한국교통대학교 융·복합기술연구소 논문집, Vol. 8, No. 1, pp. 15-20 (2018)
https://doi.org/10.22710/JICT.2018.8.1.015

ISSN : 2233-8667

GPU 기반 행렬 덧셈 및 스칼라 곱셈 알고리즘

박 상 근*

한국교통대학교 기계공학과

Matrix Addition & Scalar Multiplication on the GPU

Sangkun Park*

Department of Mechanical Engineering, Korea National University of Transportation, Daehak-ro 50, Chungju-si, Chungbuk 27469, Korea
(Received 2018.07.20 / Accepted 2018.10.29)

Abstract : Recently a GPU has acquired programmability to perform general purpose computation fast by running thousands of threads concurrently. This paper presents a parallel GPU computation algorithm for dense matrix-matrix addition and scalar multiplication using OpenGL compute shader. It can play a very important role as a fundamental building block for many high-performance computing applications. Experimental results on NVIDIA Quad 4000 show that the proposed algorithm runs 21 times faster than CPU algorithm and achieves performance of 16 GFLOPS in single precision for dense matrices with size 4,096. Such performance proves that our algorithm is practical for real applications.

Key words : GPU algorithm, Matrix addition, OpenGL compute shader

1. 서 론

최근에 출시된 그래픽 카드는 그래픽 처리의 주목적을 넘어서 다양한 공학 분야의 여러 응용 분야에서 새로운 가능성을 제공하고 있다. 2017년 5월에 발표된 엔비디아(NVIDIA)사의 GeForce GTX 1080Ti[1]는 3,584개의 GPU Cores, 11 GB의 비디오 메모리, 484 (GB/s)의 메모리 대역폭, 10,609 GFLOPS(초당 부동소수점 연산 회수)의 계산 성능을 바탕으로 대용량 계산을 필요로 하는 과학적 문제를 해결하고 있으며, 저비용 고효율의 병렬처리 플랫폼으로서 많은 가능성을 보여주고 있다. 본 연구는 이러한 GPU을 활용하여 공학 및 과학 분야에서 컴퓨팅 연산을 위해 가장 많이 사용되고 있는 행렬 연산을 신속히 수행할 수 있는 병렬처리 행렬 덧셈 알고리즘 및 그 계산 성능을 소개하고자 한다.

*Corresponding author, E-mail: skpark@ut.ac.kr

행렬 덧셈은 행렬 곱셈과 더불어 수치 알고리즘을 위한 필수 구성요소로서 계산량 측면에서 큰 부분을 차지하고 있기 때문에 수치 알고리즘의 성능을 좌우한다. 이러한 이유로 대부분의 수치 라이브러리는 보다 빠른 행렬 덧셈 혹은 곱셈을 구현하기 위해 적잖은 시간과 노력을 경주한다. 대표적인 라이브러리로서 BLAS(Basic Linear Algebra Subprograms) 라이브러리가 있으며, 이밖에 인텔(Intel)사의 MKL(Math Kernel Library) 라이브러리, 에이엠디(AMD)사의 ACML(AMD Core Math Library) 라이브러리, 엔디비아(NVIDIA)사의 CUBLAS (CUDA BLAS) 라이브러리 등이 널리 사용되고 있다.

BLAS[2]는 포트란 언어를 사용하여 벡터와 행렬 연산에 필요한 기본 루틴을 제공한다. 즉 BLAS Level 1은 벡터-벡터 연산 루틴을, BLAS Level 2는 행렬-벡터 연산을, BLAS Level 3는 행렬-행렬 연산 루틴을 제공한다. 대표적인 사용 사례로서 흔히 사용되고 있는

선형대수 소프트웨어인 LAPACK이 이 BLAS를 사용하고 있다. 현재 비용없이 다운로드하여 사용할 수 있다.

인텔사의 MKL[3]는 인텔 프로세서에 최적화된 계산용 라이브러리이다. 코어 함수로서 BLAS, LAPACK을 포함하며 FFT, 확률통계, 데이터 피팅 등의 기능을 제공한다. 주요 특징으로 멀티코어 프로세서에 최적화되어 있고, out-of-core 루틴을 제공한다.

