Interdisciplinary Information Sciences Vol. 15, No. 1 (2009) 67–78
©Graduate School of Information Sciences, Tohoku University
ISSN 1340-9050 print/1347-6157 online
DOI 10.4036/iis.2009.67

Performance Evaluation of Finite-Difference Time-Domain (FDTD) Computation Accelerated by FPGA-based Custom Computing Machine

Kentaro SANO, Yoshiaki HATSUDA, WANG Luzhou, and Satoru YAMAMOTO

Graduate School of Information Sciences, Tohoku University,
6-6-01 Aramaki Aza Aoba, Sendai 980-8579, Japan

Received May 31, 2008; final version accepted December 1, 2008

This paper evaluates the performance of the 2D FDTD computation on our FPGA-based array processor. So far, we have proposed the systolic computational-memory architecture for custom computing machines tailored for numerical computations with difference schemes, and implemented the array-processor based on this architecture with a single ALTERA StratixII FPGA. The array processor is composed of a two-dimensional array of programmable PEs with mesh network so that computations on a grid are performed in parallel. We wrote and executed codes for the 2D FDTD computation on the array-processor. We obtained almost the same results by FPGA as those by AMD Athlon64 processor. In comparison with AMD Athlon64 processor running at 2.4 GHz, the array-processor operating at 106 MHz achieved over 7 times faster computation for the 2D FDTD problem, which corresponds to the actual performance of 16.2 GFlop/s. The high utilization of the adders and the multipliers of the array processor means that the architecture is also suitable for the FDTD method.

KEYWORDS: FDTD method, Custom Computing Machine, FPGA

1. Introduction

It is getting more and more important and necessary to understand electromagnetic behavior in the advanced electrical engineering technologies including cellular phones, smart antennas, high-speed electronics and mobile computing. Finite Difference Time Domain (FDTD) method is a powerful and useful tool that is commonly used for solving a lot of different electromagnetic problems with good accuracy, as reported in Yee (1966) and Allen and C. (1996). The FDTD method is a numerical algorithm to simulate linear wave propagation phenomena, especially electromagnetic wave propagation based on Maxwell's Equations.

However, the software execution of the FDTD method has the problem of limited computational speed. Even though attempts have been made to exploit the parallelism of the FDTD method with parallel computers, the serial nature of software-based execution does not take advantages of the inherent parallelism. On the other hand, the custom computing machines (CCMs) implemented as application specific integrated circuits (ASICs) provide much faster computing speed by exploiting the inherent parallelism of the FDTD method. The operations that can be carried out in parallel are expanded as a data-flow graph, which is directly implemented in the circuits tailored for each individual application. However, this approach with ASICs is not feasible. Users hesitate to use ASICs for building CCMs because of their high cost of development and inflexibility.

Another approach to construct CCMs is the implementation with field-programmable gate arrays (FPGAs). FPGA is one of the programmable-logic devices where end users can configure their own logic circuits over and over on the semiconductor chip. Although FPGAs have some overhead of area and speed on a chip, the flexibility derived from hardware reconfigurability makes them feasible and useful to implement CCMs for a variety of applications. Furthermore, potential performance of floating-point computations on FPGAs is now overcoming that of general-purpose microprocessors for floating-point computations as reported in Underwood and Hemmert (2004) and Underwood (2004).

So far, a lot of researchers have studied fundamental floating-point operations and their applications on FPGAs. For example, FPGA-based floating-point co-processors are proposed for the matrix-vector multiplication and the matrix multiplication in deLorimier and DeHon (2005), Zhuo and Prasanna (2005), and Dou *et al.* (2005). Although such fundamental operations can be utilized for the FDTD method, these co-processors have an essential problem that data transfer to FPGAs becomes a bottleneck. Since the I/O bandwidth of an FPGA chip is limited, the approach building CCMs as a co-processor is essentially not scalable due to the I/O bottleneck in streaming data to FPGAs from external components.

To avoid the bottleneck of data transfer to FPGAs, the whole of necessary computations should be performed by FPGAs independently of a host processor. We proposed the systolic computational-memory architecture for CCMs that perform the difference schemes, which the FDTD method is based on, in Sano *et al.* (2007). The difference schemes provide regularity, locality and parallelism in computing on a grid. We designed CCMs based on the systolic computational-memory architecture for exploiting these properties of the difference schemes without the I/O bottleneck. The systolic architecture proposed in Kung (1982) is a regular arrangement of many simple processing-elements (PEs) in an array where data are processed and synchronously flow across the array between neighbors. In the systolic computational-memory architecture, each PE has a local memory and a programmable data-path so that the array can compute independently of a host processor. Such a structure is suitable to fully utilize the embedded block-RAMs of FPGAs and exploit their total bandwidth as reported in Sano *et al.* (2004) and Sano *et al.* (2005).

In this paper, we show that our systolic computational-memory architecture is also applicable to the FDTD method based on the difference scheme, speedup is efficiently obtained with the CCMs implemented on FPGAs. We demonstrate that the two-dimensional FDTD computation can be performed in parallel on the systolic computational-memory array, and show that the data-path of the PE is efficiently utilized for the multiply-and-accumulate computations in the FDTD method.

Related work is as follows. The first FPGA-based implementation of the FDTD method is presented in Schneider *et al.* (2002). They proposed a pipelined bit-serial cell performing integer arithmetic computations of the computational kernels of the FDTD method. They showed the 1D computation with 10 cells. In their design, boundary conditions are not considered. Because of the direct hardware implementation of computation at each grid-point, it is difficult to compute the large 2D grid without a lot of FPGAs. The 3D FDTD computation with 32 bit floating-point arithmetic is reported in Durbano and Ortiz (2004). In this design, boundary conditions and varying materials are considered. However, only poor performance was achieved because of complicated and slow floating-point units, no pipelining and slow memory interface. In Chen *et al.* (2004), better performance is obtained for pseudo-2D FDTD computation, which is applied to the detection system of buried land-mines with a ground penetrating radar. Actually, 24 times faster computation was achieved in comparison with a PC running at 3.0 GHz. In their design, fixed-point arithmetic is used instead of floating-point arithmetic, though they evaluated errors. In addition, the lack of programmability requires time-consuming hardware design and implementation for different computations, such as different boundary computations and wave sources. Some researchers made an attempt to accelerate the FDTD method with a graphics processing unit (GPU) as reported in Krakiwsky *et al.* (2004).

