerno, Italy. From 1990 to 2001, he was an Associate Professor of Antennas at the Dipartimento di Ingegneria Elettronica e delle Telecomunicazioni, Università di Napoli Federico II, Naples, Italy, where, since 2001, he has been a full Professor of Electromagnetic Fields.
Prof. D'Elia has been a visiting scientist at many microwave labs, such as the Electrical Engineering Research Laboratory, University of Texas at Austin; the Popov Society, Moscow, Russia; the Institute of Electrical Engineering of the Queen Mary College at the University of London; the Microwave Laboratory of the Marconi GEG, Great Britain; and JPL, Pasadena, California. His main fields of interest include the transient behavior of antennas in dispersive media, efficient and non-redundant techniques for analysis and synthesis of reflector and array antennas, phase-less near-field antenna characterization, wavefront reconstruction from amplitude data by blind deconvolution, NF-FF transformation techniques, non-redundant representation of radiated or scattered fields, and inverse scattering and remote sensing from polarimetric data. Prof. D'Elia was awarded the 1999 Honorable Mention for the H. A. Wheeler Applications Prize Paper Award of the IEEE Antennas and Propagation Society.

Angelo Liseno was born in Italy in 1974. He received the Laurea degree (summa cum laude) and the PhD degree in 1998 and 2001, respectively, both in Electrical Engineering, from the Seconda Università di Napoli, Italy. In 2001-2002, he held a postdoctoral position at the Seconda Università di Napoli. In 2003-2004, he was a research scientist with the Institut flir Hochfrequenztechnik und Radarsysteme of the Deutsches; Zentrum für Luft- und Raumfahrt (DLR), Oberpfaffenhofen, Germany. Since 2005**,** he has been a Researcher with the Università di Napoli Federico II**,** Dipartimento di Ingegneria Elettronica e delle Telecomunicazioni, Naples, Italy. His main fields of interest are phaseless

near-field/far-field transformation techniques, remote sensing, and inverse scattering.

Pietro Vinetti was born in Naples in 1978**.** He received the Laurea degree in Telecommunication Engineering from the University of Naples Federico II**,** Naples, Italy, in 2003**.** Since 2005**,** he has been a PhD student in Electronic and Telecommunication Engineering at University of Naples Federico II, with electromagnetic as his area of interest. His research activity is mainly focused on the development of innovative near-field antenna-characterization systems, based on non-invasive dielectric probes and phaseless near-field/far- field transformation techniques.

A New Software and Hardware Parallelized Floating Random-Walk Algorithm for the Modified Helmholtz Equation Subject to Neumann and Mixed Boundary Conditions

Kausik Chatterjee[1], McCullen Sandora[1], Christopher Mitchell[1], Deian Stefan[1], Dave Nummey[1], and Jonathan Poggie[2]

[1]Department of Electrical Engineering
The Cooper Union for the Advancement of Science and Art, New York, NY 10003-7185, USA
chatte@cooper.edu, sandor@cooper.edu, mitche2@cooper.edu,
stefan@cooper.edu, nummey@cooper.edu

[2]The Air Force Research Laboratory, AFRL/RBAC,
Wright-Patterson Air Force Base, OH 45433-7512, USA
jonathan.poggie@wpafb.af.mil

Abstract– A new floating random-walk algorithm for the one-dimensional modified Helmholtz equation subject to Neumann and mixed boundary conditions problems is developed in this paper. Traditional floating random-walk algorithms for Neumann and mixed boundary condition problems have involved "reflecting boundaries" resulting in relatively large computational times. In a recent paper, we proposed the elimination of the use of reflecting boundaries through the use of novel Green's functions that mimic the boundary conditions of the problem of interest. The methodology was validated by a solution of the one-dimensional Laplace's equation. In this paper, we extend the methodology to the floating random-walk solution of the one-dimensional modified Helmholtz equation, and excellent agreement has been obtained between an analytical solution and floating random-walk results. The algorithm has been parallelized and a near linear rate of parallelization has been obtained with as many as thirty-two processors. These results have previously been published in [1]. In addition, a GPU implementation employing 4096 simultaneous threads displayed a similar near-linear parallelization gain and a one to two orders of magnitude improvement over the CPU implementation. An immediate application of this research is in the numerical solution of the electromagnetic diffusion equation in magnetically permeable and electrically conducting objects with applications in dielectrometry and magnetometry sensors that have the ability to detect sub-surface objects such as landmines. The ultimate goal of this research is the application of this methodology to the solution of aerodynamical flow problems.

Index Terms– Floating random-walk, Monte Carlo, modified Helmholtz equation, parallelizable algorithm, CUDA, GPU.

I. INTRODUCTION

The floating random-walk (FRW) method [2] is a statistical technique for the numerical solution of deterministic boundary value problems. It is a generalization of the Monte Carlo integration method [3], which is a statistical approach to estimating integrals, which, unlike many other techniques, is well-suited to evaluating multi-dimensional integrals. We will discuss one such method, "Sample Mean Monte Carlo" [3], and then demonstrate how the technique is modified to form the basis for the FRW method.

Consider a function $f(x)$ defined over the interval $a \le x \le b$. Our problem is to estimate the integral

$$\mathbf{I} = \int_a^b dx\, f(x). \qquad (1)$$

In the event that the integral is improper, absolute convergence is assumed. We select an arbitrary probability density function $p(x)$, with a corresponding random variable ξ. We define another random variable η as

$$\eta = \frac{f(\xi)}{p(\xi)}. \tag{2}$$

The expectation value of the random variable η, written as $E(\eta)$, is equal to the integral $\mathbf{I}$, which can be expressed as

$$\mathbf{I} = E(\eta) = \int_a^b dx \left[\frac{f(x)}{p(x)}\right] p(x). \tag{3}$$

This integral can be evaluated by sampling the integrand with the help of a random-number generator, and averaging over a statistically significant number of samples. This approach is particularly suited to evaluating higher dimensional integrals, because the computational work of sampling the integrand does not increase substantially with the dimensionality of the integral. We will now describe how this Monte Carlo integration method can be generalized into the FRW method for the solution of boundary value problems.

We consider a differential equation, with a differential operator L,

$$L[U(\mathbf{r})] = f(\mathbf{r}), \tag{4}$$

where the solution $U(\mathbf{r})$ is a function of the three-dimensional position vector $\mathbf{r}$. The function $f(\mathbf{r})$ is a source term. The Green's functions for (4) are the solutions of the differential equation

$$L[G(\mathbf{r} \mid \mathbf{r}_0)] = \delta(\mathbf{r} - \mathbf{r}_0), \tag{5}$$

subject to specified boundary conditions. We assume that the operator L is of the Sturm-Liouville [4] form:

$$L = \nabla\cdot[p(\mathbf{r})\nabla] + q(\mathbf{r}), \tag{6}$$

where $p(\mathbf{r})$ and $q(\mathbf{r})$ are known functions of $\mathbf{r}$. Using Green's integral representation [4] $U(\mathbf{r})$ can be written as

$$\begin{aligned} U(\mathbf{r}_0) &= \iiint_V dv G(\mathbf{r} \mid \mathbf{r}_0) f(\mathbf{r}) \\ &- \oiint_S [d\mathbf{s}\cdot\nabla_{\mathbf{r}} U(\mathbf{r})] p(\mathbf{r}) G(\mathbf{r} \mid \mathbf{r}_0) \\ &+ \oiint_S [d\mathbf{s}\cdot\nabla_{\mathbf{r}} G(\mathbf{r} \mid \mathbf{r}_0)] p(\mathbf{r}) U(\mathbf{r}). \end{aligned} \tag{7}$$

The first term on the right hand side of (7) is a volume integral involving the source term in the entire volume V of interest. The second and third terms are vector surface integrals over the surface S enclosing V, where $d\mathbf{s}$ is a vector whose magnitude is equal to that of an infinitesimally small area unit on the surface S and directed normally outward from the center of the area unit. The term $G(\mathbf{r}|\mathbf{r_0})$ is often referred to as the volumetric Green's function and the term $\nabla_{\mathbf{r}} G(\mathbf{r} \mid \mathbf{r}_0)$ is called the surface Green's function. The second term corresponds to the Neumann [4] boundary condition, whereas the third term corresponds to the Dirichlet boundary condition [4]. In traditional FRW algorithms, homogeneous Dirichlet boundary conditions are imposed on the Green's function given by (5). As a result, the second integral in the right hand side of (7) goes to zero. To evaluate the solution to (4) at a particular point in the domain of interest, we consider [2] maximal spheres, cubes, or any geometrical object for which the solution to (5) is known. We then make random hops to the surface of that geometrical object based on any predefined probability density. The weights for such random hops are determined by sampling the remaining two integrands in (7). For example, in the case of a Dirichlet problem with no source term [i.e., $f(\mathbf{r}) = 0$], the contribution of the volume integral also goes to zero and the problem reduces to a Monte Carlo integration of an infinite-dimensional integral, as given by:

$$\begin{aligned} U(\mathbf{r_0}) &= \oint_{S_1} ds_1 K(\mathbf{r_0} \mid \mathbf{r_1}) \ldots \oint_{S_n} ds_n K(\mathbf{r_{n-1}} \mid \mathbf{r_n}) U(\mathbf{r_n}), \\ K(\mathbf{r_{n-1}} \mid \mathbf{r_n}) &= \left|\nabla_{\mathbf{r_n}} G(\mathbf{r_{n-1}} \mid \mathbf{r_n})\right| \cos(\gamma_{n-1,n}), \end{aligned} \tag{8}$$

where $\gamma_{n-1,n}$ is the angle between $\nabla_{\mathbf{r_n}} G(\mathbf{r_{n-1}} \mid \mathbf{r_n})$ and $d\mathbf{s_n}$. The successive surface integrals in (8) relate to successive random hops across the problem domain and the weight factors of the form $K(\mathbf{r_{n-1}} \mid \mathbf{r_n})$ are derived from

the third integral term on the right hand side of (7) that corresponds to the Dirichlet boundary condition. A particular random walk is terminated at the boundary, where the solution is known, and the samples of successive weight factors multiplied by the solution at the boundary yield a particular sample of the solution. A numerical solution of (4) is obtained by averaging over a statistically significant number of such samples.

