ECE GY 9143
Introduction to High Performance Machine Learning

Parijat Dube

CUDA Basics

Lecture 8 03/23/23

Kaoutar El Maghraoui
Parijat Dube

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

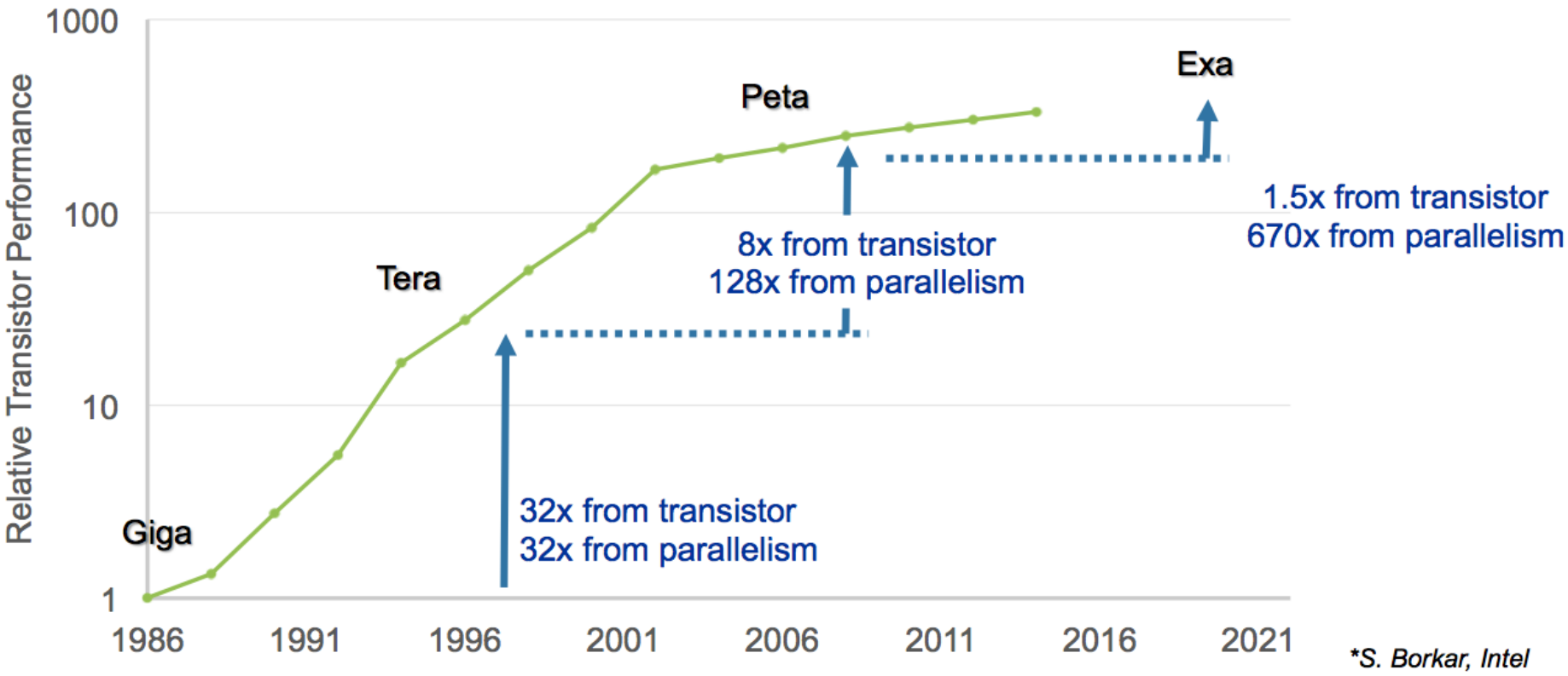
- CPU, DRAM, **GPU, HBM,** Tensor Units, Disk/Filesystem, Network