ACML[4]는 멀티스레드(multi-threading) 기능을 제공하며, 최근에 발표한 AMD Opteron 프로세서의 장점을 살릴 수 있는 기능 등을 제공한다. 현재 OpenMP을 지원하며 OpenCLTM devices을 위한 특화된 루틴도 제공하고 있다.

CUBLAS[5]는 GPU기반의 계산 루틴을 사용하여 BLAS 라이브러리를 가속화시킨 라이브러리이다. 최근에 GPU 뿐만 아니라 CPU에서도 동적으로 BLAS 호출이 가능하도록 지원함으로써 BLAS Level 3을 더욱 가속화시킨 NVBLAS 라이브러리가 출시되었다. CUBLAS의 주요 특징으로 응용 프로그램 개발을 지원하기 위해 벡터 및 행렬 데이터를 GPU 메모리에 올리고, 계산 결과를 다시 CPU host 메모리로 가져올 수 있는 CUBLAS API 함수를 제공한다. 단점으로 열-우선(column-major) 방식의 스토리지 구조를 사용하는데, C/C++의 행-우선(row-major) 스토리지 구조와 맞지 않아 기존 C/C++로 작성된 응용 프로그램에서 사용하기에 다소 불편함이 있다. 또한 NVIDIA사의 그래픽 카드에서만 모든 기능이 지원된다.

2. 병렬 계산

2.1 컴퓨터 쉐이더

컴퓨터 쉐이더(compute shader)[6]는 OpenGL이 제공하는 shader 중의 하나로서 일반적인 그래픽 렌더링 이외의 병렬처리 계산 작업에 GPU가 사용될 수 있도록 GPGPU(General-Purpose computing on Graphics Processing Units) 프로그래밍을 지원하는 범용 shader를 말한다. 현재 GPGPU 프로그래밍을 지원하는 가장 알려진 기술로 엔디비아사의 CUDA 및 AMD에서 추진하는 OpenCL이 있는데, 이들 모두 GPU기반 병렬처리 부동 소수점 계산을 위한 개발 환경을 지원한다. 그러나 오직 병렬처리 계산만을 위한 용도로 사용할 수 있고 CUDA의 경우 자사의 그래픽 카드에서만 작동하는 단점을 가지고 있다. 이에 비해 컴퓨터 쉐이더는 일반적인 그래픽 렌더링 및 이미지 프로세싱 등의 작업에서 직접 사용 가능하며 이들이 사용하는 데이터 리소스(resources) 등을 공유할 수 있어 이들이 필요로 하는 대규모 실시간 계산 작업을 가능토록 지원할 수 있다. 또한 특정 회사의 그래픽 카드 의존성이 없고 컴퓨터 OS에 무관하며 모바일(mobile) 환경에서도 어려움 없이 사용 가능하다. 원래 이러한 목적으로 컴퓨터 쉐이더가 처음 등장하였고, 점차 그 사용 영역이 늘어나고 있다. 마이크로소프트사의 Direct3D도 이러한 목적으로 2009년 등장하였고 현재 OpenGL의 컴퓨터 쉐이더와 경쟁하고 있다. 참고로 컴퓨터 쉐이더는 2012년 중반에 OpenGL 버전 4.3의 핵심 부분으로 포함되었다.

2.2 메모리 사용 전략

컴퓨터 쉐이더는 3가지 종류의 메모리가 사용된다.

1) Register memory : Invocation전용의 국부 메모리 (데이터 공유 불가능)
2) Shared memory : 국부 work group내에 존재하는 invocation들 간의 데이터 공유를 위해 지원하는 메모리
3) Global memory : 모든 work group내의 invocation 들 간의 데이터 공유를 위해 지원하는 메모리

메모리의 최대 저장 크기는 사용하는 그래픽 카드마다 다르나, register memory 8,192 ~ 65,536 bytes, shared memory 32,768 ~ 49,152 bytes, global memory 0.5 ~ 24 Gbytes이다. 한편 각 메모리에 데이터를 읽고 쓰는데 소요되는 메모리 접근 시간은 위에서 언급된 순서대로 GPU register가 가장 빠르고, global memory가 가장 느리다. 가장 빠른 GPU register를 최대한 사용하고 가장 느린 global memory를 최소화하면 될 것 같으나 CPU의 데이터가 곧바로 GPU register 혹은 shared memory로 전송되지 않기 때문에 메모리 사용 전략이 필요하다. 더불어 GPU 병렬처리 계산 시 invocation 간의 데이터 공유가 수반되며 따라서 이들 간의 동기화 전략 또한 필요하다. 본 연구에서 구현한 메모리 사용 및 동기화 전략은 다음과 같다.

