논문 2011-48CI-1-12

BioFET 시뮬레이션을 위한 CUDA 기반 병렬 Bi-CG 행렬 해법

(CUDA-based Parallel Bi-Conjugate Gradient Matrix Solver for BioFET Simulation)

박 태 정*, 우 준 명**, 김 창 헌***

(Taejung Park, Jun-Myung Woo, and Chang-Hun Kim)

요 약

본 연구에서는 연산 부하가 매우 큰 Bio-FET 시뮬레이션을 위해 낮은 비용으로 대규모 병렬처리 환경 구축이 가능한 최신 그래픽 프로세서(GPU)를 이용해서 선형 방정식 해법을 수행하기 위한 병렬 Bi-CG(Bi-Conjugate Gradient) 방식을 제안한다. 제안하는 병렬 방식에서는 반도체 소자 시뮬레이션, 전산유체역학(CFD), 열전달 시뮬레이션 등을 포함한 다양한 분야에서 많은 연산량이 집중되어 전체 시뮬레이션에 필요한 시간을 증가시키는 포아송(Poisson) 방정식의 해를 병렬 방식으로 구한다. 그 결과, 이 논문의 테스트에서 사용된 FDM 3차원 문제 공간에서 단일 CPU 대비 연산 속도가 최대 30 배 이상 증가했다. 실제 구현은 NVIDIA의 테슬라 아키텍처(Tesla Architecture) 기반 GPU에서 범용 목적으로 병렬 프로그래밍이 가능한 NVIDIA사의 CUDA(Compute Unified Device Architecture) 환경에서 수행되었으며 기존 연구가 주로 32 비트 정밀도(single floating point) 실수 범위에서 수행된 것과는 달리 본 연구는 64 비트 정밀도(double floating point) 실수 범위로 수행되어 Bi-CG 해법의 수렴성을 개선했다. 특히, CUDA는 비교적 코딩이 쉬운 반면, 최적화가 어려운 특성이 있어 본 논문에서는 제안하는 Bi-CG 해법에서의 최적화 방향도 논의한다.

Abstract

We present a parallel bi-conjugate gradient (Bi-CG) matrix solver for large scale Bio-FET simulations based on recent graphics processing units (GPUs) which can realize a large-scale parallel processing with very low cost. The proposed method is focused on solving the Poisson equation in a parallel way, which requires massive computational resources in not only semiconductor simulation, but also other various fields including computational fluid dynamics and heat transfer simulations. As a result, our solver is around 30 times faster than those with traditional methods based on single core CPU systems in solving the Possion equation in a 3D FDM (Finite Difference Method) scheme. The proposed method is implemented and tested based on NVIDIA's CUDA (Compute Unified Device Architecture) environment which enables general purpose parallel processing in GPUs. Unlike other similar GPU-based approaches which apply usually 32-bit single-precision floating point arithmetics, we use 64-bit double-precision operations for better convergence. Applications on the CUDA platform are rather easy to implement but very hard to get optimized performances. In this regard, we also discuss the optimization strategy of the proposed method.

Keywords: BioFET 시뮬레이션, CUDA, GPGPU, 병렬처리, 선형방정식해법, Bi-CG

---

* 정회원, *** 정회원-교신저자, 고려대학교 컴퓨터학과 CG Lab
(Computer Graphics Lab., Dept. of Computer Science, Korea University)
** 학생회원, 서울대학교 전기공학부
(Dept. of Electrical Engineering, Seoul National University)
※ This research is supported by Ministry of Culture, Sports and Tourism (MCST), Korea Creative Content Agency (KOCCA) in the Culture Technology (CT) Research & Development Program 2010, and the Second Brain Korea 21 Project.
접수일자: 2010년9월16일, 수정완료일: 2010년12월30일

I. 서 론

최근 학문의 융합의 관점에서 반도체 소자의 바이오 센서로의 이용을 목적으로 한 연구 동향이 점차 확대되고 있다.[1~2] 일반적인 어피니티 기반 바이오 FET (affinity-based Bio-FET)의 경우, 탐지하고자 하는 대상 분자와 상보적인 프로브(prove) 분자를 미리 반도체 표면에 기능화(functionalize)시켜 놓고, 전하를 띈 target 분자와의 반응을 반도체에서 전계 효과로써 감지해내는 동작 원리가 적용된다. 이러한 기술의 개발은 대량 생산이 가능한 반도체 소자를 이용하여 질병의 조기 진단이나 독성물질의 선감지 등을 쉽게 하는 데에 그 목적이 있다. Bio-FET의 연구가 확대됨에 따라 그 모델링에 대한 노력 또한 활발히 진행되고 있다.[3~4] 전해질 내에서 전하를 띈 바이오 분자에 의한 영향이 시뮬레이션의 핵심이 되는데, 시스템의 스케일에 비해 분자의 크기나 전기이중층의 스케일이 매우 작으므로 시뮬레이션에 앞서 메시 정의가 매우 중요하다.[5] 따라서 단위 분자 스케일의 메시를 적용하고 특히, 이를 3차원 시뮬레이션으로 고려할 경우 메시를 통해 생성되는 행렬 크기가 매우 커지게 된다. 더군다나 과도상태 응답을 구하기 위한 시뮬레이션의 경우 반복하여 3차원 메시의 행렬을 풀게 되는데 이는 과도한 계산량을 요하는 과정이다. 본 논문에서는 이 문제를 신속하게 해결하기 위해 행렬의 단위 계산 당 걸리는 시간을 줄이는 방법으로서 그래픽 프로세서(GPU)에 기반한 병렬처리를 도입한다.

그래픽 프로세서(GPU) 기술에서의 지금까지의 발전 방향은 단순히 그래픽 파이프라인을 통해 화면에 3차원 기하 정보 또는 2차원 영상 정보를 신속하게 출력하기 위한 고정된 하드웨어라는 한계를 벗어나 개발자가 직접 원하는 방식으로 화면에 출력할 수 있도록 프로그래밍이 가능한 방향으로 발전되고 있다. GPU에서의 프로그래밍 범위는 초기에 주로 화면 출력 방식(예. 카툰 렌더링 등)을 위한 목적으로 개발된 쉐이더(shader) 프로그래밍에 집중되었다. 그러나, GPU 하드웨어가 병렬 하드웨어 아키텍처로 발전해 온 특성에 주목한 결과 몇몇 연구자들에 의해 GPU를 활용해서 일반적인 수치 해석의 병렬화를 목표로 하는 GPGPU(General Purpose computing on Graphics Processing Units) 분야가 등장하게 되었다. 그러나 GPGPU는 초기에 GPU 하드웨어 및 소프트웨어 환경의 한계(예를 들어 실수 연산을 24 비트에서만 지원, 까다로운 메모리 액세스 등)로 인해 범용성과 가용성, 확장성 측면에서 많은 제약이 있었다.

이후 NVIDIA사에서 GPGPU에 적합한 하드웨어 및 소프트웨어 아키텍처로 개발한 CUDA(Compute Unified Device Architecture)[6]가 등장함으로써 많은 연구자들과 기관에서 이 아키텍처를 기반으로 한 여러 병렬 연구를 수행[7]하고 있다. 특히 본 연구에서 사용한 테슬라 아키텍처(Tesla Architecture) 기반 소비자용 GPU인 NVIDIA GTX 280 모델의 경우 제조사의 사양 문서[8]에서 명시한 이론 상 최대 연산 성능이 1 테라 FLOPS에 근접한다. 이러한 이론상 연산 성능은 90년대 중반 세계에서 가장 빠른 수퍼 컴퓨터들 중 하나인 CRAY 컴퓨터의 당시 성능보다 약 10배 정도 빠른 성능이다.

