http://dx.doi.org/10.5392/JKCA.2015.15.09.021

반도체 검증을 위한 MPI 기반 클러스터에서의 대용량 FDTD 시뮬레이션 연산환경 구축

Implementation of Massive FDTD Simulation Computing Model Based on MPI Cluster for Semi-conductor Process

이승일, 김연일, 이상길, 이철훈
충남대학교 컴퓨터공학과

Seung-Il Lee(silee@cnu.ac.kr), Yeon-Il Kim(kyi3426@cnu.ac.kr),
Sang-Gil Lee(sk0137@cnu.ac.kr), Cheol-Hoon Lee(clee@cnu.ac.kr)

요약

반도체 공정에서는 소자 내부의 물리량 계산을 통해 불순물의 움직임을 해석하여 결점을 검출하는 시뮬레이션을 수행하게 된다. 이를 위해 유한 차분 시간 영역 알고리즘(Finite-Difference Time-Domain, 이하 FDTD)과 같은 수치해석 기법이 사용된다. 반도체 칩의 집적도 향상으로 인하여 소자의 크기는 나노스케일 시대로 접어들었으며, 시뮬레이션 사이즈 또한 커지고 있는 추세이다. 이에 따라 CPU와 GPU 같은 하나의 연산 장치에서 수행할 수 없는 문제와 다중의 연산 장치로 구성된 한 대의 컴퓨터에서 수행할 수 없는 문제가 발생하기도 한다. 이러한 문제로 인해 분산 병렬처리를 통한 FDTD 알고리즘 연구가 진행되고 있다. 하지만 기존의 연구들은 단일 연산장치만을 이용하기 때문에 GPU를 사용하는 경우 연산 속도는 빠르나 메모리의 제한이 있으며 CPU의 경우 GPU에 비해 연산 속도가 느린 단점이 존재한다. 이를 해결하기 위해 본 논문에서는 CPU, GPU의 이기종 연산 장치를 포함하는 컴퓨터로 구축된 클러스터 상에서 작업 사이즈에 제한되지 않고 시뮬레이션 수행이 가능한 컴퓨팅 모델을 구현하였다. 점대점 통신 기반의 MPI 라이브러리를 이용하여 연산 장치 간 통신을 통한 시뮬레이션을 테스트 하였고 사용하는 연산 장치의 종류와 수에 상관없이 시뮬레이션이 정상 동작함을 확인하였다.

■ 중심어 : | 병렬컴퓨팅 | 하이브리드컴퓨팅 | MPI | CUDA |

Abstract

In the semi-conductor process, a simulation process is performed to detect defects by analyzing the behavior of the impurity through the physical quantity calculation of the inner element. In order to perform the simulation, Finite-Difference Time-Domain(FDTD) algorithm is used. The improvement of semiconductor which is composed of nanoscale elements, the size of simulation is getting bigger. Problems that a processor such as CPU or GPU cannot perform the simulation due to the massive size of matrix or a computer consist of multiple processors cannot handle a massive FDTD may come up. For those problems, studies are performed with parallel/distributed computing. However, in the past, only single type of processor was used. In GPU's case, it performs fast, but at the same time, it has limited memory. On the other hand, in CPU, it performs slower than that of GPU. To solve the problem, we implemented a computing model that can handle any FDTD simulation regardless of size on the cluster which consist of heterogeneous processors. We tested the simulation on processors using MPI libraries which is based on 'point to point' communication and verified that it operates correctly regardless of the number of node and type. Also, we analyzed the performance by measuring the total execution time and specific time for the simulation on each test.

■ keyword : | Parallel Computing | Hybrid Computing | MPI | CUDA |

* 이 연구는 2014 충남대학교 학술연구비에 의해 지원되었음
접수일자 : 2015년 03월 10일
심사완료일 : 2015년 08월 05일
수정일자 : 2015년 07월 31일
교신저자 : 이철훈, e-mail : clee@cnu.ac.kr

I. 서 론

최근 발달하는 컴퓨터 성능에 맞추어 어플리케이션에서 처리해야 하는 데이터의 연산량이 점차적으로 증가하는 추세이다. 이에 따른 요구에 맞추어 높은 성능을 가지는 단일 컴퓨터 한 대를 이용한 순차 프로그래밍에서 네트워크를 통해 연결되어있는 다수의 컴퓨터를 이용하거나 혹은 단일 컴퓨터 내의 다수의 코어를 이용하는 병렬 프로그래밍이 발전하고 있다. 초창기 병렬 프로그래밍은 CPU에서 다수의 코어를 사용하여 연산을 수행하였으나, IT 시장에서 하드웨어 기술이 지속적으로 발전함에 따라 그 구성요소들의 크기가 소형화 되었다. GPU 또한 그래픽 처리를 위해 사용되는 트랜지스터를 충분하게 가짐으로써, 다른 활용 방안에 대해 논의 되었다.

