

[CSE 4152] 고급 소프트웨어 실습 I

『CUDA 프로그래밍의 기초 1』

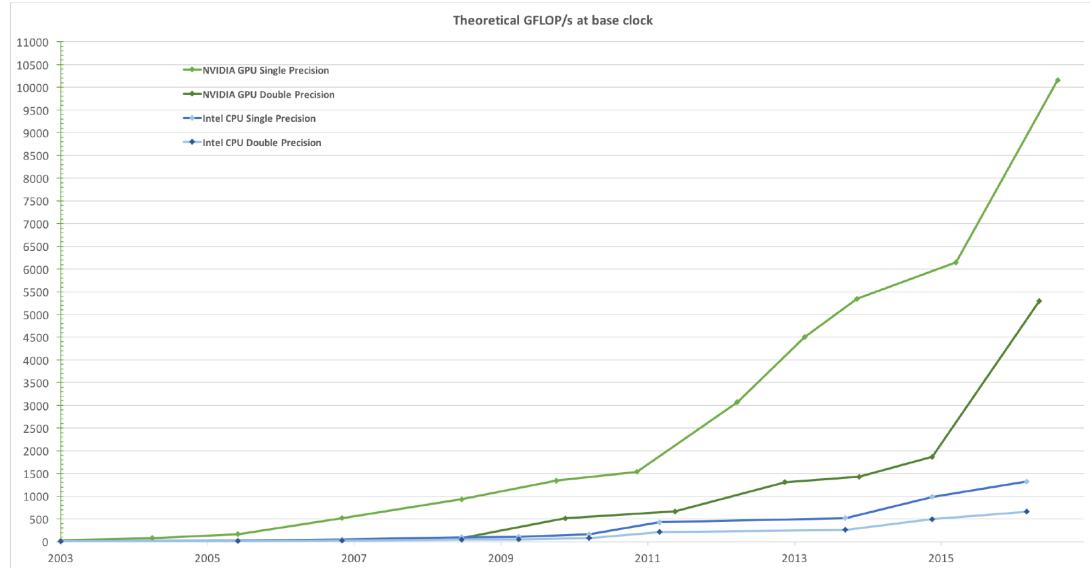
GPU 기반의 Manycore Processing의 기초

담당교수: 컴퓨터공학과 임인성 (AS-905, 02-705-8493, ihm@sogang.ac.kr)

담당조교: 김영욱 (AS-914, 02-711-5278, kimyu7@sogang.ac.kr)

Why GPU programming?

Driven by the insatiable market demand for realtime, high-definition 3D graphics, the programmable Graphic Processor Unit or GPU has evolved into a highly parallel, multithreaded, manycore processor with tremendous computational horsepower and very high memory bandwidth.



Floating-Point Operations per Second for the CPU and GPU (CUDA C PROGRAMMING GUIDE, 2016)

1 SIMD 기반의 병렬 처리의 이해

다음과 같은 간단한 C 코드를 고려하자.

```
#define MAX_N_ELEMENTS 1048576 // pow(2,20)

void combine_two_arrays(float *x, float *y, float *z, int n) {
    int i;
    for (i = 0; i < n; i++) {
        z[i] = 1.0f/(sin(x[i])*cos(y[i]) + cos(x[i])*sin(y[i]));
    }
}

void main(void) {
    int n_elements;
    float A[MAX_N_ELEMENTS], B[MAX_N_ELEMENTS], C[MAX_N_ELEMENTS];
    n_elements = MAX_N_ELEMENTS;
    :
    combine_two_arrays(A, B, C, n_elements);
    :
}
```

여기서 `combine_two_arrays()` 함수의 `for` 문장이 1,048,576($= 2^{20}$)번 반복이 되면서 `z` 배열의 값이 한 번에 한 개씩 순차적으로 계산이 되어 저장되고 있다. 이 함수의 수행과정을 살펴보면 동일한 계산 과정이 서로 다른 데이터에 대하여 반복이 됨을 알 수가 있다. 만약에 `for` 문장 안의 코드(프로그램)를 서로 동기화하면서 병렬적으로 수행시킬 수 있는 프로세서가 1,048,576개가 있다면 이론적으로 계산 시간을 $\frac{1}{1,048,576}$ 로 줄일 수가 있다.

그림 1은 이러한 병렬 계산 방식을 간략히 도시하고 있는데, 이러한 부류의 병렬 처리 방식을 SIMD, 즉 Single Instruction, Multiple Data라 부르는데 자연스러운 이름이라 할 수 있다. 이 방식에서는 병렬성이 존재하는 명령어들 간의 동시성(concurrency)을 이용하는 것 이 아니라, 하나의 프로그램을 동일한 형태로 존재하는 서로 다른 데이터 각각에 대하여 병렬적으로 수행시키는 방식을 취함으로서 처리 속도를 높이는 방식을 취한다. 즉, 주어진 문제의 data level parallelism을 이용한 병렬 처리 방식이라 할 수 있는데, 1970년대에 등장한 여러 슈퍼 컴퓨터들이 이러한 vector processing에 기반을 두었고, 따라서 이러한 컴퓨터들을

