Introduction to High-Performance Machine Learning

Lecture 5 02/22/24

Dr. Kaoutar El Maghraoui

Dr. Parijat Dube

CUDA Basics

Performance Factors

Algorithms Performance

- Algorithm choice

Hyperparameters Performance

- Hyperparameters choice

Implementation Performance

- Implementation of the algorithms on top of a framework

Framework Performance

- Python, Python, Cpython

Libraries Performance

- **CUDA,** cuDNN, Communication Libraries (MPI, GLOO)

Hardware Performance

- CPU, DRAM, **GPU, HBM,** Tensor Units, Disk/Filesystem, Network

Performance Factors

Algorithms Performance

- Algorithm choice

Hyperparameters Performance

- Hyperparameters choice

Implementation Performance

- Implementation of the algorithms on top of a framework

Framework Performance

- Python, Python, Cpython

Libraries Performance

- **CUDA,** cuDNN, Communication Libraries (MPI, GLOO)

Hardware Performance

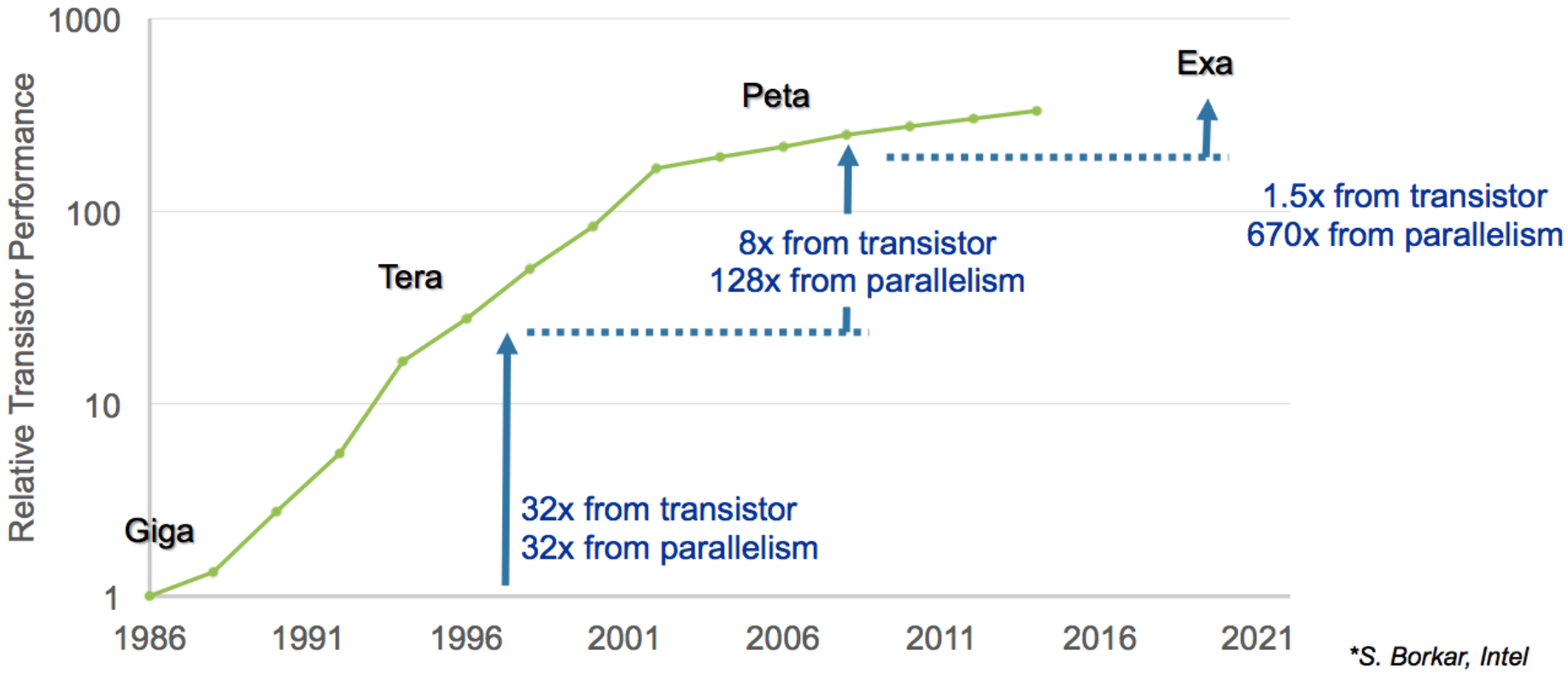
- CPU, DRAM, **GPU, HBM,** Tensor Units, Disk/Filesystem, Network

Lesson Outline

- Heterogenous architectures motivations
- Hierarchy of Computations:
  - Threads
  - Blocks
  - Grids
- Corresponding Memory Spaces
  - Local
  - Shared
  - Global
- Synchronization Primitives
  - Implicit Barriers
  - Thread Synchronization
- NVIDIA GPUs and CUDA:
  - Compute capability
- CUDA Hardware
- CUDA Compilation and Runtime:
  - CUDA Runtime, CUDA Driver, AoT and JIT compilation
- CUDA Programming Model:
  - Grid, Block, Thread
  - UVM
- CUDA Warp Scheduling
- Context and Stream
- CUDA Profiling and Debugging

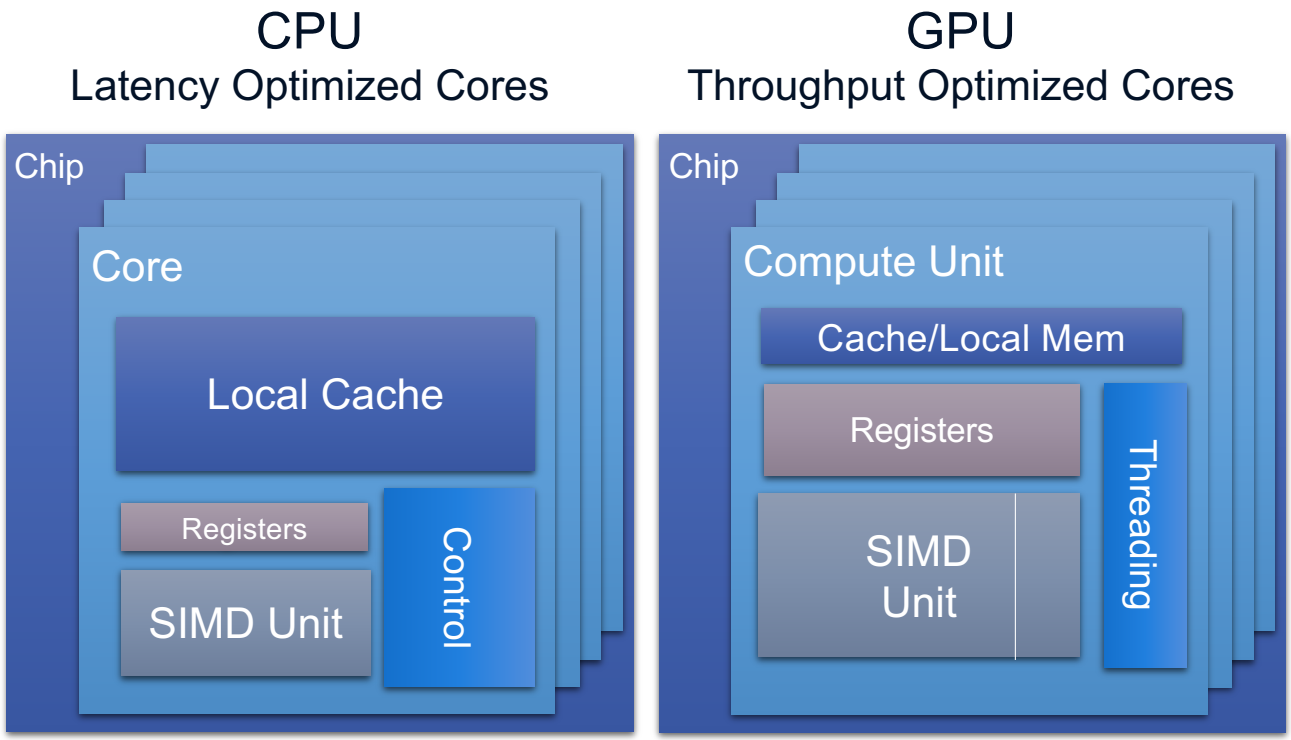
Heterogenous computing motivations

Heterogeneous Computing Motivation



Difference between CPU and GPU

- Latency Optimized:
  - Optimized to compute operation in a minimum time
- Throughput Optimized:
  - Optimized to compute many operations on the same time



latency oriented

From: Nvidia – U Illinois

No Control unit

throughput oriented

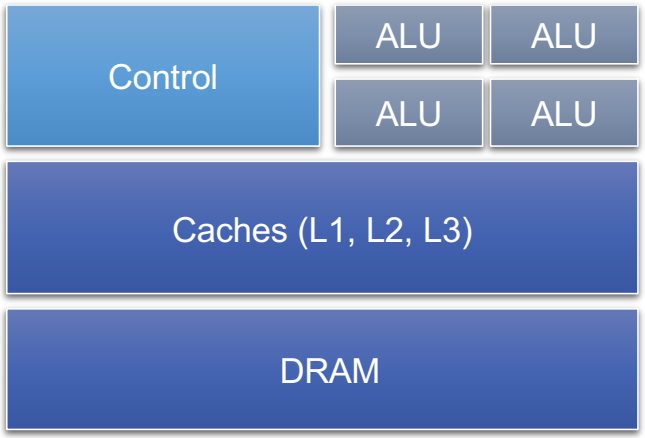
CPU: Latency Optimized

ALU is responsible for performing arithmetic and logical operations on binary data.

- Powerful **Arithmetic Logic Unit**
  - Optimized for latency
- Sophisticated **Control logic**
  - Branch prediction for reduced branch latency
  - Data forwarding for reduced data latency
- Large **L1, L2, L3 Cache** Hierarchy
  - Convert long latency memory accesses to short latency cache accesses thus reducing the latency

- https://cacm.acm.org/magazines/2010/11/100622-understanding-throughput-oriented-architectures/fulltext

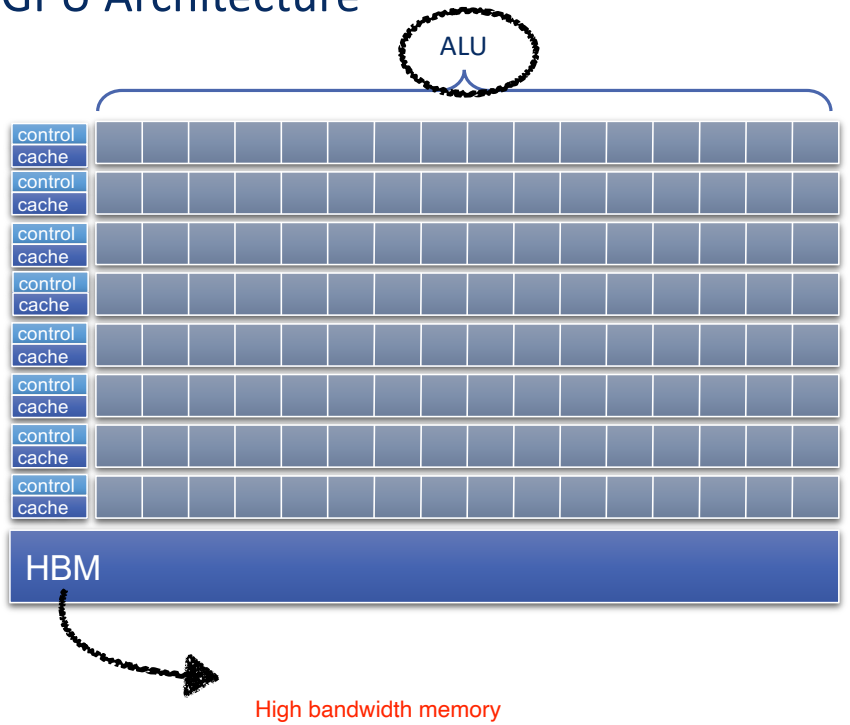
CPU Architecture



GPU: Throughput Optimized

- Many small **ALU**s
  - Many
  - Long latency
  - Parallelism
- Small **Caches (shared memory)**
  - To boost memory throughput
- Simple **Control**
  - No branch prediction
  - No data forwarding
  - Require massive number of threads to tolerate latencies
  - Threading logic and state

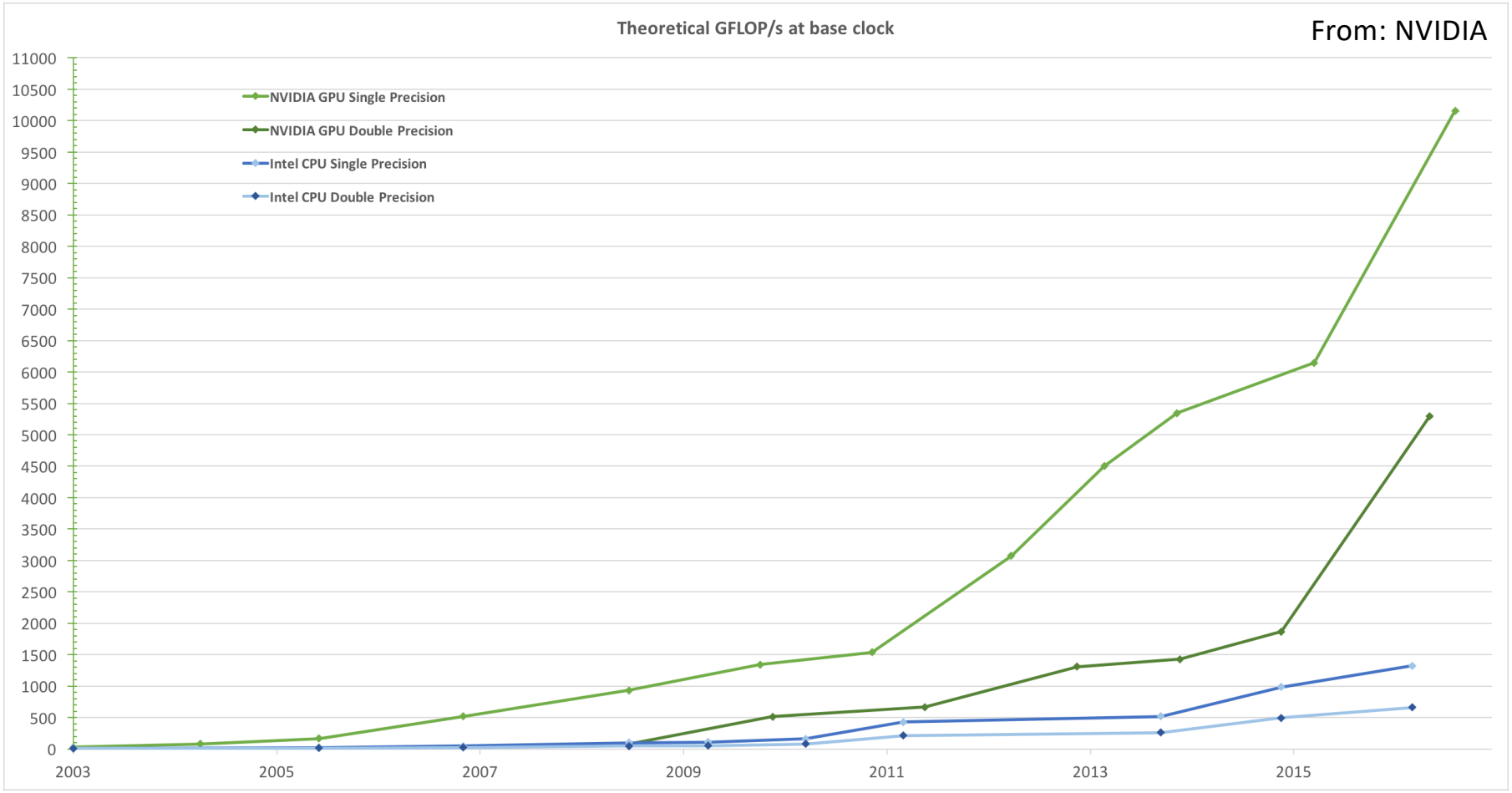
GPU Architecture



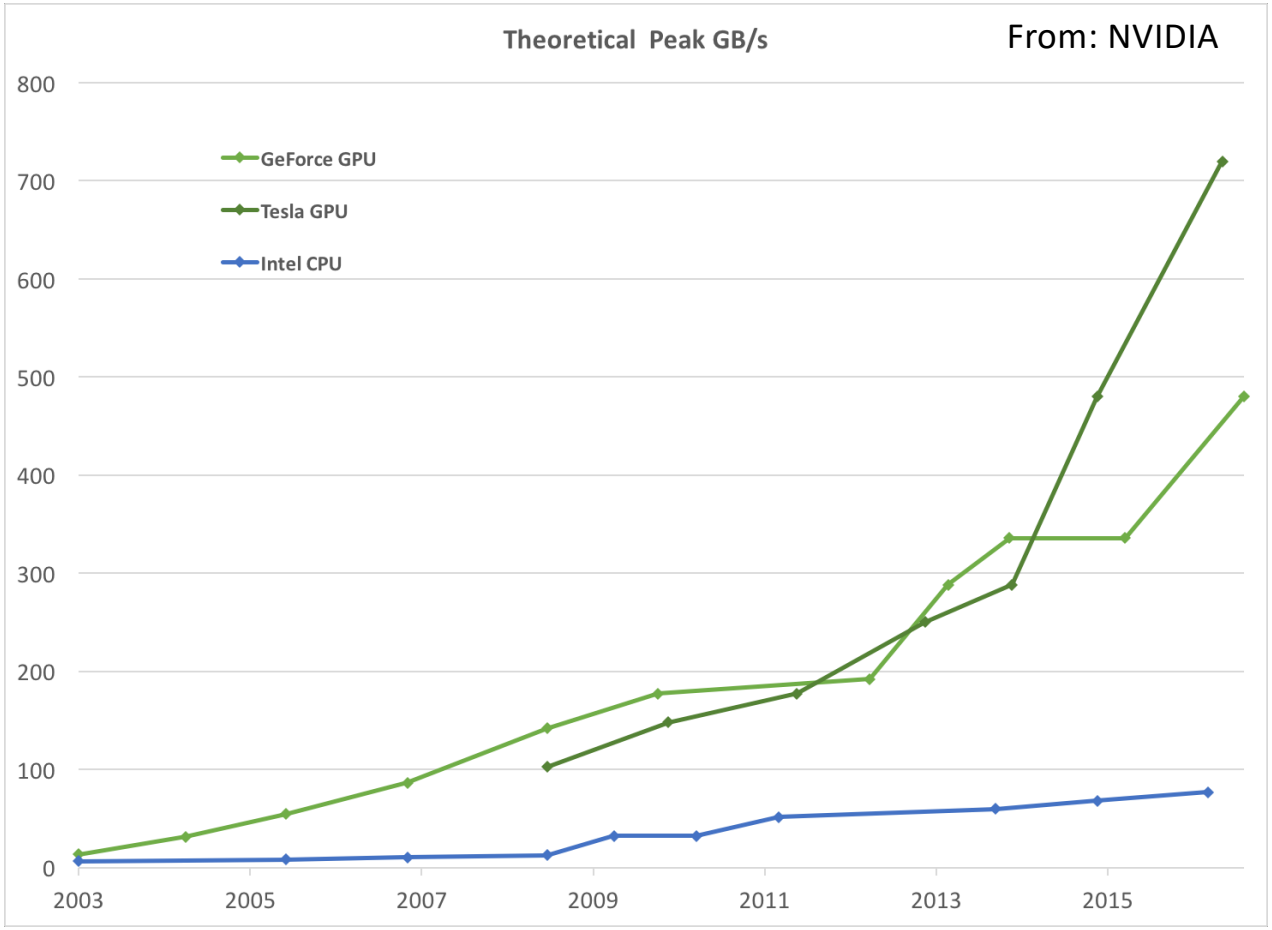
CPU/GPU Performance Comparison

- Sequential Code:
  - CPU about 10x faster than GPU ex - if else blocks
- Parallel Code:
  - GPUs can be 10X+ faster than CPUs for parallel code
- Latencies:
  - CPU: latency of operations is in **nsec**
  - GPU: launching a kernel can take 10s **micro-sec** or more high latency

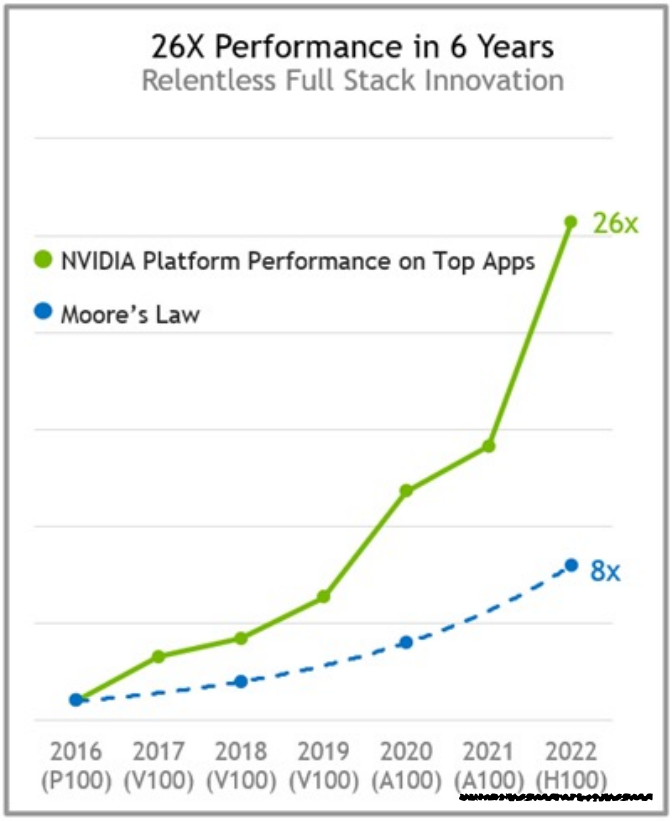
GPU Performance Advantage 1: FLOPS

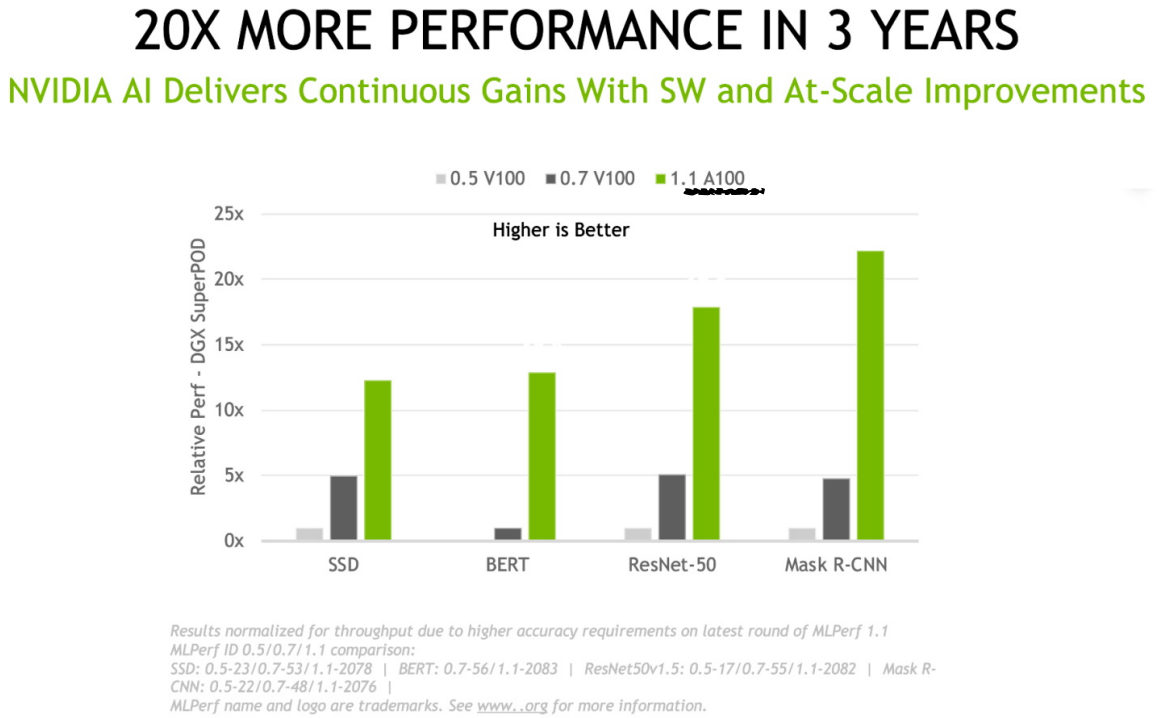


GPU Performance Advantage 2: Memory BW



NVIDIA GPU Performance Improvements Over the years





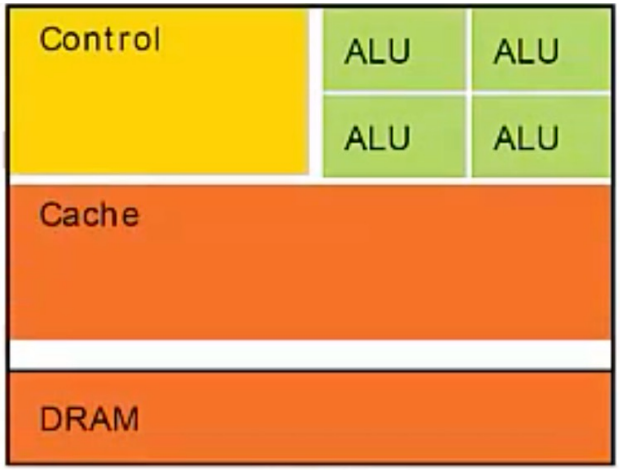
Source: https://developer.nvidia.com/blog/fueling-high-performance-computing-with-full-stack-innovation/

