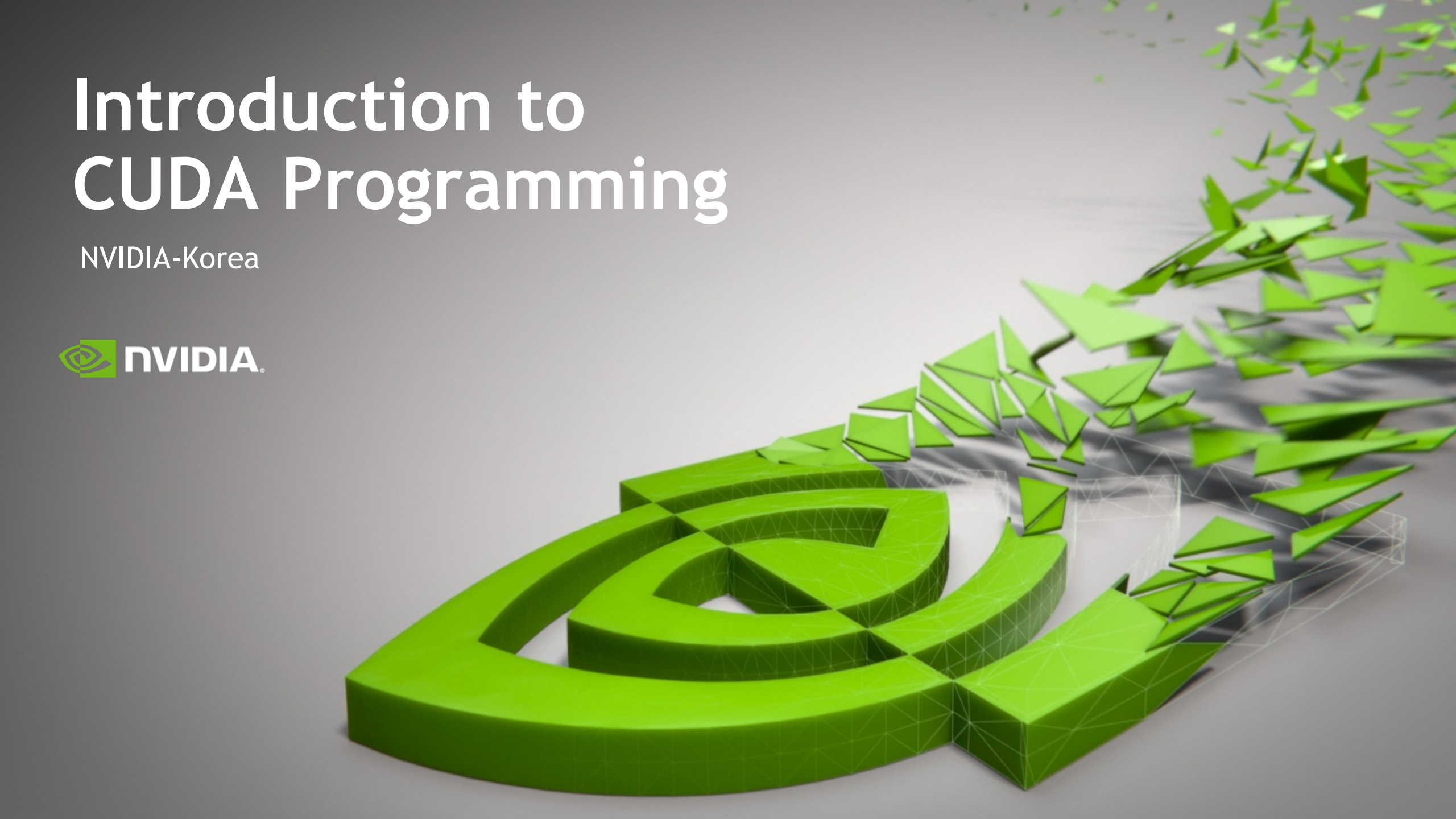
Introduction to CUDA Programming

NVIDIA-Korea



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

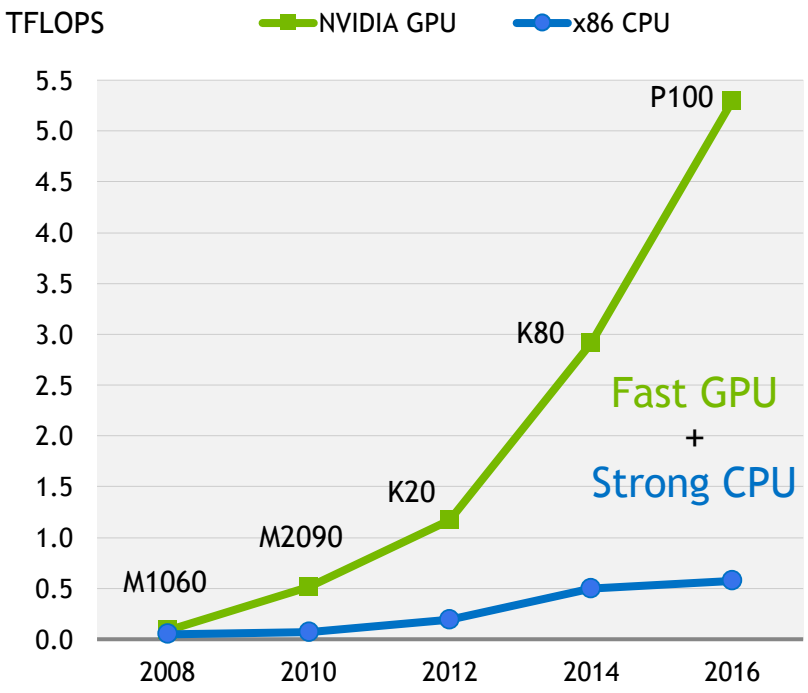
GPGPU 프로그래밍 개요

GPU COMPUTING

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

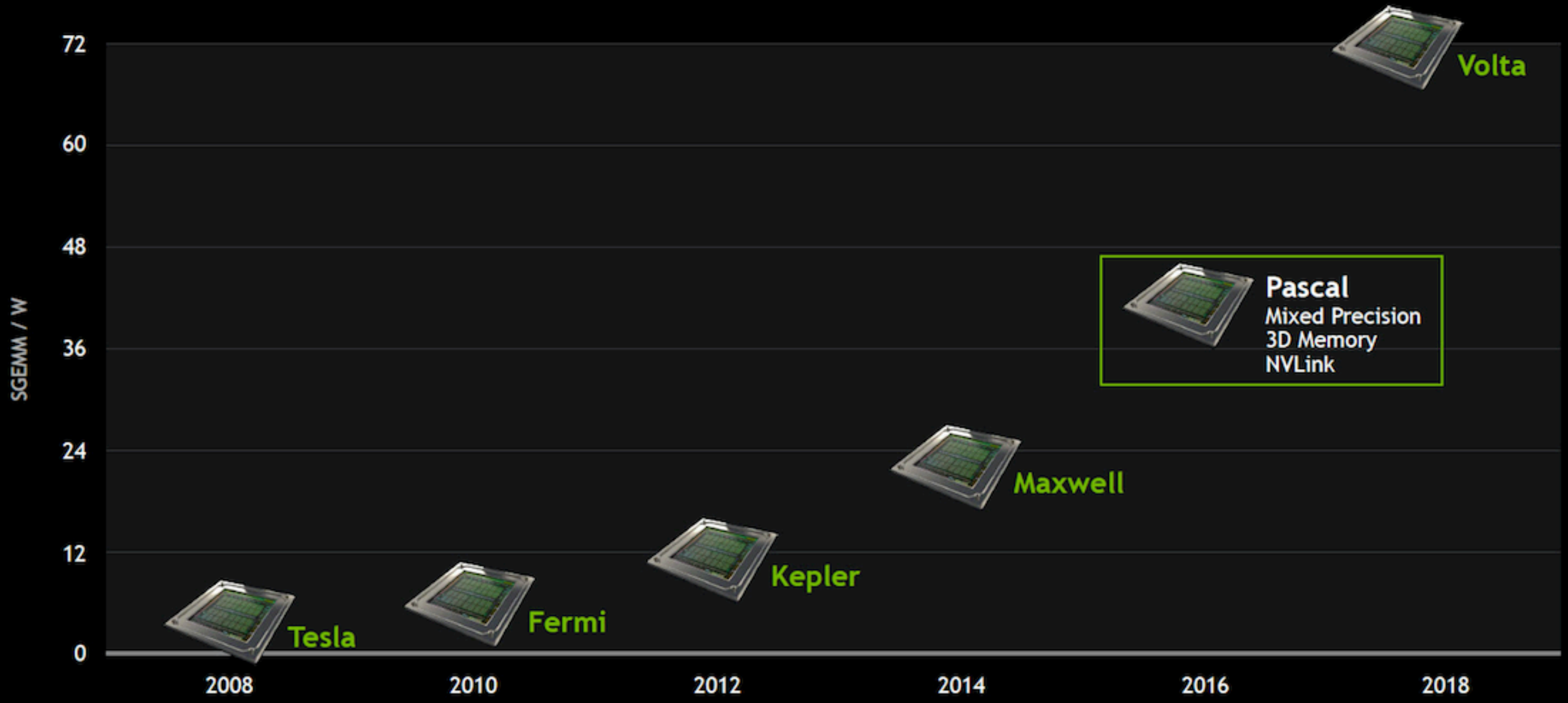
Accessibility

阿里云计算 Alibaba Cloud Computing

amazon web services™ IBM SOFTLAYER

Microsoft Azure

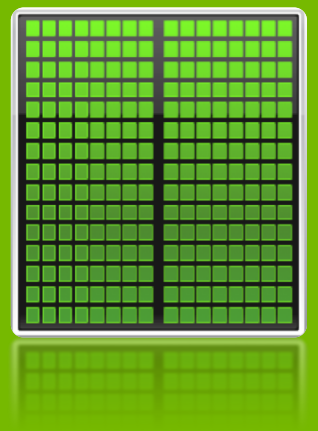
GPU ROADMAP


72
60
48
36
24
12
0
SGEMM / W
2008
2010
2012
2014
2016
2018
Tesla
Fermi
Kepler
Maxwell
Pascal
Mixed Precision
3D Memory
NVLink
Volta

“ It’s time to start planning for the end of Moore’s Law, and it’s worth pondering how it will end, not just when.”

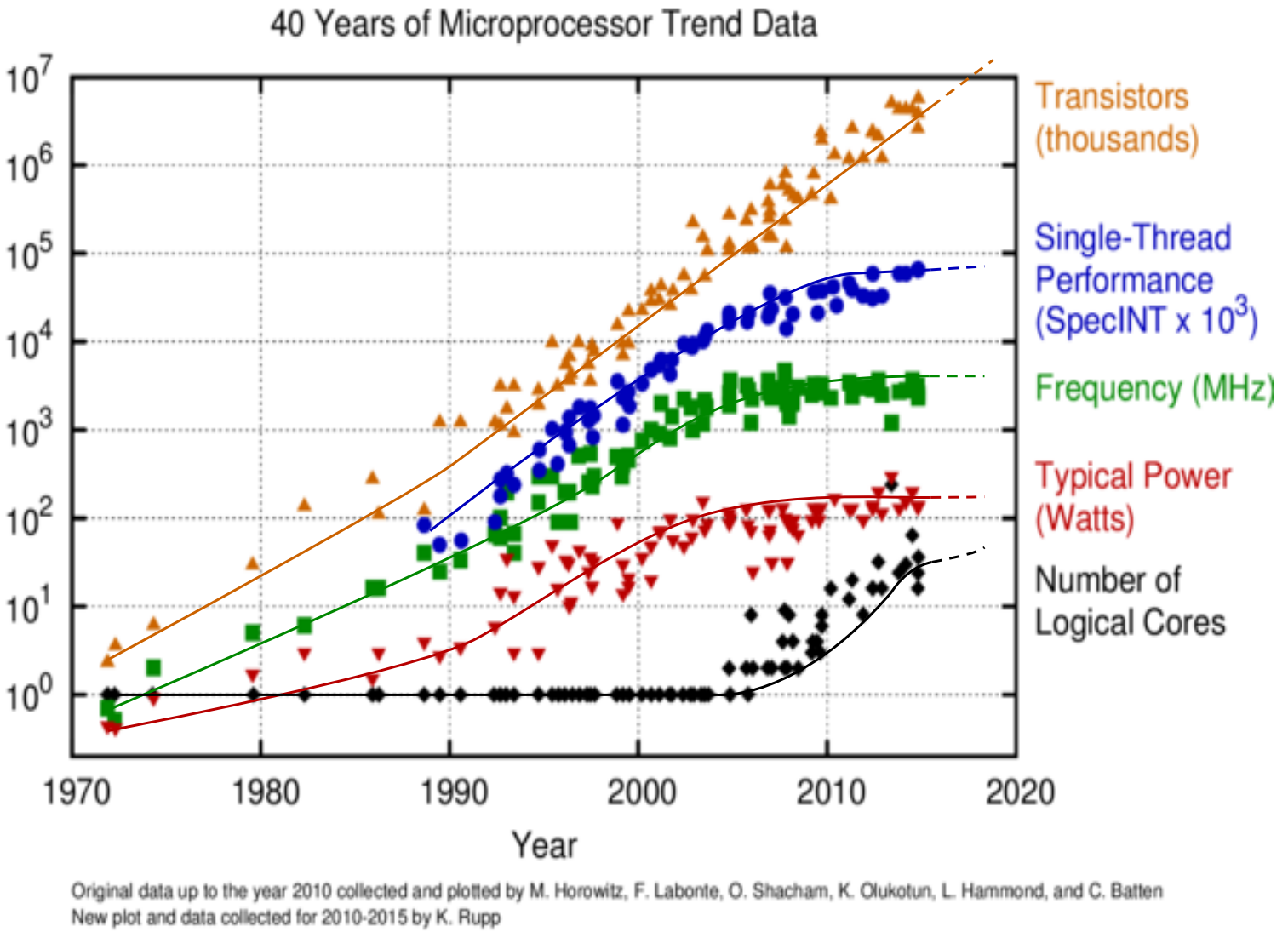
Robert Colwell
Director, Microsystems Technology Office, DARPA

GPU Accelerator



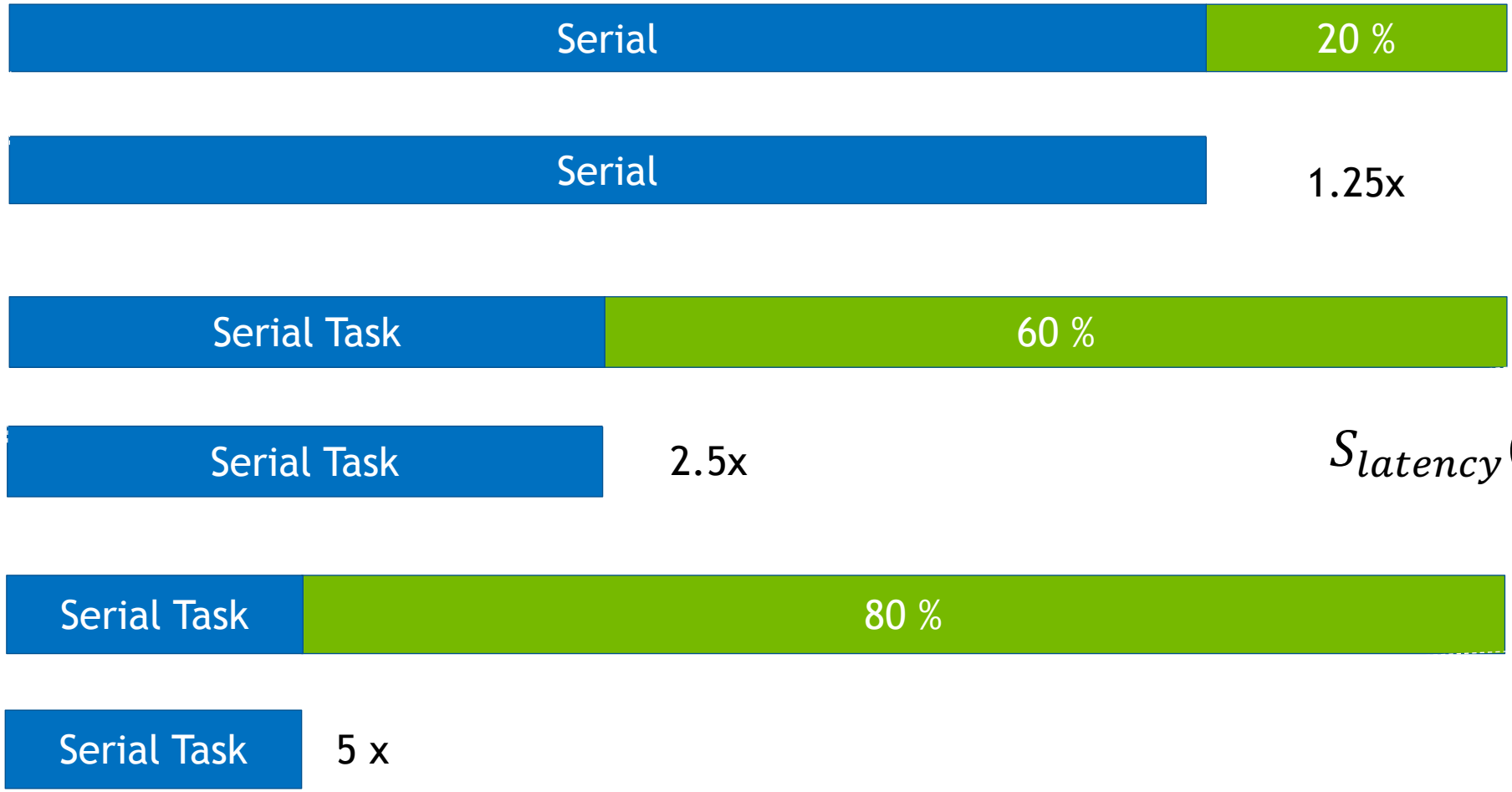
+

CPU



Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2015 by K. Rupp

AMDAHL'S LAW



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

CUDA

Common Unified Device Architecture

CUDA란?

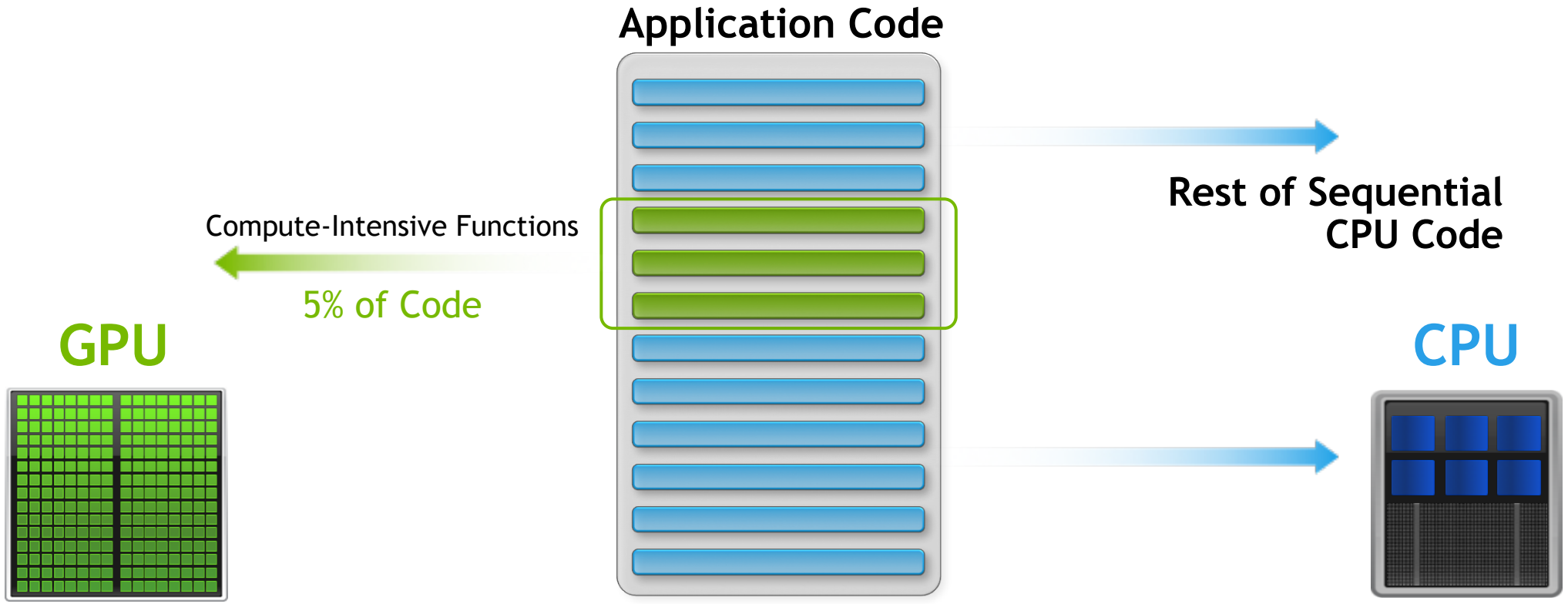
NVIDIA GPGPU 프로그래밍 플랫폼

C 언어를 기반으로 한 GPU 프로그래밍 가능

다양한 언어 및 라이브러리와 함께 사용가능

CUDA 가속 지원 라이브러리와 연동하여 다양한 어플리케이션 개발 가능

APPLICATION의 CUDA 가속 절차



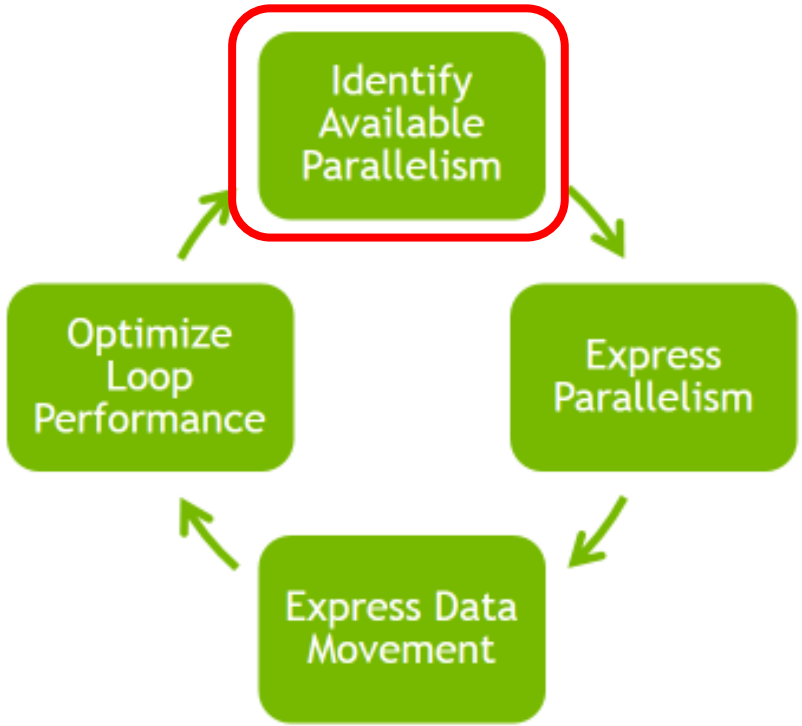
NVIDIA.

APPLICATION의 CUDA 가속 절차

성능 분석

병목 탐색

알고리즘 분석 및 병렬화 전략 수립



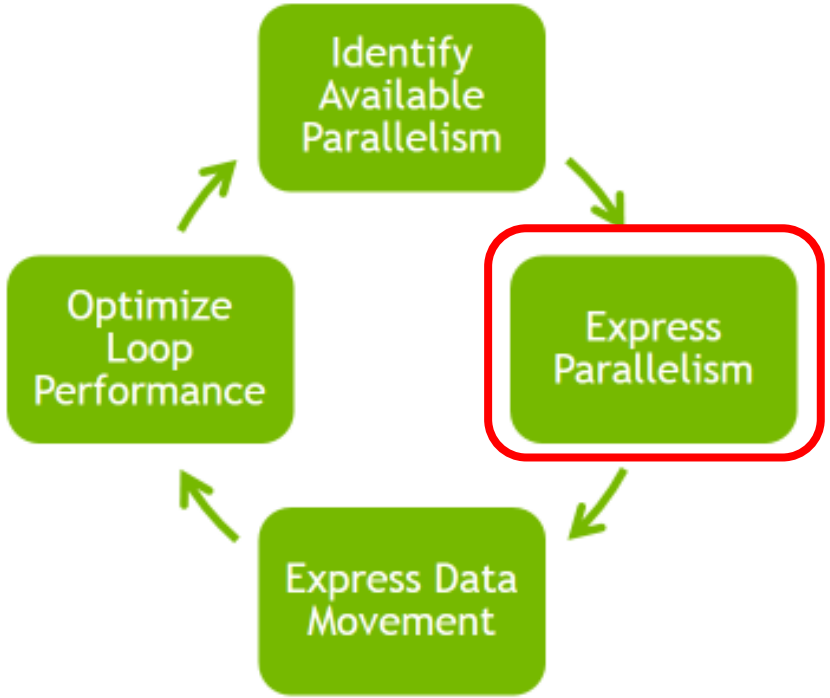
APPLICATION의 CUDA 가속 절차

성능 분석

병목 탐색

알고리즘 분석 및 병렬화 전략 수립

데이터 의존성 해결 및 병렬화



APPLICATION의 CUDA 가속 절차

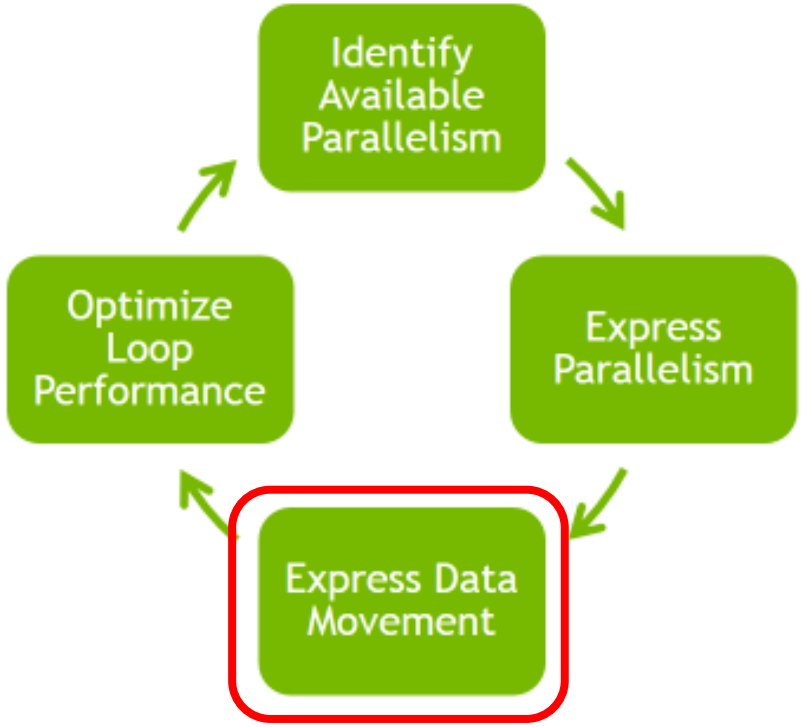
성능 분석

병목 탐색

알고리즘 분석 및 병렬화 전략 수립

데이터 의존성 해결

데이터 전송 최적화



APPLICATION의 CUDA 가속 절차

성능 분석

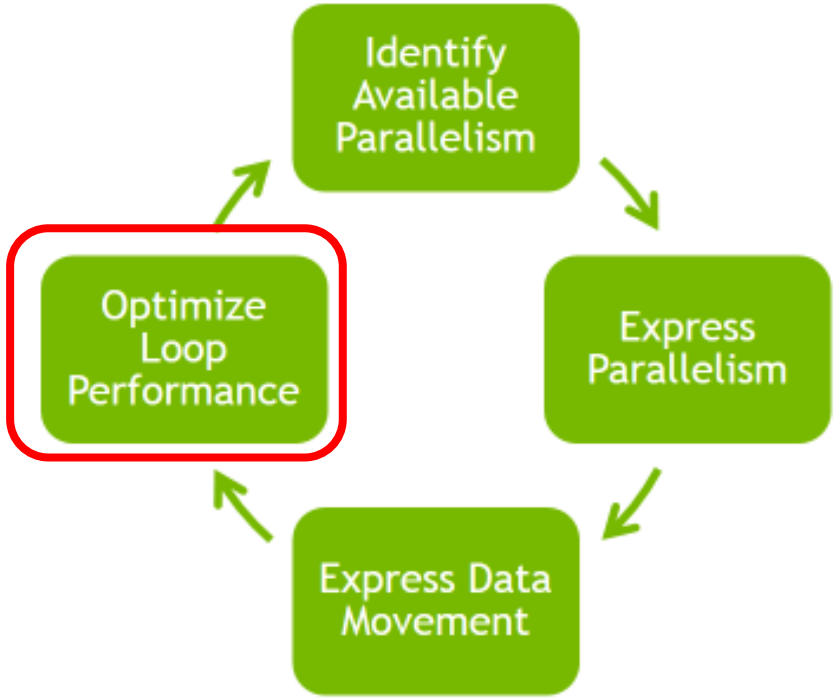
병목 탐색

알고리즘 분석 및 병렬화 전략 수립

데이터 의존성 해결

데이터 전송 최적화

Kernel 레벨의 최적화



APPLICATION의 CUDA 가속 절차

성능 분석

병목 탐색

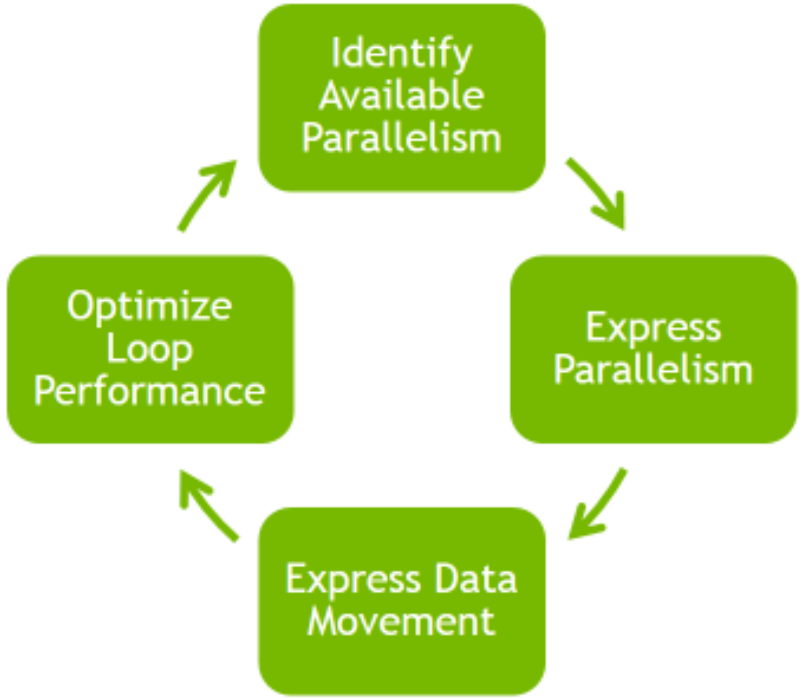
알고리즘 분석 및 병렬화 전략 수립

데이터 의존성 해결

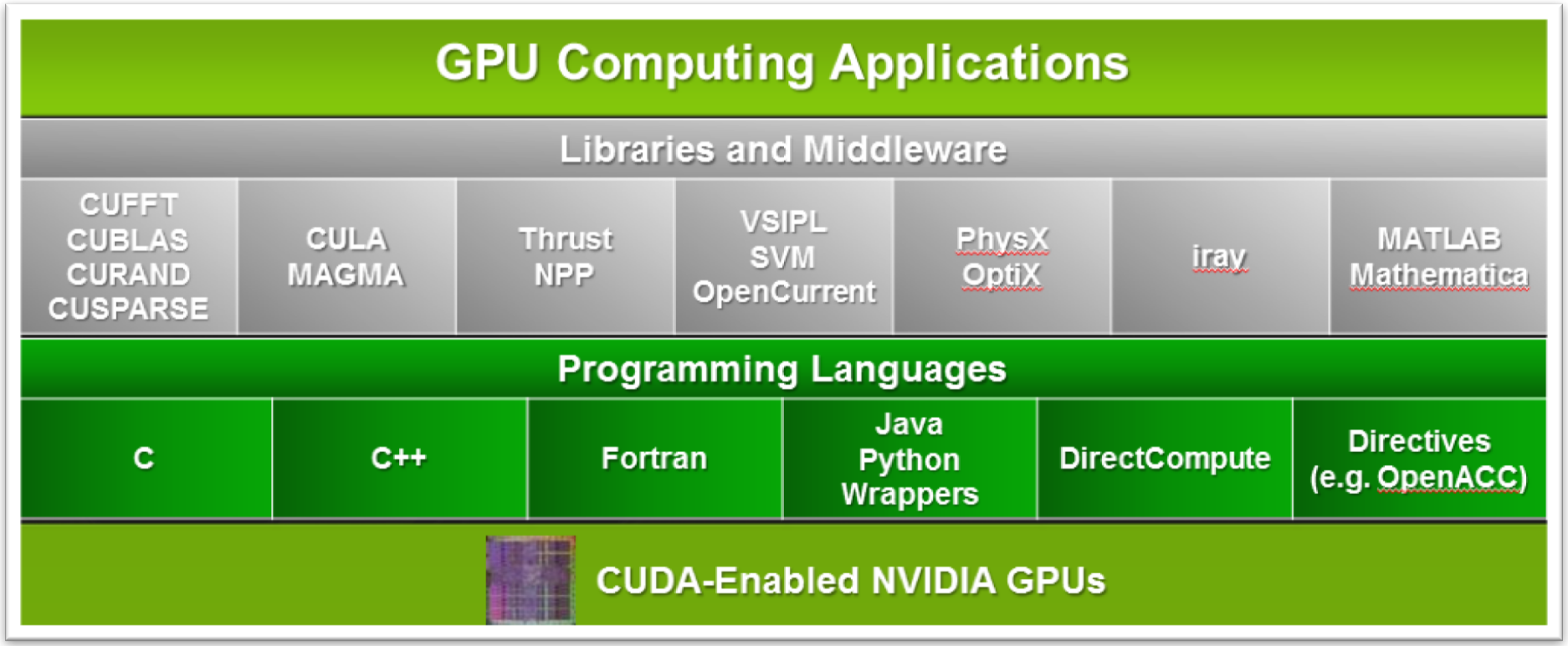
데이터 전송 최적화

Kernel 레벨의 최적화

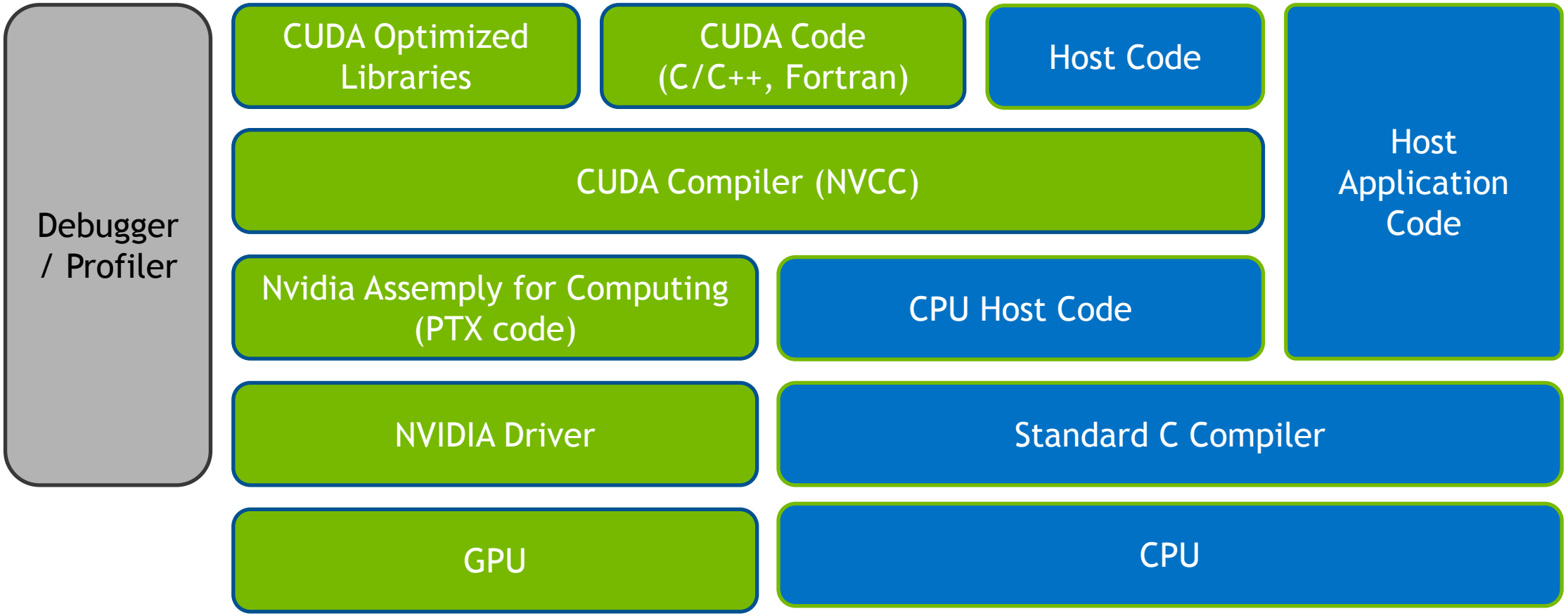
목표성능 달성



CUDA = NVIDA GPU 컴퓨팅



GPU COMPILTER: NVCC



NVIDIA GPGPU COMPUTING EVERYWHERE, EVERY PLATFORM

TITAN X
For every desktops

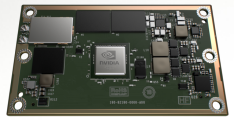
DGX-1
The HPC appliance for instant productivity

TESLA
Servers in every shape and size

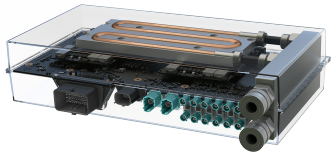
CISCO · CRAY · DELL · hp · IBM · Lenovo · Quanta Computer

CLOUD
Everywhere

Alibaba.com · amazon · Bai百度 · Google · IBM · Microsoft



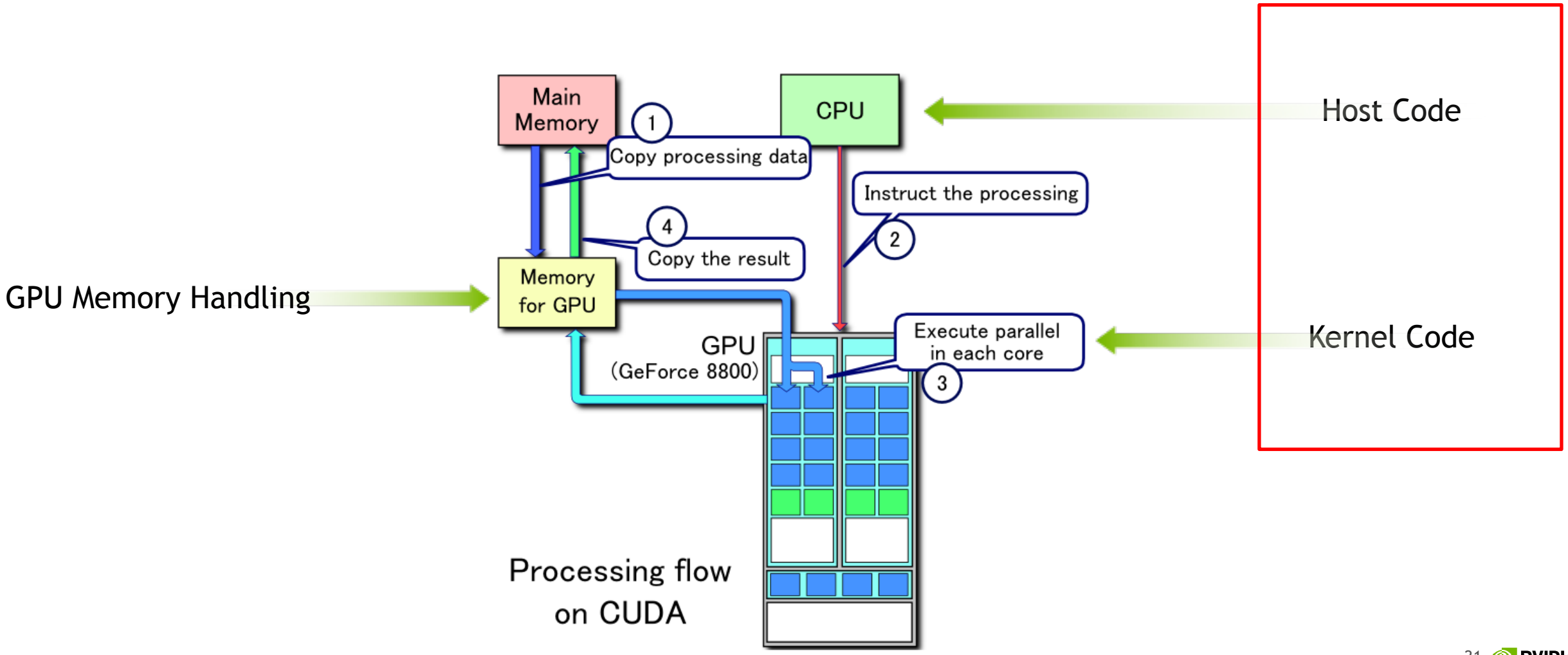
JETSON
For embedded Devices



Drive PX
Autonomous Car

CUDA PROGRAMMING MODEL

CUDA 실행 절차


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

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);
```

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
- Argument 중 Pointer는 반드시 device memory를 가리켜야 함

GPU 코드 키워드

Keyword를 이용하여 NVCC 컴파일러로 하여금 code block이 호출하는 위치와 target을 알려줌

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

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

KERNEL 호출

```
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

HELLO WORLD

cpu.c

```
#include <stdio.h>

void message_kernel() {
    printf("Hello World!\n");
}

int main() {
    message_kernel();
    return 0;
}
```

```
[~/workspace/cuda/hello_world]$ gcc -o cpu.out cpu.c
[~/workspace/cuda/hello_world]$ ./cpu.out
Hello World!
```

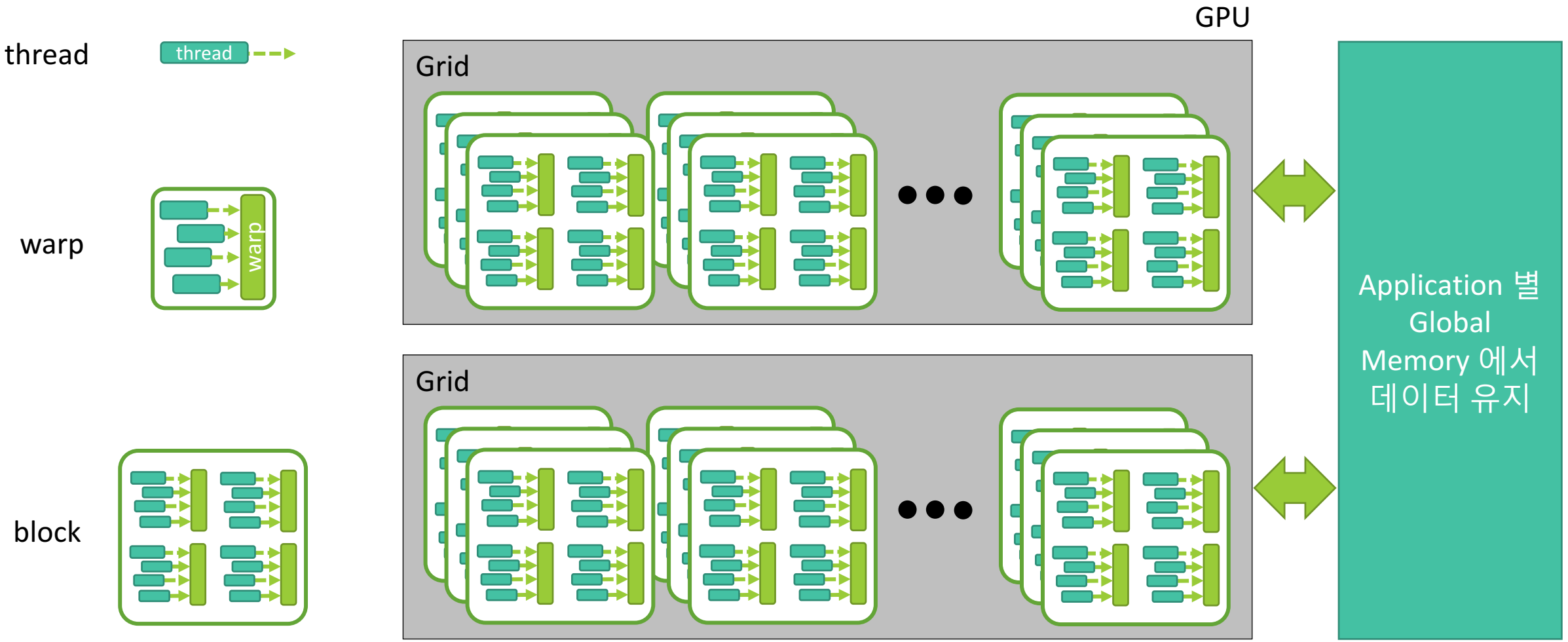
HELLO WORLD

gpu.cu

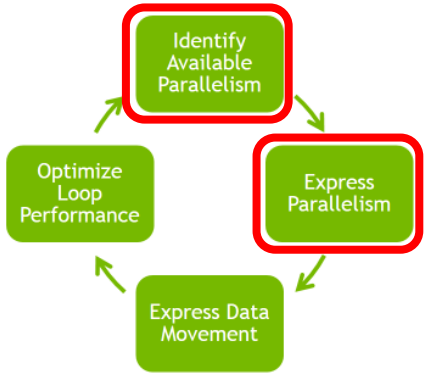
```
#include <stdio.h>

__global__ void message_kernel()
{
    printf("Hello World!\n");
}

int main() {
    message_kernel<<<1, 10>>>();
    cudaDeviceSynchronize();
    return 0;
}
```

```
[~/workspace/cuda/hello_world]$ nvcc -o gpu.out
 -gencode arch=compute_30,code=sm_30 gpu.cu
[~/workspace/cuda/hello_world]$ ./gpu.out

Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
```

2. 계층적인 실행 제어 구조



SAXPY EXAMPLE



C

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

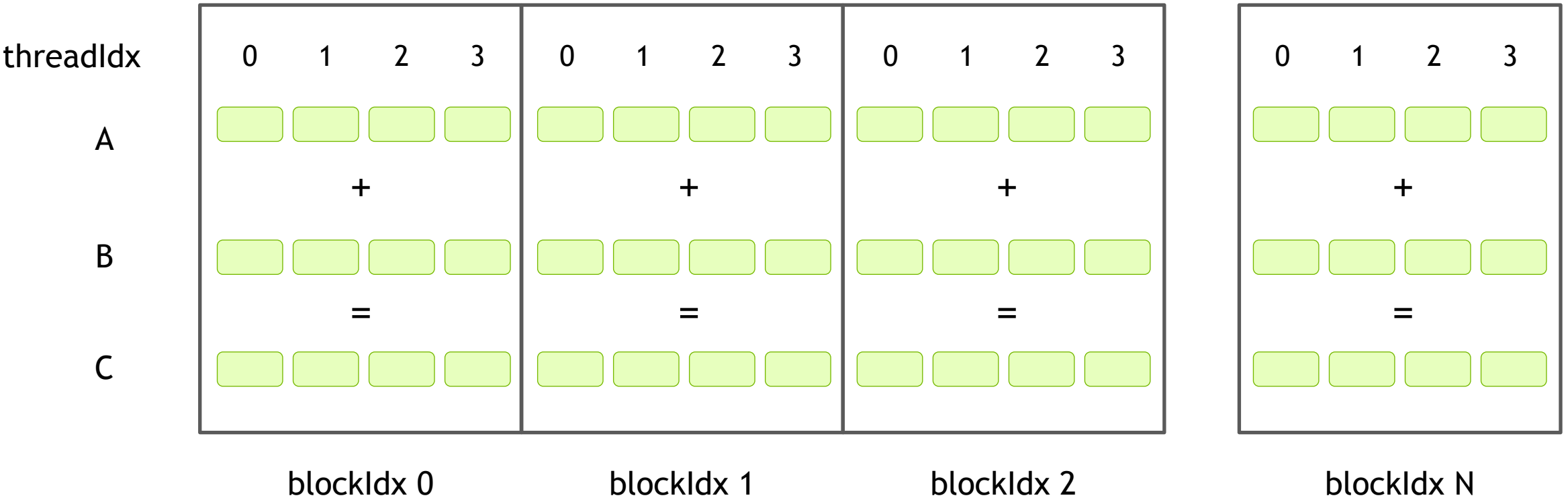
```
saxpy(py, px, alpha, N);
```

CUDA

```
__global__
void d_saxpy(float* d_a, float* d_b, float * d_c, float alpha) {
    int idx = blockDim.x * blockIdx.x + threadIdx.x;

    d_c[idx] = alpha * d_a[idx] + d_b[idx];
}
```

```
d_saxpy<<<N/block_size, block_size>>>(d_y, d_x, alpha, N);
```

GPU PARALLELIZATION




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


NVIDIA.