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

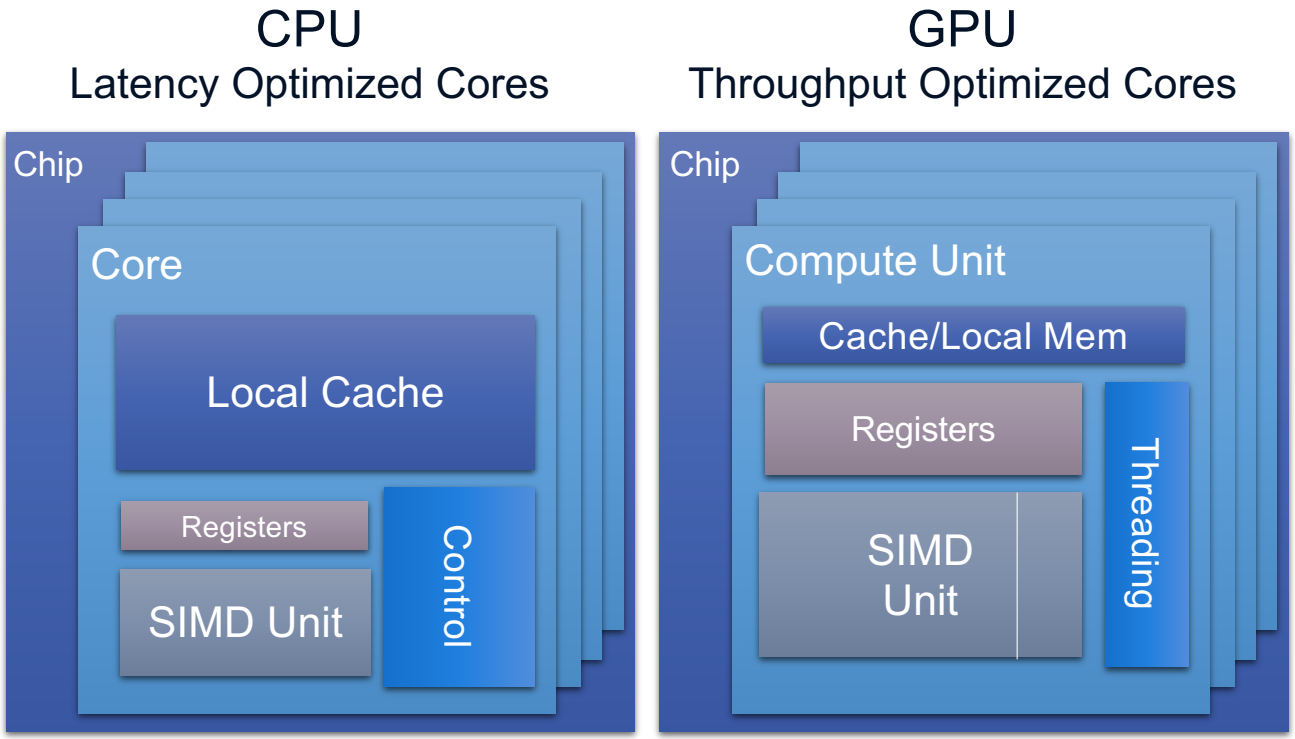
Heterogenous computing motivations

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



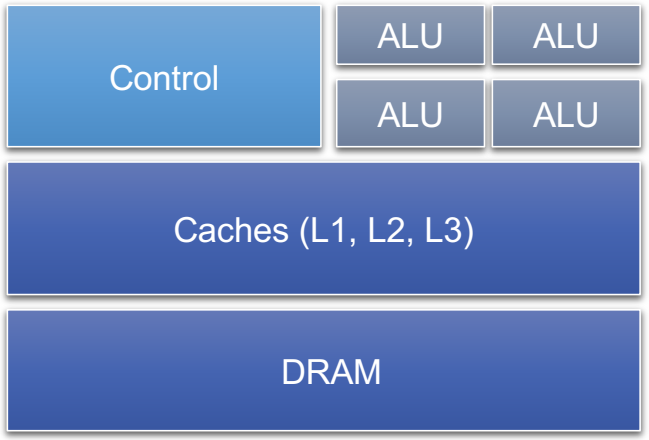
From: Nvidia – U Illinois

CPU: Latency Optimized

- 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

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

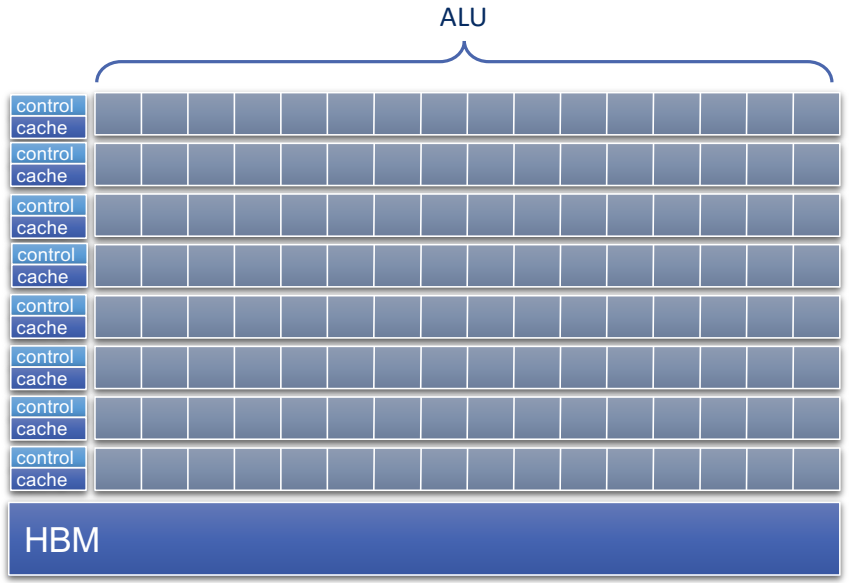
CPU Architecture



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

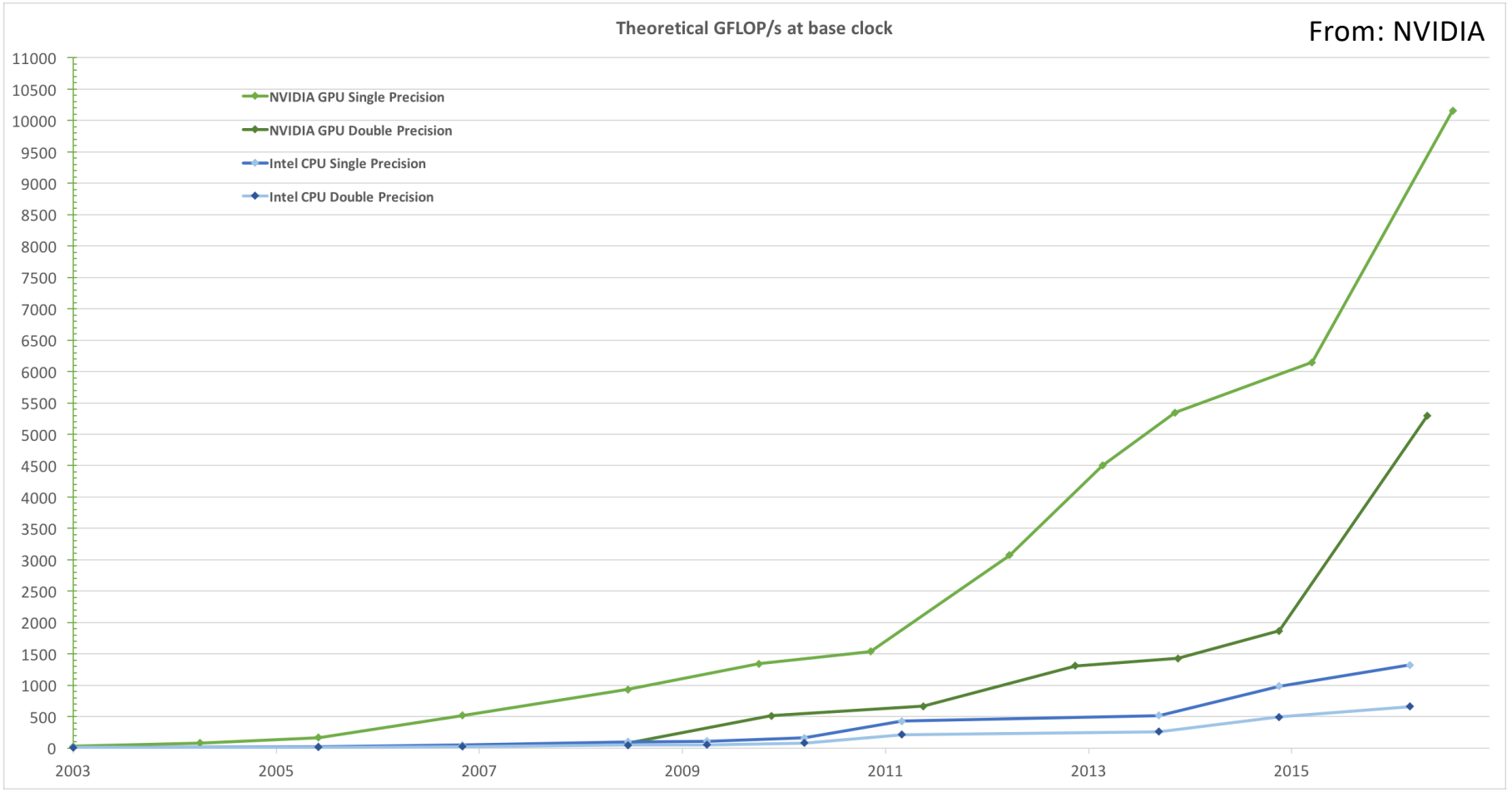
GPU Architecture



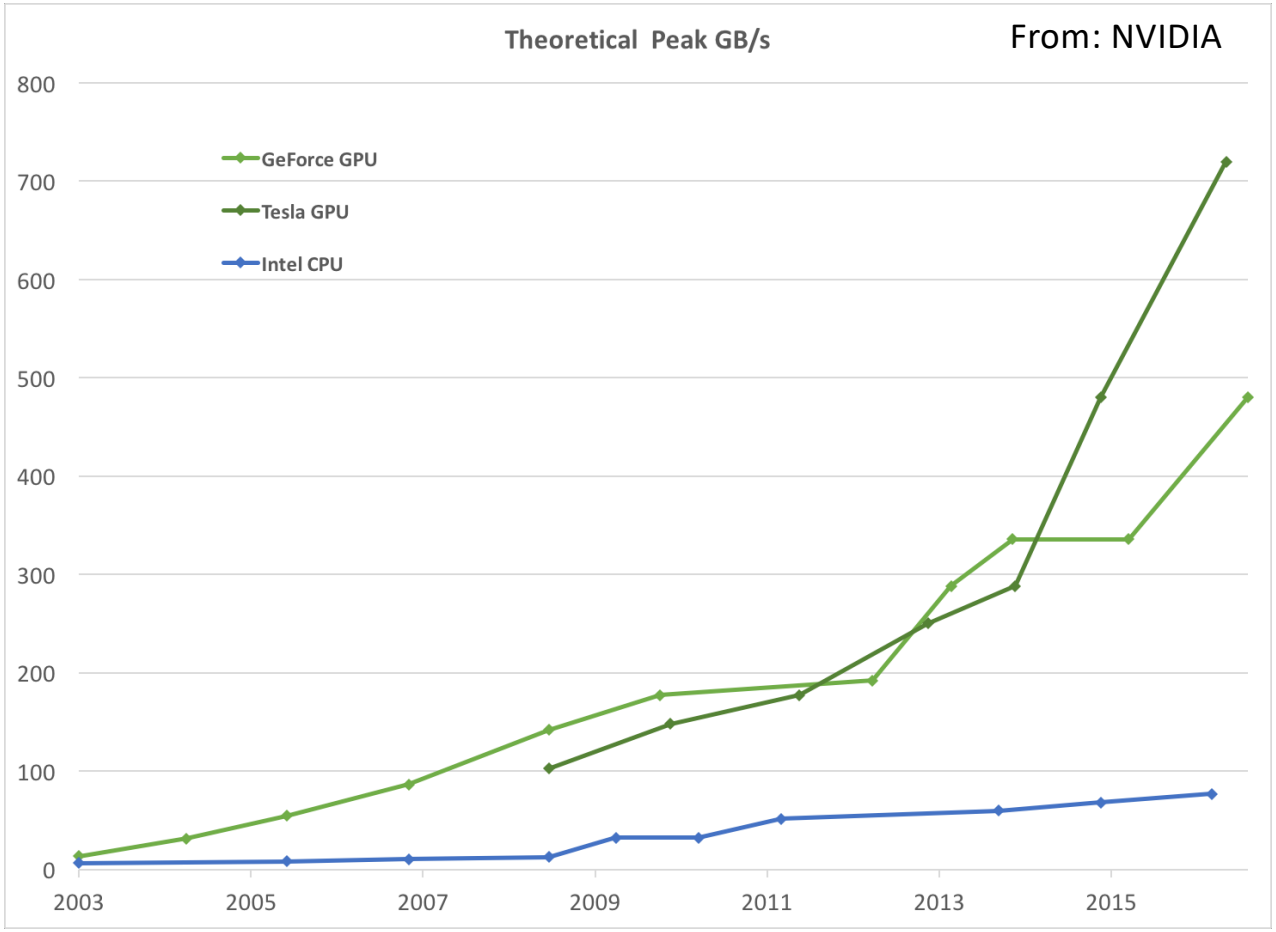
CPU/GPU Performance Comparison

- Sequential Code:
  - CPU about 10x faster than GPU
- 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

GPU Performance Advantage 1: FLOPS



GPU Performance Advantage 2: Memory BW