CPU vs. GPU Summary

CPU	GPU
A smaller number of larger cores (up to 24)	A larger number (thousands) of smaller cores
Low latency	High throughput
Optimized for serial processing	Optimized for parallel processing
Designed for running complex programs	Designed for simple and repetitive calculations
Performs fewer instructions per clock	Performs more instructions per clock
Automatic cache management	Allows for manual memory management
Cost-efficient for smaller workloads	Cost-efficient for bigger workloads

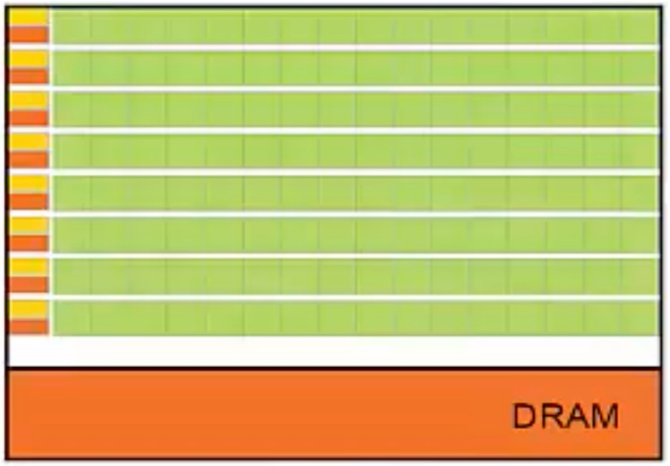
CUDA Programming Model

Difference between CPU's and GPU's



CPU

- CPUs have few strong cores
- CPU- Designed to minimize latency
  - Majority of the silicon area is dedicated to:
    - Advanced Control Logic
    - Large Cache

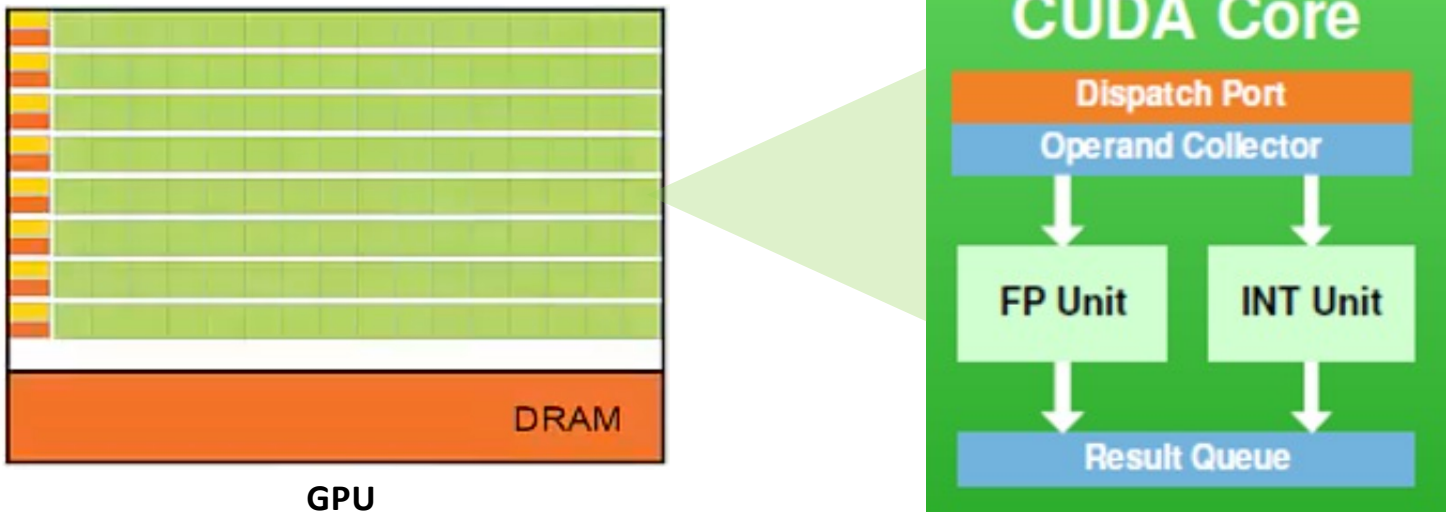


GPU

- GPUs have thousands of weaker cores
- GPU- Designed to maximize throughput
  - Majority of the silicon area is dedicated to:
    - Massive number of arithmetic units – CUDA cores

What is a CUDA Core

CUDA cores are the basic processing units responsible for executing parallel threads on the GPU.
They are designed to perform arithmetic and logic operations in parallel, making them suitable for parallelizable tasks.

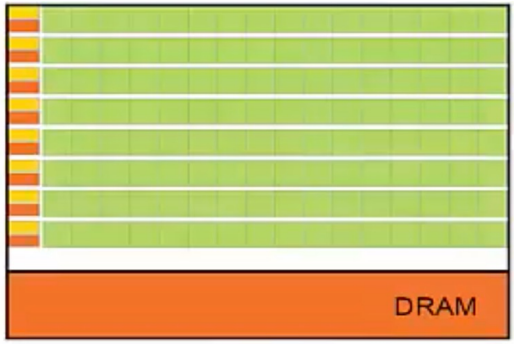


GPU

A CUDA core, also known as a streaming multiprocessor (SM) core, is a fundamental processing unit within the architecture of NVIDIA GPUs (Graphics Processing Units) that support CUDA (Compute Unified Device Architecture). CUDA cores are designed to handle parallel processing tasks and are a key component in the GPU's ability to accelerate a wide range of computations.

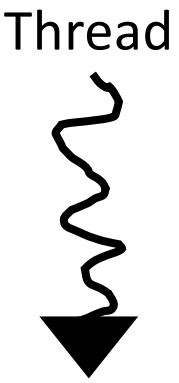
Key Concepts

To properly utilize the GPU, the parts of the program that can be parallelize must be decomposed into a large number of threads that can run concurrently in CUDA.



GPU

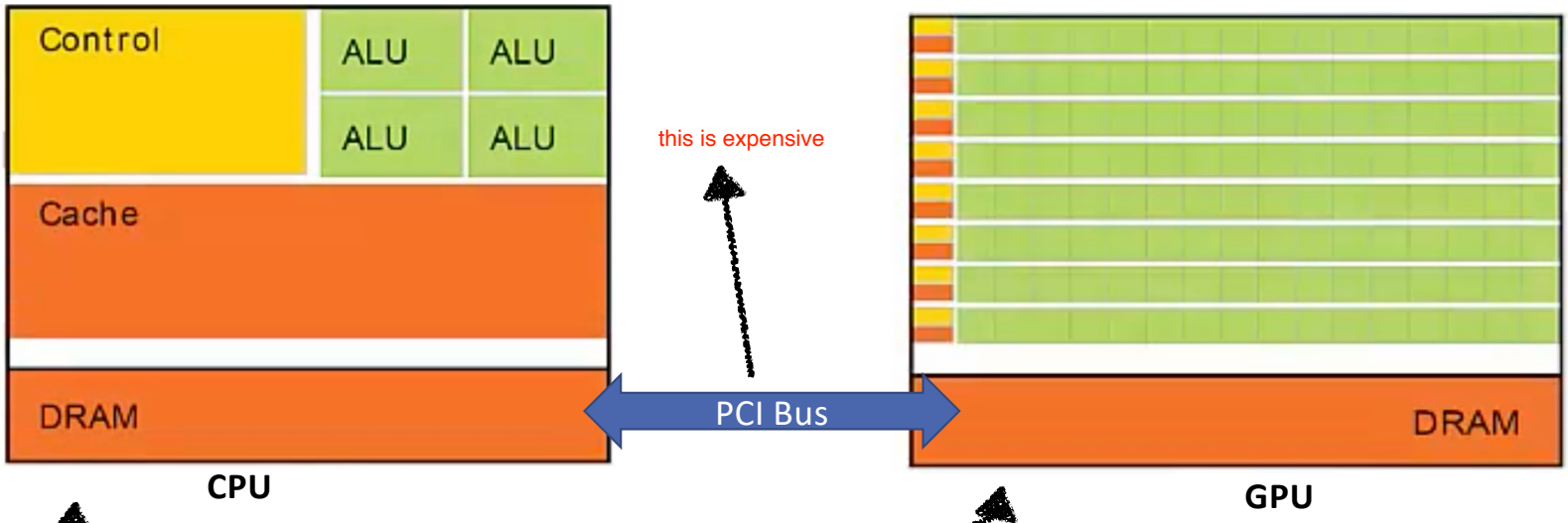
Kernels	Threads
• Functions that run on the GPU • Kernel are lunched on the CPU	• Kernels execute as a set of parallel threads • Each thread is mapped to a singled CUDA core on the GPU



Kernels are functions that execute on the GPU and are invoked with a specific syntax in CUDA code.

Threads are organized into blocks, and blocks are organized into a grid.

Key Concepts

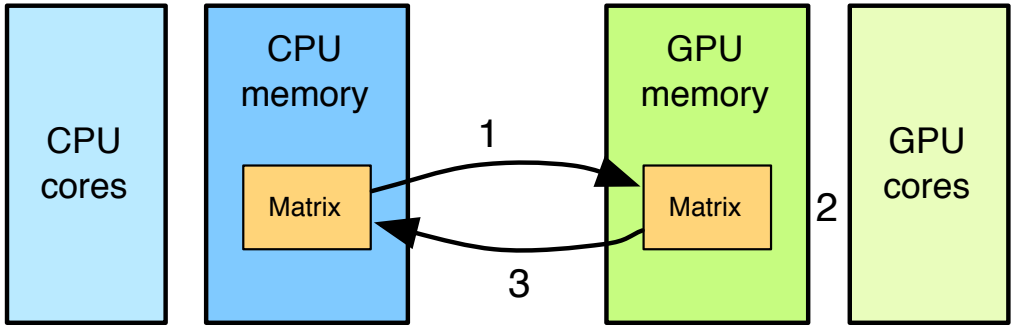


Host	Device	Heterogenous	CUDA
• CPU and its on chip memory	• GPU + dedicated DRAM	• Host + Device • Take the best of both worlds	• C with extensions • Allows for heterogenous programming

hetrogenous computing

GPU Computing

- Computation is offloaded to the GPU
- Three steps
  - CPU-GPU data transfer (1)
  - GPU kernel execution (2)
  - GPU-CPU data transfer (3)

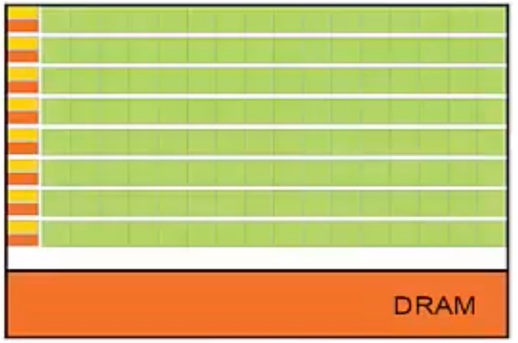


Kernels are functions that execute on the GPU

CUDA Threads

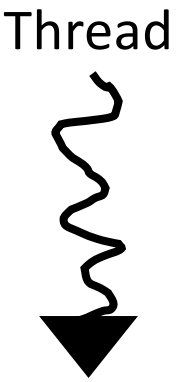
- Kernels execute as a **set of parallel threads**
- CUDA is designed to execute **thousands of threads**
- **CUDA Threads execute in a SIMD** (Single Instruction Multiple Data) fashion
  - NVIDIA calls this SIMT (Single Instruction Multiple Thread)

all of them exectuing same instruction but on diff data



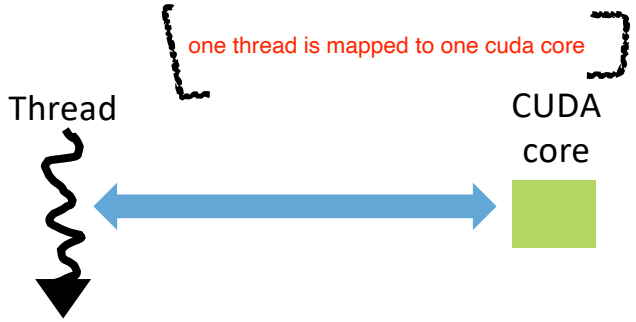
GPU

- **Threads are similar to data-parallel tasks**
  - Each thread performs the same operation on a subset of data
  - Threads execute independently
- **Threads do not execute at the same rate**
  - Different paths are taken in if/else statements
  - Different number of iterations in a loop, etc.



this is how GPU achieves high throughput

Threads Hierarchy

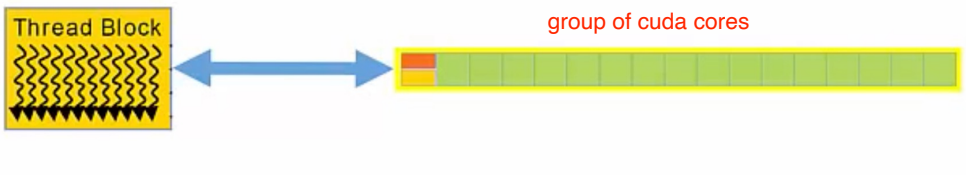


Threads

- Kernels execute as a set of Threads

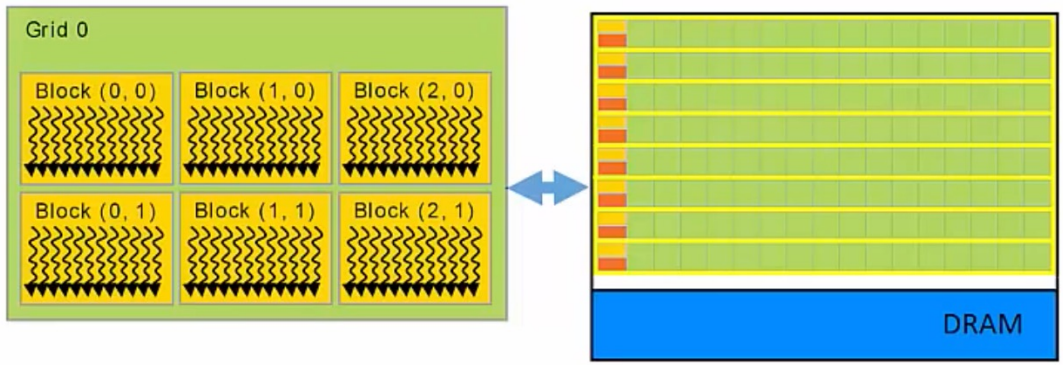
Blocks

- Threads are groups into blocks



Grid

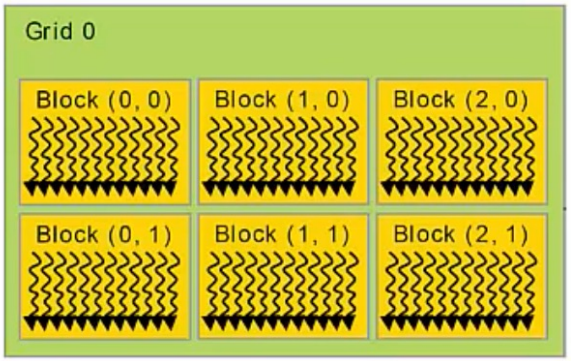
- Blocks are grouped into Grids
- **Each Kernel launch creates a single Grid**



Thread $\in$ Blocks $\in$ Grids

Dimensions of Gids and Blocks

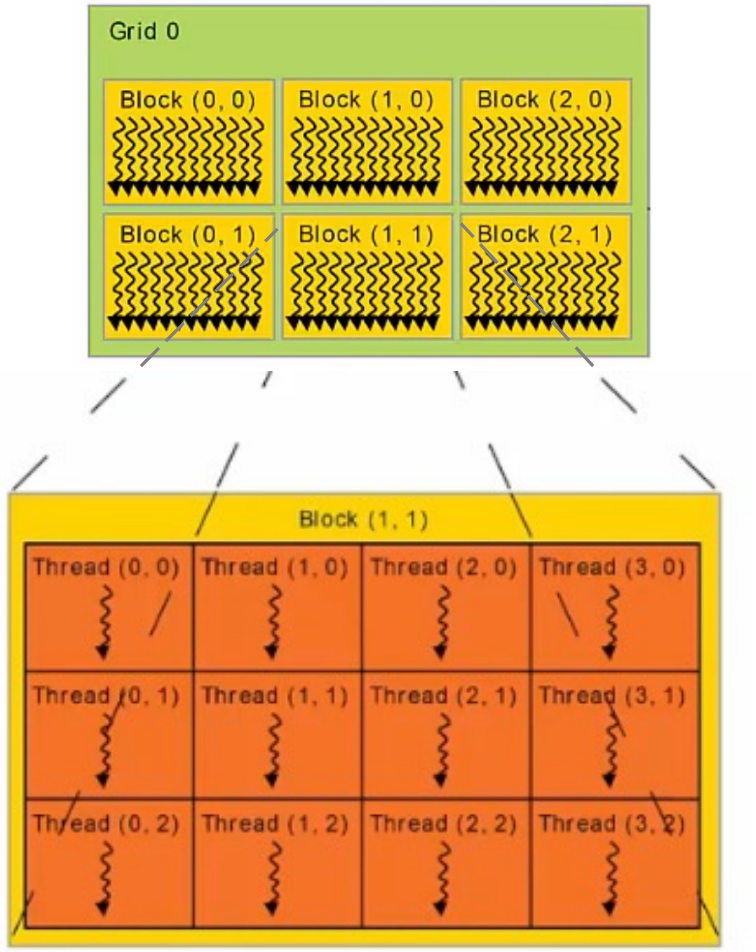
Grid dimension
- Block structure of each Grid
- 1D, 2D, or 3D



Grid dimension: 3 x 2
=> 6 block

Dimensions of Gids and Blocks

Grid dimension
- Block structure of each Grid
- 1D, 2D, or 3D



Grid dimension: 3 x 2
=> 6 block

Grid dimension
- Thread structure of each Block
- 1D, 2D, or 3D

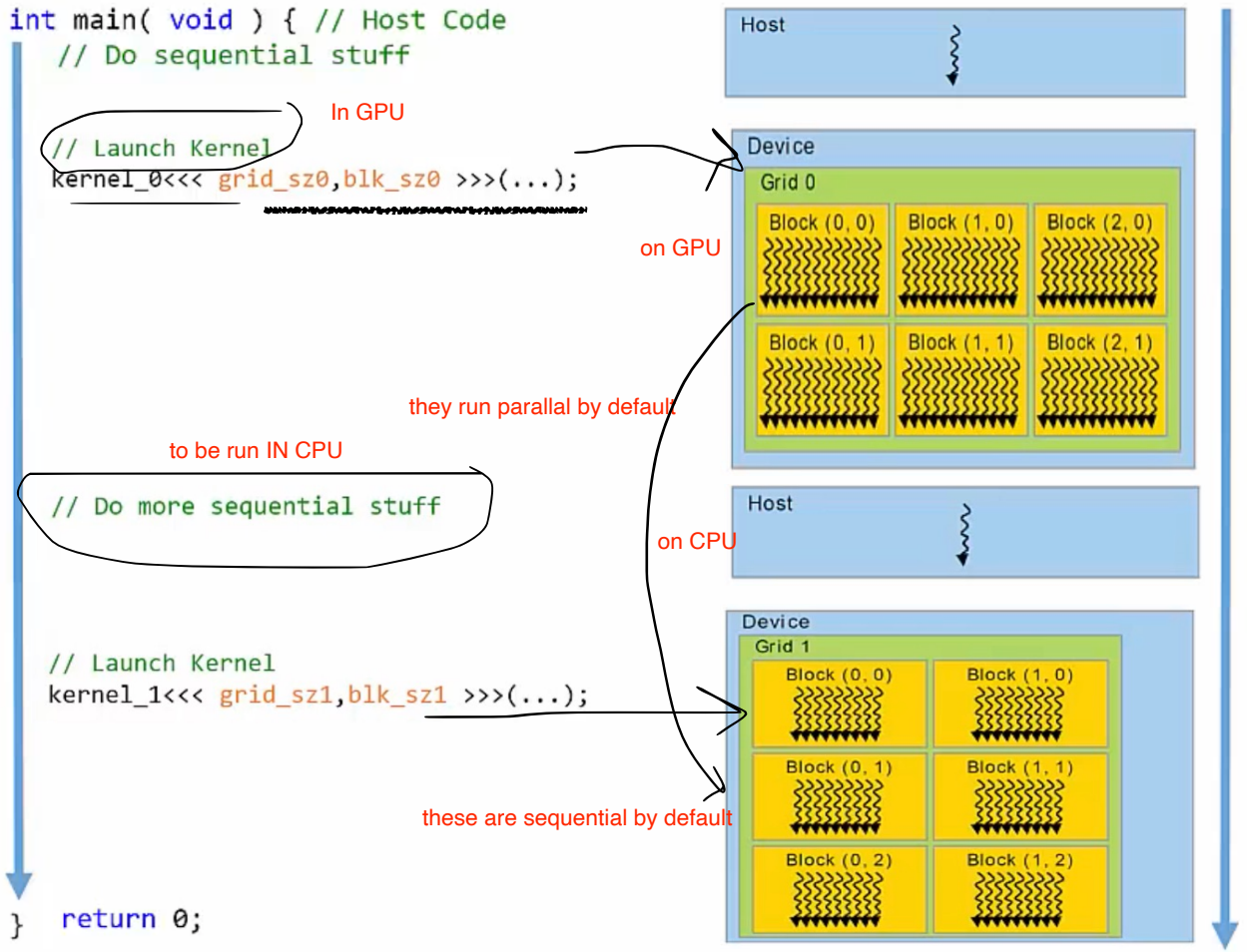
Block Dimension: 4x3
⇒ 4x3 = 12 Threads/Block
⇒ 6 Blocks x 12 Threads/Block
⇒ 72 Total Threads in Grid

i.e. 72 parallal operations

Program Flow

- Host Code
  - Serial
  - Runs on CPU
  - Launches Kernels

- Device Code
  - Parallel
  - Runs on GPU



Host code does not wait for the GPU kernel to finish unless explicitly told so

Kernel Launch

```
// Block and Grid dimensions
dim3 grid_size(x, y, z)
dim3 block_size(x, y, z)
```

Configuration Parameters

<<<grid_size,block_size>>>

- dim3 is a CUDA data structure
- Default values are (1,1,1)

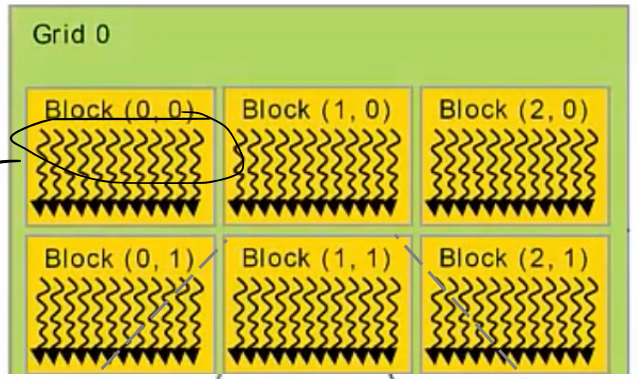
```
// Launch Kernel
KernelName<<< grid_size, block_size >>>( . . . );
```

Kernel Launch Configuration Example

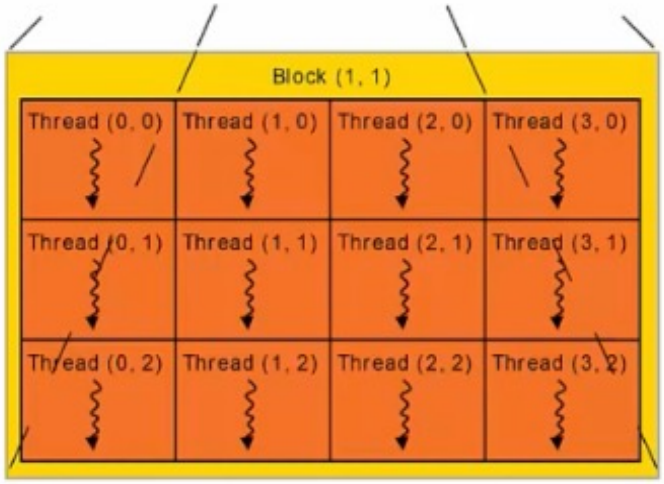
6 blocks

```
// Block and Grid dimensions
// a.k.a. Configuration Parameters
dim3 grid_size(3,2);
dim3 block_size(4,3);
```

12

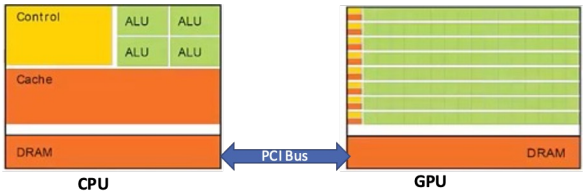



```
// Launch Kernel
kernelName<<< grid_size, block_size >>>( ... );
```



