병렬처리론 GPGPU(CUDA)

* Matrix Multiplication Using CUDA

2011253020 이화중

# Abstract

CUDA는 GPU를 프로그램 연산에 참여시켜 속도 향상을 얻는다. GPU에는 ALU가 CPU에 비해 월등히 많으므로, 많은 수의 연산을 병렬처리하여 속도 향상을 얻는 데 큰 도움을 줄 수 있다. CUDA는, 이러한 CPU에 계산량이 많은 부분을 맡겨 속도를 향상시킨다. 이 때 CPU는 순차적인 부분에 대한 연산을 맡는다. 이 Architecture에서, CPU를 포함한 전통적인 연산부를 Host, 실 연산을 수행할 GPU를 Device라고 한다.

이 프로그램은 같은 동작(행렬 곱셈)을 CUDA에서 수행할 때와 CPU에서만 연산할 때의 성능 향상을 비교한다. 성능 비교에 사용하는 지표는, 각 행렬 연산 함수의 시작부터 종료까지 소요되는 시간(㎳)이다. 각 함수 1회 수행에 소요되는 시간은 조금씩 차이가 있을 수 있으므로, 각 10회를 수행하여 각각의 평균을 내어 비교한다.

GPU의 ALU는 CPU에 비해 월등히 많으므로 이를 병렬처리한다면, 연산량이 많을 경우, CPU만을 사용하여 연산한 경우보다 성능이 월등히 좋을 수 밖에 없다. 이 차이를 시각적으로 확인한다. 그리고 행렬 곱셈의 계산 시간은 행렬의 크기에 종속적이다. 이에 따른 속도 차이를 확인한다.

추가적으로 CUBLAS Library를 이용한 행렬 곱셈과 비교한다. CUBLAS는 CUDA에서 제공하는 선형 대수 연산 Library로, Fortran으로 작성된 BLAS Library를 CUDA 버전으로 Porting한 Library이다.

# Results

## GPU로 함께 처리 했을 때(CUDA)와 CPU로 처리했을 때의 비교

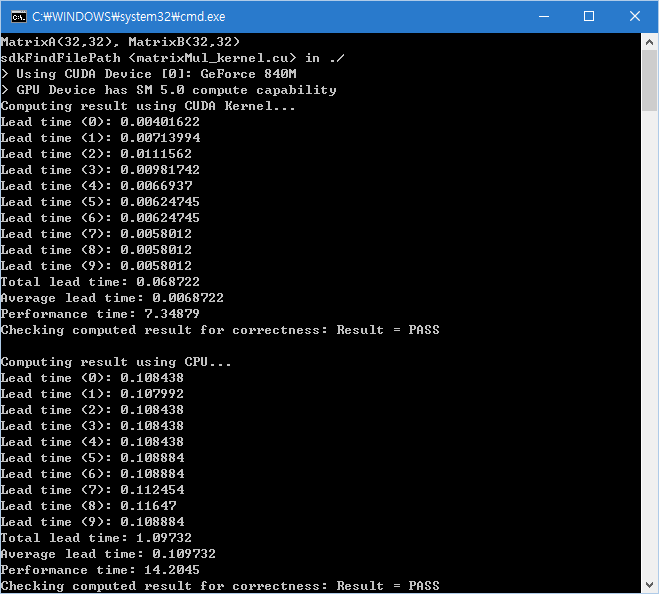


Figure 1 CUDA와 CPU 연산 시간 비교 (행렬 크기 32\*32)

정방형의 행렬 곱셈을 수행하는 데 소요된 시간의 결과이다. 행렬의 너비와 높이는 블록 크기가 32일 때의 최소 크기(32)와 같게 하였다. CUDA로 처리했을 때 평균 소요 시간은 0.0068722㎳, CPU로 처리했을 때는 0.10973㎳로, GPU로 함께 처리했을 때 처리속도가 빨랐다. CPU 대비 CUDA의 속도 향상은 이다.