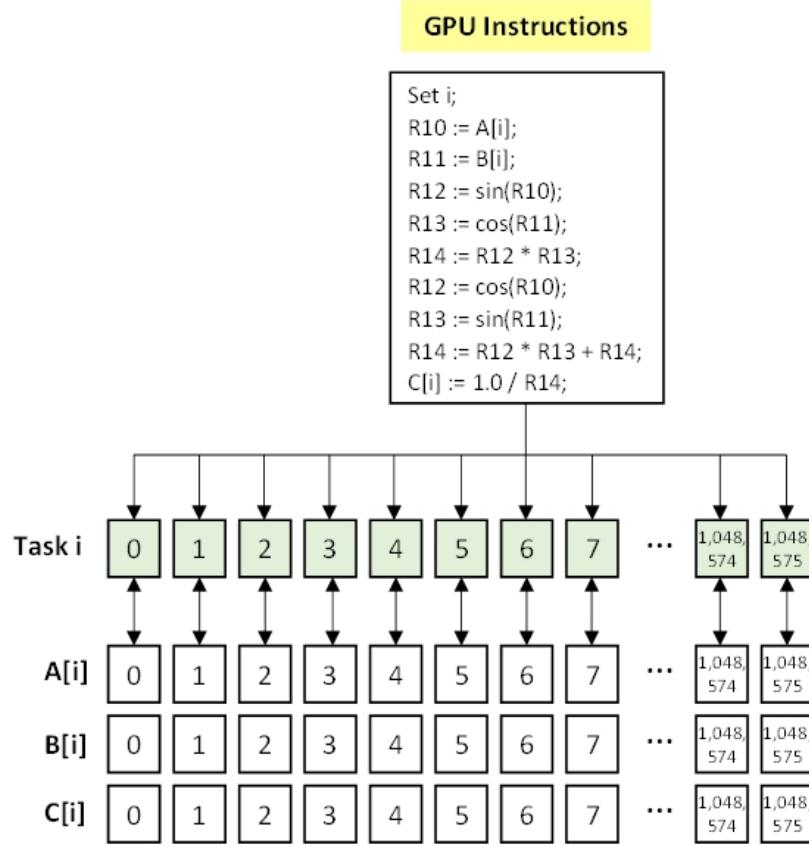


그림 1: SIMD 방식의 병렬 처리 개념

vector machine이]라 불렸었다. 다음은 <https://en.wikipedia.org/wiki/SIMD>에서 발췌한 내용이다.

Single instruction, multiple data (SIMD), is a class of parallel computers in Flynn's taxonomy. It describes computers with multiple processing elements that perform the same operation on multiple data points simultaneously. Thus, such machines exploit data level parallelism, but not concurrency: there are simultaneous (parallel) computations, but only a single process (instruction) at a given moment. SIMD is particularly applicable to common tasks like adjusting the contrast in a digital image or adjusting the volume of digital audio. Most modern CPU designs include SIMD instructions in order to improve the performance of multimedia use.

2 GPU 구조에 대한 개략적인 이해

앞 절의 프로그램 예에서 1,048,576개의 원소를 가지는 배열에 대한 처리를 SIMD 방식의 병렬화 관점에서 간략히 살펴보았는데, 실제로 현실 세계에는 이보다 훨씬 큰 크기의 데이

터에 대하여 SIMD 방식을 통한 병렬 처리를 하고 싶은 경우가 자주 발생한다. 그렇다면 과연, 예를 들어, 1,048,576개의 병렬 프로세서를 장착한 컴퓨터를 쉽게 사용할 수 있을까? 미래에는 가능할지는 모르나 당분간 그러한 것이 현실화 되기는 쉽지 않을 것이다. 그러면 현재 널리 쓰이고 있는 GPU는 이러한 문제를 공학적인 관점에서 어떠한 방식으로 해결하여 SIMD, 더 정확하게 이야기해서 SIMT(Single Instruction, Multiple Thread) 방식의 병렬 프로세싱을 제공할까?

이를 이해하기 위하여 현재 사용이 가능한 GPU의 구조에 대하여 이해하여 보자(GPU 구조에 관한 이 절의 그림은 NVIDIA사의 문서 *NVIDIA's Next Generation CUDA Compute Architecture: Kepler TM GK110(Whitepaper)*에서 발췌함). 그림 2는 NVIDIA사의 Maxwell architectural family(이 절에서는 평의상 GM204라 칭함)에 속하는 GPU의 전체적인 아키텍처를 도시하고 있다(참고로 이 계열에 속하는 최초의 GPU는 GeForce GTX 750 Ti이며, 이후 GeForce GTX 980 계열의 고성능 GPU도 출시됨).

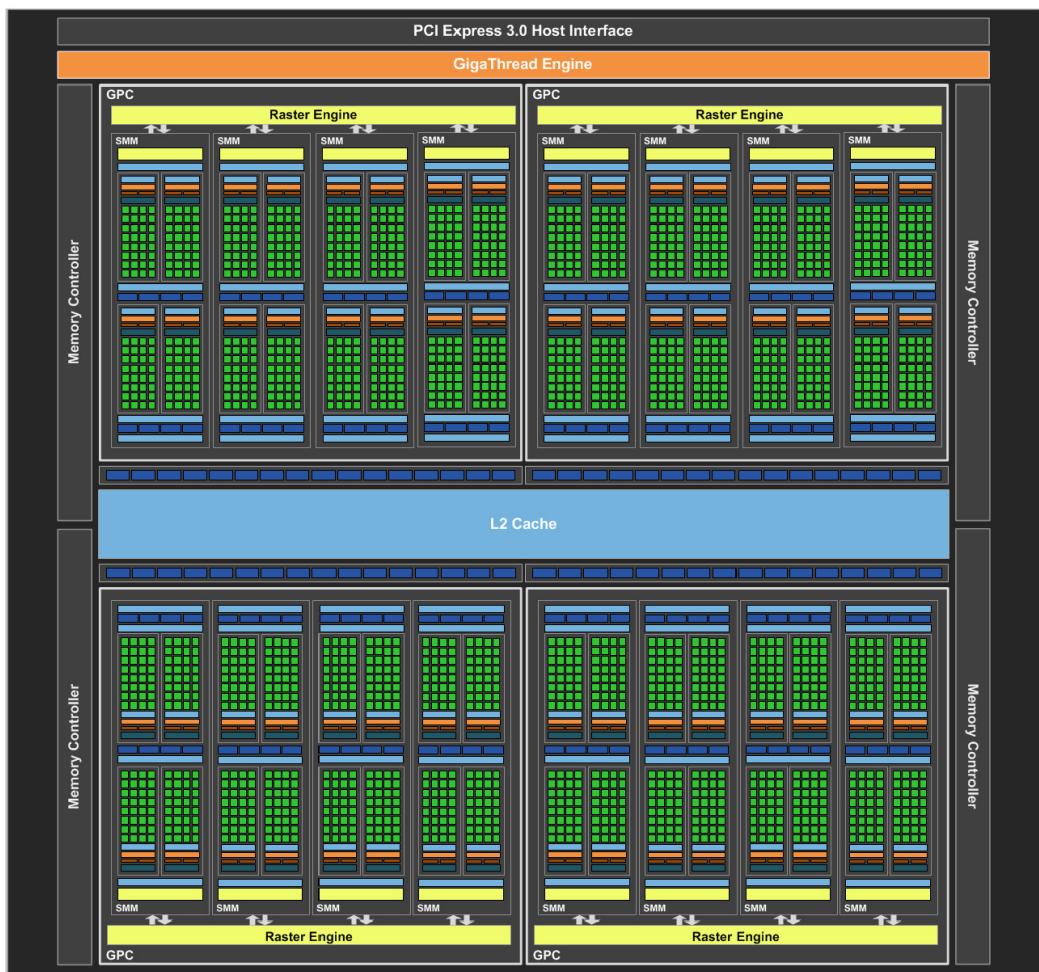


그림 2: GM204 Full-chip block diagram (NVIDIA)

