

The background of the slide is a close-up photograph of numerous water bubbles of various sizes rising through clear blue water. The bubbles are in sharp focus in the foreground, showing their spherical shape and the way they reflect light. The water has a deep blue hue, and the overall scene is dynamic and textured.

GPU computing & CUDA programming

PDS Lab

Spring 2022

Outline

- **Introduction to Graphic Processing Unit (GPU)**
- **GPU basics**
 - Hardware
 - GPU memory hierarchy
 - Execution Model
- **CUDA basics**
 - What is CUDA ?
 - Function and Variable Qualifiers.
 - Examples of CUDA Programs

Outline

- **Introduction to Graphic Processing Unit (GPU)**
- **GPU basics**
 - Hardware
 - GPU memory hierarchy
 - Execution Model
- **CUDA basics**

What is GPU?

- **GPU (Graphic Processing Unit)** GPUs originated as devices used to accelerate the rendering of images for display (e.g. video games)
- Today, these devices are used to accelerate many different computations
- **GPUs do not typically have:**
 - Out-of-order execution
 - Prefetching
 - Branch prediction



Applications for GPUs

- **GPU is specialized for compute-intensive, highly data parallel computation**
- **Ideal GPU applications have:**
 - Large data sets
 - High parallelism
 - Minimal dependency between data elements

Applications for GPU

- GPU is specialized for compute-intensive, highly data parallel computation

- Visual Computing
- Physics Engines
- Image Processing
- Augmented Reality
- Natural Speech Recognition
- Computational Photography
- Cryptography
- 3D Graphics

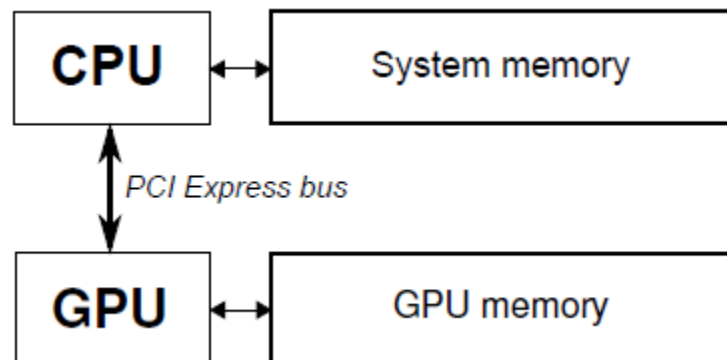


GPU computing

- GPUs as (single instruction, multiple data) SIMD processors.
- The GPUs address problems that can be expressed as **data-parallel** computations.
- Because the same program is executed for each data element, there is a **lower requirement for sophisticated flow control**;

GPU computing

- GPUs traditionally support **graphics computing** to increase processing speed.
- GPUs computing power is huge → be interested by scientists for scientific computing
- Modern GPUs now also allow **general purpose computing** easily



GPU computing

- A Graphics Processor Unit (GPU) is mostly known for the hardware device used when running applications that weigh heavy on graphics.
- Today, GPGPU's (General Purpose GPU) are the choice of hardware to accelerate computational workloads in modern High-Performance Computing (HPC) landscapes.
- GPGPU is not only about ML computations that require image recognition anymore. Calculations on tabular data is also a common exercise.

Oxen vs. Chickens

“If you were plowing a field, which would you rather use? Two strong oxen or 1024 chickens?”

—Seymour Cray

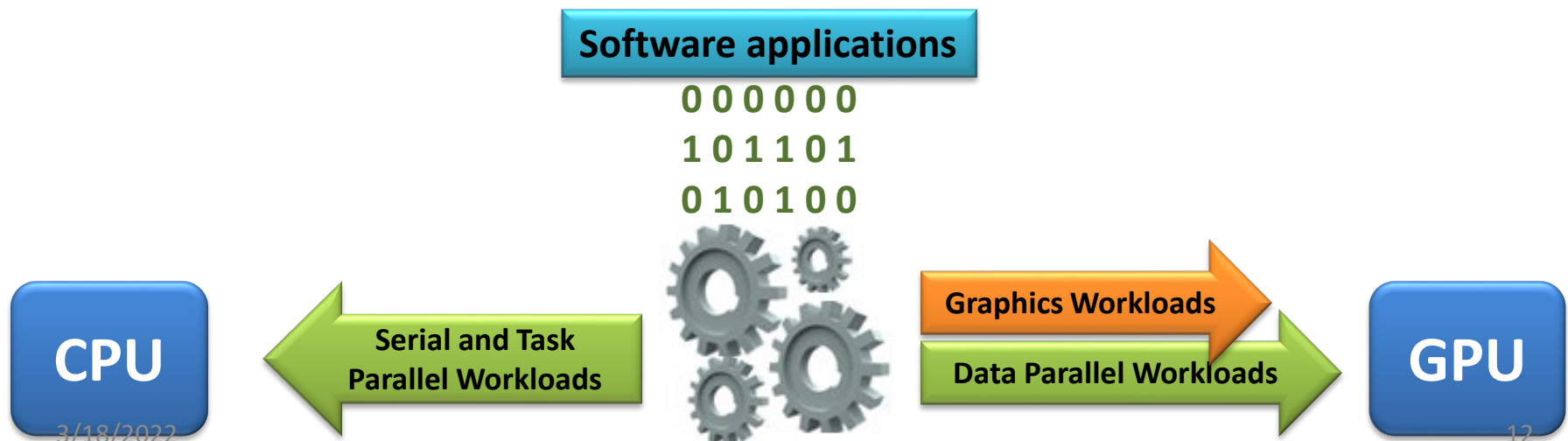


CPU vs. GPU

A CPU is a small collection of very powerful cores where a GPU is a large collection of only modestly powerful cores.