This paper is organized as follows. Section 2 briefly describes the basics of the FDTD method and its numerical scheme. We show that the common computation found in the scheme is formulated as a systolic algorithm, which can be performed in parallel on a systolic array. Section 3 describes the systolic computational-memory architecture and the custom computing machine based on the architecture. We mention how to parallelize the FDTD computation with PEs of a systolic array. Section 4 evaluates the performance of the 2D FDTD computation executed with the FPGA-based CCMs. We describe implementation and computational results. Finally, Section 5 gives conclusions and future work.

2. Finite Difference Time Domain Method

2.1 The Fundamentals

The finite difference time domain (FDTD) method is a powerful tool to solve electromagnetic problems as reported by Allen and C. (1996). Originally, Yee (1966) introduced the FDTD method in 1966, which provides a direct time-domain solution of Maxwell's Equations discretized by difference schemes on a uniform grid and at time intervals. Since the FDTD method is very flexible and gives accurate results for many non-specific problems, it is widely used for solving a wide variety of electromagnetic problems.

Maxwell's Equations, which are the governing equations in electromagnetism, are a set of the following four partial differential equations.

$$\nabla \times E = -\frac{\partial B}{\partial t}, \tag{1}$$

$$\nabla \times H = \frac{\partial D}{\partial t} + J, \tag{2}$$

$$\nabla \cdot D = \rho, \tag{3}$$

$$\nabla \cdot B = 0 \tag{4}$$

where E is the electric field $[V/m]$, H is the magnetic field $[A/m]$, D is the electric flux density $[C/m^2]$, B is the magnetic flux density $[T]$, ρ is the charge density $[C/m^3]$ and J is the electric current density $[A/m^2]$. These equations denote the relationship of the sources, charge density and current density in the the electric and magnetic fields.

We also have the following equations of the constitutive relations specifying the relations between B and H, D and E, and J and E, which are necessary to apply Maxwell's Equations.

$$B = \mu H, \tag{5}$$

$$D = \varepsilon E, \tag{6}$$

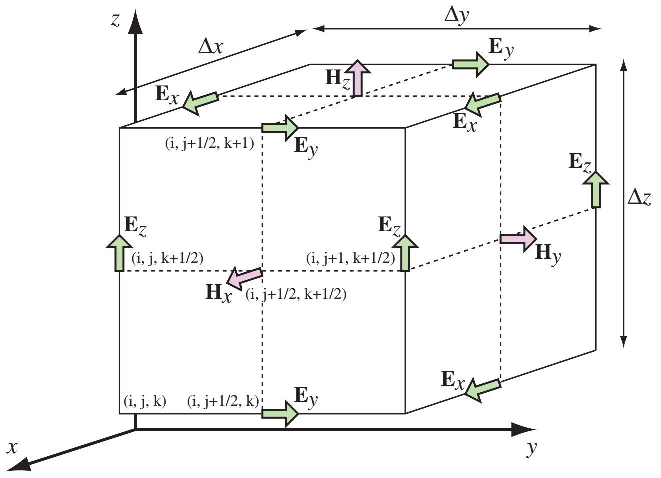


Fig. 1. 3D Yee cell.

$$J = \sigma E. \tag{7}$$

By substituting these equations to Eqs. (1) and (2), we obtain the following equations containing E and H.

$$\frac{\partial E}{\partial t} = -\frac{\sigma}{\varepsilon}E + \frac{1}{\varepsilon}\nabla \times H, \tag{8}$$

$$\frac{\partial H}{\partial t} = -\frac{1}{\mu}\nabla \times E \tag{9}$$

where μ is the magnetic permitivity $[H/m]$, ε is the electric permitivity $[F/m]$ and σ is the electrical conductivity $[C/m^3]$. Here we just show the equations only for the xy-plane. These parameters are functions of the field. Equations (8) and (9) denote the interaction between E and H.

2.2 Numerical Scheme

In the Yee's algorithm Yee (1966), Eqs. (8) and (9) are approximated by discretizing on a uniform grid at time intervals with difference schemes. Figure 1 shows the 3D grid for discretizing space, referred to as *Yee cell*. Parameters Δx, Δy and Δz define the size of the Yee cell. We use (i, j, k) to specify the point in the grid, while the real coordinate is denoted by $(i\Delta x, j\Delta y, k\Delta z)$ in the field.

On the Yee cell, discretized components of E and H are not located at the same point, but located at the interlaced positions. Every E component is surrounded by the four circulating H components, and vise versa. For example the x-component of H, $Hx_{i,j+\frac{1}{2},k+\frac{1}{2}}$, is surrounded by the four components of E: $Ez_{i,j,k+\frac{1}{2}}$, $Ey_{i,j+\frac{1}{2},k+1}$, $Ez_{i,j+1,k+\frac{1}{2}}$ and $Ey_{i,j+\frac{1}{2},k}$ as shown in Fig. 1. The E and H components are interlaced not only in space but also in time. Here Δt denotes time interval. The E components are defined at $n\Delta t$, and the H components are defined at $(n + \frac{1}{2}\Delta t)$.

The central difference approximation that is commonly used is as follows.

$$\frac{\partial E_{i,j,k}}{\partial x} = \frac{E_{i+\frac{1}{2},j,k} - E_{i-\frac{1}{2},j,k}}{\Delta x}. \tag{10}$$

With the central difference approximation and the backward difference of $\frac{\partial E}{\partial t} = \frac{E^n - E^{n-1}}{\Delta t}$ for the time integral, we obtain the following equations.

$$Ex^n_{i+\frac{1}{2},j,k} = aEx^{n-1}_{i+\frac{1}{2},j,k} + b\left(Hz^{n-\frac{1}{2}}_{i+\frac{1}{2},j+\frac{1}{2},k} - Hz^{n-\frac{1}{2}}_{i+\frac{1}{2},j-\frac{1}{2},k}\right), \tag{11}$$

$$Ey^n_{i,j+\frac{1}{2},k} = aEy^{n-1}_{i,j+\frac{1}{2},k} - c\left(Hz^{n-\frac{1}{2}}_{i+\frac{1}{2},j+\frac{1}{2},k} - Hz^{n-\frac{1}{2}}_{i-\frac{1}{2},j+\frac{1}{2},k}\right), \tag{12}$$

$$\begin{aligned} Hz^{n+\frac{1}{2}}_{i+\frac{1}{2},j+\frac{1}{2},k} &= Hz^{n-\frac{1}{2}}_{i+\frac{1}{2},j+\frac{1}{2},k} - d\left(Ey^n_{i+1,j+\frac{1}{2},k} - Ey^n_{i,j+\frac{1}{2},k}\right) \\ &\quad + e\left(Ex^n_{i+\frac{1}{2},j+1,k} - Ex^n_{i+\frac{1}{2},j,k}\right) \end{aligned} \tag{13}$$

where

$$a = \frac{1 - \dfrac{\sigma \Delta t}{2\varepsilon}}{1 + \dfrac{\sigma \Delta t}{2\varepsilon}}, \tag{14}$$

$$b = \frac{\Delta t}{\varepsilon \Delta y \left(1 + \dfrac{\sigma \Delta t}{2\varepsilon}\right)}, \tag{15}$$

$$c = \frac{\Delta t}{\varepsilon \Delta x \left(1 + \dfrac{\sigma \Delta t}{2\varepsilon}\right)}, \tag{16}$$

$$d = \frac{\Delta t}{\mu \Delta x}, \tag{17}$$

$$e = \frac{\Delta t}{\mu \Delta y}. \tag{18}$$

Based on these equations, the FDTD algorithm computes E and H along a time step. These computations are very simple because only additions, subtractions and multiplications are required.

