논문 2010-47CI-4-10

CUDA를 이용한 FDTD 알고리즘의 병렬처리

(Parallel Computation of FDTD algorithm using CUDA)

이 호 영*, 박 종 현*, 김 준 성**

(HoYoung Lee, JongHyun Park, and JunSeong Kim)

요 약

CPU를 능가하는 GPU의 연산능력 향상으로 범용 계산에 그래픽 프로세서를 사용하는 GP-GPU연구가 활발히 전개되고 있으며, 그 응용분야가 확대되고 있다. 본 논문에서는 전자기학 관련 분야에서 널리 사용되는 FDTD 알고리즘을 nVIDIA에서 제공하는 소프트웨어 플랫폼인 CUDA를 사용하여 구현한다. FDTD 알고리즘의 주요 연산과정을 병렬화하고, 그래픽 카드 내 각기 다른 메모리의 사용에 따라 최적화하며, 단일 프로세서에서 FDTD 알고리즘을 실행시킨 경우와 비교하여 그 성능 향상 정도를 측정한다. 실험결과 단일 프로세서로 구현하였을 때에 비해 실행시간이 45배까지 향상됨을 확인할 수 있었다.

Abstract

Modern GPUs(Graphic Processing Units) provide computing capability higher than that of the general CPUs(Central Processor Units). With supports of programmability of graphics pipeline GP-GPU(General Purpose computation on GPU) has gained much attention expanding its application area. This paper compares sequential and massively parallel implementations of FDTD(Finite Difference Time Domain) algorithm using CUDA(Compute Unified Device Architecture). Experimental results show upto 45X speedup over conventional CPU execution.

Keywords: FDTD 알고리즘, GP-GPU, CUDA, 병렬처리, 성능측정

I. 서 론

자연과학과 공학분야를 포함한 학문 전 분야의 연구 과정에서 복잡한 연산알고리즘의 활용과 대용량의 데이터 처리가 필요한 경우가 자주 발생한다. 연산 처리과정에서 기존의 범용 프로세서를 사용하는 경우 낮은 IPC (Instruction Per Clock)와 과도한 전력소모로 성능 향상의 한계를 갖게 된다.[1] 이러한 한계를 극복하기 위해 다수의 프로세서를 가진 시스템에서 하나의 프로그램을 태스크 단위로 분리하여 처리하는 병렬처리 방식이 사용되나, 하드웨어 구현의 복잡성과 많은 비용으로 인해 쉽고 간편하게 사용하기 어려운 점이 있다.[1~2] 이에 대한 대안으로 이미지 프로세싱 분야에서 렌더링과 같은 응용프로그램에 특별한 목적으로 사용되던 GPU(Graphics Processing Unit)를 범용 계산에 활용하는 GP-GPU (General Purpose computing on Graphics Processing Units)가 제시되었다. 오늘날의 GPU는 CPU를 뛰어넘는 연산능력을 갖고 있으므로 GPU를 단순히 영상처리 분야에서의 사용 뿐 만이 아닌 일반 연산 처리에도 이용하여 기존에 CPU가 처리하던 응용프로그램의 연산을 GPU에서 나누어 처리하자는 것이다. 특히 nVIDIA사에서 C 또는 C++와 같은 고급언어의 추가지원이 가능한 소프트웨어 플랫폼인 CUDA (Compute Unified Device Architecture)의 발표로 개발자가 보다 손쉽게 GPU의 연산능력을 이용할 수 있게 되었으며, 현재 다양한 응용 분야의 알고리즘을 개발하는 연구가 활발히 진행되고 있다.[3~4, 6, 8]

본 논문에서는 전자기학 분야에서 많이 사용되는

* 학생회원, ** 정회원, 중앙대학교 전자전기공학부
(School of Electrical and Electronics Engineering, Chung-Ang University).
접수일자: 2010년4월30일, 수정완료일: 2010년7월7일

FDTD(Finite-Difference Time-Domain) 알고리즘을 CUDA를 이용하여 구현하고 최적화를 통하여 성능 향상을 극대화시킨다. 실험을 통해 CPU를 이용하는 경우와 비교하여 GPU를 이용한 병렬처리를 통해 얻게 되는 성능 향상에 대해 분석한다. 또한 GPU에서 지원하는 다양한 메모리 중 글로벌메모리와 공유메모리의 사용에 따른 성능변화를 확인하였다.

