韓國컴퓨터情報學會 論文誌
第16卷 第2號, 2011. 2.

2011-16-2-1-2

CUDA 기반 GPU에서 효율적인 Power Method의 구현

김 정 환*, 김 진 수*

Implementation of Efficient Power Method on CUDA GPU

Junghwan Kim *, Jinsoo Kim*

요 약

GPU는 저렴한 비용으로 쉽게 대규모 데이터 병렬성을 활용할 수 있는 장점을 갖고 있어 많은 고성능 컴퓨팅 응용 분야에서 사용되고 있는 추세다. 행렬의 고유벡터를 구하는 power method는 웹 페이지의 중요도를 계산하는 PageRank 알고리즘 등 여러 응용 분야에서 활용되고 있는 방법으로써, 본 연구에서는 power method를 GPU에서 병렬화하여 구현하였으며, 성능을 최적화하기 위한 개선 방법을 제시하였다. Power method는 행렬과 벡터의 곱셈 연산이 반복적으로 수행되며 GPU에서 쉽게 병렬화가 가능하다. 그러나, 고유벡터의 수렴 여부 판단을 위한 연산 등의 작업과 다음 곱셈을 위한 벡터 크기의 조정 등의 작업이 부가적으로 필요하며, 이러한 작업은 GPU 내의 커널 코드를 여러 차례 호출하고 불필요한 데이터 이동을 유발하는 문제점이 있다. 본 연구에서는 커널 호출 회수를 줄이고 스레드 배치를 최적함과 동시에 수렴 여부 판단을 위한 연산을 최적함으로써 power method의 성능을 향상시켰다.

▸ Keyword : 파워 메소드, GPU 컴퓨팅, GPGPU, 병렬컴퓨팅, CUDA

Abstract

GPU computing is emerging in high performance application area since it can easily exploit massive parallelism in a way of cost-effective computing. The power method which finds the eigen vector of a given matrix is widely used in various applications such as PageRank for calculating importance of web pages. In this research we made the power method efficiently parallelized on GPU and also suggested how it can be improved to enhance its performance. The power method mainly consists of matrix-vector product and it can be easily parallelized. However, it should decide the convergence of the eigen vector and need scaling of the vector subsequently. Such operations incur several calls to GPU kernels and data movement between host and GPU memories. We improved the performance of the power method by means of reduced calls to GPU kernels, optimized thread allocation and enhanced decision operation for the convergence.

▸ Keyword : power method, GPU computing, GPGPU(General Purpose Computing on GPU), parallel computing, CUDA(Compute Unified Device Architecture)

• 제1저자, 교신저자 : 김정환
• 투고일 : 2010. 11. 19, 심사일 : 2010. 12. 21, 게재확정일 : 2010. 12. 27.
* 건국대학교 컴퓨터응용과학부 교수(Dept. of Computer Science, Konkuk University)
※ 이 논문은 2008년도 건국대학교 학술진흥연구비 지원에 의한 논문임.

Ⅰ. 서 론

최근 고성능 컴퓨팅 분야에서 GPU(Graphic Processing Unit)를 활용하는 사례가 점점 늘고, 이에 대한 연구가 활발히 진행되는 추세이다. GPU는 원래 고속의 그래픽 처리를 위해 사용되는 장치이지만, 다수의 병렬 연산 장치들을 포함하고 있어 고성능을 요구하는 분야에서의 활용이 주목받고 있다[1].

GPU의 범용적인 활용이 주목받게 된 배경으로 기존 GPU 프로그래밍보다 훨씬 용이한 프로그래밍 모델의 출현을 들 수 있다. NVIDIA사에서 나온 CUDA(Compute Unified Device Architecture) 플랫폼도 그러한 예라고 할 수 있다[2]. CUDA는 기존 C언어를 확장하여 사용함으로써, 기존 프로그래머가 쉽게 적응할 수 있게 해준다. CUDA에서 병렬성은 스레드에 의해 표현되는데, 스레드 간의 복잡한 동기화가 존재하지 않아 비교적 용이하게 기존 프로그램을 병렬화할 수 있다.

NVIDIA 사의 GPU는 그림 1과 같이 여러 개의 "멀티프로세서"로 구성되어 있으며, 그 개수는 기종에 따라 수십 개에 이른다. 각 멀티프로세서는 8개의 스레드프로세서(또는 스트림프로세서)로 구성되어 전체적으로 병렬처리가 가능한 스레드프로세서의 총 개수는 수백 개에 이르기도 한다.