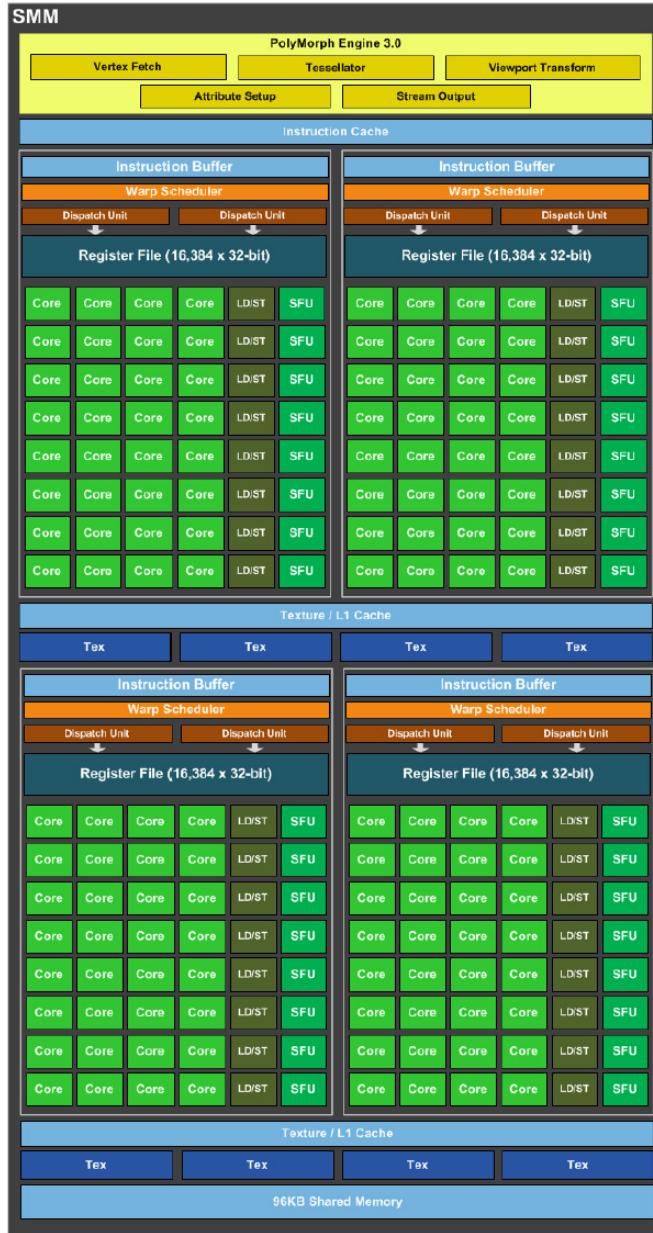


그림 3: GM204 SMM Diagram: GM204 also features 4 DP units per SMM, which are not depicted on this diagram (NVIDIA).

이 그림에 도시된 스펙의 GPU는 16개의 Streaming Multiprocessor (SM)로 구성되어 있는데 (GM 204의 경우 Maxwell SM이라 하며 SMM이라 지칭함), 각 SM의 구조를 좀 더 자세히 살펴보면 그림 3과 같다. SM은 GPU 기반의 병렬처리 구조를 이해하는데 있어 가장 핵심이 되는 부분으로서, 32개의 CUDA 코어 (CUDA core)로 구성된 네 개의 프로세싱 블럭으로 나누어져 있음을 알 수 있다. 각 CUDA 코어는 병렬적으로 계산을 수행해주는 개개의 프로세서에 해당하는데, 그림 2에 도시한 GPU의 경우 $16 \cdot 4 \cdot 32$, 즉 2,048개의 CUDA core

가 장착되어 있다. 이 시점에서 이해해야 할 중요한 사항은 2,048개의 CUDA 코어가 동시에 높은 클럭 속도로 작동을 하나, 전체 GPU의 프로세서가 128개 단위의 SM으로 분할이 되어 있고, 각 SM의 CUDA 코어들은 다른 SM의 CUDA 코어들과 독립적으로 작동한다는 점이다.

3 SIMT 방식에 기반을 둔 GPU 병렬 처리의 이해

그러면, 앞에서 예를 든, 동일한 처리가 필요한 1,048,576개의 태스크를 겨우 2,048개의 프로세서를 장착한 GPU상에서 어떻게 SIMD 방식으로 처리를 할까? 바로 SIMT(Single Instruction, Multiple Thread) 방식으로 프로그래밍을 하고 그에 따라 처리를 하게 되는데, 이에 대하여 좀 더 살펴보자. 먼저 명심을 해야할 사항은 SIMD 처리를 수행하는데 있어 GPU 종류마다 사용할 수 있는 코어 프로세서의 개수가 상이하고, 또한 처리하고자 하는 문제의 태스크의 수가 서로 다르다는 점이다.

이러한 상황에 효과적으로 대처할 수 있는 방법 중의 하나는, 프로그래머는 몇 개의 태스크가 몇 개의 코어 프로세서에서 수행이 되건, 한 개의 태스크가 한 개의 코어에서 수행되는 과정에만 집중하여 프로그램을 작성 하는 것이다(일반적으로는 풀고자 하는 문제의 특성상, 그리고 GPU 프로세싱의 특성상 병렬 프로그램 작성 시 계산이 수행되는 코어 프로세서간의 상호 작용도 고려해주어야 하나, 일단 이에 대한 설명은 생략토록 한다). 앞의 예로 돌아가서, 1,048,576개의 태스크가 SIMD 방식으로 병렬적으로 계산이 수행되는 과정을 생각해보면, 개개의 서로 독립적인 처리 과정에 대해 한 개의 쓰레드(thread)가 수행이 된다고 할 수 있다. 참고로 [https://en.wikipedia.org/wiki/Thread_\(computing\)](https://en.wikipedia.org/wiki/Thread_(computing))에서 발췌한 쓰레드에 대한 설명을 읽어보자.

