GPU(Graphic Processing Unit)

프로세서 속도 향상을 위한 병렬처리 동향

- 멀티코어(multi-core) 프로세서
  - 2/4/8-core 프로세서
  - 여러 개의 CPU 코어들을 하나의 칩에 집적
  - Multi-threading, multiple-issue 및 non-sequential execution 기법들을 이용하여 순차적 프로그램의 처리속도 향상 시도

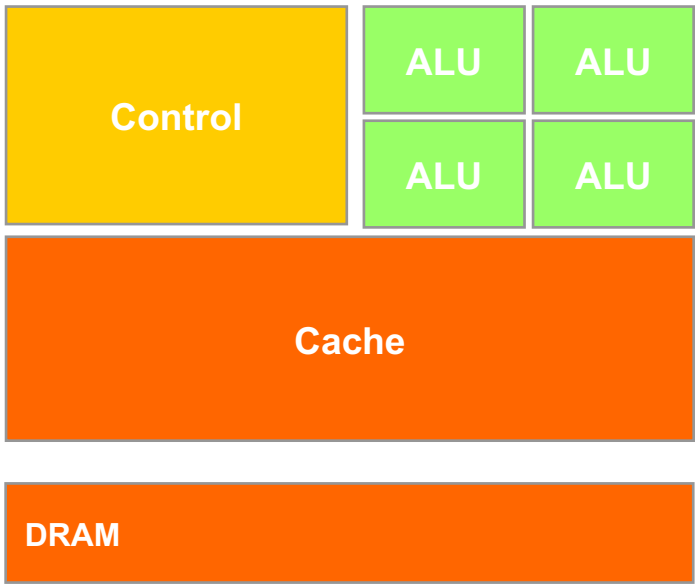
    [예] Intel Core i7 : 4~16개의 CPU core들로 구성

- 매니코어(many-core) 프로세서
  - 매우 작은 코어들을 대량으로 칩에 집적: 칩당 수백~수천 개 코어

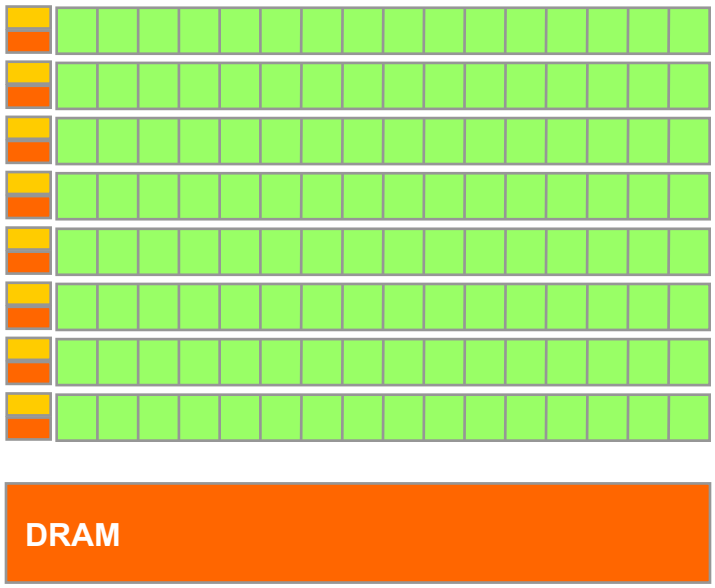
    [예] NVIDIA TITAN X GPU: 3584 cores, 1.4 GHz clock
  - 각 코어: single-issue, multi-threading 지원
  - 프로그램의 계산량이 많은 부분(부동소수점 연산들)을 병렬처리
  - 성능: 프로세서 당 1 TFLOPS 이상의 속도(멀티-코어의 수십 배)
  - GPGPU(General-Purpose Graphic Processing Unit): GPU 구조를 기반으로 발전한 범용 GPU
  - SPMD(Single-Program Multiple-Data) 모델