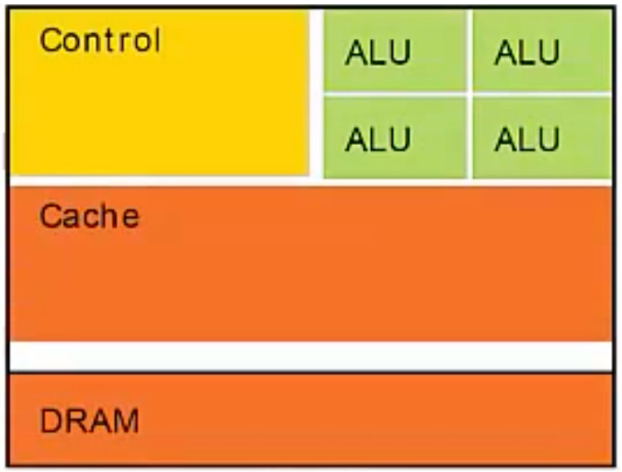


CPU vs. GPU Summary

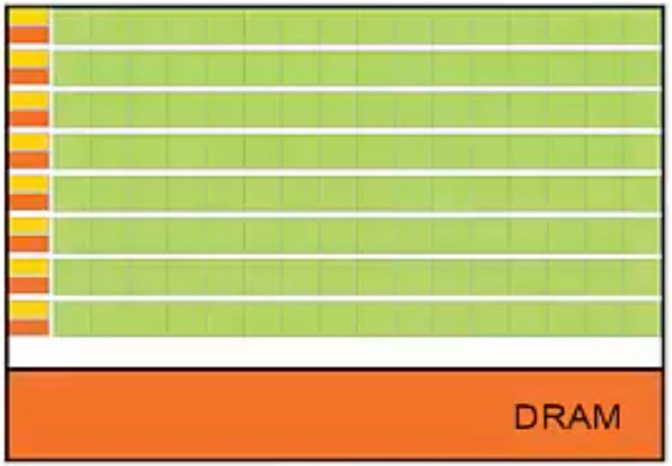
CPU	**GPU**
A smaller number of larger cores (up to 24)	A larger number (thousands) of smaller cores
Low latency	High throughput
Optimized for serial processing	Optimized for parallel processing
Designed for running complex programs	Designed for simple and repetitive calculations
Performs fewer instructions 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



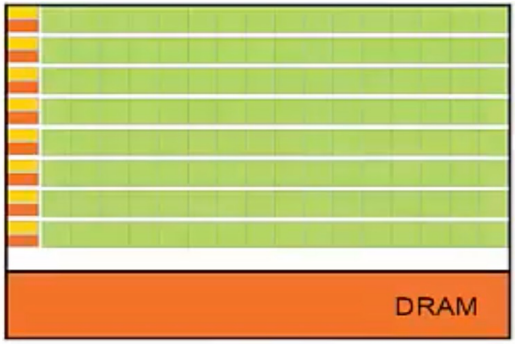
GPU

- CPU- Designed to minimize latency
  - Majority of the silicon area dedicated to:
    - Advanced Control Logic
    - Large Cache

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

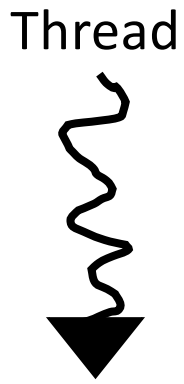
Key Concepts

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.