On the boundary cells, we apply the Mur's first-order absorbing boundary condition described in Mur (1981), which is given by

$$Hz^{n+\frac{1}{2}}_{\frac{1}{2},j+\frac{1}{2},k} = Hz^{n-\frac{1}{2}}_{\frac{3}{2},j+\frac{1}{2},k} + \frac{v\Delta t - \Delta x}{v\Delta t + \Delta x}\left(Hz^{n+\frac{1}{2}}_{\frac{3}{2},j+\frac{1}{2},k} - Hz^{n-\frac{1}{2}}_{\frac{1}{2},j+\frac{1}{2},k}\right) \tag{19}$$

where v is the speed of light. Then, the entire computation of the FDTD method is described as the following procedure.

1. Initialization
2. Computation of the electric field with Eqs. (11) and (12) and so on
3. Boundary computation
4. Computation of the magnetic field with Eqs. (13) and so on
5. Termination decision. If not, go back to 2.

2.3 Properties to be Exploited for Acceleration

For simplicity, we explain with a 2D grid below. As written in Eqs. (11) to (13), the major computations of the FDTD method is formulated as the following common form.

$$q^{\text{new}}_{i,j} = c_0 + c_1 q_{i,j} + c_2 q_{i+1,j} + c_3 q_{i-1,j} + c_4 q_{i,j+1} + c_5 q_{i,j-1} \tag{20}$$

where $q_{i,j}$ is a certain value at grid-point (i, j), and c_0 to c_5 are values obtained only with values at (i, j). We refer to this computation as *accumulation*. In the case of the 3D grid, the accumulation contains at most eight terms. This fact means that all grid-points just require the accumulation computations only with data of the adjacent grid-points. The computations at all grid-points are independent, so that they can be performed in parallel.

To exploit these properties of the numerical computations with difference schemes, so far, we have proposed and designed the custom computing machines based on the systolic architecture in Sano *et al.* (2007), Sano *et al.* (2004) and Sano *et al.* (2005). Since the FDTD algorithm expressed in the form of Eq. (20) has parallelism and homogeneity with local and regular dependence at each grid-point, it is considered as a systolic algorithm, which can be efficiently performed on the systolic array in parallel. In the next section, we describe the systolic architecture for acceleration of the numerical computations based on difference schemes.

3. Custom Computing Machine Based on the Systolic Computational-Memory Architecture

3.1 Systolic Computational-Memory Architecture

As shown in Fig. 2, a systolic array is a regular arrangement of many cells comprised of simple processing elements (PEs) in an array, where data are processed and synchronously flow across the array between neighbors Kung (1982), Johnson *et al.* (1993). *Systolic algorithms* are highly parallel algorithms suitable for a direct hardware implementation on a systolic array, which specifically have the following features; simple and parallel operations with fine granularity, homogeneity in the operations, and regularity in the communication pattern. Since such an array is suitable for pipelining and spatially parallel processing with input data while they pass through the array, it gives scalable computing performance according to the array size. The systolic array also mitigates the bottleneck of data supply from external memories by enabling re-use of input data passing through the array. Moreover, the regularity of operations on

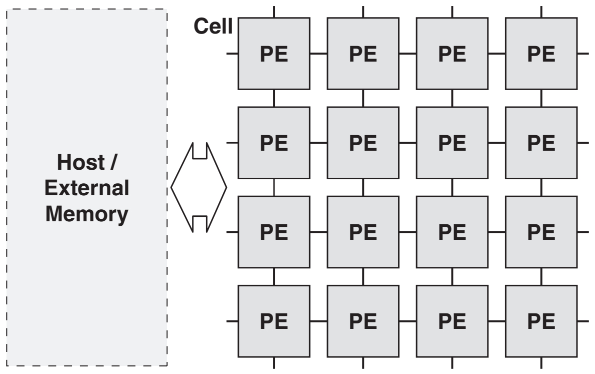


Fig. 2. Systolic Architecture.

PEs results in cost-effective design and implementation of on-chip large-scale systems.

However, the systolic array still has a bottleneck for the access of an external-memory. Let's suppose that the systolic array is implemented on a single chip. If necessary data are supplied from the off-chip memory, the external I/O bandwidth limits the computing performance of the array. It is difficult to increase pins or I/O frequency for extending the off-chip bandwidth. Accordingly, it is important for the external bandwidth requirement to be restricted for high-performance computing on a systolic array.

Computational-memory approach meets this design policy as reported in Sano *et al.* (2004) and Sano *et al.* (2005). In this approach, computing logic and memory are arranged very close to each other on a chip. We refer to the combination of the systolic architecture and the computational-memory approach as *systolic computational-memory architecture*, where each PE contains both computing logic and large local memory that stores all necessary data. Since the on-chip bandwidth is much wider than the off-chip bandwidth, this architecture allows the total memory bandwidth of PEs to be very wide and scalable to the size of the array. Consequently, the external-memory bandwidth is not a bottleneck of scaling the performance with the increased number of PEs.

In addition, a systolic architecture with this approach matches well the state-of-the-art FPGAs that have a lot of logic elements (LEs) and embedded block-RAMs inside. First, the regular arrangement of cells in an array is favorable to high LE utilization. LE utilization more than 99% has been reported in systolic array implementation on an FPGA as reported in Sano *et al.* (2005). Second, the locality of the cell control and the communication is suitable for high operation frequency on FPGAs. Third, the embedded block-RAMs distributed throughout an FPGA are available for the local memories of cells.

Since the size of the on-chip memory is limited, all the applications probably do not always fit this approach. We believe that this problem can be solved by an array of multiple FPGAs connected to each other with high-speed I/O, while an FPGA is having larger embedded memories.