- Step 1: GPU로 전송된 global memory의 데이터를

shared memory에 복사한다.
- Step 2: 복사된 데이터와 GPU register를 사용하여 병렬처리 계산 후 결과를 shared memory에 저장한다.
- Step 3: 최종 결과를 CPU로 전송하기 위해 shared memory의 데이터를 다시 global memory에 복사한다.

GPU global memory에 전송된 각 입력 데이터를 shared memory를 거치지 않고 직접 global memory에서 읽어올 경우 상당한 메모리 접근 시간이 소요된다. 그러나 각 데이터가 아닌 데이터 집합을 한꺼번에 shared memory에 복사한 후 읽어올 경우 메모리 접근 시간이 상당히 절약된다. 본 연구에서 테스트한 결과 shared memory latency가 global memory latency보다 대략 100배 정도 낮다. 또한 shared memory의 크기가 데이터 크기보다 작을 경우 모든 데이터를 한꺼번에 shared memory에 복사할 수 없고, shared memory 사용 시 bank conflict 문제가 존재한다. 즉 다수 개의 invocation이 서로 다른 bank에 접근할 경우 메모리 접근 시간이 빠르나 만약 겹쳐진다면 충돌이 일어나 오히려 느려지게 된다.

3. 행렬 연산 알고리즘

본 연구에서 제안하는 GPU 병렬처리 행렬 덧셈 및 스칼라 곱셈 알고리즘은 크게 CPU 상에서 OpenGL에게 데이터를 전송하고 병렬 계산을 명령하는 CPU 프로그램(알고리즘 1-1 및 1-2)과 명령에 의한 병렬 계산을 수행하는 GPU 프로그램(알고리즘 2)으로 구성된다.

3.1 전후처리 CPU 프로그램

알고리즘 1-1은 알고리즘 1-2에서 호출되는 함수들로서 CPU에서 GPU로 데이터 전송에 필요한 기본 함수들을 기술하고 있다. 함수 glWriteBuffer()는 행렬 요소 값들을 그래픽 카드에 전송키 위해 비디오 메모리를 할당하는 함수를 설명하고 있고, 함수 glReadBuffer()는 병렬 계산 이후 계산 결과를 CPU로 전송받기 위한 함수를 기술하고 있다. 또한 함수 glCloseBuffer()는 계산을 위해 할당되었던 비디오 메모리를 삭제하는 함수를 나타낸다.

Algorithm 1-1

```
void glWriteBuffer( GLuint buffer, GLintptr offset, GLsizeiptr
size, const GLvoid* data, GLuint index, GLenum usage )
{
 glBindBuffer(GL_SHADER_STORAGE_BUFFER, buffer);
 glBufferData(GL_SHADER_STORAGE_BUFFER, size, data,
usage);
 glBindBufferBase(GL_SHADER_STORAGE_BUFFER,
index, buffer);
}

void glReadBuffer( GLuint buffer, GLintptr offset, GLsizeiptr
size, GLvoid * data )
{
 glBindBuffer(GL_SHADER_STORAGE_BUFFER, buffer);
 glGetBufferSubData(   GL_SHADER_STORAGE_BUFFER,
offset, size, data );
}

void glCloseBuffer( GLuint buffer, GLuint index )
{
 glBindBufferBase(GL_SHADER_STORAGE_BUFFER,
index, 0);
 glDeleteBuffers( 1, &buffer );
}
```

알고르즘 1-2는 알고리즘 1-1의 함수들을 사용하여 GPU에 데이터를 전송하고, 병렬계산에 필요한 파라미터 값들을 설정한 후 GPU 병렬 프로그램을 호출하는 역할을 수행한다. 즉 행렬 데이터를 그래픽 비디오 메모리로 전송하는 부분(Data to GPU), 전송된 데이터를 가지고 병렬처리 계산하는 부분(GPU Processing), 마지막으로 비디오 메모리로부터 결과 데이터를 전송받는 부분(Data from GPU)으로 구성된다.