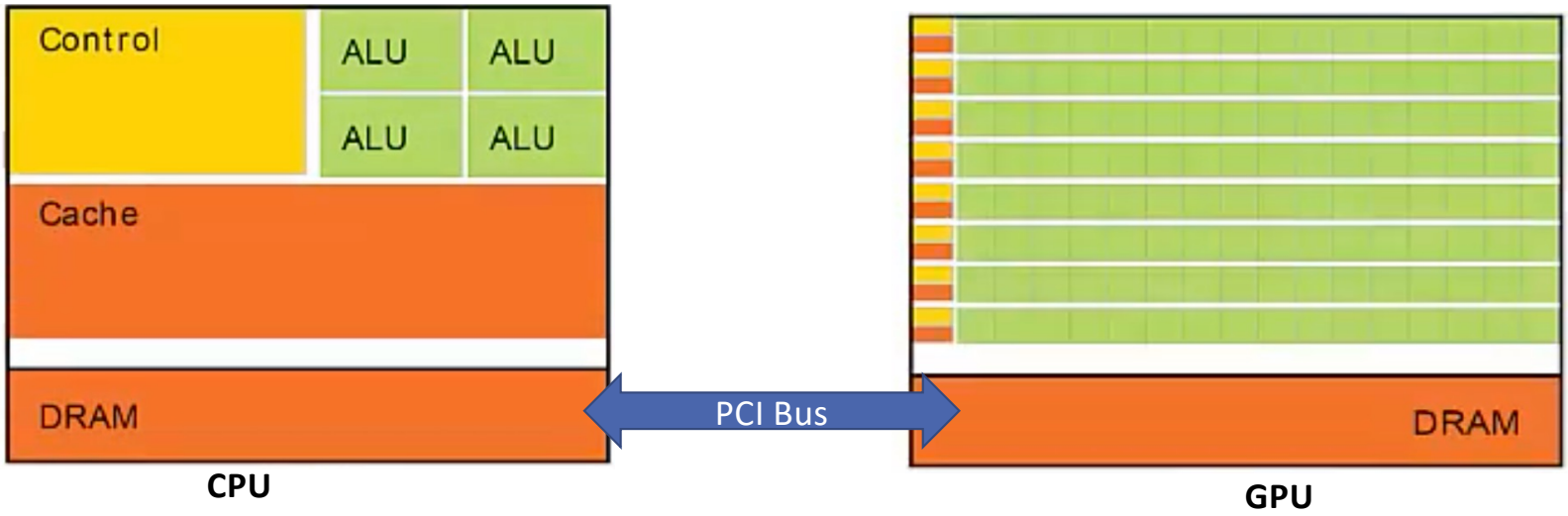


GPU

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



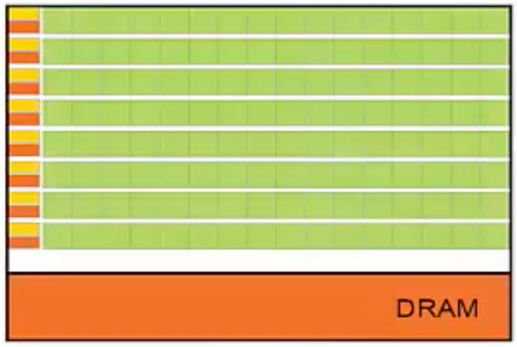
Key Concepts



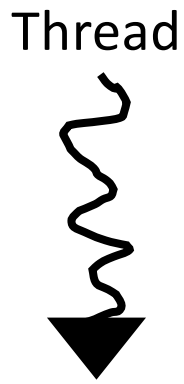
Host	Device	Heterogenous	CUDA
• CPU and its on chip memory	• GPU + dedicated DRAM	• Host + Device • Take the best of both worlds	• C with extensions • Allows for heterogenous programming

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)
- **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 taken in if/else statements
  - Different number of iterations in a loop, etc.



GPU



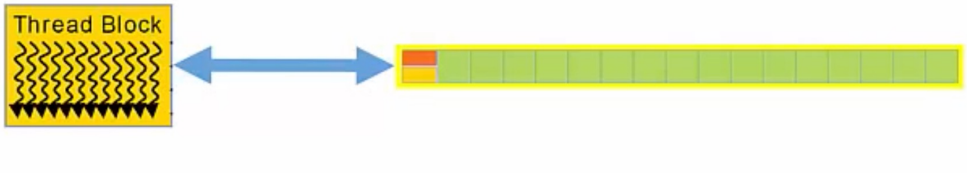
Threads Hierarchy

Threads

- Kernels execute as a set of Threads

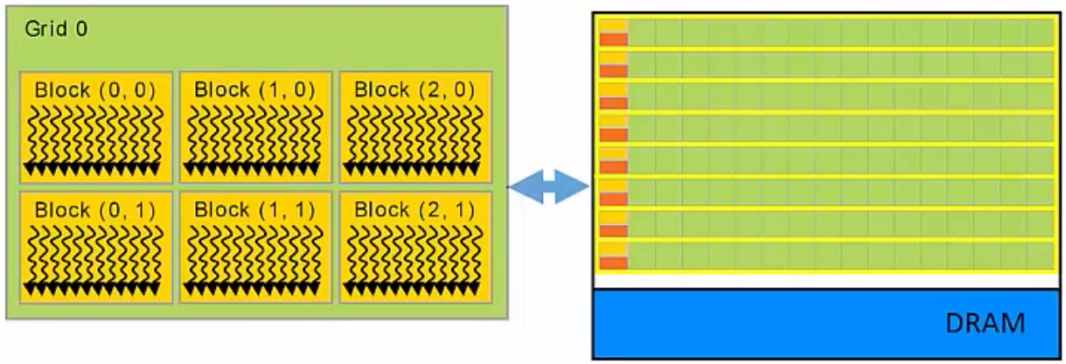
Blocks

- Threads are groups into blocks



Grid

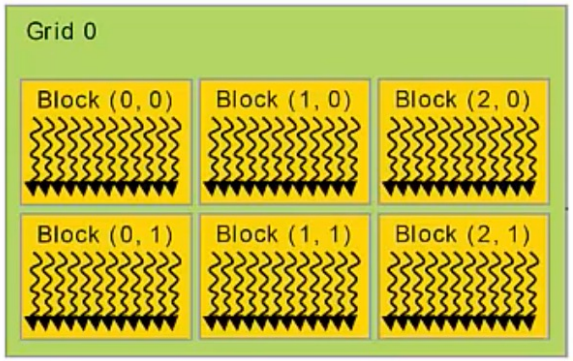
- Blocks are grouped into Grids
- **Each Kernel launch creates a single Grid**