그러나 CPU에서의 일반적인 수치 해석 프로그래밍과는 달리 GPU는 고유한 하드웨어 아키텍처로 인한 특성을 고려하지 않고 CUDA 프로그래밍을 수행할 경우, 이러한 이론적인 최대 성능을 이끌어내지 못한 뿐만 아니라 단일 코어 CPU 보다도 더 성능이 저하될 수 있다. 이러한 성능 저하의 원인은 III 장에서 보다 자세히 논의한다.

본 논문에서는 CUDA를 바탕으로 반도체 소자 시뮬레이터, 전산유체역학 등에서 높은 연산 자원을 필요로 하는 포아송(Poisson) 방정식의 3차원 FDM(Fintite Difference Method) 해법을 위한 병렬 Bi-CG (Bi-Conjugate Gradient Solver)를 구현하고 Bio-FET 시뮬레이션의 근간이 되는 수용액 내 이온 이동에 대한 시뮬레이션을 수행한다. 반도체 시뮬레이션의 경우, 전체 시뮬레이션 루프 내에서 이전 단계에서 반도체 방정식(Drift-Diffusion 또는 Density Gradient)이나 몬테카를로(Monte-Carlo) 방식을 통해 계산된 전하의 분포에 따라 새로운 전위의 계산을 위해 포아송 방정식이 사용되며 전산유체역학에서는 유체 속력(또는 점성력)과 압력 사이의 관계가 포아송 방정식으로 표현된다. 이 이외에도 전산 기하에서 3차원 도형이나 2차원 이미지의 형태 변형 등에서도 포아송 방정식이 사용된다. 따라서 포아송 방정식의 병렬화, 고속화는 여러 분야에서 요청되는 연구라고 할 수 있다.

제 IV 장에서는 이러한 포아송 방정식의 병렬 해법을 위해 앞서 언급한 대로 CUDA 아키텍처에서 성능이 저하될 수 있는 문제점들을 논의하고 그 해법을 제시한다.

II. Bi-CG (Bi-Conjugate Gradient) 해법

Bi-CG 해법은 선형 방정식을 풀기 위한 반복 행렬 해법(iterative matrix solution) 중 하나이다. 이 장에서는 Bi-CG에 대한 기본적인 설명과 함께 병렬화 원리를 논의한다.

1. Krylov 부분 공간

FDM이나 FEM 등의 이산화 방식을 주어진 편미분 방정식과 경계 조건에 적용하면 다음과 같은 일반적인 선형 방정식을 얻는다.

$$A\vec{x}=\vec{b} \tag{1}$$

이 때 A는 이산화 결과로 얻은 희소 행렬이며 $\vec{x}$는 구하고자 하는 미지수 벡터, $\vec{b}$는 경계 조건 등으로 얻게 되는 주어진 벡터이다.

이 선형 방정식의 풀이를 위한 방식은 여러 가지가 있지만 크게 직접 해법(direct method)와 간접 해법(indirect method)가 있다. 간접 해법은 반복 해법(iterative method)라고도 한다.

반복 해법에는 상당히 다양한 방식이 존재하지만 k 번째 단계에서 근사한 미지수 $\vec{x_k}$로 인한 식 (1)의 오차(residual), $\vec{r_k}$를 다음과 같이 정의하고 이 값을 줄여 나가는 것을 목표로 한다.

$$\vec{r_k}=\vec{b}-A\vec{x_k} \tag{2}$$

k+1 번째 미지수 $\overrightarrow{x_{k+1}}$은 이 오차값을 기존 k 번째 미지수 벡터에 더한 값으로 갱신된다. 즉,

$$\overrightarrow{x_{k+1}}=\vec{x_k}+\vec{r_k}=\vec{x_k}+(b-A\vec{x_k}) \tag{3}$$

이 때 미지수 벡터 초기값 $\vec{x_1}$을 $\vec{b}$로 설정하면

$$\begin{aligned}
\vec{x_1}&=\vec{b}\\
\vec{x_2}&=\vec{x_1}+(\vec{b}-A\vec{x_1})=2\vec{b}-A\vec{b}\\
\vec{x_3}&=\vec{x_2}+(\vec{b}-A\vec{x_2})=A^2\vec{b}-3A\vec{b}+3\vec{b}\\
&\cdots
\end{aligned} \tag{4}$$

식 (4)에서 볼 수 있듯이 j 번째 $\vec{x_j}$는 $\vec{b}$, $A\vec{b}$, $A^2\vec{b}$..., $A^{j-1}\vec{b}$의 선형 결합이다. 이 때 벡터 $A^k\vec{b}\,(0\le k\le j-1)$를 기저 벡터(basis vector)로 해서 스팬(span)되는 부분 공간을 j 번째 Krylov 부분 공간(subspace), $\mathbb{K}_i$라고 한다. 이러한 수렴 과정에서 j 번째 $\vec{x_j}$를 결정하는 방법 중 하나로 $\vec{r_j}$를 $\mathbb{K}_i$에 직교화(orthogonal)하는 방법을 CG (Conjugate Gradient) 방식이라고 하고 또 다른 공간 $\mathbb{K}_i(A^T)$에 직교화하는 방법을 Bi-CG(Bi-Conjugate Gradient) 방식이라고 한다[9]

표 1. CG 알고리듬
Table 1. Algorithm for conjugate gradient method.

$$\begin{aligned}
&1.\ \alpha_k=\frac{r_{k-1}^T r_{k-1}}{d_{k-1}^T A d_{k-1}}\\
&2.\ x_k=x_{k-1}+\alpha_k d_{k-1}\\
&3.\ r_k=r_{k-1}-\alpha_k A\,d_{k-1}\\
&4.\ \beta_k=\frac{r_k^T r_k}{r_{k-1}^T r_{k-1}}\\
&5.\ d_k=r_k+\beta_k d_{k-1}
\end{aligned}$$

(r, d, x는 벡터)

2. Bi-CG(Bi-Conjugate Gradient)

본 논문에서 구현한 Bi-CG 해법은 비대칭 희소 행렬도 처리할 수 있는 방법으로, 대략적으로 대칭 희소 행렬만 풀 수 있는 CG를 두 번 풀게 된다(A와 A^T에 대해 각각 적용). 따라서 본 논문에서 적용한 병렬화 방법은 CG를 기준으로 설명한다.