3.2 Design of Custom Computing Machine

As described in Sano *et al.* (2007), we applied the systolic computational-memory architecture to designing the custom computing machine for numerical computations based on difference schemes. In this paper, we refer to this machine as *the array processor*. The array processor formed as a two-dimensional systolic array with a mesh network is shown in Fig. 2. Although this 2D array can perform computation of 2D or 3D grids, here we describe the computing model with a 2D grid for brief explanation.

Let's suppose that the 2D grid is divided into $N \times M$ partial grid-blocks, each of which is composed of $n \times m$ grid points. This means that the original 2D grid has $nN \times mM$ grid points. Each PE performs computation with the partial grid block allocated to the PE, communicating necessary data with the adjacent ones.

Although the computation based on the difference scheme has the common form of Eq. (20), each step actually performs different computations. Moreover, different simulation problems require different computations for their own boundary conditions. Therefore, we designed a PE that has a data-path optimized for Eq. (20) with programmability, instead of a completely fixed data-path.

Figure 3 is the overview of a PE. The PE has a computational-memory component including a programmable data-path and a local memory. The data-path is designed to be suitable for summing-up computation of Eq. (20). The local memory stores all the necessary data for the partial grid-block allocated to the PE. The PE also has *the north (N-)*, *south (S-)*, *west (W-)* and *east (E-) First-In First-Out queues (FIFOs)* that are each connected to the four adjacent PEs. These connections give a mesh-network topology of the PEs in an array, which is also adequate for acquiring data for the summing-up computation of Eq. (20). The FIFOs allow PEs to avoid too rigorous requirement for synchronization in sending and receiving data to/from the adjacent PEs.

Figure 4 shows the data-path of the computational memory component, which has *a sequencer*, *a local memory* and *a MAC (Multiplication and Accumulation) unit*. The data-path is pipelined with eight stages; *MS (Microoperation*

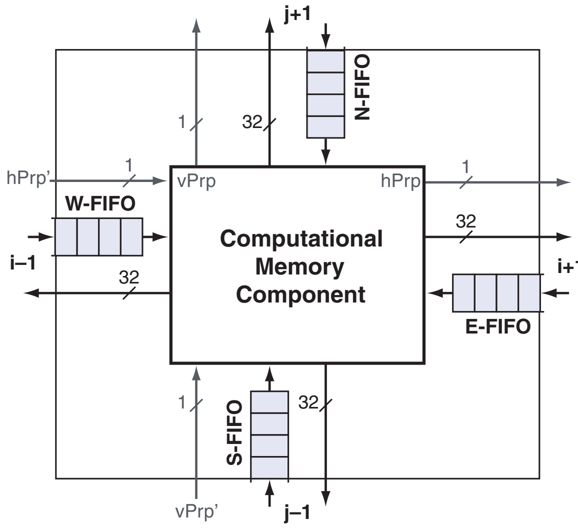


Fig. 3. Structure overview of the processing element.

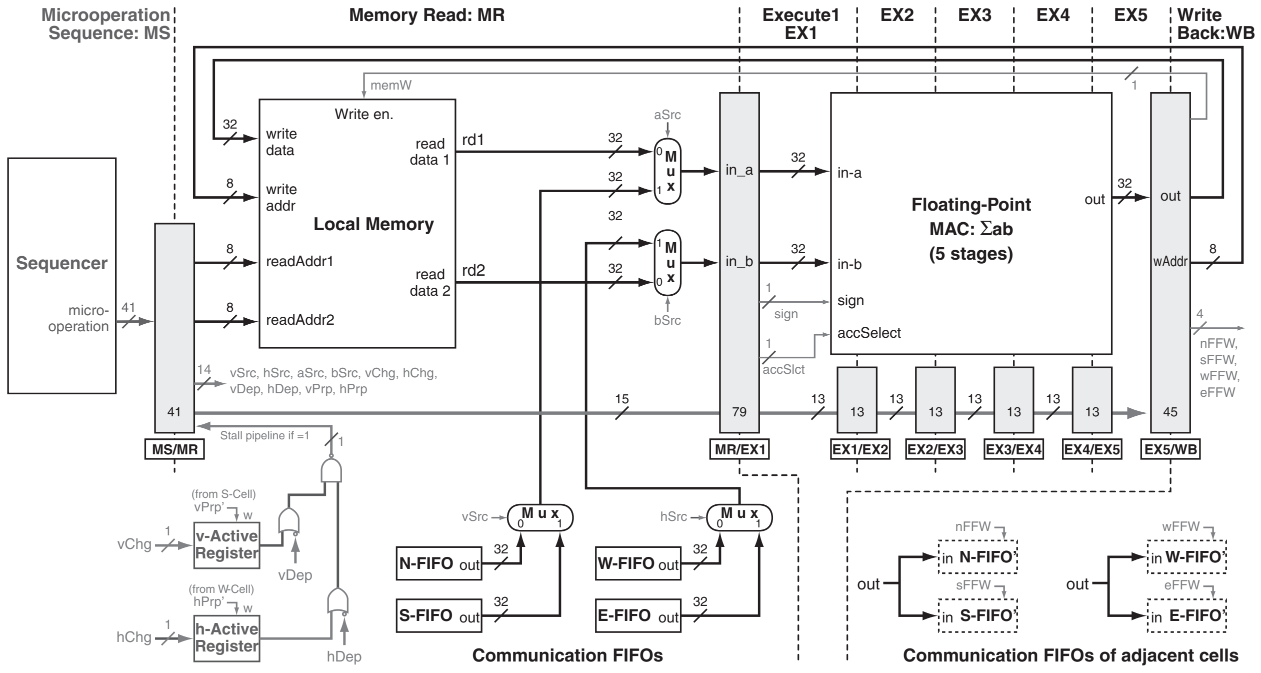


Fig. 4. Data-path of the computational memory component of PE.

Sequence) stage, *MR (Memory Read) stage*, *five EX (Execution) stages* and *WB (Write Back) stage*. The data-path does not have an instruction decoder for simple implementation. The sequencer stores control signals, i.e., microprograms, and provides them to the remaining part of the data-path. Each PE can have its own sequencer, however, we made a choice of shared sequencers for SIMD (Single Instruction stream, Multiple Data streams) control. The detail of shared sequencers is described in Section 4. The local memory stores single precision floating-point data, and temporal or intermediate results of computations. It has two read-ports and one write-port. Two data are read and input to the MAC unit, and the output of MAC unit is written to the local memory.