Thread $\in$ Blocks $\in$ Grids

Dimensions of Gids and Blocks

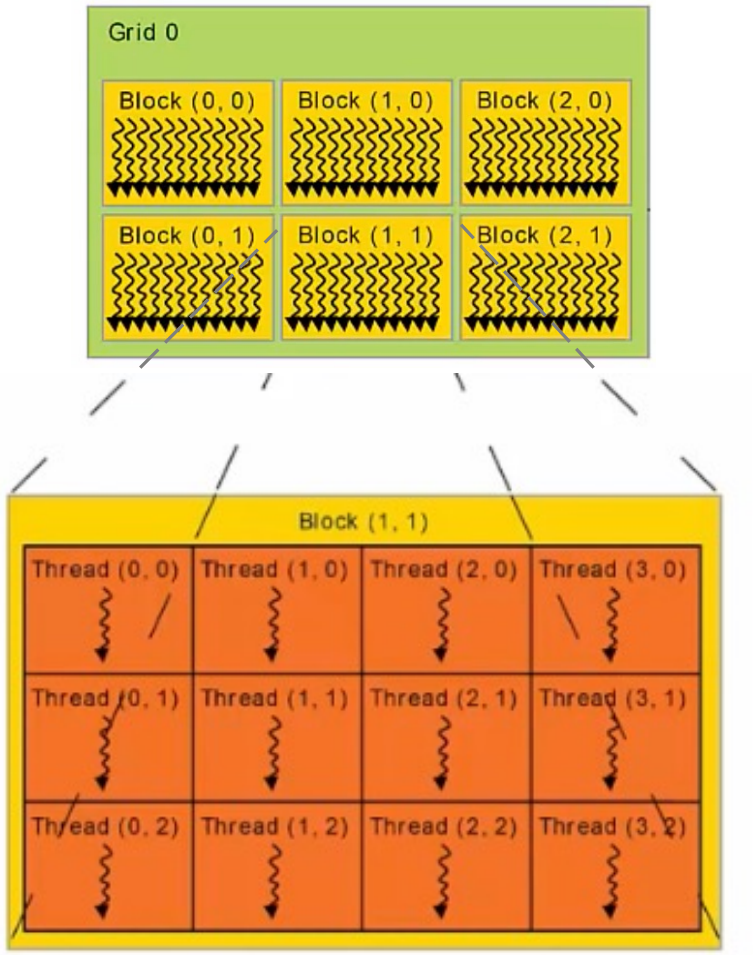
Grid dimension
- Block structure of each Grid
- 1D, 2D, or 3D



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

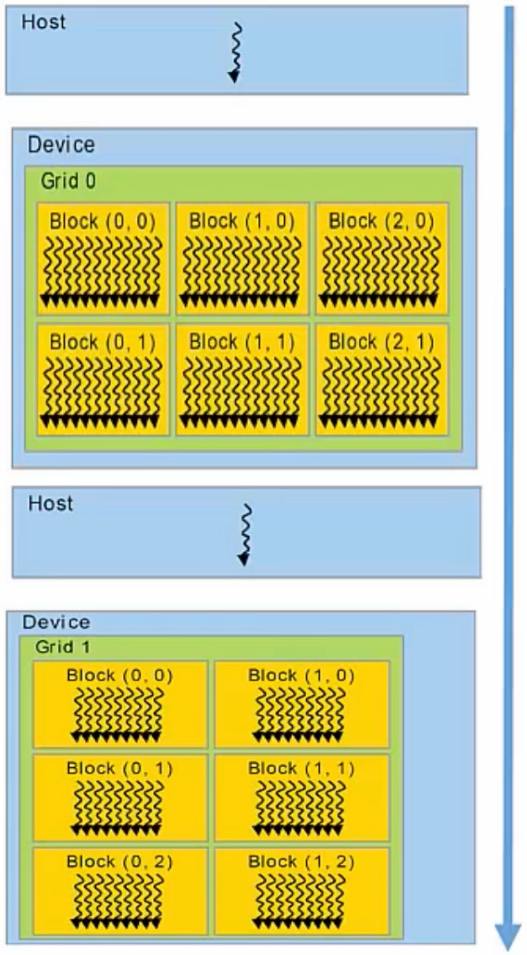
Block Dimension: 4x3
⇒ 4x3 = 12 Threads/Block
⇒ 6 Blocks x 12 Threads/Block
⇒ 72 Total Threads in Grid

Program Flow

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

- Device Code
  - Parallel
  - Runs on GPU

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

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

  // Do more sequential stuff

  // Launch Kernel
  kernel_1<<< grid_sz1,blk_sz1 >>>(...);

} return 0;
```



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>>>
```

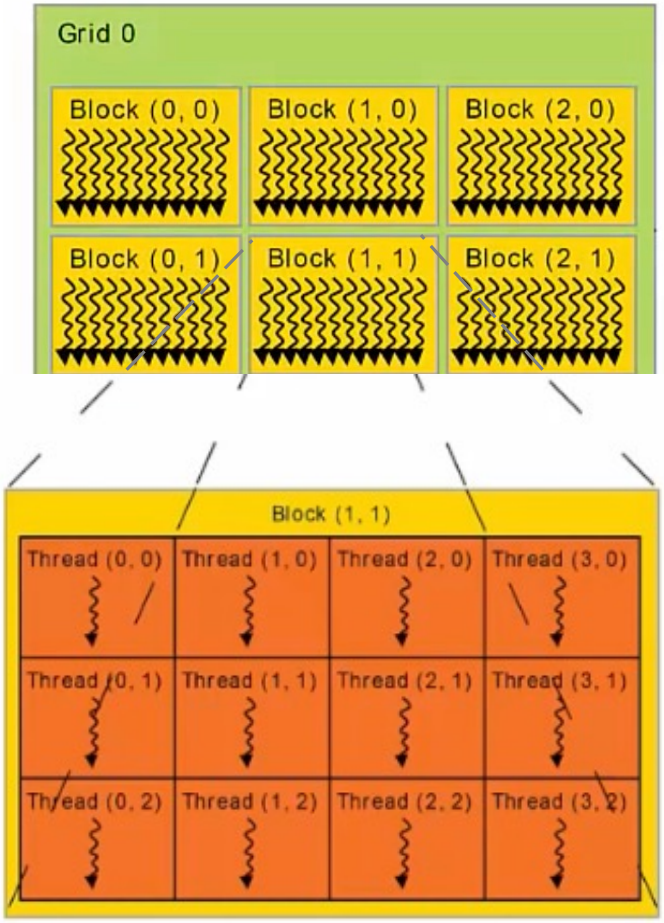
- dim3 is a CUDA data structure
- Default values are (1,1,1)