앞 절에서 설명한 대로 CG에서는 $\vec{r_k}$가 $\mathbb{K}_k$ 내 모든 열 벡터와 직교해야 하는 조건을 둔다. 특정한 방식으로 모든 열 벡터가 직교화*된 Krylov 부분 집합에 대해 $\vec{r_k}=b-A\vec{x_k}$는 $\mathbb{K}_{k+1}$에 포함되며 $\vec{r_k}$는 $\mathbb{K}_{k+1}$의 k+1 번째 직교화된 열 벡터의 실수배가 된다. 이러한 과정을 통해 표 1과 같은 k 번째 CG 해법 루틴을 얻을 수 있다(자세한 내용은 [9] 참고). 이 루틴을 프로그래밍 관점에서 보면 모든 연산이 1) 벡터 간의 내적(예. $r_{k-1}^T r_{k-1}$), 2) 희소 행렬과 벡터의 곱(Ad_{k-1}), 3) 벡터 간의 선형 연산(예. $x_{k-1}+\alpha_k d_{k-1}$)으로만 구성됨을 볼 수 있다. 이러한 벡터들의 차원은 메시 크기로 결정되기 때문에 (예, 3차원 FDM에서 메시 크기가 100×100×100이라면 벡터의 차원은 106) 단일 프로세서 환경에서 프로그래밍을 할 경우 일반적으로 for 루프를 통해 벡터 차원만큼의 반복을 통해 계산된다. 특히 2)

* Arnoldi 알고리듬 사용. 자세한 내용은 [9]참고.

희소 행렬과 벡터의 곱의 경우 희소 행렬의 데이터 구조에 따라 그 연산 루프 길이는 더욱 길어진다.

CUDA에서는 동시에 병렬로 실행 가능한 실행 단위(스레드)의 개수가 다중 CPU 코어 시스템이나 CPU 클러스터링에 비해 상당히 많고(테슬라 아키텍처의 경우 30,720개) 이 보다 더 많은 데이터를 동시에 처리할 경우에도 CPU에 비해서 오버헤드 없이 매우 효율적인 파이프라인을 수행하기 때문에 매우 큰 크기의 벡터와 희소 행렬에 대해서도 위의 1), 2), 3) 연산을 단 몇 클럭 내에 처리할 수 있다. 이러한 측면에서 Bi-CG, 또는 CG 해법은 CUDA에 매우 적합한 방식이다. III 장에서는 이러한 성능을 가능하게 하는 CUDA의 하드웨어적, 소프트웨어적 측면에 대해 살펴 본다.

III. CUDA 아키텍처

1. CPU-GPU 하이브리드 아키텍처

CUDA는 기본적으로 CPU(host)와 GPU(device)가 함께 운용되는 하이브리드 아키텍처를 기반으로 하며 컴파일러를 통해 CPU 측 실행코드와 GPU 측 실행코드를 각각 생성한다. 이러한 서로 다른 특성을 가지는 프로세서군들이 함께 작동하기 때문에 가장 이상적인 작동 방식은 GPU와 CPU 각각 가장 적합한 유형의 연산을 동시에 수행하는 것이라고 할 수 있다. 최근 이러한 연구가 많이 수행되고는 있으나 아직 하드웨어적으로 GPU-CPU 간의 데이터 버스 대역폭이 제한적이라는 한계가 있다. 따라서 일반적인 애플리케이션에서는 최적 성능 보장을 위해 GPU-CPU 사이의 통신의 최소화가 권장되며 본 논문에서 제안하는 방식 역시 GPU-CPU 사이의 데이터 통신 규모의 최소화에 중점을 두고 설계되었다. 실제 구현에서는 GPU-CPU 사이의 데이터 통신을 최소화하기 위해서 CPU의 경우 힙(heap) 영역, GPU의 경우 디바이스 메모리(device memory) 영역에 행렬 및 기타 연산에 필요한 벡터 정보를 실행 초기에 한 번에 서로 복사하도록 했다. 이러한 데이터 통신 대역폭의 효율성은 CUDA Visual Profiler로 총 실행 시간 중 각 루틴이 차지하는 비중을 표시한 그림 7에서 살펴 볼 수 있다. 이 그림에서 CPU에서 GPU로의 메모리 전송(memcpyHtoD)과 GPU에서 CPU로의 메모리 전송(memcpyDtoH) 루틴은 전체 실행 시간 중에서 차지하는 비중이 상당히 낮으며 따라서 대역폭이 효율적으로 사용되고 있다고 결론을 내릴 수 있다.

2. 테슬라(Tesla) 아키텍처

테슬라 아키텍처는 NVIDIA에서 설계한 GPU 구조로서 세계 최초로 64비트 배정도 실수형(double)을 하드웨어적으로 구현한 GPU이다. 테슬라 아키텍처를 기반으로 소비자용으로 개발된 GTX285/280 모델의 경우 GPU 칩 1개 당 내장된 멀티프로세서(MP)의 개수가 30개이며 이 멀티프로세서 1개 당 8개의 스칼라 프로세서(SP)가 포함되어 총 240개의 SP가 내장된다. 이 SP는 동시에 여러 스레드를 실행할 수 있으며 MP는 스레드 전환 시 레지스터 스택 처리 등 오버헤드가 큰 일반적인 CPU와는 달리 거의 오버헤드 없이 하드웨어적으로 스레드를 생성하고 스레드 스케줄을 처리할 수 있다. 이렇게 동시에 처리할 수 있는 스레드 개수는 총 3만개 정도로 알려져 있으며 이 보다 더 큰 데이터의 경우, 병렬처리가 가능한 최대 부분으로 분할해서 병렬처리를 수행하며 이 경우에도 스레드 전환 오버헤드 및 스케줄링이 하드웨어를 통해 최적화되기 때문에 어느 정도까지는 메모리 등으로 인한 시간 지연이 감춰지는 것으로 알려져 있다.

메모리 측면에서는 각 MP별로 일종의 프로그래머블 캐시 역할을 담당하는 shared memory가 내장되며 그 외 읽기 전용인 일정(constant) 메모리와 텍스처 메모리가 내장된다. 가장 큰 크기를 가지는 디바이스 메모리의 경우 칩 외부 보드 상에 존재하기 때문에 메모리 액세스 패턴으로 인한 성능 차이가 크게 나타난다. IV 장에서는 희소 행렬과 벡터의 곱 연산에서 메모리 액세스 패턴으로 인한 성능 저하를 줄이기 위한 희소 행렬 압축 형식을 소개하고 여러 병렬 연산에서 최종 결과를 얻기 위해 사용되는 감소(reduction) 연산의 최적화 방식을 소개한다.

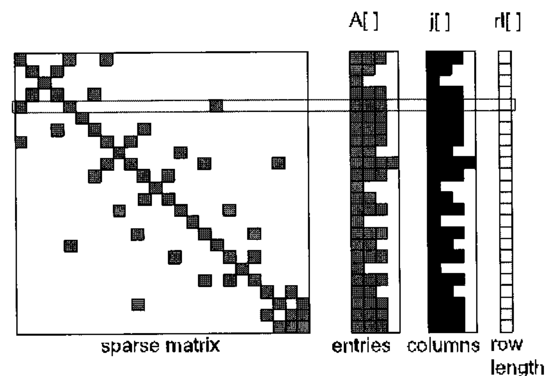

그림 1. ELLPACK-R 자료 구조
Fig. 1. ELLPACK-R data structure.