이에 Scalar 연산을 수행하는 방안으로 활용하기 위해 GPGPU(General Purpose GPU)라는 모델이 개발되었다. GPGPU는 현재 Scalar 연산 병렬 프로그래밍 모델에서 가장 좋은 성능을 보이고 있다고 평가 받고 있다.

GPGPU를 이용한 성능개선의 대상 중 하나로, 공정 시뮬레이터로는 반도체 소자 내부의 물리량을 계산하고 있기 때문에 반도체 소자 내부의 불순물의 거동을 해석하는 것이 가능하다. 이러한 모의실험은 3차원적 형상을 표현하는 물리적 미분방정식을 계산하여 그 결과를 구하게 되는데, 복잡한 구조의 비선형 미분 방정식을 정확하게 계산하기 위해 FDTD와 같은 수치해석 기법에 적용 가능하다.

시간이 흐름에 따라 반도체의 집적도가 향상되고 있는 추세이며 이에 수행하고자 하는 FDTD의 크기도 증가하고 있다. 따라서 1대의 연산장치의 메모리를 FDTD의 크기가 초과하는 문제가 발생한다. 이를 만족하기 위해서는 클러스터 내 다수의 연산장치를 이용하여 병렬적으로 수행하는 FDTD 연산환경 모델 구축이 필요하다. 또한 클러스터 내 사용가능한 GPU로 수행할 수 없는 용량의 FDTD 이거나, 해당 FDTD의 우선순위가 다른 FDTD에 비해 상대적으로 낮아 빠른 수행시간이 필요하지 않은 경우 CPU를 포함하여 연산할 수 있는 이기종 연산 장치 간 FDTD 연산 환경 모델 구축이 추가적으로 필요하다.

본 논문은 2장에 관련 연구로서 FDTD, CUDA, MPI에 대해 소개하고 3장에서는 MPI를 이용하여 대용량 FDTD를 처리하기 위한 알고리즘의 구현을, 4장에서는 테스트 환경과 결과를 기술하며 마지막으로 5장에서는 결론 및 향후 연구과제를 기술한다.

II. 관련연구

1. CUDA

CUDA(Compute Unified Device Architecture)는 GPU에서 수행하는 병렬 처리 알고리즘을 C 프로그래밍 언어를 비롯한 프로그래밍 언어를 사용하여 작성할 수 있도록 하는 GPGPU 기술이다. CUDA는 C 프로그래밍 언어를 기본으로 별도의 라이브러리를 추가하여 사용하는 방식이다. 그래픽스 파이프라인의 별도의 이해가 필요 없이 C/C++ 언어를 사용하여 프로그램을 개발할 수 있는 환경을 제공하기 때문에 프로그래머가 보다 손쉽게 병렬처리 프로그램을 개발할 수 있다. CUDA는 2006년 10월 NVIDIA가 웹에 처음 공개하였으며, 이후 지속적으로 발전하고 있으며 슈퍼 컴퓨터 제작과 고성능 연산처리에 많이 사용되고 있다. 특히 데이터 사용에 초점을 둔 어플리케이션을 CUDA를 통하여 병렬화 한 뒤, GPU 내부의 코어 프로세서를 사용하여 고속의 부동소수점 연산을 할 수 있다. 이러한 CUDA 아키텍처를 사용하려면 NVIDIA GPU와 특별한 스트림 처리 드라이버가 필요시 된다[1][2].

2. MPI

MPI(Message Passing Interface)는 각자의 메모리를 지역적으로 따로 가지는 프로세스들로 구성된 분산 시스템 환경에서 프로세스들 사이의 통신을 오직 메시지들의 송신(sending) 과 수신(receiving)으로만 구현하는 프로그래밍 모델을 말한다[2][3].

MPI는 집합 연산, 사용자 정의 데이터 타입과 토폴로지, 다양한 방식의 통신 등의 기능들과 이기종 병렬 아키텍처(Heterogeneous Parallel Architecture)에 대한

지원 등을 제공하며, 코드가 실행되는 동안 프로세스의 수를 변화시키는 동적 프로세스 관리, 원격 메모리 접근 병렬 I/O 등의 지원을 목표로 한다[3].

