한국교통대학교 융·복합기술연구소 논문집, Vol. 9, No. 1, pp. 1-6 (2019)
https://doi.org/10.22710/JICT.2019.9.1.001

ISSN : 2233-8667

GPU 기반 행렬 곱셈 병렬처리 알고리즘

박 상 근*

한국교통대학교 기계공학

Parallel Algorithm for Matrix-Matrix Multiplication on the GPU

Sangkun Park*

Department of Mechanical Engineering, Korea National University of Transportation, Daehak-ro 50, Chungju-si, Chungbuk 380-702, Korea
(Received 2019.09.26 / Accepted 2019.11.12)

Abstract : Matrix multiplication is a fundamental mathematical operation that has numerous applications across most scientific fields. In this paper, we presents a parallel GPU computation algorithm for dense matrix-matrix multiplication using OpenGL compute shader, which 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 about 208 times faster than previous CPU algorithm and achieves performance of 75 GFLOPS in single precision for dense matrices with matrix size 4,096. Such performance proves that our algorithm is practical for real applications.

Key words : GPU algorithm, Matrix-matrix multiplication, OpenGL compute shader

1. 서 론

GPU(Graphics Processing Unit)는 매우 높은 부동 소수점 처리 성능, 거대한 메모리 대역폭 및 비교적 저렴한 비용으로 일반 목적의 계산을 위한 매우 매력적인 하드웨어 플랫폼으로 발전하였다. 성능, 아키텍처 및 프로그래밍 가능성에서의 GPU 의 급속한 발전은 그래픽 처리의 주요 목적을 넘어서 다양한 분야의 여러 응용 가능성을 제공하게 되었다.

본 연구는 이러한 GPU을 활용하여 공학 및 과학 분야에서 컴퓨팅 연산을 위해 가장 많이 사용되고 있는 행렬 연산을 신속히 수행할 수 있는 병렬처리 행렬 곱셈 알고리즘 및 그 계산 성능을 소개하고자 한다. 행렬 곱셈은 수많은 수치 알고리즘을 위한 필수 구성요소로서 계산량 측면에서 상당 부분을 차지하고 있기 때문에 해당 수치 알고리즘의 성능을 좌우한다. 이러한 이유로 대부분의 수치 라이브러리는 보다 빠른 행렬 곱셈을 구현하기 위해 적잖은 시간과 노력을 경주한다. 대표적인 라이브러리로서 BLAS[1] 라이브러리가 있으며, 이밖에 BLAS 와 유사하나 보다 개선된 형태의 라이브러리로서 Intel사의 MKL[2] 라이브러리, AMD사의 ACML[3] 라이브러리, NVIDIA사의 CUBLAS[4] 라이브러리 등이 많이 사용되고 있다. 이들 라이브러리 중에서 CUBLAS는 CUDA 아키텍처에서 제공하는 GPU 기반 BLAS 라이브러리로서 가장 주목을 받고 있다.

CUBLAS(CUDA BLAS)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[5] 라이브러리가 출시되었다.

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

2. GPU 병렬 계산

2.1 컴퓨터 쉐이더

OpenGL이 제공하는 shader 중의 하나인 컴퓨터 쉐이더(compute shader)[6]는 일반적인 그래픽 렌더링 이외의 병렬처리 계산 작업을 위해 GPU가 사용될 수 있도록 GPGPU(General-Purpose computing on Graphics Processing Units) 프로그래밍을 지원하는 범용 shader이다. 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. 참고로 CUDA의 경우 병렬처리 계산만을 위한 용도로 사용될 수 있고 엔디비아사의 그래픽 카드에서만 작동하는 단점을 가지고 있다. 이것과 비교하여 컴퓨터 쉐이더는 일반적인 그래픽 렌더링 및 이미지 프로세싱 등의 작업에서도 사용 가능하며 이들이 사용하는 데이터 및 관련 리소스(resources) 등을 공유할 수 있기 때문에 대규모 실시간 계산 작업을 가능토록 지원할 수 있다. 또한 특정 그래픽 카드의 의존성이 없고, 컴퓨터 OS에 무관하며, 모바일(mobile) 환경에서도 큰 어려움 없이 사용 가능하다는 특징을 가지고 있어 점차 그 응용 분야가 늘어나고 있다. 마이크로소프트사의 Direct3D도 이러한 목적으로 2009년 등장하였고 현재 OpenGL의 컴퓨터 쉐이더와 경쟁하고 있다.