The termination of the random walk becomes a problem for Neumann and mixed boundary condition problems where the solution is not known at all points of the domain boundary. In traditional random walk literature [5], these boundary conditions are formulated as partially "reflecting" as the random-walker has a chance of either being absorbed in the problem boundary or being thrown back into the problem domain. In a recent paper [6], we formulated a FRW algorithm for this particular problem where the reflection at problem boundaries was eliminated through the development of a Green's function whose boundary conditions mimicked the boundary conditions of the problem of interest. In this paper, we extend the methodology to the solution of the one-dimensional modified Helmholtz equation, subject to mixed boundary conditions.

II. THE NEW FORMULATION

Consider the equation

$$\frac{d^2U}{dx^2} = 0, \tag{9}$$

where U is the dependent variable of interest defined in the problem domain $0 \le x \le L$. The boundary conditions imposed on this problem are $U(0) = \alpha$ and $U(L) = \beta$. A traditional FRW algorithm for this problem will be based on a Green's function given by

$$\frac{d^2G}{dx^2} = \delta(x - x_0), \tag{10}$$

defined on a problem domain $-h \le x \le h$, with homogeneous Dirichlet boundary conditions $G(-h \mid x_0) = 0$ and $G(+h \mid x_0) = 0$. The solution to (10) in a zero-centered notation (i.e., $x_0 = 0$) is given by

$$G(x \mid 0) = \begin{Bmatrix} \frac{1}{2}(x-h), & x \ge 0 \\ -\frac{1}{2}(x+h), & x \le 0 \end{Bmatrix}. \tag{11}$$

Based on the 1D version of the Green's integral representation (7), the solution to (9) at the center of the one-dimensional problem domain can be written as

$$U(0) = \left[U\frac{dG}{dx} - G\frac{dU}{dx} \right]_{x=h} - \left[U\frac{dG}{dx} - G\frac{dU}{dx} \right]_{x=-h}, \tag{12}$$

where no specific boundary conditions have been imposed on the Green's function. Using the Green's function given by (11), (12) can be reduced to

$$U(0) = \frac{1}{2}U(h) + \frac{1}{2}U(-h). \tag{13}$$

Thus, the solution to (9) at the center of the problem domain $-h \le x \le h$ can be expressed in terms of the solution at the two end-points. In a traditional FRW algorithm, (13) is used to generate the random walks. The random walker either hops to the left or to the right with equal probability (without any restriction on the hop size) until it is absorbed at one of the boundaries. An estimate of the solution at any given point $x = x^*$ within the problem domain $0 \le x \le L$ is given by

$$U(x^*) = \frac{N_\alpha \alpha + N_\beta \beta}{N_\alpha + N_\beta}, \tag{14}$$

where the number of times the random walker hits the $x = 0$ and the $x = L$ boundary are N_α and N_β respectively. Now let us consider the solution of (9) defined on the problem domain $0 \le x \le L$, but with the boundary conditions $U(0) = \alpha$ and $\left\{ dU/dx \right\}_{x=L} = \beta$. It is obvious that a FRW scheme based on (13) will not find a reward at the $x = L$ boundary. The termination at this boundary is based on a finite-difference based representation of the Neumann boundary condition [5] and the random-walker is either absorbed or reflected back into the problem domain. If the random walker is reflected back in the problem domain, once again random walks are generated based on (13). This partial

reflection at the boundary increases the computational time and as a result, Neumann and mixed boundary condition problems are considered difficult to be handled with the FRW method.

In a recent paper [6], we proposed a philosophically different approach for Neumann and mixed boundary condition problems and applied it to the problem given in (9). In our approach, the boundary conditions imposed on the Green's function mimic those of the problem of interest and as a result, the reflecting boundaries are converted to absorbing boundaries. In this paper, we use this approach to develop a FRW algorithm for the one-dimensional modified Helmholtz equation given by

$$\frac{d^2U}{dx^2} - k^2U = 0, \tag{15}$$

where U is the dependent variable of interest defined in the problem domain $0 \le x \le L$, and k is a real number independent of x. The boundary conditions imposed are $U(0) = \chi$ and $\left\{dU/dx\right\}_{x=L} = \delta$. Our approach is motivated by the one-dimensional version of Green's integral representation given by (12) and is based on a Green's function $G(x \mid x_0)$ of (15) given by

$$\frac{d^2G}{dx^2} - k^2G(x \mid x_0) = \delta(x - x_0), \tag{16}$$

defined in the problem domain $-h \le x \le h$ with the boundary conditions $G(-h \mid x_0) = 0$ and $\left\{dG/dx\right\}_{x=h} = 0$. This Green's function is explicitly given by

$$\begin{aligned} G(x \mid x_0) &= -\frac{\cosh[k(x_0 - h)]}{k\cosh[2kh]} \\ &\quad \times \sinh[k(x + h)], \quad x \le x_0, \\ G(x \mid x_0) &= -\frac{\sinh[k(x_0 + h)]}{k\cosh[2kh]} \\ &\quad \times \cosh[k(x - h)], \quad x \ge x_0. \end{aligned} \tag{17}$$

We use the boundary conditions that have been imposed on the Green's function given by (17) and the one-dimensional Green's integral representation given by (12) to obtain a representation of the solution $U(x_0)$ at a point $-h \le x_0 \le h$ given by

$$\begin{aligned} U(x_0) = &-\left[G(x \mid x_0)\frac{dU}{dx}\right]_{x=h} \\ &-\left[U(x)\frac{dG}{dx}\right]_{x=-h}. \end{aligned} \tag{18}$$

We now obtain a derivative of (18) with respect to x_0 and obtain a representation of the derivative of the variable of interest U given by

$$\frac{dU}{dx_0} = -\left[\frac{dG}{dx_0}\frac{dU}{dx}\right]_{x=h} - \left[U(x)\frac{d^2G}{dxdx_0}\right]_{x=-h}. \tag{19}$$

Equations (18) and (19) are used to generate a FRW scheme that is different from the scheme based on (13). In order to estimate the solution at a given point, the random walker hops to either the left or the right with probability $1/2$ as given by (18). If the random walker moves to the left, there is a multiplicative weight factor given by $W_{LL} = \left[-2G_x(x \mid x_0)\right]_{x=-h}$ and (18) is again used to generate the random walks in the next hop. On the other hand, if the random walker moves to the right, there is a multiplicative weight factor given by $W_{LR} = \left[-2G(x \mid x_0)\right]_{x=h}$ and (19) is used to generate the random walks in the next hop. As (19) is used to generate the random walks, the random walker moves to the left or the right with probability $1/2$. If the random walker moves to the left, there is a multiplicative weight factor given by $W_{RL} = \left[-2G_{xx_0}(x \mid x_0)\right]_{x=-h}$ and (18) is used to generate the random walks in the next hop. On the other hand, if the random walker moves to the right, there is a multiplicative weight factor given by $W_{RR} = \left[-2G_{x_0}(x \mid x_0)\right]_{x=h}$ and (19) is used to generate the random walks in the next hop. A particular random walk terminates either in the left boundary with a reward χ or at the right boundary with a reward δ, and an estimate of the solution is obtained by averaging over a statistically significant number of random walks. Thus, through the use of the Green's function in (17), the partially reflecting boundary at $x = L$ is converted to an absorbing boundary and there is no reflection. The results for the problem given by (15) will now be presented.

III. SOFTWARE RESULTS

A mixed boundary condition problem for the modified Helmholtz equation given by (15) is chosen with $L = 3$, $k = 0.5$, $\chi = 1$ and $\delta = 3$. Fig. 1 plots the exact analytical solution and the FRW results and these are seen to be in excellent agreement. In the FRW simulations, 2×10^6 random walks have been carried out for each solution point and the average error between the analytical results and the FRW solution was seen to be about 0.1 percent. Fig. 2 shows the relative speed of parallelization with respect to single processor computation. It is seen that the relative speed of parallelization gets closer to unity as the number of random-walks per solution point is increased. The increased deviation from linearity with decrease in the number of random-walks can be interpreted as the percentage increase in inter-processor communication time with respect to actual computation time that occurs with decrease in the number of random-walks.