MPI를 이용한 FDTD 병렬처리의 경우 [그림 1]과 같은 환경에서 FDTD를 사용자가 분할하고자 원하는 축을 기준으로 분할하여 각 연산 장치가 할당받은 만큼의 FDTD를 수행하고 각 스텝 종료 시 자계 및 전계 영역에 대한 결과값 송·수신 과정을 추가하여 하나의 FDTD와 동일한 연산을 수행하는 구현이 가능하다[4].

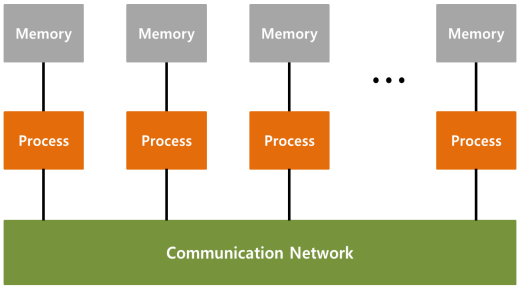

그림 1. MPI를 사용하기 위한 연산 환경

3. FDTD

FDTD는 널리 사용되고 있는 전자계 해석기법으로서 반도체 공정에서 불순물을 해석하는데 사용된다.

계산영역 내부의 모든 점에서 전계(E)와 자계(H)가 결정되며 각 셀에서의 일반적으로 매질은 유전율, 투자율, 전도율 등이 정의되는 공기(free space), 금속, 유전체 등이 매질이 사용될 수 있다.

계산영역과 격자, 매질이 결정되면 입력 소스가 정의된다. 소스는 평면파 입사, 도선상의 전류, 금속면 사이의 전압으로 모델의 상황에 맞게 결정된다. 계산과정에서 전계와 자계가 직접 계산되기 때문에 시뮬레이션의 출력은 일반적으로 각 점이나 계산영역 내부의 점들의 행렬에 대한 전계와 자계의 값이다.

FDTD는 동일한 동작을 반복적으로 수행하기 때문에 SIMD(Single Instruction Multiple Data) 구조를 가진다고 볼 수 있다[5][6].

3.1 사용 연산 장치에 따른 FDTD

3.1.1 다수의 CPU를 이용한 FDTD

클러스터 구축을 통하여 단일의 FDTD에 대하여 다수의 CPU를 사용한 연산이 가능하다. 각 연산 장치의 수에 맞게 매트릭스를 분할하고 이를 할당한다. MPI를 통한 통신을 이용하여 결과 값 송·수신 한다. 단일의 CPU를 이용하는 것보다는 성능상의 이점을 얻을 수 있으나 GPU를 사용하는 것에 비하여 상대적으로 느린 연산 시간을 가진다는 단점이 있다[7].

3.1.2 다수의 GPU를 이용한 FDTD

GPU를 이용한 FDTD의 경우 CPU를 이용한 경우와 크게 다르지 않다. NVIDIA에서 GPGPU 컴퓨팅 연산을 수행하기 위한 Architecture인 CUDA를 이용하여 병렬 연산을 수행하여 높은 성능을 기대할 수 있다. 하지만 CPU에 상대적으로 적은 메모리를 가지고 있어 대용량 사이즈의 FDTD를 수행하기 위해서는 클러스터 내 많은 수의 GPU를 유지하여야 한다. [8, 9] GPUDirect가 적용되어있는 GPU를 사용하는 경우에는 GPU 간 통신시 GPU 메모리에서 바로 결과 값 송·수신이 가능하여 HostPC의 메인 메모리 접근에 소요되는 시간을 단축시킬 수 있다[7].

3.1.3 CPU와 GPU를 모두 이용한 FDTD

CPU와 GPU를 이용하는 FDTD의 경우 각 연산 장치 간의 데이터 교환을 위한 추가적인 절차가 필요하다. [그림 2]은 PC와 그래픽 카드 사이의 메모리 참조 방식을 설명한 그림이다[10].

