Hardware Accelerated Design of Millimeter Wave Antireflective Surfaces: A Comparison of Field-Programmable Gate Array (FPGA) and Graphics Processing Unit (GPU) Implementations

Ozlem Kilic[1], Miaoqing Huang[2], Charles Conner[1], and Mark S. Mirotznik[3]

[1]Department of Electrical Engineering and Computer Science
The Catholic University of America, Washington, DC 20064, USA
{kilic,connerc}@cua.edu

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

[3]Department of Electrical and Computer Engineering
University of Delaware, Newark, DE 19716, USA
mirotzni@ece.udel.edu

***Abstract*—**Engineered materials that demonstrate a specific response to electromagnetic energy incident on them in antenna and radio frequency component design applications are in high demand due to both military and commercial needs. The design of such engineered materials typically requires numerically intensive computations to simulate their behavior as they may have electrically small features on a large area or often the overall system performance is required, which means modeling the entire integrated system. Furthermore, to achieve an optimal performance these simulations need to be run many times until a desired solution is achieved, presenting a major hindrance in arriving at a feasible solution in a reasonable amount of time. One example of such applications is the design of antireflective (AR) surfaces at millimeter wave frequencies, which often involves sub-wavelength gratings in an electrically large multilayer structure. This paper investigates the use of field-programmable gate arrays (FPGAs) and graphics processing units (GPUs) as coprocessors to the CPU in order to expedite the computation time. Preliminary results show that the hardware implementation (100 MHz) on Xilinx Virtex4LX200 FPGA is able to outperform a single-thread software implementation on Intel Itanium 2 processor (1.66 GHz) by 20 folds. However, the performance of the FPGA implementation lags behind the single-thread implementation on a modern Xeon (2.26 GHz) by 3.6×. On the other hand, modern GPUs demonstrate an evident advantage over both CPU and FPGA by achieving 20× speedup than the Xeon processor.

***Index Terms*—**Antireflective Surface, Engineered Materials, FPGA, GPU, Parallel Computing, Reconfigurable Programming, High-Performance Computing.

I. INTRODUCTION

The design of engineered materials that demonstrate a specific response to incident electromagnetic energy often requires the use of periodic structures with dimensions that are much smaller than the wavelength for electrically large structures (i.e., overall size of many wavelengths). As a result, the accurate and fast modeling of these large scale structures with fine features often becomes a major challenge. The challenge is even bigger when these models have to be run iteratively to identify an optimal solution. Recently, hardware accelerated computing has been gaining momentum due to its applicability to parallel computing while using a fraction of the power requirement of the conventional microprocessors and requiring much less cost in comparison to supercomputers.

The objective of this paper is two folds: (i) investigate the use of FPGAs and GPUs as coprocessors to CPU in electromagnetic simulations, (ii) utilize the hardware acceleration in simulating complex devices and optimizing their performance. These objectives will be achieved in the context of the design of antireflective (AR) surfaces with sub-wavelength gratings.

A common approach to the design of AR surfaces in optical regimes is to coat the surface with multiple layers of thin films with specific dielectric properties that result in the desired performance. This approach is not practical at millimeter wave frequencies as there is limited availability of dielectric materials with the desired material properties. For this purpose, alternative techniques using gratings in the substrate can be used to simulate the same effect [1]. The gratings in essence modify the effective dielectric property of each layer. As a first order approximation, an effective permittivity for each layer can be computed using the effective media theory [2]. However, this approach is only suitable

Table 1: Comparison between FPGA and GPU

Criterion		FPGA	GPU
Power Consumption		Low (Virtex4LX200: ~10 W)	High (GTX 480: 450 W)
Cost		High (Virtex4LX200: ~$6,000)	Low (GTX 480: $500)
Programming	Learning Curve	Long	Short
	Difficulty	High (Use hardware discription language)	Low (Use high level language)
	Flexibility	High	Low
	Portability	Low	High
Floating-Point Performance		Low	High

for gratings with dimensions that are much smaller than the incident wavelength. For the case of sub-wavelength gratings (i.e., resonant regime) considered in this paper, a more precise approach is required as the assumptions of the effective media theory are no longer valid.

This paper uses the rigorous coupled wave (RCW) algorithm, which employs an eigenmode approach as described in [3], to model the AR surface created with sub-wavelength gratings. RCW algorithm applies to structures with periodic gratings. The electric field and the periodic permittivity values inside the structure are expanded into a Fourier series in spatial harmonics, resulting in a matrix of coupled wave equations. With this approach, the field inside the medium is expanded in terms of the space harmonics in the periodic structure and phase matched to the fields outside the grating. The fields can be treated as waveguide modes in the grating region, and the total field is expressed as a sum of all possible modes.

The remaining part of the paper is organized as follows. Section II describes the underlying principles of hardware acceleration and presents the features of FPGA and GPU with particular attention to the systems used in this implementation. The details of the RCW algorithm are provided and its numerically intensive components are identified in Section III. Section IV describes the implementation of the RCW algorithm on two different platforms: a state-of-the-art reconfigurable computer, SGI Altix RASC RC100 [4], and the NVIDIA GPUs (i.e., Tesla C1060 and GeForce GTX480). The platform specifications for the hardware implementation and the interaction between the CPU and the two hardware platforms are also presented in this section. Significant performance improvement has been achieved on both FPGA and GPU platforms compared with the software implementation on Intel Itanium 2 and Xeon E5520 processors. Finally, the conclusion remarks are given in Section VI.