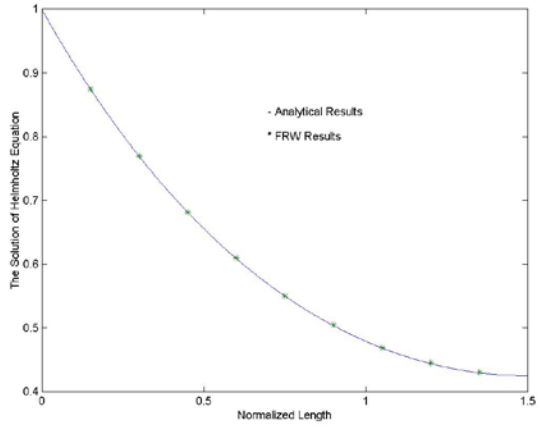


Fig. 1. Analytical and FRW results for the solution of the modified Helmholtz equation plotted against normalized length $(x' = kx)$.

IV. GPU IMPLEMENTATION

While multiprocessor environments provide exceptional throughput advantages in scientific computing, GPUs have recently emerged as an alternative highly-parallel technology for general purpose computing. GPUs contain a significantly higher core density than any commercially-available CPU, with the tradeoff

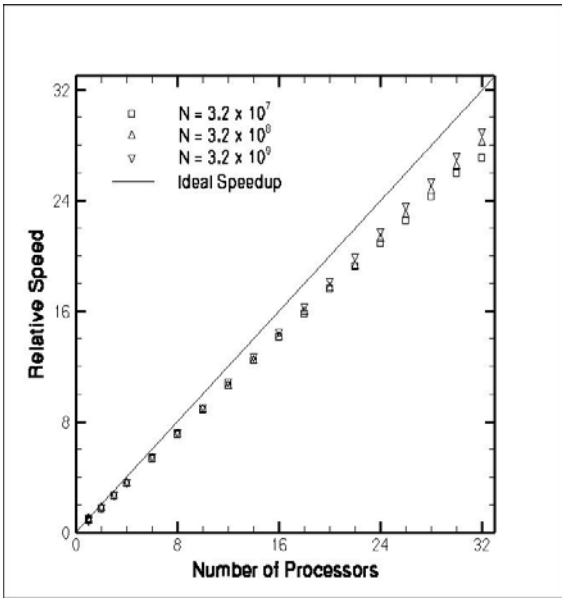


Fig. 2. Relative speed of parallelization with respect to single processor computation.

of a restricted and relatively more specific instruction set and the need for complex memory management by the designer. With the recent introduction [7] of the Compute Unified Device Architecture (CUDA) to developers on commodity NVDIA graphics cards, throughput gains of scientific applications can be increased by orders of magnitude. Using NVIDIA's CUDA application programming interface (API), we have implemented the previously-developed algorithm to explore and evaluate the merit and value of the approach. Efficient parallelization is a significant logical problem that may be solved in different ways depending on the nature of the architectural environment and the algorithm under parallelization. CUDA's shared memory model and layout of threads into blocks, for example, must be taken into account to maximally utilize the GPU's resources.

The Helmholtz problem under study lends itself well to parallelization. As the problem requires executing highly-repetitive code for millions of iterations per point at which the equation is solved, it can directly be parallelized by assigning each point to a thread within a thread block. Our final program was divided into three sections:

1) A serial (CPU) part that set up initial conditions.
2) A parallel version that ran 200,000 iterations of the algorithm for each point.
3) A serial finalization section that collected and displayed the results from the GPU.

The implementation has been simplified through the use of global GPU memory that is retained for the execution lifetime of the application. Additional improvements of 100x to 150x speedup can be achieved by using the shared memory, visible between the threads of a block but not shared among other blocks. This requires some overhead to load from global to shared memory at the beginning of the block and restore results to global memory at the end, but 'reads' and 'writes' from shared memory are comparable to register access time which is drastically faster than the 400-600 cycles required for global memory 'reads/writes.' The initial copy from global to shared memory is hidden by using thousands of threads that are more computationally intensive.

A 200,000-iteration test has been run on both a CPU and a GPU, with 16 points for the CPU and 4096 points for the GPU, the CPU completed its task in 2.57 seconds, while the GPU took 16.56 seconds. This indicates a speedup of 39.7 (0.16s/point vs. 0.004s/pt) in favor of the GPU. With additional improvements in memory handling, significant additional speedup may be possible. The GPU implementation is based on 4096 parallel threads organized into 32 blocks of 128 threads each for purposes of coalesced memory access and thread scheduling. To circumvent an 8-second execution time limit imposed by the Linux drivers for the graphics card, tests involving more than 100,000 random walks were broken down into multiple sub-steps each running at most 100,000 walks. Note that the imprecision visible in Figure 3 below is attributable both to the use of single-precision floating point calculations instead of double-precision in our tests (the GPU in question, GeForce GTS 8800, does not contain double-precision floating point units) and to the relatively low number of walks (i.e., 200000) for each point. Although only nine representative points are shown on the graph, the Helmholtz solution was calculated for a full 4096-point span for $x' = [0,1.5]$, where $x' = kx$ is the normalized length scale shown in Fig. 1.

The GPU implementation utilized the Mersenne Twister pseudorandom number generator (PRNG) written by NVIDIA [8]. It was used to generate a very large array of pseudorandom numbers before the random walk algorithm began; the algorithm then picked successive numbers out of the pre-calculated array. The random number array of about 24 million values was recalculated on the GPU in roughly 1 second using 4096 parallel threads; when the test was broken into multiple sub-steps, a new set of PRNGs was calculated with a new seed for each sub-step.

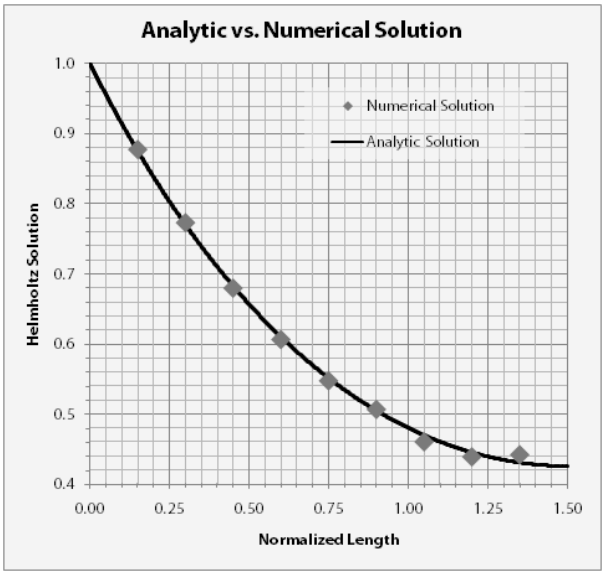


Fig. 3. GPU solution to the Helmholtz equation utilizing random walks. The solution at nine representative points from a total of 4096 is shown.

V. CONCLUSION

Summarizing, a previously-developed FRW methodology [6] for Neumann and mixed boundary problems has been extended to the solution of the 1D modified Helmholtz equation. In this methodology, reflecting boundaries are converted into absorbing boundaries through the development of Green's functions that mimic the boundary conditions of the problem of interest. The algorithm has been validated by an

analytical solution and excellent agreement has been obtained between analytical and numerical results. The algorithm has been parallelized in software and a near linear rate of parallelization has been obtained for as many as thirty-two processors. On a commodity graphics card, a speedup of over two orders of magnitude over the software implementation has been obtained. Further work involving GPU implementation will therefore begin with further optimizing our current implementation and considering a method of higher parallelization by scheduling threads on the per-walk level rather than the per-point level, which while introducing more overhead will allow for similar (and therefore lower) execution time between threads. Our future work in this area will involve the extension of this methodology to other important equations and to problems in two and three dimensions. The ultimate goal of this research is the utilization of this methodology for the solution of aerodynamical problems with Neumann and mixed boundary conditions.

ACKNOWLEDGMENT

This research has been supported by the Air Force Office of Scientific Research through a grant (FA9550-06-1-0439) monitored by Dr. F. Fahroo.

REFERENCES

[1] K. Chatterjee, M. Sandora, C. W. Yu, S. Srinivasan, J. Poggie, "A New Parallelized Floating Random-Walk Algorithm for the Modified Helmholtz Equation Subject to Neumann and Mixed Boundary Conditions: Validation with a 1D Benchmark Problem," *The 25th International Review of Progress in Applied Computational Electromagnetics*, March 2009.

[2] Y. L. Le Coz and R. B. Iverson, "A Stochastic Algorithm for High Speed Capacitance Extraction in Integrated Circuits," *Solid-State Electronics*, Vol. 35, pp. 1005-1012, 1992.

[3] M. Sobol, *A Primer for the Monte Carlo Method*, CRC Press: Boca Raton, 1994.

[4] R. Haberman, *Elementary Applied Partial Differential Equations*, 3rd ed., Prentice-Hall: New Jersey, 1998.

[5] A. Haji-Sheikh, *Application of Monte Carlo Methods to Thermal Conduction Problems*, Ph.D. dissertation, University of Minnesota, 1965, pp. 106-108.

[6] K. Chatterjee, C. Yu, S. Srinivasan and J. Poggie, "A New Floating Random Walk Methodology for Neumann and Mixed Boundary Condition Problems Without Reflections at Boundaries: Validation with Laplace's Equation in One Dimension," *Far East Journal of Applied Mathematics*, Vol. 26, No. 3, pp. 705-713, 2007.