CUDA THREAD/BLOCK INDEX

```
int idx_x = blockDim.x * blockIdx.x + threadIdx.x
int idx_y = blockDim.y * blockIdx.y + threadIdx.y
```

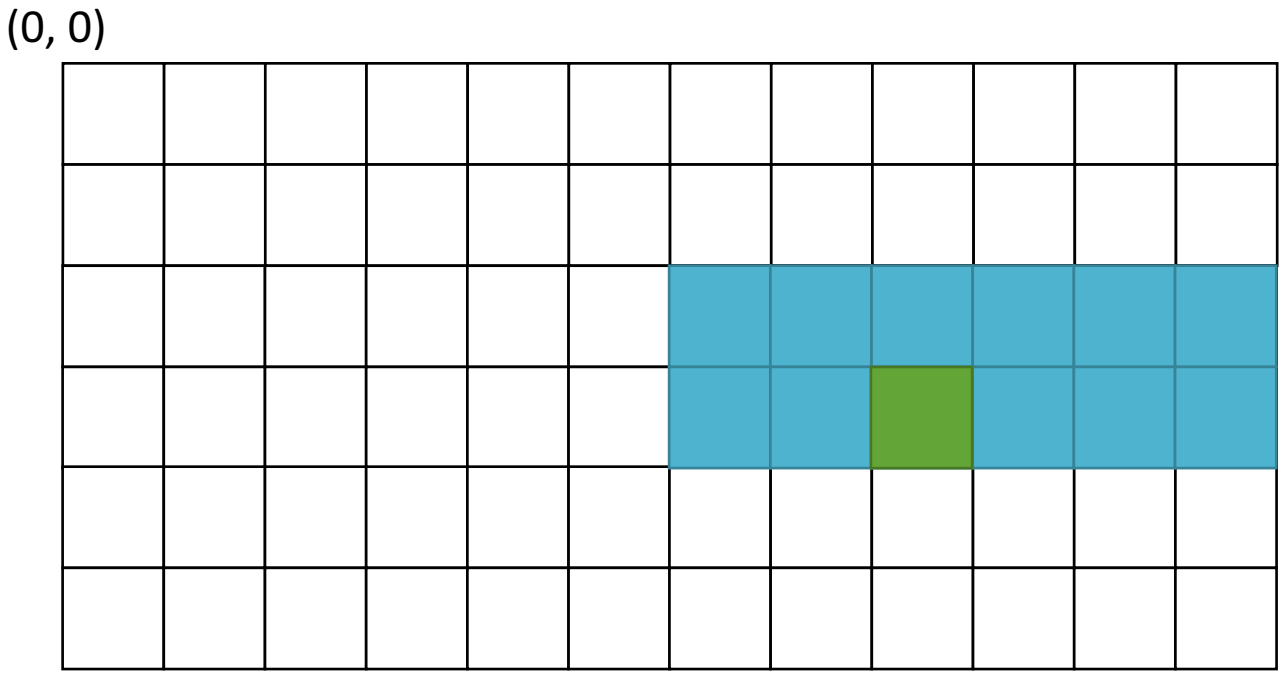
Index 지시어

blockDim (6, 2)

gridDim (2, 3)

blockIdx (1, 1)

threadIdx (2, 1)



CUDA THREAD 계층구조

계층적 병렬처리

Grid (Kernel)은 여러개의 block을 실행

Block은 여러개의 thread를 실행

각 요소의 dimension

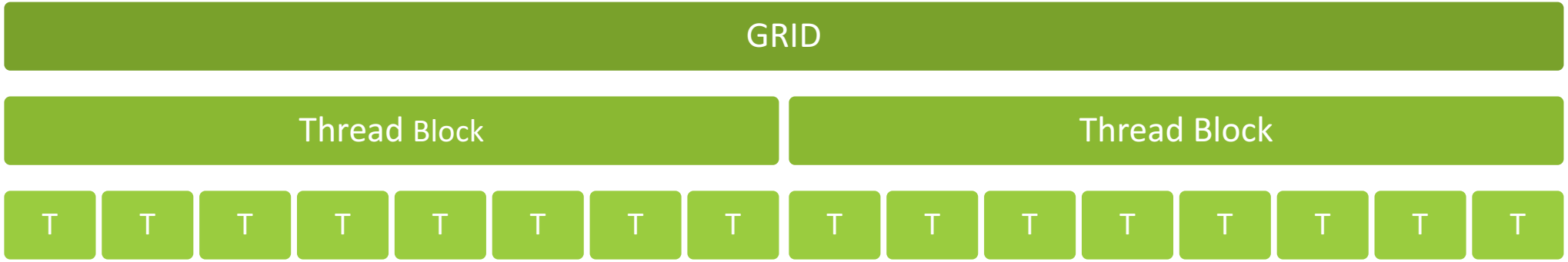
gridDim	blockIdx	blockDim	threadIdx
(x, y)	(x, y)	(x, y, z)	(x, y, z)

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



CUDA INDEX에 대한 쉬운 관점

dimension 하나마다 for문이 하나씩 제거됨

```
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];
```

```
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];
```

2D INDEX 예제

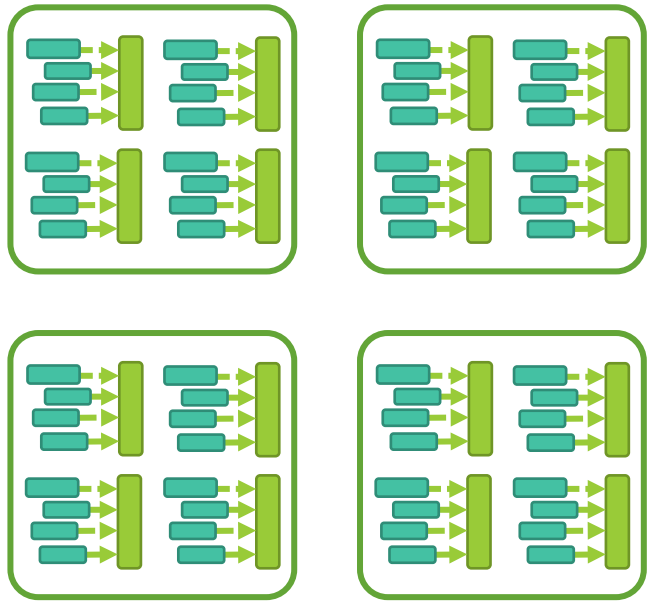
(x, y) index 표현

```
__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];
}
```

```
__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];
}
```

CUDA SM과 CUDA BLOCK의 관계

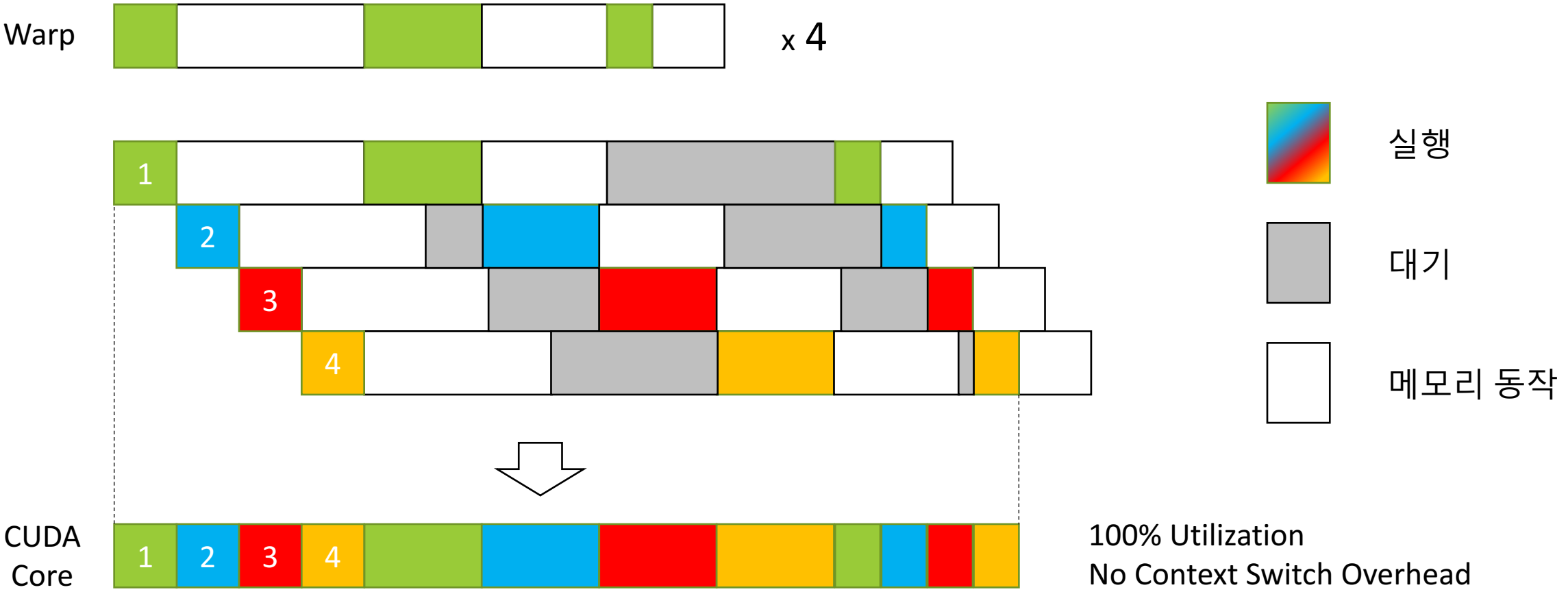
CUDA Blocks



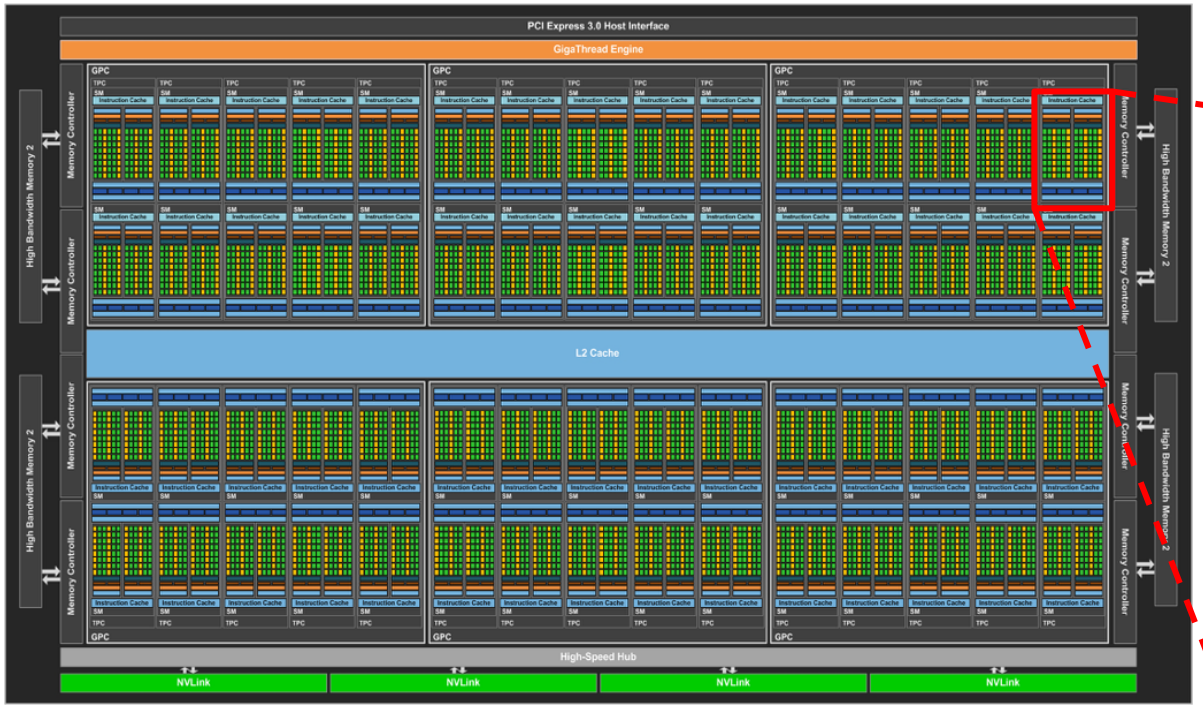
CUDA SM의 자원을 공유함

NVIDIA.

WARP SCHEDULING (EXAMPLE)

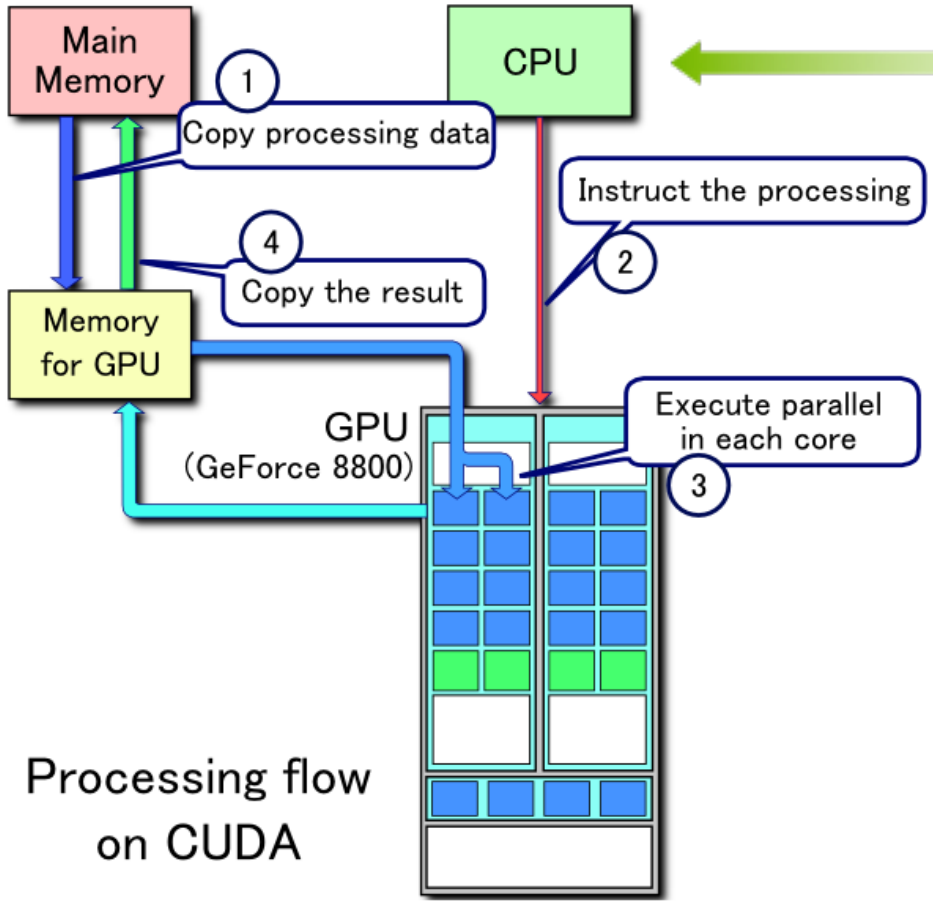


NVIDA PASCAL ARCHITECTURE



NVIDIA.

CUDA 실행 절차



GPU Memory Handling

Host Code

Kernel Code

DEVICE 메모리 할당

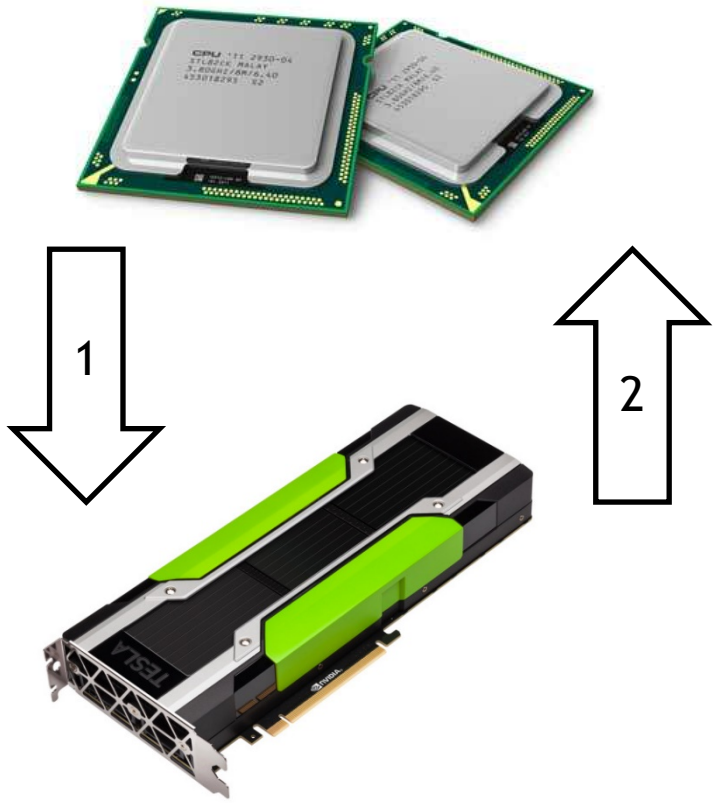
할당	`cudaMalloc(**pVoid, size_t size)`
해제	`cudaFree(*pVoid)`
초기화	`cudaMemset(*pVoid)`

CUDA MEMCPY

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

Direction

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



SAXPY EXAMPLE

```
__global__ void saxpy
    (float *a, float *b, float *c, float alpha)
{
    int i = blockIdx.x*blockDim.x+threadIdx.x;
    c[i] = alpha * a[i] + b[i];
}
```

4. Kernel Code

```
int main() {
    int N = 10;
    float *a = (float*)malloc(N*sizeof(float));
    float *b = (float*)malloc(N*sizeof(float));
    float *c = (float*)malloc(N*sizeof(float));
    float *d_a, *d_b, *d_c;

    /* data preparation */
```

1. GPU Memory Allocation

```
    cudaMalloc((void**)&d_a, N*sizeof(float));
    cudaMalloc((void**)&d_b, N*sizeof(float));
    cudaMalloc((void**)&d_c, N*sizeof(float));
```

2. DataCopy Host → Device

```
    cudaMemcpy(d_a, a, N*sizeof(float),
        cudaMemcpyHostToDevice);
    cudaMemcpy(d_b, b, N*sizeof(float),
        cudaMemcpyHostToDevice);
```

```
    vec_add<<<N/32,32>>>(d_a, d_b, d_c);
```

3. Kernel Call

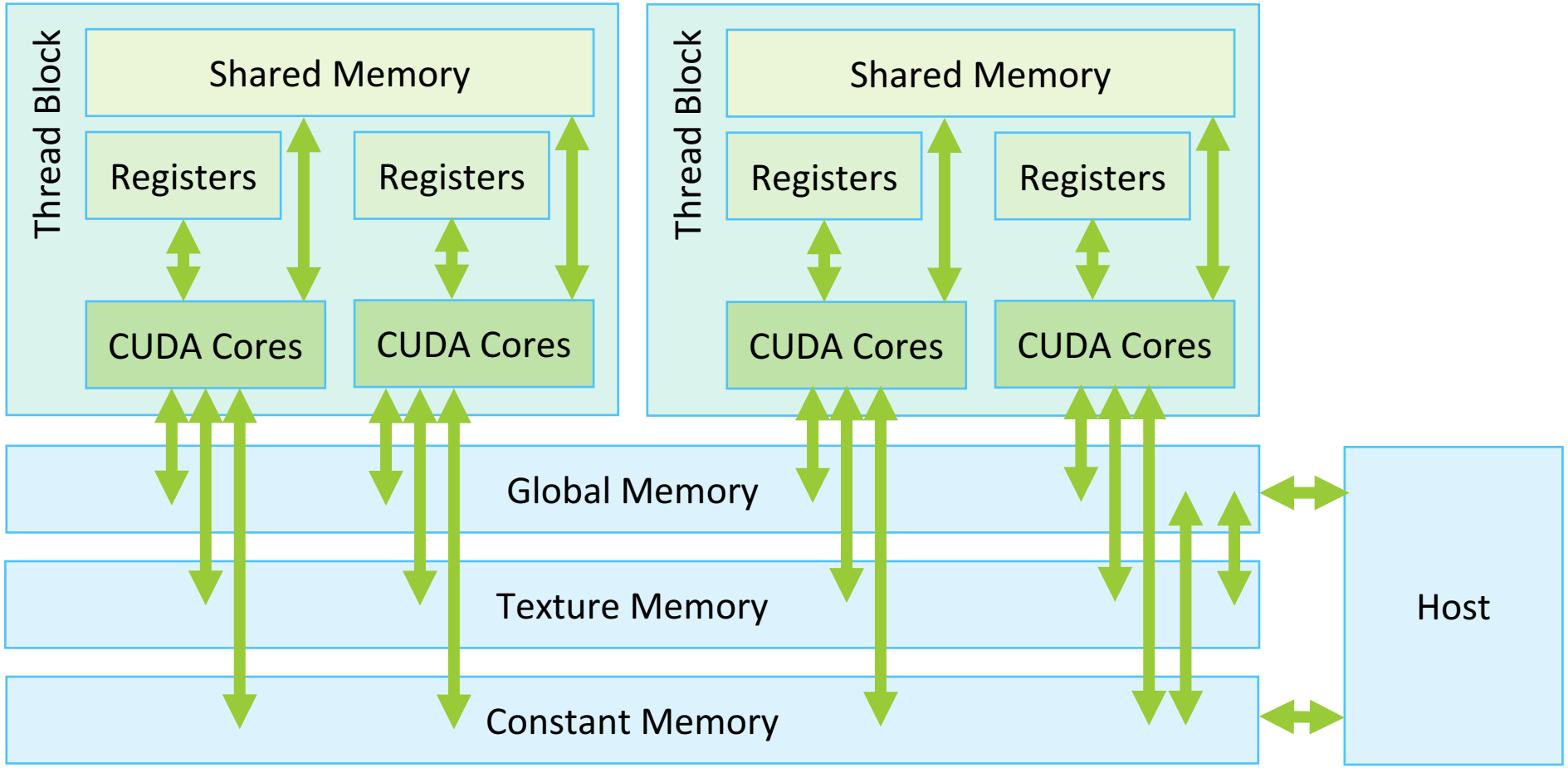
```
    cudaMemcpy(d_c, c, N*sizeof(float),
        cudaMemcpyDeviceToHost);
```

5. DataCopy Device → Host

```
    cudaFree(d_a);
    cudaFree(d_b);
    cudaFree(d_c);
    free(a);
    free(b);
    free(c);
}
```

6. GPU Memory Free

3. 계층적인 공유 메모리 구조



NVIDIA.

ON-CHIP MEMORIES

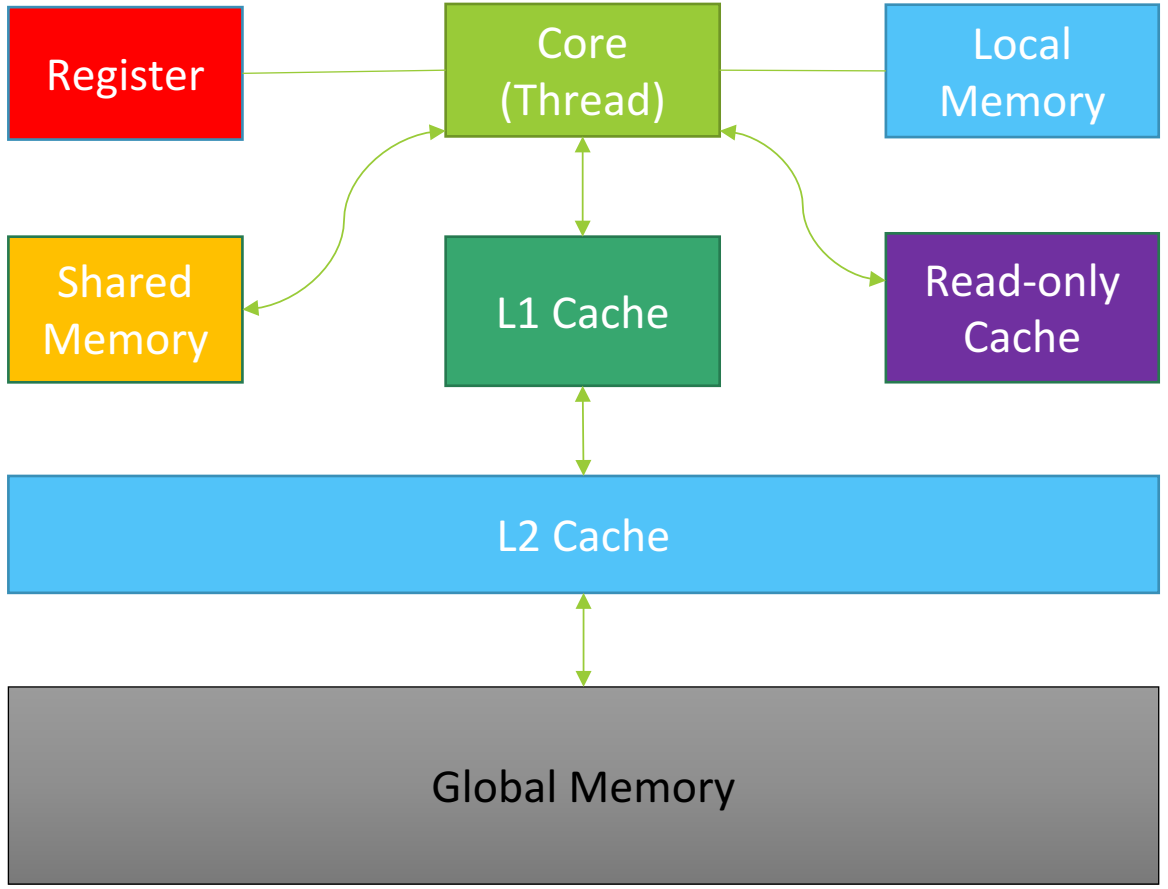
Register File

L1 Cache

Shared Memory

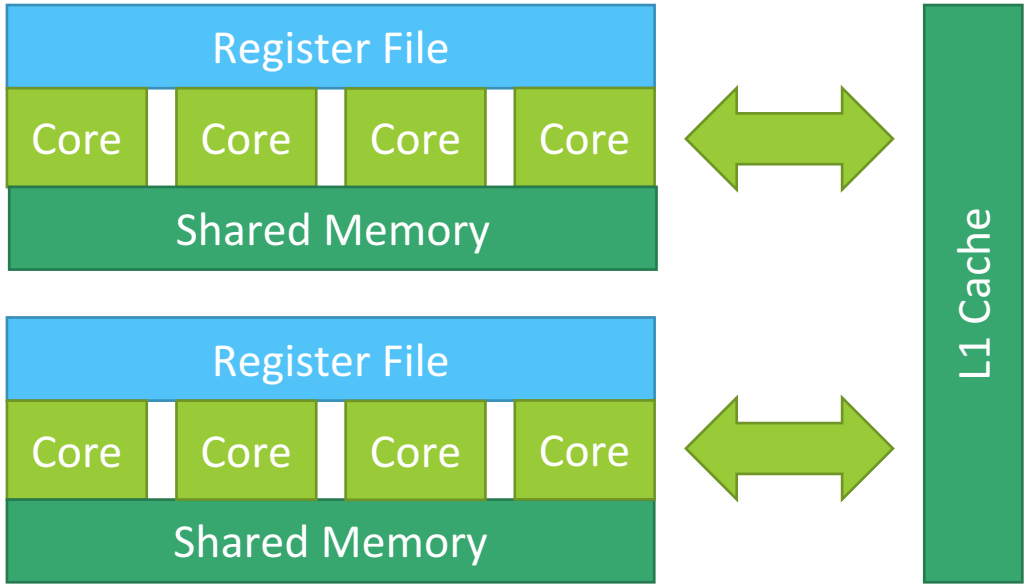
NVIDIA.

GPU CORE & MEMORY



NVIDIA.

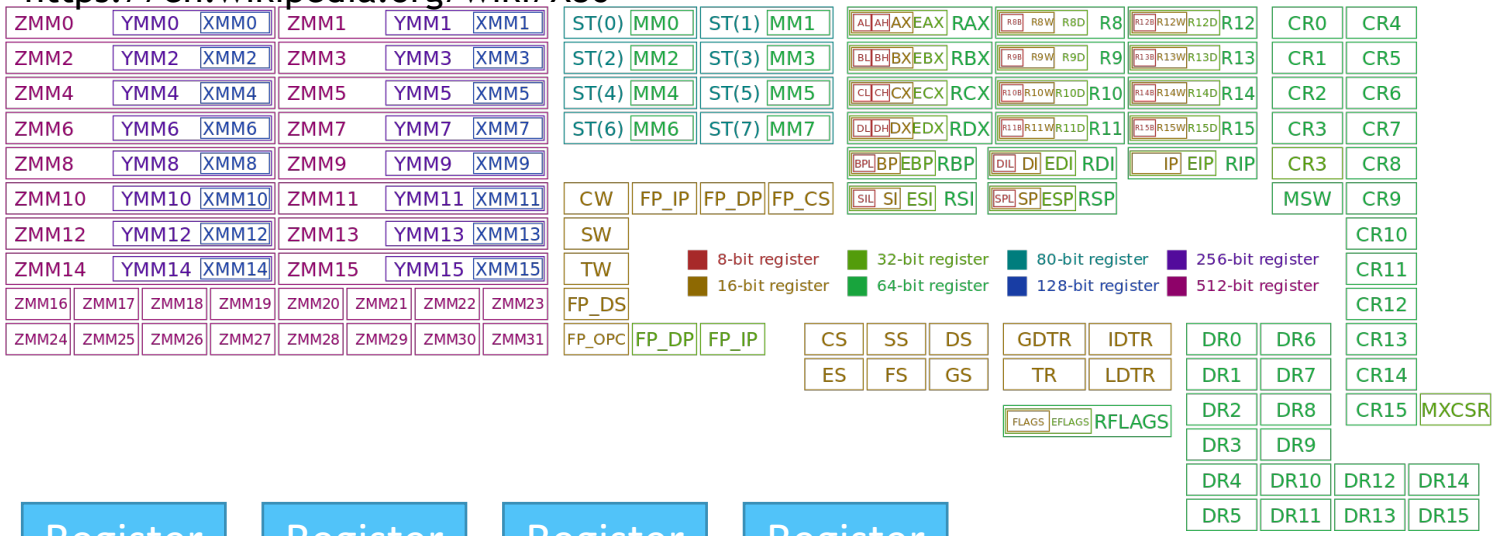
REGISTER & SHARED MEMORY SHARING

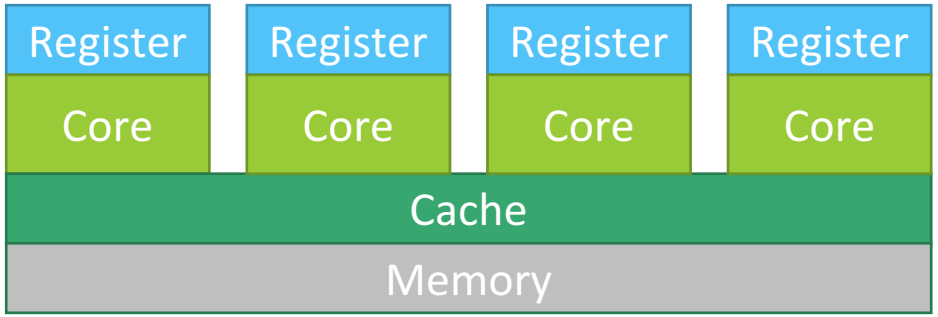


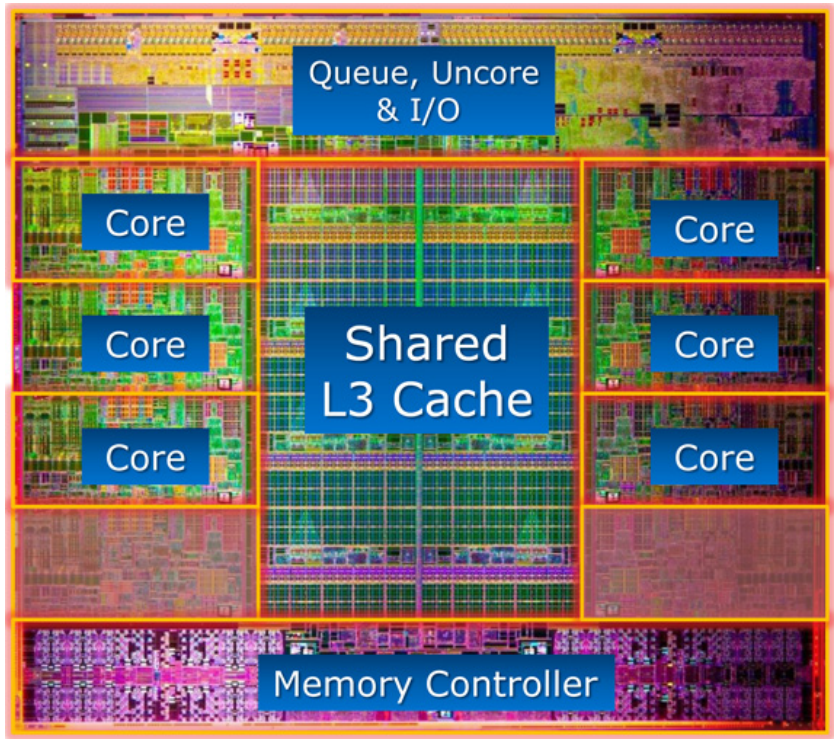
Application에 따라서 필요한 만큼의
Register를 할당받아 사용가능

CPU ARCHITECTURE

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





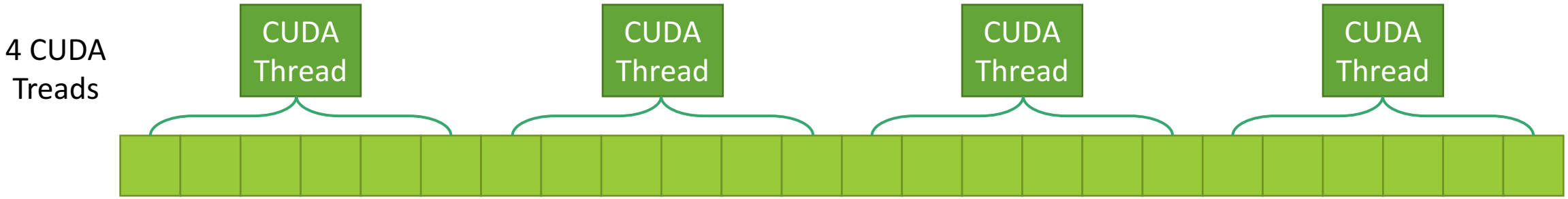


REGISTER FILE

CUDA는 SM당 256KB의 Register를 갖고 있음

CUDA Thread가 필요한 만큼 Register를 나눠가짐

- Compile 시점에 알고리즘에서 필요한 만큼을 사용할 수 있도록 정함



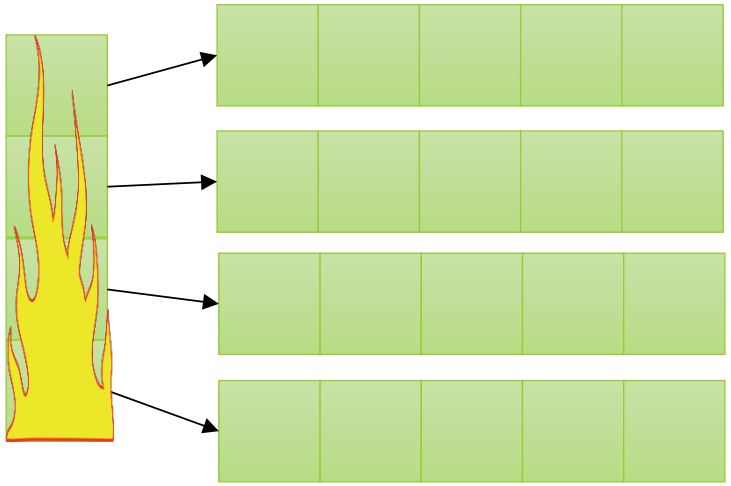
여러개의 CUDA Thread가 동시에 동작 가능

CUDA Core의 효율적인 활용 가능

DEVICE MEMORY 사용시 POINTER 문제

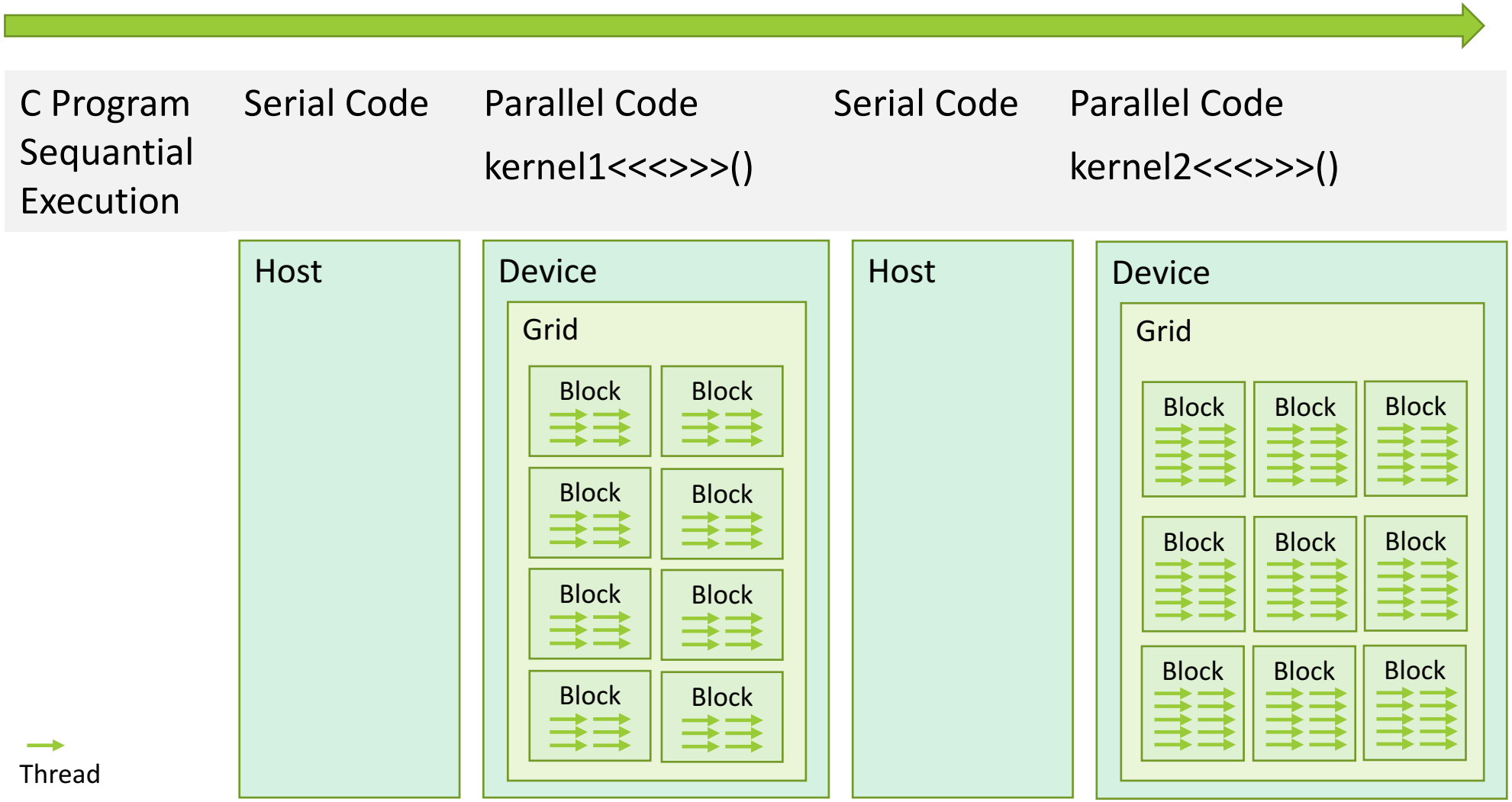
- 동적 할당 된 2차원 배열 사용시 GPU 병렬성 저하

A[4][5]



- 관리가 어려움
  - 각각의 array 들과 array of pointer를 일일이 GPU 메모리로 보내야 함
- Structure내의 pointer 변수가 있다면, pointer가 가르키는 메모리에 대해서도 GPU메모리 할당 필요
- `struct` 는 linear buffer로 이슈 없음

4. HETEROGENEOUS COMPUTING



CUDA 초기화

CUDA runtime에서 자동으로 초기화

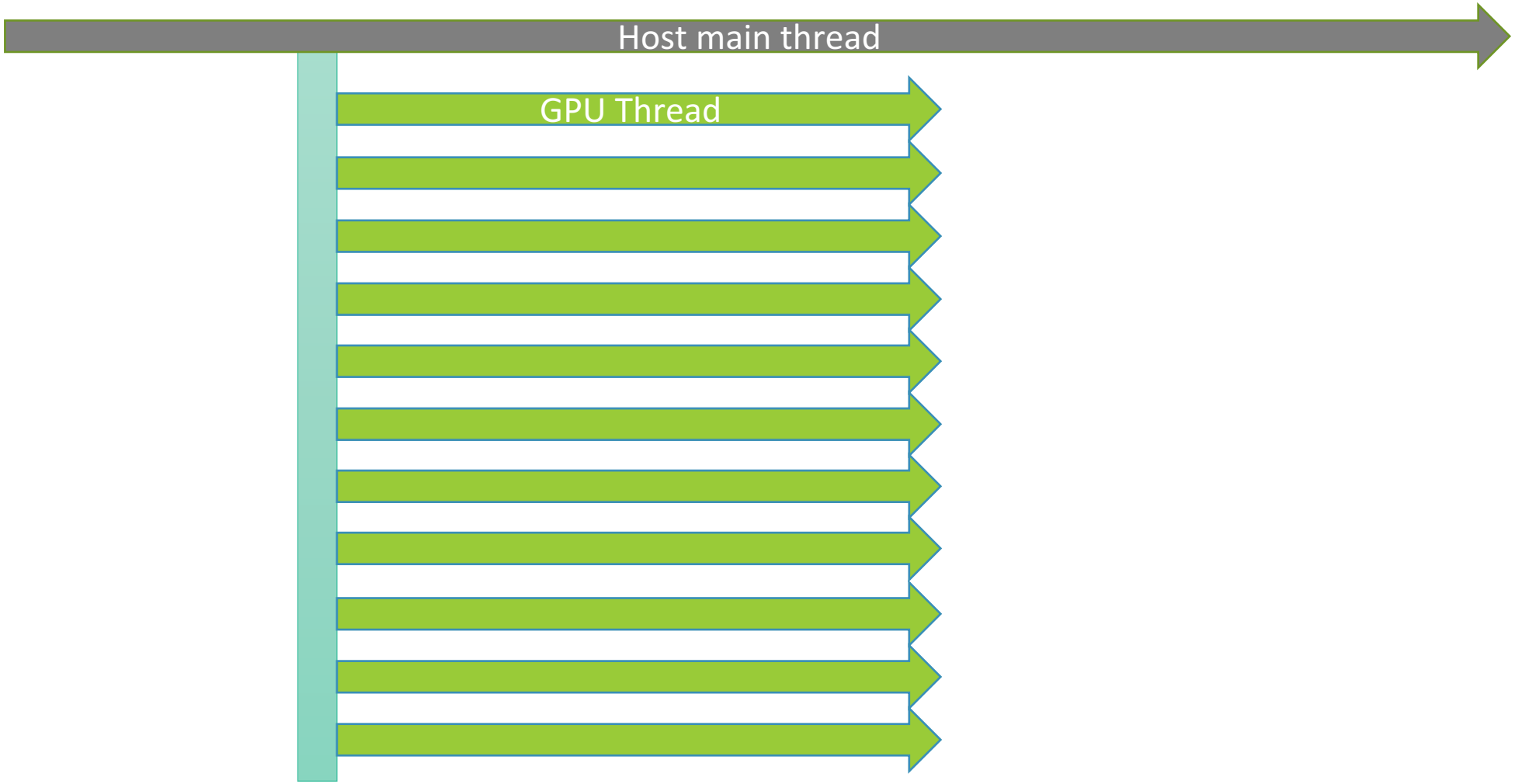
하나의 Process에 CUDA Device 별로 CUDA Context가 생성됨

Reset 명령은 cudaDeviceReset()

호출한 Host Thread가 선택한 Device에 대하여 Reset

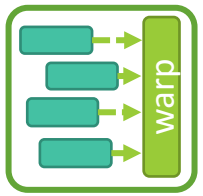
실무에서는 CUDA카드의 설치 유무, 버전 확인등 수행

CUDA PARALLEL OPERATION

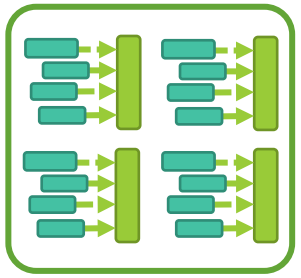


계층적 동기화 지원

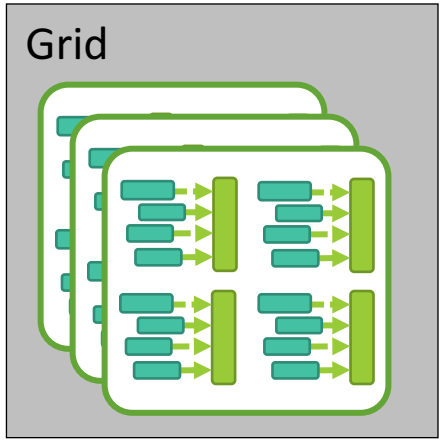
warp



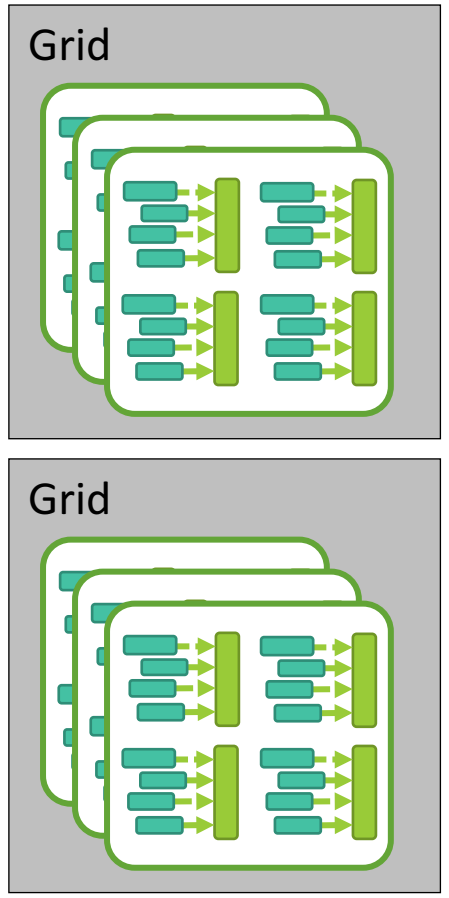
block



__syncthreads()

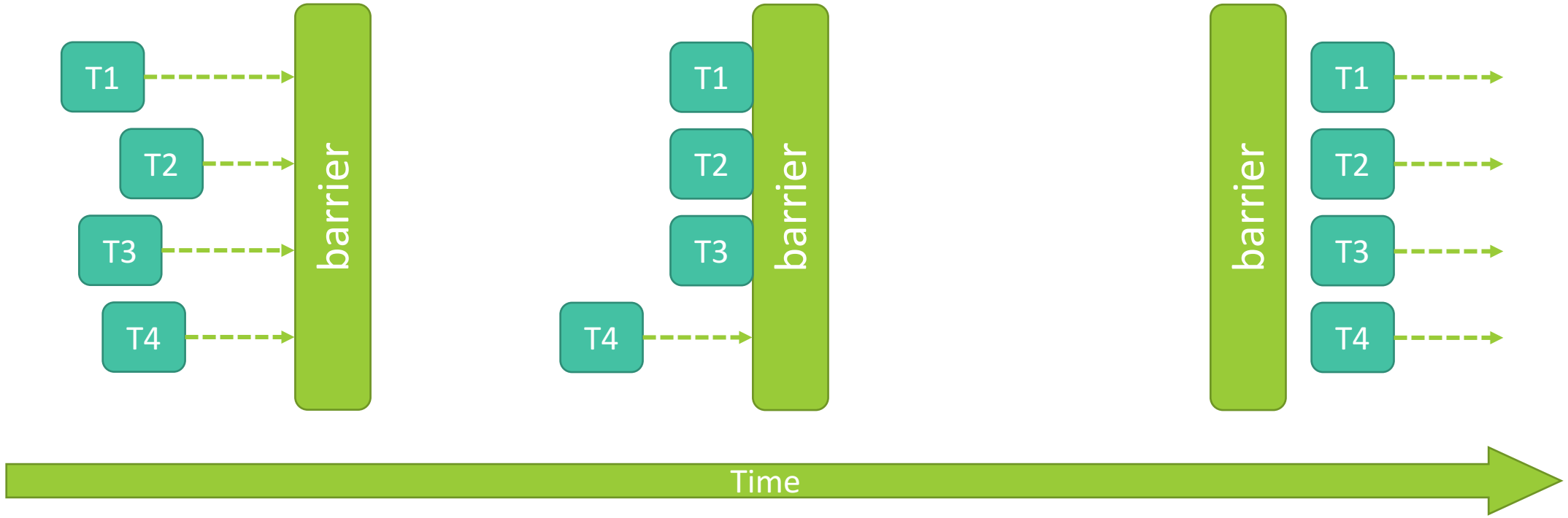


cudaStreamSynchronize ()



cudaDeviceSynchronize ()