Ⅳ. 병렬 연산 최적화 기법

제 Ⅱ 장에서 논의한 것처럼 Bi-CG 해법은 반복 해법(iterative solver)으로서 1) 스칼라-벡터 곱, 2) 벡터-벡터 간의 벡터 덧셈(뺄셈 포함), 내적, 3) 벡터-행렬 간 곱셈 연산으로 구성된다. 특히 벡터-행렬 간의 연산의 경우 제 Ⅲ 장에서 설명된 메모리 액세스 방식에 따라 성능 상의 영향이 매우 크기 때문에 특별한 주의와 적절한 방식이 필요한 부분이며 1절을 통해 이 논문에서 사용한 방식을 논의한다. 또한 벡터 간의 내적의 경우 각 벡터 원소들의 합산 과정에 순차적 특성이 내포되어 있기 때문에 병렬 처리가 어려운 측면이 있다. 이러한 문제 해결에 가장 적합한 방식은 병렬 감소(parallel reduction) 연산으로 알려져 있으며 N 차원 벡터 내적 계산에서 $O(\log_2^N)$ 연산 복잡도를 보장한다. 그러나 CUDA 환경에서는 실제 구현 방식에 따라 실행 성능 차이가 매우 크다. 이러한 특성은 하드웨어(CPU, 캐시 계층, 메모리, 버스 등) 측면에서 최적화가 이루어져서 소프트웨어 측에서 최적화를 위해 고려해야 할 경우가 드문 일반적인 CPU 아키텍처와는 달리, GPU를 이용한 대규모 병렬시스템에서는 대규모 멀티스레드 실행을 위해 메모리 뱅크 충돌과 캐시 관리, 메모리 버스 대역폭 최적화 문제까지 소프트웨어 개발 환경에 그대로 노출되기 때문에 발생한다. 병렬 감소 연산에 대한 자세한 내용은 2절에서 소개한다.

1. 희소 행렬과 벡터 곱셈 최적화 행렬 자료구조

가. 관련 연구

FDM이나 FEM과 같은 PDE(partial differential equations) 이산화 방식을 통해 얻은 행렬의 형태는 대부분의 원소가 0이고 일부만 의미 있는 수치를 가지는 희소 행렬(sparse matrix)이다. 이러한 형태의 행렬을 일반적인 단정도(float) 또는 배정도(double) 2차원 N x N 배열을 이용해서 컴퓨터 메모리에 저장할 경우 메모리가 많이 낭비된다. 따라서 단일 CPU 프로그래밍 기반 수치해석 기법에서도 희소 행렬의 여러 효율적인 저장 형식이 제안된 바 있다. CUDA에서의 경우, 이러한 희소 행렬 내 원소들의 배치가 단지 메모리의 저장 효율성 측면뿐만 아니라 Ⅲ 장에서 논의한 대로, 무작위적인 메모리 액세스 패턴으로 인한 병렬 처리 성능에 악영향을 미치게 된다. 특히 여러 수치 해석 알고리즘의 근간이 되는 희소 행렬과 벡터 사이의 곱셈의 경우 희소 행렬의 메모리 저장 특성 상 CUDA 환경에서 여러 선형 연산에 대한 병렬 처리 성능을 저해하는 주요한 요인으로 지목되고 있다.

이러한 이유로 인해 최근 1~2 년 전부터 이 문제에 대한 여러 연구가 수행되고 있다. 그러나 한 가지 주목해야 할 사실은 여러 연구에서 제시한 각각의 방법들의 성능이 희소 행렬에서 원소들의 분포 구조에 따라 각기 다른 성능을 제공한다는 점이다. 따라서 최근 희소 행렬의 구조에 따라 최적화되는 희소 행렬 구조가 제안[10]되기도 했으나, 아직 까지 이론 수립 과정 단계이며 추가 연구가 필요한 상황이다.

본 연구에서는 최근 발표된 방식들 중에서 CUDA 아키텍처에 최적화된 희소 행렬 데이터 저장 구조인 ELLPACK-R을 사용했다.

나. ELLPACK-R

ELLPACK-R[11]은 기존 ELLPACK 방식을 수정하여 GPU 메모리 액세스 패턴을 보다 통합적인 방식(coalesced memory access pattern)에 가깝게 하고자 ELLPACK의 형식에 추가로 각 행(row)에서 0이 아닌 원소들의 개수를 보관하는 정수형 1차원 배열 *rl[]*을 추가한다. 그림 1은 ELLPACK-R의 자료 구조를 보여주고 있다. 가장 왼쪽에 표시된 일반적인 $N \times N$ 희소 행렬은 행 방향으로 압축되어 0이 아닌 원소들이 배정도 실수형(double) 또는 단정도 실수형(float) 형의 $N \times m$ 크기 2차원 배열 *A[]*에 저장된다. 이 때, m은 다음 식으로 주어진다.

$$m = \max(rl[i]), 0 \le i < N \tag{5}$$

또한 정수형 $N \times m$ 크기 2차원 배열 *j[]*에는 각 행의 0이 아닌 원소들의 열(column)에서의 색인 번호가 저장된다. 그림 1에서 표시한 빨간색 사각형은 희소 행렬과 각 저장 변수 *A[], j[], rl[]*의 상관 관계를 한 행에 대해 도식화하고 있다.

이러한 ELLPACK-R 형식의 특징은 이 형식이 발표된 논문[11]에서 논의한 대로 기존 방식들(예. CRS, CRSN 등)에 비해 행렬과 벡터의 곱셈 연산이 GPU 하드웨어에서의 메모리 액세스 패턴에 보다 부합된다는 장점을 제공한다. 이 방식에서는 분산되어 있는 0이 아닌 원소들을 곱셈이 수행되는 방향인 행 방향(대칭 행

렬인 경우에는 열 방향과 행 방향 동일)으로 압축시켜 배열 *A[]*에 저장함으로써 메모리 액세스 패턴을 개선시켰다. 이러한 특성은 단지 하드웨어 캐시가 메모리에 별도로 구현되지 않은 NVIDIA 테슬라 아키텍처 GPU에서 뿐만 아니라 일반 CPU에서도 캐시를 고려한 데이터 구조(cache-aware data structure)[12]로 활용될 수 있다.

그러나 GPU 환경에서의 ELLPACK-R은 논의한 대로 *A[]*에서는 메모리 액세스 패턴 개선이 가능하나, 이 행렬과 곱셈을 수행할 벡터의 경우 *j[]*에서 해당 벡터의 위치 색인을 읽어 와서 불규칙적으로 메모리 액세스를 수행해야 하기 때문에 해당 벡터가 저장된 메모리 공간에 대해서는 앞서 설명한 메모리 액세스 패턴의 장점이 어느 정도 희석된다. 따라서 이 방식이 가장 적합한 희소 행렬 구조는 많은 다른 직접/간접 해법에서와 마찬가지로 0이 아닌 원소들이 대각 행렬 원소들 주변으로 연속적으로 밀집해 있는 구조라고 할 수 있다.

2. 병렬 감소(Reduction) 연산

병렬 감소(reduction) 연산은 병렬 연산 시스템에서도 부득이 하게 부분적으로 순차적으로 수행해야 하는 연산에 적용되는 방식이다. 병렬 감소 연산이 적용되는 연산은 여러 가지가 있지만 본 논문에서 제안하는 방식에서는 벡터 내적, 벡터 원소들 중 최대값 (또는 최소값) 검색 연산에 사용된다. 이러한 병렬 감소 연산에 대한 연구는 이론적인 측면[13]과 실제 CUDA 애플리케이션에서의 성능 최적화 등 실무적인 측면[14]에서 오래 전부터 활발하게 진행되어 왔다. 그림 2는 병렬 감소 연산의 기본적인 구조를 설명하는 그림이다. 이 그림에서처럼 병렬 감소 연산은 상하가 뒤집어진 2진 트리 형태를 가진다. 각 사각형 노드는 연산의 대상이 되는 데이터 요소를 나타낸다. 예를 들어 N 개의 실수 중에서 가장 큰 수를 찾는 문제가 주어졌다고 할 경우, 가장 위쪽에 N 개의 실수가 입력되며 각각 2개씩 묶어 둘 중 큰 수를 다음 단계로 보낸다(위에서 두 번째 줄).