[7] NVIDIA, 2007, Compute Unified Device Architecture: http://developer.download.nvidia.com/compute/cuda/1_0/NVIDIA_CUDA_Programming_Guide_1.0.pdf.

[8] V. Podlozhnyuk, Parallel Mersenne Twister: http://developer.download.nvidia.com/compute/cuda/sdk/website/projects/MersenneTwister/doc/MersenneTwister.pdf, 2007.

Kausik Chatterjee was born in India in 1969. In 1992, he received a Bachelor of Engineering degree in Electrical Engineering from Jadavpur University, Calcutta, India. Subsequently, in 1995, he received a Master of Technology degree in Nuclear Engineering from Indian Institute of Technology, Kanpur, India, and in 2002, he received his PhD degree in Electrical Engineering from Rensselaer Polytechnic Institute, Troy, New York. In 2002, he joined the faculty at California State University, Fresno as an Assistant Professor of Electrical and Computer Engineering, a position he held till 2005. He had been a visiting scientist at MIT Laboratory for Electromagnetic and Electronic Systems and had held faculty fellowships at Air Force Research Laboratory (Wright-Patterson Air Force Base) and NASA, Langley Research Center. He is currently an Assistant Professor in the Department of Electrical and Computer Engineering at The

Cooper Union for the Advancement of Science and Art in New York City. His current research interests include the development of random-walk algorithms for important equations in nature and a theory for high temperature superconductors.

McCullen Sandora was born on March 21, 1987. In May 2009, he received a Bachelor of Science degree in Interdisciplinary Engineering from The Cooper Union for the Advancement of Science and Art in New York City. He is currently a doctoral student in the Department of Physics at University of California, Davis and his current research focus is in the area of High Energy Theory.

Christopher Mitchell was born on March 11, 1987. In May 2009, he received his BE degree in Electrical Engineering from The Cooper Union for the Advancement of Science and Art in New York City, and expects to receive his Masters of Engineering in Electrical Engineering from the same institution in May, 2010. His current research interests include augmented reality and wearable computing hardware and applications, neural network applications in the areas of image recognition and self-training syntactical parsing and the implementation of high-functioning software applications on low-resource portable devices.

Deian Stefan received his B.E. degree in Electrical Engineering from The Cooper Union for the Advancement of Science and Art in 2009 and will receive his M.E. degree in Electrical Engineering in 2010. His current research interests include the analysis and implementation of cryptographic and coding algorithms.

David Nummey was born on April 18, 1987. In May 2009, he received his Bachelor of Engineering degree in Electrical Engineering from The Cooper Union for the Advancement of Science and Art in New York City. He is now working toward his Masters of Engineering degree in Electrical Engineering at the same institution, and expects to finish in May 2010. His current research interests include statistical signal processing methods, pattern recognition and machine learning techniques, and applications towards medical imaging, electrophysiology, neuroscience and assistive devices.

Jonathan Poggie was born in the United States in 1966. He studied mechanical engineering at the University of Rhode Island, receiving a B.S. degree in 1988. He went on to obtain a Ph.D. degree in mechanical and aerospace engineering from Princeton University in 1995, specializing in fluid mechanics. Since then, he has worked at the U.S. Air Force Research Laboratory, focusing on physical problems associated high speed flight. His work has encompassed laminar/turbulent transition in hypersonic boundary layers, unsteadiness of shock waves in separated flow, and the control of ionized gas flow by electromagnetic means.

An Efficient Parallel Multilevel Fast Multipole Algorithm for Large-scale Scattering Problems

Hu Fangjing, Nie Zaiping, Hu Jun

School of Electronic Engineering,
University of Electronic Science and Technology of China
Chengdu 610054, PRC
hfj1010@gmail.com

Abstract—In this paper, we present an efficient parallel multilevel fast multipole algorithm (MLFMA) for three dimensional scattering problems of large-scale objects. Several parallel implantation tricks are discussed and analyzed. Firstly, we propose a method that reduces truncation number without loss of accuracy. Furthermore, a matrix-sliced technique, allowing data in the memory transforming into the hard disk, is applied here, in order to solve the problem of extremely large targets. Finally, a transition level scheme is adopted to improve the parallel efficiency. We demonstrate the capability of our code by considering a sphere of 220λ discretized with 48,879,411 unknowns and a square patch of 200λ discretized with 10,150,143 unknowns. The bi-static RCS is calculated within 41.5 GB memory for the first object and 14.7 GB for the second one.

Index terms—parallel algorithm, RCS calculation, multilevel fast multipole algorithm, electromagnetic scattering.

I. INTRODUCTION

Integral equation methods are widely used for solving electromagnetic scattering problems, and the multilevel fast multipole algorithm (MLFMA) has established itself as one of the most powerful among the different acceleration methods [1]. However, for many real-life problems, the discretization of these large-scale targets lead to millions of unknowns. The maximum size that can be solved is limited even with modern computers. Thus, it is necessary to develop an efficient parallel algorithm in order to solve these very large-scale problems.

Of the various parallelization schemes for MLFMA, the most popular is the distributed memory architectures by constructing clusters of computers with local memories connected via fast networks [1]-[5]. However, the parallel implementation of MLFMA is not trivial, owing to the complicated structure of this algorithm. Without careful parallel schemes, the algorithm may fail to produce accurate results. Thus, a series of implementation tricks have been developed for the efficient parallelization of MLFMA in [2]-[5]. But even with these implementations, the algorithm has to face memory-hungry problem for many extremely large problems.

In this paper, we present an efficient parallel MLFMA algorithm that integrating a series of implementation tricks proposed in [2]-[4]. In particular, a novel trick for reducing the truncation number is presented and the technique, that slices the matrix data and save it to the hard disk, is applied in our code, in order to optimize the memory usage and solve the memory-hungry problem. To demonstrate the capability of our parallel MLFMA code, the bi-static RCS of a sphere with a diameter of 220λ, containing more than 50 million unknowns, and a patch of 200λ, containing about 10 million unknowns, are successfully solved.

II. PARALLEL IMPLEMENTATION OF MLFMA

A series of implementation tricks for parallel MLFMA are developed in [2]-[4], most of which focus on memory optimization. We can say that reducing the RAM requirement can never be over-emphasized. In this section, several tricks that integrating in our code will be introduced and analyzed.

A. Integral Equation Formulation

In this section, we consider the scattering of electromagnetic waves from perfectly conducting objects.

For a perfectly conducting object, the well-known electric-field integral equation (EFIE) can be written as [2]

$$\frac{ik\eta}{4\pi}\hat{t}\cdot\int_S \bar{G}(\vec{r},\vec{r}\,')\cdot\vec{J}(\vec{r}\,')dS' = -\hat{t}\cdot\vec{E}^i(\vec{r}) \qquad r\in S \tag{1}$$

with

$$\bar{G}(\vec{r},\vec{r}\,') = \left[\bar{I}-\frac{1}{k^2}\nabla\nabla'\right]g(\vec{r},\vec{r}\,') = \left[\bar{I}-\frac{1}{k^2}\nabla\nabla'\right]\frac{e^{ik|\vec{r}-\vec{r}\,'|}}{|\vec{r}-\vec{r}\,'|}$$

In Equation (1), η is the impedance of free space, S is the surface boundary of the scatterer, and $\hat{t}$ is the unit tangent vector at any given point on S. Furthermore, $\vec{J}$ is the unknown surface electric current, $\vec{E}^i$ is the incident electric-field vector, and $\bar{I}$ is the unit dyad.

If the surface of the object is closed, it can also be described using the magnetic-field integral equation (MFIE)

$$-\hat{t}\cdot\frac{\vec{J}(\vec{r})}{2}+\frac{1}{4\pi}\hat{t}\cdot\vec{n}\times\nabla\times\int_S g(\vec{r},\vec{r}\,')\cdot\vec{J}(\vec{r}\,')dS' = -\hat{t}\cdot\vec{n}\times\vec{H}^i(\vec{r}) \tag{2}$$

where $\vec{n}$ is the unit normal vector, and $\vec{H}^i$ is the incident magnetic-field vector.

However, both EFIE and MFIE suffer from nonunique solutions at resonant frequencies. To alleviate this problem, for a surface-closed target, we used the combined-field integral equation (CFIE) which is defined by the relation

$$\alpha\ \text{EFIE}+\eta(1-\alpha)\,\text{MFIE} \tag{3}$$

where $\alpha \in [0,1]$ is called the combination coefficient.

In order to solve these equations numerically, we should model the surface with flat triangular patches and expand the current in term of RWG basis functions [7]. Applying Galerkin's method, the integral equation is then reduced to a system of linear equations. The matrix element of EFIE and MFIE is given by

$$Z_{mn}^E = \frac{ik\eta}{4\pi}\int_S dSf_m(\vec{r})\cdot\int_S \bar{G}(\vec{r},\vec{r}\,')\cdot f_n(\vec{r}\,')dS' \tag{4}$$

$$Z_{mn}^M = -\int_S dSf_m(\vec{r})\cdot\frac{f_n(\vec{r})}{2} + \frac{1}{4\pi}\int_S dSf_m(\vec{r}\,')\cdot\hat{n}\times\nabla\times\int_S g(\vec{r},\vec{r}\,')\cdot f_n(\vec{r}\,')dS' \tag{5}$$

where $f_n(\vec{r})$ is the nth RWG basis function.