본 논문의 구성은 다음과 같다. Ⅱ장에서는 CUDA 라이브러리와 실험에서 사용된 응용 프로그램인 FDTD 알고리즘에 대해 알아본다. Ⅲ장에서는 CUDA를 사용하여 FDTD 알고리즘을 구현하고 최적화한다. Ⅳ장에서는 실험 환경 소개와 구현된 FDTD 알고리즘의 실행시간 측정을 통한 성능비교와 그 결과를 분석한다. 마지막으로 결론 및 본 논문의 전반적인 요약을 Ⅴ장에 기술한다.

Ⅱ. 배경 이론

1. GP-GPU와 CUDA 라이브러리

GPU는 컴퓨터에서 그래픽 처리를 위한 특정 목적을 갖는 프로세서를 말한다. GPU가 등장하기 이전까지는 모든 그래픽 처리 및 계산 작업을 CPU가 담당함으로써 프로세스를 처리함에 있어 병목현상이 자주 발생하였지만 GPU의 등장으로 인하여 영상처리 및 화면 출력과 같은 CPU의 그래픽 작업 부하가 줄어들게 되었다. GPU는 그래픽을 사용하는 응용 분야의 확대와 그에 따른 하드웨어의 급격한 진화로 인해 CPU를 뛰어넘는 연산능력을 갖게 되었다. 이는 결과적으로 GPU의 높은 연산능력을 범용 연산처리에 사용하려는 노력으로 이어졌고, CPU가 전통적으로 취급했던 응용프로그램들의 연산까지도 GPU를 이용하여 처리할 수 있게 되었다.[4~5] 현재까지 디지털 영상처리 분야에서 두각을 나타내고 있는 GP-GPU는 향후 여러 분야로 확대될 전망이다. 그림 1. 은 GPU의 특징을 CPU와 비교하여 그 구조상의 차이를 개념적으로 보여준다. GPU는 제어 기능보다 연산 기능이 강조된 구조로서, CPU에 비해 coltrol과 cache에 관한 기능이 적고, ALU(Arithmetic Logic Unit)의 수가 현저하게 많다. 이는 동일한 연산에 대한 다량의 데이터를 처리하는 SIMD(Single Instruction Multiple Data) 구조에 적합하며 GPU를 이용하여 병렬 프로그래밍을 할 수 있는 환경을 제공한다.[5~6]

하지만 GPU의 뛰어난 연산능력에도 불구하고 GPU

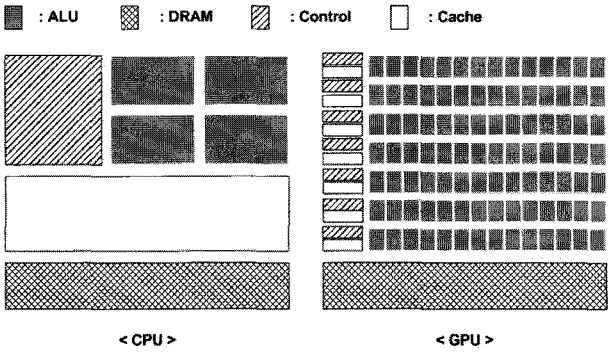


그림 1. CPU와 GPU의 구조적 차이
Fig. 1. Architectural differences between CPU and GPU.

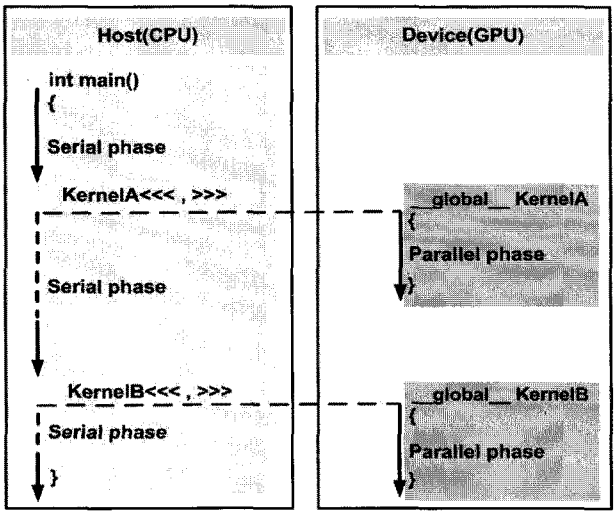


그림 2. CUDA를 사용한 프로그램의 수행
Fig. 2. Programming code procedures in CUDA.