## 행렬의 크기를 증가시키고 처리 시간 측정 및 속도 증가 계산

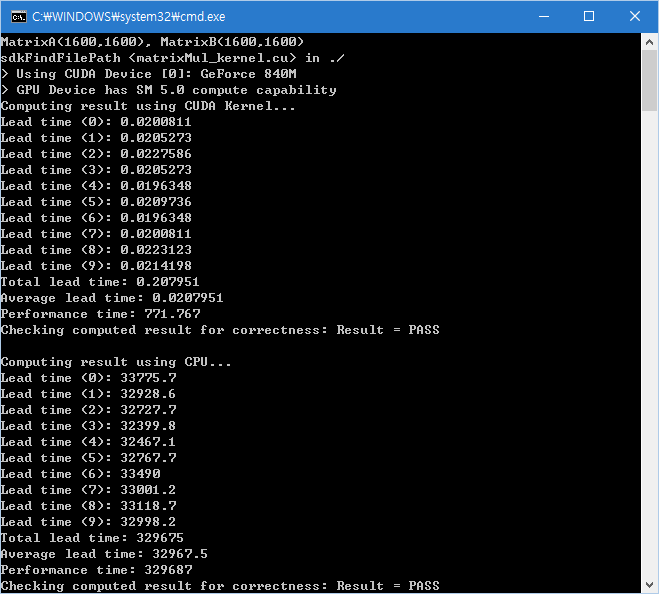


Figure 2 행렬 크기를 늘렸을 때의 CUDA와 CPU 연산 시간 증가 비교 (행렬 크기 1600\*1600)

Figure 1에 비해 너비와 높이를 각각 50배 증가시켰다. 따라서 행렬 셀의 수는 2,500배 증가하였다. CUDA로 처리했을 때 평균 소요 시간은 0.020795㎳, CPU로 처리했을 때는 32,968㎳로, GPU로 함께 처리했을 때 처리속도가 월등히 빨랐다. 소요된 시간은 각각 3.0260 배, 300,440 배가 증가했다. 이때의 CPU 대비 CUDA의 속도 향상은 이다. 이 수치는, 2.1의 결과인 과 비교하여 99,282배 증가한 수치이며, CUDA의 진가는 병렬 처리하는 연산량이 많을 때 더욱 크게 발휘한다는 것을 보여준다.

## CUBLAS Library를 사용한 결과와 속도 향상 비교

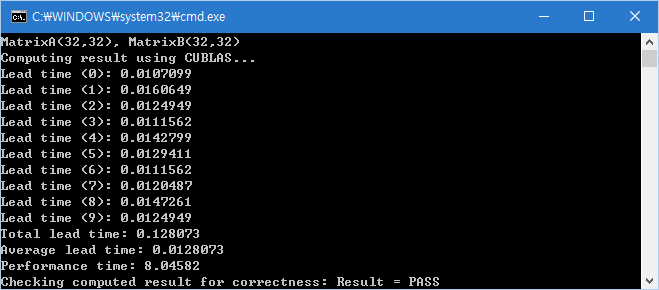


Figure 3 CUBLAS Library를 사용한 행렬 곱셈 (행렬 크기 32\*32)

앞서 2.1에서 수행한 결과와 함께 비교하기 위하여 행렬 크기를 32\*32로 하여 수행하였다. CUBLAS Library로 10회 연산한 결과의 평균은 0.012807㎳이다. Figure 1의 CPU로 처리한 연산 시간 평균과 비교했을 때 속도 향상은 이다. CPU 대비 CUDA의 속도 향상인, 에 비교해 0.53658배였다.

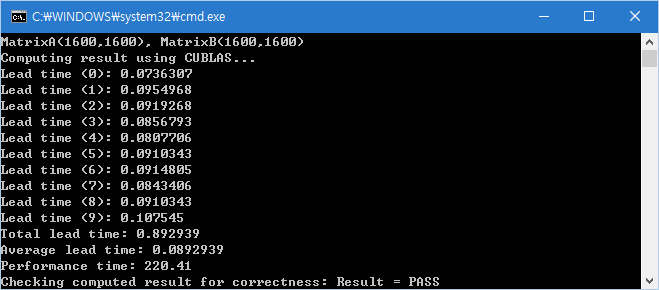


Figure 4 CUBLAS Library를 사용한 큰 규모의 행렬 곱셈 (행렬 크기 1600\*1600)

앞에서와 마찬가지로, 앞에서의 결과와 함께 비교하기 위하여 행렬 크기를 2.2에서 수행한 것과 같게 하였다. 이때 CUBLAS Library로 10회 연산한 결과의 평균은 0.089293㎳이다. Figure 3의 결과에 비해 연산 시간은 6.9721배 증가하였다. 2.2에서 CUDA의 시간 증가폭인 3.0260배보다 크다. 이는 연산량 증가에 따른 CUBLAS의 연산 성능이 CUDA보다 조금 못하다는 것을 의미한다. Figure 2의 CPU로 처리한 연산 시간 평균과 비교한 속도 향상은 으로, 여전히 CPU에 비해서 엄청난 속도 향상을 보여준다.