2.2 GPU 메모리 사용 및 동기화 전략

OpenGL compute shader는 Fig. 1과 같이 3차원 형태의 work group이 생성된 후 수행된다. 즉 CPU 상에서 OpenGL API 함수인 glDispatchCompute() 를 호출하면 OpenGL은 GPU 병렬 처리를 위해 1개의 global work group을 생성하고, 이어서 global work group 내에 다수개의 local work group을 생성한다. 생성되는 local work group의 개수는 예를 들어 Fig. 1과 같이 x축, y축, z축 방향으로 각각 3개, 4개, 8개이고 함수 glDispatchCompute (3, 4, 8)의 입력 인자로서 사용자가 설정할 수 있다. 또한 위의 함수에 의해 생성된 1개의 local work group 은 그 내부에 다시 각 방향으로 다수 개의 work items를 생성한다. 여기서 각 work item은 컴퓨터 쉐이더의 invocation (일종의 thread에 해당)에 의해 처리된다. 각 방향으로의 work item 개수는 다음과 같이 shader 소스 코드 안에 사용자가 직접 입력한다.

```
layout ( local_size_x = 6,
         local_size_y = 4,
         local_size_z = 1) in;
```

위의 layout() 함수 예는 x방향으로 work item이 6개, y방향으로 4개, z방향으로 1개임을 나타낸 다. 즉 1개의 work group 내에 $6 \times 4 \times 1 = 24$개의 work items 혹은 invocation이 생성됨을 의미한다. 여기서 work item의 최대 개수는, 사용하는 그래 픽 카드마다 다르지만, x방향, y방향, z방향으로 각 각 1024개, 1024개, 64개를 보장하며, 세 방향의 총 개수로 1024개를 보장한다. 한편 최적의 local work group 개수 및 work item 개수는 그래픽 카드의 성능 및 해당 어플리케이션의 문제 상황을 고려하여 사용자가 설정해야 한다. 본 연구의 경우 $32 \times 32 \times 1$개의 work item을 사용하였고, local work group 개수는 데이터의 크기 (행렬의 크기)를 work item의 개수로 나눈 값을 사용하였다.

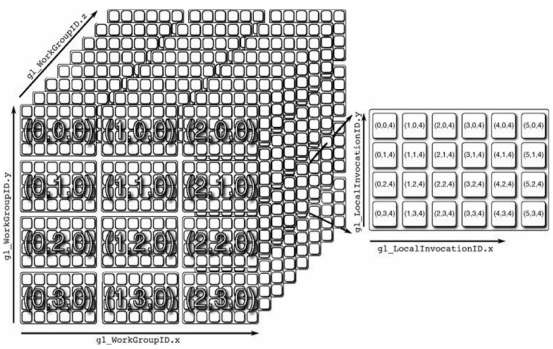

Fig. 1 Compute work group and invocations

한편, 컴퓨터 쉐이더는 3가지 종류의 메모리를 사용한다.

1) Register memory : Invocation전용의 국부 메모리로서 데이터 공유가 불가능하다.
2) Shared memory : local work group내에 존재하는 invocation들 간의 데이터 공유를 위한 메모리이다.
3) Global memory : 모든 work group내의 invocation들 간의 데이터 공유를 위한 메모리이다.

각 메모리 접근 시간을 살펴보면 GPU register memory가 가장 빠르고, global memory가 가장 느리다. 가장 빠른 GPU register memory를 최대한 사용하고 가

장 느린 GPU global memory를 최소화 하면 계산 성능 측면에서 큰 이득을 얻을 것 같으나 CPU에서 생성된 데이터가 곧바로 GPU register 혹은 shared memory로 전송되지 않기 때문에 메모리 사용에 관한 알고리즘이 필요하다. 본 연구에서 구현한 메모리 사용 기본 알고리즘은 다음과 같이 요약할 수 있다.