내부 구조 비교

CPU

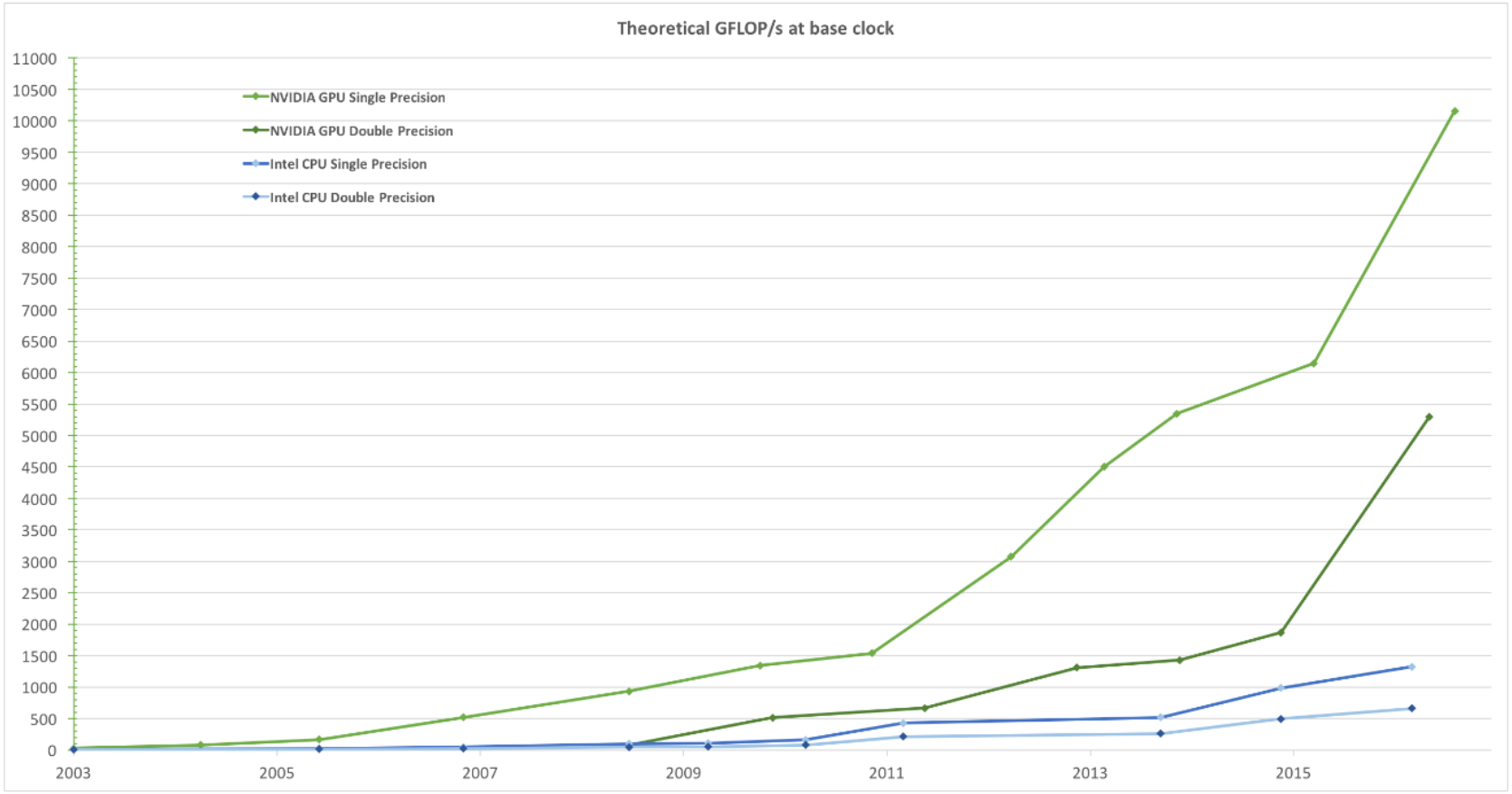


GPU



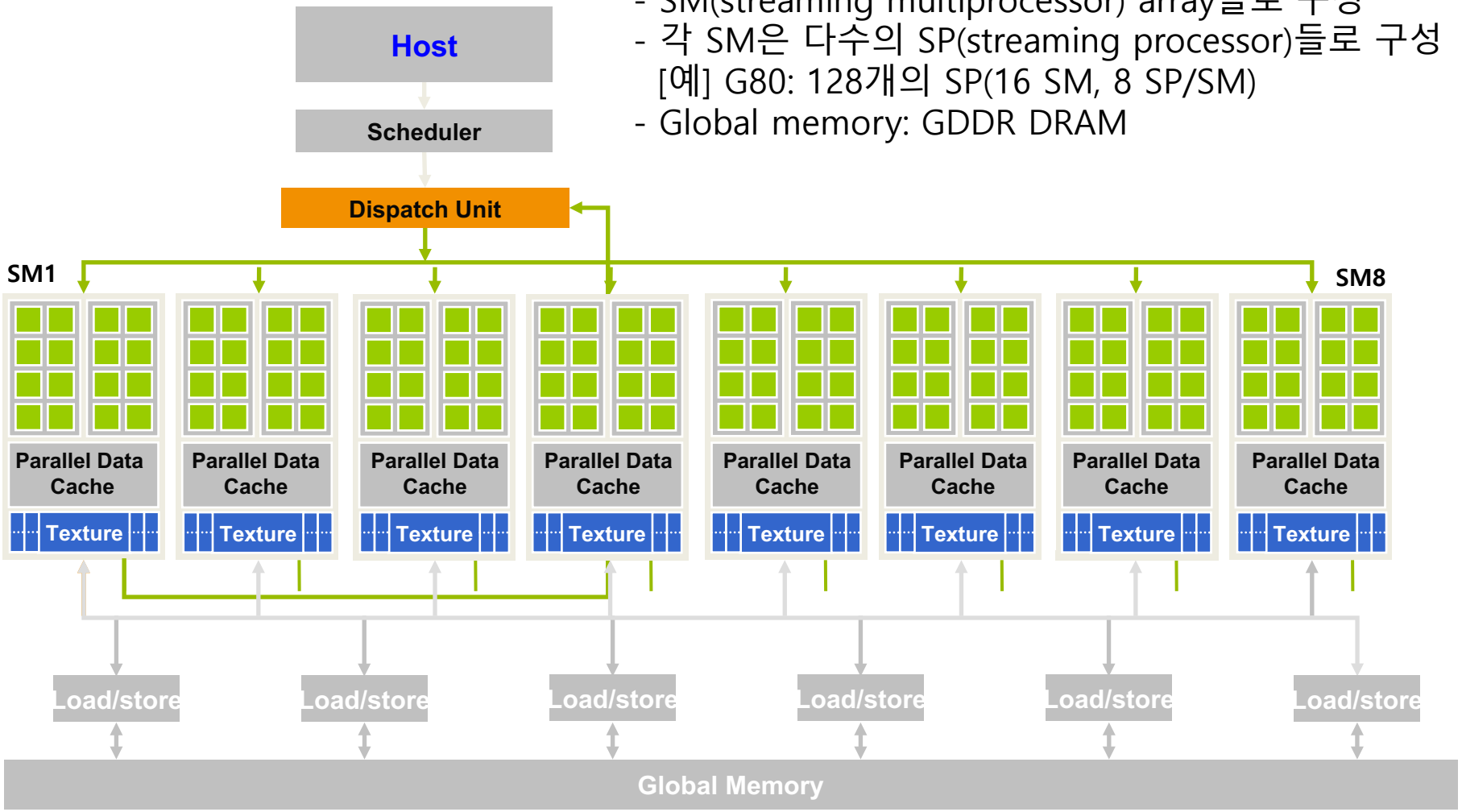
GPU와 일반 CPU의 성능 비교

Figure 1. Floating-Point Operations per Second for the CPU and GPU

