

Parallel Development Community (PDC) Study

Introduction to CUDA

Sang-Soo Park

September 6th, 2020

GPGPU (General Purpose Graphic Processing Unit)

■ Manycore Architecture

NVIDIA TITAN RTX D6 24GB

TITAN RTX / 12nm / 1350 MHz, 부스트 1770MHz / 4608개 / PCIe3.0x16 / GDDR6(DDR6) / 14000MHz / 24GB / 384-bit / HDMI / DP / USB Type-C / 최대 280W / 정격파워 650W 이상 / 2개 팬



최저가

3,399,000원

최저가 구매하기

딜러 가격

	Titan RTX	Quadro RTX 6000	RTX 2080 Ti
GPU	TU102	TU102	TU102
CUDA cores	4,608	4,608	4,352
Tensor cores	576	576	544
RT cores	72	72	68
Base clock	1,350MHz	1,455MHz	1,350MHz
Boost clock	1,770MHz	1,770MHz	1,635MHz
Video memory	24GB GDDR6	24GB GDDR6	11GB GDDR6
Memory interface	384-bit	384-bit	352-bit
Bandwidth	672GB/s	672GB/s	616GB/s
TDP	280W	260W	260W
Price	\$2,499 £2,399	\$6,300 £5,850	\$1,199 £1,099

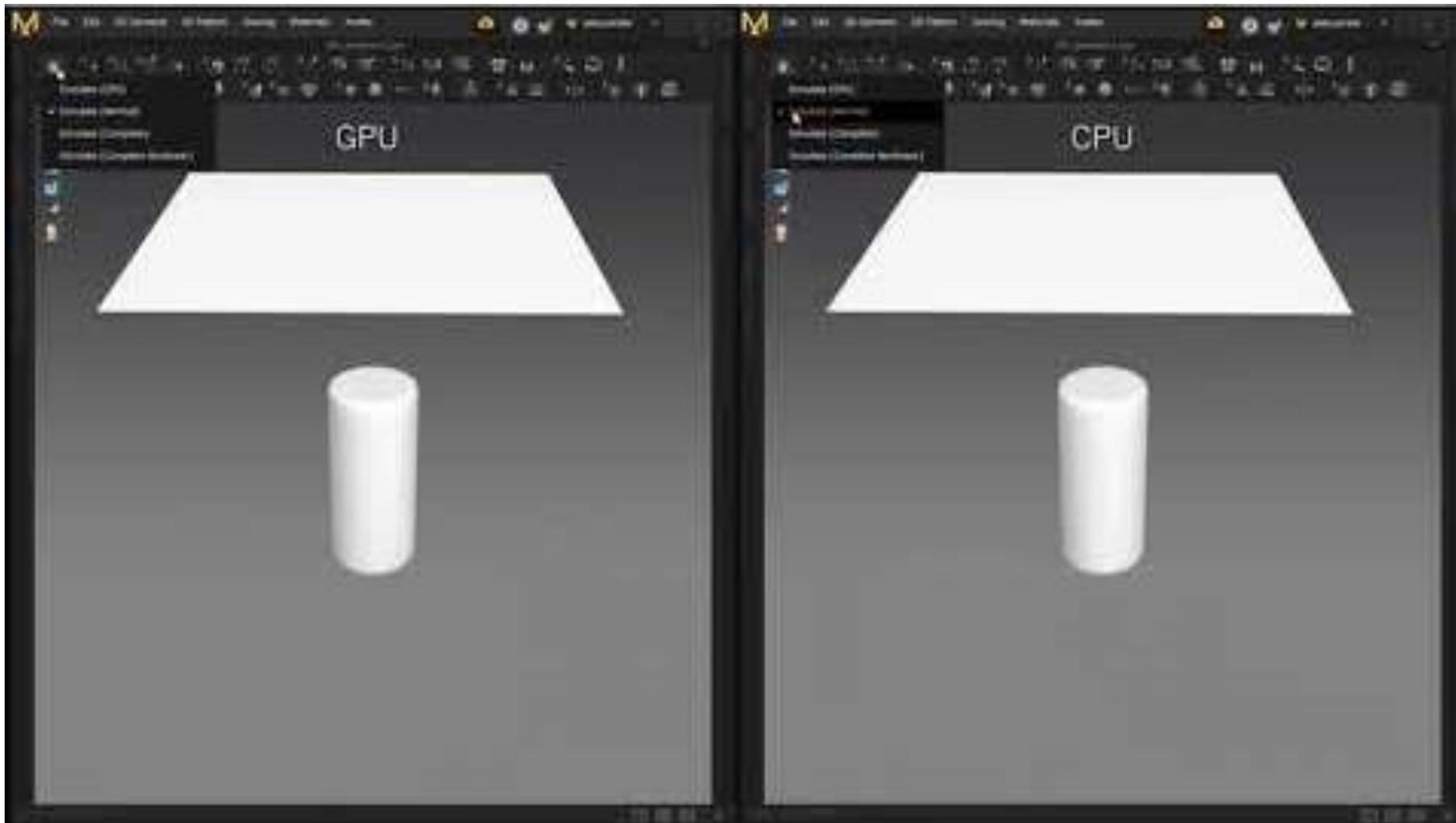
GPGPU (General Purpose Graphic Processing Unit)

- Game Application



GPGPU (General Purpose Graphic Processing Unit)

- Physics Simulation



GPGPU (General Purpose Graphic Processing Unit)

■ 컴퓨터 그래픽스 연산에 사용되는 GPU를 범용적인 어플리케이션에 적용

- Definition in Wikipedia
- [Parallel processing](#)

General-purpose computing on graphics processing units

From Wikipedia, the free encyclopedia

General-purpose computing on graphics processing units (GPGPU, rarely GPGP) is the use of a [graphics processing unit \(GPU\)](#), which typically handles computation only for [computer graphics](#), to perform computation in applications traditionally handled by the [central processing unit \(CPU\)](#).^{[1][2][3][4]} The use of multiple video cards in one computer, or large numbers of graphics chips, further parallelizes the already parallel nature of graphics processing.^[5] In addition, even a single GPU-CPU framework provides advantages that multiple CPUs on their own do not offer due to the specialization in each chip.^[6]

Essentially, a GPGPU pipeline is a kind of parallel processing between one or more GPUs and CPUs that analyzes data as if it were in image or other graphic form. While GPUs operate at lower frequencies, they typically have many times the number of cores. Thus, GPUs can process far more pictures and graphical data per second than a traditional CPU. Migrating data into graphical form and then using the GPU to scan and analyze it can create a large speedup.

GPGPU pipelines were developed at the beginning of the 21st century for [graphics processing](#) (e.g., for better [shaders](#)). These pipelines were found to fit [scientific computing](#) needs well, and have since been developed in this direction.

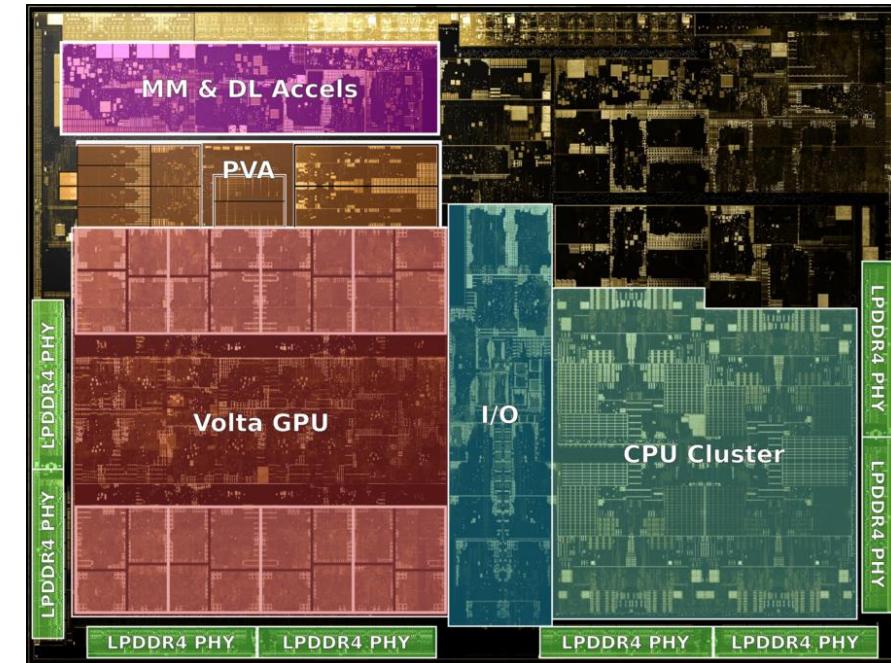
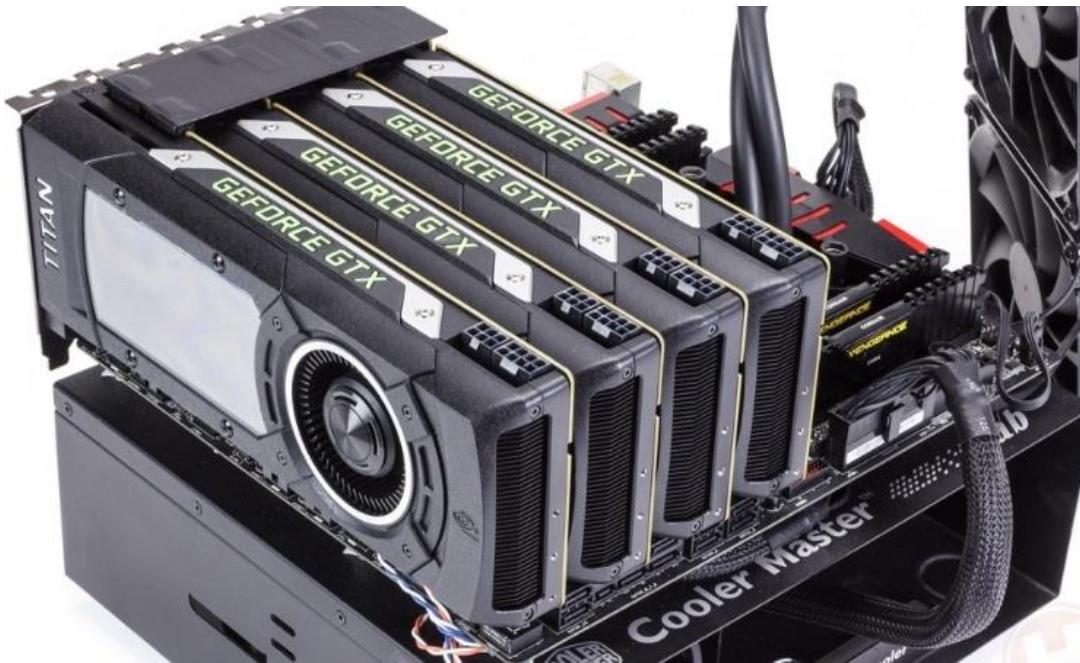
목차

- GPGPU를 위한 가속 시스템
- GPU History and Architecture
- CUDA #1 (개념 및 Thread Model)
- CUDA #2 (Execution/Memory Model)

GPGPU를 위한 다양한 가속 시스템

■ 가속기가 시스템에 연결되는 방식에 따른 구분

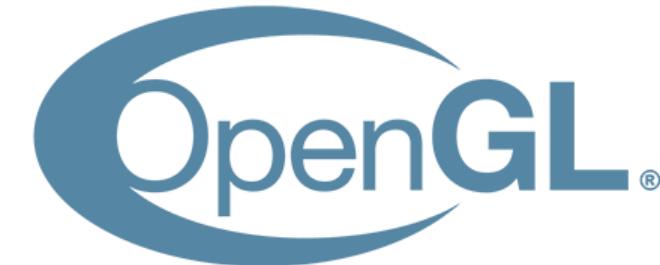
- 서버 시스템: PCI-Express에 연결하여 동작하는 방법
- 임베디드 시스템: 칩 내부에서 CPU의 작업을 도와주는 보조 프로세서 (Co-processor)



GPGPU를 위한 다양한 가속 시스템

■ GPGPU 프레임워크의 종류에 따른 구분

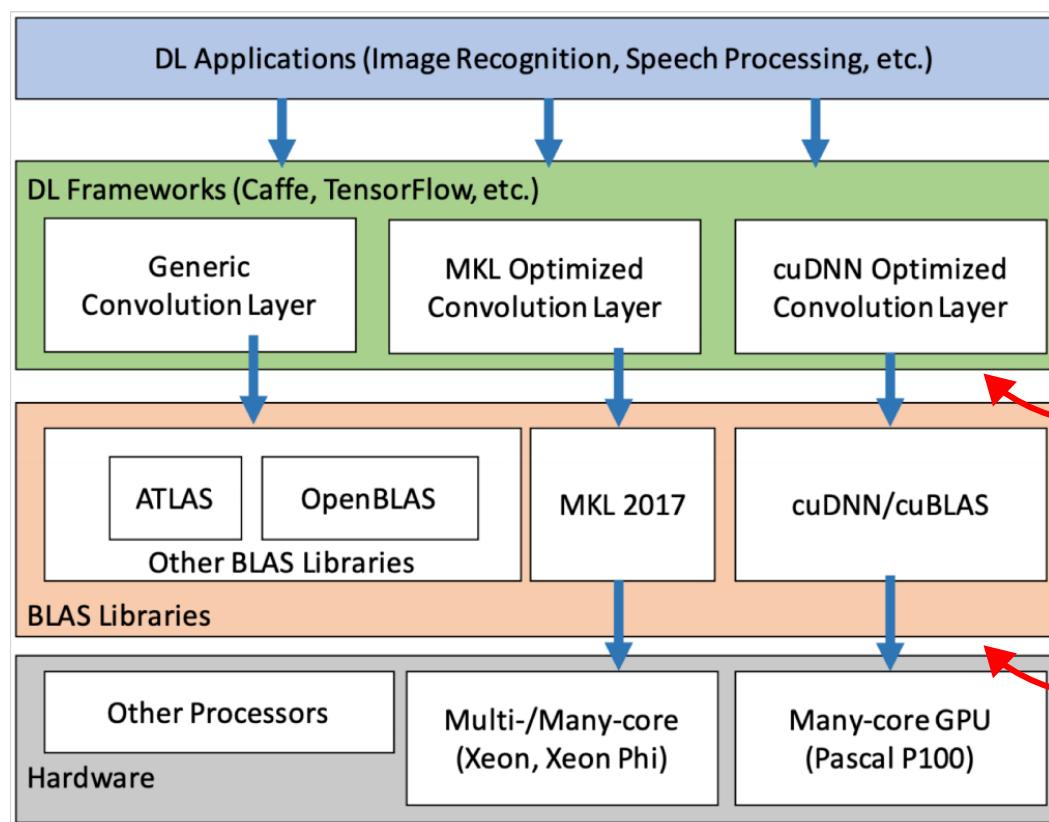
- CUDA: NVIDIA GPU에서만 동작하는 GPGPU 프레임워크
- OpenCL: CPU, GPU, FPGA (TPU) 등 다양한 가속기에서 동작하는 프레임워크



GPGPU를 위한 다양한 가속 시스템

■ 딥러닝 프레임워크에서의 GPGPU

- 프레임워크에서 모델을 정의하기 위해서 Python 사용
- Python으로 작성된 코드는 프레임워크 내부 (Backend)에서 CUDA 기반의 가속 라이브러리 사용



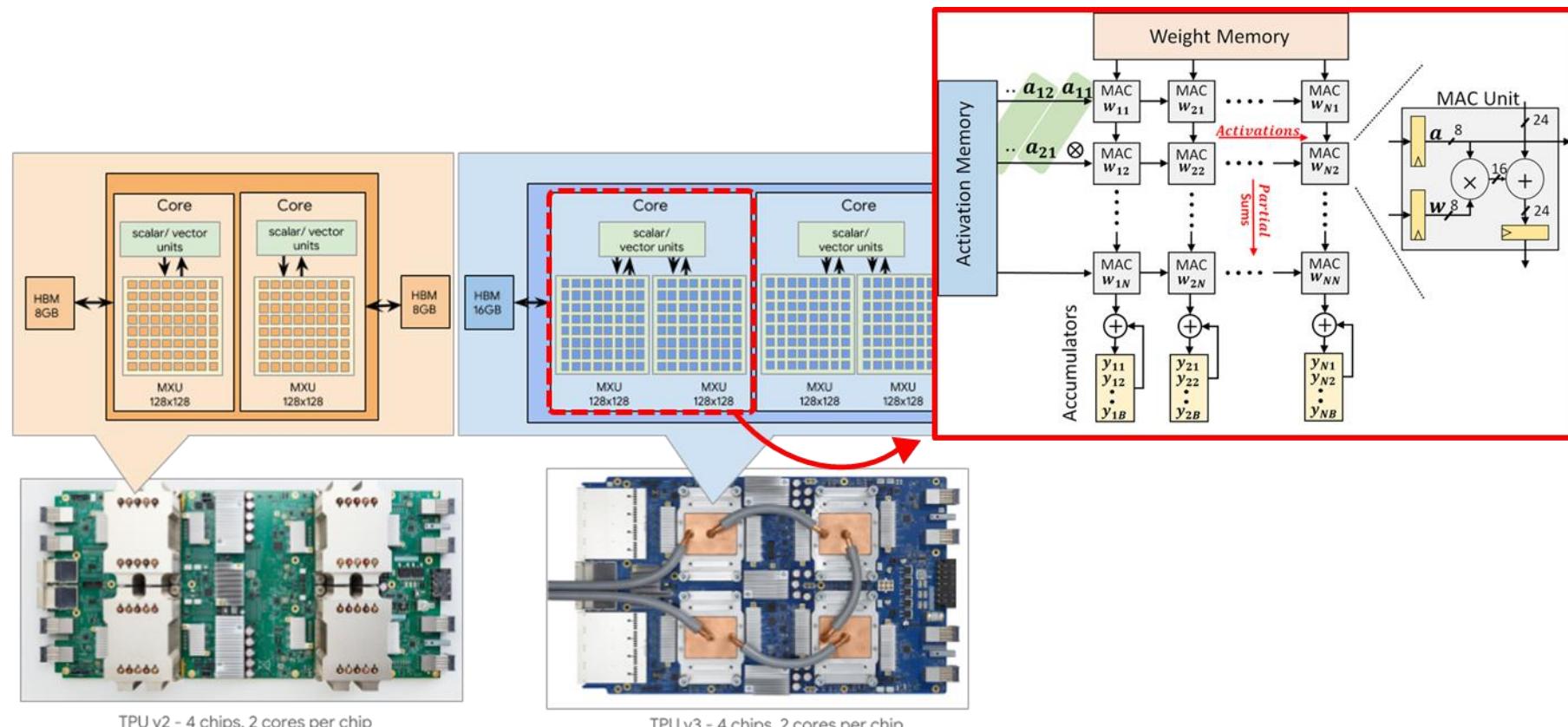
```
1 # Variables Init
2 init = tf.initialize_all_variables()
3
4 # Session Run
5 sess = tf.Session()
6 sess.run(init)
7
8 # Training
9 for step in xrange(0, 201):
10    sess.run(train)
11    if step % 20 == 0:
12        print step, sess.run(W), sess.run(b)
```

```
1 __global__ void VecAdd(float* A, float* B, float* C)
2 {
3     int i = blockDim.x * blockIdx.x + threadIdx.x;
4     C[i] = A[i] + B[i];
5 }
6
7 int main()
8 {
9 ...
10    // Kernel invocation with N threads
11    VecAdd<<<M, N>>>(A, B, C);
12 ...
13 }
```

GPGPU를 위한 다양한 가속 시스템

Tensor Processing Unit (TPU)