GPUs for parallel computing

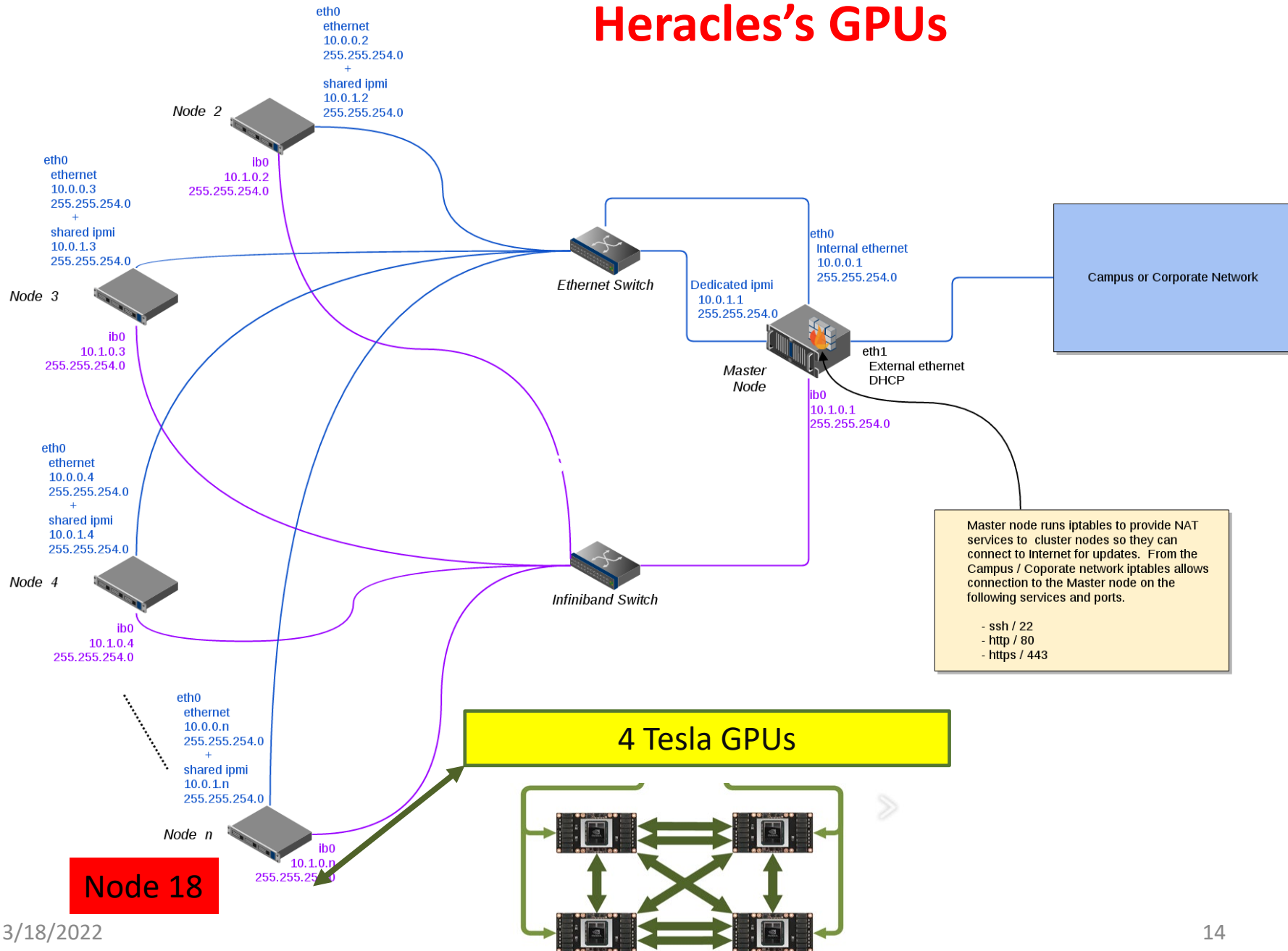
CPU	GPU
Designed with Big Caches	Designed with Arithmetic Intensity
Latency optimized	Throughput optimized
Best for Task Parallelism	Best for Data Parallelism



Outline

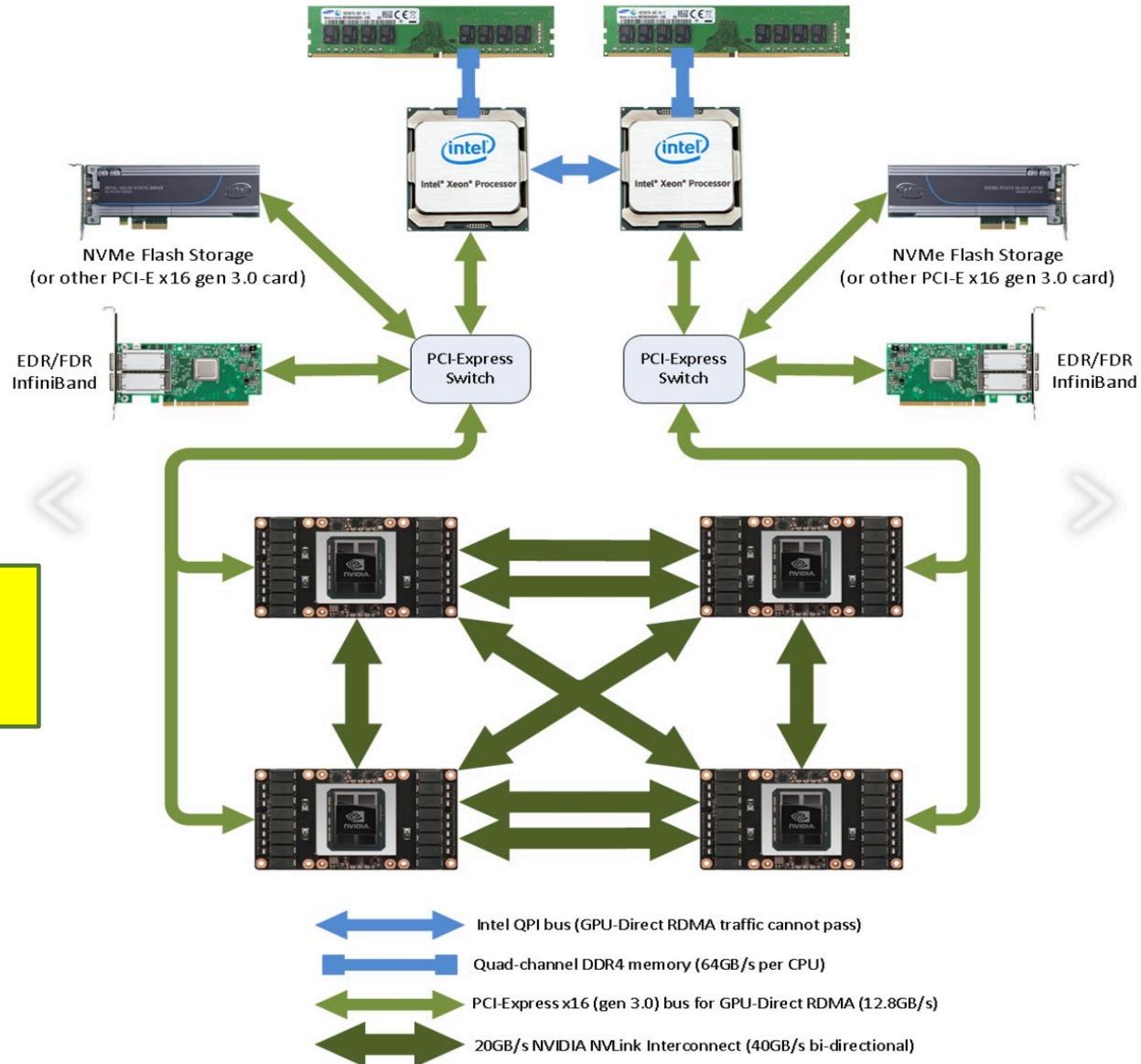
- Introduction to Graphic Processing Unit (GPU)
- GPU basics
 - Hardware
 - GPU memory hierarchy
 - Execution Model
- CUDA basics