한 가지 흥미로운 점은, CUBLAS Library를 사용하여 연산할 때와 그렇지 않을 때의 GPU를 동기화하는 시간에 차이가 있다는 것이다. 행렬의 크기가 클 때 특히, 전자의 경우가 후자의 경우보다 짧았다. 이러한 점 때문인지, 전자의 개별 행렬 연산 속도는 더 느림에도 불구하고, 동기화를 포함한 전체 연산의 수행 시간은 전자의 경우(Figure 4 Performance time: 220.41㎳)가 후자(Figure 2 Performance time (CUDA): 771.77㎳)보다 더 빠른 수행 속도를 보여줬다. CUDA 연산 시 GPU 동기화가 필요하다는 점을 감안한다면, 행렬의 크기가 클 때 전반적인 연산 시간을 단축시키는 데는 CUBLAS Library를 사용하는 편이 좋을 것으로 예상된다.

# Source Code

CUDA의 소스코드는 크게, CPU에서 수행하는 Host Code와 GPU에서 수행하는 Device Code로 나뉜다. Host Code는 Host에서 수행할 순차적인 연산과, Device에 대한 kernel의 launch를 담당한다. Device Code에는 실제로 GPU가 수행할 연산들을 담는다. 일반적으로, Host Code는 일반적인 C++ 프로그래밍과 같이 .cpp 파일에 작성하고, Device Code는 .cu 파일에 작성한다. 이 프로그램에서는 Host Code는 MatrixMul.cpp, Device Code는 MatrixMul\_kernel.cu에 구현되어 있다.

CUDA를 이용한 배열 곱셈은 크게 두 단계로 나뉜다. 첫 번째로 Device에 메모리를 할당하고 Host에서 Device로 각종 데이터를 전달하는 부분과, Device Kernel을 실행하여 실제 행렬 곱셈을 수행하는 부분으로 나뉜다. CUBLAS Library를 사용한 배열 곱셈은, Device Kernel을 직접 실행하는 코드 대신 CUBLAS의 행렬 곱셈 Library 함수를 호출하는 코드를 작성한다.

## Device에게 행렬 곱셈을 수행시키는 Host Code의 핵심

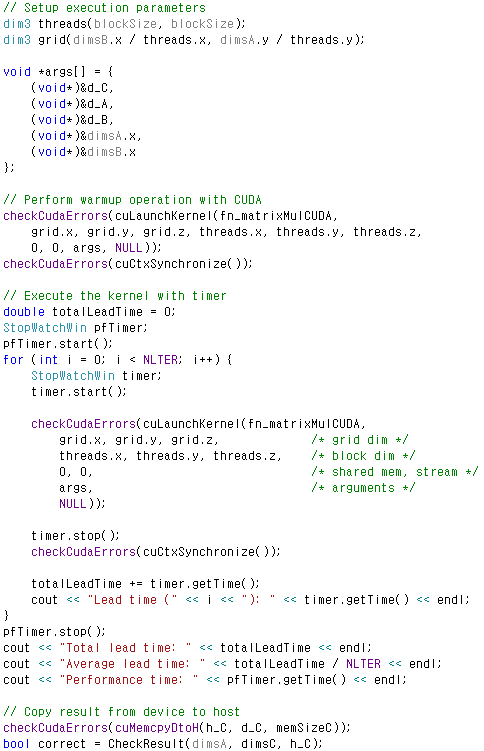


Figure 5 Host에서 CUDA Kernel 실행 및 Host로의 결과 반환 (MatrixMul.cpp:   
matrixMultiplyUsingCUDA()) 여기서 NLTER은 행렬 곱셈의 테스트 횟수이다.

여기서 Host는 사전에, 수행할 함수(fn\_matrixMulCUDA), 그리드 크기(grid dim)와 스레드 크기(block dim), 공유 메모리 크기와 스트림(shared mem, stream), 함수 아규먼트(arguments) 등의, 데이터들을 설정하고 CUDA Kernel을 실행시킨다. 여기서 CUfunction형의 변수인 fn\_matrixMulCUDA는 실행될 CUDA Kernal 상에서 수행될 배열 곱셈 함수인 matrixMulCUDA() 수행에 대한 데이터를 갖는다. 이를 통해 각 스레드에서 수행되는 코드인 MatrixMul\_kernel.cu의 matrixMulCUDA() 함수가 각 스레드에서 동작할 수 있게 된다. 그리고 시간 측정에 들어가기 앞서 warm-up 연산을 1회 먼저 행하였는데, 처음 계산을 수행했을 때의 튀는 값을 제거하기 위함이다.