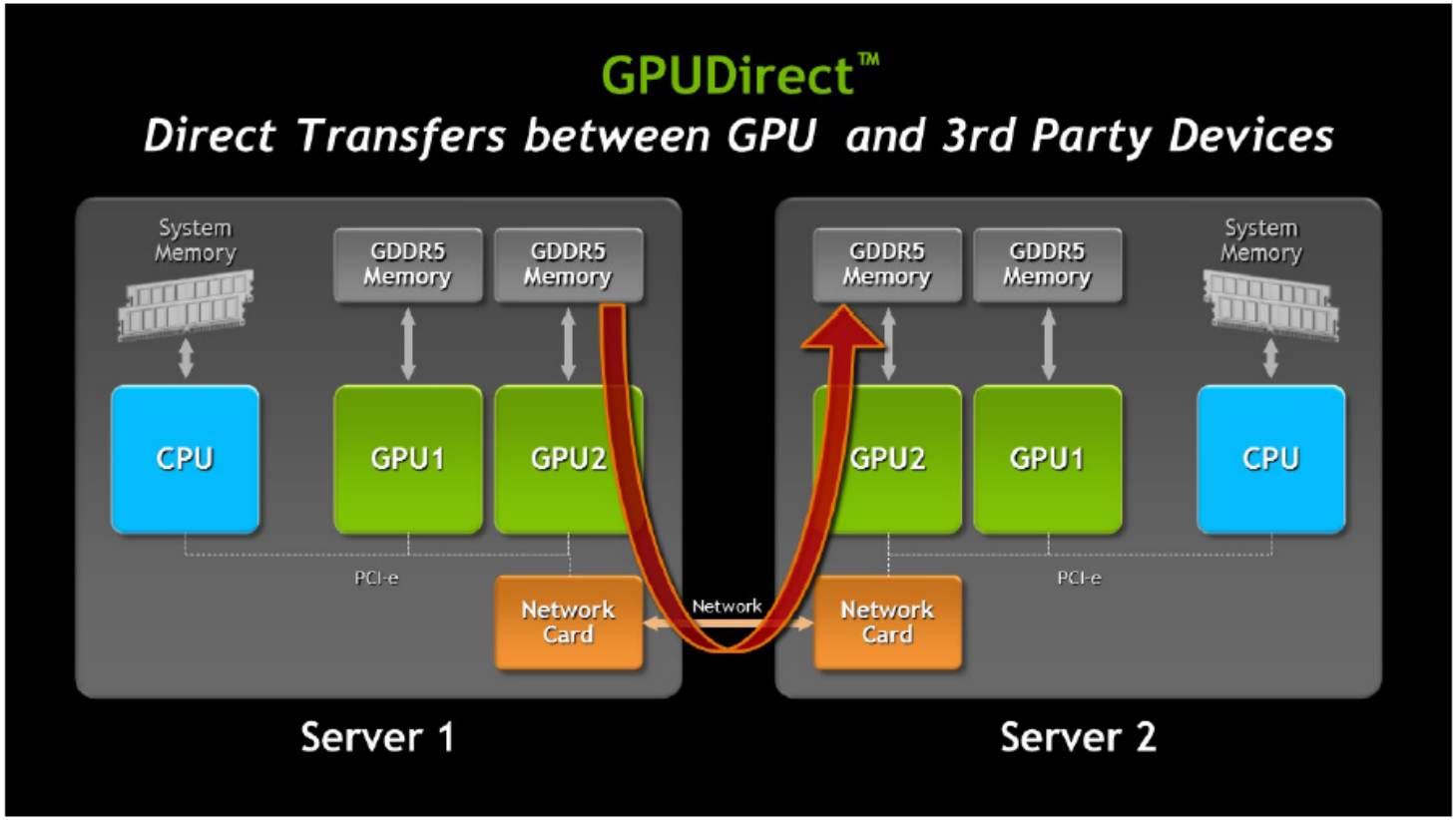


전형적인 GPU 구조

- SM(streaming multiprocessor) array들로 구성
- 각 SM은 다수의 SP(streaming processor)들로 구성
  [예] G80: 128개의 SP(16 SM, 8 SP/SM)
- Global memory: GDDR DRAM