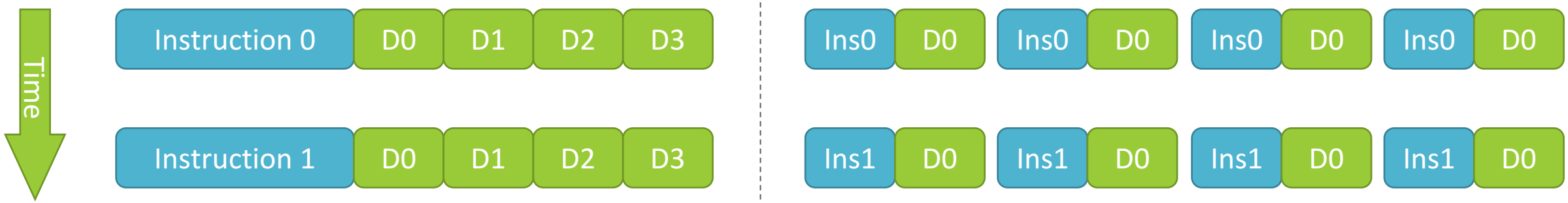
BARRIER 동기화



SIMT 동작

Single Instruction Multiple Thread

SIMD(Single Instruction Multiple Data)와 대비됨



5. COMPUTE CAPABILITY

CUDA Compute Capability

Target GPU Arch. verson

Version	Architecture 명
6.x	Pascal
5.x	Maxwell
3.x	Kepler
2.x	Fermi
1.x	Tesla

CUDA Version

CUDA API Version

현재 8.0, 9.0rc 공개

Version	특징
9.0rc	Pascal 지원 추가
8.0	Pascal 지원

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);
}
```

CUDA 개발 환경

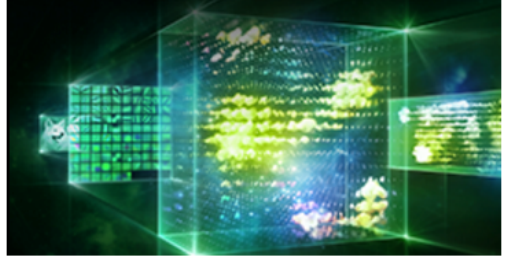
그리고 실습 환경

NVIDIA SDK

The Essential Resource for GPU Developers

NVIDIA SDK

DEEP LEARNING
Deep Learning SDK
High-performance tools and libraries for deep learning



SELF-DRIVING CARS
NVIDIA DriveWorks™
Deep learning, HD mapping and supercomputing solutions, from ADAS to fully autonomous

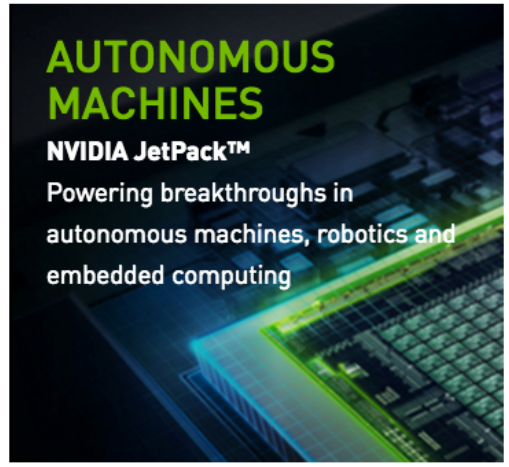
VIRTUAL REALITY
NVIDIA VRWorks™
A comprehensive SDK for VR headsets, games and professional applications

GAME DEVELOPMENT
NVIDIA GameWorks™
Advanced simulation and rendering technology for game development

ACCELERATED COMPUTING
NVIDIA ComputeWorks™
Everything scientists and engineers need to build GPU-accelerated applications

DESIGN & VISUALIZATION
NVIDIA DesignWorks™
Tools and technologies to create professional graphics and advanced rendering applications

AUTONOMOUS MACHINES
NVIDIA JetPack™
Powering breakthroughs in autonomous machines, robotics and embedded computing



ADDITIONAL RESOURCES
More resources for GPU Developers

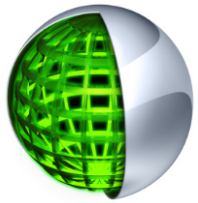
COMPUTEWORKS

Libraries, Languages and Development Tools for GPU Computing

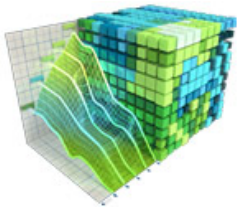
Programming Approaches

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

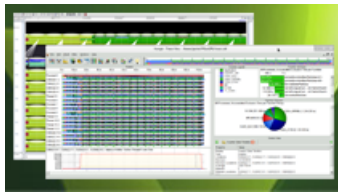
Development Environment



Nsight IDE

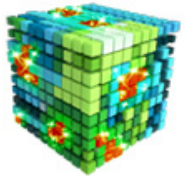


NVIDIA Visual Profiler



CUDA Profiling Tools Interface

CUDA-GDB Debugger



CUDA MEMCHECK

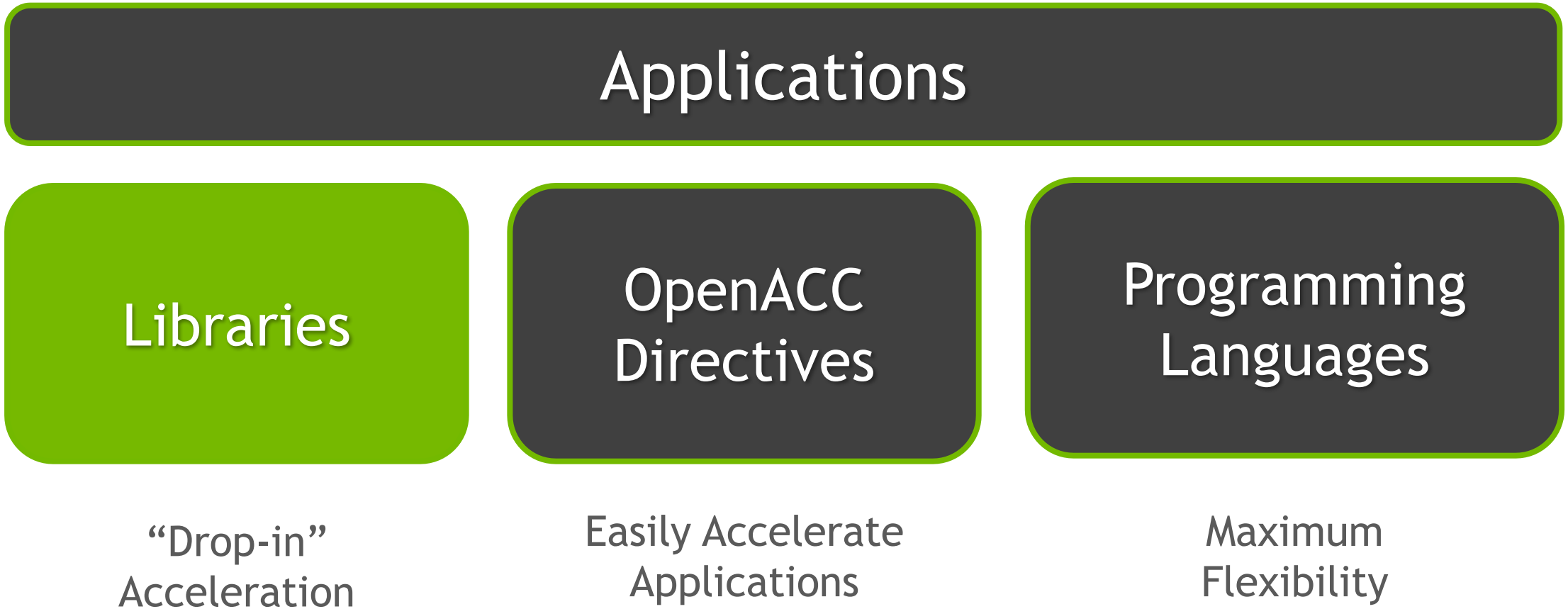
Language Support

C C++ Fortran python

LLVM COMPILER INFRASTRUCTURE

Compile new languages to CUDA

3 WAYS TO ACCELERATE APPLICATIONS

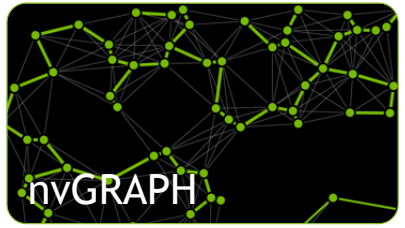


GPU ACCELERATED LIBRARIES

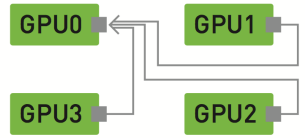
"Drop-in" Acceleration for Your Applications

Domain-specific

Deep Learning, GIS, EDA, Bioinformatics, Fluids



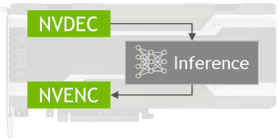

NVIDIA TensorRT



NCCL

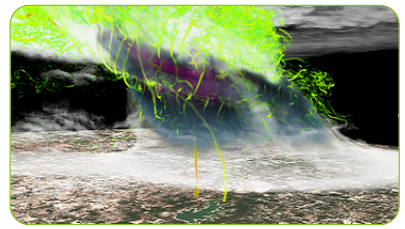
Visual Processing

Image & Video



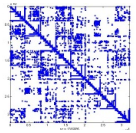
DeepStream SDK

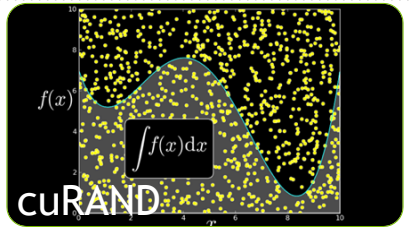

NVIDIA CODEC SDK

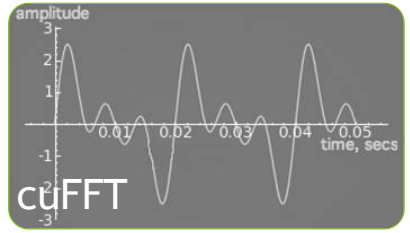


Linear Algebra

Dense, Sparse, Matrix

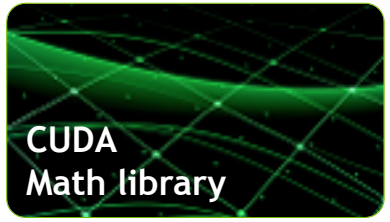

cuSPARSE





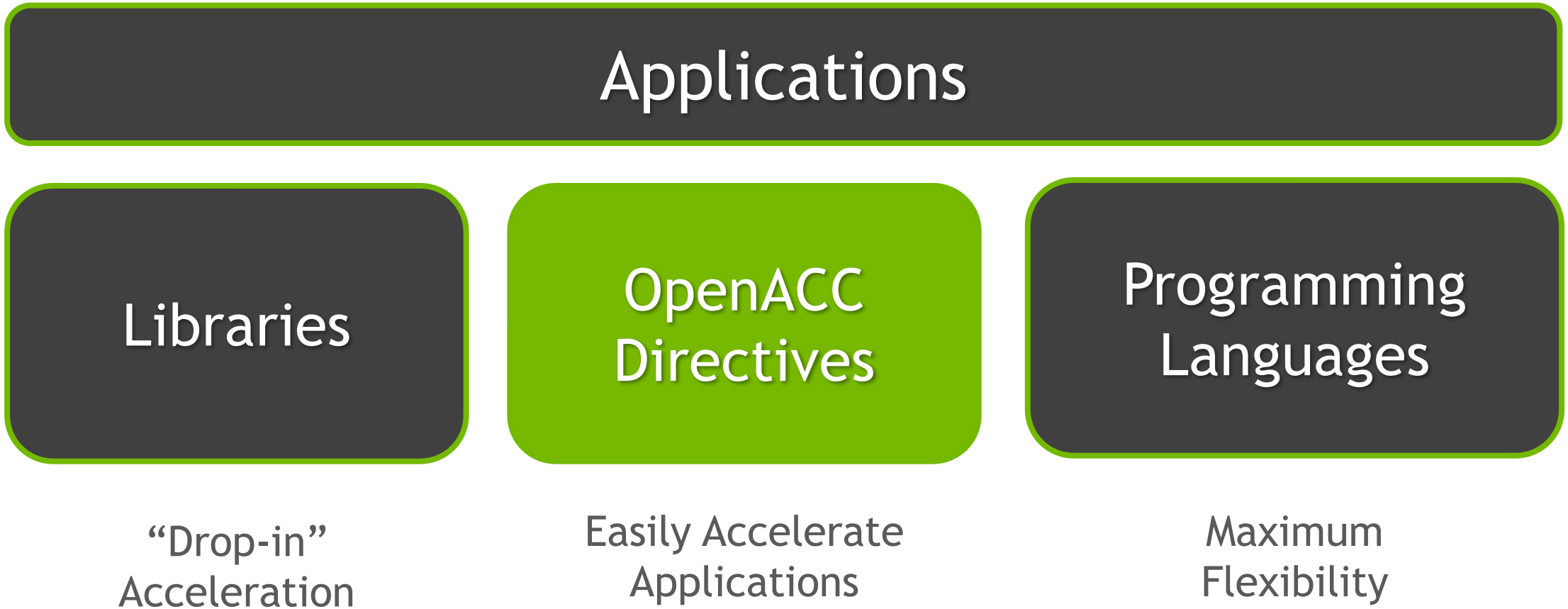
Math Algorithms

AMG, Templates, Solvers



developer.nvidia.com/gpu-accelerated-libraries

3 WAYS TO ACCELERATE APPLICATIONS



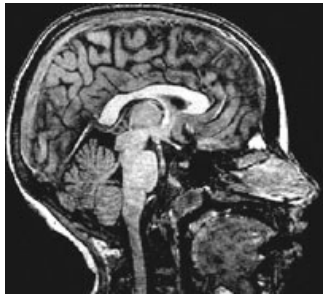
OpenACC

Simple | Powerful | Portable

Fueling the Next Wave of Scientific Discoveries in HPC

```
main()
{
  <serial code>
  #pragma acc kernels
  //automatically runs on GPU
  {
    <parallel code>
  }
}
```

University of Illinois
PowerGrid- MRI Reconstruction



70x Speed-Up
2 Days of Effort

RIKEN Japan
NICAM- Climate Modeling

7-8x Speed-Up
5% of Code Modified

8000+

Developers

using OpenACC

Compiler: PGI Compiler (Free for Comm. Edition)
http://www.pgroup.com/
http://www.pgroup.com/products/community.htm

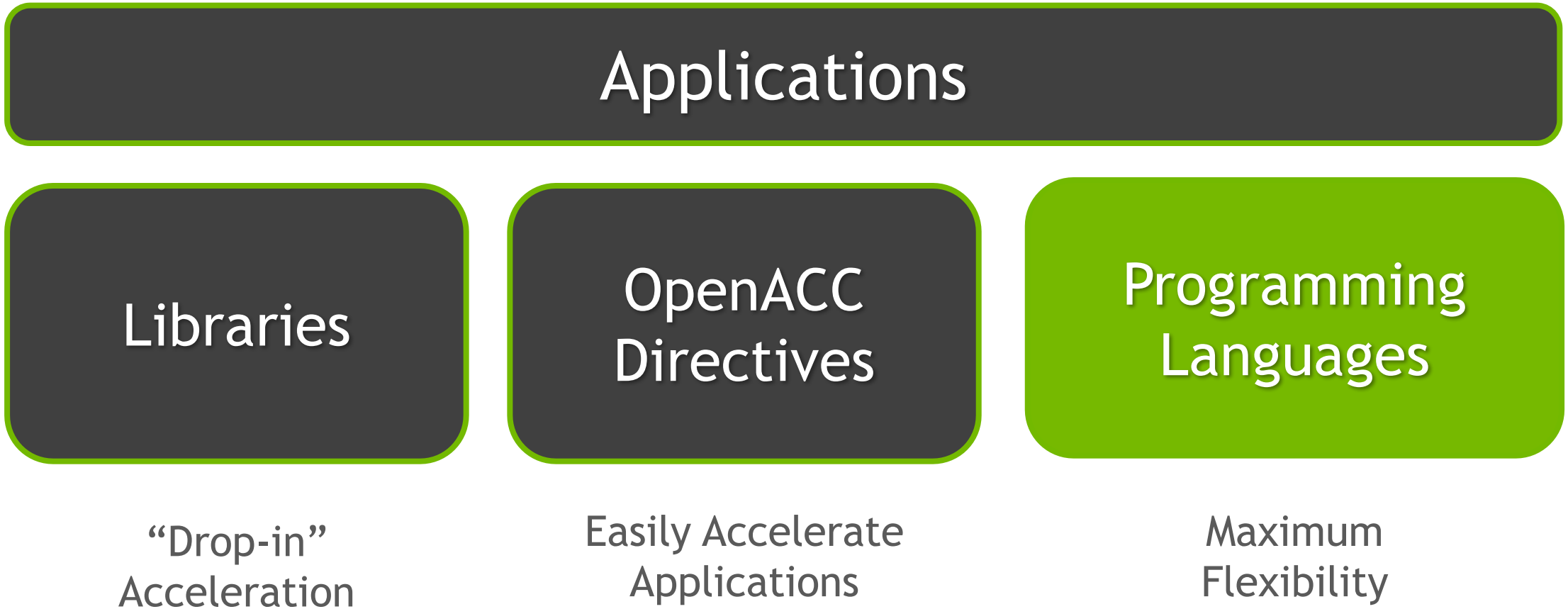
http://www.cray.com/sites/default/files/resources/OpenACC_213462.12_OpenACC_Cosmo_CS_FNL.pdf
http://www.hpcwire.com/off-the-wire/first-round-of-2015-hackathons-gets-underway
http://on-demand.gputechconf.com/gtc/2015/presentation/S5297-Hisashi-Yashiro.pdf
http://www.openacc.org/content/experiences-porting-molecular-dynamics-code-gpus-cray-xk7

PGI COMPILERS & TOOLS FOR EVERYONE

PGI Community Edition Now Available

	FREE PGI Community Edition	PGI Professional Edition	PGI Enterprise Edition
PROGRAMMING MODELS OpenACC, CUDA Fortran, OpenMP, C/C++/Fortran Compilers and Tools	✓	✓	✓
PLATFORMS X86, OpenPOWER, NVIDIA GPU	✓	✓	✓
UPDATES	1-2 times a year	6-9 times a year	6-9 times a year
SUPPORT	User Forums	PGI Support	PGI Premier Services
LICENSE	Annual	Perpetual	Volume/Site

3 WAYS TO ACCELERATE APPLICATIONS



GPU PROGRAMMING LANGUAGES

Numerical analytics	MATLAB, Mathematica, LabVIEW
Fortran	CUDA Fortran, OpenACC
C, C++	CUDA C++, OpenACC
Python	CUDA Python, PyCUDA
C#	Altimesh Hybridizer, Alea GPU

CUDA C

```
void saxpy_serial(int n,
                  float a,
                  float *x,
                  float *y)
{

  for (int i = 0; i < n; ++i)
    y[i] = a*x[i] + y[i];
}

// Perform SAXPY on 1M elements
saxpy_serial(4096*256, 2.0, x, y);
```

```
__global__
void saxpy_parallel(int n,
                    float a,
                    float *x,
                    float *y)
{
  int i = blockIdx.x*blockDim.x +
          threadIdx.x;
  if (i < n) y[i] = a*x[i] + y[i];
}

// Perform SAXPY on 1M elements
saxpy_parallel<<<4096,256>>>(n,2.0,x,y);
```

http://developer.nvidia.com/cuda-toolkit

CUDA TOOLKIT & DRIVER

CUDA Toolkit 설치시 Driver와 Toolkit이 함께 설치 가능

NVIDIA Driver가 설치된 후, Toolkit만 별도로 설치해도 무방

Toolkit에는 CUDA Sample Code/Debugger/Profiler 포함

개발 가능 OS

TOOLKIT DOWNLOAD

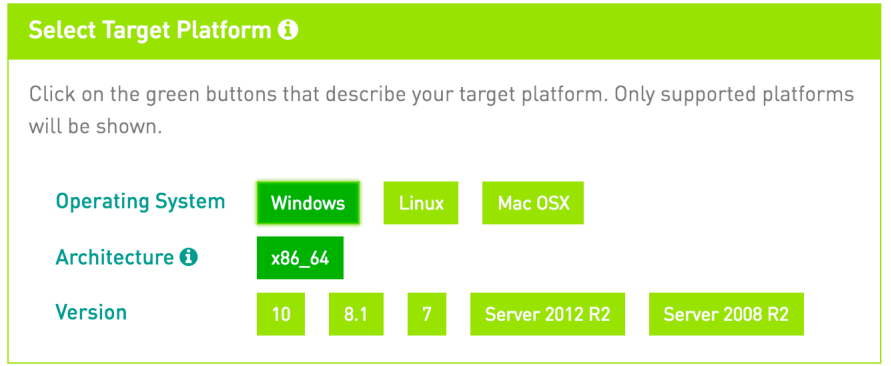
https://developer.nvidia.com/cuda-downloads

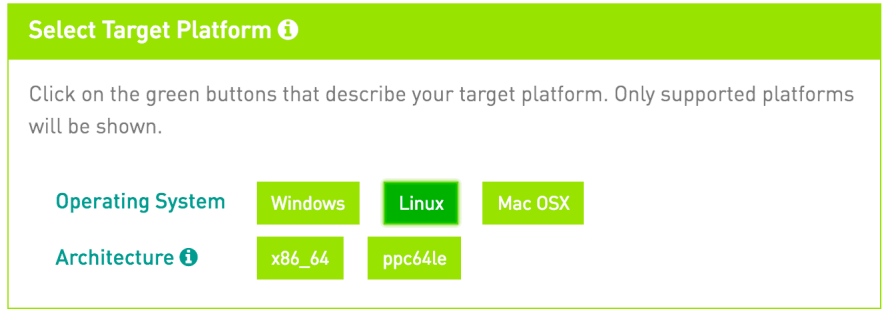
Windows, Linux, OSX 지원

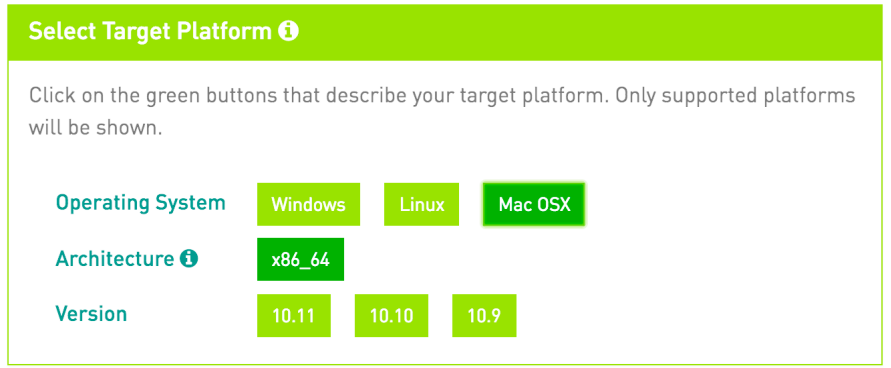
개발 환경에 맞춰서 설치 후 사용

약 1.2 GB

드라이버, 컴파일러, 라이브러리, 샘플, 디버거 등이 설치됨





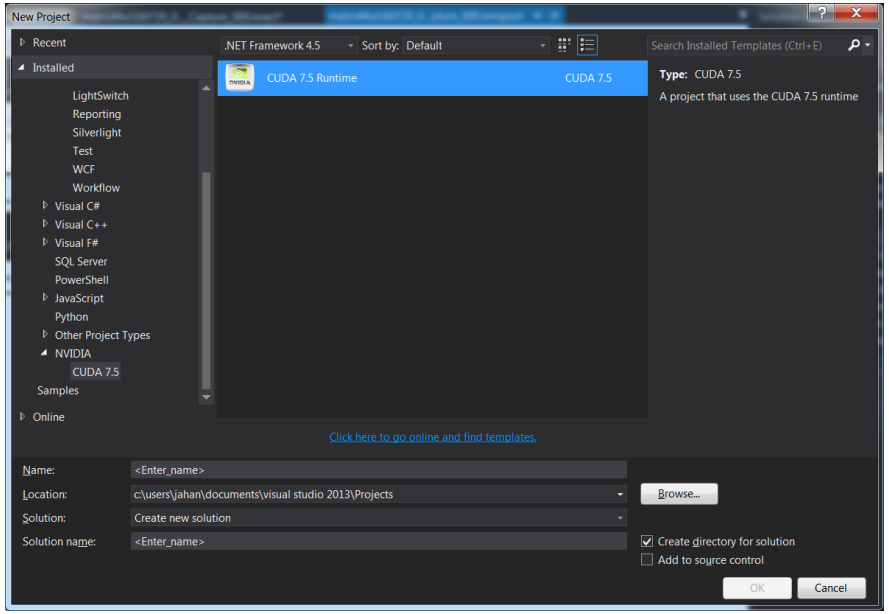


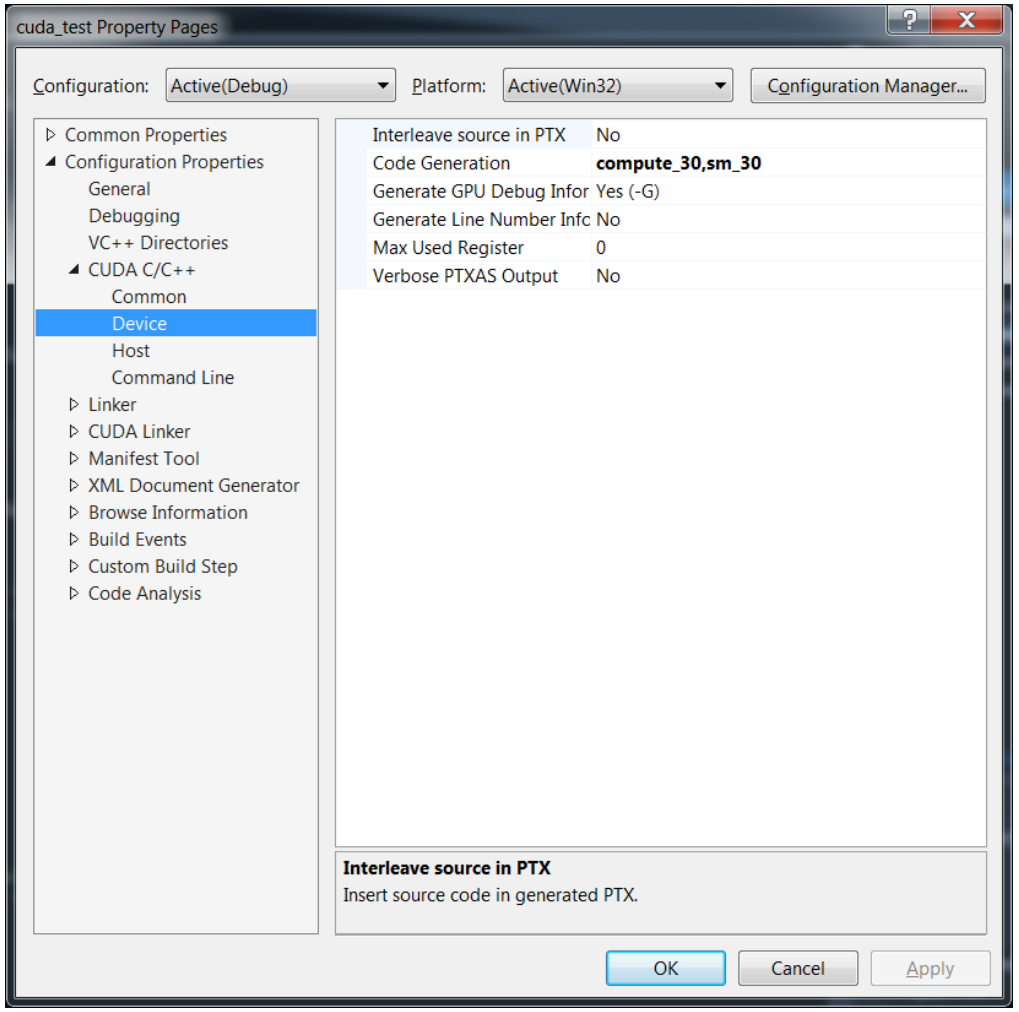
WINDOWS 개발환경 구성

Windows 7 ~ 10

Visual Studio 2010-2013

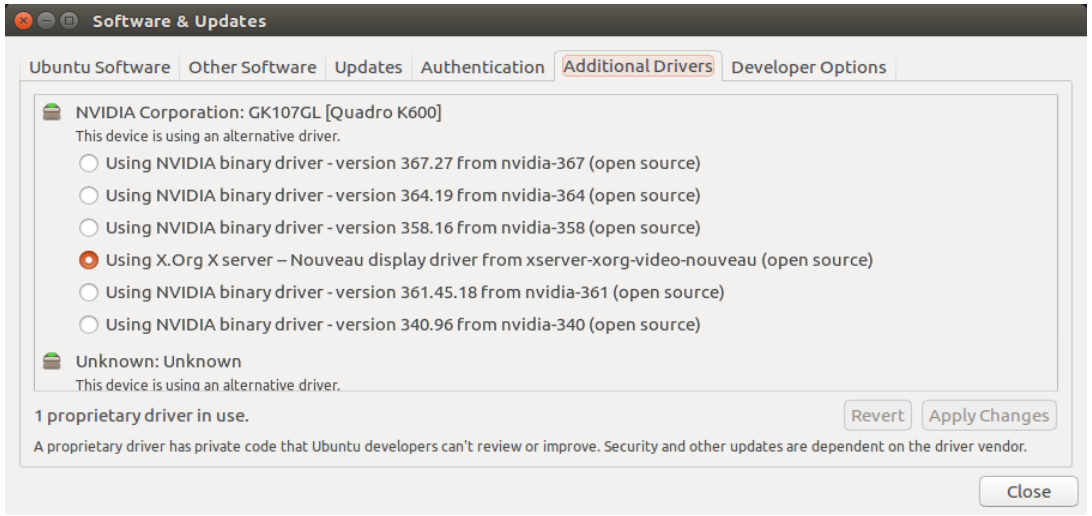
CUDA Toolkit 8.0





LINUX 개발환경 구성

Driver 설치




```
$ sudo add-apt-repository ppa:graphics-drivers/ppa
$ sudo apt update
$ sudo apt-get install nvidia-387
```

Toolkit 설치

Ubuntu

runfile　deb

```
sudo dpkg -i <toolkit>.deb
sudo apt-get update
sudo apt-get install cuda-toolkit-8-0
```

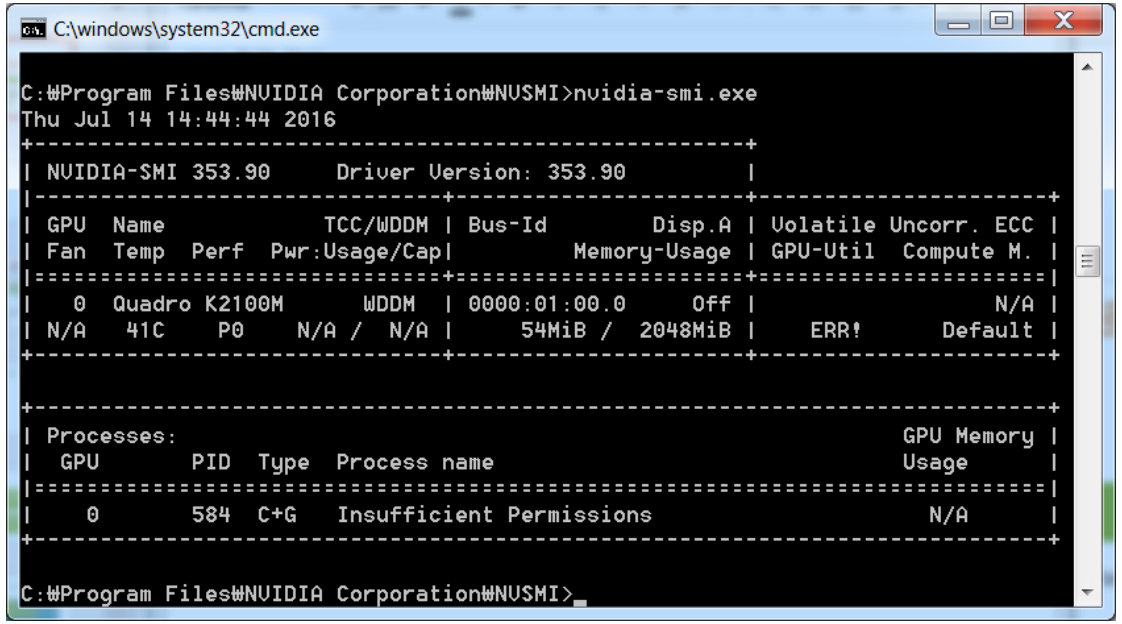
CentOS

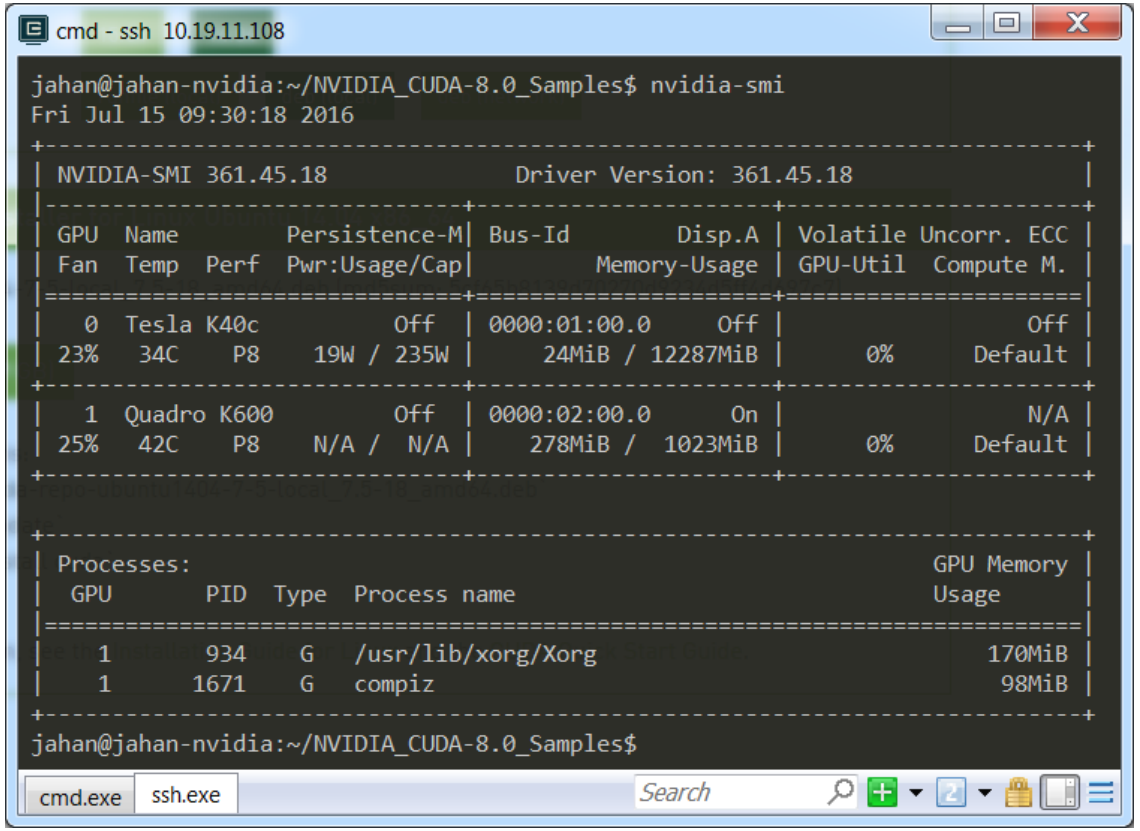
runfile　rpm

```
sudo rpm -i <toolkit>.rpm
sudo yum clean all
sudo yum install cuda-toolkit-8-0
```

Runfile 설치시 nouveau driver disable 필수

설치 확인 - NVIDIA-SMI





-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 연동

선행 요구 사항

C/C++ 프로그래밍 경험 필요

그래픽스 경험 없어도 익힐 수 있음

병렬 프로그래밍 경험은 익혀야 함

NVIDIA.

실습 개발환경

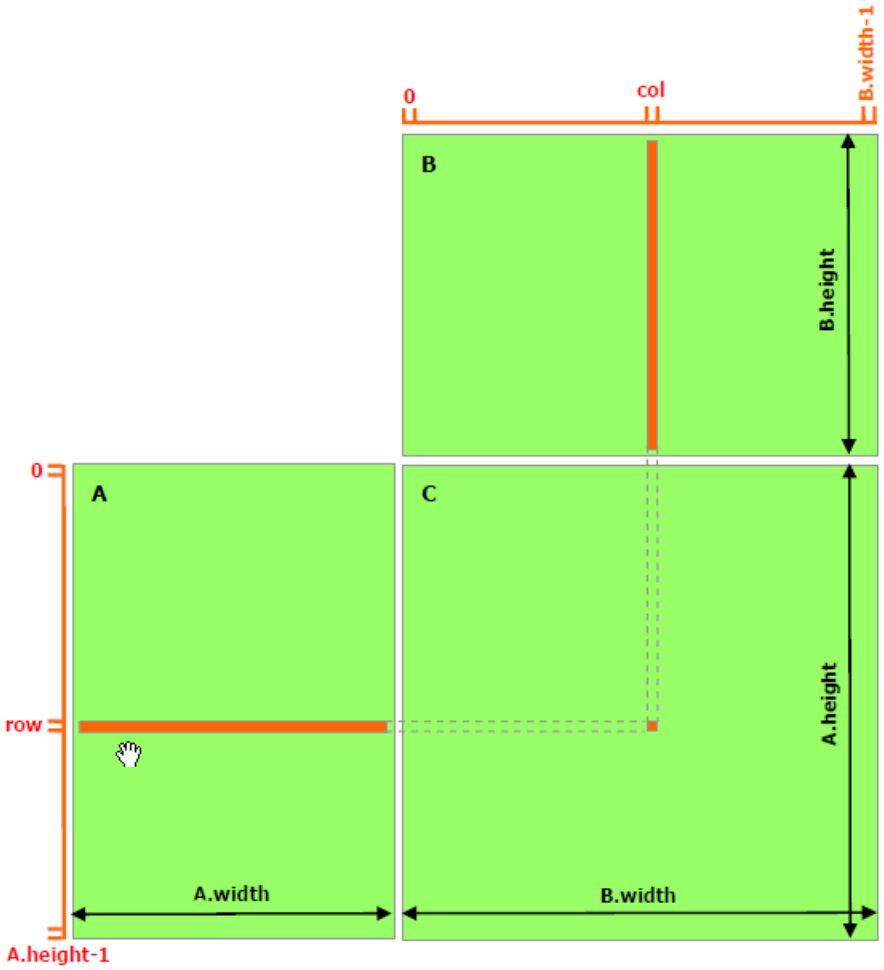
- Ubuntu 14.04
- Jupyter / Shell Interface
- K520 GRID GPU

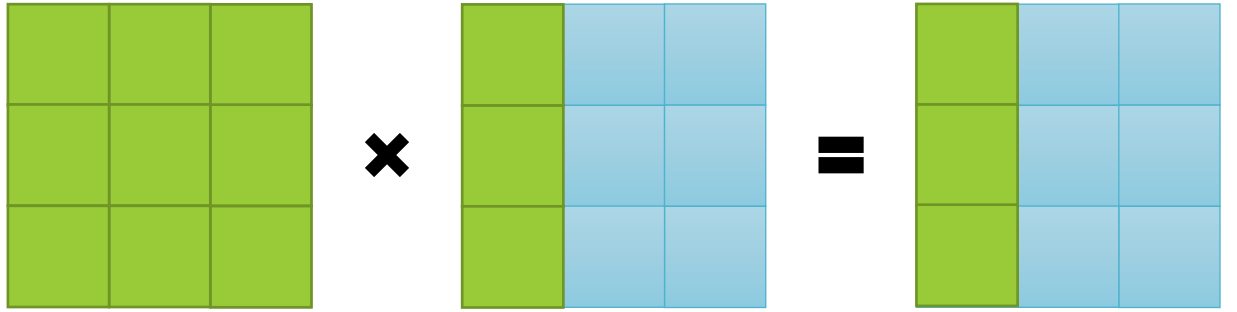
- Jupyter
  - *.jypnb 파일
  - 40가지 언어 지원 (python, markdown 기본)
  - 코드와 실행화면 함께 볼 수 있음
  - Data Science 분야에서 많이 사용되는 웹앱
  - 단축키
    - Shift + enter: 코드 실행
    - l : 줄번호

CUDA EXAMPLE 1

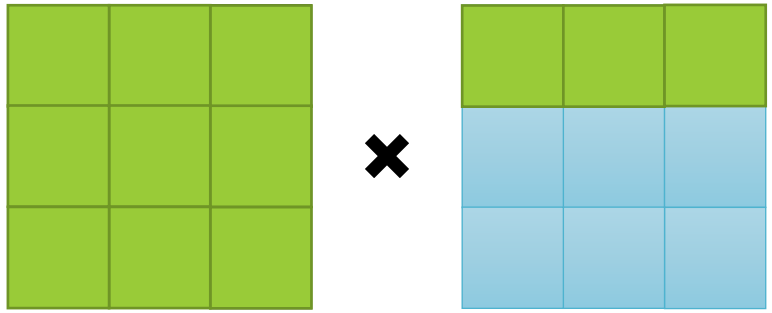
SGEMM with shared memory

MATRIX MULTIPLICATION

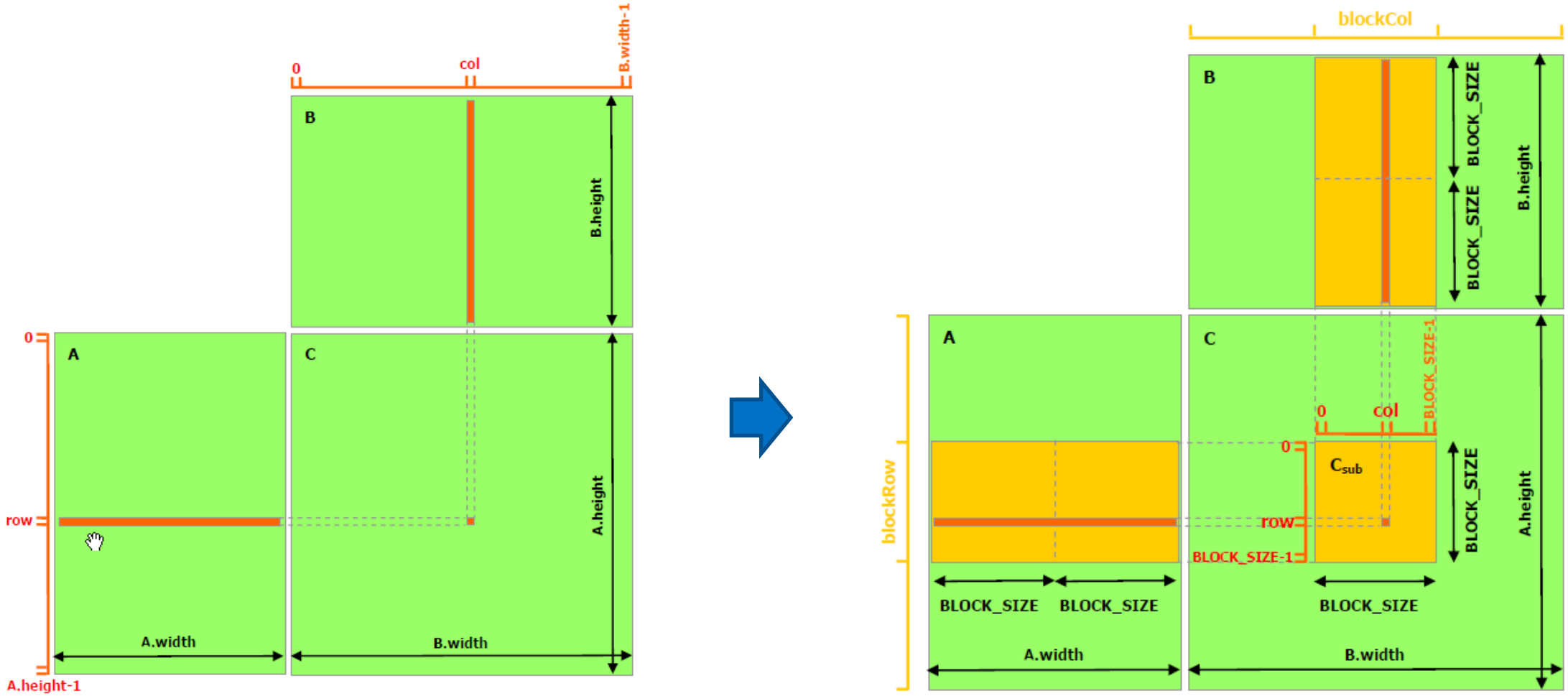




transpose



MATRIX MULTIPLICATION WITH SHARED MEMORY



NVIDIA.

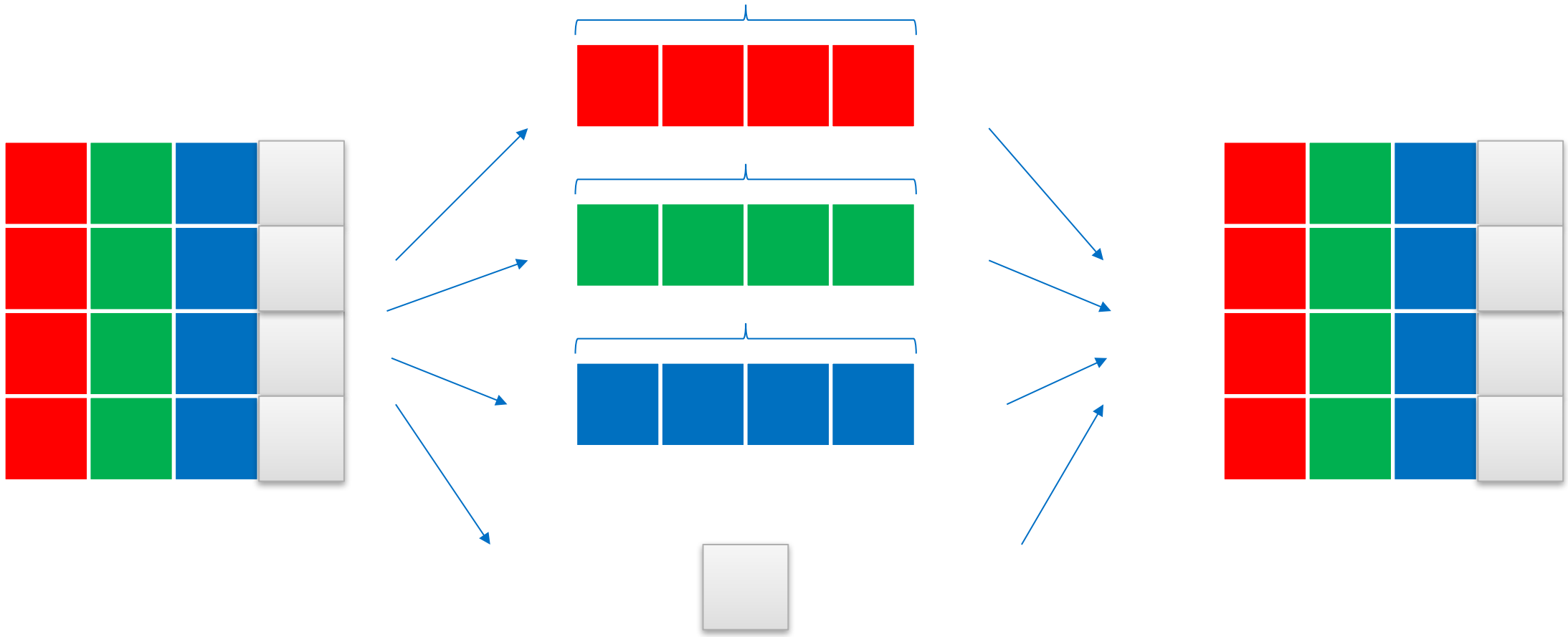
CUDA EXAMPLE 2

Blurring Filter 만들기

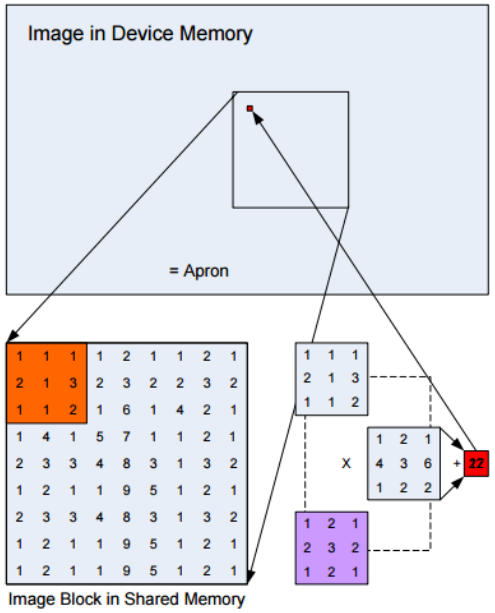
(Image Convolution)

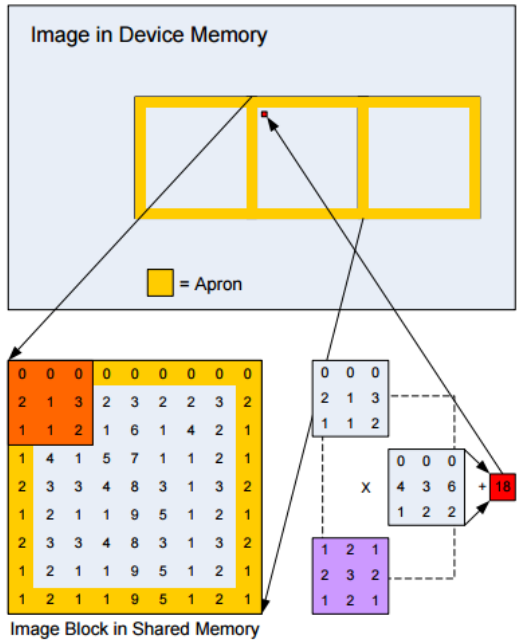
IMAGE BLURRING FILTER (BLUR_FILTER.IPUB)

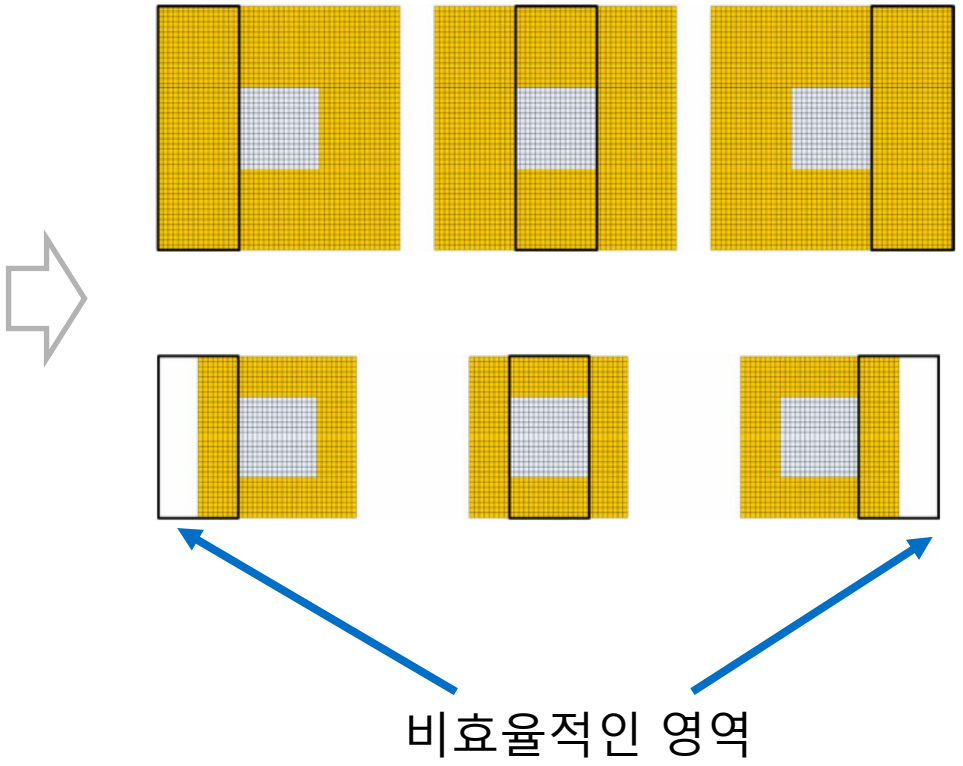
IMAGE BLURRING FILTER OPERATION



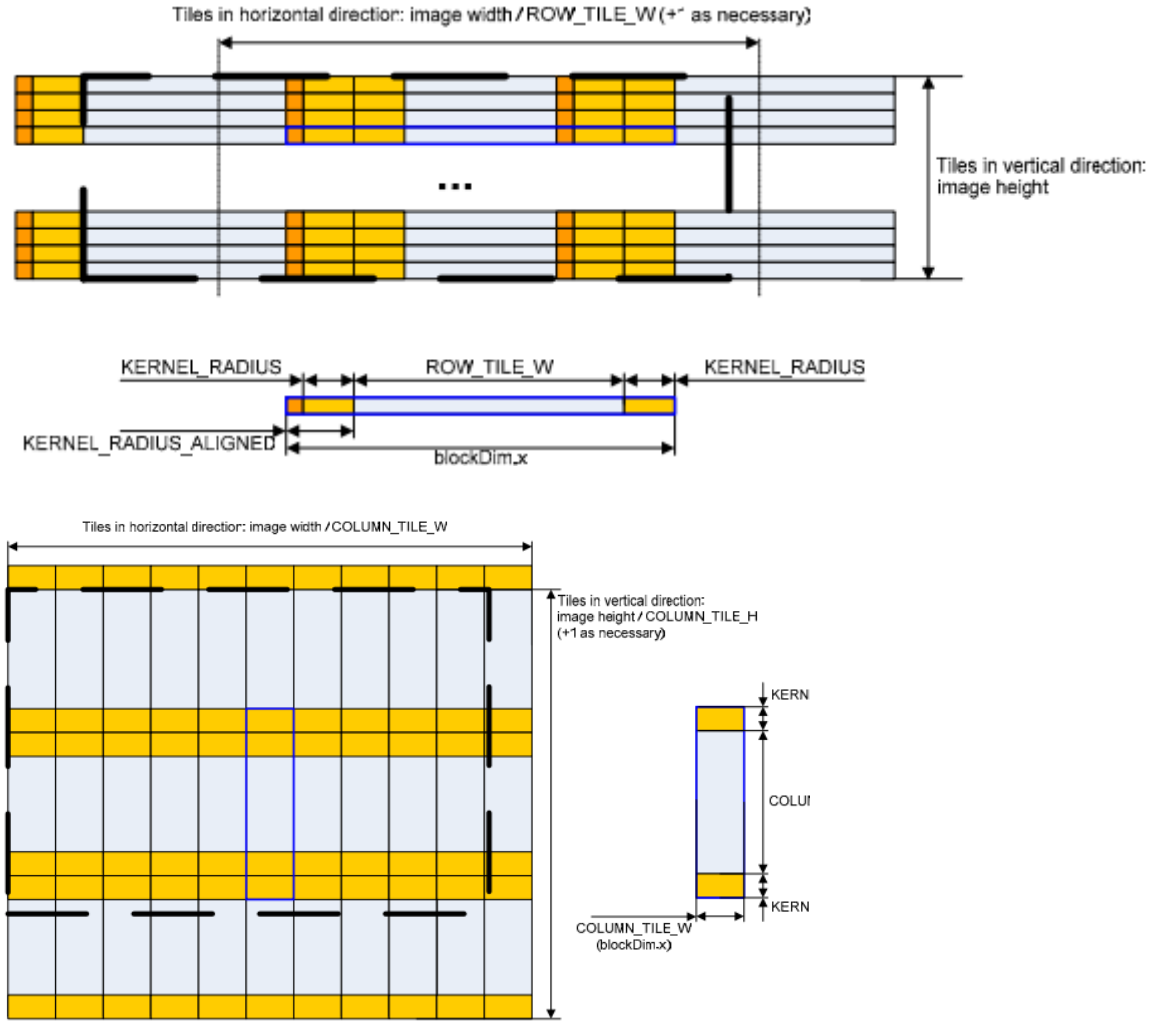
NAÏVE APPROACH VS SHARED MEMORY

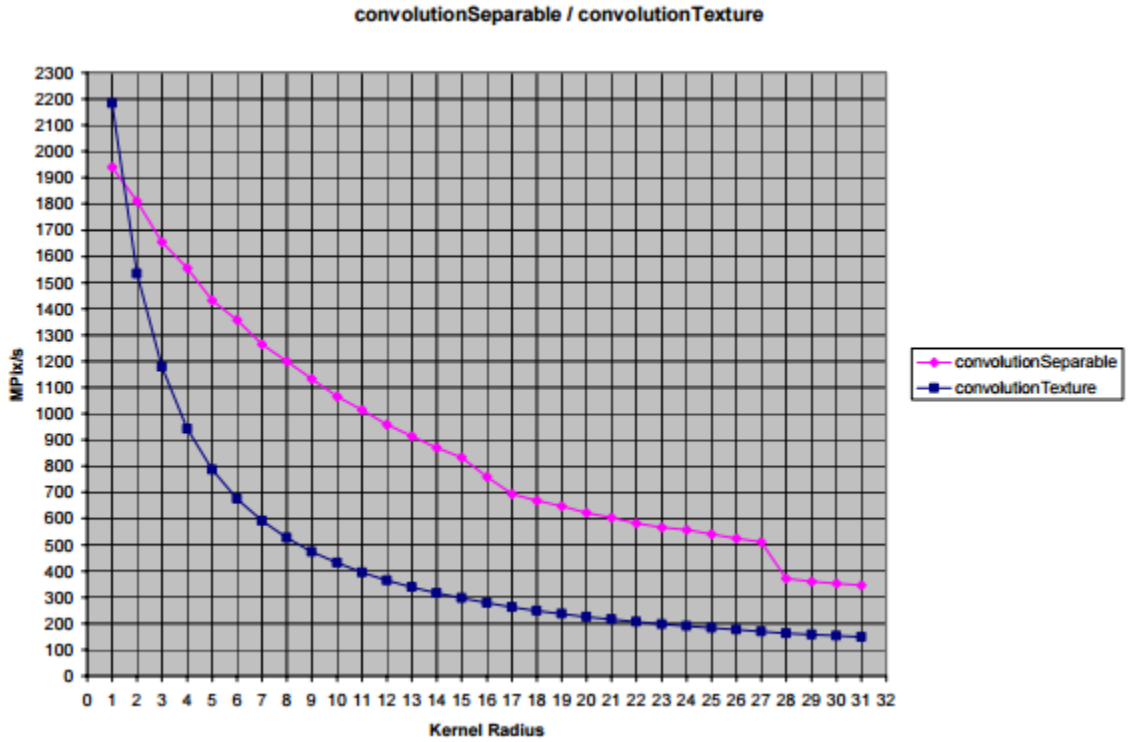






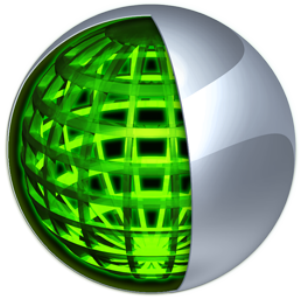
SEPARATED CONVOLUTION





CUDA DEBUGGING & PROFILING

PARALLEL NSIGHT



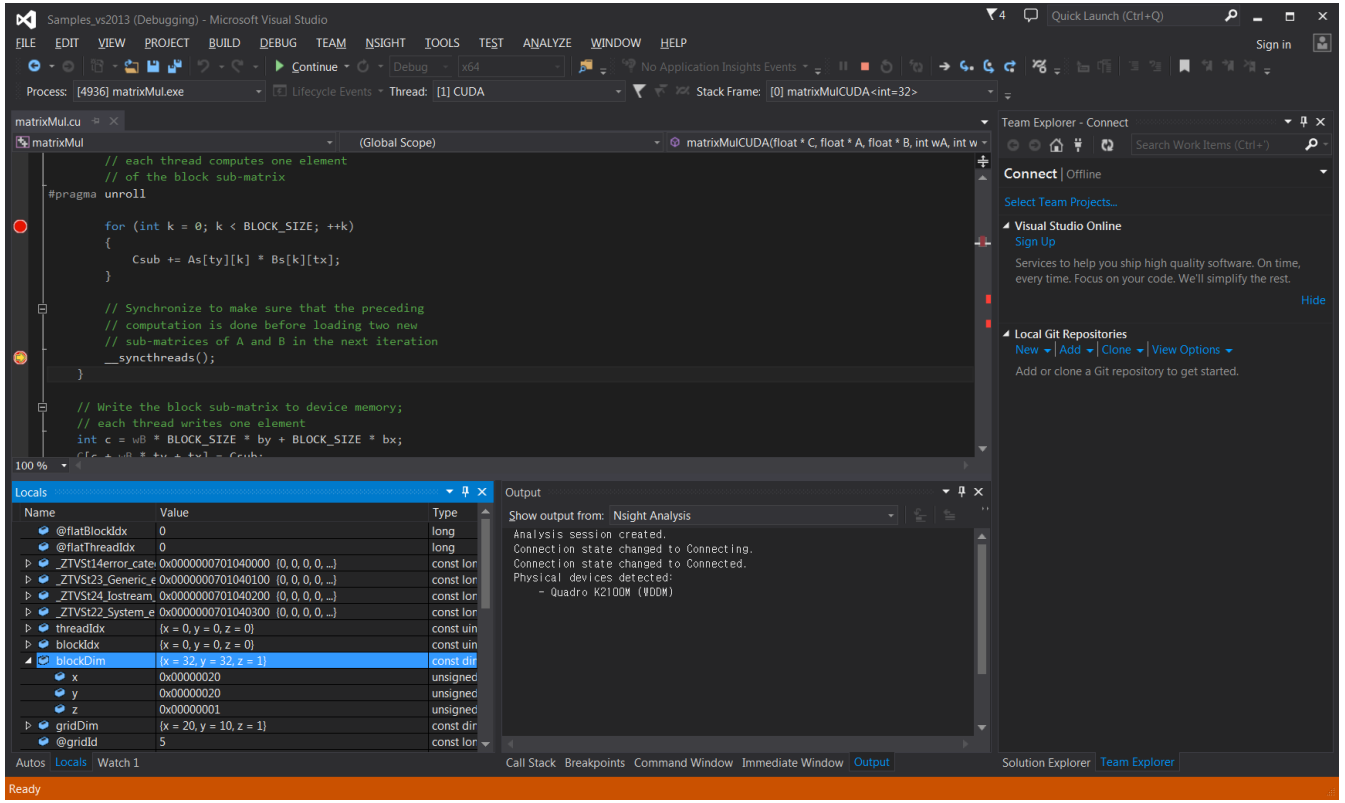
CUDA 프로그래밍 디버거

Windows

Visual Studio와 연동해서 동작

Linux / Mac

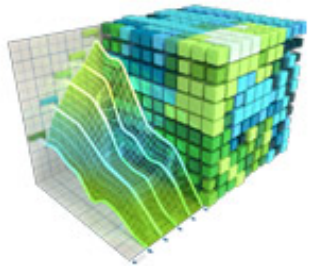
Eclipse Edition



Toolkit과 함께 설치 (무료)

모든 CUDA Thread의 모든 연산에 대하여 확인 가능

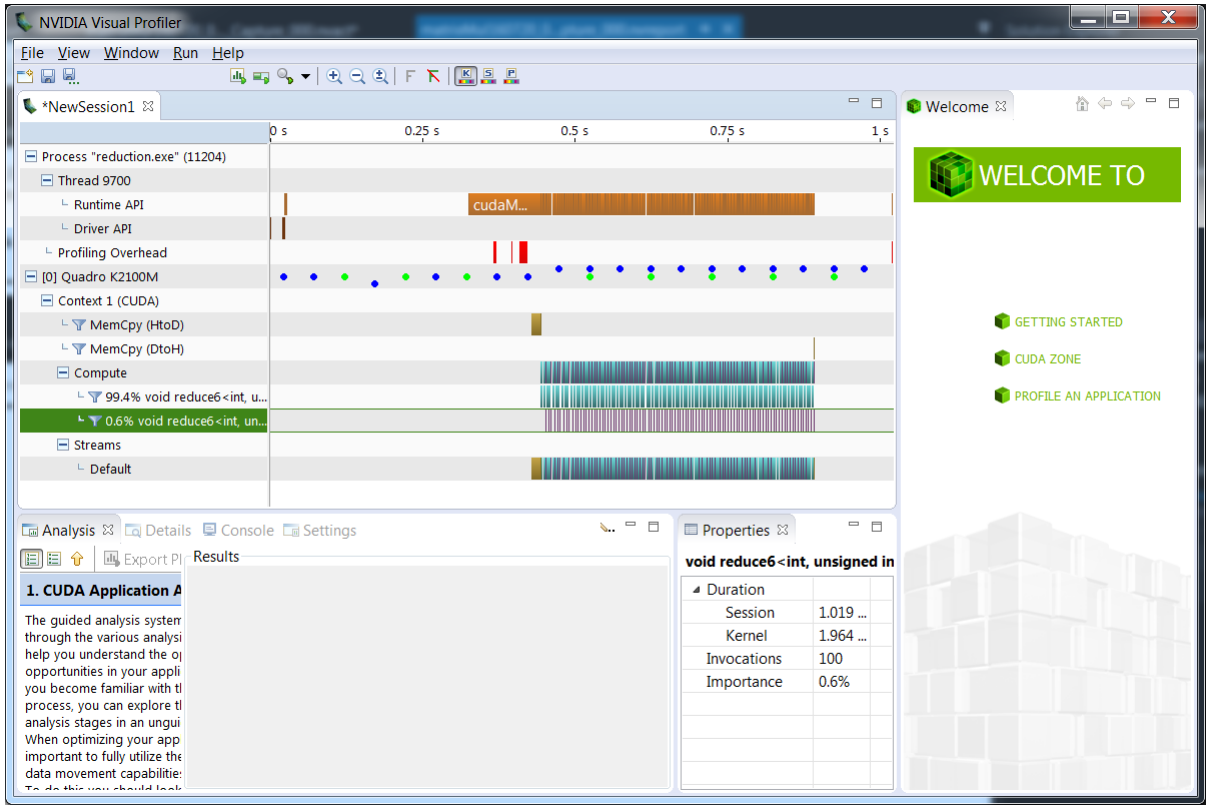
VISUAL PROFILER



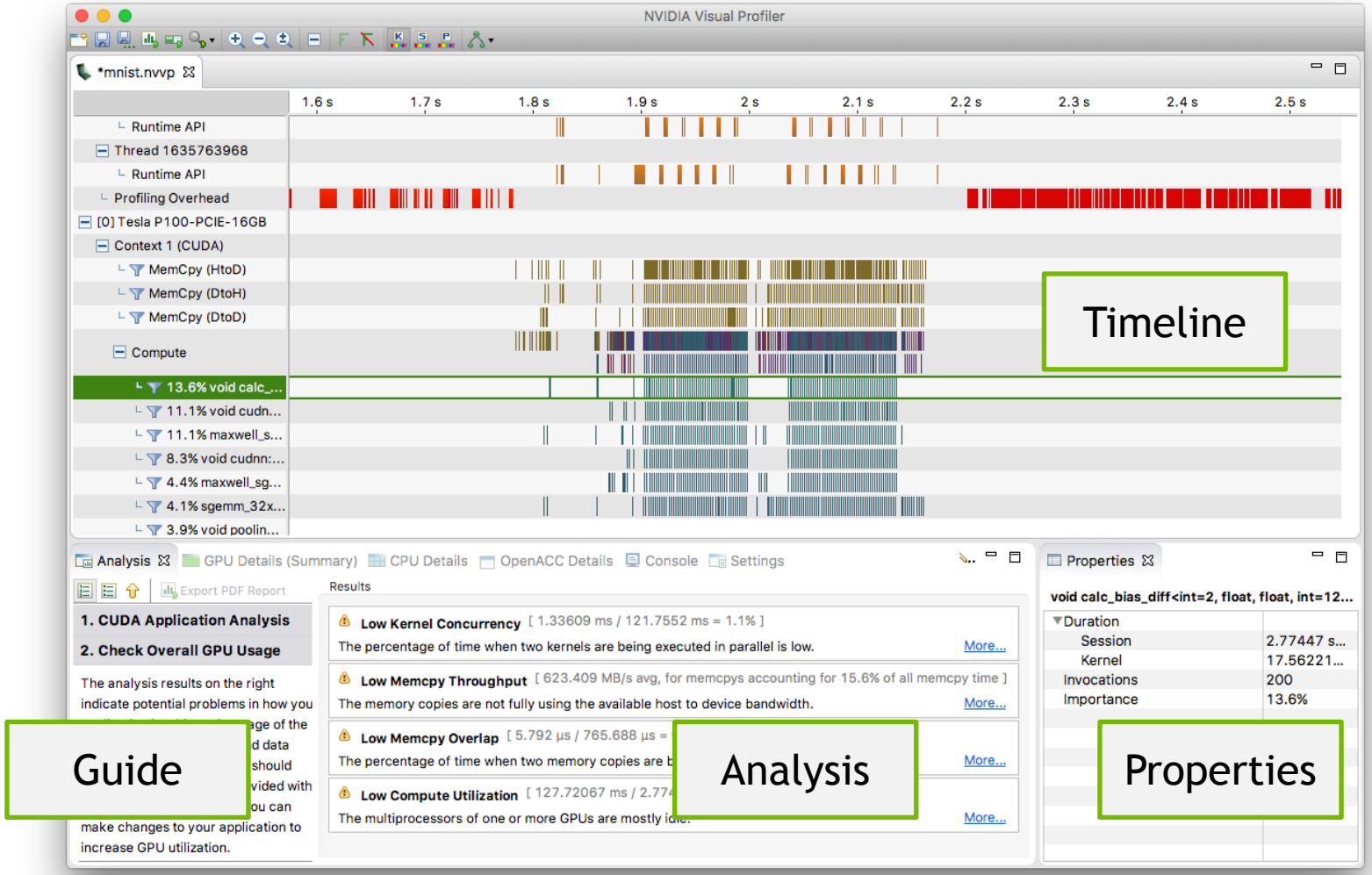
nSight eclipse edition과는 통합 사용 가능

개별적으로 실행 가능

GPU가 설치되지 않은 곳에서 리뷰 가능



PROFILER WINDOW



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

- 파일 확장자를 cu, cuh로 지정
  - 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

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);
```