Closer look at Program Flow

- Host Code
  - Do sequential staff
  - Prepare for Kernel Launch
- Allocate Memory on Device
- Copy Data Host->Device
  - Copy data from CPU to GPU
- Launch Kernel
  - Execute Threads on the GPU in Parallel
- Copy Data Device->Host

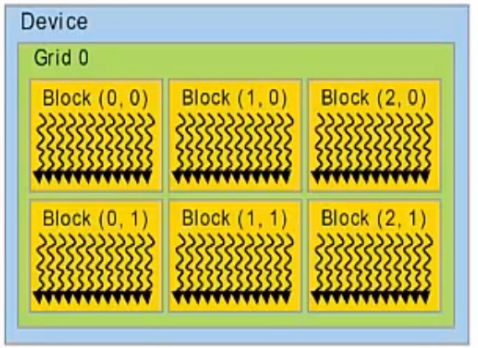



```
int main( void ) { // Host Code
  // Do sequential stuff

  // Allocate memory on the device
  cudaMalloc(...);

  // Copy data from Host to Device
  cudaMemcpy(...);

  // Launch Kernel
  kernel_0<<< grid_sz0,blk_sz0 >>>(...);

  // Copy data from Device to Host
  cudaMemcpy(...);

  // Do sequential stuff
  // ...

  return 0;
}
```

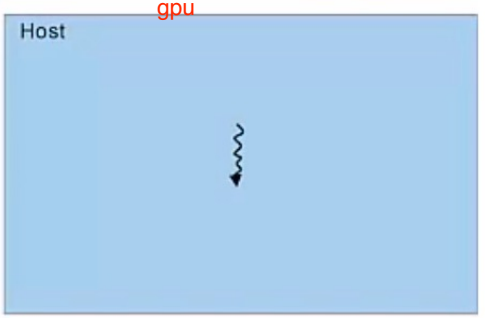
cude device

from CPU to GPU

cpu

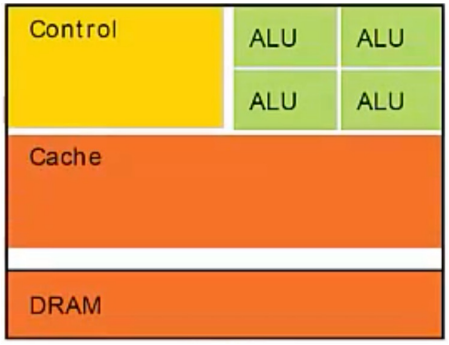


gpu

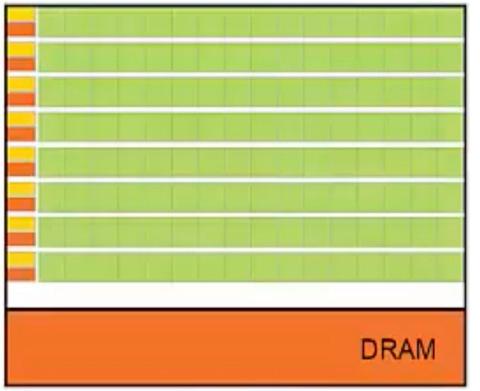


Allocation Device Memory

- Similar to allocation of memory in C
  - malloc(. . .);
- De-Allocate memory in C
  - free(. . .);
- Allocate memory in CUDA
  - cudaMalloc(LOCATION, SIZE);
  - 1st Argument:
    - Memory location on Device to allocate memory
    - An address in the GPU's memory
  - 2nd Argument:
    - Number of bytes to allocate
- De-Allocate memory in CUDA
  - cudaFree(. . .);



CPU

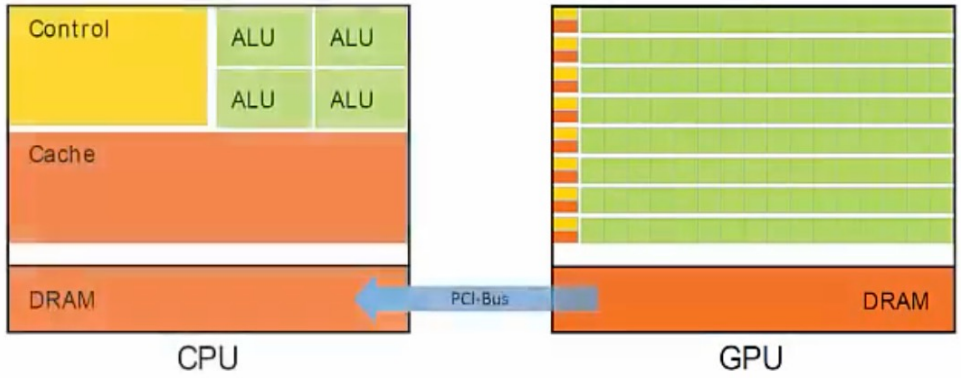


GPU

Copy Data Host <-> Device

```
cudaMemcpy( dst, src, numBytes, direction );
```

- numBytes = N*sizeof(type)
- direction
  - cudaMemcpyHostToDevice
  - cudaMemcpyDeviceToHost



Example

```
int main( void ) {

    // Declare variables
    int *h_c, *d_c;

    // Allocate memory on the device
    cudaMalloc( (void**)&d_c, sizeof(int) );

    // Allocate memory on the device
    cudaMemcpy(d_c, h_c, sizeof(int), cudaMemcpyHostToDevice );

    // Configuration Parameters
    dim3 grid_size(1);    dim3 block_size(1);

    // Launch the Kernel
    kernel<<<grid_size, block_size>>>(...);

    // Copy data back to host
    cudaMemcpy( h_c, d_c, sizeof(int), cudaMemcpyDeviceToHost );

    // De-allocate memory
    cudaFree( d_c );  free( h_c );
    return 0;
}
```

// Declare variable good practice

Convention

- Variables that live on Host
  - h_
- Variables that live on Device
  - d_

Example

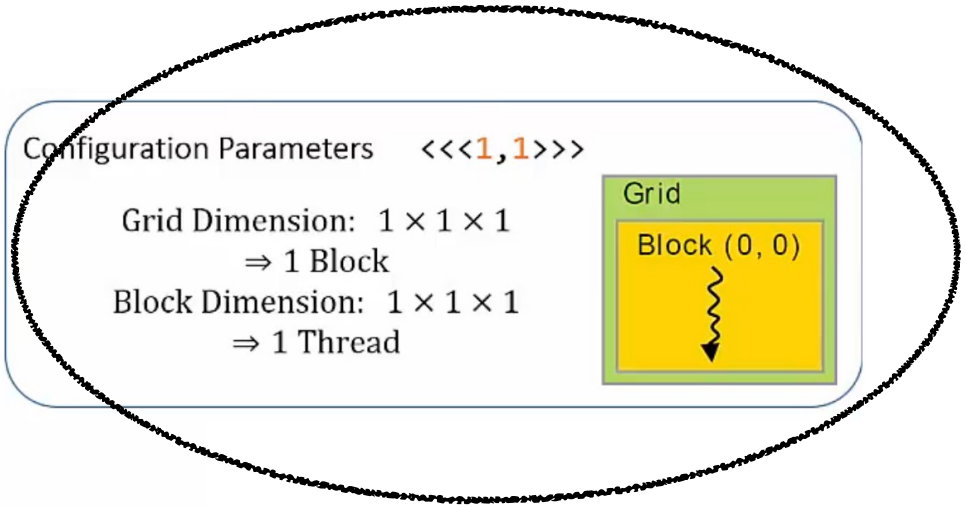
```
int main( void ) {

    // Declare variables
    int *h_c, *d_c;

    // Allocate memory on the device
    cudaMalloc( (void**)&d_c, sizeof(int) );

    // Allocate memory on the device
    cudaMemcpy(d_c, h_c, sizeof(int), cudaMemcpyHostToDevice );

    // Configuration Parameters
    dim3 grid_size(1);    dim3 block_size(1);

    // Launch the Kernel
    kernel<<<grid_size, block_size>>>(...);

    // Copy data back to host
    cudaMemcpy( h_c, d_c, sizeof(int), cudaMemcpyDeviceToHost );

    // De-allocate memory
    cudaFree( d_c );  free( h_c );
    return 0;
}
```

// Allocate memory on device.

cudaMalloc(Location of Memory on Device, Amount of Memory)

Example

```
int main( void ) {

    // Declare variables
    int *h_c, *d_c;

    // Allocate memory on the device
    cudaMalloc( (void**)&d_c, sizeof(int) );

    // Allocate memory on the device
    cudaMemcpy(d_c, h_c, sizeof(int), cudaMemcpyHostToDevice );

    // Configuration Parameters
    dim3 grid_size(1);    dim3 block_size(1);

    // Launch the Kernel
    kernel<<<grid_size, block_size>>>(...);

    // Copy data back to host
    cudaMemcpy( h_c, d_c, sizeof(int), cudaMemcpyDeviceToHost );

    // De-allocate memory
    cudaFree( d_c );  free( h_c );
    return 0;
}
```



Kernel Definition

```
__global__ void kernel( int *d_out, int *d_in )
{
    // Perform this operation for every thread
    d_out[0] = d_in[0];
}
```

`__global__` is a "Declaration Specifier" that alerts the compiler that a function should be compiled to run on device.

Kernels must return type `void`
⇒ Variables operated on in the kernel need to be passed by reference

Any results that the kernel computes are stored in the device memory.

To manipulate data:

C uses "pass-by-value"

- Functions receive copies of their arguments
- The actual parameters to the function will not be modified.
- We simulate "pass-by-reference".
  - Pass the address of the variable as parameter to the kernel.

How to distinguish Threads from one another?

Thread Index

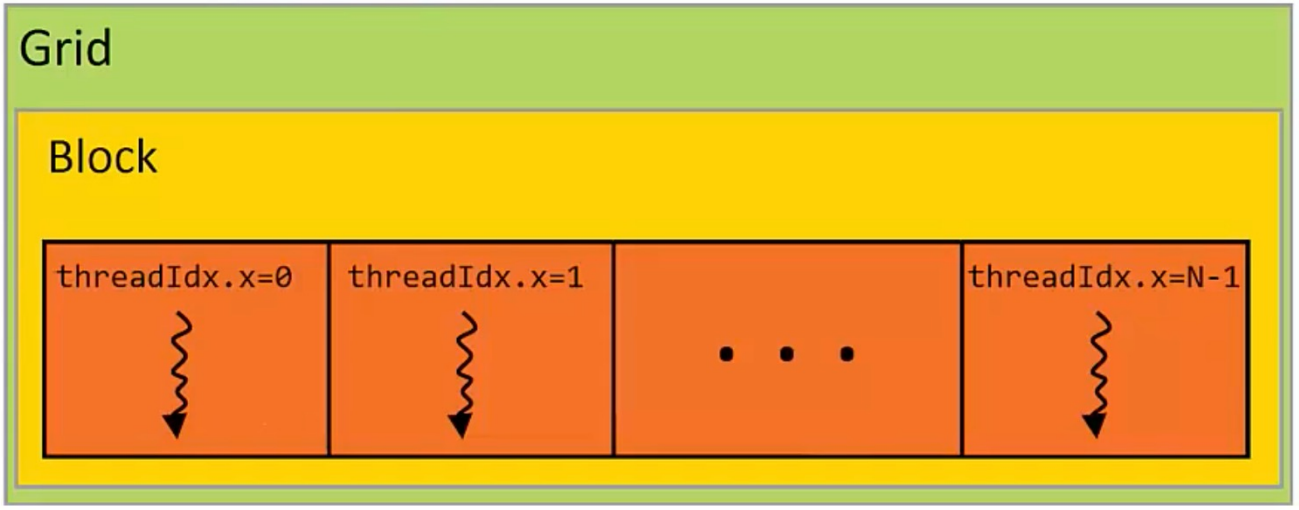
- Each Thread has its own thread index
  - Accessible within a Kernel through the built in threadIdx variable
- Thread Blocks can have as many as 3-dimensions, therefore there is a corresponding index for each dimension:

```
threadIdx.x
threadIdx.y
threadIdx.z
```

```
// Configuration Parameters
dim3 grid_size(1);
dim3 block_size(N);
```

Grid Dimension: $1 \times 1 \times 1$
$\Rightarrow$ 1 Block

Block Dimension: $N \times 1 \times 1$
$\Rightarrow N$ Threads



Parallelize for loop example

CPU Program

```
// Function Definition
void increment_cpu( int *a, int N)
{
    for (int i=0; i<N; i++)
        a[i] = a[i] + 1;
}

int main( void )
{
    int a[N] =  // ...

    // Call Function
    increment_cpu( a , N );
    // ...
    return 0;
}
```

CUDA Program

```
// Kernel Definition
__global__ void increment_gpu(int *a, int N)
{
    int i = threadIdx.x;
    if (i < N)
        a[i] = a[i] + 1;
}

int main( void )
{
    int h_a[N] =  // ...

    // Allocate arrays in Device memory
    int* d_a; cudaMalloc( (void**)&d_a, N * sizeof(int) );

    // Copy memory from Host to Device
    cudaMemcpy(d_a, h_a, N*sizeof(int), cudaMemcpyHostToDevice );

    // Block and Grid dimensions
    dim3 grid_size(1);   dim3 block_size(N);

    // Launch Kernel
    increment_gpu<<<grid_size, block_size>>>( d_a , N );
    // ...
    return 0;
}
```

to be executed on a single thread

N threads, one for one index of array

Parallelize for loop example

CPU Program

```
// Function Definition
void increment_cpu( int *a, int N)
{
    for (int i=0; i<N; i++)
        a[i] = a[i] + 1;
}

int main( void )
{
    int a[N] =  // ...

    // Call Function
    increment_cpu( a , N );
    // ...
    return 0;
}
```

CUDA Program

```
// Kernel Definition
__global__ void increment_gpu(int *a, int N)
{
    int i = threadIdx.x;
    if (i < N)
        a[i] = a[i] + 1;
}

int main( void )
{
    int h_a[N] =  // ...

    // Allocate arrays in Device memory
    int* d_a; cudaMalloc( (void**)&d_a, N * sizeof(int) );

    // Copy memory from Host to Device
    cudaMemcpy(d_a, h_a, N*sizeof(int), cudaMemcpyHostToDevice );

    // Block and Grid dimensions
    dim3 grid_size(1);   dim3 block_size(N);

    // Launch Kernel
    increment_gpu<<<grid_size, block_size>>>( d_a , N );
    // ...
    return 0;
}
```

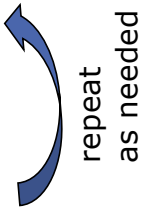
If (i < N)
Ensures that the kernel does not execute more Threads than the length of the array.

Traditional Program Structure in CUDA

- Function prototypes

```
float serialFunction(…);
__global__ void kernel(…);
```

- `main()`
  - 1) Allocate memory space on the device – `cudaMalloc(&d_in, bytes);`
  - 2) Transfer data from host to device – `cudaMemCpy(d_in, h_in, …);`
  - 3) Execution configuration setup: #blocks and #threads
  - 4) Kernel call – `kernel<<<execution configuration>>>(args…);`
  - 5) Transfer results from device to host – `cudaMemCpy(h_out, d_out, …);`



- Kernel – `__global__ void kernel(type args,…)`
  - Automatic variables transparently assigned to registers
  - Shared memory: `__shared__`
  - Intra-block synchronization: `__syncthreads();`

Slide credit: Hwu & Kirk

CUDA Programming Language

- Memory allocation

```
cudaMalloc((void**)&d_in, #bytes);
```

- Memory copy

```
cudaMemcpy(d_in, h_in, #bytes, cudaMemcpyHostToDevice);
```

- Kernel launch

```
kernel<<< #blocks, #threads >>>(args);
```

- Memory deallocation

```
cudaFree(d_in);
```

- Explicit synchronization

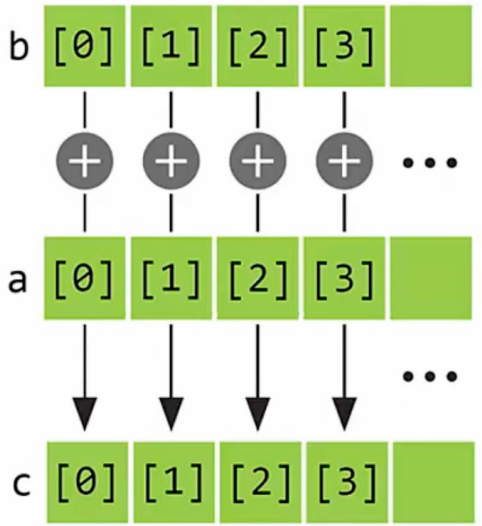
```
cudaDeviceSynchronize();
```

Vector Addition – A Very Parallel Problem

$$\boldsymbol{a}, \boldsymbol{b} \in \mathbb{R}^N$$

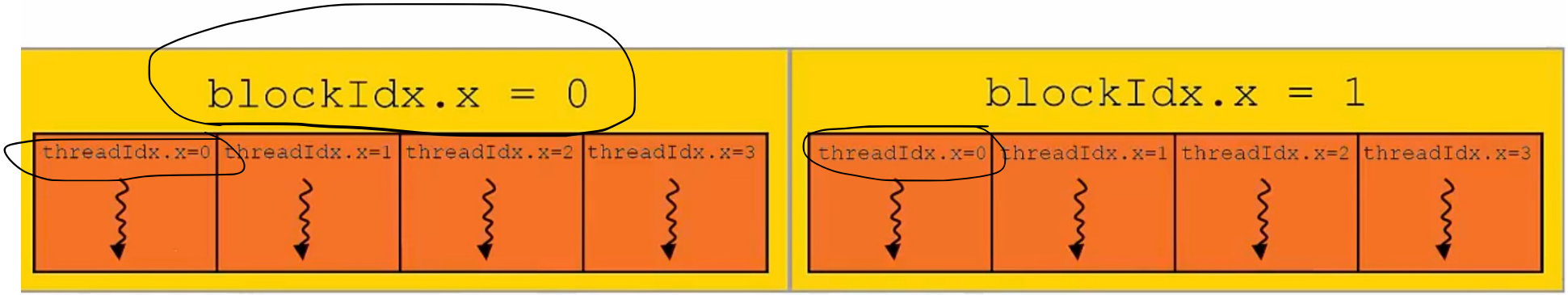
$$\boldsymbol{a} + \boldsymbol{b} = \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \\ \vdots \\ a_{N-1} \end{bmatrix} + \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \\ \vdots \\ b_{N-1} \end{bmatrix}$$

$$= \begin{bmatrix} a_0 + b_0 \\ a_1 + b_1 \\ a_2 + b_2 \\ a_3 + b_3 \\ \vdots \\ a_{N-1} + b_{N-1} \end{bmatrix} = \begin{bmatrix} c_0 \\ c_1 \\ c_2 \\ c_3 \\ \vdots \\ c_{N-1} \end{bmatrix} = \boldsymbol{c} =$$

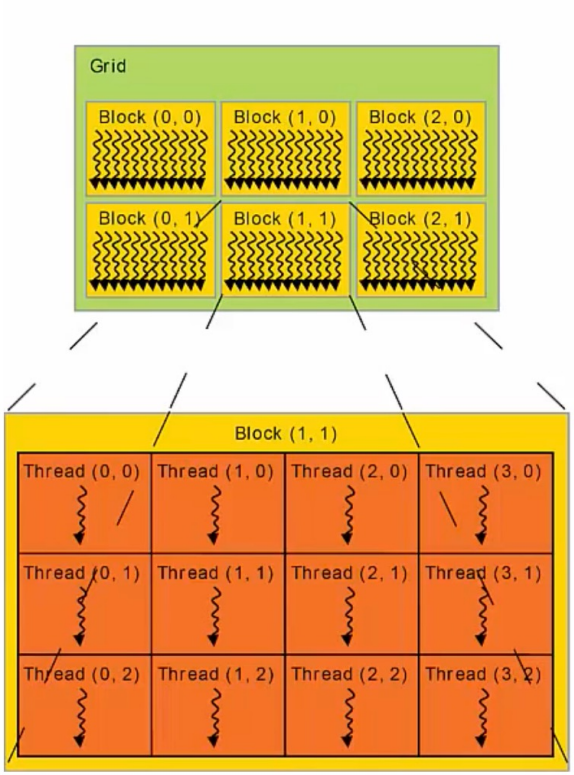


Thread Index

- Each Thread has its own unique thread index.
  - Accessible within a Kernel through the built in `threadIdx` variable.



Built-in Variables



Dimension of a Grid

```
dim3 gridDim;
int gridDim.x;
int gridDim.y;
int gridDim.z;
```

Index of a Block

```
dim3 blockIdx;
int blockIdx.x;
int blockIdx.y;
int blockIdx.z;
```

Dimension of a Block

```
dim3 blockDim;
int blockDim.x;
int blockDim.y;
int blockDim.z;
```

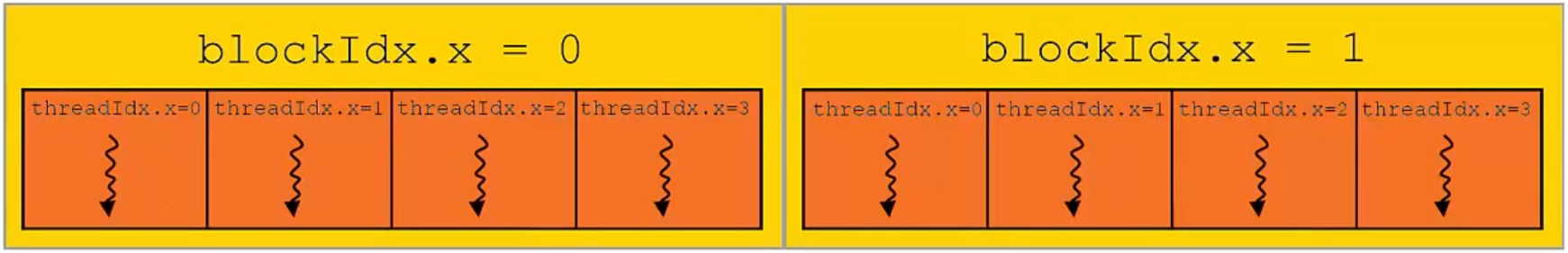
Index of a Thread

```
dim3 threadIdx;
int threadIdx.x;
int threadIdx.y;
int threadIdx.z;
```

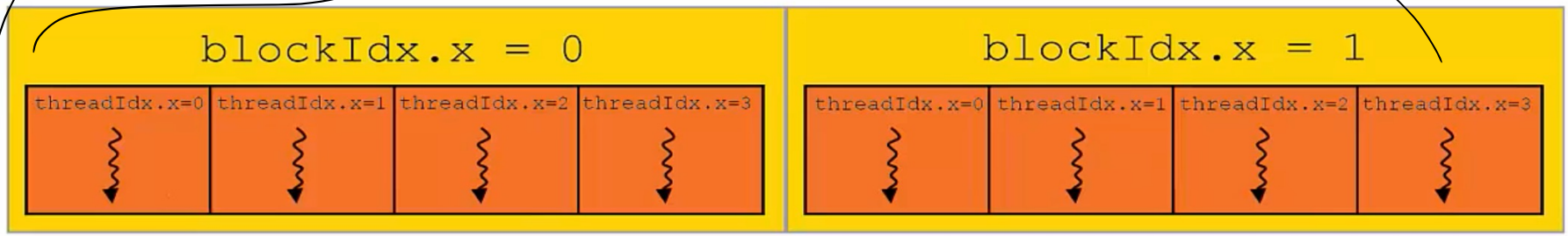
Indexing Within Grid

- `threadIdx` is only unique within its own Thread Block
- To determine the unique Grid index of a Thread:

```
i = threadIdx.x + blockIdx.x * blockDim.x;
```



Indexing Within Grid




```
i = threadIdx.x + blockIdx.x * blockDim.x;
```

i	threadIdx.x	blockIdx.x * blockDim.x
0	0	0 * 4 = 0
1	1	0 * 4 = 0
2	2	0 * 4 = 0
3	3	0 * 4 = 0
4	0	1 * 4 = 4
5	1	1 * 4 = 4
6	2	1 * 4 = 4
7	3	1 * 4 = 4

unique IDs w.r.t. grid

3blocks, 4 threads per block

Examples

```
// Launch Kernel
kernel<<< 3, 4 >>>(a);
```

grid size

block size

4 threads per block.
Block is just 1-D in this case

```
__global__ void kernel( int* a ) {
    int i = threadIdx.x + blockIdx.x * blockDim.x;
    a[i] = blockDim.x;
}
```

```
__global__ void kernel( int* a ) {
    int i = threadIdx.x + blockIdx.x * blockDim.x;
    a[i] = threadIdx.x;
}
```

```
__global__ void kernel( int* a ) {
    int i = threadIdx.x + blockIdx.x * blockDim.x;
    a[i] = blockIdx.x;
}
```

```
__global__ void kernel( int* a ) {
      int i = threadIdx.x + blockIdx.x * blockDim.x;
      a[i] = i;
}
```

total 12 times. As, 12 threads are created (3x4)