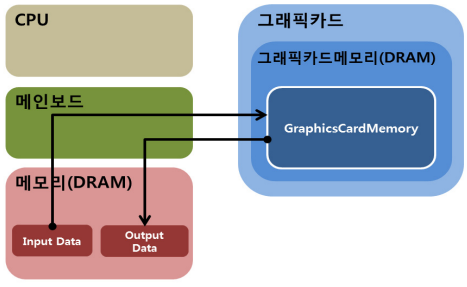

그림 2. Host PC와 그래픽 카드 사이의 메모리 참조 방식

CPU의 경우 FDTD 할당 및 처리 데이터를 이용하는

메모리가 DRAM을 이용하지만 GPU는 장치 내에 존재하는 그래픽스 카드 메모리를 사용하게 된다. 따라서 CPU와 GPU를 동시에 사용하는 경우, 각 연산장치의 결과 값에 대한 송·수신 과정에서 cudaMemcpy() 함수를 이용하여 전달하고자 하는 연산장치의 메모리로 옮기는 과정이 필요하다[11].

Ⅲ. 이기종 연산 장치를 이용한 FDTD 구현

1. 연산 장치 수에 따른 FDTD 분할 방법

1대의 연산 장치를 사용하는 경우 기존의 코드를 이용하였으며, 2대 이상의 연산 장치를 이용하는 코드를 구현하였다. FDTD의 분할을 사용자가 각 연산장치에 할당하고자 하는 Z축의 크기만큼을 분할한다. [그림 3]는 기존의 큰 사이즈의 매트릭스를 2대의 연산장치에 분할하는 과정을 설명한 그림이다. 분할된 두 개의 매트릭스는 각각의 연산 장치에서 수행되어지고 연산에 대한 결과 값을 자계영역의 경우 위쪽 매트릭스를 담당하는 연산장치에게, 전계의 경우 아래쪽 매트릭스를 담당하는 연산 장치에게 송신하게 된다[9].

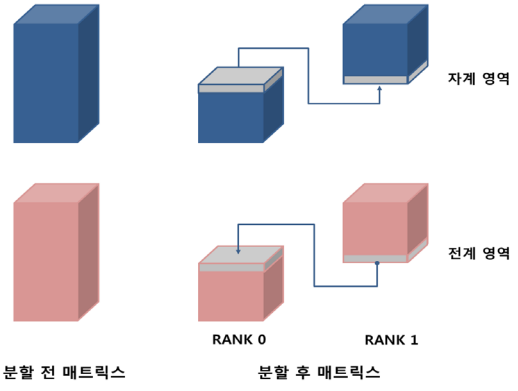

그림 3. FDTD를 위한 매트릭스 분할 전 · 후에 따른 결과값 전송 과정

2. 구현 알고리즘

시뮬레이션은 visual studio 2010을 사용하여 구현된다. 시뮬레이션을 시작하면 인자로 전달 받은 (스텝 * 사이클) 수만큼 동일한 코드를 반복하여 수행 한다. FDTD 시뮬레이션 수행 명령과 함께 프로그램이 시작되어 사용자가 설정한 연산장치가 동시에 사용되어지게 되고 각 연산 장치간의 통신을 위한 MPI초기화를 MPI_Init() 함수를 이용하여 하게 된다. 이 함수에 대한 반환 값을 RANK 식별자로 사용한다. 이 후 각자의 분할된 매트릭스를 메모리에 할당하고 시뮬레이션에 필요한 변수들을 저장하는 환경 설정을 수행한다. 그 후 사이클 수만큼 FDTD() 함수를 반복하여 수행하고 매 수행이 종료 될 때마다 종료 조건을 판별하여 종료 조건일 경우 MPI_Finalize() 함수를 이용하여 시뮬레이션에 사용되어진 모든 연산 장치에게 종료를 알린 후 프로그램이 종료한다.

아래의 [그림 4]는 3대의 연산 장치를 이용하여 실질적으로 FDTD를 수행하는 FDTD()에서 한 스텝에 수행되어지는 과정을 나타낸 그림이다.