II. HARDWARE ACCELERATION ON FPGA AND GPU

Parallelism and pipelining are in the essence of hardware accelerated computing. A more conventional way of hardware acceleration based on von-Neuman architecture, where instructions and data are stored in the same memory, is typically achieved by the use of multiple processors in a system. In this approach, instruction stream programming can be used as in any traditional computer. An example of such a system is the Beowulf cluster [5]. As an alternative, this paper focuses on a different kind of hardware acceleration, where a coprocessor is used to support the CPU for specific tasks in an algorithm. Traditional von-Neuman architectures tend to create bottlenecks between the CPU and the main memory. The use of a dedicated coprocessor with its own memory can accelerate numerically intensive computations. One of the early uses of such coprocessors is the digital signal processor (DSP), which is a highly specialized form of a microprocessor. While the use of DSPs was universal for hardware acceleration in its early stages, the growing need for flexibility for many computationally intensive applications outstripped the functionalities offered by these chips. As a result, FPGAs, which are a form of highly configurable hardware, began entering the market. FPGAs contain programmable logic components called "logic blocks", and a hierarchy of reconfigurable interconnects that allow the blocks to be connected together to perform custom computation. With the abundance of available transistors, modern FPGAs are capable of carrying out big scientific applications. Thousands of performance speedup has been observed on reconfigurable computers [6].

While FPGAs are highly reconfigurable and energy efficient, there is a prize for the flexibility offered by these platforms, i.e., the difficulty in hardware implementation. Typically, hardware description languages, such as Verilog and VHDL, are required to program the FPGAs in order to achieve desirable performance speedup.

Recently, another platform, i.e., graphics processing unit, has been gaining popularity due to their relatively easier learning curve. GPU was presented as early as in 1989 as a stream computing engine [7]. Modern GPUs from both NVIDIA and AMD consist of hundreds of stream processing units and are capable of achieving remarkable processing parallelism. Both companies provide SDK to facilitate the end user use high level languages (e.g., C language) to program the GPUs, therefore significantly lowering the programming difficulty.

FPGAs and GPUs, have demonstrated the ability to speed up a wide range of applications from image processing to encryption, as demonstrated in previous work [8]–[12]. Each technology has its advantage and disadvantage, as listed in Table 1. In general, GPU provides the ease of use and higher parallelism. On the other hand, FPGA consumes much less power, has better programming flexibility, and provides deep pipelining, which is very useful for many applications.

III. RIGOROUS COUPLED WAVE ALGORITHM

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

$$\begin{bmatrix} \partial^2 S_{l,y}/\partial z'^2 \\ \partial^2 S_{l,x}/\partial z'^2 \end{bmatrix} = \Omega_l \begin{bmatrix} S_{l,y} \\ S_{l,x} \end{bmatrix}, \tag{1a}$$

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

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

matrix approach in [13] and adopts Lalanne's improved eigenvalue formalism [14]. A detailed discussion on the RCW algorithm can be found in these references. 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.

$$\varepsilon_l(x,y) = \sum_{g,h} \varepsilon_{l,gh} \exp\left(j\frac{2\pi gx}{\Lambda_x} + j\frac{2\pi hy}{\Lambda_y}\right), \tag{3a}$$

$$\varepsilon_l^{-1}(x,y) = \sum_{g,h} A_{l,gh} \exp\left(j\frac{2\pi gx}{\Lambda_x} + j\frac{2\pi hy}{\Lambda_y}\right). \tag{3b}$$

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 (1), 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 (1b) are matrices given as

$$B = k_x\varepsilon_l^{-1}k_x - I, \tag{4a}$$

$$D = k_y\varepsilon_l^{-1}k_y - I. \tag{4b}$$

The k_x and k_y in (1b) and (4) are diagonal matrices formed by k_{x_m} and k_{y_n} as shown in (5), in which k_0 is the free space wave number.

$$k_x = \frac{k_{x_m}}{k_0}, \tag{5a}$$

$$k_y = \frac{k_{y_n}}{k_0}. \tag{5b}$$

k_{x_m} and k_{y_n} are the wave vector components along x and y, respectively. They are computed from phase matching and Floquet conditions as (6).

$$k_{x_m} = k_0\left(n_1\sin\theta\cos\varphi - m\left(\frac{\lambda_0}{\Lambda_x}\right)\right), \tag{6a}$$

$$k_{y_n} = k_0\left(n_1\sin\theta\sin\varphi - n\left(\frac{\lambda_0}{\Lambda_y}\right)\right). \tag{6b}$$

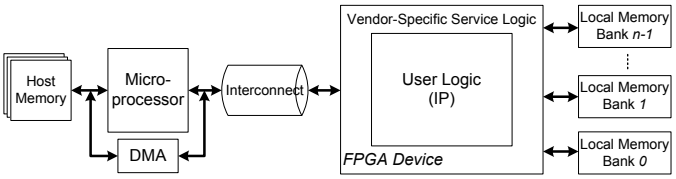
Λ_x and Λ_y in (3) and (6) represent the periodicity of the gratings along x and y respectively. α in (1b) is a grating geometry dependent parameter, which is a real positive number between $[0, 1]$ as introduced in [14].

Therefore, 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 RCW algorithm is this eigenvalue computation. The hardware platforms will be used to implement the eigenvalue computations of the RCW algorithm to achieve acceleration.

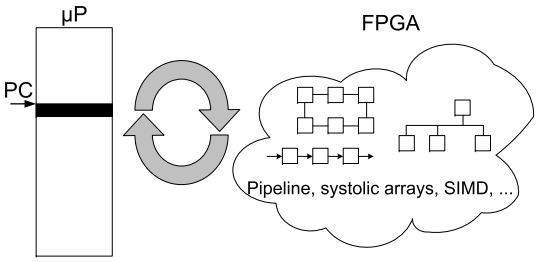
A. QR eigenvalue algorithm

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 [15] 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 [16]–[18] is accepted as a practical solution and adopted in most applications to deal with general square matrices.