Examples

```
// Launch Kernel
kernel<<< 3, 4 >>>(a);
```

```
__global__ void kernel( int* a ) {
    int i = threadIdx.x + blockIdx.x * blockDim.x;
    a[i] = blockDim.x;
}
```

a: 4 4 4 4 4 4 4 4 4 4 4 4

```
__global__ void kernel( int* a ) {
    int i = threadIdx.x + blockIdx.x * blockDim.x;
    a[i] = threadIdx.x;
}
```

a: 0 1 2 3 0 1 2 3 0 1 2 3

```
__global__ void kernel( int* a ) {
    int i = threadIdx.x + blockIdx.x * blockDim.x;
    a[i] = blockIdx.x;
}
```

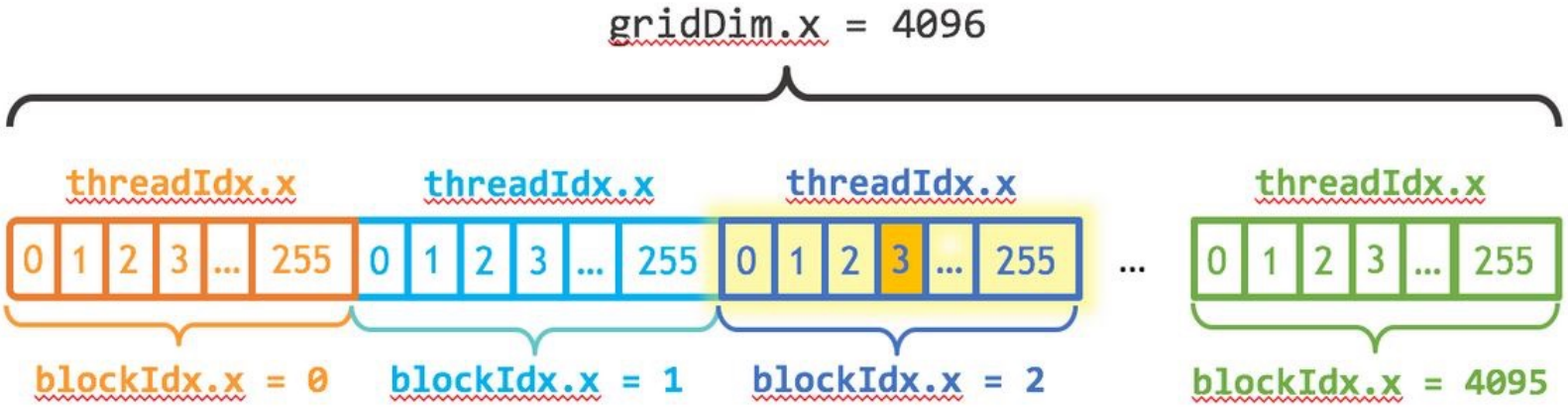
a: 0 0 0 0 1 1 1 1 2 2 2 2

```
__global__ void kernel( int* a ) {
      int i = threadIdx.x + blockIdx.x * blockDim.x;
      a[i] = i;
}
```

a: 0 1 2 3 4 5 6 7 8 9 10 11 12

Summary: CUDA – Grid Block and Thread

- One Grid per CUDA Kernel
- Multiple Blocks per Grid
- Multiple Threads per Block




```
index = blockIdx.x  * blockDim.x + threadIdx.x
index =    (2)      *   (256)    +     (3)      = 515
```

From: NVIDIA

Indexing and Memory Access

- Images are 2D data structures
  - height x width
  - Image[j][i], where 0 ≤ j < height, and 0 ≤ i < width

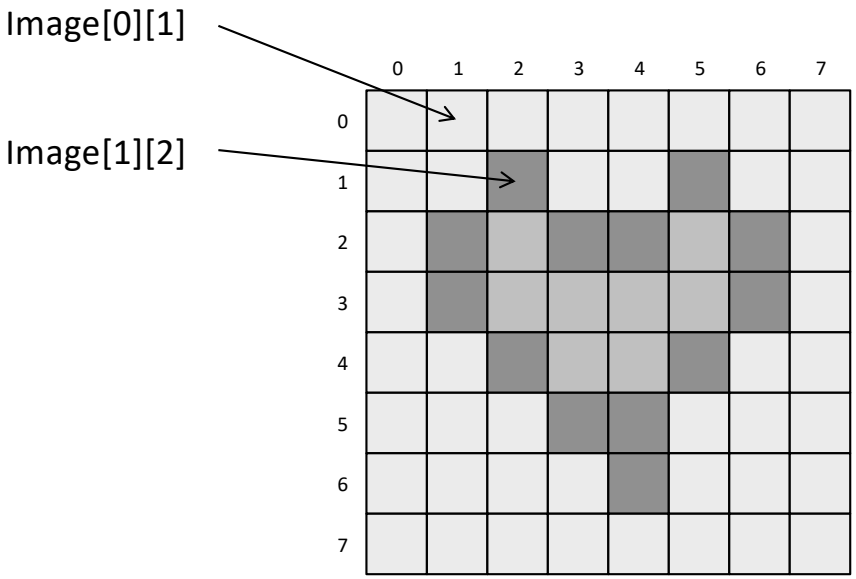
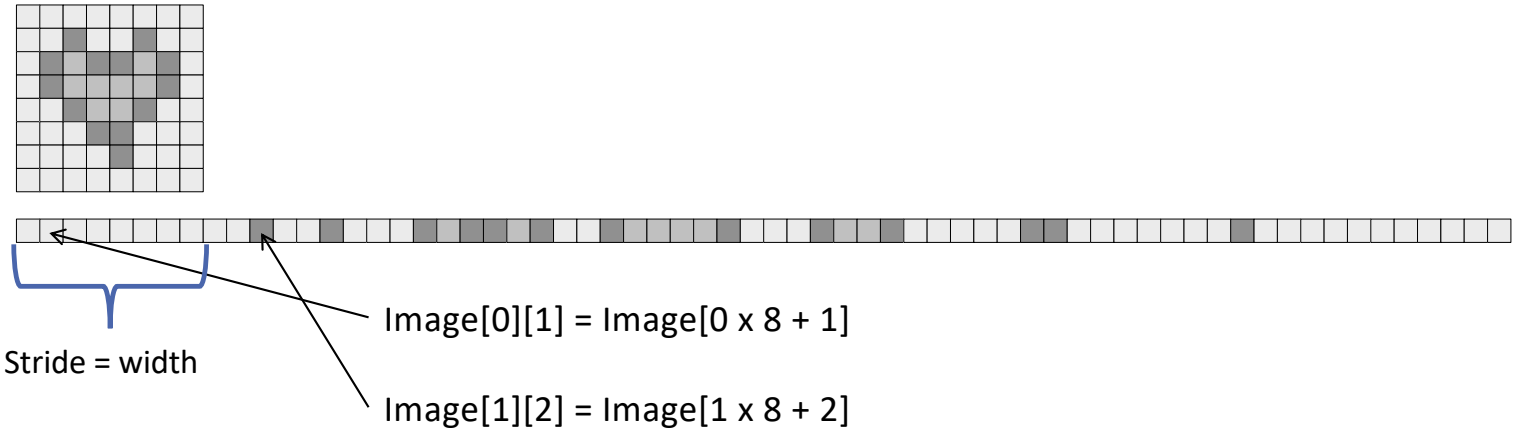


Image Layout in Memory

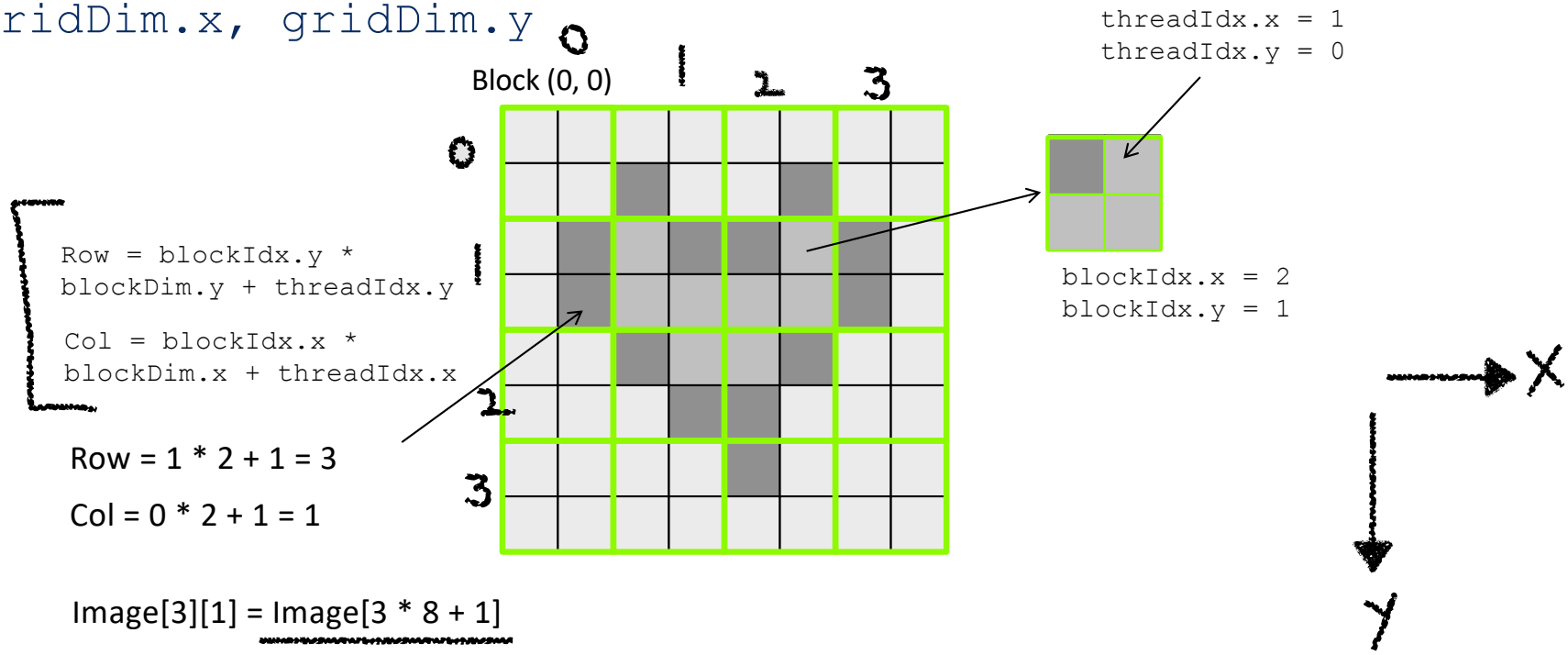
- Row-major layout
- Image[j][i] = Image[j x width + i]



Indexing and Memory Access: 2D Grid

- 2D blocks
  - `gridDim.x, gridDim.y`

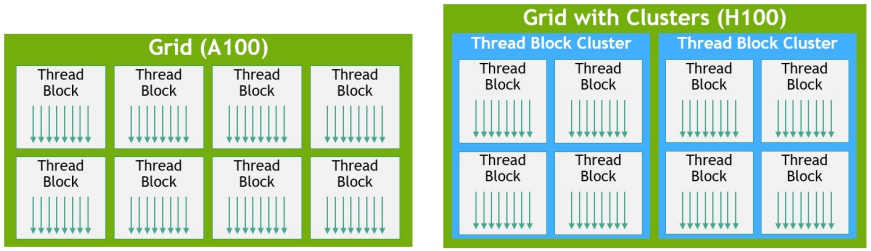
each block is having 4 threads in 2D fashion



As it is saved in a row-major fashion

NVIDIA H100: Thread Block Clusters

- GPUs grow beyond 100 GPU cores (SMs): a new level in the software hierarchy can improve execution efficiency
  - Programmatic control of locality at a granularity larger than a single thread block on a single SM
- Thread blocks in the same cluster can synchronize and exchange data
- Thread blocks in the same cluster are guaranteed to be concurrently scheduled
  - Thread blocks in the same cluster run on the SMs within a GPU Processing Cluster (GPC)



https://developer.nvidia.com/blog/nvidia-hopper-architecture-in-depth/

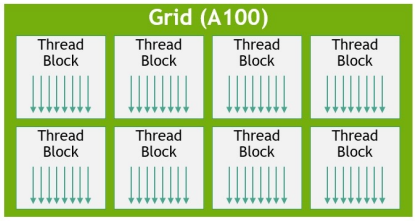
NVIDIA H100: Thread Block Clusters

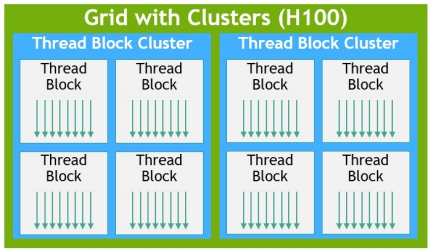
- GPUs grow beyond 100 GPU cores (SMs): a new level in the software hierarchy can improve execution efficiency
  - Programmatic control of locality at a granularity larger than a single thread block on a single SM

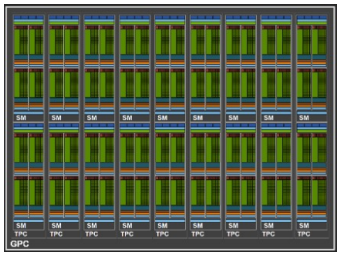
https://developer.nvidia.com/blog/nvidia-hopper-architecture-in-depth/

NVIDIA H100: Thread Block Clusters

- GPUs grow beyond 100 GPU cores (SMs): a new level in the software hierarchy can improve execution efficiency
  - Programmatic control of locality at a granularity larger than a single thread block on a single SM
- Thread blocks in the same cluster can synchronize and exchange data
- Thread blocks in the same cluster are guaranteed to be concurrently scheduled
  - Thread blocks in the same cluster run on the SMs within a GPU Processing Cluster (GPC)
  - Data sharing via SM-to-SM network in a GPC







Thread - Thread block - Thread block cluster - Grid

https://developer.nvidia.com/blog/nvidia-hopper-architecture-in-depth/

CUDA Memory Model

Thread Memory Correspondence

Threads ⇔ Local Memory (and Registers)

- Scope: Private to its corresponding Thread
- Lifetime: Thread

registers are on-chip
Local memory is off-chip

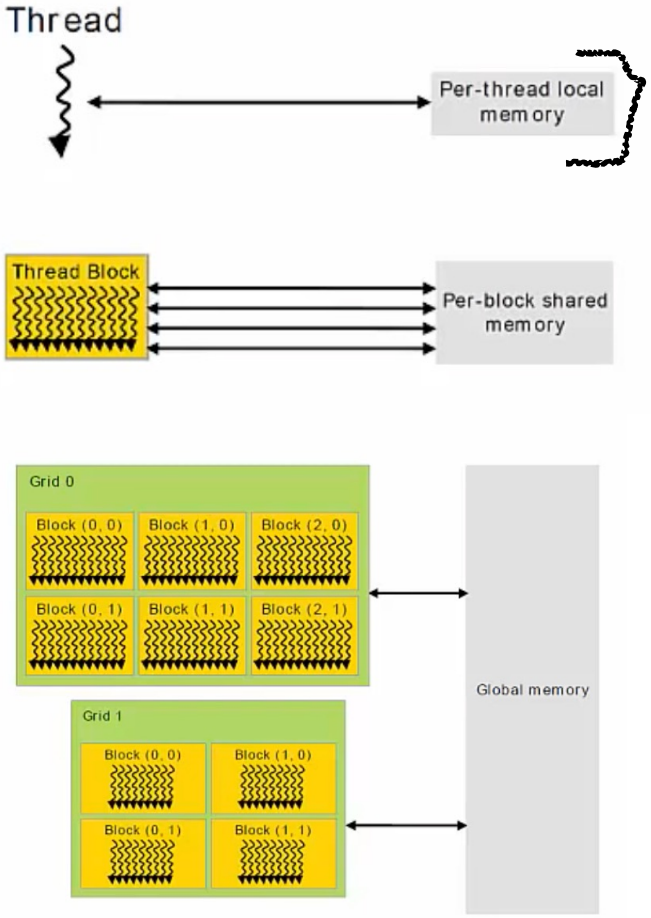
Blocks ⇔ Shared Memory

- Scope: Every Thread in the Block has access
- Lifetime: Block

Grids ⇔ Global Memory

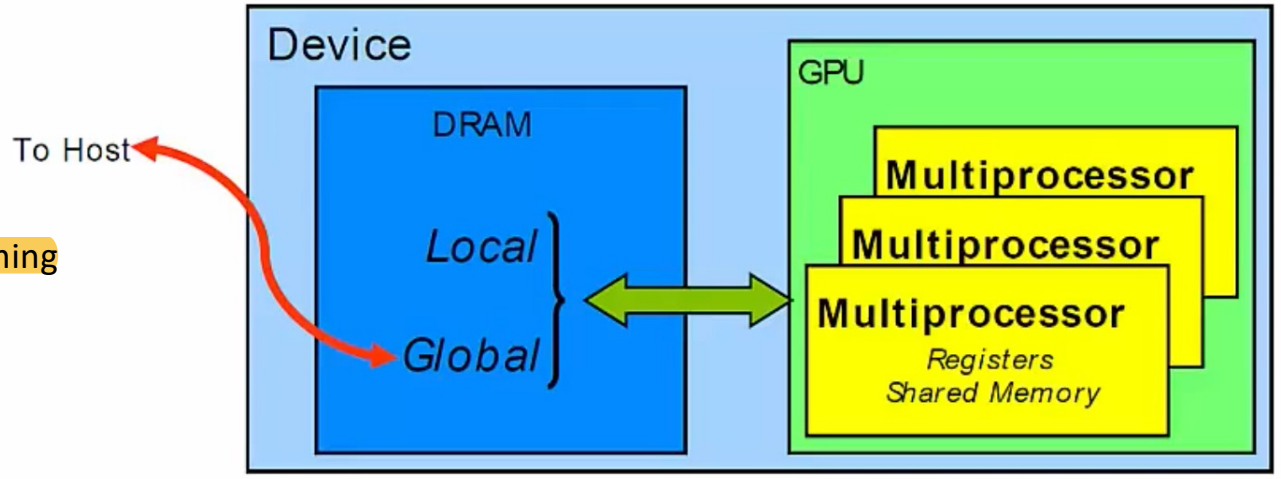
- Scope: Every Thread in all Grids have access
- Lifetime: Entire program in Host code – `main()`

multiple grids have access to it

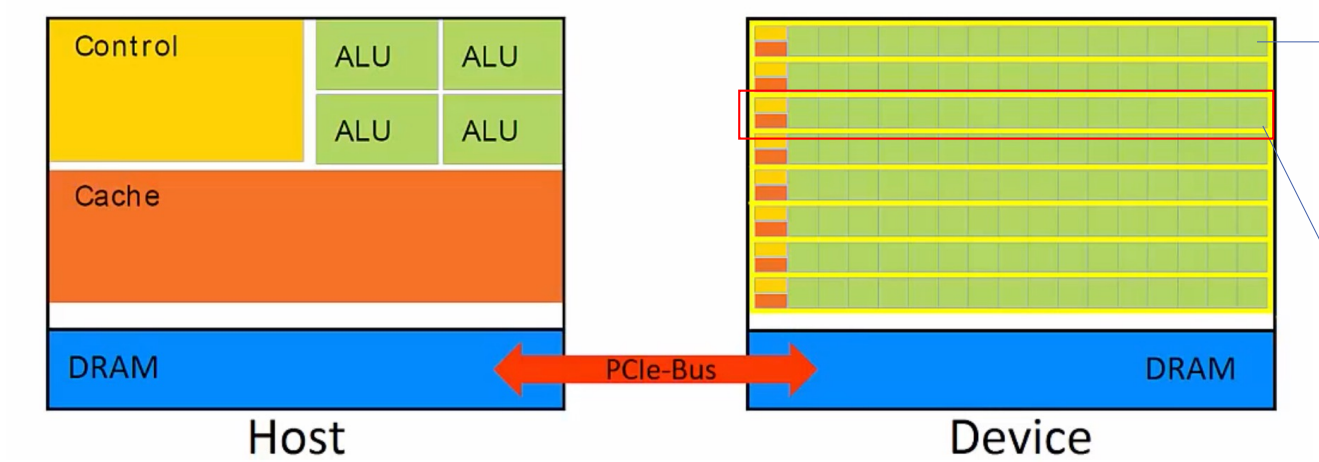


Memory Model

- Yellow rectangles represent SM: Streaming Multiprocessors
- Memory located in SMs is called:
  - **"On-chip"** Device memory
- Memory not in SM is called
  - **"Off-chip"** Device memory

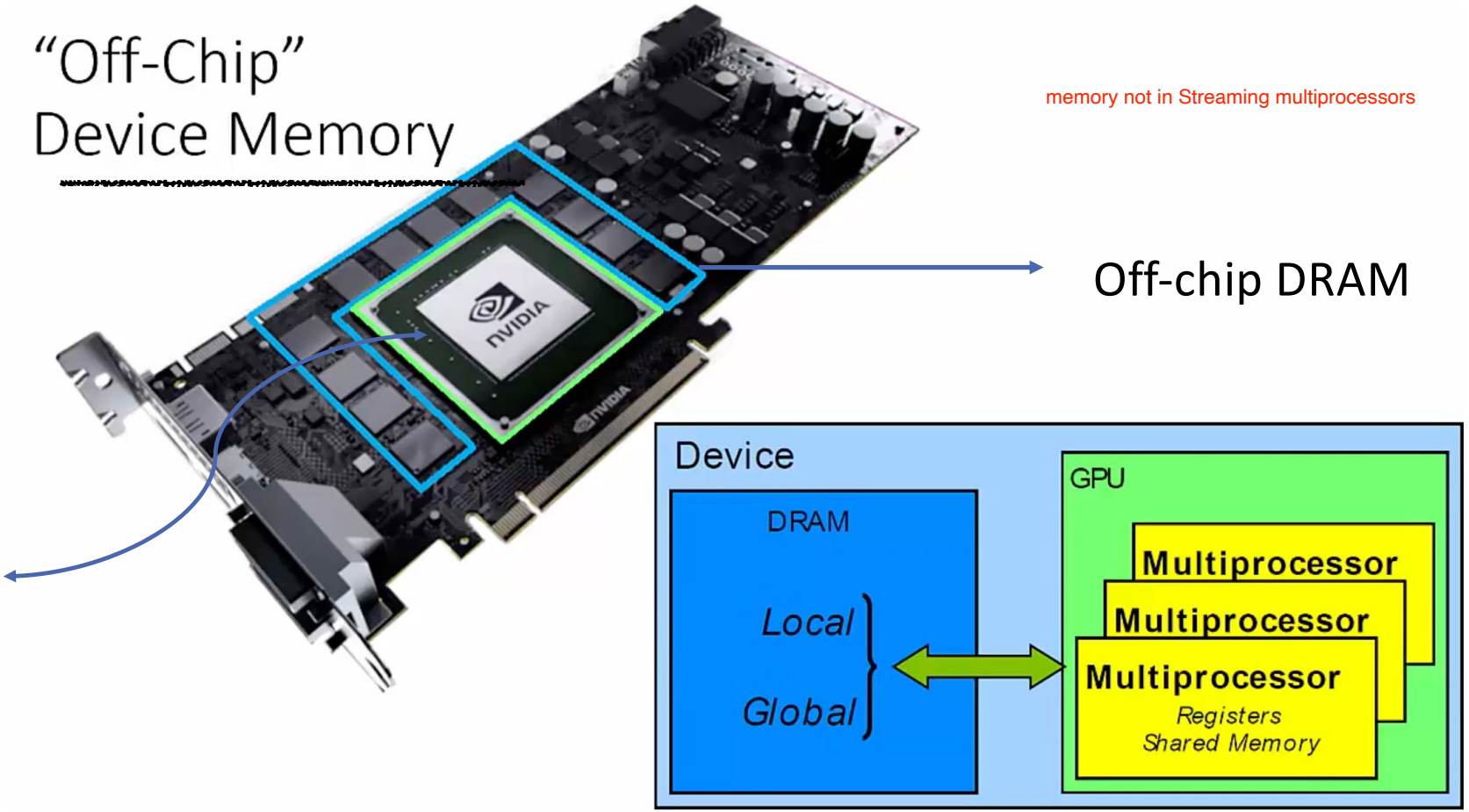


Term **local** refers to the scope of lifetime and not physical location

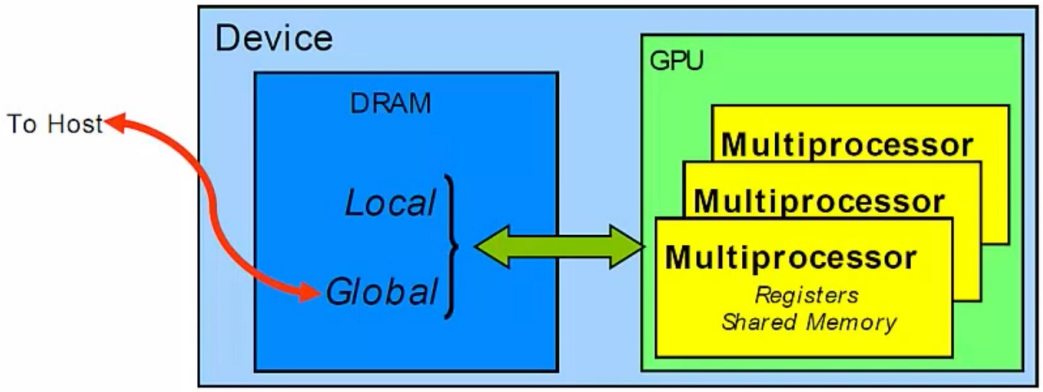


Streaming Multiprocessors

- An SM is a collection of processing units that work together to execute CUDA kernels. Each SM consists of several CUDA cores responsible for executing instructions in parallel.
- Each SM has "CUDA" cores, which are ALU units that can execute SIMD instructions
- The stream multiprocessor (SM) is a key computational grouping within a GPU, although "stream multiprocessor" is Nvidia's terminology. AMD would call them "compute units".
- "CUDA cores" would be called "shader units" or "stream processors" by AMD.


