Introduction to CUDA Programming

한재근 과장 | Solutions Architect

jahan@nvidia.com

TOPICS

- GPGPU 프로그래밍 개요
- CUDA Programming 모델
- CUDA Programming 기초
- Parallel Programming Pattern with CUDA
- CUDA Debugging
- OpenACC 개요
- GPU 최적화
- CUDA Library 소개
- DirectGPU 기술
- Multi-GPU Programming

GPGPU 프로그래밍 개요

GPU Computing

NVIDIA.

CPU vs GPU

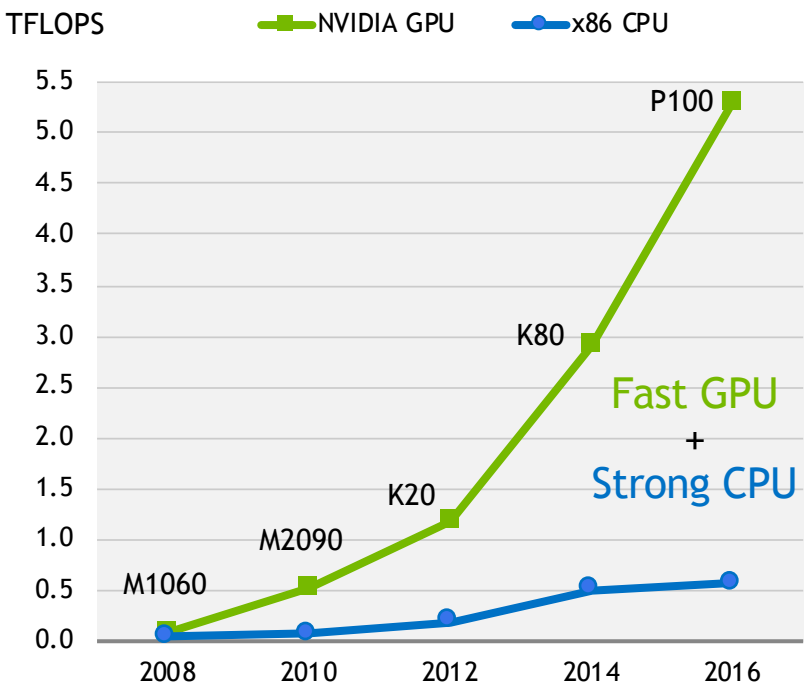
CPU
Multiple Cores

GPU
Many Cores

GPU ACCELERATED COMPUTING PLATFORM

Focused on Co-Design for Accelerated Data Center

Fast GPU Engineered for High Throughput



Productive Programming Model & Tools

NVIDIA CUDA

OpenACC
Directives for Accelerators

Expert Co-Design

APPLICATION

MIDDLEWARE

SYS SW

LARGE SYSTEMS

PROCESSOR

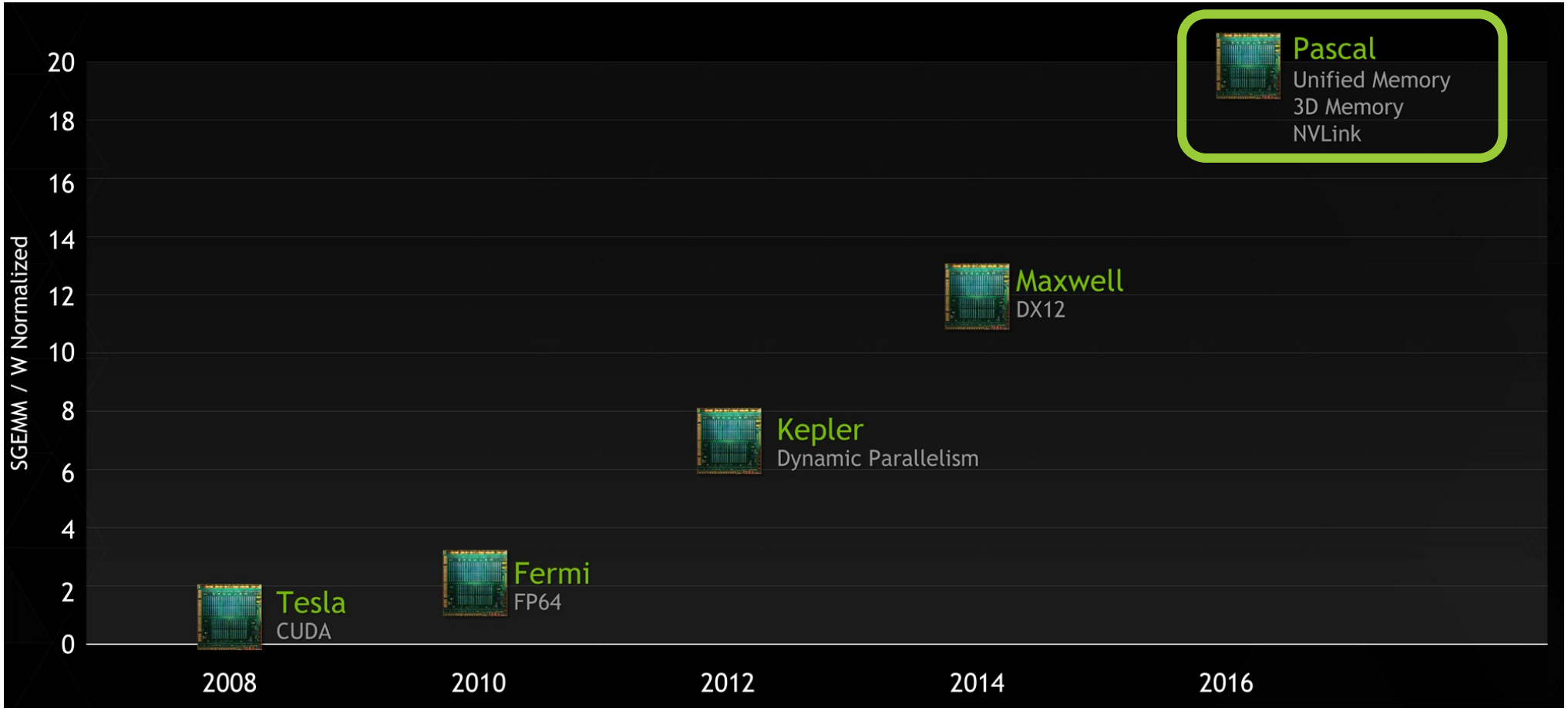
Accessibility

阿里云计算 Alibaba Cloud Computing

amazon web services™ IBM SOFTLAYER

Microsoft Azure

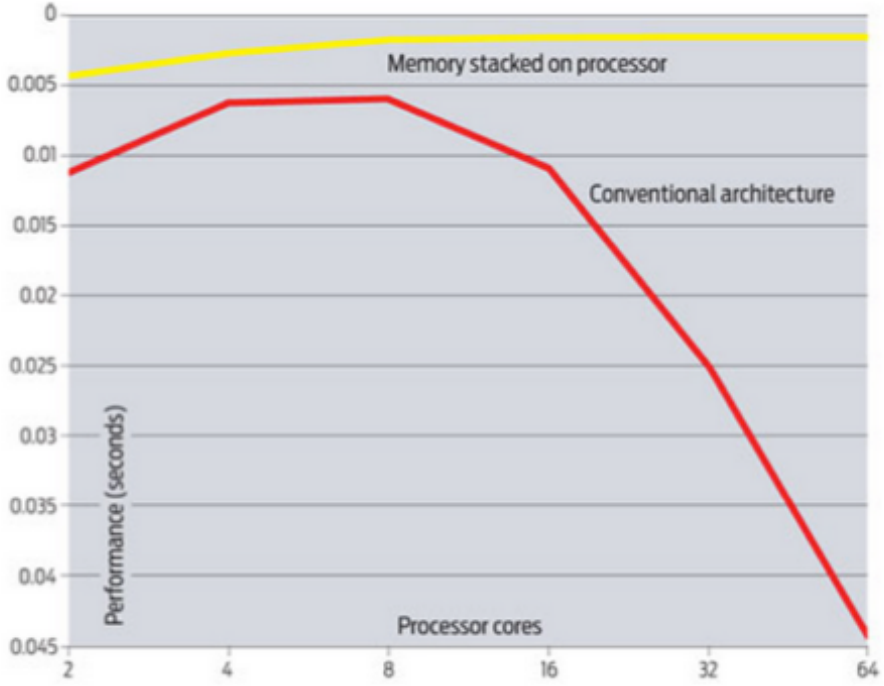
NVIDA GPU Roadmap



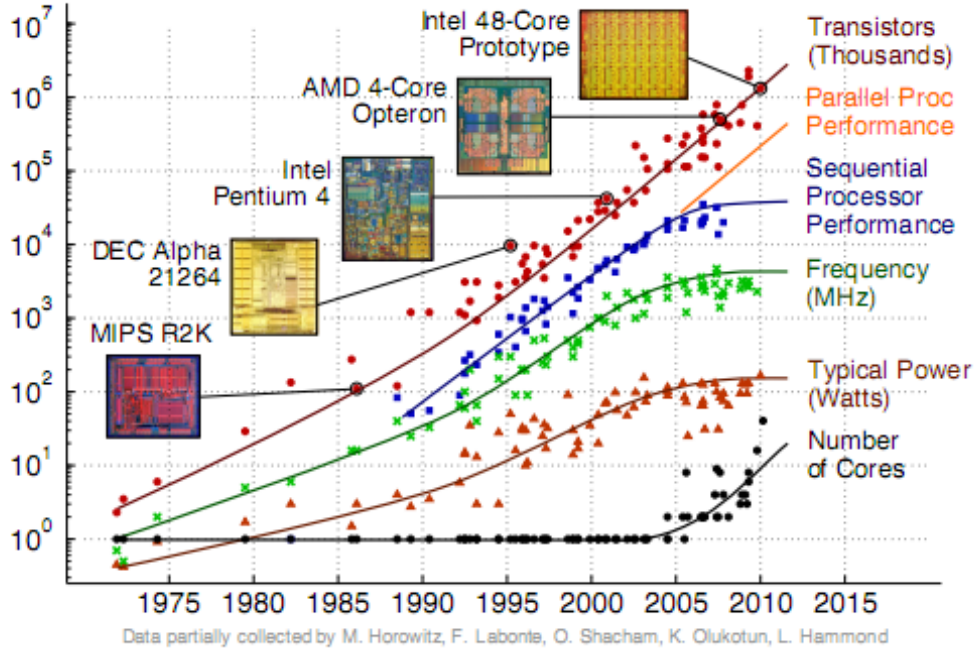
NVIDIA.

CPU의 성능향상이 제한되는 이유

- CPU가 사용하는 공유메모리 기반 멀티코어 프로세서는 Core 개수에 제한이 있음
  - 공유메모리 통신 대역폭 및 동기화 등이 성능 제한 요인
- 고성능 연산을 위해 분산된 코어 및 메모리 구조를 가진 아키텍처가 필요함: GPU!!



https://share.sandia.gov/news/resources/news_releases/more-chip-cores-can-mean-slower-supercomputing-sandia-simulation-shows/#.V4kBjUaLRD9

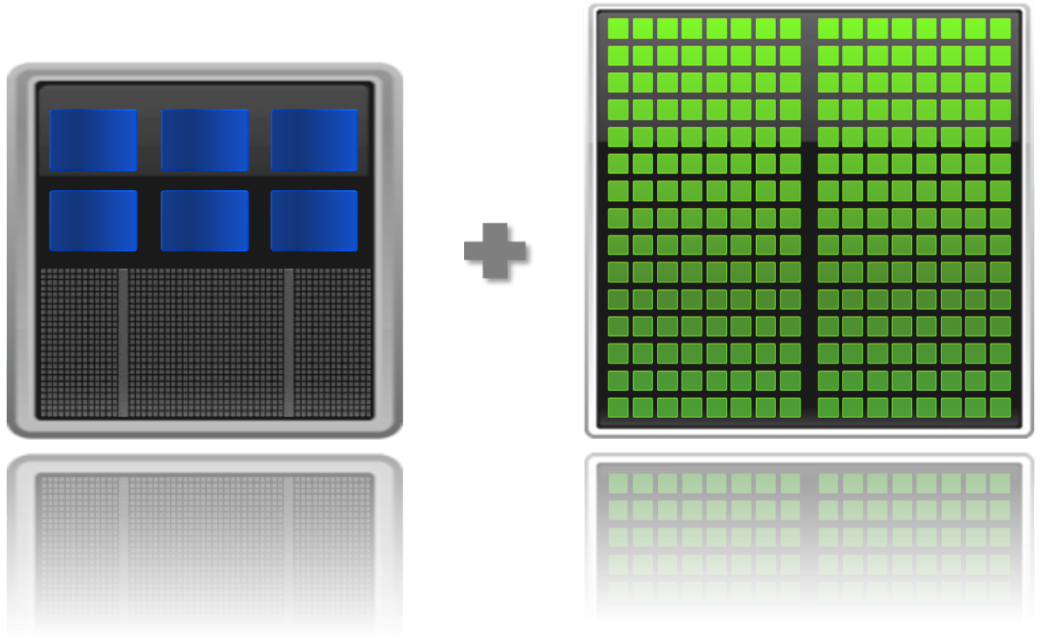


Prepared by C. Batten - School of Electrical and Computer Engineering - Cornell University - 2005 - retrieved Dec 12 2012 - http://www.csl.cornell.edu/courses/ece5950/handouts/ece5950-overview.pdf

Heterogeneous Computing with GPU

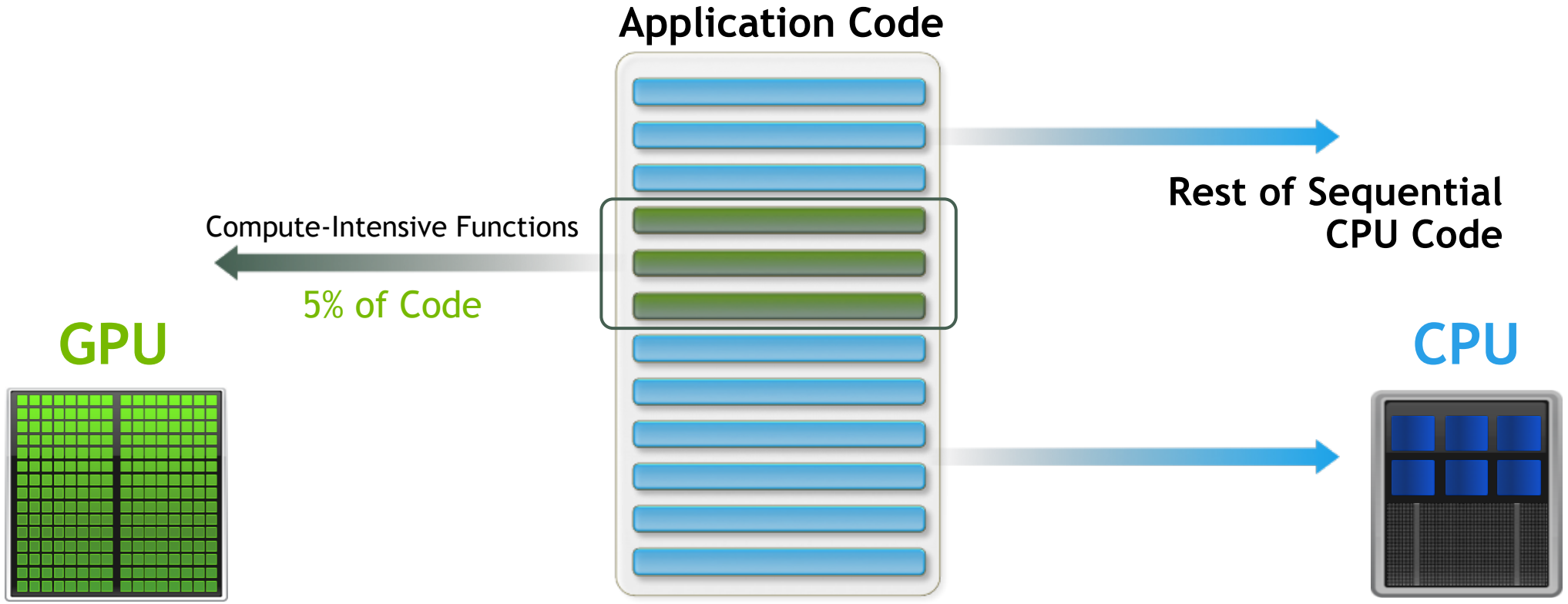
CPU
Serial Tasks

GPU
Parallel Tasks



NVIDIA.

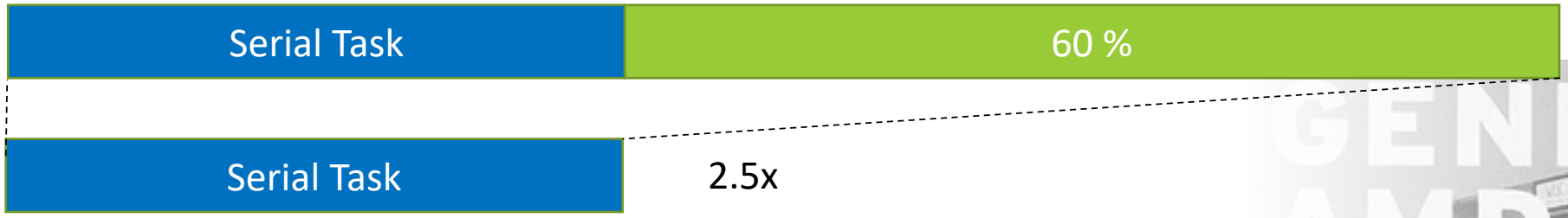
Heterogeneous Computing의 개념

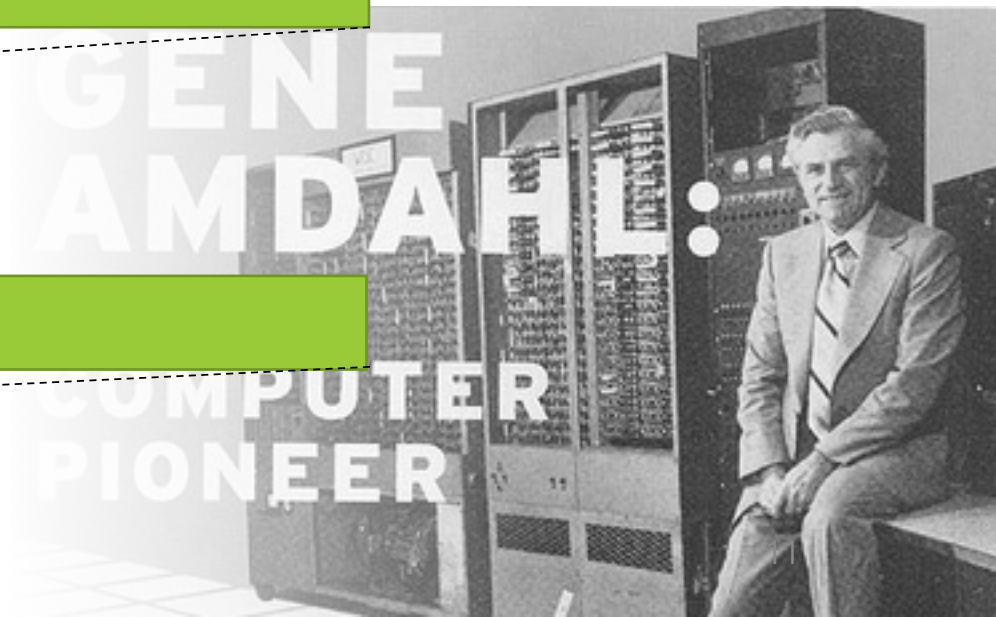


NVIDIA.

Amdahl's Law

$$S_{latency}(s) = \frac{1}{(1-p) + \frac{p}{s}}$$





CUDA

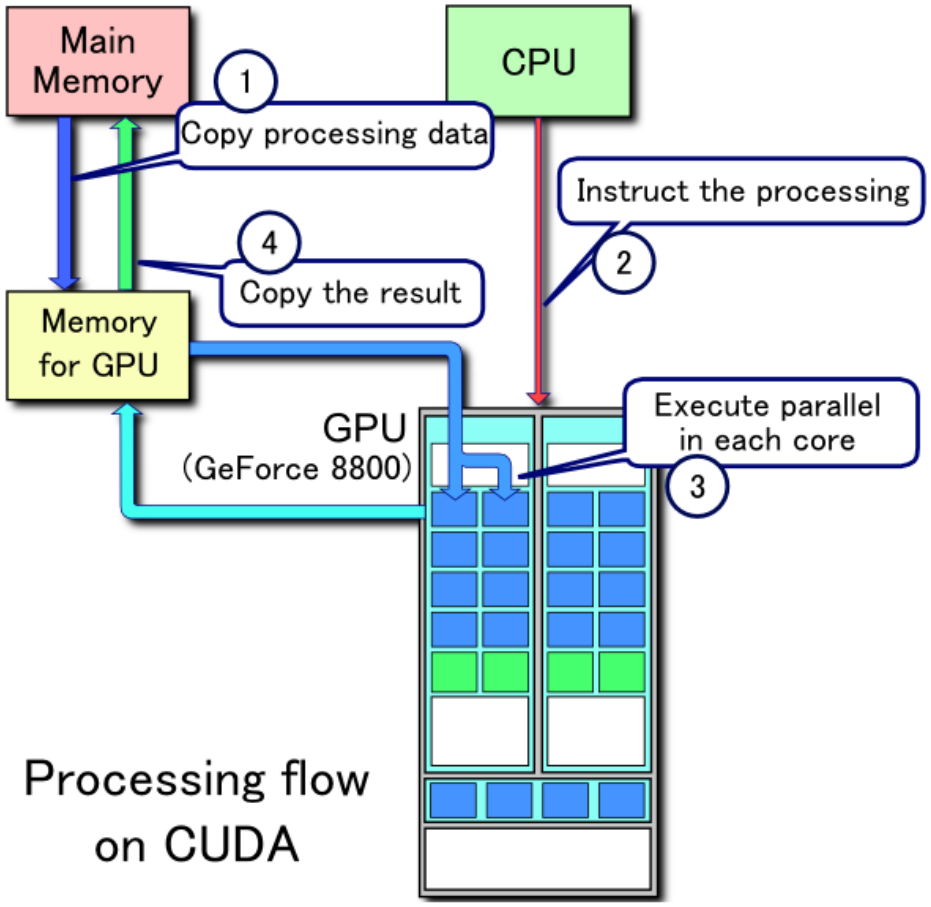
Common Unified Device Architecture

CUDA란?

- NVIDIA GPGPU 프로그래밍 플랫폼
- C 언어를 기반으로 한 GPU 프로그래밍 가능
  - 다양한 언어 및 라이브러리와 함께 사용가능
- CUDA 가속 지원 라이브러리와 연동하여 다양한 어플리케이션 개발 가능

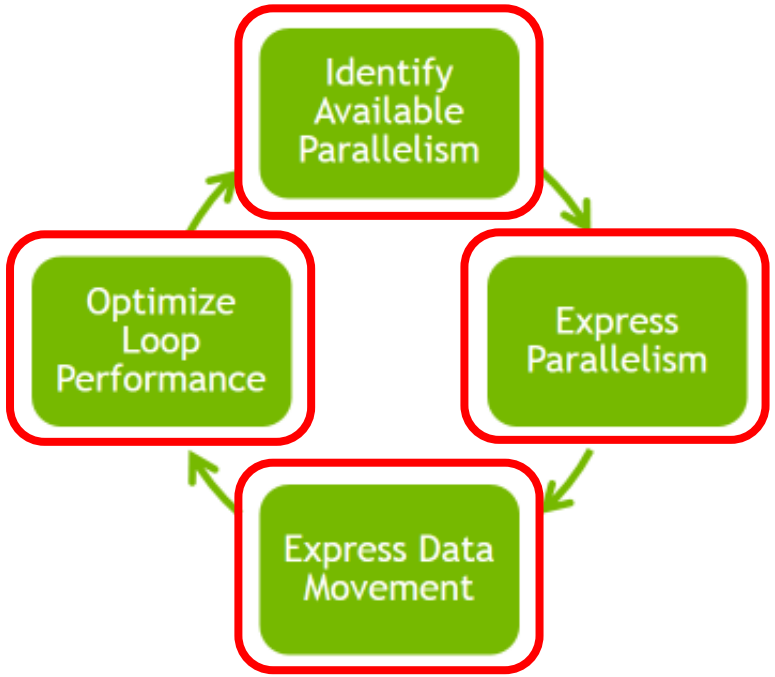
NVIDIA.

CUDA 실행 절차


Main Memory
CPU
1
Copy processing data
Instruct the processing
2
4
Copy the result
Memory for GPU
GPU (GeForce 8800)
Execute parallel in each core
3
Processing flow on CUDA

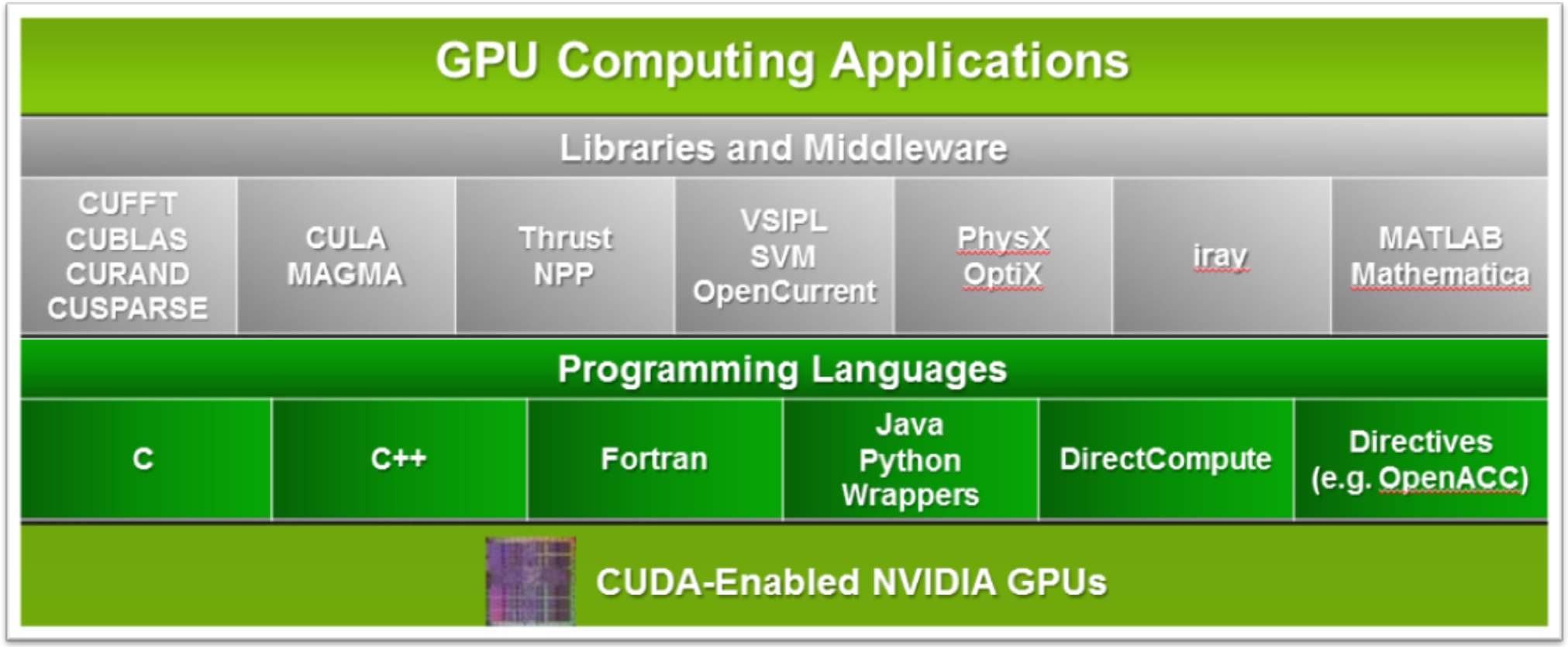
Application의 CUDA 가속 절차

- 성능 분석
- 병목 탐색
- 알고리즘 분석 및 병렬화 전략 수립
- 데이터 의존성 해결
- 병렬화
- 최적화



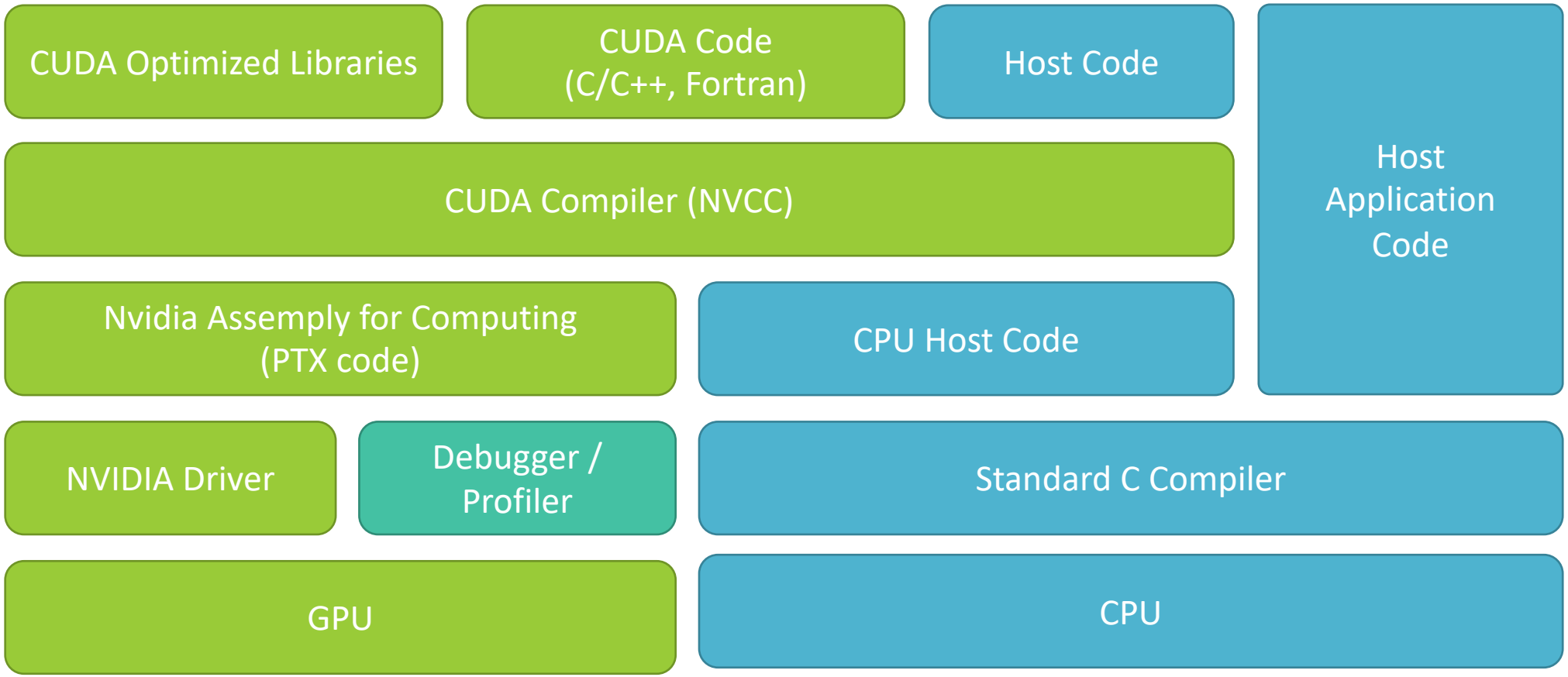
NVIDIA.

CUDA = NVIDA GPU 컴퓨팅



NVIDIA.

CUDA Compiler? = NVCC



NVIDIA.

그래서 어떤 그래픽카드를 사야하나요

- 모든 NVIDIA 그래픽카드 지원
- 최신 카드를 살 수록 유리
  - 최신 architecture 명: Pascal (2016)
  - Maxwell(2014), Kepler(2012), Fermi(2010)
- Application의 종류에 따라 선택 필요
  - DP연산, 대용량 연산엔 TESLA 필수

서버/연산용

워크스테이션

임베디드

게임밍/데스크탑

CUDA 프로그래밍 모델

1. CUDA Kernel

- GPU에 동작하는 함수를 Kernel이라 함

Device Code

```
__global__ void MatAdd(float A[N][N], float B[N][N], float C[N][N])
{
    . . .
}
```

- 호출 시 GPU에게 동작 조건을 명시함

Host Code

```
dim3 threadsPerBlock(N, N);
MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
```

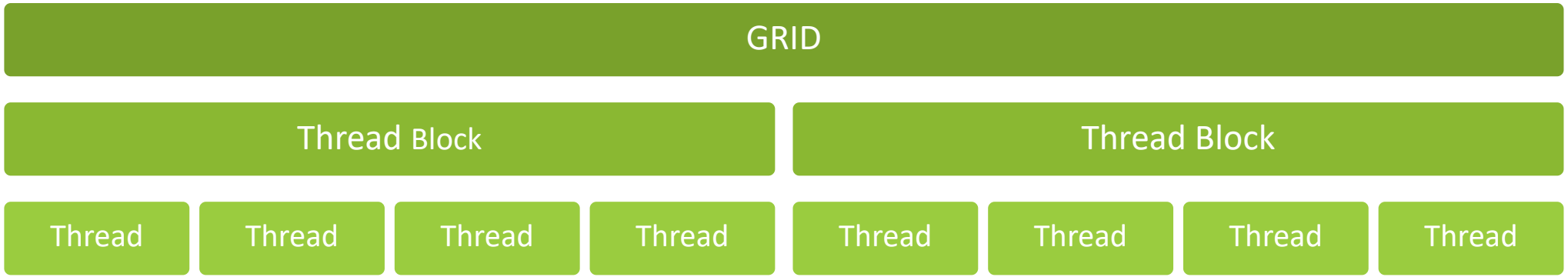
NVIDIA.

2. 계층적인 CUDA Thread/Block 구성

- Thread 단위의 개념

	특징	Index
CUDA Thread	순차적으로 연산하는 Thread	x, y, z
Thread Block	CUDA 연산을 제어하기 위한 block	x, y, z
Grid	Kernal	Index 없음
Warp	CUDA의 최소 제어 Thread 묶음	32개 고정. Index 없음

- 계층의 구성



NVIDIA.

TESLA V100

21B transistors
815 mm²

80 SM
5120 CUDA Cores
640 Tensor Cores

16 GB HBM2
900 GB/s HBM2
300 GB/s NVLink

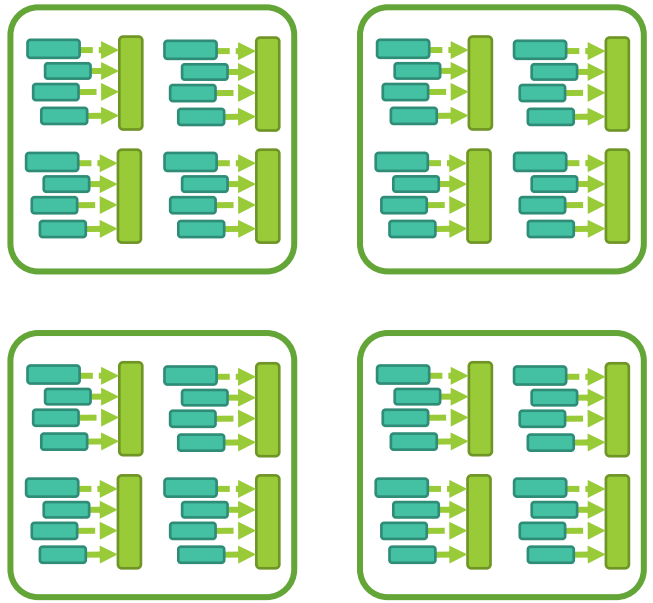
*full GV100 chip contains 84 SMs

VOLTA GV100 SM

	GV100
FP32 units	64
FP64 units	32
INT32 units	64
Tensor Cores	8
Register File	256 KB
Unified L1/Shared memory	128 KB
Active Threads	2048

CUDA SM과 CUDA Block의 관계

CUDA Blocks



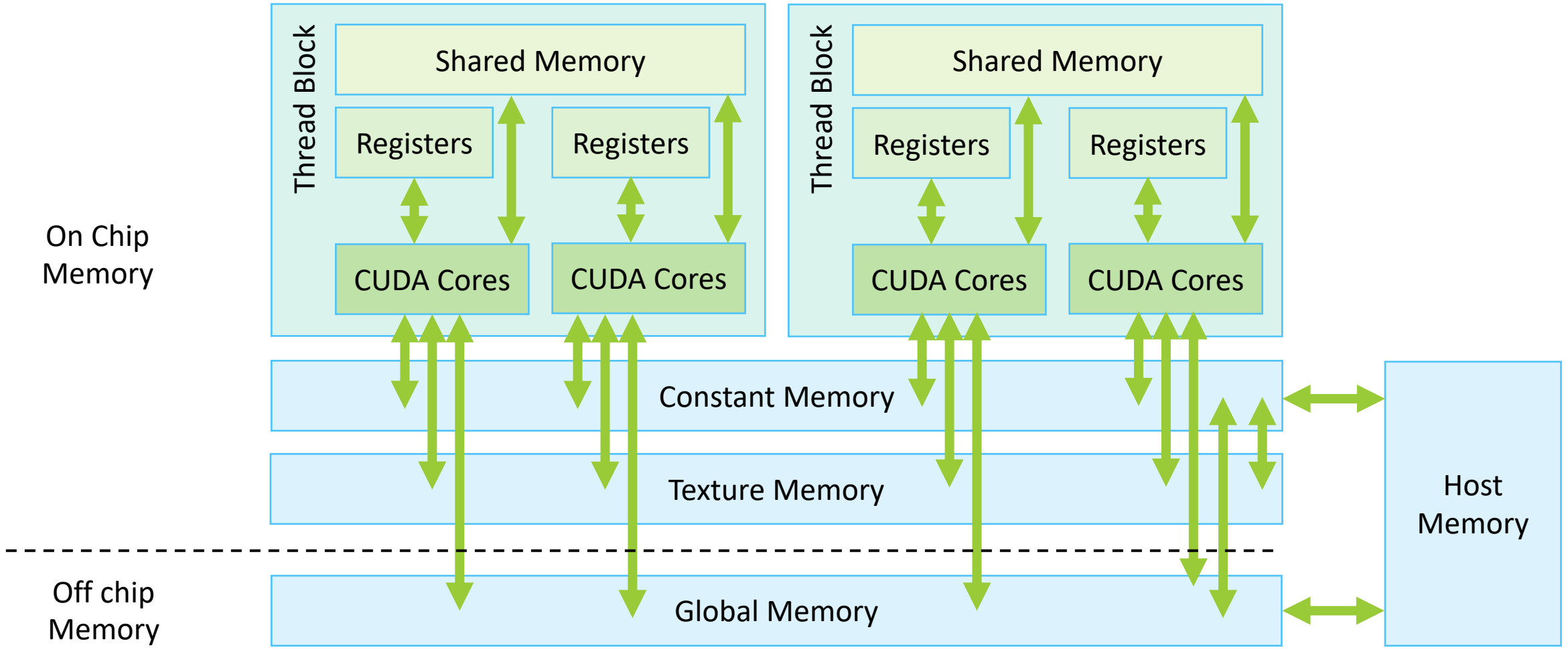
CUDA Block은 CUDA SM의 자원을 공유하는 범위로 Scheduling 됨

NVIDIA.

CUDA 실행 모델

NVIDIA.

3. 계층적인 공유 메모리 구조



NVIDIA.

GPU Architecture

NVIDIA.