The matrix element corresponding to CFIE can then be derived as $Z_{mn} = \alpha Z_{mn}^E + \eta(1-\alpha)Z_{mn}^M$. Then the integral equations reduce to the matrix equation

$$[Z]_{N\times N}\bullet[J]_{N\times 1} = [V]_{N\times 1} \tag{6}$$

where N is the number of unknowns.

Equation (6) can be solved using an iterative method such as the Generalized Minimal Residual Algorithm (GMRES). The detailed discussions for the parallelized version of GMERS can be found in the literature [2].

B. A Novel Method for Reducing the Truncation Number

In MLFMA, the memory requirement and the CPU time depend heavily on the truncation number, L, which is normally determined by the size of box, D [6]. We should determine the minimum value of D in order to reduce the truncation number, and thus save the memory requirement and CPU time. The relation between the truncation number L and D can be expressed as

$$L = kD + \ln(\pi + kD) \tag{7}$$

Previously, D is determined by the real size of box in each level. The truncation number L then can be calculated using equation (7). However, the value of D obtained in this way is not a minimum, for there are spaces for many boxes [4]. In [4], one method is proposed to determine the minimal D at each level by finding the maximum distance of the edge-distance located in each box.

In this section, we present another method, by finding the equivalent maximum distance in each box, to determine the value of D. The equivalent maximum distance on level L is defined as

$$d(l)=\sqrt{\left(x_{\max}-x_{\min}\right)^2+\left(y_{\max}-y_{\min}\right)^2+\left(z_{\max}-z_{\min}\right)^2}$$

where x_{min}, y_{min}, z_{min} and x_{max}, y_{max}, z_{max} are the minimum and maximum coordinates, respectively, among the triangular patch pair center points for each box on one level. Figure 1 shows the relation of the equivalent maximum distance, d, and D concerning with the real size of box for a two-dimensional problem. The solid and dashed line represents d and D respectively, and the white nodes are the patch pair center points in the box.

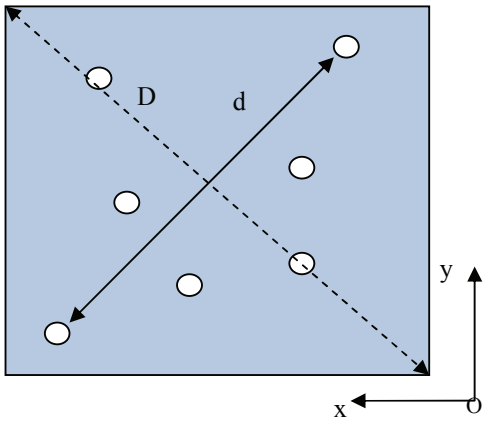


Fig. 1. The relation of d and D for a two-dimensional problem.

We could calculate every d in every box at each level, find the maximal one, and designate it as the equivalent value of D. Thus, the truncation number at each level can be determined by

$$L=kd(l)+\ln\left(\pi+kd(l)\right) \qquad (8)$$

For the equivalent maximum distance d is less than the value of D, thus the truncation number, L, can be reduced without loss of the accuracy. For a target discretized with tens of millions unknowns, the time consuming on finding the equivalent maximum distance can be neglected. Also, this method can be efficiently parallelized. Figure 2 shows the Bi-static RCS of a sphere with a diameter of 4λ，the result shows our method agrees well with the MIE series.

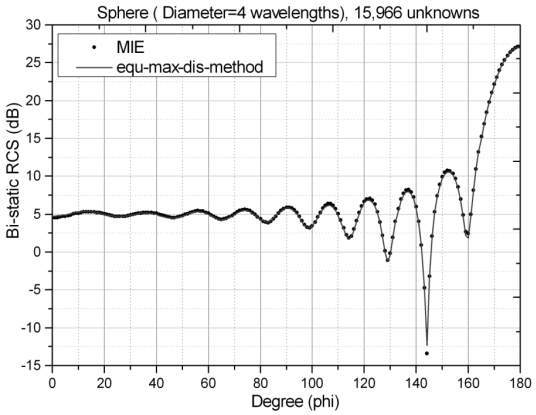


Fig. 2. The Bi-static RCS of our method and the MIE series.

C. The Matrix-Sliced-to-Disk Technique

For many extremely large targets even modern servers and computers will encounter memory-hungry problems. The memory in MLFMA is mainly consumed in setting up the matrix equation. The idea that transforms the matrix data into the hard disk is straight-forward. There are three main reasons for adopting this technique to our parallel MLFMA code:

1) This approach allows us to solve extremely big problems without having to worry about the memory consumption. Memory is almost used for other parts of MLFMA such as the oct-tree rather than the matrix equation.

2) From an economic and convenient point of view, this approach helps our code to be more scalable and meet the demand of some low-performance computers. With this approach, we can solve a problem with about one million unknowns on a single computer of only 2 GB memory.

3) With the swift improvements in the hard drive storage technology, the difference of the I/O speed of memory and the hard disk will be reduced, making this approach more and more attractive, as shown in Fig. 3.

Actually, this technique will cost slightly more time than without it, for the I/O operation of hard disk is relatively slower than that of memory. Thus it is necessary to compare the CPU time and the elapsed time (which express the total time from the start of a program to the

end of it), in order to evaluate the efficiency of this technique. Sphere with different electrically sizes are considered here. The diameters of sphere range from 20 wavelengths to 220 wavelengths. The CPU and elapsed time for different sizes are depicted in Figure. 3. All the calculations are carried out on one computer with 4 CPUs and a high performance SAS hard disk. The result shows that up to nearly 50 million unknowns, the performance with this technique is only 4.5% slower than that without it. For the situation that the number of unknowns less than 10 million, the differences of with and without this technique can be well neglected.

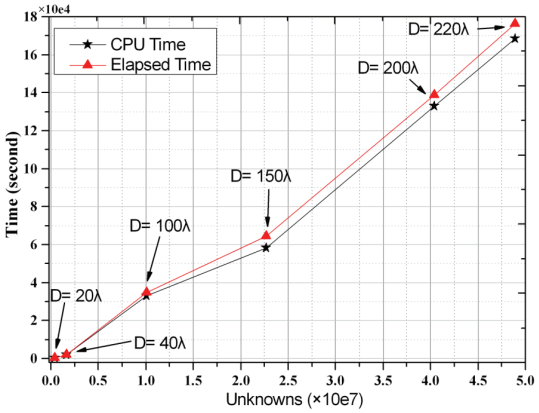


Fig. 3. The comparisons of CPU time and elapsed time for different electrical sizes of sphere.

The solid-state store technique is the trend. With the technical improvement of high performance hard disk, the influence of this relatively slower I/O operation will be less significant. Thus, we can say that this technique will be much more practical in the future.

D. The Transition Level Scheme

An important part of parallel MLFMA is the parallel efficiency. Previously, the boxes were distributed equally among the processors. It is natural that this parallel approach can achieve good load-balancing in fine levels. However, it is difficult to achieve good load-balancing in the coarse levels with this approach, since the number of boxes is small in those levels. This usually degrades the parallel efficiency and performance of parallel MLFMA code.

A transition level scheme is proposed in [3] in order to improve the parallel efficiency. In the levels that are finer than the transition level, the boxes are distributed equally among the processors; in the levels that are coarser than the transition level, the far-field pattern and translation matrix are distributed equally among the processors. However, this scheme causes additional communication between processors. In order to reduce this problem of communication and to restrict each processor to communicate with only two nearby processors at most, it is proved in [4] that the transition level should be the level where the truncation number, L, is not less than twice the number of processors, p. To obtain good parallel efficiency, we usually choose L=2p.

E. The Efficiency of Parallel MLFMA

The efficiency of parallel algorithm is defined as

$$\eta = \frac{T_1}{pT_p} \times 100\%$$

where p is the number of processors, T_p is the CPU time consumed for p processors.

To demonstrate the efficiency of our Parallel MLFMA, the bi-static RCS of a sphere of diameter 40λ is calculated. The total parallel efficiency and matrix-vector multiplication parallel efficiency is shown in Fig. 4. We can see that the efficiency is above 80% even for 16 processors.

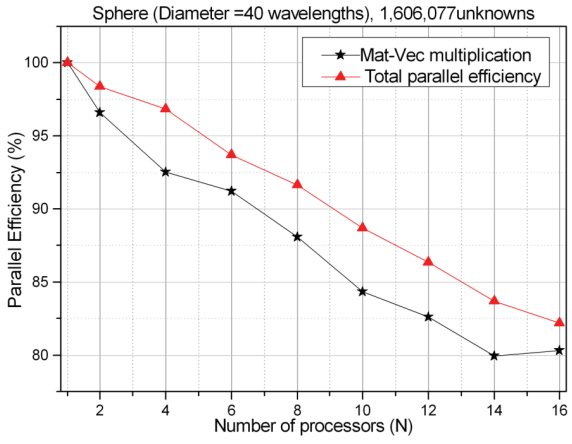


Fig. 4. The parallel efficiency for a sphere of diameter 40λ from 1 to 16 processors.