In the EX stages, the MAC unit performs multiplication and accumulation in order to efficiently sum up the terms of Eq. (20). The MAC unit computes $a \times b$ with the two inputs of a and b, and then adds or subtracts ab with the output of the MAC unit. Figure 5 shows the structure of the MAC unit. The MAC unit has the five pipelined stages for a single precision floating-point multiplier and an adder. Integrating the multiplier and the adder in one unit allows the MAC unit to have relatively smaller circuit and faster operation frequency by partially simplifying the rounding and normalization circuits.

The MAC unit has the forwarding path from the 5th stage to the 2nd stage for accumulation. The sign bit selects addition or subtraction in the 4th stage of the MAC unit. According to the accSlct bit, the forwarding is activated or not. Suppose that a_i and b_i are input at cycle i. a_1 and b_1 are input at cycle 1. Since their multiplication result a_1b_1 reaches

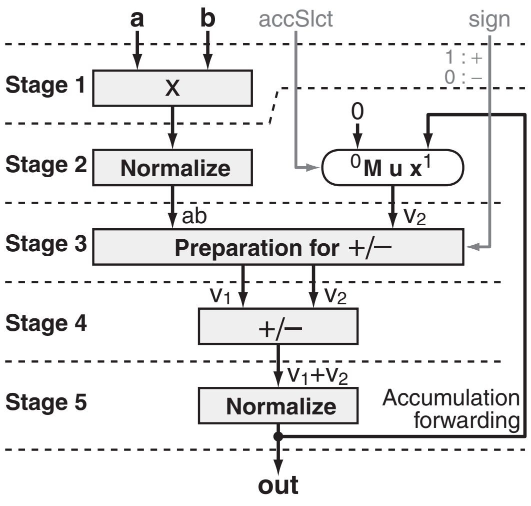


Fig. 5. MAC unit pipelined with 5 stages.

Table 1. Instruction set of a processing element.

	op-code	dst1,	dst2,	src1,	src2	description
1	mulp	—,	L1,	L2,	SFIFO	MACout = M[L2] × SFIFO, M[L1] := MACout
2	mulm	SN,	—,	L2,	L3	MACout = −M[L2] × M[L3], {S,N}FIFOs := MACout,
3	accp	—,	L1,	L2,	L3	MACout = MACout + M[L2] × M[L3] M[L1] := MACout,
4	nop					No operation
5	halt					Halt the array processor.
6	lset	Num,	Addr			Loop-counter := Num. Bne-reg$_i$:= Addr (for i-th nested loop).
7	bne					Branch if loop-counter not eq. to zero.
8	accpbne	—,	L1,	L2,	L3	accp instruction & bne

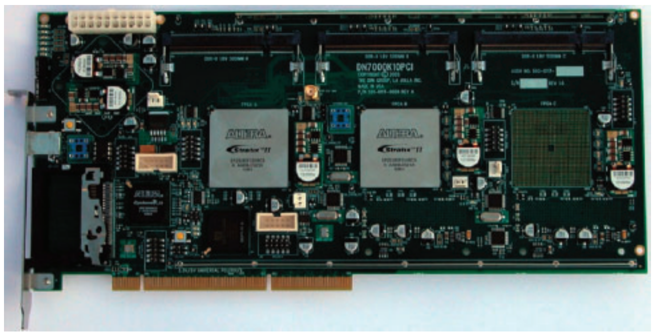


Fig. 6. DiNI group DN7000k10PCI with two Stratix II FPGAs (EP2S180).

the 5th stage at cycle 5 and forwarded to the 2nd stage, a_4b_4 can be added to a_1b_1 at cycle 6. Thus this unit accumulates the products of inputs fed every three cycles. This means that three sets of Eq. (20) have to be concurrently performed in order to fully utilize the multiplier and the adder of the MAC unit. In the WB stage, the output of the MAC unit is written into the local memory, or the FIFOs of the adjacent cells.

Table 1 shows the instruction set of the PE. The instruction set is composed of *computing instructions* and *controlling instructions*. There is no comparison instruction and no conditional branch. The computing instruction takes an operation code (op-code), two destinations (dst), and two sources (src). The op-codes of *mulp*, *mulm* and *accp* are *multiply and add with zero*, *multiply and subtract with zero* and *multiply and accumulate with the previous output of the MAC unit*, respectively. Since the forwarding path connects the 2nd and 5th stages of the MAC unit for accumulation, the accp instruction sums up the result of multiplication with the value accumulated 3 cycles before. The first destination, dst1, specifies FIFOs which the computing result is sent to. S, N, E and W of dst1 corresponds to S-FIFO,

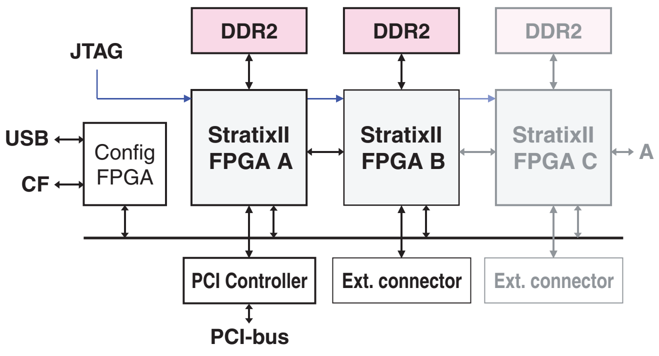


Fig. 7. Block diagram of DN7000k10PCI.

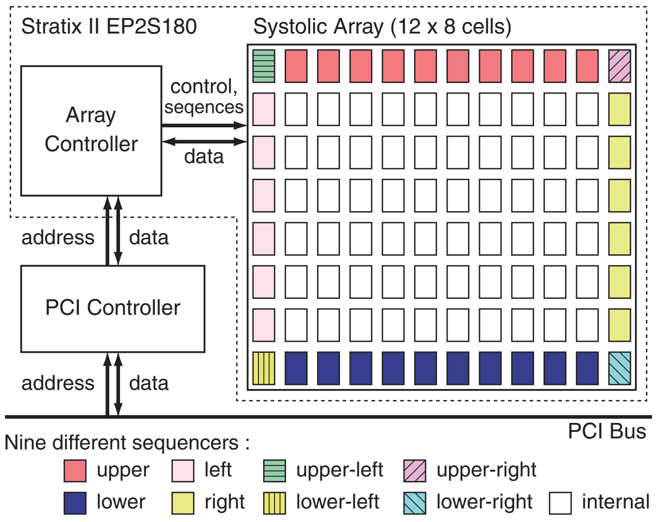


Fig. 8. Implemented System with a single Stratix II FPGA.