프로그래밍이 그래픽 라이브러리를 이용한 범위 내에서만 가능했기 때문에 그 활용은 제한적이었다. 따라서 그래픽 하드웨어가 단순한 가속기의 개념을 뛰어넘어 직접적인 프로그래밍이 가능하도록 개발자의 입장에서 사용하기 편리하고 디버깅이 쉬운 개발언어의 필요성이 제기되었다. 이에 nVIDIA사는 2007년, C 또는 C++와 같은 고급언어의 추가지원이 가능한 CUDA를 공식적으로 발표했다.[7~9]

CUDA는 GPU를 사용하여 대량의 데이터를 병렬로 처리하기 위한 소프트웨어 플랫폼으로, C 프로그래밍 언어를 기본으로 별도의 라이브러리를 추가하여 사용한다. 그림 2. 에서 볼 수 있는 것과 같이 프로그램이 실행되면 우선 호스트(Host) 컴퓨터에서 CPU를 사용한 일반적인 직렬 코드를 수행한다. 진행 중에 커널을 호출하면 GPU를 사용하는 병렬코드가 실행된다. 이때 호스트 컴퓨터의 직렬코드는 커널을 통해 실행되는 병렬코드와의 동기화 여부에 따라 진행방법이 달라진다.

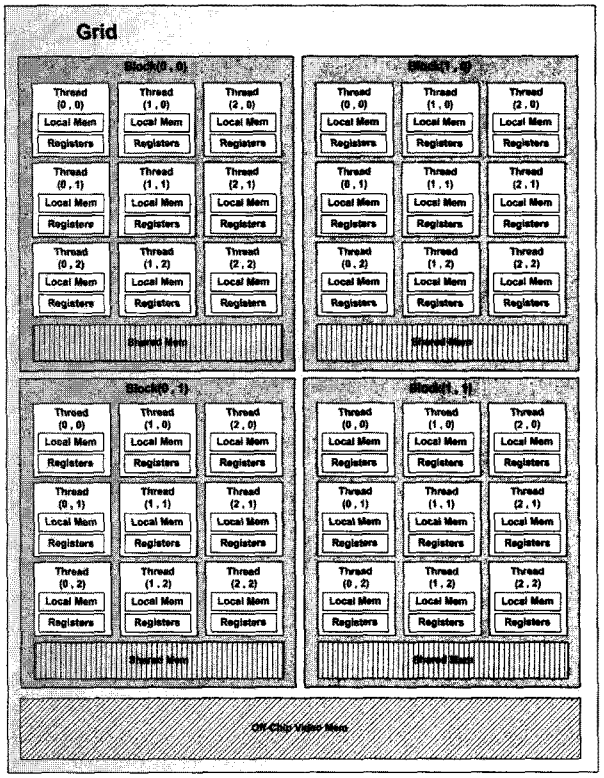


그림 3. CUDA의 병렬 처리 구조 및 메모리 모델
Fig. 3. Paralle processing architecture and memory model of CUDA.

GPU에서 연산된 결과값이 직렬코드 연산에 필요한 경우, 직렬코드는 정지되었다가 커널의 연산이 끝나면 진행된다. 그러나 직렬코드가 병렬코드와 별개의 내용을 처리할 경우 직렬코드는 CPU에서, 병렬코드는 GPU에서 각각의 코드를 실행한다.

CUDA의 병렬처리 구조는 그림 3. 과 같다. 호스트 프로그램에서 커널의 호출은 하나의 그리드(Grid)를 생성하며 그리드는 블록(Block)을, 블록은 쓰레드(Thread)로 구성되는 구조를 갖는다. 그림 3. 은 CUDA의 병렬처리 구조의 예로써 하나의 그리드가 2X2의 2차원 블록으로, 각 3X3의 2차원 쓰레드로 구성된 경우를 보여준다. 사용자는 병렬로 구현하고자 하는 어플리케이션의 알고리즘 특성에 맞게 쓰레드와 블록, 그리드의 개수와 구조를 적절하게 조절할 수 있으며 각각의 쓰레드, 블록은 고유한 ID를 부여받아 연관된 데이터를 할당받고 연산을 수행한다.