III. CAPABILITY OF THE PARALLEL MLFMA

A parallel MLFMA code has been developed by implementing several tricks presented in the above sections. To demonstrate the capability of our code, we first calculate the bi-static RCS of a sphere of diameter 220λ discretized with 48,879,411 unknowns. The incident angle is (90°, 0°), and the scanning plane is *xoy* plane with 1801 sampling points from (0°, 180°). The parallelized GMRES is adopted to solve the matrix equation, and the residual error is 0.005. The simulation is carried out on one single computer with 8 Xeon 3.0 GHz CPUs and 64 GB memory. The detail resources used in this calculation is shown in Table 1.

Table 1. The computational resources for a sphere of 220λ by the parallel MLFMA.

General Information		CPU Time (min)	
Geometry size (wavelength)	220	Geometry information	5.6
Number of processors	8	Set up of near-field matrix	199.1
Number of iterations	19	Set up of far-field matrix	67.3
Total memory (GB)	41.5	Iteration and solution	1121.5
Total time (hr)	26.9	RCS calculation	0.63

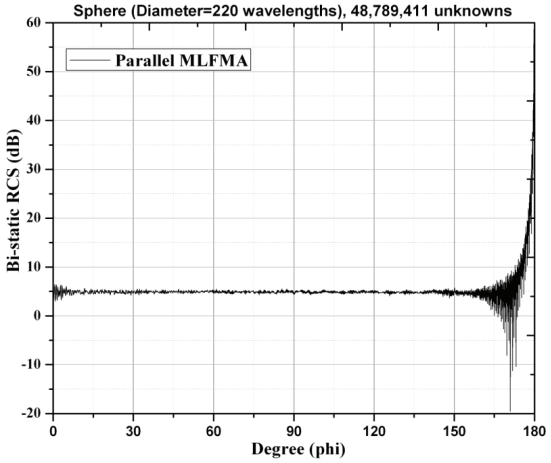


Fig. 5. The bi-static RCS for a sphere of diameter 220λ.

To further demonstrate the capability of our parallel MLFMA code, we calculate the bi-static RCS of a square patch of size of 200×200 wavelengths with 10,150,143 unknowns. Since this is an open structure, EFIE is used to solve this problem. The patch is located in the *yoz* plane with its center at the origin. The incident angle is (90°,0°) and the scanning plane is *xoy* with 1801 sampling points from (0°,180°). The parallelized GMRES is adopted to solve the matrix equation, and the residual error is 0.001. This simulation is carried out on one computer with 8 Xeon 3.0 GHz CPUs and 64GB memory. In this problem, only 4 CPUs are used. The detailed resources for this calculation are shown in Table 2.

Table 2. The computational resources for a square patch of 200λ by the parallel MLFMA.

General Information		CPU Time (min)	
Geometry size (wavelength)	200	Geometry information	2.6
Number of processors	4	Set up of near-field matrix	40.1
Number of iterations	142	Set up of far-field matrix	13.4
Total memory (GB)	14.7	Iteration and solution	624.9
Total time (hr)	11.5	RCS calculation	0.37

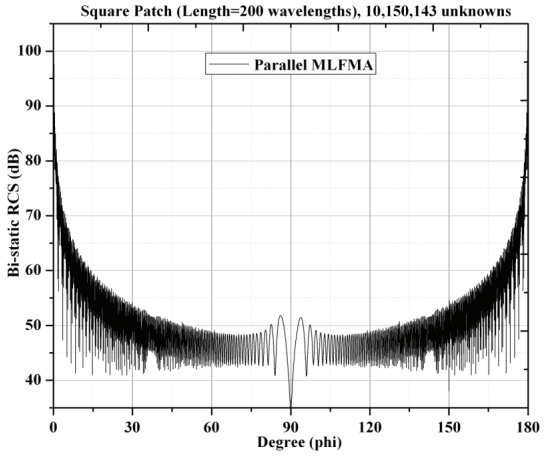


Fig. 6. The bi-static RCS for a square patch of length 200λ.

IV. CONCLUSIONS

In this paper, several implementation tricks of parallel MLFMA have been introduced and analyzed. Firstly, we proposed a modified truncation number method in order to reduce the memory and CPU time usage; secondly, a technique that sliced matrix to hard disk is applied to fulfill the memory demand for extremely large problems; finally, a transition level scheme is introduced in order to improve the parallel efficiency. With these tricks, memory usage can be reduced. We demonstrate the capability of our code by considering a sphere of diameter 220λ, containing nearly 50 million unknowns and a square patch with a length of 200λ, involving approximately 10 million unknowns.

REFERENCES

[1] W. C. Chew, J. M. Jin, E. Michielssen, and J. M. Song, *Fast and Efficient Algorithms in Computational Electromagnetics.* Norwood, MA, Artech House, 2001.

[2] S. Velamparambil, W. C. Chew, and J. M Song. "10 Million Unknowns: Is It That Big?" *IEEE Transaction on Antennas Propagation Magazine*, vol. 45, no. 2, pp. 43-58, April 2003.

[3] S. Velamparambil and W. C. Chew, "Analysis and performance of a distributed memory multilevel fast multipole algorithm," *IEEE Transactions on Antennas and Propagation*, vol. 53, no. 8, pp. 2719–2727, August 2005.

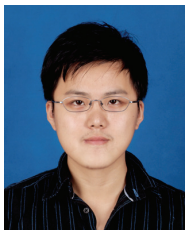
[4] X. M. Pan and X. Q. Sheng. "A Sophisticated Parallel MLFMA for Scattering by Extremely Large Targets". *IEEE Antennas and Propagation Magazine*, vol. 50, no. 3, pp. 129-138, June 2008.

[5] O. Ergul and L. Gurel, "Efficient Parallelization of the Multilevel Fast Multipole Algorithm for the Solution of Large-Scale Scattering Problems," *IEEE Transactions on Antennas and Propagation*, vol. 56, no. 8, pp. 2335-2345, August 2008.

[6] R. Coifmnan, V. Rokhlin, and S. Wandzura, "The Fast Multipole Method for the Wave Equation: A Pedestrian Prescription," *IEEE Antennas and Propagation Magazine*, vol. 35, no.3, pp. 7-12, June 1993.

[7] S. M. Rao, D. R. Wilton, and A. W. Glisson, "Electromagnetic scattering by surfaces of arbitrary shape," *IEEE Transactions on Antennas and Propagation*, vol. 30, no. 3, pp. 409–418, May 1982.

[8] T. Malas, O. Ergul and L. Gurel, "Sequential and Parallel Preconditioners for Large-Scale Integral- Equation Problems," *Computational Electromagnetics Workshop,* pp.35–43, August 2007.



Hu Fangjing received the B.S. degree in electromagnetic field and microwave technique from the University of Electronic Science and Technology of China (UESTC), Chengdu, in 2007, where he is currently working toward the M.S. degree. His research has mainly focused on fast integral equation methods and in particular, multilevel fast multipole method and its parallelization.

Nie Zaiping was born in Xi'an, China, in 1946. He received the B.S. degree in radio engineering and the M.S. degree in electromagnetic field and microwave technology from the Chengdu Institute of Radio Engineering (now UESTC: University of Electronic Science and Technology of China), Chengdu, China, in 1968 and 1981, respectively. From 1987 to 1989, he was a Visiting Scholar with the Electromagnetics Laboratory, University of Illinois, Urbana. Currently, he is a Professor with the Department of Electromagnetic Engineering, University of Electronic Science and Technology of China, Chengdu, China. He has published more than 380 papers. His research interests include antenna theory and techniques, field and waves in inhomogeneous media, computational electromagnetics, electromagnetic scattering and inverse scattering, new techniques for antenna in mobile communications, transient electro-magnetic theory and applications.

Hu Jun received the B.S., M.S., and Ph.D. degrees in electromagnetic field and microwave technique from the University of Electronic Science and Technology of China (UESTC), Chengdu, in 1995, 1998, and 2000, respectively. During 2001 he was with the Center of Wireless Communication in the City University of Hong Kong, Kowloon, as a Research Assistant. He is currently an Associate Professor with the School of Electronic Engineering of UESTC. He is the author or coauthor of over 90 technical papers. His current research interests include computational electromagnetics, electromagnetic scattering, and radiation.