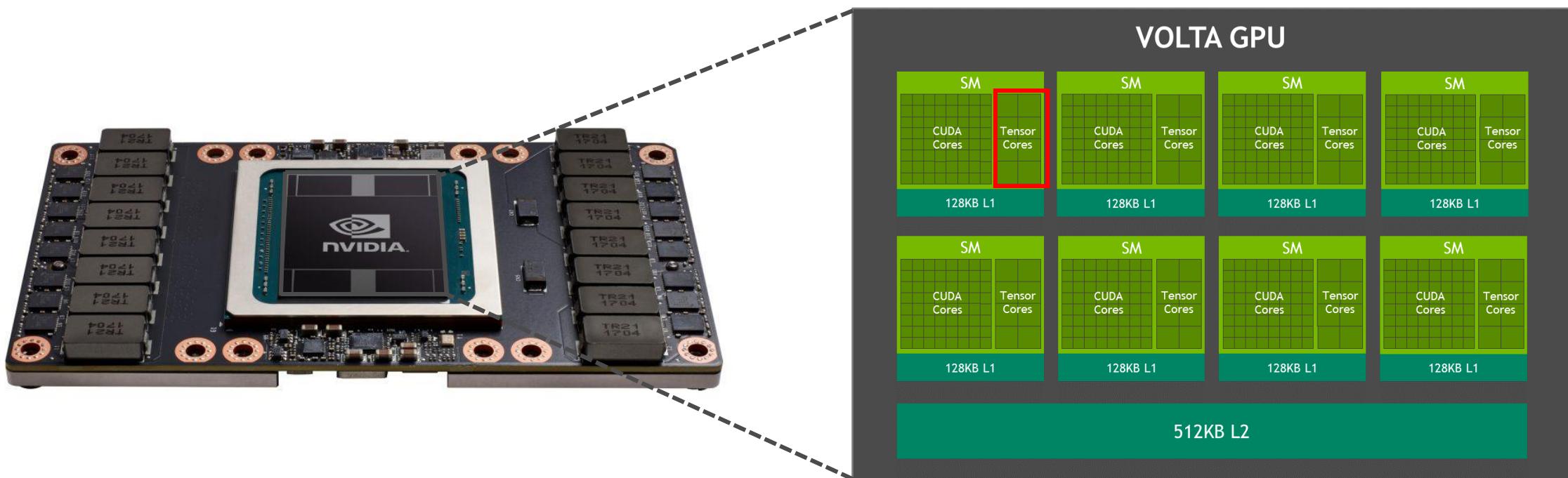
- 딥러닝 연산에서 자주 사용되는 행렬/벡터의 곱셈에 특화된 전용 장치
- 많은 개수의 Multiply and Accumulate (MAC) 연산기를 포함하는 구조



GPGPU를 위한 다양한 가속 시스템

■ GPU 내부의 텐서코어 (Tensor Core)

- 4x4 크기의 행렬 곱셈을 위한 전용 하드웨어 (TPU)
- 4x4 행렬을 계산하기 위해서 기존 GPU는 수천 cycle이 소요
 - 텐서 코어를 사용하여 4x4 행렬의 계산을 1 cycle에 해결



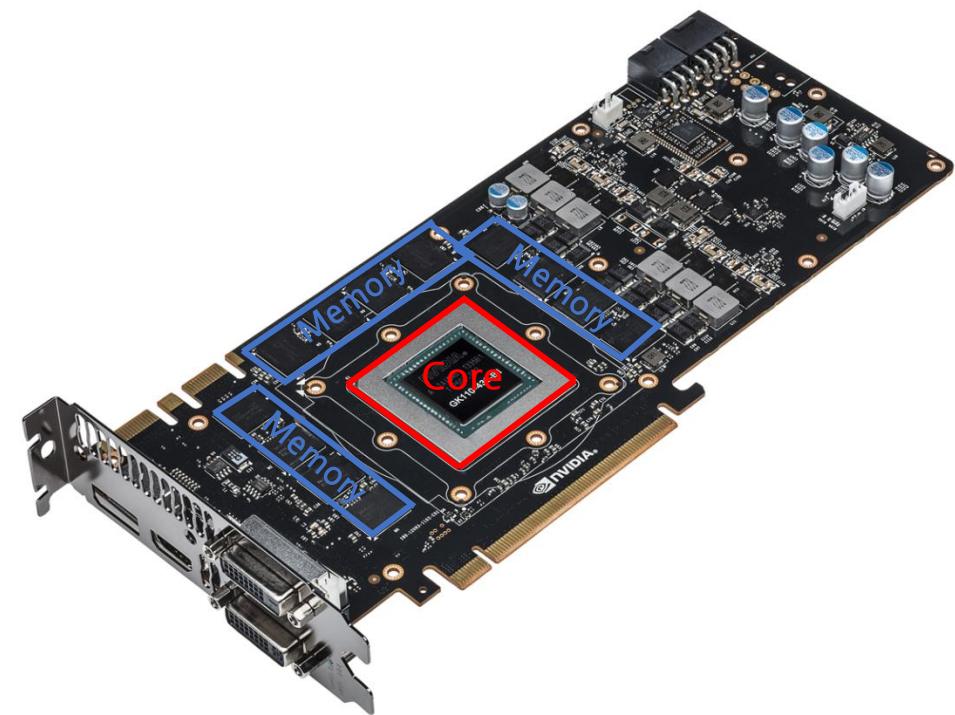
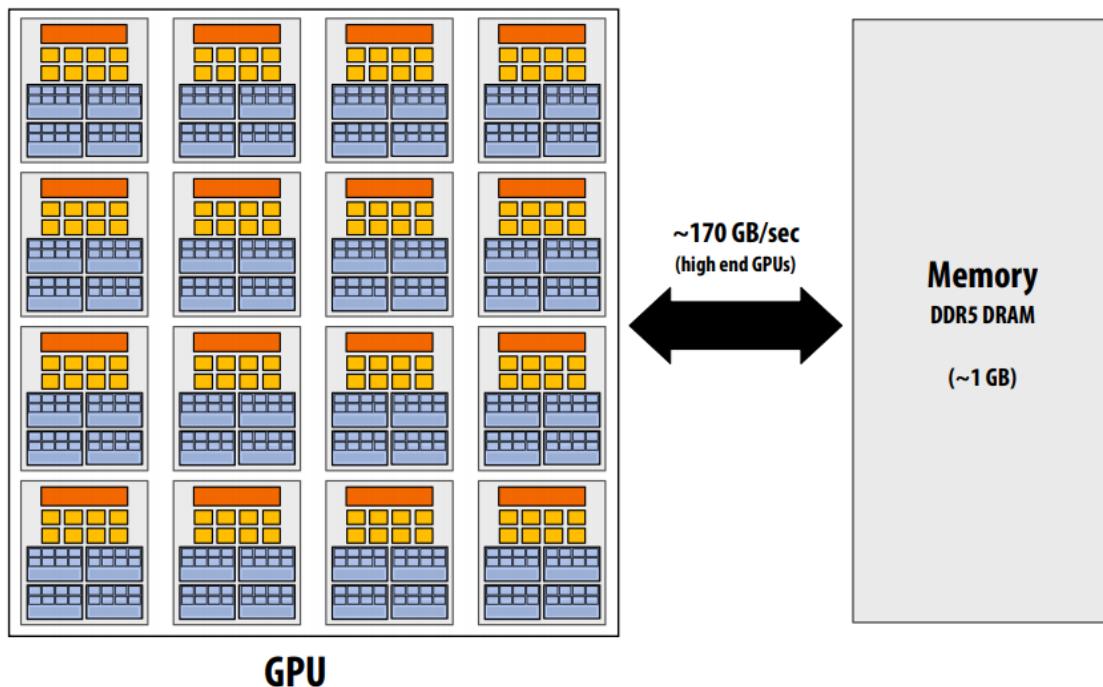
목차

- GPGPU를 위한 가속 시스템
- **GPU History and Architecture**
- CUDA #1 (개념 및 Thread Model)
- CUDA #2 (Execution/Memory Model)

Basic GPU Architecture

■ 높은 메모리 대역폭으로 동작하는 GPU

- Ryzen Threadripper 2990WX (32C, 87.42GiB/s), i9-9980XE (18C, 79.47GiB/s)
- Titan Volta (5,120C, 625.8GB/s), Titan RTX (4,608C, 672 GB/s)



3D Rendering

■ 초창기의 GPU는 Rendering을 위해 고안됨

- Rendering: 컴퓨터 프로그램을 사용하여 모델로부터 영상을 만들어내는 과정

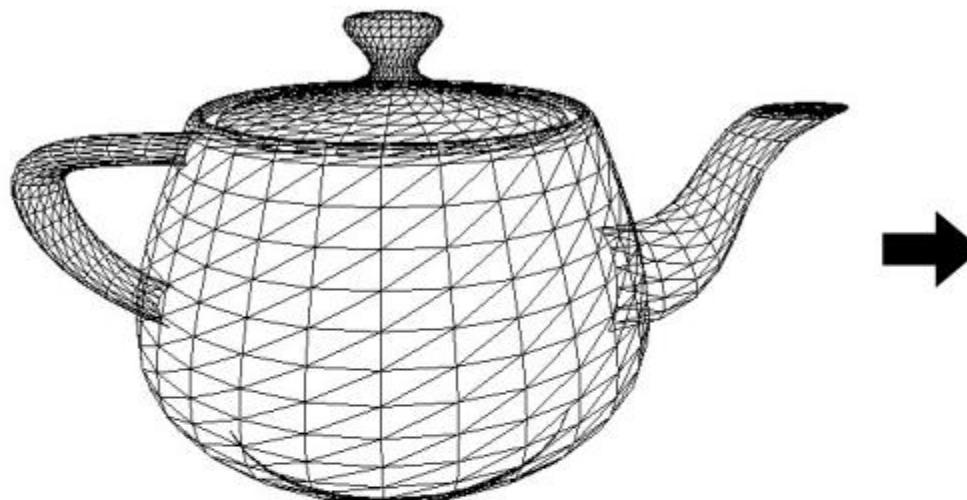


Image credit: Henrik Wann Jensen

Model of a scene:

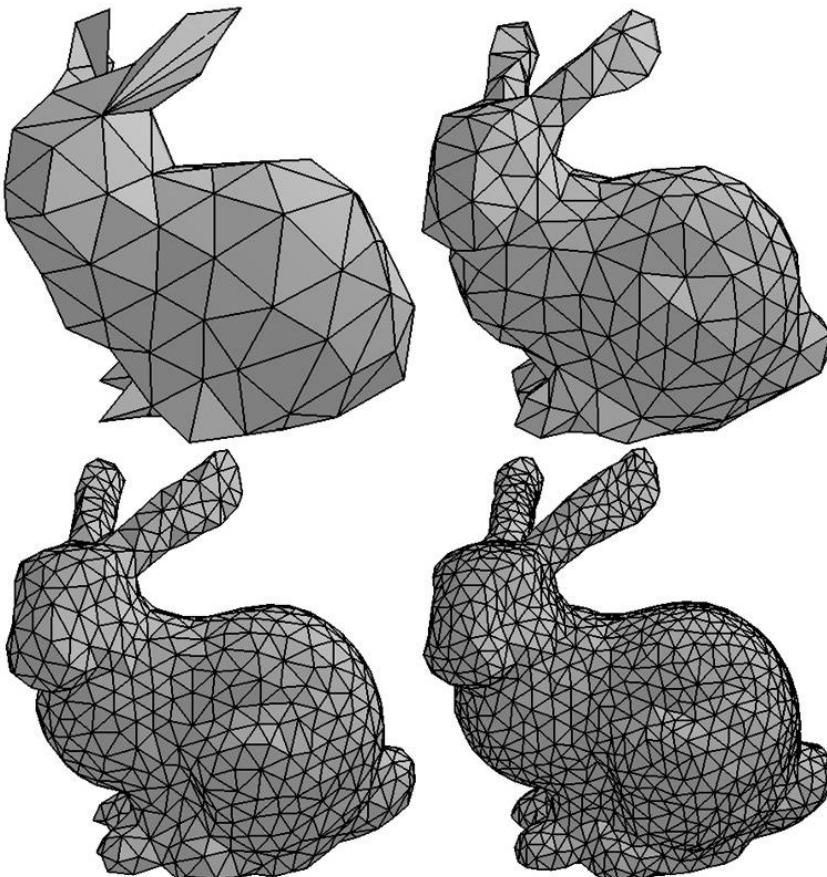
3D surface geometry (e.g., triangle mesh)
surface materials
lights
camera

Image

3D Rendering

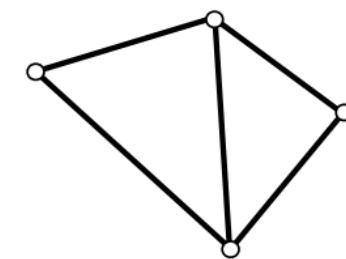
■ Real-time Graphics Primitives

- Rendering: 컴퓨터 프로그램을 사용하여 모델로부터 영상을 만들어내는 과정
- 폴리곤 (물체의 형태)과 텍스처 (물체 이미지, 표면)을 합쳐서 3D Rendering

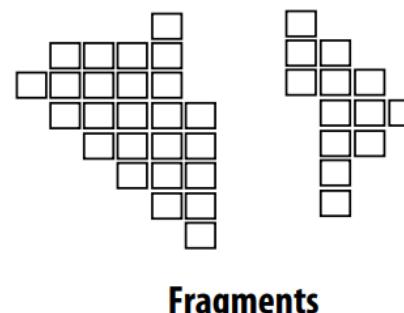


○ 1
○ 2
○ 3
○ 4

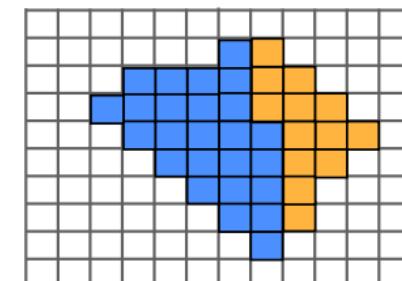
Vertices



Primitives
(triangles, points, lines)



Fragments

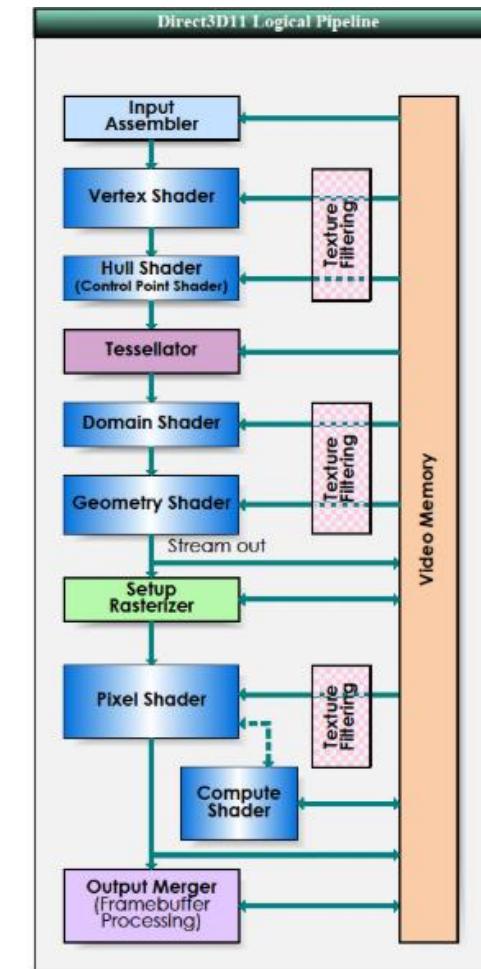
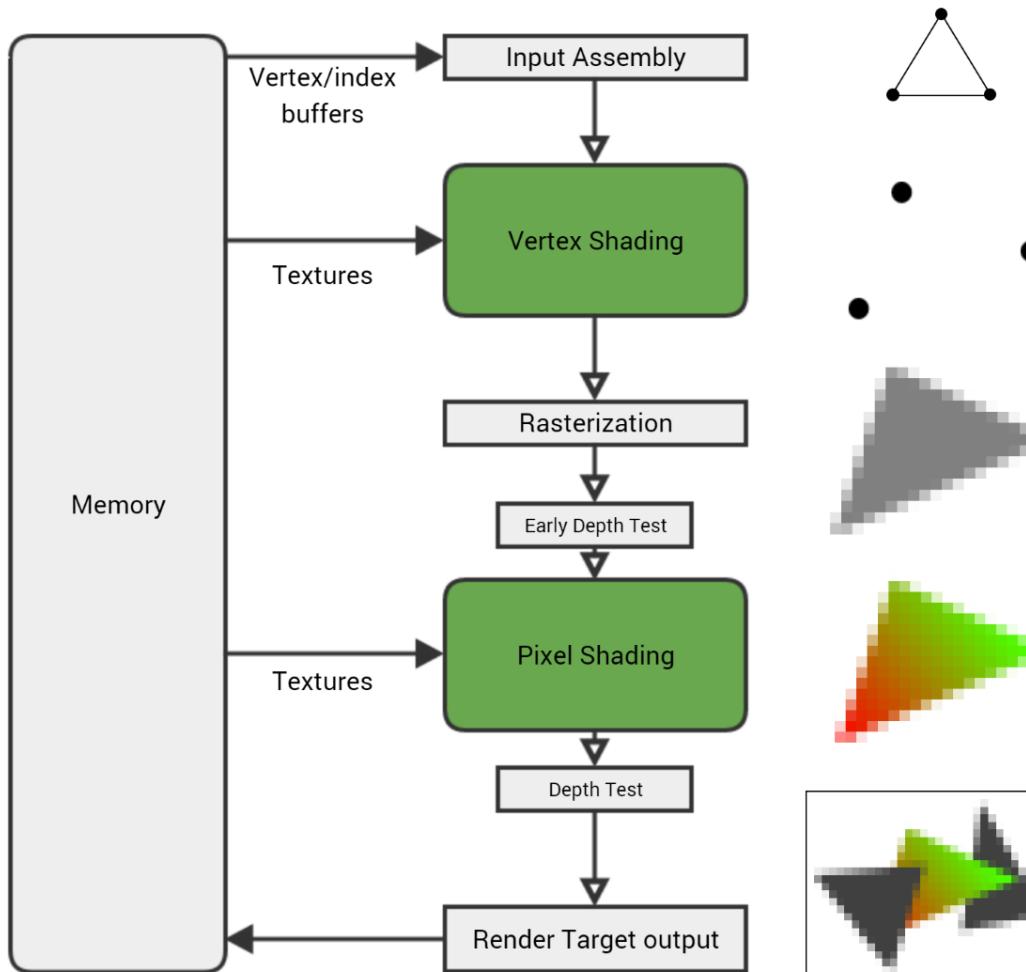


Pixels

3D Rendering

■ 독립적인 연산 장치로 구성된 GPU 구조

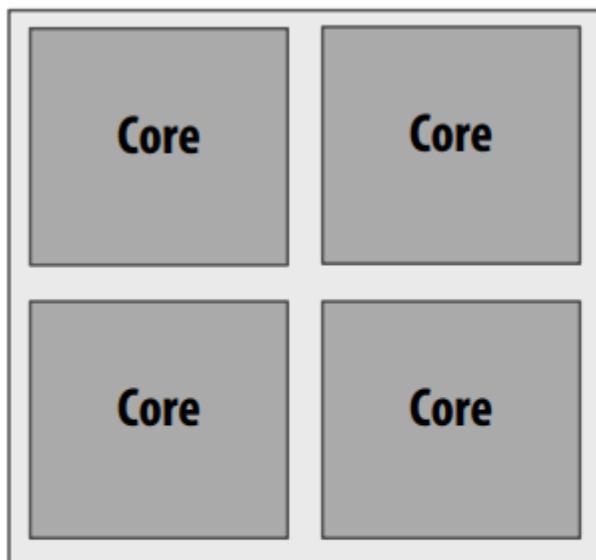
- Vertex/Pixel Shader Core: 서로 다른 기능을 수행하는 연산 장치로 구성



NVIDIA TESLA Architecture (2007)

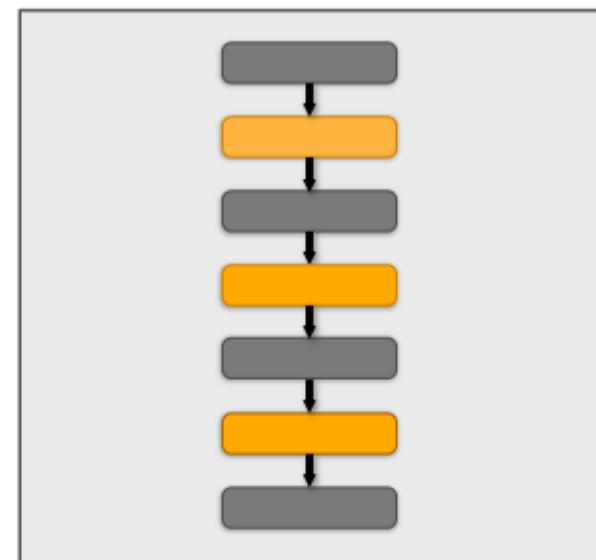
■ Non-graphic Specific Software Interface to GPU

- Multi-core CPU architecture: multi-thread, multi-process programming
- Pre-2007 GPU architecture : Separate shader programming
- Post-2007 “compute mode” GPU architecture: Set kernel program (Unfired shader)