There are two phases in the practical QR algorithm, as described in (2). In the first phase, the original matrix A is re-



(a) General architecture of a reconfigurable computer



(b) Execution model

Fig. 1. Using FPGAs as coprocessors in general-purpose computing.

duced to the upper Hessenberg form H using the Householder transformation [19]. 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 (2), and the eigenvalue problem is solved once this form is achieved. If the corresponding eigenvectors are required, they can be calculated using Gaussian elimination and back substitution after the eigenvalues are available.

IV. HARDWARE IMPLEMENTATION OF THE RCW ALGORITHM

Both hardware platforms (e.g., FPGA and GPU) will be used as coprocessors to the CPU to accelerate the most numerically intensive part of the RCW algorithm, which is the eigenvalue calculation for the large matrices required in the RCWA design. Promising results are demonstrated to prove the efficiency of the hardware implementation compared with the software implementation of the same algorithm in C. The acceleration in computation time allows for the design and optimization of complex AR surfaces as numerous iterations can be run rapidly on hardware coprocessors.

A. Implementation on Altix RASC RC100 reconfigurable computer

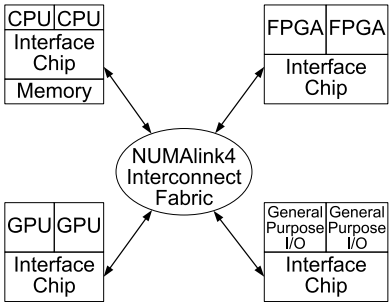
Reconfigurable computers (RCs) are traditional computers extended with coprocessors based on reconfigurable hardware like FPGAs. These enhanced systems are capable of providing significant performance improvement for applications in many scientific and engineering domains, such as the electromagnetics [20], [21]. Due to the limited size of the internal block RAM memory, multiple SRAM modules are generally connected to the hardware coprocessor for data storage, such as the example shown in Figure 1(a).

The implementation of an application on a reconfigurable computer consists of a hardware part and a software part. The implementation on the hardware part requires the use of either hardware description languages (e.g., VHDL, Verilog) or high level languages, such as Impulse-C [22] or Mitrion-C [23], to carry out the design in hardware. Multiple techniques, e.g., pipelining and single instruction multiple data (SIMD), can be applied to take advantage of the hardware acceleration. Multiple dependent tasks in an application can form a pipeline so that the output of a producer can be forwarded to the input of a consumer directly. Take the circuit in Figure 3(a) as one example, multiple primitive operators form a pipeline to accomplish an advanced operation. Another typical technique, SIMD as shown in Figure 3(c), is to instantiate multiple identical processing elements (PEs) so that multiple data items can be processed in parallel. The theoretical performance of N identical PEs is N times of a single PE.

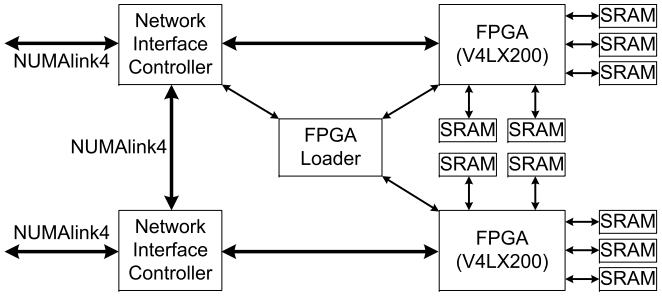
Since the hardware implementation depends on the available resources on the FPGA device (e.g., memory, built-in multipliers, slices), it might be necessary to distribute the hardware part into multiple FPGA configurations, each of which is called a *bitstream*. Once the bitstreams are available, they can be integrated into the software part, which is executed on the CPU. From the point of view of a software programmer, a bitstream can be treated as a software subroutine during the integration process in spite of the fact that the functionality is realized in hardware, as shown in Figure 1(b). The integration process always involves the use of vendor application programming interfaces (APIs).

In the following text, the numerically intensive part of the RCW algorithm, i.e., the eigenvalue solver, is described in terms of the mathematical approach used for the implementation. This discussion is followed by the details of the implementation of the eigenvalue algorithm on the Altix RASC RC100 reconfigurable computer along with a description of the system specifications and architecture of the platform.

1) The FPGA Platform: SGI's Altix RASC RC100 reconfigurable computer is a blade-based heterogeneous supercomputer in which NUMAlink™4 interconnect is used to connect different types of computing blades, as shown in Figure 2(a). Each blade itself is a homogeneous node consisting of the same type of processors, e.g., the CPU or the FPGA coprocessors. The Altix 450 at The Catholic University of America includes two CPU blades and one FPGA blade. The CPU used in the system is Intel Itanium 2 (1.66 GHz). The detailed architecture of a RASC RC100 FPGA blade is shown in Figure 2(b). There are two FPGA devices on a single RASC blade. Each FPGA device, Xilinx Virtex-4LX200, is equipped with 5 banks of SRAM for local data storage. The size of each SRAM bank is 8 MB. Every bank has separate 64-bit read and write ports directly connected to the FPGA device. Besides the local memory, each FPGA device is capable of communicating with CPU blades through



(a) Multi-paradigm computing



(b) RASC RC100 FPGA blade architecture

Fig. 2. Altix RASC RC100 reconfigurable computer.

NUMAlink™4 interconnect, achieving a theoretical 3.2 GB/s on both directions at the same time. Since the two FPGA devices have their separate network interface controller, their communication with other components in the RASC system is independent to each other. However, there is no direct communication channel between the two FPGA device on the same board. This limitation prevents an application from being implemented on two devices, i.e., part of an application on one device and part of the same application on the other device. In other words, the hardware part of an application can be only implemented on a single FPGA device.