printf

커널 함수에서 printf를 통해서 debugger를 거치지 않고 실행 내용 확인 가능

pseudo 값이므로 debugger를 통해 보이는 값과 부동소수점 정밀도 오차 범위 내로 차이가 있을 수 있음

DIRECTIVE PARALLEL PROGRAMMING

OpenACC

OPENMP

공유메모리 병렬 프로그래밍

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

OPENACC

Vector Processing을 지원하는 모든 코어에 동작

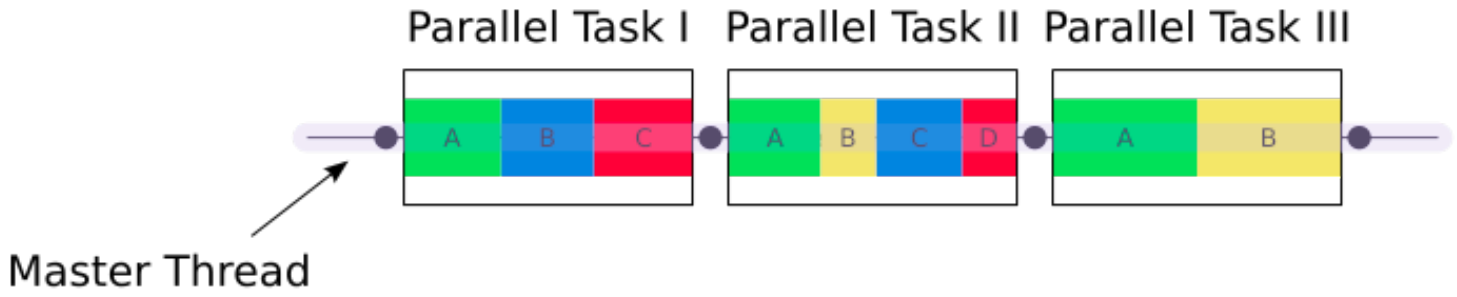
GPU, CPU, ARM, OpenPower, etc

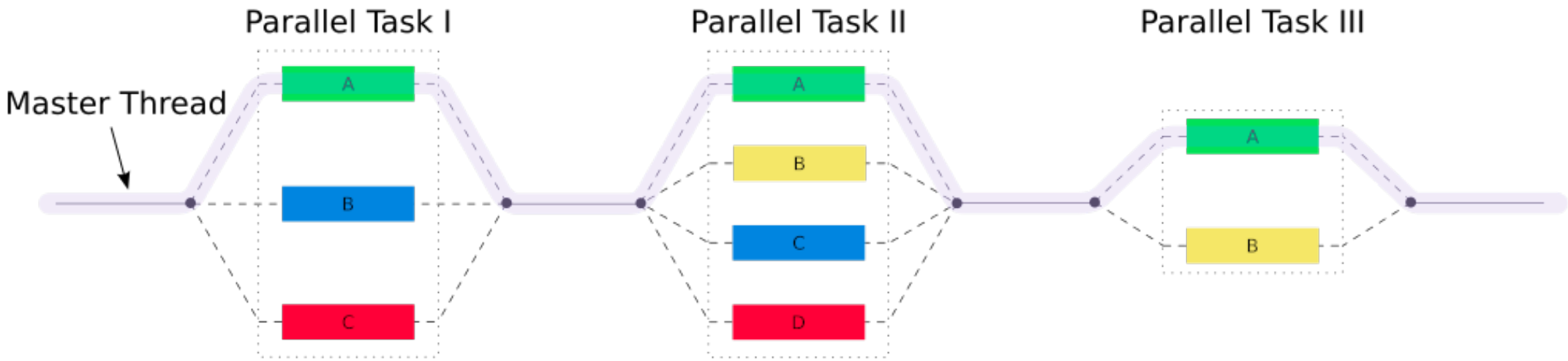
동일한 코드로 compile 옵션 변경을 통해 target 변경

Compiler 및 Profiler 무료 사용가능

PGI Compiler Community Edition
http://www.pgroup.com/products/community.htm

FORK-JOIN MODEL





NVIDIA.

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];
}
```

CUDA OPTIMIZATION

Overview

On-Chip level

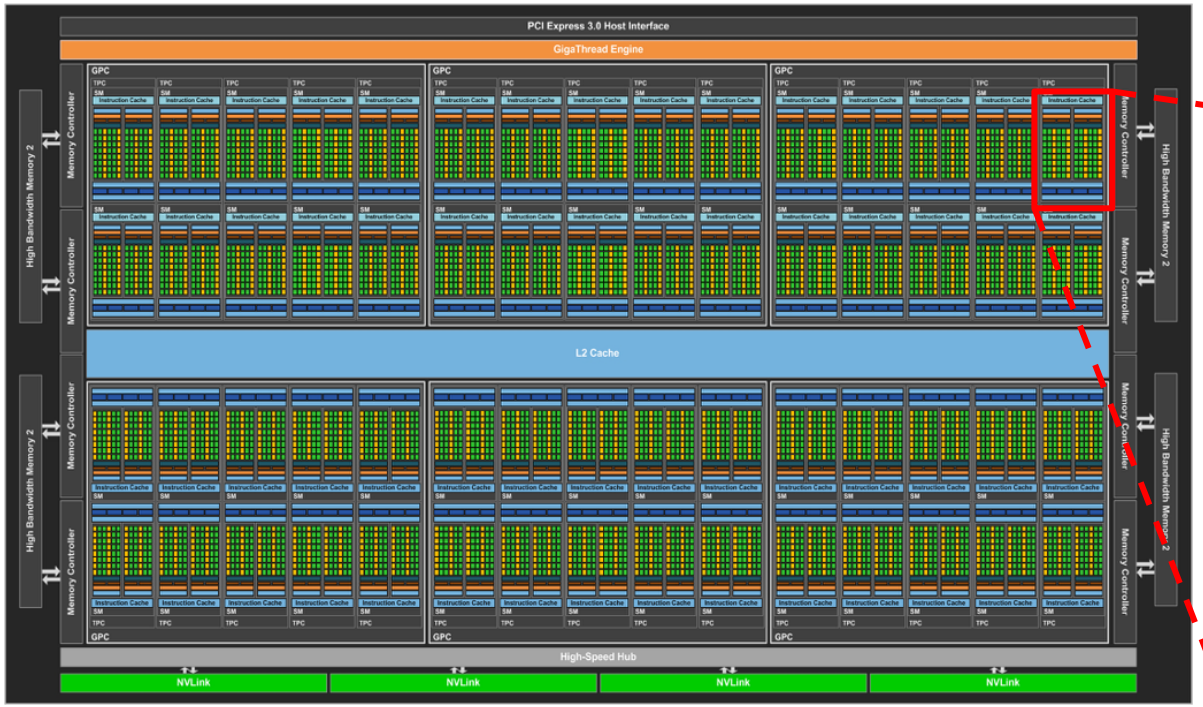
CUDA 최적화 전략

적용 대상	대역폭 최적화	병렬처리	동작 최적화
System	• Pinned Memory • Access Coalescing • Pitch Memory • Unified Memory	• Asynchronous Operation • CUDA Stream • Multi-Process Service • GPUDirect	

On-chip	• GPU Cache utilization
	  ○ Shared Memory
	  ○ Constant/Texture
	• CUDA Array
		• CUDA Occupancy
		  ○ Occupancy Calculator
		  ○ Block Size
		  ○ Register / Thread
			• Warp divergence avoiding
			• Intrinsic function

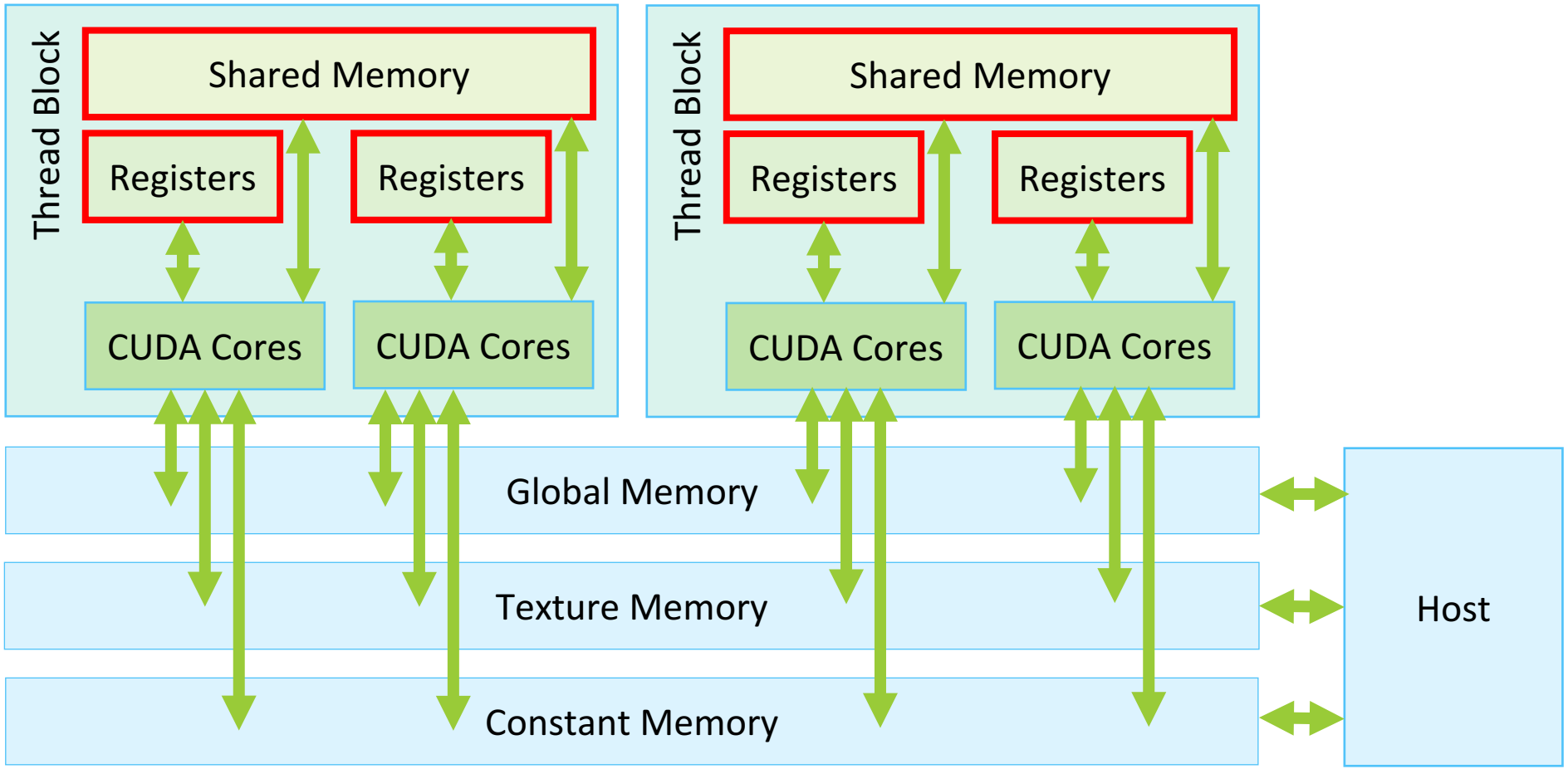
NVIDA PASCAL ARCHITECTURE

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



NVIDIA

GPU ON-CHIP 메모리



NVIDIA.

CUDA OCCUPANCY

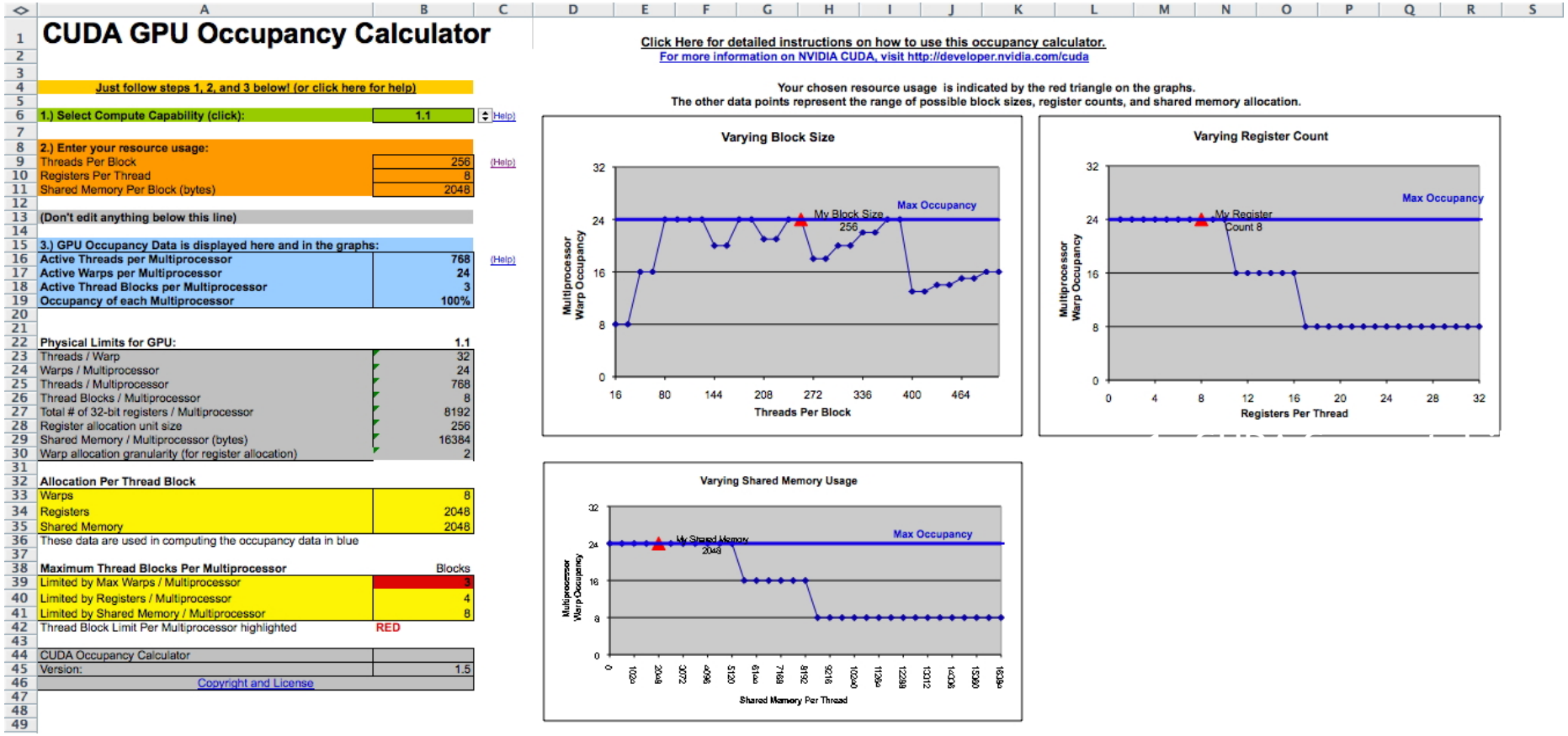
CUDA Core에 대한 SM의 자원 비율

- Register
- Shared Memory
- Active Threads
- Active Blocks

CUDA Thread block은 SM의 메모리 자원을 나눠서 실행함

CUDA SM의 자원 배분의 효율성에 대한 이론적 계산

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

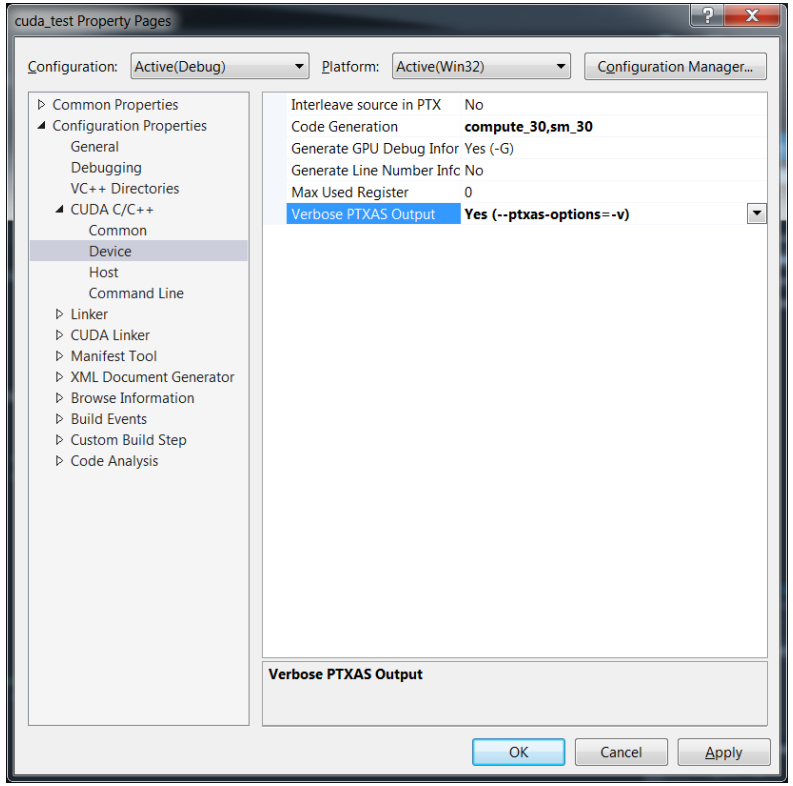
Just follow steps 1, 2, and 3 below! (or click here for help)

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.

PTX OUTPUT

Kernel 별로 사용하는 CUDA 자원의 크기 알림



CUDA SM & THREAD BLOCK

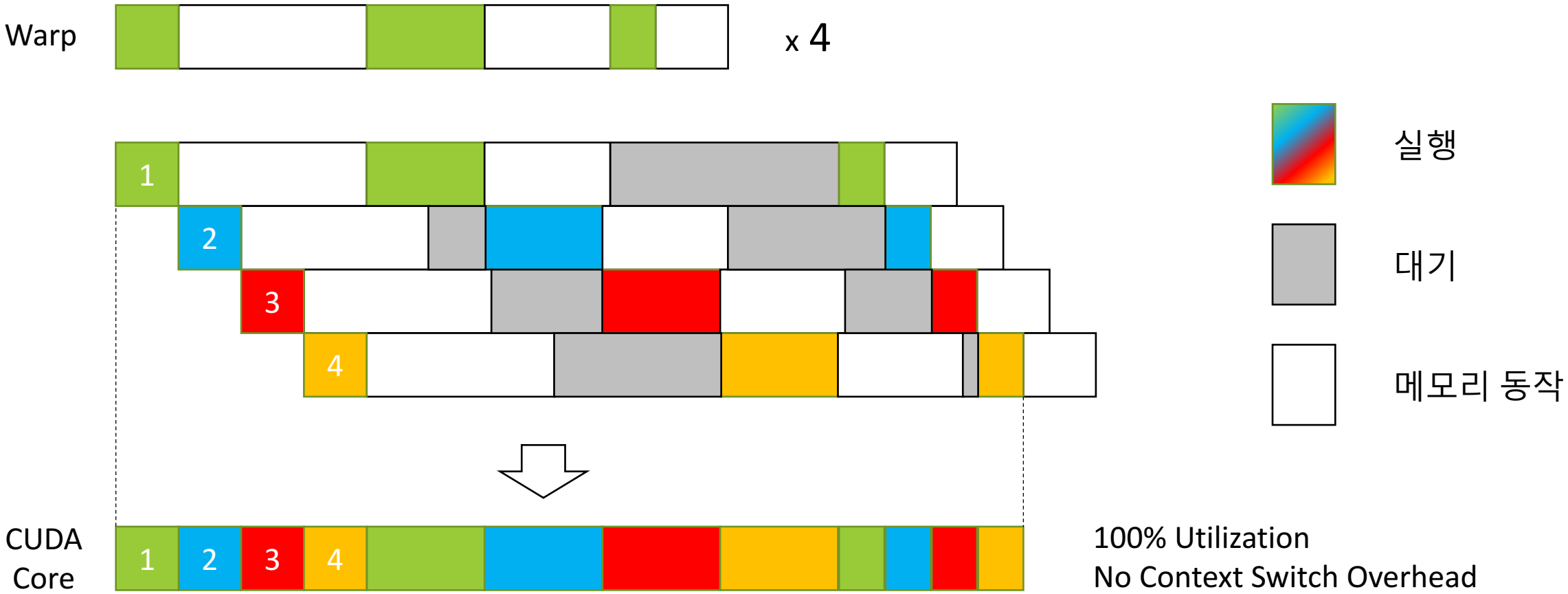
CUDA Core(SP)는 64개인데…

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

WARP SCHEDULING (REMIND)

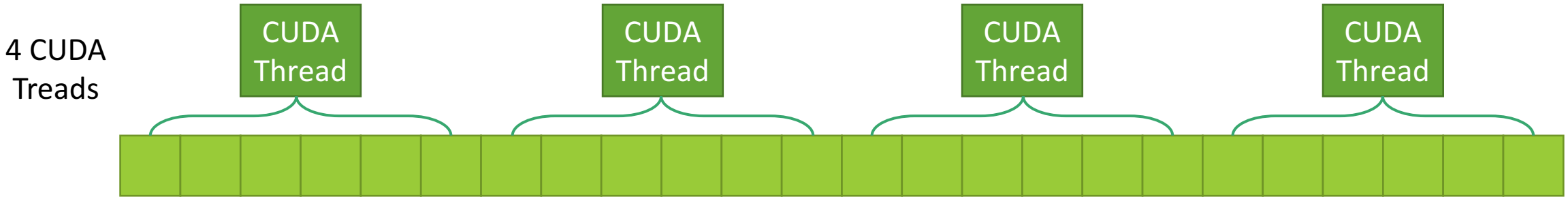


REGISTER FILE

CUDA는 SM당 256KB의 Register를 갖고 있음

CUDA Thread가 필요한 만큼 Register를 나눠가짐

- Compile 시점에 알고리즘에서 필요한 만큼을 사용할 수 있도록 정함



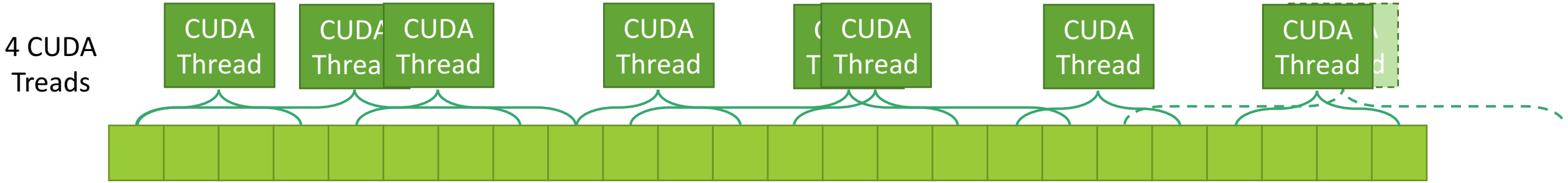
여러개의 CUDA Thread가 동시에 동작 가능

CUDA Core의 효율적인 활용 가능

REGISTERS PER THREAD

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

반대로 사용량이 작으면, 더 많은 병렬화가 가능



NVIDIA.

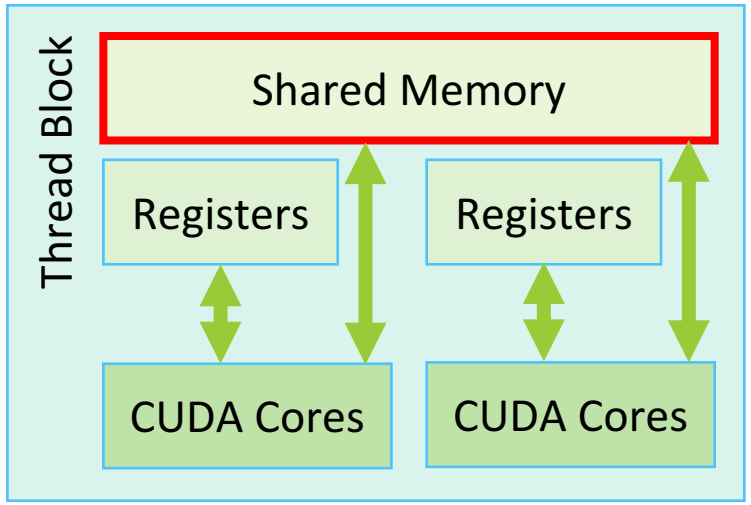
CUDA SHARED MEMORY

데이터를 재활용하기 위한 방법

공유 Cache 메모리 활용

Thread Block 내 Thread간 데이터 공유

개발자가 필요에 따라 반복 사용 빈도가
높은 데이터를 보관하는데 사용



__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

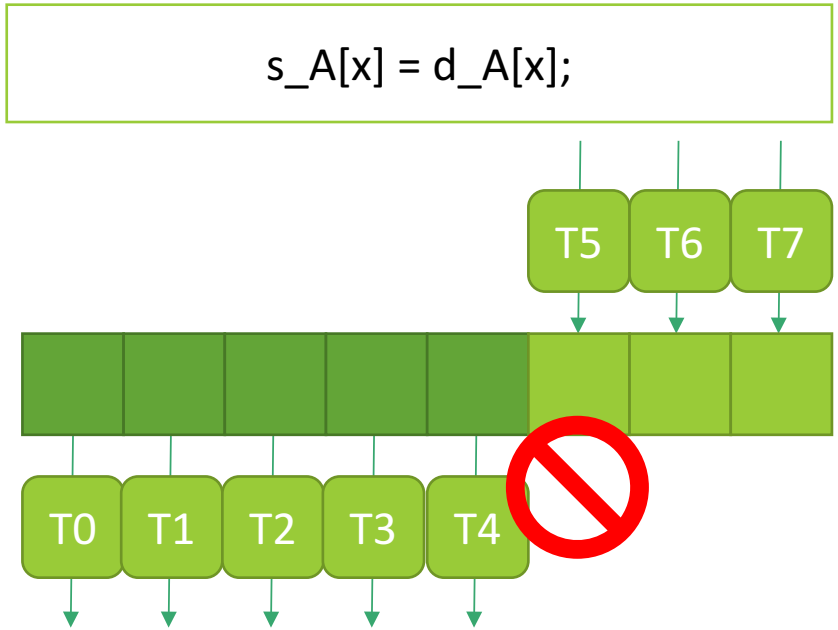
```
__shared__ float *s_A;
__global__ foo(float* B, float *A)
{
    { … }
    s_A[idx] = A[idx];
    { … }
}
```

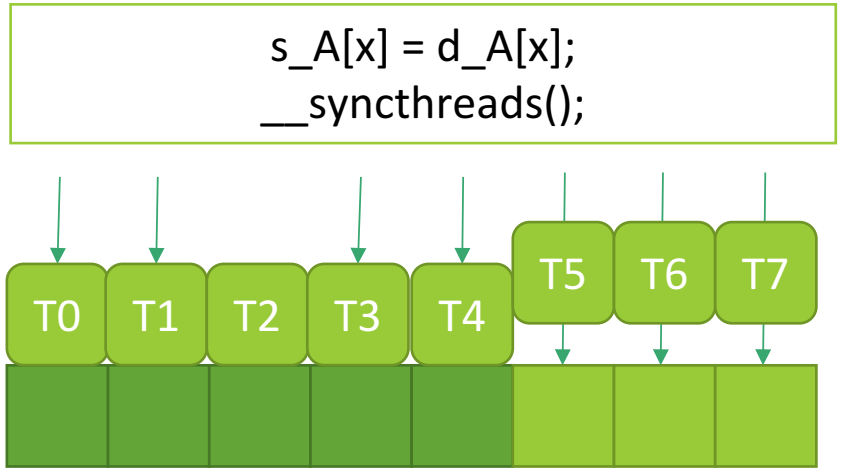
  - 필요한 만큼 공유메모리 사용
  - Kernel간 공유메모리 이름 공유

__SYNCTHREADS()

Thread block의 동기화 (Block barrier)

Shared memory에 필요한 데이터가 모두 복사되었음을 보장하는데 필요

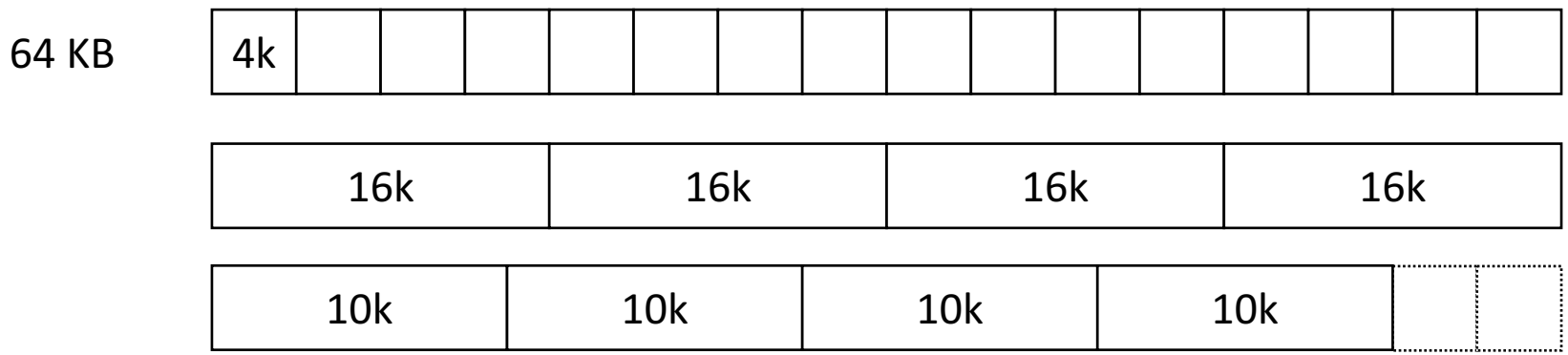




NVIDIA.

SHARED MEMORY PER BLOCK & SM

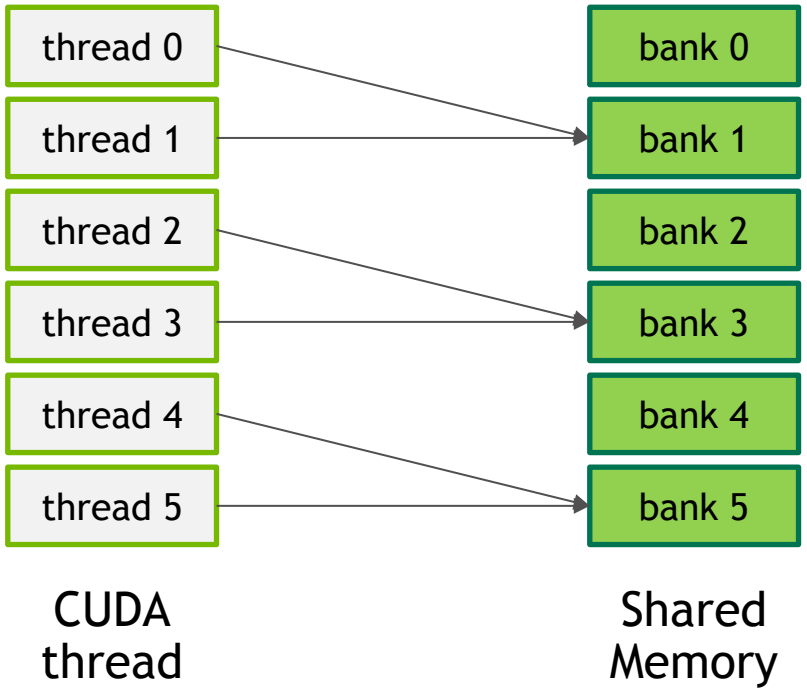
- SM의 shared memory 크기는 64 KB
- Shared memory는 thread block 내에서 공유 가능한 메모리
  - SM단위로 공유 불가
    - 어떤 index의 thread block이 같은 SM에 있을지 예측 불가
  - Thread block의 shared memory는 SM의 shared memory를 나눠 씀
    - SM에 동시에 실행 가능한 block 개수에 제한이 발생



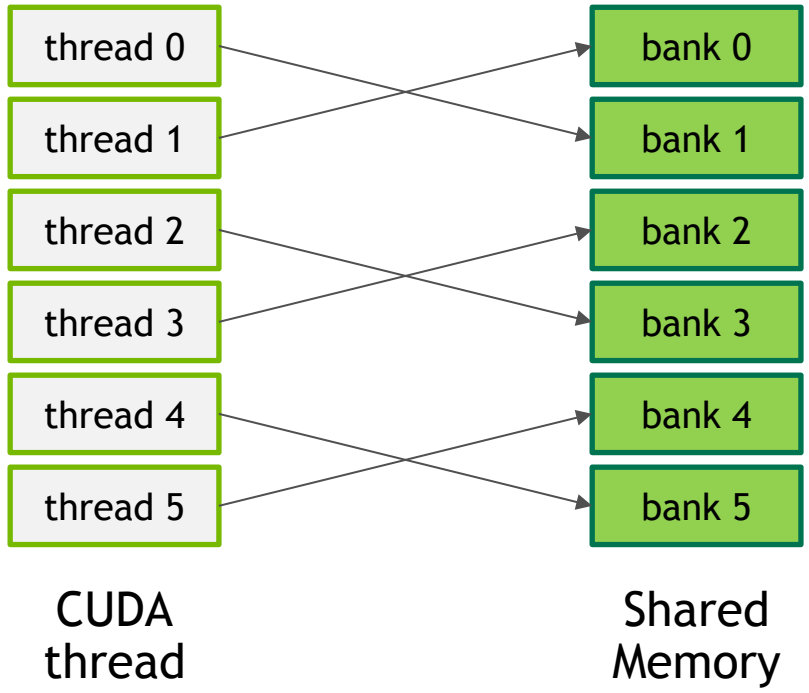
NVIDIA.

BANK CONFLICT

Shared memory의 memory bank에 복수의 CUDA thread가 access하는 경우 지연 발생



Bank Conflict



No Bank Conflict

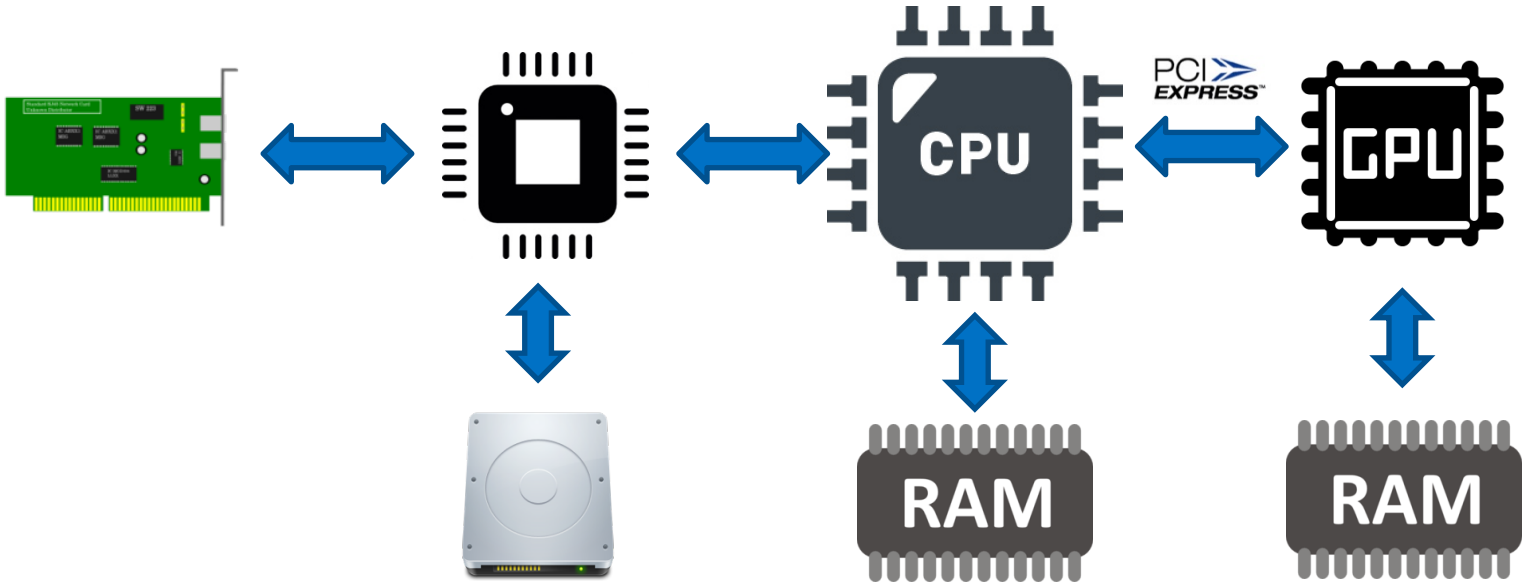
NVIDIA.

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

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

ABOUT PCI EXPRESS

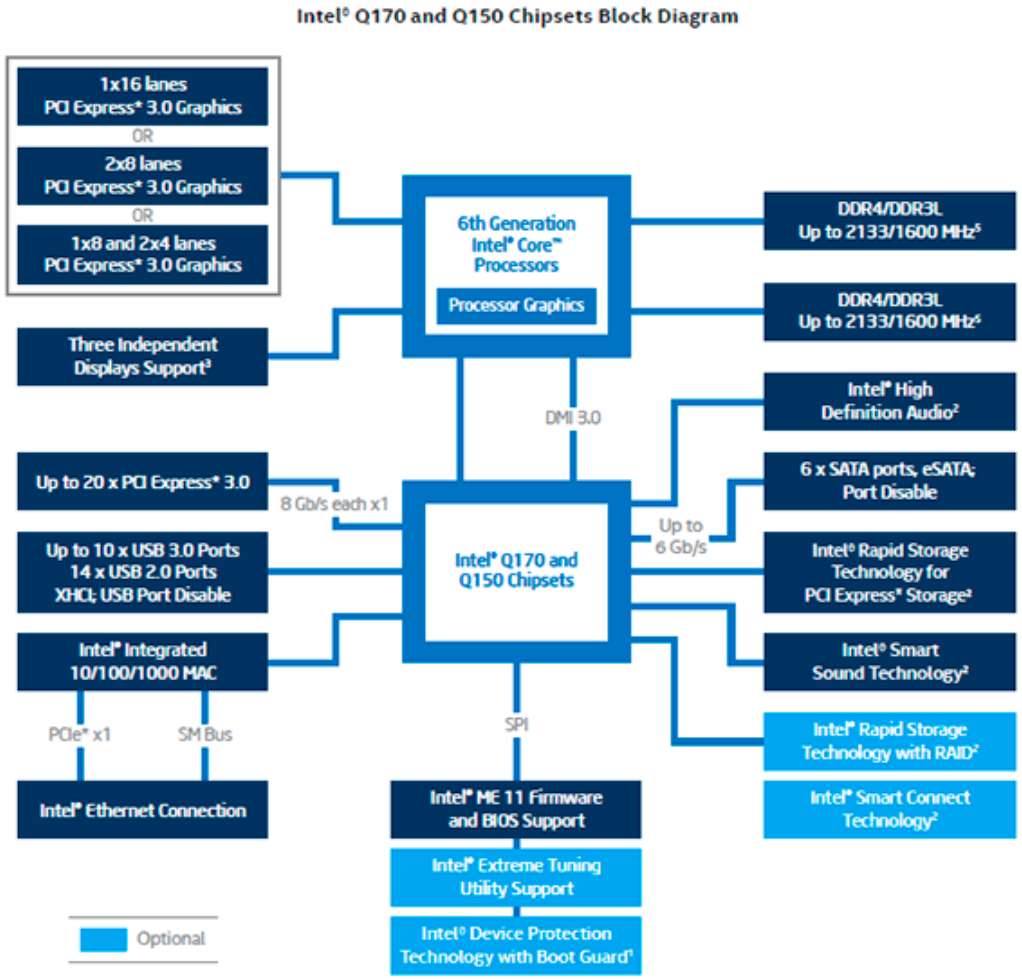
대용량 데이터 전송

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

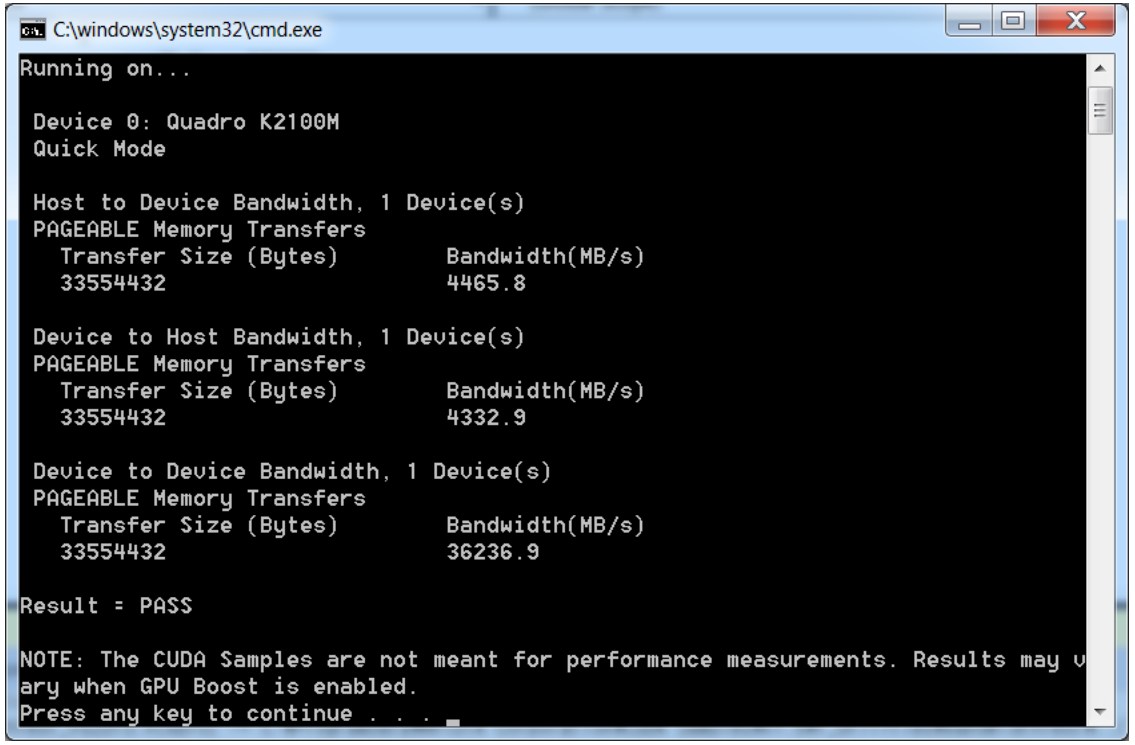
- Page Locked Memory란?
  - CPU-GPU간 Scatter-gather DMA를 할 수 있도록 구성되는 메모리
  - malloc()을 사용하여 할당한 메모리는 Page 단위로 DMA
  - 데이터 전송에서 CPU의 개입을 최소화 하여 성능 향상
- 할당
  - 모듈 내부터 활용
  - 이미 할당된 메모리의 변환

cudaHostAlloc(void* p, int_64 bufsize)	cudaHostRegister(void* p, int_64 bufsize)

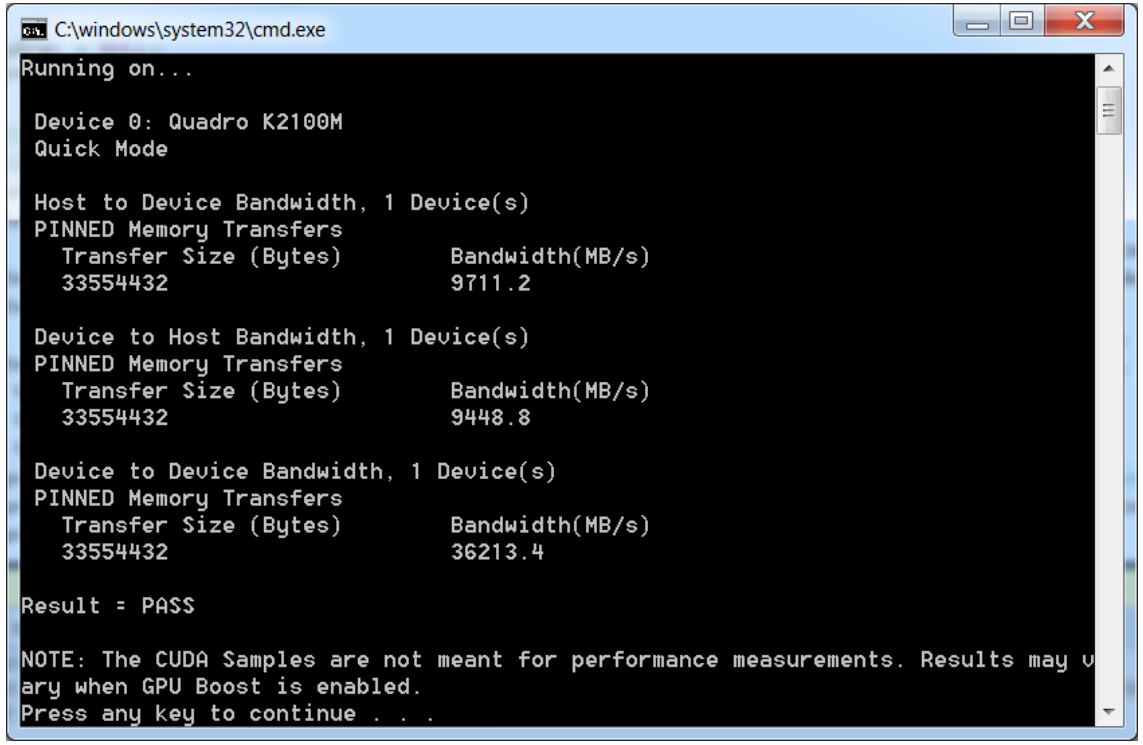
- 해제

cudaFreeHost(void* p)	cudaHostUnregister(void* p)

CUDA MEMCPY BANDWIDTH TEST

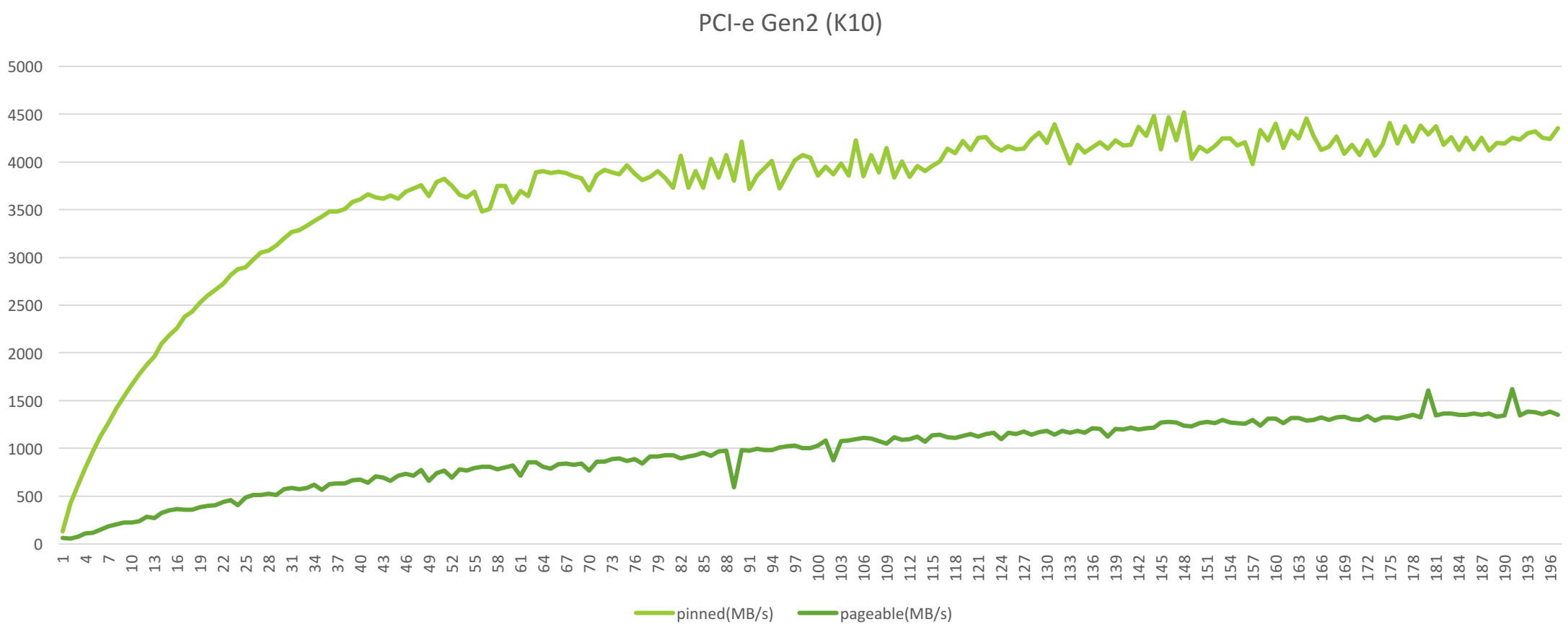


Pageable memory



Pinned memory

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

CUDA STREAM

Stream Creation and Destroy

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

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

CUDA Stream API

Kernel call with CUDA Stream

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

cudaMemcpyAsync

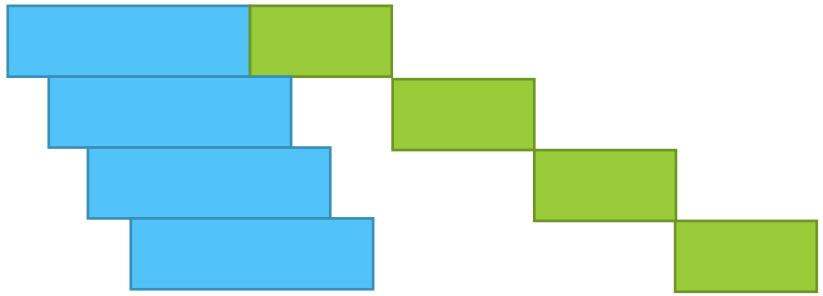
```
cudaMemcpyAsync(void* dst, void* src, int size, int mode, cudaStream_t stream = 0);
```

CUDA ASYNC OP. WITH MULTI-THREAD

Synchronous Operation

Asynchronous Operation

Synchronous Operation + Multi-thread



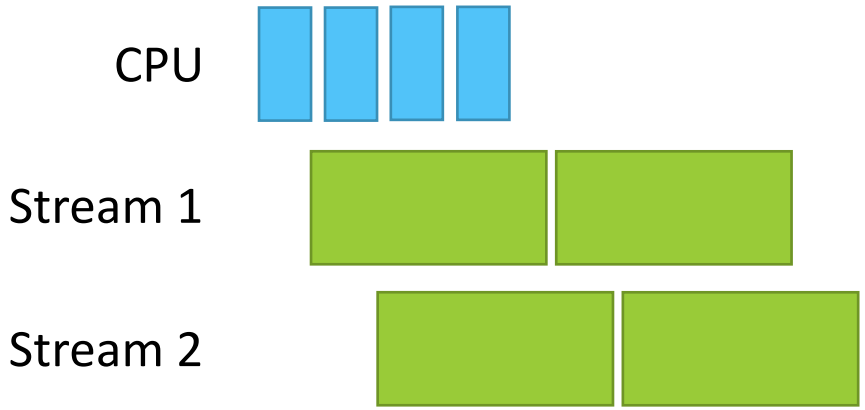
Synchronous Operation + Multi-thread + **CUDA Stream**

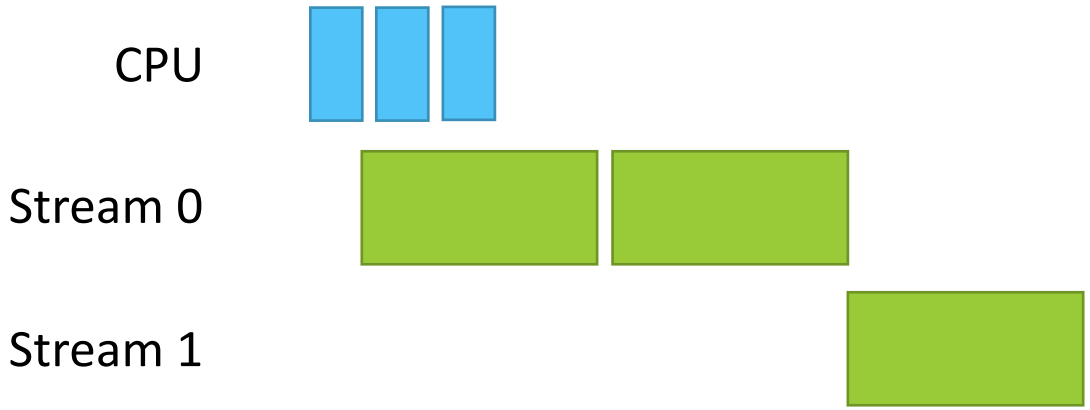
CUDA STREAM UTILIZATION

Multiple CUDA thread operation

Create CUDA Stream which you need

Default Stream (Stream 0) does not allow duplicated operation





STREAM SYNCHRONIZATION

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

Synchronization for CUDA stream

vs. cudaDeviceSynchronize() – synchronization for GPU operation

NVIDIA.

ADVANCED CUDA STREAM

Supports preemption

Getting available priority

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

CUDA Stream creation with priority

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

CUDA Stream callback function

Call back function setting with CUDA stream

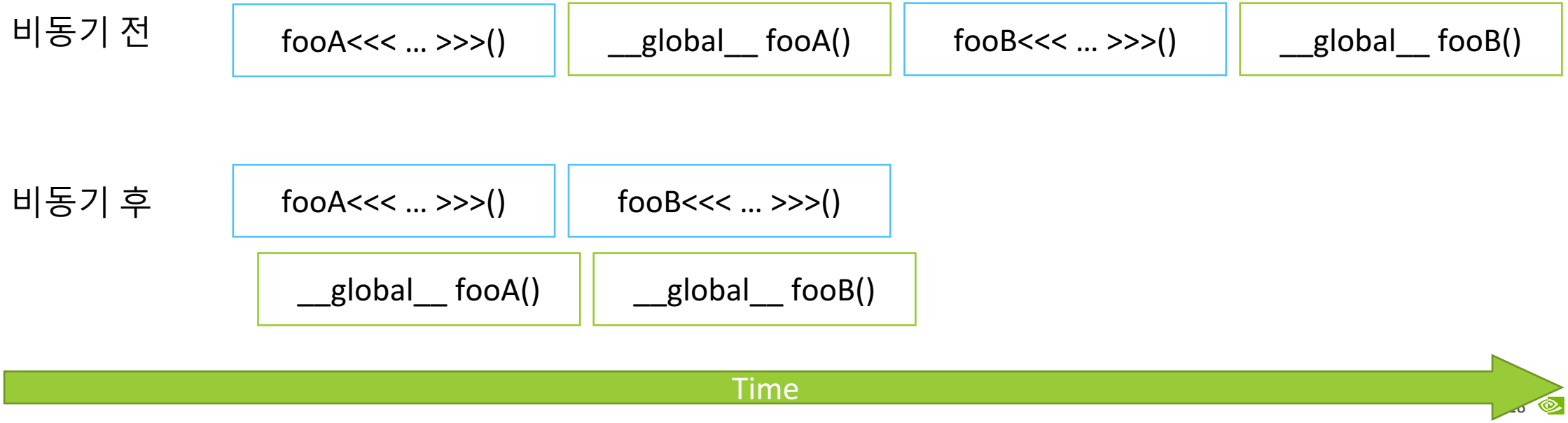
```
void CUDART_CB MyCallback(void* data)
cudaStreamAddCallback(stream, MyCallback, (void*)i, 0);
```

CPU / GPU 동작의 비동기화

Asynchronous Concurrent Execution

Host와 Kernel의 동작이 비동기로 동작

Host code와 device code가 동시에 실행 가능



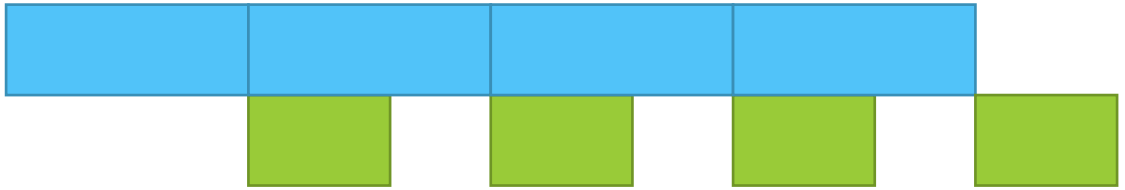
CUDA ASYNCHRONOUS OPERATION

CUDA Kernel은 원래 비동기로 동작함, cudaMemcpy의 특성상동기화발생

CUDA 비동기가 가능한 cudaMemcpy 사용

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

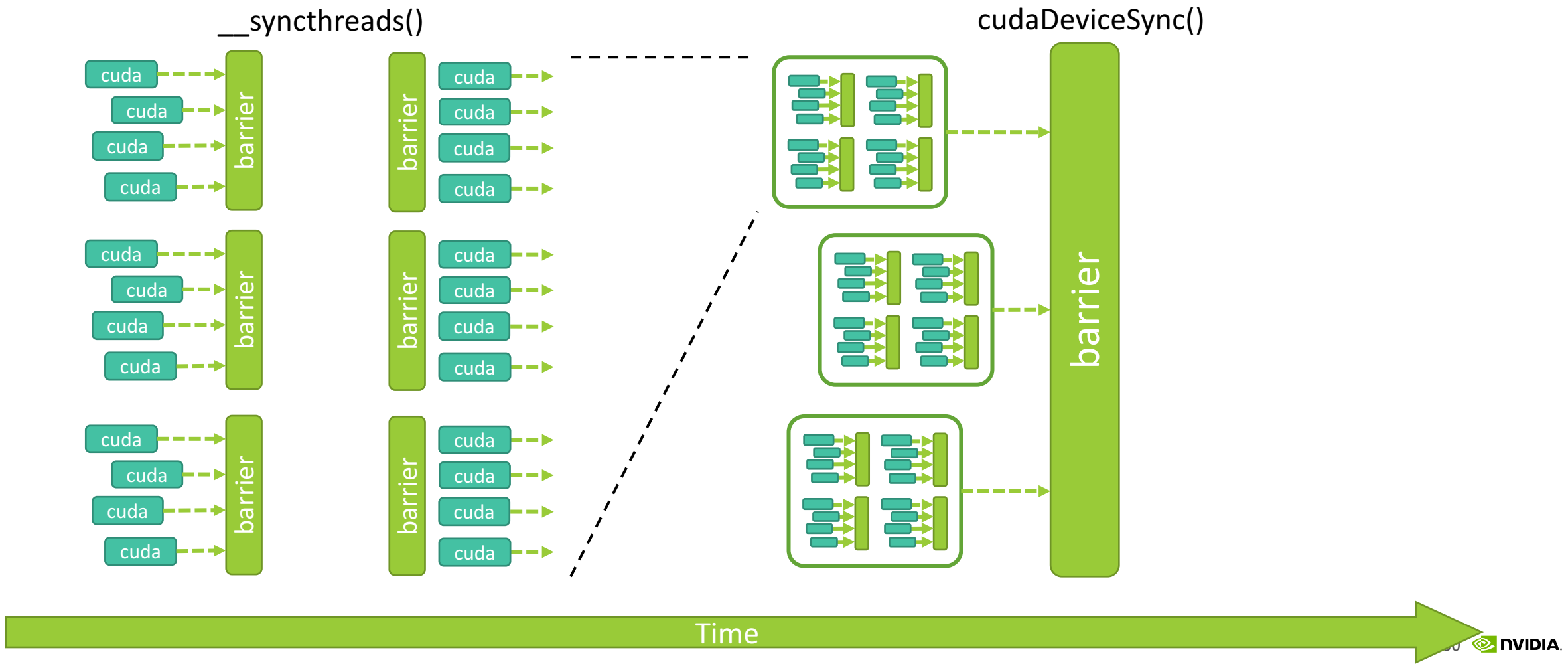
결과 데이터를 host memory로 옮기지 않는다면?



Host Memory는 반드시 Pinned Memory를 사용해야 함

CPU 간 동기화에 대한 전략도 고려해야 함

CUDA THEAD & BLOCK SYNC.

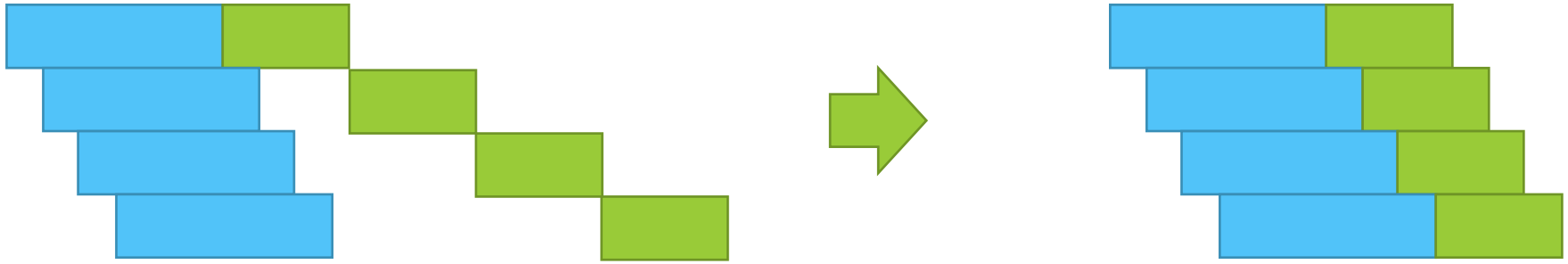


ZERO COPY

- 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);
```