In computer science, a thread of execution is the smallest sequence of programmed instructions that can be managed independently by a scheduler, which is typically a part of the operating system. The implementation of threads and processes differs between operating systems, but in most cases a thread is a component of a process. Multiple threads can exist within one process, executing concurrently and sharing resources such as memory, while different processes do not share these resources. In particular, the threads of a process share its executable code and the values of its variables at any given time.

Systems with a single processor generally implement multithreading by time slicing: the central processing unit (CPU) switches between different software threads. This context switching generally happens very often and rapidly enough that users perceive the threads or tasks as running in parallel. On a multiprocessor or multicore system, multiple threads can execute in parallel, with every processor or core executing a separate thread simultaneously; on a processor or core with hardware threads, separate software threads can also be executed concurrently by separate hardware threads.

다시 원래의 문제로 돌아와서, 이때 각각의 쓰레드는 동일한 처리 과정을 거치므로, 프로그래머는 몇 개의 태스크가 존재하건 한 개의 쓰레드에서 수행이 되는 계산 과정을 CUDA Programming API를 사용하여 코딩을 하는 방식을 취하면, 태스크와 CUDA 코어의 개수에 상관없이 병렬 프로그램을 작성할 수 있다. 하지만 아직 1,048,576개의 태스크가 2,048개의 CUDA 코어에서 어떻게 처리가 되는지가 불분명한데, 이에 대해 간략히 살펴보자.

쓰레드 블록을 통한 태스크의 분할

우선 프로그래머는 각 쓰레드에서 동일하게 수행되어야 할 한 개의 프로그램을 작성 해야하는데, 병렬 처리 분야에서는 이러한 프로그램을 계산 커널 (compute kernel)이라 부른다(편의상 커널 프로그램이라 하자). 이제 커널 프로그램을 GPU상에서 구동시키기 위해서는 먼저 0번부터 1,048,575번까지의 번호를 가지는 1,048,576개의 태스크를 각각 동일한 개수의 태스크로 구성된 쓰레드 블록 (thread block)으로 나누어주어야 한다. 예를 들어, 각 블록을 256개의 태스크로 구성할 수 있는 쓰레드의 최대 개수는 자신이 사용하는 CUDA Compute Capability에 따라 다름을 명심하자). 이때 각 블록은 256이라는 쓰레드 블록 크기 (thread block size 또는 thread block dimension)를 가진다고 하는데, 한 블록 내에서 256개의 태스크 각각은 0번부터 255번까지 고유한 번호가 부여되는데, 이를 쓰레드 ID (thread ID)라 부른다. 또한, 4,096개의 블록 각각에 대하여 0번부터 4,095번까지 순차적으로 고유 번호가 부여되는데 이를 쓰레드 블록 ID (thread block ID)라 부른다(그림 4 참조).

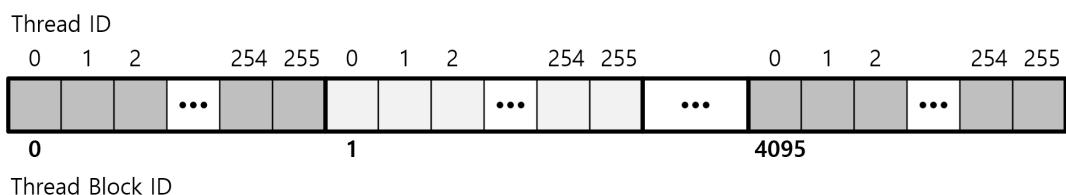


그림 4: 쓰레드 블록을 통한 태스크의 분할

SIMT 방식의 태스크의 구동

이제 1,048,576개의 태스크로 구성된 전체 문제를 256개의 크기를 가지는 4,096개의 쓰레드 블록을 통하여 커널 프로그램을 구동시키면, CUDA 콘트롤러에 의해 각 블록들이 GPU가 제공하는 SM으로 매핑이 되어 계산이 수행된다. 예를 들어, 앞절에서 도시한 GPU의 경우 총 16개의 SM으로 구성되어 있는데, 이 경우 4,096개의 블록이 16개의 SM으로 할당되는데 어떤 블록이 물리적으로 어떤 SM에 매핑이 될지는 알 수가 없으며, 이는 전체 프로그램 수행 시 상황에 따라 CUDA 콘트롤러가 정하게 된다.

또한, 한 개의 블럭 내의 모든 쓰레드들은 모두 동일한 SM에서 처리가 되지만, 실제로 한 블럭 내의 쓰레드들은 모두 동시에 SIMD 방식으로 처리되는 것이 아니라 다시 32개의 쓰레드로 구성된 와프(warp) 단위로 나누어져 계산이 수행된다. 즉 앞의 예라면, 한 블럭의 256개의 쓰레드들이 총 8개의 와프로 분할이 된 후, 와프 단위로 한 SM 내의 CUDA 코어로 매핑이 되어 32개 각 쓰레드에 해당하는 커널 프로그램이 동시에 수행이 된다. 다시 말해서, 한 개의 블럭이 여러 개의 와프로 구성될 경우, 한 와프 내의 32개 쓰레드들은 동시에 SIMD 방식으로 처리가 된다. 하지만 동일한 블럭의 와프들이 어떠한 순서대로 처리가 되어 종료가 되는지는 응용 프로그래머 입장에서 알 수가 없다. 또한 전체 블럭들이 GPU상의 어떠한 SM에 할당되어 어떠한 순서대로 처리되어 종료가 될지도 예측할 수가 없다. 다만 전체 GPU 운용 입장에서 가장 효율적으로 전체 태스크가 병렬 처리가 될 것이라는 사실만 알 수 있다. 따라서 GPU상에서의 처리는 여러 하드웨어적인 그리고 소프트웨어적인 제약으로 인하여, 물리적으로는 추상적인 SIMD 처리와는 다른 방식으로 계산이 수행이 되며, 따라서 이러한 방식을 SIMD라 하지 않고 SIMT(Single Instruction, Multiple Thread) 방식이라 부른다.