There are several factors that can limit the problem size an application can deal with when it (or part of it) is implemented on FPGA device. The first one is the number of basic lookup tables (LUTs) or combined as slices on Xilinx FPGAs. The bigger an application is, the more hardware resource its implementation is going to take. The second one is the number of built-in multipliers. Many scientific and engineering applications involve the double precision floating-point operations, particularly multiplications. Basic LUTs can be used to construct double precision multipliers. However, a more economic way is to use the built-in multipliers so that LUTs can be used for other part of the application. The third limiting factor is the size and the bandwidth of the off-chip memory. The size of the memory will decide how much data (including source, intermediate and result data) can be stored. The bandwidth of the memory will decide the data processing parallelism the logic can achieve. In this work, it is mainly the

Algorithm 1: Hessenberg Reduction (Vector-based)

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 $v = \frac{u}{\|u\|}$;

size of the memory that decide the maximum problem size the application can deal with, as elaborated in the following text.

2) FPGA implementation of QR algorithm: The RCW algorithm in the most general sense creates a square matrix with complex entries. Both real part and imaginary part 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 complete matrix entry. Therefore, the real part and the imaginary part of a complex value can be accessed simultaneously.

As described in Section III-A, there are two phases in the QR algorithm. The first phase, i.e., the Hessenberg reduction, is completely implemented in one FPGA configuration. Part of the second phase in which the computation is close to the one in Hessenberg reduction is implemented in another separate FPGA configuration. Since the computation in both configurations is close, we focus on the description of Hessenberg reduction in this paper.

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 2. 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. More specifically, the 10 sub-steps are carried out in a sequence during the execution. The advantage of hardware implementation comes from the pipelined processing within each sub-step. For example, Step 1.1 involves multiplication, addition, accumulation, and square root operation to calculate the norm of a vector. In hardware

Table 2: 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.

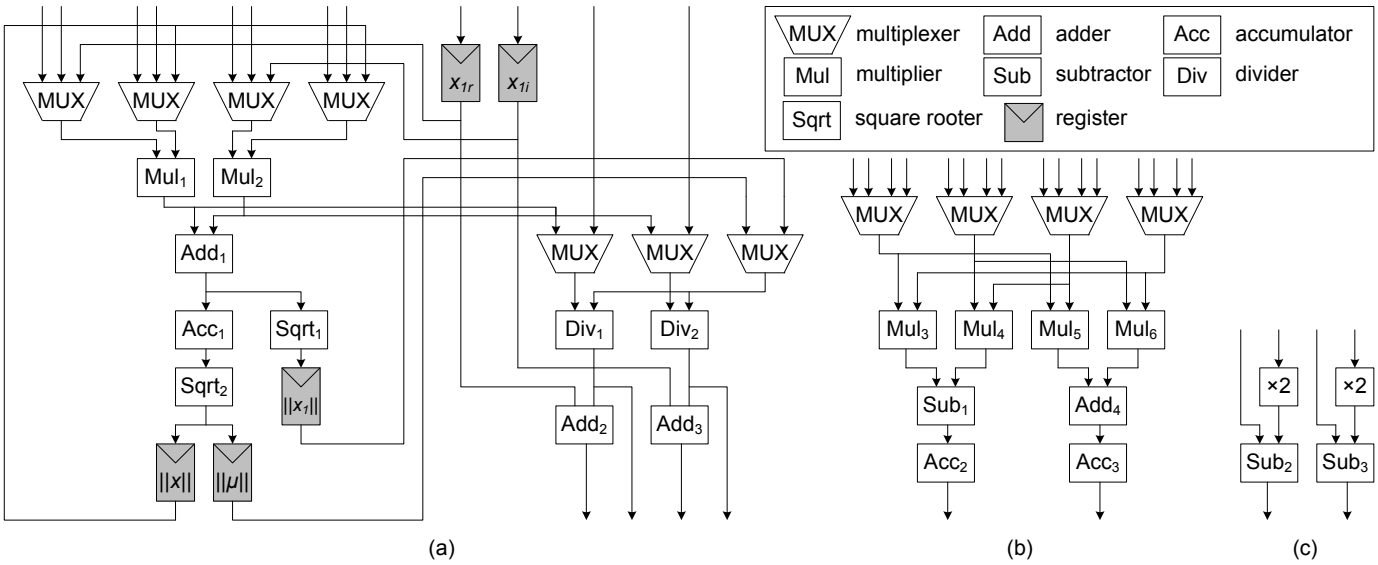


Fig. 3. The computing blocks in the hardware implementation: (a) the computing block used in Step 1; (b) the computing block used in Step 2.1, 2.2, 3.1, 3.2; (c) the computing block used in Step 2.3, 3.3. (*Note: (1) all inputs and outputs are connected to the local memory interface; (2) the control logic is not illustrated in the figure*).

implementation, these four operations are carried out in four operators, which are concatenated together to form a pipeline, as shown in Figure 3(a). These primitive operators are all fully pipelined in our design such that one new data item can be fed into the pipeline every clock cycle. Therefore, it will take roughly $n-k-1$ clock cycles to finish this sub-step (if we ignore all potential latencies). Table 2 lists the number of required clock cycles for each sub-step. By putting all iterations together, the total number of clock cycles required to reduce a matrix of rank n to its Hessenberg form can be computed as:

$$\sum_{k=0}^{n-3}(3k^2-9nk+6n^2-3n-2)=\frac{5}{2}n^3-\frac{9}{2}n-11. \quad (7)$$