2010 INSTITUTIONAL MEMBERS

DTIC-OCP LIBRARY
8725 John J. Kingman Rd, Ste 0944
Fort Belvoir, VA 22060-6218

AUSTRALIAN DEFENCE LIBRARY
Northcott Drive
Canberra, A.C.T. 2600 Australia

BEIJING BOOK CO, INC
701 E Linden Avenue
Linden, NJ 07036-2495

BUCKNELL UNIVERSITY
69 Coleman Hall Road
Lewisburg, PA 17837

ROBERT J. BURKHOLDER
OHIO STATE UNIVERSITY
1320 Kinnear Road
Columbus, OH 43212

DARTMOUTH COLLEGE
6025 Baker/Berry Library
Hanover, NH 03755-3560

DSTO EDINBURGH
AU/33851-AP, PO Box 830470
Birmingham, AL 35283

SIMEON J. EARL – BAE SYSTEMS
W432A, Warton Aerodome
Preston, Lancs., UK PR4 1AX

ELLEDIEMME
Libri Dal Mondo
PO Box 69/Poste S. Silvestro
Rome, Italy 00187

ENGINEERING INFORMATION, INC
PO Box 543
Amsterdam, Netherlands 1000 Am

ETSE TELECOMUNICACION
Biblioteca, Campus Lagoas
Vigo, 36200 Spain

OLA FORSLUND
SAAB MICROWAVE SYSTEMS
Nettovagen 6
Jarfalla, Sweden SE-17588

GEORGIA TECH LIBRARY
225 North Avenue, NW
Atlanta, GA 30332-0001

HRL LABS, RESEARCH LIBRARY
3011 Malibu Canyon
Malibu, CA 90265

IEE INSPEC
Michael Faraday House
6 Hills Way
Stevenage, Herts UK SG1 2AY

IND CANTABRIA
PO Box 830470
Birmingham, AL 35283

INSTITUTE FOR SCIENTIFIC INFO.
Publication Processing Dept.
3501 Market St.
Philadelphia, PA 19104-3302

KUWAIT UNIVERSITY
Postfach/po box 432
Basel, Switzerland 4900

LIBRARY – DRDC OTTAWA
3701 Carling Avenue
Ottawa, Ontario, Canada K1A OZ4

LIBRARY of CONGRESS
Reg. Of Copyrights
Attn: 407 Deposits
Washington DC, 20559

LINDA HALL LIBRARY
5109 Cherry Street
Kansas City, MO 64110-2498

RAY MCKENZIE – TELESTRA
13/242 Exhibition Street
Melbourne, Vic, Australia 3000

MISSISSIPPI STATE UNIV LIBRARY
PO Box 9570
Mississippi State, MS 39762

MISSOURI S&T
400 W 14th Street
Rolla, MO 56409

MIT LINCOLN LABORATORY
Periodicals Library
244 Wood Street
Lexington, MA 02420

OSAMA MOHAMMED
FLORIDA INTERNATIONAL UNIV
10555 W Flagler Street
Miami, FL 33174

NAVAL POSTGRADUATE SCHOOL
Attn:J. Rozdal/411 Dyer Rd./ Rm 111
Monterey, CA 93943-5101

NDL KAGAKU
C/0 KWE-ACCESS
PO Box 300613 (JFK A/P)
Jamaica, NY 11430-0613

OVIEDO LIBRARY
PO BOX 830679
Birmingham, AL 35283

PENN STATE UNIVERSITY
126 Paterno Library
University Park, PA 16802-1808

DAVID J. PINION
1122 E PIKE STREET #1217
SEATTLE, WA 98122

KATHERINE SIAKAVARA-
ARISTOTLE UNIV OF
THESSALONIKI
Gymnasiou 8
Thessaloniki, Greece 55236

SWETS INFORMATION SERVICES
160 Ninth Avenue, Suite A
Runnemede, NJ 08078

TIB & UNIV. BIB. HANNOVER
DE/5100/G1/0001
Welfengarten 1B
Hannover, Germany 30167

UNIV OF CENTRAL FLORIDA
4000 Central Florida Boulevard
Orlando, FL 32816-8005

UNIVERSITY OF COLORADO
1720 Pleasant Street, 184 UCB
Boulder, CO 80309-0184

UNIVERSITY OF KANSAS –
WATSON
1425 Jayhawk Blvd 210S
Lawrence, KS 66045-7594

UNIVERSITY OF MISSISSIPPI
JD Williams Library
University, MS 38677-1848

UNIVERSITY LIBRARY/HKUST
CLEAR WATER BAY ROAD
KOWLOON, HONG KONG

UNIV POLIT CARTAGENA
Serv Btca Univ,
Paseo Alfonso XIII, 48
Cartagena, Spain 30203

THOMAS WEILAND
TU DARMSTADT
Schlossgartenstrasse 8
Darmstadt, Hessen, Germany 64289

STEVEN WEISS
US ARMY RESEARCH LAB
2800 Powder Mill Road
Adelphi, MD 20783

YOSHIHIDE YAMADA
NATIONAL DEFENSE ACADEMY
1-10-20 Hashirimizu
Yokosuka, Kanagawa,
Japan 239-8686

ACES COPYRIGHT FORM

This form is intended for original, previously unpublished manuscripts submitted to ACES periodicals and conference publications. The signed form, appropriately completed, MUST ACCOMPANY any paper in order to be published by ACES. PLEASE READ REVERSE SIDE OF THIS FORM FOR FURTHER DETAILS.

TITLE OF PAPER:

AUTHORS(S)

PUBLICATION TITLE/DATE:

RETURN FORM TO:

Dr. Atef Z. Elsherbeni
University of Mississippi
Dept. of Electrical Engineering
Anderson Hall Box 13
University, MS 38677 USA

PART A - COPYRIGHT TRANSFER FORM

(NOTE: Company or other forms may not be substituted for this form. U.S. Government employees whose work is not subject to copyright may so certify by signing Part B below. Authors whose work is subject to Crown Copyright may sign Part C overleaf).

The undersigned, desiring to publish the above paper in a publication of ACES, hereby transfer their copyrights in the above paper to The Applied Computational Electromagnetics Society (ACES). The undersigned hereby represents and warrants that the paper is original and that he/she is the author of the paper or otherwise has the power and authority to make and execute this assignment.

Returned Rights: In return for these rights, ACES hereby grants to the above authors, and the employers for whom the work was performed, royalty-free permission to:

1. Retain all proprietary rights other than copyright, such as patent rights.
2. Reuse all or portions of the above paper in other works.
3. Reproduce, or have reproduced, the above paper for the author's personal use or for internal company use provided that (a) the source and ACES copyright are indicated, (b) the copies are not used in a way that implies ACES endorsement of a product or service of an employer, and (c) the copies per se are not offered for sale.
4. Make limited distribution of all or portions of the above paper prior to publication.
5. In the case of work performed under U.S. Government contract, ACES grants the U.S. Government royalty-free permission to reproduce all or portions of the above paper, and to authorize others to do so, for U.S. Government purposes only.

ACES Obligations: In exercising its rights under copyright, ACES will make all reasonable efforts to act in the interests of the authors and employers as well as in its own interest. In particular, ACES REQUIRES that:

1. The consent of the first-named author be sought as a condition in granting re-publication permission to others.
2. The consent of the undersigned employer be obtained as a condition in granting permission to others to reuse all or portions of the paper for promotion or marketing purposes.

In the event the above paper is not accepted and published by ACES or is withdrawn by the author(s) before acceptance by ACES, this agreement becomes null and void.

AUTHORIZED SIGNATURE — TITLE (IF NOT AUTHOR)

EMPLOYER FOR WHOM WORK WAS PERFORMED — DATE FORM SIGNED

Part B - U.S. GOVERNMENT EMPLOYEE CERTIFICATION

(NOTE: if your work was performed under Government contract but you are not a Government employee, sign transfer form above and see item 5 under Returned Rights).

This certifies that all authors of the above paper are employees of the U.S. Government and performed this work as part of their employment and that the paper is therefor not subject to U.S. copyright protection.

AUTHORIZED SIGNATURE — TITLE (IF NOT AUTHOR)

NAME OF GOVERNMENT ORGANIZATION — DATE FORM SIGNED

PART C - CROWN COPYRIGHT

(NOTE: ACES recognizes and will honor Crown Copyright as it does U.S. Copyright. It is understood that, in asserting Crown Copyright, ACES in no way diminishes its rights as publisher. Sign only if *ALL* authors are subject to Crown Copyright).

This certifies that all authors of the above Paper are subject to Crown Copyright. (Appropriate documentation and instructions regarding form of Crown Copyright notice may be attached).

AUTHORIZED SIGNATURE TITLE OF SIGNEE

NAME OF GOVERNMENT BRANCH DATE FORM SIGNED

Information to Authors

ACES POLICY

ACES distributes its technical publications throughout the world, and it may be necessary to translate and abstract its publications, and articles contained therein, for inclusion in various compendiums and similar publications, etc. When an article is submitted for publication by ACES, acceptance of the article implies that ACES has the rights to do all of the things it normally does with such an article.

In connection with its publishing activities, it is the policy of ACES to own the copyrights in its technical publications, and to the contributions contained therein, in order to protect the interests of ACES, its authors and their employers, and at the same time to facilitate the appropriate re-use of this material by others.

The new United States copyright law requires that the transfer of copyrights in each contribution from the author to ACES be confirmed in writing. It is therefore necessary that you execute either Part A-Copyright Transfer Form or Part B-U.S. Government Employee Certification or Part C-Crown Copyright on this sheet and return it to the Managing Editor (or person who supplied this sheet) as promptly as possible.

CLEARANCE OF PAPERS

ACES must of necessity assume that materials presented at its meetings or submitted to its publications is properly available for general dissemination to the audiences these activities are organized to serve. It is the responsibility of the authors, not ACES, to determine whether disclosure of their material requires the prior consent of other parties and if so, to obtain it. Furthermore, ACES must assume that, if an author uses within his/her article previously published and/or copyrighted material that permission has been obtained for such use and that any required credit lines, copyright notices, etc. are duly noted.

AUTHOR/COMPANY RIGHTS

If you are employed and you prepared your paper as a part of your job, the rights to your paper initially rest with your employer. In that case, when you sign the copyright form, we assume you are authorized to do so by your employer and that your employer has consented to all of the terms and conditions of this form. If not, it should be signed by someone so authorized.