Algorithm 1-2

```
void add( float a, Matrix& A, float b, Matrix& B, Matrix& R )
{
  // Data to  GPU
 GLuint ssbo_A, ssbo_B, ssbo_R;
 glOpenBuffer(ssbo_A);
 glWriteBuffer(ssbo_A, 0, (A.m*A.n) * sizeof(float),
 &A.v[0], 0, GL_STATIC_DRAW);
 glOpenBuffer(ssbo_B);
 glWriteBuffer(ssbo_B, 0, (B.m*B.n) * sizeof(float),
 &B.v[0], 1, GL_STATIC_DRAW);
 glOpenBuffer(ssbo_R);
```

```
glWriteBuffer(ssbo_R, 0, (R.m*R.n) * sizeof(float),
NULL, 2, GL_STATIC_READ);
glBindBuffer(GL_SHADER_STORAGE_BUFFER, 0);

// GPU Processing
uint BLOCK_SIZE = 512;
uint gsize =  (m*n + BLOCK_SIZE - 1) / BLOCK_SIZE);
glUseProgram(math_add);
  glUniform1ui(glGetUniformLocation(
                math_add, "m"), m);
  glUniform1ui(glGetUniformLocation(
                math_add, "n"), n);
   glUniform1f(glGetUniformLocation(
                math_add, "a"), a);
   glUniform1f(glGetUniformLocation(
                math_add, "b"), b);
   glDispatchCompute(gsize, 1, 1);
glUseProgram(0);

// Data from GPU
glUseProgram(math_add);
glReadBuffer(ssbo_R, 0, (R.m*R.n) * sizeof(float), &R.v[0]);
glUseProgram(0);

glCloseBuffer(ssbo_A, 0);
glCloseBuffer(ssbo_B, 1);
glCloseBuffer(ssbo_R, 2);
}
```

본 알고리즘 코드에서 Matrix A와 Matrix B는 행렬 덧셈을 위한 입력 행렬이고 (행렬의 크기는 m x n) Matrix R은 계산이후 결과 행렬이다. math_add는 사전에 미리 컴파일해 놓은 GPU 프로그램 ID이고, glUniform1ui()와 glGetUniformLocation()는 math_add에서 사용할 uniform 데이터 값을 전송하는 OpenGL함수이다. 그리고 glDispatchCompute()는 상기 입력된 데이터를 가지고 병렬계산을 위한 invocation들을 생성하는 OpenGL 컴퓨터 쉐이더 함수이다. 실제로 이 부분에서 다음 절에 소개할 병렬 계산이 수행된다.

Algorithm 2

```
#version 430 core

#define USE_SSBO_R

layout(local_size_x = 512,  local_size_y = 1,  local_size_z = 1 ) in;
uniform uint m;
uniform uint n;
uniform float a;
uniform float b;

layout(std430, binding=0) buffer ssbo_A { float A[]; };
layout(std430, binding=1) buffer ssbo_B { float B[]; };
#ifdef USE_SSBO_R
layout(std430, binding=2) buffer ssbo_R { float R[]; };
#endif

const uint BLOCK_SIZE = gl_WorkGroupSize.x;

void main()
{
  uint g = gl_GlobalInvocationID.x;
  uint offset =
  gl_WorkGroupSize.x * gl_NumWorkGroups.x;
  uint mn = m*n;
  while( g < mn )
  {
    #ifdef USE_SSBO_R
      R[g] = a*A[g] + b*B[g];
    #else
      A[g] = a*A[g] + b*B[g];
    #endif
    g += offset;
  }
}
```