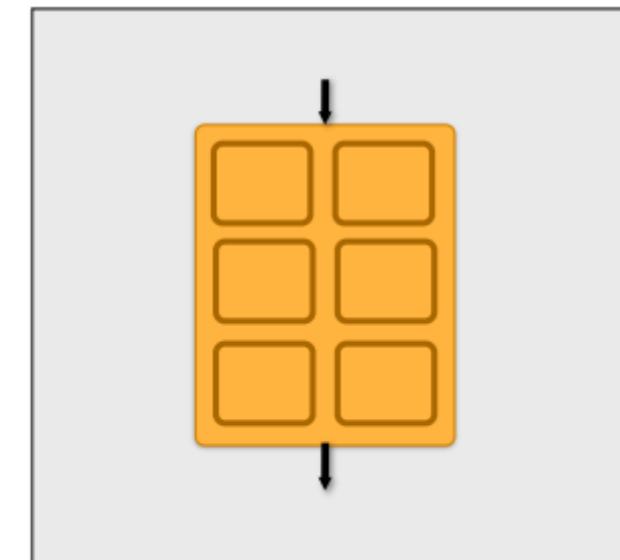
Multi-core CPU architecture

CPU presents itself to system software
(OS) as multi-processor system



Pre-2007 GPU architecture

GPU presents following interface**
to system software (driver):



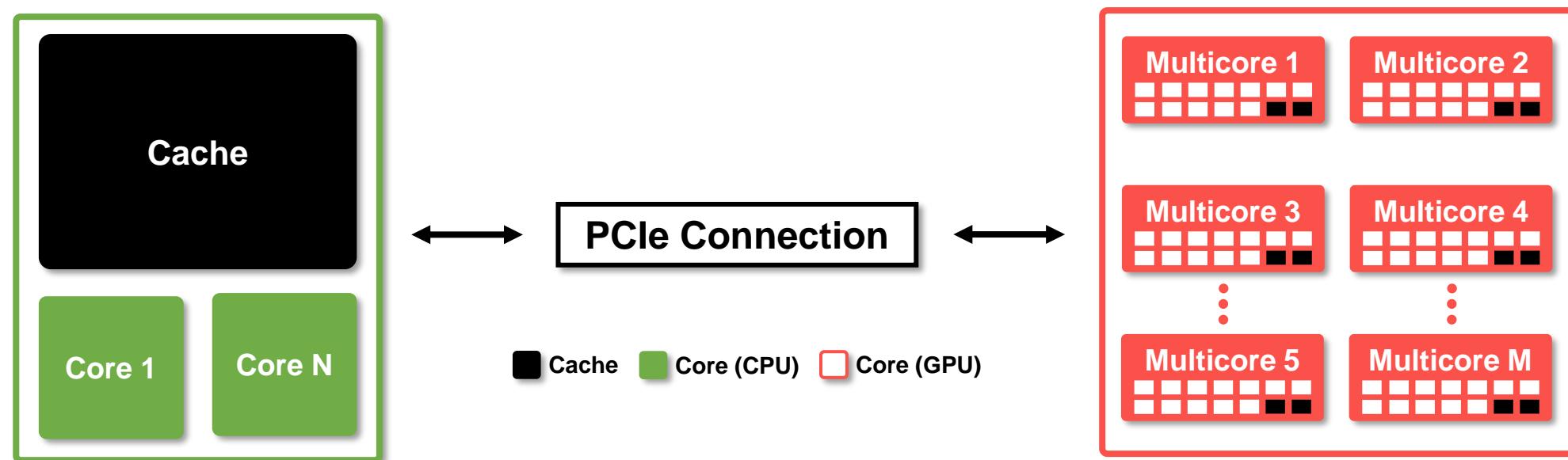
Post-2007 “compute mode”

GPU architecture

CPU와 GPU의 차이점

■ 캐시와 연산장치의 상이한 구성 정

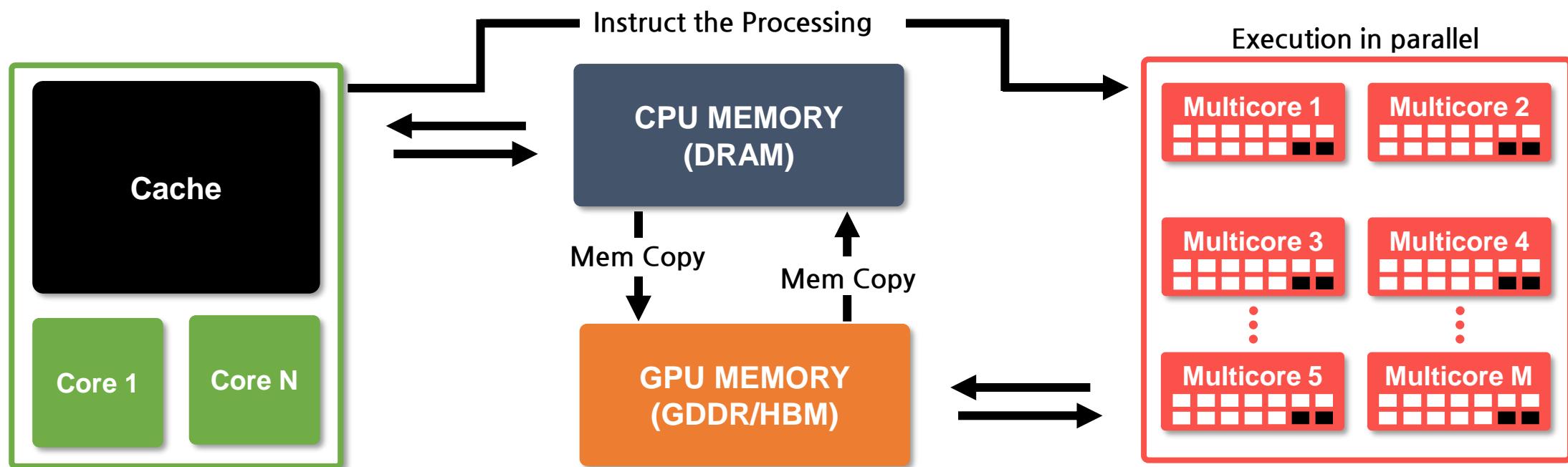
- CPU: 큰 크기의 캐시, 작은 크기의 연산 장치
- GPU: 작은 크기의 캐시, 큰 크기의 연산 장치
 - 성능은 좋지 않지만, 다수의 연산 장치로 구성
 - 여러 개의 연산 장치가 하나의 제어 장치를 공유하는 구조



GPU에서 Application을 가속하는 과정

■ 메모리의 할당/복사, 연산으로 구성된 동작 과정

- 메모리 할당과 복사: CPU (Host) to GPU (Device) or GPU (Device) to CPU (Host)
- 작업 실행 (Instruct the processing)
- 병렬처리 (Execution in parallel)



PCI Express 3.0 Host Interface

GigaThread Engine



목차

- GPGPU를 위한 가속 시스템
- GPU History and Architecture
- **CUDA #1 (개념 및 Thread Model)**
- **CUDA #2 (Execution/Memory Model)**

Compute Unified Device Architecture (CUDA)

■ C와 유사한 언어를 사용하여 GPU에서 실행되는 프로그램을 표현하는 인터페이스

▪ NVIDIA Tesla architecture (2007)년 부터 사용 가능

- 초기에는 C/C++만 지원하였으나 지금은 포트란, C# 등 다양한 언어에서 사용 가능
- Device Code: GPU에서 동작하는 코드
- Host Code: Host (CPU)에서 동작하는 코드

Normal Code

```
void c_hello() {  
    printf("Hello World!\n");  
}  
  
int main() {  
    c_hello();  
    return 0;  
}
```

CUDA Code

```
__global__ void cuda_hello () {  
    printf("Hello World from GPU!\n");  
}
```

Device Code

```
int main() {  
    cuda_hello<<<1,1>>>();  
    return 0;  
}
```

Host Code

CUDA: C/C++ Keywords

■ 함수에 해당되는 Kernel

- Kernel은 실행되는 환경에 따라 커널의 시작 부분이 상이함
 - `_global_`: 호스트에 의해 호출이 되는 디바이스 코드 (Kernel)
 - `_host_`: 호스트에서 동작하는 코드 (Default)
 - `_device_`: 디바이스에서 동작하는 코드, 디바이스에 의해 호출
- 디바이스 코드는 특정 레이블을 사용하여 데이터 수준의 병렬화 가능
 - `<<<A, B>>>`: 호스트가 디바이스 코드를 호출
 - Execution configuration syntax
 - A, B의 값에 따라 실행하는 스레드의 개수와 형태를 조절

```
_global_ void cuda_hello () {
    printf("Hello World from GPU! \n");
}

int main() {
    cuda_hello<<<1,1>>>();
    printf("Hello CUDA \n");
    return 0;
}
```

Kernel

CUDA: Hello CUDA (Example #1)

■ 다음의 Command를 사용하여 컴파일

- nvcc cuda_hello.cu
- CUDA 프로그램을 컴파일하기 위해서 nvcc 컴파일러 사용

```
#include <stdio.h>
#include <cuda.h>

__global__ void helloCUDA (int number) {
    printf("Hello CUDA frm GPU (%d), (%d)\n", threadIdx.x, number);
}

int main (void) {
    printf("Hello GPU fron CPU !\n");
    helloCUDA<<<1, 10>>>(1000);
    cudaDeviceSynchronize();
    return 0;
}
```

CUDA: CUDA Program

■ Host와 Device의 코드로 구성된 CUDA Program

▪ 호스트 (Host)

- CPU를 지칭, Host Memory는 시스템 메모리에 해당

▪ 디바이스 (Device)

- GPU를 지칭, Device Memory는 GPU 메모리에 해당

Device
Code

```
#include <stdio.h>
#include <cuda.h>

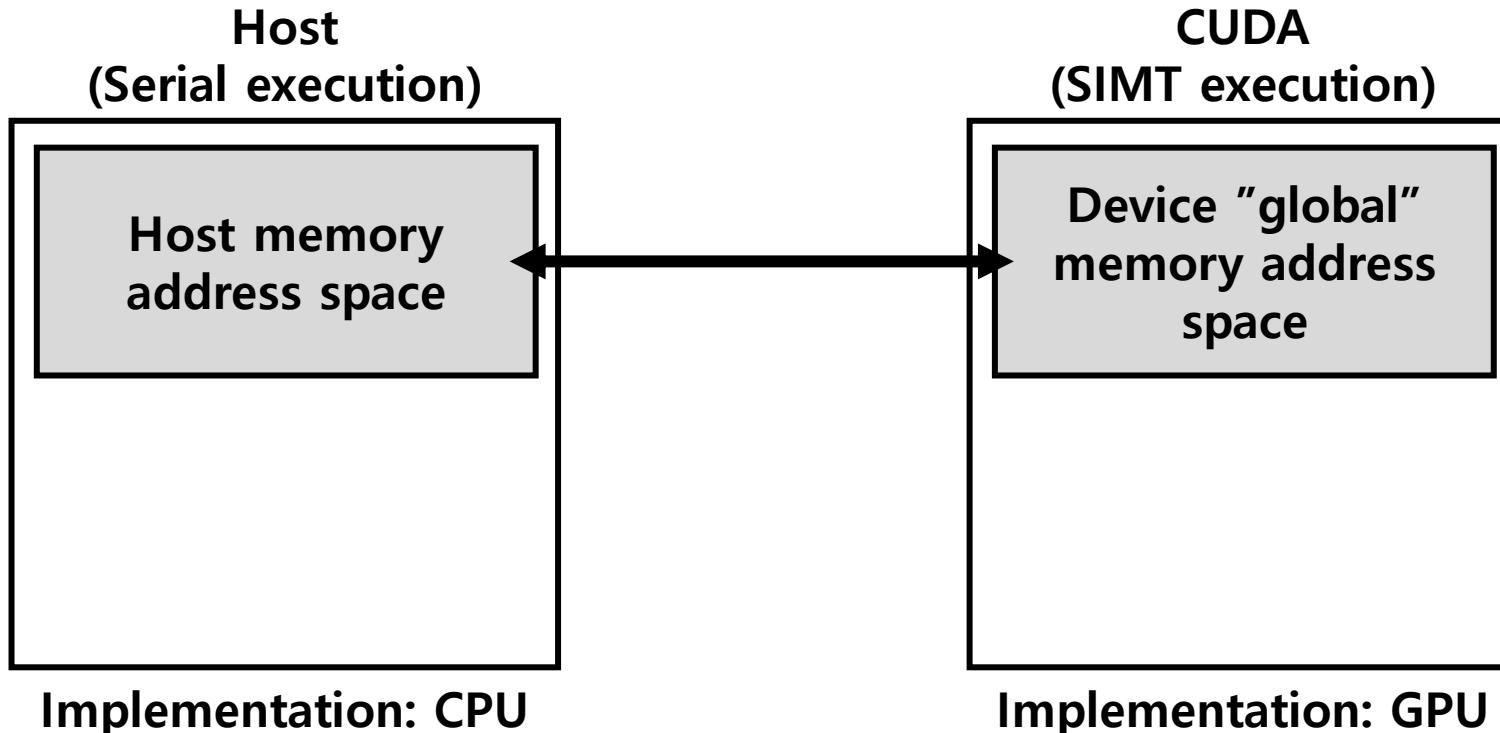
__global__ void helloCUDA (int number) {
    printf("Hello CUDA frm GPU (%d), (%d)\n", threadIdx.x, number);
}

int main (void) {
    printf("Hello GPU fron CPU !\n");
    helloCUDA<<<1, 10>>>(1000);
    cudaDeviceSynchronize();
    return 0;
}
```

Host
Code

CUDA: Device Memory

- GPU에서 연산을 하기 위해서는 메모리 복사가 사전에 수행되어야 함
 - CPU와 GPU는 물리적으로 서로 다른 메모리를 사용
 - 물리적으로 서로 다른 메모리 뿐만 아니라 서로 다른 주소의 메모리 사용



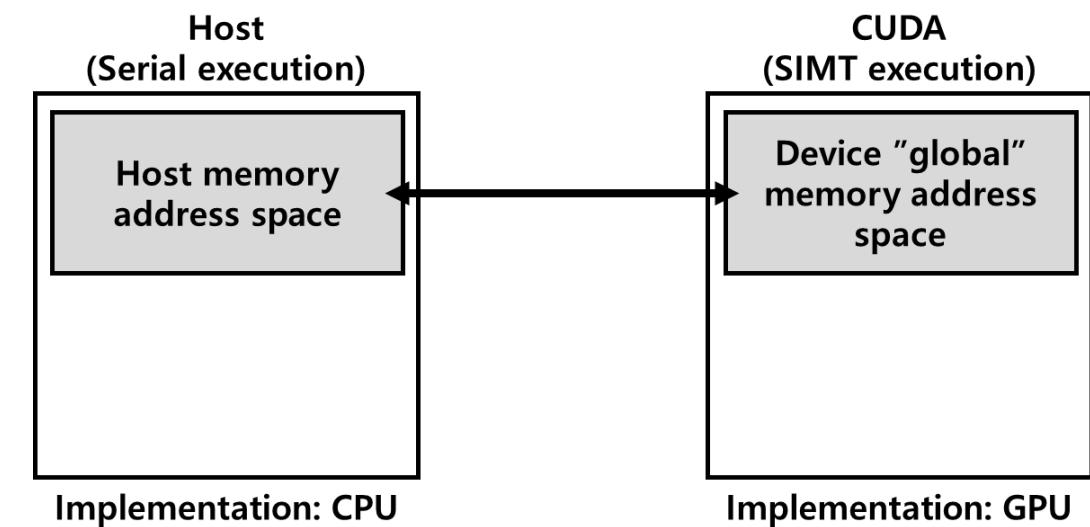
CUDA: Device Memory

Memory Copy Primitive

GPU 메모리를 사용하기 위해서는 메모리를 할당하고 복사하는 과정 필요

- 메모리 할당 (cudaMalloc), 메모리 복사 (cudaMemcpy)
- 메모리 해제 (cudaFree), 메모리 초기화 (cudaMemset)

```
float* A = new float[N]; // allocate buffer in host mem  
  
// populate host address space pointer A  
for (int i=0; i<N; i++)  
    A[i] = (float)i;  
  
int bytes = sizeof(float) * N;  
Float* deviceA;  
cudaMalloc (&deviceA, bytes);  
  
// populate deviceA;  
cudaMemcpy(deviceA, A, bytes, cudaMemcpyHostToDevice)
```



CUDA: Vector Sum

■ 벡터 데이터의 덧셈을 수행하는 Vector Sum

- gcc vector_sum.c –o vector_sum을 사용하여 컴파일
- OpenMP를 사용하여 병렬화

```
#include <stdio.h>
#define N 10000000

void vector_add(float *out, float *a, float *b, int n) {
    #pragma omp parallel for
    for(int i = 0; i < n; i++) {
        out[i] = a[i] + b[i];
    }
}

int main() {
    float *a, *b, *out;
    // Allocate memory
    a = (float*)malloc(sizeof(float) * N);
    b = (float*)malloc(sizeof(float) * N);
    out = (float*)malloc(sizeof(float) * N);
    // Initialize array
    for(int i = 0; i < N; i++) {
        a[i] = 1.0f; b[i] = 2.0f;
    }
    vector_add(out, a, b, N);
    free(a); free(b); free(c);
}
```

CUDA: Vector Sum (Example #2)

■ 벡터 데이터의 덧셈을 수행하는 Vector Sum

- nvcc vector_sum_cuda.c –o vector_sum_cuda를 사용하여 컴파일

```
#include <stdio.h>
#define N 10000000

void vector_add(float *out, float *a, float *b, int n){
    for(int i = 0; i < n; i++) {
        out[i] = a[i] + b[i];
    }
}

int main() {
    float *a, *b, *out;
    // Allocate memory
    a = (float*)malloc(sizeof(float) * N);
    b = (float*)malloc(sizeof(float) * N);
    out = (float*)malloc(sizeof(float) * N);
    // Initialize array
    for(int i = 0; i < N; i++) {
        a[i] = 1.0f; b[i] = 2.0f;
    }
    vector_add(out, a, b, N);
    free(a); free(b); free(c);
}
```

```
// Allocate device memory
cudaMalloc((void**)&d_a, sizeof(float) * N);
cudaMalloc((void**)&d_b, sizeof(float) * N);
cudaMalloc((void**)&d_out, sizeof(float) * N);
// Transfer data from host to device memory
cudaMemcpy(d_a, a, sizeof(float) * N, cudaMemcpyHostToDevice);
cudaMemcpy(d_b, b, sizeof(float) * N, cudaMemcpyHostToDevice);
// Executing kernel
vector_add<<<1,1>>>(d_out, d_a, d_b, N);
// Transfer data back to host memory
cudaMemcpy(out, d_out, sizeof(float) * N, cudaMemcpyDeviceToHost);

// Verification
for(int i = 0; i < N; i++) {
    assert(fabs(out[i] - a[i] - b[i]) < MAX_ERR);
}
printf("out[0] = %f\n", out[0]);
printf("PASSED\n");

// DeAllocate device memory
cudaFree(d_a); cudaFree(d_b); cudaFree(d_out);
```

CUDA: Performance Profiling

■ CUDA 프로그램의 실행시간 프로파일링

- **프로파일링 (Profiling)**: 프로그램의 실행시간을 측정하고, 병목의 구간을 확인하는 작업
- 실행시간 = Computation + Data Transfer Overhead

Data Transfer Time

```
// Send input data from host to device
cudaMemcpy(d_a, a, memSize, cudaMemcpyHostToDevice);
cudaMemcpy(d_b, b, memSize, cudaMemcpyHostToDevice);

// Send result from device to host
cudaMemcpy(c, d_c, memSize, cudaMemcpyDeviceToHost);
```

Computation Time

```
// Kernel call
vecAdd<<<1, NUM_DATA >>>(d_a, d_b, d_c);
```

CUDA: Kernel은 어떻게 동작하는 것일까요 ?