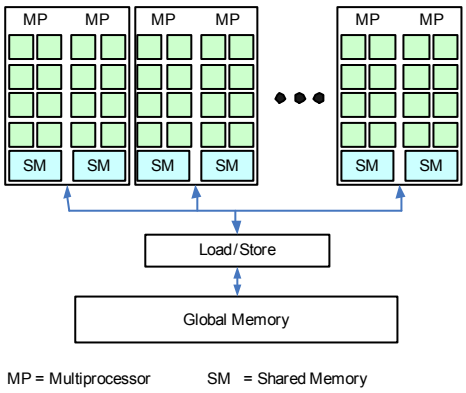


그림 1. NVIDIA GPU 구조
Fig. 1. Architecture of NVIDIA GPU

GPU에서 수행되는 코드는 커널(kernel) 코드라고 불리는데, 보통의 C언어를 통해 작성되며, 각각의 명령어는 여러 개의 스레드프로세서에 의해 병렬적으로 수행되어 SIMT(Single Intruction Multiple Threads)라고 불린다. SIMT는 SIMD(Single Instruction Multiple Data)와 유사한 개념이며, 대규모의 데이터 병렬성을 활용할 수 있게 해준다. 커널 코드는 동시에 수많은 스레드의 의해 서로 다른 데이터에 대해 수행된다[3].

GPGPU(General Purpose Computing on GPU)로 상징되는 GPU의 활용은 유체 역학[4], 분자동력학[5], 수치선형대수[6], 의료 영상[7, 8], 자원 탐사 등 다양한 분야에서 활발히 진행되고 있다.

한편, 행렬의 고유벡터(eigen vector)와 고유치(eigen value)를 계산하는 문제는 다양한 분야에서 다루어지고 있다. 가령, 웹 기반 시스템의 이상 탐지[9], 검색 엔진에서 웹 페이지의 중요도 산정을 위한 PageRank 알고리즘[10, 11] 등에서 필요로 한다. 이러한 문제들은 행렬-벡터 곱셈 연산의 반복적인 연산을 수행하는 power method를 통해 해결할 수 있다.

Power method는 기존에 공유메모리 기반 또는 MPI와 같은 메시지 전달 기반의 프로그래밍 모델에서 병렬화 방법이 연구되어 왔지만[12], GPU를 활용한 사례는 아직 없다. 본 연구에서는 CUDA 기반의 GPU를 사용함으로써, 거대 희소행렬(sparse matrix)에 대한 power method를 적은 비용으로 빠르게 수행하기 위한 방법을 제시한다.

본 논문의 2절에서는 power method 알고리즘과 이를 구성하는 각각의 요소들을 설명하고 병렬화에 대해 기술한다. 3절에서는 각 요소들을 효과적으로 결합함으로써 성능을 개선하는 방법에 대해 설명한다. 여기에서는 단순 병렬화로부터 성능 향상을 위한 방법을 단계적으로 기술한다. 4절에서는 각 향상 기법들에 대해 성능 평가 및 비교하며, 마지막으로 5절에서 결론을 제시한다.

Ⅱ. Power Method의 병렬화

1. CUDA 플랫폼

CUDA(Compute Unified Device Architecture)는 NVIDIA사 GPU에서 대규모 병렬 컴퓨팅을 하기 위해 개발된 소프트웨어 플랫폼이다. 여기에는 기존 C언어에 대한 몇 가지 확장과 GPU 디바이스를 접근하기 위한 함수들의 라이브러리가 포함되어 있다.

병렬성은 스레드에 의해 표현되며, 각 스레드는 자신의 스레드 ID를 이용하여 서로 다른 데이터에 접근한다. 여러 개의 스레드는 블록 단위로 묶여 관리되는데, 하나의 스레드 블록

은 최대 512개의 스레드를 가질 수 있다. 하나의 블록에 포함된 스레드들 간에는 __syncthreads()를 통해 배리어(barrier) 형태의 동기화를 수행할 수 있으며, 그림 1의 Shared Memory에 접근할 수 있다. 블록 내의 스레드들은 Shared Memory를 통해 데이터를 공유할 수 있을 뿐 아니라, 캐쉬와 같은 매우 빠른 접근이 가능하다. Global Memory는 상대적으로 느리지만, 여러 블록들이 함께 사용할 수 있다. 블록 간의 동기화 방법은 없으며, 필요한 경우 두 개의 커널 함수로 나누어 수행해야 한다. 이때 두 커널 함수 호출 사이에 묵시적인 동기화가 이루어진다.

CUDA는 사용자가 기존의 C 프로그램을 확장, 수정함으로써 병렬화를 이루도록 하며, 스레드들의 스케줄링이나 동기화에 대해 신경쓸 필요가 없도록 한다. 올바른 CUDA 코드는 GPU 내부 구조에 대해 사용자가 고려하지 않아도 얻어지지만, 좋은 성능을 얻기 위해서는 Global Memory, Shared Memory, 레지스터 등의 메모리 계층과 워프(warp) 단위의 스케줄링 등 내부적인 것을 고려해야 한다[3].