- Step 1: CPU에서 GPU로 전송된 global memory의 데이터를 shared memory에 복사한다.
- Step 2: shared memory에 복사된 데이터와 GPU register를 사용하여 얻은 데이터를 활용하여 병렬처리 계산을 수행하고 그 결과를 shared memory에 저장한다.
- Step 3: 최종 계산 결과를 CPU로 전송하기 위해 shared memory의 데이터를 global memory에 복사한다.

또한 GPU 병렬처리 계산 시 invocation 간의 데이터 공유가 수반되며 따라서 이들 간의 동기화 알고리즘이 필요하다. 본 연구에서 구현한 두 가지 동기화 알고리즘은 다음과 같이 요약할 수 있다.

1) 계산 작업 순서 동기화 : 수행속도가 느린 어떤 invocation이 memory의 특정 주소에 접근하여 데이터를 읽을 때 수행속도가 빠른 invocation 이 수정해 놓은 상태라면 수행속도가 느린 invocation 은 잘못된 데이터를 가지고 계산을 하게 된다. 이와 같은 상황을 막기 위해 OpenGL 컴퓨터 쉐이더의 barrier() 함수를 사용하여 작업 수행의 동기화 메커니즘을 구현하였다.
2) 메모리 쓰기/읽기 동기화 : 메모리 쓰기 작업이 완료되지 않은 상태에서 읽기 작업이 시작될 수 있다. 쓰기 작업이 완료된 데이터만이 읽혀지도록 보장하기 위해 OpenGL 컴퓨터 쉐이더의 memoryBarrier() 함수를 사용하여 메모리 쓰기와 읽기 순서를 조정하는 메카니즘을 구현하였다.

3. GPU 행렬 곱셈 알고리즘

본 연구에서 제안하는 GPU 병렬처리 행렬 곱셈 알고리즘은 크게 CPU 상에서 OpenGL에게 데이터를 전송하고 병렬 계산을 명령하는 병렬 설정 프로그램(Algorithm 1)과 명령에 의한 병렬 계산을 수행하는 병렬 계산 프로그램(Algorithm 2)으로 구성된다.

3.1 병렬 설정 프로그램

주어진 문제의 데이터 크기와 차원을 고려하여 최적의 work group 개수와 shader invocation 개수를 결정해야 한다. 본 연구에서 사용한 그래픽 카드의 invocation 최대값이 1024이므로 32 × 32 × 1의 shader invocation 개수 (wsize)를 사용하였고, local work group

Algorithm 1: CPU codes for parallel computing

```
void mul( Matrix& A, Matrix& B, Matrix& R )
{
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);

uint BLOCK_SIZE = 32;
uvec3 wsize(BLOCK_SIZE, BLOCK_SIZE, 1);
uvec3 gsize(64, 64, 1);
uvec3 wcnt = (uvec3(l, m, 1) + wsize - uvec3(1)) / wsize;
uvec3 gcnt = (wcnt + gsize - uvec3(1)) / gsize;
uvec3 nelem = wsize * gsize;

glUseProgram(math_mul);
glUniform1ui(glGetUniformLocation(
`math_mul, "m"), m);
glUniform1ui(glGetUniformLocation(
 math_mul, "n"), n);
glUniform1ui(glGetUniformLocation(
 math_mul, "l"), l);
glUseProgram(0);

glUseProgram(math_mul);
for (uint yi = 0; yi < gcnt.y; yi++)
{
 uint y0 = nelem.y * yi;
 for (uint xi = 0; xi < gcnt.x; xi++)
 {
  uint x0 = nelem.x * xi;
  glUniform1ui(glGetUniformLocation(
  math_mul, "x0"), x0);
  glUniform1ui(glGetUniformLocation(
  math_mul, "y0"), y0);
  glDispatchCompute(gsize.x, gsize.y, 1);
 }
}
glUseProgram(0);

glUseProgram(math_mul);
 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);
}
```