Heracles's GPUs



Heracles's GPUs

NumberSmasher 1U Tesla GPU Server with NVLink



GPU Architecture

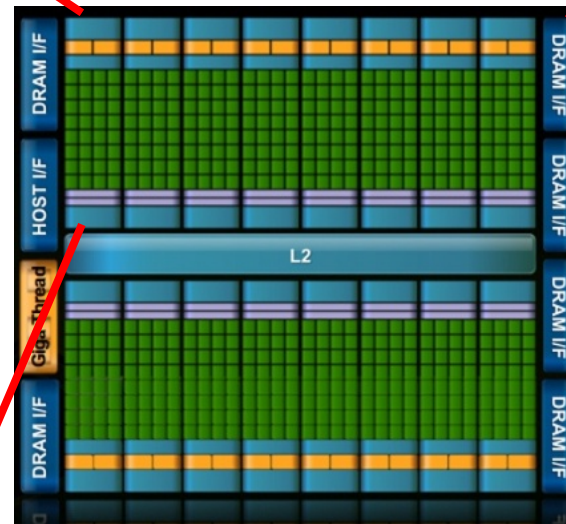
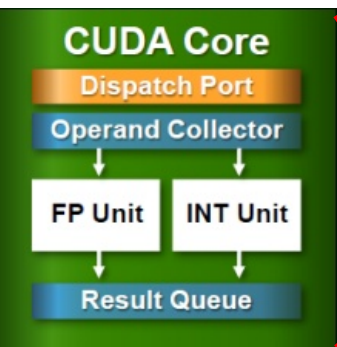
Fermi S2050 Tesla GPU (on Hydra) – Launch 2011

One SM has 32 cores

16 Stream Multiprocessors (SM)

GPU

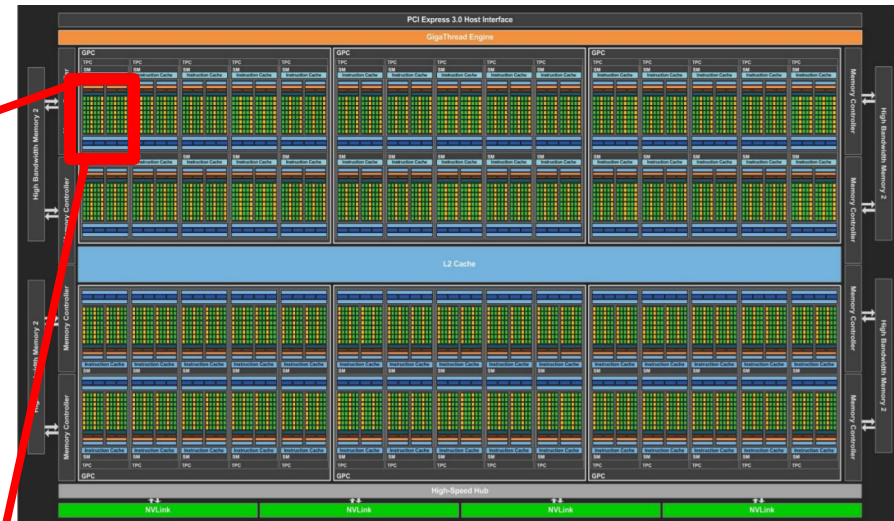
- Each GPU has ≥ 1 Streaming Multiprocessors (SMs)
- Each SM has a simple SIMD/SIMT Processors



GPU Architecture

Pascal GP100 Tesla GPU (on Heracles) Launch 2016

A Streaming Multiprocessor)