CPU - GPU connection

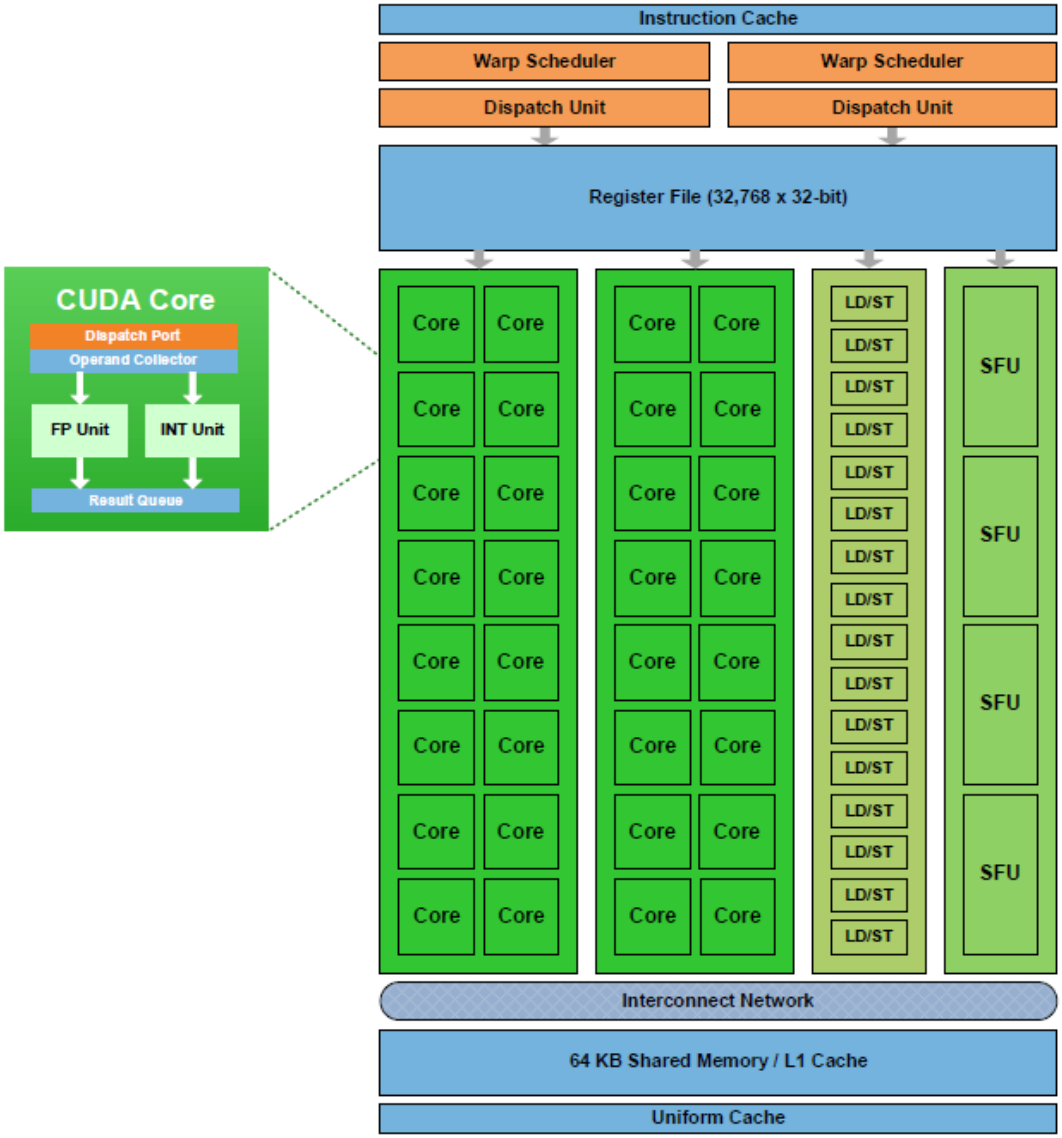


GPUDirect RDMA allows direct access to GPU memory from 3rd-party devices such as network adapters, which translates into direct transfers between GPUs *across* nodes as well.

SM(streaming multiprocessor) Array

Kepler GK110 Full chip block diagram

SM(streaming multiprocessor) 내부 구조



프로그래밍 모델

- GPGPU: 산술연산 엔진(계산전용 보조프로세서)
- 응용프로그램 처리 방식
  - CPU: 순차적인 부분 처리
  - GPU: 계산량이 많고 병렬처리 가능한 부분 처리
- CUDA(Compute Unified Device Architecture) 프로그래밍 모델
  - 2007년 NVIDIA사에서 개발한 병렬프로그래밍 모델
  - CPU 및 GPGPU를 이용한 응용프로그램의 병렬처리 지원
  - GPGPU측의 범용 병렬프로그래밍 인터페이스가 CUDA 프로그램의 요청을 처리
  - 프로그래머-친화적 병렬컴퓨팅 플랫폼(programmer-friendly parallel computing platform)으로서, GPU 활용 증대에 크게 기여

CUDA 소개

- CUDA 프로그램을 처리하는 컴퓨팅시스템: host와 한 개 이상의 device로 구성
  - Host: 일반적인 CPU를 가진 컴퓨터(PC, SMP 등)
  - Device: 병렬프로세서(예: GPU)로서, 대규모 병렬 데이터 처리를 담당

- CUDA 프로그램의 구조
  - CUDA 프로그램: host code와 device code로 이루어진 통합 소스 코드로서, 각각 host 혹은 device에서 실행
  - NVIDIA C compiler(nvcc)가 컴파일하는 과정에서 분리시킴
  - Host code: ANSI C code (순차적 프로그램 코드)
  - Device code: ANDI C code + kernel(데이터 병렬함수들 및 관련 자료구조를 명시하는 키워드를 확장한 형태로 작성된 코드)

CUDA Devices and Threads

- A compute device
  - Is a coprocessor to the CPU or host
  - Has its own DRAM (device memory)
  - Runs many threads in parallel
  - Is typically a GPU but can also be another type of parallel processing device
- Data-parallel portions of an application are expressed as device kernels which run on many threads
- Differences between GPU and CPU threads
  - GPU threads are extremely lightweight: very little creation overhead
  - GPU needs 1000s of threads for full efficiency

    (c.f.) Multi-core CPU needs only a few

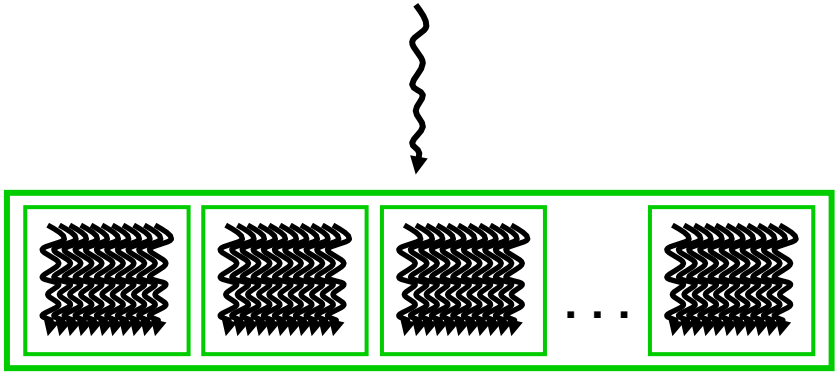
CUDA 소개 (계속)

- CUDA 프로그램의 실행 모델
  - 커널 함수가 launch되면, 커널 코드는 device로 보내져 실행됨
  - Device에서 커널이 다수의 스레드들을 생성하여 SM(streaming processor)들에게 할당
    - 그리드(grid): 하나의 응용을 위해 생성된 전체 스레드들을 통칭
  - 커널의 모든 스레드들의 수행이 완료되면, 그리드 종료
  - Host가 다시 host code를 실행하다가, 다른 커널이 호출되면 device 실행 반복
  - SPMD형 실행

CUDA 프로그램 실행 모델

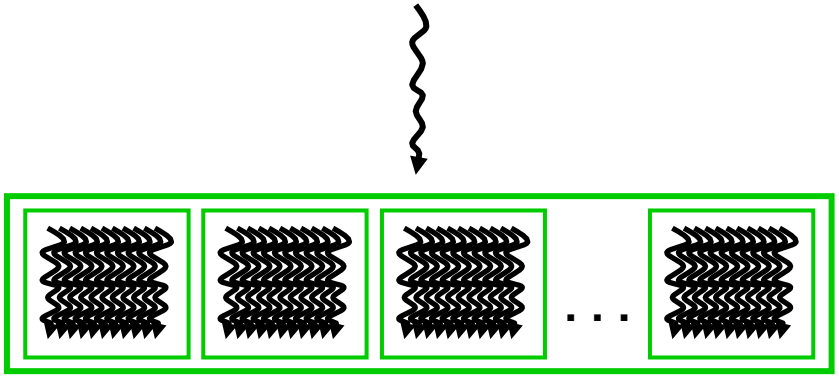
Serial Code (host)

Parallel Kernel (device)
KernelA<<< nBlk, nTid >>>(args);



Serial Code (host)

Parallel Kernel (device)
KernelB<<< nBlk, nTid >>>(args);



커널(kernel)