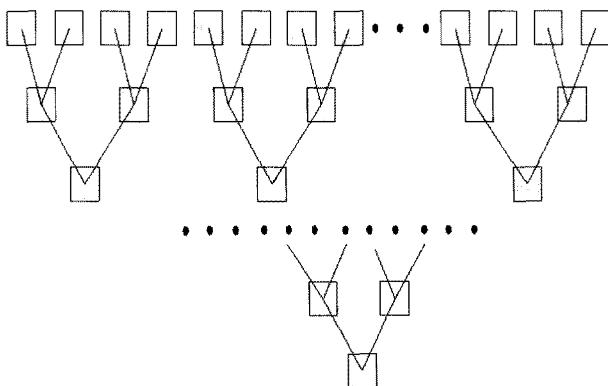

그림 2. 병렬 감소 연산
Fig. 2. Parallel reduction operation.

이렇게 $\log_2^N$ 단계를 거친 후 최종적으로 가장 큰 실수가 마지막 단계에 남게 된다. 앞서 언급한 대로 이 루틴은 최대/최소값 검색 뿐만 아니라 벡터 내적 연산에도 이용 가능하다. 일반적으로 OpenMP 등과 같이 성능이 높은 프로세서를 상대적으로 적은 개수를 적용하는 구조(coarse-grained parallel architecture)의 경우에는 제일 상위 단계에서 한 프로세서 내에 여러 개의 데이터 요소들을 처리하지만 CUDA와 같이 성능이 상대적으로 낮은 프로세서를 많이 사용하는 구조(dense-grained parallel architecture)에서는 프로세서 (또는 스레드) 하나 당 데이터 요소 하나를 할당하게 된다.

CUDA 기술 백서[14]에서는 CUDA에서의 구현 시 메모리 액세스 패턴과 루프 풀기(loop unrolling) 기법을 적용함으로써 6단계에 걸친 최적화를 소개하고 있다. 그 결과로 가장 낮은 단계의 최적화 대비 최대 최적화 성능이 약 20배 이상 증가함을 보이고 있다. 본 논문에서는 Bi-CG 루틴 중에서 벡터의 내적과 벡터 내 최대 요소를 찾는 루틴 ($\|\bullet\|_{max}$ norm 연산)에 병렬 감소 연산을 적용한다.

V. 실 험

본 논문에서 제안하는 CUDA 기반 병렬 Bi-CG 루틴은 BioFET 시뮬레이션을 위한 전해질 용액 내 이온 분포의 물리적 해석을 위해 적용되었다. 기본적으로 전체 시뮬레이션 시스템은 반도체 방정식(DD)과 동일하다고 볼 수 있다.

반도체 방정식 해법에서 포아송 방정식은 반도체 소자 내에서의 전자와 정공 등의 분포를 기술한다. 이와 마찬가지로 전해질 용액 내에서의 양이온과 음이온 역시 전하를 띈 반송자(carrier)이므로 포아송 방정식에 의해 기술될 수 있다. [n+]와 [n-]를 각각 양이온과 음이온의 농도라고 할 때, 다음 방정식으로 전위 ψ와의 관계를 기술한다.

$$\nabla\cdot\varepsilon(-\nabla\psi)=q([n^+]-[n^-]) \quad (6)$$

또한 양이온과 음이온은 전자와 정공처럼 DD (drift-diffusion)에 의한 연속방정식에 의해 그 운동을 기술할 수 있는데 이는 다음과 같이 표현이 된다.

$$\frac{\partial[n^{\pm}]}{\partial t}=-\frac{1}{q}\nabla\cdot J_{[n^{\pm}]},\ J_{[n^{\pm}]}=-q\mu_{n^{\pm}}[n^{\pm}]\nabla\psi\mp qD_{n^{\pm}}\nabla[n^{\pm}] \quad (7)$$

기서 $J_{[n^{\pm}]}$는 양(음)이온의 전속밀도, $\mu_{n^{\pm}}$는 양(음)이온의 이동도, $D_{n^{\pm}}$는 양(음)이온의 확산계수를 의미한다.

전해질 용액에서 전극에 전압이 가해지거나 전하를 띈 분자를 고려할 경우, 이로 인해 발생한 전위가 전해질 용액 속의 이온들에 의해 짧은 거리 안에서 차폐(screening)가 된다. 이 결과로 전기이중층(electric double layer)이 생기게 되는데, 이는 전해질 속의 목적 분자의 탐지(sensing)에 중요한 역할을 한다.

양(음)이온의 유한한 속도 때문에 전기이중층의 생성에는 유한의 시간이 걸리게 되는데, 이 과도상태에서 정상상태에 이르기까지 전위의 계산이 필요하다. 이를 위해 전극에 스텝 펄스로 가해진 전위를 3차원 시뮬레이션의 경계조건으로 하여, 정상상태에 이르기까지 포아송 방정식과 연속 방정식을 교차적으로 갱신하면서 풀어준다.

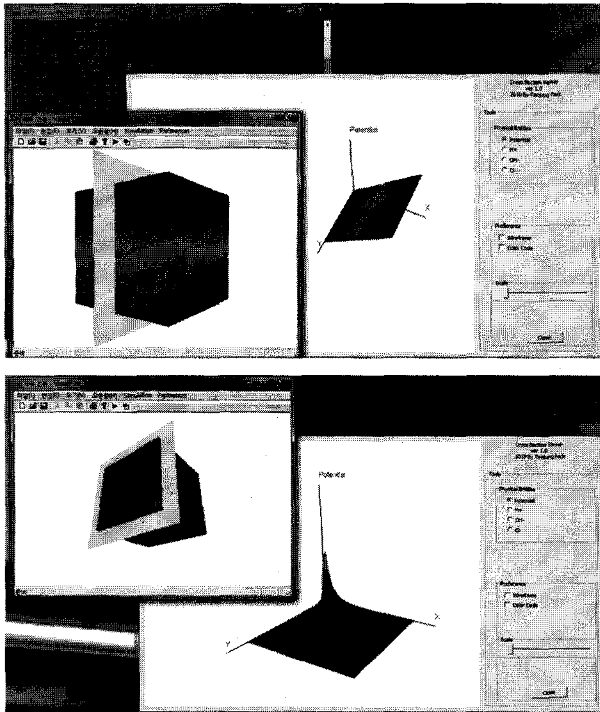

그림 3. MS Windows GUI 환경 병렬 CUDA FDM 시뮬레이터 (파란색: 0.5V, 녹색: 0.25V, 빨간색: 0V, 상자 주변의 노란색 사각평면은 문제 영역 내부 전위 관찰을 위한 절단면을 의미)

Fig. 3. Our parallel CUDA FDM simulator implemented in MS Windows GUI environment (blue: 0.5V, green: 0.25V, red: 0V. The yellow square around the box indicates a cross-sectioning plane for the problem domain.)