■ CUDA Kernel

- GPU에서 동작하는 프로그램
- CUDA Kernel이 호출될 때, 실질적인 연산이 시작

CUDA Kernel

```
__global__ void vecAdd(int *_a, int *_b, int *_c) {
    int tID = threadIdx.x;
    _c[tID] = _a[tID] + _b[tID];
}
```

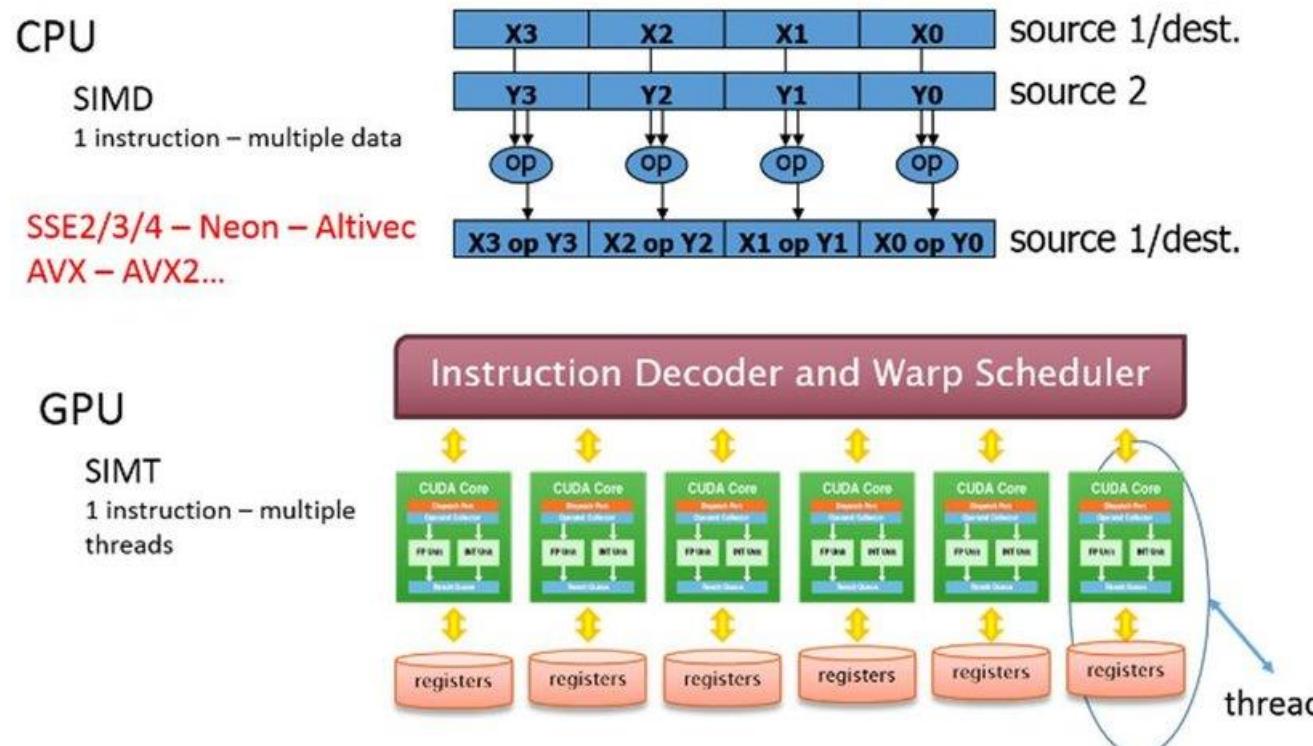
CUDA Kernel Call

```
// Kernel call
vecAdd<<<1, NUM_DATA >>>(d_a, d_b, d_c);
```

CUDA: Thread Model

■ Single Instruction Multiple Thread (SIMT) Architecture

- 하나의 명령어로 여러 개의 스레드를 실행하는 구조
- 모든 스레드는 동일한 코드 (Kernel)를 공유
 - A group of threads is controlled by a control unit (32 Threads = Warp)
 - 각 스레드는 스레드 고유의 control context를 사용



CUDA: Revisit to Vector Sum Kernel

- SIMT Architecture에서 모든 스레드는 커널 코드를 공유
 - 이러한 코드에서 얼마나 많은 Thread, Warp, Thread Block이 생성 될까 ?

```
__global__ void vecAdd(int *_a, int *_b, int *_c){  
    int tID = threadIdx.x;  
    _c[tID] = _a[tID] + _b[tID];  
}
```

CUDA: Thread Hierarchy

■ 계층적인 스레드 모델 (프로그래밍 모델)

- Thread, Thread Block, Grid는 3차원으로 사용 가능 !

▪ Thread (스레드)

- 멀티코어에서의 thread와 동일한 개념
- 하나의 thread는 하나의 CUDA Core에서 동작

▪ Warp (워프)

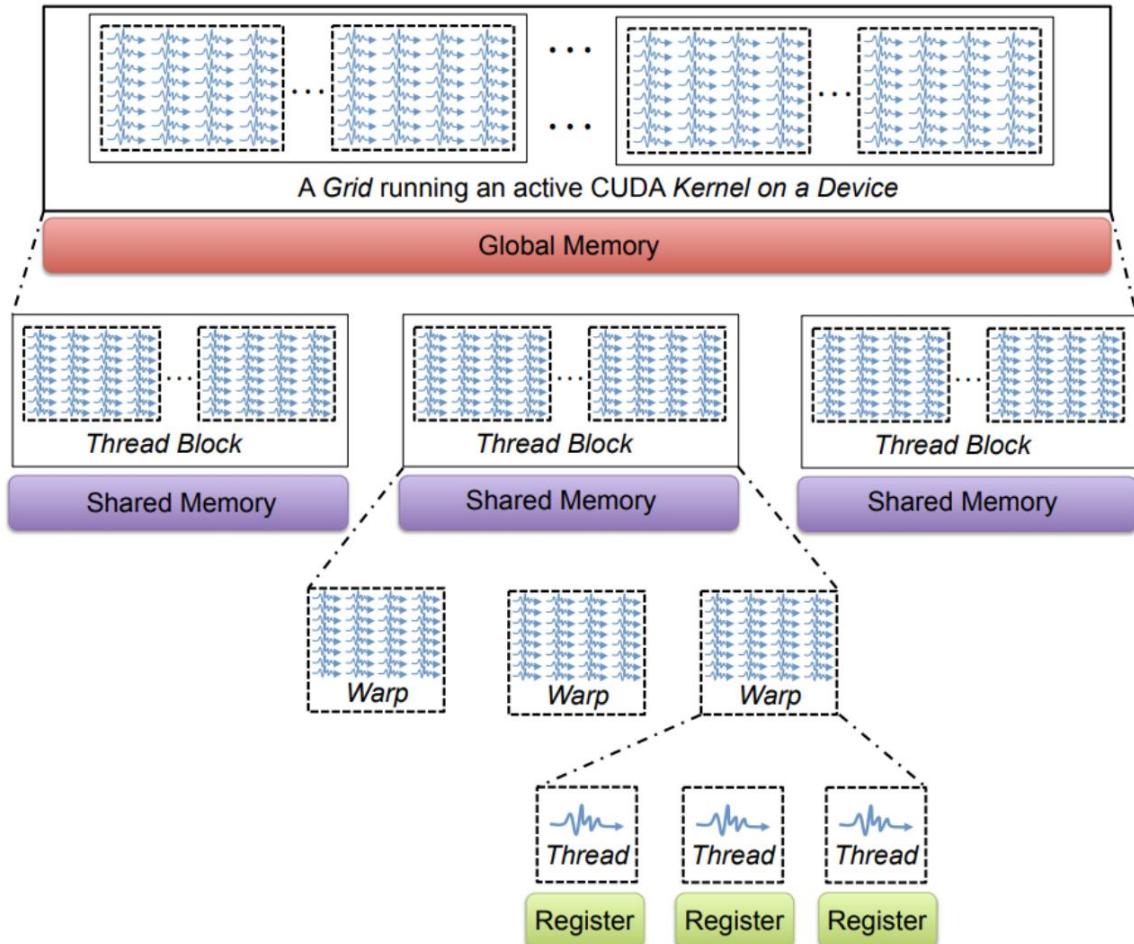
- 하드웨어에서 스레드를 처리하는 단위
- 32개의 스레드가 묶인 단위

▪ Thread Block (스레드 블록)

- 프로그래밍 모델에서 스레드를 처리하는 단위
- 여러 개의 스레드가 묶여 Thread Block

▪ Grid (그리드)

- Thread block이 여러 개 묶인 단위



CUDA: Built-in Variables for CUDA Thread Hierarchy

■ CUDA에 내장된 Kernel에서 사용되는 변수들

■ gridDim

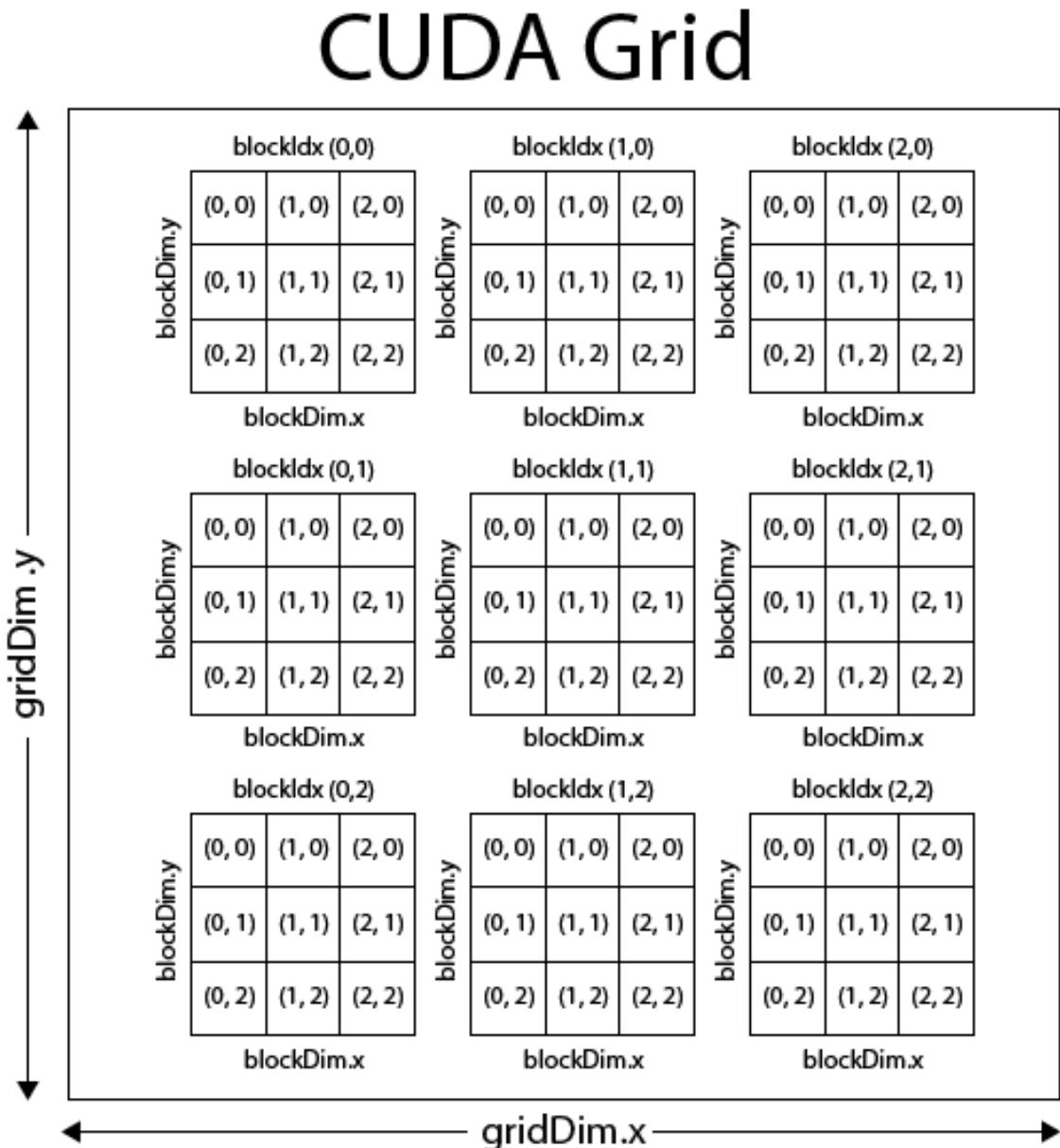
- Dimension of the current grid
- Determine the number of blocks in a grid

■ blockIdx

- Block ID of the current block
- A unique ID of a block in the grid

■ blockDim

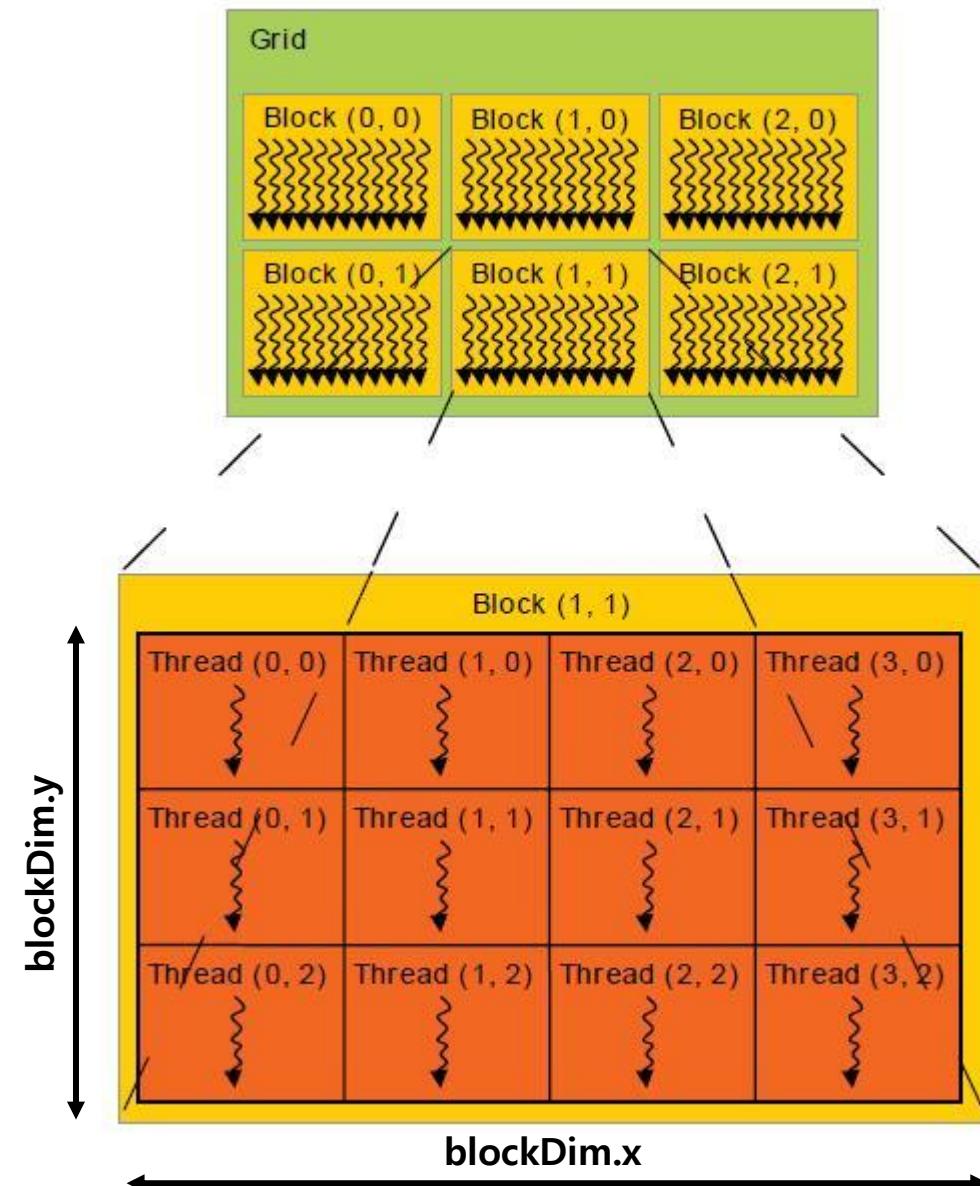
■ threadIdx



CUDA: Built-in Variables for CUDA Thread Hierarchy

■ CUDA에 내장된 Kernel에서 사용되는 변수들

- **gridDim**
- **blockIdx**
- **blockDim**
 - Dimension of the current block
 - Determine the number of threads in a block
- **threadIdx**
 - Thread ID of the current thread in the block
 - A unique ID of a thread in the block

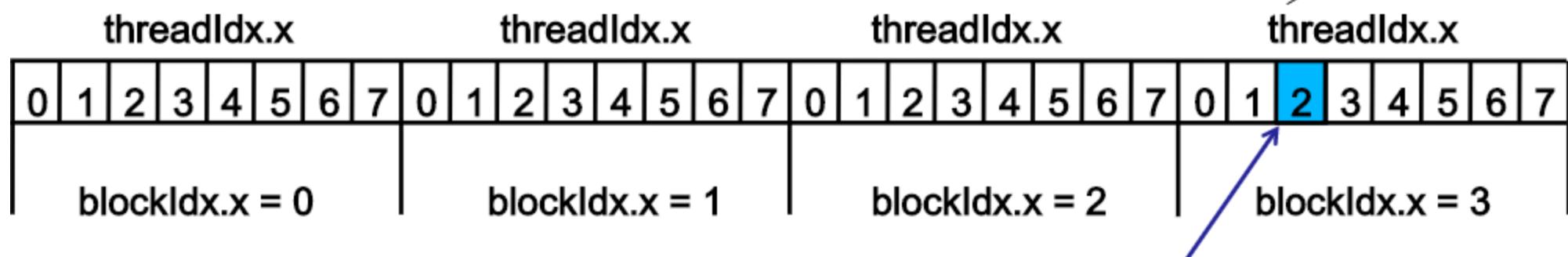


CUDA: CUDA Thread Hierarchy & Kernel Call

■ 3차원 형태로 설정 가능한 그리드, 블록, 스레드

- 1차원, 2차원, 3차원 형태로 사용 가능
- Dim3 data type
 - UInt3 타입, x, y, z 차원에 해당되는 값의 저장 가능

```
Dim3 dimGrid (4, 1, 1);
Dim3 dimBlock (8, 1, 1);
vecAdd <<<dimGrid, dimBlock>>>(d_a, d_b, d_c);
```



Global thread ID = **blockIdx.x * blockDim.x + threadIdx.x**
= $3 * 8 + 2$ = thread 26 with linear global addressing

CUDA: Thread Index Check Example (Example #3)

```
#include <stdio.h>
#include <stdlib.h>
#include <cuda.h>
#include <cuda_runtime.h>

__global__ void checkIndex(void) {
    printf("threadIdx (%d, %d, %d) blockIdx (%d, %d, %d) blockDim (%d, %d, %d) gridDim (%d, %d, %d)
\n", threadIdx.x, threadIdx.y, threadIdx.z, blockIdx.x, blockIdx.y, blockIdx.z, blockDim.x, blockDim.y, blockDim.z, gridDim.x, gridDim.y, gridDim.z);
}

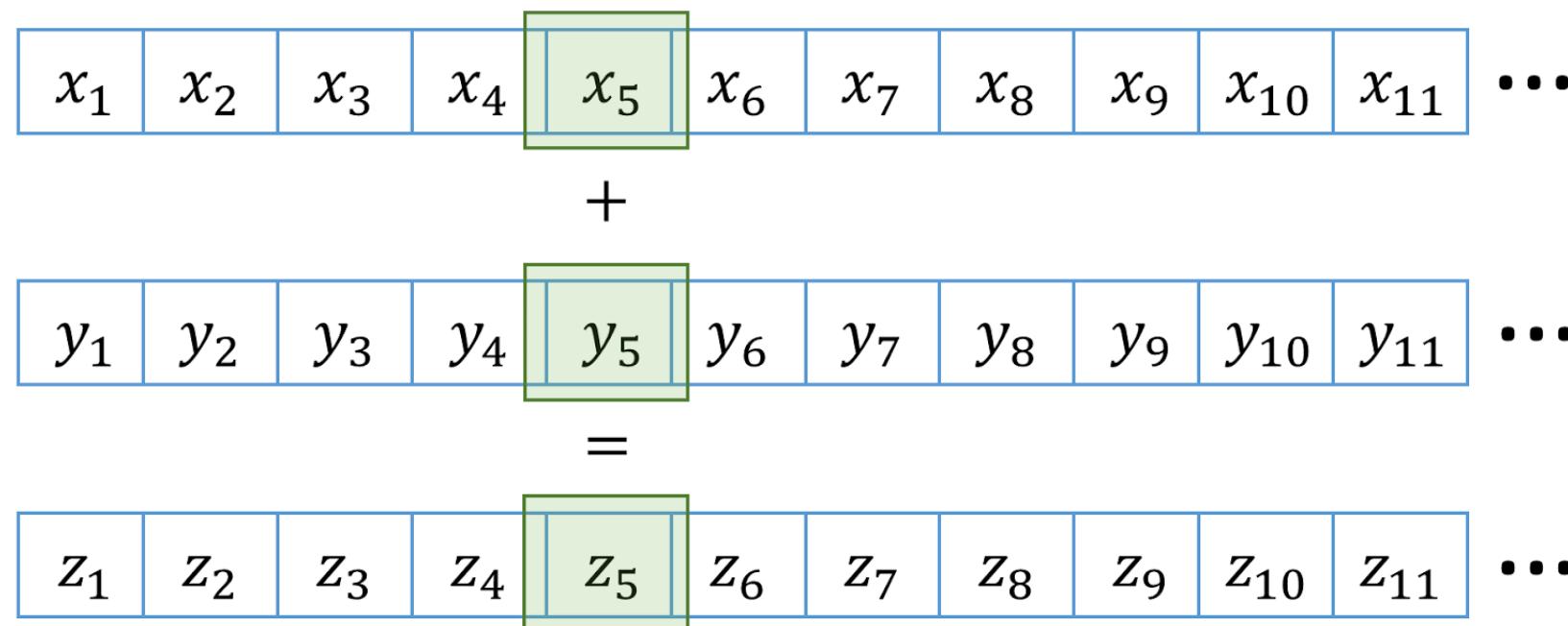
int main() {
    int nElem = 6;
    dim3 block(3);
    dim3 grid((nElem + block.x - 1) / block.x);
    printf("grid.x %d grid.y %d grid.z %d\n", grid.x, grid.y, grid.z);
    printf("block.x %d block.y %d block.z %d\n", block.x, block.y, block.z);
    checkIndex <<<grid, block>>> ();
    cudaDeviceReset();
    return(0);
}
```