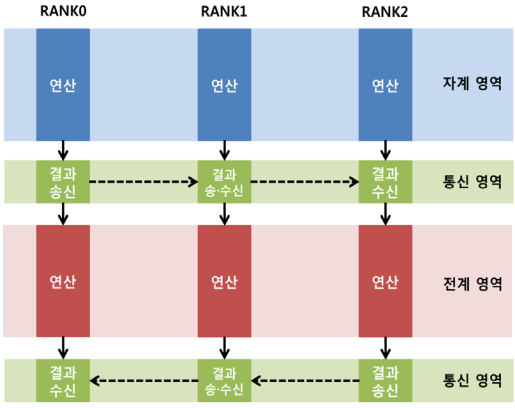

그림 4. 3대의 연산 장치를 이용한 FDTD() 함수

FDTD() 함수가 실행되면 사용자가 지정한 스텝수만큼 반복문을 수행하게 된다. 각 스텝 별로 해당 영역의 연산 및 결과 값을 송·수신 하며 이에 따른 오류 검출을 수행하게 되고 사이클의 경우 2대 이상의 연산 장치를 사용하는 경우 해당 매트릭스 자계 및 전계 영역에 대한 연산에 대한 결과 값을 전송하고 인접한 매트릭스로부터 결과 값을 수신하는 과정이 추가된다. MPI 초기화 과정에서 호출한 MPI_Init() 함수 수행에 대한 반환 값으로 연산 장치에 배정 된 고유 식별 값(RANK) 값이 작을수록 매트릭스의 아랫면을 담당하여 연산하

게 되어 0의 값을 가지는 RANK의 경우 최하면을 담당하고 시뮬레이션에 참여하는 연산 장치의 수를 가지는 전역 변수인 mpi_size와 같은 값을 가지는 RANK의 경우 최상면을 담당하게 된다.

각 연산 장치 마다 할당된 전계 및 자계 영역에 대하여 연산을 수행하게 되고 필요시 MPI_Isend()와 MPI_Irecv()를 이용하여 결과 값을 주고받게 된다. 비동기 통신을 지원하는 위 두 함수를 사용하여 결과 값의 전송이 완료될 때까지 대기하지 않고 다음 영역의 수행이 가능케 하였다. 각 연산 장치가 담당하는 매트릭스에 따라 최하면 매트릭스, 최상면 매트릭스, 중간 매트릭스의 총 세가의 수행 방법으로 나뉜다.

가장 아랫면을 담당하는 RANK가 수행하는 과정에 대한 의사코드는 [그림 5]과 같다.

```
if( node_type == CPU ){
    CPU 코드를 이용한 자계 영역 수행;
    자계 영역 PML 수행;
    다음 RANK(RANK+1)로 자계 영역 연산 결과 송신;
    CPU 코드를 이용한 전계 영역 수행;
    전계 영역 PML 수행;
    다음 RANK(RANK+1)의 전계 영역 연산 결과 수신;
}
if( node_type == GPU ){
    GPU 코드를 이용한 자계 영역 수행;
    자계 영역 PML 수행;
    다음 RANK(RANK+1)로 자계 영역 연산 결과 송신;
    GPU 코드를 이용한 전계 영역 수행;
    전계 영역 PML 수행;
    다음 RANK(RANK+1)의 전계 영역 연산 결과 수신;
}
```

그림 5. 가장 아랫면을 담당하는 RANK의 수행에 대한 의사코드

가장 아랫면을 담당하는 RANK의 경우 해당 매트릭스의 다음 사이클 연산을 위하여 자계 연산에 대한 결과를 위의 매트릭스로 송신하여야 하며 위의 매트릭스의 전계 연산에 대한 결과를 수신해야 한다. 추가적으로 자계영역과 전계영역 연산 수행 후 매트릭스의 아랫부분을 감싸고 있는 자유공간인 PML에 대한 연산을 수행한다.

가장 윗면을 담당하는 RANK가 수행하는 과정에 대한 의사코드는 [그림 6]과 같다.

```
if( node_type == CPU ){
    CPU 코드를 이용한 자계 영역 수행;
    자계 영역 PML 수행;
    이전 RANK(RANK-1)의 자계 영역 연산 결과 수신;
    CPU 코드를 이용한 전계 영역 수행;
    전계 영역 PML 수행;
    이전 RANK(RANK-1)의 전계 영역 연산 결과 송신;

    }
    if( node_type == GPU ){
    GPU 코드를 이용한 자계 영역 수행;
    자계 영역 PML 수행;
    이전 RANK(RANK-1)의 자계 영역 연산 결과 수신;
    GPU 코드를 이용한 전계 영역 수행;
    전계 영역 PML 수행;
    이전 RANK(RANK-1)의 전계 영역 연산 결과 송신;
    }
```

그림 6. 가장 윗면을 담당하는 RANK의 수행에 대한 의사코드

가장 윗면을 담당하는 RANK는 자계 연산에 대한 아래의 매트릭스의 결과를 수신하고 해당 매트릭스에 대한 전계 연산 결과를 아래의 매트릭스로 송신해야 한다. 추가적으로 가장 아랫면을 담당하는 RANK와 유사하게 자계영역과 전계영역 연산 수행 후 매트릭스의 윗부분을 감싸고 있는 자유공간인 PML에 대한 연산을 수행한다.