- Programmer가 kernel이라 불리는 C function을 정의
- Kernel이 호출되면, N개의 CUDA thread들이 N개의 SP(streaming processor)들에 의해 병렬로 실행
- Kernel 코드는 declaration specifier인 __global__ 을 사용하여 정의
- 그 kernel 호출을 실행할 CUDA thread의 수는 새로운 execution configuration syntax인 <<< . . . >>> 을 이용하여 지정
- 그 kernel을 실행할 각 thread에게는 thread ID가 주어지는데, kernel 함수 내에서 built-in variable인 'threadIdx'로 액세스

[예] 행렬 덧셈(Matrix addition): 크기가 N인 두 개의 벡터 A와 B를 더하고, 그 결과를 벡터 C에 저장

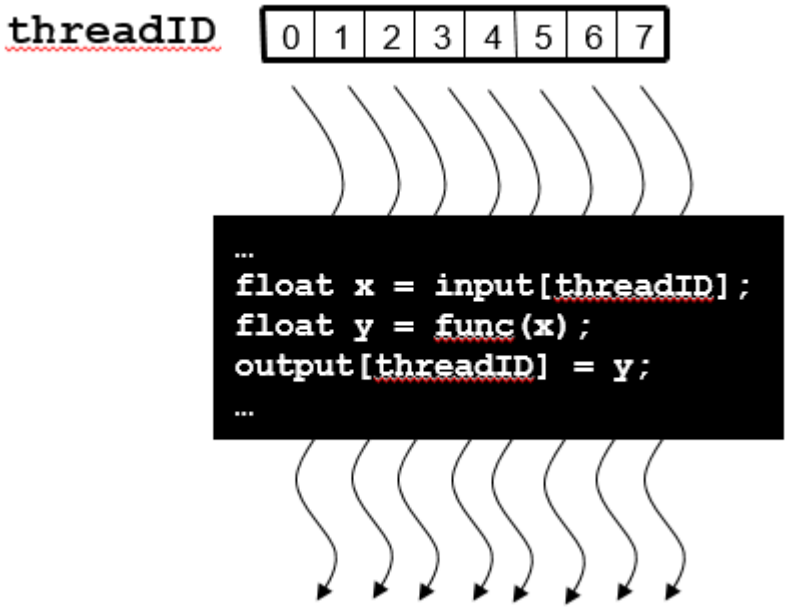
```
// Kernel definition
__global__ void VecAdd(float* A, float* B, float* C)
{
    int i = threadIdx;
    C[i] = A[i] + B[i];
}

int main()
{
    . . .
    // Kernel invocation with N threads
    VecAdd <<< 1, N >>> (A, B, C);
    . . .
}
```

➔ N개의 thread들이 각 data pair에 대하여 한 번씩의 VecAdd( )를 수행

C[1] = A[1] + B[1], C[2] = A[2] + B[2], . . .

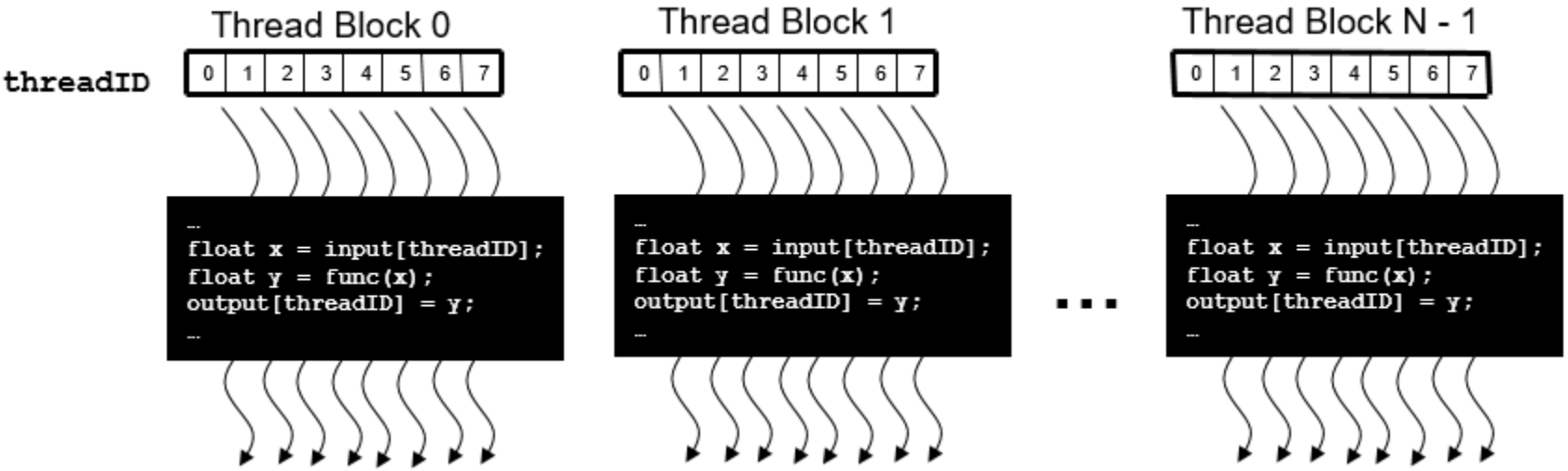
Arrays of Parallel Threads

- A CUDA kernel is executed by an array of threads
  - All threads run the same code (SPMD)
  - Each thread has an ID that it uses to compute memory addresses and make control decisions