쓰레드 ID, 쓰레드 블럭 ID, 그리고 쓰레드 블럭 크기

1,048,576개의 태스크 각각에 대한 쓰레드들이 응용 프로그래머 입장에서는 예측하기 어려운 방식으로 CUDA 코어에 할당이 되어 커널 프로그램이 수행될 때, 가장 기본적으로 해결이 되어야 할 사항은 바로 해당 쓰레드가 1,048,576개의 태스크중 어떤 태스크에 대응이 되는지를 알아낼 수 있어야 한다는 점이다. 이러한 문제를 효과적으로 해결하기 위하여 CUDA 커널 프로그램의 내장 변수인 `blockDim.x`, `blockIdx.x`, 그리고 `threadIdx.x`를 활용한다. 즉 한 번만 작성하는 커널 프로그램은 동일한 이름의 변수를 사용지만, 실제로 i 번째 블럭의 j 번째 태스크가 GPU상에서 쓰레드 형태로 수행이 될 때, i 와 j 값이 각각 `blockIdx.x`와 `threadIdx.x` 변수를 통하여 전달이 되고, 따라서 각 GPU 쓰레드는 자신이 어떤 데이터에 대한 처리를 수행해야 하는지를 알수 있다. 예를 들어, 커널 프로그램 내에서 다음과 같은 간단한 계산을 수행하면,

```
int i = blockIdx.x*blockDim.x + threadIdx.x;
```

자신이 몇 번째 태스크에 대한 계산을 수행해야하는지를 알 수가 있다.

이제 이 강의 자료의 맨 앞에 있는 C 프로그램에 대응하는 CUDA 코드를 살펴보면 다음과 같다. 이 프로그램에서는 좀 더 일반적인 상황을 고려하기 위하여 위에서 언급한 1차원 배열 데이터의 원소들을 `width`개의 뮁음으로 분할하여 `width` 행 `height` 열 크기의 2차원 배열로 저장을 한 후 2차원 쓰레드 블럭을 사용하여 계산을 수행하고 있는데, 이 경우 바로 위에서 언급한 주소 계산이 약간 복잡해지게 된다(자세한 내용은 수업 시간에 설명할 예정인데, 커널 프로그램의 내장 변수인 `gridDim.x`, `gridDim.y`, `blockDim.x`, `blockDim.y`,

`threadIdx.x`, 그리고 `threadIdx.y`가 의미하는 바를 고려하여 주소 계산 방식에 대하여 생각해볼 것).

```
#include<stdio.h>
#include<stdlib.h>
#include <math.h>
#include <Windows.h>
#include<cuda.h>
#include<device_launch_parameters.h>
#include<cuda_runtime.h>

#define MAX_N_ELEMENTS 1048576 // pow(2, 20)
:

int BLOCK_SIZE;

typedef struct {
    int width;
    int height;
    float *elements;
} Array;

__global__ void CombineTwoArrraysKernel(Array A, Array B, Array C) {
    int row = blockDim.y*blockIdx.y + threadIdx.y;
    int col = blockDim.x*blockIdx.x + threadIdx.x;
    int id = gridDim.x*blockDim.x*row + col;

    C.elements[id] = 1.0f / (sin(A.elements[id])*cos(B.elements[id])
        + cos(A.elements[id])*sin(B.elements[id]));
}

void combine_two_arrays(const Array A, const Array B, Array C) {
    Array d_A, d_B, d_C;
    size_t size;
```

```
d_A.width = A.width; d_A.height = A.height;
size = A.width * A.height * sizeof(float);
cudaMalloc(&d_A.elements, size);
cudaMemcpy(d_A.elements, A.elements, size, cudaMemcpyHostToDevice);

d_B.width = B.width; d_B.height = B.height;
size = B.width * B.height * sizeof(float);
cudaMalloc(&d_B.elements, size);
cudaMemcpy(d_B.elements, B.elements, size, cudaMemcpyHostToDevice);

d_C.width = C.width; d_C.height = C.height;
size = C.width * C.height * sizeof(float);
cudaMalloc(&d_C.elements, size);

// Assume that width and height are multiples of BLOCK_SIZE.
dim3 dimBlock(BLOCK_SIZE, BLOCK_SIZE);
dim3 dimGrid(A.width/dimBlock.x, A.height/dimBlock.y);
CombineTwoArrraysKernel <<< dimGrid, dimBlock >>> (d_A, d_B, d_C);

cudaMemcpy(C.elements, d_C.elements, size, cudaMemcpyDeviceToHost);
cudaFree(d_A.elements);
cudaFree(d_B.elements);
cudaFree(d_C.elements);
}

int main(void) {
:
Array A, B, C;

// Assume that MAX_N_ELEMENTS is a multiple of 1,024.
A.width = B.width = C.width = 1024;
A.height = B.height = C.height = MAX_N_ELEMENTS/1024;
BLOCK_SIZE = 16;
```

```
A.elements = (float *) malloc(sizeof(float)*MAX_N_ELEMENTS);  
B.elements = (float *) malloc(sizeof(float)*MAX_N_ELEMENTS);  
C.elements = (float *) malloc(sizeof(float)*MAX_N_ELEMENTS);  
  
generate_random_float_array(A.elements, MAX_N_ELEMENTS);  
generate_random_float_array(B.elements, MAX_N_ELEMENTS);  
  
combine_two_arrays(A, B, C);  
:  
}
```

요약을 하면, SIMD 병렬성을 가지는 (일반적으로) 매우 큰 개수의 태스크를 GPU상에서 효과적으로 처리를 하기 위해서는

- 한 개의 태스크, 즉 한 개의 쓰레드에 해당하는 커널 프로그램을 작성하고,
- 전체 태스크를 동일한 개수의 쓰레드로 구성된 블럭 단위로 분할한 후,
- 블럭 개수와 블럭 크기 인자를 통하여 커널 프로그램을 구동시키면,
- 각각의 블럭들이 적절히 GPU의 SM에 매핑이 되어, 와프 단위로 병렬처리가 수행이 된다.