3.2 본처리 GPU 프로그램

알고리즘 2는 본 연구에서 가장 중요한 부분인 병렬처리 계산 부분이다. m, n은 행렬의 크기(m은 행의 개수, n은 열의 개수)를 나타내며, a, b는 행렬에 곱해지는 스칼라 값으로 a는 행렬 A에 곱해지며, b는 행렬 B에 곱해진다. buffer ssbo_A는 행렬 A의 데이터(행렬 요소 값들)가 저장된 버퍼를 가리키며, 유사하게 buffer ssbo_B는 행렬 B의 데이터가 저장된 버퍼를 가리킨다. 계산 결과는 행렬 R이 가리키는 buffer ssbo_R에 저장되며 알고리즘 1-2의 함수 glReadBuffer()에 의해 CPU로 전송된다. 함수 main()은 본 병렬 계산의 핵심 부분으로 입력된 데이터를 가지고 병렬 계산하는 부분으로 R = a * A + b * B를 수행한다. 매우 간단하며 단순함을 알 수 있다.

4. 수치 실험

본 연구에서 사용한 그래픽 카드는 엔디비아 Quadro 4000이고, 이 그래픽 카드의 비디오 메모리 크기는

2,048 MB, 코어(core) 개수는 256개, 블록당 스레드 개수는 1024개이다. 본 연구에서 사용한 알고리즘 평가 지표는 수행 시간과 유효(effective) GFLOPS이다. 일반적으로 총 수행 시간은 다음과 같이 구성된다.

총 수행 시간(milli-seconds) =
GPU로의 데이터 전송 시간(Data to GPU)
+ GPU에서의 계산 시간(GPU processing)
+ GPU로부터의 데이터 전송 시간(Data from GPU)

즉 CPU RAM에 존재하는 데이터를 GPU로 전송하는데 걸린 시간, 전송된 데이터를 가지고 실제 계산하는데 소요된 시간, 마지막으로 계산 결과를 GPU에서 다시 CPU로 전송하는데 걸린 시간, 이 3개의 시간 총합을 milli-seconds단위로 측정한다.

그리고 알고리즘 계산 성능을 수치로 나타내기 위해 초당 부동소수점 연산 회수를 기가 단위로 표시하는 GFLOPS를 주요 지표로 사용하는데, 본 연구 행렬 덧셈의 경우, 각 행렬의 계수(A의 경우 a, B의 경우 b) 곱셈 연산 횟수가 (m x n)이고, 두 행렬 간의 덧셈 연산 횟수가 (m x n)이므로 아래와 같이 나타낼 수 있다.

$$GFLOPS = (m \times n \times 3) / (10^9 \times \text{수행시간(sec)})$$

위의 GFLOPS 값은 GPU 제조사가 발표하는 최상(peak)의 이론적 수치와 다르며, 어떤 그래픽 카드에서 무슨 메모리를 얼마만큼 사용하여 어떻게 계산했느냐에 따라 다르게 측정된다. 결국, 알고리즘의 객관적 평가를 위해선 동일한 하드웨어가 필수 조건이 된다. 그러나 충분 조건이 되진 않는다. 수행 시간 측정 당시 모든 것이 동일 조건이여야 하는데 이는 불가능한 일이다. 대개 GPU의 수행 시간은 매우 짧기 때문에 약간의 비동일 조건으로 큰 오차가 발생할 수 있다. 그래서 1회 이상의 측정이 필요하다. 본 연구의 경우 10회 측정하여 평균하였다.

한편 본 연구에서는 구현한 GPU 알고리즘의 성능을 평가하기 위해 Table 1과 같이 CPU 상에서의 수행 결과와 비교하였다. 사용한 행렬은 정방 행렬로써 그 크기가 N = 128, 256, 512, 1024, 2048, 4096일 때 각 경우에 대해 CPU-알고리즘과 GPU-알고리즘의 GFLOPS 값을 측정하였고, 상대적 비교를 위해 CPU 대비 GPU의 GFLOPS 값을 속도비로 표시하였다. Fig. 1의 왼쪽