VOLTA GV100 SM

	GV100
FP32 units	64
FP64 units	32
INT32 units	64
Tensor Cores	8
Register File	256 KB
Unified L1/Shared memory	128 KB
Active Threads	2048

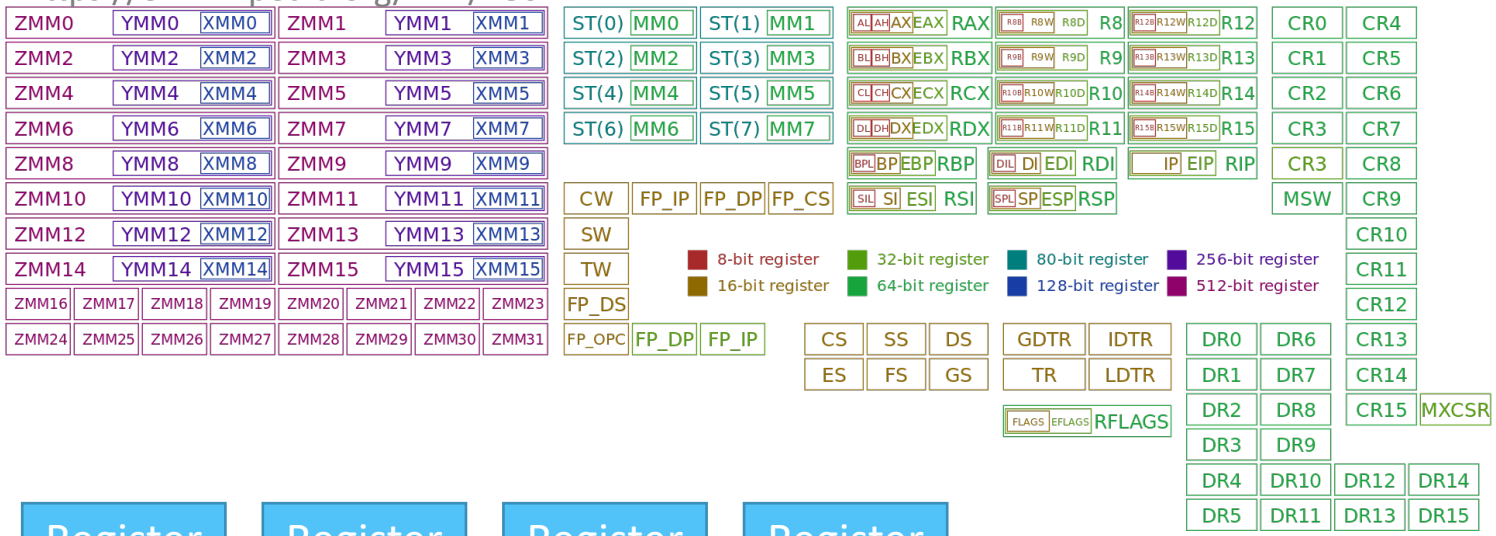
Texture Memory →

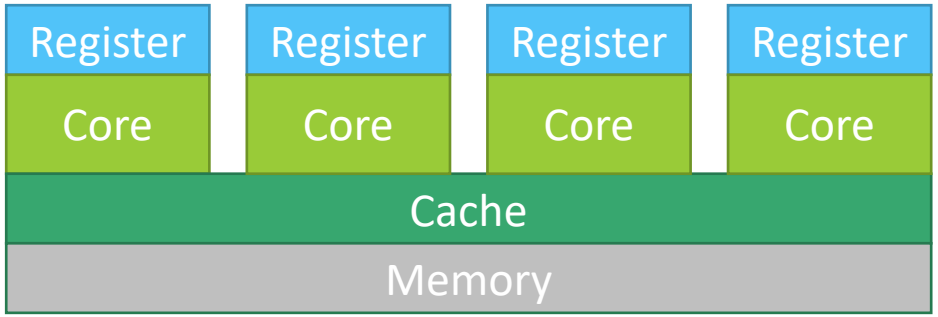
NVIDIA.

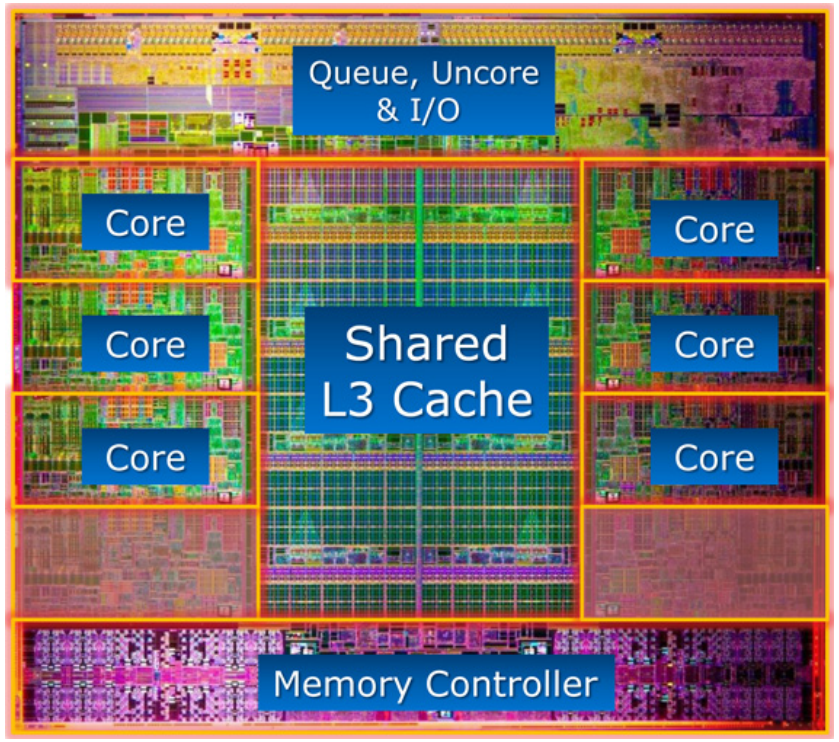
CPU Architecture

Registers available in the x86 instruction set

https://en.wikipedia.org/wiki/X86



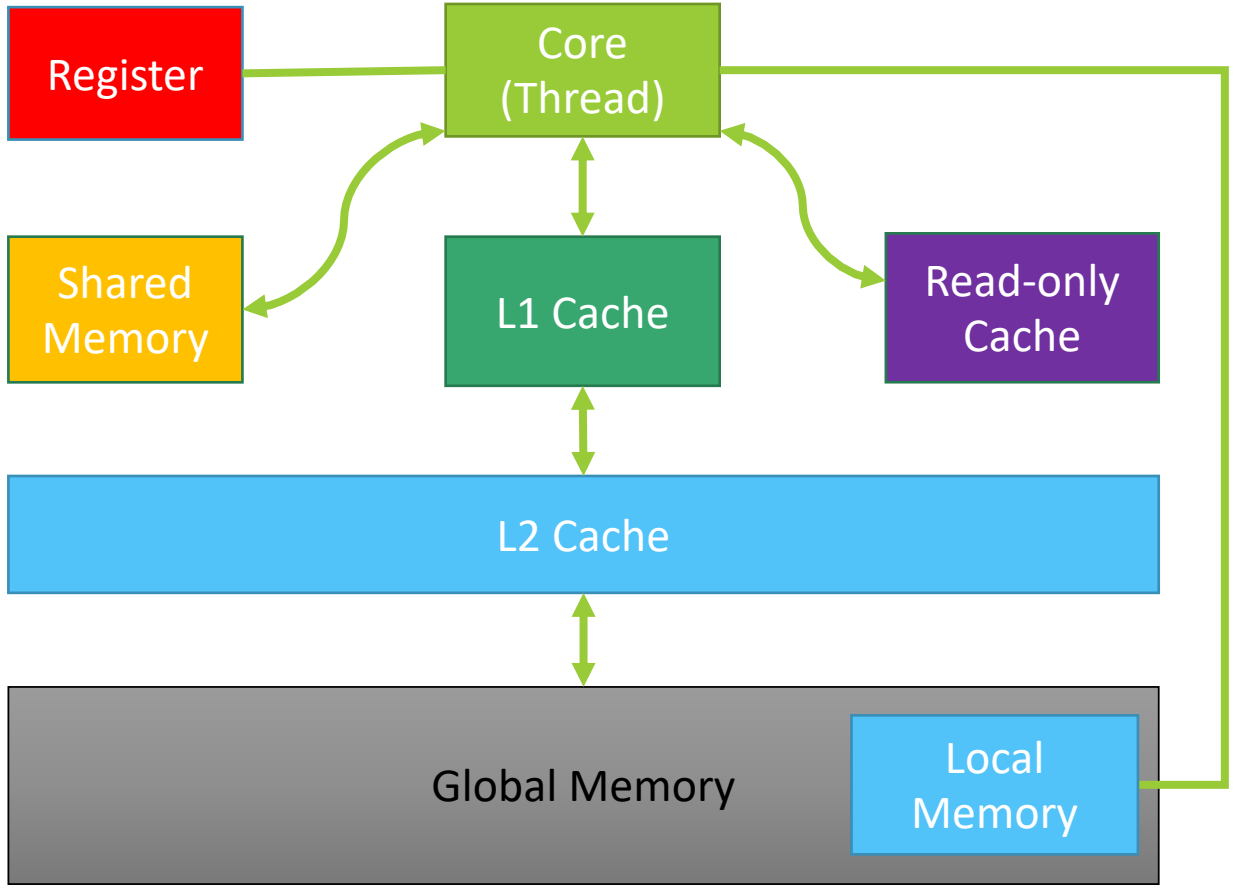




Core당 레지스터의 갯수와 목적이 정해져있음

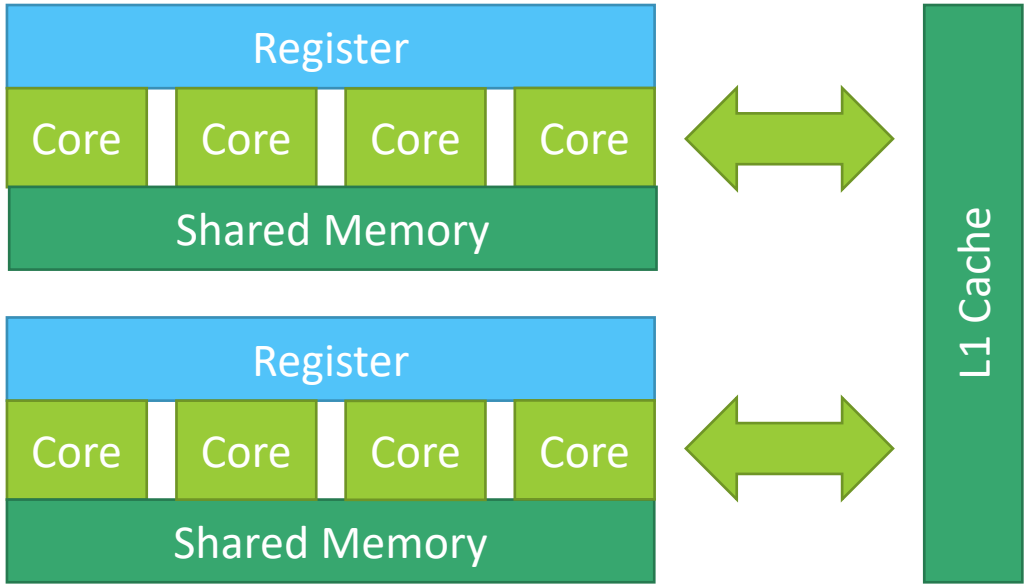
NVIDIA.

GPU Core & Memory



NVIDIA.

Register & Shared Memory Sharing

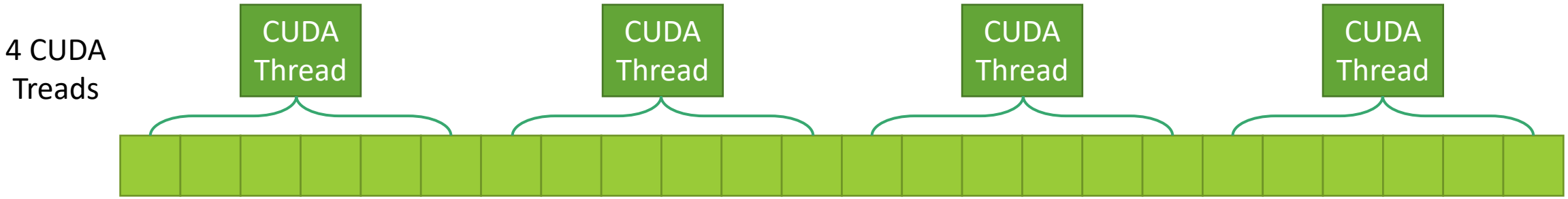


Application에 따라서 필요한 만큼의
Register를 할당받아 사용가능

NVIDIA.

Register File

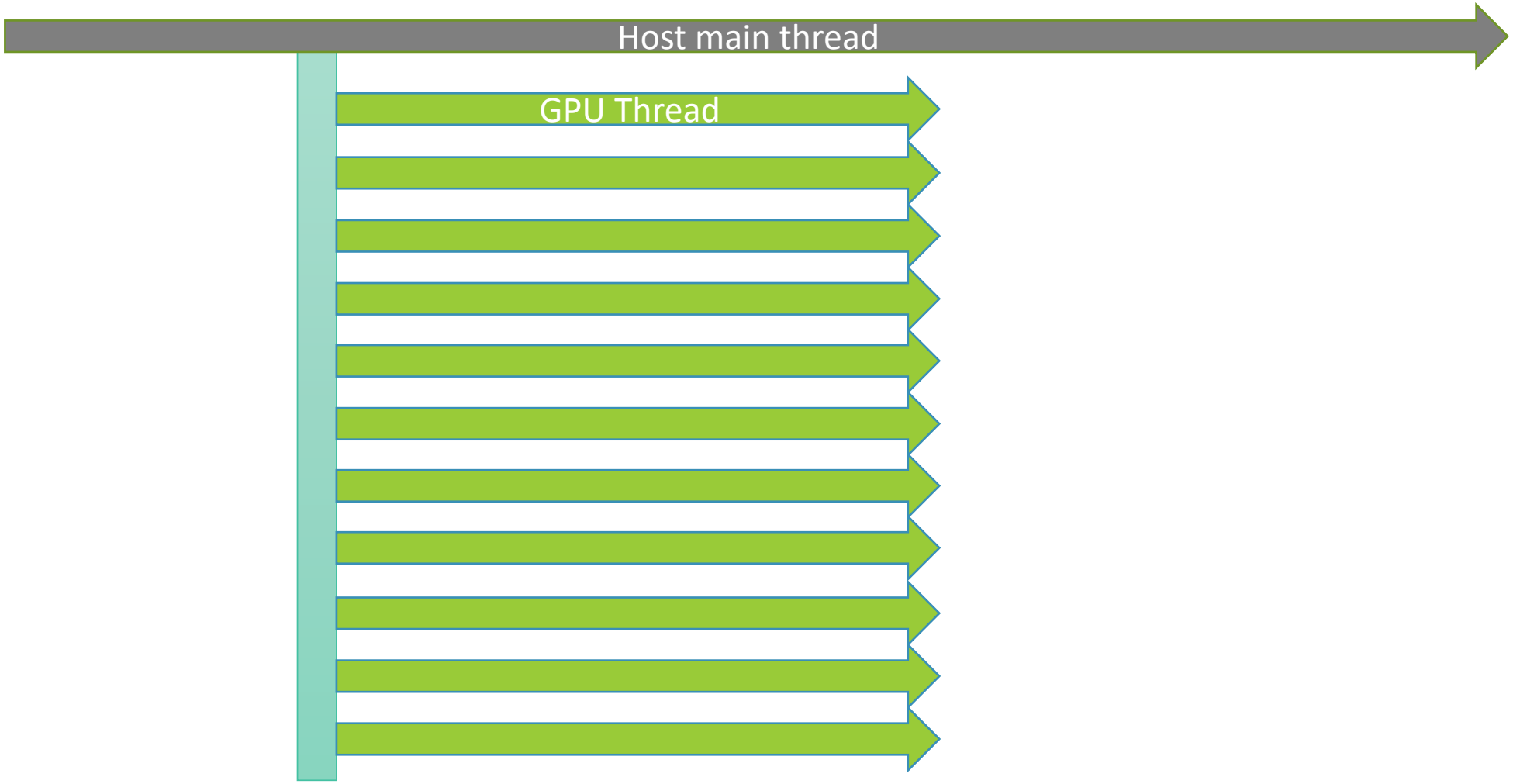
- CUDA는 SM당 64KB의 Register를 갖고 있음
- CUDA Thread가 필요한 만큼 Register를 나눠가짐
  - 필요한 양은 Compile 시점에 결정



- 여러개의 CUDA Thread가 동시에 동작 가능
- CUDA Core의 효율적인 활용 가능

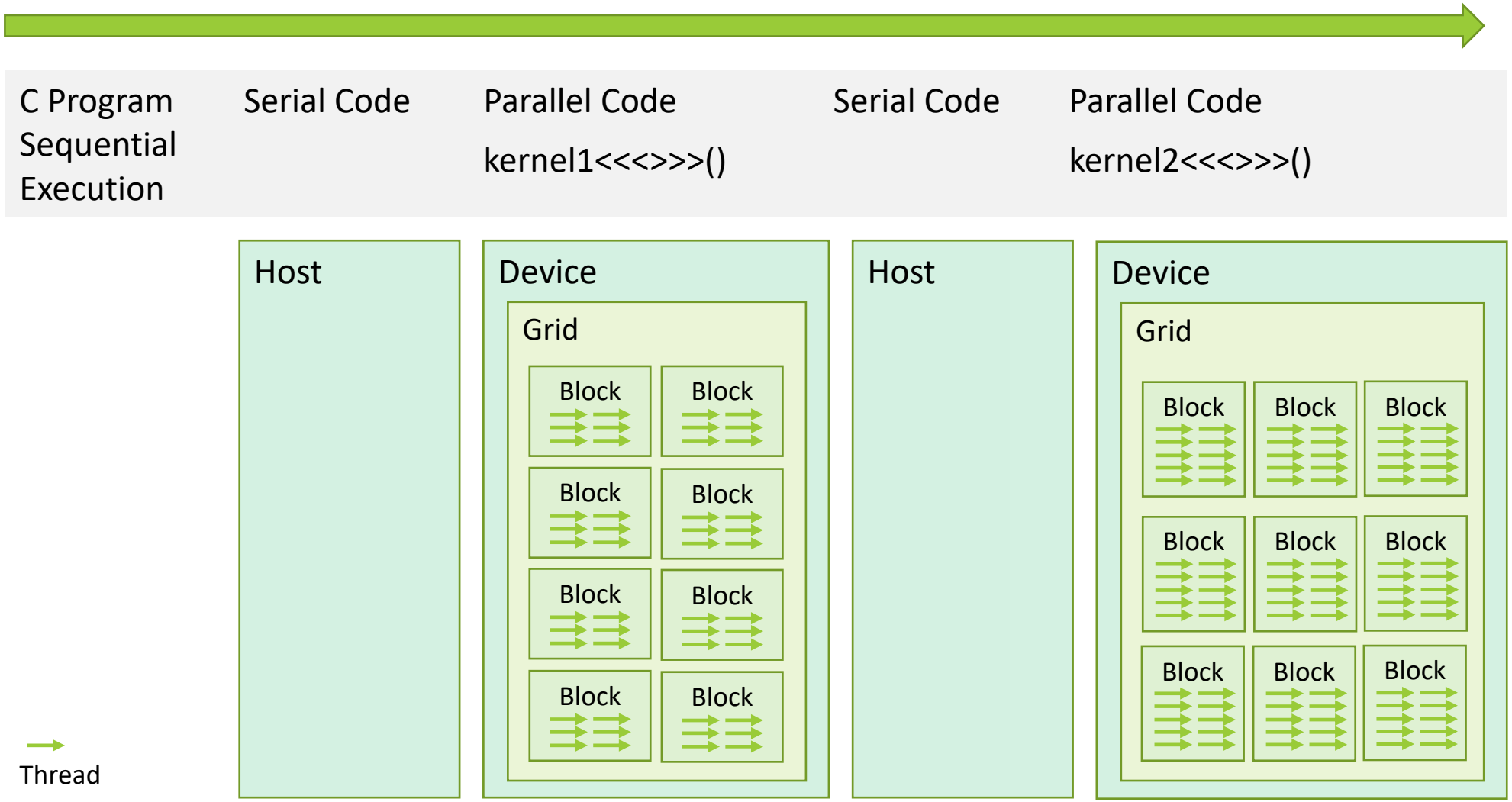
NVIDIA.

CUDA Parallel Operation

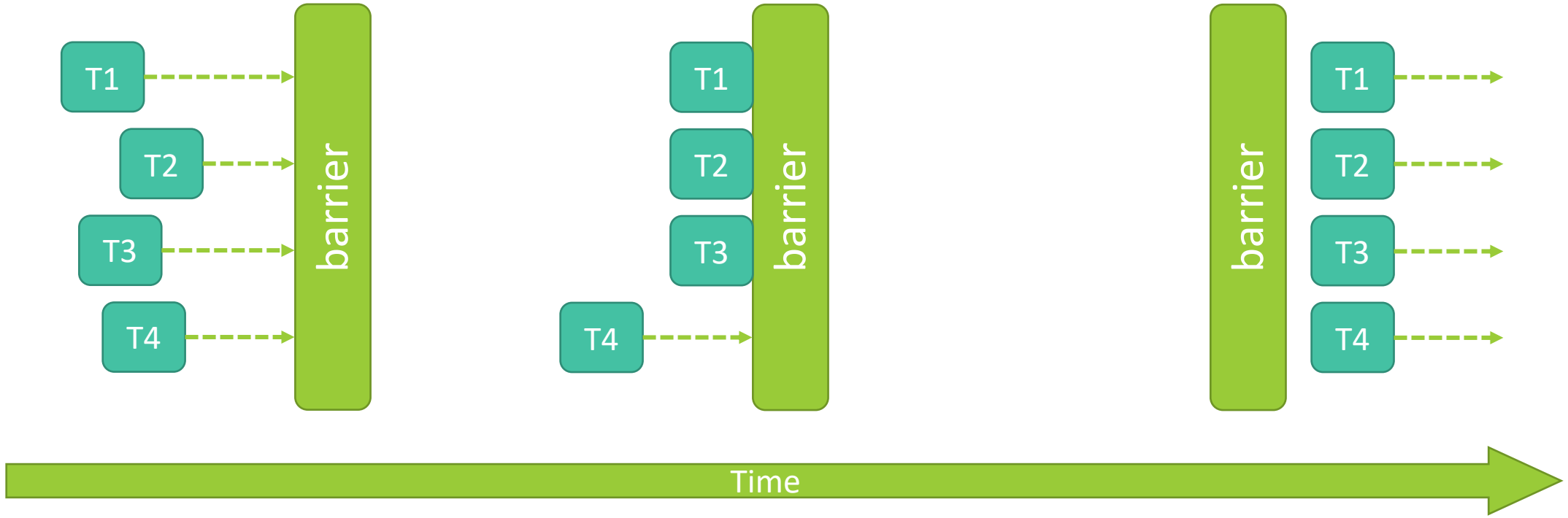


NVIDIA.

4. Heterogeneous Computing



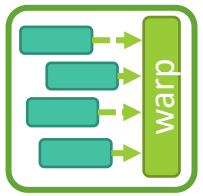
Barrier 동기화



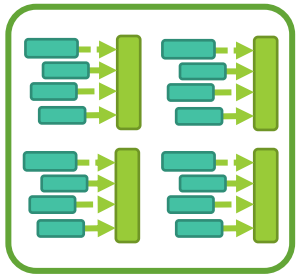
NVIDIA.

계층적 동기화 지원

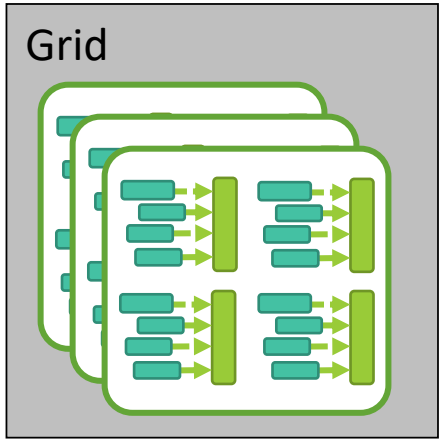
warp



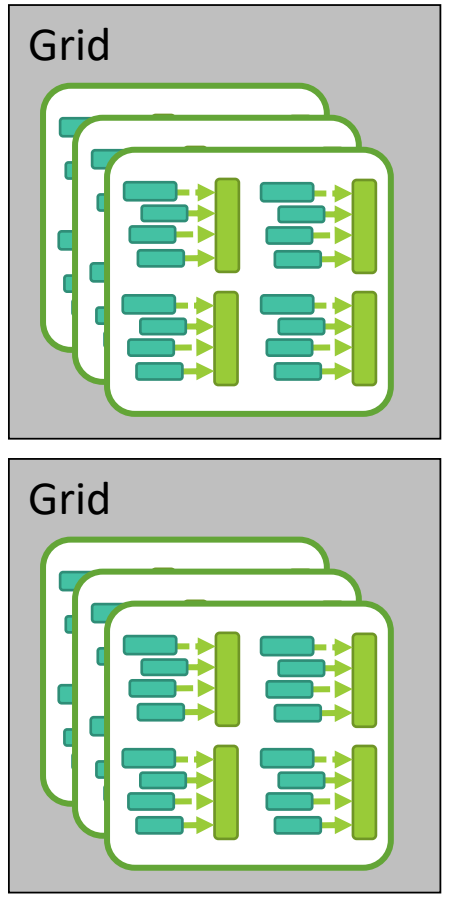
block



__syncthreads()



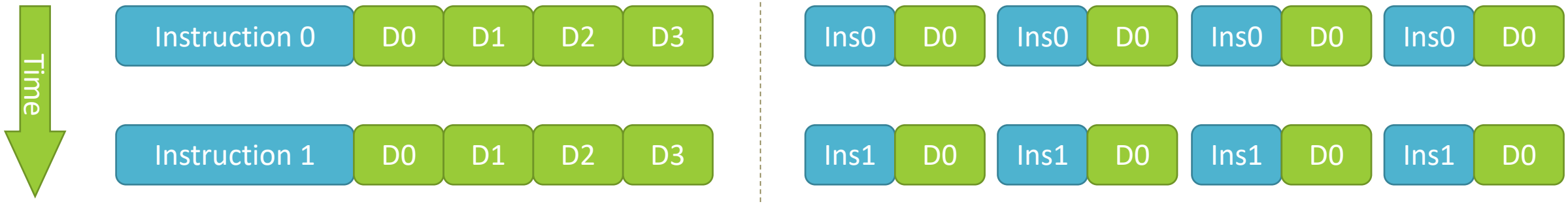
cudaStreamSynchronize ()



cudaDeviceSynchronize ()

SIMT 동작

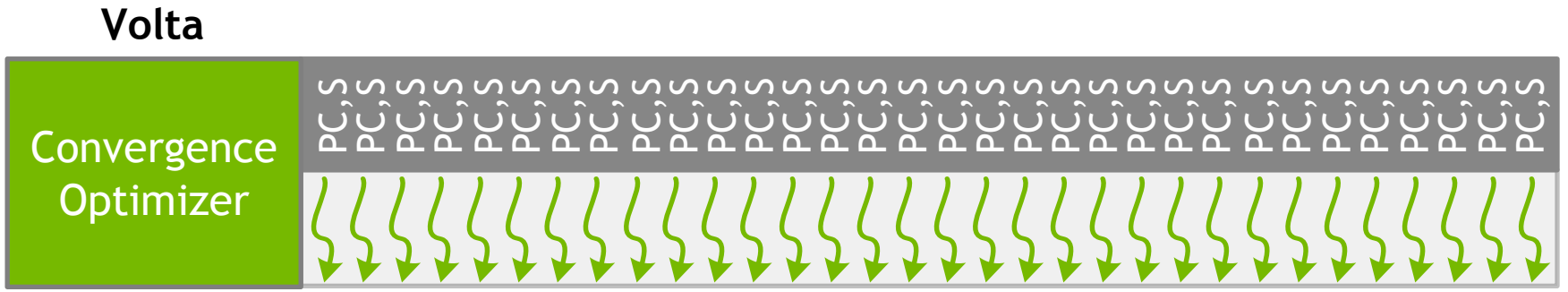
- Single Instruction Multiple Thread
- SIMD(Single Instruction Multiple Data) 유사
- Thread 별로 분기가 가능하기에 차이가 존재함



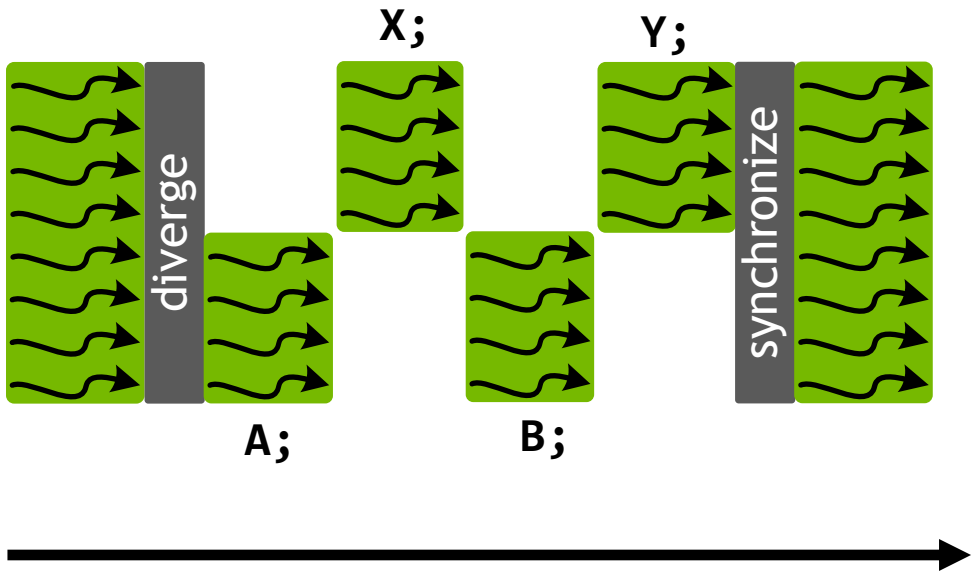
SIMD **SIMT**

NVIDIA.

최신 GPU의 Warp의 동작




```
if (threadIdx.x < 4) {
    A;
    __syncwarp();
    B;
} else {
    X;
    __syncwarp();
    Y;
}
__syncwarp();
```



NVIDIA.

5. Compute Capability

CUDA Compute Capability

- Target GPU Arch. verson

Version	Architecture 명
7.5	Turing
7.0	Volta
6.x	Pascal
5.x	Maxwell
3.x	Kepler
2.x	Fermi

CUDA Version

- CUDA API Version
- 현재 10.1이 최신 버전

Version	특징
10.x	Turing 지원
9.x	Volta 지원

CUDA 개발 환경

그리고 실습 환경

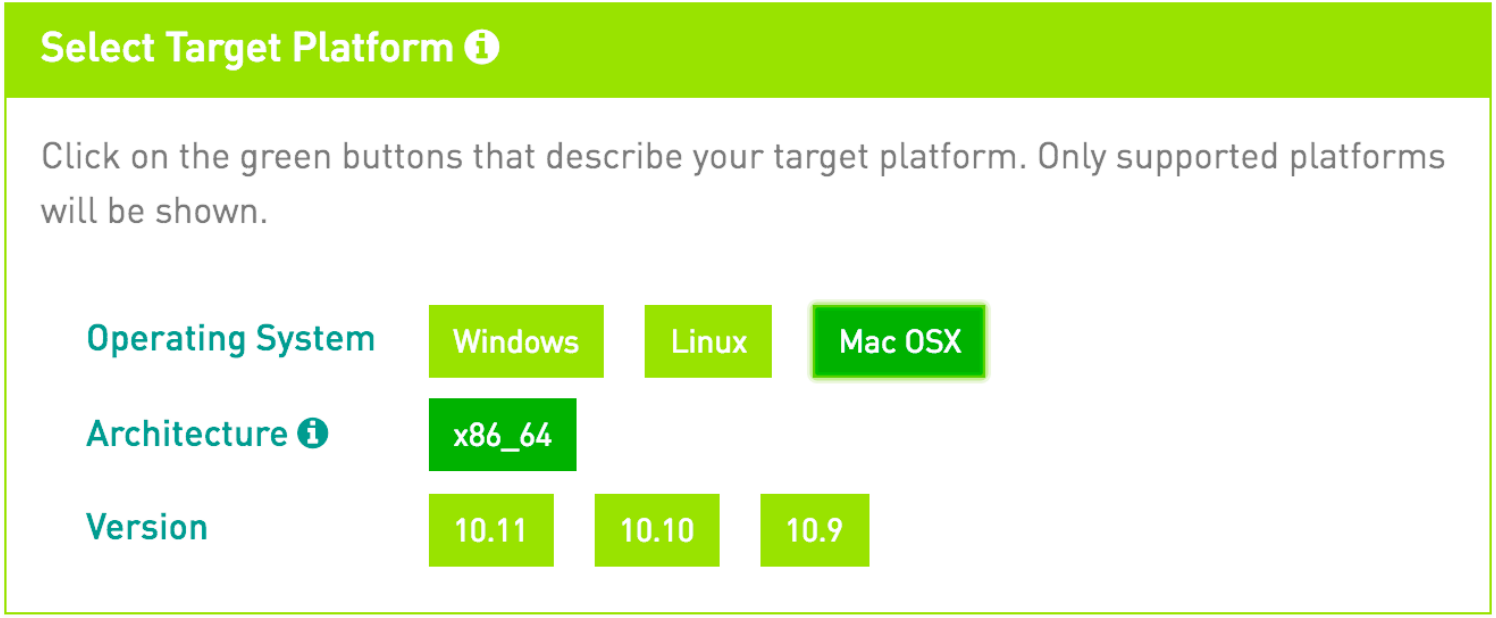
CUDA Toolkit & Driver

- CUDA Toolkit 설치시 Driver와 Toolkit이 함께 설치 가능
  - NVIDIA Driver가 설치된 후, Toolkit만 별도로 설치해도 무방
- Toolkit에는 CUDA Sample Code/ Profiler / Debugger 포함
- 개발 가능 OS

Pascal driver
No debugger

Toolkit Download

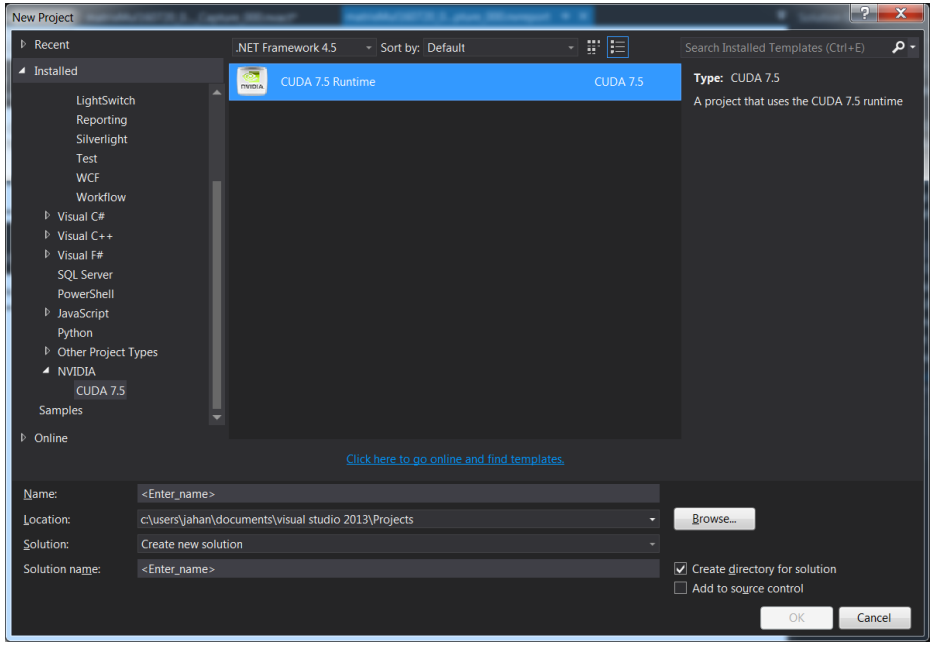
- https://developer.nvidia.com/cuda-downloads
- Windows, Linux, OSX 지원
  - 개발 환경에 맞춰서 설치 후 사용

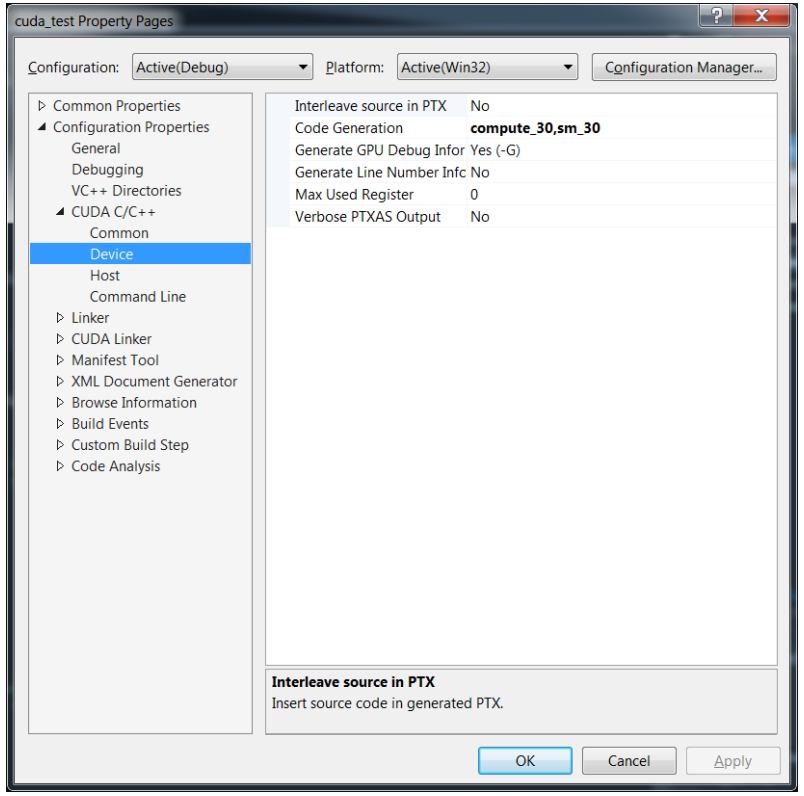


NVIDIA.