N-FIFO, E-FIFO and W-FIFO, respectively. The second destination, dst2, specifies the address of the local memory where the computing result is written. The first and second sources, src1 and src2, specify the addresses of the local memory or FIFOs which values are read from to the MAC unit.

As the controlling instructions, we have *nop*, *halt*, *lset* and *bne* instructions. The lset and bne instructions are dedicated to the nested loop control, which is necessary in numerical simulations. The lset instruction is used to set a loop-counter and a jump-register in the sequencer with Num and Addr, respectively. Then, when bne is executed, the program counter is set to be the address stored in the jump-register if the loop-counter is not zero. Simultaneously, the loop-counter is decremented. The accpbne instruction executes the accp instruction and the bne instruction at the same clock cycle.

4. Performance Evaluation

4.1 Implementation

We implemented the array processor with a PCI prototyping board, DN7000k10PCI shown in Fig. 6, which has two ALTERA Stratix II FPGAs (EP2S180-5) and a PCI controller as depicted in Fig. 7. The StratixII FPGA totally has 143520 adaptive look-up tables (ALUTs), which can emulate up to more than 1.2 million ASIC gates. The FPGA also contains totally 96 embedded 36-bit multipliers and three types of configurable SRAMs; 512-bit M512 blocks, 4-Kbit M4K blocks and 512-Kbit M-RAM blocks.

Figure 8 shows the overview of the system implemented as a PCI target device. For prototyping, the systolic array and its controller were implemented by using only FPGA-A. The systolic array has *the idle mode* and *the computing mode*. In the idle mode, all the local memories and the sequence memories of cells are arranged in a single memory space, which is accessed by the PCI controller. In this mode, these memories are initialized, and the computational results in the local memories are read by a host computer. In the computing mode, the array performs computation based on sequences in the sequence memories.

The implemented array-processor is composed of 96(12×8) PEs, which consumed 74880 ALUTs (52.1%), 96 embedded 36-bit multipliers (100%), 384 ($= 4 \times 96$) M4K blocks (50%), 768 ($= 8 \times 96$) M512 blocks (82.6%). The

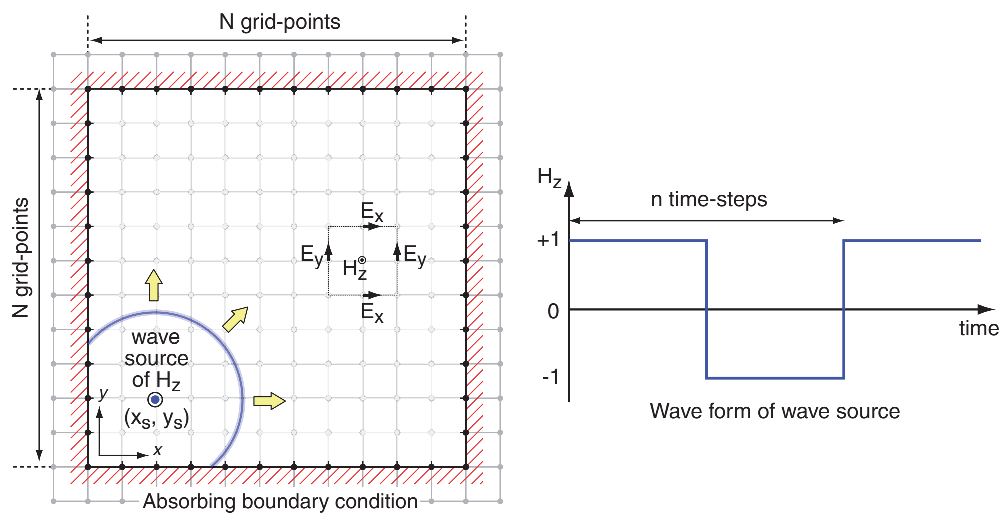


Fig. 9. Configuration of 2D FDTD computation.

size of each local memory is 1 KBytes where 256 32-bit floating-point numbers can be stored. The 96 cells share 9 sequencers that were implemented by using 16992 ALUTs (11.8%) and 9 M-RAM blocks (100%). The 9 sequencers control 9 groups of PEs as shown in Fig. 8 for different boundary-computations. The size of each sequence memory is 64 KBytes and 8192 sequences can be stored. The four FIFOs of a cell each have 32 entries. Since the array-processor operates at 106 MHz, each PE has the peak performance of $106 \times 10^6 \times 2 = 0.212$ GFlop/s. Accordingly, the entire array achieves the peak performance of $0.212 \times 96 = 20.4$ GFlop/s.

4.2 FDTD Computation

We carried out two-dimensional FDTD computation with the condition of Fig. 9. The computational grid consists of $N \times N$ grid-points. There is a square wave source of Hz at (x_s, y_s) close to the lower left corner. We computed two different sizes of a grid, $N = 48$ with $(x_s, y_s) = (3, 5)$ and $N = 72$ with $(x_s, y_s) = (5, 8)$, where Δx and Δy correspond to 5.0×10^{-3} m. The amplitude of the wave source is 1, and its period is 80 time-steps. Δt corresponds to $\frac{1.0}{80.0 \times 2.45 \times 10^9} \simeq 5.102 \times 10^{-12}$ sec. We used the parameters of $\varepsilon = 8.854 \times 10^{-12}$, $\sigma = 0.0$ and $\mu = 4\pi \times 10^{-7}$. On the border, Mur's first-order absorbing boundary condition is applied.

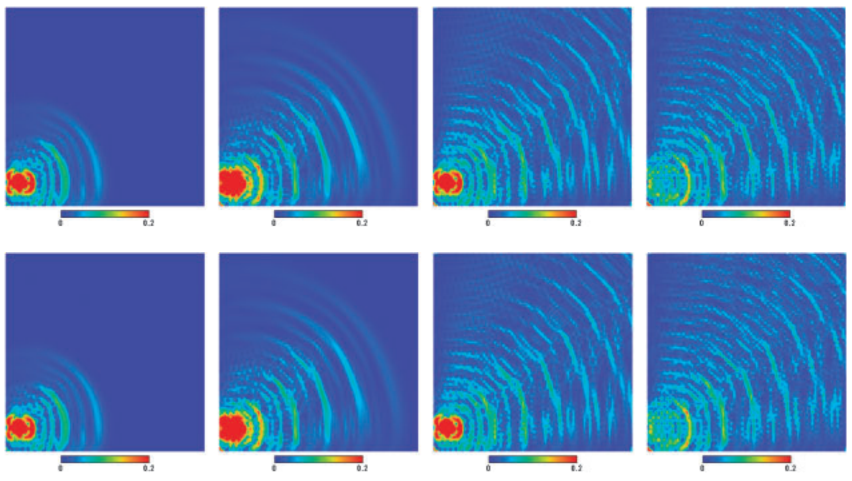
For comparison, we wrote a program of the same computation in C to be executed on a Linux PC with AMD Athlon64 4000+ running at 2.4 GHz. All the computations are performed in single precision. This program was compiled by using gcc with option of "-O3."