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

Kernel Launch Configuration Example

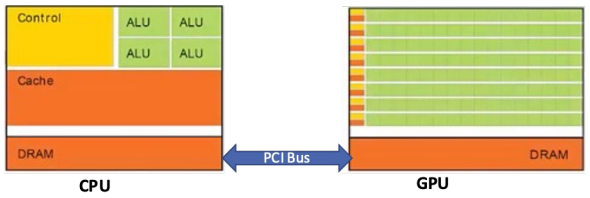
```
// Block and Grid dimensions
// a.k.a. Configuration Parameters
dim3 grid_size(3,2);

dim3 block_size(4,3);
```

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

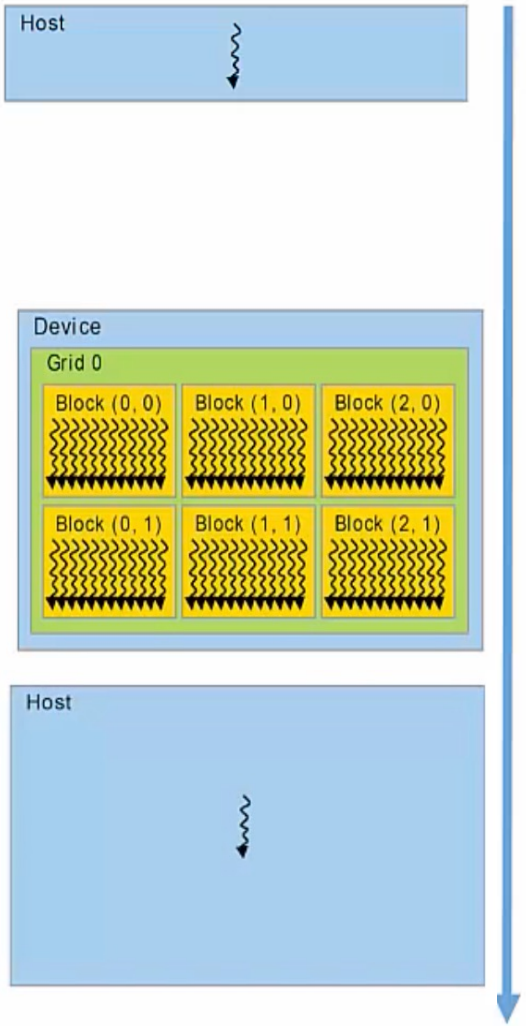


Closer look at Program Flow

- Host Code
  - Do sequential stuff
  - 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

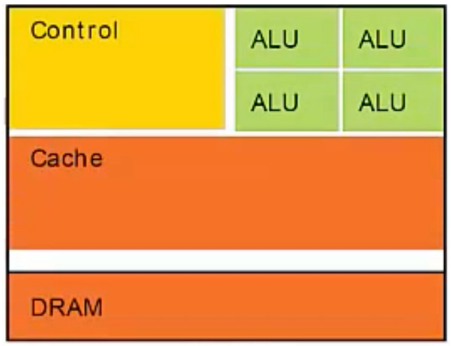



```
int main( void ) { // Host Code
  // Do sequential stuff

  // 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;
}
```