Windows 개발환경 구성

- Windows 7 ~ 10
- Visual Studio 2010-2013
- CUDA Toolkit 8.0





Linux 개발환경 구성

- Ubuntu repository 추가

$ sudo apt-key adv --fetch-keys
"http://developer.download.nvidia.com/compute/cuda/repos/ubuntu1804/x86_64/7fa2af80.pub”
$ sudo sh -c 'echo "deb http://developer.download.nvidia.com/compute/cuda/repos/ubuntu1804/x86_64 /" > /etc/apt/sources.list.d/nvidia-cuda.list’
$ sudo sh -c 'echo "deb http://developer.download.nvidia.com/compute/machine-learning/repos/ubuntu1804/x86_64 /" > /etc/apt/sources.list.d/nvidia-machine-learning.list'

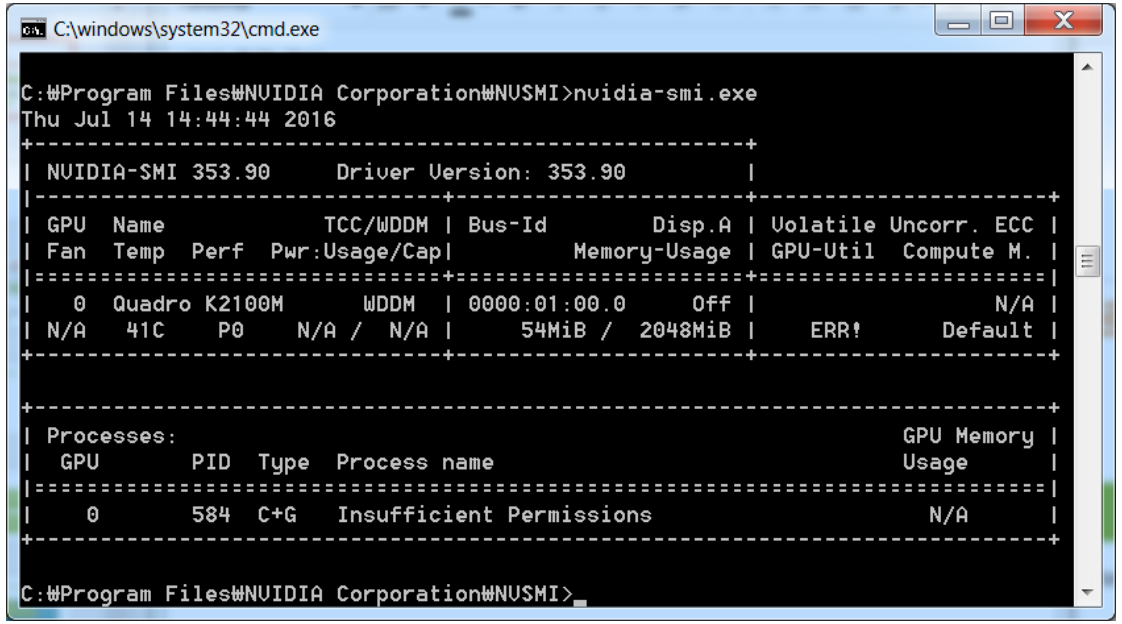
- 드라이버 설치 및 CUDA 설치

$ sudo apt-get install nvidia-driver-418 # desktop
$ sudo apt-get install nvidia-headless-418 nvidia-utils-418 # server

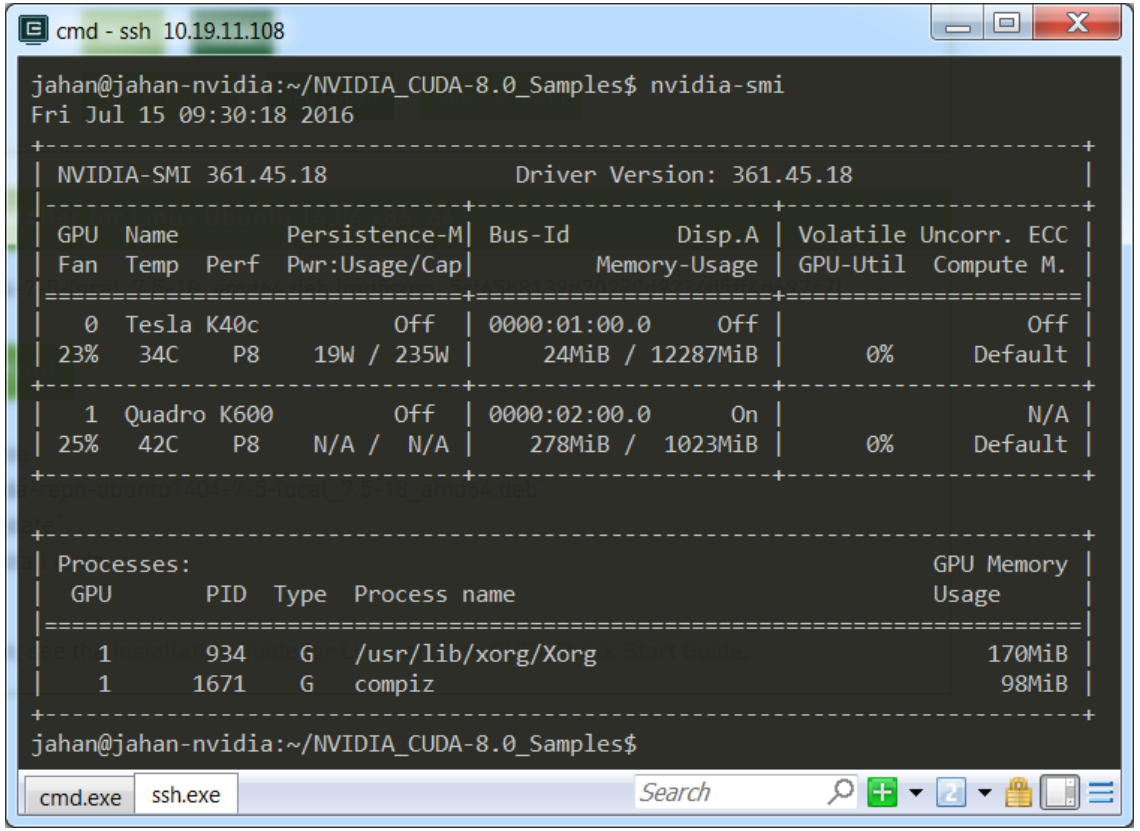
$ sudo apt-get install cuda-toolkit-10-1

NVIDIA.

설치 확인 – nvidia-smi



Windows



Linux

-q: 상세정보 확인
-l #: #s마다 정보갱신
-i #: GPU #의 정보 확인

CUDA C/C++, Fortran으로 할 수 있는 것

- 사칙연산/math.h
- 자료형
  - 기본형 / 벡터형 (float2, int4 등)
- 다른 application과 연동
- 다른 Language와 연동
  - Python, C#, Java, R, etc… (Wrapper 사용)
- OpenGL/DirectX, OpenCV, NPP 등 다양한 라이브러리와 연계
- Deep Learning…?

선행 요구 사항

- C/C++ 프로그래밍 경험 필요
- 그래픽스 경험 없어도 익힐 수 있음
- 병렬 프로그래밍 경험은 익혀야 함

NVIDIA.

CUDA 프로그래밍 기초

Hello CUDA

CUDA 초기화

- CUDA runtime에서 자동으로 초기화
- 하나의 Process에 CUDA Device 별로 CUDA Context가 생성됨
- Reset 명령은 cudaDeviceReset()
  - 호출한 Host Thread가 선택한 Device에 대하여 Reset
- 실무에서는 CUDA카드의 설치 유무, 버전 확인등 수행

NVIDIA.

Device 메모리 할당

- 할당

```
cudaMalloc(**pVoid, size_t size)
```

- 해제

```
cudaFree(*pVoid)
```

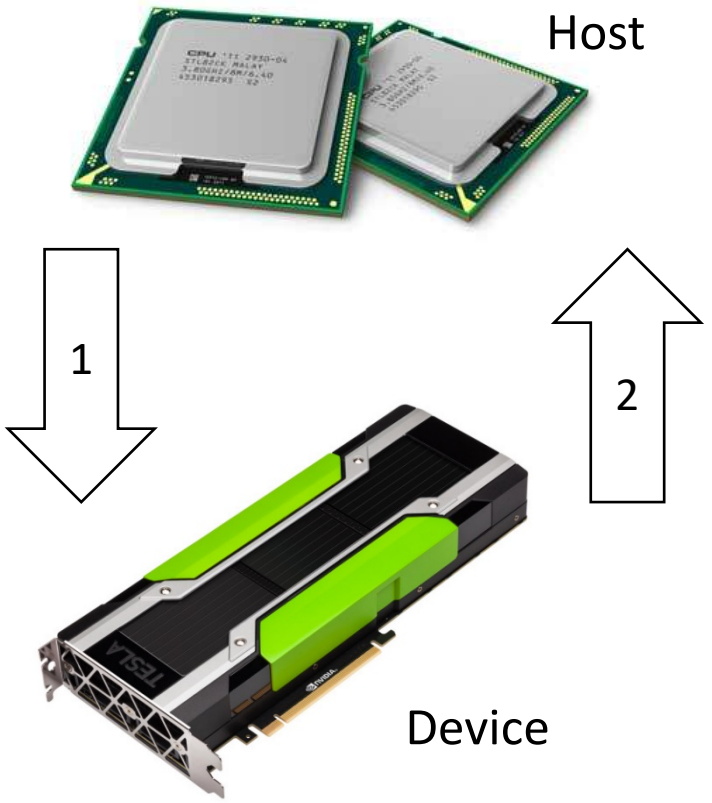

NVIDIA.

CUDA Memcpy

```
cudaMemcpy(void* dst, void* src, size_t length, enum direction)
```

Direction

1. cudaMemcpyHostToDevice
2. cudaMemcpyDeviceToHost
3. cudaMemcpyDeviceToDevice
4. cudaMemcpyHostToHost



Kernel 함수

```
__global__ void saxpy(float* d_y, float* d_x, float a) {
    int idx = blockDim.x * blockIdx.x + threadIdx.x;
    d_y[idx] = a * d_x[idx] + d_y[idx];
}
```

- CPU에서 호출하는 GPU 함수
- __global__ 지시어 지정
- return은 항상 void
- argument는 값 또는 device memory pointer이어야 함

NVIDIA.

Kernel 호출

kernel_call

```
dim3 dimBlock(64);
dim3 dimGrid((vecSize + dimBlock.x – 1) / dimBlock.x);
dMemcpy<<< dimBlock, dimGrid >>>(d_y, d_x);
```

- Kernel의 thread 크기를 계산해서 GPU에 알림
- dim3 자료형 사용
  - 3차원까지 사용 가능
- Grid의 크기 계산시 Ceiling Operation 사용

NVIDIA.

SAXPY Example

C

```
void saxpy(float* py, float* px, float alpha, int n_size) {
    for (int i = 0; i < n_size; i++) {
        py[i] = alpha * px[i] + py[i];
    }
}
```

CUDA

```
__global__
void d_saxpy(float* d_y, float* d_x, float alpha, int n_size) {
    int idx = blockDim.x * blockIdx.x + threadIdx.x;

    d_y[idx] = alpha * d_x[idx] + d_y[idx];
}
```

NVCC에게 CPU/GPU 코드를 알려주는 방법

- 지시어를 이용하여 알려줌

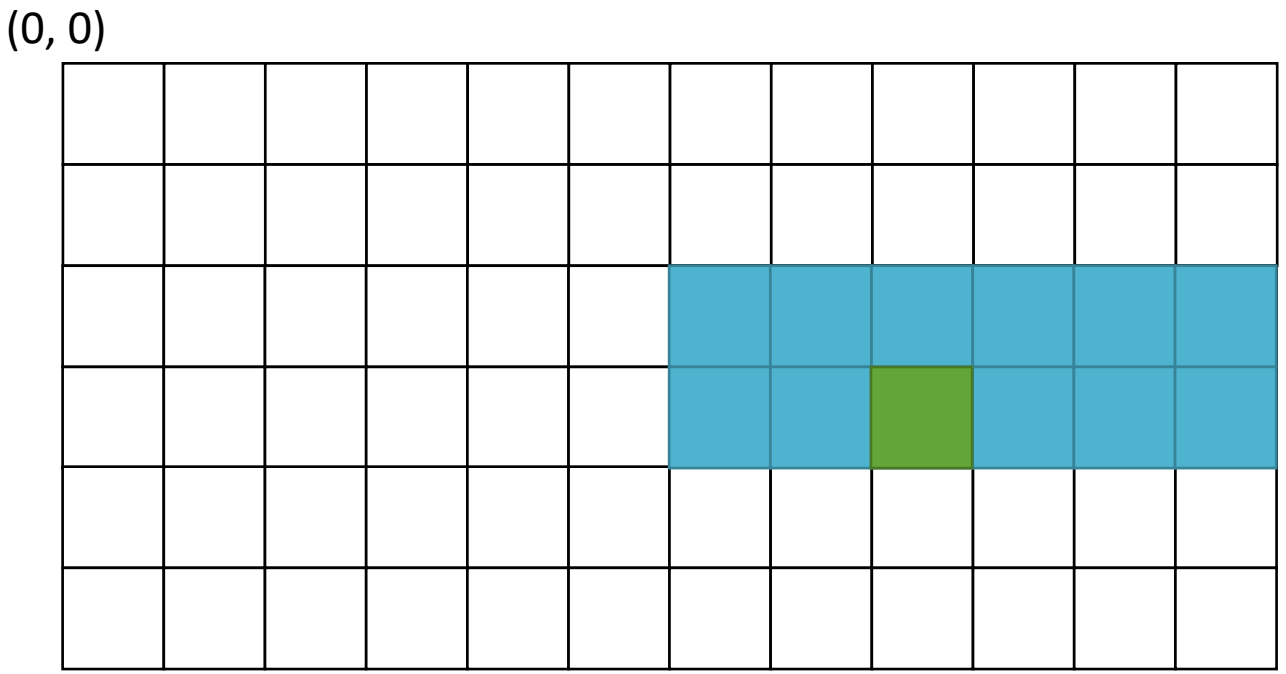
	호출위치	실행위치
`__global__`	host	device
`__host__`	host	host
`__device__`	device	device
`__host__ __device__`	host, device	host, device

NVIDIA.

CUDA Thread/Block Index

```
int idx = blockDim.x * blockIdx.x + threadIdx.x
```

- Index 지시어
  - blockDim (6, 2)
  - gridDim (2, 3)
  - blockIdx (1, 1)
  - threadIdx (2, 1)



NVIDIA.

CUDA Thread 계층구조

- 계층적 병렬처리
  - Grid (Kernel)은 여러개의 block을 실행
  - Block은 여러개의 thread를 실행

- 각 요소의 dimension

gridDim	blockIdx	blockDim	threadIdx
(x, y)	(x, y)	(x, y, z)	(x, y, z)

NVIDIA.

CUDA Index에 대한 쉬운 관점

- dimension 하나마다 for문이 하나씩 제거됨

$O(n^2) \rightarrow O(n)$

  - 1D

```
for (i = 0; i < N; i++) {
    y[i] = a*x[i] + y[i];
}
```

```
int i = blockDim.x * blockIdx.x + threadIdx.x;
C[i] = A[i] + B[i];
```

  - 2D

```
for (j = 0; j < N; j++) {
    for (i = 0; i < N; i++) {
        y[j][i] = a*y[j][i] + y[j][i];
    }
}
```

```
int j = blockDim.y * blockIdx.y + threadIdx.y;
int i = blockDim.x * blockIdx.x + threadIdx.x;
C[j][i] = A[j][i] + B[j][i];
```


NVIDIA.

2D 예제

- (x, y) index 표현

```
__global__ saxpy(float* d_y, float* d_x, float a) {
   int idx_y = blockDim.y * blockIdx.y + threadIdx.y;
   int idx_x = blockDim.x * blockIdx.x + threadIdx.x;

   d_y[idx_y * N + idx_x] = a * d_x[idx_y * N + idx_x] + d_y[idx_y * N + idx_x];
}
```

- 하나의 index로 표현

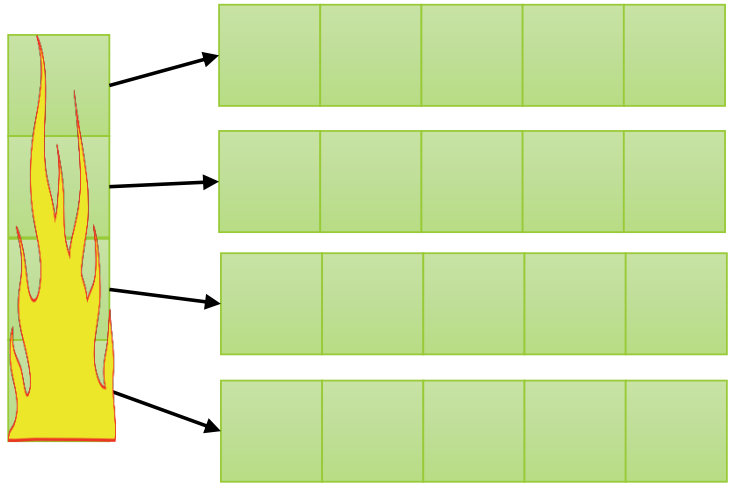
```
__global__ saxpy(float* d_y, float* d_x, float a) {
   int idx = gridDim.x * blockDim.x * blockDim.y * blockIdx.y +
             gridDim.x * blockDim.x * threadIdx.y +
             blockDim.x * blockIdx.x + threadIdx.x;

   d_y[idx] = a * d_x[idx] + d_y[idx];
}
```


NVIDIA.

Device memory 사용시 1차원 배열 권장

- 동적 할당 된 2차원 배열 사용시 GPU 병렬성 저하

A[4][5]



- 관리가 어려움
  - 각각의 array 들과 array of pointer를 일일이 GPU 메모리로 보내야 함
- Struct는 배열이므로 가능
  - Structure내의 pointer 변수가 있다면, pointer가 가르키는 메모리에 대해서도 GPU메모리 할당 필요

NVIDIA.

Device Function

- Kernel에서 호출 가능한 함수

```
__device__ float d_saxpy(float* x, float* y, float a) {
    return a*x + y;
}

__global__void saxpy(float* d_y, float* d_x, float a) {
    int idx = blockDim.x * blockIdx.x + threadIdx.x;
    d_y[idx] = d_saxpy(d_x + idx, d_y + idx, a);
}
```

  - Kernel Code의 재 사용성 향상

NVIDIA.

CUDA 변수, 함수에 대한 Naming 권장

- Device memory 및 Kernel 함수에 대한 코드상 구분이 불가함

- 프로그래밍 시 naming을 통해 실행되는 위치의 명시 권장

```
__device__ float d_saxpy(float* x, float* y, float a) {
    return a*x + y;
}

__global__void saxpy(float* d_y, float* d_x, float a) {
    int idx = blockDim.x * blockIdx.x + threadIdx.x;
    d_y[idx] = d_saxpy(d_x + idx, d_y + idx, a);
}
```

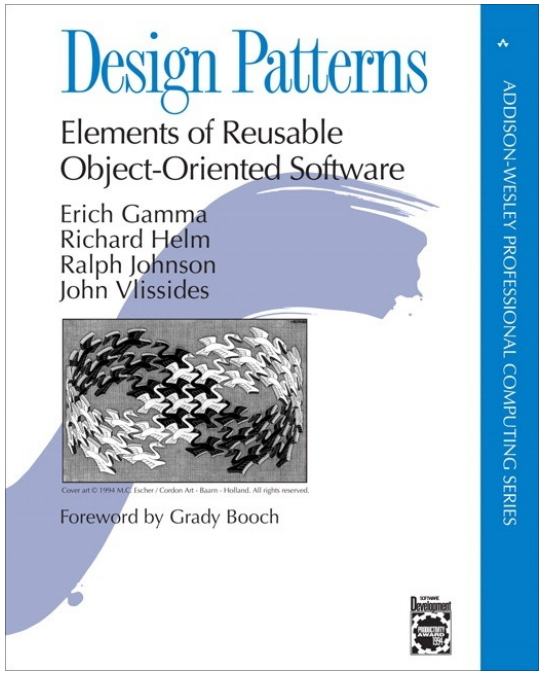

NVIDIA.

CUDA Programming Practice

Parallel Programming Pattern 1

Parallel Communication Pattern

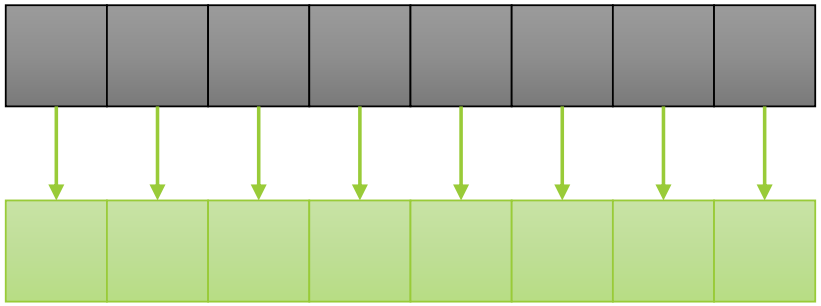
- Map
- Gather
- Stencil
- Scatter
- Transpose
- Reduce
- Histogram
- Scan
- Pack
- Sort

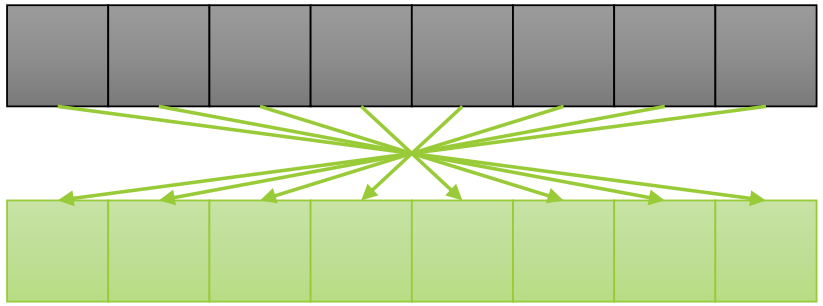


NVIDIA.

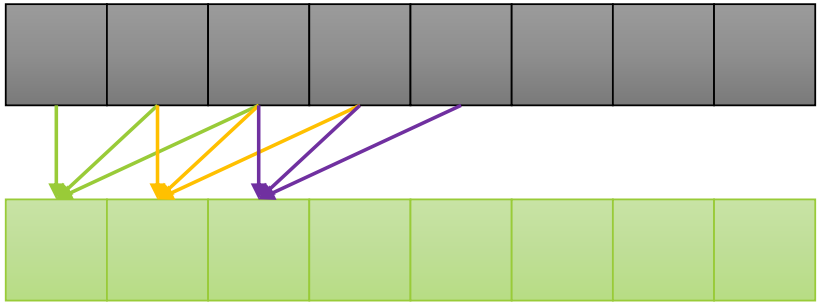
Map & Gather

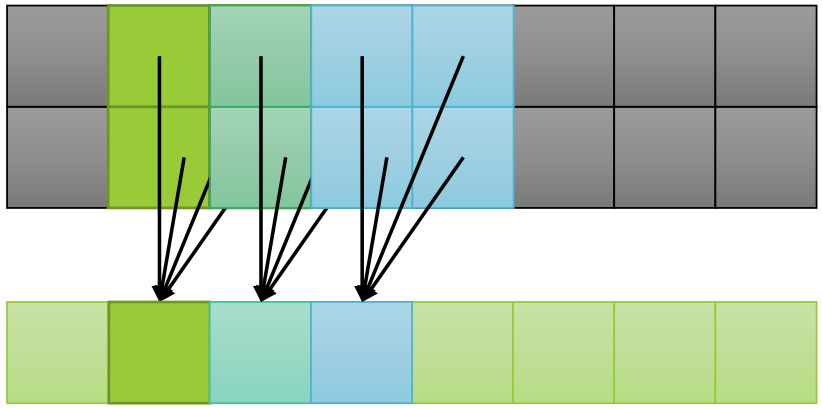
- Map (1:1)





- Gather (n:1)

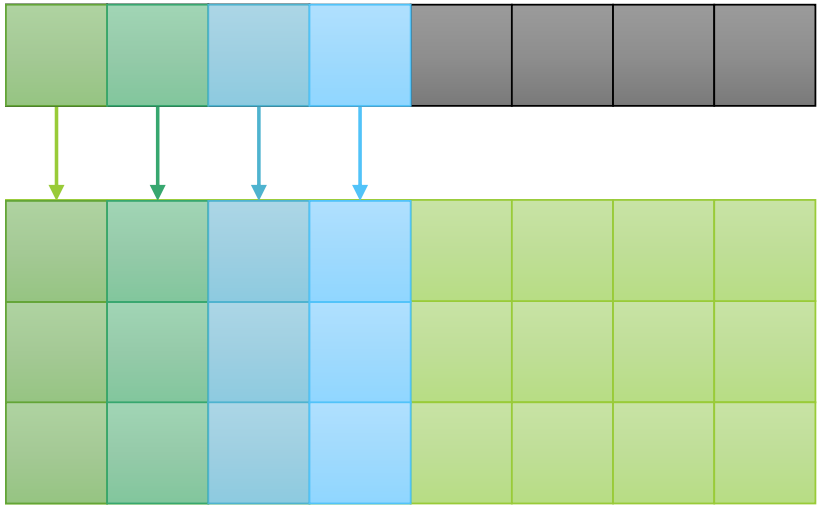




NVIDIA.

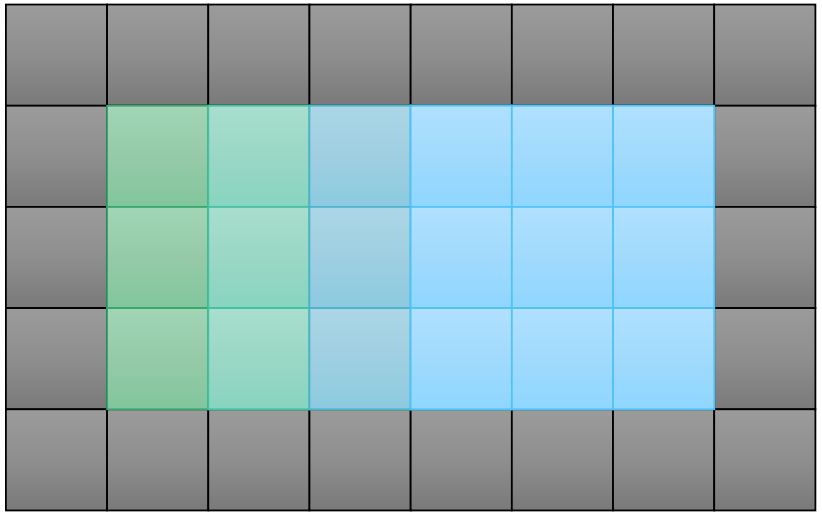
Scatter

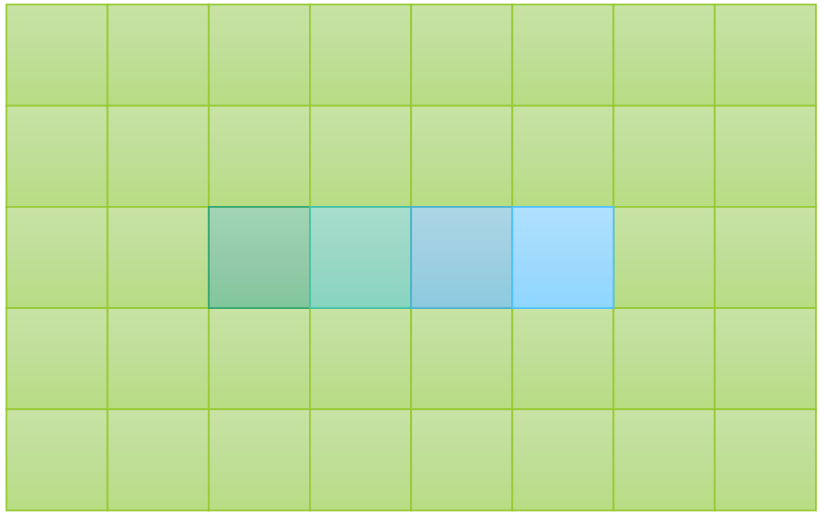
- 1:n



Stencil

- 정해진 수만큼 주변의 데이터를 함께 참고하는 동작



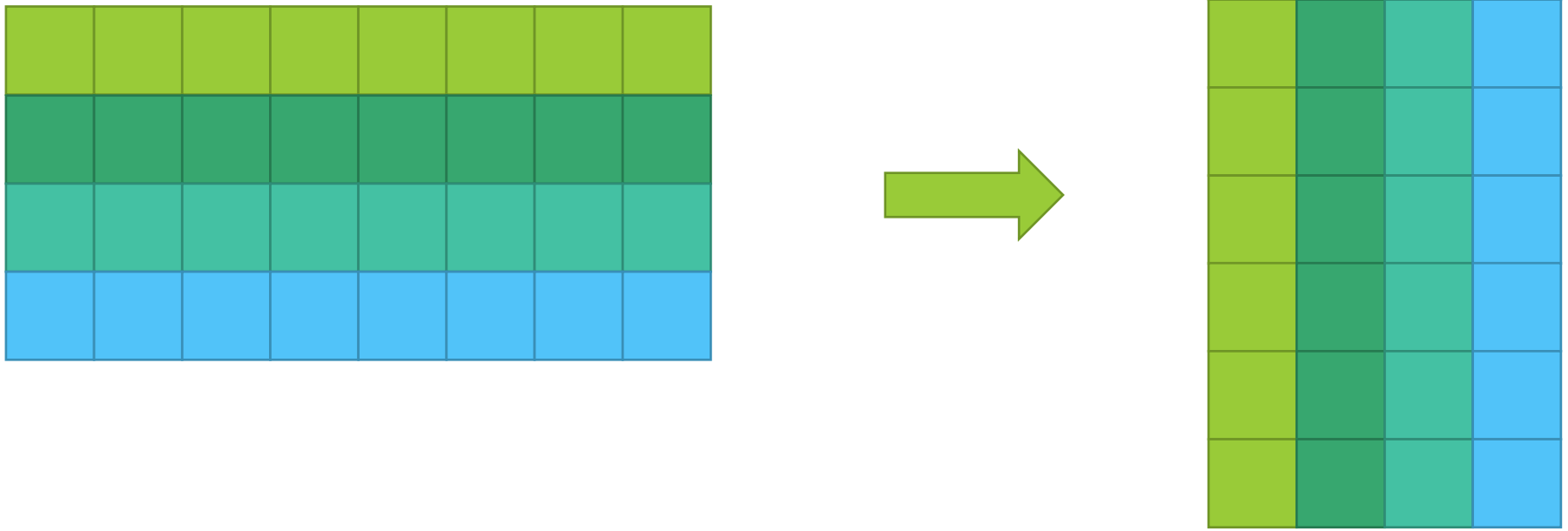


NVIDIA.

Transpose

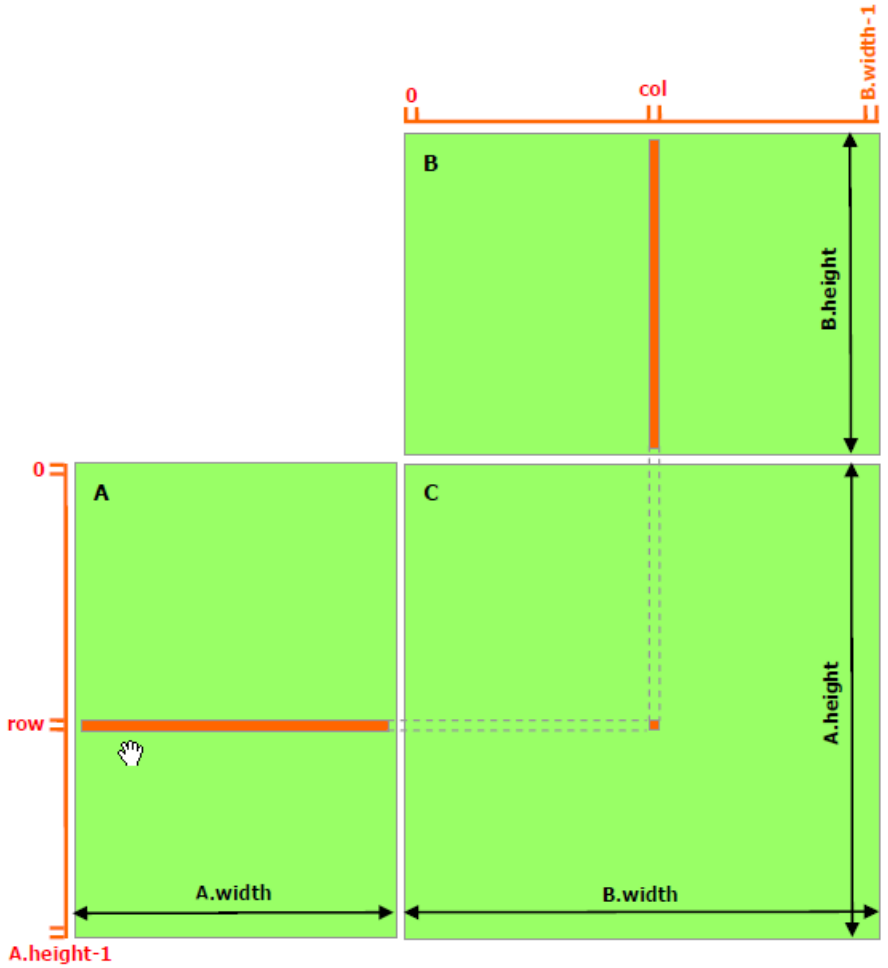
- Buffer 데이터의 Transpose
  - 이미지, 행렬 등의 연산에 활용

$$A \rightarrow A^T$$



Practice Code

- Matrix Multiplication을 해봅시다.
  - Map
  - Transpose
  - Gather



NVIDIA.

Parallel Communication Pattern

- **Selection**
- **Map**
- **Gather**
- **Stencil**
- **Scatter**
- **Transpose**
- Histogram
- Reduce
- Scan
- Pack
- Sort

NVIDIA.

CUDA Debugging & Profiling

cudaError_t

- CUDA 동작에서 발생하는 에러가 정의된 변수
  - 모든 CUDA Runtime API는 cudaError_t를 return
  - Kernel Call은 내부 전역 변수로 CUDA error 상태를 저장

- CUDA Kernel Call의 에러 확인

```
cudaError_t cudaGetLastError();
```

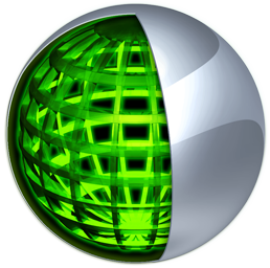
  - 가장 최근에 호출된 Kernel Call의 결과 확인 가능

- Toolkit sample code에서 에러메시지를 string으로 변환해서 볼 수 있음

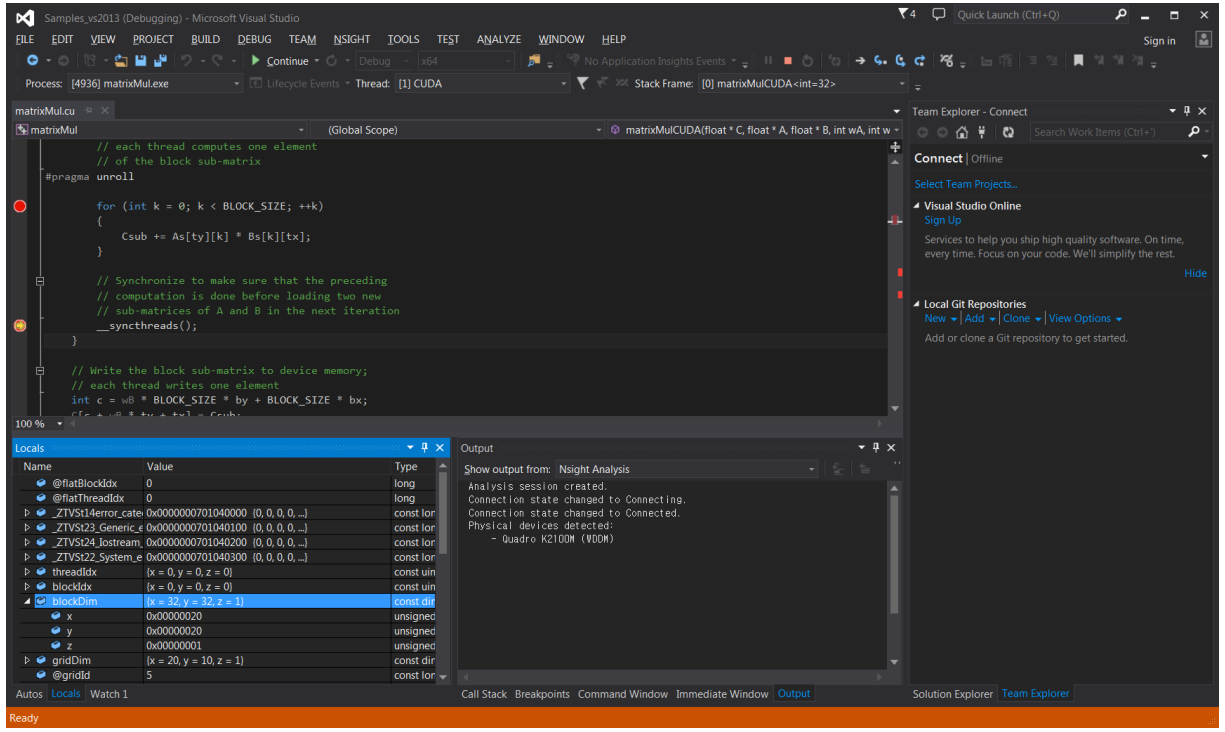
```
#include “$(SAMPLE)/inc/helper_cuda.h”
…
printf(“%s\n”, _cudaGetErrorEnum(cudaError_t error);
```


NVIDIA.

Parallel nSight

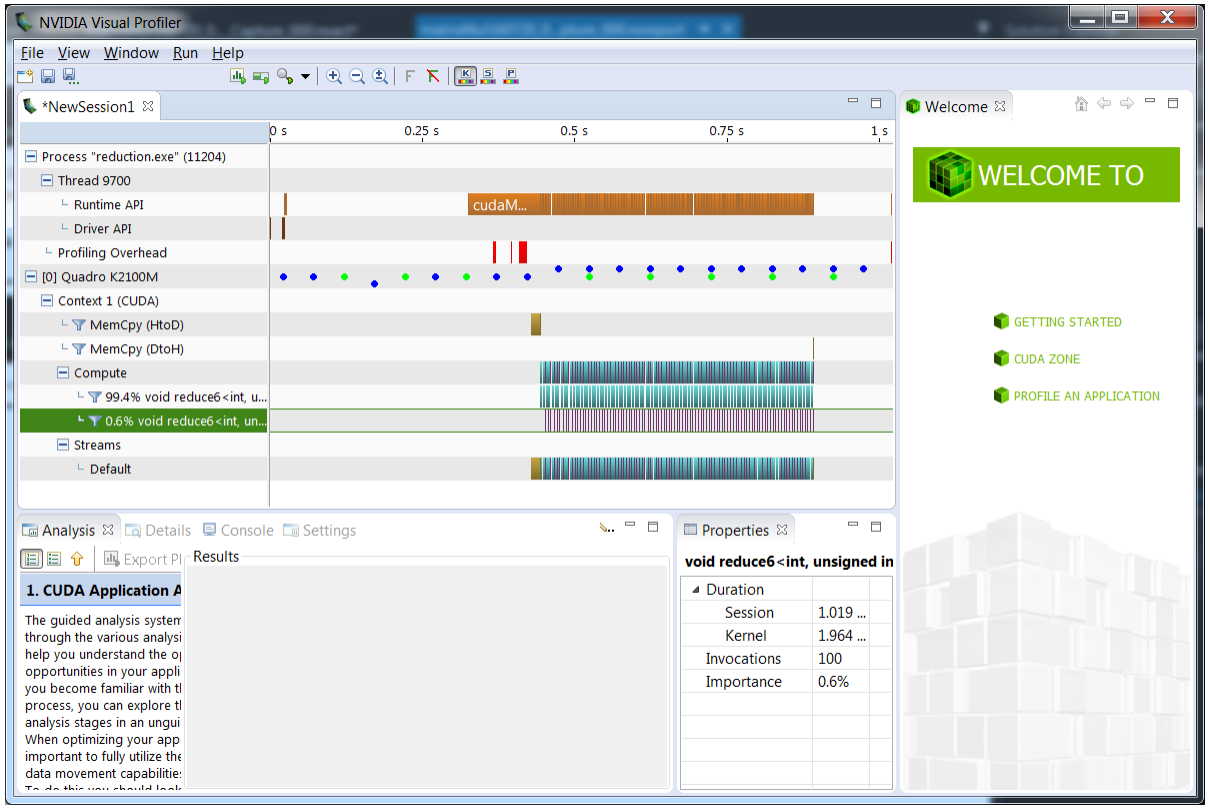


- CUDA 프로그래밍 디버거
  - Windows
    - Visual Studio와 연동해서 동작
  - Linux / Mac
    - Eclipse Edition
- Toolkit과 함께 설치 (무료)
- 모든 CUDA Thread의 모든 연산에 대하여 확인 가능



Visual Profiler

- Windows는 별도, Linux/Mac에서는 nSight와 함께 볼 수 있음



nvprof & nvvp

- nvprof
  - Benchmark GPU program
  - 옵션
    - -o: Benchmark 정보를 file로 dump
    - --print-gpu-trace: GPU Kernel 별로 호출되는 정보를 terminal로 보여줌

- nvvp: NVIDIA Visual Profiler
  - nvprof을 통해서 생성한 log를 UI를 통한 분석 제공

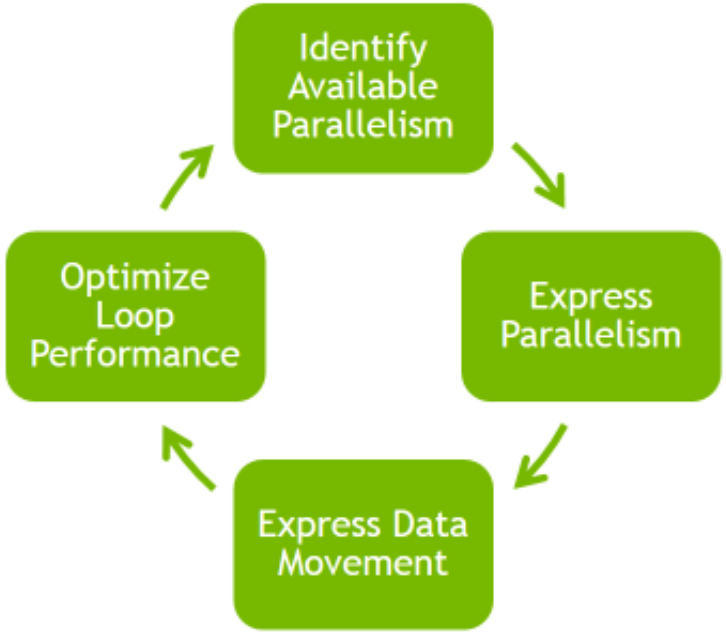
CUDA 개발시 주의할 점

- cu, cuh 파일
  - nvcc에게 CUDA 코드임을 알려주는 키워드
    - CUDA Keyword(blockDim, threadIdx, __global__ 등) 인지 가능
    - <<<…>>> 같은 키워드는 host compiler는 인지 불가
  - 최소한 Kernel코드 및 호출 코드는 cu 파일 내에 작성 해야 함

- 원격으로 개발할 경우,
  - nSight에서 원격 디버깅 지원
  - 원격 모니터로 개발 가능, 단 연산 전용 카드가 필요함
  - 화면 출력도 함께 하는 카드로 개발하는 경우,
    VNC와 같이 화면을 캡쳐→압축→전송→디코딩 해주는 앱 사용할 것

Application의 CUDA 가속 절차

- 성능 분석
- 병목 탐색
- 알고리즘 분석 및 병렬화 전략 수립
- **데이터 의존성 해결**
- **병렬화**
- 최적화



NVIDIA.