CUDA 메모리 모델은 GPU 내부의 온칩 메모리와 외부의 오프칩 메모리로 구분된다. 온칩 메모리는 데이터를 읽고 쓰는 시간이 짧지만 메모리 크기에 제한이 있다. 반면 오프칩 메모리는 용량이 크지만 데이터를 읽어오거나 저장하는데 온칩 메모리보다 많은 시간이 소요된다. 때문에 비효율적인 메모리 사용은 GPU를 통한 병렬처리의 성능향상을 제한하게 되므로 CUDA를 사용하여 최대의 성능을 얻기 위해서는 온칩 메모리와 오프칩 메모리 사이의 적절한 사용 및 관리가 매우 중요하다.

2. FDTD Method

FDTD는 임의의 공간에서 맥스웰 방정식을 수치해석적으로 풀어내는 방법이다. 1966년 Yee가 처음으로 제안한 이후, Taflove 등 많은 사람들에 의해 발전된 FDTD는 알고리즘이 비교적 간단하며 다양한 전파해석 문제에 적용할 수 있어 복잡한 구조를 갖는 마이크로파 해석에 적합하다.[10~11] 자유공간에서의 시간 의존적(time-dependent) 맥스웰 방정식은 다음과 같다.

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

$$\nabla \times H = -\sigma E + \epsilon \frac{\partial E}{\partial t} \tag{2}$$

이를 직교좌표계 각 축에 관한 스칼라 방정식으로 표현하여 각각의 식에서 구해진 E와 H요소를 yee cell로 정의된 공간에 배치한 후, half-time step을 고려하여 유한차분근사법(finite difference approximation)으로 풀이하여 나타낸다.

$$H_z^{n+1}(i+\frac{1}{2},j+\frac{1}{2},k) = H_z^n(i+\frac{1}{2},j+\frac{1}{2},k) - \frac{\Delta t}{\mu(i+\frac{1}{2},j+\frac{1}{2},k)}(E_y^{n+\frac{1}{2}}(i+1,j+\frac{1}{2},k) - E_y^{n+\frac{1}{2}}(i,j+\frac{1}{2},k) - E_x^{n+\frac{1}{2}}(i+\frac{1}{2},j+1,k) + E_x^{n+\frac{1}{2}}(i+\frac{1}{2},j,k)) \tag{3}$$

식 3. 은 H_z를 유한차분근사법으로 전개한 것이다. 같은 방법으로 각각의 직교좌표계에서 유한차분근사법으로 전개하면 FDTD 알고리즘의 수식이 완성된다. 위와같이 자유공간에서의 복잡한 전자계 맥스웰 방정식을 FDTD를 사용하여 계산할 수 있다.

III. FDTD 알고리즘의 구현

본 논문에서 구현한 프로그램은 메모리 할당부와 메인함수, 그리고 GPU에서 실행되는 커널 등 크게 3부분으로 구성된다. 메모리 할당부에서는 호스트 영역과 디바이스

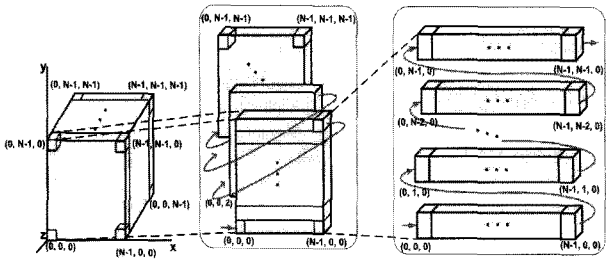


그림 4. FDTD 알고리즘의 메모리 할당
Fig. 4. Memory allocation of FDTD algorithm.

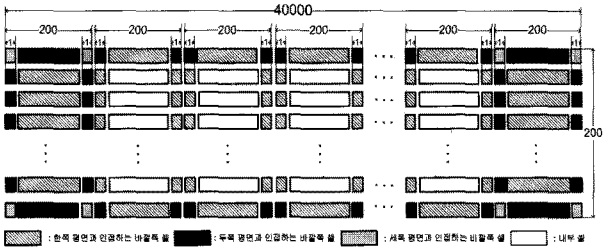


그림 5. CUDA로 구현한 FDTD 알고리즘의 셀 구조
Fig. 5. Structure of cells of FDTD algorithm.