"Off-Chip"
Device Memory
memory not in Streaming multiprocessors
Off-chip DRAM
GPU
Device
DRAM
Local
Global
GPU
Multiprocessor
Multiprocessor
Multiprocessor
Registers
Shared Memory
Off-chip DRAM
On-chip

Memory Speed



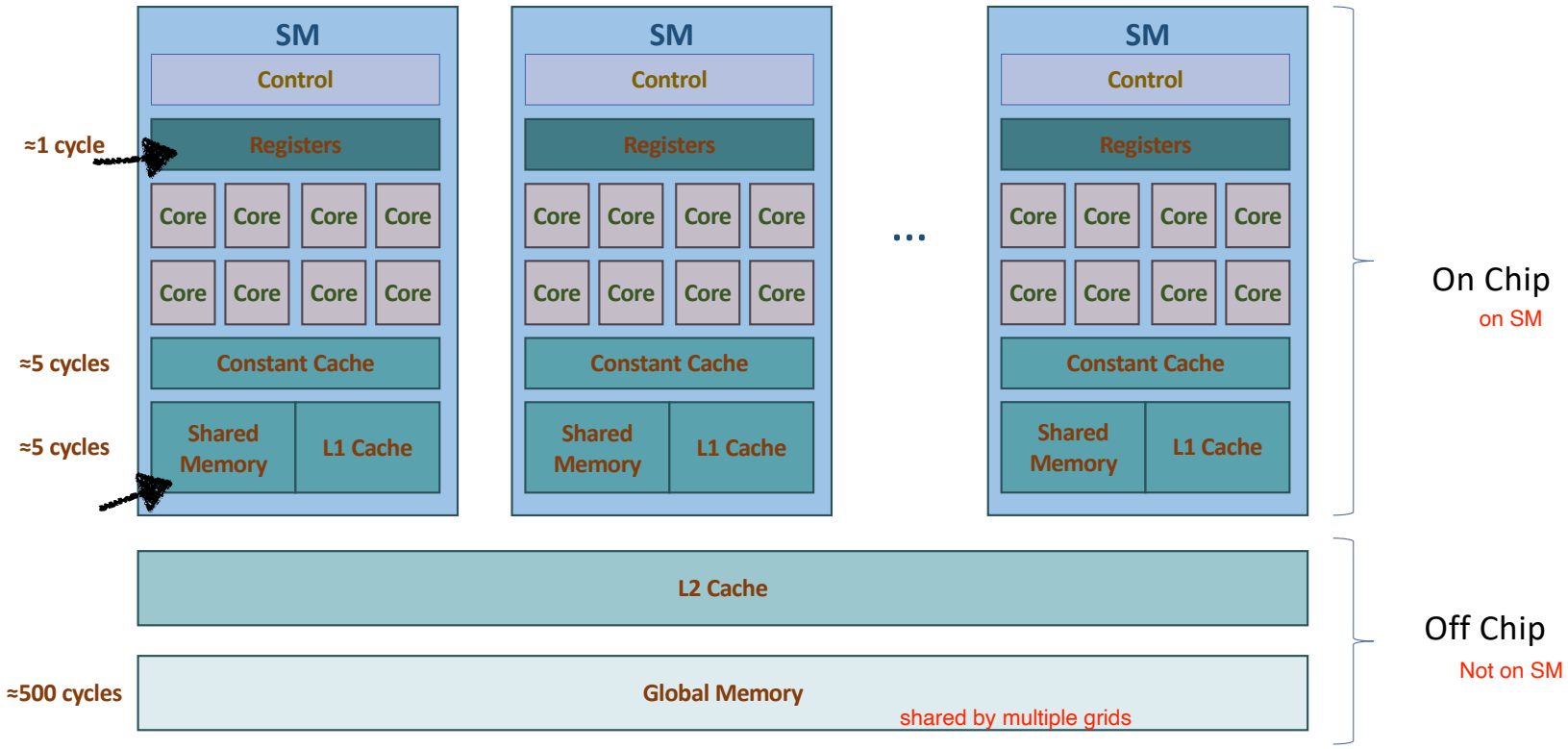
- Relative speed of memory spaces: $\underbrace{\text{"Memory Space"}}_{\substack{\text{"Bandwidth"}\\ \text{"Latency"}}}$

as both are off-chip memory

$$\underbrace{\text{Registers}}_{\substack{\text{bandwidth } \sim 8\text{TB/s}\\ \text{latency } \sim 1\text{clock}}} < \underbrace{\text{Shared}}_{\substack{\sim 1.5\text{TB/s}\\ \sim 32\text{clocks}}} \ll \underbrace{\text{Local} \approx \text{Global}}_{\substack{\sim 200\text{GB/s}\\ \sim 800\text{clocks}}} \ll \underbrace{\text{Host (PCIe)}}_{\sim 5\text{GB/s}}$$

but local memory is private to a thread

Memory in the GPU Architecture



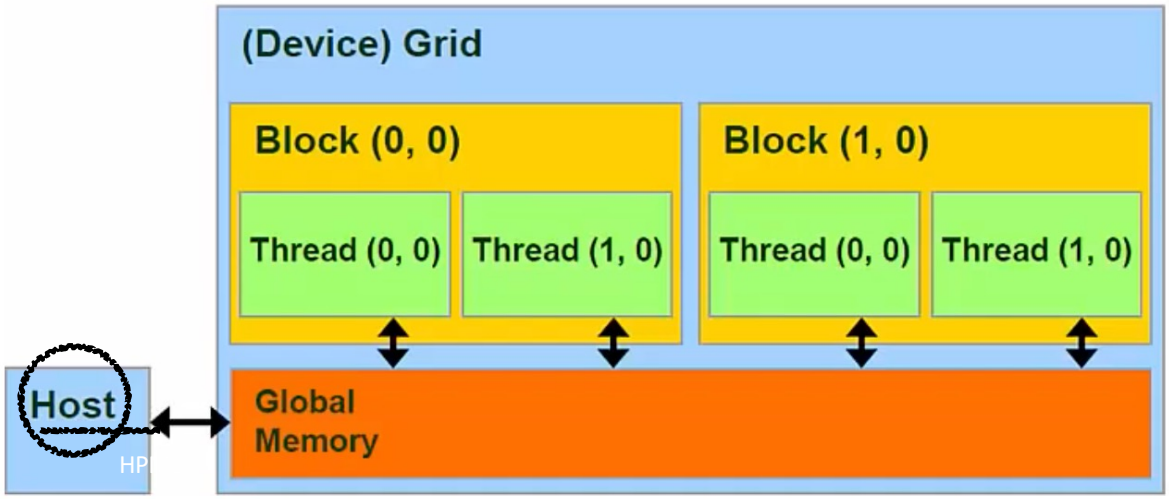
Slide credit: Izzat El Hajj

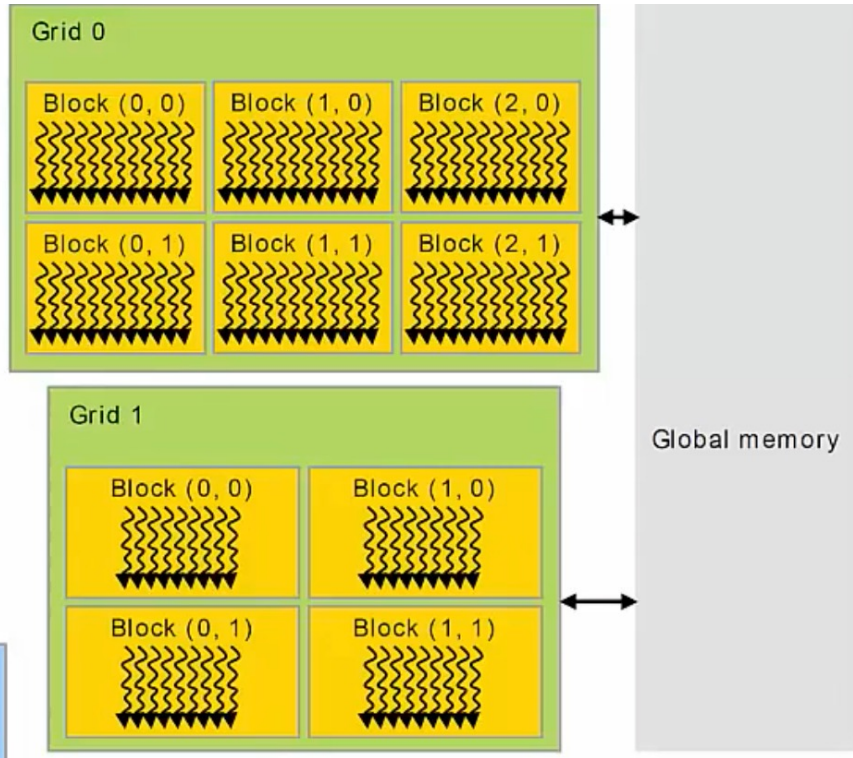
memory which is shared by multiple grids

Global Memory

Accessed with

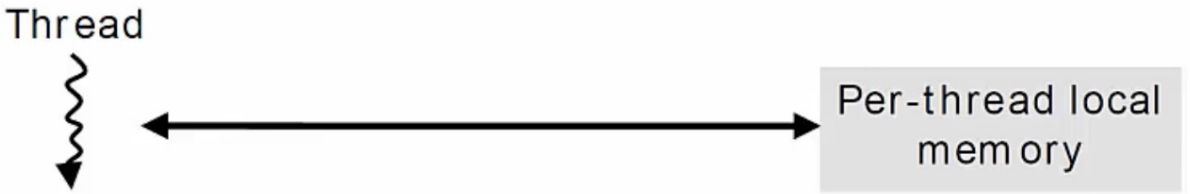
- `cudaMalloc()`
- `cudaMemset()`
- `cudaMemCopy()`
- `cudaFree()`



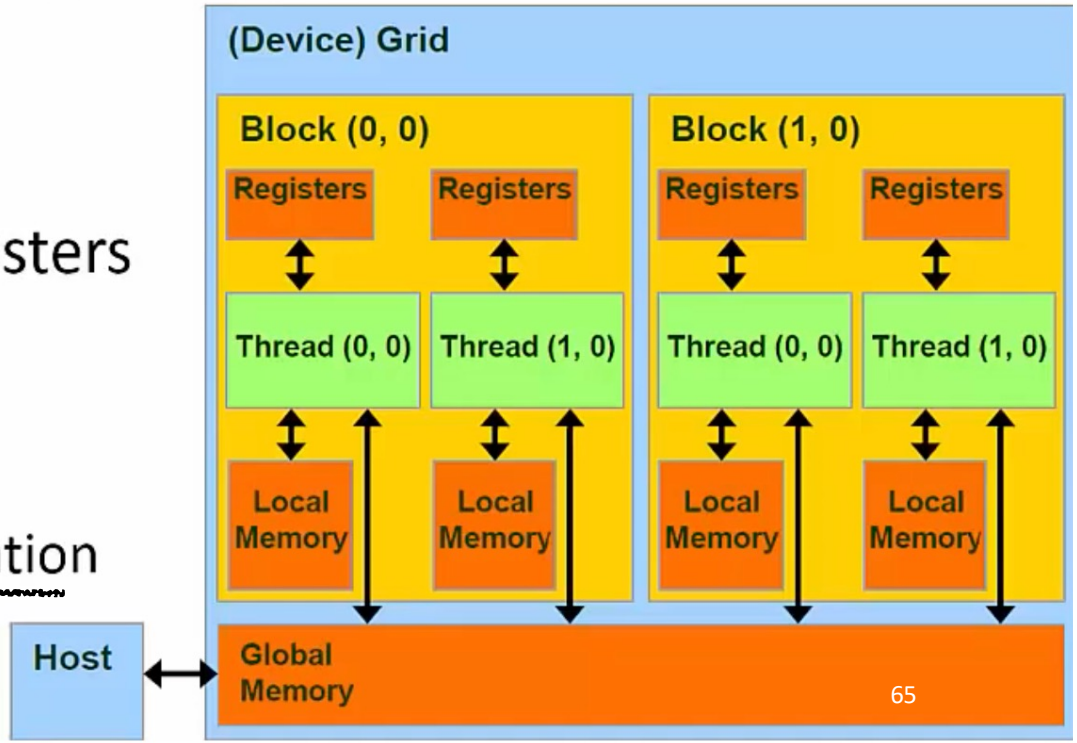


i.e. host can only communicate with global memory

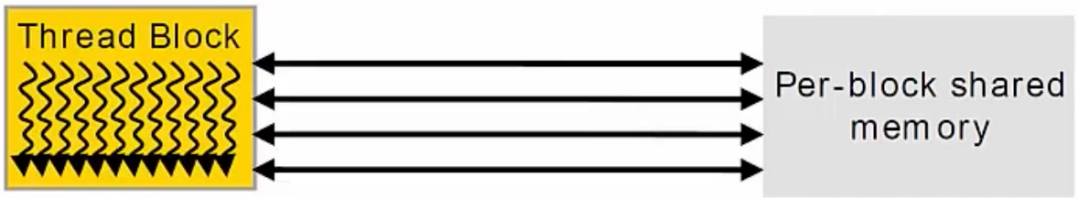
Registers and Local Memory



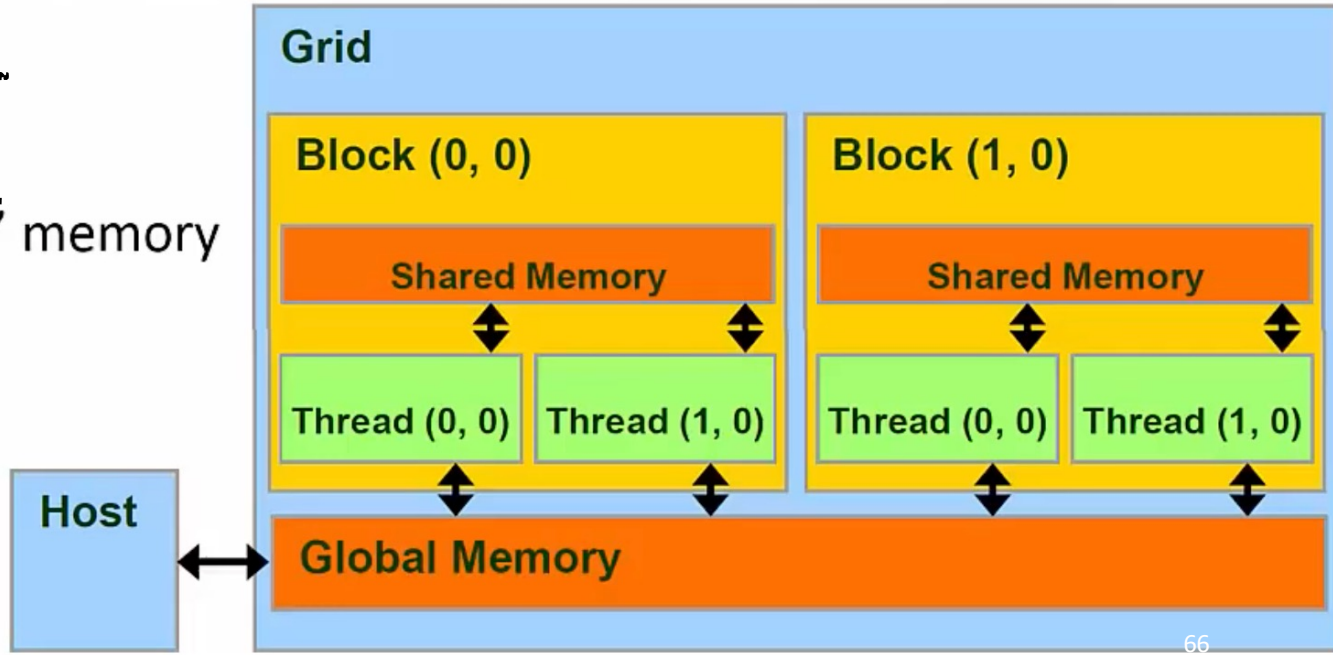
- Variables declared in a Kernel are stored in Registers
  - *On-Chip*
  - Fastest form of memory
- Arrays too large to fit into Registers spill over into Local memory
  - *Off-Chip*
  - Compiler controlled
  - "Local" refers to scope, not location
    - Local to each Thread



Shared Memory



- Allows Threads within a Block to communicate with each other
  - Use synchronization
- Very fast
  - Only Registers are faster
- Can use as "Scratch-pad" memory

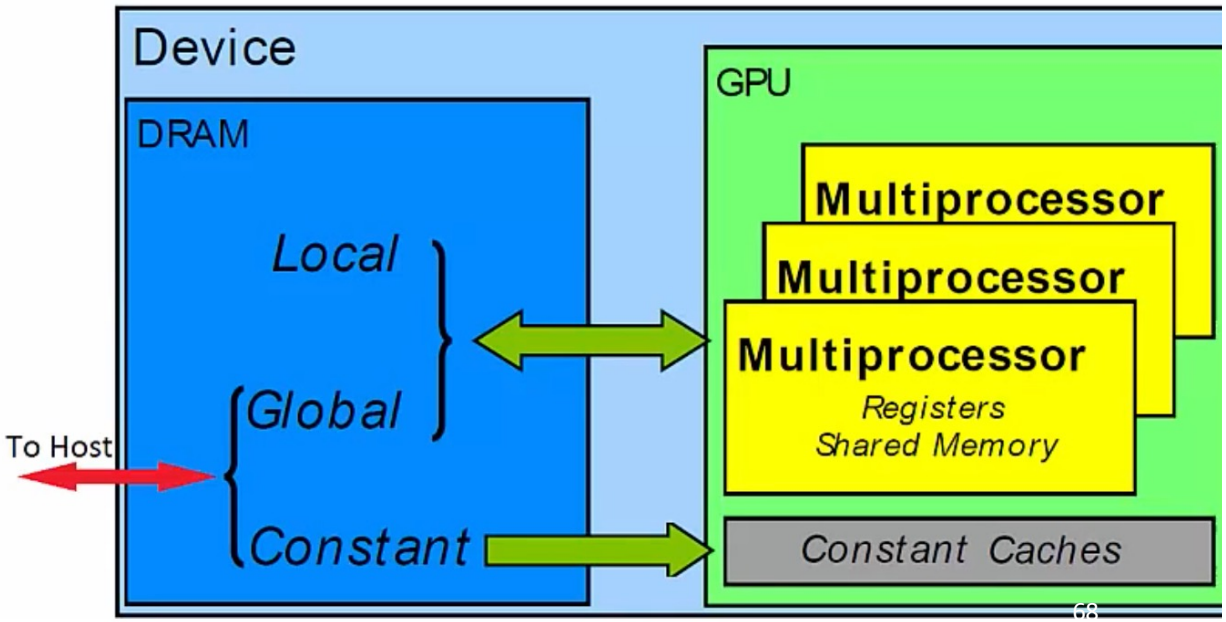


Using Shared Memory

```
__global__ void kernel( int* in, int N )
{
    int i  = threadIdx.x + blockDim.x * blockIdx.x;

    // Allocate a shared array
    __shared__ int shared_array[N];

    // each thread writes to one element of shared_array
    shared_array[i] = in[i];

    // Do more stuff
    // ...
}
```

Constant Memory

- Special Region of Device Memory
  - Used for data with unchanging contents throughout kernel execution
  - Read–Only from Kernel
- *Off Chip*
- Constant memory is aggressively cached into *On-Chip* memory



Memory Model

Registers & Local Memory

- Regular variables declared within a Kernel

spills from registers go into local memory

Shared Memory

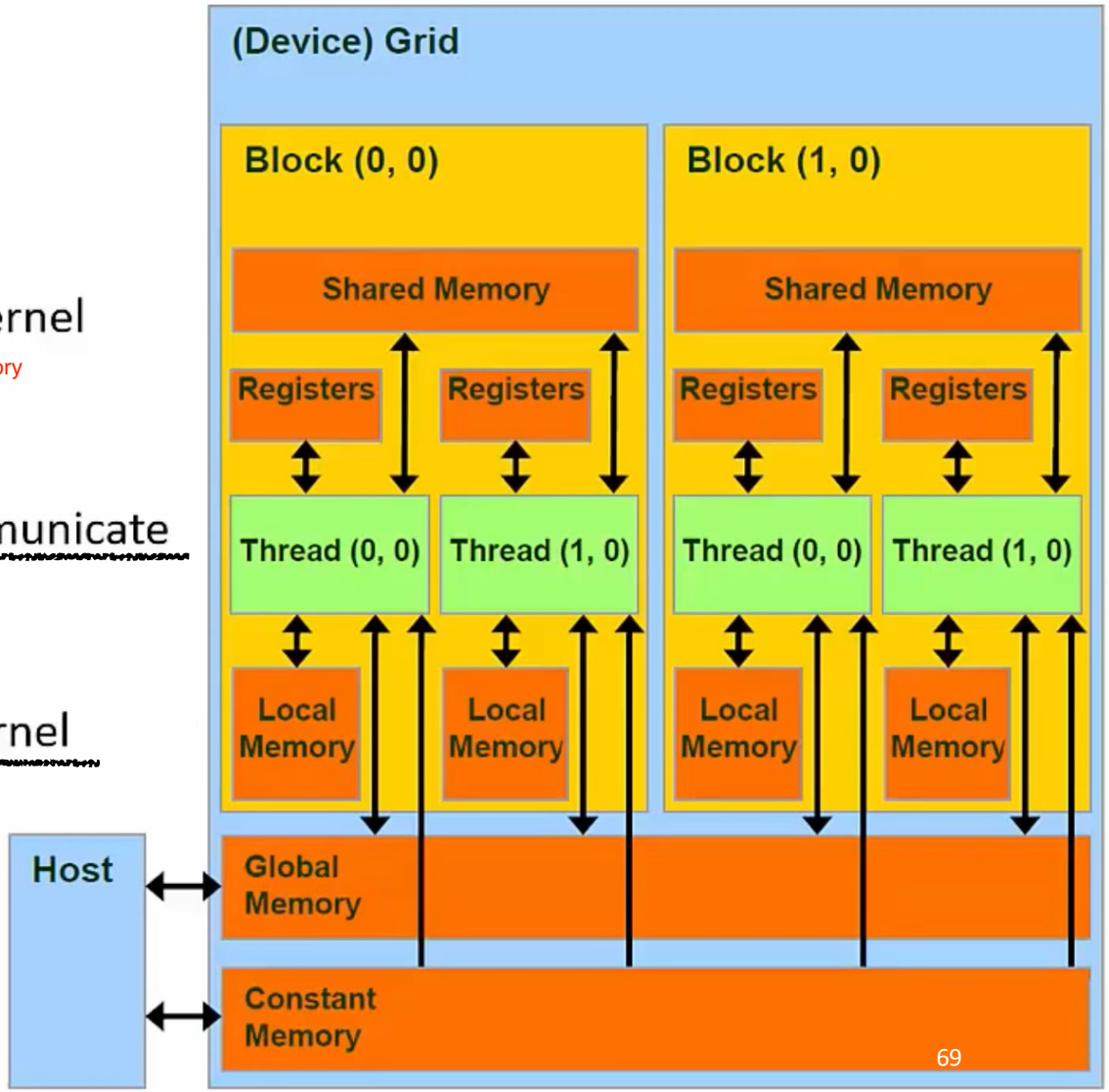
- Allows threads within a block to communicate

Constant

- Used for unchanging data through Kernel

Global Memory

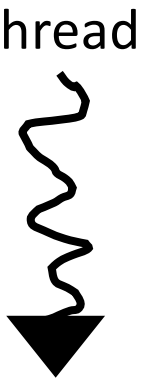
- Stores data copied to and from Host



CUDA Thread Synchronization

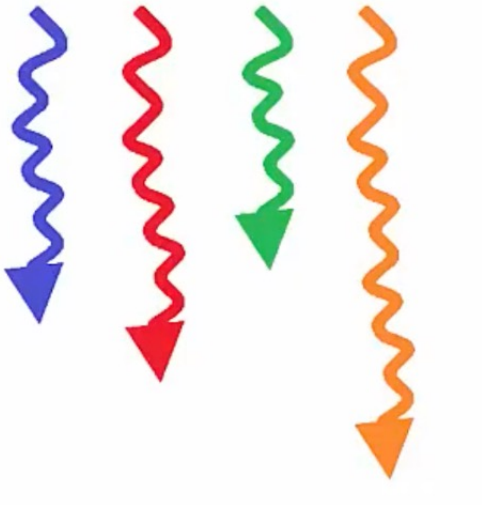
Threads Execute in Parallel

- Threads can read a result before another thread writes to that address
  - Race condition