Whats on Next

- CUDA Optimization
  - Host level / GPU level / On-chip level optimization
- GPU Technologies
  - Multi-GPU
  - DirectGPU
- Advanced CUDA Programming
  - Parallel Communication Pattern – part 2
- CUDA libraries

Directive Parallel Programming

OpenMP

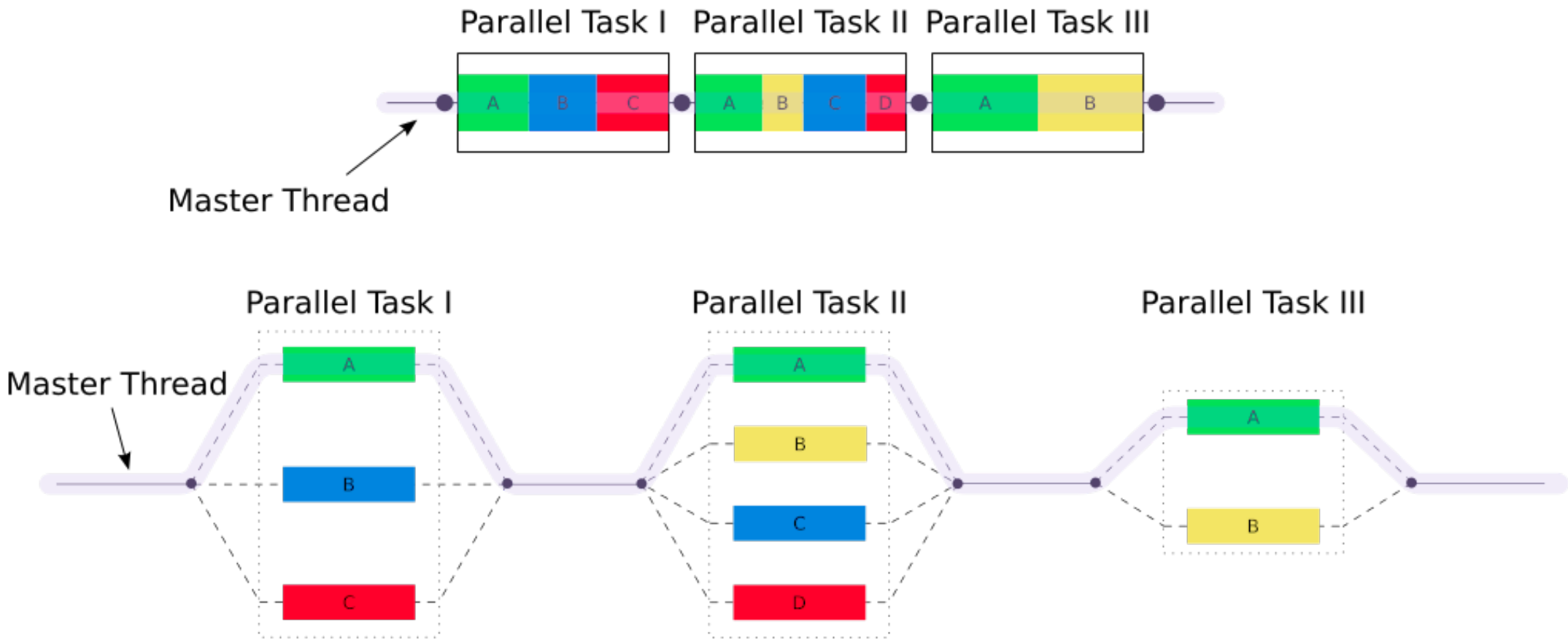
- 공유메모리 병렬 프로그래밍
  - CPU에서 지원
    - 컴파일러 지원 사항 확인
    - 대부분의 컴파일러에서 지원
  - CPU의 코어를 자동으로 병렬화
  - pragma omp 라인을 추가하여 병렬프로그래밍 가능

NVIDIA.

OpenACC

- Vector Processing을 지원하는 모든 코어에 동작
  - GPU, CPU, ARM, OpenPower, etc
- 동일한 코드로 compile 옵션 변경을 통해 target 변경
- Compiler 및 Profiler 무료 사용가능
  - PGI Compiler Community Edition
    http://www.pgroup.com/products/community.htm

Fork-join Model



https://en.wikipedia.org/wiki/Fork%E2%80%93join_model

SAXPY with OpenMP

OpenMP

```
void saxpy(float* y, float* x, float a, int n) {
    #pragma omp parallel shared (n,a,x,y) private (i)
    #pragma omp for
    for (int i = 0; i < n; ++i) {
        y[i] = a*x[i] + y[i];
    }
}
```

CUDA

```
__global__ void saxpy(float* y, float* x, float a)
    int i = blockDim.x * blockIdx.x + threadIdx.x;

    y[i] = a*x[i] + y[i];
}
```

OpenACC

- Directive programming for CUDA

```
void saxpy_parallel(int n, float a, float *x, float *restrict y) {
    #pragma acc kernels
    for (int i = 0; i < n; ++i)
        y[i] = a*x[i] + y[i];
}
```


NVIDIA.

CUDA Optimization

Overview

On-Chip level

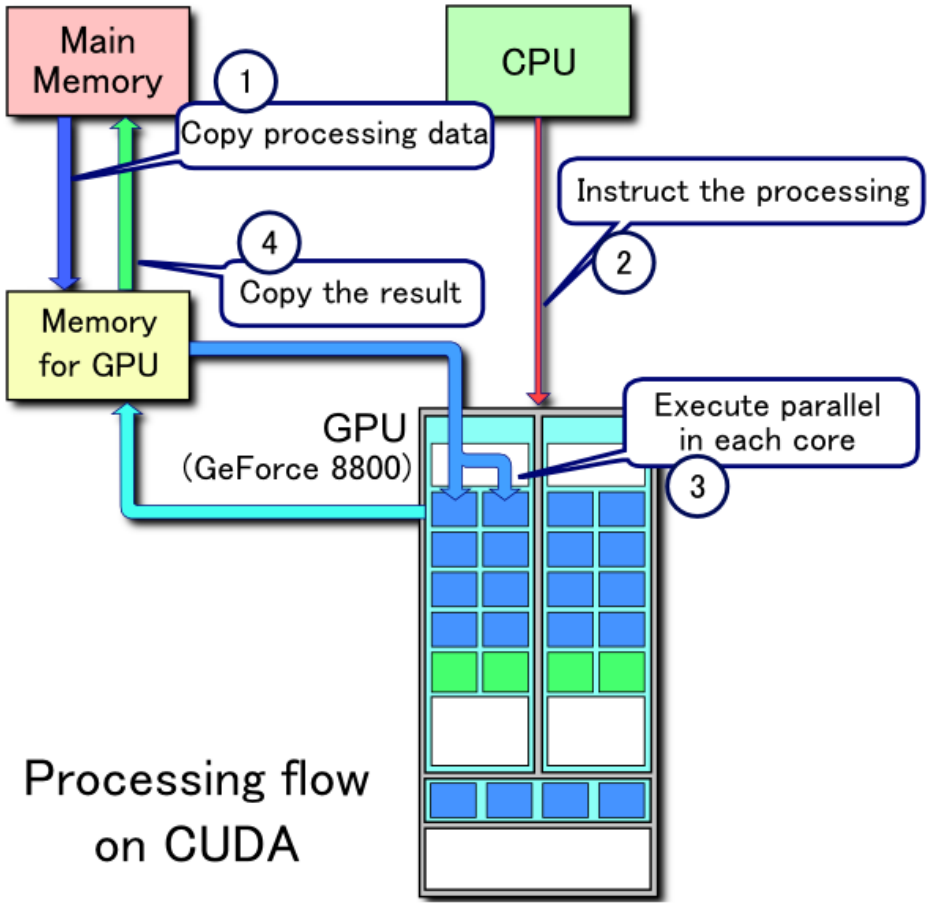
CUDA 최적화 전략

적용 대상	병렬처리	대역폭 최적화	동작 최적화

On-chip	• CUDA Occupancy 최적화
	  ○ Occupancy Calculator
	  ○ Block 크기 조정
	  ○ Register / Thread 조정
		• GPU 메모리 최적화
		  ○ Shared Memory
		  ○ Constant/Texture
		• CUDA Array
			• Warp divergence avoiding
			• Intrinsic function

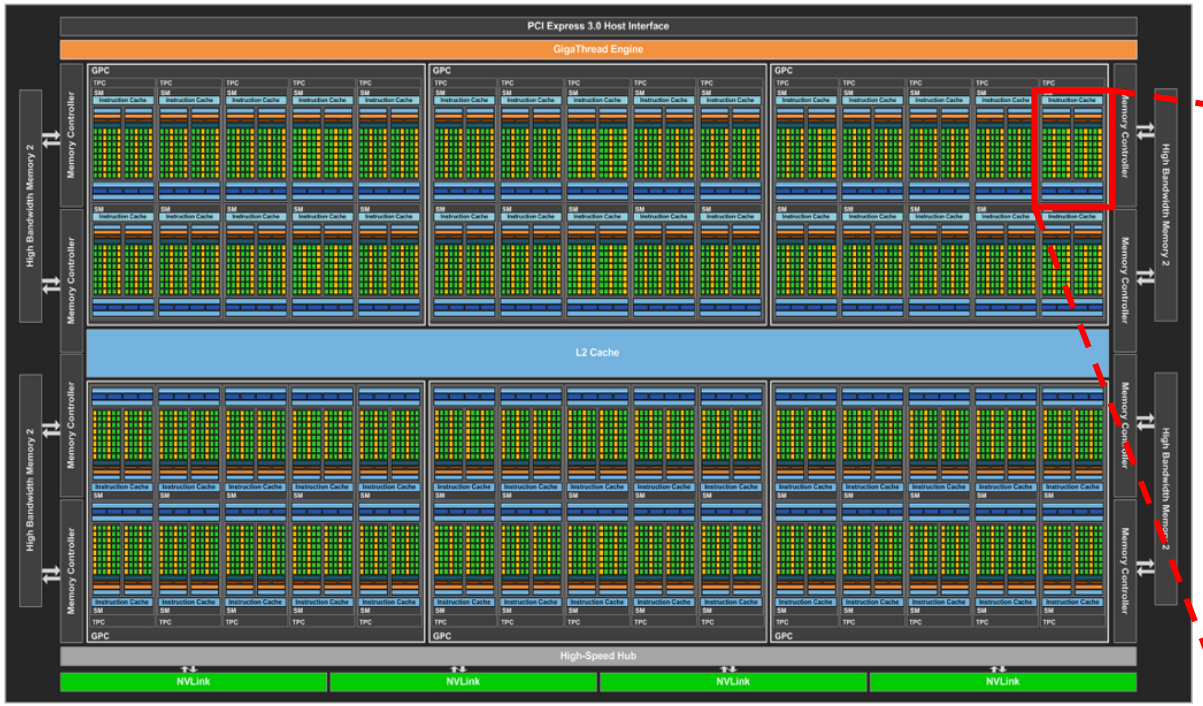
System	• Asynchronous Operation • CUDA Stream • GPUDirect	• Pinned Memory • Access Coalescing • Pitch Memory	

CUDA 실행 절차


Main Memory
CPU
1
Copy processing data
Instruct the processing
2
4
Copy the result
Memory for GPU
GPU (GeForce 8800)
Execute parallel in each core
3
Processing flow on CUDA

NVIDA Pascal Architecture

	Pascal SM
CUDA Cores SP	64
CUDA Cores DP	32
Register File	256 KB
Shared Memory	64 KB
Active Threads	2048
Active Blocks	32

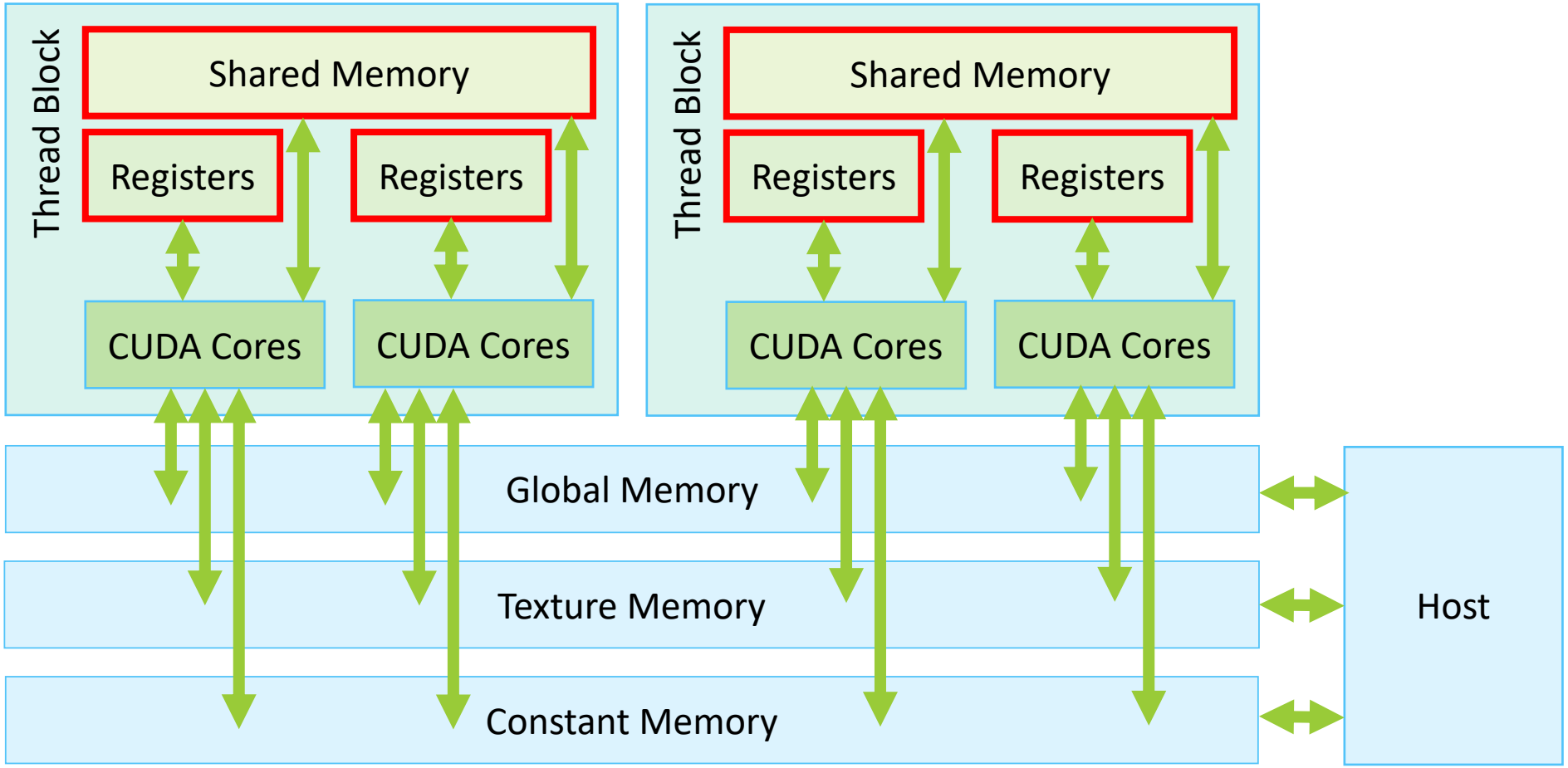


Tesla P100: 60 SMs
GTX 1080: 40 SMs
GTX 1070: 30 SMs

SM: Stream Multiprocessor

NVIDIA

GPU On-Chip 메모리



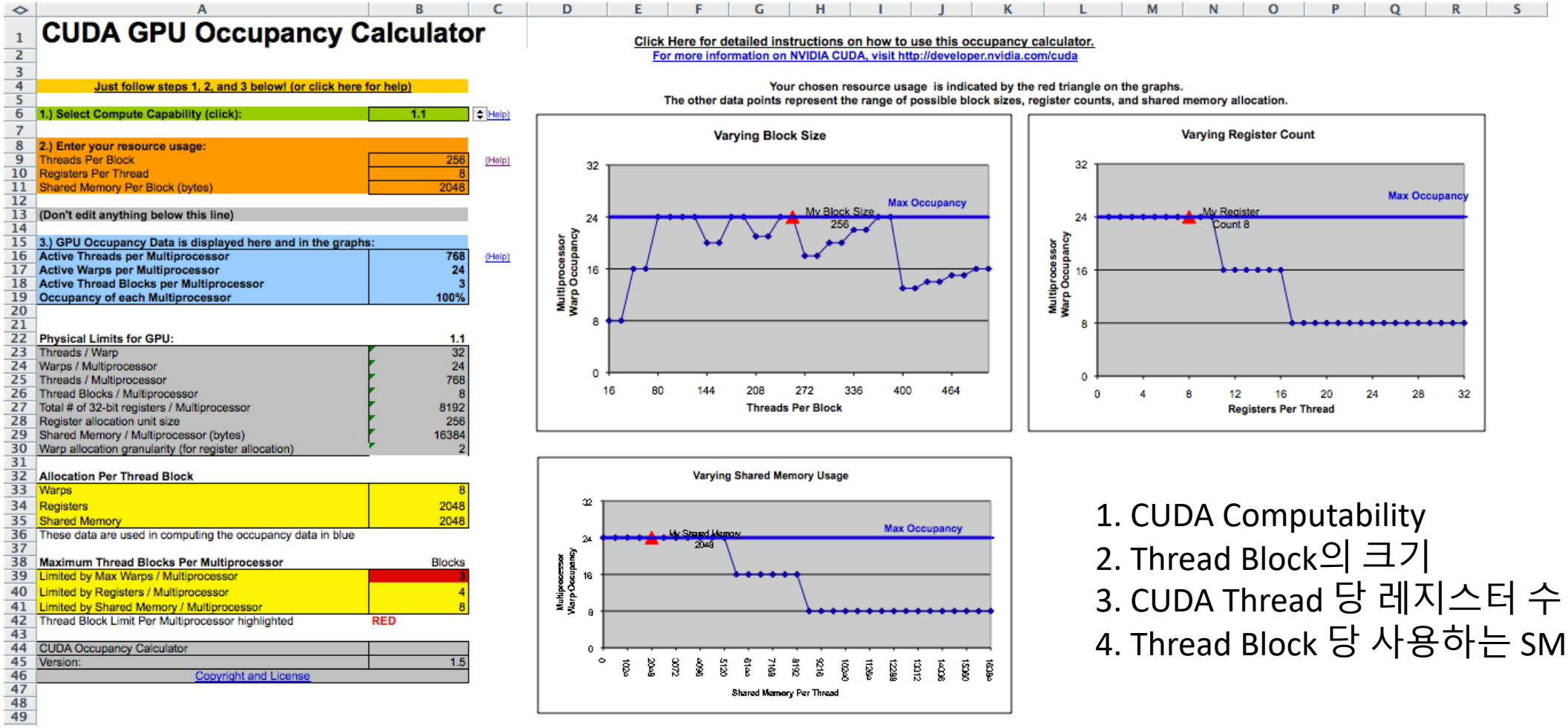
NVIDIA.

CUDA Occupancy

- CUDA Core에 대한 SM의 자원 비율
  - Register
  - Shared Memory
  - Active Threads
  - Active Blocks
- CUDA Thread block은 SM의 메모리 자원을 나눠서 실행함
- CUDA SM의 자원 배분의 효율성에 대한 이론적 계산

NVIDIA.

Occupancy Calculator



CUDA GPU Occupancy Calculator

Click Here for detailed instructions on how to use this occupancy calculator.
For more information on NVIDIA CUDA, visit http://developer.nvidia.com/cuda

Just follow steps 1, 2, and 3 below! (or click here for help)

1.) Select Compute Capability (click):	1.1	(Help)
2.) Enter your resource usage:		
Threads Per Block	256	(Help)
Registers Per Thread	8	
Shared Memory Per Block (bytes)	2048	
(Don't edit anything below this line)		
3.) GPU Occupancy Data is displayed here and in the graphs:		
Active Threads per Multiprocessor	768	(Help)
Active Warps per Multiprocessor	24	
Active Thread Blocks per Multiprocessor	3	
Occupancy of each Multiprocessor	100%	

Physical Limits for GPU:	1.1
Threads / Warp	32
Warps / Multiprocessor	24
Threads / Multiprocessor	768
Thread Blocks / Multiprocessor	8
Total # of 32-bit registers / Multiprocessor	8192
Register allocation unit size	256
Shared Memory / Multiprocessor (bytes)	16384
Warp allocation granularity (for register allocation)	2

Allocation Per Thread Block	
Warps	8
Registers	2048
Shared Memory	2048

These data are used in computing the occupancy data in blue

Maximum Thread Blocks Per Multiprocessor	Blocks
Limited by Max Warps / Multiprocessor	3
Limited by Registers / Multiprocessor	4
Limited by Shared Memory / Multiprocessor	8
Thread Block Limit Per Multiprocessor highlighted	RED

CUDA Occupancy Calculator	
Version:	1.5
Copyright and License	

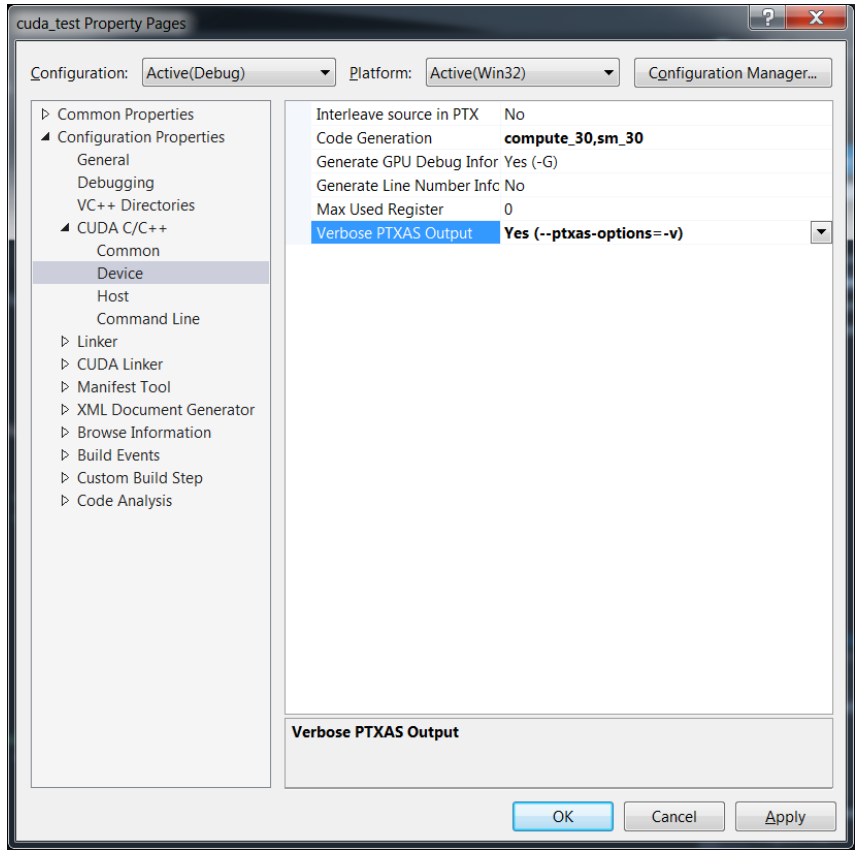
Your chosen resource usage is indicated by the red triangle on the graphs.
The other data points represent the range of possible block sizes, register counts, and shared memory allocation.

1. CUDA Computability
2. Thread Block의 크기
3. CUDA Thread 당 레지스터 수
4. Thread Block 당 사용하는 SM 크기

NVIDIA.

PTX output

- Kernel 별로 사용하는 CUDA 자원의 크기 알림

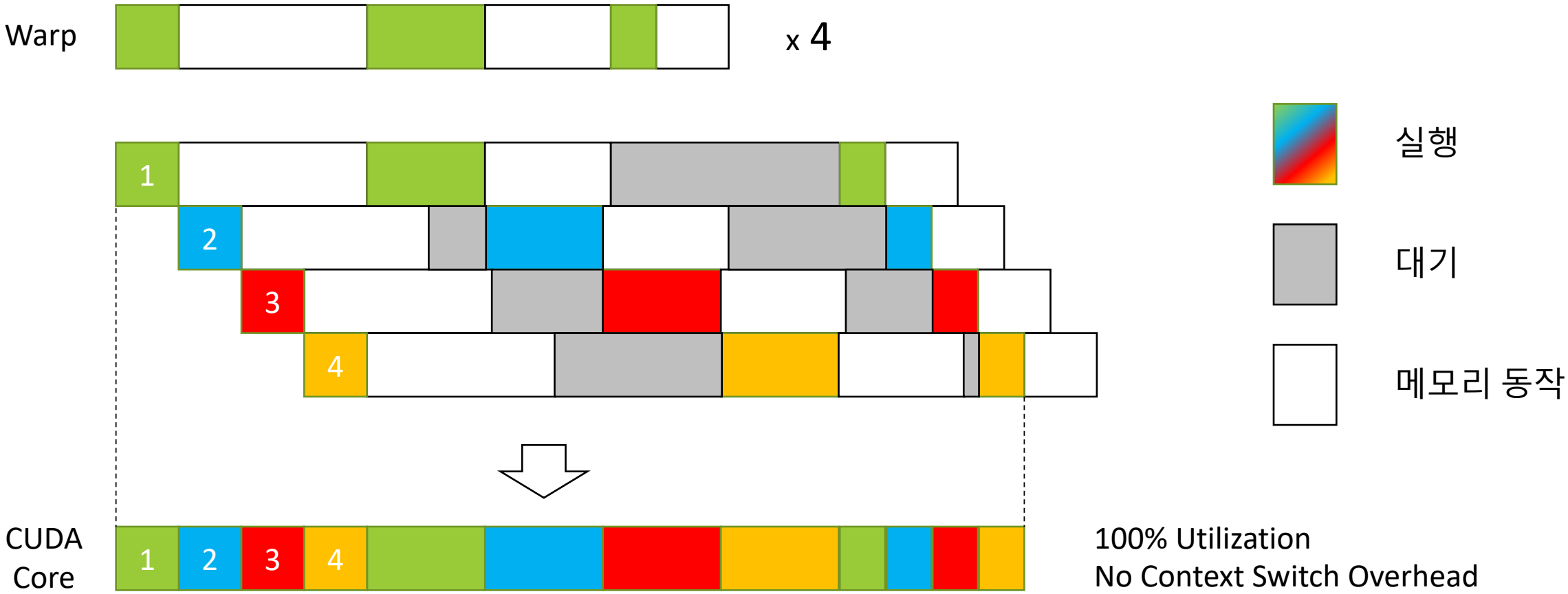


CUDA SM & Thread Block

- CUDA Core(SP)는 64개인데…
  - Active Thread는 최대 2048 / Active Block은 최대 32개 지원
- Thread Block이 CUDA SM의 자원을 나눠 쓰는 구조
- 데이터 및 Thread를 병렬화 한 뒤, CUDA Core가 순차적으로 실행

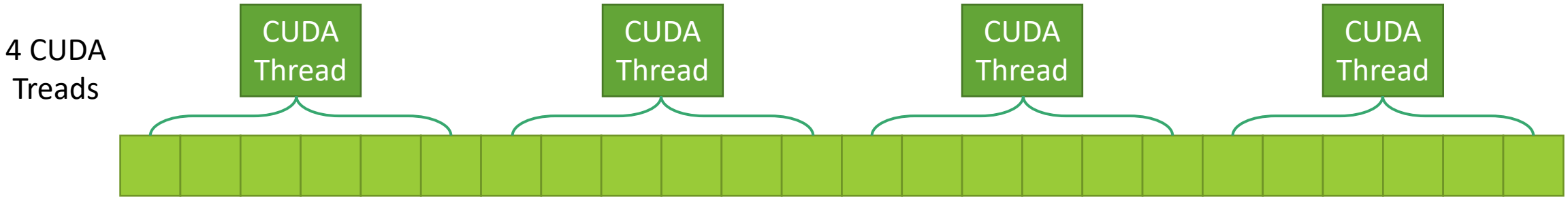
NVIDIA.

Warp Scheduling (example)



Register File

- CUDA는 SM당 256KB의 Register를 갖고 있음
- CUDA Thread가 필요한 만큼 Register를 나눠가짐
  - 필요한 양은 Compile 시점에 결정



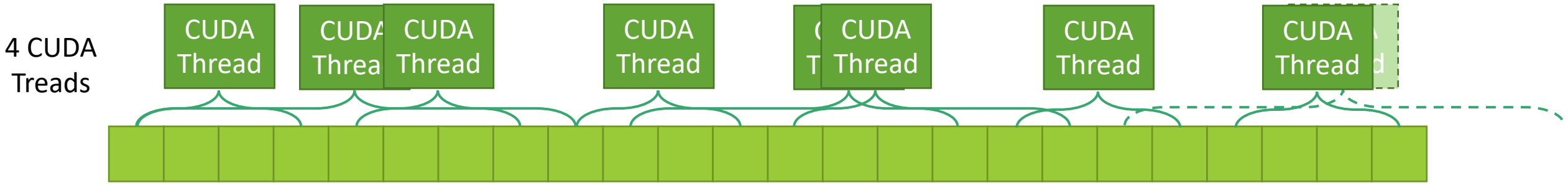
- 여러개의 CUDA Thread가 동시에 동작 가능
- CUDA Core의 효율적인 활용 가능

NVIDIA.

Registers per Thread

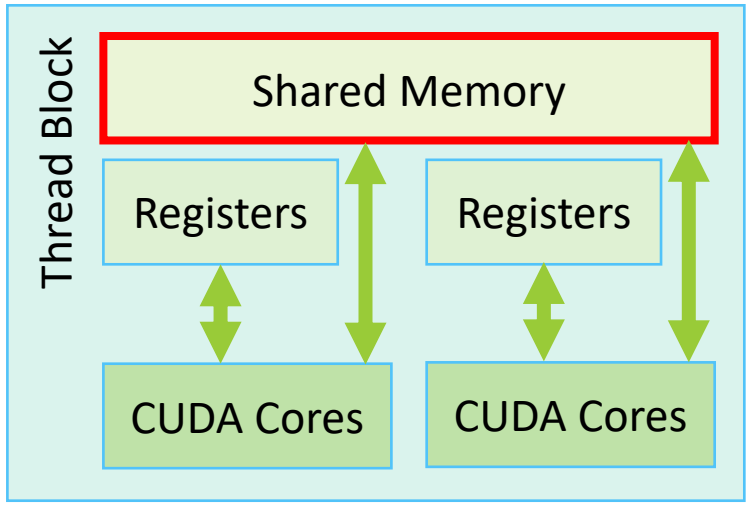
- CUDA Thread에서 사용하는 Register의 수가 많으면, 동시에 실행가능한 CUDA Thread의 수가 제한됨

- 반대로 사용량이 작으면, 더 많은 병렬화가 가능



CUDA Shared memory

- 데이터를 재활용하기 위한 방법
  - 공유 Cache 메모리 활용
- Thread Block 내 Thread간 데이터 공유
- 개발자가 필요에 따라 반복 사용 빈도가 높은 데이터를 보관하는데 사용



NVIDIA.

__shared__ 지시어

- 공유메모리임을 알려주는 지시어를 통해 사용

- 정적 선언

```
__global__ foo(float* B, float *A) {
    __shared__ float s_A[n];
    { … }
    s_A[idx] = A[idx];
    { … }
}
```

  - 사용하는 공유메모리크기 명시
  - Kernel 내에서만 활용

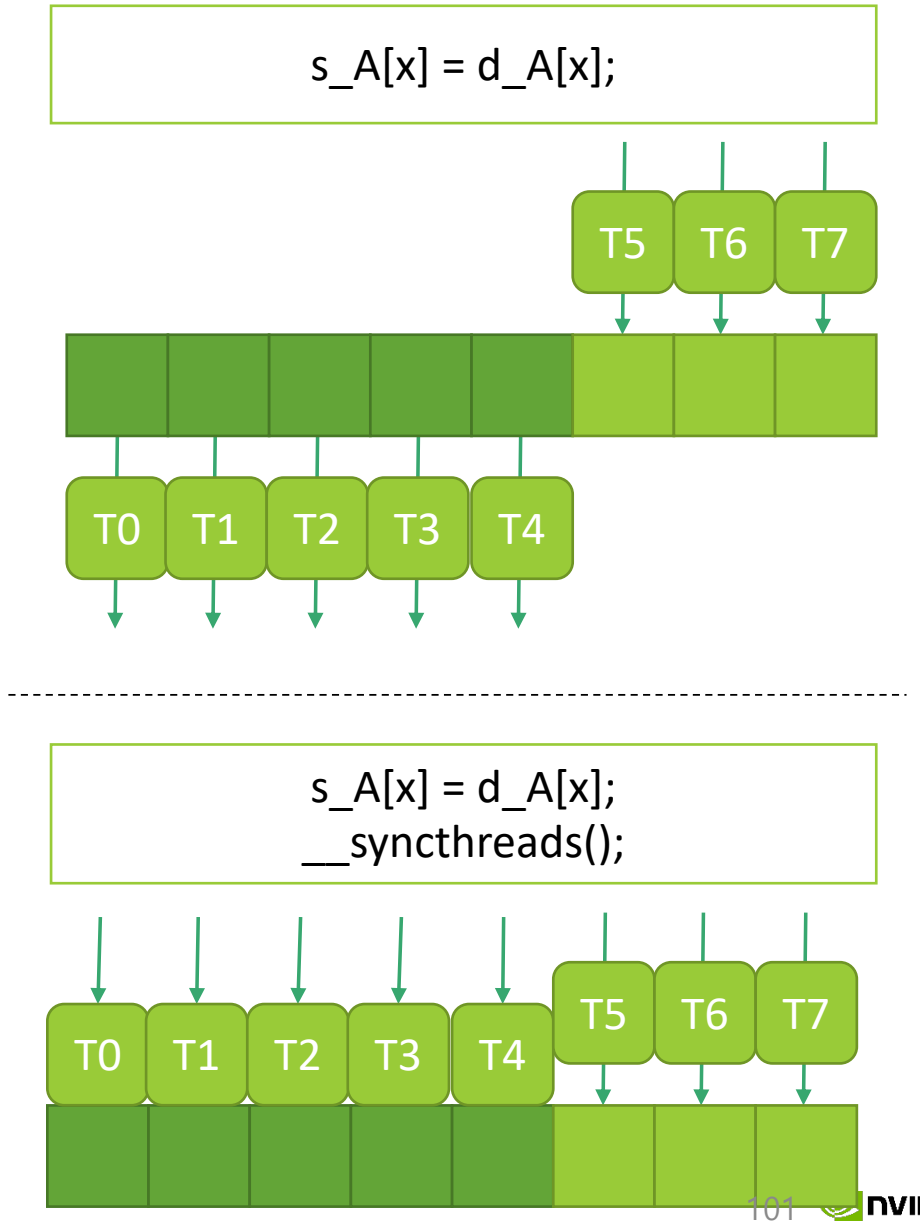
- 동적 선언

```
__shared__ float *s_A;
__global__ foo(float* B, float *A)
{
    { … }
    s_A[idx] = A[idx];
    { … }
}
```

  - 필요한 만큼 공유메모리 사용
  - Kernel간 공유메모리 이름 공유

__syncthreads()

- Thread block의 동기화 (Block barrier)
- Shared memory에 필요한 데이터가 모두 복사되었음을 보장하는데 필요

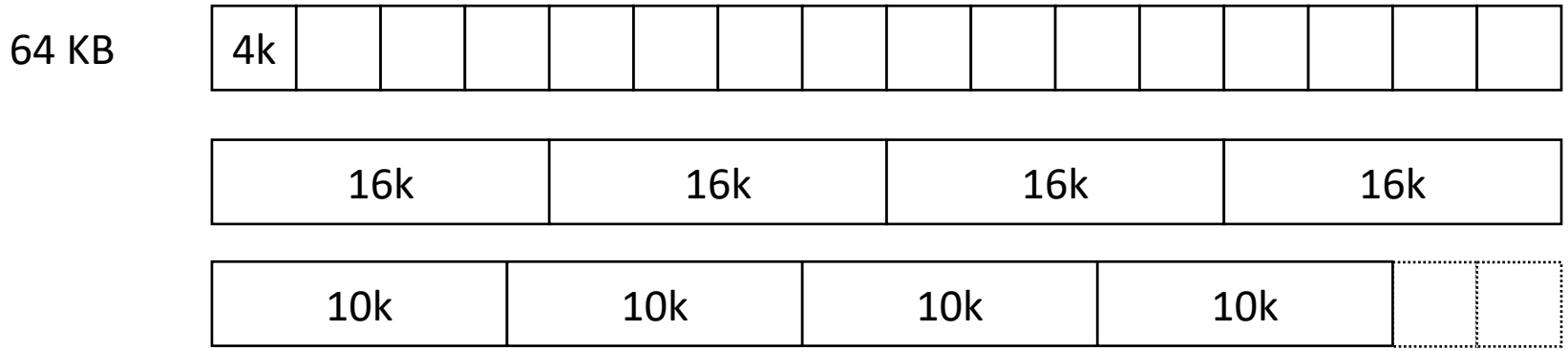


NVIDIA.