2. Power Method 알고리즘

Power method는 행렬에 대한 가장 큰 고유치(eigenvalue)와 고유벡터(eigenvector)를 구하는 방법으로 행렬-벡터 곱셈을 반복적으로 수행한다[13].

그림 2는 power method 알고리즘을 보여준다. 매 반복에서 행렬-벡터 곱셈의 결과 벡터 y는 무한히 커지거나 작아지는 문제를 피하기 위해 $|y_j| = \|y\|_\infty$인 y_j에 대해 $y = y / y_j$ 를 수행한다. $\|y\|_\infty$는 최대 노옴(maximum norm)으로 $y = [y_1, y_2, ..., y_m]^T$ 일 때 다음과 같은 정의를 갖는다.

$$\|y\|_\infty = \max\{|y_1|, |y_2|, \cdots, |y_m|\}$$

알고리즘에서 MaxElem()는 그러한 y_j를 찾기 위한 함수이다. 반복이 끝났을 때 해당 값은 고유치가 된다. 또한 그 때의 x 벡터가 고유벡터가 된다.

알고리즘의 수렴 여부의 결정은 매 반복의 고유치 차이가 일정 기준 이하(τ)인지를 판별하는 것과 고유벡터의 차이가 일정 기준 이하인지를 판별하는 것이 있다. 고유치가 수렴하더라도 고유벡터는 수렴하지 않을 수 있으므로 이 알고리즘에서는 고유벡터의 수렴 여부를 판별한다(알고리즘에서 6번 문장). 수렴하지 않을 경우에는 무한히 반복될 수 있으므로 최대 반복 회수를 둔다.

```
PowerMethod(A, λ, x)
Input: matrix A
Output: λ and vector x (eigenvalue and eigenvector)
1    x ← [1.0, 1.0, ..., 1.0]^T;
2    for k ← 1, 2, ..., MAX_ITERS do
3        y ← Ax;
4        λ ← MaxElem(y);
5        y ← y / λ;
6        if ( |MaxElem(y - x)| < τ)
7            return success;
8        x ← y;
9    return failure;
```

그림 2. Power method 알고리즘
Fig. 2. Power Method Algorithm

3. 희소 행렬-벡터의 곱셈

y = Ax 형태의 행렬-벡터의 곱셈은 power method에서 가장 중심적인 역할을 하는 계산 작업이다(그림 2 알고리즘에서 3번 문장). 특히 본 연구에서는 희소 행렬(sparse matrix)을 사용하며, 이를 위한 데이터 포맷으로 CRS(Compressed Row Storage)가 이용되었다[14, 15]. 그림 3은 CRS 형태로 희소 행렬을 표현한 예를 보여준다.

0	0	3	9	0
0	5	0	0	0
0	0	2	0	0
0	0	0	0	8
1	7	0	0	0

<예시 행렬>

val = {3, 9, 5, 2, 8, 1, 7}
col_idx = {2, 3, 1, 3, 4, 0, 1}
row_ptr = {0, 2, 3, 4, 5, 7}

<CRS>

그림 3. 희소 행렬의 CRS 표현
Fig. 3. CRS Format of a Sparse Matrix

CRS 표현에서는 3개의 배열이 사용된다. 이들은 0이 아닌 요소 데이터를 저장하는 val 배열, 요소 데이터의 열 인덱스 값을 저장하는 col_idx 배열, 그리고 val 및 col_idx에서 각 행의 시작 위치를 나타내는 row_ptr 배열이다.

그림 4는 희소 행렬과 벡터 x의 곱 연산을 위한 커널 코드를 보여준다. 희소 행렬은 앞서 언급한 CRS 표현에 따라 val, col_idx, row_ptr에 저장되어 있다. 하나의 스레드는 희소 행렬의 한 개 행을 담당하며, 스레드가 담당한 행의 인덱스는 스레드 ID와 블록 ID인 tx와 bx에 의해 결정된다. blockDim.x는 블록의 크기를 의미한다. 각 행의 연산은 독립적으로 수행 가능하므로 스레드 간의 동기화는 필요치 않다.

```
i = bx * blockDim.x + tx;
if (i < matsize) {
  ysub = 0.0;
for (k = row_ptr[i]; k < row_ptr[i + 1]; k++)
  ysub = ysub + val[k] * x[col_idx[k]];
  y[i] = ysub;
  }
```

그림 4. 희소 행렬-벡터 곱
Fig. 4. Sparse Matrix-Vector Product

4. Reduction

Reduction 연산은 벡터에 대해 한 개의 스칼라 결과를 갖는다. 가령, 벡터 요소의 총 합을 구하는 일이나 또는 벡터 요소 중에서 가장 큰 값을 찾는 일 등이다. 순차 실행에서는 $O(m)$의 시간이 소요되지만, 병렬 실행의 경우 $\log_2 m$ 단계를 거쳐 완료될 수 있다. Power method에서는 그림 2 알고리즘에서 4번 또는 6번 문장과 같이 벡터에서 최대값을 갖는 요소를 찾기 위해 reduction이 필요하다.