이러한 BioFET 시뮬레이션의 경우 일반적인 MOSFET 단일 소자 시뮬레이션과는 달리 문제 영역이 더 크고 유기체 등으로 인한 경계 조건(BC)을 다루어야 하기 때문에 더 크고 복잡한 기하 구조를 가진 메시(mesh)를 사용해야 한다. 이러한 상황에서, FDM을 통해 얻는 희소 행렬의 크기는 메시 노드(mesh node) 수의 제곱에 비례(비압축 형식에서)하며 단일 프로세서 기반의 선형 방정식 기법의 연산 복잡도는 보통 O(N3)(N은 버텍스 개수)이기 때문에 연산 부담이 크게 증가한다. 따라서 앞서 논의한 시뮬레이션 루틴 중에서 포아송 방정식 해법이 전체 연산에서 차지하는 부분이 상당하기 때문에 병렬화를 통한 이 부분의 적절한 해법이 필수적이라고 할 수 있다.

그림 3에서는 실험을 위해 제작된 전해질 용액 내 이온 분포 해석 시뮬레이터의 화면을 보여 주고 있다. 이 시뮬레이터는 64비트 Windows 환경에서 MFC (Microsoft Foundation Class library)를 통해 GIU 환경에서 프로그래밍 되었으며 NVIDIA사의 CUDA 환경에서 본 논문에서 제시한 병렬화 알고리듬이 구현되었다. 또한 과도 응답(transient) 시뮬레이션 중간 결과를 시뮬레이션 도중에 시각적으로 확인할 수 있도록 하기 위해 다중 스레드 프로그래밍 기법과 OpenGL이 적용되었다.

2. 결과

가. CPU와 GPU의 실행 시간 비교

그림 4는 일반적인 CPU 환경과 GPU에서의 Bi-CG 방법을 이용한 포아송 방정식 해법 처리 시간을 비교한 그래프이다. 실험에 사용된 PC 시스템은 2대로, 각각 인텔사의 i7과 Core2Duo CPU를 탑재한 64 비트 Windows 7 환경에서 NVIDIA 사의 GTX285 GPU (1GByte)를 설치해서 테스트를 수행했다*. GTX285 GPU는 최초로 배정도(double) 실수 연산을 지원하는 그래픽칩으로서 이 실험에서도 모든 연산은 배정도 실수 연산으로 수행되었다. 참고로 GTX285의 단정도 ALU (Arithematic Logic Unit) 개수는 배정도 ALU의 8배이기 때문에 단정도로 실행할 경우 대체적으로 8배

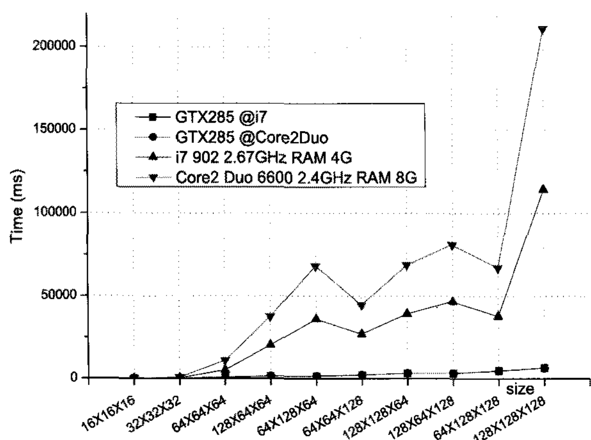

그림 4. 일반적인 CPU 환경과 GPU에서의 포아송 방정식 해석 시간 비교(블록 당 스레드 개수는 256개 기준).

Fig. 4. Time comparison between CPU vs. GPU to solve the Poisson equation (the number of threads per block is 256).

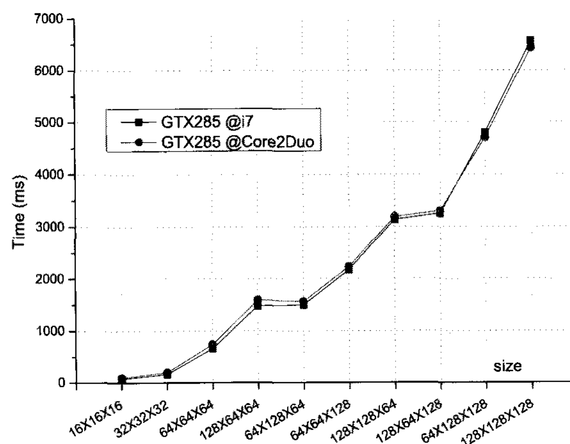

그림 5. 서로 다른 성능을 가지는 CPU가 설치된 환경에서 동일한 GPU를 설치한 CUDA 시스템의 성능 비교(블록 당 스레드 개수는 256개).

Fig. 5. CUDA system performance comparison on two CUDA systems which have different CPUs but with one GPU (number of threads / block is 256).

성능 향상을 기대할 수 있다. 이 그래프의 Y축은 실행시간(msec)을 의미하고 X축은 문제 공간에서 x 축, y 축, z 축 방향으로 메시 노드의 개수(예. x 축 64개, y 축 128개, z 축 64개일 경우 64×128×64로 표시)를 의미한다. 주목할 점은 메시 크기가 가장 작은 경우 (즉, 16×16×16) CPU에서 실행한 속도가 GPU 보다도 오히려 빨랐다는 점이다(i7에서 17.496 ms, i7 시스템에 설치한 GTX285에서 75.7 ms). 이러한 결과는 문제 크기가 작을 경우 프로세서의 개수 보다는 개별 프로세서의 성능이 전체 성능에 더 큰 영향을 주며, GPU에 여러 개의 프로세서가 포함되어 있으나 개별 성능으로만 본다면 CPU의 성능이 월등히 우수하기 때문이다. 그러나 16×16×16의 경우를 제외하고 모든 경우에 GPU에서의 처리 시간이 월등하게 짧아진다는 사실을 확인할 수 있다. 특히, 문제 공간이 가장 큰 128×128×128의 경우, 최대 33배 정도 차이가 나는 것을 확인할 수 있다 (Core2Duo에서 실행 시간은 2.11×10^{5}ms, GTX285에서는 6.43×10^{3}ms).

나. 다른 성능의 CPU 환경에 설치된 GPU (CUDA) 시스템의 실행 시간 비교

III 장에서 논의한대로 CUDA 환경은 전적으로 GPU만 이용하는 것이 아니라 일반적인 CPU 시스템을 동시에 이용할 수 있는 혼성(하이브리드) 시스템이다. 다시 말해서 이론적으로는 반드시 직렬화 처리 루틴을 사용해야 하는 코드는 개별 프로세서 차원에서 성능이 더 뛰어난 CPU에서 처리하도록 하고 CPU의 처리 동안 대규모 병렬화 루틴을 GPU에서 동시에 처리하도록 구성할 수 있다. 그러나 CPU와 GPU의 동시 처리는 이 두 가지 소자 사이를 연결하는 메모리 버스의 느린 속도 때문에 아직 한계가 있으며 병렬 부분은 GPU가 처리하고 병렬로 처리하기 힘든 부분은 CPU가 맡아서 처리하는 것이 보통이다. 따라서 CUDA 시스템에서의 애플리케이션 성능은 전적으로 GPU에 의해서만 결정되는 것이 아니라 CPU 및 기타 시스템(예. 데이터 버스 및 RAM)의 성능도 부분적으로 영향을 미친다고 할 수 있다. 그림 5에서는 본 논문에서 제안하는 방식에 대해 CPU의 성능이 미치는 영향을 비교한다. 그림 3에서 볼 수 있듯이 비교적 성능차이가 분명한 CPU (즉, i7과 Core2Duo) 시스템 환경에서 동일한 GPU(GTX285)를 설치하고 그 결과를 비교했다. 그림 5에서 확인할 수 있듯이 미미한 결과 차이가 있기는 했으나 유의미한 수준은 아니라고 결론 내릴 수 있으며 따라서 제안하는 Bi-CG 솔버는 CPU에 거의 의존하지 않고 GPU에만 거의 의존한다고 할 수 있다. 또 다른 측면에서 보자면, 루틴 중 대부분이 병렬화되어 GPU에서 실행된다는 것으로 볼 수도 있다.