Shared memory per block & SM

- SM의 shared memory 크기는 64 KB

- Shared memory는 thread block 내에서 공유 가능한 메모리
  - SM단위로 공유 불가
    - 어떤 index의 thread block이 같은 SM에 있을지 예측 불가
  - Thread block의 shared memory는 SM의 shared memory를 나눠 씀
    - SM에 동시에 실행 가능한 block 개수에 제한이 발생



NVIDIA.

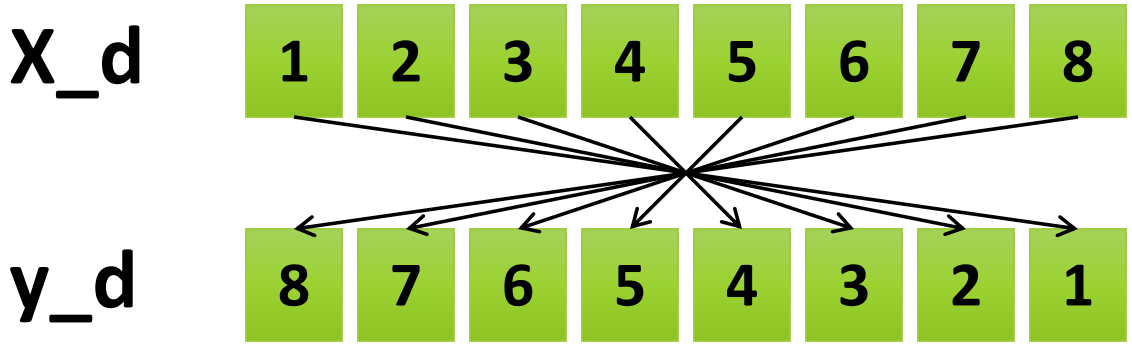
CUDA의 병렬 처리

- 동시에 수행한다는 것의 의미
  - Kernel 호출과 함께 여러개의 CUDA Threads가 각자 할당된 메모리에서 데이터 처리
- CPU와 비교
  - CPU에서 Multi-Thread와 같이 상주의 개념과 다름
  - 실제 CPU도 OS Scheduler에 의해서 Time Sharing을 함
- GPU의 메모리의 데이터를 CUDA Core가 준비되는대로 임의로 처리함
  - 의존성을 낮춘 데이터
  - GPU에 최적화된 병렬화 전략

CUDA Practice 1

Reverse

Reverse buffer

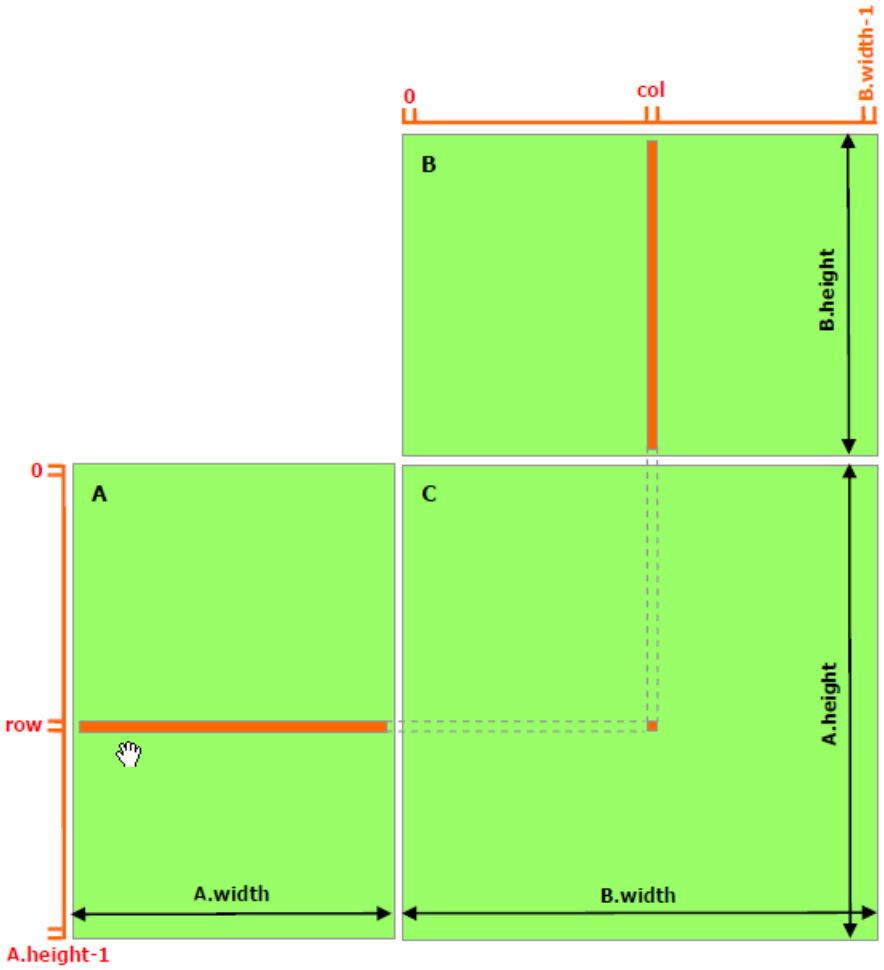


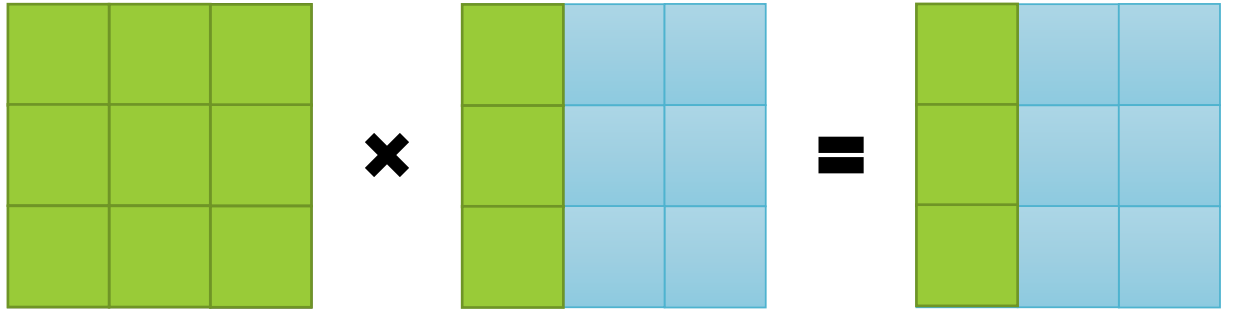
NVIDIA.

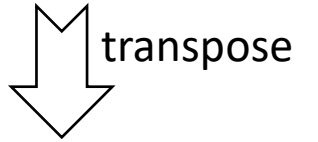
CUDA Practice 2

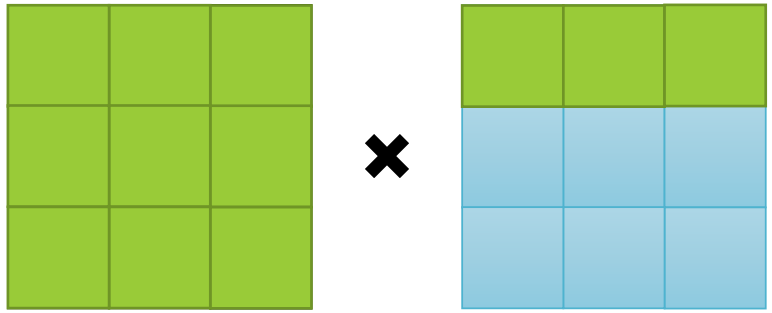
SGEMM with shared memory

Matrix Multiplication

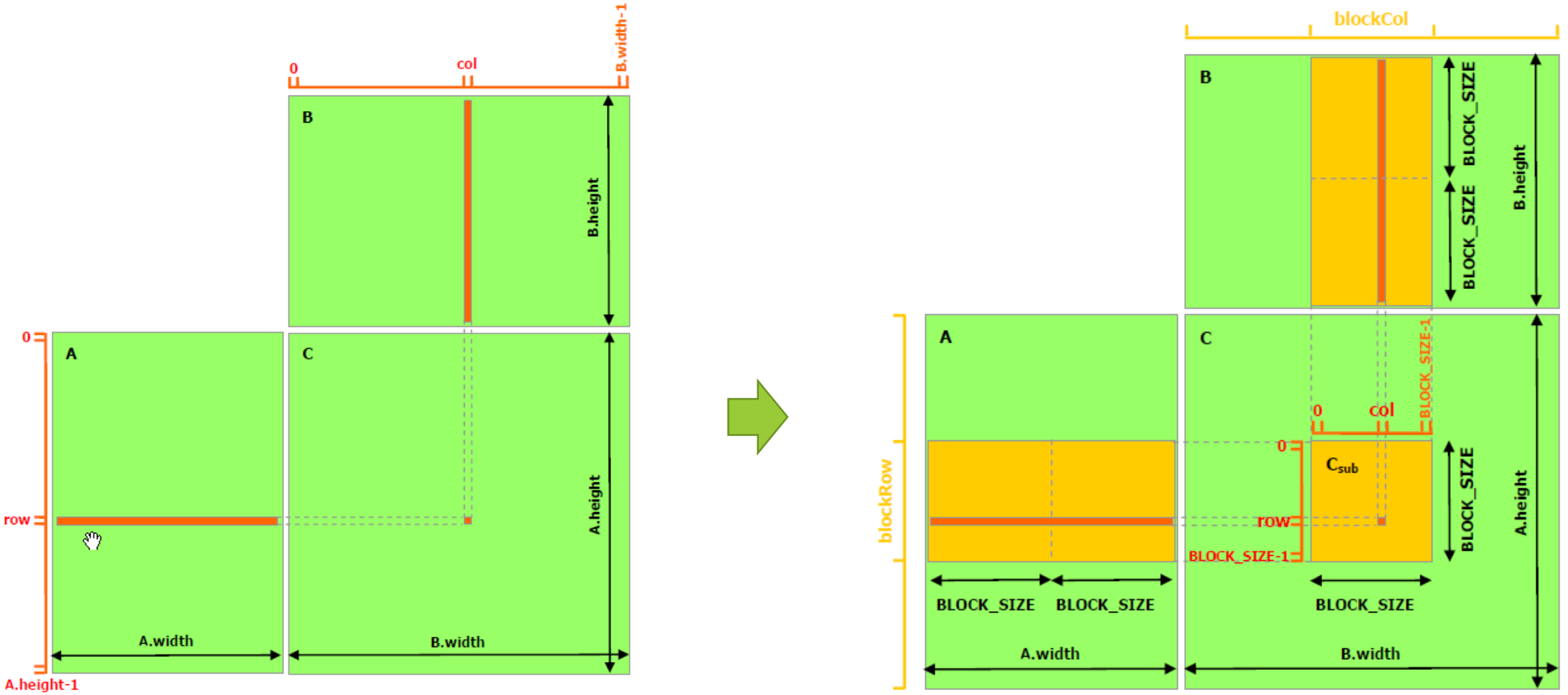








Matrix Multiplication with Shared Memory



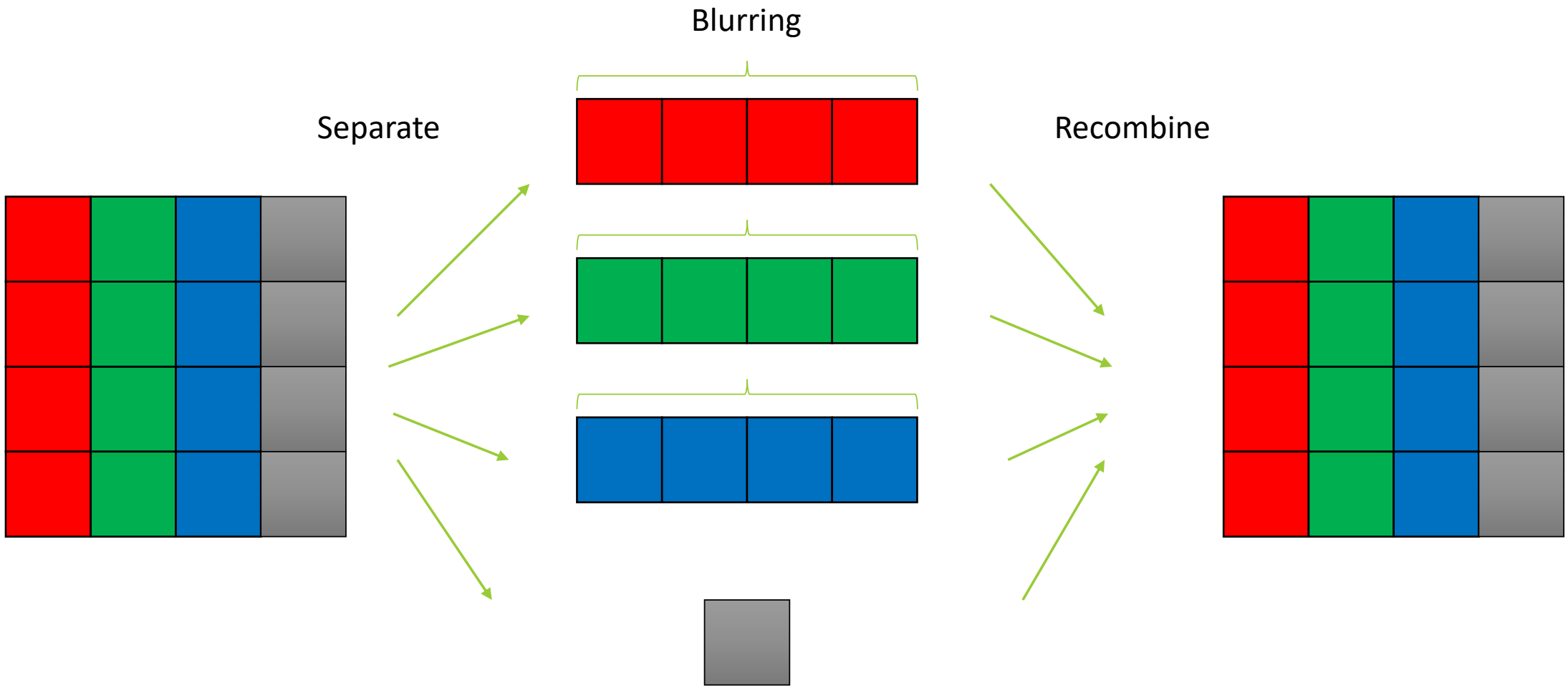
CUDA Practice 3

Blurring Filter 만들기

(Image Convolution)

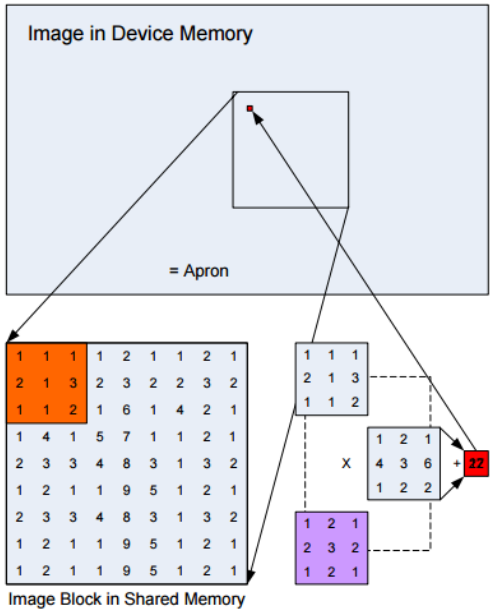
Image Blurring Filter (blur_filter.ipub)

Image Blurring Filter Operation

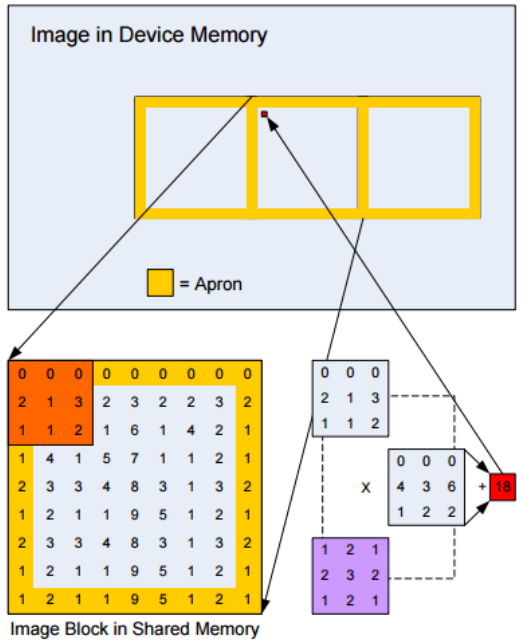


NVIDIA.

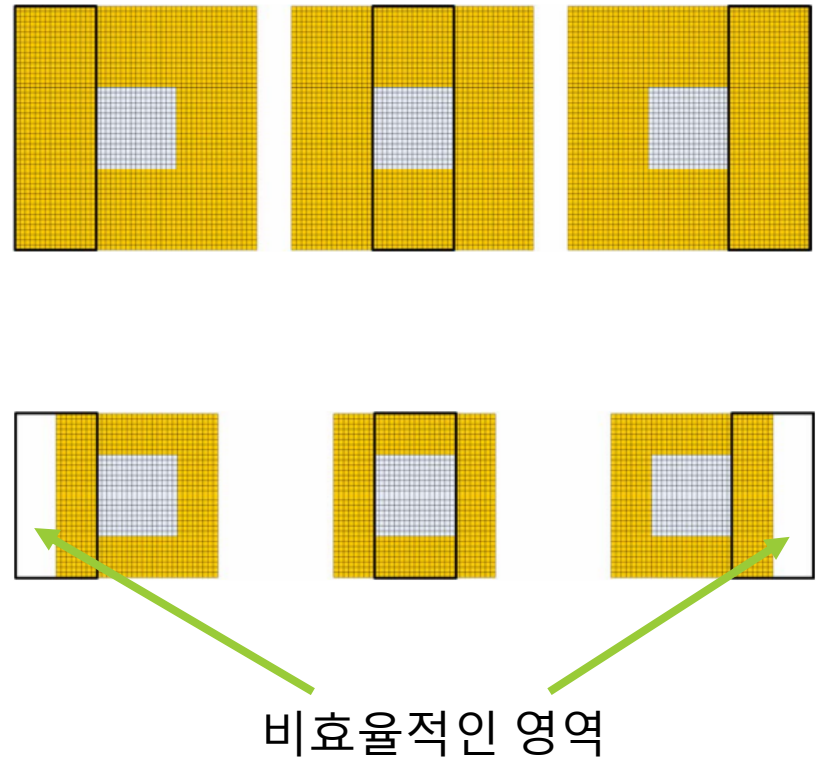
Naïve approach vs Shared Memory



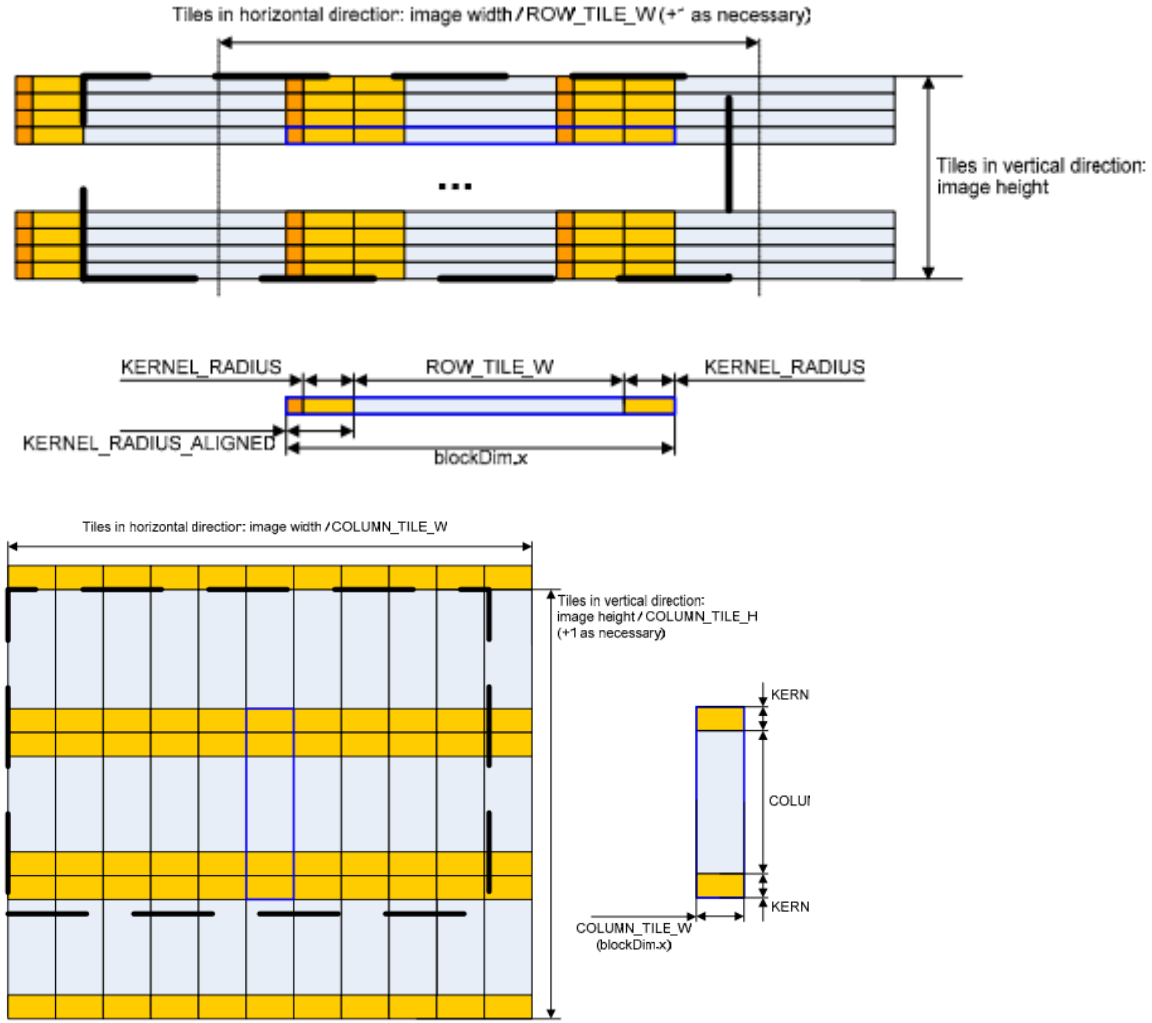
Naïve Approach

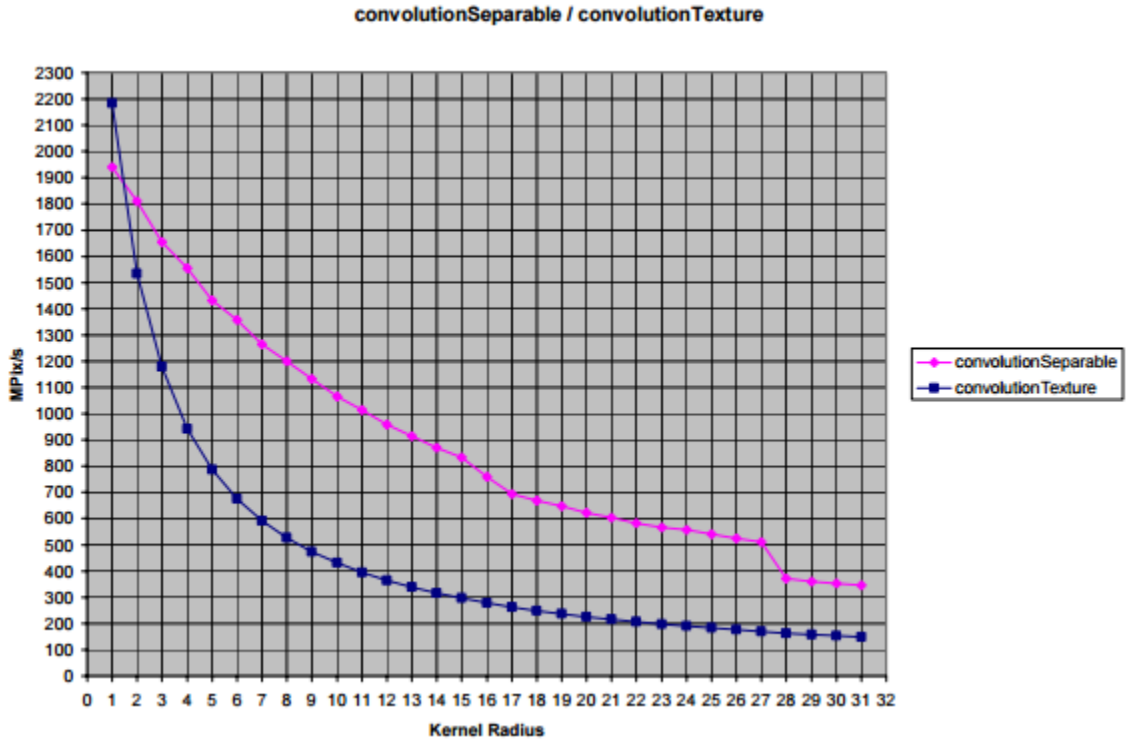


Apron & shared memory



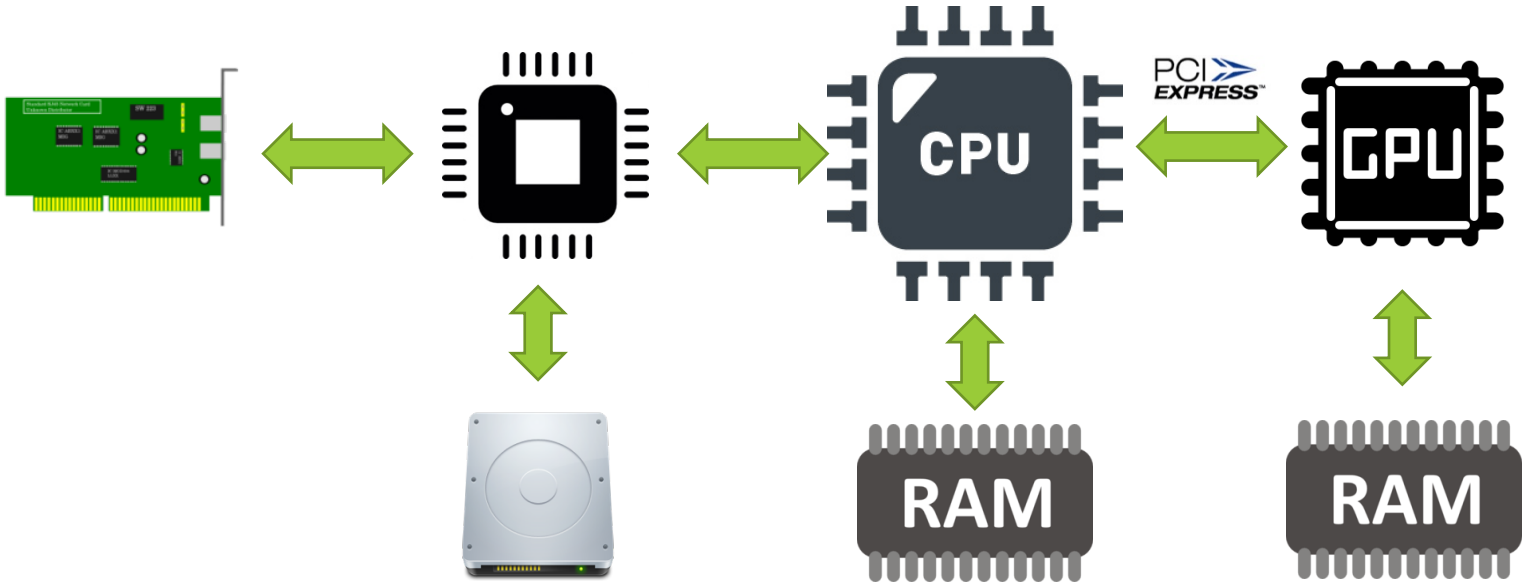
Separated Convolution





데이터 전송 최적화

- IO 병목과 같이 CPU, GPU 간 데이터 전송 지연시간이 큼

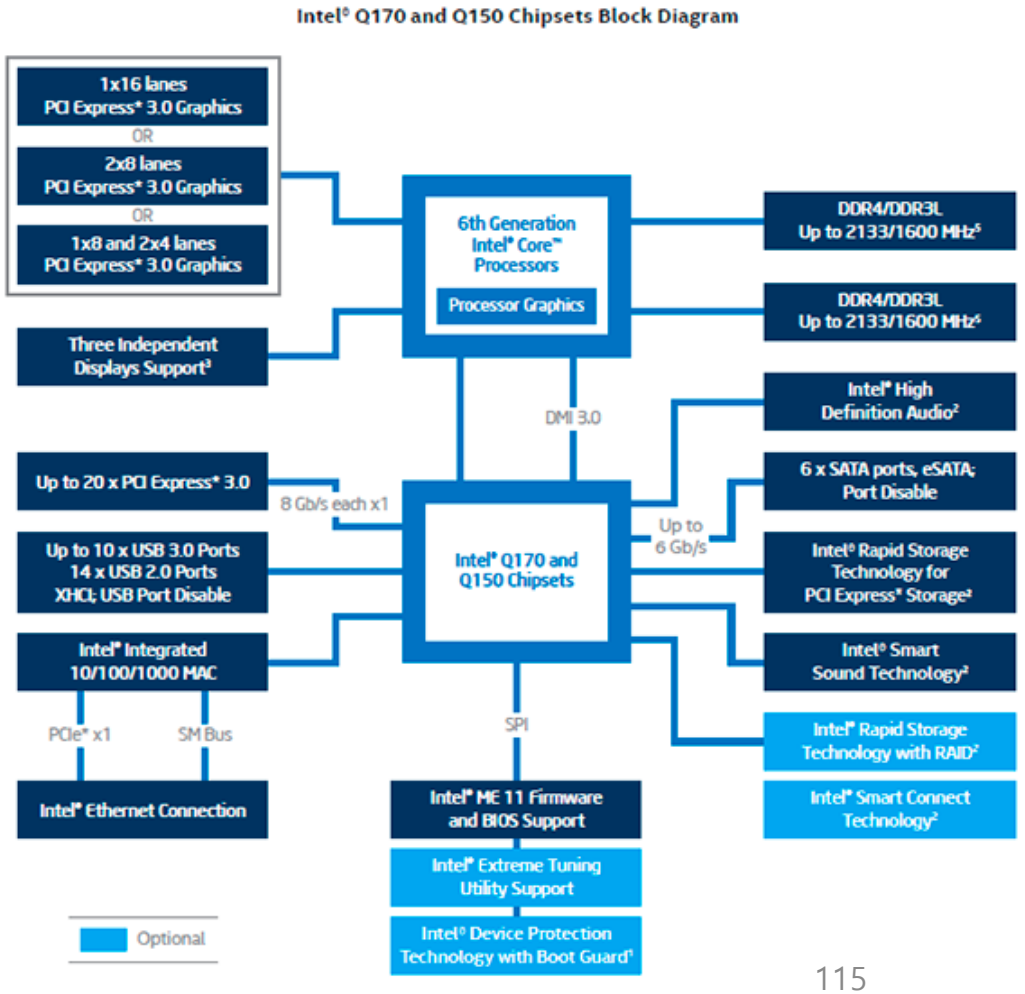


	BandWidth x1	BandWidth x16
PCIe 2.x	500 MB/s	16 GB/s
PCIe 3.x	1 GB/s	32 GB/s

NVIDIA.

About PCI EXPRESS

- 대용량 데이터 전송
- DMA 지원
- 지연시간 예측
  - $E(t) = {Data\ Size}/{PCIe\ BandWidth}$
  - ${4MB}/{32GB/s} \times 60 \times 2 \cong 15\mu s$ (이론 성능)



Page-Locked(Pinned) Memory Use

- Page Locked Memory란?
  - CPU-GPU간 Scatter-gather DMA를 할 수 있도록 구성되는 메모리
  - malloc()을 사용하여 할당한 메모리는 Page 단위로 DMA
  - 데이터 전송에서 CPU의 개입을 최소화 하여 성능 향상

- 할당
  - 모듈 내부터 활용
  - 이미 할당된 메모리의 변환

cudaHostAlloc(void* p, int_64 bufsize)

cudaHostRegister(void* p, int_64 bufsize)

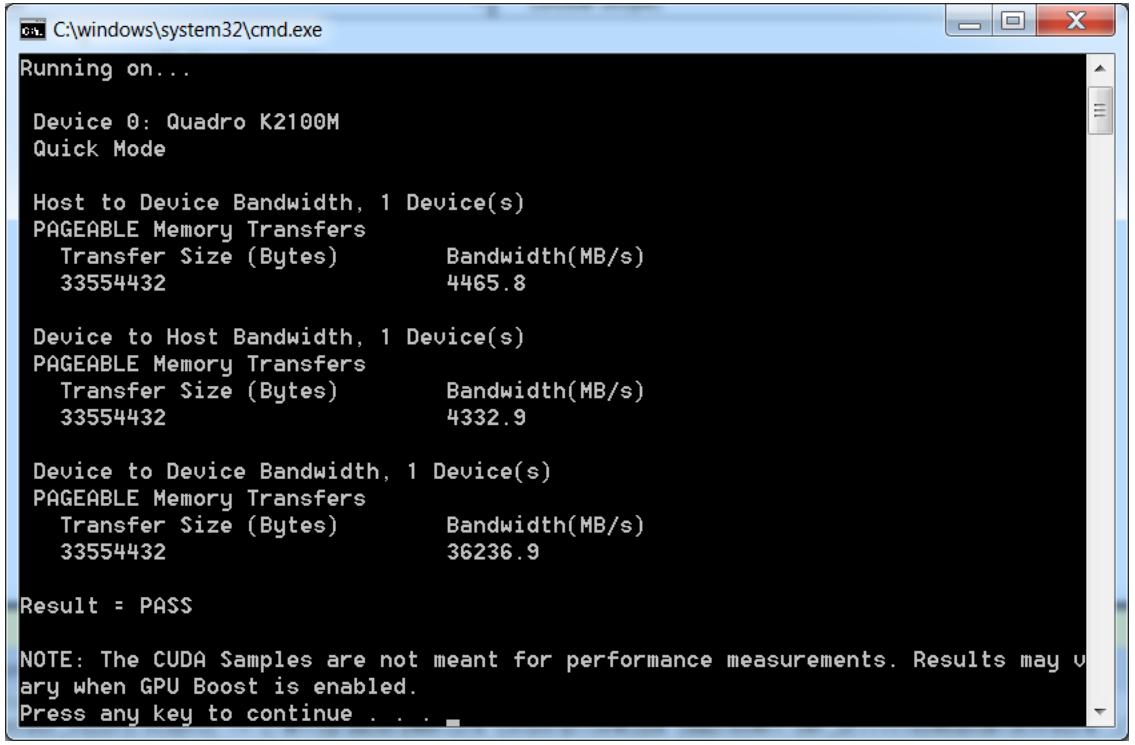
- 해제

cudaFreeHost(void* p)

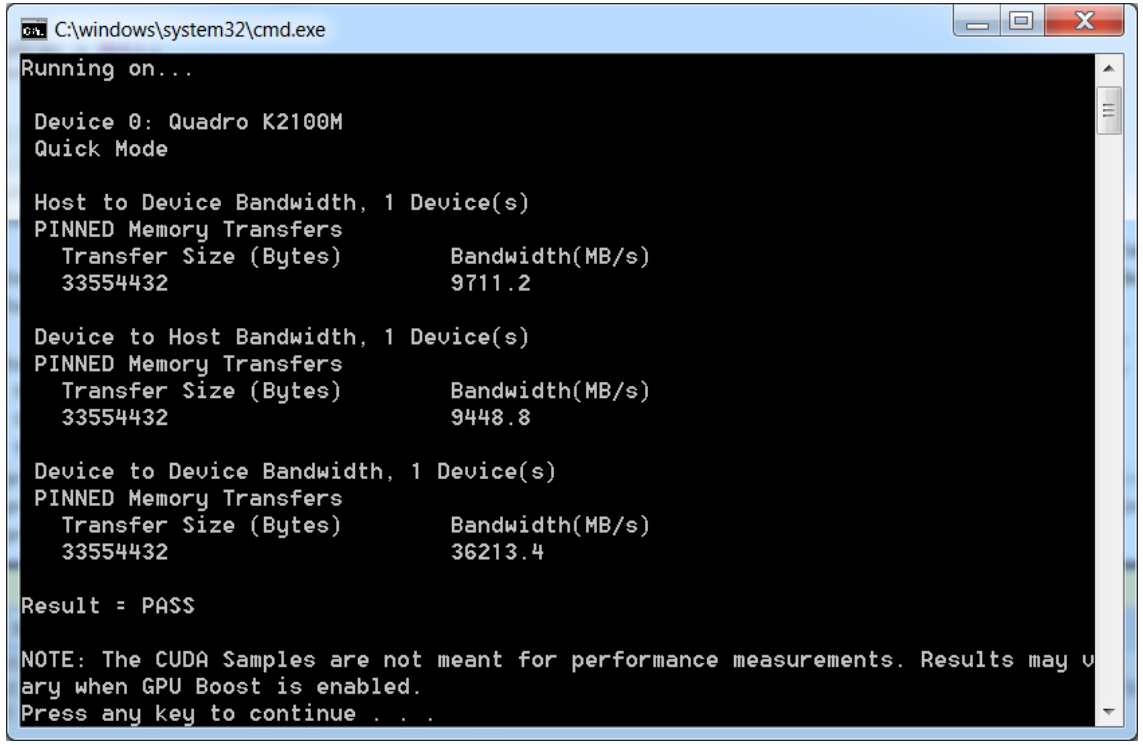
cudaHostUnregister(void* p)

NVIDIA.

CUDA Memcpy Bandwidth Test



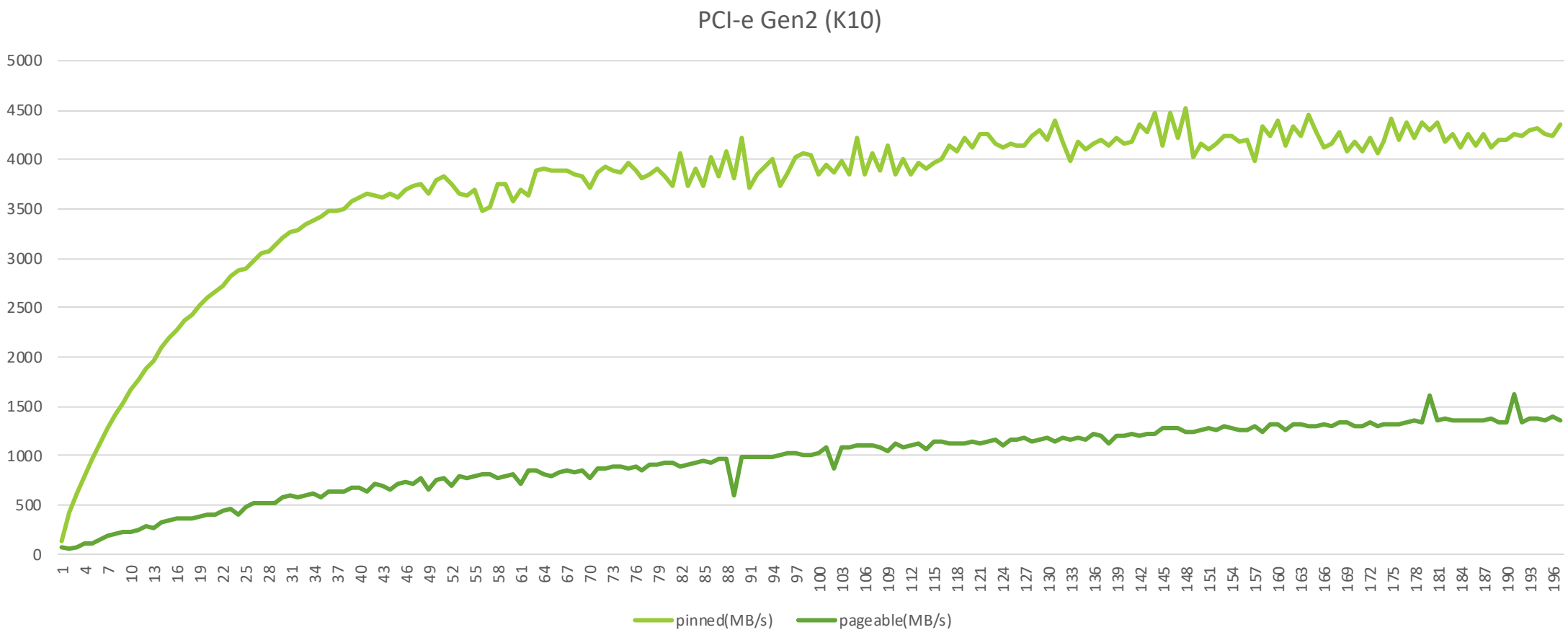
Pageable memory



Pinned memory

NVIDIA.