Table 1 Performance comparison of CPU and GPU-algorithm

N x N	CPU		GPU		speedups
	계산 시간 (ms)	GFLOPS	계산 시간 (ms)	GFLOPS	
128 x 128	0.67	0.73	0.01	4.92	7
256 x 256	2.56	0.77	0.02	8.55	11
512 x 512	10.21	0.77	0.08	9.36	12
1024 x 1024	40.77	0.77	0.25	12.43	16
2048 x 2048	174.44	0.72	0.96	13.05	18
4096 x 4096	654.82	0.77	3.12	16.11	21

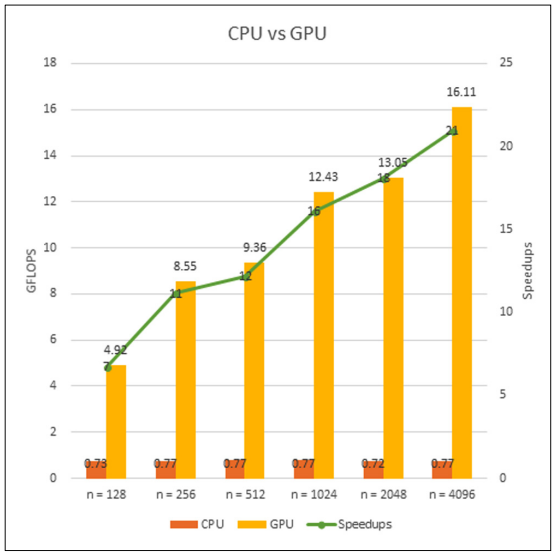

Fig. 1 Performance comparison of CPU-algorithm and GPU-algorithm

축은 GFLOPS 축이고, 오른쪽 축은 속도비 축이다. 예상대로 GPU-알고리즘이 상당히 빠르며, n = 4096일 때 무려 21배의 속도 차이가 나타냄을 확인할 수 있다.

5. 결 론

본 연구는 GPU 상에서 병렬 처리에 의해 수행되는 OpenGL compute shader 행렬 덧셈 및 스칼라 곱셈 알고리즘을 상세히 서술하였고, 수치 실험을 통하여 본 연구 알고리즘의 계산 성능을 기술하였다. 또한 GPU 메모리 구조 및 쉐이더 invocation 등에 관해 서술하였다.

본 연구에서 제시한 GPU 행렬 연산은 CUBLAS에

서 이미 구현된 기술이다. 그러나 OpenGL의 컴퓨터 쉐이더를 사용하여 구현된 사례를 관련 문헌에서 찾아보기 힘들다. 즉 본 연구 알고리즘은 저자의 판단으로 첫 시도이며 컴퓨터 쉐이더를 시작하는 독자에게 참고 문헌이 될 것이다. OpenGL 라이브러리는 CUDA와는 다르게 컴퓨터 OS에 무관하게 인스톨되는 기본 라이브러리이며, 특정 그래픽 카드에 의존성이 전혀 없어 어떤 컴퓨터에서도 사용 가능하다. 따라서 그 활용 가능성은 CUDA보다 높다고 볼 수 있다. 대표적인 활용 분야로서 그래픽 렌더링 분야에서 바로 사용 가능하며, 게임 분야의 물리 엔진에도 기반 기술로서 활용도가 높을 것으로 기대된다.

추후 연구 과제로 본 연구에서 수행한 병렬 처리 방식을 바탕으로 전치 행렬(transposed matrix), 역행렬을 계산하고, 나아가 수치해석 기법으로 널리 알려진 PCG, GMRES 등에 관해 GPU가속화 프로그램을 개발할 예정이다. 현재 사용자의 편의성 및 확장성을 위해 DLL형태의 라이브러리를 개발 중에 있다.

Acknowledgement

이 논문은 2018년 한국교통대학교 지원을 받아 수행하였음.

References

1) https://www.nvidia.com/en-us/geforce/products/10series/geforce-gtx-1080-ti/.
2) http://www.netlib.org/blas.
3) https://software.intel.com/en-us/mkl.
4) http://developer.amd.com/tools-and-sdks/archive/acml-product-features/.
5) https://developer.nvidia.com/cublas.
6) G. Sellers, R. S. Wright, and N. Haemel, OpenGL SuperBible (7th ed.), Addison-Wesley, 2015.