한 블럭 내의 모든 와프의 수행이 끝나면 해당 블럭의 수행이 종료하고, 이렇게 모든 블럭의 수행이 끝나면 전체 프로그램의 수행이 종료하게 된다. 지금까지 CUDA API를 통한 GPU 프로그래밍에 대하여 아주 간략하게 살펴보았는데, 실제로는 최고의 성능을 보장하기 위해서는 고도의 프로그래밍 스킬을 필요로 하는 작업으로서, 상당한 노력과 경험을 통하여 효율적인 CUDA 프로그램을 작성할 수 있는데, 현재 그리고 장래의 GPU 프로그래밍의 중요성을 고려하여 심도 깊게 공부하기 바란다.

4 실습 문제

이번 실습에서는 조교의 지도 하에 자신의 컴퓨터에 CUDA 프로그래밍 시스템을 설치한 후 간단한 CUDA 프로그램을 수행시켜보자.

실습 문제 1

먼저 조교가 설명하는 방식대로 자신의 PC에 CUDA 프로그래밍 시스템을 사용 할 수 있는 환경을 구축하라. 다음 자신의 CUDA 환경을 이해하기 위하여 다음 과 같은 내용을 파악한 후 보고서에 기술하여 제출하라.

1. 자신이 사용하는 컴퓨터에 장착된 GPU의 기종
2. 현재 설치된 CUDA 시스템의 Compute Capability
3. 현재 CUDA Compute Capability가 제공하는 각종 성능 및 스펙

실습 문제 2

- (i) 금주 강의 자료에서 설명한 C/C++ 프로그램에 조교가 제공하는 시간 측정 코드를 삽입하여 수행 시간을 측정할 수 있도록 하라. 어떻게 하면 가급적 정확한 시간을 측정할 수 있을지 생각해볼 것.
- (ii) 금주 강의 자료에서 설명한 CUDA 프로그램에 조교가 제공하는 시간 측정 코드를 삽입하여 수행 시간을 측정할 수 있도록 하라. 어떻게 하면 가급적 정확한 시간을 측정할 수 있을지 생각해볼 것.
- (iii) 다음 이 CUDA 프로그램에 대하여 블럭 크기를 다양하게 변화 시켜가면서 시간을 측정한 후, 그 결과를 보고서에 테이블로 요약하라. 참고로 블럭의 크기는 와프의 크기인 32의 배수로 하고, 한 블럭이 가질 수 있는 크기에 어떠한 제한이 있는지 파악하기 위하여 실습 문제 1-1을 통하여 얻은 정보를 활용하라. 또한 의미있는 결과를 얻기 위하여 전체 데이터의 크기 (1,048,576)를 자신의 시스템이 허용하는 범위 내에서 최대로 한 후 실험을 진행할 것.
 - (이 프로그램은 매우 간단한 프로그램이라 큰 차이가 없을 수 있으나) 블럭 크기에 따른 수행 시간 변화가 있는지 확인하고 자신이 발견한 사항을 보고서에 기술하라.

- 현재 커널 프로그램이 수행해주는 계산을 좀 더 (의미가 있고) 복잡하게 하여 한 쓰레드의 계산 시간을 길게 한 후, 블럭 크기에 따른 수행 시간 변화를 분석하라.

실습 문제 3

이번 실습 문제에서 해결해야 할 문제는 다음과 같다.

- 지금 메인 메모리에는 두 개의 배열이 저장되어 있다. 첫 번째 배열에는 32 행 32열 크기의 행렬 M 이 행 우선 순서로 저장이 되어 있고, 두 번째 행렬에는 32개의 크기를 가지는 n 개의 벡터 \vec{x}_i 가 순차적으로 저장이 되어 있다 ($i = 0, 1, \dots, n - 1$). 따라서 첫 번째 배열은 총 $32 \cdot 32$ 개, 그리고 두 번째 배열은 총 $32n$ 개의 원소를 가지고 있는 원소를 가지고 있는 배열이 되어 있는데, 이 문제에서는 각 원소가 4바이트를 사용하는 float 타입으로 표현 된다고 가정함.
- 메인 메모리에서 두 배열 데이터를 읽어들여 $\vec{y}_i = M\vec{x}_i$ 에 해당하는 벡터 \vec{y}_i ($i = 0, 1, \dots, n$)를 구한 후, 그 결과를 메인 메모리에 $32n$ 개의 원소를 가지는 배열 공간에 순차적으로 저장 해주는 프로그램을 작성하라.

이제 이 문제에 대하여 CPU 및 GPU 기반의 코드를 각각 작성하여 그 성능을 비교하여 보자.

- (i) 먼저 for 문장을 사용하여 순차적으로 n 번의 행렬-벡터 곱셈을 수행해주는 C/C++ 함수를 작성한 후 CPU 수행 시간을 측정하라. 시간 측정 방법은 이전 실습 시간에 사용한 방법을 사용하고, 가급적 정확한 시간 측정을 위하여 여러 번 반복적으로 함수를 수행시킨 후 평균값을 취할 것.
- (ii) 다음 이 문제를 해결해주는 CUDA 커널 프로그램을 작성한 후 가급적 정확하게 GPU 수행 시간을 측정하라. 앞의 문제에서와 같이 쓰레드 블럭의 크기를 변화시켜가면서 수행 시간 관점에서 CPU 방법과 비교 분석한 후, 그 결과를 보고서 기술하라.

주의 1 자신의 PC 환경이 허용하는 범위 내에서 최대한 큰 n 값을 찾은 후 실험을 하라. 샘플 데이터는 n 값이 주어지면 두 배열을 생성한 후 파일에 binary 형식으로 저장을 해주는 프로그램을 조교가 제공할 예정임. 따라서 여러분은 해당 데이터를 파일에서 메인 메모리로 읽어 들인 후 실험을 진행하면 됨(참고로 모든 원소는 -1.0에서 1.0사이의 값을 가짐).

주의 2 당연히 CPU 함수 수행 시간 측정 시 데이터 로딩 시간은 제외함. 마찬가지로 GPU 커널 프로그램 수행 시간 측정 시 CPU 메모리와 GPU 메모리

간의 데이터 이동 시간은 제외를 하고 순수한 행렬-벡터 곱셈 시간만 측정
을 해야 함.

주의 3 당연히 여러분의 프로그램을 정확한 행렬-벡터 결과를 생성해야 하며, 이는
조교가 자신의 방법으로 확인할 예정임.