그림 5는 GPU에서 reduction 연산을 수행하는 예를 보여준다. 매 단계마다 동기화가 필요하지만, 블록 간에는 동기화가 지원되지 않으므로 블록 내에서의 최종 결과가 나오면 커널을 종료한다. 커널에서 호스트 코드로 복귀하면 커널 내의 모든 블록이 종료되었다는 묵시적인 동기화가 이루어진다. 이후 각 블록 결과들을 새로운 입력으로 하여 동일한 커널 코드를 반복적으로 다시 호출한다.

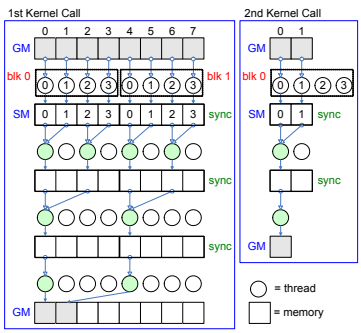


그림 5. Shared Memory를 이용한 reduction 수행 과정
Fig. 5. Reduction Using Shared Memory

5. AXPY

AXPY는 aX +Y 형태(a는 스칼라)의 벡터 연산으로 벡터의 모든 요소에 대해 독립적으로 동일한 연산이 수행된다. 그림 2 power method 알고리즘의 5번 문장에서 **(1/λ)x + 0** 이나, 6번 문장에서 **y - x**와 같은 것이 AXPY 형태라고 할 수 있다. 그림 6은 GPU에서의 AXPY 연산 코드 부분을 보여준다.

```
i = bx * blockDim.x + tx;
if (i < vectsize) {
z[i] = alpha * x[i] + y[i];
  }
```

그림 6. AXPY
Fig. 6. AXPY

하나의 스레드는 벡터의 한 요소에 대해 연산을 수행하며, 이때 해당 요소는 블록 ID인 bx와 스레드 ID인 tx에 의해 결정된다. 모든 요소는 독립적으로 계산되므로 스레드 간의 동기화는 필요치 않다.

III. 성능 개선

1. 단순 병렬화

병렬성을 가장 많이 내포한 부분은 행렬-벡터 곱을 계산하는 y = Ax 부분일 것이다. 이 부분만을 II.3절에서 언급한 GPU 커널 코드와 같이 병렬화한다. 곱셈 결과인 y벡터는 GPU에서 호스트 컴퓨터의 메모리로 옮겨진 후, 그림 2 알고리즘의 4~8번 문장을 호스트 컴퓨터에서 순차적으로 실행한다.

2. 완전 병렬화

그림 2의 power method 알고리즘을 병렬화하기 위해 ① 행렬-벡터 곱셈 연산, ② reduction, ③ AXPY 연산을 모두 II절에서 언급한 것처럼 개별적으로 병렬화하여 이용한다. 행렬-벡터 곱셈 부분뿐 아니라, 알고리즘의 4~8번 문장 모두에 대해 가능한 한 GPU에서 수행될 수 있도록 한다. 이 경우, 병렬 수행을 통해 수행 시간을 단축시킬 수 있을 뿐 아니라, GPU와 호스트 컴퓨터 사이의 데이터 이동이 줄어들어 수행 시간을 단축시킬 수 있다. 그러나, 고유치 λ와 고유벡터의 수렴을 판별하기 위한 $\|y-x\|_\infty$ 값의 호스트메모리 이동은 피할 수 없다.

그림 7은 GPU 커널 함수를 호출하도록 그림 2 알고리즘의 3~8번 문장 부분을 수정한 것을 의사 코드로 나타낸 것이다. SpMV_GPU()는 희소행렬-벡터 곱셈을, MaxElem_GPU()는 최대 요소를 찾는 연산을, 그리고 DAXPY_GPU()는 AXPY의 배정도 수행을 각각 GPU에서 수행하는 커널 함수를 의미한다. MaxElem_GPU()는 II.4절에서 언급한 reduction 연산을 병렬로 수행한다.

```
3    SpMV_GPU(y, A, x);                    // y ← Ax
4    MaxElem_GPU(L, y);                    // L ← MaxElem(y)
5    CopyFromGPU(λ, L);                    // λ ← L
6    DAXPY_GPU(y, 1/λ, y, 0.0);            // y ← 1/λ*y +0.0
7    DAXPY_GPU(z, -1.0, x, y);             // z ← (-1.0)*x + y
8    MaxElem_GPU(e, z);                    // e ← MaxElem(z)
9    CopyFromGPU(err, e);                  // err ← e
10   if ( |err| < τ)
11   CopyFromGPU(x, x);                    // x ← x
12   return success;                       // with λ and x
13   x ← y;
```