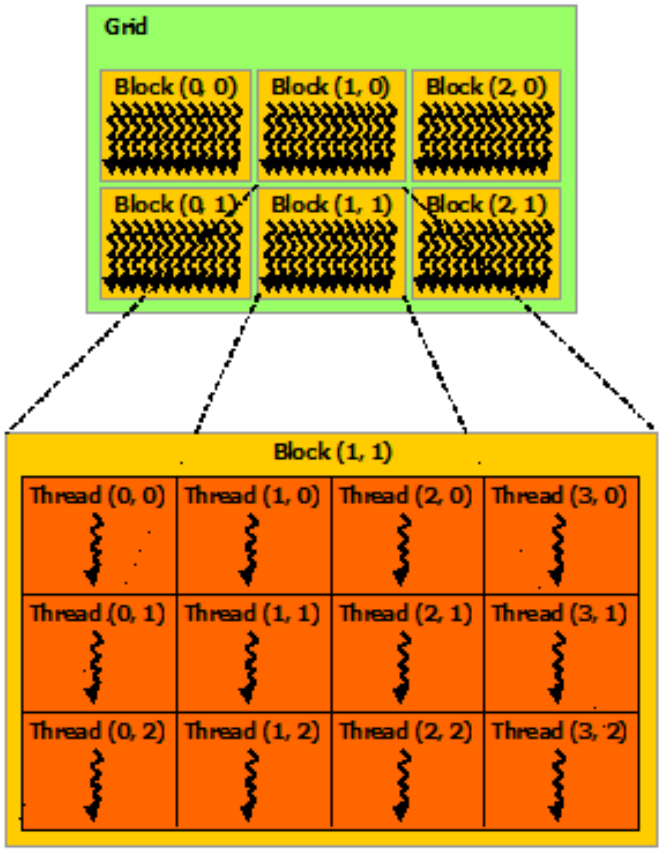


Thread Blocks

- Divide monolithic thread array into multiple blocks
  - Threads within a block cooperate via shared memory, atomic operations and barrier synchronization
  - Threads in different blocks cannot cooperate



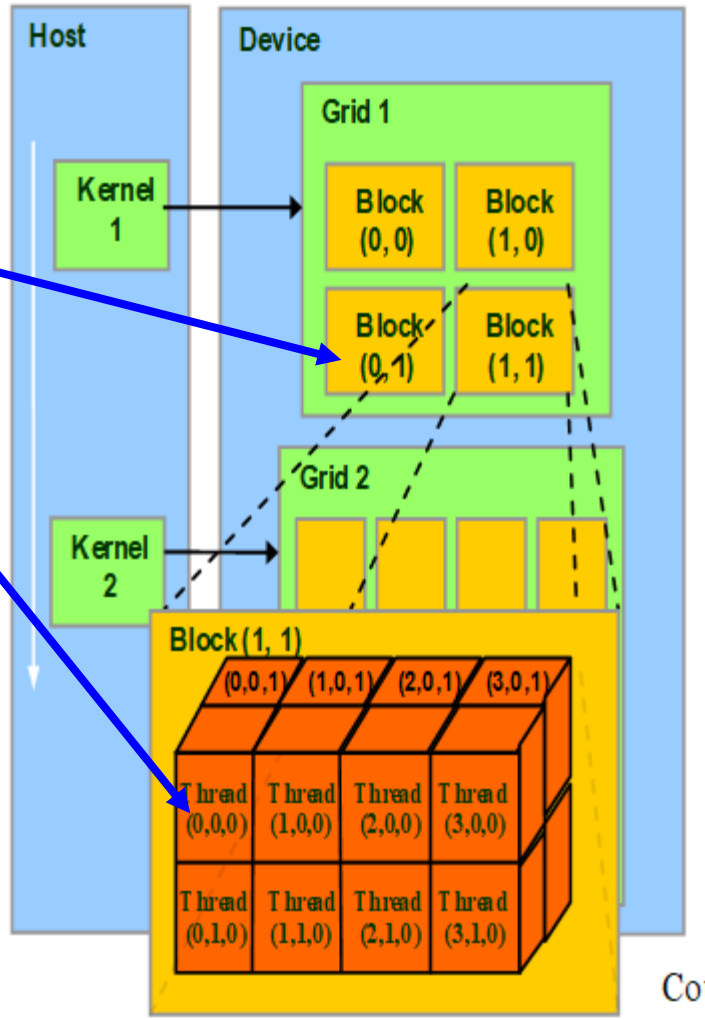
- Kernel creates a Grid
- Grid is composed of Blocks
- Block is composed of Threads



Block IDs and Thread IDs

- Each thread uses IDs to decide what data to work on
  - Block ID: 1D or 2D
  - Thread ID: 1D, 2D, or 3D

- Simplifies memory addressing when processing multidimensional data
  - Image processing
  - Solving PDEs on volumes
  - …



CUDA Memory Model

- Global memory
  - Main means of communicating data between host and device
  - Contents visible to all threads (threads can access the contents)
  - Long access latency
- Shared memory
  - Used by threads in the same block
  - Main means of communicating data among threads in the same block