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

CUDA STREAM 활용

Stream 생성 및 제거

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

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

CUDA Stream API

Kernel 호출

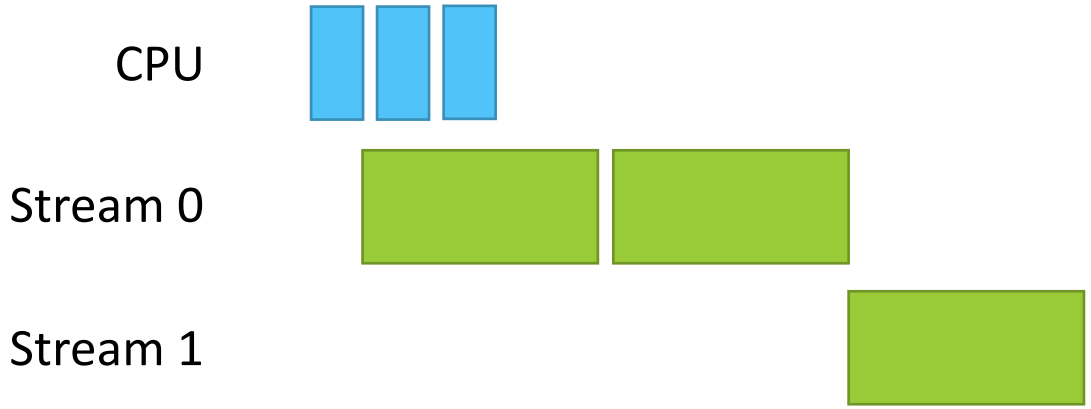
```
foo<<<dim3 gridDim, dim3 gridDim, int smem, cudaStream_t stream = 0>>>()
```

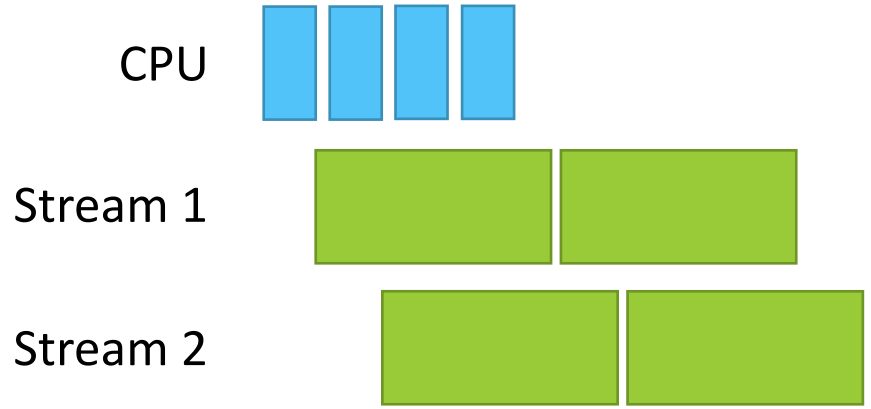
cudaMemcpyAsync

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

DEFAULT STREAM & CUDA STREAM

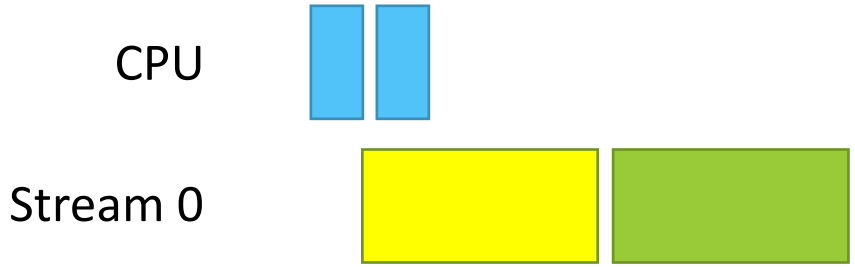
- Stream 0는 다른 Stream과 중복이 불가함
  - 모든 CUDA Stream이 동기화 됨
  - 필요한 stream 수만큼 CUDA stream 생성하여 사용
  - cudaStreamCreateWithFlags(&stream,cudaStreamNonBlocking)
    - stream 0에 대해서도 비동기 동작 가능
- 동일한 CUDA stream에서는 순차적으로 실행

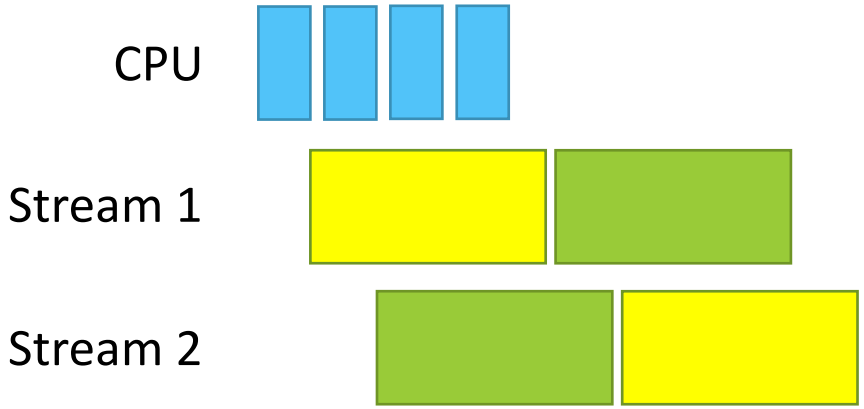




CUDA STREAM & cudaMemcpyAsync

cudaMemcpyAsync를 활용하여 GPU 동작 중 데이터 복사 가능

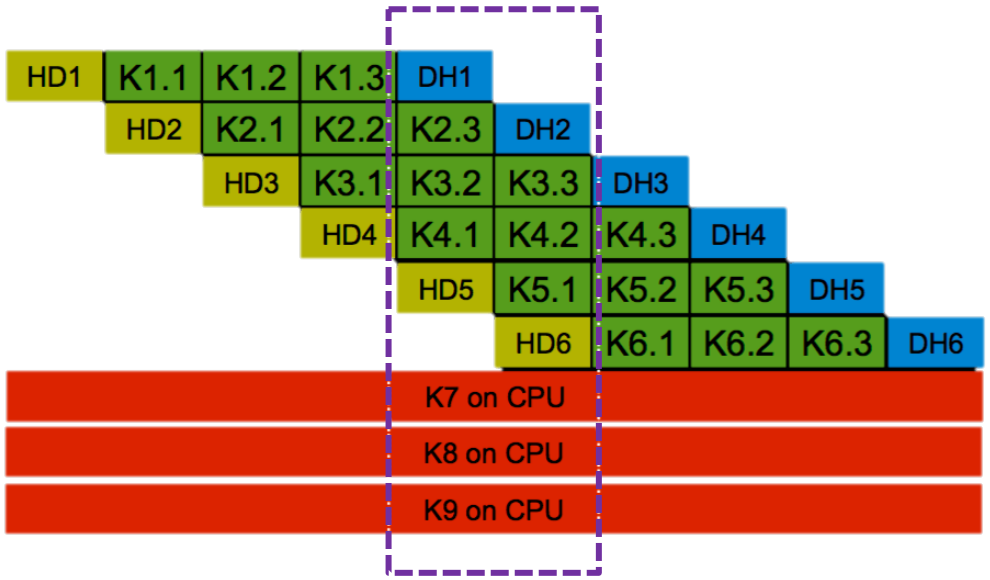




데이터 복사에 따른 지연시간을 줄일 수 있음

NVIDIA.

이런 오해가 있을 수 있습니다.



- GPU Core가 Kernel 별로 할당되는 구조가 아님
- CUDA Stream에 맞춰서 multi-thread 처럼 동작 할 수 있음

STREAM SYNCHRONIZATION

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

CUDA stream에 대하여 동기화 가능

cudaDeviceSynchronize()는 GPU의 모든 stream에 대하여 동기화

ADVANCED CUDA STREAM

우선순위 부여

가능한 Priority 값 알기

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

Priority를 가진 CUDA Stream 생성

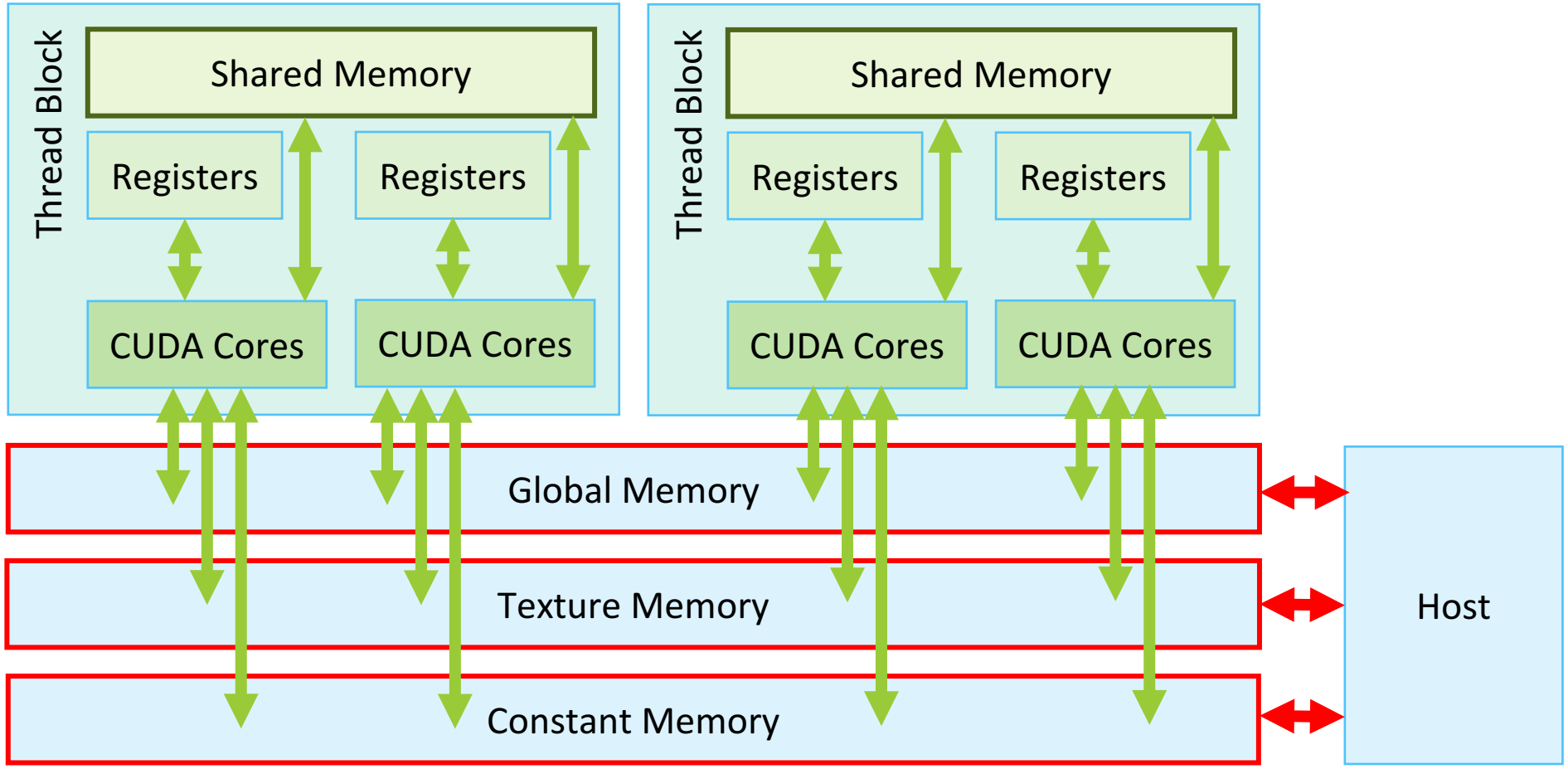
```
cudaStreamCreateWithPriority(&stream, flags, priority)
```

Stream Callback 함수

Stream에 의존한 Host 함수 호출 가능

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

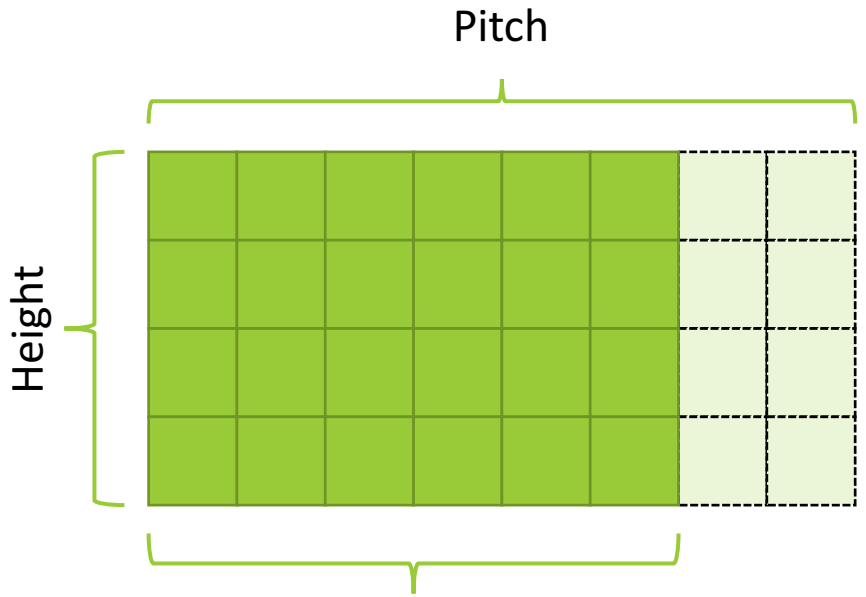
GPU 메모리


Thread Block
Shared Memory
Registers
Registers
CUDA Cores
CUDA Cores
Thread Block
Shared Memory
Registers
Registers
CUDA Cores
CUDA Cores
Global Memory
Texture Memory
Constant Memory
Host

GLOBAL MEMORY 최적화

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

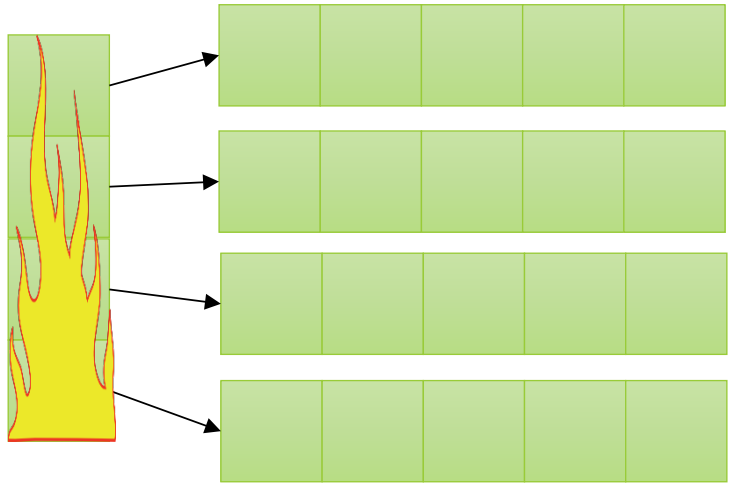
- 2차원 배열에 대한 최적화
  - cudaMallocPitch/Async()
    - Global Memory에 대한 자동 정렬
  - cudaMemcpy2D/Async()
    - Pitched 메모리로 데이터 복사시 사용



DEVICE MEMORY 사용시 POINTER 문제

- 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
- Structure내의 pointer 변수가 있다면, pointer가 가르키는 메모리에 대해서도 GPU메모리 할당 필요
- `struct` 는 linear buffer로 이슈 없음

COALESCING MEMORY ACCESS

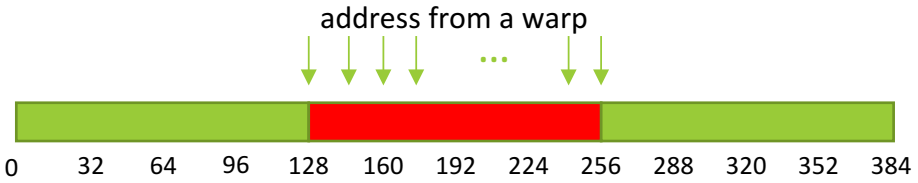
CUDA Core는 Global메모리에 접근시 block으로 access함

사용시 128 byte 단위 access

Cache 미사용시 32 byte 단위 access

Nvcc에 Xptxas -dlcm=cg 옵션으로 L1 Cache off 가능

Coalesced Memory Access

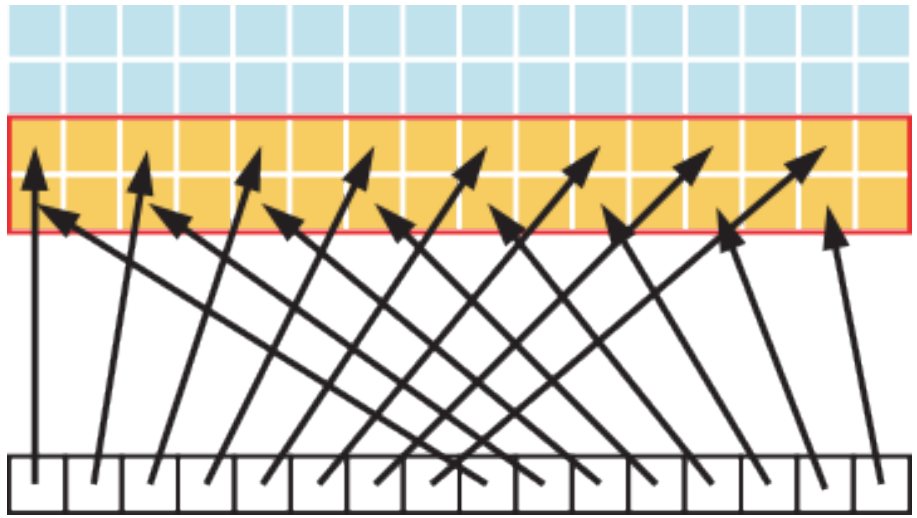


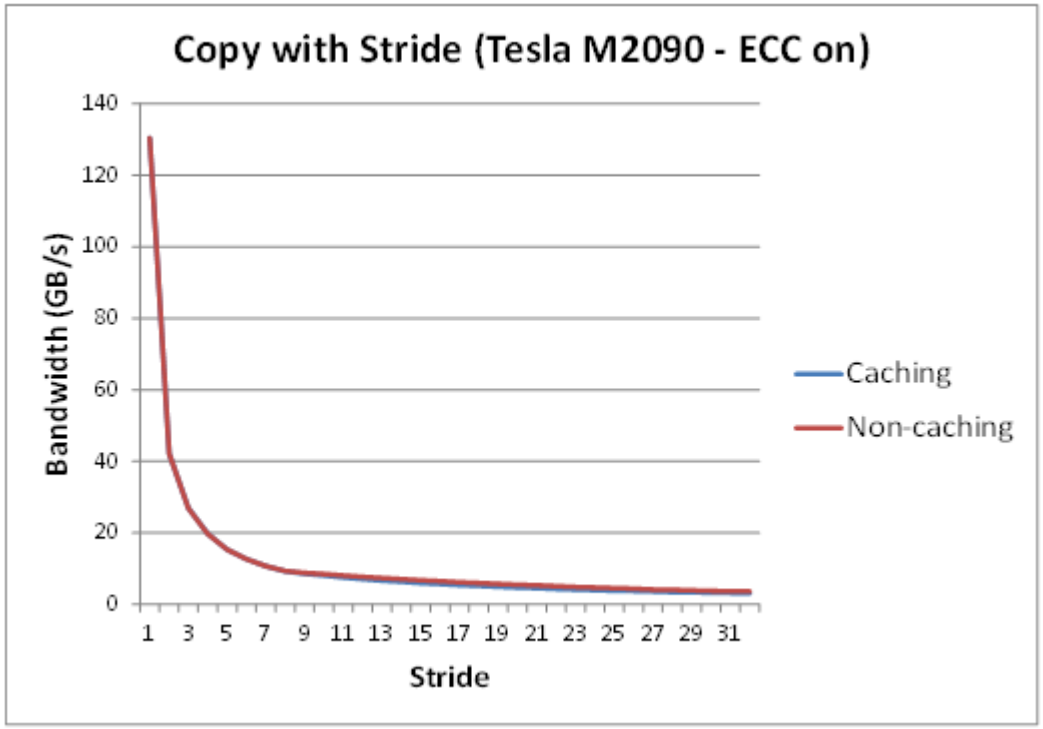
Misaligned Memory Access

Misaligned Memory Access with L1 Cache-off

STRIDED ACCESS

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





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

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

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

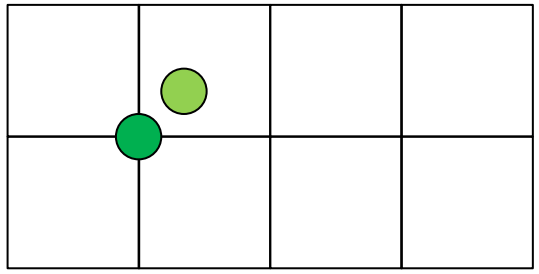
```
texture<float, 3, cudaReadModeElementType> texRef;
cudaChannelFormatDesc channelDesc =
                        cudaCreateChannelDesc <32, 32, 32, 32, cudaChannelFormatKindUnsigned);
cudaBindTextureToArray(texRef, cudaArray, channelDesc);
```

TEXTURE FILTER MODE

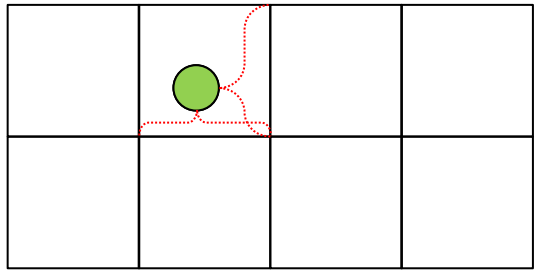
하드웨어 설정으로 Texture Address 모드 선택

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

Filter Mode



cudaFilterModePoint

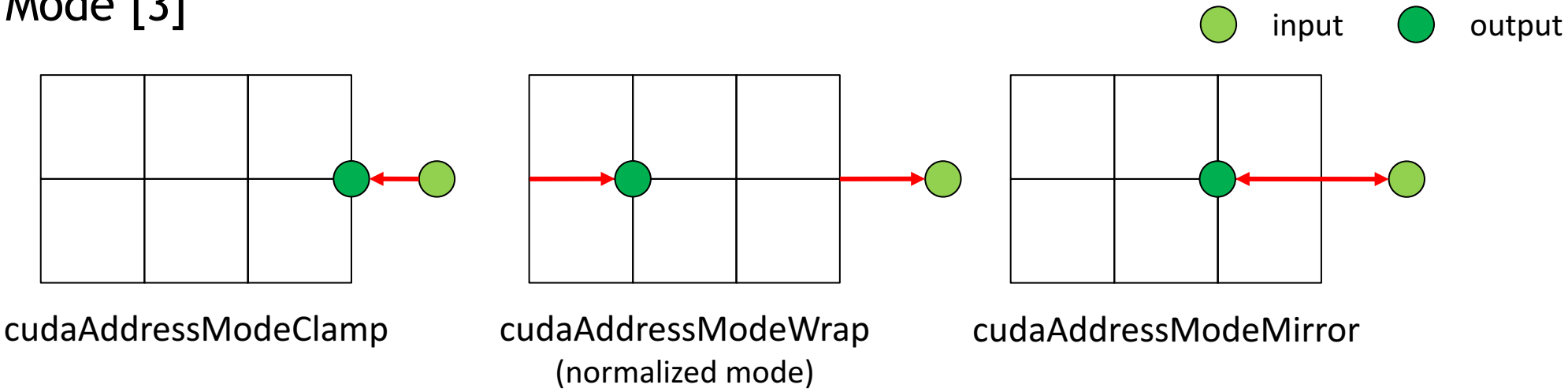


cudaFilterModeLinear

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

CONSTANT MEMORY

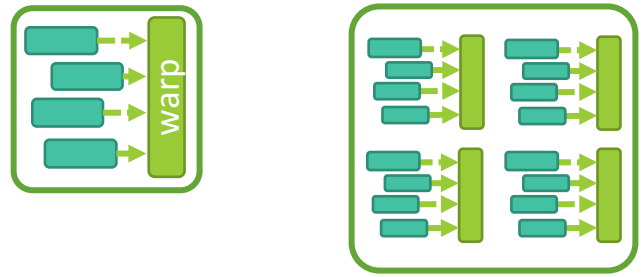
- 읽기 전용
  - Kernel 실행 전 Data 복사
  - 새로 덮어쓰기 전까지 Application 내에서 데이터 보존
- API
  - 선언
    - __constant__ float table[N];
  - cudaMemcpyToSymbol() / cudaMemcpyFromSymbol()
- 크기가 작음 (64 KB)

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

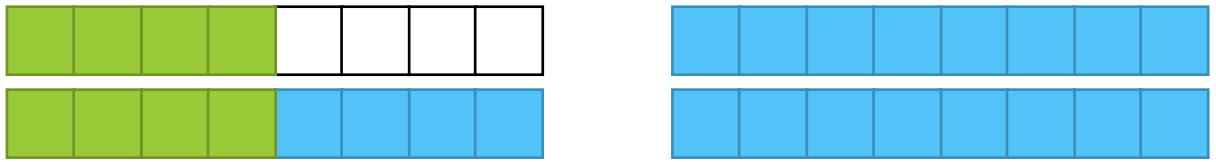
WARP DIVERGENCE

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

A[idx] = idx < 64 ? : ;

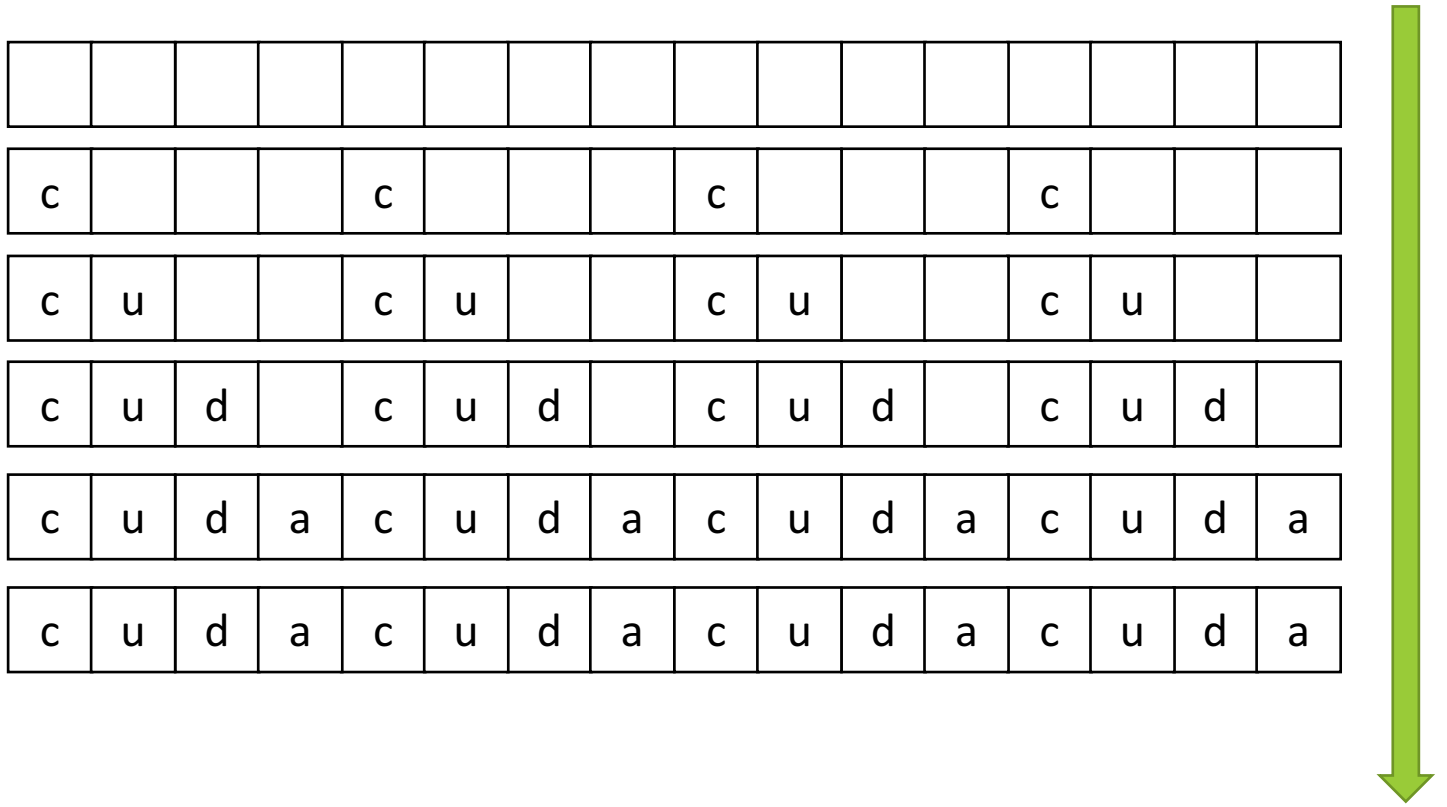
A[idx] = idx < 32 ? : ;

A[idx] = idx < 16 ? : ;