NOTE RE RETURNED RIGHTS: Just as ACES now requires a signed copyright transfer form in order to do "business as usual", it is the intent of this form to return rights to the author and employer so that they too may do "business as usual". If further clarification is required, please contact: The Managing Editor, R. W. Adler, Naval Postgraduate School, Code EC/AB, Monterey, CA, 93943, USA (408)656-2352.

Please note that, although authors are permitted to re-use all or portions of their ACES copyrighted material in other works, this does not include granting third party requests for reprinting, republishing, or other types of re-use.

JOINT AUTHORSHIP

For jointly authored papers, only one signature is required, but we assume all authors have been advised and have consented to the terms of this form.

U.S. GOVERNMENT EMPLOYEES

Authors who are U.S. Government employees are not required to sign the Copyright Transfer Form (Part A), but any co-authors outside the Government are.

Part B of the form is to be used instead of Part A only if all authors are U.S. Government employees and prepared the paper as part of their job.

NOTE RE GOVERNMENT CONTRACT WORK: Authors whose work was performed under a U.S. Government contract but who are not Government employees are required so sign Part A-Copyright Transfer Form. However, item 5 of the form returns reproduction rights to the U. S. Government when required, even though ACES copyright policy is in effect with respect to the reuse of material by the general public.
January 2002

APPLIED COMPUTATIONAL ELECTROMAGNETICS SOCIETY JOURNAL
http://aces.ee.olemiss.edu

INFORMATION FOR AUTHORS

PUBLICATION CRITERIA

Each paper is required to manifest some relation to applied computational electromagnetics. **Papers may address general issues in applied computational electromagnetics, or they may focus on specific applications, techniques, codes, or computational issues.** While the following list is not exhaustive, each paper will generally relate to at least one of these areas:

1. **Code validation.** This is done using internal checks or experimental, analytical or other computational data. Measured data of potential utility to code validation efforts will also be considered for publication.

2. **Code performance analysis.** This usually involves identification of numerical accuracy or other limitations, solution convergence, numerical and physical modeling error, and parameter tradeoffs. However, it is also permissible to address issues such as ease-of-use, set-up time, run time, special outputs, or other special features.

3. **Computational studies of basic physics.** This involves using a code, algorithm, or computational technique to simulate reality in such a way that better, or new physical insight or understanding, is achieved.

4. **New computational techniques** or new applications for existing computational techniques or codes.

5. **"Tricks of the trade"** in selecting and applying codes and techniques.

6. **New codes, algorithms, code enhancement, and code fixes.** This category is self-explanatory, but includes significant changes to existing codes, such as applicability extensions, algorithm optimization, problem correction, limitation removal, or other performance improvement. **Note: Code (or algorithm) capability descriptions are not acceptable, unless they contain sufficient technical material to justify consideration.**

7. **Code input/output issues.** This normally involves innovations in input (such as input geometry standardization, automatic mesh generation, or computer-aided design) or in output (whether it be tabular, graphical, statistical, Fourier-transformed, or otherwise signal-processed). Material dealing with input/output database management, output interpretation, or other input/output issues will also be considered for publication.

8. **Computer hardware issues.** This is the category for analysis of hardware capabilities and limitations of various types of electromagnetics computational requirements. Vector and parallel computational techniques and implementation are of particular interest.

Applications of interest include, but are not limited to, antennas (and their electromagnetic environments), networks, static fields, radar cross section, inverse scattering, shielding, radiation hazards, biological effects, biomedical applications, electromagnetic pulse (EMP), electromagnetic interference (EMI), electromagnetic compatibility (EMC), power transmission, charge transport, dielectric, magnetic and nonlinear materials, microwave components, MEMS, RFID, and MMIC technologies, remote sensing and geometrical and physical optics, radar and communications systems, sensors, fiber optics, plasmas, particle accelerators, generators and motors, electromagnetic wave propagation, non-destructive evaluation, eddy currents, and inverse scattering.

Techniques of interest include but not limited to frequency-domain and time-domain techniques, integral equation and differential equation techniques, diffraction theories, physical and geometrical optics, method of moments, finite differences and finite element techniques, transmission line method, modal expansions, perturbation methods, and hybrid methods.

Where possible and appropriate, authors are required to provide statements of quantitative accuracy for measured and/or computed data. This issue is discussed in "Accuracy & Publication: Requiring, quantitative accuracy statements to accompany data," by E. K. Miller, *ACES Newsletter,* Vol. 9, No. 3, pp. 23-29, 1994, ISBN 1056-9170.

SUBMITTAL PROCEDURE

All submissions should be uploaded to ACES server through ACES web site (http://aces.ee.olemiss.edu) by using the upload button, journal section. Only pdf files are accepted for submission. The file size should not be larger than 5MB, otherwise permission from the Editor-in-Chief should be obtained first. Automated acknowledgment of the electronic submission, after the upload process is successfully completed, will be sent to the corresponding author only. It is the responsibility of the corresponding author to keep the remaining authors, if applicable, informed. Email submission is not accepted and will not be processed.

PAPER FORMAT (INITIAL SUBMISSION)

The preferred format for initial submission manuscripts is 12 point Times Roman font, single line spacing and single column format, with 1 inch for top, bottom, left, and right margins. Manuscripts should be prepared for standard 8.5x11 inch paper.

EDITORIAL REVIEW

In order to ensure an appropriate level of quality control, papers are peer reviewed. They are reviewed both for

technical correctness and for adherence to the listed guidelines regarding information content and format.

PAPER FORMAT (FINAL SUBMISSION)

Only camera-ready electronic files are accepted for publication. The term **"camera-ready" means that the material is neat, legible, reproducible, and in accordance with the final version format listed below.**

The following requirements are in effect for the final version of an ACES Journal paper:

1. The paper title should not be placed on a separate page. The title, author(s), abstract, and (space permitting) beginning of the paper itself should all be on the first page. The title, author(s), and author affiliations should be centered (center-justified) on the first page. The title should be of font size 16 and bolded, the author names should be of font size 12 and bolded, and the author affiliation should be of font size 12 (regular font, neither italic nor bolded).

2. An abstract is required. The abstract should be a brief summary of the work described in the paper. It should state the computer codes, computational techniques, and applications discussed in the paper (as applicable) and should otherwise be usable by technical abstracting and indexing services. The word "Abstract" has to be placed at the left margin of the paper, and should be bolded and italic. It also should be followed by a hyphen (—) with the main text of the abstract starting on the same line.

3. All section titles have to be centered and all the title letters should be written in caps. The section titles need to be numbered using roman numbering (I. II. ….)

4. Either British English or American English spellings may be used, provided that each word is spelled consistently throughout the paper.

5. Internal consistency of references format should be maintained. As a guideline for authors, we recommend that references be given using numerical numbering in the body of the paper (with numerical listing of all references at the end of the paper). The first letter of the authors' first name should be listed followed by a period, which in turn, followed by the authors' complete last name. Use a coma (,) to separate between the authors' names. Titles of papers or articles should be in quotation marks (" "), followed by the title of journal, which should be in italic font. The journal volume (vol.), issue number (no.), page numbering (pp.), month and year of publication should come after the journal title in the sequence listed here.

6. Internal consistency shall also be maintained for other elements of style, such as equation numbering. As a guideline for authors who have no other preference, we suggest that equation numbers be placed in parentheses at the right column margin.

7. The intent and meaning of all text must be clear. For authors who are not masters of the English language, the ACES Editorial Staff will provide assistance with grammar (subject to clarity of intent and meaning). However, this may delay the scheduled publication date.

8. Unused space should be minimized. Sections and subsections should not normally begin on a new page.

ACES reserves the right to edit any uploaded material, however, this is not generally done. It is the author(s) responsibility to provide acceptable camera-ready pdf files. Incompatible or incomplete pdf files will not be processed for publication, and authors will be requested to re-upload a revised acceptable version.

COPYRIGHTS AND RELEASES

Each primary author must sign a copyright form and obtain a release from his/her organization vesting the copyright with ACES. Copyright forms are available at ACES, web site (http://aces.ee.olemiss.edu). To shorten the review process time, the executed copyright form should be forwarded to the Editor-in-Chief immediately after the completion of the upload (electronic submission) process. Both the author and his/her organization are allowed to use the copyrighted material freely for their own private purposes.

Permission is granted to quote short passages and reproduce figures and tables from and ACES Journal issue provided the source is cited. Copies of ACES Journal articles may be made in accordance with usage permitted by Sections 107 or 108 of the U.S. Copyright Law. This consent does not extend to other kinds of copying, such as for general distribution, for advertising or promotional purposes, for creating new collective works, or for resale. The reproduction of multiple copies and the use of articles or extracts for commercial purposes require the consent of the author and specific permission from ACES. Institutional members are allowed to copy any ACES Journal issue for their internal distribution only.

PUBLICATION CHARGES

All authors are allowed for 8 printed pages per paper without charge. Mandatory page charges of $75 a page apply to all pages in excess of 8 printed pages. Authors are entitled to one, free of charge, copy of the journal issue in which their paper was published. Additional reprints are available for a nominal fee by submitting a request to the managing editor or ACES Secretary.

Authors are subject to fill out a one page over-page charge form and submit it online along with the copyright form before publication of their manuscript.

ACES Journal is abstracted in INSPEC, in Engineering Index, DTIC, Science Citation Index Expanded, the Research Alert, and to Current Contents/Engineering, Computing & Technology.