개수 (gsize)는 Microsoft TDR (Timeout Detection & Recovery) 문제를 피하기 위해 최대 64 × 64 × 1로 설정하였다. TDR이란 윈도우 운영체제가 일정시간 내에 그래픽 카드의 응답을 받지 못하면 운영체제가 그래픽 카드를 재설정하며 이어서 운영체제의 재부팅이 요구되는 현상을 말한다. 그리고 행렬 크기에 영향을 받지 않기 위해 행렬의 크기가 2048 × 2048 이상인 경우에 부행렬(submatrix)로 분할하고 각 부행렬에 1개의 global work group을 설정한다. 이상의 초기 설정은 Algorithm 1의 위쪽 부분에 해당한다. Algorithm 1의 아래 부분은 수립된 분할 계획에 의해 각 분할 행렬의 병렬 처리 계산을 지시하는 반복 과정을 보여준다. 즉 두 행렬 A와 B의 곱을 R = AB이라 하고 행과 열의 방향을 각각 x, y방 향이라 할 때, 행렬 A는 gcnt.y × 1로 분할되고 B는 1 × gcnt.x로, R은 gcnt.y × gcnt.x로 분할되며 이러한 분할 계획은 OpenGL glDispatchCompute() 함수에 의해 미리 사전에 컴파일한 GPU-프로그램 (Algorithm 2)으로 넘어가게 된다. 여기서 gcnt는 global work group의 개수를 의미한다.

3.2 병렬 계산 프로그램

GPU-프로그램에 입력되는 데이터는 행렬 크기를 나타내는 m, n, l와 부행렬의 시작 위치를 나타내는 (x0, y0)가 있다. 그리고 shared memory에 해당하는 2차원 행렬 As와 Bs가 있는데 그 크기는 모두 32 × 32이다.

Algorithm 2의 계산 과정을 살펴보면 Fig. 2와 같이 32 × 32개의 invocation에 의해 행렬 A의 부분 데이터

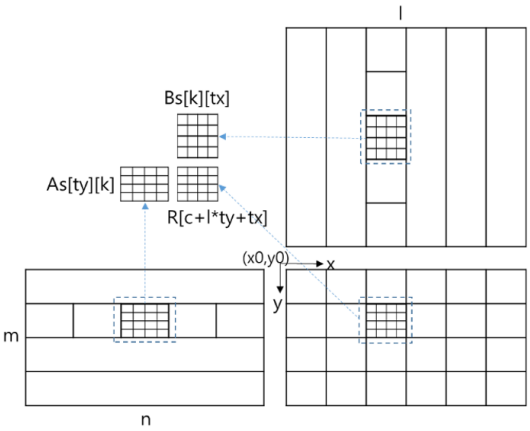

Fig. 2 Parallel computing using shared memory: As & Bs represent the shared memory for matrix A & B, respectively, (x0, y0) denotes the starting indices of R-submatrix.

Algorithm 2: GPU codes (which runs on GPU)

```
#version 430 core

layout( local_size_x = 32, local_size_y = 32, local_size_z = 1 ) in;

uniform uint m;
uniform uint n;
uniform uint l;
uniform uint x0;
uniform uint y0;

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

const uint BLOCK_SIZE = gl_WorkGroupSize.x;

shared float As[BLOCK_SIZE][BLOCK_SIZE];
shared float Bs[BLOCK_SIZE][BLOCK_SIZE];

void __memoryBarrierShared()
{
 memoryBarrierShared(); barrier();
}

void main()
{
 uint hA = m;
 uint wA = n;
 uint wB = l;

 uint bx = gl_WorkGroupID.x;
 uint by = gl_WorkGroupID.y;

 uint tx = gl_LocalInvocationID.x;
 uint ty = gl_LocalInvocationID.y;

 uint gx = x0+gl_GlobalInvocationID.x;
 uint gy = y0+gl_GlobalInvocationID.y;

 uint aBegin = wA*(y0 + BLOCK_SIZE*by);
 uint aEnd = aBegin + wA - 1;
 uint aStep = BLOCK_SIZE;

 uint bBegin = x0 + BLOCK_SIZE*bx;
 uint bStep  = BLOCK_SIZE*wB;

 float Rsub = 0.0f;

 for (uint a = aBegin, b = bBegin, blkStart = 0;
  a <= aEnd;
  a += aStep, b += bStep, blkStart+=BLOCK_SIZE)
 {
  As[ty][tx] = (gy < hA && blkStart + tx < wA) ?
  A[a + wA*ty + tx] : 0.0f;
  Bs[ty][tx] = (gx < wB && blkStart + ty < wA) ?
  B[b + wB*ty + tx] : 0.0f;
  __memoryBarrierShared();

  #pragma unroll
  for (uint k = 0; k < BLOCK_SIZE; k++)
  {
    Rsub += As[ty][k] * Bs[k][tx];
  }
  __memoryBarrierShared();
 }

 if( gy < hA && gx < wB )
 {
  uint c = wB*(y0 + BLOCK_SIZE*by) +
  (x0 + BLOCK_SIZE*bx);
  R[c + wB*ty + tx] = Rsub;
 }
}
```