The detailed hardware implementation of the computing blocks is illustrated in Figure 3. Since multiple steps have to be carried out sequentially, many basic computing units are re-used to reduce the resource cost. For example, the pipeline chain consisting of Mul_1, Mul_2, Add_1, Acc_1, and $Sqrt_2$ are re-used in Step 1.1 and Step 1.3 to compute $\|x\|$ and $\|\mu\|$, respectively. $\cos\varphi$ and $\sin\varphi$ are calculated on the fly by using division, i.e., $x_{1r}/\|x_1\|$ and $x_{1i}/\|x_1\|$. Therefore, the outputs of Step 1.2 correspond to the output of Add_2 (i.e., $x_{1r}+\|x\|\cdot x_{1r}/\|x_1\|$) and Add_3 (i.e., $x_{1i}+\|x\|\cdot x_{1i}/\|x_1\|$). The multiplication between matrix/vector and vector/vector in Step 2.1, 2.2, 3.1, and 3.2 is realized using the pipeline chain in Figure 3(b). Both the real part and the imaginary part of a complex entry are computed simultaneously. The control logic is not illustrated in Figure 3. It is mainly composed by three components, i.e., (1) a finite state machine whose statuses represent different steps and sub-steps, (2) the logic to generate correct read and write address for memory access, and (3) the logic to control the operations of the units in Figure 3.

The hardware implementation of Hessenberg reduction occupies 56,520 (63%) slices on the target FPGA device and runs at 100 MHz. The primitive operators, i.e., the double precision floating-point adder, multiplier, divider, are generated by using Xilinx CORE Generator. The accumulator is composed of adders and FIFOs. The hardware design is coded in Verilog, synthesized by Xilinx XST, placed and routed by Xilinx

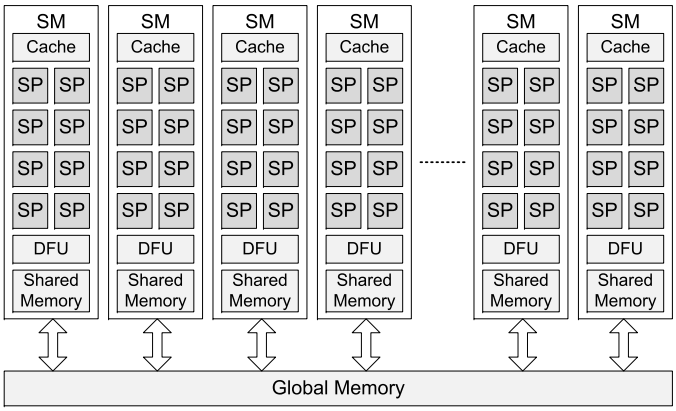


Fig. 4. The general architecture of an NVIDIA GT200 GPU (*SM: streaming multiprocessor, SP: streaming processor, DFU: double-precision floating-point unit*).

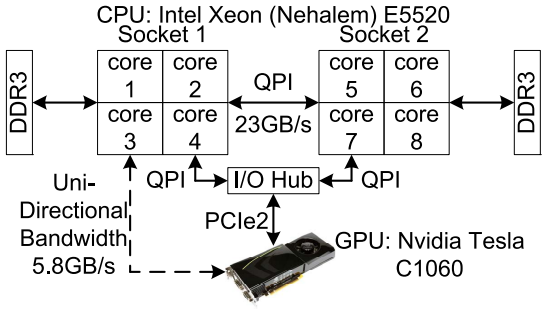


Fig. 5. The heterogeneous CPU-GPU board.

ISE 10.1. The operating frequency of the design is mainly limited by the control logic. The hardware design is capable of handling the matrix with a rank up to 480. The maximum size of the matrix is limited by the size of the off-chip memory in this case as a 480×480 complex matrix takes almost 8 MB to store its entries. The other off-chip memory is used to store the intermediate result in the execution. During the runtime, the rank of the object matrix is passed to the hardware design as a parameter through a register. Before the FPGA starts processing, the original matrix as well as its rank are transferred from the host to the FPGA. After the processing is finished, the upper Hessenberg matrix is transferred back to the host memory.

B. Implementation on NVIDIA GPUs

General-purpose computing on graphics processing units (GPGPU) is the technique of using GPUs to perform computation in applications traditionally handled by the microprocessors. GPUs are designed traditionally for graphics and thus are very restrictive in terms of operations and programming. Due to their nature, GPUs are only effective at tackling problems that can be solved using stream processing and the hardware can only be used in certain ways. More precisely, GPUs are efficient to process the independent elements belonging to a stream in a parallel fashion. Kernels are the functions that are applied to each element in the stream. Figure 4 illustrates a general architecture of an NVIDIA GT200 GPU consisting of many streaming processors (SPs).

We have implemented Alg. 1 on both NVIDIA Tesla C1060 and GeForce GTX 480 (Fermi) GPUs. The Tesla C1060 is installed on a dual-socket Intel Xeon workstation, as shown in Figure 5. The GTX 480 is installed on an Intel Core i7 workstation. On both workstations, GPU communicate with CPU through PCI Express 2 $\times$16 bus. Tesla C1060 (architecture code-named GT200) features 30 Streaming Multiprocessors, each of which is further composed of eight single precision floating-point CUDA streaming processors and one double precision floating-point unit, with 16KB on-chip storage called shared memory and 64KB of register windows for massive threading. The total 240 (single precision) + 30 (double precision) floating-point processors can achieve an observed peak performance of 78 GFLOPS for double precision operation. The Tesla GPU is equipped with 4 GB GDDR3 memory on board with the theoretical memory bandwidth of 102 GB/s. The uni-directional bandwidth of the PCI Express 2 bus on the platform is observed at 5.8 GB/s.