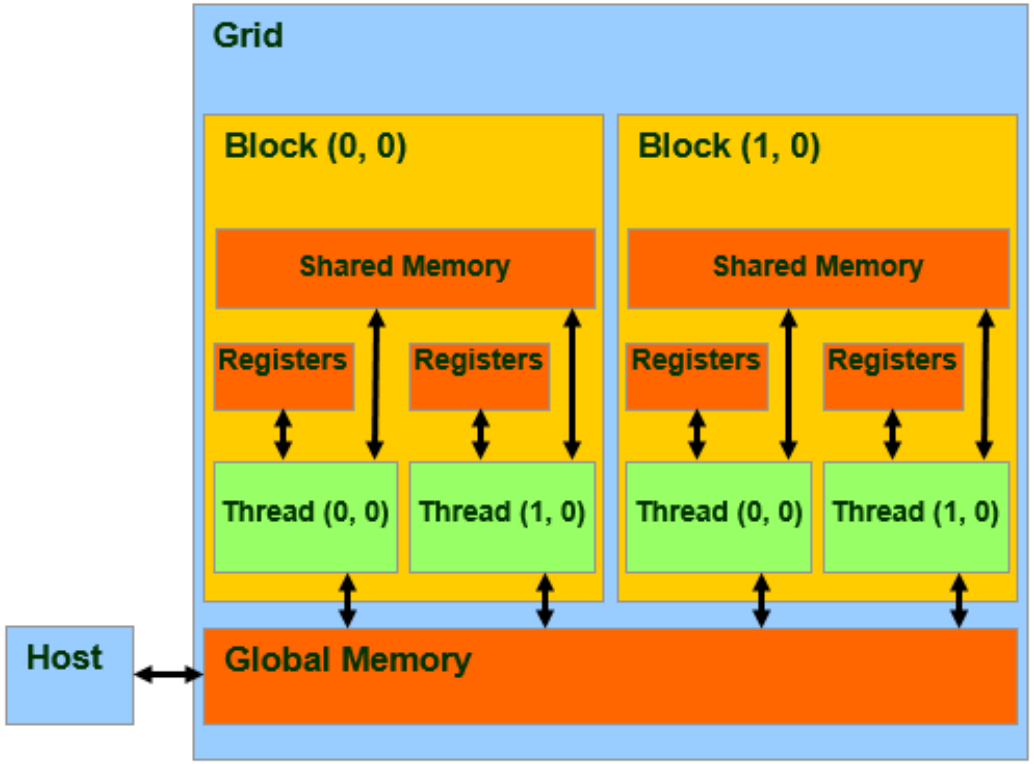
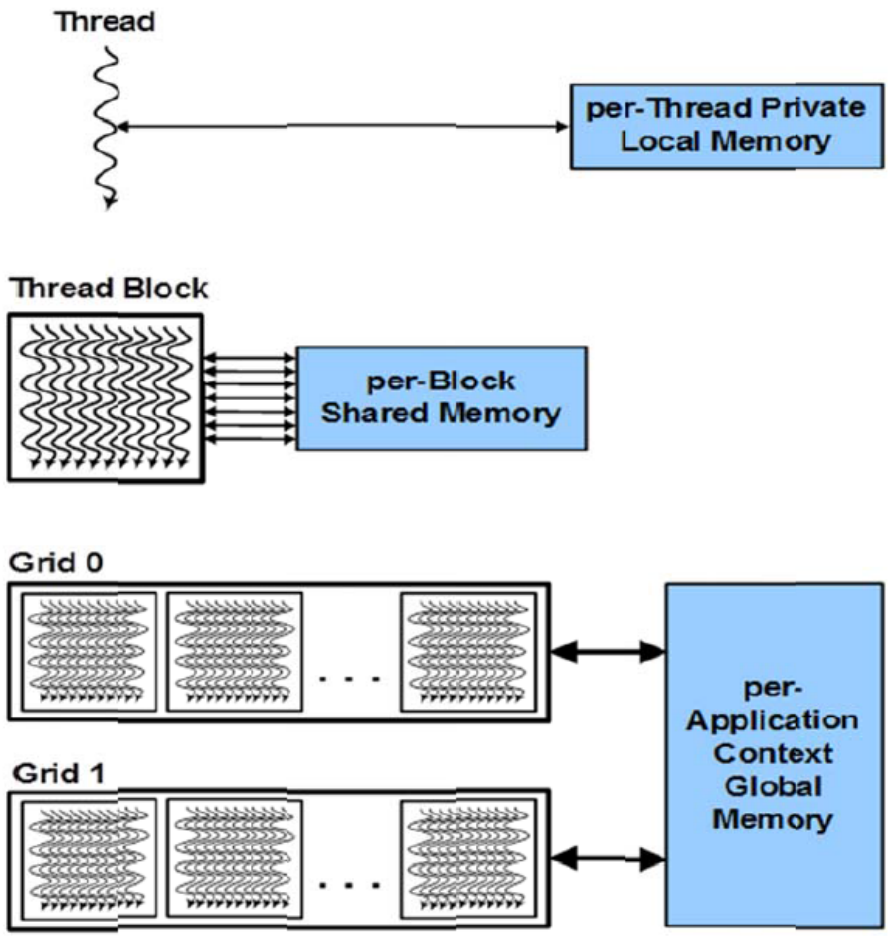


Figure 1: CUDA Hierarchy of threads, blocks, and grids, with corresponding per-thread private, per-block shared, and per-application global memory spaces.



CUDA Device Memory Allocation

- cudaMalloc( )
  - Allocates object in the device Global Memory
  - Requires two parameters
    - **Address of a pointer** to the allocated object
    - **Size of** allocated object

- cudaFree( )
  - Frees object from device Global Memory
    - **Pointer** to freed object

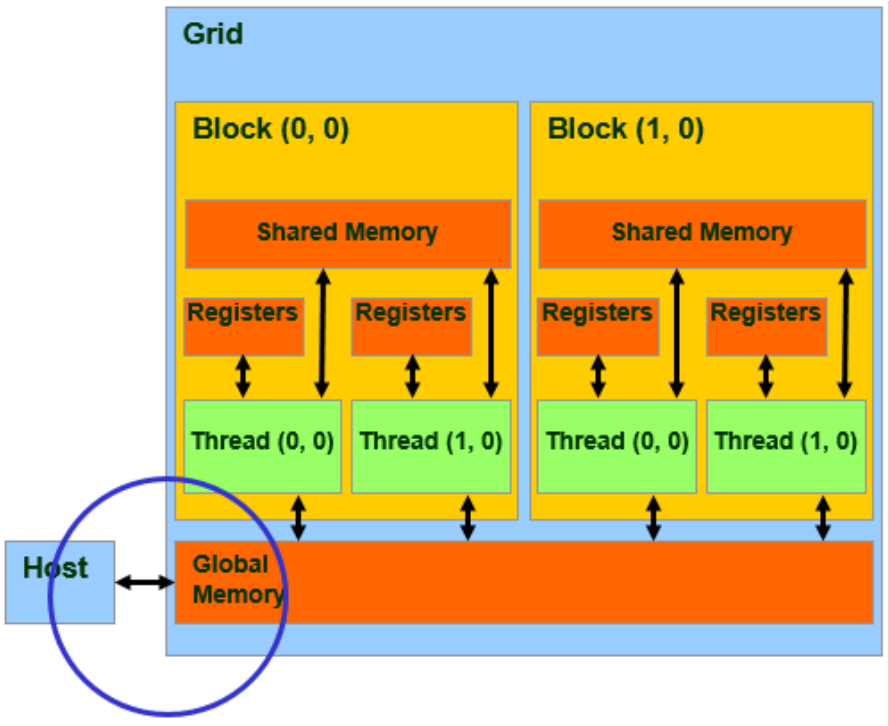
CUDA Device Memory Allocation (cont'd)

- Code example:
  - Allocate a 64 * 64 single precision float array
  - Attach the allocated storage to Md
  - "d" is often used to indicate a device data structure

```
TILE_WIDTH = 64;
Float* Md
int size = TILE_WIDTH * TILE_WIDTH * sizeof(float);

cudaMalloc((void**)&Md, size);

cudaFree(Md);
```

CUDA Host-Device Data Transfer

- cudaMemcpy( )
  - Memory data transfer
  - Requires four parameters
    - Pointer to destination
    - Pointer to source
    - Number of bytes copied
    - Type of transfer
      - Host to Host
      - Host to Device
      - Device to Host
      - Device to Device
- Asynchronous transfer



CUDA Host-Device Data Transfer (cont'd)