Pinned Memory



NVIDIA.

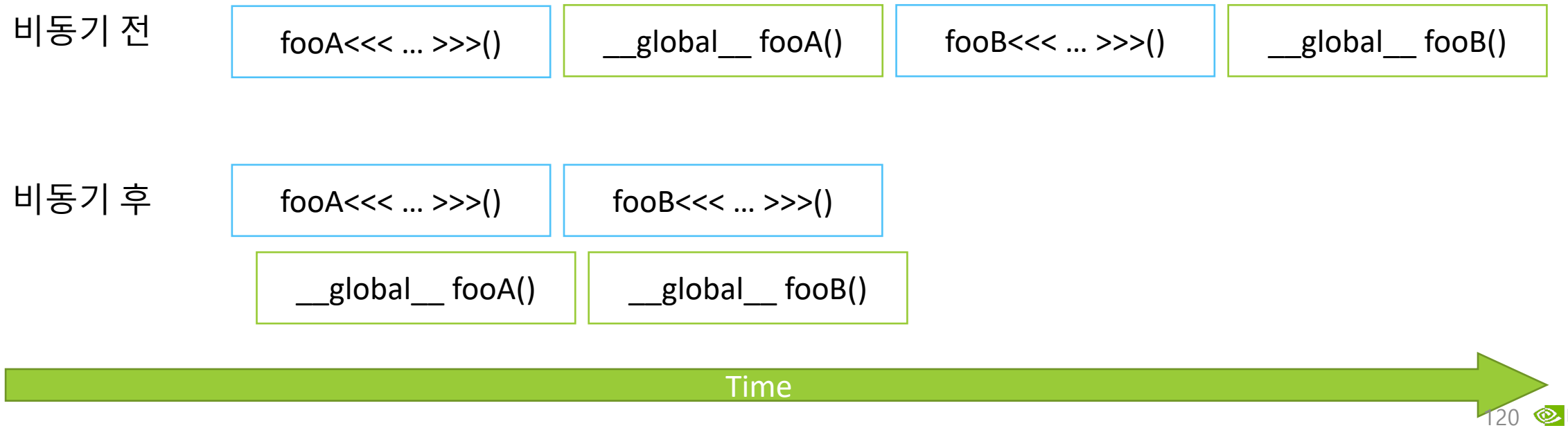
Pinned Memory의 종류

- cudaHostAllocDefault
  - 기본 동작
- cudaHostAllocMapped
  - Zero Copy에 활용
- cudaHostAllocPortable
  - Multi-GPU 환경에서 모든 GPU에 대하여 Pinned Memory 기능 활성화
- cudaHostAllocWriteCombined
  - Device로 데이터를 전송하는데 더 많은 대역폭을 할애

NVIDIA.

CPU / GPU 동작의 비동기화

- Asynchronous Concurrent Execution
  - Host와 Kernel의 동작이 비동기로 동작
  - Host code와 device code가 동시에 실행 가능



NVIDIA.

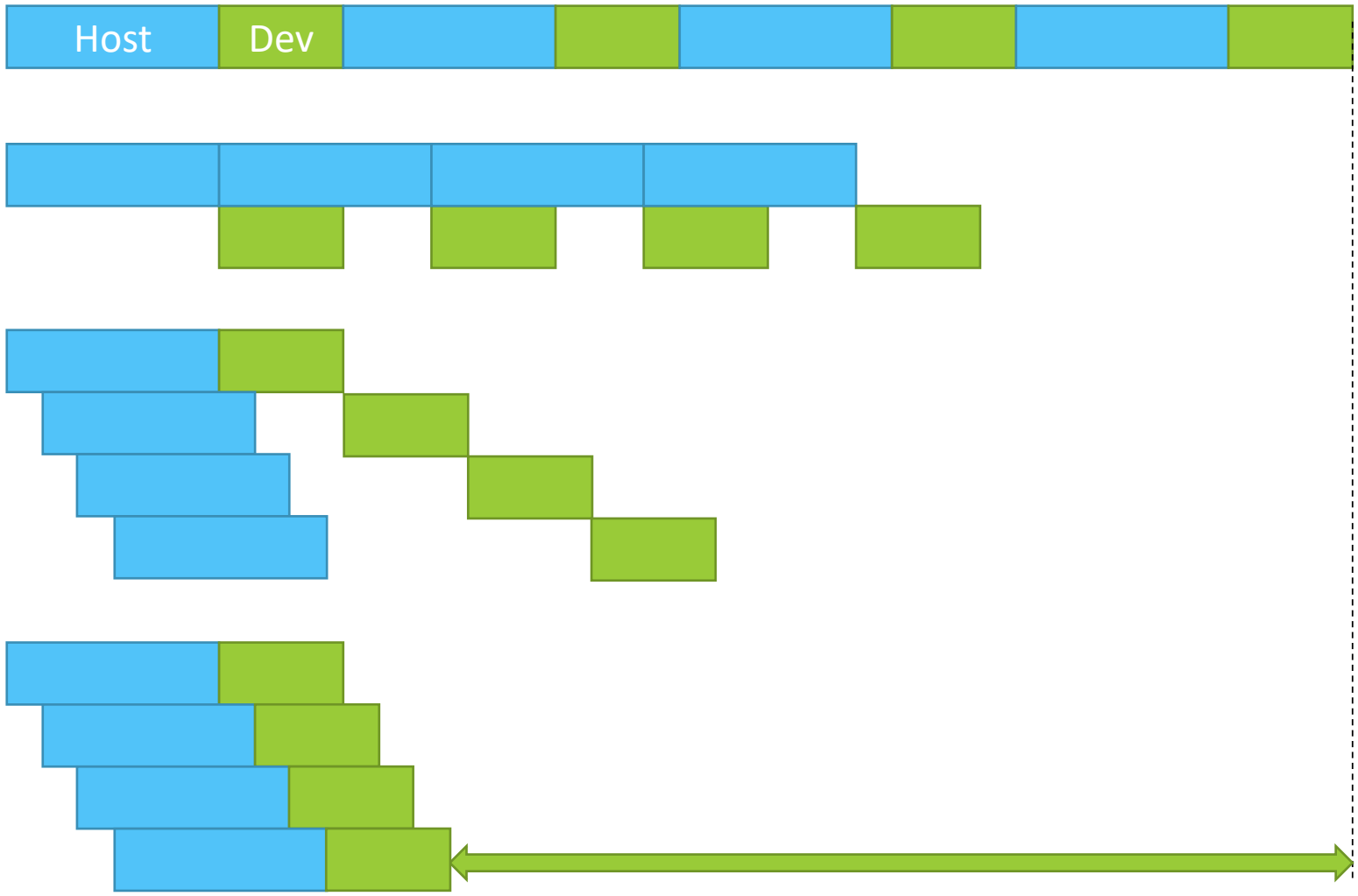
CUDA Async. 동작과 성능 향상

Synchronous Operation

Asynchronous Operation

Synchronous Operation + Multi-thread

Synchronous Operation + Multi-thread + CUDA Stream



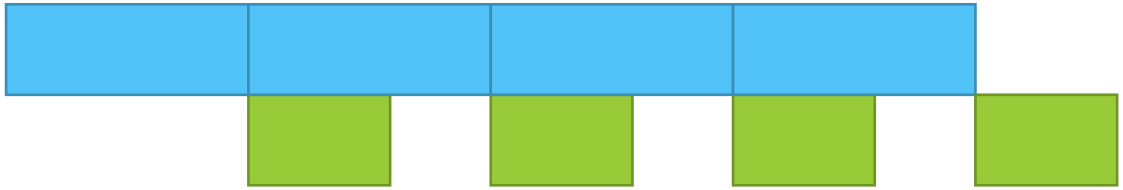
NVIDIA.

CUDA Asynchronous Operation

- CUDA Kernel은 원래 비동기로 동작함, cudaMemcpy의 특성상동기화발생
- CUDA 비동기가 가능한 cudaMemcpy 사용

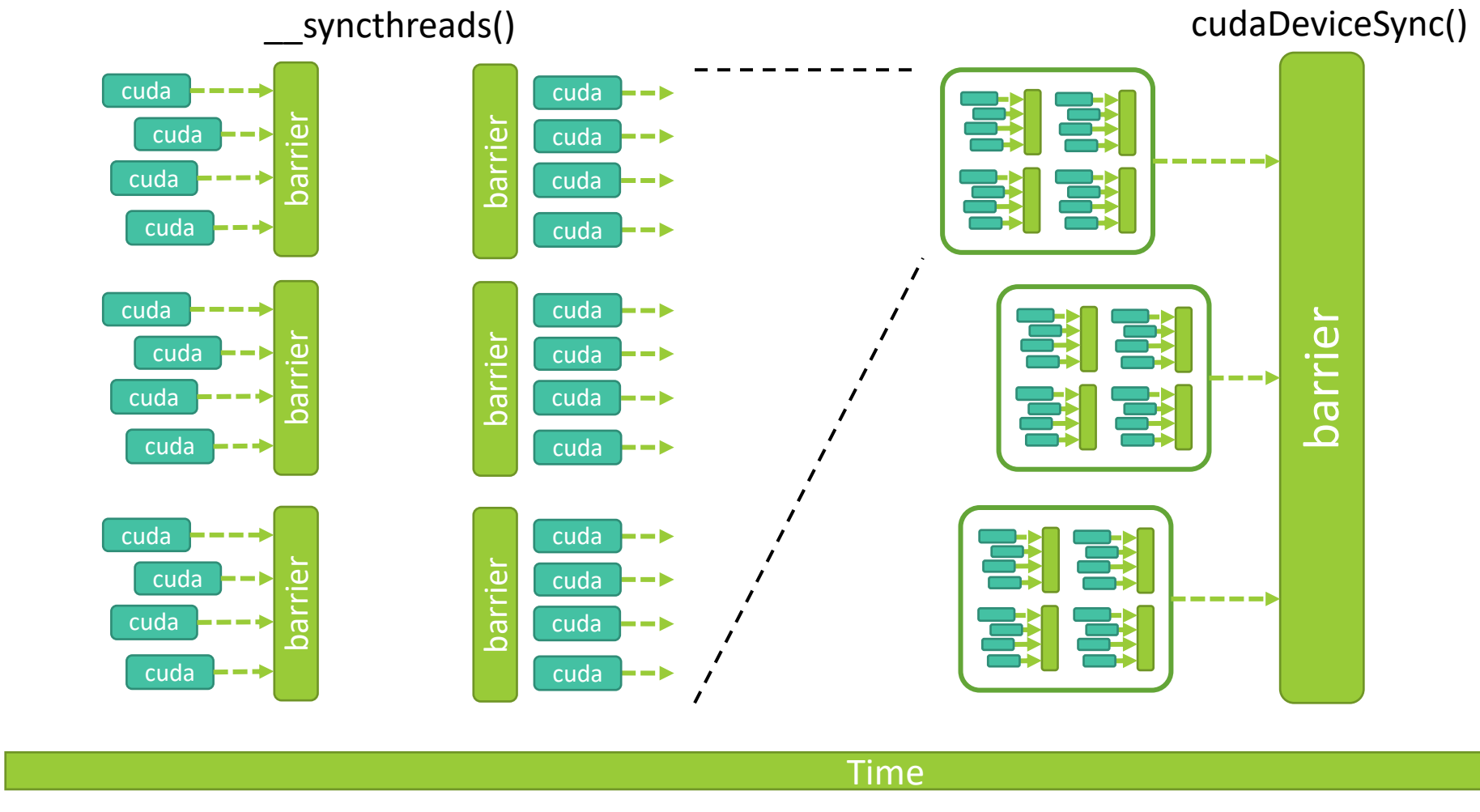
```
cudaMemcpyAsync(void* p1, void* p2, int size, {direction option}, stream_t = 0)
```

- 결과 데이터를 host memory로 옮기지 않는다면?



- Host Memory는 반드시 Pinned Memory를 사용해야 함
- CPU 간 동기화에 대한 전략도 고려해야 함

CUDA Thead & Block Sync.



Zero Copy

- 필요한 만큼만 데이터를 복사해서 사용

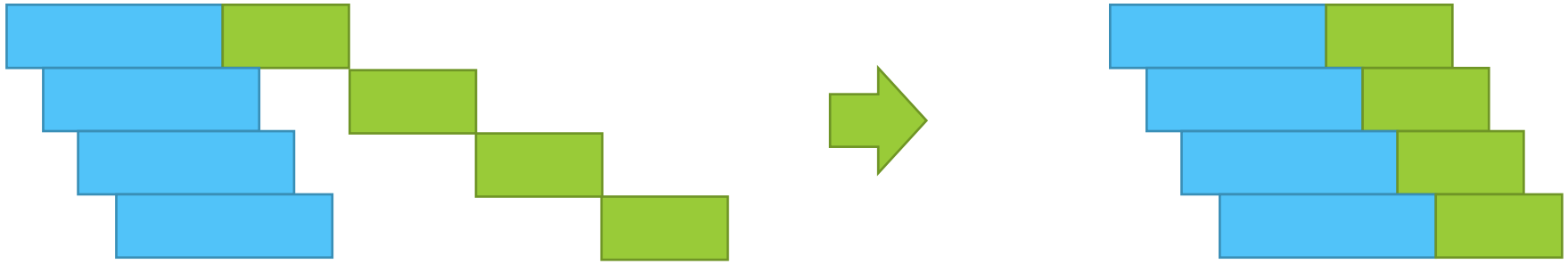
- Host 메모리에 대하여 GPU에서 바로 접근이 가능
  - 필요한 만큼만 데이터를 가져와 쓸 수 있게 함

- cudaMalloc 및 cudaMemcpy를 명시적으로 하지 않음
  - 대신 Mapping된 GPU 메모리 주소를 알아야 함

```
cudaHostGetDevicePointer(void** dev, void* host, 0);
```


NVIDIA.

GPU에서 Pipeline 구성하기



- CUDA Stream의 갯수대로 병렬로 실행가능

- Stream의 수는 필요한 만큼 생성 가능
  - 기본 CUDA Stream ID는 0
  - 비동기 CUDA runtime API에서 모두 지원함

NVIDIA.

CUDA Stream 활용

- Stream 생성 및 제거

```
cudaStreamCreate(cudaStream_t* stream);
```

```
cudaStreamDestroy(cudaStream_t* stream);
```

- CUDA Stream API
  - Kernel 호출

```
foo<<<dim3 gridDim, dim3 gridDim, int smem, cudaStream_t stream = 0>>>()
```

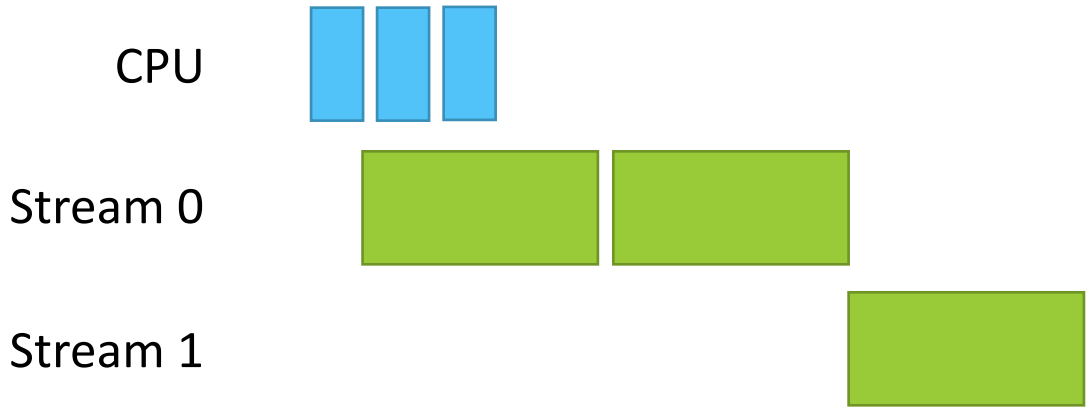
  - cudaMemcpyAsync

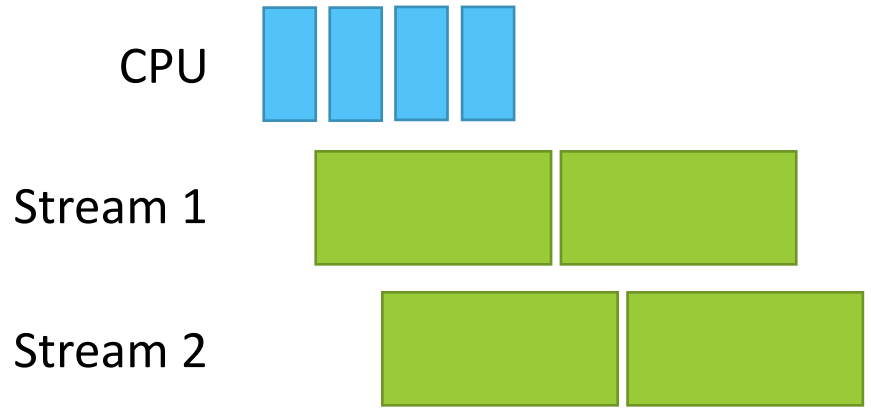
```
cudaMemcpyAsync(void* dst, void* src, int size, int mode, cudaStream_t stream = 0);
```


NVIDIA.

Default Stream & CUDA Stream

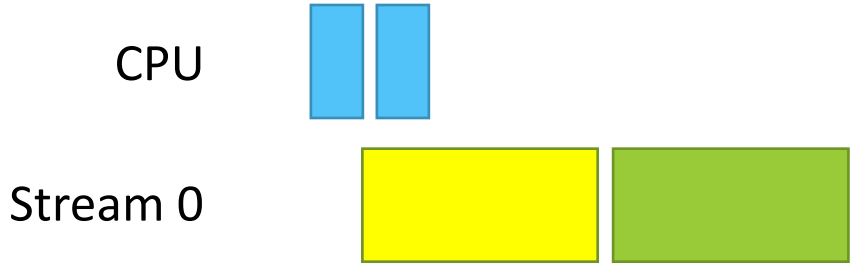
- Stream 0는 다른 Stream과 중복이 불가함
  - 모든 CUDA Stream이 동기화 됨
  - 필요한 stream 수만큼 CUDA stream 생성하여 사용
  - cudaStreamCreateWithFlags(&stream,cudaStreamNonBlocking)
    - stream 0에 대해서도 비동기 동작 가능
- 동일한 CUDA stream에서는 순차적으로 실행

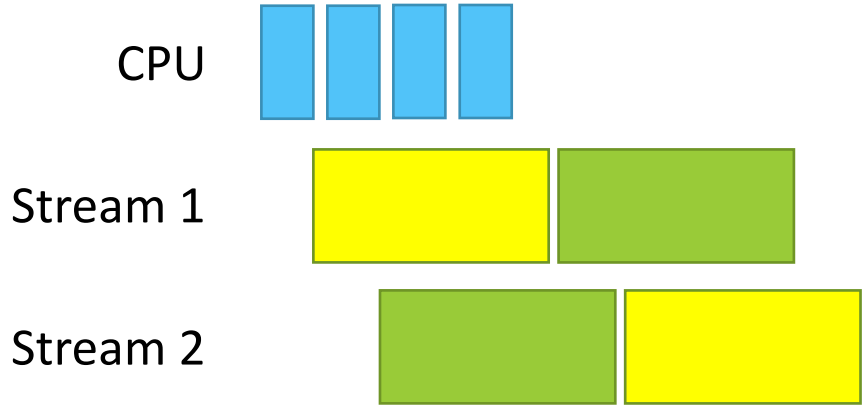




CUDA Stream & cudaMemcpyAsync

- cudaMemcpyAsync를 활용하여 GPU 동작 중 데이터 복사 가능

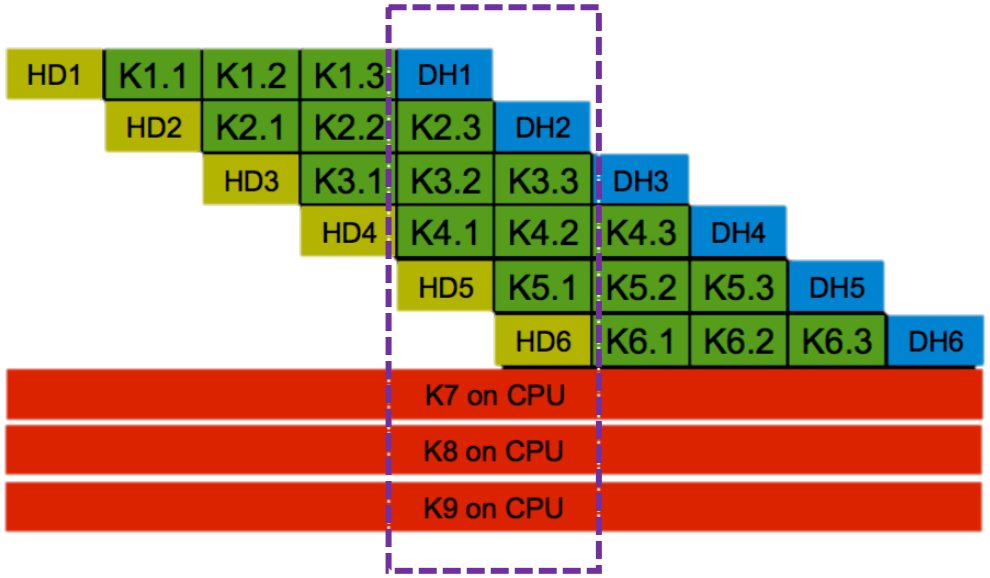




- 데이터 복사에 따른 지연시간을 줄일 수 있음

NVIDIA.

이런 오해가 있을 수 있습니다.



- 실제 병렬로 실행되기 위해서는 Occupancy가 허용 가능해야함

NVIDIA.

Stream Synchronization

```
cudaStreamSynchronize(cudaStream_t stream);
```

- CUDA stream에 대하여 동기화 가능
- cudaDeviceSynchronize()는 GPU의 모든 stream에 대하여 동기화

NVIDIA.

Advanced CUDA Stream

- 우선순위 부여
  - 가능한 Priority 값 알기

```
cudaDeviceGetStreamPriorityRange(int* lower, int* higher)
```

  - Priority를 가진 CUDA Stream 생성

```
cudaStreamCreateWithPriority(&stream, flags, priority)
```

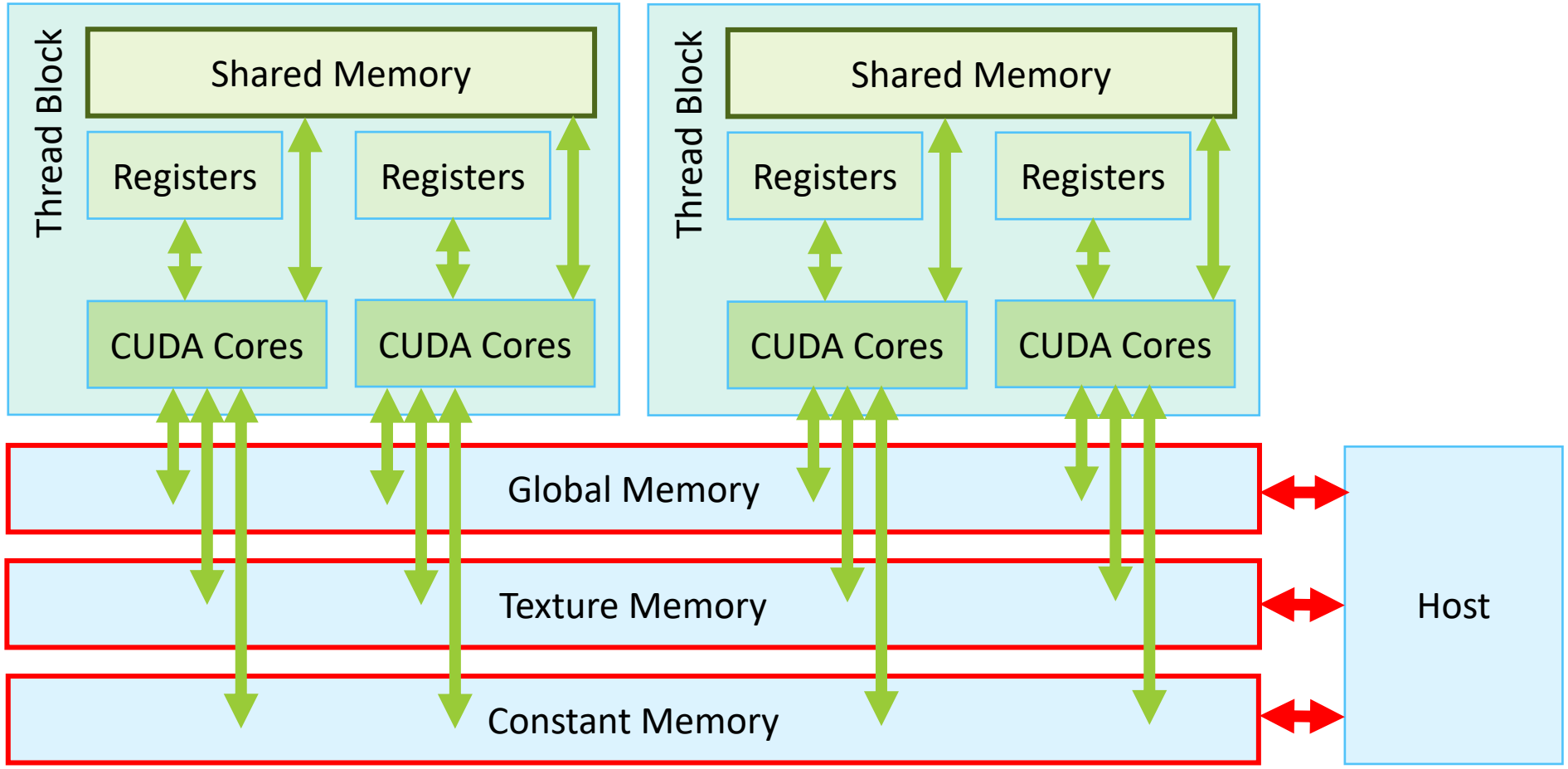
- Stream Callback 함수
  - Stream에 의존한 Host 함수 호출 가능

```
Void CUDART_CB MyCallback(void* data)
```

```
cudaStreamAddCallback(stream, MyCallback, (void*)i, 0);
```


NVIDIA.

GPU 메모리

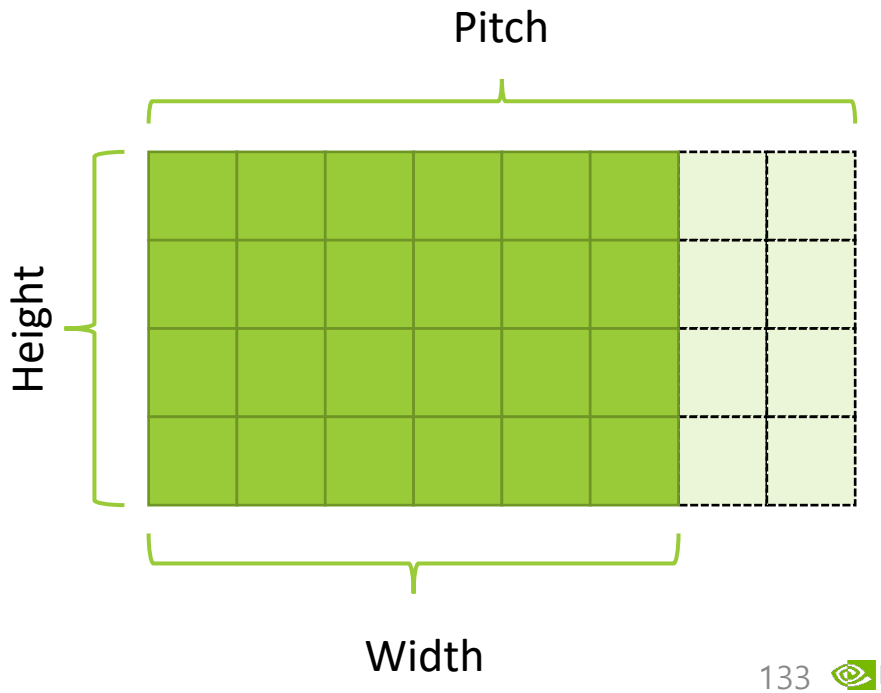


NVIDIA.

Global Memory 최적화

- 데이터 정렬
  - 32/64/128 byte 단위 데이터 통신
  - Global Memory는 256 byte 단위로 정렬

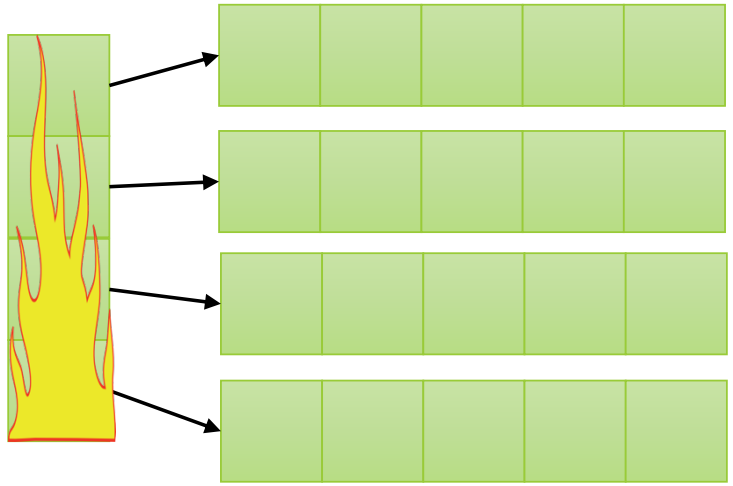
- 2차원 배열에 대한 최적화
  - cudaMallocPitch/Async()
    - Global Memory에 대한 자동 정렬
  - cudaMemcpy2D/Async()
    - Pitched 메모리로 데이터 복사시 사용



NVIDIA.

Device memory 사용시 1차원 배열 권장

- 동적 할당 된 2차원 배열 사용시 GPU 병렬성 저하



- Array of Pointer를 이용
  - Global Memory 접근이 2회로 증가
  - Global Memory 대역폭 낭비가 심함

NVIDIA.

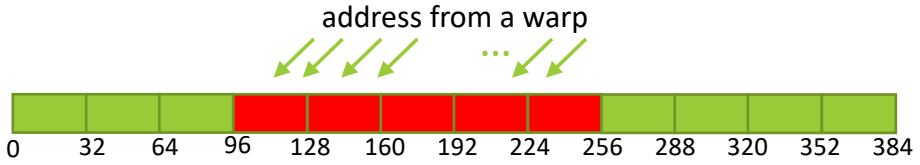
Coalescing Memory Access

- CUDA Core는 Global메모리에 접근시 block으로 access함
  - 사용시 128 byte 단위 access
  - Cache 미사용시 32 byte 단위 access
    - Nvcc에 Xptxas –dlcm=cg 옵션으로 L1 Cache off 가능

Coalesced Memory Access

Misaligned Memory Access

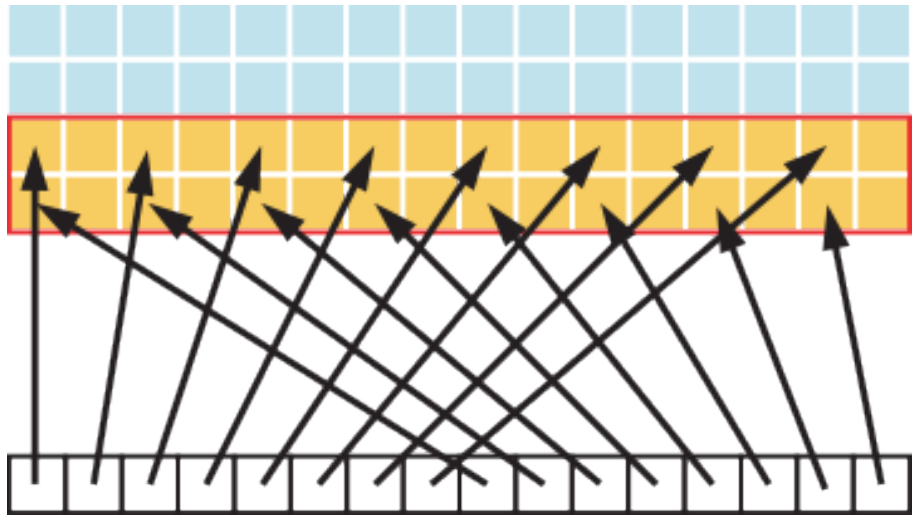
Misaligned Memory Access with L1 Cache-off

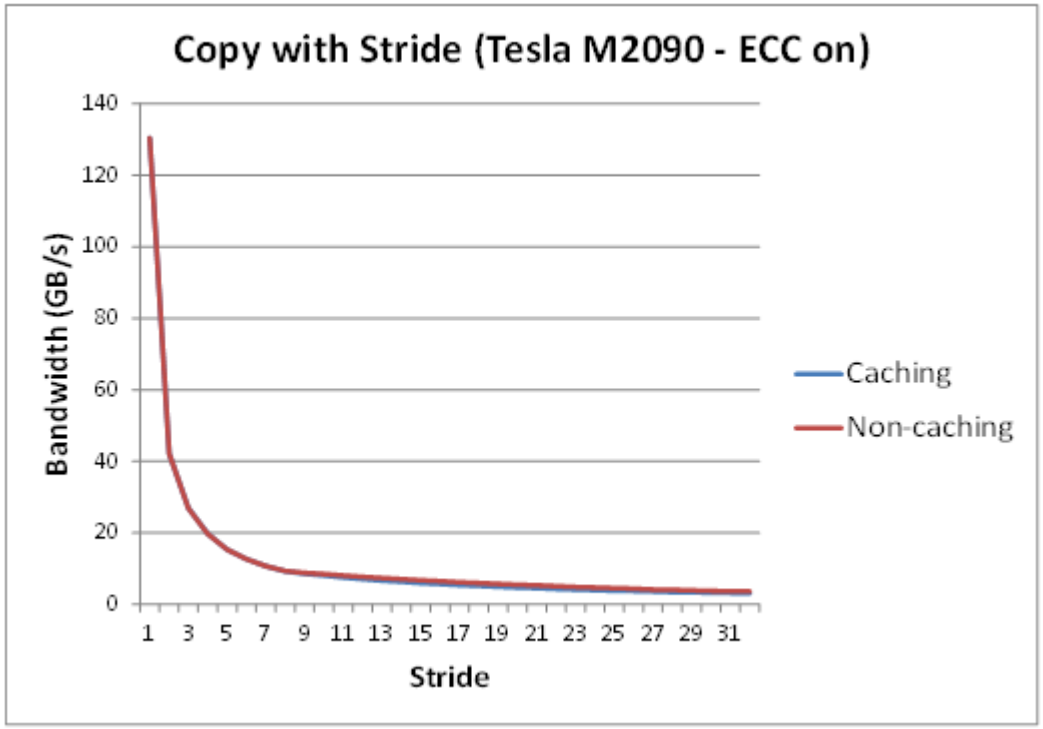


NVIDIA

Strided Access

- CUDA Thread간 간격(stride)를 갖고 memory에 Access 하는 경우의 성능효과





NVIDIA.

Texture Memory

- 읽기 전용 Cache 메모리
- Texture object / Texture reference API 제공
- GPU에게 Texture memory가 반환할 값에 대한 자동 설정 가능
  - Boundary Addressing
  - Value Normalization
  - Interpolation

NVIDIA.

Texture 메모리 활용

Texture Object – Create & Pass

- Linear memory / CUDA array 준비
- Texture target 메모리 명세
- Texture channelDesc 명세
- Texture object parameter 명세
- Texture object 생성
- Kernel call시에 Paramter로 전달
- Kernel 내에서 fetch

Texture Reference – Bind

- Texture Reference 선언
- Linear memory / CUDA array 준비
- Texture channelDesc 명세
- Texture Reference 명세
- Texture Reference와 CUDA memory bind
- Kernel 내에서 fetch

CUDA Array & Fetch

- Texture / Surface 메모리에 최적화된 memory layout
  - 1D, 2D, 3D 인덱스 지원
  - 다양한 type(8-, 16-, 32-bit)의 element 구성(1, 2, or 4)에 대한 최적화
  - Kernel 메모리에서만 활용 가능 (cudaMemcpyToArray 필요)

- 할당 및 해제

```
cudaMallocArray(cudaArray* cuArray, channelDesc, width, height);
```

```
cudaFreeArray(cuArray)
```

- Fetch
  - 1D: tex1D<T>(texObj, idx); / tex1Dfetch(texObj, idx);
  - 2D: tex2D<T>(texObj, idx_x, idx_y);
  - 3D: tex3D<T>(texObj, idx_x, idx_y, idx_z)

CUDA Object only

NVIDIA.

Texture Memory Binding

- Texture Memory 선언 및 Binding

1D

```
texture<float, 1, cudaReadModeElementType> texRef;
cudaChannelFormatDesc channelDesc = cudaCreateChannelDesc<float>();
cudaBindTexture(offset, texRef, devPtr, &channelDesc, size);
```

2D

```
texture<float, 2, cudaReadModeElementType> texRef;
cudaChannelFormatDesc channelDesc = cudaCreateChannelDesc<float>();
cudaBindTexture2D(offset, texRef, devPtr, &channelDesc, width, height, pitch);
```

3D

```
texture<float, 3, cudaReadModeElementType> texRef;
cudaChannelFormatDesc channelDesc =
                        cudaCreateChannelDesc <32, 32, 32, 32, cudaChannelFormatKindUnsigned);
cudaBindTextureToArray(texRef, cudaArray, channelDesc);
```

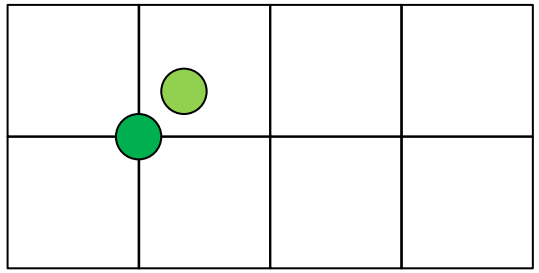

NVIDIA.