가운데에 위치한 RANK가 수행하는 과정에 대한 의사코드는 [그림 7]과 같다.

```
if( node_type == CPU ){
        CPU 코드를 이용한 자계 영역 수행;
        이전 RANK(RANK-1)의 자계 영역 연산 결과 수신;
        다음 RANK(RANK+1)로 자계 영역 연산 결과 송신;
        CPU 코드를 이용한 전계 영역 수행;
        다음 RANK(RANK+1)의 전계 영역 연산 결과 수신;
        이전 RANK(RANK-1)의 전계 영역 연산 결과 송신;
    }
    if( node_type == GPU ){
        GPU 코드를 이용한 자계 영역 수행;
        이전 RANK(RANK-1)의 자계 영역 연산 결과 수신;
        다음 RANK(RANK+1)로 자계 영역 연산 결과 송신;
        GPU 코드를 이용한 전계 영역 수행;
        다음 RANK(RANK+1)의 전계 영역 연산 결과 수신;
        이전 RANK(RANK-1)의 전계 영역 연산 결과 송신;
    }
```

그림 7. 가운데 면을 담당하는 RANK의 수행에 대한 의사코드

가운데에 위치한 RANK의 경우에는 각각의 전계 및 자계에 대하여 결과를 아래의 매트릭스로 부터 수신후 담당 매트릭스 연산에 대한 결과를 위의 매트릭스로 송신하는 구조를 가지게 되어 결과 값 송·수신을 위한 한 단계가 추가적으로 발생하게 된다.

연산 장치가 할당 받은 매트릭스에 대한 자·전계 영역 연산과 결과 값에 대한 송·수신을 각 스텝마다 반복

하게 되고 총 스텝 수만큼 수행 후 RANK가 0을 가지는 연산 장치를 Master로 하여 Intensity() 함수를 수행하게 한다. 그 후 MPI_Bcast() 함수를 이용하여 Master 연산 장치가 나머지 모든 Slave 연산 장치에게 FDTD() 함수의 정상적 종료를 MPI_Bcast() 함수를 통하여 알리게 되고, 시뮬레이션에 참여하는 모든 연산 장치가 FDTD() 함수를 동시에 종료한다.

Ⅳ. 실험환경 및 결과

1. 실험환경 및 실험방법

실험을 위하여 구축한 이기종 연산 장치로 구성된 클러스터 상에서 FDTD가 정상적으로 작동함을 검증하고 수행시간 분석을 위한 실험환경은 [그림 8]과 같으며 [표 1]과 같은 성능을 가지는 컴퓨터 세대를 연결하여 클러스터를 구축하였다. 데이터 교환을 위해서 패스트 이더넷을 이용하여 통신하였으며 GPUDirect를 지원하지 않는 환경에서 테스트를 진행하였다.

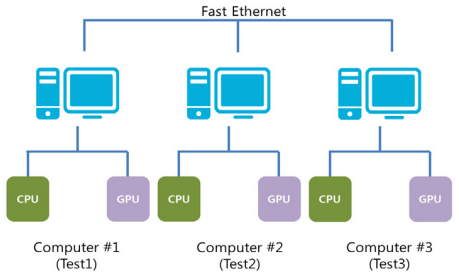

그림 8. MPI기반 대용량 FDTD 시뮬레이션 동작 검증을 위한 실험환경

표 1. 클러스터를 구성하는 컴퓨터 환경구성

Host Name	연산 장치 종류	연산 장치명	동작환경
Test1	CPU	Intel Core i5-2500 CPU 3.30GHz	windows 7
	GPU	NVIDIA GeForce GTX 550 Ti	
Test2	CPU	Pentium Dual-Core CPU 3.06GHZ	
	GPU	NVIDIA GeForce GTX 550 Ti	
Test3	CPU	Intel Core i5-2500 CPU 3.30GHz	
	GPU	NVIDIA GeForce GTX 550 Ti	

테스트에 사용되어진 시뮬레이션은 200*200*200, 200*200*400, 200*200*600의 크기를 가지며 각 각 약 750MB, 1GB, 1.5GB의 사이즈를 가진다. 이는 약 600MB의 할당 가능한 메모리를 가지는 테스트 GPU 연산 장치 1대에서 수행 가능한 사이즈보다 크므로 다른 추가적인 연산 장치의 도움을 필요로 하는 크기이다. 해당 사이즈의 시뮬레이션을 이용하여 사용 연산 장치와 수를 다르게 하여 테스트를 수행하였으며 사용자가 지정한 RANK 0에 해당하는 Master 연산장치가 사이클 수행 후 출력하는 Intensity가 동일한 값을 출력하는지 비교하였다. [표 2], [표 3]은 FDTD에 사용되는 할당 가능한 연산장치에 대한 시뮬레이션 테스트 케이스이다.