For FPGA-based computation, we wrote sequences for the computation of the electric field, the boundary computation and the computation of the magnetic field with the two nested loops. Since the array processor has 12×8 PEs, each PE takes charge of a partial grid-block containing 6×9 points for a 72×72 grid. As shown in Fig. 8, the nine sequencers for the nine sorts of partial grid blocks containing *the upper boundary*, *the lower boundary*, *the left boundary*, *the right boundary*, *the upper-left corner*, *the upper-right corner*, *the lower-left corner*, *the lower-right corner* and *the internal grid points*, respectively, because they need different sequences for different boundary conditions.

4.3 Results and Discussions

Figure 10 are the visualized results of the 2D FDTD computation with the 72×72 grid by the FPGA-based array-processor and AMD Athlon64 4000+ running at 2.4 GHz. The value of $|\mathbf{E}| = \sqrt{E_x^2 + E_y^2}$ is shown in color. As seen in the figure, these results by FPGA and CPU are almost the same. The root mean square error (RMSE) of $|\mathbf{E}|$ is only 1.563×10^{-4} at time-step of 1600, which is 5.04×10^{-1}% of the mean value of $|\mathbf{E}|$. For the 48×48 grid, RMS of $|\mathbf{E}|$ is 1.263×10^{-7}, which is 3.27×10^{-4}% of the mean absolute value of $|\mathbf{E}|$. Although these errors are very small, they are not zero. The reason why the error is not zero is that the implemented adder and multiplier do not support the denormalized numbers and all the rounding modes of the IEEE754 format. However, such errors can be given by different CPUs, e.g., Intel Pentium4 and AMD Athlon64, and therefore they are practically not significant.

For the array-processor, the computing sequences of PEs are scheduled so that the MAC unit is almost fully utilized. In the case of the sequence for the internal PEs, the computations takes 594 cycles in each time step. Therefore, the FDTD computation for 1600 time-steps on the 72×72 grid totally takes $594 \times 1600 + 103$ cycles including initialization sequences. The average utilizations of the adder and the multiplier in the MAC unit are 70.4 and 88.2%, respectively, for all the PEs. Therefore, the average utilization of the MAC unit is $(70.4 + 88.2\%)/2 = 79.3\%$, which gives the actual performance of $20.4 \times 79.3\% = 16.2$ GFlop/s.



a. Time-step=100. b. Time-step=200. c. Time-step=500. d. Time-step=1600.

Fig. 10. Computational results of 2D FDTD computation. Absolute values of $|\mathbf{E}| = \sqrt{E_x^2 + E_y^2}$ are visualized in color. Upper: FPGA, Lower: CPU (AMD Athlon64 4000+).

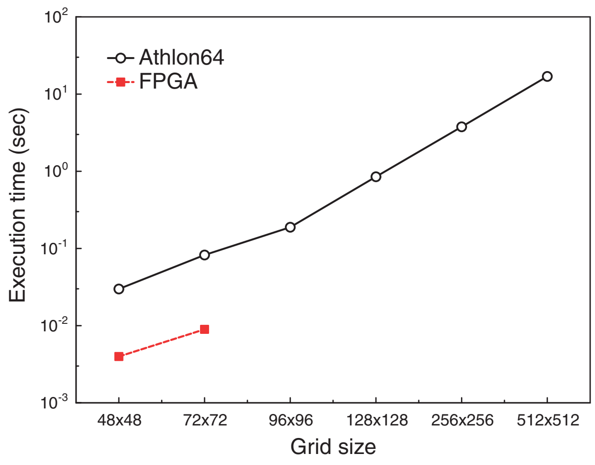


Fig. 11. Execution time vs. grid size.

We measured the execution time on AMD Athlon64 for comparing computational-speed with FPGA. We used *gettimeofday() system call* for measurement, and ten measured data are averaged to have precision of 1 μsec. For the computation of 1600 time-steps on the 72 × 72 grid, the computational time on AMD Athlon64 is 8.236×10^{-2} sec. On the other hand, the array-processor operating at 106 MHz takes 8.967×10^{-3} sec for 950503 cycles spent for the entire computation of 1600 time-steps. Thus, the FPGA-based array-processor achieved 9.19 times faster computations than AMD Athlon64 running at 2.4 GHz in spite of low operation frequency, 106 MHz, of FPGA. For the computation of 1600 time-steps on the 48 × 48 grid, AMD Athlon64 and the array-processor take 2.989×10^{-2} and 3.986×10^{-3} sec, respectively, resulting in 7.50 times faster computation on FPGA.

Figure 11 shows the execution time for the different sizes of a grid. Here, the size of the grid computed on the array processor is limited to 72 × 72 due to the limited size of local memories of the current implementation. In the case of Athlon64, the time increases with a certain incline until the grid size becomes 96^2. However, for the grid sizes larger than 96^2, the incline gets higher than that for the grid sizes less than 96^2. We consider that this is due to the L2 cache effect. AMD Athlon64 4000+ has an L2 cache of 512 KBytes. Since the grid size of 128^2 requires 192 KBytes for *Ex*, *Ey* and *Hz*, the L2-cache miss increases around 128^2. On the other hand, the execution time of the array processor is not influenced by L2 cache misses, and therefore the time is almost linear to the number of grid-points. Consequently, the speedup of the array processor gets relatively higher in comparison with microprocessors.

Figure 12 shows the execution time for the different time-steps. The FPGA-based array processor provides linear increase to the number of time steps. On the other hand, Athlon64 takes relatively shorter time per grid-point as the number of time steps increases. We consider that this is due to the effect of compulsory miss in the cache. However, Athlon64 is also having almost the same incline as that of FPGA for larger time steps, the array processor keeps the speedup.

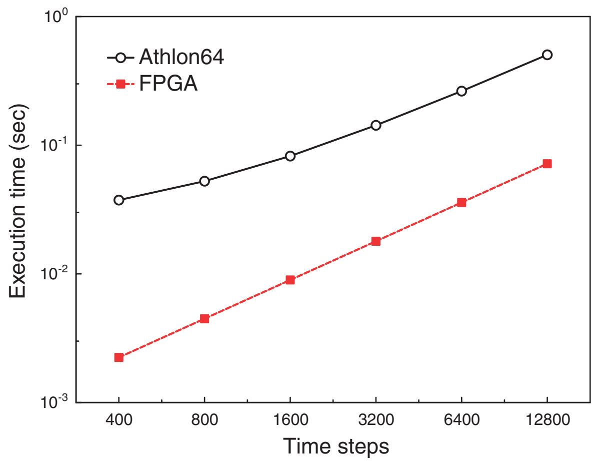


Fig. 12. Execution time vs. time steps.