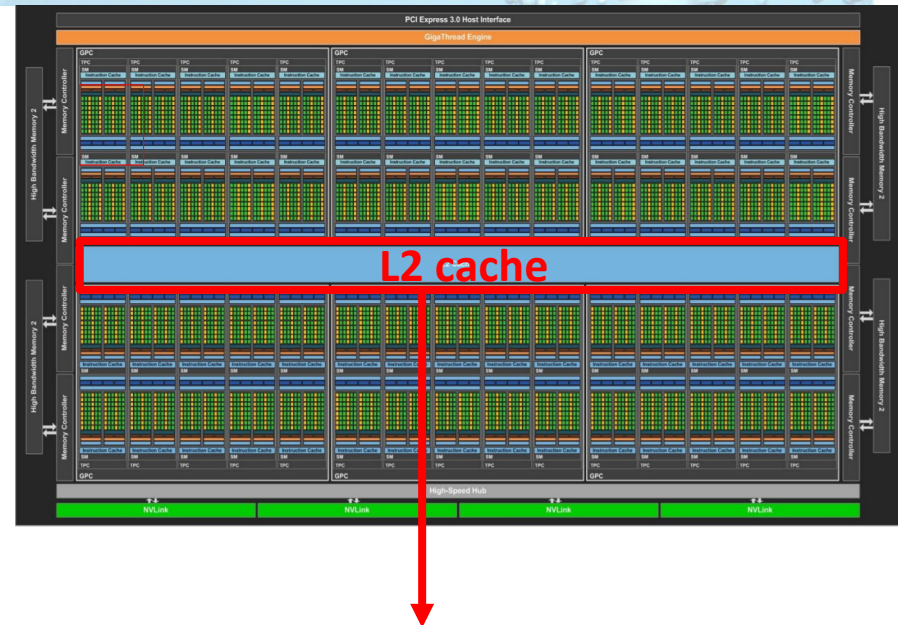
- 56 SM
- 64 CUDA cores per SM
- 3584 cuda cores/GPU (56*64)
- 14336 total cores (3584 * 4GPUs)
- Dual Warp Scheduler

Outline

- **Introduction to Graphic Processing Unit (GPU)**
- **GPU basics**
 - Hardware
 - **GPU memory hierarchy**
 - Execution Model
- **CUDA basics**

GPU Architecture

Pascal GP100 Tesla GPU (on Heracles) Launch 2016



4 MB **L2 cache** shared across SMs

24KB **L1/Texture cache** shared across threads in a same block

49KB **shared memory** shared across threads in a same block

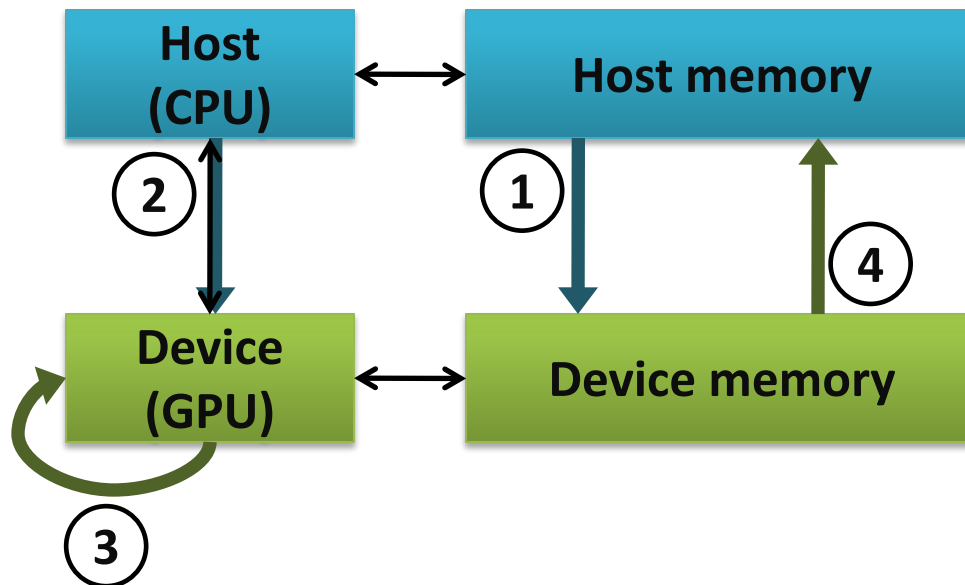
16GB Global memory is separate hardware from the **GPU** core (containing SM's, caches, etc)

Outline

- **Introduction to Graphic Processing Unit (GPU)**
- **GPU basics**
 - Hardware
 - GPU memory hierarchy
 - **Execution Model**
- **CUDA basics**

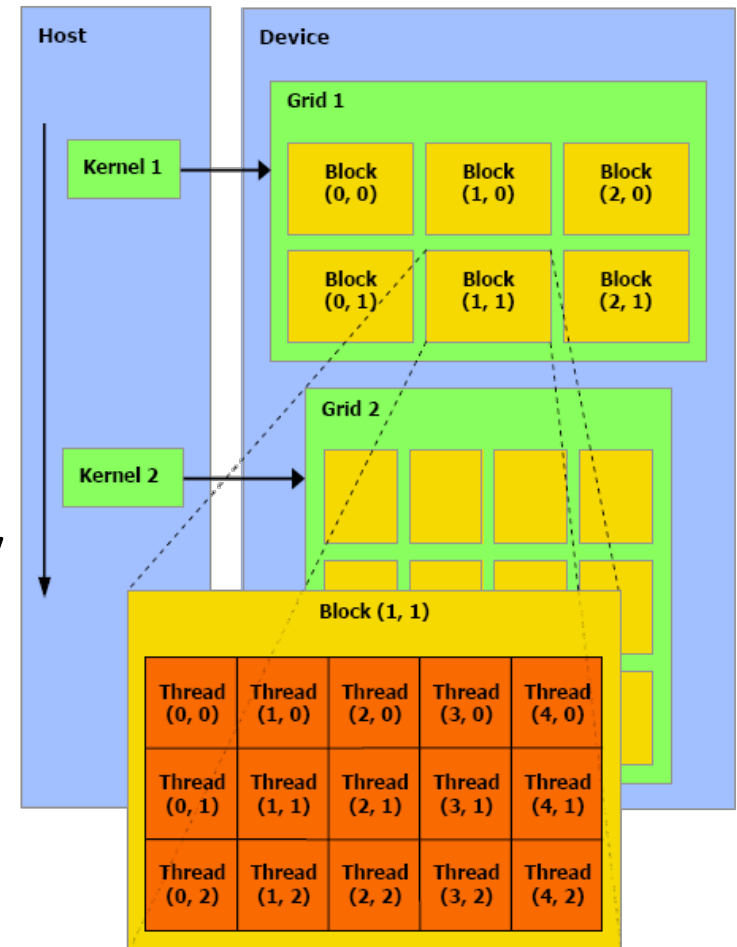
Execution Model

1. CPU sends data to the GPU
2. CPU instructs the processing on GPU
3. GPU processes data
4. CPU collects the results from GPU