Thread Block (3, 1, 1)
Grid (2, 1, 1)

CUDA: Vector Addition

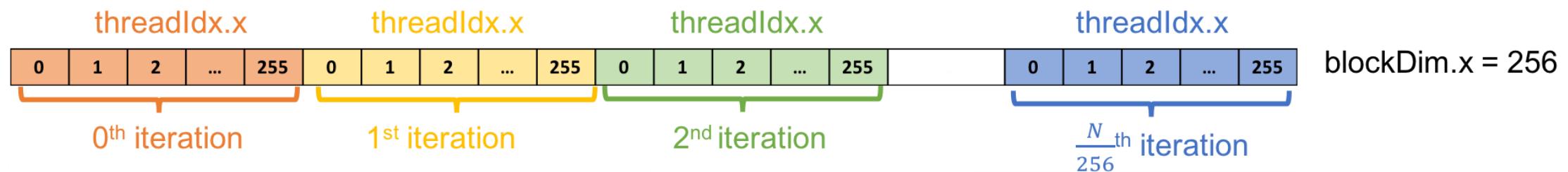
■ Use following Thread Layouts for Vector Addition Kernel

- Total computation (T), Divider (N)
- Grid $(1, 1, 1)$ with Block $(T, 1, 1)$
- Grid $(N, 1, 1)$ with Block $(T/N, 1, 1)$



CUDA: Vector Addition

- Maps Each Dimension to x Coordinates
 - $\text{Index} = \text{threadIdx.x} + \text{blockIdx.x} * \text{blockDim.x}$



CUDA: Vector Addition at Grid (1,1,1) with Block (T, 1, 1) (Example #4)

■ CUDA Kernel

```
_global_ void Vec_add(float x[], float y[], float z[], int n) {
    int thread_id = threadIdx.x;
    int block_dim = blockDim.x;
    int block_id = blockIdx.x;

    if (thread_id < n){
        z[thread_id+block_id * block_dim] = x[thread_id+block_id * block_dim] + y[thread_id+block_id * block_dim];
    }
}
```

■ Thread Layout

```
dim3 blockDim(T);
dim3 gridDim(1);
```



```
Vec_add<<<1, T>>>(d_x, d_y, d_z, n);
```

CUDA: Vector Addition at Grid (1,1,1) with Block (T, 1, 1) (Example #5)

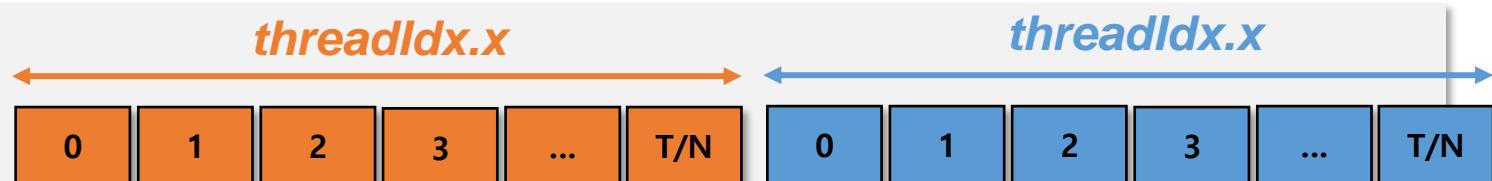
■ CUDA Kernel

```
_global_ void Vec_add(float x[], float y[], float z[], int n) {  
    int thread_id = threadIdx.x;  
    int block_dim = blockDim.x;  
    int block_id = blockIdx.x;  
  
    if (thread_id < n){  
        z[thread_id+block_id * block_dim] = x[thread_id+block_id * block_dim] + y[thread_id+block_id * block_dim];  
    }  
}
```

■ Thread Layout

```
dim3 blockDim(T/N);  
dim3 gridDim(N);
```

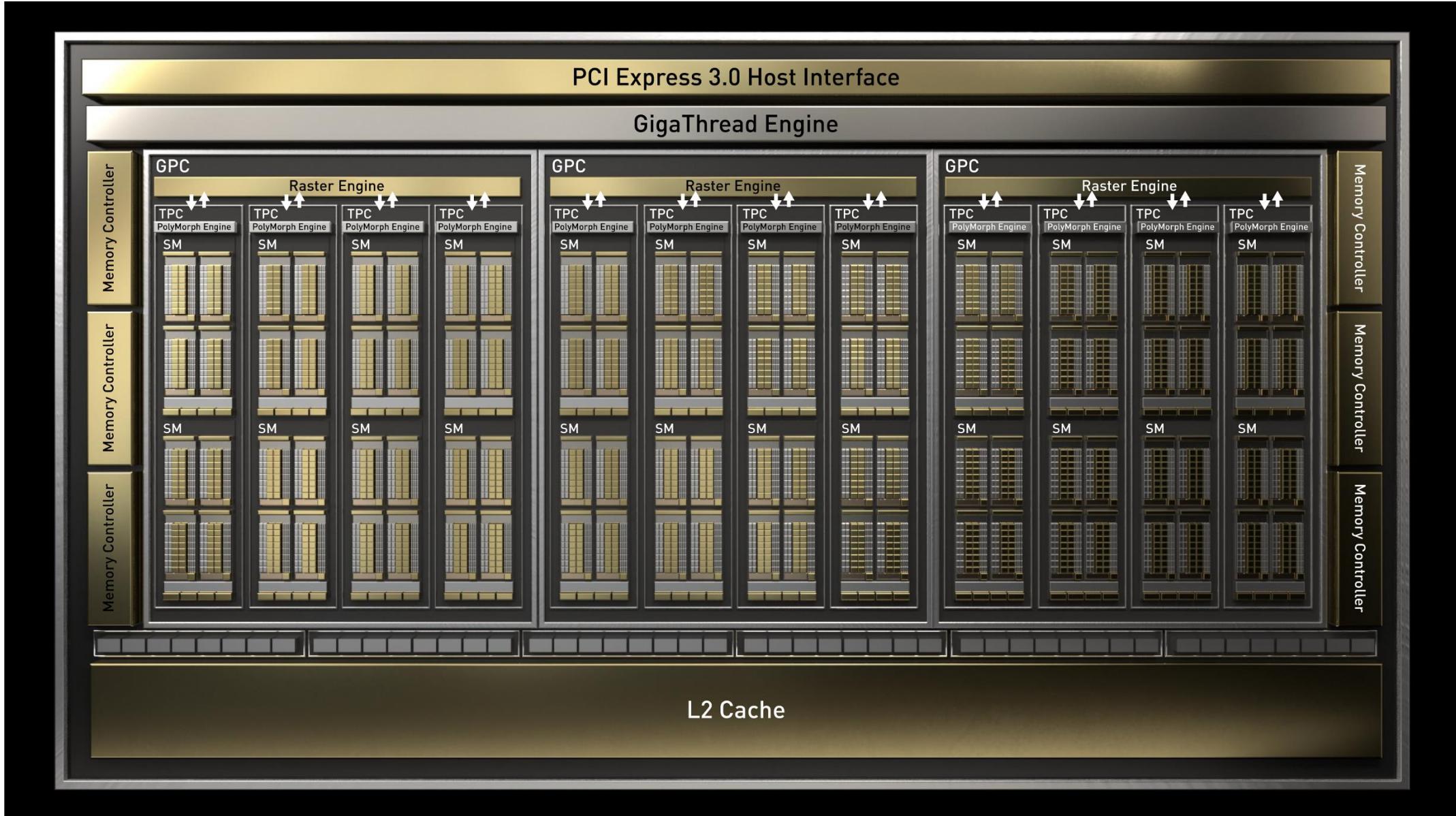
```
Vec_add<<<N, T/N>>>(d_x, d_y, d_z, n);
```



목차

- GPGPU를 위한 가속 시스템
- GPU History and Architecture
- CUDA #1 (개념 및 Thread Model)
- **CUDA #2 (Execution/Memory Model)**

CUDA: Architecture (Turing)



CUDA: Architecture (Fermi)



CUDA: Execution Model

■ 계층적인 하드웨어 구조

■ CUDA Core (Streaming Processor, SP)

- GPU 구조에서 가장 기본적인 연산 단위
- 멀티코어 CPU에서의 하나의 코어에 해당
- 하나의 스레드는 CUDA Core에 매핑됨

■ Streaming Multi-processor (SM)

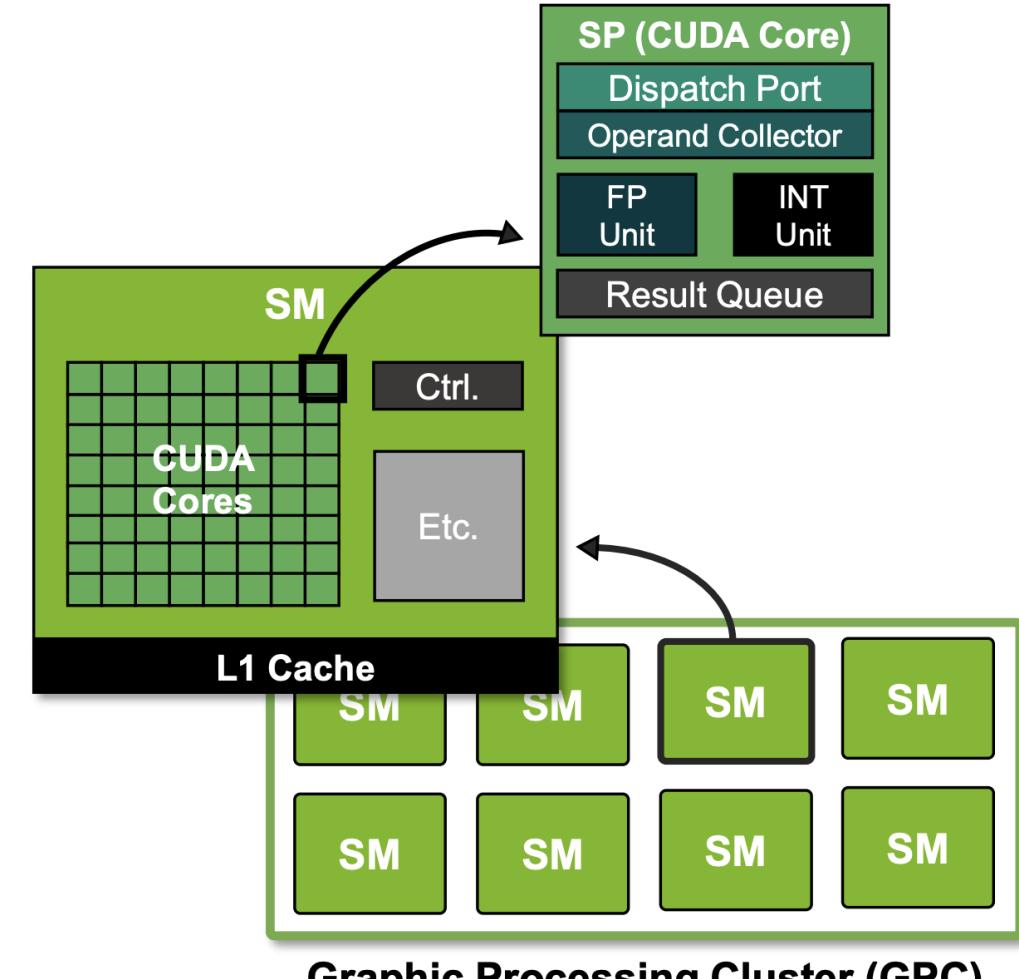
- 여러 개의 CUDA Core와 L1 캐시를 포함
- CUDA Core 여러 개는 하나의 컨트롤 로직 공유
- 32개의 CUDA Core를 포함

■ Graphic Processing Cluster (GPC)

- SM의 그룹

■ GPC, SM, SP의 개수

- 아키텍처의 종류에 따라 다름 (Pascal, Volta, Turing)



CUDA: Execution Model

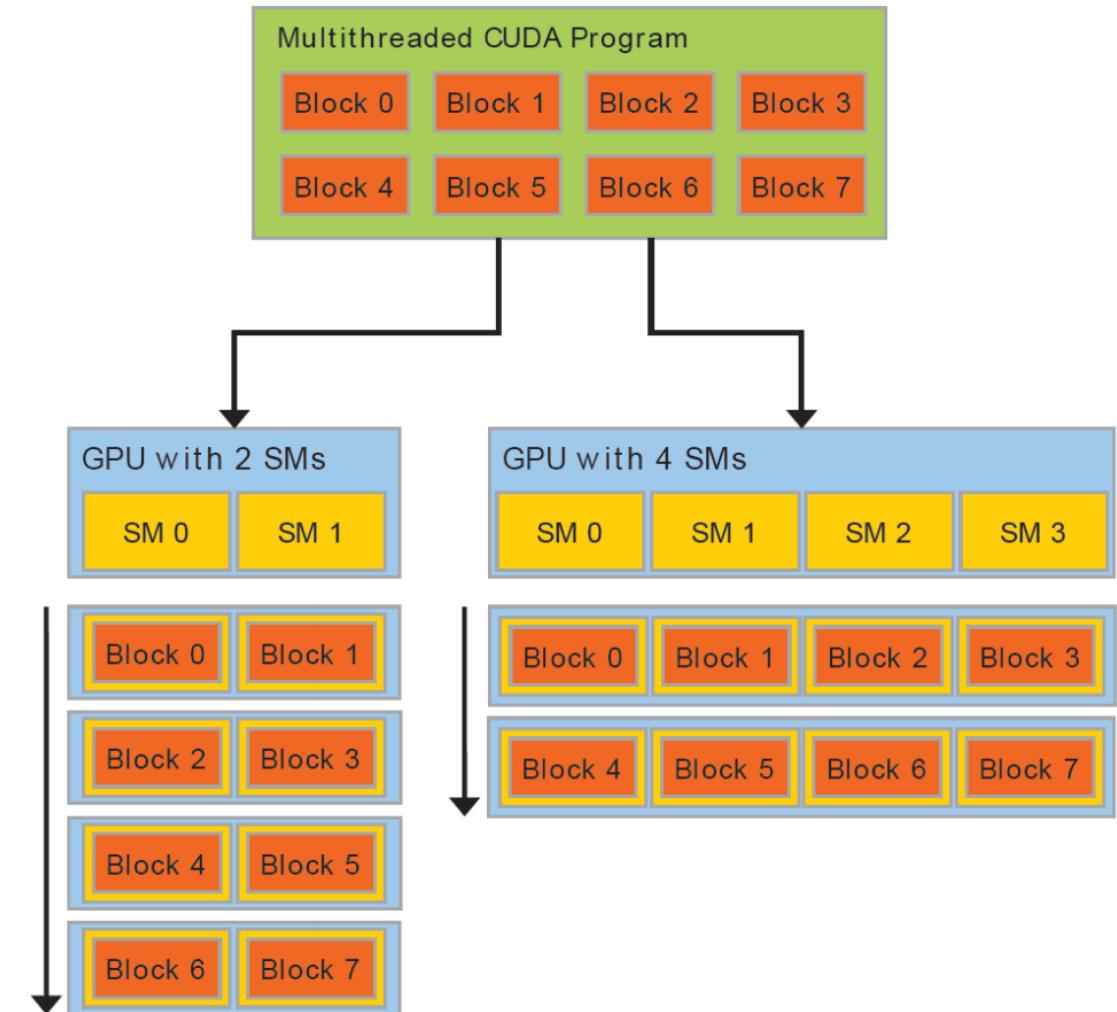
■ Thread Hierarchy & GPU

■ Grid/Kernel

- Kernel이 실행될 때 Grid가 생성
- GPU에서 동작하는 프로그램의 기본 단위 (Kernel)

■ Block

- Block들은 여러 개의 SM에 분산되어 계산됨
- Block은 현재 SM에서 계산되는 대상



CUDA: Execution Model

■ Thread Hierarchy & GPU

■ Warp

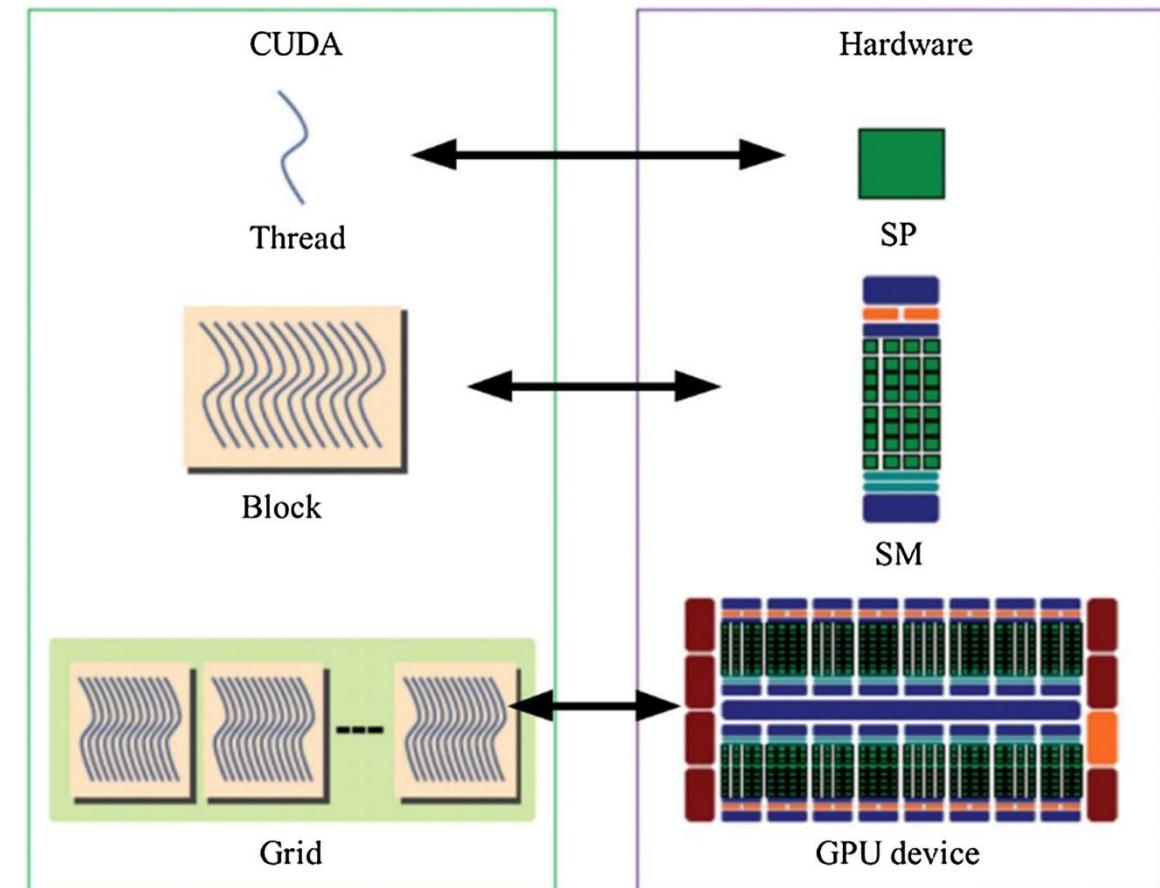
- 하나의 Block은 Warp 단위로 나눠져 계산됨
- Warp는 하드웨어에서 계산 가능한 CUDA Thread의 숫자