* i7 902 2.67GHz, RAM 4 GBytes Core2Duo 6600 2.4GHz, RAM 8 GBytes

다. 메모리 액세스 패턴과 블록 당 스레드 개수의 영향

개별 프로세서의 성능이 큰 폭으로 향상되면서 현재 병렬 시스템(단일 칩, 단일 보드 상의 병렬 시스템, 네트워크로 연결되는 클러스터링 병렬 시스템 등)에서 가장 큰 성능 병목 현상을 보이는 부분은 메모리 대역폭이다. 즉, 대규모 데이터를 여러 프로세서로 동시에 짧은 시간 내에 푼다고 하더라도 계산이 끝난 데이터를 메모리에 다시 저장하고 다음 단계에서 계산할 데이터를 읽어 오는 과정에서 메모리 대역폭이 좁기 때문에 여러 프로세서들이 필요한 작업을 수행하지 못하고 대기해야 하는 문제가 성능 저하의 가장 큰 원인이 되는 것이다.

앞서 언급한 대로 일반적인 CPU 시스템에서는 캐시의 하드웨어적인 발전으로 인해 메모리 대역폭 문제가 상당 부분 해소된 상황이지만 GPU의 경우 여러 개의 프로세서가 개별적으로 캐시를 가질 경우 공유 메모리와의 데이터 불일치 문제가 심각해 질 수 있기 때문에 이러한 하드웨어적인 문제를 소프트웨어적인 해결 방법으로 해결해야 한다*.

그러나 하드웨어 캐시가 높은 수준으로 최적화되었다고 인식되는 CPU의 경우에도 메모리 액세스 방식에 따라 여전히 성능의 차이가 발생한다. 이러한 이유로 그림 4 CPU 실행 시간 그래프에서 i7과 Core2Duo 모두 전체 메시 크기가 같더라도 메모리 액세스 패턴에 영향을 주는 공간상의 메시 노드 분포에 따라 상대적으로 성능 차이가 확연하게 드러난다. 예를 들어, 64×128×64와 128×64×64, 64×64×128은 모두 메시 내 포함된 노드의 개수가 524,288개로 행렬 크기도 동일하지만 메모리 액세스 패턴이 달라지기 때문에 64×128×64에서 분명한 성능 저하를 확인할 수 있다. 이 구조에서의 실행 시간은 메시 노드 개수가 2배(즉, 희소 행렬 크기는 비압축에서 4배)에 해당되는 128×128×64와 64×128×128의 수행 시간과 거의 동일함을 볼 수 있다. 또한 128×64×128 구성은 같은 크기임에도 불구하고 128×128×64와 64×128×128 구성에 비해서 실행 시간이 증가함을 비교적 분명하게 확인할 수 있다[14].

그림 6에서는 GPU의 경우 각 블록[8]에 포함된 스레드의 개수를 달리 함으로써 메모리 액세스 패턴으로 인한 영향이 좀 더 복잡한 양상으로 나타남을 볼 수 있다.

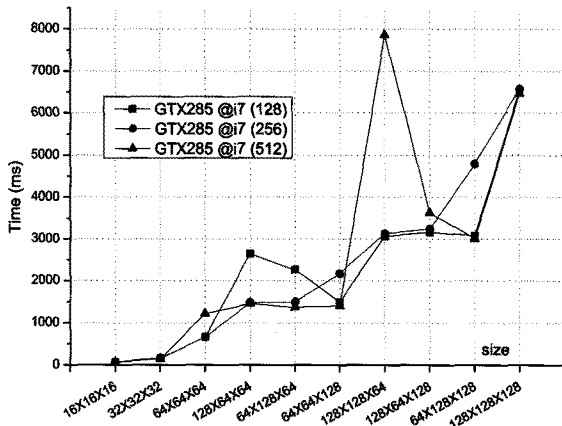

그림 6. 동일한 CPU, 동일한 GPU에서 블록 당 스레드 개수의 변화에 따른 성능 변화.

Fig. 6. Performance differences according to various number of threads per block in the same CPU and GPU.

라. 각 하위 루틴의 비중

그림 7에서는 본 논문에서 구현한 전체 루틴에서, 각 개별 루틴이 전체 루틴에 대해 차지하는 상대적인 실행 시간 비중을 표시한 그래프로 CUDA Visual Profiler를 이용해서 측정했다. Y 축 방향을 따라 표시된 이름은 각 하위 루틴의 이름을 의미하고 그 옆의 괄호 속에 표시된 숫자는 실행 회수를 나타낸다. 그리고 X 축을 따라 표시되는 녹색 막대는 전체 실행 시간 대비 각 하위 루틴의 상대적인 실행 시간을 의미한다. 결과 그래프에서 보면 SpMV로 표시되는 희소 행렬과 벡터의 곱 연산이 가장 큰 비중(38.77%)을 차지하고 있다는 사실을 확인할 수 있다. 따라서 IV장에서 설명한 이 부분에서 더 많은 최적화가 필요하다고 볼 수 있다. 그 다음으로

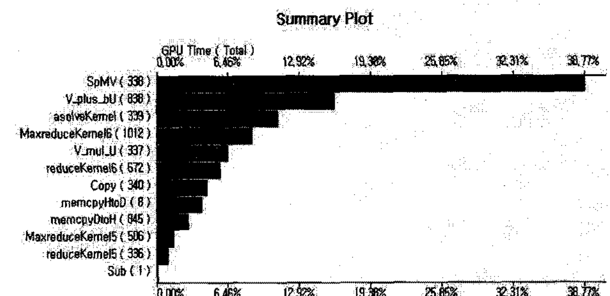

그림 7. CUDA Visual Profiler 결과

Fig. 7. A result from CUDA Visual Profiler.

* 최근 NVIDIA에서 발표한 페르미(Fermi) 아키텍처의 경우 하드웨어 캐시를 적절하게 이용함으로써 이 문제를 어느 정도 해결했다. 본 논문은 그 이전 세대 아키텍처인 테슬라(Tesla) 아키텍처를 기준으로 작성되었다.

표 2. 그림 7의 하위 루틴 설명
Table 2. Descriptions of the subroutines in Fig. 7.