Execution Model

- A kernel is executed as a **grid** of **thread blocks**
- A **thread block** is a batch of **threads** that can cooperate with each other by:
 - Sharing data through shared memory
 - Synchronizing their execution
- **Threads** from different blocks cannot cooperate



Execution Model

Software

Thread



Thread Block

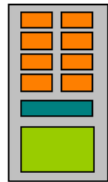


Grid

Hardware



Scalar
Processor



Multiprocessor



Device

Threads are executed by scalar processors

Thread blocks are executed on multiprocessors

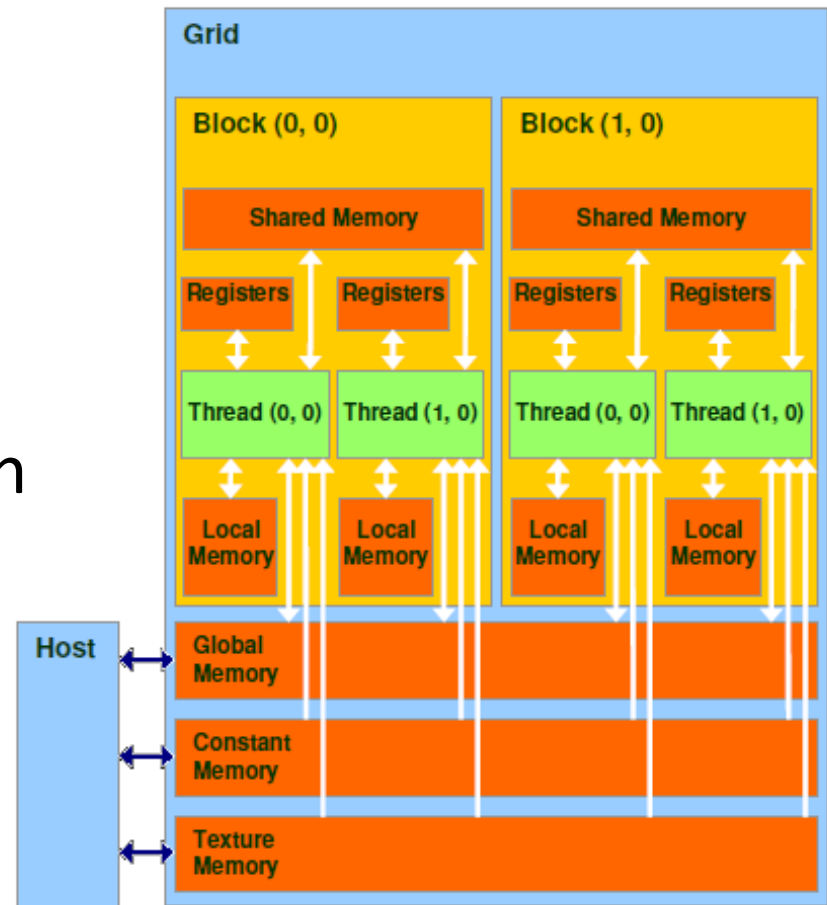
Thread blocks do not migrate

Several concurrent thread blocks can reside on one multiprocessor - limited by multiprocessor resources (shared memory and register file)

A kernel is launched as a grid of thread blocks

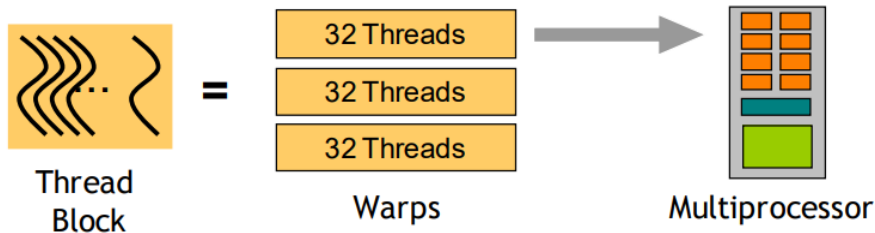
Memory model

- **Local Memory: per-Thread**
 - Private per thread
 - Auto variables, register spill
- **Shared Memory: per-Block**
 - Shared by thread Block
 - Inter-thread communication
- **Global Memory: per-Grid**
 - Shared by all threads
 - Inter-Grid communication