영역에서 사용할 메모리 크기를 할당한다. FDTD 알고리즘을 구현하기 위해서 그림 4. 와 같이 자유공간 내에 N×N×N 크기의 임의의 셀을 정의한 뒤 (0, 0, 0)에서부터 (N-1, N-1, N-1) 좌표를 갖는 총 N^3개의 셀을 순서대로 일렬로 나열하였다. 즉, xyz축의 3차원에서 구성된 총 N^3개의 셀을 z축을 따라 N^2개의 셀로 구성된 N개의 xy평면으로 분할하고, 각 xy평면을 y축을 따라 다시 N개의 셀로 구성된 N개의 x축상의 선으로 분할하여 순차적으로 나열함으로써 N^3개의 셀로 구성된 3차원 공간을 1차원 배열로 구성한다. 3차원 공간에 분포된 데이터는 FDTD 연산시 세 개의 축 방향 전후로 인접한 주변셀의 데이터를 참조한다. 본 실험에서 구현한 프로그램의 경우 3차원 공간의 데이터를 1차원 배열에 저장하였으므로 임의의 셀이 인접데이터를 참조할 때 임의의 셀을 기준으로 첫번째, N번째, N^2번째 전후 셀의 데이터를 참조한다. 즉, 임의의 셀에 대한 데이터 array[a]의 계산을 위해 x축 인접셀인 array[a±1], y축 인접셀인 array[a±N], 그리고 z축 인접셀인 array[a±N^2]의 6개 인접셀에 대한 데이터의 접근이 필요하다.

메모리 할당이 끝나면, 메인 함수에서는 쓰레드와 블록의 크기를 선언하고 커널 호출을 실행하며 연산이 종료되면 사용한 메모리를 해제한다. 프로세서당 처리할 수 있는 쓰레드와 블록의 크기가 제한되어 있기 때문에 CUDA를 이용한 프로그램에서 쓰레드와 블록의 크기는 성능에 큰 영향을 미친다. 따라서 제한된 프로세서 자원 내에서 최대한의 데이터를 효율적으로 분배하고 연산할 수 있도록 쓰레드와 블록의 크기를 선정해야 한다. 더불어 쓰레드의 크기를 결정할 때 CUDA의 스케줄링 단위인 워프(warp)크기를 고려한다. 한 개의 워프는 32개의 쓰레드로 구성되어 있는데 GPU 내의 데이터 처리는 워프단위로 이루어진다. 예를들면, 33개의 쓰레드로 연산할 경우나 64개의 쓰레드로 연산할 경우 동일하게 2개의 워프를 통하여 처리하게 된다. 따라서 데이터를 할당할 때 쓰레드의 크기가 워프크기 단위와 일치하지 않을 경우 데이터 할당 및 연산시 불필요한 메모리 사용이 이루어져 성능향상에 제약이 따른다.

마지막으로 커널 함수에서는 설정된 쓰레드와 블록에 따른 ID를 부여하여 연관된 데이터를 메모리에서 가져와 연산을 실시한다. 그림 5. 는 N=200인 경우 커널 내부에서 실행되는 FDTD 알고리즘의 셀 구조이다. 좌에서 우로 그리고 위에서 아래로의 셀 연산이 전개된다. 내부 셀이 아닌, 3차원 공간에서 가장 바깥의 6면체의 면을 이루는 셀의 경우는 그 위치에 따라서 특정 축 방향의 인접셀이 존재하지 않는다. 이것은 전후 셀을 참조하는 알고리즘 연산의 경우 바깥면에 위치하는 셀들은 모든 인접셀의 값을 참조할 필요가 없다는 것을 의미한다. 따라서 N^3개의 셀 중 3차원 공간에서 바깥쪽 영역에 있는 셀에 대해서는 불필요한 데이터 참조를 실시하지 않도록 설계하였다.

Ⅳ. 실험 및 결과분석

실험에 사용된 시스템은 intel Quad core 2.66Ghz 클럭으로 동작하며 4Gbyte 크기의 메모리를 사용한다. 그래픽 디바이스의 오프칩 메모리를 초기화하기 위해서는 CPU 호스트의 DRAM 내부 데이터를 GPU 디바이스의 오프칩 메모리로 복사하는 과정이 포함된다. 또한 GPU 연산을 통하여 도출해낸 결과데이터는 실질적으로 사용하기 위해서 호스트의 DRAM으로 다시 복사되어야 한다. 때문에 호스트의 DRAM은 그래픽 하드웨어의 오프칩 메모리를 충분히 포함하는 것이 바람직하다. 사용한 그래픽 카드는 nVIDIA 사의 GeForce GTX 260으로 890Mbyte의 글로벌메모리와 멀티프로세서당 16Kbyte의 공유메모리를 제공하며 GPU내에 총 216(27 × 8 SM)개의 core를 내장하고 있다.