행렬 곱셈 시간 측정은 StopWatchWin (helper\_timer.h에 정의) 객체를 생성하여 수행한다. 이 객체의 start()로 타이머를 시작하고, stop()로 정지한다. 실제 타이머를 동작시키는 시간은 CUDA kernel 실행 시작부터 종료 시까지 이다. getTime()을 통해 측정된 시간을 가져올 수 있는데, 시간의 단위는㎳이다. 시간 측정에서 GPU를 동기화하는 시간은 제외하였다.

## 실제로 행렬 곱셈을 수행하는 Device Code의 핵심

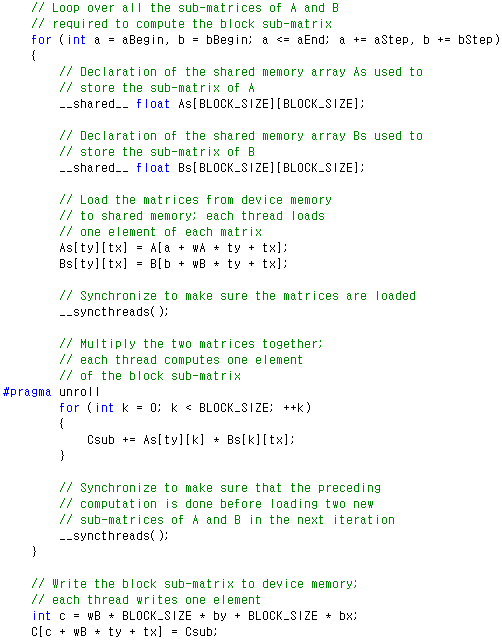


Figure 6 각 스레드 별로 수행할 행렬 연산 코드 (MatrixMul\_kernel.cu: matrixMulCUDA())   
bx, by는 각 블록의 인덱스이며, tx, ty는 각 스레드의 인덱스이다.

이 코드는 하나의 스레드가 결과 행렬의 한 셀에 대한 연산을 맡는다. 다시 얘기하면, 좌측 행렬의 한 행과 우측 행렬의 한 열에 대한 곱셈을 담당한다는 것이다. 그리고 각 행 또는 열의 길이를 블록 크기 만큼씩 나누어 연산한다. 블록에 해당하는 반복문 1회 수행 시, Kernel 실행 시에 전달받은 행렬에서, 각 스레드가 맡은 한 셀을 공유 메모리에 올린다. 이 작업은 병렬로 처리되어 동시에 수행된다. 다만, 각 스레드 간 동작에 시간 차이가 미세하게나마 발생할 수 있으므로, \_\_syncthreads()를 호출하여 스레드를 동기화해준다. 다음으로, 블록 사이즈만큼에 해당하는, 행렬의 실제 곱셈을 수행한다. 여기서도 마찬가지로 \_\_syncthreads()를 호출하여 스레드를 동기화해준다. 이런 식으로, 반복문 전체가 수행되면 결과적으로 결과 행렬의 한 셀에 대한 값을 얻게 된다. 얻은 이 값을 각 셀에 저장한다.

## CUBLAS Library를 사용한 행렬 곱셈 Code의 핵심

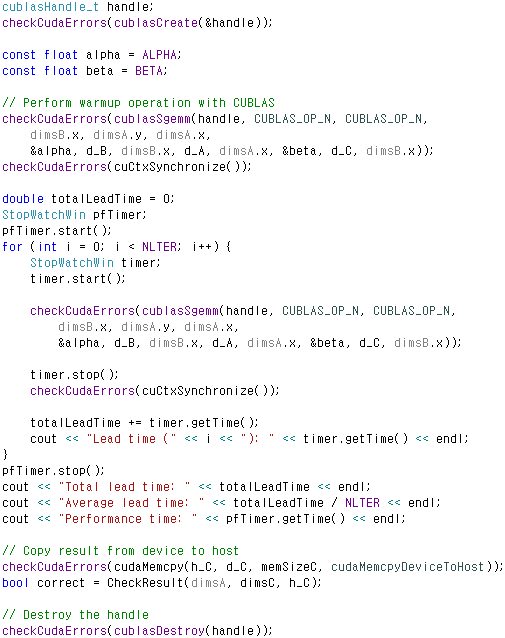


Figure 7 CUBLAS Library를 사용한 행렬 곱셈 (MatrixMul.cpp: matrixMultiplyUsingCUBLAS())

CUBLAS Library에는 cublasSgemm()이라는 2차원 행렬 곱셈 함수가 있다. (사실, cublasSgemm()은 매크로이며, 실제로 호출되는 함수는 cublasSgemm\_v2()이다.) 이 Library 함수는, CUBLAS를 위한 핸들 및 디바이스 메모리 상에 할당된 각 행렬의 시작 주소와 각 행렬의 크기 등을 받아, 행렬 곱셈을 수행한다. 행렬 곱셈을 수행하는 부분 이외의 코드에 대한 설명은 3.1과 같다.