그림 7. Power method의 완전 병렬화
Fig. 7. Fully Parallelization of Power Method

```
2    for k ← 1, 2, ..., MAX_ITERS do
3    Step1(y, A, x, mlist);
4    Step2(L, mlist);
5    CopyFromGPU(λ, L);                    // λ ← L
6    m = 1/λ;
7    Step3(m, x, y, τ, b);
8    CopyFromGPU(b, b);                    // b ← b
9    if ( !b )
10   CopyFromGPU(x, x);                    // x ← x
11   return success;                       // with λ and x
12   x ← y;
```

그림 8. Power method 최적화
Fig. 8. Optimized Power Method

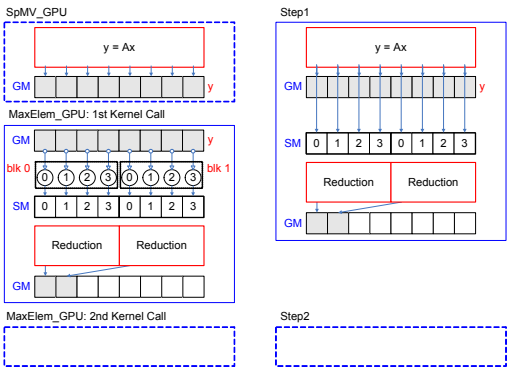
3. 최적화

그림 7의 SpMV_GPU(), MaxElem_GPU(), DAXPY_GPU() 등은 GPU에서 수행됨으로써 수행 시간을 단축시키긴 하지만, 이들 커널 코드들이 나뉘어져 있음으로 인해 단점을 갖고 있다. 첫째, 불필요한 GPU 메모리 내에서의 데이터 이동을 유발하고, 둘째, 여러 차례 커널 호출에 따른 오버헤드, 그리고 임시 벡터 z를 위한 GPU 내의 추가 메모리 소요 등의 문제를 갖고 있다.

이러한 문제를 해소하고 성능을 보다 향상시키기 위한 방안으로써, 본 연구에서는 전체 커널 함수 호출을 세 부분으로 합치도록 구현하였다.

그림 8은 Step1(), Step2(), Step3()의 3개 커널 함수에 대한 호출로 구성되어 있다. 각 커널 함수가 그림 7에서 해당하는 부분은 다음과 같다. Step1()은 문장 3과 문장 4 전반부, Step2()는 문장 4 후반부까지, Step3()은 문장 6~9에 각각 해당된다. 그림 7에서 MaxElem_GPU()는 사실 한 번의 커널 호출로 되지 않는다. 이는 II.4절에서 언급한 것처럼 스레드 블록 간에는 동기화가 되지 않기 때문이다. Step1()은 여기서 첫 번째 커널 호출까지 만 포함하며, 나머지는 Step2()에서 수행한다.

그림 9의 Step1()은 한 번의 커널 호출로 행렬-벡터 곱셈을 수행하고 또한 결과로 나온 y벡터에 대해 최대 요소를 구하기 위한 앞 부분의 단계들을 수행한다. 이렇게 함으로써, 커널 호출 회수가 줄어들 뿐 아니라, 결과 벡터 y의 요소 값들을 지연시간이 긴 Global Memory에서 다시 읽는 대신 레지스터에 들어있는 값에서 바로 읽어 처리하는 이점이 있다. 읽어들인 값은 Shared Memory에 저장한 후, reduction의 앞 단계를 수행한다. reduction은 Step2()에서 수행하는 코드와 같다.



(a) 최적화 이전 (b) 최적화 이후

그림 9. Step1( ) 최적화
Fig. 9. Optimized Step1()

Step2()는 Step1()의 결과로 얻어진 배열을 입력으로 받아 최대 요소를 찾기 위한 reduction을 수행한다. 최종 결과가 1개 남을 때까지 그림 10과 같은 커널이 반복적으로 수행된다.

GPU에 있는 각 멀티프로세서에는 스레드들이 워프(warp) 단위로 스케줄링된다. 따라서 워프 크기 32이므로

32개의 스레드가 하나의 수행단위가 된다. reduction을 수행할 때 매 단계마다 수행되는 스레드는 1/2씩 줄어들게 되며, 이때 각 단계에서 수행되는 스레드들이 여러 워프에 있다면, 스레드 프로세서의 이용률이 떨어지게 될 것이다. 그림 10에서 제시한 스레드 선택은 워프의 이용을 최대화하도록 고려한 것이다.