표 2. 200*200*200, 200*200*400의 시뮬레이션 테스트 케이스

테스트 번호	CPU 연산 장치 수	GPU 연산 장치 수
1	2	0
2	0	2
3	1	1
4	1	2
5	2	1
6	3	0
7	0	3

표 3. 200*200*600의 시뮬레이션 테스트 케이스

테스트 번호	CPU 연산 장치 수	GPU 연산 장치 수
1	1	2
2	2	1
3	3	0
4	0	3

2. 실험 결과

2.1 테스트 유효성 검증

사용 연산 장치의 종류와 수를 다르게 하여 수행한 테스트의 유효성 검증은 사용 연산 장치를 다르게 하여 각 사이클 후 출력되는 Intensity값을 비교한다. Intensity는 매트릭스에 입사광에 의한 반사광 에너지의 x축, y축, z축의 제곱의 합을 나타낸 값으로 빛의 에너지량을 나타낸다. 이는 입사광의 세기, 매질의 구조와 크기, 불순물의 유무 등의 요소로 변화하게 된다. 시뮬

레이션 테스트의 결과는 [표 4], [표 5] [표 6]과 같다.

표 4. 200*200*200의 시뮬레이션 테스트 케이스

사이클	테스트 번호						
	1	2	3	4	5	6	7
1	3.00	3.00	3.00	3.00	3.00	3.00	3.00
2	2.92	2.92	2.92	2.92	2.92	2.92	2.92
3	2.44	2.44	2.44	2.44	2.44	2.44	2.44
4	1.71	1.71	1.71	1.71	1.71	1.71	1.71
5	1.04	1.04	1.04	1.04	1.04	1.04	1.04
6	5.57	5.57	5.57	5.57	5.57	5.57	5.57
7	2.54	2.54	2.54	2.54	2.54	2.54	2.54

표 5. 200*200*400의 시뮬레이션 테스트 케이스

사이클	테스트 번호						
	1	2	3	4	5	6	7
1	3.00	3.00	3.00	3.00	3.00	3.00	3.00
2	3.00	3.00	3.00	3.00	3.00	3.00	3.00
3	2.94	2.94	2.94	2.94	2.94	2.94	2.94
4	2.59	2.59	2.59	2.59	2.59	2.59	2.59
5	1.71	1.71	1.71	1.71	1.71	1.71	1.71
6	8.25	8.25	8.25	8.25	8.25	8.25	8.25
7	2.91	2.91	2.91	2.91	2.91	2.91	2.91

표 6. 200*200*600의 시뮬레이션 테스트 케이스

사이클	테스트 번호			
	1	2	3	4
1	3.00	3.00	3.00	3.00
2	3.00	3.00	3.00	3.00
3	2.98	2.98	2.98	2.98
4	2.71	2.71	2.71	2.71
5	2.06	2.06	2.06	2.06
6	1.33	1.33	1.33	1.33
7	7.31	7.31	7.31	7.31

사이클이 진행됨에 따라 이전 사이클의 입사광의 영향, 매질의 구조에 따라 Intensity 값의 차이는 존재하지만 사용 연산 장치에 따른 값은 동일한 결과인 것을 확인할 수 있다. 이를 통하여 반도체 집적도 향상으로 인하여 발생되는 대용량 FDTD 시뮬레이션 수행이 가능하다.

V. 결론 및 향후 연구과제

본 논문에서는 이기종 연산 장치로 구성된 클러스터 상에서 대용량의 FDTD 수행 기능을 제공하기 위하여 수행하고자 하는 매트릭스를 분할 후 각 연산 장치 간 MPI 라이브러리를 통한 통신을 가능하게 하였다. 또한, 분할된 매트릭스를 담당하는 연산 장치의 위치에 따른 FDTD 처리 방식을 다르게 함으로써, 대용량 FDTD를 한 대의 연산 장치를 사용하여 수행한 결과와 다수의 연산 장치를 사용하여 수행한 결과가 같은 값을 출력하는 것을 확인할 수 있었다.