The latest GPU offered by NVIDIA is code-named as Fermi, which takes a significant leap forward in architecture highlighted by features such as improved double precision performance and configurable cache hierarchy. The model GTX 480 used in our experiments is composed of 15 newly designed streaming multiprocessors (SMs). Each SM features 32 CUDA streaming processors and is capable of 16 double precision fused multiply-add operations per clock, which is an 8$\times$ improvement over the GT200 architecture. Another key architectural difference is that Fermi has two instruction dispatch units and most instructions can be dual-issued, which is different from the HyperThreads used in the Intel Nehalem processors. Two HyperThreads within a single core of Nehalem processors share a single instruction fetch and decoding unit.

The GPU implementations are developed using CUDA [24]. The vector-based diagonal factorization is composed of a major outer loop that factorizes one column/row per step. Unfortunately, advanced features offered on the GPU such as asynchronized communication/computation and concurrently kernel execution cannot be used for such an algorithm, as dependency exists among the outer loops and all inner steps. Therefore the GPU implementation suffers from low occupancy for small problem sizes. In order to optimize the GPU implementation, firstly we managed to squeeze every inner computation step except the Householder generator (i.e., Step 1 in Table 2) into the GPU to keep the entire matrix remained in the GPU memory throughout the computation. In other words, the Hessenberg reduction is a CPU-GPU co-design on the hybrid platform as shown in Figure 5. Step 1 in Table 2 is carried out on CPU and the remaining two steps are executed on GPU. Fortunately the calculation of Step 1 only needs the transportation of one column (or part of a column) of a matrix. Therefore we managed to minimize the round trip communication overhead to approximately 5% of overall execution time. All kernels are further incrementally optimized

Table 3: Platform Characteristics

Criteria	Xeon (Nehalem)	Tesla C1060	GTX 480
Cores	4	240/30	480
Frequency (GHz)	2.26	1.3	1.4
Double Precision GFLOPs	36	78	672
Memory Bandwidth (GB/s)	25.6	102	177.4

through memory coalescing, using of shared memory and assigning more work per thread. The configurable L1 cache on the Fermi GPU introduces more design tradeoffs for users. In our experiments, for kernels with limited or no usage of shared memory, configuring the L1 to be 48KB can yield an approximately 10% improvement on GTX 480. Moreover, we found that the multi-dimensional threads and blocks configuration can also affect the cache performance, especially when the performance differences are examined on both GT200 and Fermi. We achieved the best performance mostly at the thread configuration of 32×8 for the Fermi GPU.

V. RESULTS

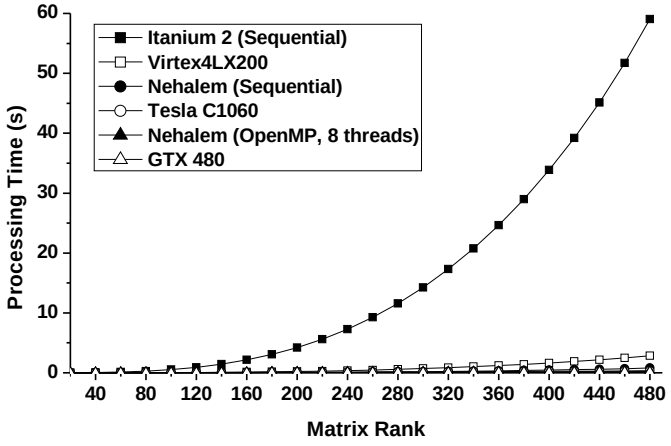
Due to the data dependency within the QR eigenvalue algorithm, it is found that the first phase, i.e., the Hessenberg reduction, is able to get significant performance improvement through hardware acceleration technologies. Therefore, we present the performance result of Hessenberg reduction on different platforms in this section. In order to demonstrate the benefit of FPGA and GPU implementations, we implemented Alg. 1 on two CPUs as reference, i.e., Intel Itanium 2 (1.66 GHz) on the RASC RC100 platform and Intel Xeon E5520 on the Tesla C1060 platform.

A. Performance comparison

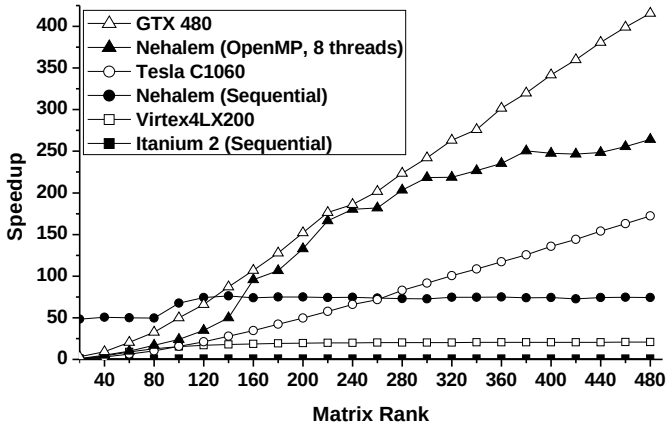
For comparison of acceleration over pure software based implementations, we coded the Hessenberg reduction phase in C++ on two software platforms.

The first platform is the RASC RC100 workstation with Intel Itanium 2 at 1.66 GHz. The size of L1 cache and L2 cache of the microprocessor is 16KB and 256KB [25], respectively. The software implementation on Itanium 2 is a sequential and direct implementation of Alg. 1. This sequential implementation is handcoded in C++ and single-threaded.