■ Warp Divergence

- Thread 내부에 조건문 존재, 해당 코드는 순차적으로 실행

■ Thread

- Streaming Processor (SP)에서 동작



CUDA: Memory Model

■ 계층적인 메모리 구조

■ Per-Thread

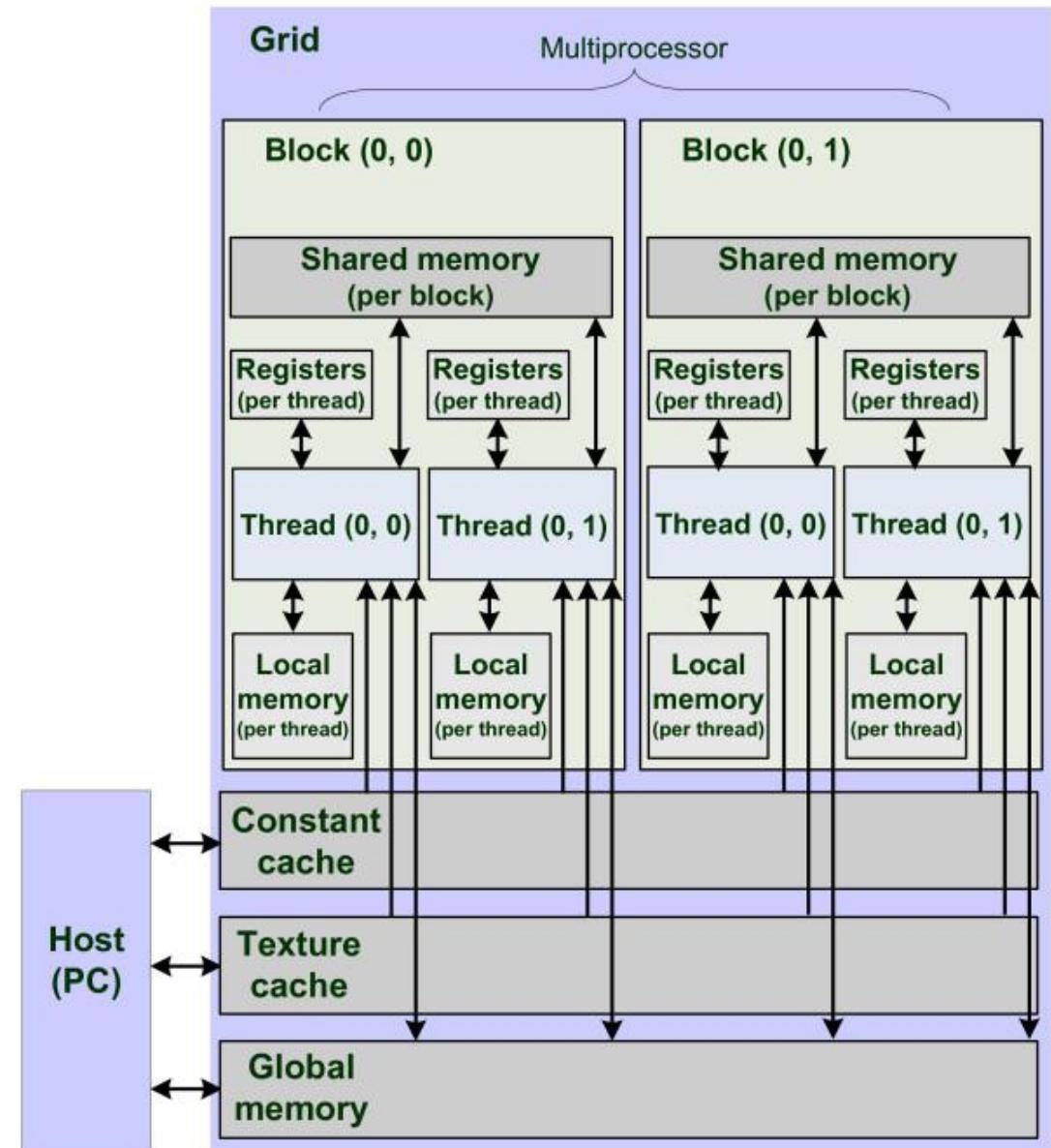
- Register
- Local memory

■ Per-Block

- Shared Memory

■ Per-Grid

- Global Memory
- Constant Memory (Read Only, Cachable Memory)
- Texture Memory (Read Only, Cachable Memory)



CUDA: Memory Model

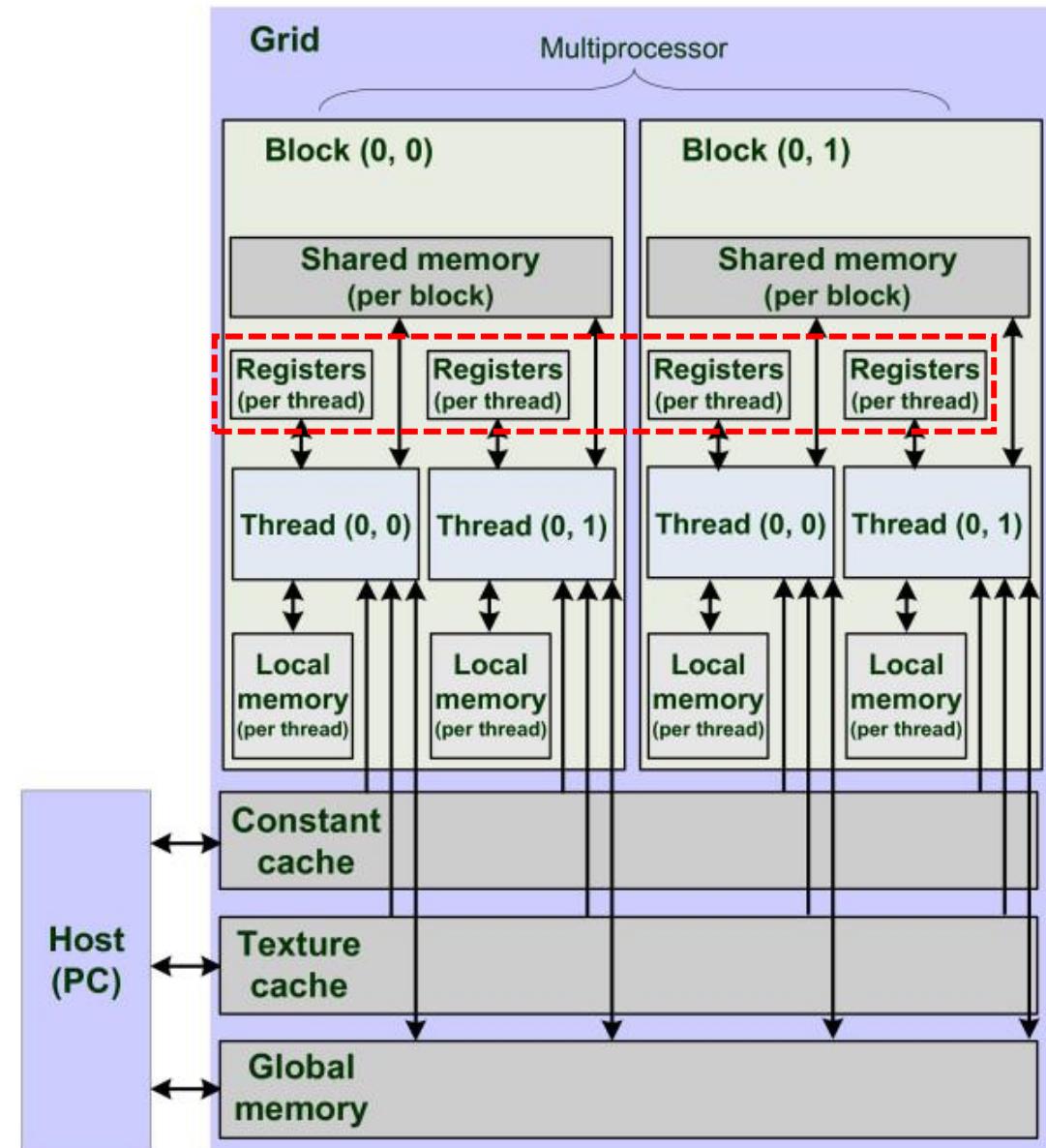
■ Register

■ 가장 빠르지만, 가장 작은 크기의 메모리

- GPU 기준으로 한 Cycle내에 데이터를 읽고 쓰는 것이 가능함
- GPU Kernel 내부에서 선언되는 변수에 이에 해당

■ GPU 칩 내부에 존재하는 메모리 (On-Chip)

- SM 내부에 포함되어 있음
- SM당 8k~64k 32bit 레지스터의 사용 가능



CUDA: Memory Model

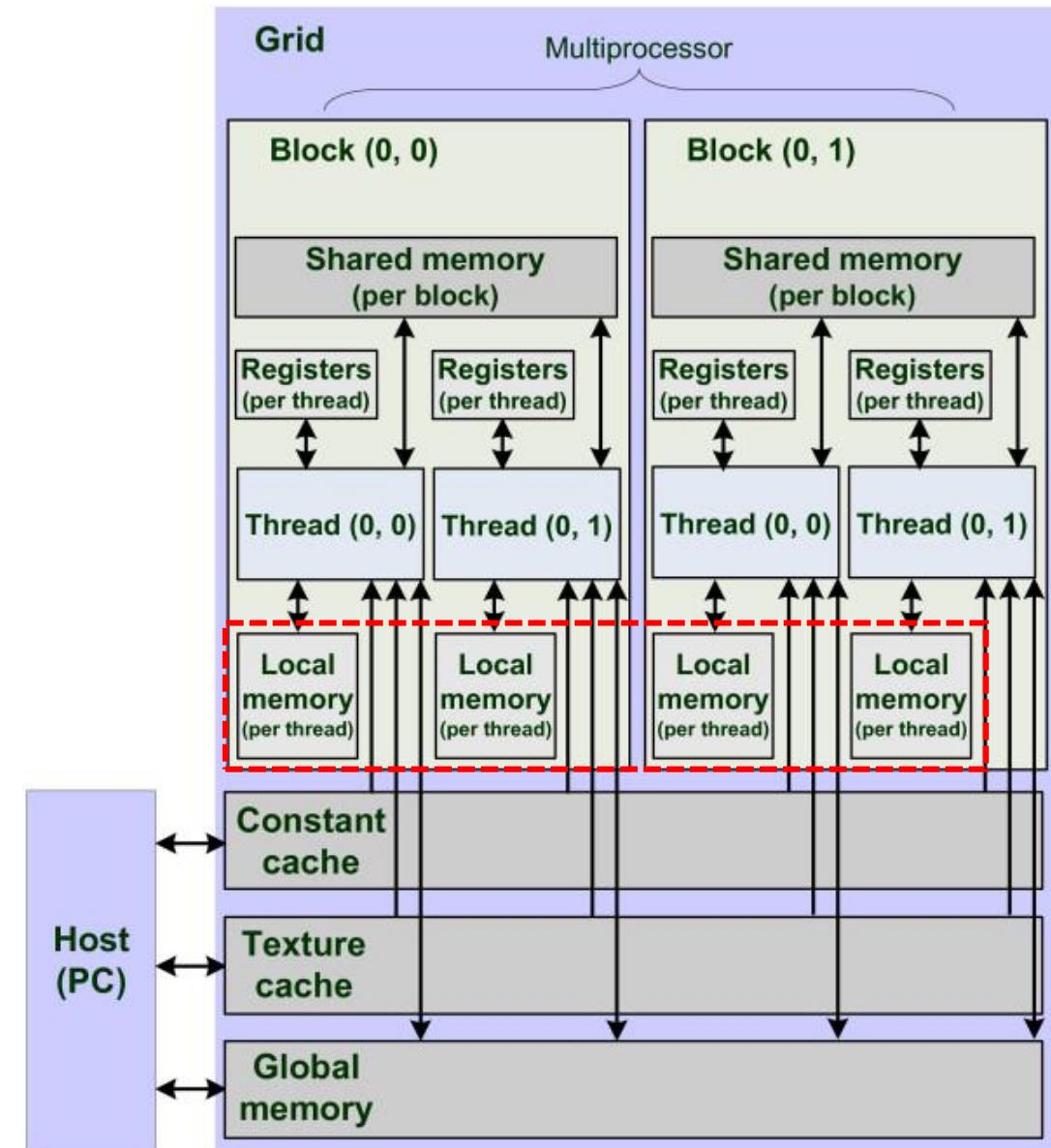
■ Local Memory

■ GPU 칩 외부에 존재하는 메모리 (Off-Chip)

- GPU PCB의 DRAM에 존재하는 메모리
- 칩 외부에 존재하기 때문에 느린 특성 (400~600 Cycles)