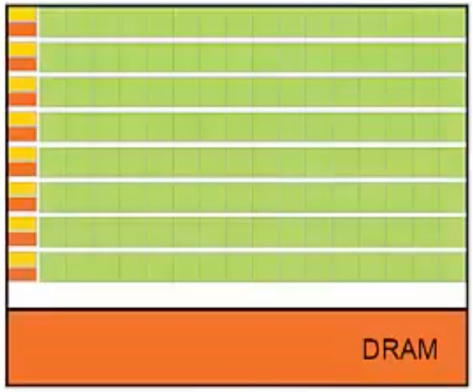


Allocation Device Memory

- 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

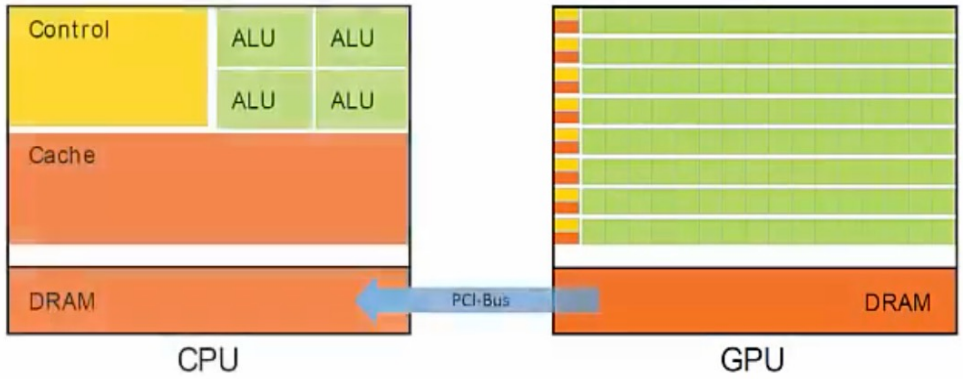


GPU

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

Convention

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

Example

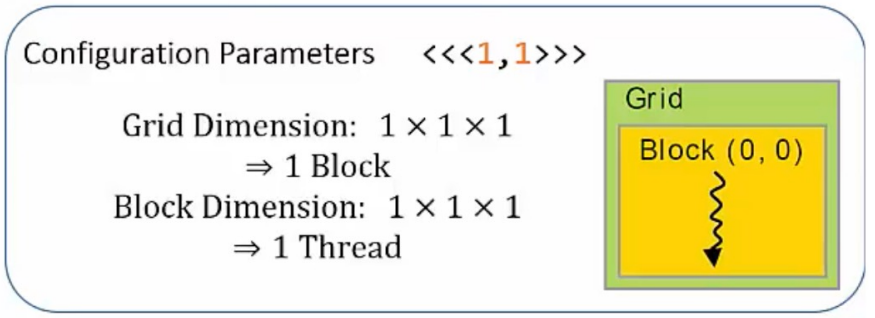
```
int main( void ) {

    // 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

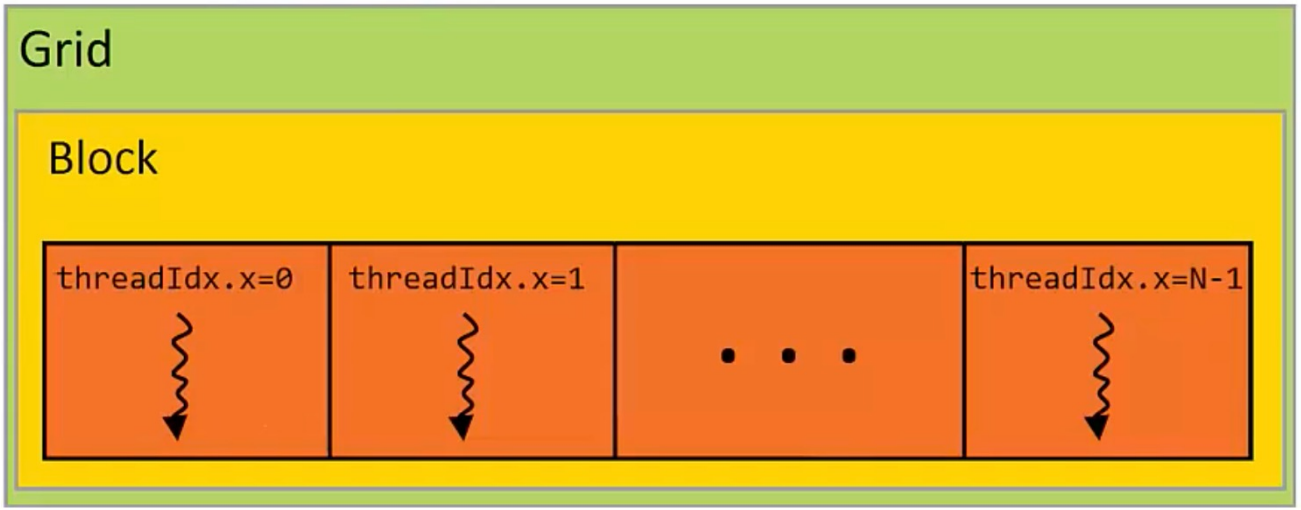
- Each Thread has its own thread index
  - Accessible within a Kernel through the built in threadIdx variable
- Thread Blocks can have as many as 3-dimensions, therefore there is a corresponding index for each dimension:

```
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;
}
```

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;
}
```

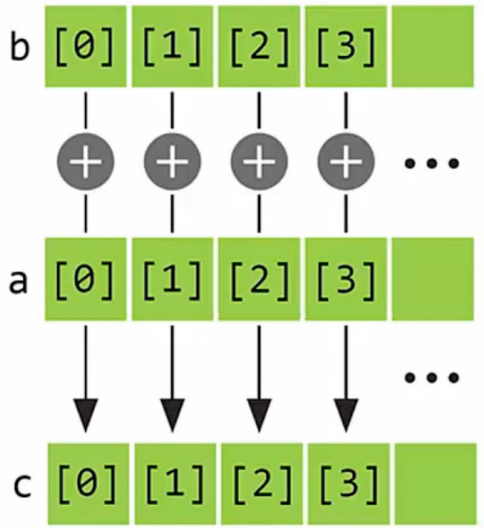
If (i < N)
Ensures that the kernel does not execute more Threads than the length of the array.

Vector Addition – A Very Parallel Problem

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

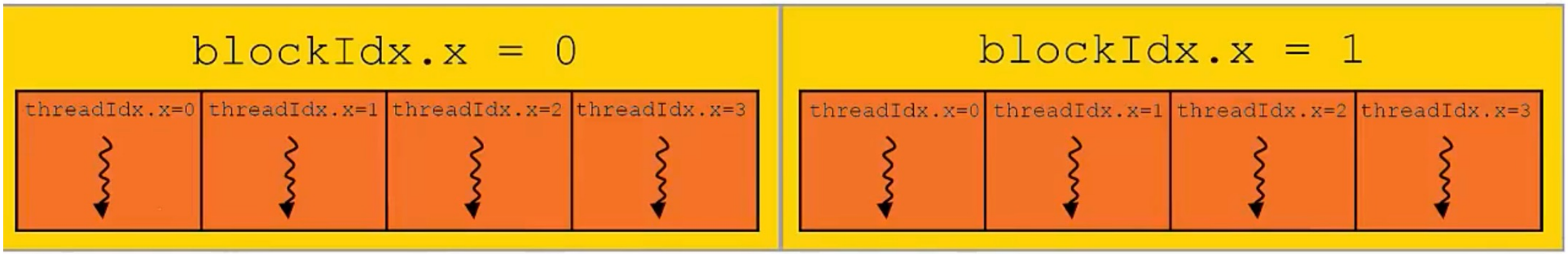
$$\boldsymbol{a} + \boldsymbol{b} = \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \\ \vdots \\ a_{N-1} \end{bmatrix} + \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \\ \vdots \\ b_{N-1} \end{bmatrix}$$