본 실험에서는 CUDA 2.3 라이브러리와 visual

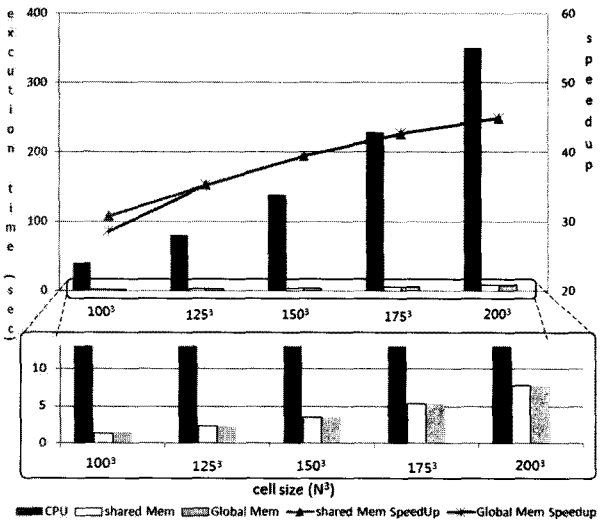


그림 6. CPU와 GPU를 사용한 FDTD알고리즘의 연산의 성능비교

Fig. 6. Performance comparison of FDTD algorithm using CPU and GPU.

studio 2005를 사용하였으며, FDTD 알고리즘 연산의 성능 비교를 위하여 clock() 함수를 사용하여 프로그램의 수행시간을 측정하고 프로그램별 각 10회 반복 측정 후 평균시간을 계산하였다.

그림 6. 은 FDTD 알고리즘을 CPU만으로 연산한 경우와 GPU의 글로벌메모리를 사용하여 연산한 경우, 그리고 GPU의 공유메모리를 사용하여 연산한 경우를 비교한 그래프이다. 그래프의 x축은 FDTD 연산을 위한 공간의 크기, 즉 FDTD를 연산한 단위공간의 셀수(N^3)를 의미하고 좌측의 주 y축은 알고리즘의 실행시간을, 그리고 우측의 보조 y축은 CPU 연산결과를 기준으로 하여 GPU를 사용하여 연산한 실행시간의 스피드업(Speedup) 정도를 나타낸다.

실험결과 CPU만을 사용한 연산의 수행보다 GPU를 함께 사용한 연산의 수행이 N의 크기에 따라 약 45배까지 성능이 향상됨을 확인할 수 있다. 성능향상의 정도는 선형적이지 않고 N의 크기가 커질수록 포화되어 가는 경향을 보인다. FDTD 알고리즘의 경우 N의 크기가 커지면 연산량과 더불어 연산에 사용하는 메모리 접근 횟수가 늘어나게 된다. 이때, N이 커질수록 전체 프로그램의 실행시간 중 연산에 소요되는 시간에 대한 메모리 접근이 차지하는 시간의 비율이 더 커지게 된다. 따라서 병렬 연산으로 얻는 이득보다 메모리 접근의 증가로 인한 손실이 커지게 되므로 스피드업의 증가 비율이 작아지게 된다.

공유메모리를 사용한 경우 글로벌메모리를 사용한 경우와 비교하였을 때 크게 성능개선이 이루어지지 않음을 확인하였다. 공유메모리를 효율적으로 사용하기 위해서는 연산에 사용되는 데이터를 지속적으로 재사용함으로써 메모리 접근시간을 줄여야 한다. FDTD의 경우 매 연산과정에서 결과값이 갱신되며, 다음 연산시 갱신된 데이터를 사용 한다. 하지만 공유메모리는 N^3개의 셀에 대하여 갱신되는 모든 결과값을 저장할 만큼의 충분한 용량을 제공하지 않기 때문에 연산 결과를 유지하기 위해 글로벌메모리의 사용이 불가피하다. 결과적으로 매 연산수행시 온칩 메모리와 오프칩 메모리의 데이터 접근이 동반되며 데이터 재사용의 결핍으로 기대되는 메모리 접근시간의 이득을 얻을 수 없다. 이는 프로그램 구현 시 프로그램에서 사용되는 메모리의 크기와 데이터의 재사용 여부 등 응용프로그램의 특성에 대한 이해가 바람직함을 보여준다.