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’;
}
```



연산량 최적화

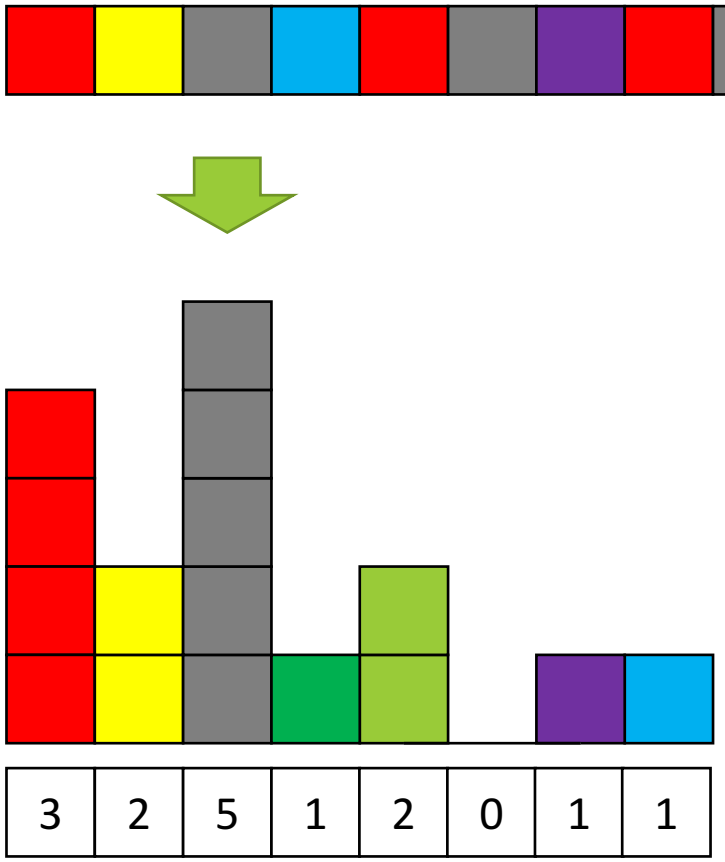
- Half Precision, Single Precision, Double Precision 최적화
  - 세부 지원 내용은 GPU 버전 및 종류마다 다름
  - 적절한 Architecture 선택 및 Precision이 낮을 수록 연산량 높음
  - HP의 경우 성능향상 가능한 GPU는 Tesla 만 지원
- Intrinsic function
  - Short, Char 데이터에 대한 SIMD 지원
  - 유효숫자를 줄인 round off 연산 가능 (오차범위 확인 필요)
- 2^n으로 나머지(modular) 연산을 하는 경우,

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

PARALLEL PROGRAMMING PATTERN

Advanced CUDA Programing

HISTOGRAM




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

결과예측 불가

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

예측한 결과

CUDA ATOMIC OPERATION

CUDA thread의 atomic 동작 보장

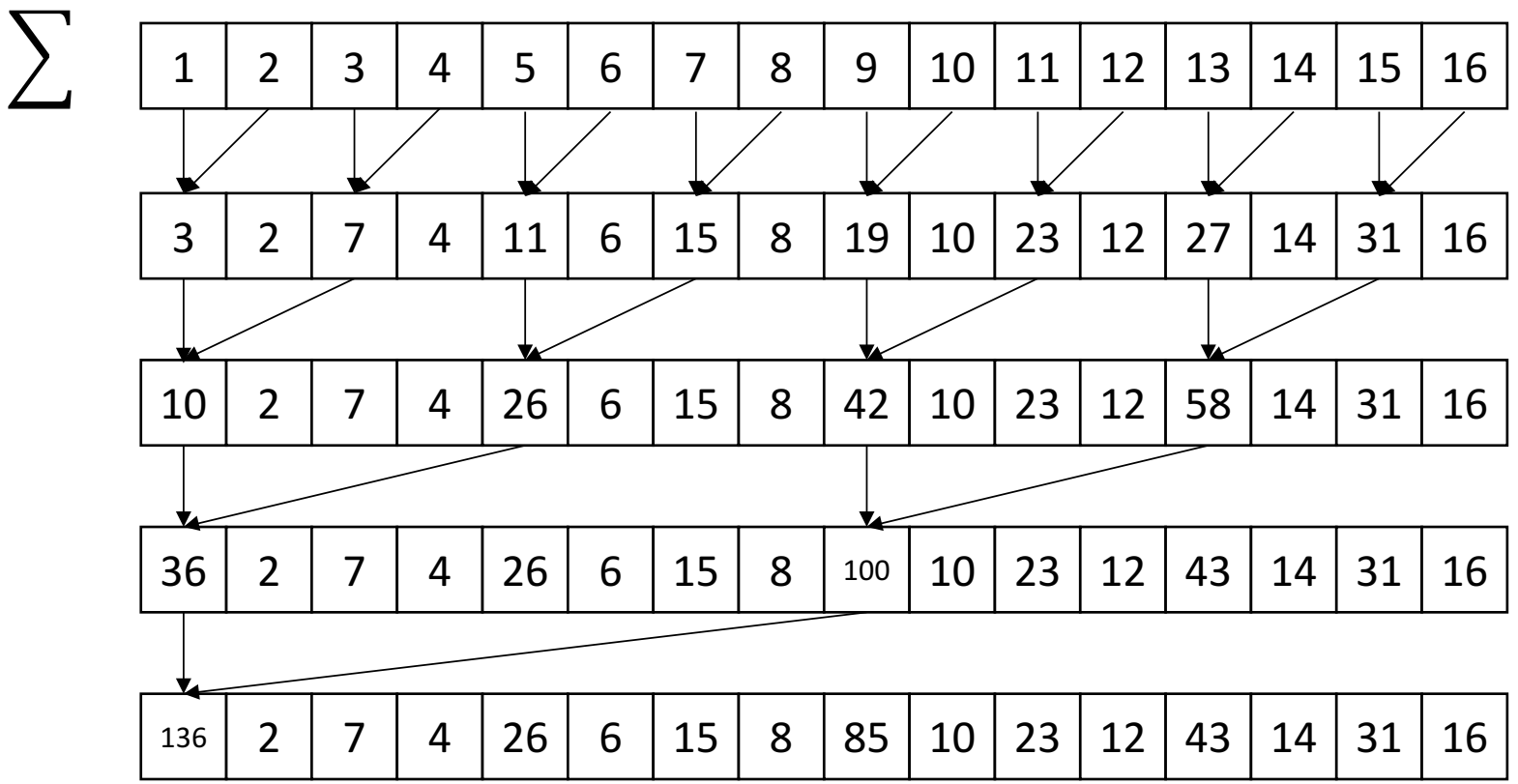
여러 Thread가 하나의 메모리 값을 갱신할 때 예상하는 값을 보장

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);
…
```

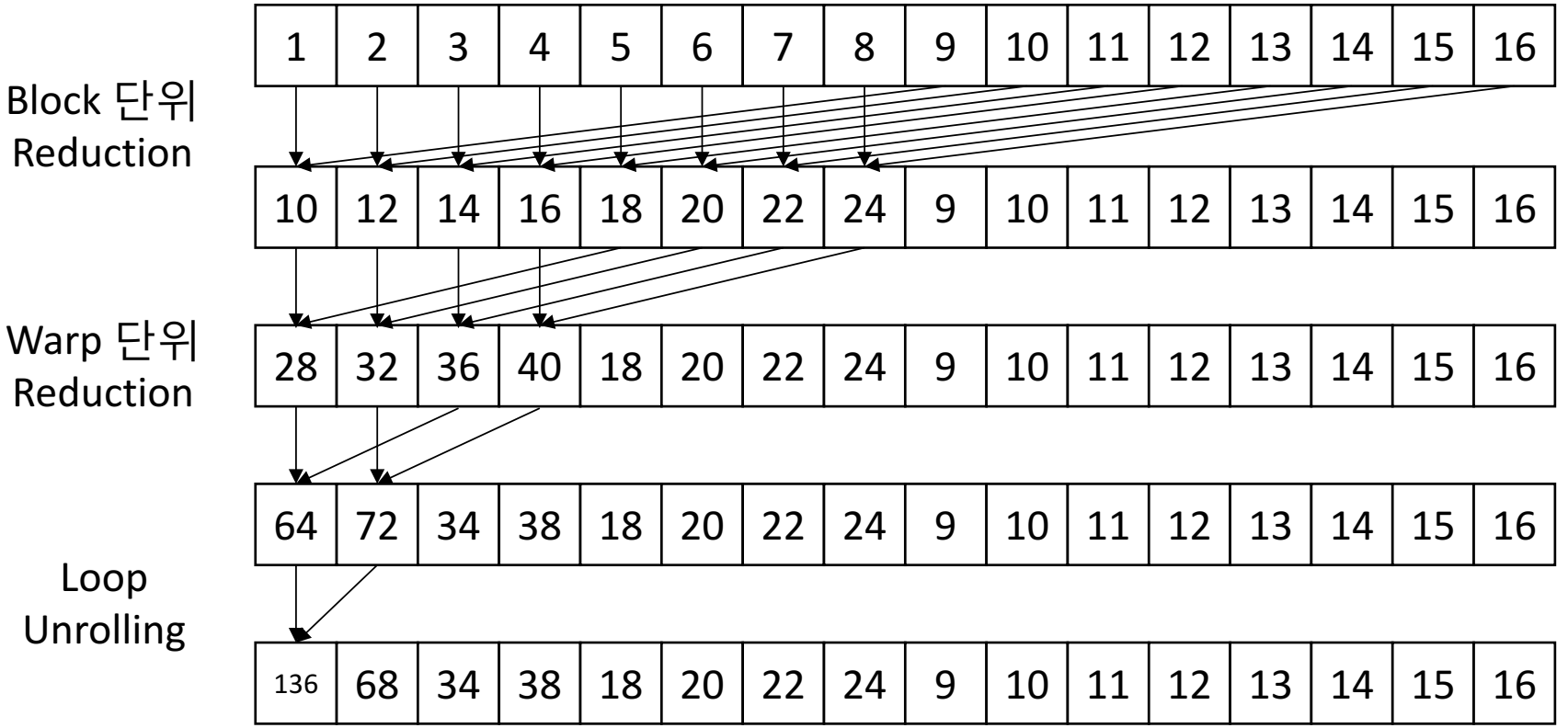
REDUCTION - SIMPLE IDEA

배열 내의 모든 값을 갖고 하나의 결과를 얻을 때,



REDUCTION - CUDA OPTIMIZED WAY

Block Size, Warp Size 및 Loop Unrolling 적용

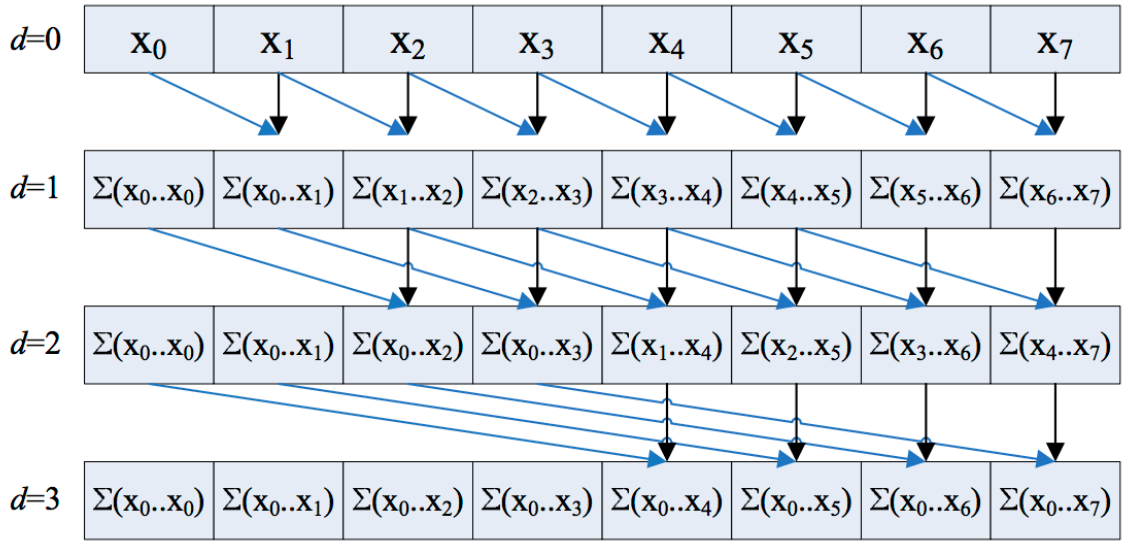


SCAN

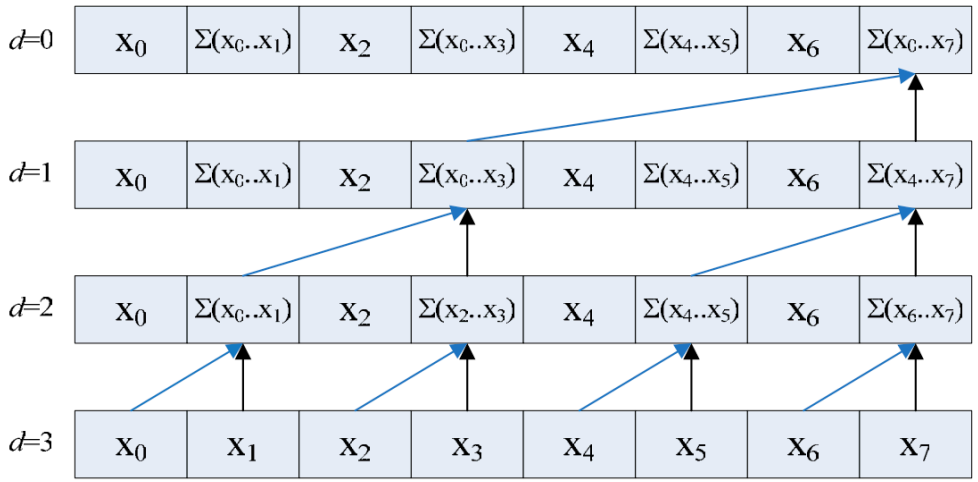
- 앞의 값을 참조해서 값을 구성하는 경우

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

- NAÏVE Implementation (Hillis & Steele)

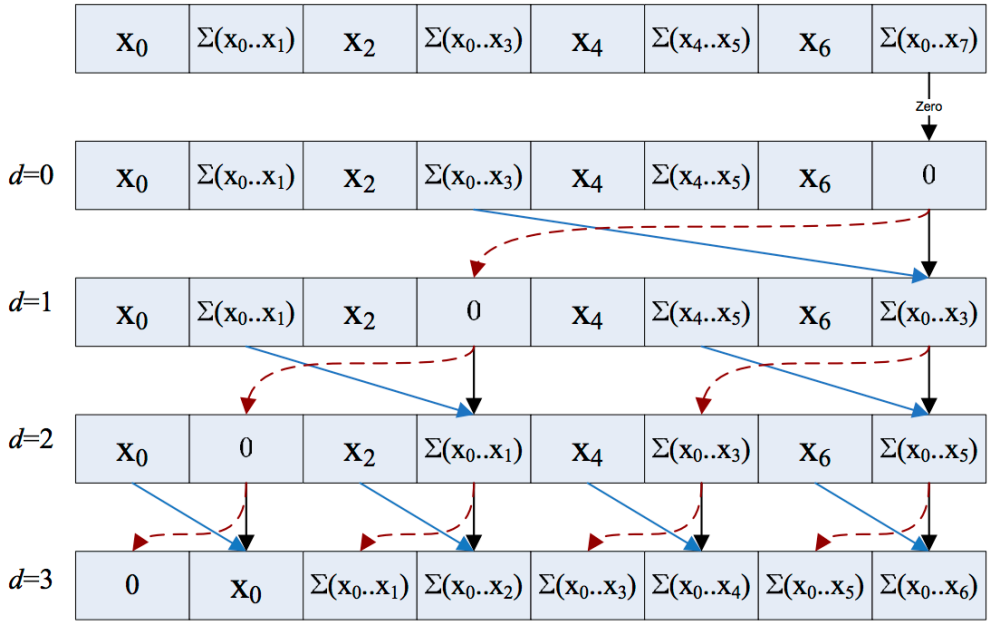


OPTIMIZED SCAN (BLELLOCH)




```
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]
```

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



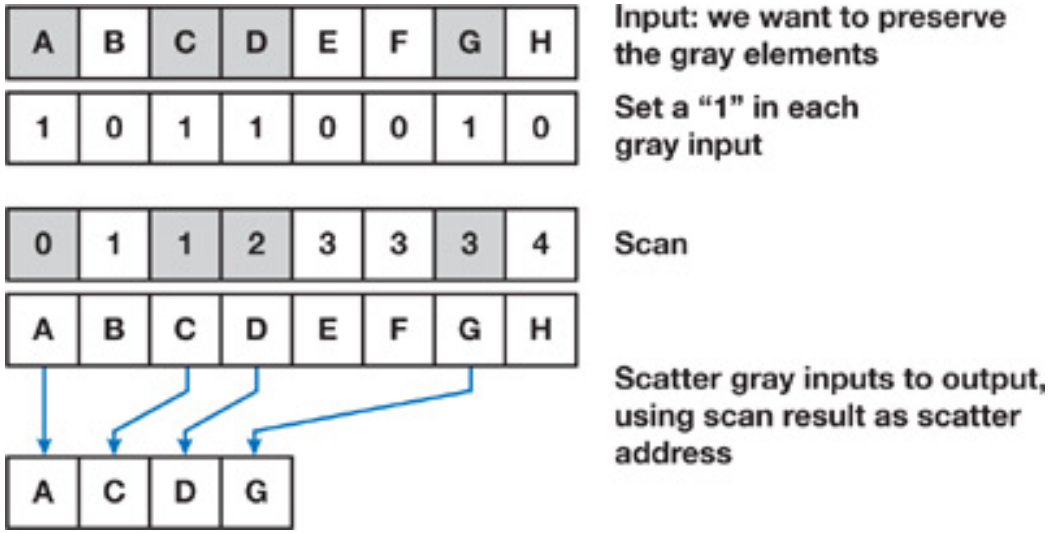

```
x[n - 1] := 0
for d := log2n down to 0 do
    for k from 0 to n – 1 by 2^(d+1) in parallel do
        t := x[k + 2^d - 1]
        x[k + 2^d - 1] := x [k + 2^(d+1) - 1]
        x[k + 2^(d+1) - 1] := t + x [k + 2^(d+1) - 1]
```

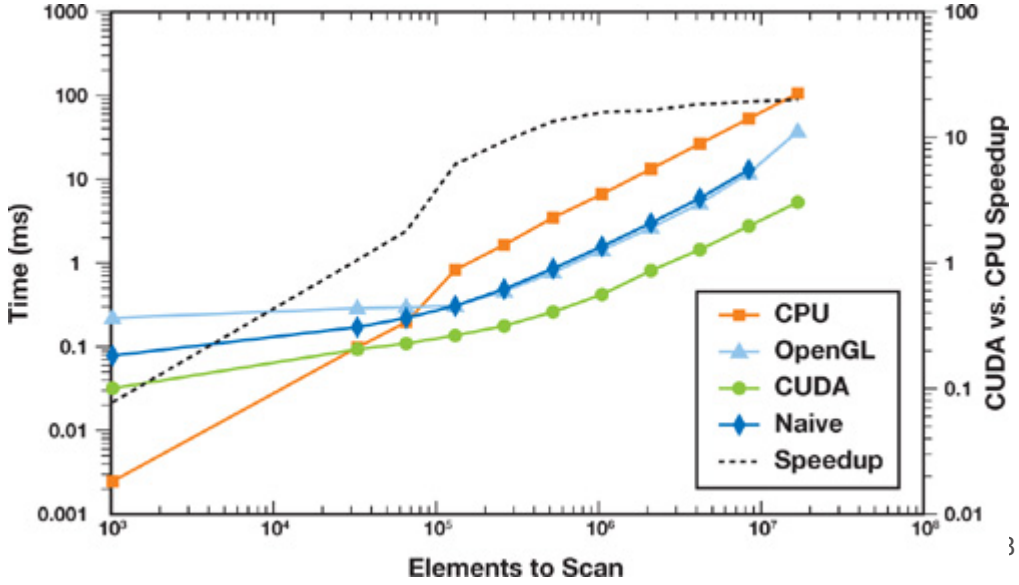

NVIDIA.

COMPACT

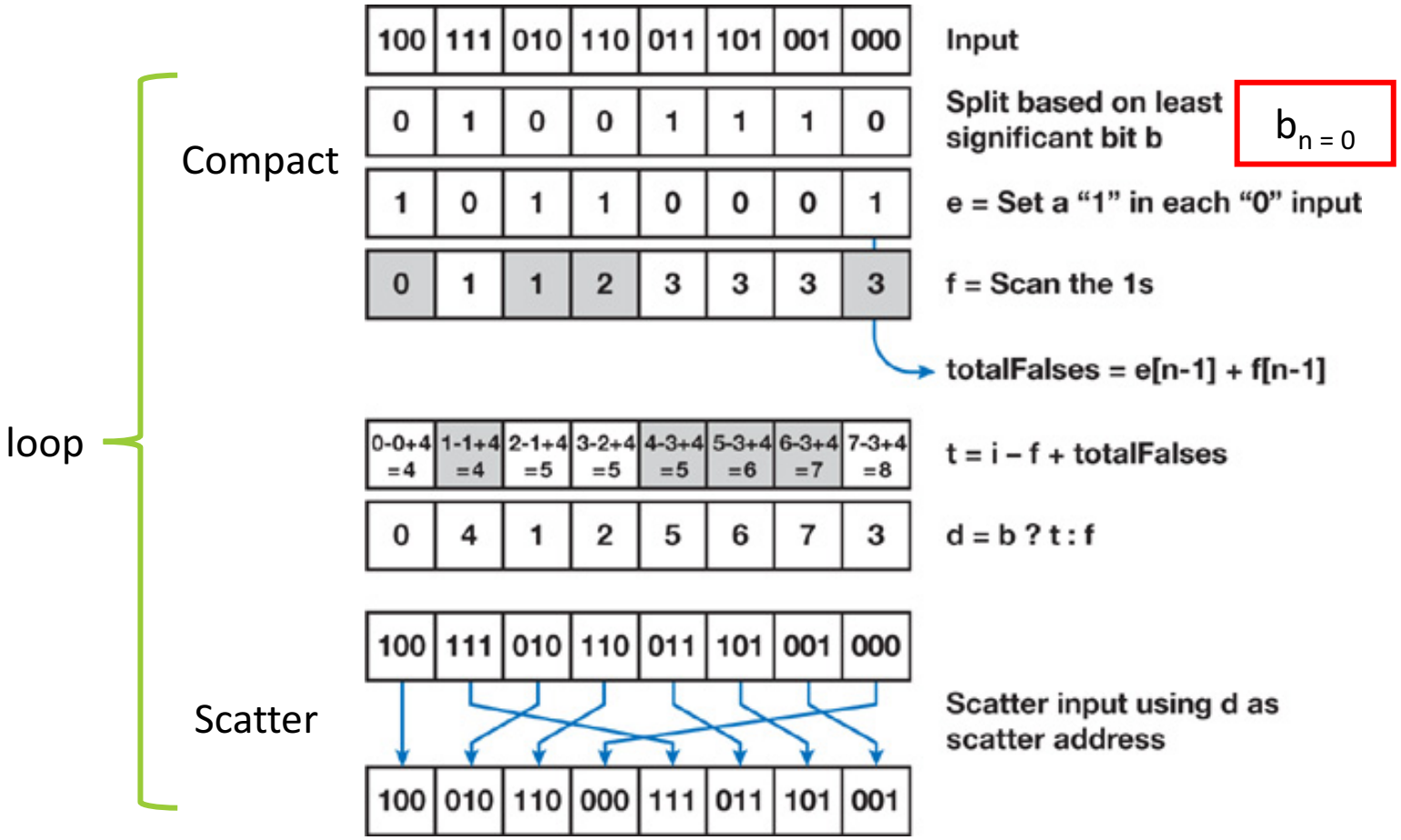
[A B C D E F G H] ➔ [A C D G]

Predicate + Scan + Scatter





RADIX SORT


loop
Compact
Scatter
100 111 010 110 011 101 001 000
Input
0 1 0 0 1 1 1 0
Split based on least significant bit b
$b_{n=0}$
1 0 1 1 0 0 0 1
e = Set a "1" in each "0" input
0 1 1 2 3 3 3 3
f = Scan the 1s
totalFalses = e[n-1] + f[n-1]
0-0+4 =4
1-1+4 =4
2-1+4 =5
3-2+4 =5
4-3+4 =5
5-3+4 =6
6-3+4 =7
7-3+4 =8
t = i – f + totalFalses
0 4 1 2 5 6 7 3
d = b ? t : f
100 111 010 110 011 101 001 000
Scatter input using d as scatter address
100 010 110 000 111 011 101 001

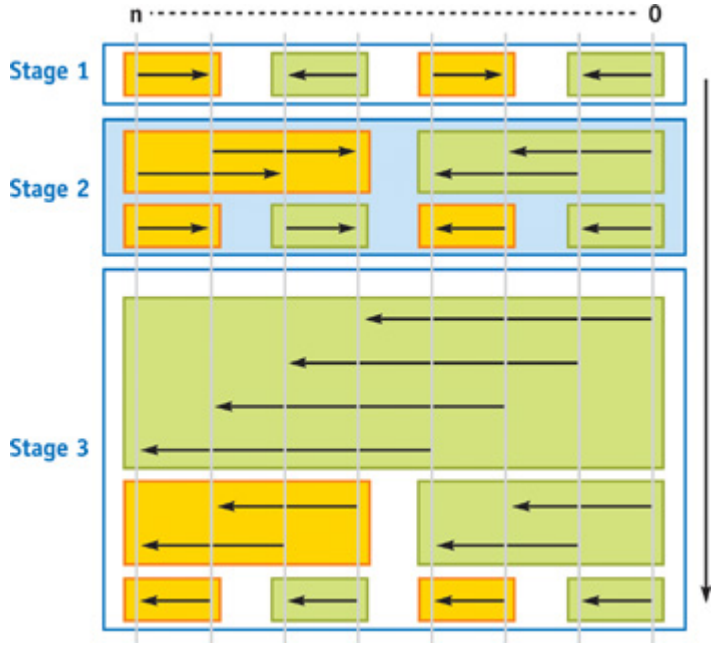
BITONIC MERGE SORT

Parallel processing을 효율적으로 활용하는 Merge Sort

$O(n) = O(n \log^2(n) + \log(n))$

Quick Sort: O(n) = O(n log(n))

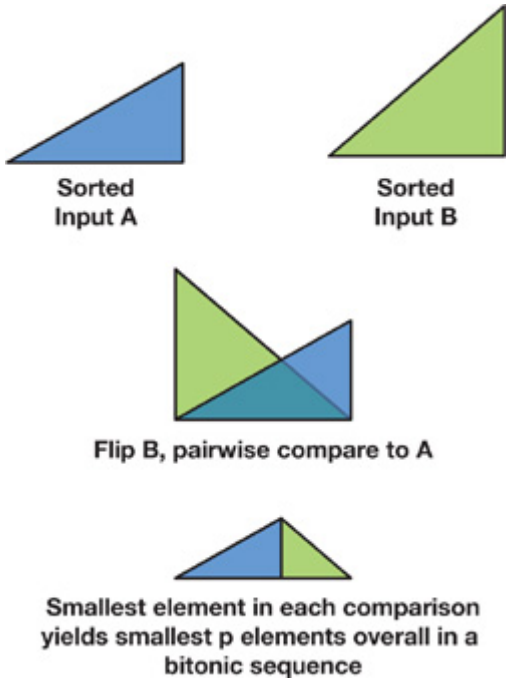
Quick sort 보다 cost가 크지만,
Stage 수만큼만의 시간만 필요하므로
더욱 효과적



NVIDIA.

MERGE SORT

Radix Sort로 Thread Block 단위로 Sort된 block을 병렬로 Merge



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

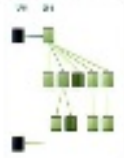
Open Compiler Tool Chain

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

Hardware Capabilities

SMX

Dynamic Parallelism



HyperQ

GPUDirect

© NVIDIA 2013

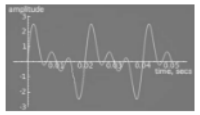
CUDA 가속 라이브러리

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.

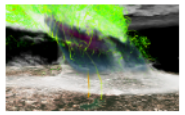
cuDNN

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.



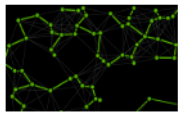
cuFFT

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

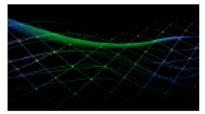
GPU-accelerated linear algebra library by EM Photonics, that utilizes CUDA to dramatically improve the computation speed of sophisticated mathematics.

MAGMA

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

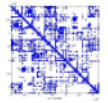
IMSL Fortran Numerical Library

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



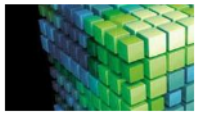
cuSOLVER

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.

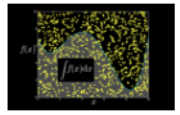


cuBLAS

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.

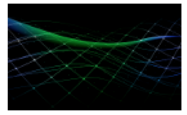
ArrayFire

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

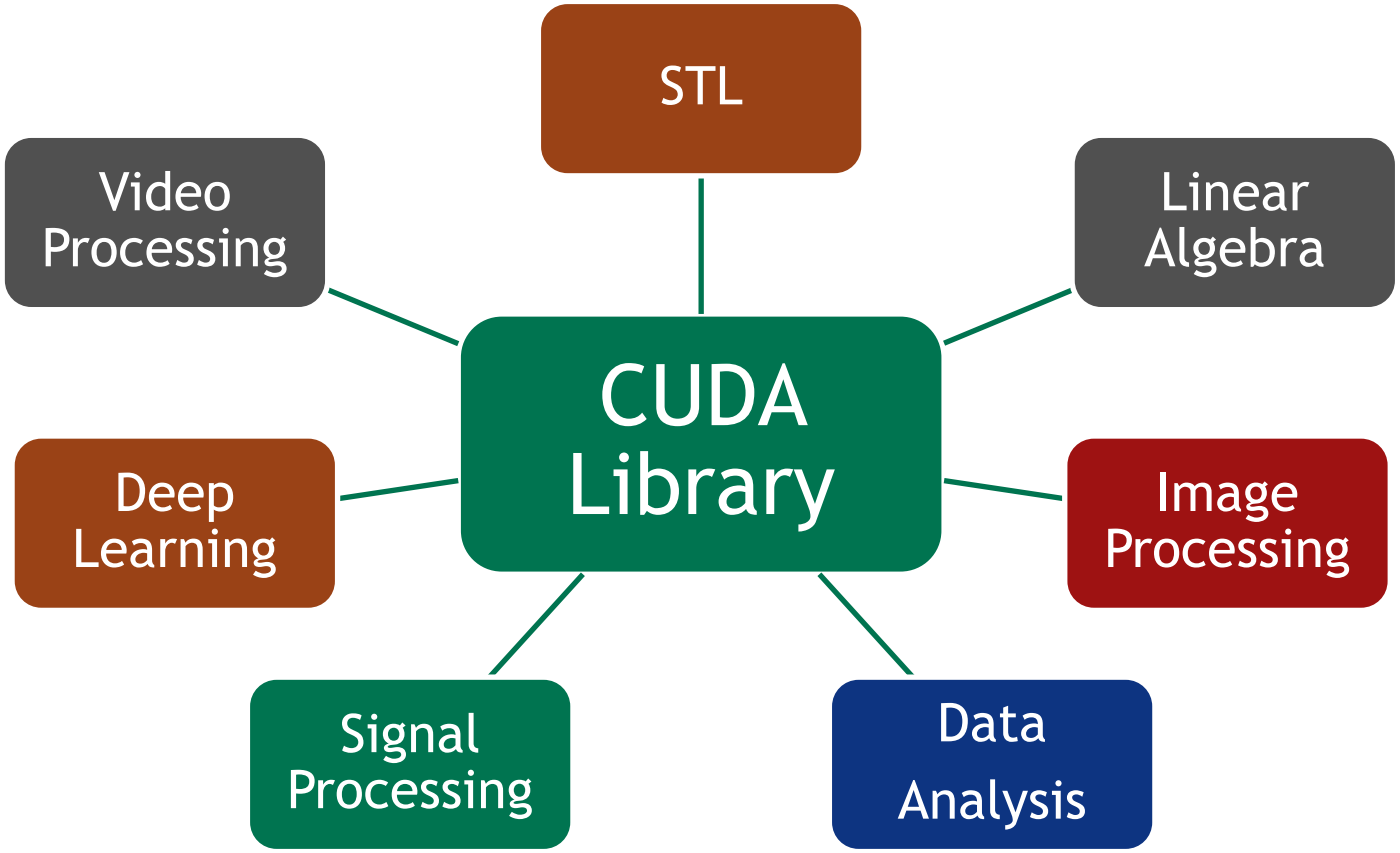
OpenCV is the leading open source library for computer vision, image processing and machine learning, and now features GPU acceleration for real-time operation.

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

CATEGORIES


STL
Video Processing
Linear Algebra
CUDA Library
Deep Learning
Image Processing
Signal Processing
Data Analysis

NVIDIA DEEP LEARNING SOFTWARE PLATFORM

TRAINING

Training Data

Data Management

Training

Model Assessment

Trained Neural Network

Caffe2 · Chainer · Microsoft Cognitive Toolkit · mxnet · TensorFlow · PYTORCH · theano

INFERENCE

Data center

GRE + TensorRT

Embedded

JETPACK SDK

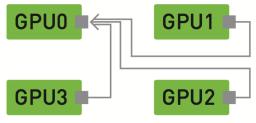
Automotive

DriveWorks SDK

NVIDIA DEEP LEARNING SDK and CUDA

cuDNN

NCCL



cuBLAS

cuSPARSE

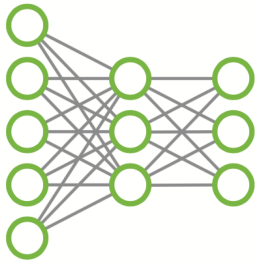
TensorRT

developer.nvidia.com/deep-learning-software

NVIDIA DEEP LEARNING SDK UPDATE

Up to 3x Faster Deep Learning on Volta

GPU-accelerated DL Primitives



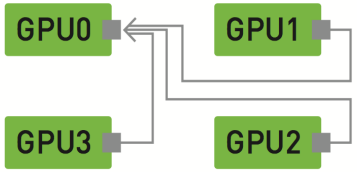
cuDNN 7

2.5x Faster Training of CNNs for computer vision

3x Faster Training of RNNs for speech and machine translations

Leading frameworks support

Multi-GPU & Multi-node



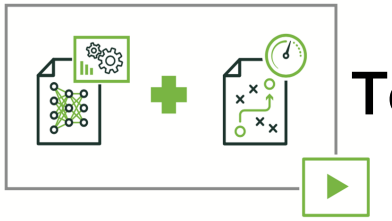
NCCL 2

Multi-node distributed training on up-to 8 servers

Near-linear scaling on PCIe and Nvink interconnect

Leading frameworks support

Inference Optimizer and Runtime Engine



TensorRT 3

3.5x Faster Inference

Optimize TensorFlow or Caffe trained models

Linux, Windows, QNX and Android support

NVIDIA cuDNN 7

Deep Learning Primitives

High performance building blocks for deep learning frameworks

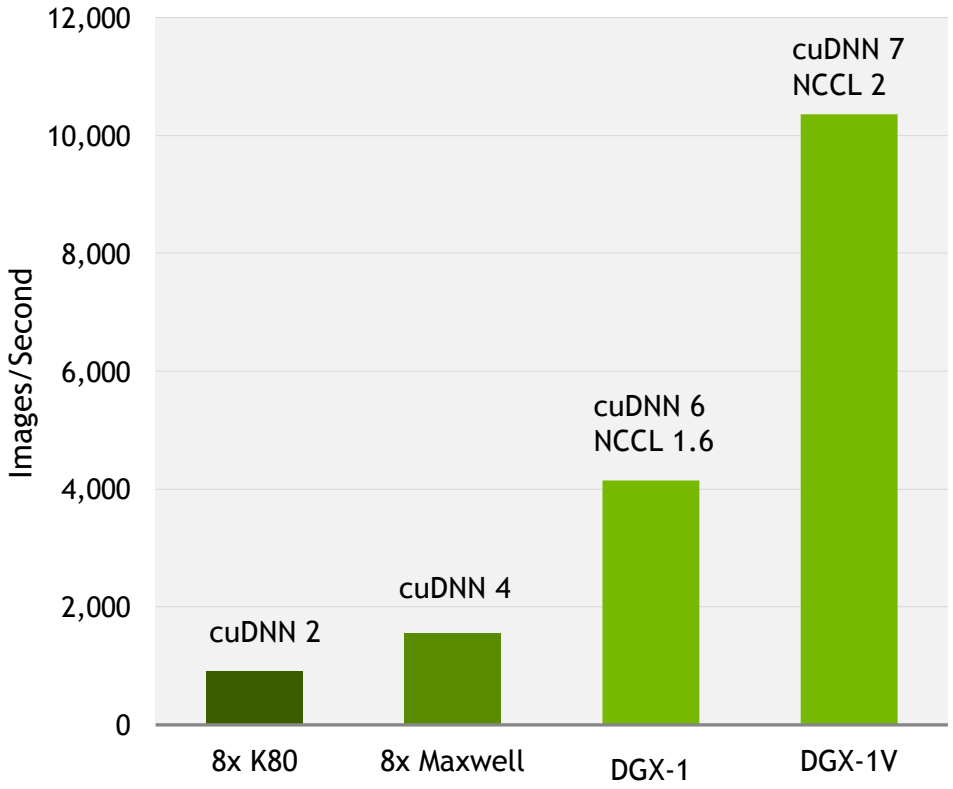
Drop-in acceleration for widely used deep learning frameworks such as Caffe2, Microsoft Cognitive Toolkit, PyTorch, Tensorflow, Theano and others

Accelerates industry vetted deep learning algorithms, such as convolutions, LSTM RNNs, fully connected, and pooling layers

Fast deep learning training performance tuned for NVIDIA GPUs

developer.nvidia.com/cudnn

Deep Learning Training Performance

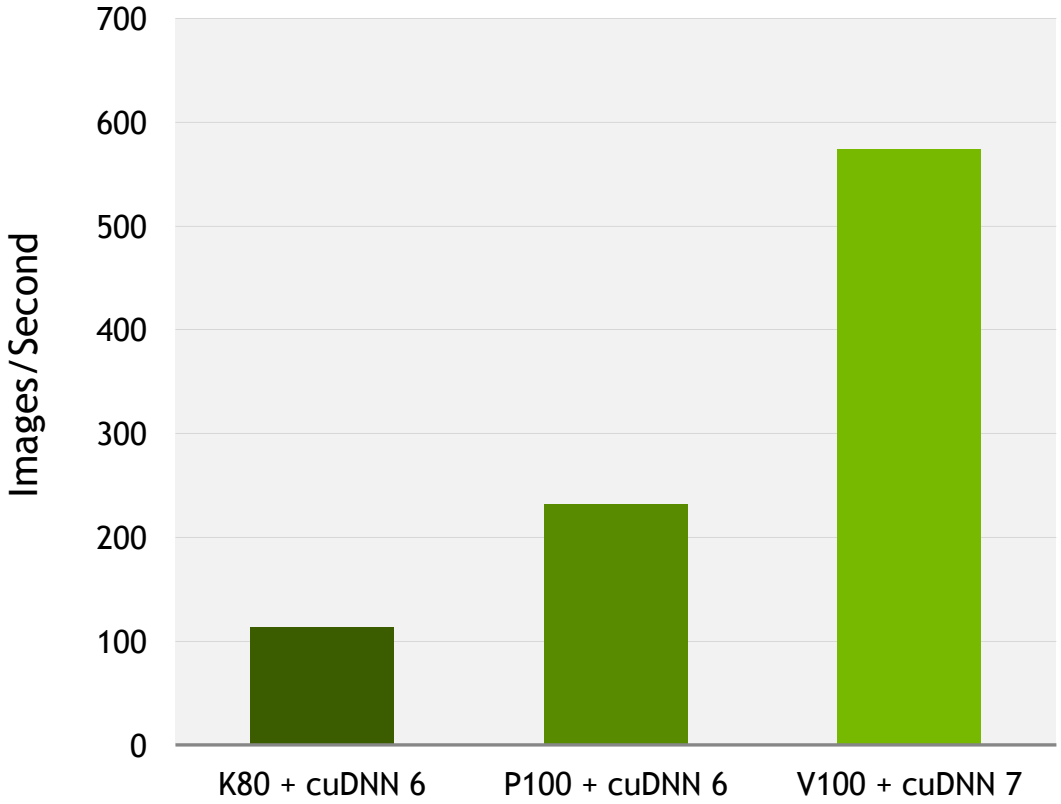


"NVIDIA has improved the speed of cuDNN with each release while extending the interface to more operations and devices at the same time."

— Evan Shelhamer, Lead Caffe Developer, UC Berkeley

cuDNN 7: 2.5X FASTER CNNs 3X FASTER RNNs

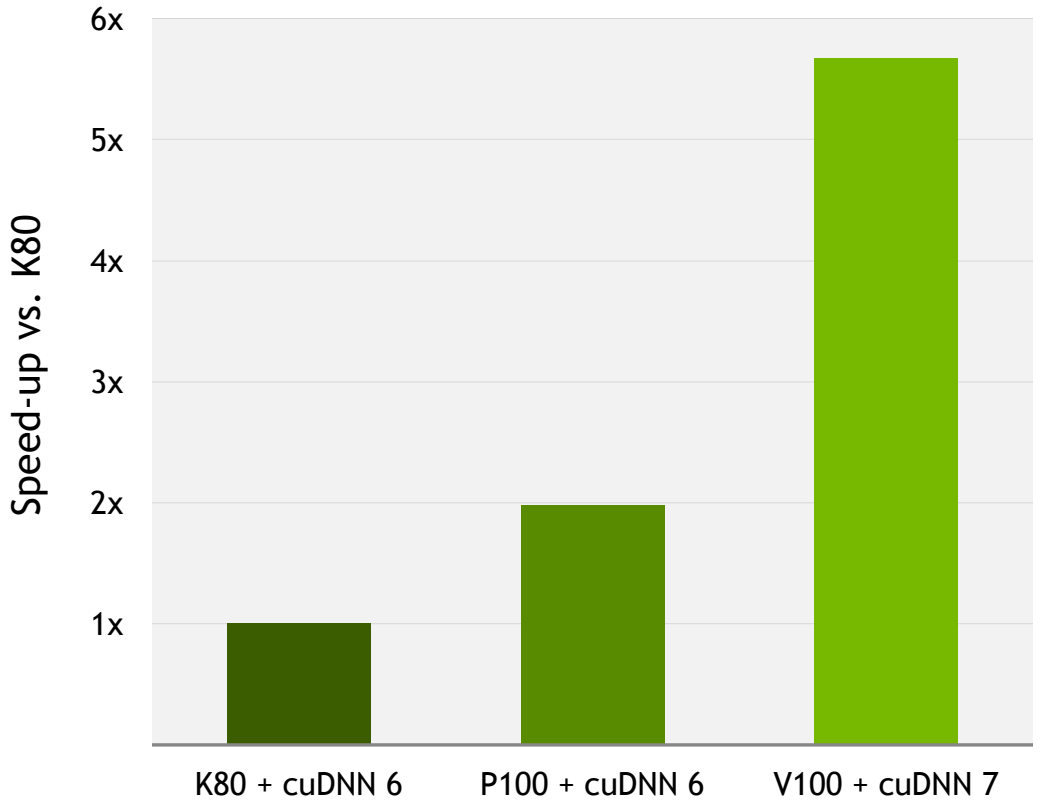
2.5x Faster Training of CNNs



Caffe2 performance (images/sec), Tesla K80 + cuDNN 6 (FP32), Tesla P100 + cuDNN 6 (FP32), Tesla V100 + cuDNN 7 (FP16). ResNet50, Batch size: 64

developer.nvidia.com/cudnn

3x Faster Training of LSTM RNNs



MXNet performance (min/epoch), Tesla K80 + cuDNN 6 (FP32), Tesla P100 + cuDNN 6 (FP32), Tesla V100 + cuDNN 7 (FP16). OpenNMT 662M seq2seq RNN (http://opennmt.net/Models/)

NVIDIA Collective Communications Library (NCCL) 2

Multi-GPU and multi-node collective communication primitives

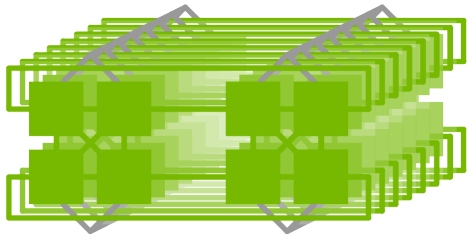
High-performance multi-GPU and multi-node collective communication primitives optimized for NVIDIA GPUs

Fast routines for multi-GPU multi-node acceleration that maximizes inter-GPU bandwidth utilization

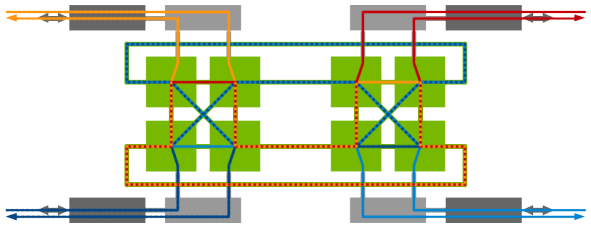
Easy to integrate and MPI compatible. Uses automatic topology detection to scale HPC and deep learning applications over PCIe and NVink

Accelerates leading deep learning frameworks such as Caffe2, Microsoft Cognitive Toolkit, MXNet, PyTorch and more

Multi-GPU:
NVLink
PCIe



Multi-Node:
InfiniBand verbs
IP Sockets

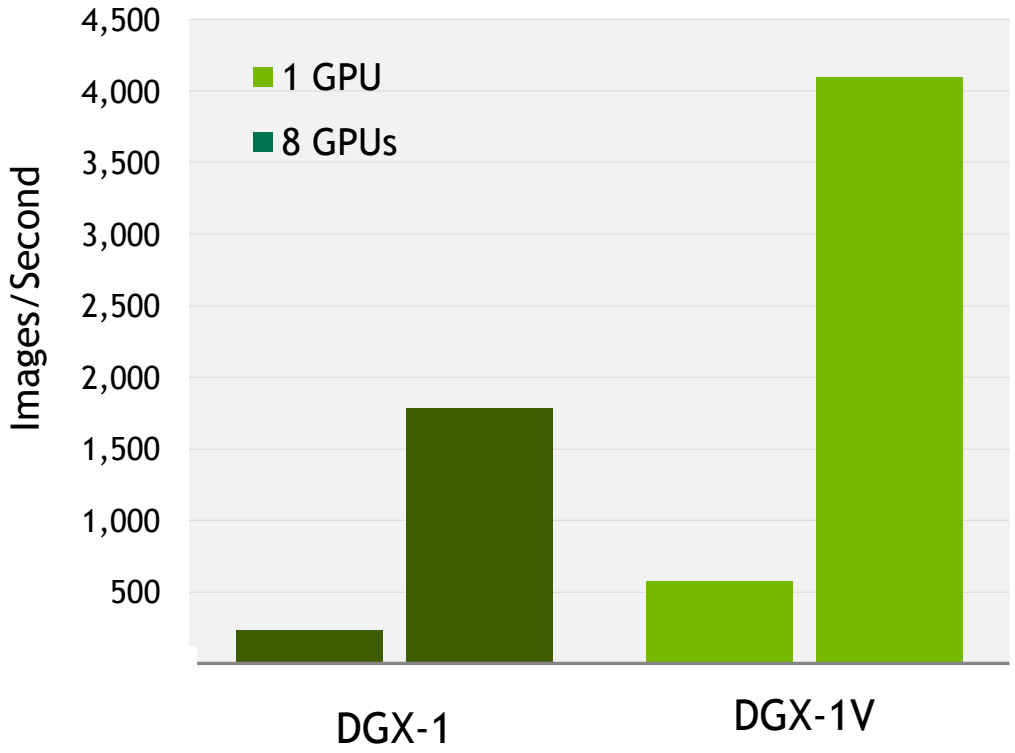


Automatic
Topology
Detection

developer.nvidia.com/nccl

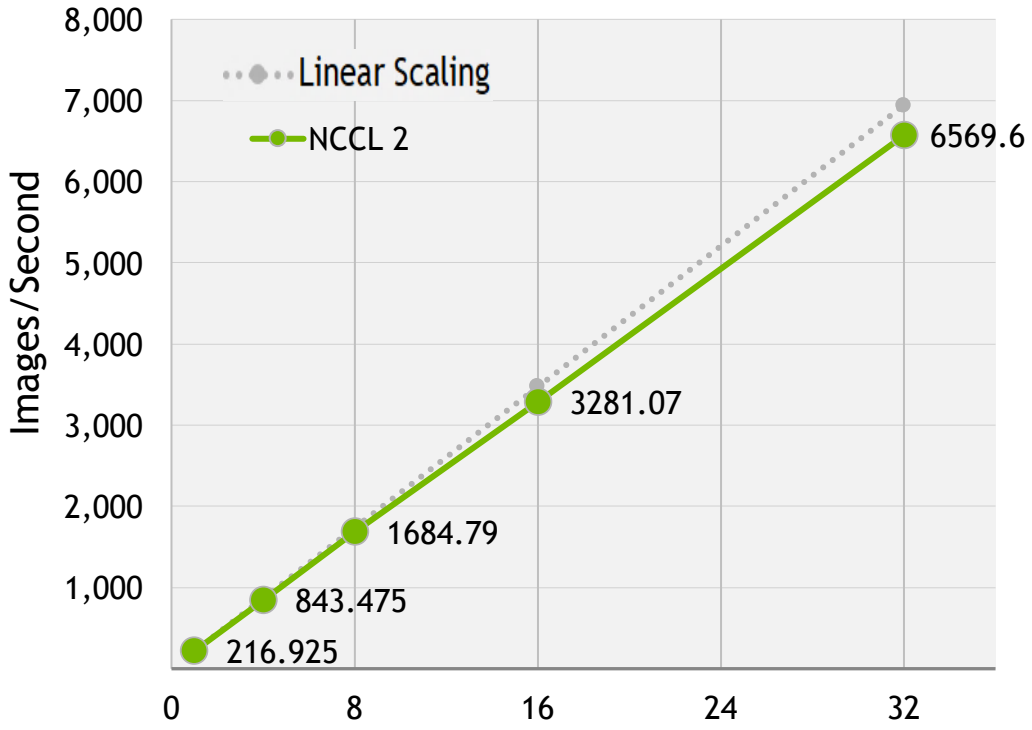
NCCL: MULTI-GPU AND MULTI-NODE SCALING

7x Faster Training on DGX vs. single GPU



Caffe2 multi-GPU performance (images/sec) DGX-1 + cuDNN 6 (FP32), DGX-1V + cuDNN 7 (FP16). ResNet50, Batch size: 64

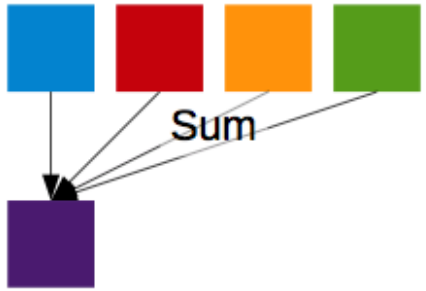
Near-Linear Multi-Node Scaling



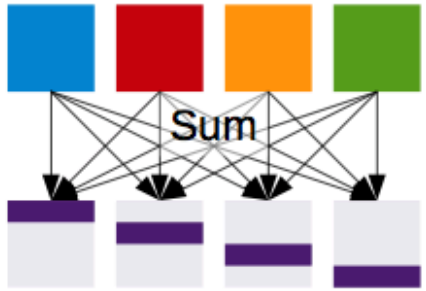
Microsoft Cognitive Toolkit multi-node scaling performance (images/sec), NVIDIA DGX-1 + cuDNN 6 (FP32), ResNet50, Batch size: 64

developer.nvidia.com/nccl

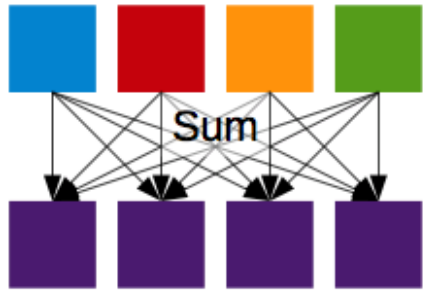
ncclReduce



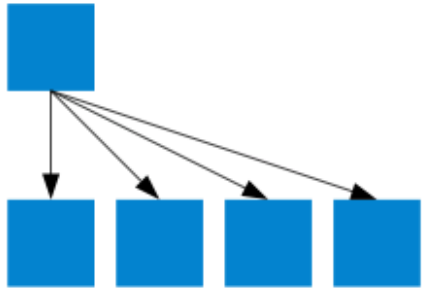
ncclReduceScatter



ncclAllreduce



ncclBcast



ncclAllgather

NVIDIA.