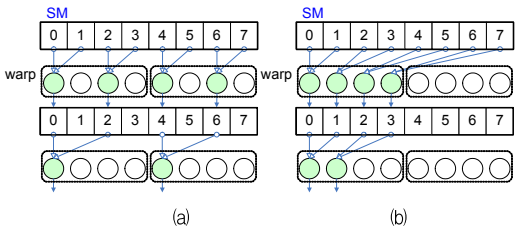


그림 10. Reduction 최적화
Fig. 10. Optimized Reduction

마지막으로 그림 11에 제시된 Step3()는 새로운 y벡터를 계산하고, 수렴 여부를 판단하는 부분이다. 주목할 점은 앞서 그림 6의 z벡터가 필요 없어졌고, 수렴 여부를 계산하기 위해 두 벡터의 차이에 대한 최대 노옴을 구할 필요가 없다는 점이다. $|y_i - x_i| > \tau$인 i번째 요소가 있는지를 알면 충분하지, 굳이 $\|y-x\|_\infty$를 계산하고 이 값이 τ보다 큰지 알 필요가 없다. 만일 그러한 항이 존재한다면 b의 값은 true가 되고, 이는 수렴하지 않았다는 의미이다. 이와 같은 방식은 reduction을 위한 커널 함수의 반복적인 호출을 제거함으로써, 단지 한번의 호출로 결과를 산출하게 해준다.

```
Step3(m, size, x, y, tolerance, B)

    i = (bx * blockDim.x + tx);
        if (i < size) {
        res = y[i]  * (m);
            y[i] = res;

    if (fabs(res - x[i]) > tolerance)
                *B = 1;
                }
```

그림 11. Step3() 최적화
Fig. 11. Optimized Step3()

Ⅳ. 성능 평가

성능 평가에는 NVIDIA의 GTX280이 장착된 리눅스 서버가 사용되었으며, 구체적인 사양은 다음과 같다. GTX280은 30개의 멀티프로세서와 1GB의 Global Memory를 갖고 있다. 스레드프로세서 코어는 30×8개이며, 블록 당 Shared Memory는 16KB, 블록 당 최대 레지스터 수는 16K개다. 리눅스 서버는 Intel Xeon 2.66GHz 쿼드 코어와 8GB의 메모리가 사용되었다.

데이터의 정밀도는 모두 배정도(double precision)가 사용되었다. 실험에 사용된 행렬은 정방행렬로 한 차원의 크기가100000, 200000, 500000, 1000000, 2000000의 총 5가지가 사용되었다. 또한 0이 아닌 값의 존재 비율에 따라 각 종류별로 8개의 다양한 행렬들을 랜덤하게 발생시켰다. 결과적으로 40개의 희소행렬이 실험에 사용되었다.

그림 12는 행렬 크기별로 호스트컴퓨터 대비 속도 향상을 보여준다. 그림에서 HOST는 호스트컴퓨터, G-NAIVE는 단순 병렬, G-FULL은 완전 병렬, G-OPT는 최적화를 각각 의미한다. 그림은 HOST < G-NAIVE < G-FULL < G-OPT 순으로 성능이 좋음을 보여준다. 같은 행렬 크기라도 0이 아닌 값의 비율에 따라 속도 향상 정도는 다른데, 0이 아닌 값이 적게 있을 수록 G-OPT의 성능은 우수하다.

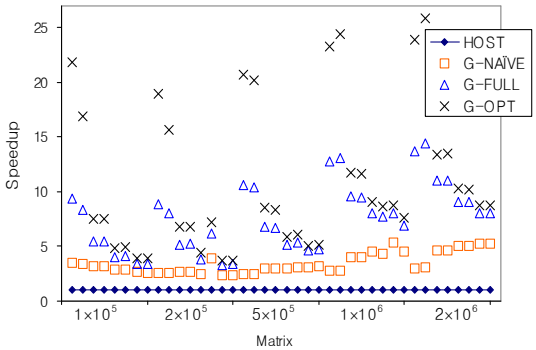


그림 12. 성능 개선 비교
Fig. 12. Comparison of Performance

그림 13은 행렬에 포함된 0이 아닌 값의 개수에 따라 각 버전의 속도 향상을 보여준다. 실험에 사용된 행렬의 크기는 2000000×2000000이다.

x축의 첫 번째 항목은 0이 아닌 값의 개수가 가장 작은 경우로 G-OPT 버전의 속도 향상이 23.9로써 다른 것들에 비해 가장 좋다. 0이 아닌 값의 개수가 늘어남에 따라 세 버전의 격차는 점점 줄어든다. 이는 행렬-벡터 곱셈 부분에 대해 성능 개선을 한 것이 아니라, 다른 부분들에 대해 한 것이기 때문에 행렬-벡터 곱셈에 소요되는 시간이 늘어날수록 성능 향상의 격차는 줄어들게 된다. 하지만 희소성 관점에서 보면, 이 실험 결과는 희소성의 정도가 커질수록 성능의 향상도는 증가하고 있음을 시사하고 있다.