Texture Filter Mode

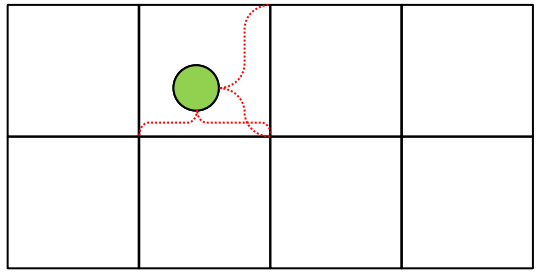
- 하드웨어 설정으로 Texture Address 모드 선택

```
texRef.addressMode[0] = cudaAddressModeWrap;
texRef.addressMode[1] = cudaAddressModeWrap;
texRef.filterMode     = cudaFilterModeLinear;
texRef.normalized     = true;
```

  - Filter Mode



cudaFilterModePoint

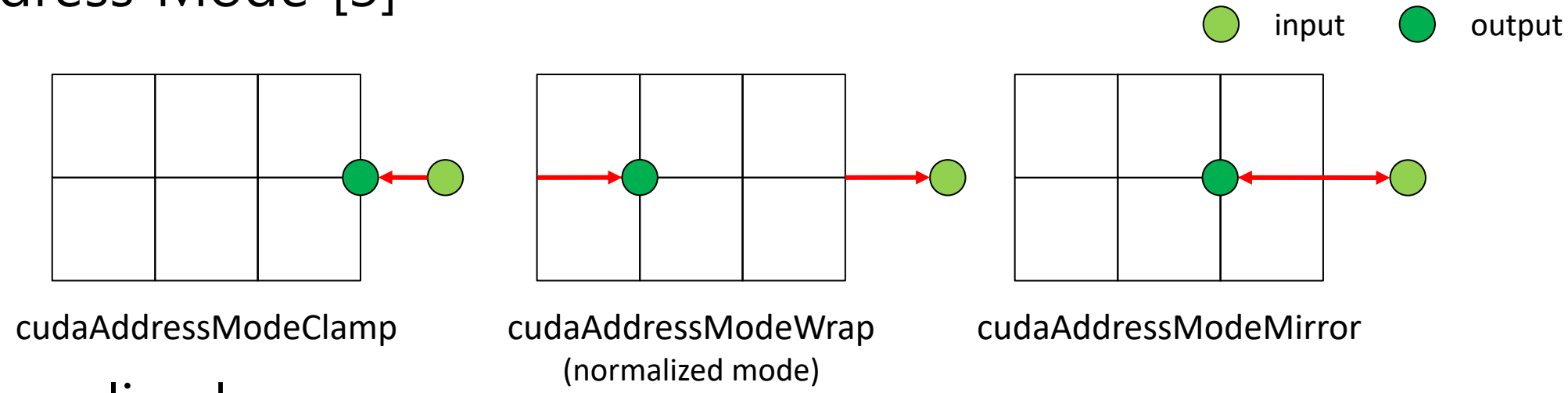


cudaFilterModeLinear

input output

Texture Address Mode & Normalized

- Address Mode [3]



- Normalized
  - 1(true): [0:1]로 normalized 된 값으로 index
  - 0(false): x, y, z 값으로 지정된 위치로 index

NVIDIA.

Constant Memory

- 읽기 전용
  - Kernel 실행 전 Data 복사
  - 새로 덮어쓰기 전까지 Application 내에서 데이터 보존

- API
  - 선언
    - __constant__ float table[N];

  - cudaMemcpyToSymbol() / cudaMemcpyFromSymbol()

- 크기가 작음 (64 KB)

NVIDIA.

Register 최적화

- Thread block당 사용할 수 있는 register 수에 제한이 있음
  - CUDA Computability 별 spec. 참조
  - Block당 thread 수를 64의 배수로 구성한 것을 기준으로 nvcc에서 register 최적화

- 필요한 경우 register 수 제한가능
  - -maxregcount=N

	Compute Capability							
Technical Specifications	**2.x**	**3.0**	**3.2**	**3.5**	**3.7**	**5.0**	**5.2**	**5.3**
Warp size	32							
Maximum number of resident blocks per multiprocessor	8	16				32		
Maximum number of resident warps per multiprocessor	48	64						
Maximum number of resident threads per multiprocessor	1536	2048						
Number of 32-bit registers per multiprocessor	32 K	64 K			128 K	64 K		
Maximum number of 32-bit registers per thread block	32 K	64 K						32 K
Maximum number of 32-bit registers per thread	63	255						

- Warp
  - CUDA Thread의 최소 제어단위
  - = 32 CUDA Threads
  - Occupancy Calculator를 통해 최대화하는 대상
  - 하나의 warp 내의 모든 CUDA thread는 동일한 동작을 함

- 분기 발생시 효율이 떨어짐

NVIDIA.

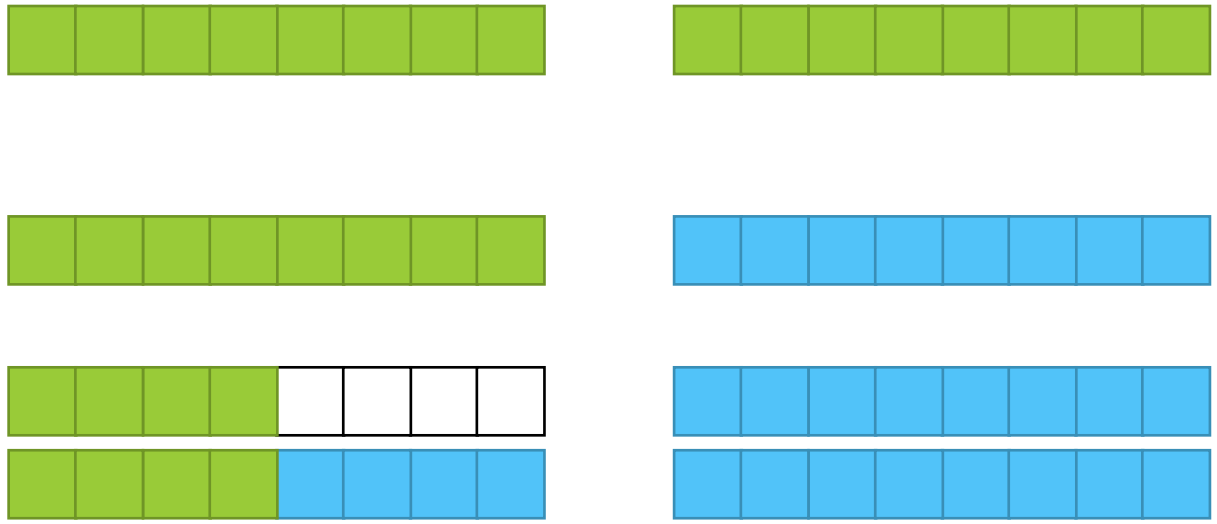
Warp Divergence

int idx = blockDim.x * blockIdx.x + threadIdx.x;

A[idx] = idx < 64 ? : ;

A[idx] = idx < 32 ? : ;

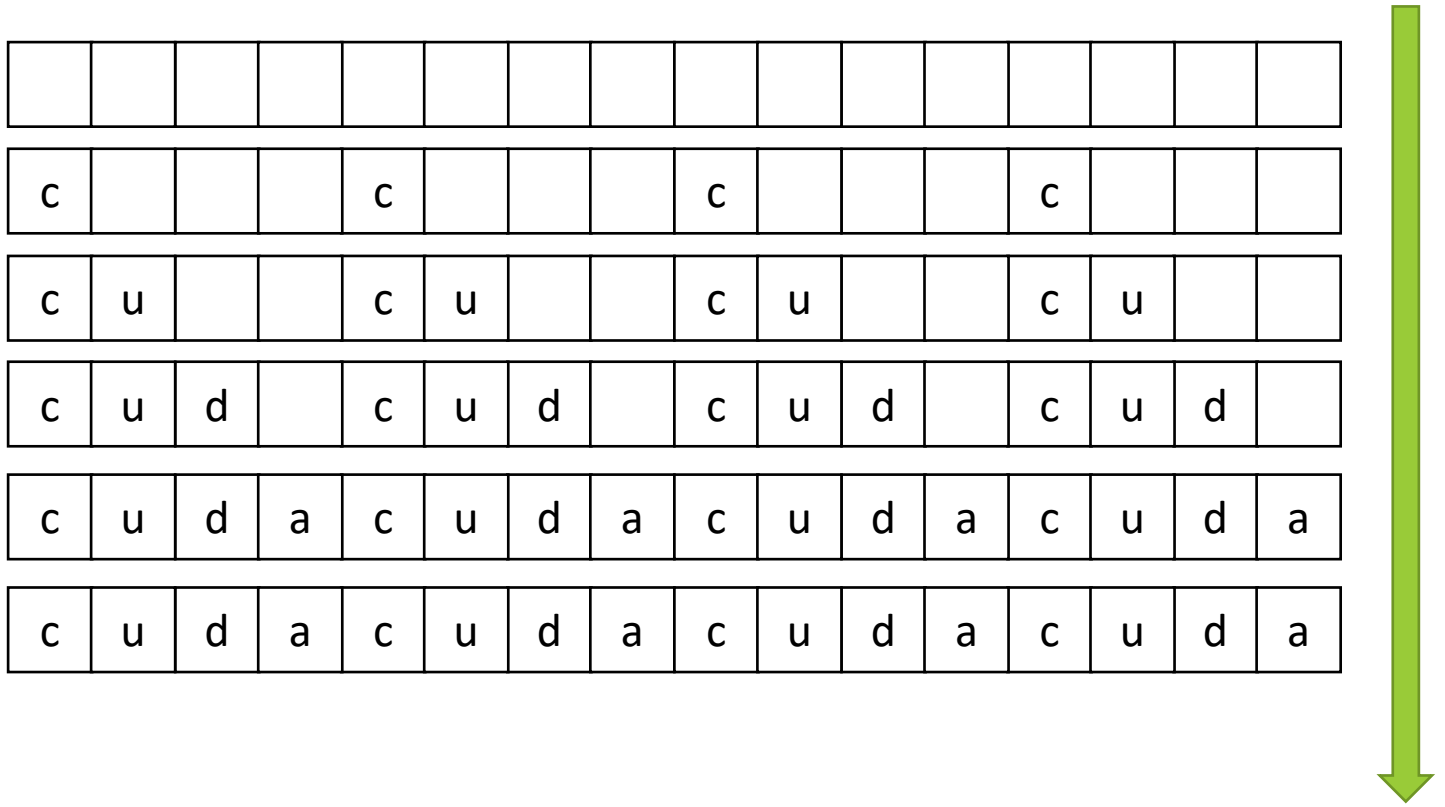
A[idx] = idx < 16 ? : ;



NVIDIA.

More Warp Divergence

```
switch (idx % 4) {
    case 0: A[idx] = ‘c’;
    case 1: A[idx] = ‘u’;
    case 2: A[idx] = ‘d’;
    case 3: A[idx] = ‘a’;
}
```



NVIDIA.

연산량 최적화

- Half Precision, Single Precision, Double Precision 최적화
  - 세부 지원 내용은 GPU 버전 및 종류마다 다름
  - 적절한 Architecture 선택 및 Precision이 낮을 수록 연산량 높음
  - HP의 경우 성능향상 가능한 GPU는 Tesla 만 지원

- Intrinsic function
  - Short, Char 데이터에 대한 SIMD 지원
  - 유효숫자를 줄인 round off 연산 가능 (오차범위 확인 필요)

- 2^n으로 나머지(modular) 연산을 하는 경우,

$$m \bmod 2^n = m \,\&\, (2^n - 1)$$

NVIDIA.

Parallel Programming Pattern II

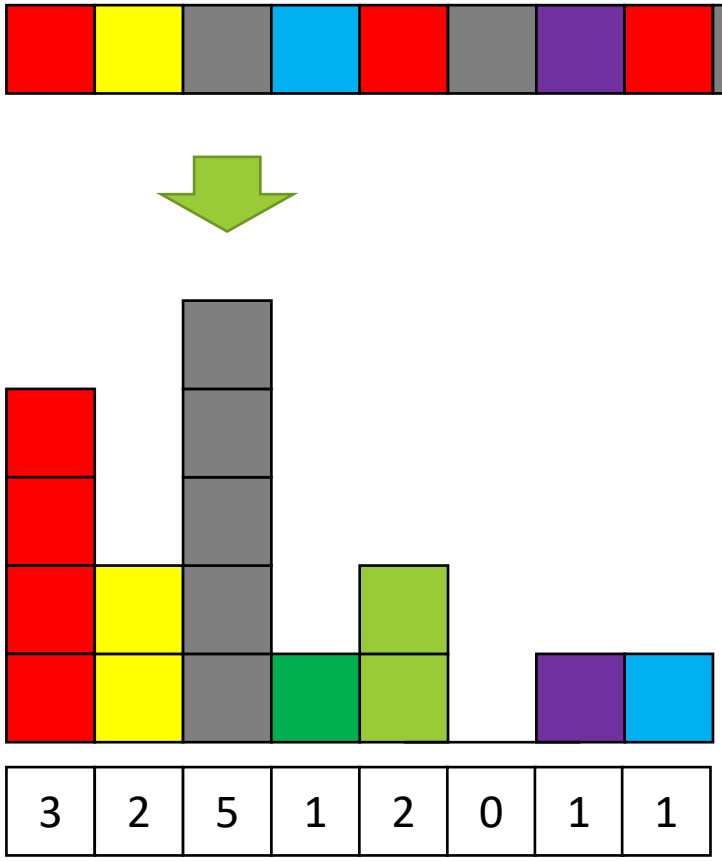
Parallel Communication Pattern

- **Selection**
- **Map**
- **Gather**
- **Stencil**
- **Scatter**
- **Transpose**
- *Histogram*
- *Reduce*
- *Scan*
- *Pack*
- *Sort*

NVIDIA.

Histogram

- Data의 분포를 분석하는데 사용




```
int value = (int)data[idx];
histogram[value]++;
```

결과예측 불가

```
int value = (int)data[idx];
atomicAdd(&histogram[value], 1);
```

예측한 결과

NVIDIA.

CUDA atomic operation

- CUDA thread의 atomic 동작 보장
  - 여러 Thread가 하나의 메모리 값을 갱신할 때 예상하는 값을 보장

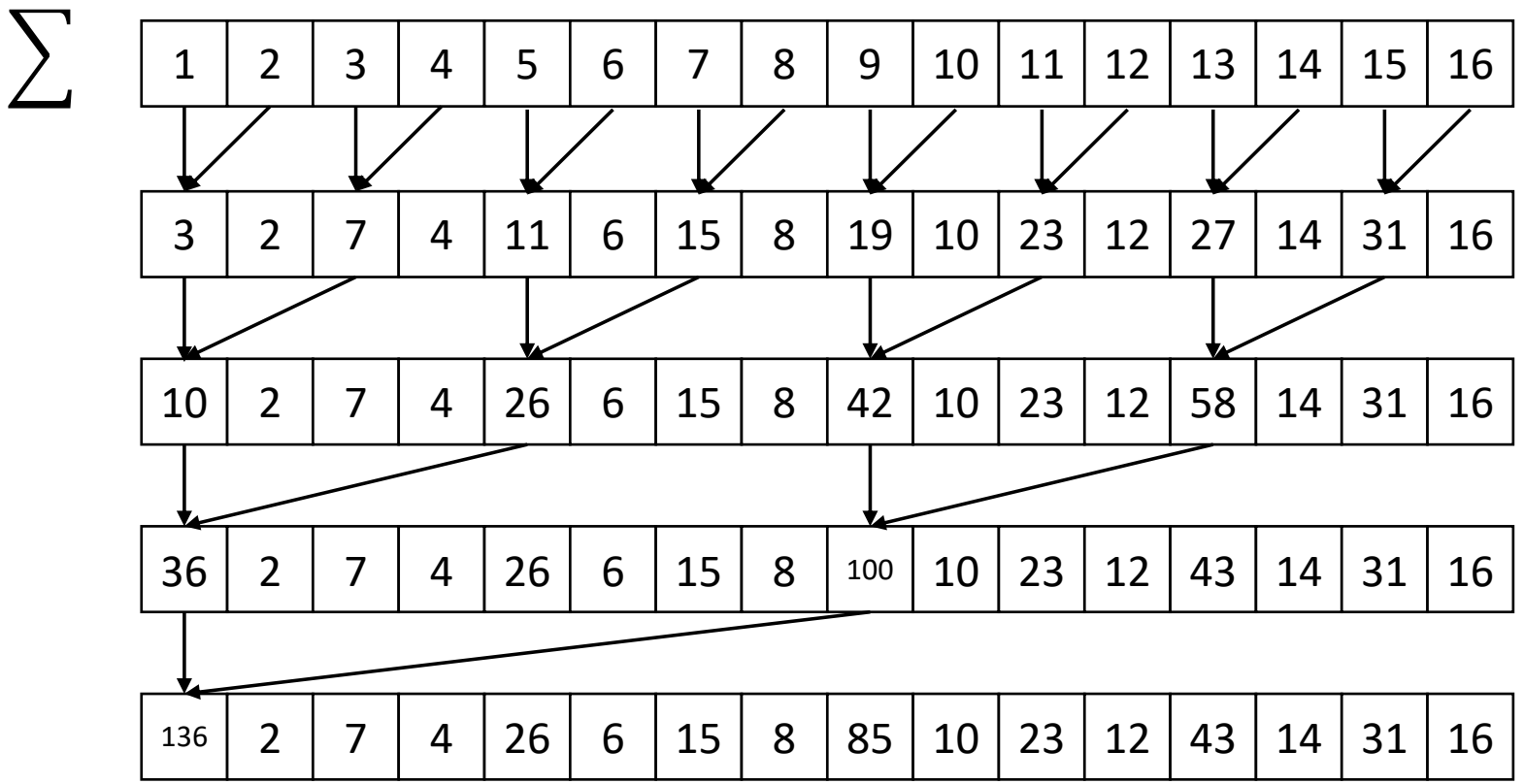
- APIs

```
int atomicAdd(int* address, int val);
int atomicExch(int* address, int val);
int atomicMax(int* address, int val);
int atomicCAS(int* address, int compare, int val);
int atomicAnd(int* address, int val);
…
```


NVIDIA.

Reduction – Simple Idea

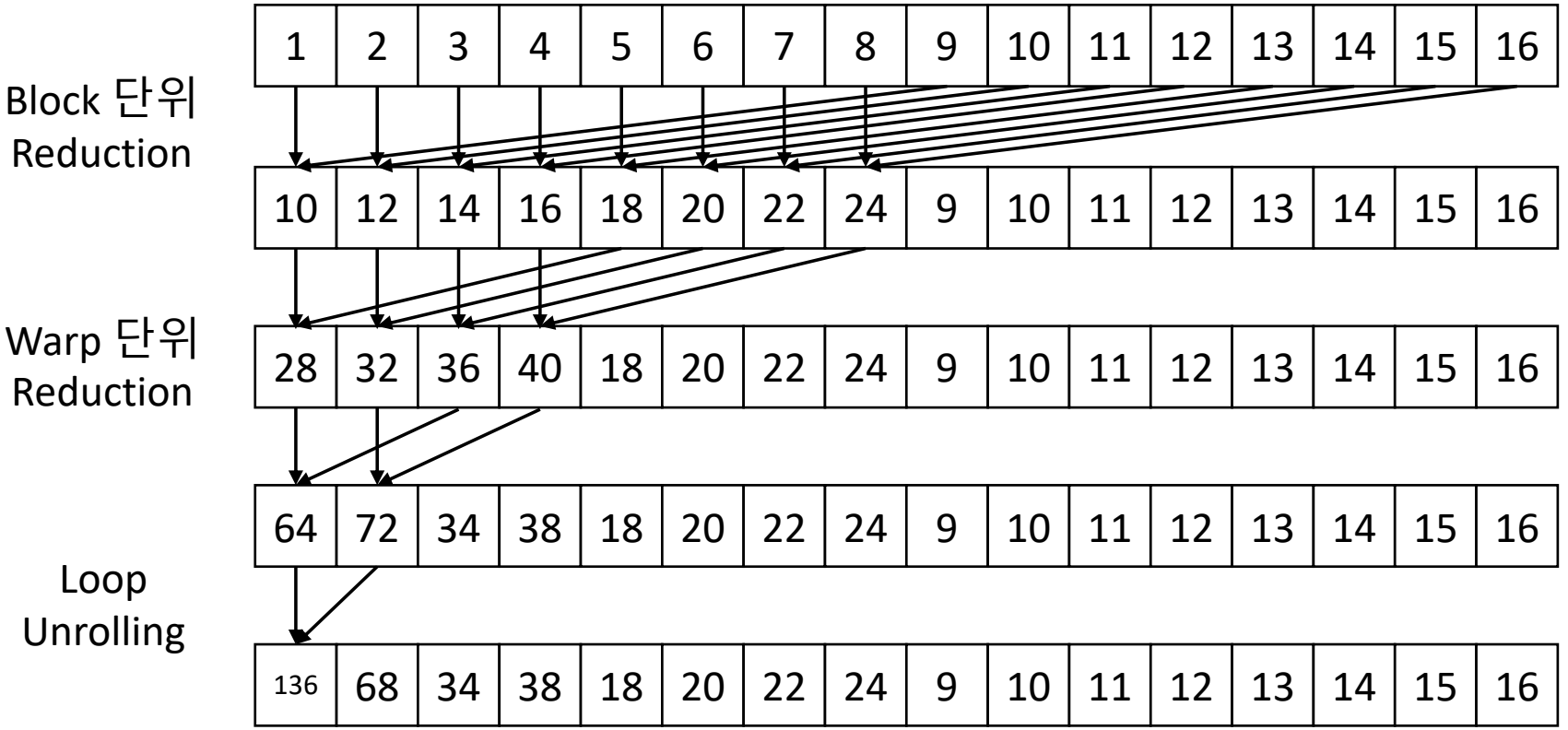
- 배열 내의 모든 값을 갖고 하나의 결과를 얻을 때,



NVIDIA.

Reduction – CUDA Optimized way

- Block Size, Warp Size 및 Loop Unrolling 적용



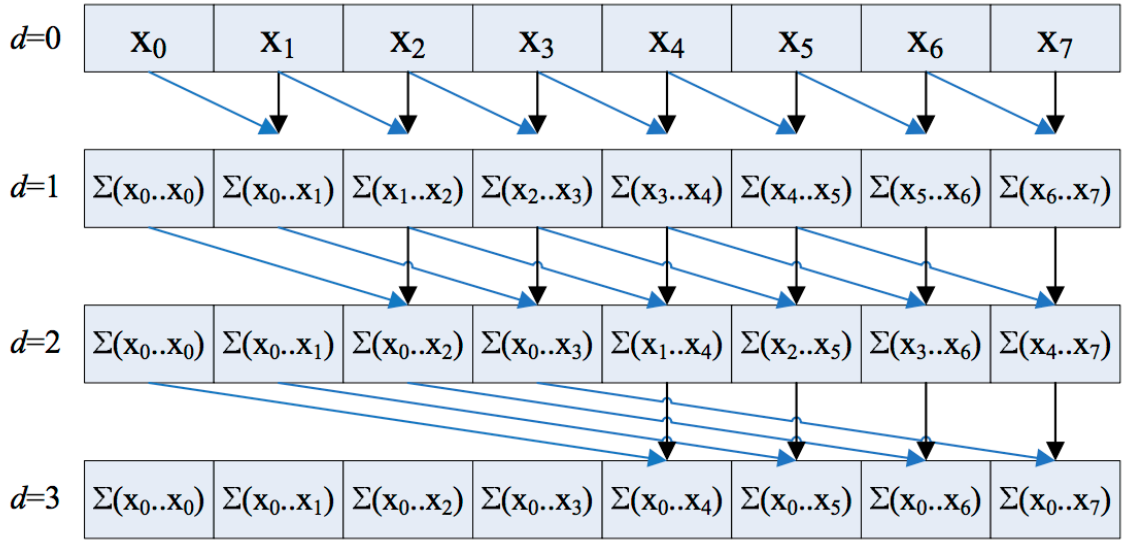
NVIDIA.

Scan

- 앞의 값을 참조해서 값을 구성하는 경우

[3 1 7 0 4 1 6 3]  [3 4 11 11 14 16 22 25].

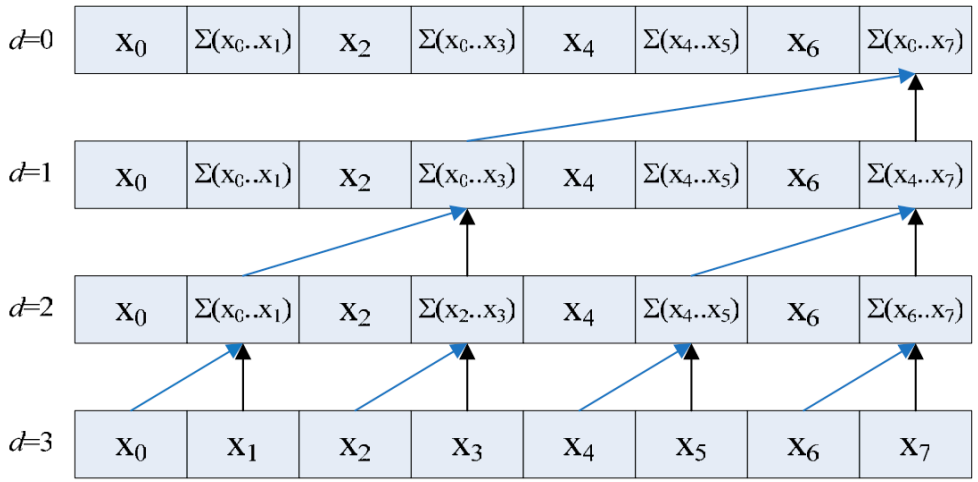
- NAÏVE Implementation (Hillis & Steele)



NVIDIA.

Optimized Scan (Blelloch)

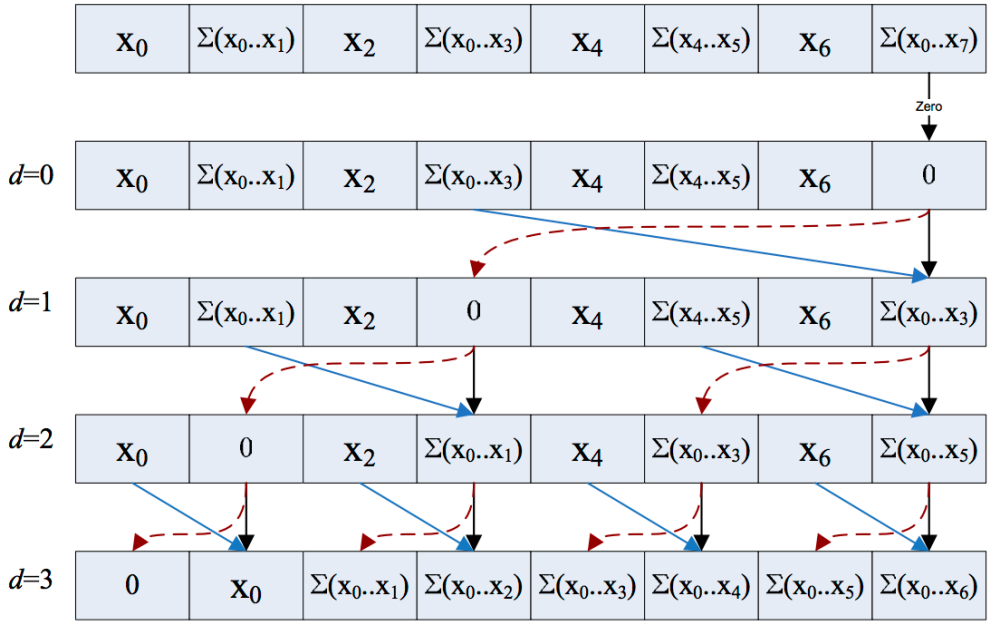
Up-sweep phase




```
for d := 0 to log2n - 1 do
    for k from 0 to n – 1 by 2^(d+1) in parallel do
        x[k + 2^(d+1) - 1] := x[k + 2^d - 1] + x[k + 2^(d+1) - 1]
```

$O(n) = \log(n)$

Down-sweep phase



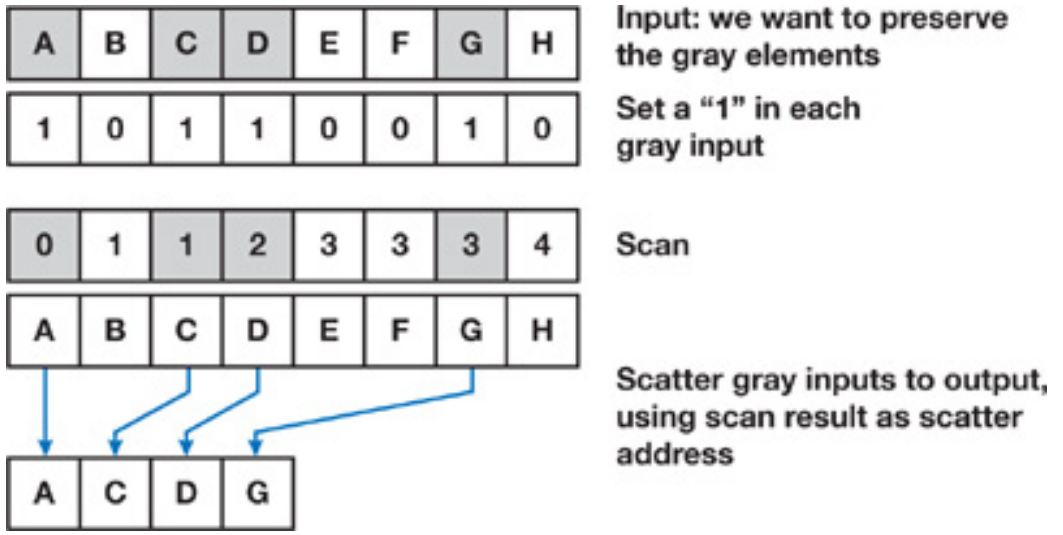

```
x[n - 1] := 0
for d := log2n down to 0 do
    for k from 0 to n – 1 by 2^(d+1) in parallel do
        t := x[k + 2^d - 1]
        x[k + 2^d - 1] := x[k + 2^(d+1) - 1]
        x[k + 2^(d+1) - 1] := t + x[k + 2^(d+1) - 1]
```

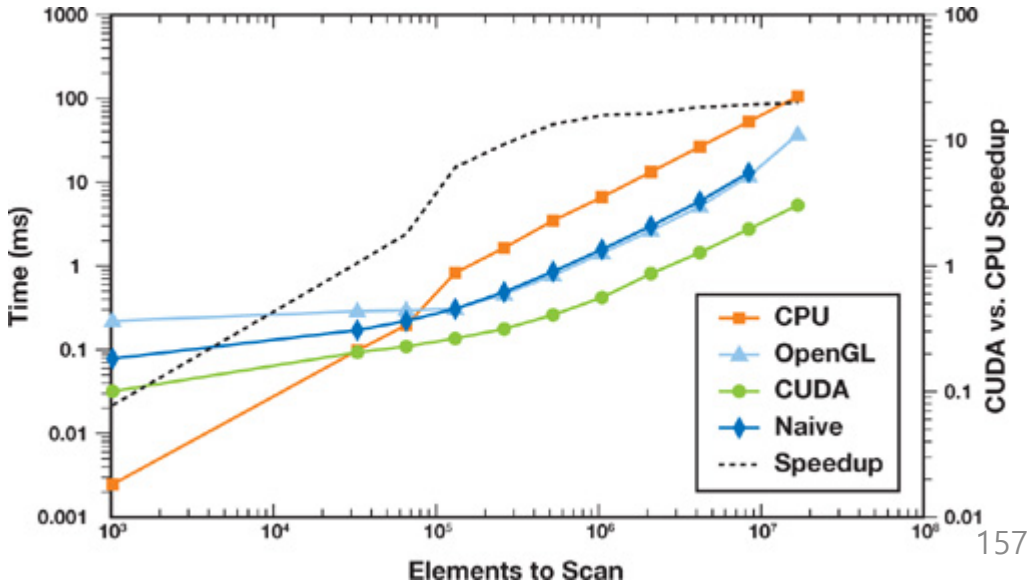

NVIDIA.

Compact

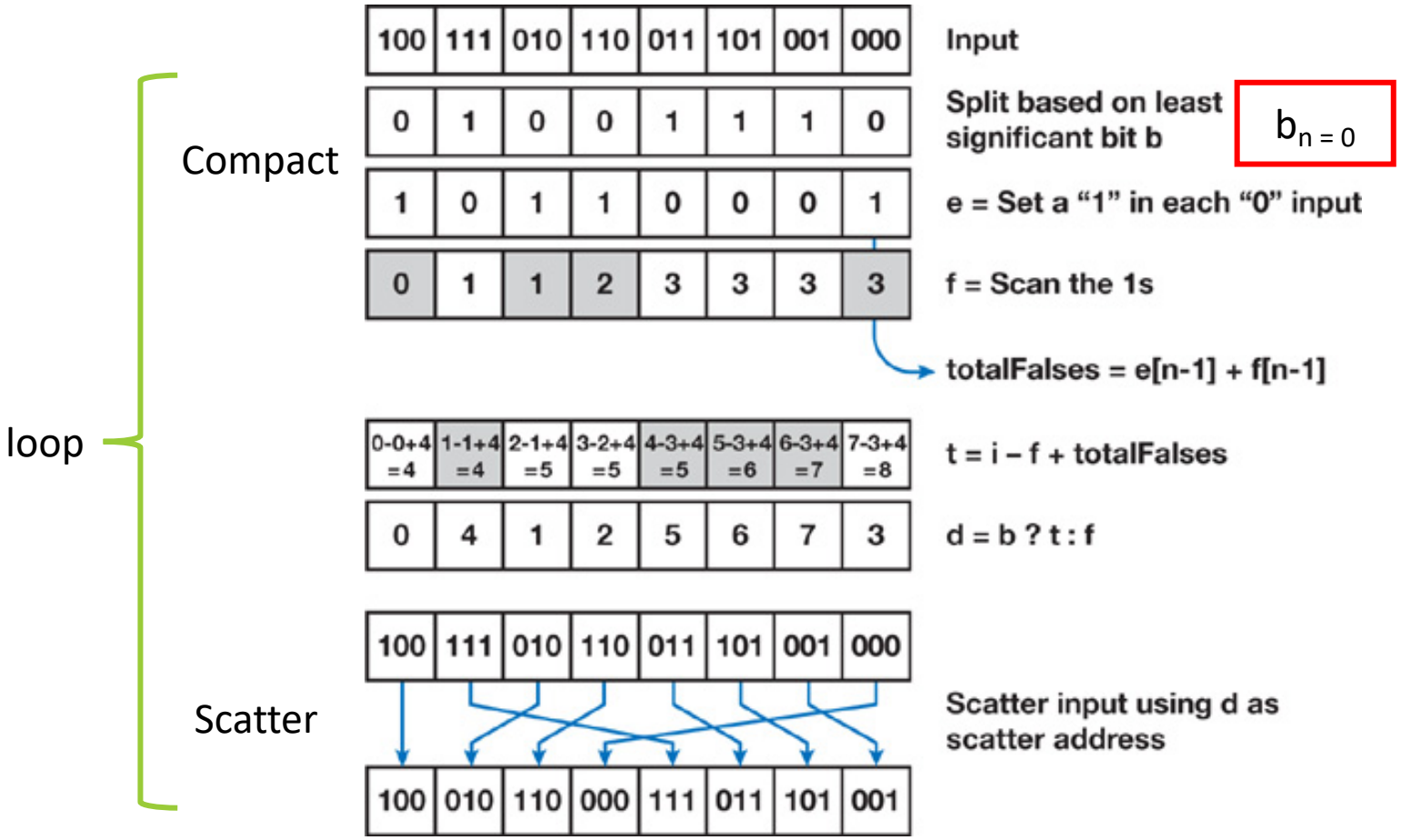
[A B C D E F G H] ➔ [A C D G]

- Predicate
- Scan
- Scatter





Radix Sort



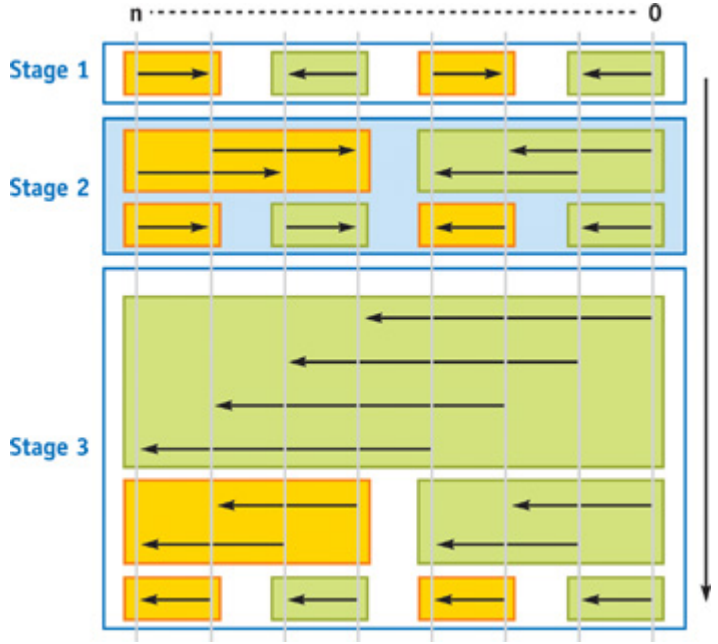
NVIDIA.

Bitonic Merge Sort

- Parallel processing을 효율적으로 활용하는 Merge Sort

- $O(n) = O(n \log^2(n) + \log(n))$
  - Quick Sort: O(n) = O(n log(n))

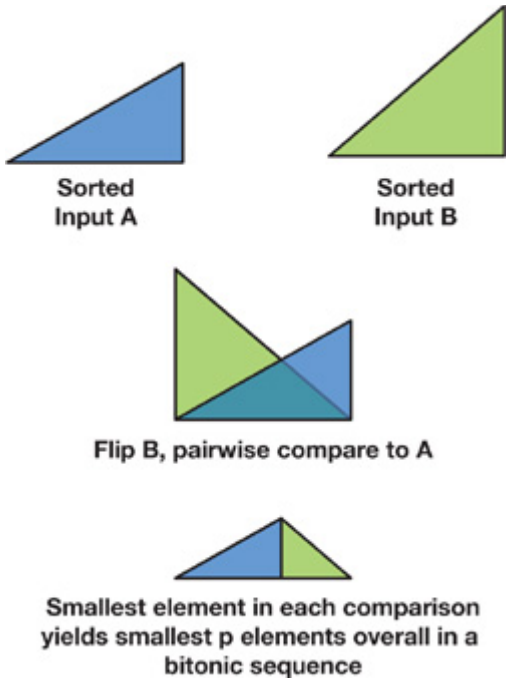
- Quick sort 보다 cost가 크지만,
  Stage 수만큼만의 시간만 필요하므로
  더욱 효과적



NVIDIA.

Merge Sort

- Radix Sort로 Thread Block 단위로 Sort된 block을 병렬로 Merge



NVIDIA.

Parallel Communication Pattern

- Selection
- Map
- Gather
- Stencil
- Scatter
- Partition
- Histogram
- Reduce
- Scan
- Sort

NVIDIA.

CUDA Libraries

https://developer.nvidia.com/gpu-accelerated-libraries

https://developer.nvidia.com/tools-ecosystem

CUDA Parallel Computing Platform

www.nvidia.com/getcuda

Programming Approaches

- **Libraries** — "Drop-in" Acceleration
- **OpenACC Directives** — Easily Accelerate Apps
- **Programming Languages** — Maximum Flexibility

Development Environment

Nsight IDE
Linux, Mac and Windows
GPU Debugging and Profiling

CUDA-GDB debugger
NVIDIA Visual Profiler

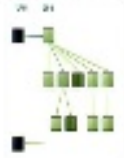
Open Compiler Tool Chain

Enables compiling new languages to CUDA platform, and CUDA languages to other architectures

Hardware Capabilities

SMX

Dynamic Parallelism



HyperQ

GPUDirect

© NVIDIA 2013

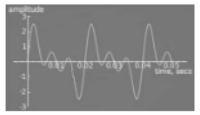
CUDA 가속 라이브러리

AmgX

A simple path to accelerated core solvers, providing up to 10x acceleration in the computationally intense linear solver portion of simulations, and is very well suited for implicit unstructured methods.

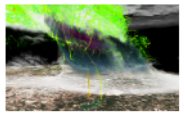
cuDNN

NVIDIA cuDNN is a GPU-accelerated library of primitives for deep neural networks, it is designed to be integrated into higher-level machine learning frameworks.