NVIDIA TensorRT 3

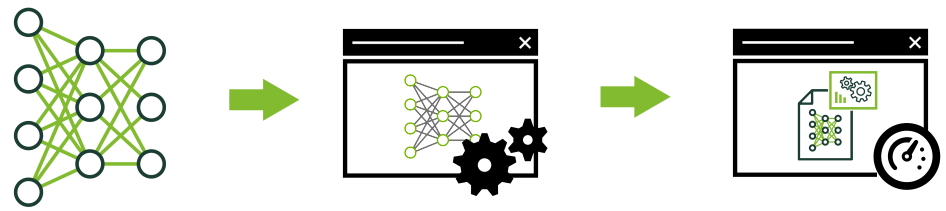
Deep Learning Inference Optimizer and Runtime

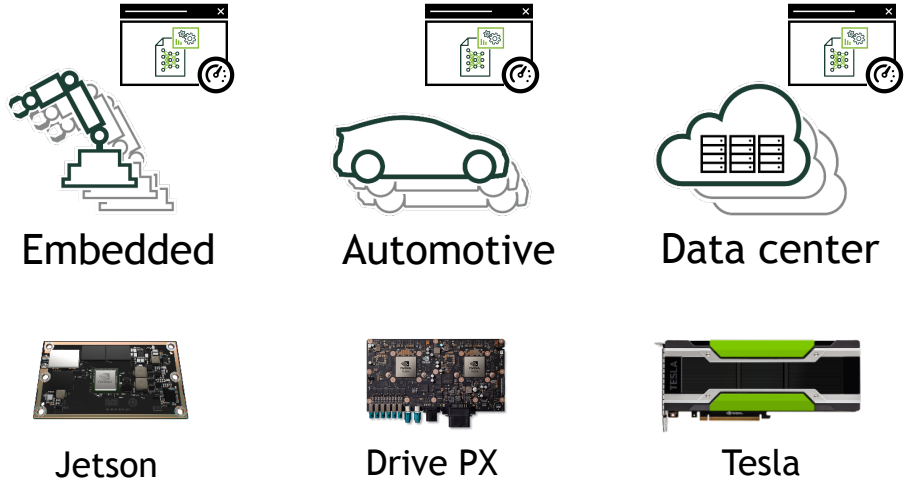
High performance neural network inference optimizer and runtime engine for production deployment

Maximize inference throughput for latency-critical services in hyperscale datacenters, embedded, and automotive production environments.

Optimize models trained in TensorFlow or Caffe to generate runtime engines that maximizes inference throughput

Deploy faster, more responsive and memory efficient deep learning applications with INT8 and FP16 optimized precision support

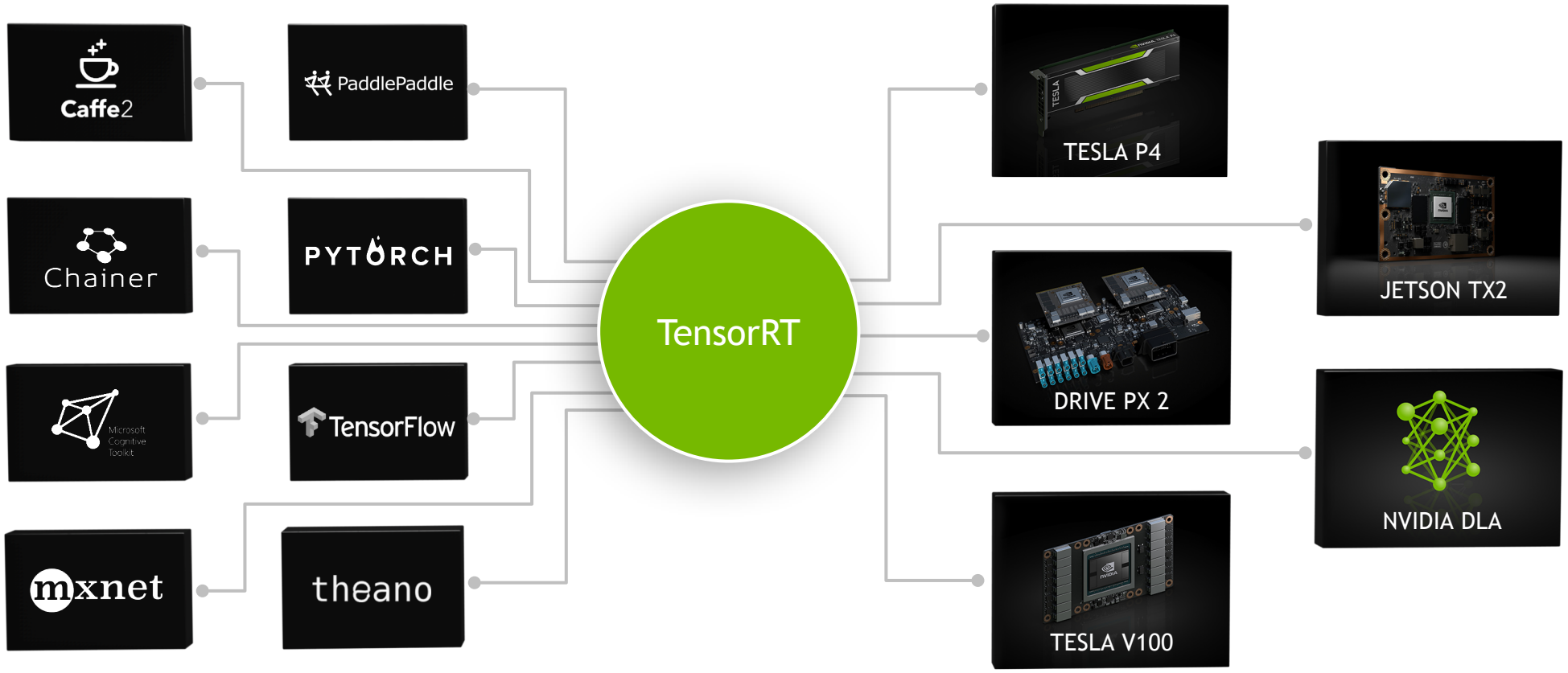




developer.nvidia.com/tensorrt

NEW NVIDIA TENSORRT 3

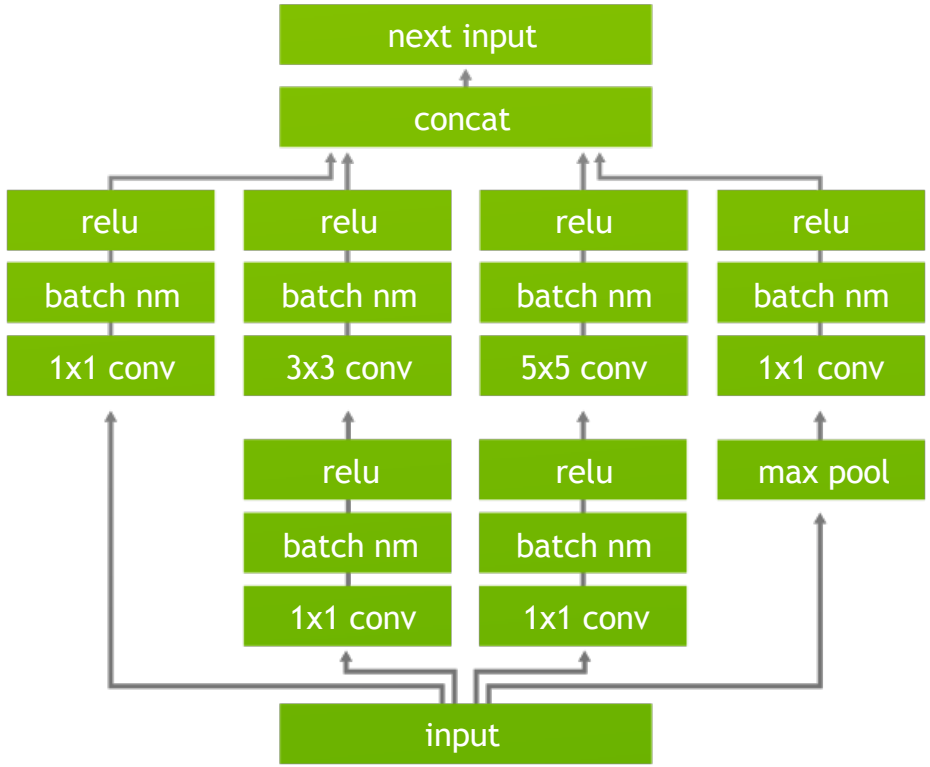
Programmable Inference Accelerator

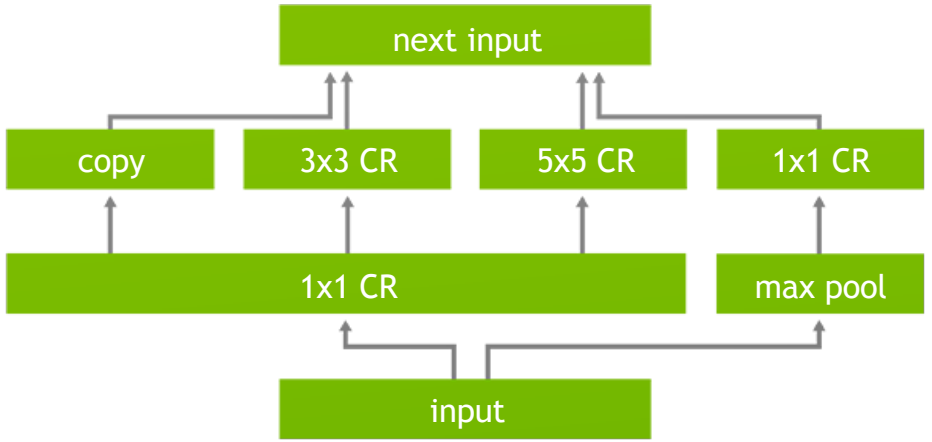


Compile and Optimize Neural Networks | Support for Every Framework
Optimize for Each Target Platform

NEW NVIDIA TENSORRT 3

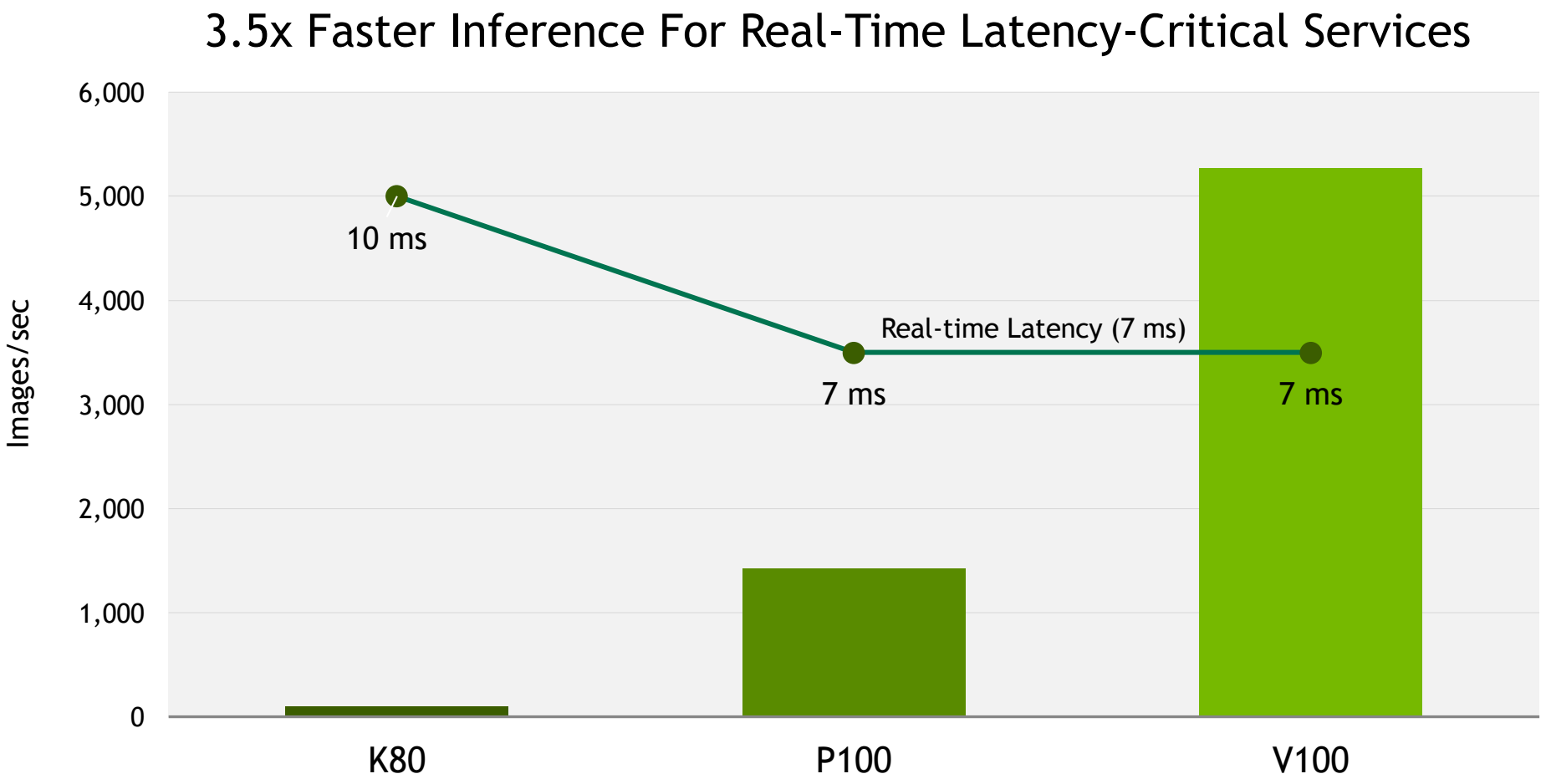
Programmable Inference Accelerator





Weight & Activation Precision Calibration | Layer & Tensor Fusion
Kernel Auto-Tuning | Multi-Stream Execution

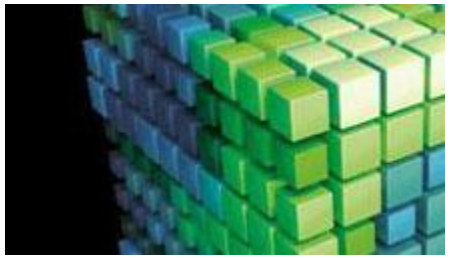
TensorRT 3: 3.5X FASTER INFERENCE



ResNet50 Inference, TensorRT performance (images/sec), TensorRT + K80: Batch Size =1, Latency = 10 ms
TensorRT + P100 (FP16): Batch Size =9 Latency= 7ms, TensorRT + V100 (FP16): Batch Size =26 Latency= 7ms,

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

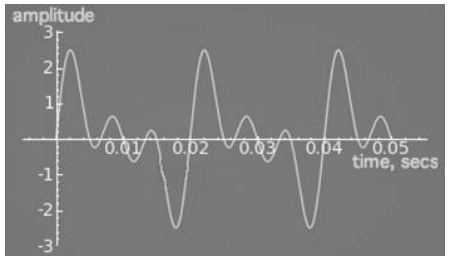
$$C = \alpha\, op(A) op(B) + \beta op(C)$$

```
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.



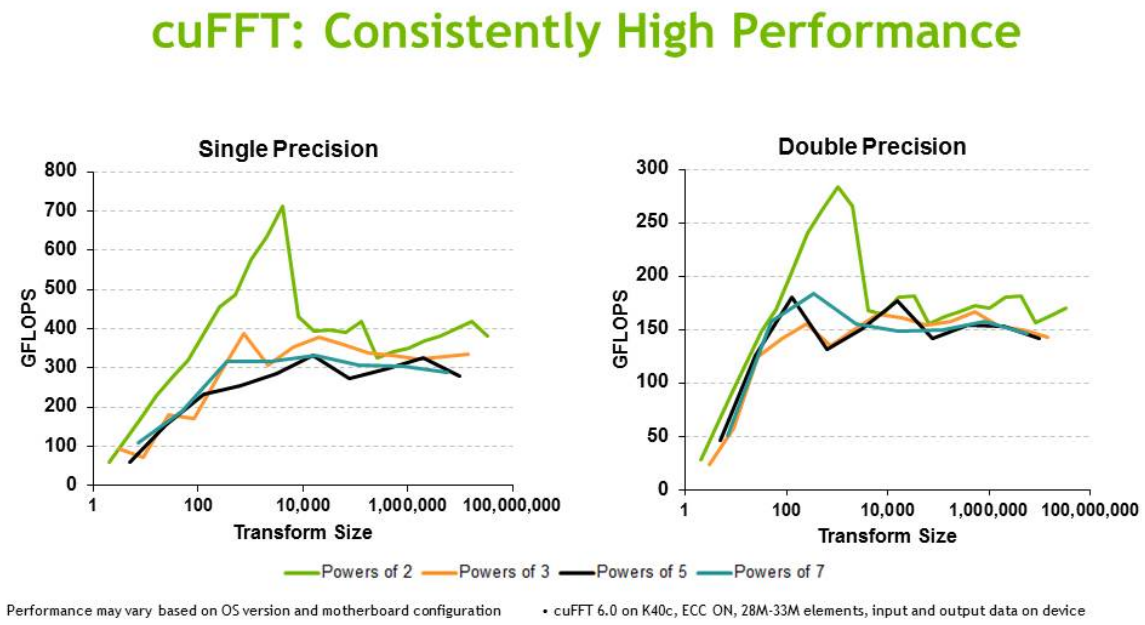
CUFFT

CUDA Library for FFTs on the GPU

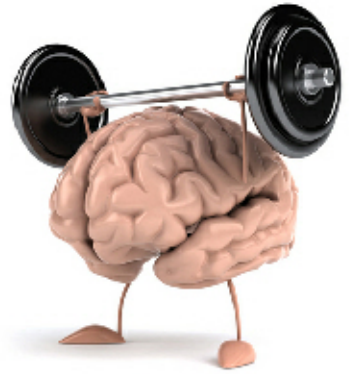
Features:

- 1D ~ 3D transforms for real & complex
- Batch execution for multiple transform
- Stream execution

http://www.fftw.org/와 유사



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

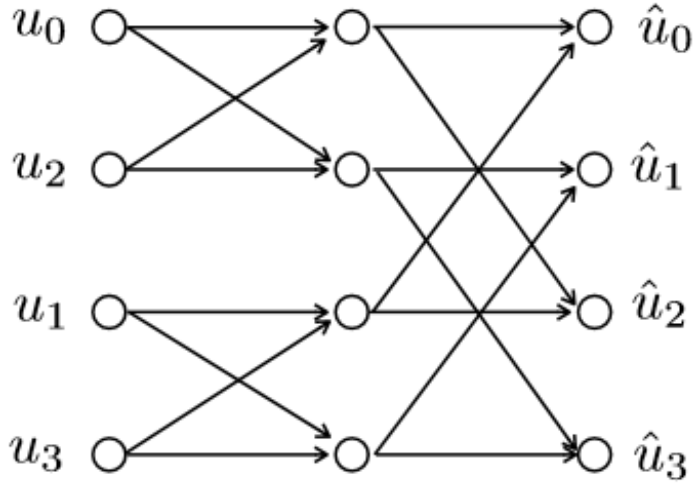
$$u(x) = \int_{-\infty}^{\infty} e^{ikx} \hat{u}(k) dk$$

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



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

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

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 FEATURES

Unified Virtual Address / Unified Memory

Multi-GPU

GPU Direct

MPI + CUDA

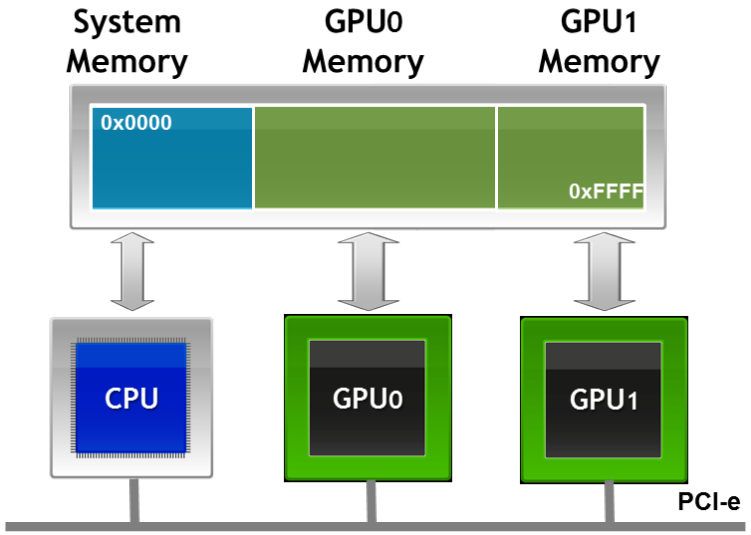
Dynamic Parallelism

UNIFIED VIRTUAL MEMORY CODE

64bit OS/Application Only

Address만으로 Host/Device 구분

UVA: Single Address Space




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

cudaMemcpy(d_mem, h_mem, bufsize);

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

cudaFree(d_mem);
```

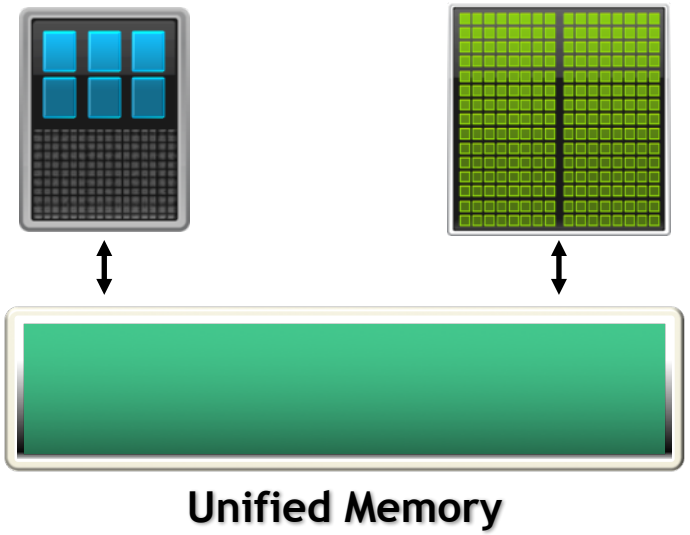
```
void *d_mem;
cudaMallocManaged(&d_mem, bufsize);

fread(d_mem, 1, bufsize, fp);

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

cudaFree(d_mem);
```

NVIDIA.

UNIFIED MEMORY

CUDA 8.0, Pascal 이후 지원

Developer View With Unified Memory



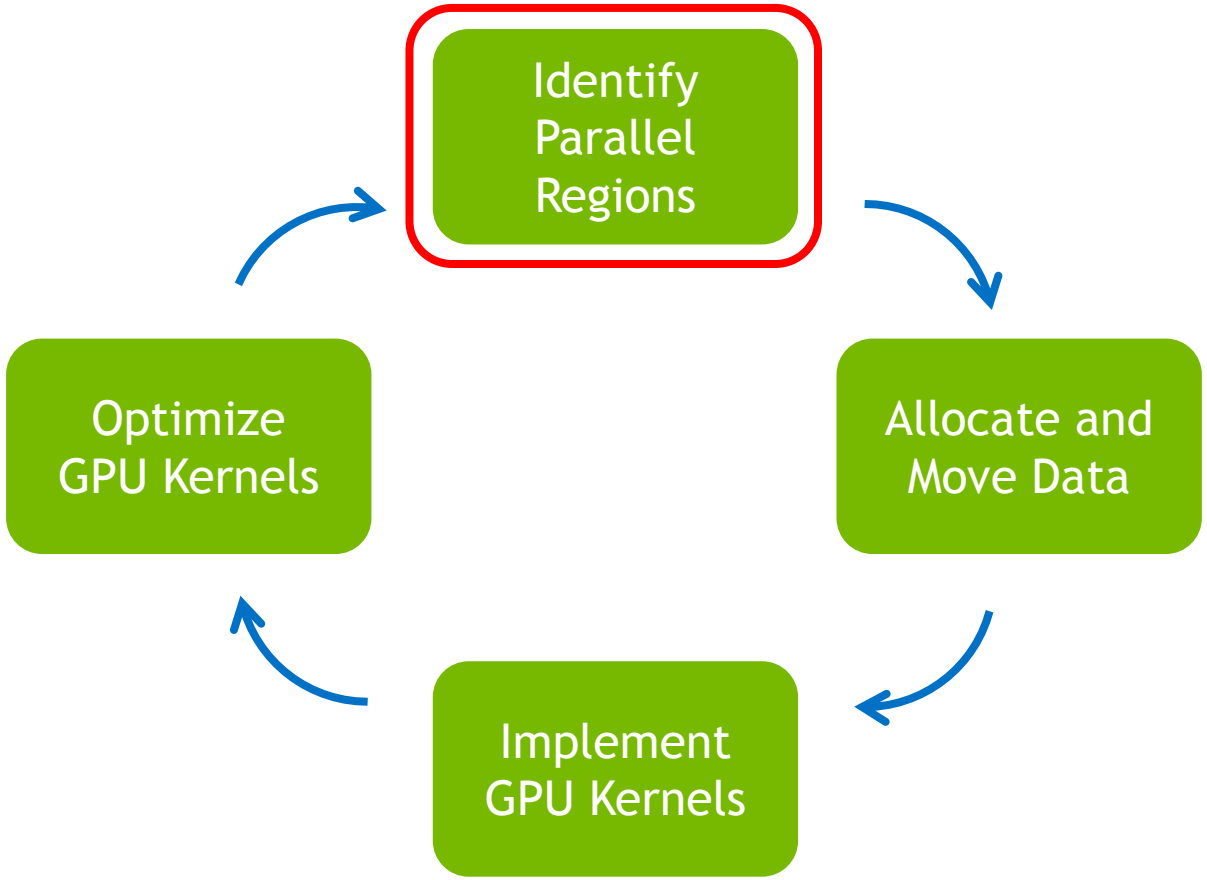
시스템 주소공간 공유

```
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);
```

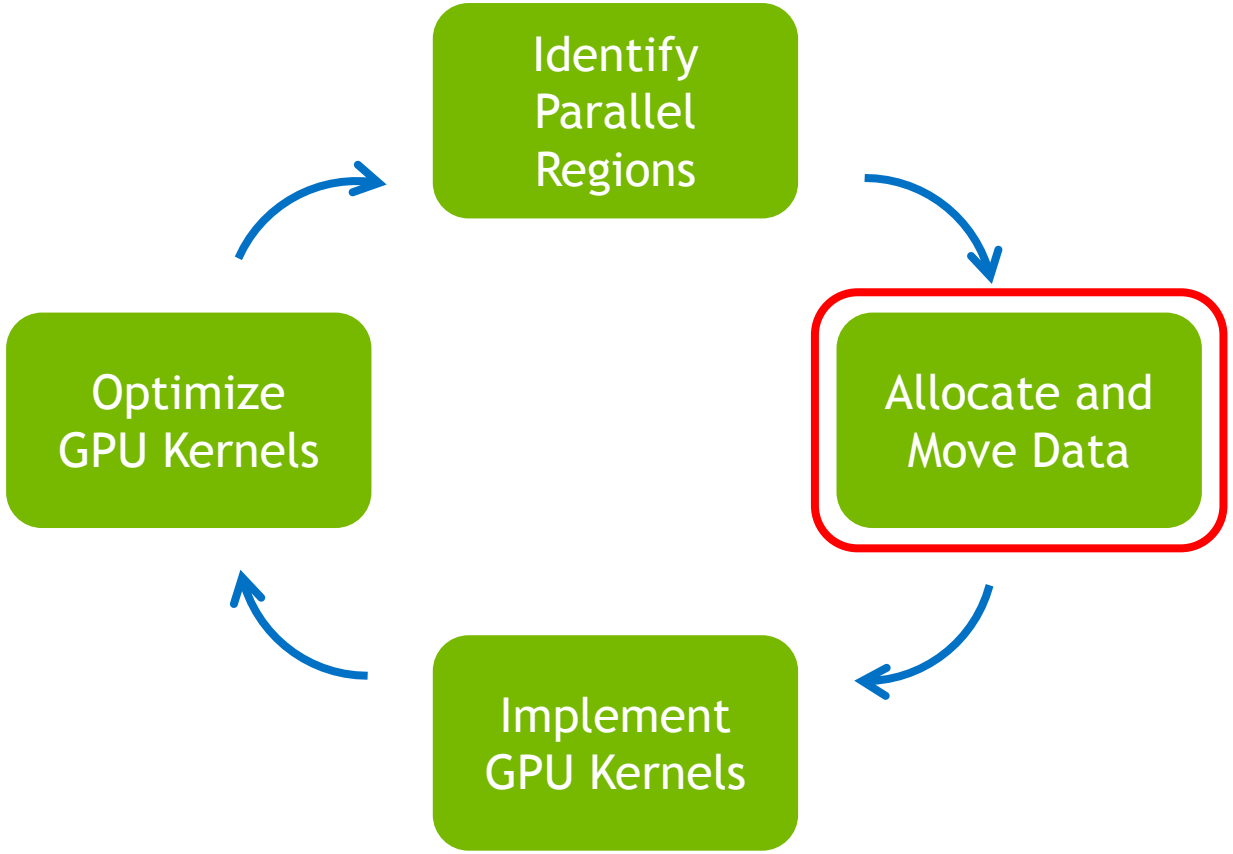
APPLICATION의 CUDA 가속 절차

1. 성능 분석
2. 병목 탐색
3. 알고리즘 분석 및 병렬화 전략 수립



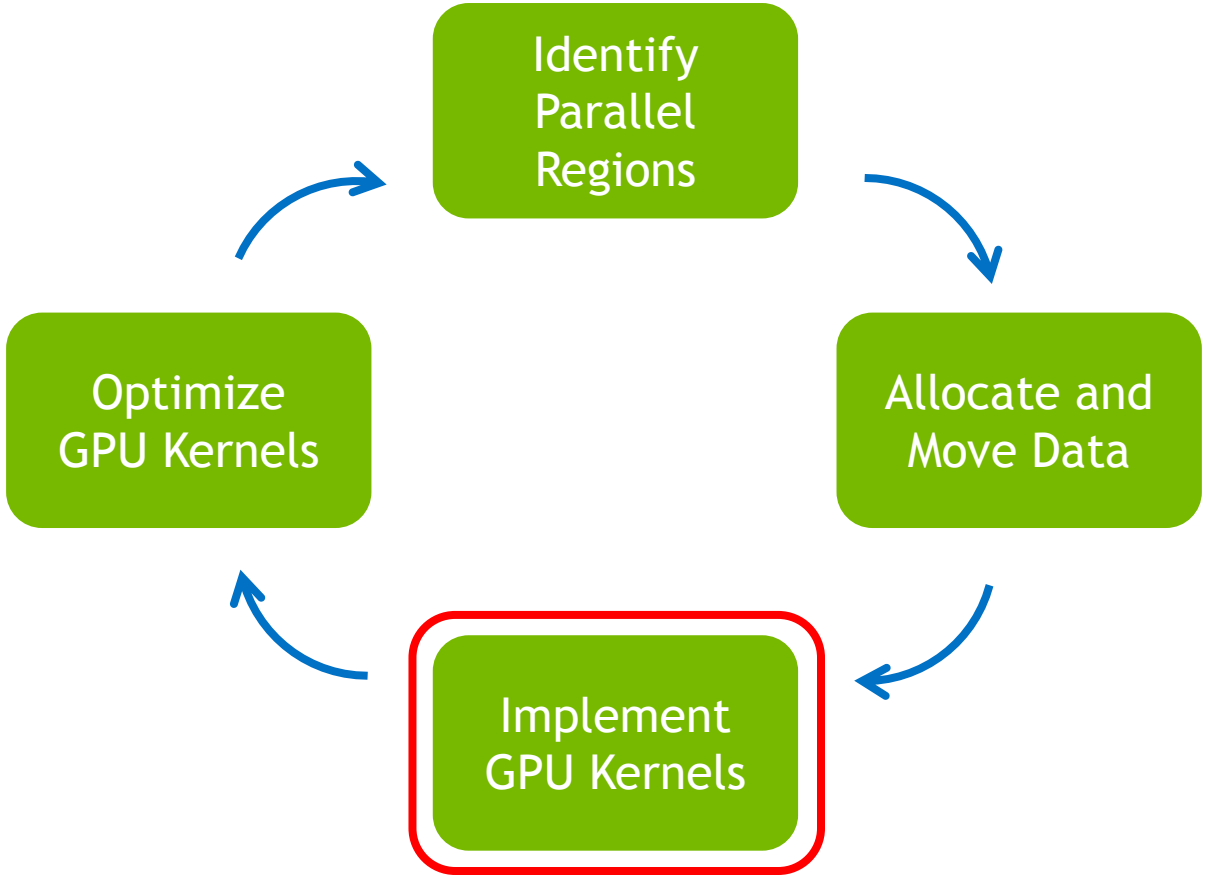
APPLICATION의 CUDA 가속 절차

1. 성능 분석
2. 병목 탐색
3. 알고리즘 분석 및 병렬화 전략 수립
4. 데이터 전송 및 최적화



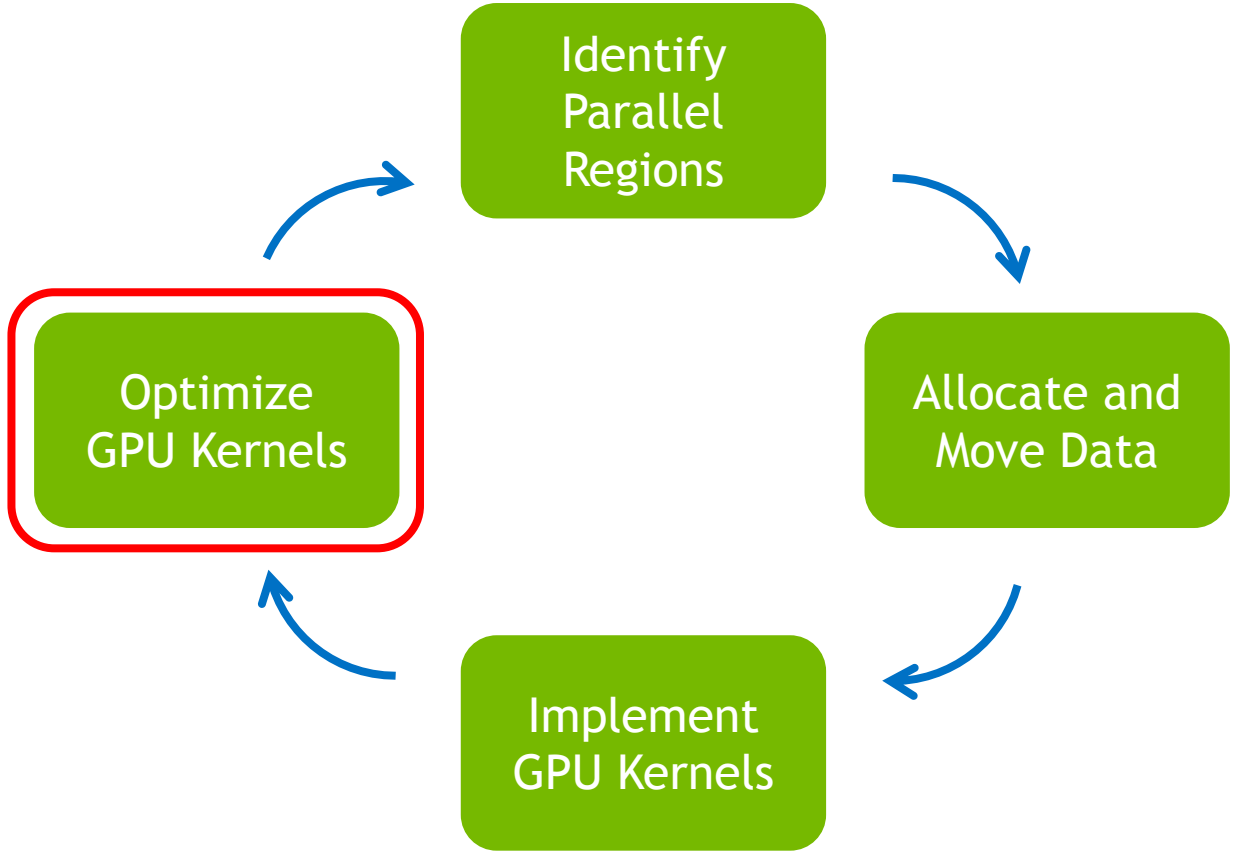
APPLICATION의 CUDA 가속 절차

1. 성능 분석
2. 병목 탐색
3. 알고리즘 분석 및 병렬화 전략 수립
4. 데이터 전송 및 최적화
5. GPU 병렬 코드 작성



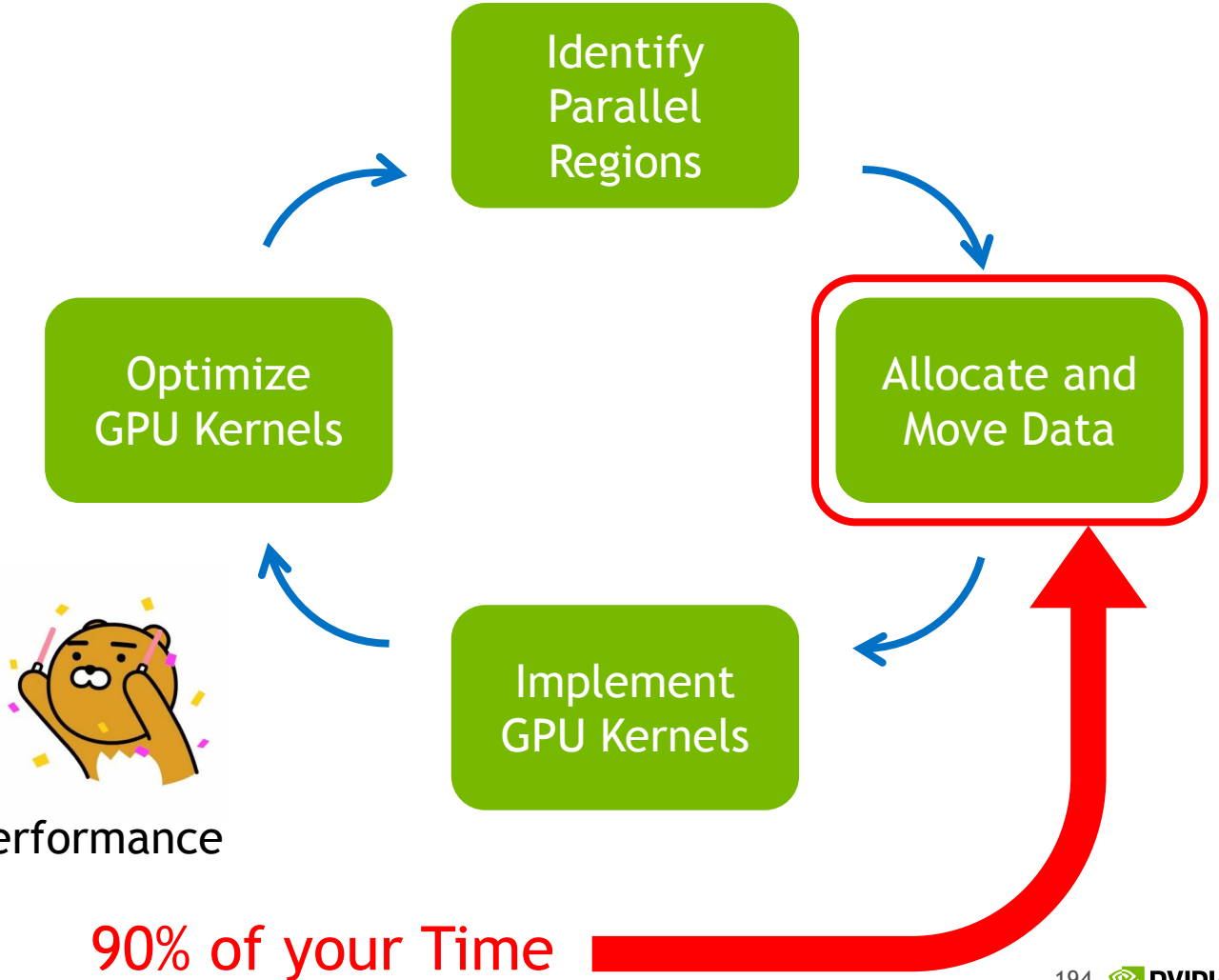
APPLICATION의 CUDA 가속 절차

1. 성능 분석
2. 병목 탐색
3. 알고리즘 분석 및 병렬화 전략 수립
4. 데이터 전송 및 최적화
5. GPU 병렬 코드 작성
6. GPU 코드 최적화



APPLICATION의 CUDA 가속 절차

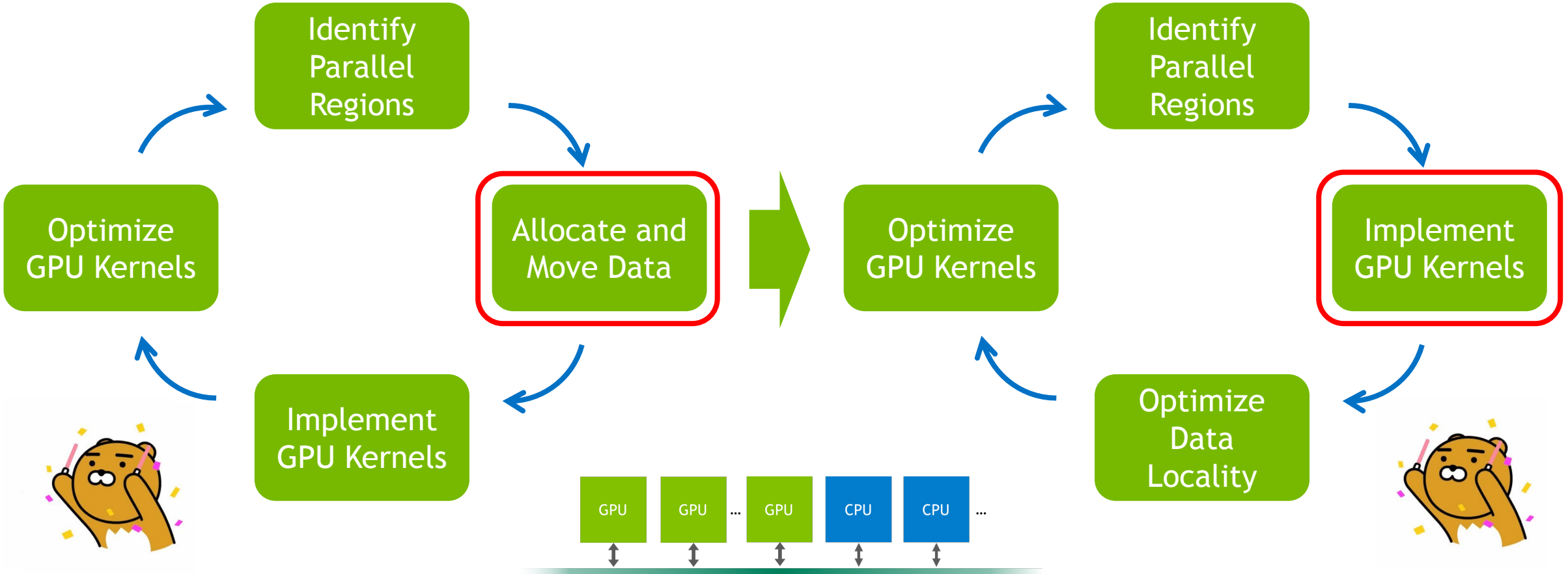
1. 성능 분석
2. 병목 탐색
3. 알고리즘 분석 및 병렬화 전략 수립
4. 데이터 전송 및 최적화
5. GPU 병렬 코드 작성
6. GPU 코드 최적화



Good Initial Performance

UNIFIED MEMORY

Easier to achieve initial good performance



UNIFIED MEMORY

Single pointer for CPU and GPU

CPU code

```
void sortfile(FILE *fp, int N) {
  char *data;
  data = (char *)malloc(N);

  fread(data, 1, N, fp);

  qsort(data, N, 1, compare);


  use_data(data);

  free(data);
}
```

GPU code with Unified Memory

```
void sortfile(FILE *fp, int N) {
  char *data;
  cudaMallocManaged(&data, N);

  fread(data, 1, N, fp);

  qsort<<<...>>>(data,N,1,compare);
  cudaDeviceSynchronize();

  use_data(data);

  cudaFree(data);
}
```

UNIFIED MEMORY ON PASCAL

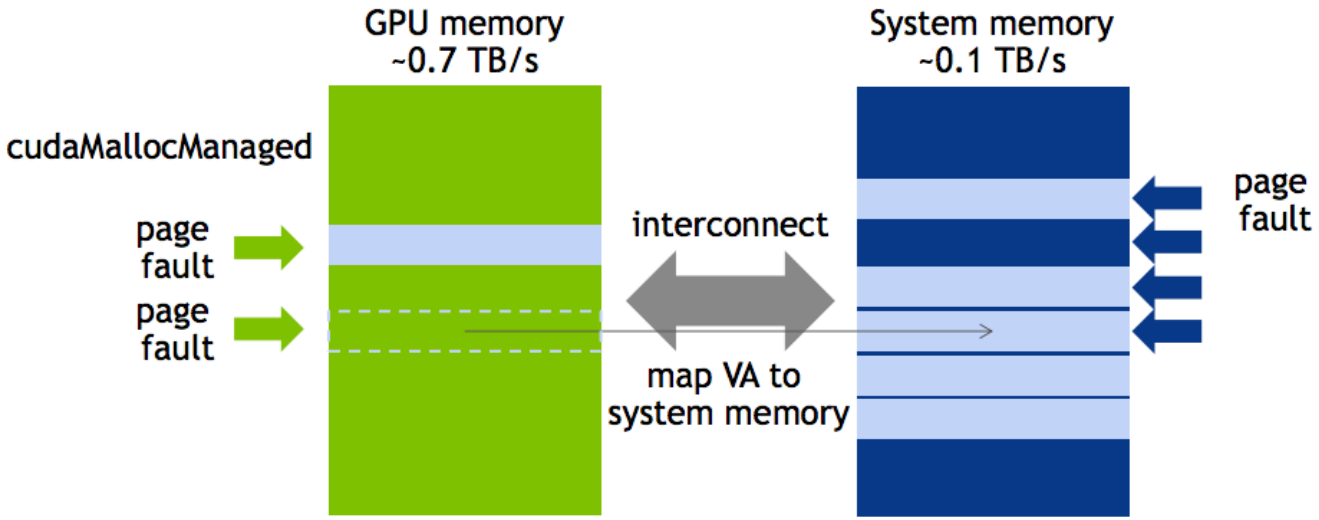
On-demand page migration

```
cudaMallocManaged(&ptr, ...);
*ptr = 1;
qsort<<<...>>>(ptr);
```

← Empty, no pages anywhere (similar to malloc)

← **CPU page fault**: data allocates on CPU

← GPU page fault: data migrates to GPU



USER HINTS

why, when, and how to use them

If you know your application well you can optimize with hints

These are also useful to override some of the driver heuristics

cudaMemPrefetchAsync(ptr, size, processor, stream)

Similar to move_pages() in Linux

cudaMemAdvise(ptr, size, advice, processor)

Similar to madvise() in Linux

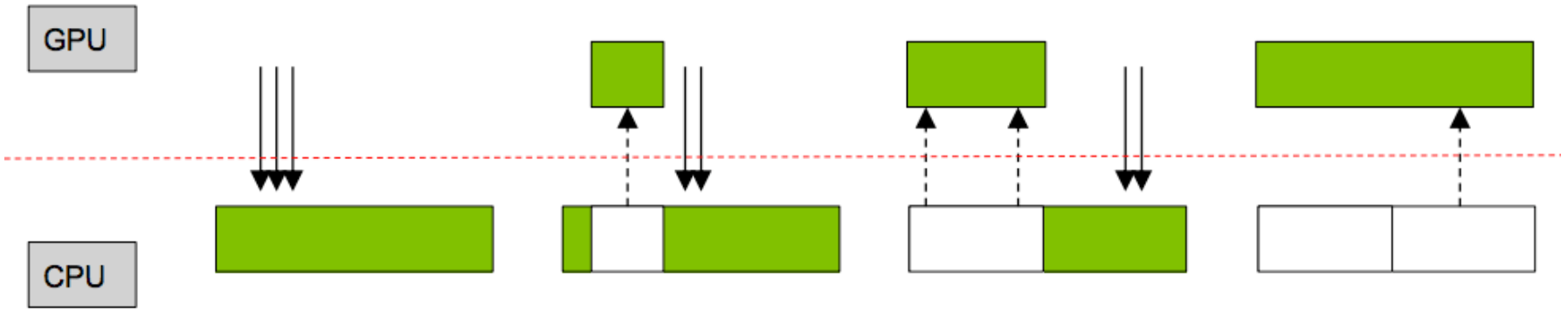
DRIVER PREFETCHING

GPU architecture supports diferent page sizes

Contiguous pages up to a larger page size are promoted to the larger size

Driver prefetches whole regions if pages are accessed densely

You can't control them but you can override most of these with hints



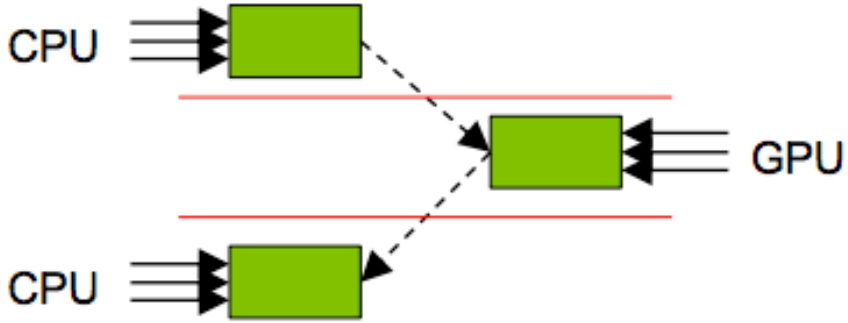
USER HINTS

Prefetching

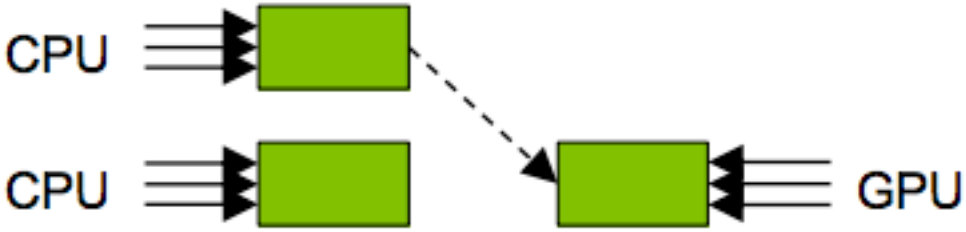
```
char *data;
cudaMallocManaged(&data, N);

init_data(data, N);

cudaMemPrefetchAsync(data, N, myGpuId, s);
mykernel<<<..., s>>>(data, N);
cudaMemPrefetchAsync(data, N, cudaCpuDeviceId, s);
cudaStreamSynchronize(s);

use_data(data, N);

cudaFree(data);
```