5 숙제

제출 마감: ?월 ?일 오후 ?시 정각

제출물 및 방법: 조교가 실습 시간에 공지

이번 주의 숙제는 다음과 같다.

숙제 1

다음은 이차 방정식 $f_i(x) = a_i x^2 + b_i x + c_i = 0$ ($i = 0, 1, \dots, n-1$)의 근을 구해주는 간단한 C/C++ 함수이다.

```
void find_roots_CPU(float *A, float *B, float *C,
                     float *X0, float *X1, float *FX0, float *FX1, int n) {
    int i;
    float a, b, c, d, x0, x1, tmp;

    for (i = 0; i < n; i++) {
        a = A[i]; b = B[i]; c = C[i];
        d = sqrtf(b*b - 4.0f*a*c);
        tmp = 1.0f / (2.0f*a);
        X0[i] = x0 = (-b - d) * tmp;
        X1[i] = x1 = (-b + d) * tmp;
        FX0[i] = (a*x0 + b)*x0 + c;
        FX1[i] = (a*x1 + b)*x1 + c;
    }
}
```

이 함수는 n 개의 이차 방정식의 근을 찾아주는 것을 목적으로 하는데, i 번째 이차 방정식의 계수 a_i , b_i , 그리고 c_i 가 각각 순차적으로 배열 A , B , 그리고 C 에 저장이 된 상태에서 이 함수로 넘어오게 된다. 이 함수는 i 번째 방정식의 두 근 x_{0i} 와 x_{1i} 를 구해 각각 $X0$ 와 $X1$ 배열에 저장을 해주고 있다. 또한 이 함수가 구한 근이 얼마나 정확한지를 확인하기 위하여 $f(x_{0i})$ 과 $f(x_{1i})$ 값을 구해 $FX0$ 와 $FX1$ 배열에 저장을 해주고 있다.

조교가 제공하는 세 개의 파일 $A.bin$, $B.bin$, 그리고 $C.bin$ 에는 n 개의 이차 방정식의 계수 a_i , b_i , 그리고 c_i 의 값이 각각 float 타입의 binary 형식으로 순차적

으로 저장되어 있다(n 값은 이러한 이진 파일의 크기를 4로 나누면 알 수 있으나, 조교가 n 값을 공지를 할 예정이며, 자신의 프로그램에서는 매크로 변수 N_EQUATIONS를 사용하여 이 값을 지정하라).

이제 이 세 개의 파일에서 n 개의 이차 방정식의 계수들을 읽어들여 GPU 상에서 위에서 기술한 x_{0i} , x_{1i} , $f(x_{0i})$, 그리고 $f(x_{1i})$ 값을 구한 후, 그 결과를 각각 이름이 X0.bin, X1.bin, FX0.bin, 그리고 FX1.bin인 파일에 저장을 해주는 CUDA 기반의 프로그램을 작성하라. (참고: 문제를 간단히 하기 위하여 본 실험에서 제공하는 데이터의 이차 방정식들은 항상 두 개의 실근이 존재한다고 가정한다.)

- (i) 첫 번째 두 배열 X0와 X1에는 자신이 구한 두 개의 실근 x_0 과 x_1 이 저장되어 있는데 반드시 $x_0 \leq x_1$ 조건을 만족시키도록 저장이 되어야 한다. 다음 두 번째 두 배열 FX0와 FX1에는 각각 x_0 과 x_1 을 대응하는 이차 방정식에 대입하여 계산한 함수값을 저장해주어야 한다. 물론 이론적으로는 모두 0 값이 계산되어야 하지만 수치 계산 시 발생하는 계산 오차로 인하여 정확히 0이 아닌 값이 나올 수 있음을 상기하라.
- (ii) 이제 조교가 지정한 N_EQUATIONS 값에 대하여 자신의 CUDA 프로그램이 가장 빠른 속도를 보이는 블럭의 크기를 실험적으로 결정한 후 그 값을 보고서에 기술하라.
- (iii) 위의 C/C++ 함수와 자신의 CUDA 커널 프로그램의 속도를 가급적 정확히 측정하여 그 결과를 보고서에서 비교 분석하라.

[참고] 숙제 제출물은 다음과 같은데, 상세한 내용은 조교의 지시를 따를 것.

- 자신이 작성한 CPU 프로그램 (program_CPU)과 GPU 프로그램 (program_GPU)
- 정량적인 실험 결과를 포함한 보고서 파일

숙제 2

다음과 같이 정의되는 피보나치 수(Fibonacci number)를 고려하자.

$$F_i = \begin{cases} 0, & i = 0 \\ 1, & i = 1 \\ F_{i-1} + F_{i-2}, & i > 1 \end{cases}$$

임의의 n 에 대해 F_n 은 다음과 같이 표현할 수 있는데,

$$F_i = \frac{1}{\sqrt{5}} \left\{ \left(\frac{1+\sqrt{5}}{2} \right)^i - \left(\frac{1-\sqrt{5}}{2} \right)^i \right\}$$

다음은 입력 배열 $x[]$ 에 저장되어 있는 n 개의 임의의 정수 값 각각에 대한 피보나치 수를 계산하여 출력 배열 $y[]$ 에 저장을 해주는 코드이다(본 수업 제공 코드 참조).

```
int Fibonacci(int n) {
    float sqrt_5, x_0, x_1, tmp_0, tmp_1;

    sqrt_5 = sqrtf(5.0f);
    x_0 = (1.0f + sqrt_5) / 2.0f; x_1 = (1.0f - sqrt_5) / 2.0f;

    tmp_0 = tmp_1 = 1.0f;
    for (int i = 0; i < n; i++) {
        tmp_0 *= x_0; tmp_1 *= x_1;
    }
    return (int) ((tmp_0 - tmp_1) / sqrt_5 + 0.5);
}

#define N 67108864 // 8192 * 8192
int *x, *y; // input/output arrays
void main(void) {
    int n, i;

    n = N;
    :
    for (i = 0; i < n; i++) {
```

```
    y[i] = Fibonacci(x[i]);  
}  
:  
}
```

조교가 제공하는 샘플 입력 파일 `x.binary`에는 먼저 `n` 값이 int 타입의 값으로 binary format으로 저장되어 있고, 이후 `n`개의 임의의 양의 정수가 동일한 형식으로 순차적으로 저장되어 있다(따라서 입력 파일의 크기는 $4 + 4*n$ 바이트임). 이제 입력 파일에서 `n` 값과 `n`개의 양의 정수 값을 배열 `x[]`로 읽어들여 각 `x[i]`에 대한 피보나치 수 $F_{x[i]}$ 를 계산하여 출력 배열 `y[i]`에 저장을 한 후, 배열 `y[]`를 출력 파일 `y.binary`에 저장을 해주는 프로그램을 작성하라. 이때 출력 배열의 형식은 입력 배열의 형식과 동일하다.