# Discussion

이 과제를 수행하는 데 사용한 Toolkit은 Nvidia CUDA 7.5, IDE는 Microsoft Visual Studio 2015이다. 하지만 CUDA Toolkit 7.5는 현재 Visual Studio 2015 상에서의 컴파일을 지원하지 않는다. (\*.cu 파일을 프로젝트의 빌드에 참여시키면 NVCC(CUDA Compiler)에 의해 컴파일 오류가 발생한다.)

프로그래밍에 있어서, CUDA Toolkit 7.5 설치 시 함께 설치한 Code Sample을 참고하였다. CUDA 7.5가 Visual Studio 2015를 지원하지 않아 Visual Studio 2015에 최적화된 Code는 제공되지 않았지만, 프로그램 작성이나 프로젝트 설정 등, CUDA 7.5를 활용한 프로그래밍을 참고하는 데 도움이 되었다.

CUDA 7.5가 Visual Studio 2015를 지원하지 않기 때문에, Visual Studio 2015 상에서 CUDA 7.5 프로그래밍을 하기 위해서는 다른 방법이 필요하다. 바로, SDK를 통하여 간접적으로 컴파일하여 그 함수를 사용하는 것이다. 그 핵심 코드는 다음과 같다. 이 방법에 관련 없는 코드는 생략하였다.

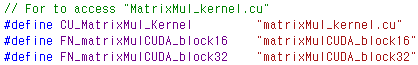


Figure 8 파일 이름과 함수 이름을 지정한 매크로 (MatrixMul.cpp)

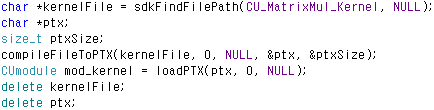


Figure 9 CUDA kernel(MatrixMul\_kernel.cu) 파일 불러오기 (matrixMultiplyUsingCUDA())

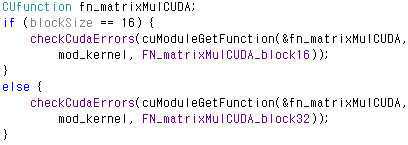


Figure 10 CUDA kernel(MatrixMul\_kernel.cu) 파일 내 함수 연동 (matrixMultiplyUsingCUDA())

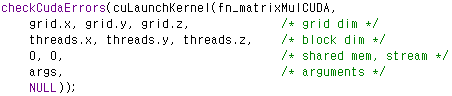


Figure 11 CUDA kernel과 함께 함수 실행시키기 (matrixMultiplyUsingCUDA()) (Figure 5의 일부)

Visual Studio 2015에서라도 CUDA Toolkit 7.5의 Library는 접근하여 사용할 수 있다는 점을 활용한다. 우선, 해당 .cu 파일에 대한 경로를 가져온다(Figure 9: sdkFindFilePath()). 그 다음, 가져온 파일을 컴파일하여 그에 대한 포인터를 얻는다(Figure 9: compileFileToPTX()). 그리고 그 포인터에 대한 CUDA Module을 얻는다(Figure 9: loadPTX()). 이제 그 Module로부터 사용하고자 하는 함수를 가져온다(Figure 10: cuModuleGetFunction()). 여기까지의 과정이 끝나면 가져온 함수를 바탕으로 하는 Kernel을 실행시킨다(Figure 11: cuLaunchKernel()). 이 때, 함수의 아규먼트를, void\* 형의 배열로 구성하여, 함께 전달한다. 이와 같이, Visual Studio 2015 상이 아닌 SDK를 이용하여 간접적으로 컴파일하여 접근하는 방법으로 Visual Studio 2015에서 CUDA 7.5 프로그래밍을 할 수 있다.

# Environment

## Hardware

CPU: Intel i7-4712MQ

GPU: Nvidia GeForce 840M

VRAM: 2GB GDDR3

## Software

Toolkit: Nvidia CUDA Toolkit 7.5

Debugger: Nvidia Nsight (Visual Studio Edition) 5.1

Language: C++, CUDA C++

IDE: Microsoft Visual Studio 2015 (v140)

OS: Microsoft Windows 10 Pro