The second platform is a dual-socket Intel Xeon (Nehalem) system, as shown in Figure 5. The CPU is clocked at 2.26GHz with 8MB shared L3 cache and 12GB DDR3 memory (total 24GB for the entire system). The theoretical peak double precision floating-point performance is 36 GFLOP/S for each CPU. We implemented both sequential and parallel versions on Xeon. The sequential implementation is same to the one on Itanium processor. The parallel version is parallelized using OpenMP [26]. The critical computing intensive paths are parallelized by multiple threads first then further vectorized by the compiler utilizing the SSE units per core. These optimizations are achieved by enabling compiler optimization flags in GCC, such as *-sse4.2 -mtune=core2*. Furthermore, in order to achieve better scalability on all eight cores of both CPUs, we manually optimized our OpenMP code for better data locality control and further applied *numactl* to



(a) Computation time



(b) Speedup against sequential implementation on Itanium 2

Fig. 6. Performance comparison of the vector-based Hessenberg reduction.

bind threads to physical CPU cores to avoid the NUMA penalty. Such an optimization significantly improves overall performance on two CPUs for up to 60%.

Overall, the vector-based Hessenberg reduction has been realized in 6 different implementations on three platforms as follows.

- The FPGA implementation;
- The Tesla C1060 GPU implementation;
- The GTX 480 GPU implementation;
- The sequential software implementation on Itanium 2;
- The sequential software implementation on Xeon E5520;
- The parallel software implementation of OpenMP.

We had another implementation by using Intel MKL library, which runs 8 threads on the two Xeon processors. However, the performance of the Intel MKL parallel implementation is close to the OpenMP implementation. Therefore, the performance result of MKL implementation is not included in this paper. The comparison among these 6 implementations

is illustrated in Figure 6, which includes both computation time and the speedup against the software implementation on Itanium 2. The computation time on both FPGA and GPU is the end-to-end time including data communication time and data processing time on the coprocessors. The FPGA configuration time is not counted, however.

From Figure 6(b), it can be found that the FPGA implementation is able to outperform the Itanium 2 by 20 folds. Both Virtex-4 and Intel Itanium 2 were technologies around Year 2005, and the FPGA implementation has the big advantage than the CPU when the device was just released to the market. However, the performance of the FPGA implementation lags behind the state-of-the-art microprocessor and the GPUs with a big margin. The inferior performance of FPGA is mainly due to three factors. (i) The FPGA device is running at a very low frequency, i.e., 100 MHz. If the FPGA device is running at the same speed as the microprocessor, their performance will tie. (ii) The direct implementation of Alg. 1 is a sequential process due to the data dependency. Although we have tried to parallelize the hardware implementation to the extreme, its performance is easily surpassed by modern multicore processors with improved design on cache and SSE when dealing with applications such as Hessenberg reduction. (iii) The 5 local memory banks on the current platform become the limiting factor to increase the parallelism in the hardware implementation. More memory banks are desired to achieve higher parallelism on FPGA device.

The state-of-the-art microprocessor used in the experiments, Intel Xeon processor, demonstrates a remarkable performance improvement than the Itanium 2. For example, the sequential implementation on Xeon outperforms the sequential implementation on Itanium 2 for 50 folds. Putting multiple cores in a single processor further improves its performance, which is contributed mainly by two factors. First, the SSE extension in modern processor provides the vector processing capability, which fits the computation pattern in the target application very well. Second, the target application is a streaming application in which the computation can be distributed onto multiple cores to parallelize the data processing. Due to the data distributing overhead, the benefit for using multiple cores can be achieved only when the problem size is big enough, e.g., the rank of the matrix reaches 150 in Fig 6(b).

It is evident that it will be beneficial to implement the application on GPU as the matrix rank increases. The Tesla implementation surpasses the sequential software implementation on Xeon at rank 260 and then approaches the parallel software implementation afterwards. Fermi consistently outperforms GT200 for approximately 4×. Two factors mainly contribute to the performance improvement on GPU architecture. The first one is the massive parallel computing capability provided by the hundreds of streaming processors. As the rank of the matrix increases, the occupancy of the streaming processors improves accordingly as well as the speedup. The second factor is the very high bandwidth provided by the graphics DDR memory. As shown in Table 3, the memory bandwidth on GPU is 7 times of the memory bandwidth on CPU. The high bandwidth is very beneficial when the data need to be frequently accessed from the memory.

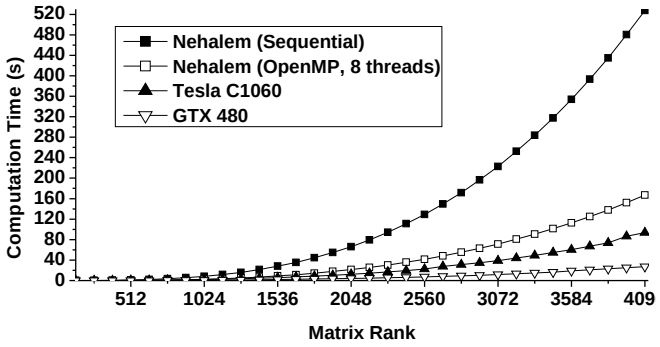


Fig. 7. Performance scalability of vector-based implementations of Hessenberg reduction.

B. Scalability

In the previous test, we limit the matrix rank at 480 because it is the biggest size the FPGA design can accommodate due to the size of the off-chip memory. In the meantime, it is clearly demonstrated that GPUs are capable of outperforming multicore CPUs as the matrix rank increases. In order to completely show the performance potential of GPUs, we compare them with the sequential and 8-thread x86 implementations on the Xeon platform (shown in Figure 5) with the matrix rank up to 4,096. By observing Figure 7, the implementation on Tesla C1060 is generally 2 times faster than the 8-thread Xeon implementation. The main reason has been described as above. The GTX 480 GPU outperforms all other versions consistently with a big margin. The Hessenberg reduction is a computation-intensive as well as communication-intensive problem. The abundant streaming processors and the high memory bandwidth on the Fermi architecture evidently give the advantage of GTX 480 compared with other technologies.