■ 큰 용량의 사용이 가능한 메모리

- 레지스터는 용량이 작기 때문에 큰 용량의 데이터 사용 불가능
- 큰 크기의 구조체나 배열을 사용하는데 효과적



CUDA: Memory Model

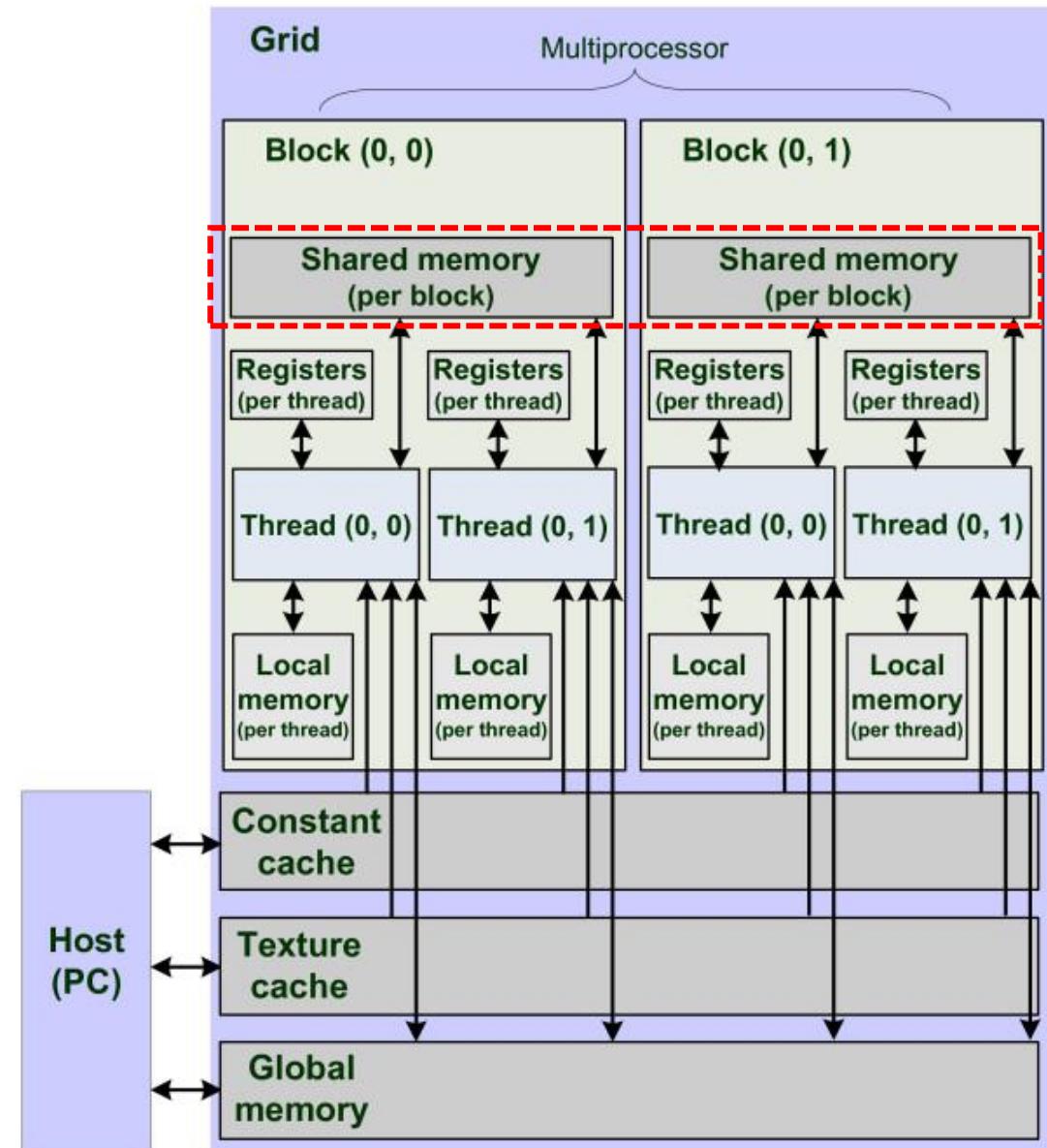
■ Shared Memory

■ 빠르지만 작은 크기의 메모리

- 칩 내부에 존재하는 메모리
- 1~4 GPU Cycle 기준으로 읽기/쓰기 가능
- 16~96 KB per thread block/SM

■ 동일한 Thread block 내에서만 사용 가능

- 동일한 Thread block 내부의 thread는 같은 데이터를 사용 가능



CUDA: Memory Model

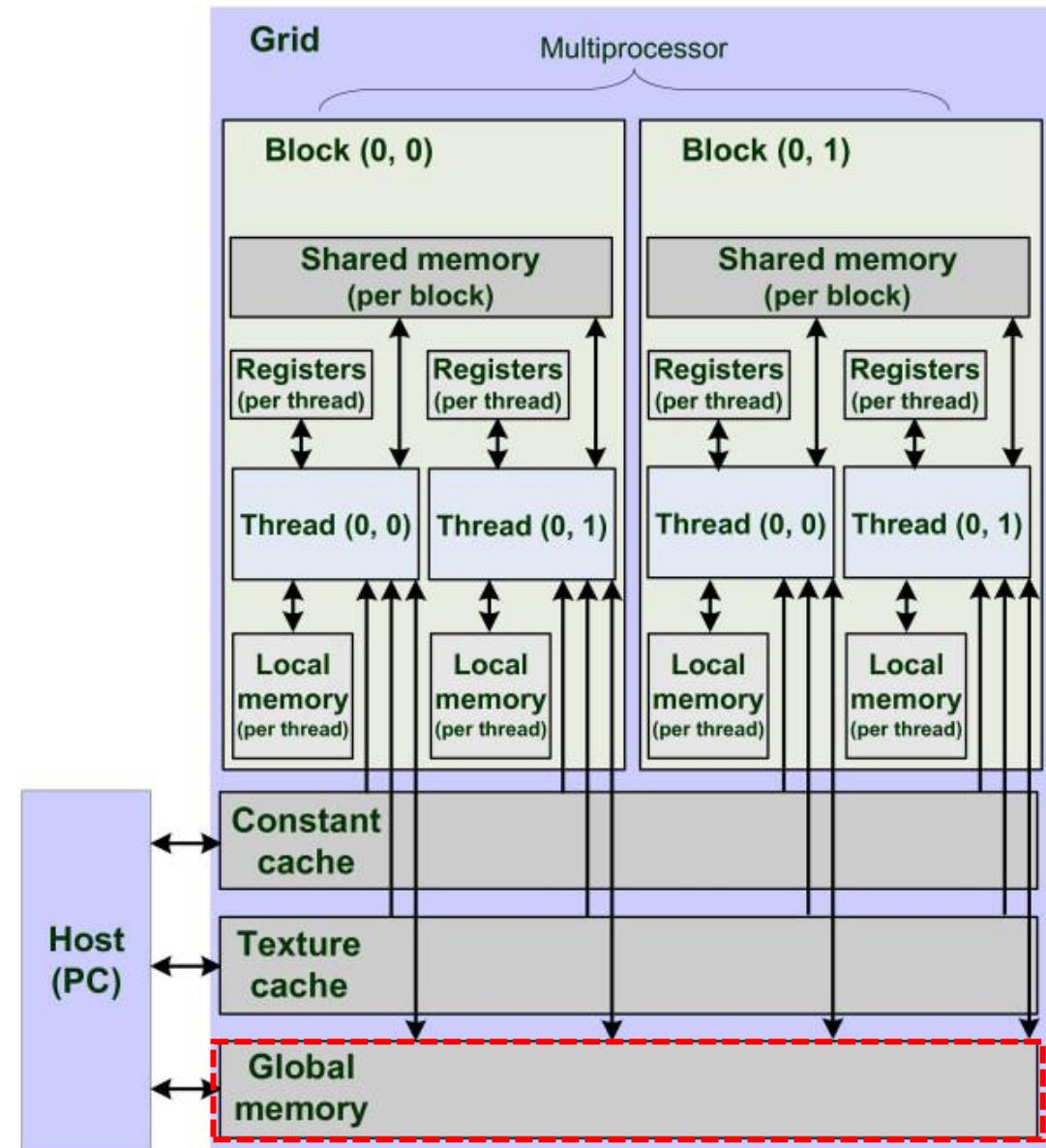
■ Global Memory

■ 가장 느리지만 큰 크기의 메모리

- 400~800 GPU Cycle 기준으로 읽기/쓰기 가능
- 2~16GB 메모리의 사용 가능

■ Grid 내의 모든 Thread에서 사용 가능

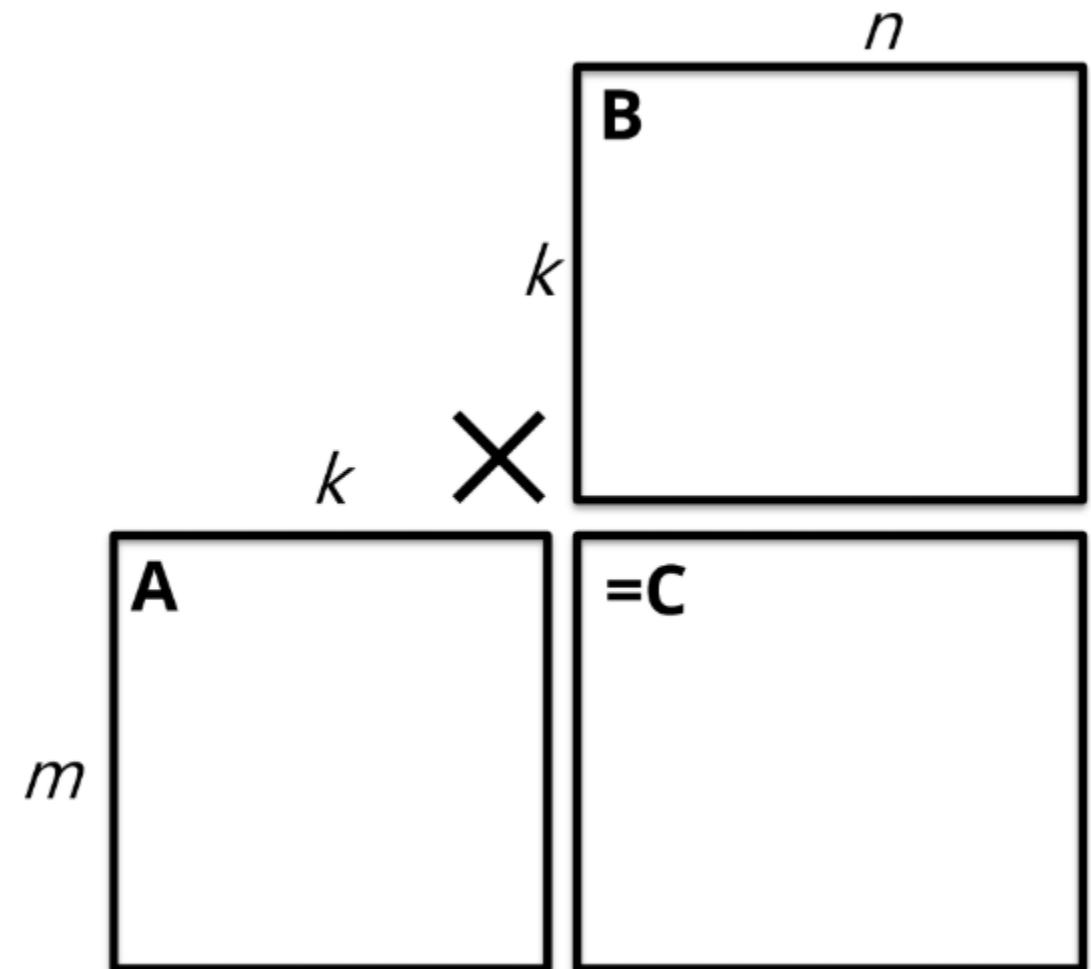
- 동일한 Thread block 내부의 thread는 같은 데이터를 사용 가능



CUDA: Matrix Multiplication

Matrix Multiplication

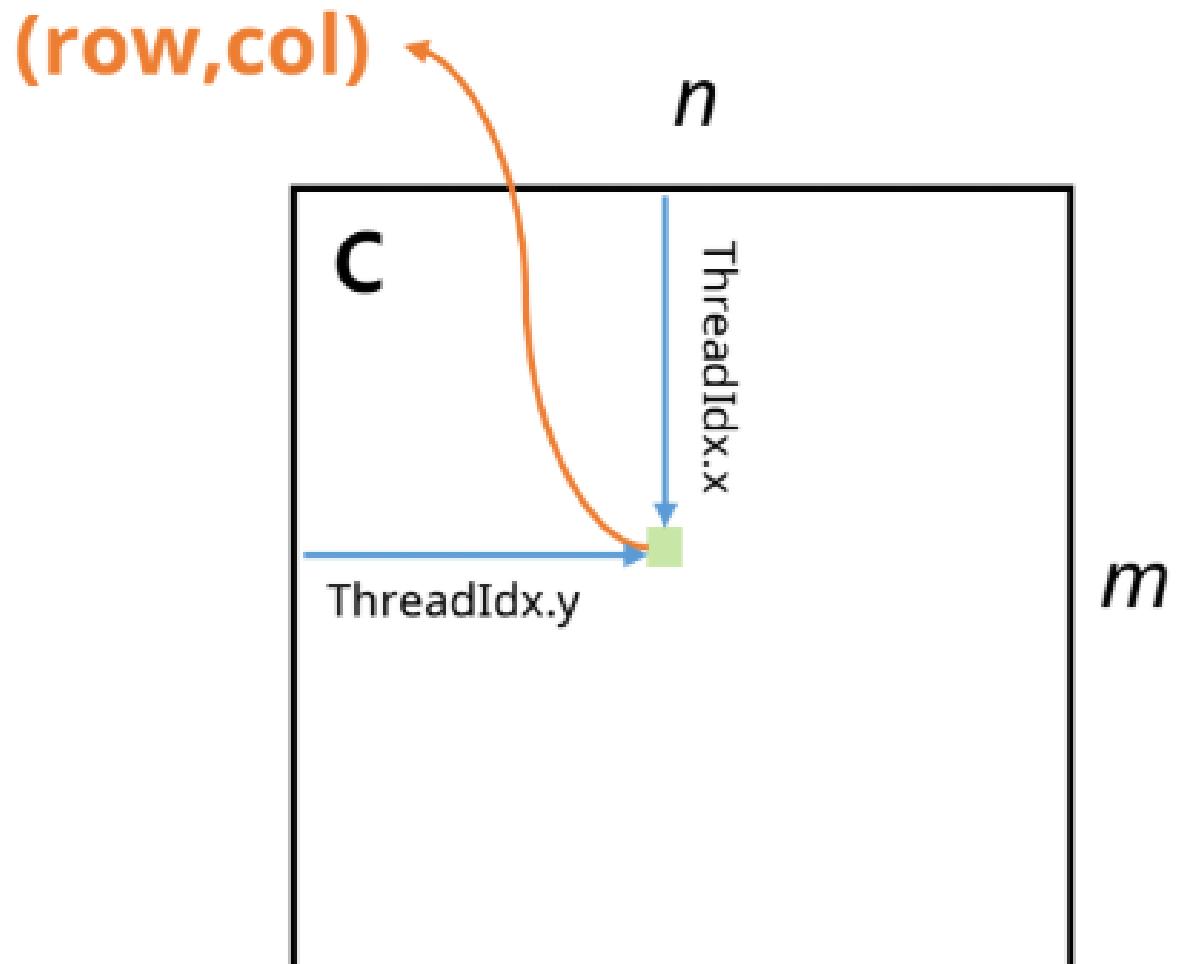
- $A = m \text{ by } k$
- $B = k \text{ by } n$
- $C = m \text{ by } n$



CUDA: Matrix Multiplication

Thread Layout

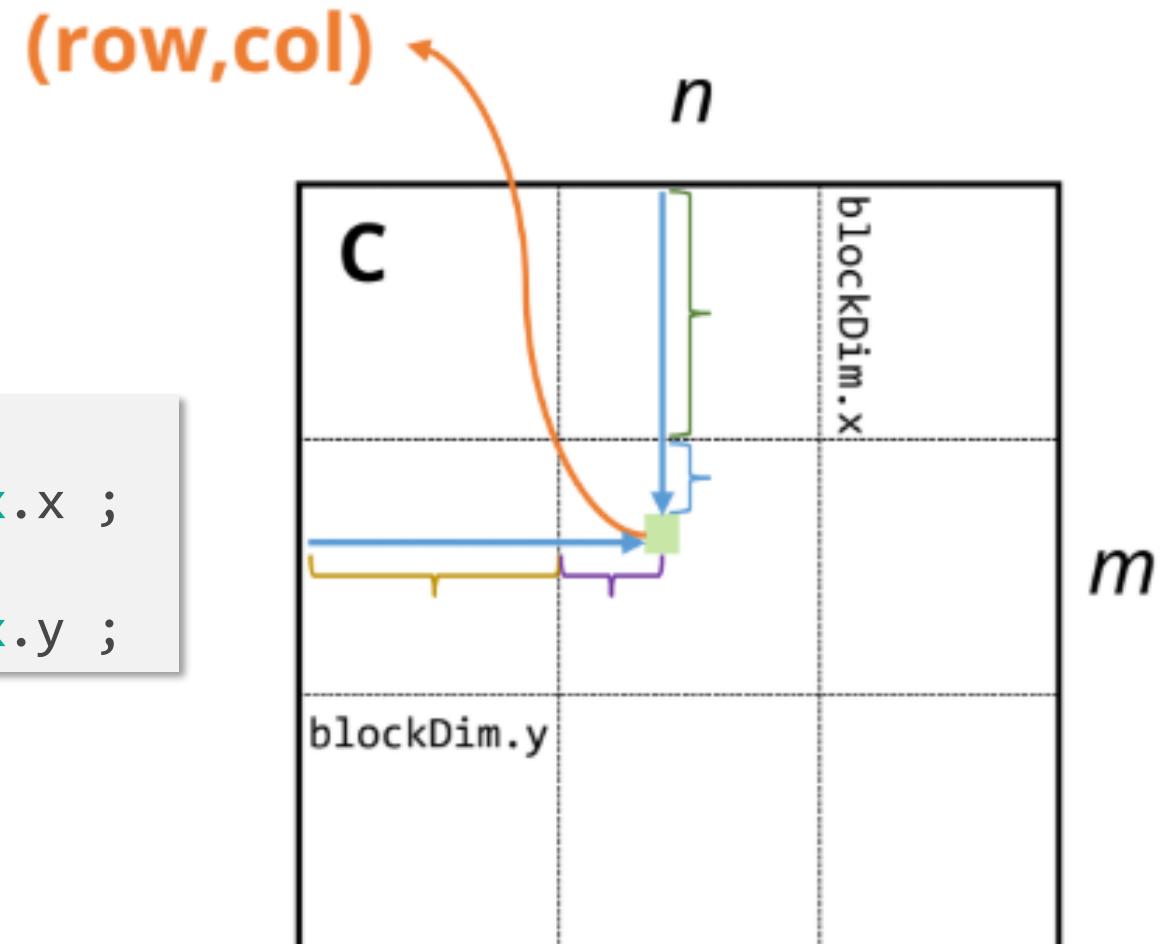
- Design a thread layout based on the matrix C
- 2D threadID (row, col)
 - Row = x-dimension
 - Col = y-dimension



CUDA: Matrix Multiplication

- 2D grid with 2D block
 - Block index in a Grid + Thread index in a Block

```
int row  
= blockDim.x * blockIdx.x + threadIdx.x ;  
  
int col  
= blockDim.y * blockIdx.y + threadIdx.y ;
```



CUDA: Matrix Multiplication (Example #6)

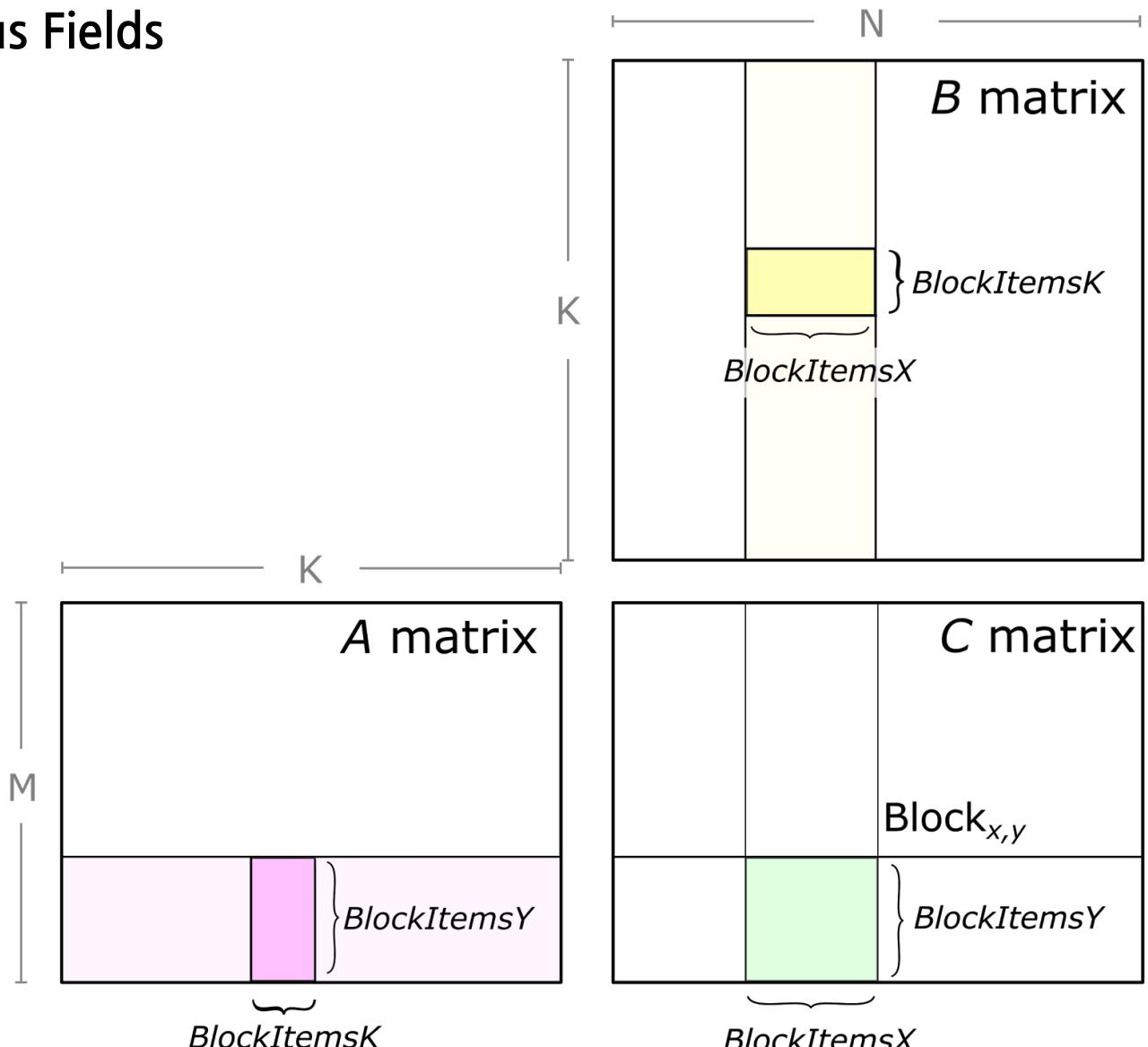
General Matrix Multiplication used in Various Fields

Assumption

- A small enough to handle by a thread block
- 예) row=32, k=128, col=32

Thread Layout

- 2D Thread block
- TILE_WIDTH is threads in the Thread block
- ***col = blockIdx.x * TILE_WIDTH + threadIdx.x;***
- ***row = blockIdx.y * TILE_WIDTH + threadIdx.y;***



CUDA: Matrix Multiplication (Example #6)

Host Code

```
for (int r = 0 ; r < ROW_SIZE ; r++)
    for (int c = 0; c < COL_SIZE; c++)
        for (int k = 0 ; k < K_SIZE; k++)
            hostC[r][c] += A[r][k] * B[k][c];
```

Kernel Code

```
__global__ void MatrixMulKernel(float* Ad, float* Bd, float* Cd)
{
    // Calculate the row index of the Cd element and A
    int Row = blockIdx.y * TILE_WIDTH + threadIdx.y;
    // Calculate the column index of Cd and B
    int Col = blockIdx.x * TILE_WIDTH + threadIdx.x;

    // Each thread computes one element of the block sub-matrix
    for (int k = 0; k < K; ++k)
        Cd[Row * K + Col] += Ad[Row * K + k] * Bd[k * K + Col];
}
```

```
dim3 blockDim(TILE_WIDTH, TILE_WIDTH);
```

CUDA: Matrix Multiplication (Example #6)

```
void MatrixMultiplication(float* A, float* B, float* C, int Width)
{
    int size = Width * Width * sizeof(float);
    float *Ad, *Bd, *Cd;

    // Transfer A and B to device memory
    cudaMalloc((void**)&Ad, size);
    cudaMemcpy(Ad, A, size, cudaMemcpyHostToDevice);
    cudaMalloc((void**)&Bd, size);
    cudaMemcpy(Bd, B, size, cudaMemcpyHostToDevice);

    // Allocate C on the device
    cudaMalloc((void**)&Cd, size);

    // Setup the execution configuration
    dim3 dimGrid(Width / TILE_WIDTH, Width / TILE_WIDTH); // Number of 2-dimension Blocks per Grid
    dim3 dimBlock(TILE_WIDTH, TILE_WIDTH); // Number of 2-dimension Threads per Block

    // Launch the device computation threads!
    MatrixMulKernel <<< dimGrid, dimBlock >>>(Ad, Bd, Cd);

    // Transfer C from device to host
    cudaMemcpy(C, Cd, size, cudaMemcpyDeviceToHost);
    // Free device matrices
    cudaFree(Ad);
    cudaFree(Bd);
    cudaFree(Cd);
}
```