- (i) 먼저 본 수업 제공 코드에 기반을 둔 CPU 코드를 작성한 후 처리 속도를 가급적 신빙성 있게 측정하라. (당연히 Visual Studio에서 Release mode를 사용할 것)
- (ii) 다음 이에 대응하는 CUDA 프로그램을 작성한 후 다양한 크기의 블럭에 대하여 속도를 측정한 후, CPU 기반 코드에 비해 얼마나 성능이 향상이 되는지 분석하라.
- (iii) 위의 CPU 코드와 CUDA 코드로 실험한 내용을 자신의 분석 결과와 함께 보고서에 명확히 기술하라.

[참고] 숙제 제출물은 다음과 같은데, 상세한 내용은 조교의 지시를 따를 것. (조교는 $n = 67108864 = 8192*8192$ 인 임의의 입력 파일에 대하여 여러분의 코드를 수행하여 올바르게 출력 파일을 생성하는지 확인할 예정임)

- 자신이 작성한 CPU 프로그램 (`program_CPU`)과 GPU 프로그램 (`program_GPU`)
- 정량적인 실험 결과를 포함한 보고서 파일

숙제 3

-100과 100 사이의 값을 가지는 n 개의 정수로 구성된 수열 $X = \{x_i\}$ ($i = 0, 1, \dots, n-1$)을 고려 하자. 이제 주어진 양의 정수 n_f 에 대하여 이 수열의 각 원소 a_i 를 중심으로 왼쪽과 오른쪽 각 방향으로 n_f 개의 원소들과 자신의 합 s_i 를 구하려 한다. 즉 또 다른 수열 $S = \{s_i\}$ ($i = 0, 1, \dots, n-1$)의 값은 다음과 같이 $(2n_f + 1)$ 개의 원소의 값으로 표현할 수 있는데,

$$S_i = \sum_{k=i-n_f}^{i+n_f} x_k,$$

만약 왼쪽이나 오른쪽에 충분한 개수의 원소가 없을 경우 합이 가능한 원소들까지만 더한다. (예를 들어, $n_f = 5$ 이고 $i = 2$ 라면 $s_2 = \sum_{k=-3}^7 x_i$ 가 되는데, i 가 0보다 작거나 n 보다 같거나 클 경우 x_i 는 0값을 가진다고 생각하면 된다.)

이제 주어진 입력 데이터 파일에서 수열 $X = \{x_i\}$ 를 읽어들여 수열 $S = \{s_i\}$ 를 계산한 후 출력 파일에 저장해주는 문제를 CPU와 GPU 기반의 코드를 각각 작성하여 그 성능을 비교하여 보자.

- (i) 먼저 for 문장을 사용하여 순차적으로 n 번의 동일한 계산을 반복적으로 수행하는 C/C++ 함수를 작성한 후 CPU 수행 시간을 측정하라. 시간 측정 방법은 이전 실습 시간에 사용한 방법을 사용하고, 가급적 정확한 시간 측정을 위하여 여러 번 반복적으로 함수를 수행시킨 후 평균값을 취하라. (이때 데이터 입출력 시간은 제외하고 순수한 수열 계산 시간만 측정할 것.)
- (ii) 다음 이 문제를 해결해주는 CUDA 커널 프로그램을 작성한 후 가급적 정확하게 GPU 수행 시간을 측정하라. 여러 n 과 n_f 값에 대하여 쓰레드 블럭의 크기를 변화시켜가면서 수행 시간 관점에서 CPU 방법과 비교 분석한 후, 그 결과를 보고서 기술하라.

주의 1 이번 숙제에서 입력 파일의 이름은 `Cuda_HW3_input.bin`이고 출력 파일의 이름은 `Cuda_HW3_output.bin`으로 한다.

주의 2 입출력 파일의 데이터는 모두 binary 형식으로 저장되며, 각 파일의 첫 4 바이트에는 원소의 개수 n 이, 그리고 다음 4 바이트에는 덧셈의 범위를 지정하는 n_f 인자가 각각 `int` 타입으로 저장된다. 이후 연달아 $4 \cdot n$ 바이트에는 각 파일에 해당하는 n 개의 원소값이 역시 `int` 타입으로 저장된다.

주의 3 CUDA 코드의 수행 시간을 측정할 때는 데이터 이동 시간은 제외한 순수한 커널 수행 시간만 측정하라.

주의 4 자신의 PC 환경이 허용하는 범위 내에서 최대한 큰 n 값을 찾은 후, n_f 값을 적절히 바꾸어 가면서 실험을 진행하라. n 은 최소 2^{24} 보다 큰 2의 제곱 수이어야 하며, n_f 인자는 다음과 같은 값을 고려할 것.

$$n_f = 1, 4, 16, 64, 256, 1024.$$

주의 5 CUDA 커널 작성 시 1차원 또는 2차원 그리드를 사용하며, 쓰레드 블럭의 크기를 적절히 바꾸어 가면서 수행 시간이 어떻게 변화하는지 실험을 통하여 확인하라. 특히 주어진 블럭 크기에 대하여 n_f 값을 증가시켜가면서 실험을 한 후 GPU 수행 시간이 어떻게 증가하는지 관찰하라.

주의 6 당연히 여러분의 프로그램을 정확한 결과를 생성해야 하며, 이는 조교가 자신의 방법으로 확인할 예정임.

주의 7 다음 주 실험 또는 숙제에서는 이번에 작성한 CUDA 커널 코드를 shared memory를 사용하여 성능을 향상 시켜보려 하며, 이번 숙제에서는 shared memory를 사용하지 말것.