Warps

Threads are executed in warps of 32 threads



A thread block consists of 32-thread warps

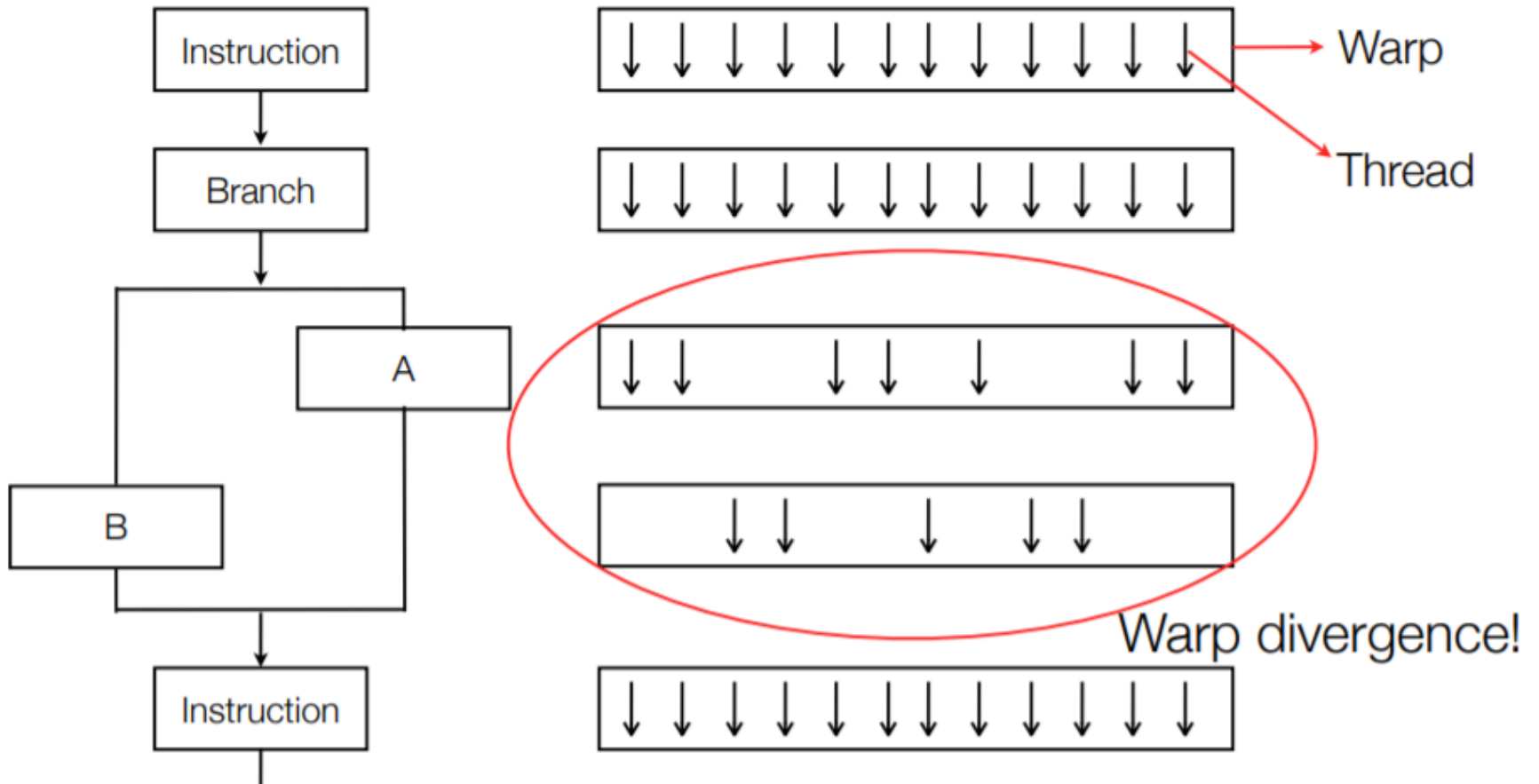
A warp is executed physically in parallel (SIMT) on a multiprocessor

Within a warp, the hardware is not capable of executing if and else statements at the same time! → warp divergence

```
__global__ void function();
{
    ....

    if (condition)
    {    ...
    }
    else
    {    ...
    }
}
```

Warp divergence



Recommendations

- ❑ Try to make every thread in the **same warp** do the **same thing**
- ❑ If the **if statement cuts at a multiple of the warp size**, there is no warp divergence and the instruction can be done in one pass
- ❑ Remember threads are placed consecutively in a warp (t0-t31, t32- t63, ...)
 - But we cannot rely on any execution order within warps
- ❑ If you can't avoid branching, try to make as **many consecutive threads as possible do the same thing**

Outline

- **Introduction to Graphic Processing Unit (GPU)**
- **GPU basics**
 - Hardware
 - Execution Models
 - GPU memory hierarchy
- **CUDA basics**
 - What is CUDA ?
 - Function and Variable Qualifiers.
 - Examples of CUDA Programs

What is CUDA ?

- **CUDA** = **C**ompute **U**nified **D**evice **A**rchitecture
- Parallel programming tool for Nvidia GPUs
- CUDA enables efficient use of the massive parallelism of NVIDIA GPUs
- Extensions of C language
- Support **NVIDIA** GeForce 8-Series & later

Function qualifiers

	Executed on the:	Only callable from the:
__host__ void HostFunc()	Host	Host
__global__ void KernelFunc()	Device	Host
__device__ void DeviceFunc()	Device	Device

Restrictions for device code (**__global__** / **__device__**)

- no recursive call
- no static variable
- no function pointer

__global__ function is asynchronous invoked

__global__ function must have void return type

Variable qualifiers

	Memory	Scope	Lifetime
Automatic none array variables	Register	Thread	Kernel
Automatic array variables	Local (physically in global memory)	Thread	Kernel
<code>__shared__</code> int SharedVar;	Shared	Block	Kernel
<code>__device__</code> int GlobalVar;	Global	Grid	Application
<code>__constant__</code> int ConstantVar;	Constant	Grid	Application

The constant memory space resides in device memory and is cached in the L1/texture cache.

CUDA = C with Language Extensions

- **Execution configuration**

```
dim3 dimGrid(100, 50); // 5000 thread blocks  
dim3 dimBlock(4, 8, 8); // 256 threads per block  
MyKernel <<<dimGrid,dimBlock>>>(...); // Launch  
kernel
```

- **Built-in variables and functions valid in device code:**

```
dim3 gridDim; // Grid dimension  
dim3 blockDim; // Block dimension  
dim3 blockIdx; // Block index  
dim3 threadIdx; // Thread index  
void __syncthreads(); // Thread synchronization
```


CUDA = C with Runtime Extensions

- **Device management**

`cudaGetDeviceCount()`, `cudaGetDeviceProperties()`

- **Device memory management**

`cudaMalloc()`, `cudaFree()`, `cudaMemcpy()`

- **Texture management**

`cudaBindTexture()`, `cudaBindTextureToArray()`

Example of CUDA Program

1. CPU sends data to the GPU

Host Code

```
int N= 1000;  
int size = N*sizeof(float);  
float A[1000], *dA;
```

2. CPU instructs the processing on GPU

```
cudaMalloc((void **)&dA, size);  
cudaMemcpy(dA , A, size, cudaMemcpyHostToDevice);
```

```
ComputeArray <<< 10, 20 >>> (dA ,N);
```

```
cudaMemcpy(A, dA, size, cudaMemcpyDeviceToHost);  
cudaFree(dA);
```