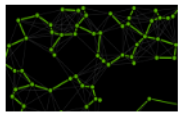
cuFFT

NVIDIA CUDA Fast Fourier Transform Library (cuFFT) provides a simple interface for computing FFTs up to 10x faster, without having to develop your own custom GPU FFT implementation.



IndeX Framework

NVIDIA IndeX Framework is a real-time scalable visualization plug-in for ParaView.



nvGRAPH

nvGRAPH Analytics Library is a GPU-accelerated graph analytics library.

GIE

NVIDIA GPU Inference Engine is a high performance neural network inference library for deep learning applications

NPP

NVIDIA Performance Primitives is a GPU accelerated library with a very large collection of 1000's of image processing primitives and signal processing primitives.

FFmpeg

FFmpeg is a popular open-source multi-media framework with a library of plugins that can be applied to various parts of the audio and video processing pipelines.

CHOLMOD

GPU-accelerated CHOLMOD is part of the SuiteSparse linear algebra package by Prof. Tim Davis. SuiteSparse is used extensively throughout industry and academia.

CULA Tools

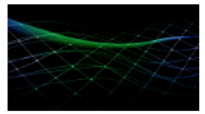
GPU-accelerated linear algebra library by EM Photonics, that utilizes CUDA to dramatically improve the computation speed of sophisticated mathematics.

MAGMA

A collection of next gen linear algebra routines. Designed for heterogeneous GPU-based architectures. Supports current LAPACK and BLAS standards.

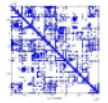
IMSL Fortran Numerical Library

Developed by RogueWave, a comprehensive set of mathematical and statistical functions that offloads work to GPUs.



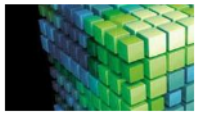
cuSOLVER

A collection of dense and sparse direct solvers which deliver significant acceleration for Computer Vision, CFD, Computational Chemistry, and Linear Optimization applications



cuSPARSE

NVIDIA CUDA Sparse (cuSPARSE) Matrix library provides a collection of basic linear algebra subroutines used for sparse matrices that delivers over 8x performance boost.

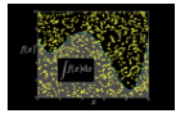


cuBLAS

NVIDIA CUDA BLAS Library (cuBLAS) is a GPU-accelerated version of the complete standard BLAS library that delivers 6x to 17x faster performance than the latest MKL BLAS.

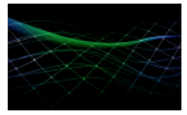
ArrayFire

Comprehensive, open source GPU function library. Includes functions for math, signal and image processing, statistics, and many more. Interfaces for C, C++, Java, R and Fortran.



cuRAND

The CUDA Random Number Generation library performs high quality GPU-accelerated random number generation (RNG) over 8x faster than typical CPU only code.



CUDA Math Library

An industry proven, highly accurate collection of standard mathematical functions, providing high performance on NVIDIA GPUs.

Thrust

A powerful, open source library of parallel algorithms and data structures. Perform GPU-accelerated sort, scan, transform, and reductions with just a few lines of code.

NVBIO

A GPU-accelerated C++ framework for High-Throughput Sequence Analysis for both short and long read alignment.

NVIDIA VIDEO CODEC SDK

Accelerate video compression with the NVIDIA Video Codec SDK. This SDK includes documentation and code samples that illustrate how to use NVIDIA's NVENC and NVDEC hardware in GPUs to accelerate encode, decode, and transcode of H.264 and HEVC video formats.

HiPLAR

HiPLAR (High Performance Linear Algebra in R) delivers high performance linear algebra (LA) routines for the R platform for statistical computing using the latest software libraries for heterogeneous architectures.

OpenCV

OpenCV is the leading open source library for computer vision, image processing and machine learning, and now features GPU acceleration for real-time operation.

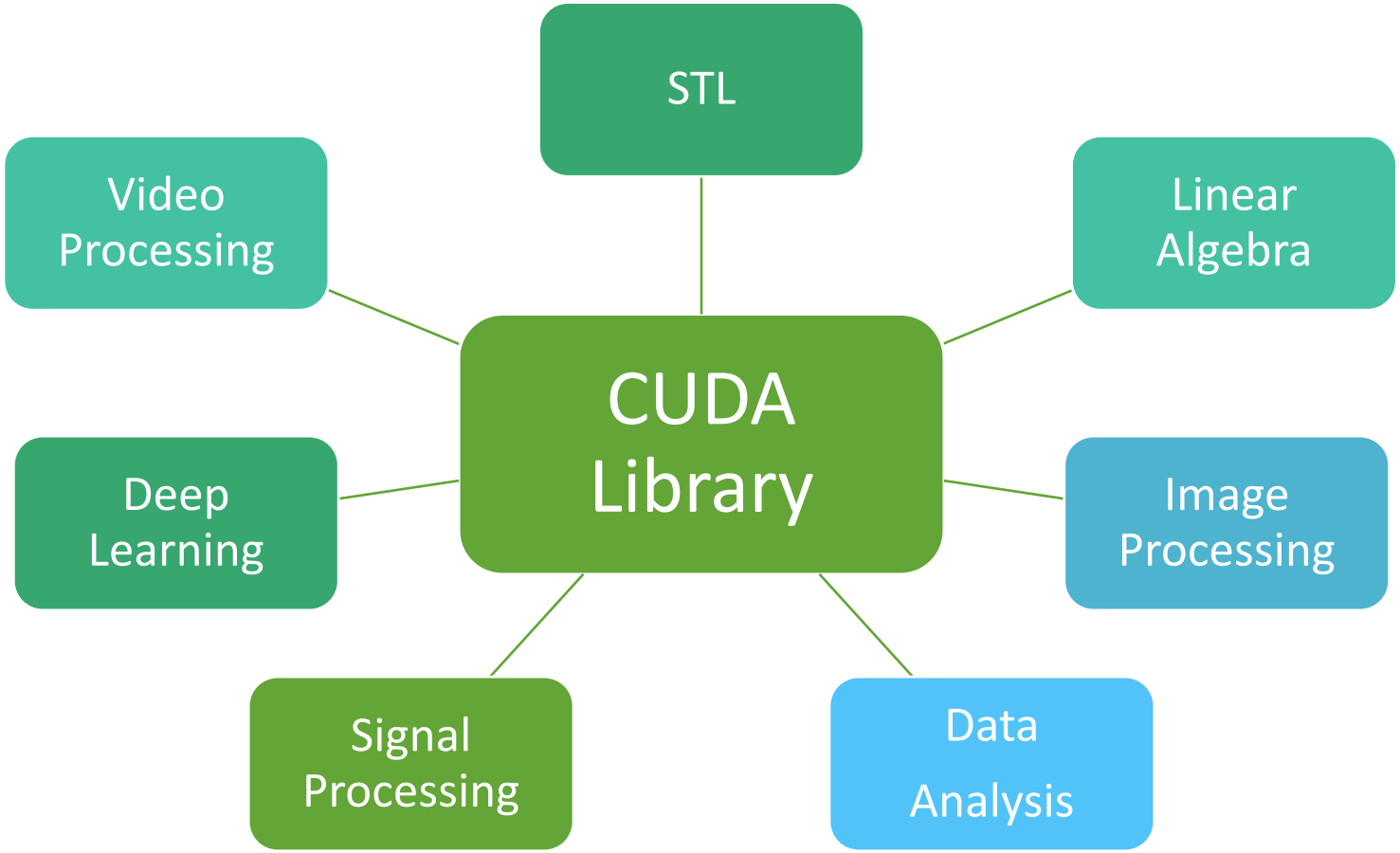
Geometry Performance Primitives(GPP)

GPP is a computational geometry engine that is optimized for GPU acceleration, and can be used in advanced Graphical Information Systems (GIS), Electronic Design Automation (EDA), computer vision, and motion planning solutions.

https://developer.nvidia.com/gpu-accelerated-libraries

NVIDIA.

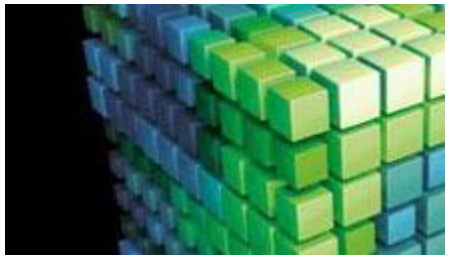
Categories



NVIDIA.

Example – vector & sort

```
#include <thrust/host_vector.h>
#include <thrust/device_vector.h>
#include <thrust/sort.h>
#include <cstdlib.h>
int main(void)
{
  // generate 32M random numbers on the host
  thrust::host_vector<int> h_vec(32 << 20);
  thrust::generate(h_vec.begin(), h_vec.end(), rand);
  // transfer data to the device
  thrust::device_vector<int> d_vec = h_vec;
  // sort data on the device (846M keys per sec on GeForce GTX 480)
  thrust::sort(d_vec.begin(), d_vec.end());
  // transfer data back to host
  thrust::copy(d_vec.begin(), d_vec.end(), h_vec.begin());
  return 0;
}
```



cuBlas Example

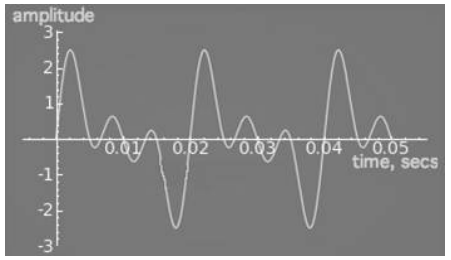
$$C = \alpha\ op(A)op(B) + \beta op(C)$$

```
cublasStatus_t cublasSgemm(cublasHandle_t handle,
                           cublasOperation_t transa, cublasOperation_t transb,
                           int m, int n, int k,
                           const float *alpha,
                           const float *A, int lda,
                           const float *B, int ldb,
                           const float *beta,
                           float *C, int ldc)
```

cublasOperation_t	
CUBLAS_OP_N	Normal matrix
CUBLAS_OP_T	Transposed Matrix
CUBLAS_OP_C	Conjucate Matrix

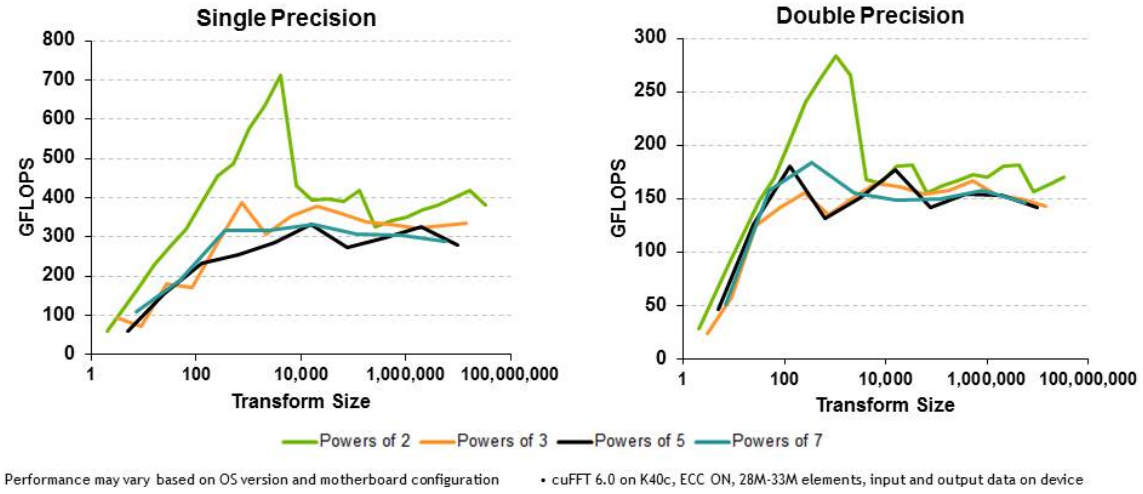
#gemm	General	Complex
Single Precision	SGEMM	CGEMM
Double Precision	DGEMM	ZGEMM

NVIDIA.



cuFFT

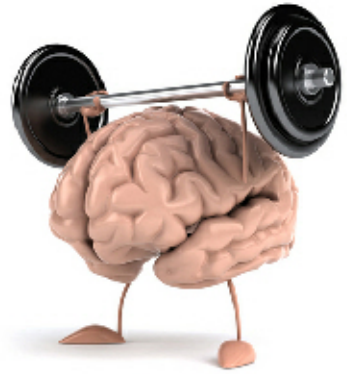
- CUDA Library for FFTs on the GPU
- Features:
  - 1D ~ 3D transforms for real & complex
  - Batch execution for multiple transform
  - Stream execution
- http://www.fftw.org/와 유사



https://developer.nvidia.com/cufft
http://docs.nvidia.com/cuda/cufft/

NVIDIA.

Fourier Transform ➔ FFT



Fourier Transform

$$\hat{u}(k) = \int_{-\infty}^{\infty} e^{-ikx} u(x) dx$$

Real → Wave

Discrete Fourier Transform

$$\hat{u}_k = \sum_{j=0}^{N-1} u_j e^{-\frac{2\pi i}{N} kj}$$

Real → Discrete, bounded

Inverse Fourier Transform

$$u(x) = \int_{-\infty}^{\infty} e^{ikx} \hat{u}(k) dk$$

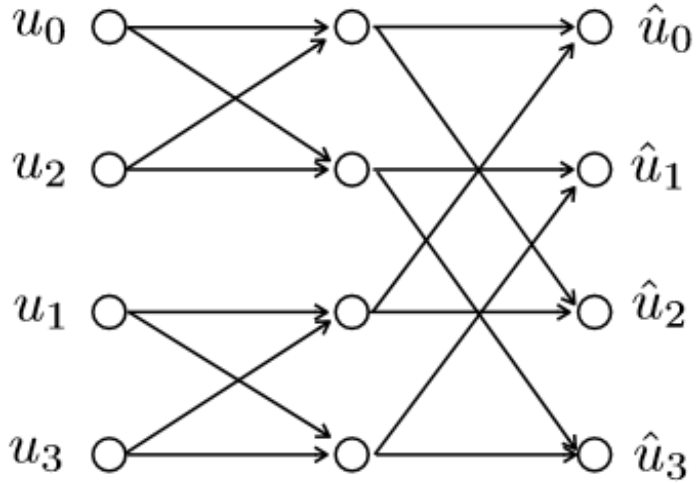
Wave → Real

Inverse Discrete Fourier Transform

$$u_j = \sum_{k=0}^{N-1} \hat{u}_k e^{\frac{2\pi i}{N} kj}$$

Wave → Bounded, discrete

FFT Butterfly



$$\hat{u}_0 = u_0 + u_2 e^0 + e^0(u_1 + u_3 e^0)$$
$$\hat{u}_1 = u_0 + u_2 e^{-\pi i} + e^{-\frac{\pi}{2} i}(u_1 + u_3 e^{-\pi i})$$
$$\hat{u}_2 = u_0 + u_2 e^{-2\pi i} + e^{-\pi i}(u_1 + u_3 e^{-2\pi i})$$
$$\hat{u}_3 = u_0 + u_2 e^{-3\pi i} + e^{-3\frac{\pi}{2} i}(u_1 + u_3 e^{-3\pi i})$$

Fast Fourier Transform (FFT) ⇨ $O(N^2) \rightarrow O(N \log(N))$

NVIDIA.

cuFFT Example

```
#define NX 64
#define NY 64
#define NZ 128

cufftHandle plan;
cufftComplex *data1, *data2;

cudaMalloc((void**)&data1, sizeof(cufftComplex)*NX*NY*NZ);
cudaMalloc((void**)&data2, sizeof(cufftComplex)*NX*NY*NZ);

/* Create a 3D FFT plan. */
cufftPlan3d(&plan, NX, NY, NZ, CUFFT_C2C);
/* Transform the first signal in place. */
cufftExecC2C(plan, data1, data1, CUFFT_FORWARD);
/* Transform the second signal using the same plan. */
cufftExecC2C(plan, data2, data2, CUFFT_FORWARD);
/* Destroy the cuFFT plan. */
cufftDestroy(plan);

cudaFree(data1); cudaFree(data2);
```

GPGPU Technologies

Unified Virtual Address / Unified Memory

Multi-GPU

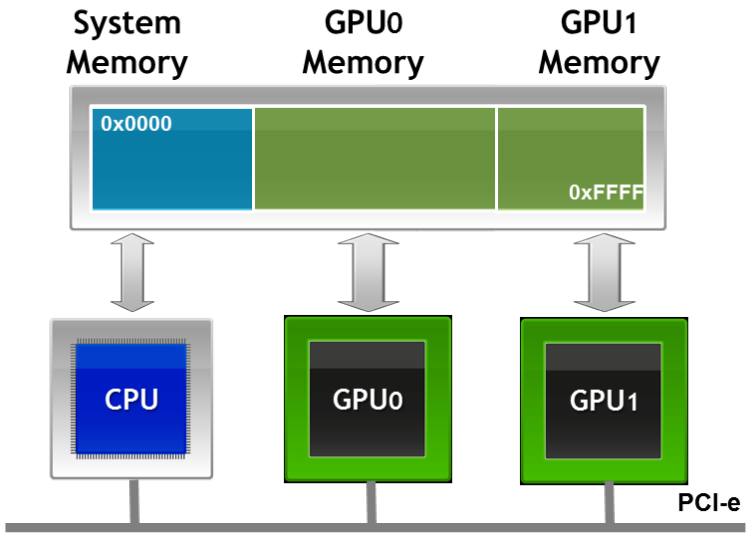
GPU Direct

MPI + CUDA

Dynamic Parallelism

Unified Virtual Memory Code

- 64bit OS/Application Only
- Address만으로 Host/Device 구분



GPU 메모리 주소 구분

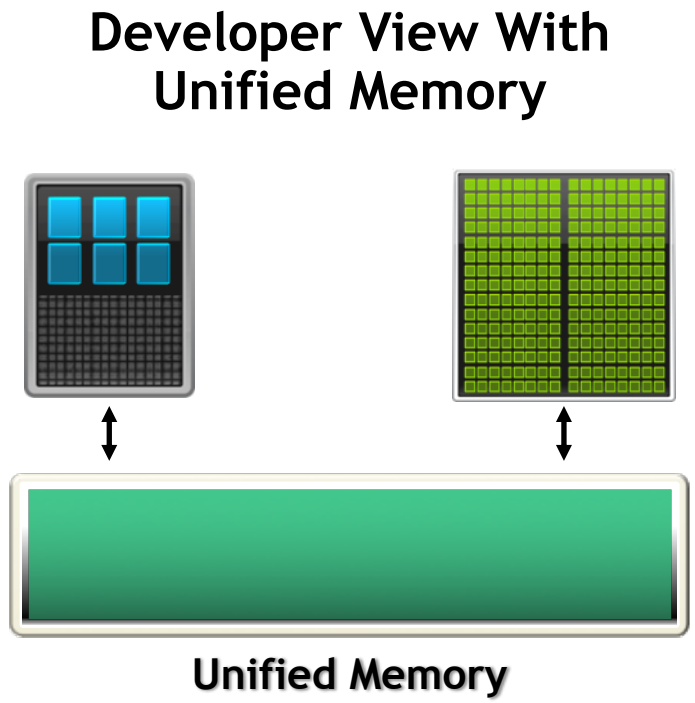
```
void *d_mem;
cudaMalloc(&d_mem, bufsize);

cudaMemcpy(d_mem, h_mem, bufsize);

d_foo<<<gridDim, gridBlock>>>(d_mem);

cudaFree(d_mem);
```

```
void *d_mem;
cudaMallocManaged(&d_mem, bufsize);

fread(d_mem, 1, bufsize, fp);

d_foo<<<gridDim, gridBlock>>>(d_mem);
cudaDeviceSynchronize();

cudaFree(d_mem);
```


NVIDIA.

Unified Memory

- **CUDA 8.0** 지원



시스템 주소공간 공유

```
void *d_mem;
cudaMalloc(&d_mem, bufsize);

cudaMemcpy(d_mem, h_mem, bufsize);

d_foo<<<gridDim, gridBlock>>>(d_mem);

cudaFree(d_mem);
```

```
void *d_mem;
d_mem = malloc(bufsize);

fread(d_mem, 1, bufsize, fp);

d_foo<<<gridDim, gridBlock>>>(d_mem);
cudaDeviceSynchronize();

cudaFree(d_mem);
```

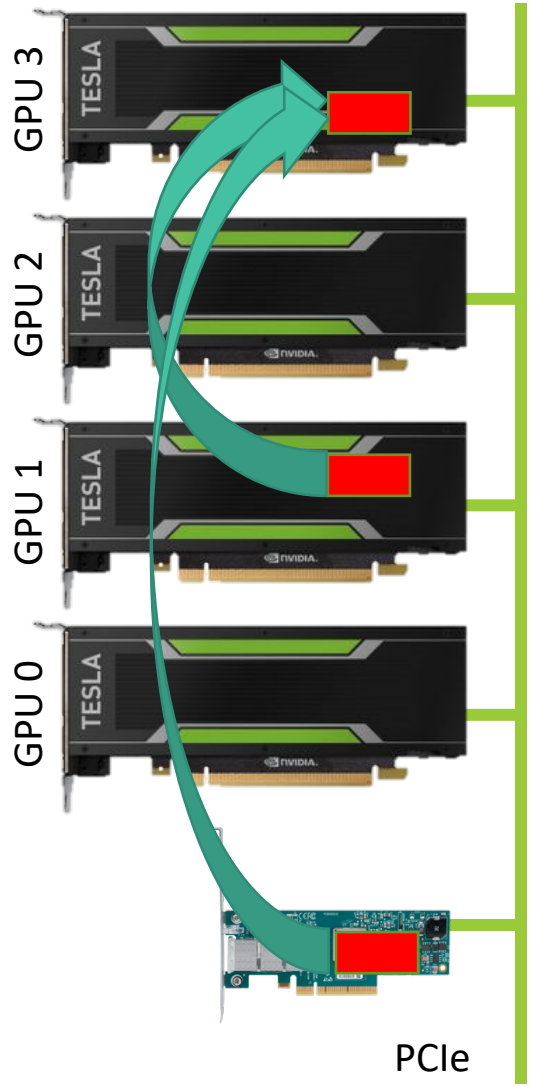
Multiple GPU

- GPU Select
  - cudaDeviceCount(int* count)
  - cudaGetDevice(int* devID)
  - cudaSetDevice(int devID)
- CUDA Memcpy
  - cudaMemcpy/Async()
- CUDA Peer-to-Peer (only Tesla/Quadro)
  - cudaDeviceEnablePeerAccess(int targetID, 0)
  - cudaMemcpyPeer/Async
    (void* dst, int dstID, void* src, int srcID, size_t count)

NVIDIA.

GPU Direct

- Peer-to-Peer memory copy
- Peer Access
- RDMA
- Sharing Pinned Memory



Peer-to-peer operation

- Peer-to-Peer memory copy

```
cudaMemcpyPeer ( void * dst, int dstDevice, const void * src, int srcDevice, size_t count )
```

  - Unified (Virtual) Memory를 사용하는 경우

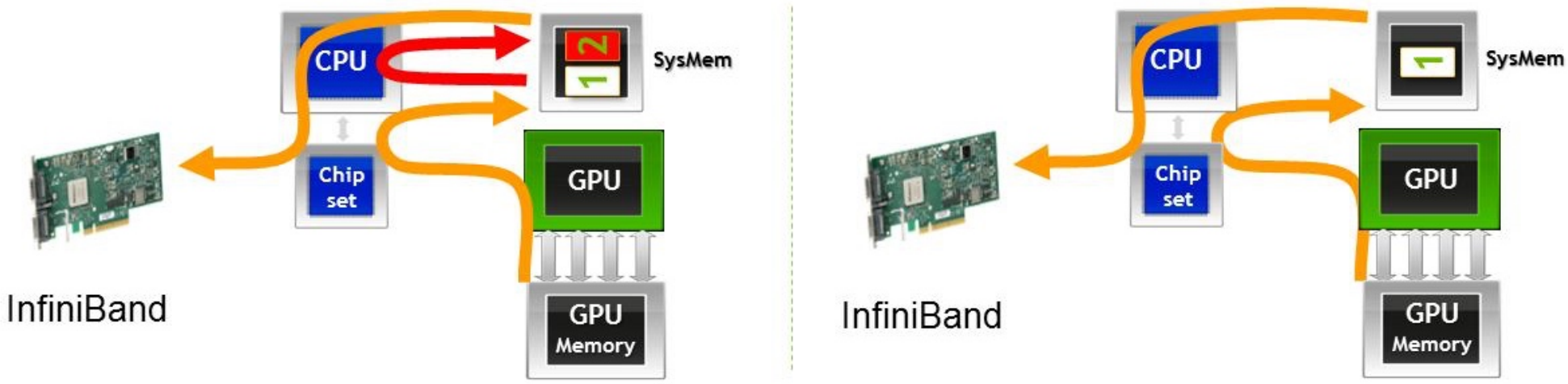
```
cudaMemcpy ( void * dst, const void * src, size_t count, cudaMemcpyKind kind )
```

- Peer Access
  - cudaDeviceEnablePeerAccess(int deviceID, 0)
  - 다른 GPU에서 PeerAcceess를 활성화한 GPU의 메모리 접근 가능
  - Tesla, Quadro Only

NVIDIA.

Sharing Pinned Memory

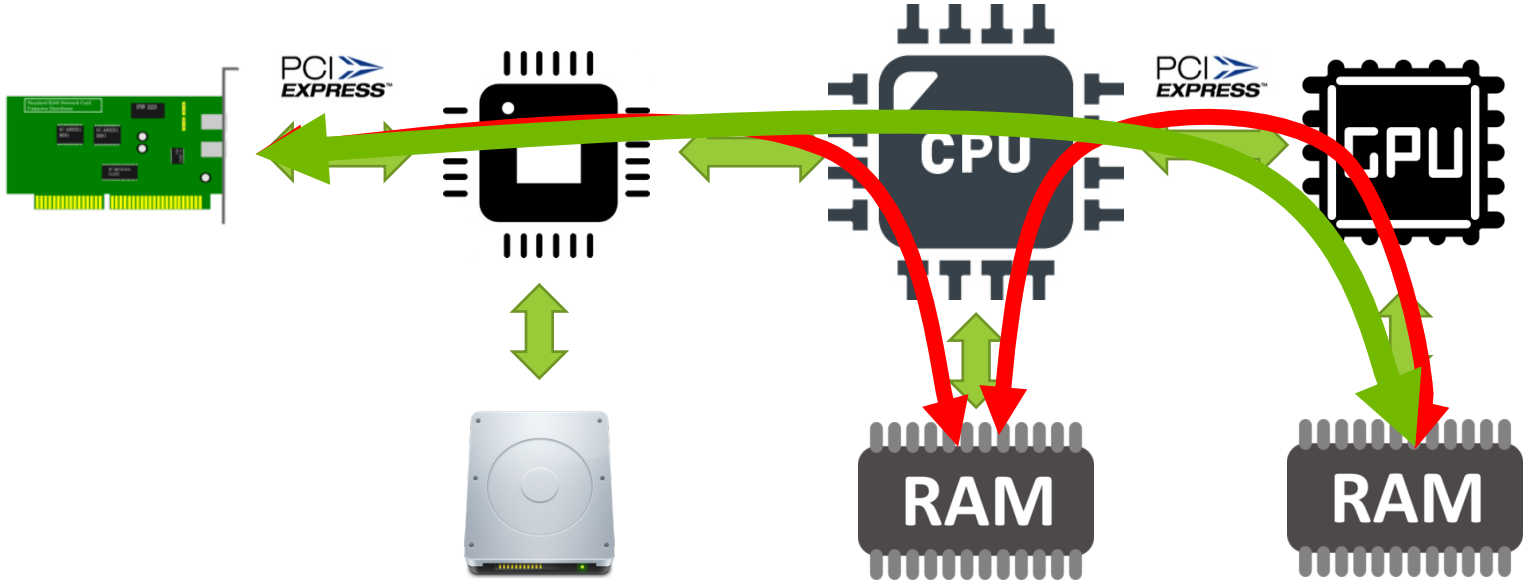
- 다른 장치의 Pinned Memory (DMA enabled memory)와 공유 가능
  - 메모리 공간 확보 및 복사에 따른 지연시간 감소



NVIDIA.

RDMA

- 3rd party DMA 장비간 데이터 전송을 효율화 하는 방법
- Quadro / Tesla만 지원

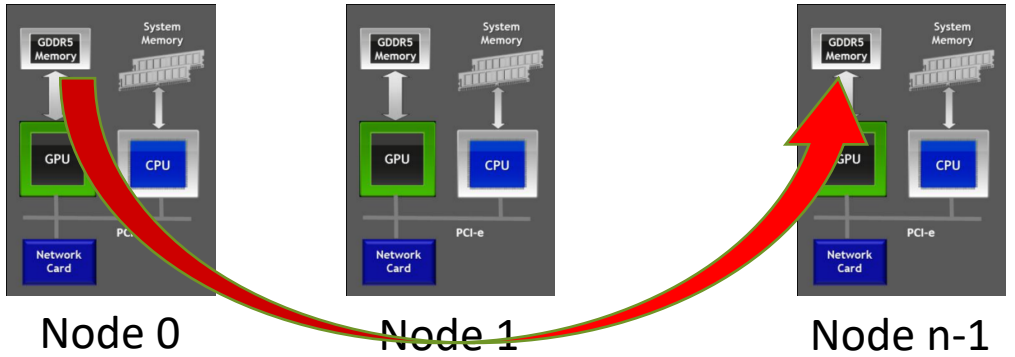


NVIDIA.

MPI + CUDA

- MPI
  - Process간 Message를 주고받기 위한 interface
  - 다양한 언어로 개발 되어 있음
  - CUDA aware MPI를 사용하면 GPUDirect 기능 활용 가능
    - MVAPICH2, OpenMPI, Cray MPI, IBM MPI

- 실행 예
  - MPI_Send(s_buf_d,size,MPI_CHAR,n-1,tag,MPI_COMM_WORLD); (rank 0)
  - MPI_Recv(r_buf_d,size,MPI_CHAR,0,tag,MPI_COMM_WORLD,&stat); (rank n-1)



UVA + CUDA-aware MPI

UVA & CUDA aware MPI

```
// MPI rank 0
MPI_Send(buf_d, size, …);

// MPI rank n-1
MPI_Recv(buf_d, size, …);
MPI_Barrier(MPI_COMM_WORLD);
```

No UVA & regular MPI

```
// MPI rank 0
cudaMemcpy(buf_h, buf_d, size, …);
MPI_Send(buf_h, size, …);

// MPI rank 1
MPI_Recv(buf_h, seize, …);
MPI_Barrier(MPI_COMM_WORLD);
cudaMemcpy(buf_d, buf_h, size, …);
```

MPI + CUDA

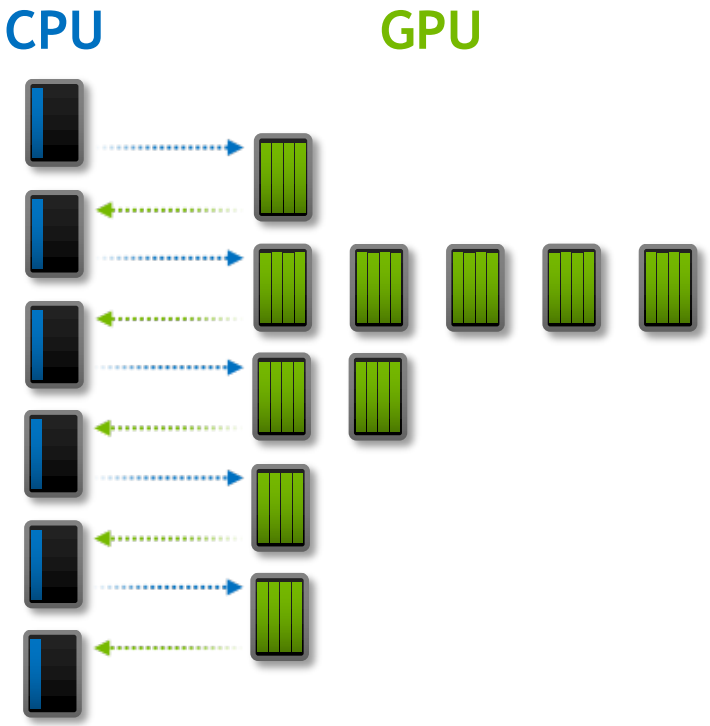
- MPI Execution

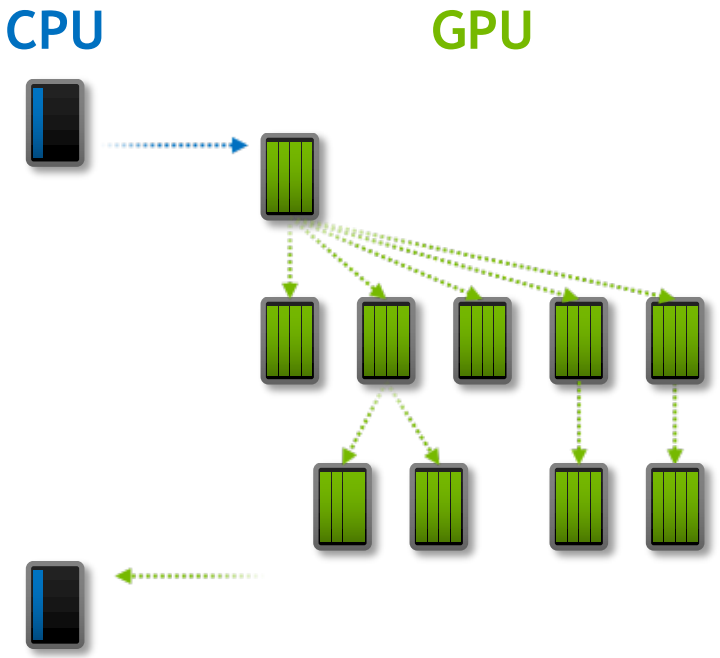
```
mpirun –np 4 ./myapp <args>
```

```
#include <mpi.h>
int main(int argc, char *argv[]) {
        int myrank;
        /* Initialize the MPI library */
        MPI_Init(&argc,&argv);
        /* Determine the calling process rank */
        MPI_Comm_rank(MPI_COMM_WORLD,&myid);
        /* Call MPI routines like MPI_Send, MPI_Recv, ... */
        /* Shutdown MPI library */
        MPI_Finalize();
        return 0;
}
```


NVIDIA.

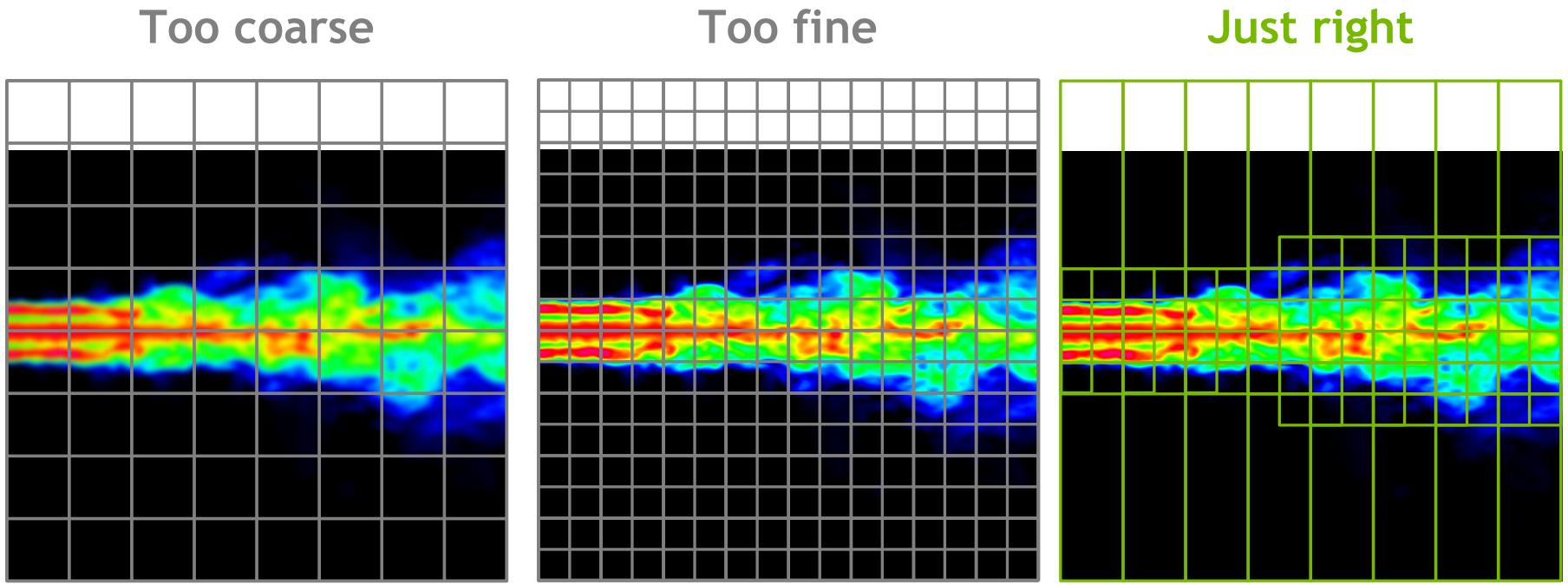
Dynamic Parallelism





NVIDIA.

적절한 단위로 병렬화 가능



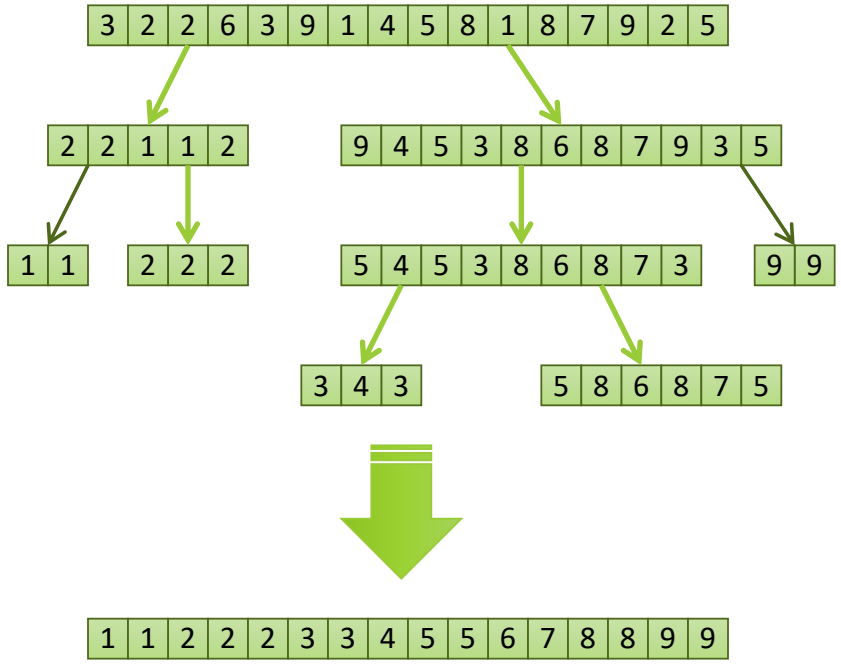
NVIDIA.

Programming Model Example

- CUDA Runtime syntax & semantics
- Launch is per-thread
- Sync includes all launches by any thread in the block
- *cudaDeviceSynchronize()* does not imply syncthreads
- Asynchronous launches only (note bug in program, here!)

```
__device__ float buf[1024];
__global__ void cnp(float *data)
{
   int tid = threadIdx.x;
   if(tid % 2)
      buf[tid/2] = data[tid]+data[tid+1];
   __syncthreads();

   if(tid == 0) {
      launch<<< 128, 256 >>>(buf);
      cudaDeviceSynchronize();
   }
   __syncthreads();

   cudaMemcpyAsync(data, buf, 1024);
   cudaDeviceSynchronize();
}
```

Example: Parallel Recursion



- Divide & conquer 알고리즘
- 재귀적으로 partition & Sort
- 데이터에 따라 동작이 변함

NVIDIA.