5. Conclusions

In this paper, we have evaluated the performance of the 2D FDTD computation on the FPGA-based array processor. The array processor is a custom computing machine for numerical computations with difference schemes, which is designed based on the systolic computational-memory architecture. The array processor is composed of a two-dimensional array of PEs with mesh network. Each PE is programmable and performs the MAC (multiply and accumulate) operations with its local memory. We implemented the array processor with 12×8 PEs on a single ALTERA Stratix II FPGA, which operates at 106 MHz. The peak performance is 20.4 GFlop/s for floating-point computation of single precision.

We computed the 2D FDTD computation for electromagnetic wave propagation with 48×48 and 72×72 grids. The FPGA-based array processor achieved over 7 times faster computation than AMD Athlon64 4000+ running at 2.4 GHz for 72×72 grids. This speedup is due to the large number of PEs and the utilization of the MAC units. The utilization of the MAC units is 79%, resulting in the actual performance of 16.2 GFlop/s. This means that the data-path tailored for difference schemes is very effective and efficient for the FDTD method.

We still have the problems of the memory size and the transcendental function for a wave source. First, the total size of local memories is a problem because the leading edge FPGA have the limited size of embedded RAMs, e.g., 9 Mbits. Although it is increasing as the FPGA technology is advanced, we need techniques to scale the size of memory. For scalable memory-size, we are planning to introduce *external-memory support*, *connection of multiple FPGAs* and *compression of floating-point data*. If we have the sufficient size of local memories, we can perform larger and more complex 3-dimensional problem on an FPGA-based array processor. By assining tall blocks like square pillars to PEs, we can parallelize the computations on a 3-dimensional grid with the 2-dimensional array structure. In order to compute with complex geometries, we can introduce generalized curvilinear coordinates into the orthogonal grid, which require more parameters per grid point.

Second, the current array processor cannot efficiently compute the transcendental function, e.g. $\sin(\theta)$, that is commonly used for a wave source. We are introducing the following approaches. One is a look-up table to store the sampled values of transcendental functions. If we have enough RAMs, this is feasible. Another approach is to introduce the custom unit to compute the transcendental function. Although such a custom unit consumes a lot of logic elements, typically we need only one unit for a PE computing a wave source. The cost to have only one unit is trivial.

Our final goal is to implement a large-scale systolic array with a lot of FPGAs connected to each other via high-speed differential I/O links. Now we have just implemented the I/O unit and demonstrated that sufficient bandwidth is obtained between two FPGAs.

REFERENCES

[1] Allen, T., and H. S. C., Computational electrodynamics—The finite difference time-domain method: Norwood, MA: Aretch House Inc. (1996).

[2] Chen, W., Kosmas, P., Leeser, M., and Rappaport, C., An FPGA implementation of the two-dimensional finite-difference time-domain (FDTD) algorithm, *Proceedings of the 2004 ACM/SIGDA 12th international symposium on Field programmable gate arrays (FPGA2004)*, 213–222 (2004).

[3] deLorimier, M., and DeHon, A., Floating-point sparse matrix-vector multiply for FPGAs, *Proceedings of the International Symposium on Field-Programmable Gate Arrays*, 75–85 (2005).

[4] Dou, Y., Vassiliadis, S., Kuzmanov, G. K., and Gaydadjiev, G. N., 64-bit floating-point fpga matrix multiplication, *Proceedings of the International Symposium on Field-Programmable Gate Arrays*, 86–95 (2005).

[5] Durbano, J. P., and Ortiz, F. E., FPGA-based acceleration of the 3d finite-difference time-domain method, *Proceedings of the*

12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM2004), 156–163 (2004).

[6] Johnson, K. T., Hurson, A., and Shirazi, B., "General-purpose systolic arrays," *Computer*, **26**: 20–31 (1993).

[7] Krakiwsky, S. E., Turner, L. E., and Okoniewski, M. M., Acceleration of finite-difference time-domain (FDTD) using graphics processor units (GPU), *Proceedings of the IEEE MTT-S International Microwave Symposium Digest*, **2**: 1033–1036 (2004).

[8] Kung, H. T., "Why systolic architecture?," *Computer*, **15**: 37–46 (1982).

[9] Mur, G., "Absorbing boundary conditions for the finite-difference approximation of the time-domain electromagnetic field equations," *IEEE Transactions on Electromagnetic Compatibility*, **4**: 377–382 (1981).

[10] Sano, K., Iizuka, T., and Yamamoto, S., Systolic architecture for computational fluid dynamics on fpgas, *Proceedings of the 15th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM2007)*, 107–116 (2007).

[11] Sano, K., Takagi, C., Egawa, R., Suzuki, K., and Nakamura, T., A systolic memory architecture for fast codebook design based on MMPDCL algorithm, *Proceedings of the International Conference on Information Technology (ITCC2004)*, 572–578 (2004).

[12] Sano, K., Takagi, C., and Nakamura, T., Systolic computational memory approach to high-speed codebook design, *Proceedings of the 5th IEEE International Symposium on Signal Processing and Information Technology (ISSPIT2005)*, 334–339 (2005).

[13] Schneider, R. N., Turner, L. E., and Okoniewski, M. M., Application of fpga technology to accelerate the finite-difference time-domain (FDTD) method, *Proceedings of the 2004 ACM/SIGDA 12th international symposium on Field programmable gate arrays (FPGA2002)*, 97–105 (2002).

[14] Underwood, K., FPGA vs. CPUs: Trends in peak floating-point performance, *Proceedings of the International Symposium on Field-Programmable Gate Arrays*, 171–180 (2004).

[15] Underwood, K. D., and Hemmert, K. S., Closing the gap: CPU and FPGA trends in sustainable floating-point blas performance, *Proceedings of the 12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, 219–228 (2004).

[16] Yee, K. S., Numerical solution of inital boundary value problems involving maxwell's equations in isotropic media, *IEEE Transactions on Antennas and Propagation*, **14**: 302–307 (1966).

[17] Zhuo, L., and Prasanna, V. K., Sparse matrix-vector multiplication on fpgas, *Proceedings of the International Symposium on Field-Programmable Gate Arrays*, 63–74 (2005).