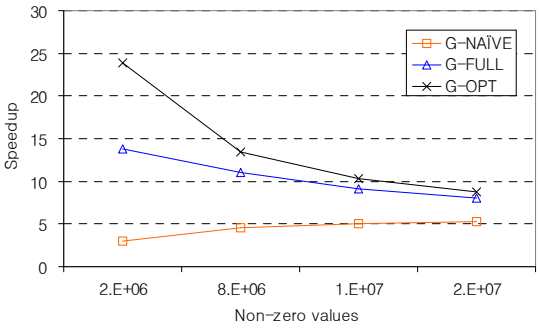


그림 13. 0이 아닌 값의 개수에 따른 성능 개선 정도
Fig. 13. Performance According to Non-zeros

그림 14는 행렬-벡터 곱셈 연산(SpMV), 최대값을 찾는 reduction 연산(MAX), 새로운 x벡터의 준비 및 수렴 판단 등 기타 부분(NewX+Stop)에 소요되는 시간을 보여준다. G-OPT는 SpMV와 MAX를 따로 측정할 수 없으므로 두 부분을 합친 소요시간(SpMV+MAX)을 표시하였다.

그림 14에서 G-OPT의 SpMV+MAX는 10.9ms로써, G-FULL의 SpMV와 MAX를 합한 시간 12.2ms보다 향상됨을 보여준다. 이는 G-OPT가 행렬-벡터 곱셈과 reduction의 앞부분을 하나의 커널에서 수행함으로써 데이터 이동과 커널 호출 오버헤드를 최소화한데 기인한다.

그 외 나머지 시간 NewX+Stop의 경우도 G-FULL이 2.0ms인데 비해 G-OPT는 0.4ms로 단축되었다. 이는 메모리 이동의 최소화와 함께 수렴 판단 부분에서 reduction을 제거함으로써 개선된 효과이다.

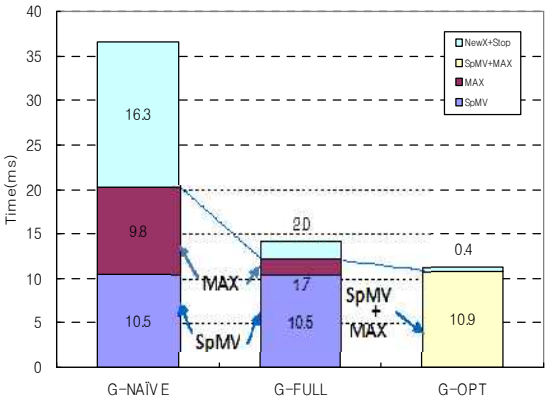


그림 14. 부분별 소요시간
Fig. 14. Elapsed Time for Each Component

V. 결 론

고성능 컴퓨팅 분야에서 GPU는 저비용과 용이성으로 인해 점점 활용 사례가 늘고 있는 추세이다. NVIDIA사의 CUDA 플랫폼은 C언어를 확장함으로써, 기존 프로그래밍 언어 사용자가 매우 쉽게 적응할 수 있게 해준다. 또한 매우 제한적인 동기화만 제공함으로써, 병렬 프로그래밍의 어려운 요소들을 제거하였다.

CUDA 플랫폼에서 프로그래머는 쉽게 올바른 동작을 하는 병렬 프로그램을 작성할 수 있지만, 보다 좋은 성능을 얻기 위해서는 여러 가지 요소들을 고려해야 한다. 가령, GPU가 갖고 있는 고유 구조와 프로세서의 개수, 스레드가 프로세서에 할당되는 상황 등에 대한 이해가 필요하며, 특히 메모리 계층 간의 특성과 데이터 배치에 세심한 고려가 필요하다.

본 연구에서는 여러 응용 분야에서 사용되는 고유치와 고유벡터를 구하기 위한 power method를 GPU에서 구현하고 이를 개선하였다. Power method를 구성하는 요소인 행렬-벡터 곱셈 연산, reduction, AXPY 연산 등은 GPU에서 쉽게 병렬화하여 구현할 수 있는 부분들이지만, 여기에서는 보다 효과적으로 이들을 개선하여 보았다.

먼저 reduction의 경우, 한 번의 커널 수행으로 해결되지 않는다. 이는 스레드 블록간의 동기화가 지원되지 않는 점에 기인한다. 여러 번의 커널 수행을 줄이기 위해 첫 번째 커널 수행을 행렬-벡터 곱셈 연산을 수행하는 커널에서 같이 이루어지게 결합하였다.