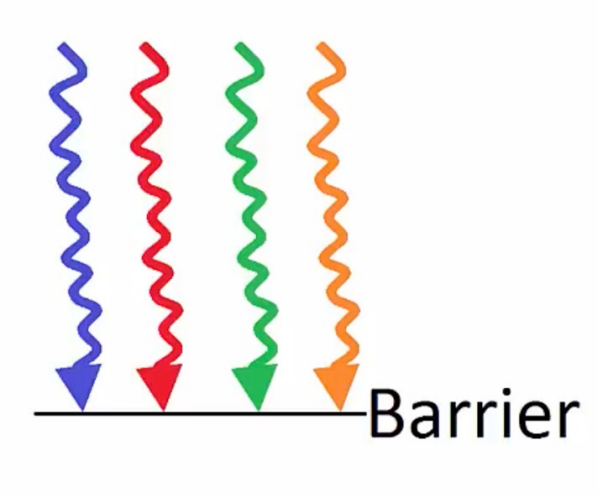
Thread Synchornization via Explicit Barrier

- Threads need to synchronize with one another to avoid race conditions



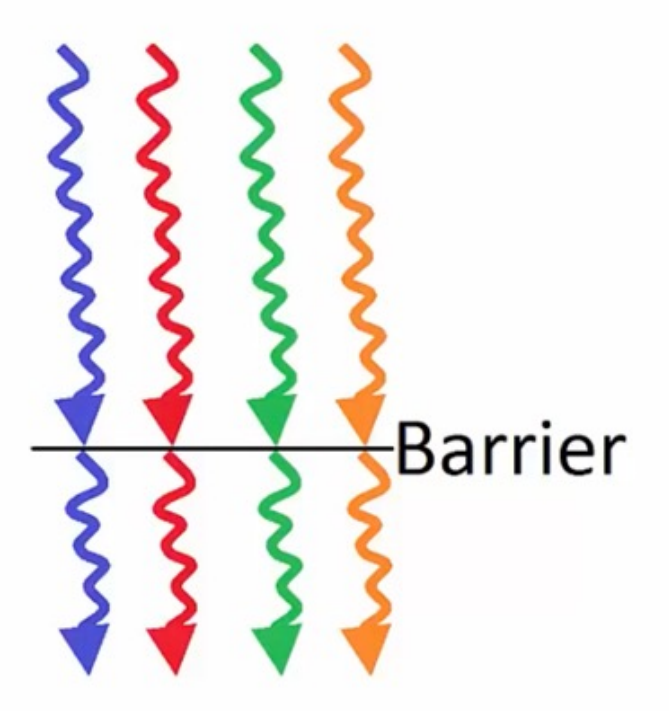
Thread Synchornization via Explicit Barrier

- Threads need to synchronize with one another to avoid race conditions
- A barrier is a point in the kernel where all the threads stop and wait on the others



Thread Synchornization via Explicit Barrier

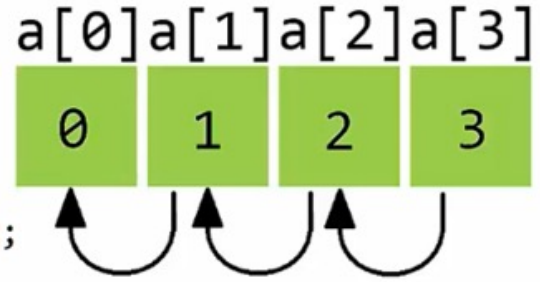
- Threads need to synchronize with one another to avoid race conditions
- A barrier is a point in the kernel where all the threads stop and wait on the others
- When all threads have reached the barrier, they can proceed
- Barriers can be implemented with:
  - __syncthreads();



syncthreads() Example

- Goal: Shift the contents of an array to the left by one elements

```
// Kernel Definition
__global__ void kernel( int* a )
{
    int i = threadIdx.x + blockIdx.x * blockDim.x;
    if( i < 3 )
    {
        int temp = a[i+1];
        __syncthreads;
        a[i] = temp;
        __syncthreads;
    }
}
```

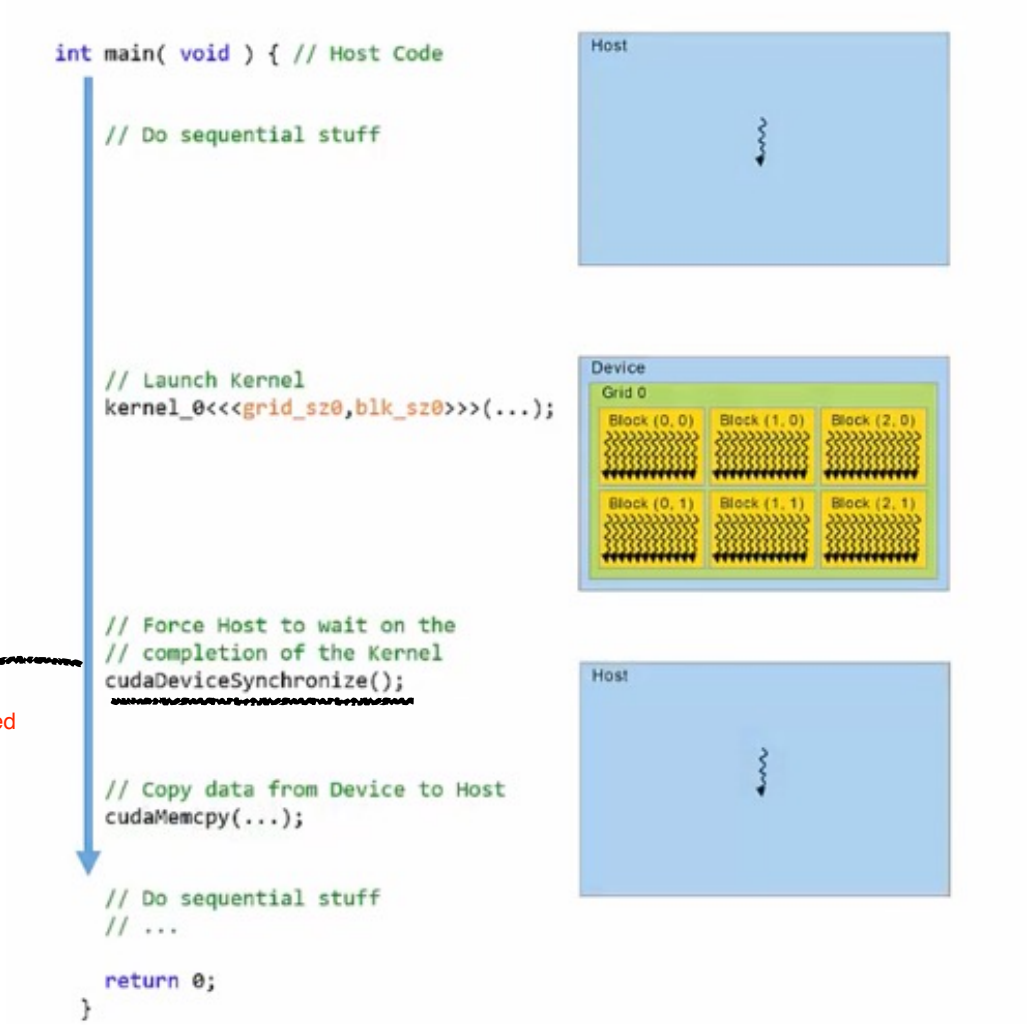


is syncthreads was not used, we might not get the desired result as all threads run independently at their own pace.

temp is a varaible local to each thread - it will be created in register memory

So, sybcthreads is important if in our program multiole threads might read/write on same data

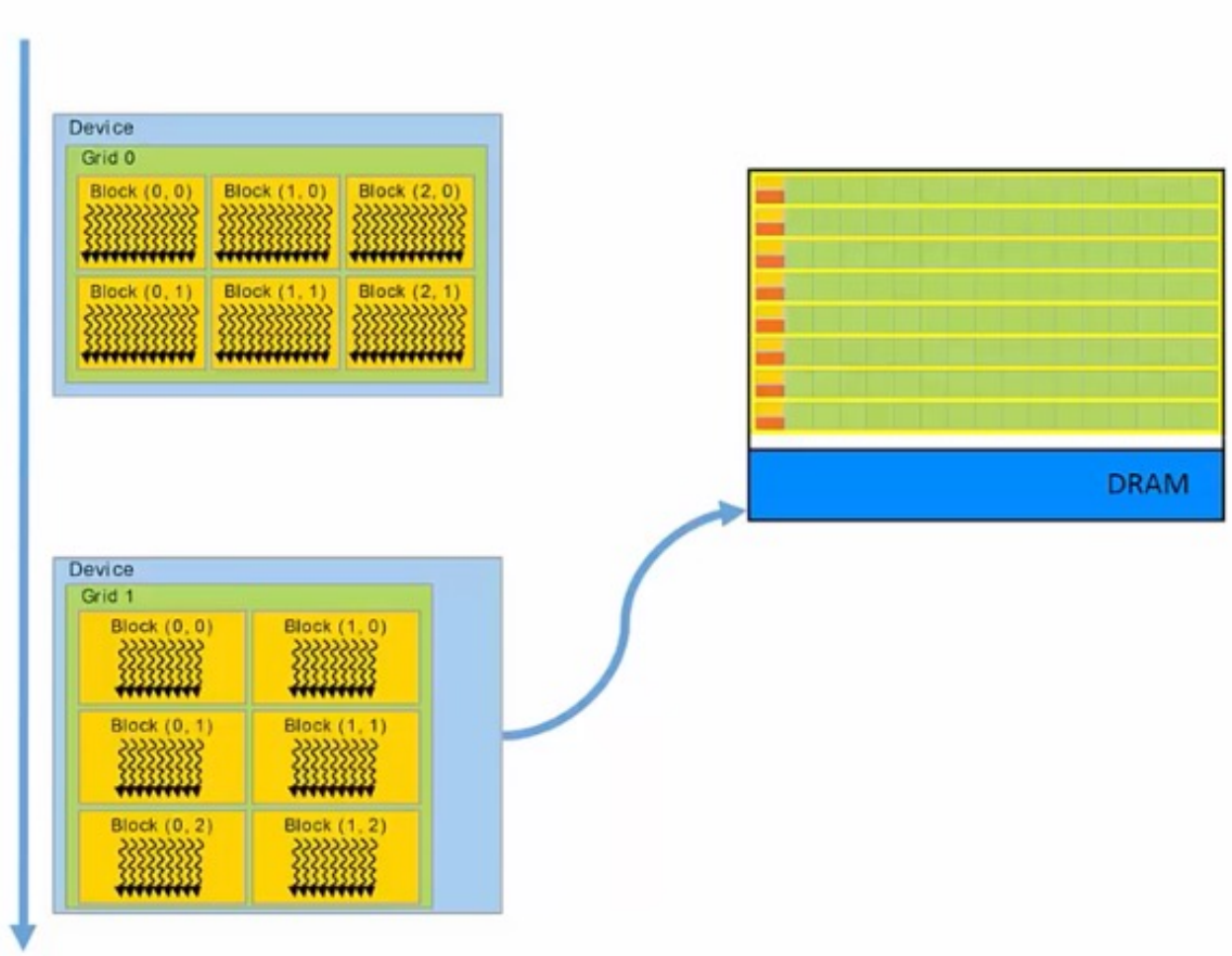
Implicit Barriers between Kernels



forces host to wait until kernel is finished

Implicit Barriers between Kernels

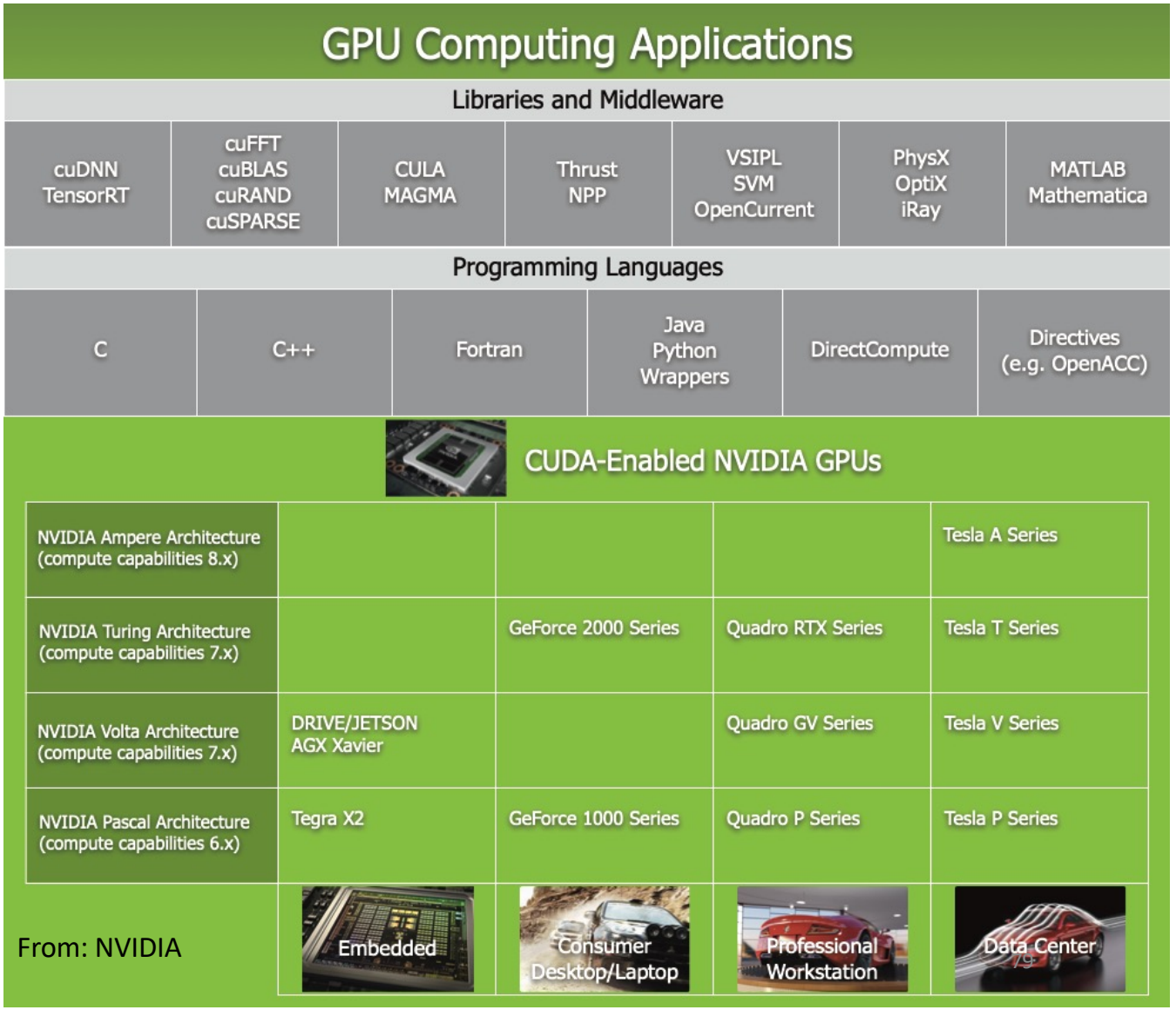
kernel launched first will be first executed and then so on



NVIDIA GPUs and CUDA

CUDA

- CUDA®: *A General-Purpose Parallel Computing Platform and Programming Model*
- Introduced in 2006 by NVIDIA
- Key objectives:
  - **Scalability**
  - **Portability**
- CUDA has different compute capabilities for each architecture
- http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#introduction



From: NVIDIA

CUDA – Compute Capability

- Compute Capability:
  - represented by a version number (ex. 6.2)
    - Major revision number (ex. 6) identifies the core architecture
    - Minor revision number identifies incremental improvements (ex. .2)
  - Identifies the features supported by the GPU hardware
  - Used by applications at runtime to identify available features
- Do not confuse with CUDA version
  - Tesla and Fermi not supported on CUDA 7.0 and 9.0 respectively

Compute Capability Major revision	NVIDIA GPU Architecture
1	Tesla
2	Fermi
3	Kepler
5	Maxwell
6	Pascal
7	Volta
7.5	Turing
8	Ampere
9	Hopper

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

NVIDIA Hardware and Compute Capabilities

Feature Support	Compute Capability					
(Unlisted features are supported for all compute capabilities)	5.0, 5.2	5.3	6.x	7.x	8.x	9.0
Atomic functions operating on 32-bit integer values in global memory (Atomic Functions)	Yes					
Atomic functions operating on 32-bit integer values in shared memory (Atomic Functions)	Yes					
Atomic functions operating on 64-bit integer values in global memory (Atomic Functions)	Yes					
Atomic functions operating on 64-bit integer values in shared memory (Atomic Functions)	Yes					
Atomic functions operating on 128-bit integer values in global memory (Atomic Functions)	No					Yes
Atomic functions operating on 128-bit integer values in shared memory (Atomic Functions)	No					Yes
Atomic addition operating on 32-bit floating point values in global and shared memory (atomicAdd())	Yes					
Atomic addition operating on 64-bit floating point values in global memory and shared memory (atomicAdd())	No	Yes				
Atomic addition operating on float2 and float4 floating point vectors in global memory (atomicAdd())	No					Yes
Warp vote functions (Warp Vote Functions)	Yes					
Memory fence functions (Memory Fence Functions)	Yes					
Synchronization functions (Synchronization Functions)	Yes					
Surface functions (Surface Functions)	Yes					
Unified Memory Programming (Unified Memory Programming)	Yes					
Dynamic Parallelism (CUDA Dynamic Parallelism)	Yes					
Half-precision floating-point operations: addition, subtraction, multiplication, comparison, warp shuffle functions, conversion	No	Yes				
Bfloat16-precision floating-point operations: addition, subtraction, multiplication, comparison, warp shuffle functions, conversion	No				Yes	
Tensor Cores	No			Yes		
Mixed Precision Warp-Matrix Functions (Warp matrix functions)	No			Yes		
Hardware-accelerated `memcpy_async` (Asynchronous Data Copies using cuda::pipeline)	No				Yes	
Hardware-accelerated Split Arrive/Wait Barrier (Asynchronous Barrier)	No				Yes	
L2 Cache Residency Management (Device Memory L2 Access Management)	No				Yes	
DPX Instructions for Accelerated Dynamic Programming	No					Yes
Distributed Shared Memory	No					Yes
Thread Block Cluster	No					Yes
Tensor Memory Accelerator (TMA) unit	No					Yes

https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#compute-capabilities

NVIDA Micro-Architectures

Year	μArch	Series	Process	Xistors	Notes
1999	—	GeForce 256	220 nm		First GPU
2008	Tesla	GeForce 8 (8xxx)	65 nm	210 M	CUDA appears
2008		GeForce 9 (9xxx)		1.4 B	
2009		GeForce 200	40 nm		
2010	Fermi	GeForce 400	40 nm		
2011		GeForce 500		3.0 B	
2012	Kepler	GeForce 600	28 nm	7.1 B	
2014		GeForce 700		7.1 B	
2014	Maxwell	GeForce 900	28 nm	8.0 B	
2016	Pascal	GeForce 10	16 nm	15.3 B	Unified memory
2017	Volta		12 nm	21.1 B	Tensor cores
2019	Turing	GeForce 16	12 nm	6.6 B	RT ray tracing
2018		GeForce 20		18.6 B	
2020	Ampere Hopper	GeForce 30A	7 nm	28.3 B	

Lesson Outline

- Heterogenous architectures motivations
- Hierarchy of Computations:
  - Threads
  - Blocks
  - Grids
- Corresponding Memory Spaces
  - Local
  - Shared
  - Global
- Synchronization Primitives
  - Implicit Barriers
  - Thread Synchronization
- NVIDIA GPUs and CUDA:
  - Compute capability
- CUDA Hardware
- CUDA Compilation and Runtime:
  - CUDA Runtime, CUDA Driver, AoT and JIT compilation
- CUDA Programming Model:
  - Grid, Block, Thread
  - UVM
- CUDA Warp Scheduling
- Context and Stream
- CUDA Profiling and Debugging

Lesson Outline

- Heterogenous architectures motivations
- Hierarchy of Computations:
  - Threads
  - Blocks
  - Grids
- Corresponding Memory Spaces
  - Local
  - Shared
  - Global
- Synchronization Primitives
  - Implicit Barriers
  - Thread Synchronization
- NVIDIA GPUs and CUDA:
  - Compute capability

- CUDA Hardware
- CUDA Compilation and Runtime:
  - CUDA Runtime, CUDA Driver, AoT and JIT compilation
- CUDA Programming Model:
  - Grid, Block, Thread
  - UVM
- CUDA Warp Scheduling
- Context and Stream
- CUDA Profiling and Debugging

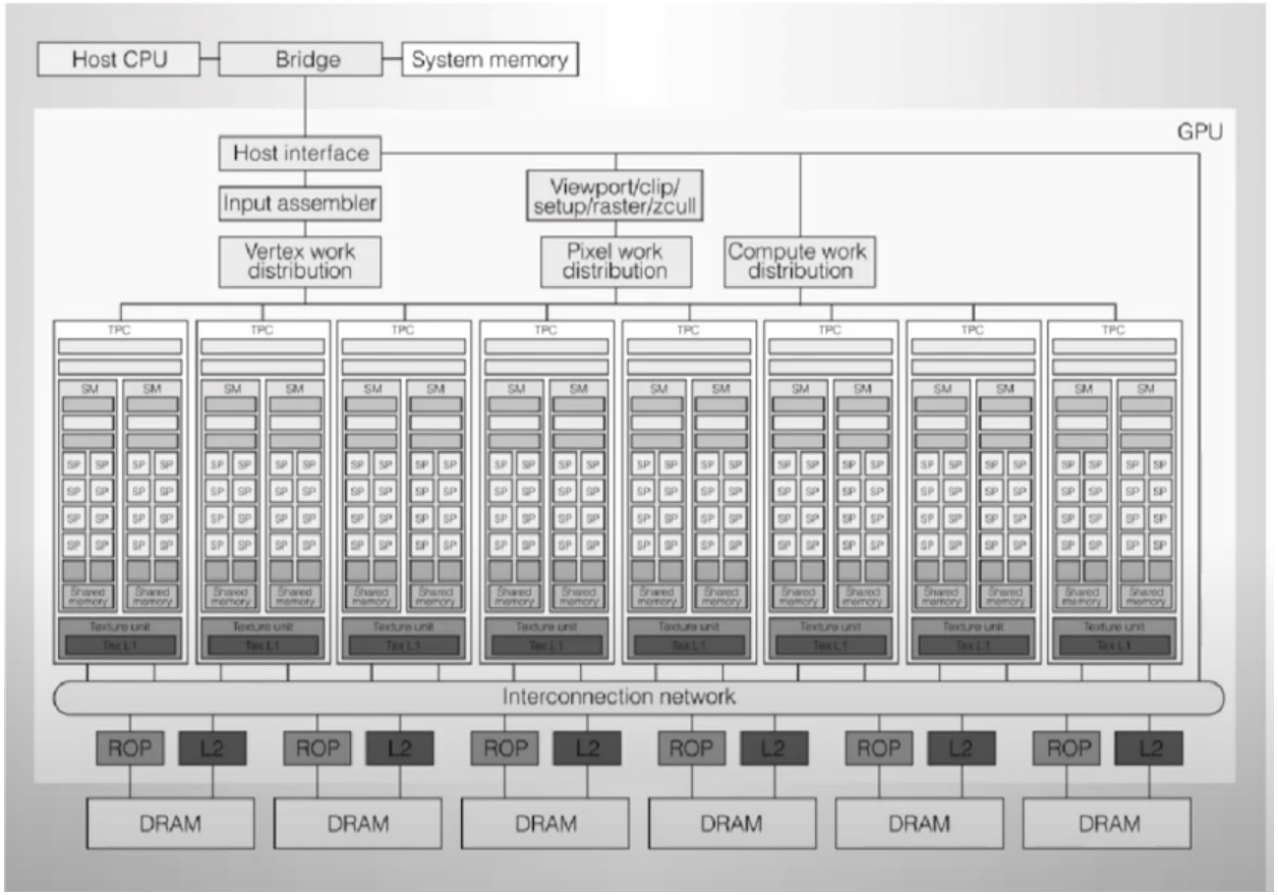
go to other slide

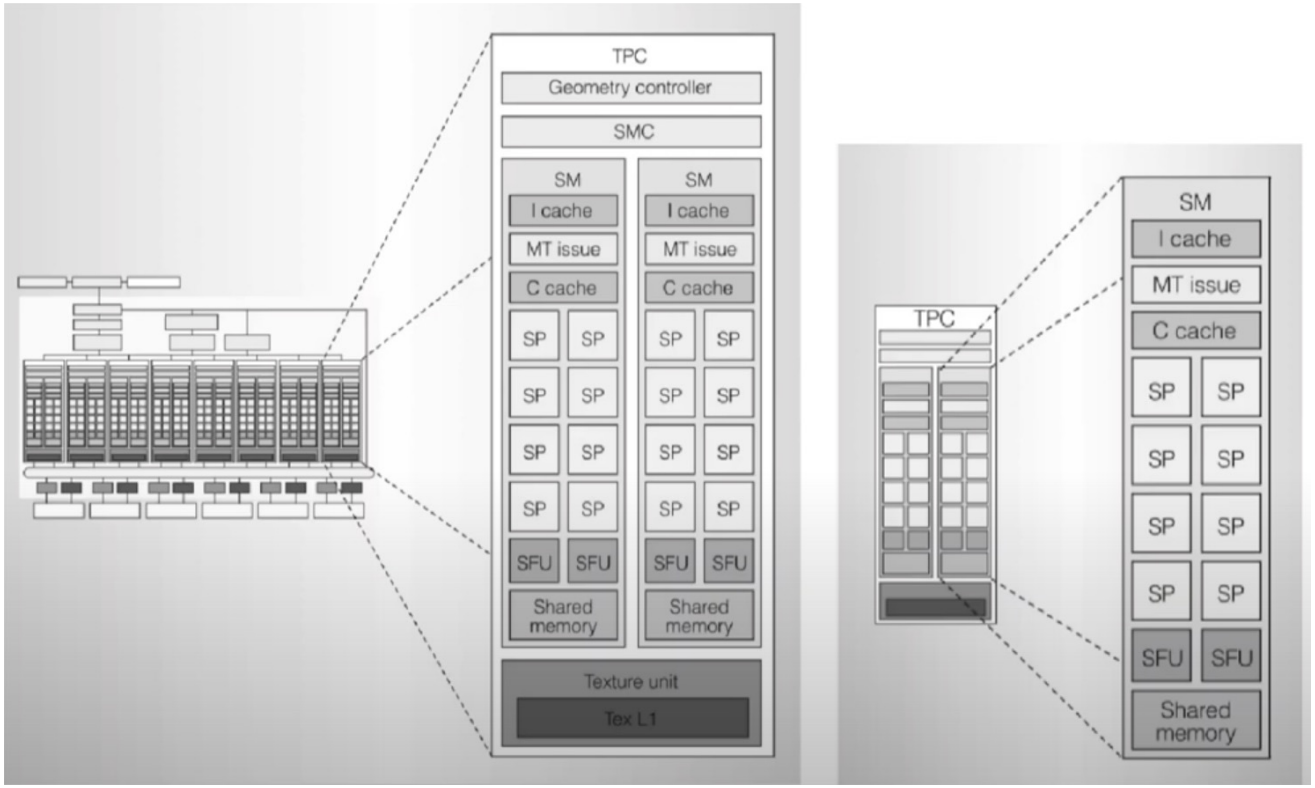
CUDA Hardware

Terminology

Term	Definition
SM	Streaming Multiprocessor
SP	Streaming Processor
TPC	Texture/Processor Cluster
GPC	Graphics Processing Cluster
SP	Single-Precision (32-bit)
DP	Double-Precision (64-bit)