인 32 × 32개의 행렬 요소(element) 가 As 배열에 복사되고, 마찬가지로 행렬 B의 부 분 데이터인 32 × 32개의 행렬 요소가 Bs 배열에 복사된다. 이후 As와 Bs가 곱해지고 그 결과가 Rsub변수에 누적되고, 이어서 동일한 방식의 계산 반복을 위해 다음 위치, 행렬 A의 a + aStep로, 행렬 B의 b + bStep로 이동한다. 32 × 32개의 invocation 이 반복 계산을 끝내면 최종적으로 누적된 Rsub 값이 행렬 R의 해당 위치 즉 R [c + wB*ty + tx]에 복사된다. 여기서 As와 Bs에 데이터가 복사될 때 OpenGL barrier() 함수가 호출되고, Rsub변수에 값이 누적될 때도 호출된다. 32 × 32 개의 shader invocation 이 shared memory인 As와 Bs의 해당 위치에 데이터를 올려놓고 서로 공유하면서 계산 결과를 누적시켜 나가는 구조이다.

4. 수치 실험

본 연구에서 사용한 그래픽 카드는 엔디비아 Quadro 4000이고, 비디오 메모리 크기는 2,048 MB, 블록 당 스레드 개수는 1024개이다. 본 연구에서 사용한 병렬 계산 평가지표는 유효(effective) GFLOPS이다. 즉 알고리즘 계산 성능을 수치로 나타내기 위해 초당 부동소수점 연산 회수를 기가 단위로 표시하는 GFLOPS를 주요 지표로 사용하는데, 본 연구 행렬 곱셈의 경우, 곱셈 연산 횟수가 (m x n x l)이고 덧셈 연산 횟수가 (m x n x l)이므로 다음과 같이 나타낼 수 있다.

$$GFLOPS = 2\,(m \times n \times l) / (10^9 \times \text{수행시간(sec)}) \quad (1)$$

만약에 크기가 n인 정방 행렬이고 수행시간 단위로 milliseconds(msec)를 쓴다면 위의 식은

$$GFLOPS = 2\,n^3 / (10^6 \times \text{수행시간(msec)}) \quad (2)$$

가 된다. 일반적으로 위의 GFLOPS 값은 GPU 제조사가 발표하는 최상(peak)의 이론적 수치와 다르며, 어떤 그래픽 카드에서 무슨 메모리를 얼마만큼 사용하여 어떻게 계산했느냐에 따라 다르게 측정된다. 또한 GPU의 수행시간은 매우 짧기 때문에 동일 컴퓨터라 하더라도 사용중인 컴퓨터 상황에 따라 적잖은 편차가 발생할 수 있다. 결국 다수의 측정을 통한 평균값 계산이 필요하다. 본 연구의 경우는 10회 측정하여 평균하였다.