성능을 검증하기 위해 테스트에 사용된 GPU 한 대에서 수행할 수 없는 사이즈의 시뮬레이션을 3대의 CPU와 3대의 GPU로 구성된 클러스터에서 연산 장치의 종류와 수를 다르게 하여 수행하였으며, 각 사이클 수행 후 출력되는 Intensity를 비교를 통해 연산 장치의 종류와 수에 상관없이 동일한 결과를 출력함을 확인 하였다.

향후 연구 과제로는 본 논문의 높은 신뢰성을 검증하기 위해 실제 반도체 테스트 공정에서 사용되는 클러스터 상에서의 테스트가 필요하고, 통신 영역 수행에 소요되는 시간을 줄이는 방안 연구가 필요하다. 또한 사용되는 연산 장치의 성능을 고려한 Z축 분할 방안에 대한 연구가 필요하다.

참 고 문 헌

[1] 데이비드 B. 커크, 원메이 W. 후, *대규모 병렬 프로세서 프로그래밍 CUDA를 이용한 실용적 접근*, BJ퍼블릭, 2010.

[2] NVIDIA CUDA Programming Guide V2.0, http://kr.nvidia.com/object/cuda_develop_kr.html, accessed on 13 April 2009.

[3] 이홍석, 김정한, 이승우, 이식, *MPI 병렬 프로그래밍 : 멀티코어 시대에 꼭 알아야할*, 어드북스, 2010.

[4] 고영관, *고속 연산처리를 위한 CUDA 기반의 병렬 처리 소프트웨어 설계 및 구현*, 충남대학교 학위논문, 2013.

[5] 박종현, 이정수, 김준성 “클러스터 환경의 병렬처리를 통한 FDTD 알고리즘의 성능 향상 분석”,

대한전자공학회 추계학술대회, 제32권, 제2호.
[6] David M. Sheen, Sami M. A, and Mohamed D. A., "Application of the three-dimensional Finite-Difference Time-Domainn method to the analysis of planar microstrip circuit," IEEE Transaction on Microwaqve Theory and Techniques, Vol.38, pp.849-857, 1990.
[7] https://developer.nvidia.com/gpudirect
[8] 정복재, "CUDA로 구현한 FDTD 알고리즘의 OpenMP기술 적용 및 성능 측정", 한국컴퓨터정보학회 통계학술대회 논문집, 제21권, 제1호.
[9] 조용희, "OpenMP, MPI, CUDA를 이용한 안테나 수치 계산 가속화", 한국콘텐츠학회 종합학술대회 논문집, 2014.
[10] Cameron Hughes and Tracey Hughes, *C++ 병렬·분산 프로그래밍*, 정보문화사.
[11] 정영훈, *CUDA 병렬 프로그래밍*, 프리렉, 2011.

저 자 소 개

이 승 일(Seung-Il Lee) 준회원

- 2013년 2월 : 충남대학교 컴퓨터공학과(공학사)
- 2013년 9월 ~ 현재 : 충남대학교 컴퓨터공학과 석사과정 재학

<관심분야> : 실시간 운영체제, 임베디드 시스템

김 연 일(Yeon-Il Kim) 준회원

- 2014년 2월 : 충남대학교 컴퓨터공학과(공학사)
- 2014년 2월 ~ 현재 : 충남대학교 컴퓨터공학과 석사과정 재학

<관심분야> : 실시간 운영체제, 임베디드 시스템

이 상 길(Sang-Gil Lee) 준회원

- 2014년 2월 : 충남대학교 컴퓨터공학과(공학사)
- 2014년 2월 ~ 현재 : 충남대학교 컴퓨터공학과 석사과정 재학

<관심분야> : 실시간 운영체제, 임베디드 시스템

이 철 훈(Cheol-Hoon Lee) 정회원

- 1983년 2월 : 서울대학교 전자공학과(공학사)
- 1988년 2월 : 한국과학기술원 전기 및 전자공학과(공학석사)
- 1992년 2월 : 한국과학기술원 전기 및 전자공학과(공학박사)
- 1983년 3월 ~ 1986년 2월 : 삼성전자 컴퓨터 사업부 연구원
- 1992년 3월 ~ 1994년 2월 : 삼성전자 컴퓨터 사업부 선임연구원
- 1994년 2월~1995년 2월 : Univ. of Michigan 객원 연구원
- 1995년 5월 ~ 현재 : 충남대학교 컴퓨터공학과 교수
- 2004년 2월 ~ 2005년 2월 : Univ. of Michigan 초빙 연구원

<관심분야> : 실시간시스템, 운영체제, 고장허용 컴퓨팅, 로봇 미들웨어