GeForce 8800 (2008)





I Cache: Instruction Cache

C Cache: Constant Cache

SF: Special Function Unit

Pascal GPU (2016)

- 6 GPC
- 10 SM/GPC
- 60 SM
- 64 SP/SM
- 3840 SP

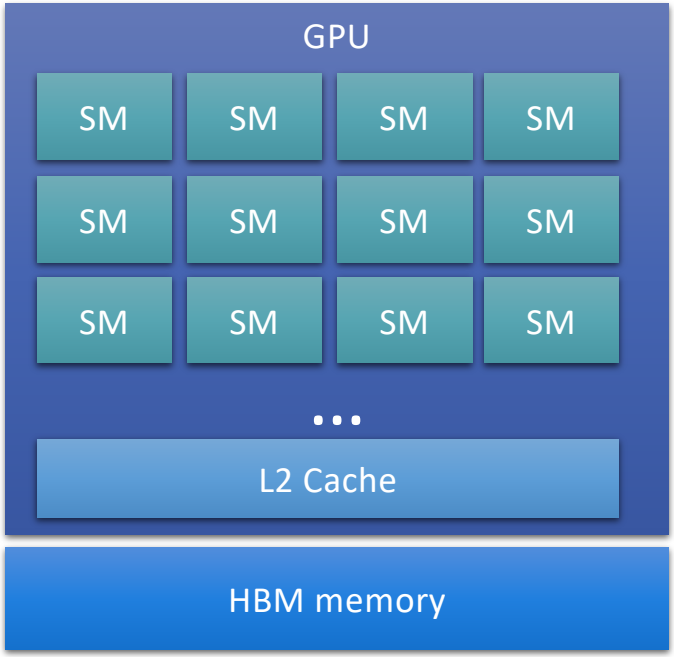
Pascal GPU (2016) - Closeup

Volta GPU (2017)

- 6 GPC
- 14 SM/GPC
- 84 SM
- 64 SP Float Cores/SM
- 64 SP Int Cores/SM
- 32 DP Float Cores/SM
- 5376 SP Float Cores
- 5376 SP Int Cores
- 2688 DP Float Cores

NVIDIA Volta Architecture (Tesla V100)

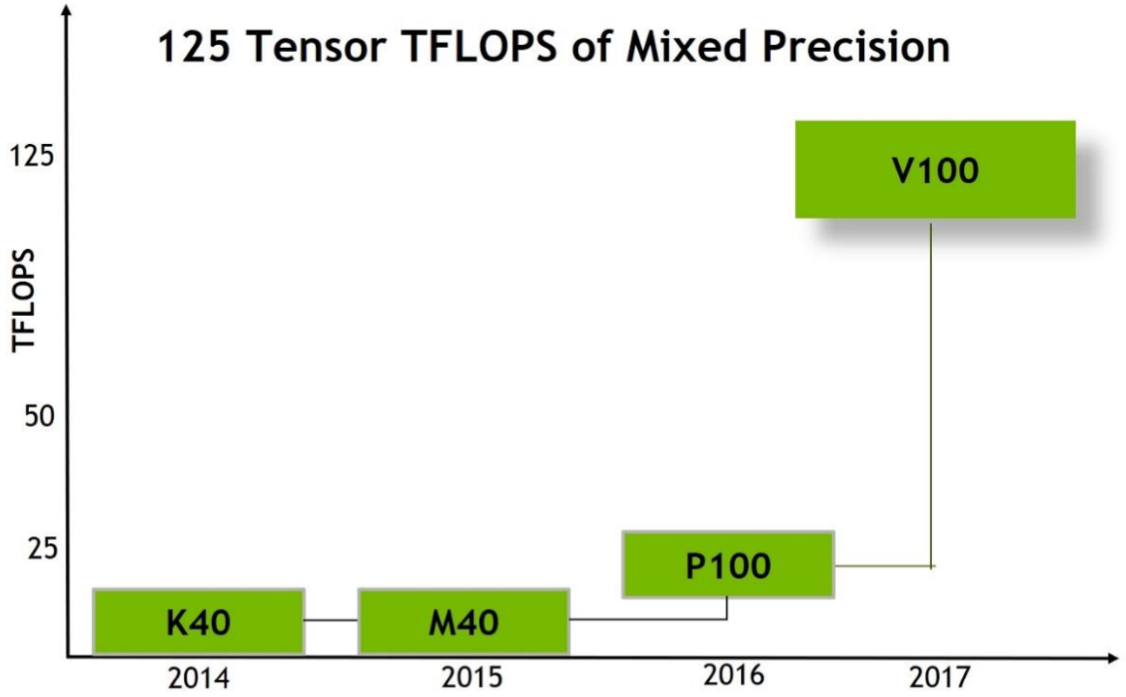
- Streaming Multiprocessors
  - 32 hardware threads Double Precision (DP: 64 bits)
  - or 64 hardware threads Single Precision (SP: 32 bits)
- DP FLOPS: 7,000 GFLOPS
- L2 size: 6MB
- High Bandwidth Memory
  - Size: 16 GB
  - Bandwidth: 900 GB/s



Volta GPU (2017) - Closeup

- Introduction of Tensor core: dedicated hardware for Deep Learning.
- Performs basic calculations for training and evaluation of neural networks
- Provide enormous speedups for AI neural network training and inferencing

Impact of Tensor Cores

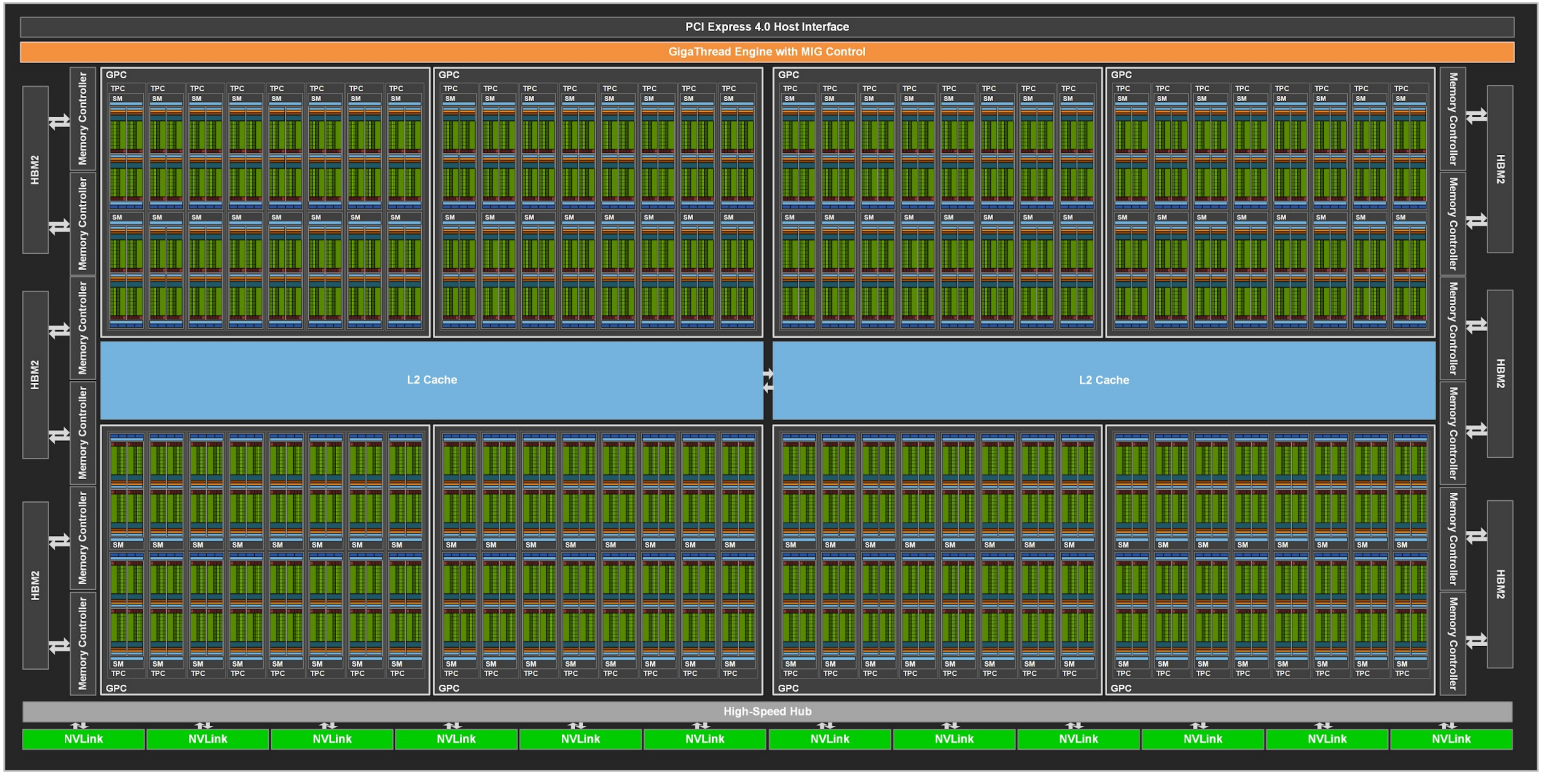


Tesla V100 provides a major leap in Deep Learning Performance with New Tensor Cores

Ampere GPU (2020) – Current Architecture

- SM
- 7 GPC
- 12 SM/GPC
- 84 SM
- 128 CUDA Cores/SM
- 28 Tensor Cores/SM
- 10752 CUDA Cores
- 336 Tensor Cores

NVIDIA A100 (2020)



108 cores on the A100 https://developer.nvidia.com/blog/nvidia-ampere-architecture-in-depth/

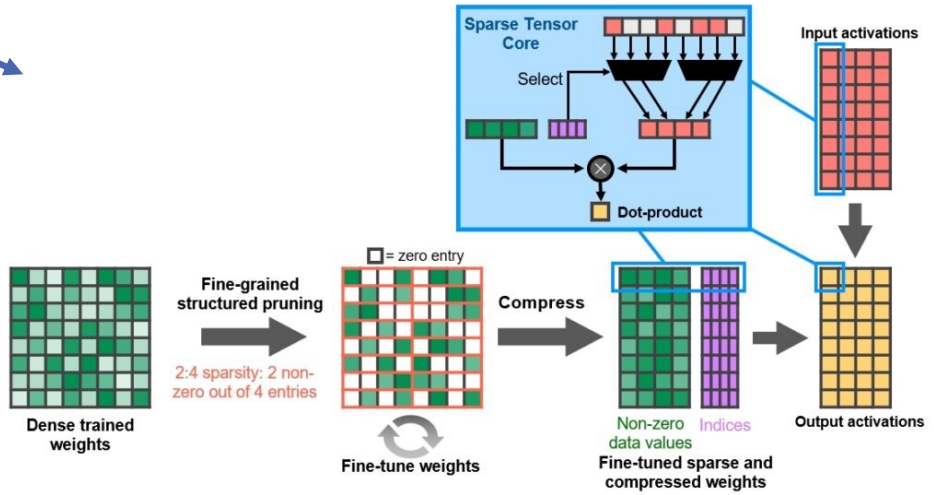
(Up to 128 cores in the full-blown chip)

40MB L2 cache

NVIDIA A100 Core

GPU compute throughput:

- 19.5 TFLOPS Single Precision
- 9.7 TFLOPS Double Precision
- 312 TFLOPS for Deep Learning (Tensor cores)



https://developer.nvidia.com/blog/nvidia-ampere-architecture-in-depth/

NVIDIA H100 Block Diagram

https://developer.nvidia.com/blog/nvidia-hopper-architecture-in-depth/

144 cores on the full GH100

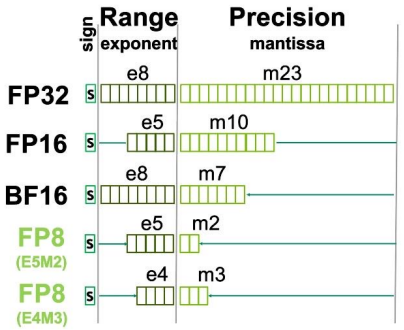
60MB L2 cache

NVIDIA H100 Core

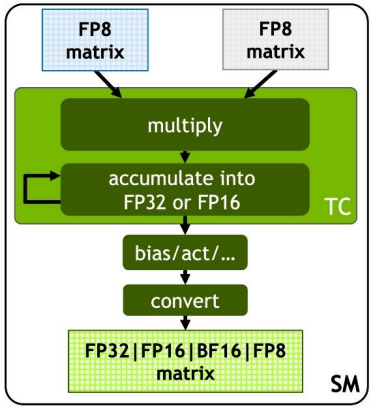
48 TFLOPS Single Precision*

24 TFLOPS Double Precision*

800 TFLOPS (FP16, Tensor Cores)*



Allocate 1 bit to either range or precision



Support for multiple accumulator and output types

https://developer.nvidia.com/blog/nvidia-hopper-architecture-in-depth/

* Preliminary performance estimates

NVIDIA A100 vs. H100

Nvidia Datacenter GPU	Nvidia A100 SXM	Nvidia H100 SXM	Nvidia H100 PCIe
GPU codename	GA100	GH100	GH100
GPU architecture	Ampere	Hopper	Hopper
GPU board form factor	SXM4	SXM5	PCIe Gen5
Launch date	May 2020	March 2022	March 2022
GPU process	TSMC 7nm N7	custom TSMC 4N	custom TSMC 4N
Die size	826mm2	814 mm2	814 mm2
Transistor Count	54 billion	80 billion	80 billion
FP64 CUDA cores	3,456	8,448	7,296
FP32 CUDA cores	6,912	16,896	14,592
Tensor Cores	432	528	456
Streaming Multiprocessors	108	132	114
Peak FP64	9.7 teraflops	30 teraflops	24 teraflops
Peak FP64 Tensor Core	19.5 teraflops	60 teraflops	48 teraflops
Peak FP32	19.5 teraflops	60 teraflops	48 teraflops
Peak FP32 Tensor Core	156 teraflops | 312 teraflops*	500 teraflops | 1,000 teraflops*	400 teraflops | 800 teraflops*
Peak BFLOAT16 Tensor Core	312 teraflops | 624 teraflops*	1,000 teraflops | 2,000 teraflops*	800 teraflops | 1,600 teraflops*
Peak FP16 Tensor Core	312 teraflops | 624 teraflops*	1,000 teraflops | 2,000 teraflops*	800 teraflops | 1,600 teraflops*
Peak FP8 Tensor Core	-	2,000 teraflops | 4,000 teraflops*	1,600 teraflops | 3,200 teraflops*
Peak INT8 Tensor Core	624 TOPS | 1,248 TOPS*	2,000 TOPS | 4,000 TOPS*	1,600 TOPS | 3,200 TOPS*
Peak INT4 Tensor Core	1,248 TOPS | 2,496 TOPS*	-	-
Interconnect	NVLink: 600GB/s PCI Gen4: 64GB/s	NVLink: 900GB/s PCI Gen5: 128GB/s	NVLink: 600GB/s PCI Gen5: 128GB/s
Max TDP	400 watts	700 watts	350 watts

*Effective TFLOPS or FLOPS using the Sparsity feature

Source: https://www.hpcwire.com/2022/03/22/nvidia-launches-hopper-h100-gpu-new-dgxs-and-grace-megachips/

Comparison of NVIDIA Tesla GPUS

- From: NVIDIA

Table 1. Comparison of NVIDIA Tesla GPUs

Tesla Product	Tesla K40	Tesla M40	Tesla P100	Tesla V100
GPU	GK180 (Kepler)	GM200 (Maxwell)	GP100 (Pascal)	GV100 (Volta)
SMs	15	24	56	80
TPCs	15	24	28	40
FP32 Cores / SM	192	128	64	64
FP32 Cores / GPU	2880	3072	3584	5120
FP64 Cores / SM	64	4	32	32
FP64 Cores / GPU	960	96	1792	2560
Tensor Cores / SM	NA	NA	NA	8
Tensor Cores / GPU	NA	NA	NA	640
GPU Boost Clock	810/875 MHz	1114 MHz	1480 MHz	1530 MHz
Peak FP32 TFLOPS[1]	5	6.8	10.6	15.7
Peak FP64 TFLOPS[1]	1.7	.21	5.3	7.8
Peak Tensor TFLOPS[1]	NA	NA	NA	125
Texture Units	240	192	224	320
Memory Interface	384-bit GDDR5	384-bit GDDR5	4096-bit HBM2	4096-bit HBM2
Memory Size	Up to 12 GB	Up to 24 GB	16 GB	16 GB
L2 Cache Size	1536 KB	3072 KB	4096 KB	6144 KB
Shared Memory Size / SM	16 KB/32 KB/48 KB	96 KB	64 KB	Configurable up to 96 KB
Register File Size / SM	256 KB	256 KB	256 KB	256KB
Register File Size / GPU	3840 KB	6144 KB	14336 KB	20480 KB
TDP	235 Watts	250 Watts	300 Watts	300 Watts
Transistors	7.1 billion	8 billion	15.3 billion	21.1 billion
GPU Die Size	551 mm²	601 mm²	610 mm²	815 mm²
Manufacturing Process	28 nm	28 nm	16 nm FinFET+	12 nm FFN

[1] Peak TFLOPS rates are based on GPU Boost Clock

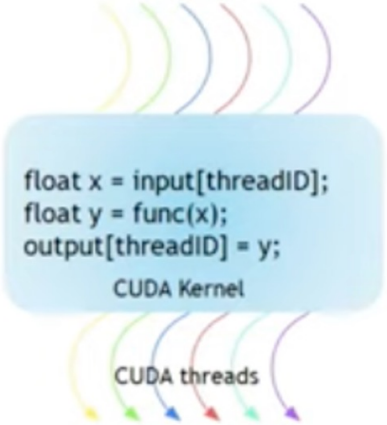
Summary/Comparisons

Year	μArch	SMs	SPs	Note
2008	Tesla	16	128	GeForce 8800
		30	240	GeForce 280
2010	Fermi	16	512	
2012	Kepler	15	2880	
			384	Pseudo Lab (Quadro K620)
2014	Maxwell	16	2048	
2016	Pascal	60	3840	
2017	Volta	84	5376	
2019	Turing	72	4608	
2020	Ampere	84	10752	

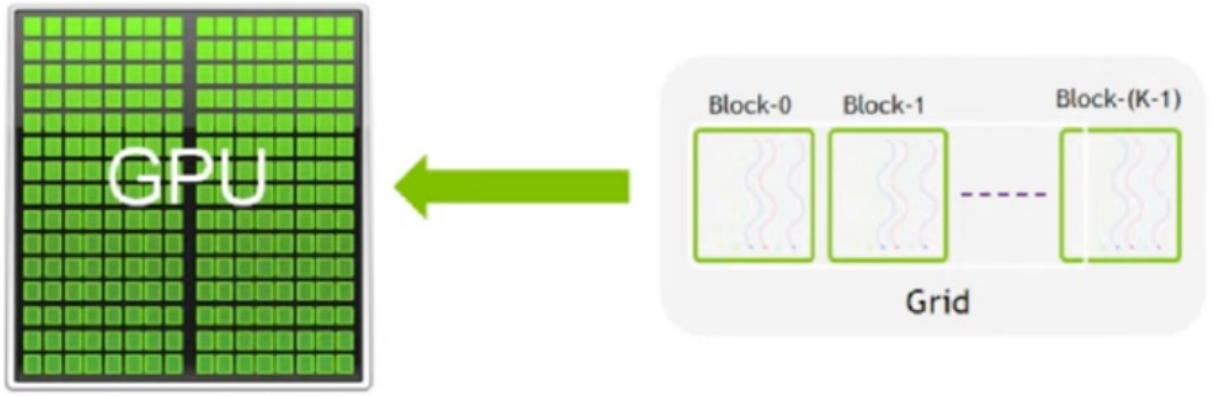
CUDA Programming Model

Part 2

Recap



- Kernel: fundamental unit of concurrency
- Executed K times in parallel
- On K different CUDA Threads