한편 본 연구에서는 구현한 GPU 알고리즘의 성능을 비교 평가하기 위해, 일반적으로 CPU 메인 메모리를 가지고 수행되는 CPU 행렬 곱셈 프로그램을 Algorithm 3과 같이 구현하였고, Table 1과 같이 본 연구 병렬 계산 알고리즘과 비교하였다.

Algorithm 3: Non-parallel CPU code for comparison

```
void mul_cpu( Matrix& A, Matrix& B, Matrix& R )
{
  for (uint i = 0; i<m; i++)  {
    for (uint j = 0; j<l; j++) {
      sum = 0.0;
      for (uint k = 0; k< n; k++)
        sum = sum + A[i][k] * B[k][j]
      R[i][j] = sum;
    }
  }
}
```

비교에 사용된 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 GPU 의 GFLOPS 값을 속도비로 표시하였다. Fig. 3의 왼쪽 축은 GFLOPS 축이고, 오른쪽 축은 속도비 축이다. 예상대로 GPU-알고리즘이 상당히 빠르며, n = 4096일 때 대략 208배의 속도 차이가 나타냄을 확인할 수 있다.

Table 1 Performance comparison of CPU and GPU-algorithm

(GFLOPS : Giga-sized floating point operations per second)

N x N	CPU		GPU		speed ups
	Computing Time (ms)	GFLOPS	Computing Time (ms)	GFLOPS	
128 x 128	2.35	3.57	0.31	13.53	4
256 x 256	19.02	3.53	0.72	46.60	13
512 x 512	236.7	2.27	4.02	66.77	29
1024 x 1024	7,967.3	0.54	29.14	73.70	137
2048 x 2048	86,792	0.40	230.85	74.42	188
4096 x 4096	763,312	0.36	1,831	75.05	208

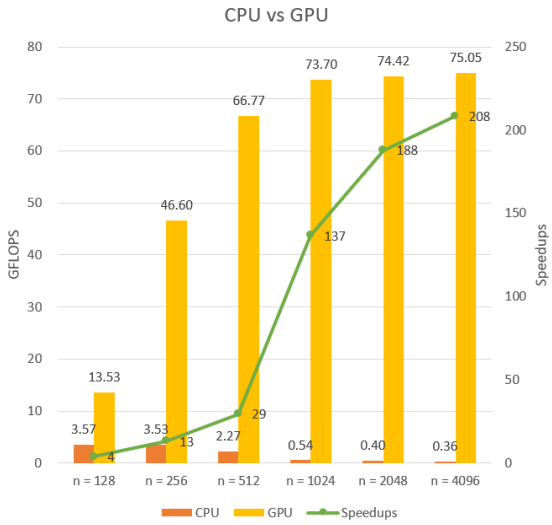

Fig. 3 Performance comparison of CPU-algorithm and GPU-algorithm

5. 결 론

본 연구는 GPU 상에서 병렬 처리에 의해 수행되는 OpenGL 컴퓨터 쉐이더 행렬 곱셈 알고리즘을 상세히 서술하였고, 수치 실험을 통하여 본 연구 알고리즘의 계산 성능을 기술하였다. 또한 알고리즘 구현에 필요한 GPU 메모리 구조 및 쉐이더 invocation, barrier 동기화 등에 관해 기술하였다.

본 연구에서 제시한 GPU 행렬 곱셈 알고리즘은 OpenGL 라이브러리를 사용하기 때문에 특정 그래픽 카드에 대한 의존성이 전혀 없다. 또한 그래픽 렌더링에 바로 사용될 수 있으며, 나아가 게임분야의 물리 엔진에도 기반 기술로서 활용도가 높을 것으로 기대된다.

추후 연구 과제로 본 연구에서 개발한 병렬 처리 방식을 바탕으로, 전치행렬 및 역행렬을 병렬 계산하는 프로그램을 개발할 예정이며, 수치해석 기법인 PCG, GMRES 등에 관해 GPU 가속화 프로그램을 개발할 예정이다. 현재 방대한 계산량을 요구하는 FFT 및 합성곱(convolution) 등을 개발 중에 있다.

Acknowledgement

이 논문은 2019년 한국교통대학교 지원을 받아 수행하였음.

References

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