Page faults can be expensive and they stall SM execution

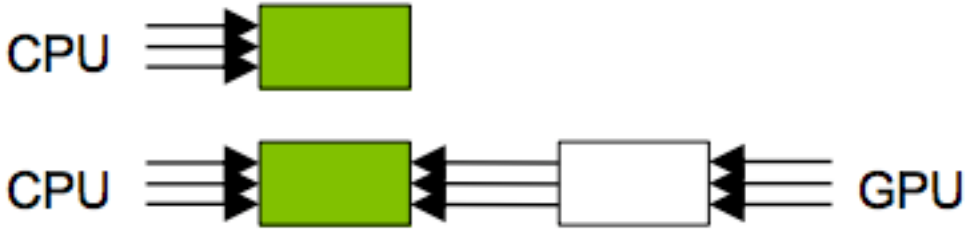
Avoid faults by prefetching data to the accessing processor



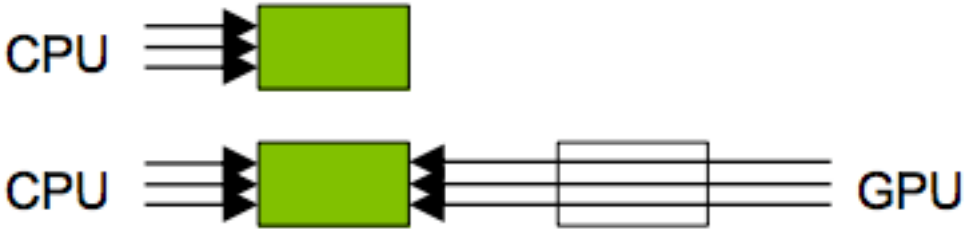
USER HINTS FOR PREFETCH




```
cudaMemAdvice(ptr, size, cudaMemAdviseSetReadMostly, gpuId)
```

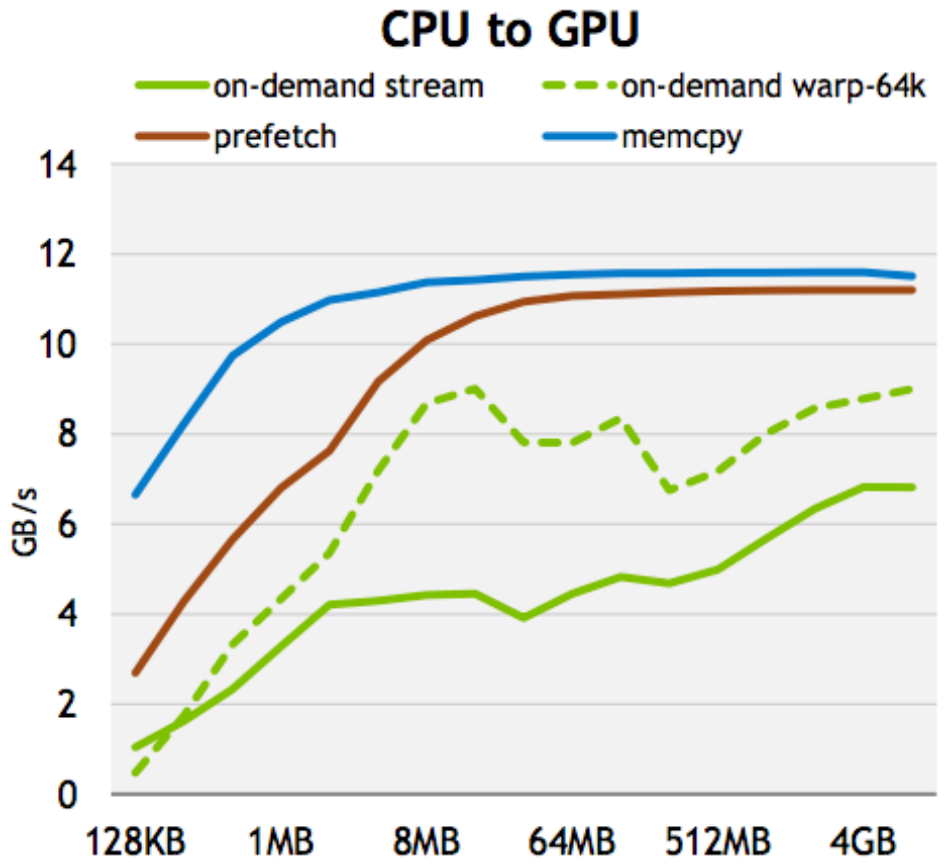


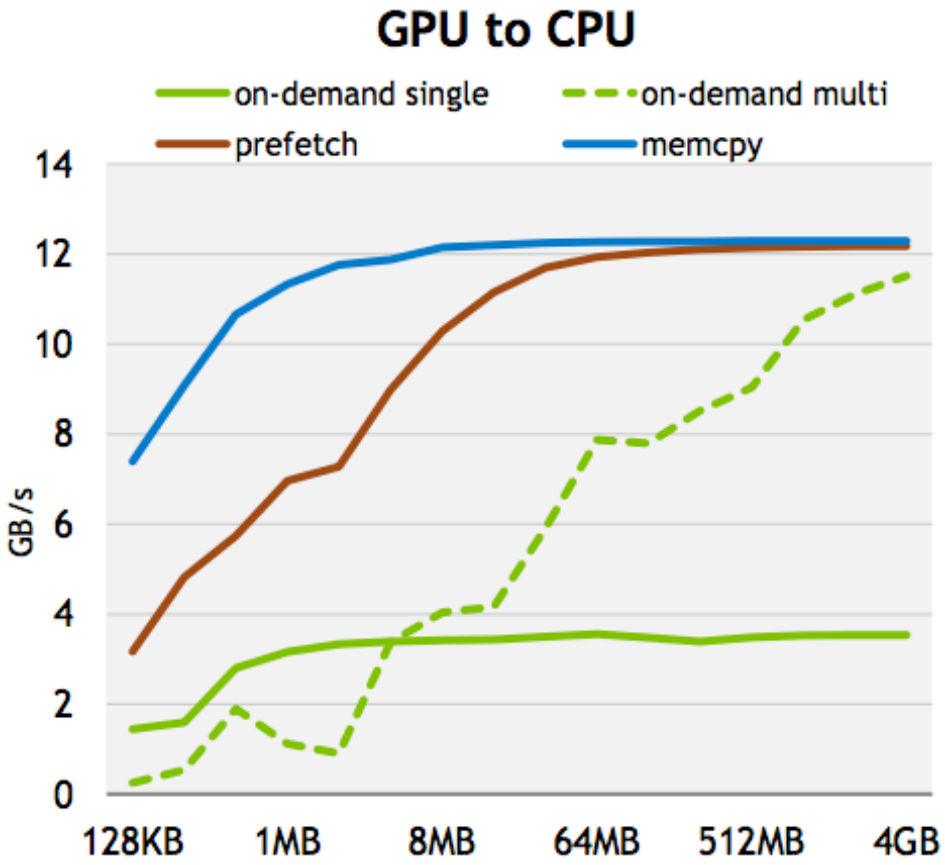

```
cudaMemAdvice(data, size, cudaMemAdvisePreferedLocation, gpuId)
```




```
cudaMemAdvice(ptr, size, cudaMemAdviceAccessedBy, gpuId)
```

UNIFIED MEMORY PERFORMANCE





GPU MULTI-PROCESS SCHEDULING

Background

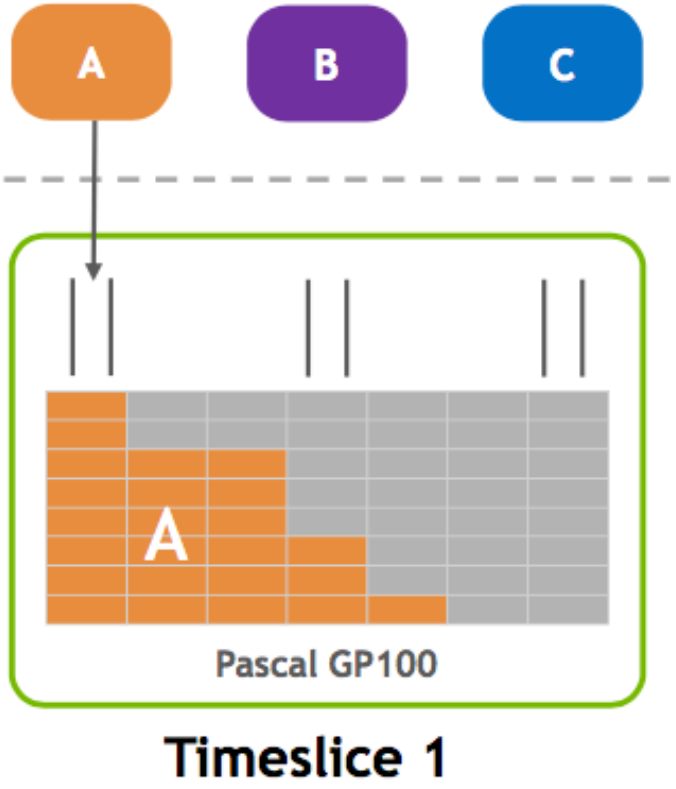
Timeslice Scheduling

Single-process throughput optimized

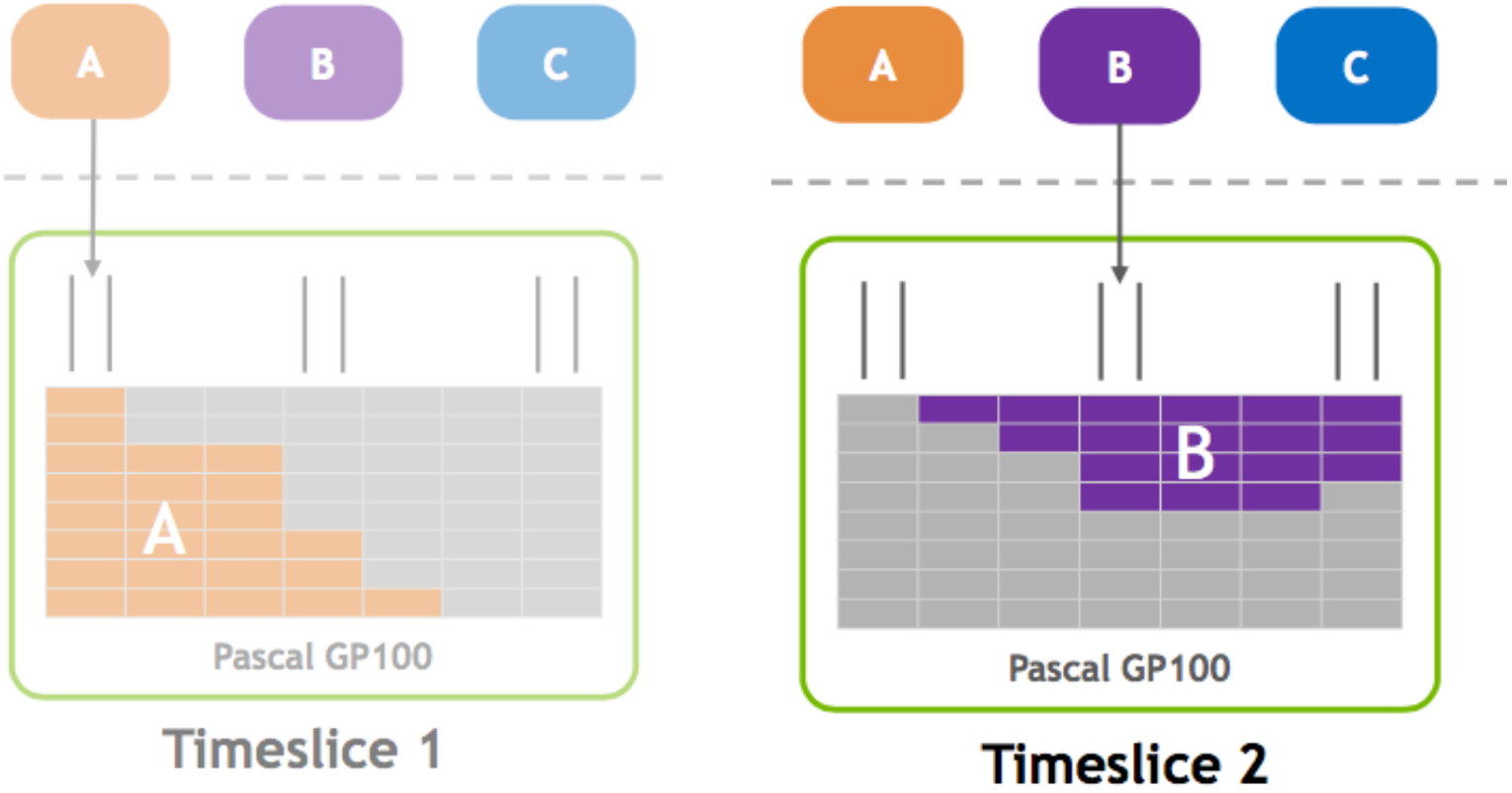
Multi-Process Service

Multi-process throughput optimized

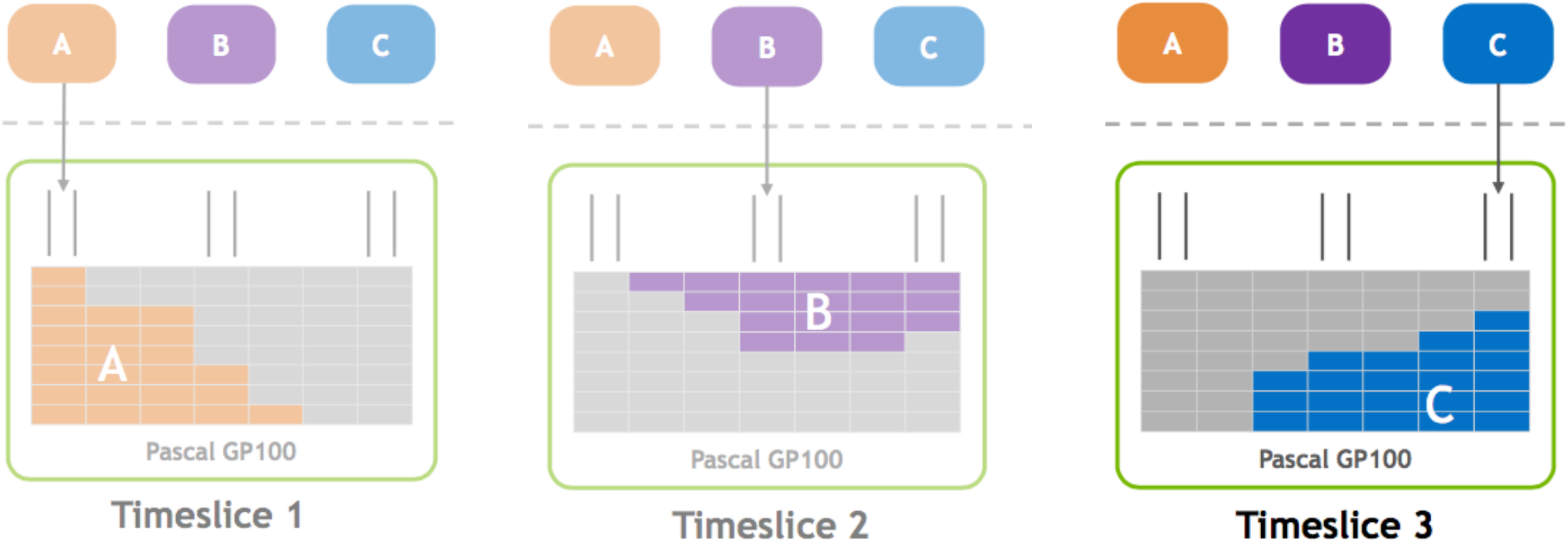
MULTI-PROCESS TIME SLICING



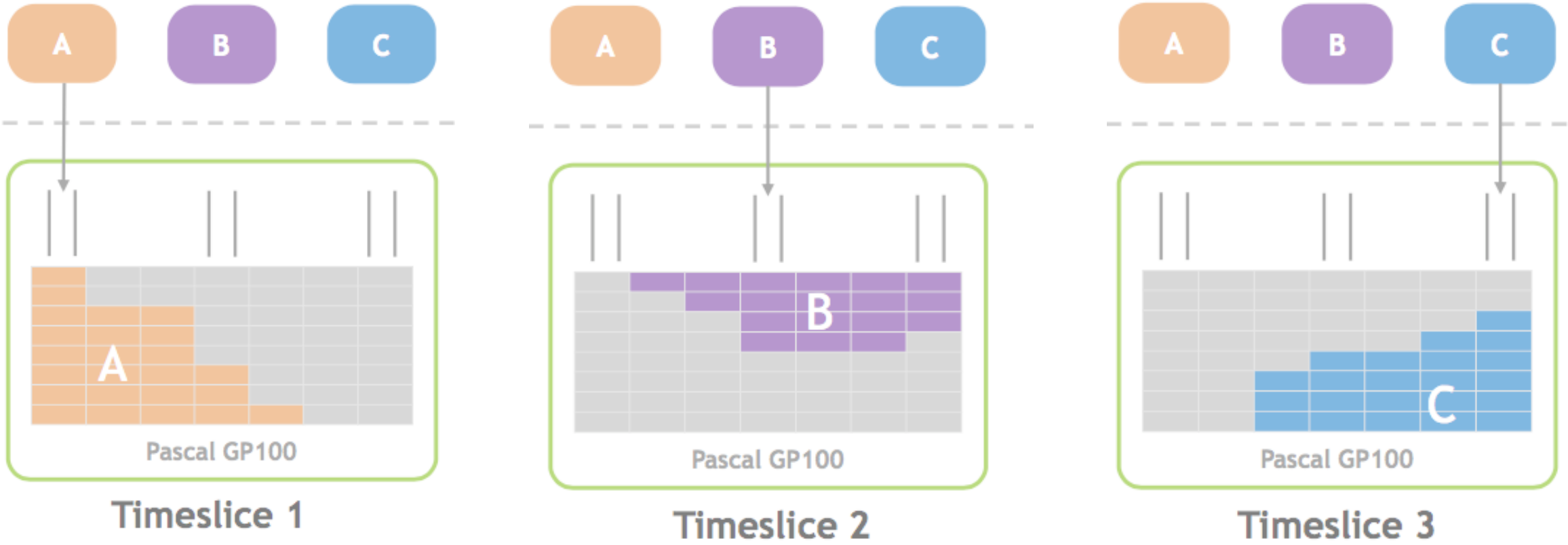
MULTI-PROCESS TIME SLICING



MULTI-PROCESS TIME SLICING



MULTI-PROCESS TIME SLICING



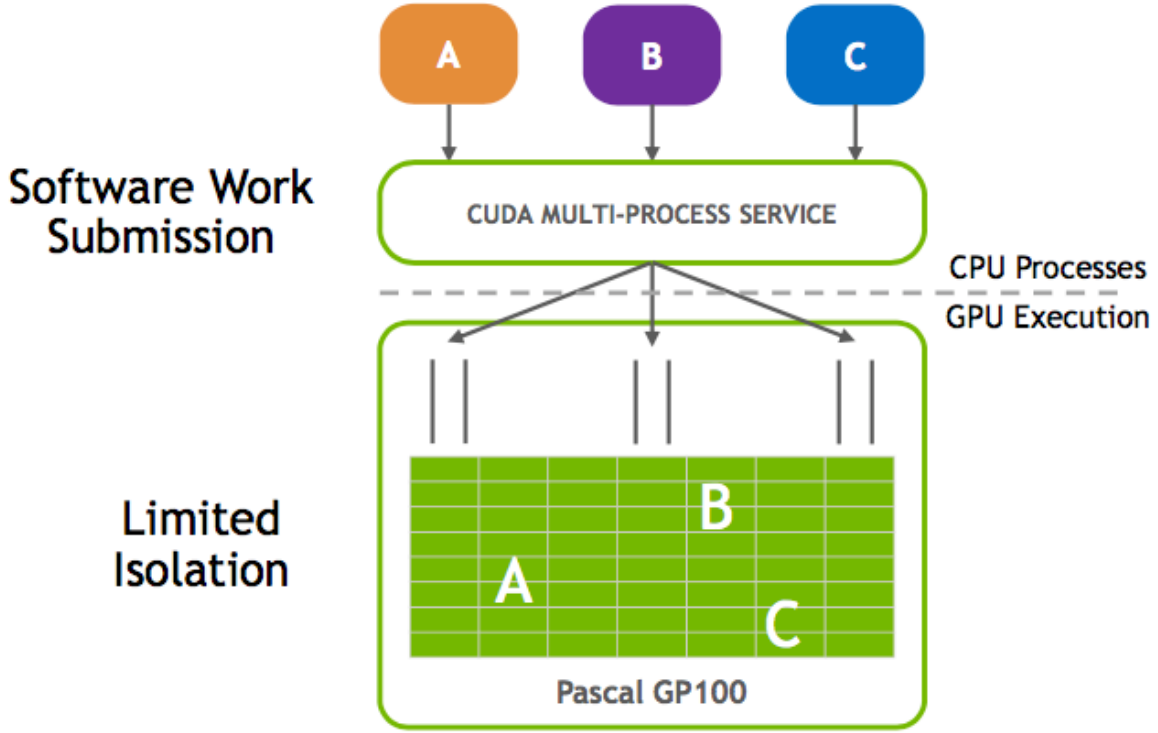
Full process isolation, peak throughput optimized for each process

PASCAL MULTI-PROCESS SERVICE

CUDA Multi-Process Service

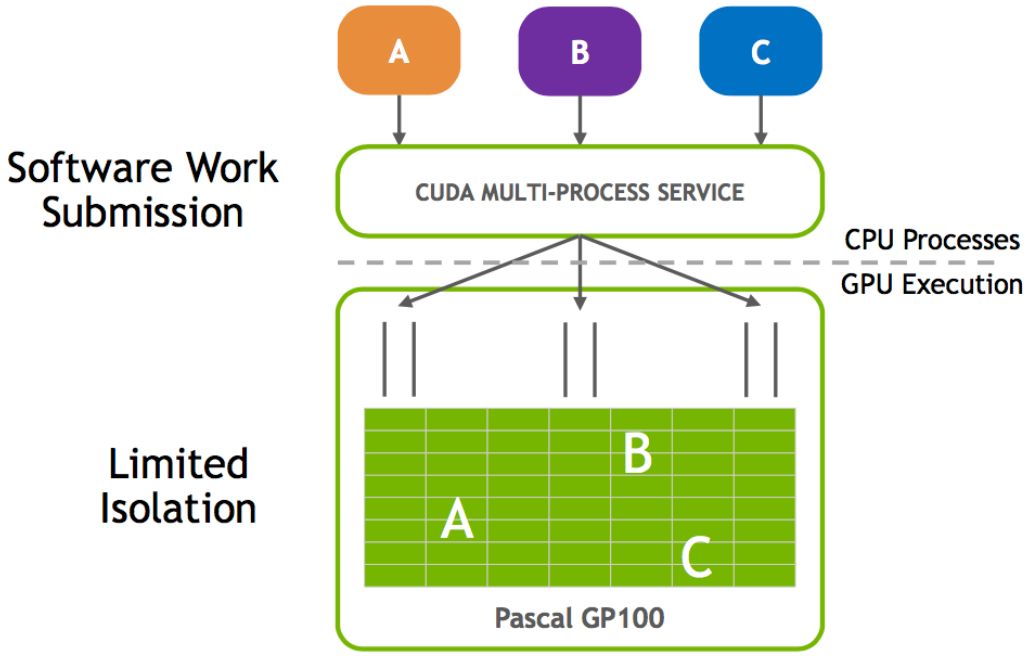
Improves GPU utilization by sharing resources amongst small jobs

Supports up to 16 processes

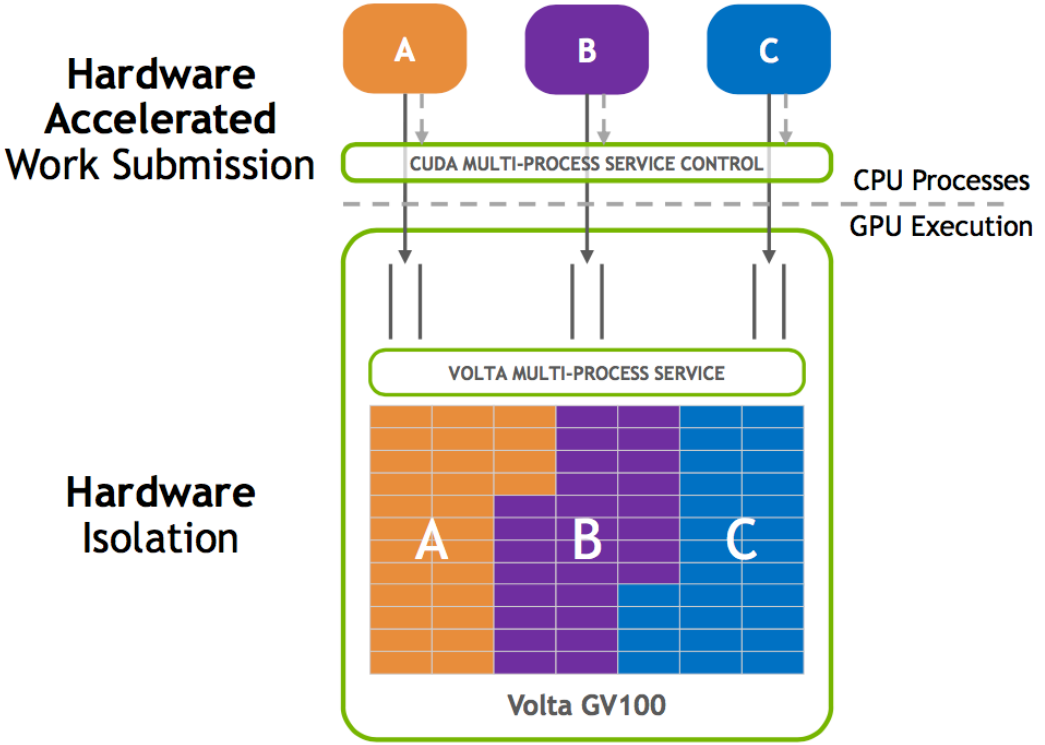


MPS ENABLEMENT

Reduce launch latency / Improve Utilization



Pascal
16 process client/card



Volta
48 process client/card

NVIDIA.

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)

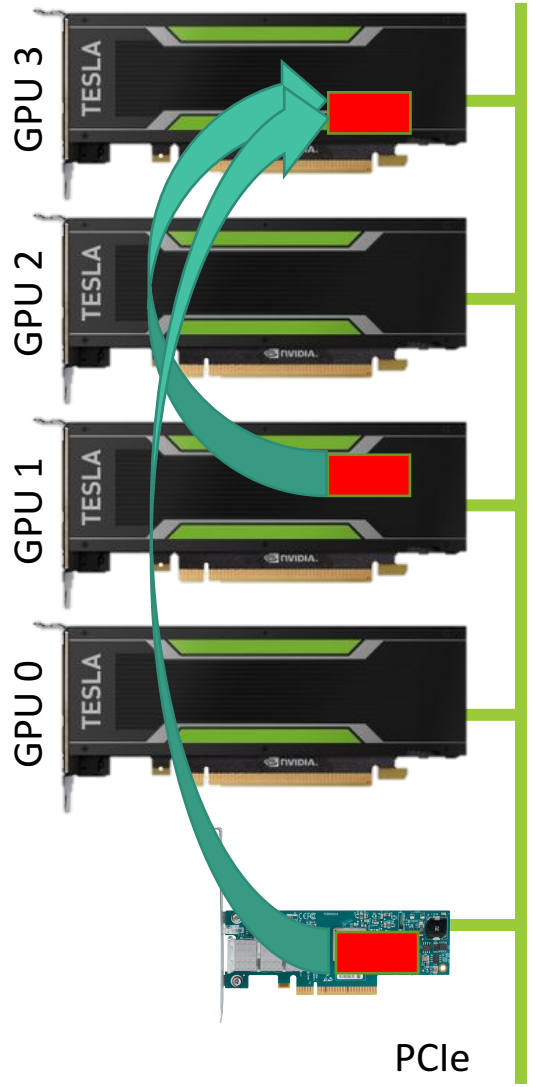
GPU DIRECT

Peer-to-Peer memory copy

Peer Access

RDMA

Sharing Pinned Memory



211

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)

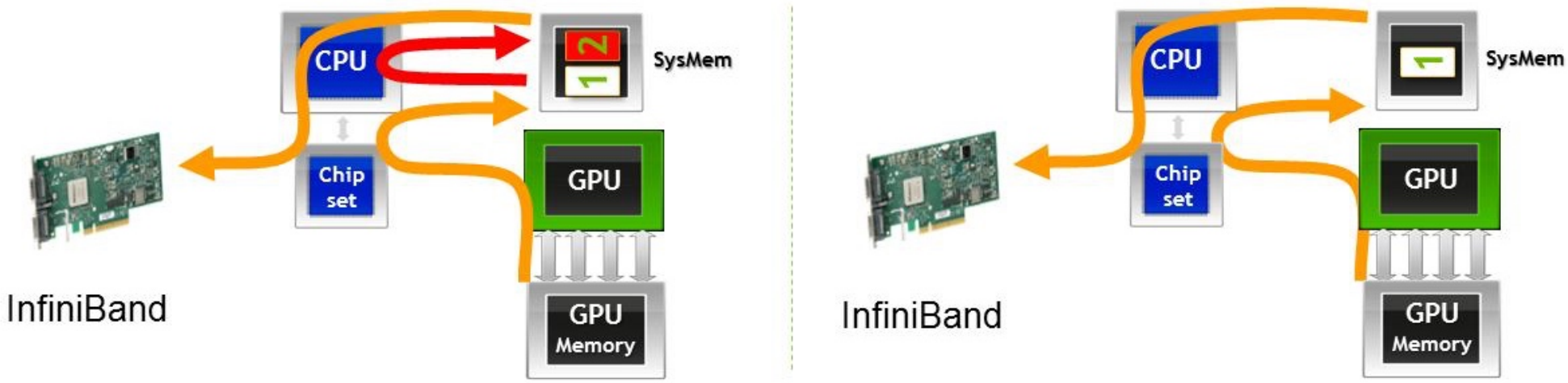
다른 GPU에서 PeerAcceess를 활성화한 GPU의 메모리 접근 가능

Tesla, Quadro Only

SHARING PINNED MEMORY

다른 장치의 Pinned Memory (DMA enabled memory)와 공유 가능

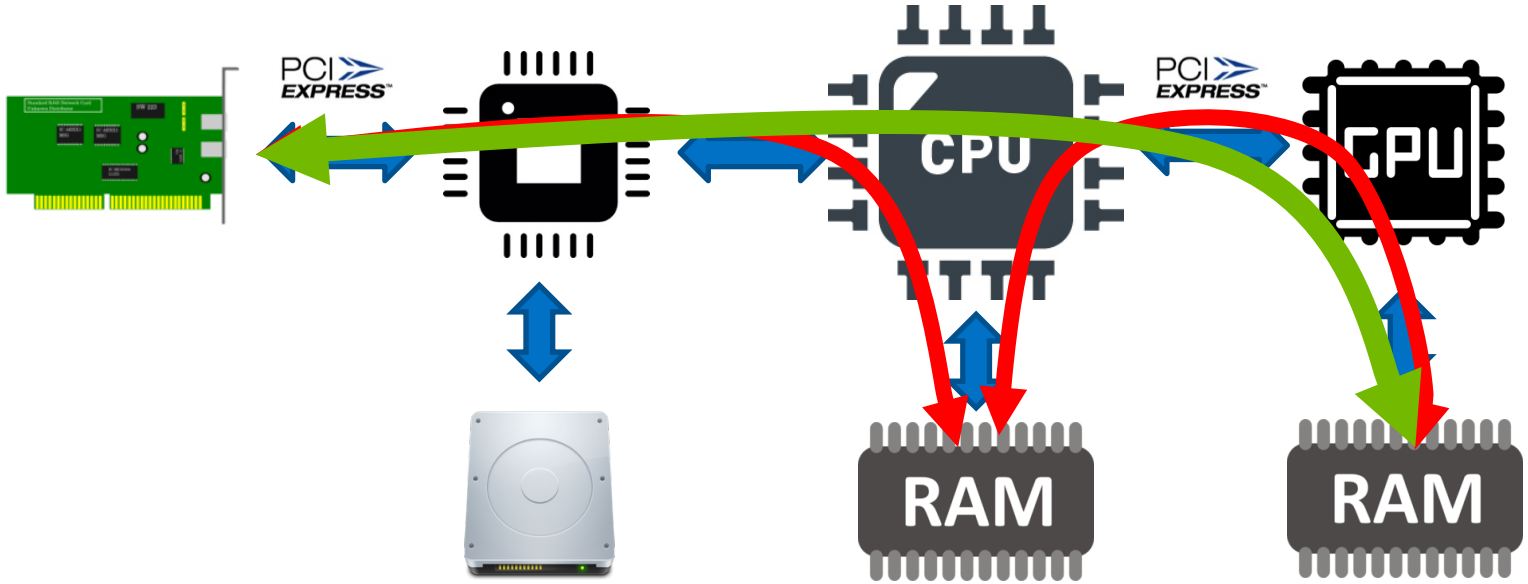
메모리 공간 확보 및 복사에 따른 지연시간 감소



RDMA

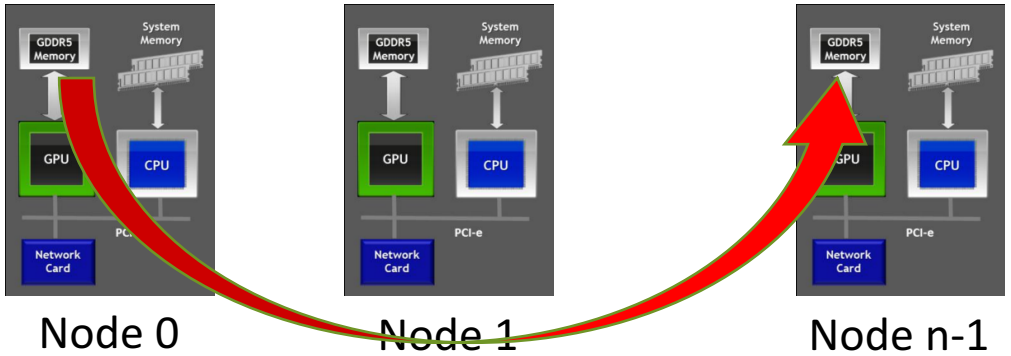
3rd party DMA 장비간 데이터 전송을 효율화 하는 방법

Quadro / Tesla만 지원



MPI + CUDA

- MPI
  - Process간 Message를 주고받기 위한 interface
  - 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, …);
```

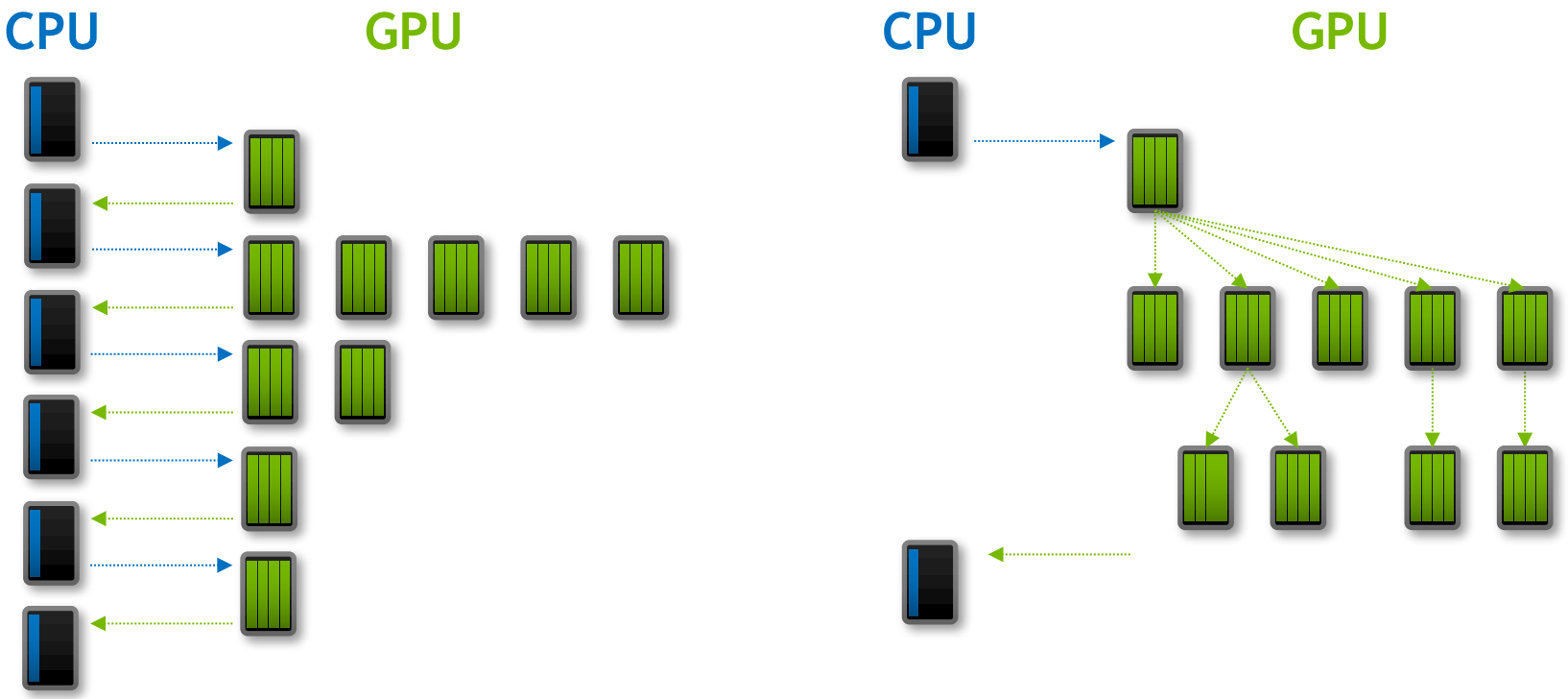
MPI + CUDA

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;
}
```

DYNAMIC PARALLELISM



FAMILIAR SYNTAX

CUDA from CPU

```
void main() {
    float *data;
    do_stuff(data);

    A <<< ... >>> (data);
    B <<< ... >>> (data);
    C <<< ... >>> (data);

    cudaDeviceSynchronize();
    do_more_stuff(data);
}
```

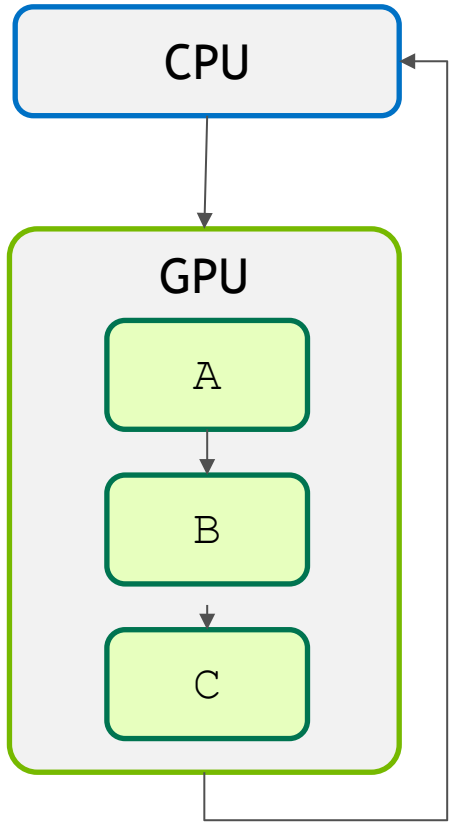
CUDA from GPU

```
__global__ void B(float *data) {
    do_stuff(data);

    X <<< ... >>> (data);
    Y <<< ... >>> (data);
    Z <<< ... >>> (data);

    cudaDeviceSynchronize();
    do_more_stuff(data);
}
```

DEPENDENCIES IN CUDA KERNELS

CUDA from CPU

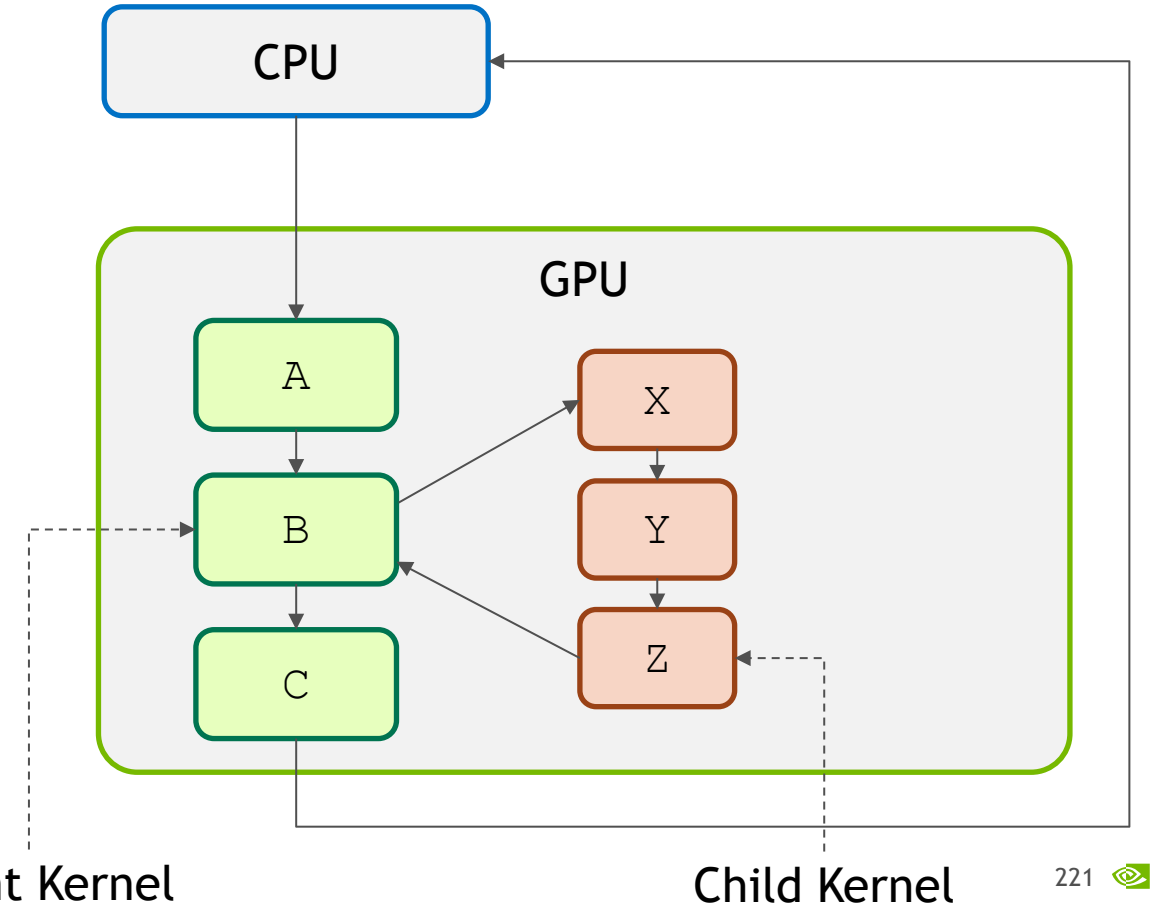
```
void main() {
    float *data;
    do_stuff(data);

    A <<< ... >>> (data);
    B <<< ... >>> (data);
    C <<< ... >>> (data);

    cudaDeviceSynchronize();
    do_more_stuff(data);
}
```



NESTED DEPENDENCIES

CUDA from CPU

```
__global__ void B(float *data) {
    do_stuff(data);

    X <<< ... >>> (data);
    Y <<< ... >>> (data);
    Z <<< ... >>> (data);

    cudaDeviceSynchronize();
    do_more_stuff(data);
}
```



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!)

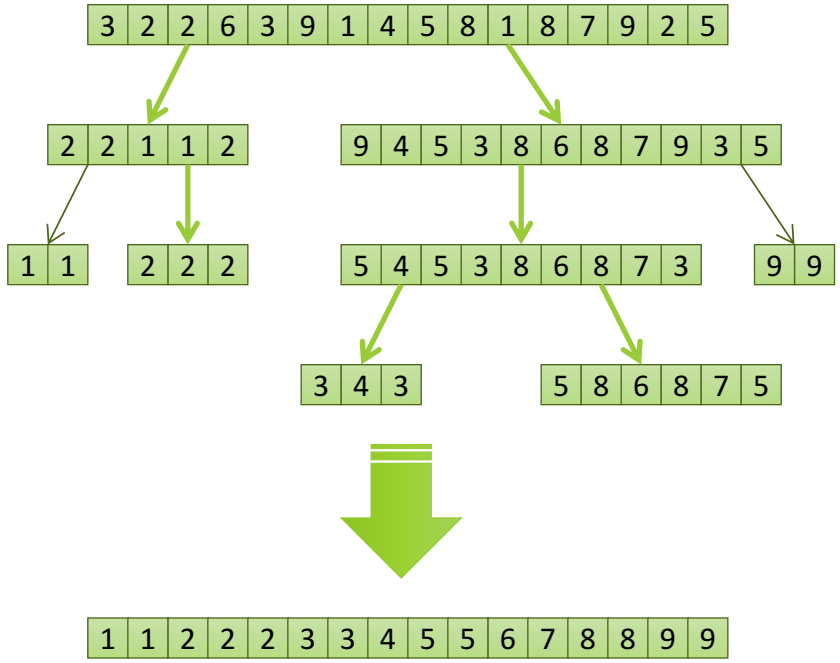
```
__device__ float buf[1024];
__global__ void cnp(float *data)
{
    int tid = threadIdx.x;
    if(tid % 2)
        buf[tid/2] = data[tid]+data[tid+1];
    __syncthreads();

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

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

EXAMPLE: PARALLEL RECURSION

Divide & conquer 알고리즘

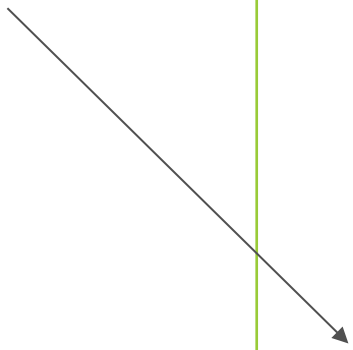
재귀적으로 partition & Sort

데이터에 따라 동작이 변함



RECURSION EXAMPLE

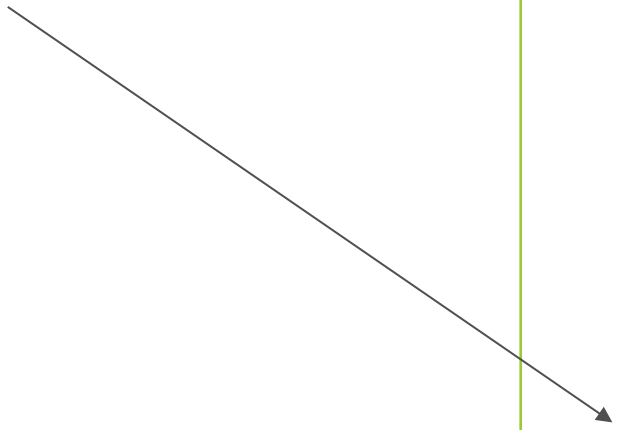
- Launch Kernels in Kernel function



```
__global__ void qsort(int *data, int l, int r) {
    int pivot = data[0];
    int *lptr = data+l, *rptr = data+r;

    // Partition data around pivot value
    partition(data, l, r, lptr, rptr, pivot);

    // Launch next stage recursively
    if(l < (rptr-data))
        qsort<<< ... >>>(data, l, rptr-data);
    if(r > (lptr-data))
        qsort<<< ... >>>(data, lptr-data, r);
}
```

PARALLEL RECURSION EXAMPLE

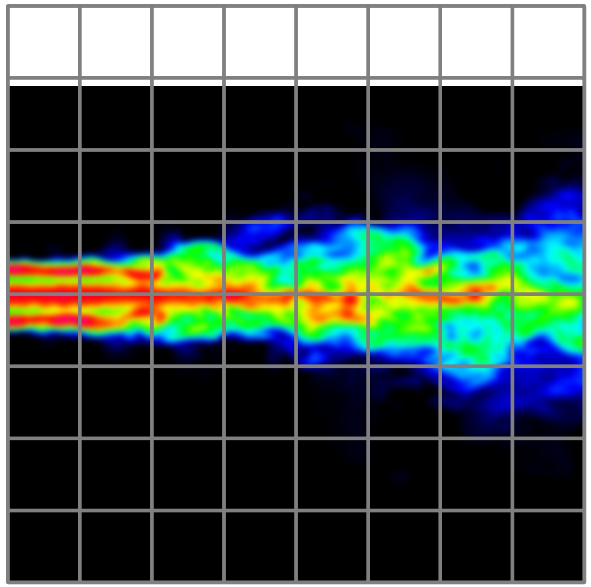
- Achieve concurrency by launching left- and right-hand sorts in saperated streams



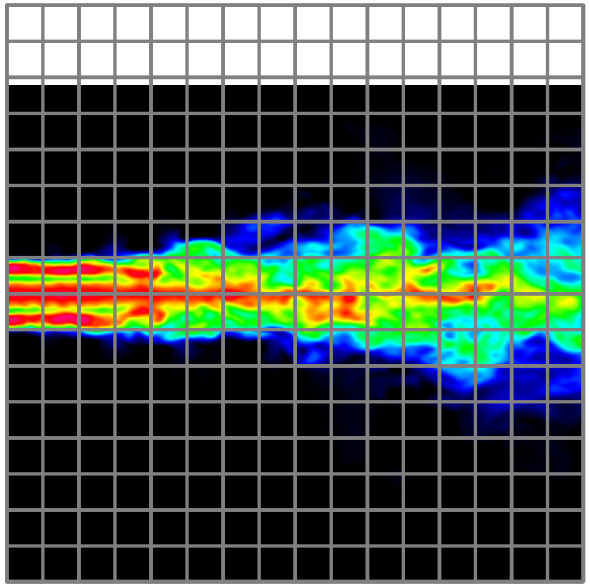
```
__global__ void qsort(int *data, int l, int r) {
    int pivot = data[0];
    int *lptr = data+l, *rptr = data+r;

    // Partition data around pivot value
    partition(data, l, r, lptr, rptr, pivot);

    // Use stream
    cudaStream_t s1, s2;
    cudaStreamCreateWithFlags(&s1, ...);
    cudaStreamCreateWithFlabs(&s2, ...);

    // Launch next stage recursively
    if(l < (rptr-data))
        qsort<<< ..., 0, s1 >>>(data, l, rptr-data);
    if(r > (lptr-data))
        qsort<<< ..., 0, s2 >>>(data, lptr-data, r);
}
```

적절한 단위로 병렬화 가능

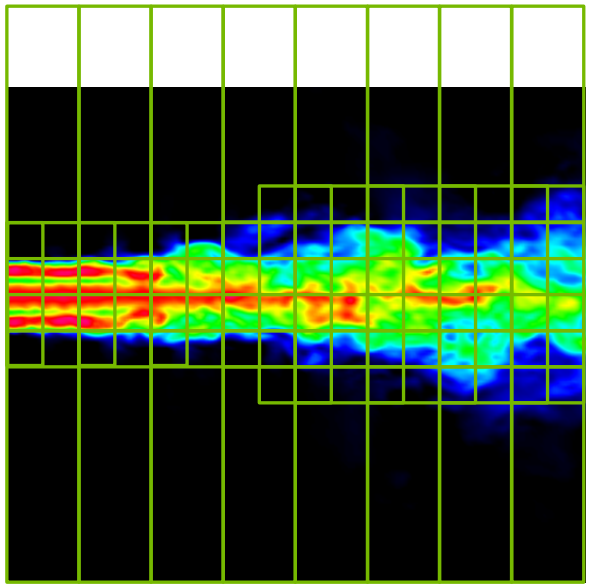
Too coarse



Too fine



Just right



THANKS