V. 결 론

신호처리, 유체역학, 입자 물리학 및 우주과학을 비롯한 다양한 응용분야에서의 연산에 있어 사용되는 알고리즘들이 복잡하고 다루는 데이터의 크기나 종류가 매우 방대하므로 현존하는 단일 프로세서를 사용한 처리 방법은 물리적인 한계를 갖게 된다. 특히, 프로그램이 연속적이고 반복적인 연산 패턴을 갖는 경우는 병렬처리를 통하여 효율성 높은 연산을 실행하는 것이 바람직하다.

본 논문에서는 전자기학 분야에서 널리 사용되는 FDTD 알고리즘을 CUDA 라이브러리를 사용하여 구현하였다. 병렬화된 FDTD 알고리즘을 GPU 환경에서 실행시켜 봄으로써 CPU에서 실행시킨 경우와 비교하여 그 성능 향상 정도를 측정하였다. 실험 결과 CPU에서 FDTD 알고리즘을 연산한 경우보다 GPU에서 연산을 실시한 경우 문제의 크기에 따라 약 45배까지 성능 향상이 된 것을 관찰할 수 있었다. 하지만 GPU에서 지원하는 제한된 크기의 공유메모리를 사용하여 FDTD연산을 수행하는 경우에는 데이터의 재사용 횟수가 적어 글로벌메모리를 사용하는 경우와 비교하여 성능차이가 미미함을 볼 수 있었다.

참 고 문 헌

[1] E. Hesham, A. Mostafa, "Advanced Computer Architecture and Parallel Processing", WILEY, 2004.

[2] T. R. Halfhill, "Parallel Processing For the x86", MICROPROCESSOR REPORT, 2007.

[3] J. Y. Chen, "GPU technology trends and future requirements" Electron Devices Meeting (IEDM), 2009.

[4] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, J. C. Phillips, "GPU Computing" Proceedings of the IEEE, Vol.96 , Issue: 5, 2008.

[5] M. Houston, "GPGPU: General-purpose Computiatin on Graphics Hardware", SIGGRAPH 2007.

[6] D. Tarditi, S. Puri, J. Oglesby, "Accelerator Using data-parallelism to program GPUs for general-purpose uses", Int. Conf. Architect. Support Program. Lang. Oper. Syst, Oct, pp. 325 - 335, 2006.

[7] T. R. Halfhill, "Parallel Processing With CUDA : Nvidia's High-Performance Computing Platform Uses Massive Multithreading", MICROPROCESSOR REPORT, 2008.

[8] W. M. Hwu, C. Rodrigues, S. Ryoo, J. Stratton, "Compute Unified Device Architecture Application Suitability", Computing in Science & Engineering Vol.11, Issue: 3, pp. 16 - 26, 2009.

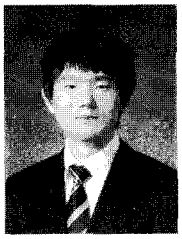
[9] S. Ryoo, "Optimization Principles and Application Performance Evaluation of a Multithreaded GPU using CUDA", ACM SIGPLAN Symp. Principles and Practice of Parallel Programming, pp. 73-82, 2008.

[10] K. S. Yee, "Numerical solution of initial boundary value problems involving Maxwell's equations in isotropic media", IEEE Trans. antennas and Propagation, Vol.14, pp. 302-307, 1996.

[11] D. M. Sullivan, "Electormagnetic simulation using the FDTD method", IEEE Press series on RF and microwave technology, 2000.

저 자 소 개

이 호 영(학생회원)
2005년 중앙대학교 전자전기공학부 학사 졸업.
2005년~현재 동대학원 석사과정
<주관심분야 : 컴퓨터구조, 병렬처리시스템>



박 종 현(학생회원)
2008년 중앙대학교 전자전기공학부 학사 졸업.
2010년 중앙대학교 대학원 전자전기공학부 석사 졸업.
<주관심분야 : 컴퓨터구조, 병렬처리시스템>

김 준 성(정회원)
1991년 중앙대학교 전자공학과 학사 졸업.
1993년 1993년 중앙대학교 전자공학과 석사과정
1998년 미네소타대학교 전기공학과 박사 졸업.
2002년~현재 중앙대학교 전자전기공학부 부교수
<주관심분야 : 컴퓨터구조, 병렬처리시스템, 시스템성능 분석, 임베디드 시스템 설계>