3. GPU processes data

Device Code

```
__global__ void ComputeArray(float *A, int N)  
{  
    int i = blockIdx.x * blockDim.x + threadIdx.x;  
    if (i<N) A[i] = A[i]*A[i];  
}
```


4. CPU collects the results from GPU

Host Synchronization

- All kernel launches are asynchronous
 - control returns to CPU immediately
 - kernel executes after all previous CUDA calls have completed.
- `cudaMemcpy()` is synchronous
 - control returns to CPU after copy completes
 - copy starts after all previous CUDA calls have completed

Example of CUDA program

Remember: CUDA uses **thread id** to select work and address shared data



```
#include <iostream>
using namespace std;
// Kernel definition
__global__ void MatAdd(float A[N], float B[N],
                      float C[N])
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;

    if (i < N )
        C[i] = A[i] + B[i];
}

int main()
{
    int N = ...;
    // Kernel invocation
    int threadsPerBlock = 16;
    int numBlocks = (N / threadsPerBlock);
    MatAdd<<<numBlocks, threadsPerBlock>>>>(A, B, C);
}
```


C++ sequential version

```
#include <iostream>
#include <math.h>

// function to add the elements of two arrays
void add(int n, float *x, float *y)
{
    for (int i = 0; i < n; i++)
        y[i] = x[i] + y[i];
}

int main(void)
{
    int N = 1<<20; // 1M elements

    float *x = new float[N];
    float *y = new float[N];

    // initialize x and y arrays on the host
    for (int i = 0; i < N; i++) {
        x[i] = 1.0f;
        y[i] = 2.0f;
    }

    // Run kernel on 1M elements on the CPU
    add(N, x, y);

    // Check for errors (all values should be 3.0f)
    float maxError = 0.0f;
    for (int i = 0; i < N; i++)
        maxError = fmax(maxError, fabs(y[i]-3.0f));
    std::cout << "Max error: " << maxError << std::endl;

    // Free memory
    delete [] x;
    delete [] y;

    return 0;
}
```

C++ GPU version 1 (1 thread)

```
#include <iostream>
#include <math.h>

// Kernel function to add the elements of two arrays
__global__
void add(int n, float *x, float *y)
{
    for (int i = 0; i < n; i++)
        y[i] = x[i] + y[i];
}

int main(void)
{
    int N = 1<<20;
    float *x, *y;

    // Allocate Unified Memory - accessible from CPU or GPU
    cudaMallocManaged(&x, N*sizeof(float));
    cudaMallocManaged(&y, N*sizeof(float));

    // initialize x and y arrays on the host
    for (int i = 0; i < N; i++) {
        x[i] = 1.0f;
        y[i] = 2.0f;
    }

    // Run kernel on 1M elements on the GPU
    add<<<1, 1>>>>(N, x, y);

    // Wait for GPU to finish before accessing on host
    cudaDeviceSynchronize();

    // Check for errors (all values should be 3.0f)
    float maxError = 0.0f;
    for (int i = 0; i < N; i++)
        maxError = fmax(maxError, fabs(y[i]-3.0f));
    std::cout << "Max error: " << maxError << std::endl;

    // Free memory
    cudaFree(x);
    cudaFree(y);

    return 0;
}
```

C++ GPU version 1 (1 thread)

```
#include <iostream>
#include <math.h>

// Kernel function to add the elements of two arrays
__global__
void add(int n, float *x, float *y)
{
    for (int i = 0; i < n; i++)
        y[i] = x[i] + y[i];
}

int main(void)
{
    int N = 1<<20;
    float *x, *y;

    // Allocate Unified Memory - accessible from CPU or GPU
    cudaMallocManaged(&x, N*sizeof(float));
    cudaMallocManaged(&y, N*sizeof(float));

    // initialize x and y arrays on the host
    for (int i = 0; i < N; i++) {
        x[i] = 1.0f;
        y[i] = 2.0f;
    }

    // Run kernel on 1M elements on the GPU
    add<<<1, 1>>>>(N, x, y);

    // Wait for GPU to finish before accessing on host
    cudaDeviceSynchronize();

    // Check for errors (all values should be 3.0f)
    float maxError = 0.0f;
    for (int i = 0; i < N; i++)
        maxError = fmax(maxError, fabs(y[i]-3.0f));
    std::cout << "Max error: " << maxError << std::endl;

    // Free memory
    cudaFree(x);
    cudaFree(y);

    return 0;
}
```

C++ GPU version 2 (256 threads)

```
#include <iostream>
#include <math.h>

// function to add the elements of two arrays
void add(int n, float *x, float *y)
{
    int index = threadIdx.x;
    int stride = blockDim.x;
    for (int i = index; i < n; i += stride)
        y[i] = x[i] + y[i];
}

int main(void)
{
    int N = 1<<20; // 1M elements

    float *x = new float[N];
    float *y = new float[N];

    // initialize x and y arrays on the host
    for (int i = 0; i < N; i++) {
        x[i] = 1.0f;
        y[i] = 2.0f;
    }

    // Run kernel on 1M elements on the CPU
    add<<<1, 256>>>>(N, x, y);

    // Check for errors (all values should be 3.0f)
    float maxError = 0.0f;
    for (int i = 0; i < N; i++)
        maxError = fmax(maxError, fabs(y[i]-3.0f));
    std::cout << "Max error: " << maxError << std::endl;

    // Free memory
    delete [] x;
    delete [] y;

    return 0;
}
```