VI. CONCLUSION

Using FPGAs and GPUs as coprocessors to CPUs in parallel computing has been demonstrated in the context of an engineered material design, where the numerically intensive components of the RCW algorithm were implemented on these hardware acceleration technologies. The performance speedup on both coprocessors compared with software implementations on modern microprocessors are very impressive, proving both platforms are very suitable in scientific applications.

ACKNOWLEDGMENT

The authors would like to thank Lingyuan Wang with The George Washington University for the implementation of Hessenberg reduction on GPUs and Xeon.

REFERENCES

[1] D. H. Raguin and G. M. Morris, "Antireflection Structured Surfaces for the Infrared Spectral Region," *Applied Optics*, vol. 32, no. 7, pp. 1154–1167, 1993.

[2] P. Lalanne and J. Hugonin, "High-Order Effective-Medium Theory of Subwavelength Gratings in Classical Mounting: Application to Volume Holograms," *Journal of the Optical Society of America A*, vol. 15, no. 7, pp. 1843–1851, 1998.

[3] E. Noponen and J. Turunen, "Eigenmode Method for Electromagnetic Synthesis of Diffractive Elements with Three-Dimensional Profiles," *Journal of the Optical Society of America A*, vol. 11, no. 9, pp. 2494–2502, 1994.

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

[5] T. Sterling, D. Becker, M. Warren, T. Cwik, J. Salmon, and B. Nitzberg, "An Assessment of Beowulf-Class Computing for NASA Requirements: Initial Findings from the First NASA Workshop on Beowulf-Class Clustered Computing," in *Proc. 1998 IEEE Aerospace Conference*, pp. 367–381, Mar. 1998.

[6] 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.

[7] A. Fournier and D. Fussell, "On the Power of the Frame Buffer," *ACM Transactions on Graphics*, vol. 7, no. 2, pp. 103–128, Apr. 1988.

[8] B. de Ruijsscher, G. N. Gaydadjiev, J. Lichtenauer, and E. Hendriks, "FPGA Accelerator for Real-Time Skin Segmentation," in *Proc. the 2006 IEEE/ACM/IFIP Workshop on Embedded Systems for Real Time Multimedia (ESTMED'06)*, pp. 93–97, 2006.

[9] J. Krüger and R. Westermann, "Linear Algebra Operators for GPU Implementation of Numerical Algorithms," in *Proc. International Conference on Computer Graphics and Interactive Techniques (ACM SIGGRAPH'03)*, pp. 908–916, 2003.

[10] L. Nyland, M. Harris, and J. Prins, "Fast N-body Simulation with CUDA," in *GPU Gems 3 (H. Nguyen: editor)*, Aug. 2007.

[11] Z. K. Baker and V. K. Prasanna, "Efficient Hardware Data Mining with the Apriori Algorithm on FPGAs," in *Proc. the 13th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'05)*, pp. 3–12, Apr. 2005.

[12] S. Che, J. Li, J. W. Sheaffer, K. Skadron, and J. Lach, "Accelerating Compute-Intensive Applications with GPUs and FPGAs," in *Proc. the 2008 Symposium on Application Specific Processors (SASP'08)*, pp. 101–107, 2008.

[13] 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.

[14] 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.

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

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

[17] J. G. F. Francis, "The QR Transformation, II," *The Computer Journal*, vol. 4, no. 4, pp. 332–345, 1962.

[18] 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.

[19] A. S. Householder, "Unitary Triangularization of a Nonsymmetric Matrix," *Journal of the ACM*, vol. 5, no. 4, pp. 339–342, Oct. 1958.

[20] O. Kilic, M. S. Mirotznik, and J. P. Durbano, "Application of FPGA Based FDTD Simulators to Rotman Lenses," in *Proc. 22nd ACES Conference*, 2006.

[21] 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.

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

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

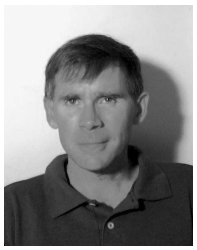
[24] *Nvidia CUDA Programming Guide 2.3.1*, Nvidia Corporation, Aug. 2009.

[25] *Dual-Core Update to the Intel Itanium 2 Processor Reference Manual*, Intel Corporation, Jan. 2006.

[26] *http://openmp.org*

Ozlem Kilic graduated from The George Washington University (1996) with a D.Sc. degree in Electrical Engineering. She is presently an Assistant Professor with The Catholic University of America. Before joining CUA, she worked at the U.S. Army Research Laboratories, Adelphi, MD and COMSAT Laboratories, Clarksburg, MD. She also holds the position of Senior Research Engineer for the Naval Surface Warfare Center (NSWC), Carderock Division. 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.



Charles Conner graduated from The Catholic University of America (Ph.D.) in 1999. He serves as a faculty member in the Electrical Engineering Department of the Capitol College, as well as an adjunct faculty at The Catholic University of America. He has been mainly working on various signal processing and computational problems for various government agencies. His specialty is digital signal processing and algorithm development.

Mark S. Mirotznik received the B.S.E.E. degree from Bradley University, Peoria, IL, in 1988 and the M.S.E.E. and Ph.D. degrees from the University of Pennsylvania, Philadelphia, in 1991 and 1992, respectively. He was a faculty member in the Department of Electrical Engineering at The Catholic University of America, Washington, DC until 2009. He is currently an Associate Professor and Director of Educational Outreach in the Department of Electrical and Computer Engineering at the University of Delaware, Newark DE. In addition to his academic positions, he is an associate editor of the Journal of Optical Engineering and also holds the position of Senior Research Engineer for the Naval Surface Warfare Center (NSWC), Carderock Division. His research interests include applied electromagnetics and photonics, computational electromagnetics, and bioelectromagnetics.