CUDA: Matrix Multiplication (Example #6)

```
#include <cuda.h>
#include <stdio.h>
#include <cuda_runtime.h>
#include <device_launch_parameters.h>

#define TILE_WIDTH 8
#define M 8
#define N 8
#define K 8

int main()
{
    int width;
    float *A, *B, *C;
    cudaEvent_t start, stop;
    float elapsedTime;
    width= N;

    A = (float*)malloc(sizeof(float)*width*width);
    B = (float*)malloc(sizeof(float)*width*width);
    C = (float*)malloc(sizeof(float)*width*width);

    for (int i = 0; i < width*width; ++i){
        A[i] = 1;
        B[i] = 1;
        C[i] = 0;
    }
    cudaEventCreate(&start);
    cudaEventCreate(&stop);
    cudaEventRecord(start, 0);
```

```
    MatrixMultiplication(A, B, C, width);

    cudaEventRecord(stop, 0);
    cudaEventSynchronize(stop);
    cudaEventElapsedTime(&elapsedTime, start, stop);

    for (int i=0; i<M; i++) {
        for (int j=0; j<N; j++) {
            printf("%lf ", C[i*N+j]);
        }
        printf("\n");
    }
    printf("%d x %d Matrix Multiplication Time : %fs \n\n", width,
           width, elapsedTime/1000);

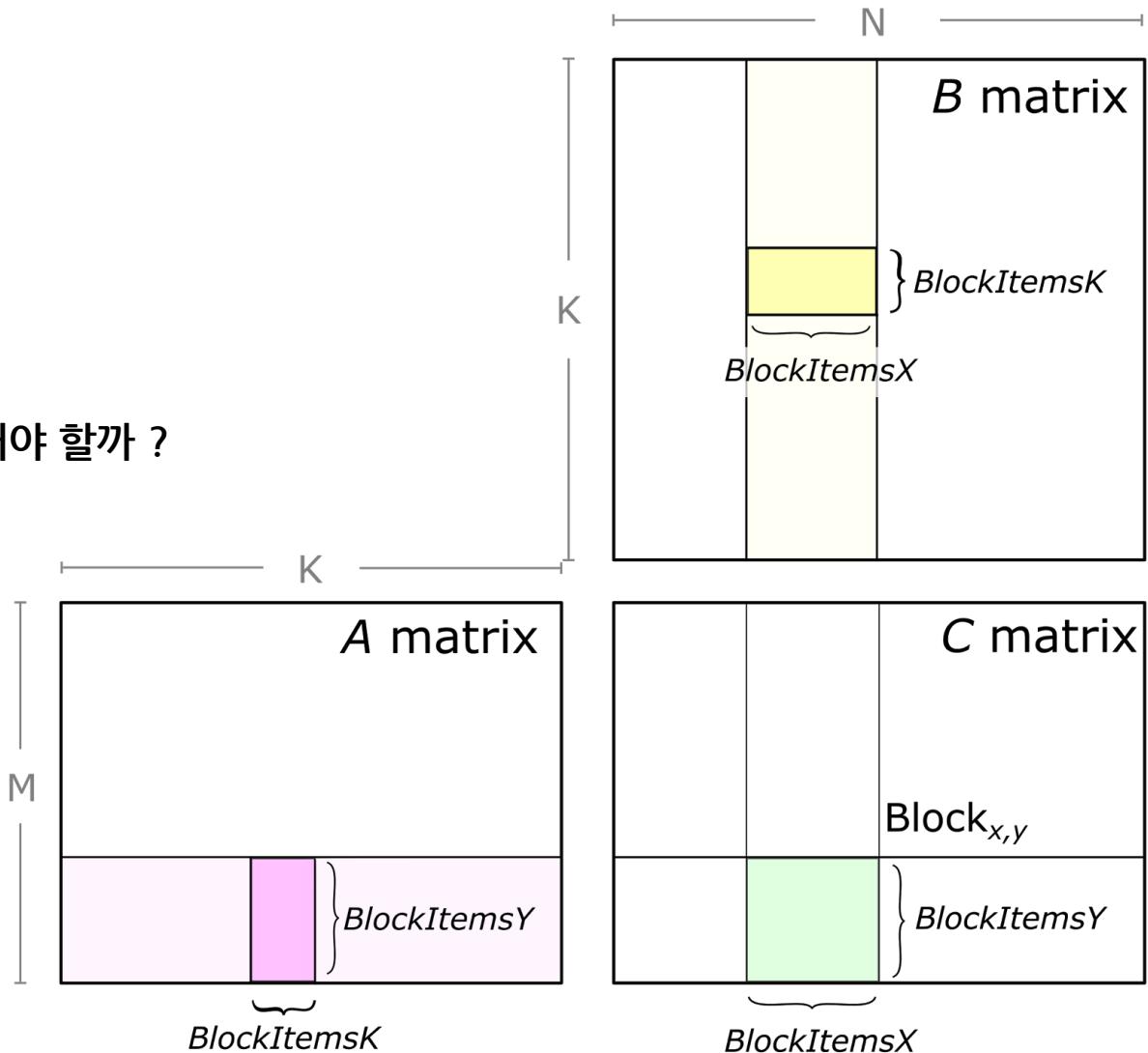
    cudaEventDestroy(start);
    cudaEventDestroy(stop);

    free(A);
    free(B);
    free(C);
    return 0;
}
```

CUDA: Matrix Multiplication

■ 연산을 위해서 메모리에 접근하는 횟수는 ?

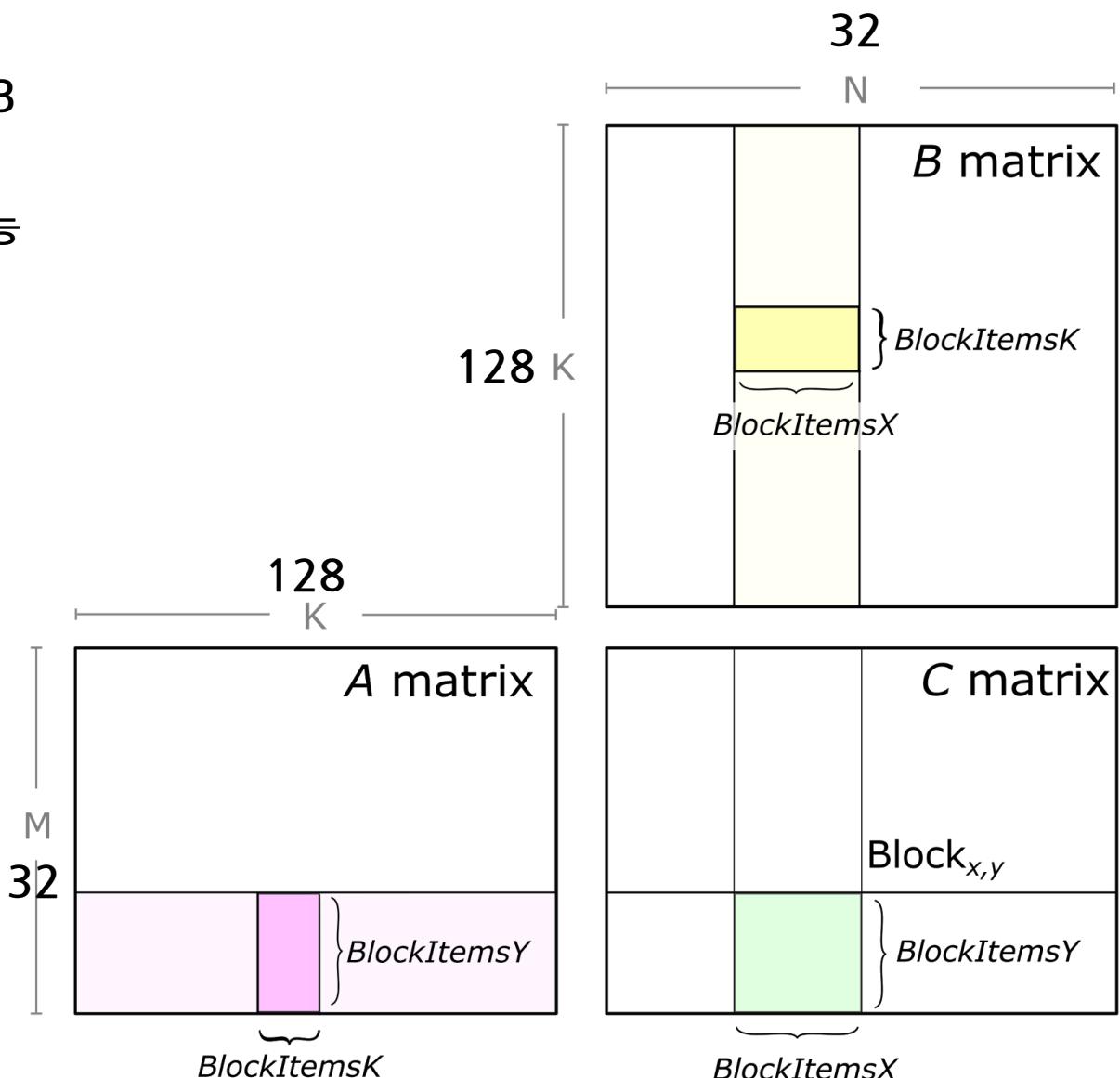
- Each row of A
 - The column length of B (K)
- Each column of B
 - The row length of A
- How many accesses to global memory ?
 - $O(|row| * |col| * k)$
 - Global memory의 접근 횟수를 줄이기 위해서는 어떻게 해야 할까 ?



CUDA: Matrix Multiplication

A=32 by 128, B = 128 by 32

- 2 * 4069 (32 * 128) * 4bytes = 2 * 16KB = 32KB
 - Shared memory size = 64KB
 - 행렬 A와 B를 모두 Shared memory에 저장하는 것이 가능
 - We can load A and B into shared memory
- Global Memory Access
 - $O(|row|+|col|)*k$



CUDA: Matrix Multiplication

Kernel using Shared Memory

_syncthreads()

- 동일한 Thread block 내에 존재하는 Thread간의 동기화
- Shared memory를 사용하기 전에, 모든 데이터가 Shared memory에 저장되는 것을 보장하기 위해 사용

```
__global__ void MatrixMulKernel(float* Ad, float* Bd, float* Cd, int Width)
{
    // Initialize shared memory like cache
    __shared__ float As[TILE_WIDTH][TILE_WIDTH];
    __shared__ float Bs[TILE_WIDTH][TILE_WIDTH];

    int Row = blockIdx.y * TILE_WIDTH + threadIdx.y;
    int Col = blockIdx.x * TILE_WIDTH + threadIdx.x;

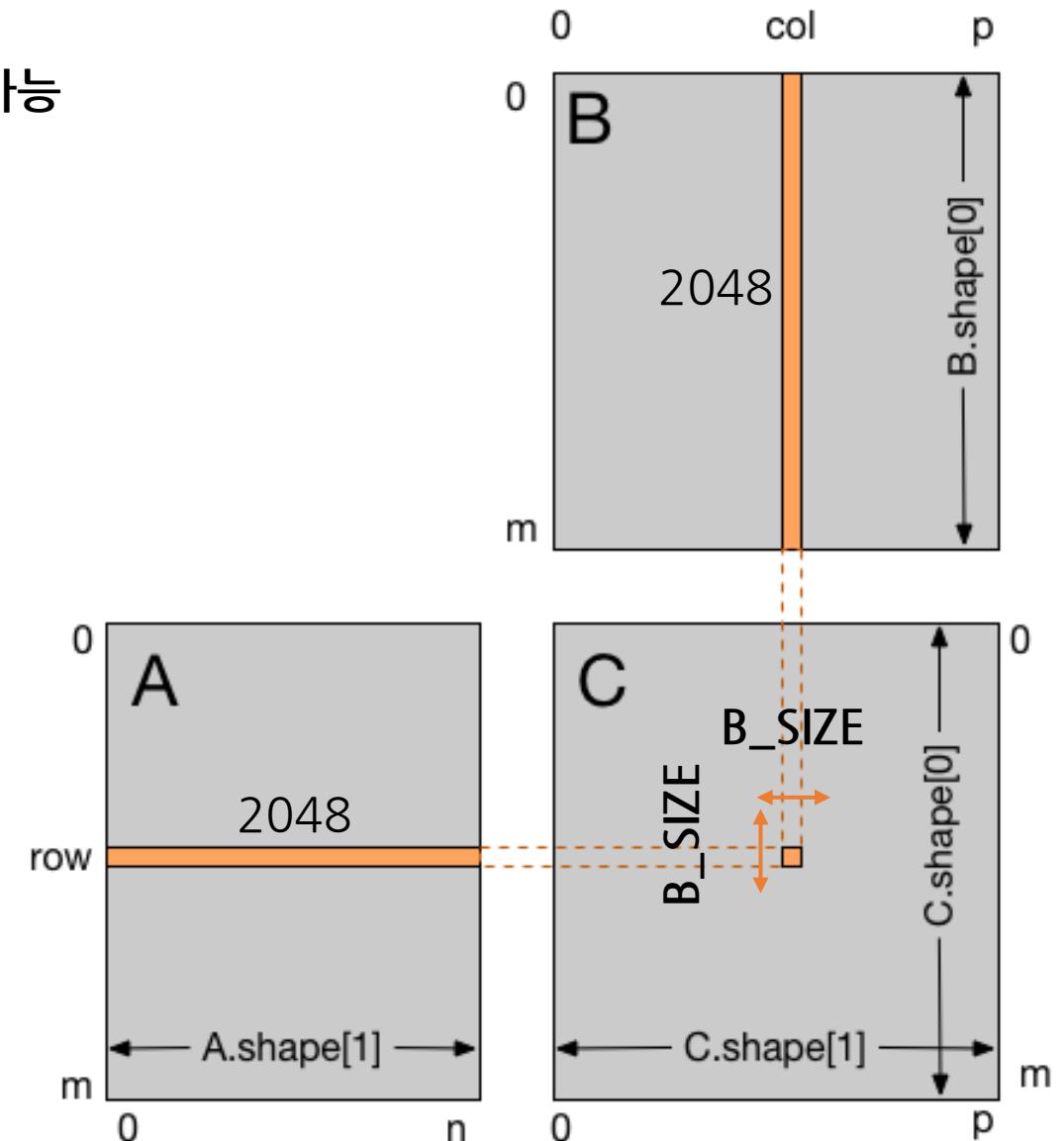
    for (int k = 0; k < (TILE_WIDTH + Width - 1)/TILE_WIDTH; k++) {
        if (k*TILE_WIDTH + threadIdx.x < Width && Row < Width)
            As[threadIdx.y][threadIdx.x] = Ad[Row*Width + k*TILE_WIDTH + threadIdx.x];
        ...
    }
    __syncthreads(); // wait until all thread Load the matrix
}
```

CUDA: Matrix Multiplication

■ Load Sub-Matrix into Shared Memory

- 큰 크기의 행렬이라면, Shared memory에 저장하는 것은 불가능
 - 보통 Shared memory는 최대 64KB의 데이터 저장 가능
 - $B_SIZE = 16$, $A = 1024 * 2048$, $B = 2048 * 4096$ 라면
 $(16*2048+16*2048)*4Byte=256KB$ 필요 !
 - 행렬의 일부를 Shared memory에 저장하여 연산에 사용
- 큰 행렬을 작은 크기의 행렬로 나눠 순차적으로 로드
 - The same size with the block

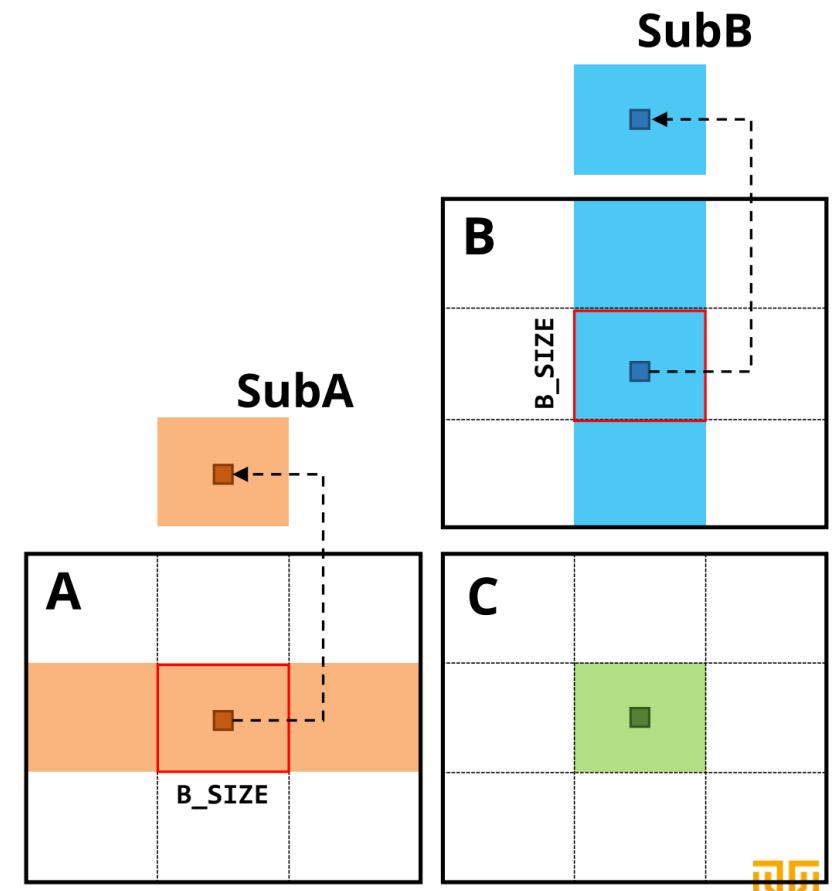
```
__global__ void MatMul ... {  
    ...  
    __shared__ float subA[B_SIZE][B_SIZE];  
    __shared__ float subB[B_SIZE][B_SIZE];  
    ...  
}
```



CUDA: Matrix Multiplication

- Load Sub-Matrix into Shared Memory
 - Each thread loads an element based on its thread ID in the block

```
__global__ void MatMul ... {  
    ...  
    __shared__ float subA[B_SIZE][B_SIZE];  
    __shared__ float subB[B_SIZE][B_SIZE];  
    ...  
}
```

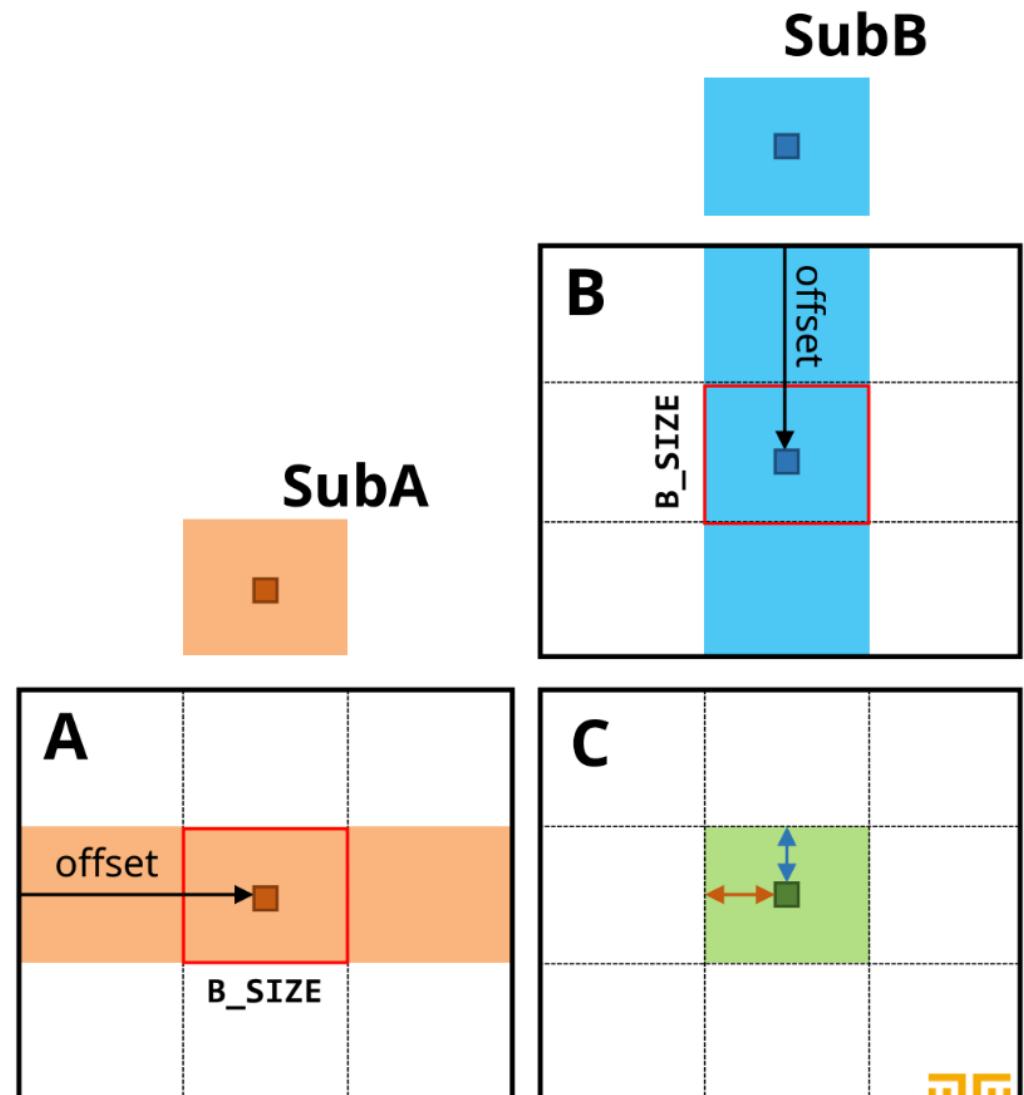


CUDA: Matrix Multiplication

Sub-block of Matrix A

- SubA[localRow][localCol] = A[row * k + offset]
- SubB[localRow][localCol] = B[col + (offset) * n]

```
__global__ void MatMul ... {  
    ...  
    __shared__ float subA[B_SIZE][B_SIZE];  
    __shared__ float subB[B_SIZE][B_SIZE];  
    ...  
}
```



감사합니다