C++ GPU version 2 (256 threads)

```
#include <iostream>
#include <math.h>

// function to add the elements of two arrays
void add(int n, float *x, float *y)
{
    int index = threadIdx.x;
    int stride = blockDim.x;
    for (int i = index; i < n; i += stride)
        y[i] = x[i] + y[i];
}

int main(void)
{
    int N = 1<<20; // 1M elements

    float *x = new float[N];
    float *y = new float[N];

    // initialize x and y arrays on the host
    for (int i = 0; i < N; i++) {
        x[i] = 1.0f;
        y[i] = 2.0f;
    }

    // Run kernel on 1M elements on the GPU
    add<<<1, 256>>>>(N, x, y);

    // Check for errors (all values should be 3.0f)
    float maxError = 0.0f;
    for (int i = 0; i < N; i++)
        maxError = fmax(maxError, fabs(y[i]-3.0f));
    std::cout << "Max error: " << maxError << std::endl;

    // Free memory
    delete [] x;
    delete [] y;

    return 0;
}
```

C++ GPU version 2 (multiple blocks)

```
#include <iostream>
#include <math.h>

// function to add the elements of two arrays
void add(int n, float *x, float *y)
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i < n)
        y[i] = x[i] + y[i];
}

int main(void)
{
    int N = 1<<20; // 1M elements

    float *x = new float[N];
    float *y = new float[N];

    // initialize x and y arrays on the host
    for (int i = 0; i < N; i++) {
        x[i] = 1.0f;
        y[i] = 2.0f;
    }

    // Run kernel on 1M elements on the GPU
    int blockSize = 256;
    int numBlocks = (N + blockSize - 1) / blockSize;
    add<<<numBlocks, blockSize>>>>(N, x, y);

    // Check for errors (all values should be 3.0f)
    float maxError = 0.0f;
    for (int i = 0; i < N; i++)
        maxError = fmax(maxError, fabs(y[i]-3.0f));
    std::cout << "Max error: " << maxError << std::endl;

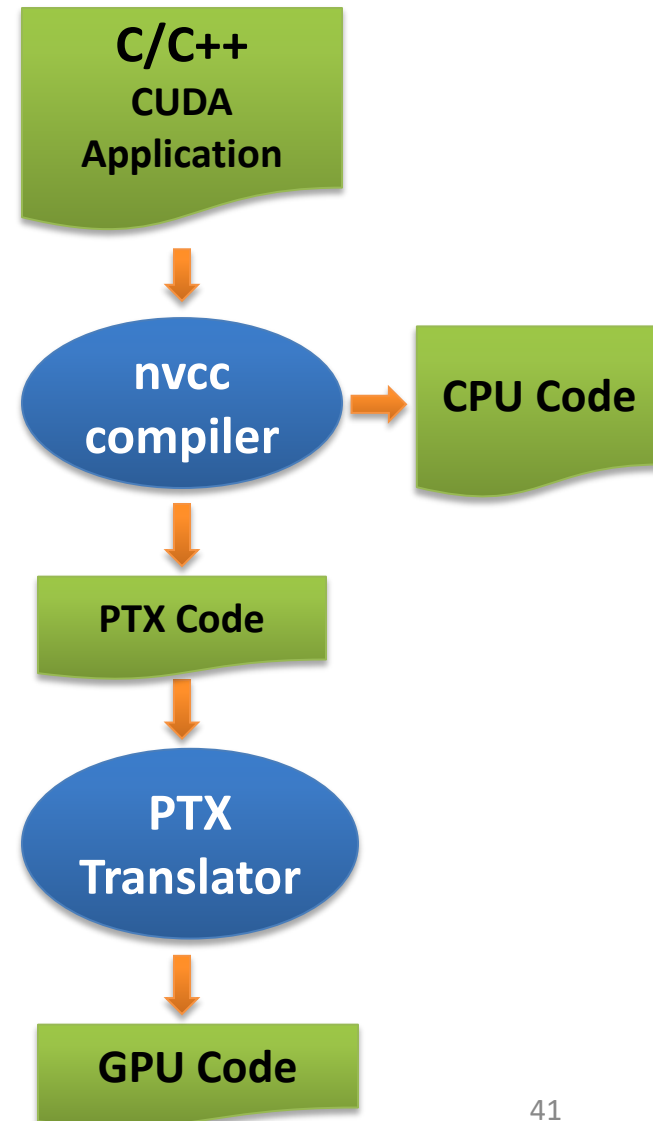
    // Free memory
    delete [] x;
    delete [] y;

    return 0;
}
```

Version	Laptop (GeForce GT 750M)		Server (Tesla K80)	
	Time	Bandwidth	Time	Bandwidth
1 CUDA Thread	411ms	30.6 MB/s	463ms	27.2 MB/s
1 CUDA Block	3.2ms	3.9 GB/s	2.7ms	4.7 GB/s
Many CUDA Blocks	0.68ms	18.5 GB/s	0.094ms	134 GB/s

Compiling CUDA

- **The CUDA source file**
 - ***.cu** extension
 - contain host and device codes
- **The CUDA Compiler **nvcc****
 - generate CPU/PTX code
 - PTX (Parallel Thread Execution) is the device independent VM code
- **PTX code** is translated for special GPU architecture.



Outline

- **Introduction to Graphic Processing Unit (GPU)**
- **GPU basics**
 - Hardware
 - Execution Models
 - GPU memory hierarchy
- **CUDA basics**
 - What is CUDA ?
 - Function and Variable Qualifiers.
 - Examples of CUDA Programs

References

Books

CUDA by Example: An Introduction to General-Purpose GPU Programming by Sanders and Kandrot

The CUDA Handbook: A Comprehensive Guide to GPU Programming by Wilt

CUDA Application Design and Development by Farber

Nvidia CUDA Programming Guide

<https://docs.nvidia.com/cuda/cuda-c-programming-guide/>

NVIDIA TESLA P100

<http://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper-v1.2.pdf>

YouTube lectures

GPU Architecture and CUDA Overview

<https://www.youtube.com/watch?v=nRSxp5ZKwhQ>

CUDA optimization

https://www.youtube.com/watch?v=FcCTHJO8_zo

Exploring the GPU Architecture and why we need it

<https://blogs.vmware.com/vsphere/2019/03/exploring-the-gpu-architecture-and-why-we-need-it.html>

NVIDIA CUDA tutorial 1: Introduction

<https://www.youtube.com/watch?v=m0nhePeHwFs>