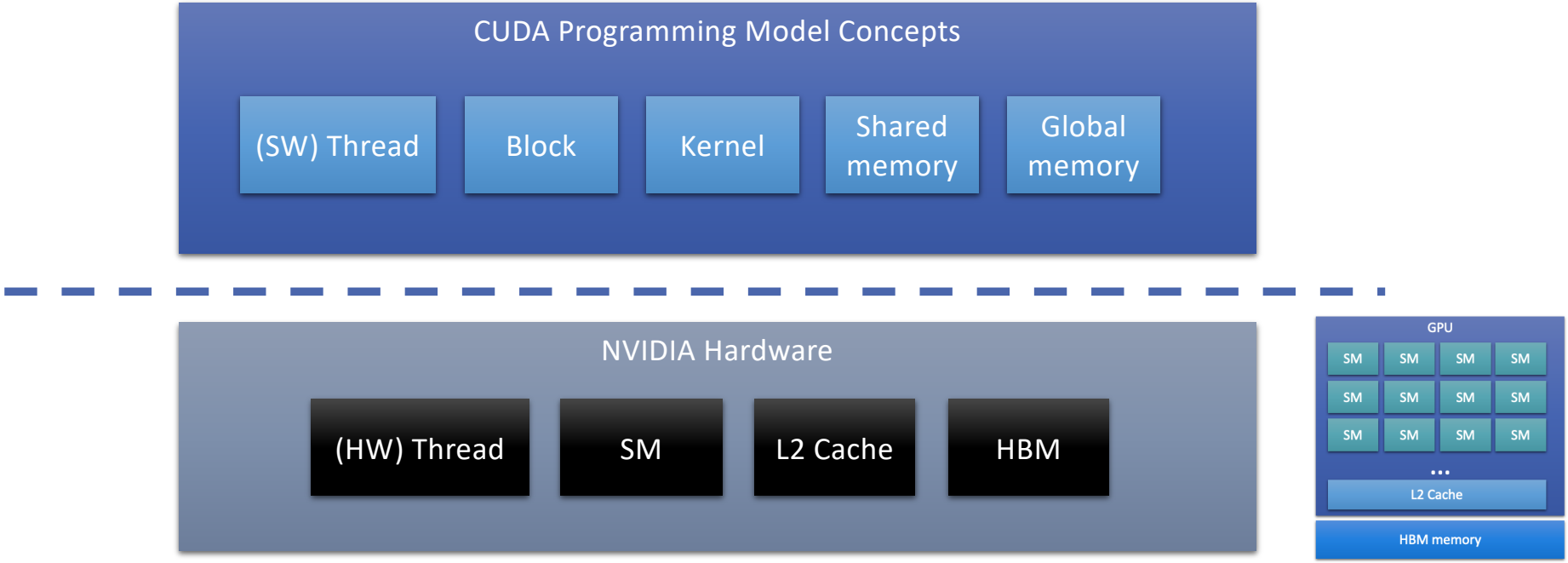


1. Kernel executes in a thread
2. Threads grouped into Thread Blocks (a.k.a. Blocks)
3. Blocks grouped into Grids

Kernel executed as a Grid of Blocks of Threads

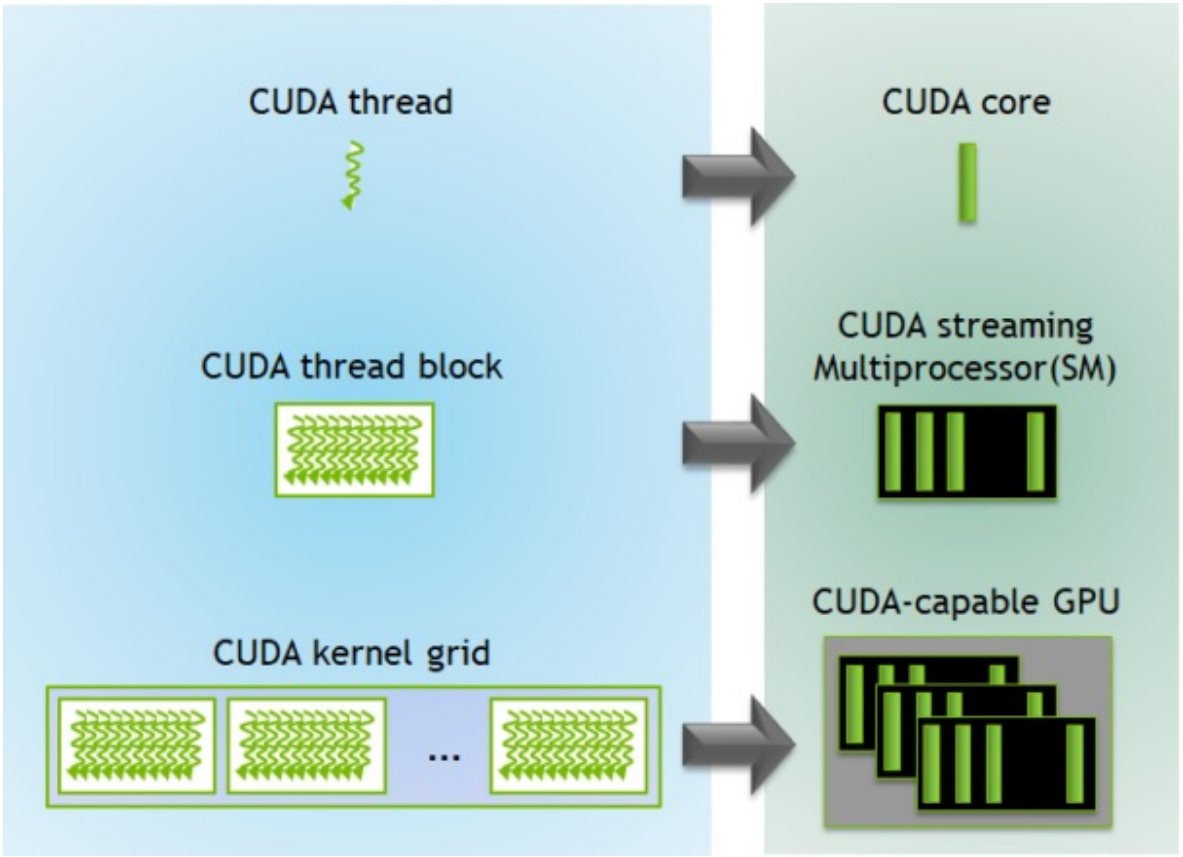
CUDA Programming model

- Programming model tries to **abstract** hardware architecture



HBM: High Bandwidth Memory
SM: Streaming Multiprocessor

Mapping to Hardware

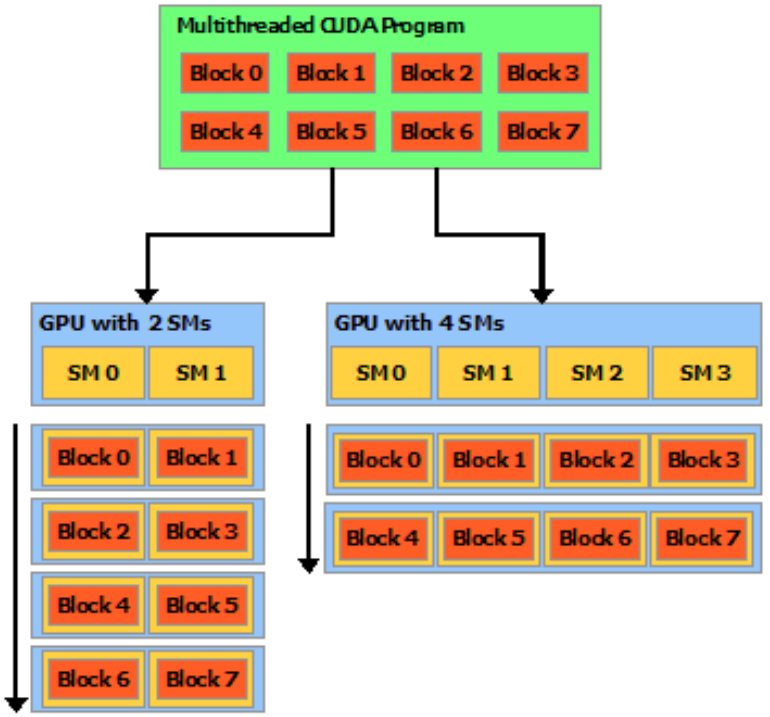


CUDA	Hardware
Thread	SP
Block	SM
Grid	GPU

https://developer.nvidia.com/blog/cuda-refresher-cuda-programming-model/

CUDA Programming Model – Definitions Recap

- ***Grid:***
  - *A group* of blocks executing a **kernel**
- ***Block***:
  - A group of threads that can be scheduled independently on a SM
  - Max threads in a block: 1024
  - Blocks can be executed in any order by the SMs
- ***Thread:*** a single context of execution



From: NVIDIA

Note on Scalability

- CUDA scales with the increasing number of cores (SPs in SMs)
- Program using CUDA language abstractions
- Divide into sub-problems, solved independently by thread blocks
- Each block solves one sub-problem in parallel threads (running a kernel)
- CUDA runtime schedules blocks on multiprocessors
- Scheduling can be done in any order
  - Allows the program to scale to any number of multiprocessors
  - Kernel code can’t rely on a specific sequencing

CUDA Programming Model - Definitions

- ***Kernel:***
  - C function that will execute in parallel on *N* hardware threads in the GPU
- ***(Software) Thread***:
  - Smallest unit of execution
  - ***threadIdx****:* 1,2, or 3 dimensions vector (x,y,z)
- These __global__ functions are known as *kernels*, and code that runs on the GPU is often called *device code*, while code that runs on the CPU is *host code*.

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

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

http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#programming-model

CUDA functions definition

	Executed on:	Callable from:
__device__ float DeviceFunc()	device	device
__global__ void KernelFunc()	device	host
__host__ float HostFunc()	host	host

- Keywords for CUDA functions:
  - **__global__** defines a kernel function
    - always returns **void**
  - **__device__** and **__host__** can be used together
  - **__host__** is optional if used alone

device functions

- __device__ functions can call other functions decorated with __device__
- Host code isn't allowed to call __device__ functions directly -- if we want access to the functionality in a __device__ function, we need to write a __global__ function to call it for us!
- You can also "overload" a function, e.g : you can declare void foo(void) and __device__ foo (void), then one is executed on the host and can only be called from a host function. The other is executed on the device and can only be called from a device or kernel function.

https://code.google.com/archive/p/stanford-cs193g-sp2010/wikis/TutorialDeviceFunctions.wiki

CUDA – Hello World

- Build instructions:
  - C code: main.c
  - CUDA code: **main.cu**
    - **nvcc** only parses .cu (or use –x)
  - First make sure to have nvcc and cuda libraries (module load…)

```
$ nvcc ./main.cu –o main
$ ./main
```

- C code

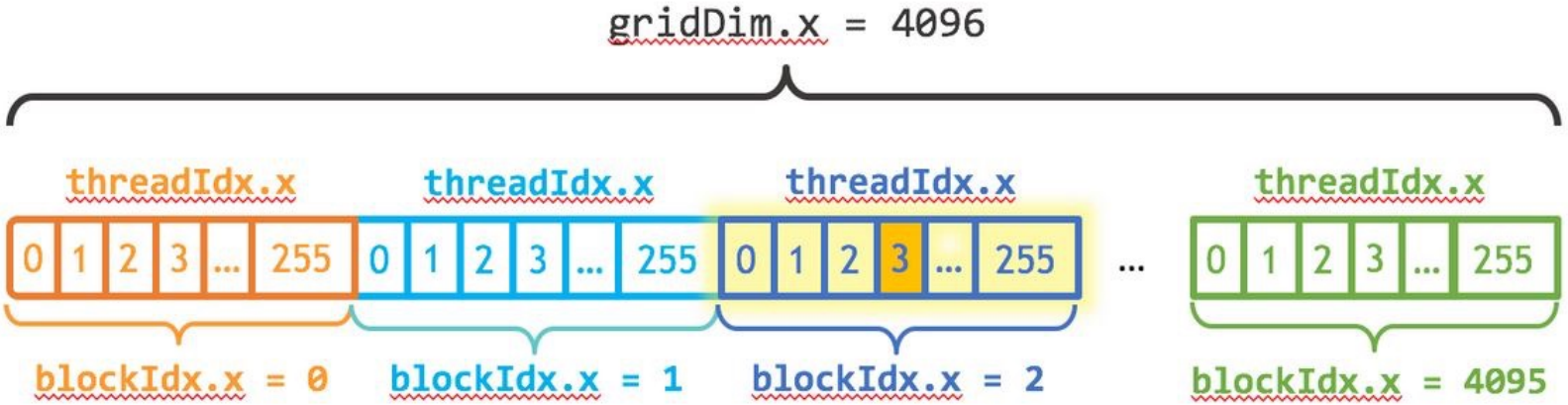
```
Int main() {
printf("Hello World!\n");
return 0;
}
```

- CUDA code

```
__global__ void mykernel(void) {
printf("Hello World!\n");
}
int main(void) {
mykernel<<<1,1>>>();
return 0;
}
```

CUDA – Grid Block and Thread

- One Grid per CUDA Kernel
- Multiple Blocks per Grid
- Multiple Threads per Block



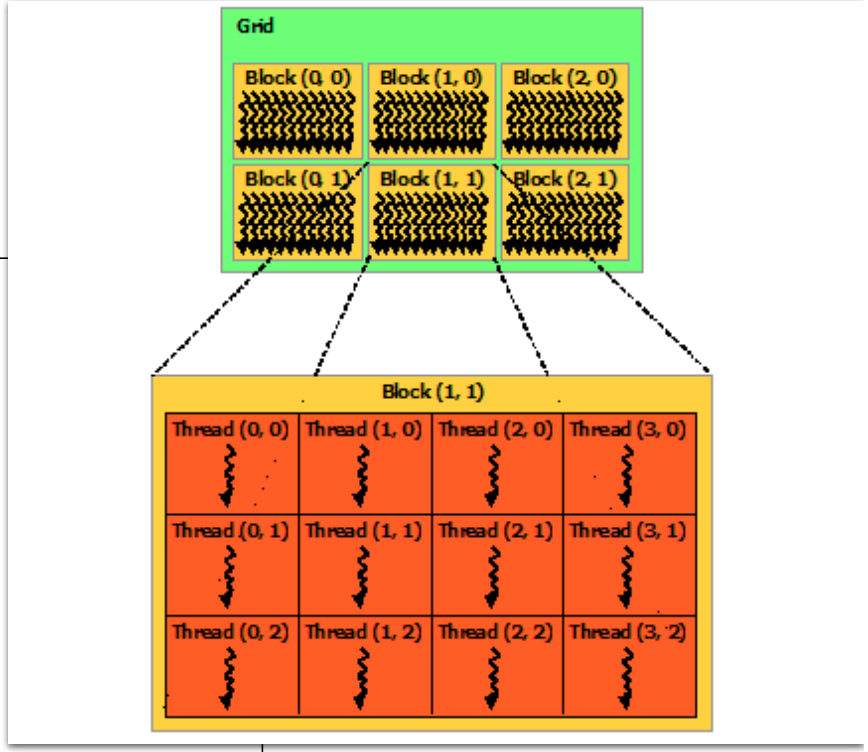

```
index = blockIdx.x  * blockDim.x + threadIdx.x
index =    (2)      *   (256)    +     (3)      = 515
```


From: NVIDIA

CUDA - Example

- Block and Threads index

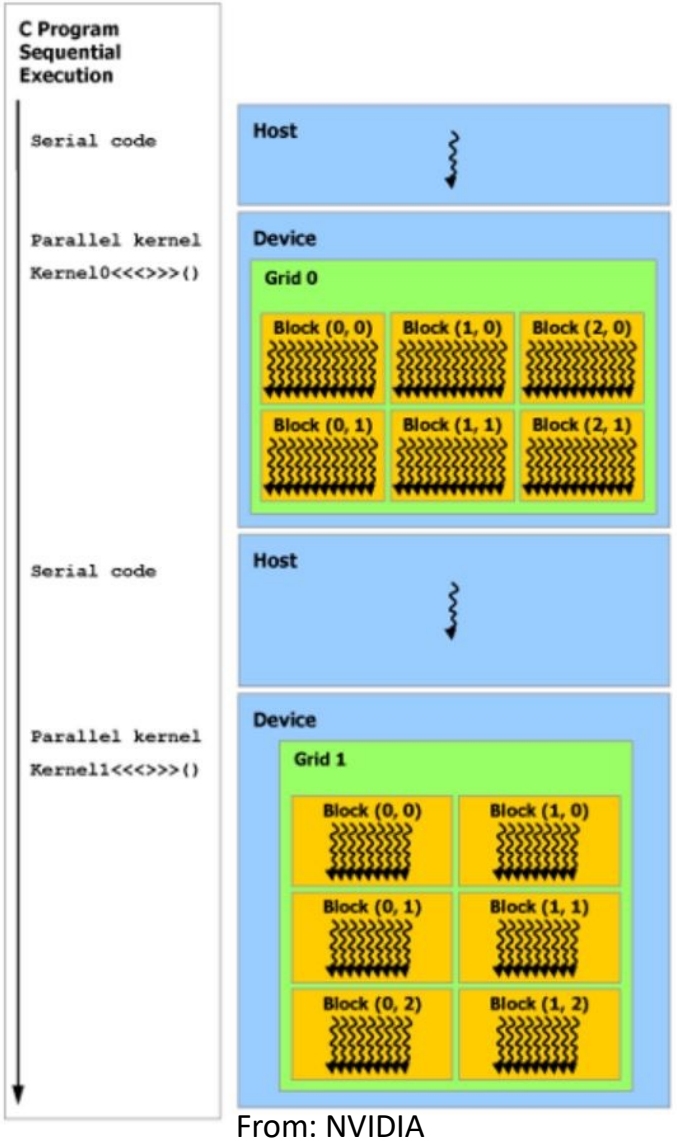
```
// Kernel definition
__global__ void MatAdd(float A[N][N], float B[N][N],
float C[N][N]) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;
    if (i < N && j < N)
        C[i][j] = A[i][j] + B[i][j];
}

int main() {
    ...
    // Kernel invocation
    dim3 threadsPerBlock(16, 16);
    dim3 numBlocks(N / threadsPerBlock.x, N / threadsPerBlock.y);
    MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
    ...
}
```



From: NVIDIA

CUDA – Serial and Parallel code

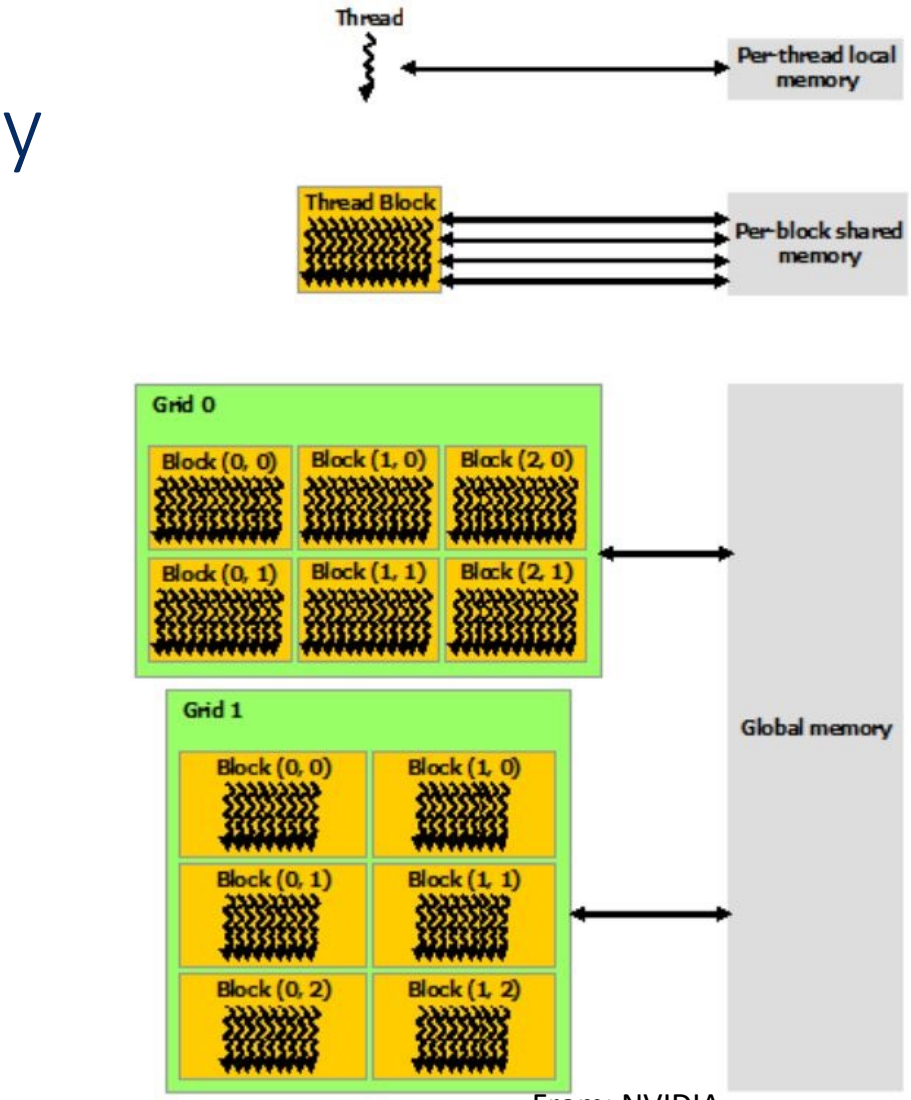
- A CUDA program starts as a sequential C/C++/Fortran process in the Host
- With a kernel launch:
  - Parallel code is executed inside the device (GPU)
  - SIMT Parallelism
    - Single Instruction Multiple Threads
- At the exit of a kernel launch the execution becomes sequential again



From: NVIDIA

CUDA Memory Hierarchy

- There are 3 levels of CUDA memory
- Per-thread local memory:
  - Available only to the thread
- Block-shared memory:
  - Shared by all the threads in a block
- Global memory:
  - Shared among all blocks and all grids



From: NVIDIA

C++ Example

- Sum two arrays in C++
- C++ code:
  - Allocate arrays
  - Add elements of two arrays
  - Check for errors

```
#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;
}
```

From: NVIDIA

CUDA Example

- Sum two arrays with C++ and CUDA

```
#include <iostream>
#include <math.h>
// Kernel function to add the elements of two arrays
__global__
void add(int n, float *x, float *y){
 int index = blockIdx.x * blockDim.x + threadIdx.x;
 int stride = blockDim.x * gridDim.x;
 for (int i = index; i < n; i += stride)
    y[i] = x[i] + y[i];
}
int main(void) {
  int N = 1<<20;
  size_t size = N*sizeof(float);
  float *x, *y;

  // Allocate input vectors h_A and h_B in host memory
  float* hx = (float*)malloc(size);
  float* hy = (float*)malloc(size);

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

```
  // Allocate vectors in device memory
  cudaMalloc(&x, size);
  cudaMalloc(&y, size);
  // Copy vectors from host memory to device global memory
  cudaMemcpy(x, hx, size, cudaMemcpyHostToDevice);
  cudaMemcpy(y, hy, size, cudaMemcpyHostToDevice);

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

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

  // Free memory
  cudaFree(x);  cudaFree(y);
  free(hx); free(hy);
  return 0;
}
```

From: NVIDIA

CUDA Vector Add – Vector size exceeds grid

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

__global__ void vector_add(int n, float *x, float *y){
 int index = blockIdx.x * blockDim.x + threadIdx.x;
 int stride = blockDim.x * gridDim.x;
 for (int i = index; i < n; i += stride)  // Needed !!!
   y[i] = x[i] + y[i];
}

int main(void) {
 int N = 1000;
 size_t size = N*sizeof(float);
 float* hx = (float*)malloc(size);      // cpu buffers
 float* hy = (float*)malloc(size);
 for (int i = 0; i < N; i++) {
   hx[i] = 1.0f;    hy[i] = 2.0f;
 }
```

```
float *x, *y;                  // gpu buffers
 cudaMalloc(&x, size);
 cudaMalloc(&y, size);
 cudaMemcpy(x, hx, size, cudaMemcpyHostToDevice);
 cudaMemcpy(y, hy, size, cudaMemcpyHostToDevice);

 int blockSize = 8;
 int numBlocks = 8;                  // only 64 'tasks'
 vector_add<<<numBlocks, blockSize>>>(N, x, y);
 cudaMemcpy(hy, y, size, cudaMemcpyDeviceToHost);

 float maxError = 0.0f;
 for (int i = 0; i < N; i++)
   maxError = fmax(maxError, fabs(hy[i]-3.0f));
 std::cout << "Max error: " << maxError << std::endl;

 cudaFree(x);  cudaFree(y);
 free(hx); free(hy);
 return 0;
}
```

CUDA Host and Device Memory

- In normal conditions Host and Device memory do not share an address space:
  - A pointer from *malloc()* or *cudaMallocHost()* (page-locked) cannot be read from GPU
  - A pointer from *cudaMalloc()* is on the device and cannot be hosted by the host
- Implications of not having a shared address space:
  - Explicit *cudaMemcpy()*
  - Pointers cannot be used: each data-structure using pointers needs to be serialized, copied and then recomposed

CUDA Unified Virtual Memory

- Unified Virtual Memory: Defines a *managed* memory space with **single coherent global address space**
  - Pointers work on CPU and GPU => No need for explicit *cudaMemcpy()*
  - Two ways: use ***__managed__*** keyword or use ***cudaMallocManaged()***

```
__device__ __managed__  int  ret[1000];
__global__ void AplusB(int a, int b) {
   ret[threadIdx.x] = a + b + threadIdx.x;
}
int main() {
   AplusB<<< 1, 1000 >>>(10, 100);
   cudaDeviceSynchronize();
   for(int i=0; i<1000; i++)
      printf("%d: A+B = %d\n", i, ret[i]);
   return  0;
}
```

```
__global__ void AplusB(int *ret, int a, int b) {
   ret[threadIdx.x] = a + b + threadIdx.x;
}
int main() {
   int *ret;
   cudaMallocManaged(&ret, 1000 * sizeof(int));
   AplusB<<< 1, 1000 >>>(ret, 10, 100);
   cudaDeviceSynchronize();
   for(int i=0; i<1000; i++)
      printf("%d: A+B = %d\n", i, ret[i]);
   cudaFree(ret);
   return  0;
}
```

cudaDeviceSynchronize()

- Need the CPU to wait until the kernel is done before it accesses the results (because CUDA kernel launches don't block the calling CPU thread).
- Call cudaDeviceSynchronize() before doing the final error checking on the CPU

CUDA Example with UVM

```
#include <iostream>
#include <math.h>
// Kernel function to add the elements of two arrays
__global__
void add(int n, float *x, float *y){
 int index = blockIdx.x * blockDim.x + threadIdx.x;
 int stride = blockDim.x * gridDim.x;
 for (int i = index; i < n; i += stride)
    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
 int blockSize = 256;
 int numBlocks = (N + blockSize - 1) / blockSize;
 add<<<numBlocks, blockSize>>>(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;
}
```

From: NVIDIA

CUDA Example Performance Results

- Latency and Bandwidth for the *add()* kernel that uses UVM

	Laptop (GeForce GT 750M)		Server (Tesla K80)	
Version	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

From: NVIDIA

- Results from: https://devblogs.nvidia.com/even-easier-introduction-cuda/

Lesson Key Points

- Heterogenous architectures motivations
- Hierarchy of Computations:
  - Threads
  - Blocks
  - Grids
- Corresponding Memory Spaces
  - Local
  - Shared
  - Global
- Synchronization Primitives
  - Implicit Barriers
  - Thread Synchronization
- NVIDIA GPUs and CUDA:
  - Compute capability
- CUDA Programming Model:
  - Grid, Block, Thread
  - UVM
- CUDA Hardware
- CUDA Warp Scheduling
- Context and Stream
- CUDA Memory Alignment and Coalescing
- CUDA Profiling and Debugging
- CUDA Compilation and Runtime:
  - CUDA Runtime, CUDA Driver, AoT and JIT compilation

Acknowledgments

Part of this material has been adapted from the “The GPU Teaching Kit” that is licensed by NVIDIA and the University of Illinois under the Creative Commons Attribution-NonCommercial 4.0 International License.

Part of the material has been adopted from Josh Holloway (CUDA Teaching Center), Prof. Dr. Juan Gómez Luna, and Prof. Onur Mutlu from ETH Zürich