- Code example:
  - Transfer a 64 * 64 single precision float array
  - M is in host memory and Md is in device memory
  - cudaMemcpyHostToDevice, and
    cudaMemcpyDeviceToHost

cudaMemcpy(Md, M, size, cudaMemcpyHostToDevice);

cudaMemcpy(M, Md, size, cudaMemcpyDeviceToHost);

CUDA Keywords

- CUDA Function Declarations

	Executed on the	Only callable from the:
`__device__ float DeviceFunc()`	device	device
`__global__ void KernelFunc()`	device	host
`__host__ float HostFunc()`	host	host

- `__global__` defines a kernel function **`(void)`**
- `__device__` and `__host__` can be used together

Calling a Kernel Function – Thread Creation

- A kernel function must be called with an execution configuration:

```
__global__ void KernelFunc(...);
dim3   DimGrid(100,50);     // 5000 thread blocks
dim3   DimBlock(4,8,8);    // 256 threads per block
size_t SharedMemBytes=64; // 64 bytes of shared memory

KernelFunc <<< DimGrid,DimBlock,SharedMemBytes >>>
    (...);
```

- Kernel 함수: 데이터 병렬성을 높이기 위하여 많은 수의 스레드(thread)들을 생성

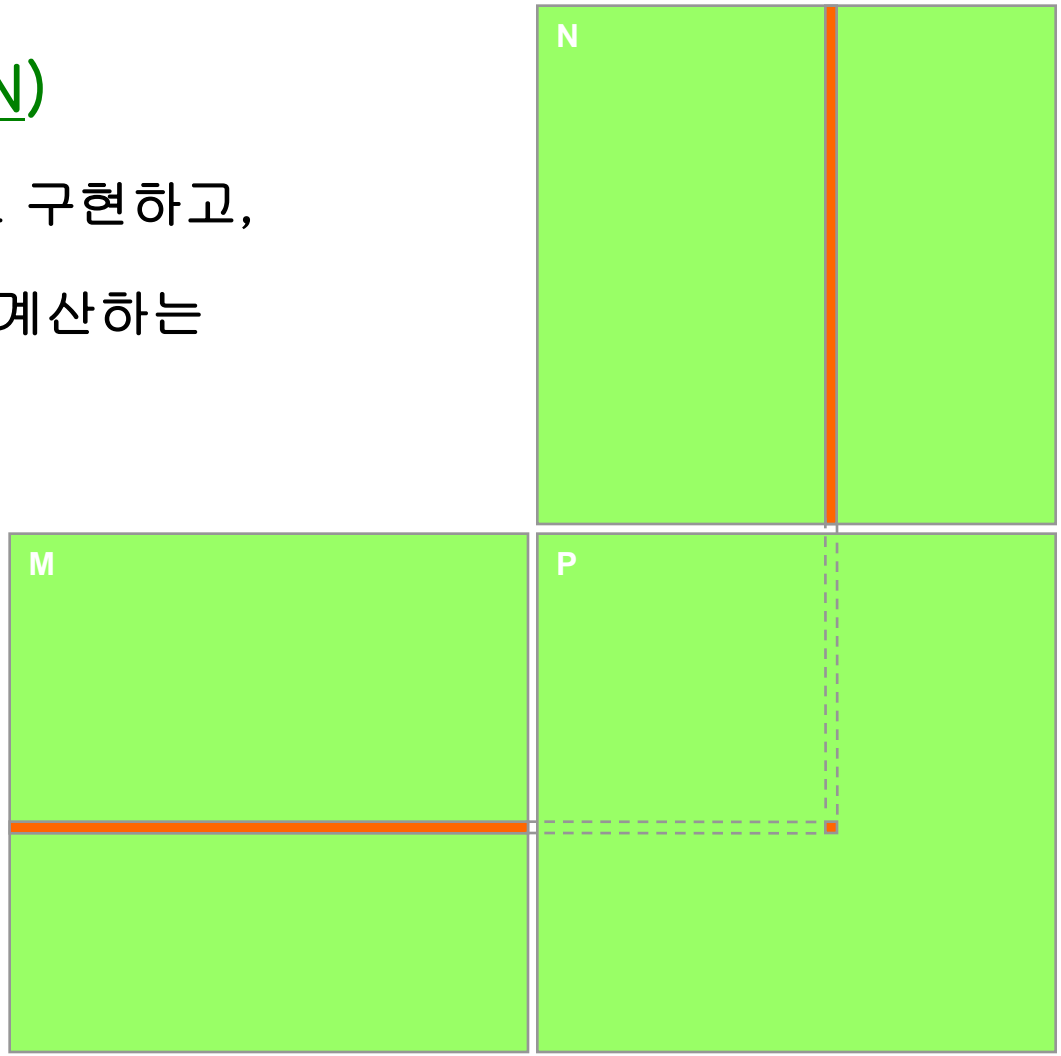
<개별 과제>

행렬 곱셈: (P = M x N)

전체를 하나의 커널 함수로 구현하고,

출력 행렬(P)의 각 원소를 계산하는

부분은 스레드로 구현



행렬 곱셈을 위한 CUDA 프로그램 구조

```
void MatrixMultiplication(float* M, float* N, float* P, int Width)
{
    int size = Width * Width * sizeof(float);
     float* Md, Nd, Pd;
    …

// Allocate device memory for M, N and P
 // and copy M and N to allocated device memory location

 // Kernel invocation code to let the device perform the actual
    multiplication
      …

 // Read P from the device

 // Free device matrices
```

NVIDIA GPUs

GPU	G80	GT200	Fermi
Transistors	681 million	1.4 billion	3.0 billion
CUDA Cores	128	240	512
Double Precision Floating Point Capability	None	30 FMA ops / clock	256 FMA ops /clock
Single Precision Floating Point Capability	128 MAD ops/clock	240 MAD ops / clock	512 FMA ops /clock
Special Function Units (SFUs) / SM	2	2	4
Warp schedulers (per SM)	1	1	2
Shared Memory (per SM)	16 KB	16 KB	Configurable 48 KB or 16 KB
L1 Cache (per SM)	None	None	Configurable 16 KB or 48 KB
L2 Cache	None	None	768 KB
ECC Memory Support	No	No	Yes
Concurrent Kernels	No	No	Up to 16
Load/Store Address Width	32-bit	32-bit	64-bit