다음으로 reduction 과정에서 워프 크기를 고려하여 스레드 배치를 함으로써 프로세서의 이용률을 극대화할 수 있도록 하였다.

마지막으로 수렴 여부를 판단하기 위해 노옴을 계산할 필요를 제거하였다. 노옴의 계산 과정은 최대값을 찾는 reduction 과정을 수반하지만, 개선된 코드에서는 결과적으로 이러한 reduction을 제거할 수 있었다.

단순 병렬화를 수행한 버전의 성능이 약 3.0~5.2 배 향상된 데 비해, 위와 같은 최적화 과정을 통해 개선된 버전은 8.8~23.9 배의 향상을 보였다. 희소 행렬에서 0이 아닌 데이터의 밀도가 커질 경우 power method에서 성능에 영향을 가장미치는 요소는 행렬-벡터 곱셈 연산이 되며, 행렬-벡터 곱셈 연산의 최적화는 향후 연구 과제로 남는다.

참고문헌

[1] John Nickolls and William J. Dally "The GPU Computing Era," IEEE Micro, Vol. 30, Issue 2, March-April 2010.

[2] Tom R. Halfhill, "Parallel Processing with CUDA," Microprocessor Report, Jan. 2008.

[3] NVIDIA CUDA C Programming Guide, Ver. 3.1.1, Nvidia, July 2010.

[4] T. Brandvik and G. Pullan, "Acceleration of a 3D Euler Solver Using Commodity Graphics Hardware," Proc. 48th AIAA Aerospace Sciences Meeting and Exhibit, AIAA Press, 2008.

[5] J.A. Anderson, C.D. Lorenz and A. Travesset, "General Purpose Molecular Dynamics Simulations Fully Implemented on Graphics Processing Units," J. Computational Physics, Vol. 227, No. 10, May 2008.

[6] S. Ryoo et al., "Optimization Principles and Application Performance Evaluation of a Multithreaded GPU using CUDA,'' Proc. 13th ACM SIGPLAN Symp. Principles and Practice of Parallel Programming, ACM Press, 2008.

[7] S.A. Johnson et al., Apparatus and Method for Imaging Objects with Wavefields, US patent 6,636,584, Patent and Trademark Office, 2003.

[8] ju Hwan Kim, Koojoo Kwon, Byeong-Seok Shin, "Large-Scale Ultrasound Volume Rendering using Bricking",Korea Society of Computer Information,No13(7) pp117-126,Dec. 2008

[9] Chinmay Karande, Kumar Chellapilla and Reid Andersen, "Speeding up Algorithms on Compressed Web Graphs," Proceedings of the Second ACM International Conference on Web Search and Data Mining, 2009.

[10] S. Brin and L. Page, "The Anatomy of a Large-Scale Hypertextual Web Search Engine," Computer Networks and ISDN Systems Vol. 33, No. 3, pp.107-117, 1998.

[11] Tianji Wu, Bo Wang, Yi Shan, Feng Yan, Yu Wang and Ningyi Xu, "Efficient PageRank and SpMV Computation on AMD GPUs," 39th International Conference on Parallel Processing, 2010.

[12] Imran Patel and John R. Gilbert, "An Empirical Study of the Performance and Productivity of Two Parallel Programming Models," IEEE International Symposium on Parallel and Distributed Processing, 2008.

[13] Brian Bradie, A Friendly Introduction to Numerical Analysis, Pearson Prentice Hall, 2006.

[14] J. D. Z. Bai, J. Dongarra, A. Ruhe and H. van der Vorst, "Templates for the solution of algebraic eigenvalue problems: A practical guide," In Society for Industrial and Applied Mathematics, 2000.

[15] Eun-jin Im,"An Efficient Computation of Matrix Triple Products",Korea Society of Computer Information,No11(3) pp141-149,

저 자 소 개

김 정 환

1991년 : 서울대학교 계산통계학과 이학사

1993년 : 서울대학교 대학원 전산과학과 이학석사

1999년 : 서울대학교 대학원 전산과학과 이학박사

1999년 ~ 2000년 : 삼성전자 통신연구소 연구원

2001년 ~ 현재 : 건국대학교 컴퓨터응용과학부 교수

관심분야 : 병렬처리, 컴퓨터네트워크, GPU 컴퓨팅

Email : jhkim@kku.ac.kr

김 진 수

1983년 : 서울대학교 컴퓨터공학과 공학사

1985년 : KAIST 전산학과 공학석사

1998년 : KAIST 전산학과 공학박사

1985년 ~ 2000년 : KT 선임연구원

2000년 ~ 현재 : 건국대학교 컴퓨터응용과학부 교수

관심분야 : 정보통신 네트워크, 병렬처리, 센서네트워크

Email : jinsoo@kku.ac.kr