이름	설명	비중(%)
SpMV	희소 행렬과 벡터 곱셈	38.76
V_plus_bU	$\vec{v}+b\vec{u}$	16.06
asolveKernel	preconditioning	10.86
MaxreduceKernel6	최대값 검색 reduction	8.51
V_mul_U	벡터 원소별 곱셈	6.35
reduceKernel6	벡터 내적 검색 reduction	5.66
Copy	벡터를 다른 벡터로 복사	4.46
memcpyHtoD	CPU -> GPU 데이터 전송	4.03
memcpyDtoH	GPU -> CPU 데이터 전송	2.8
MaxreduceKernel5	최대값 검색 reduction	1.46
reduceKernel5	벡터 내적 검색 reduction	0.97
Sub	벡터 간 뺄셈	0.01

많은 연산 시간을 소비하는 루틴은 V_plus_bU 루틴으로 이 루틴은 $\vec{v}+b\vec{u}$ ($\vec{v}, \vec{u}$는 벡터, b는 스칼라) 형태의 연산을 수행한다. 이 연산들이 차지하는 비중이 두 번째(16.06%)로 크기는 하지만, Bi-CG 루틴의 특성 상 이러한 연산이 많이 수행(838회)되어야 하기 때문에 루틴 하나의 효율성이 그리 떨어진다고 볼 수는 없다. 각 하위 루틴의 설명은 표 2를 통해 정리한다.

VI. 결 론

본 논문에서는 BioFET 시뮬레이션을 위한 전해질 용액 내 이온 분포의 물리적 해석을 위해 GPU를 사용해서 병렬화함으로써 포아송 방정식 해법에서 최대 33배의 성능 향상을 실현하였다. 본 연구에서 다루는 문제의 경우 가장 큰 값과 가장 작은 값의 차이가 매우 크기 때문에 32비트 단정도 실수를 사용할 경우 수렴하지 않는 문제가 종종 발생하며 현재 사용하는 GPU 아키텍처에서 8배 정도 속도 저하를 감수하고도 33배의 성능 향상을 얻었다. 또한 본 논문의 결과 부분에서 제시한 메모리 액세스 방식에 따른 캐시 hit/miss로 인한 성능차가 GPU 뿐만 아니라 CPU에서 발생한다는 사실은 기타 시뮬레이션 프로그래밍의 성능 향상을 위한 중요한 시사점을 제공한다고 본다.

또한 메시의 크기와 기하 특성이 매우 복잡해지고 방대해 지는 BioFET 시뮬레이션의 특성을 고려한다면 병렬 처리가 불가피하다고 결론을 내릴 수 있다. 동시에 현재 대부분의 병렬 시스템에서 프로세서의 성능보다는 메모리 통신 대역폭이 성능 상의 가장 큰 병목 지점임을 고려할 때 연구의 관심이 점차, 메모리 통신 지연이 네트워크까지 이어 지는 기존의 클러스터링 기반 수퍼컴퓨터로부터 GPU 같은 원칩/원보드 대규모 병렬 시스템으로 이동할 가능성이 높다고 결론 내릴 수 있다.

참 고 문 헌

[1] Y. Cui, Q. Wei, H. Park, C. M. Lieber, Nanowire Nanosensors for Highly Sensitive and Selective Detection of Biological and Chemical Species, Science 293, 1289, 2001.

[2] C. Stagni, C. Guiducci, L. Benini, B. Riccò, S. Carrara, B. Samorí, C. Paulus, M. Schienle, M. Augustyniak, and R. Thewes, CMOS DNA Sensor Array With Integrated A/D Conversion Based on Label-Free Capacitance Measurement, IEEE Journal of Solid-State Circuits 41, 2956, 2006.

[3] D. Landheer, G. Aers, W. R. McKinnon, M. J. Deen, and J. C. Ranuarez, "Model for the field effect from layers of biological macromolecules on the gates of metal-oxide-semiconductor transistors," Journal of Applied Physics, vol. 98, p. 044701-044701-15, 2005.

[4] Y. Liu, K. Lilja, C. Heitzinger, and R. W. Dutton, "Overcoming the screening-induced performance limits of nanowire biosensors: a simulation study on the effect of electro-diffusion flow," in IEDM Tech. Dig., San Francisco, pp. 491-494, 2008.

[5] C. Heitzinger, N. J. Mauser, and C. Ringhofer, Multiscale Modeling of Planar and Nanowire Field-Effect Biosensors, SIAM J. Appl. Math. Volume 70, Issue 5, pp. 1634-1654, 2010.

[6] NVIDIA, CUDA C Programming guide. Version 3.1.1, July, 2010. http://developer.download.nvidia.com/compute/cuda/3_1/toolkit/docs/NVIDIA_CUDA_C_ProgrammingGuide_3.1.pdf

[7] 이호영, 박종현, 김준성, "CUDA를 이용한 FDTD 알고리즘의 병렬처리," 전자공학회논문지, 제47권 CI편, 제4호, 82-87쪽, 2010년 7월

[8] 이주석, 류현곤, "GPU 병렬 컴퓨팅 기술을 이용한 개인용 수퍼 컴퓨터 현황과 전망 - CUDA 기술에 대한 이해," 전자공학회논문지, 제36권, 제5호, 18-27쪽, 2009년 5월

[9] G. Strang, Computational Science and Engineering, Wellesley-Cambridge Press, pp. 586-595, 2007.

[10] F. V´azquez, G. Ortega, J.J. Fern´andez, E.M. Garz´on. Improving the performance of the sparse matrix vector product with GPUs, In Proceedings of the *10th IEEE International Conference on Computer and Information Technology (CIT 2010)*, pp. 1146-1151. Bradford, the UK, July, 2010.

[11] F. V´azquez, E.M. Garzón, J.A. Martínez, and J.J. Fernández. Accelerating sparse matrix vector product with GPUs. In Proceedings of the *2009 International Conference on Computational and Mathematical Methods in Science and Engineering*, volume 2, pp. 1081-1092. CMMSE, 2009.

[12] C. Ericson, Real Time Collison Detection, Morgan Kaufmann Publishers, pp 525-536, 2005.

[13] X. L. Wu, N. Obeid, and W. M. Hwu, Exploiting More Parallelism from Applications Having Gneralized Reduction on GPU Architectures, In Proceedings of the *10th IEEE International Conference on Computer and Information Technology (CIT 2010)*, pp 1175-1180, Bradford, the UK, July, 2010..

[14] M. Harris, Optimizing Parallel Reduction in CUDA. http://developer.download.nvidia.com/compute/cuda/1_1/Website/projects/reduction/doc/reduction.pdf

저 자 소 개

박 태 정(정회원)
1997년 서울대학교 전기공학부 학사
1999년 서울대학교 전기공학부 석사
2006년 서울대학교 전기컴퓨터공학부 박사
2006년～현재 고려대학교 BK21소프트웨어사업단 연구교수
<주관심분야 : 수치해석, 기하 정보 압축, TCAD, 3D 도형 모델링>

김 창 헌(정회원)-교신저자
1979년 고려대학교 경제학과 학사
1988년 University of Tsukuba 전자정보 박사
1995년 3월～현재 고려대학교 컴퓨터학과 교수
2008년 3월～2010년 2월 한국컴퓨터그래픽스학회 회장
2009년 1월～현재 정보과학회 부회장
<주관심분야 : 컴퓨터그래픽스, 물리기반 시뮬레이션, Mesh Processing>

우 준 명(학생회원)
2007년 서울대학교 전기공학부 학사
2007년～현재 서울대학교 전기컴퓨터공학부 박사과정.
<주관심분야 : 반도체소자, 바이오소자 모델링>