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

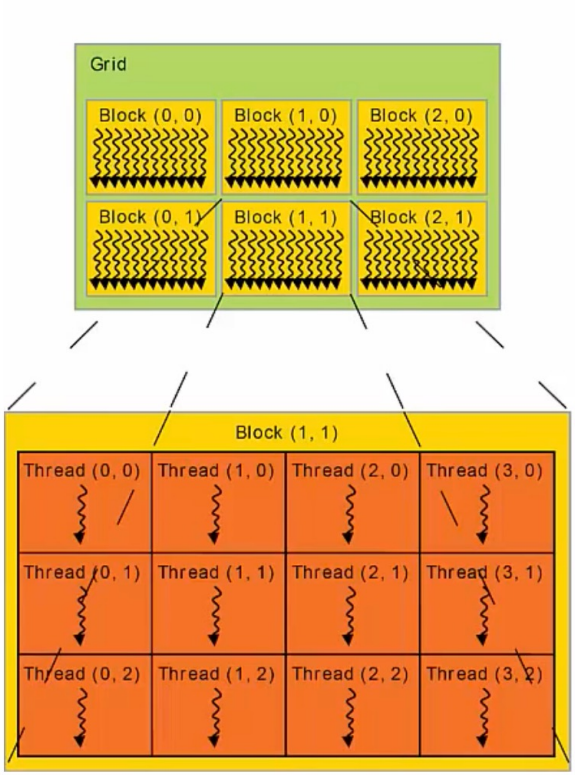


Thread Index

- 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;
```

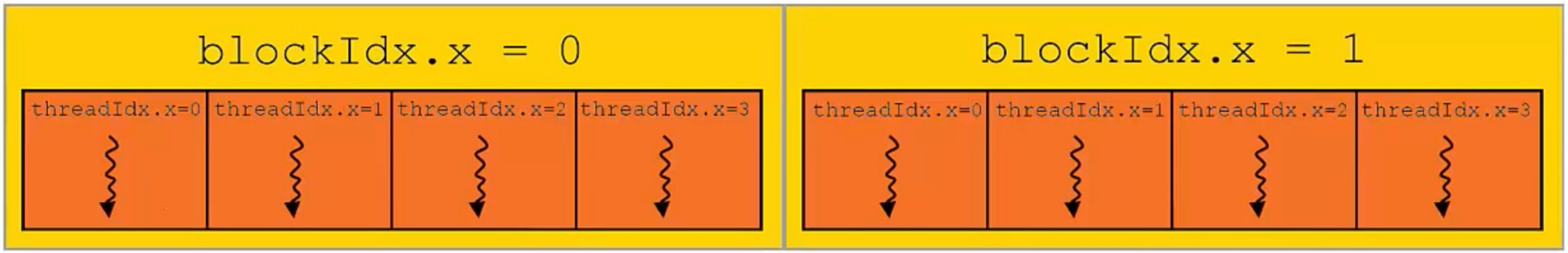
Index of a Thread

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

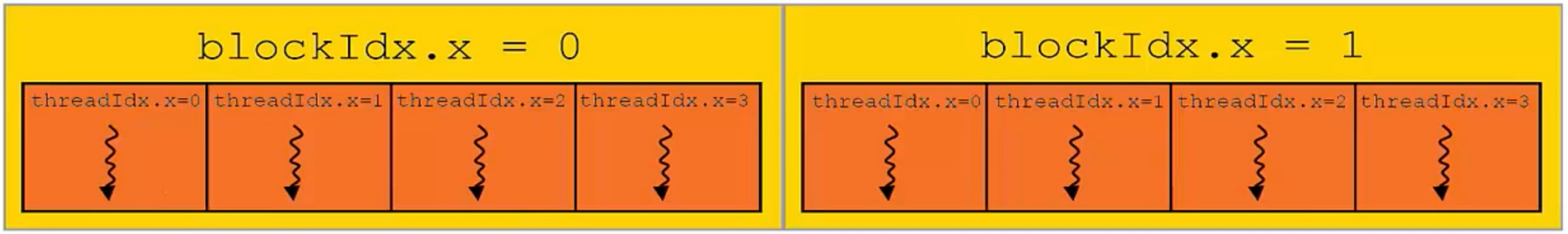
Indexing Within Grid

- `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

Examples

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

```
__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;
}
```

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

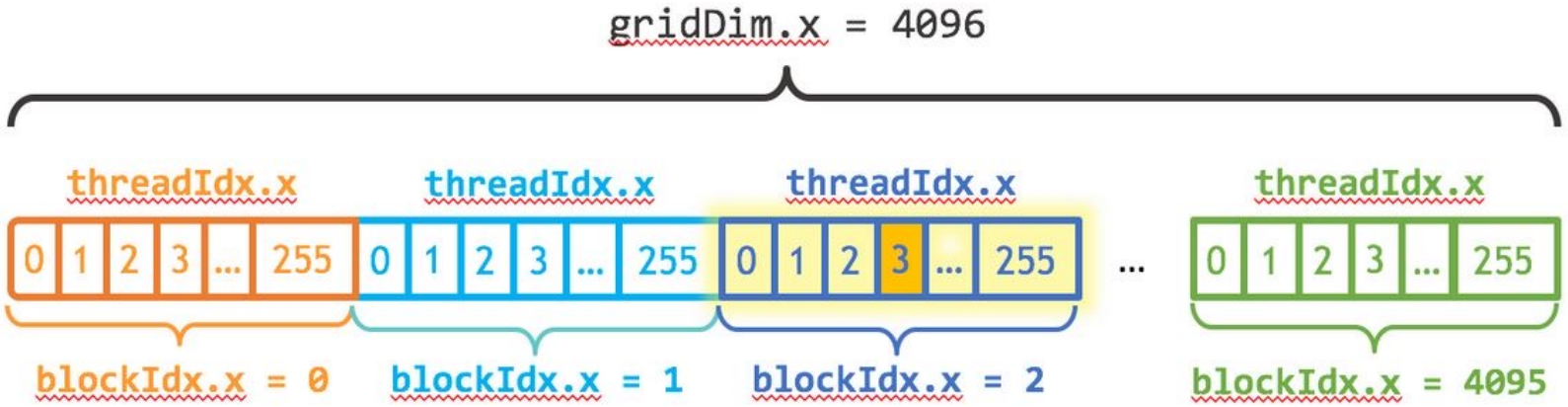
`__global__ void kernel( int* a ) {`
`  int i = threadIdx.x + blockIdx.x * blockDim.x;`
`  a[i] = threadIdx.x;`
`}`
	a: 0 1 2 3 0 1 2 3 0 1 2 3

`__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

CUDA Memory Model

Thread Memory Correspondence

Threads ⇔ Local Memory (and Registers)

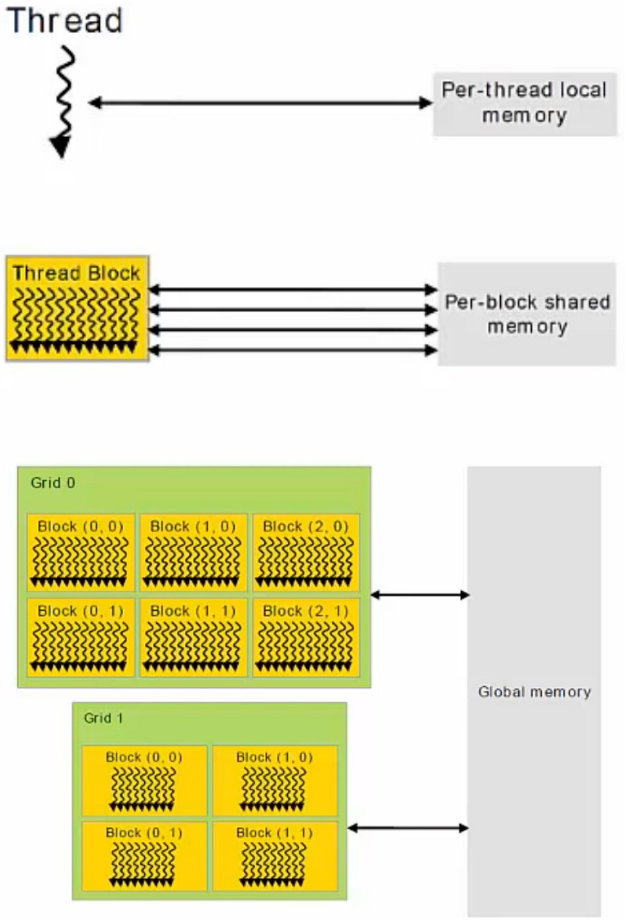
- Scope: Private to its corresponding Thread
- Lifetime: Thread

Blocks ⇔ Shared Memory

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

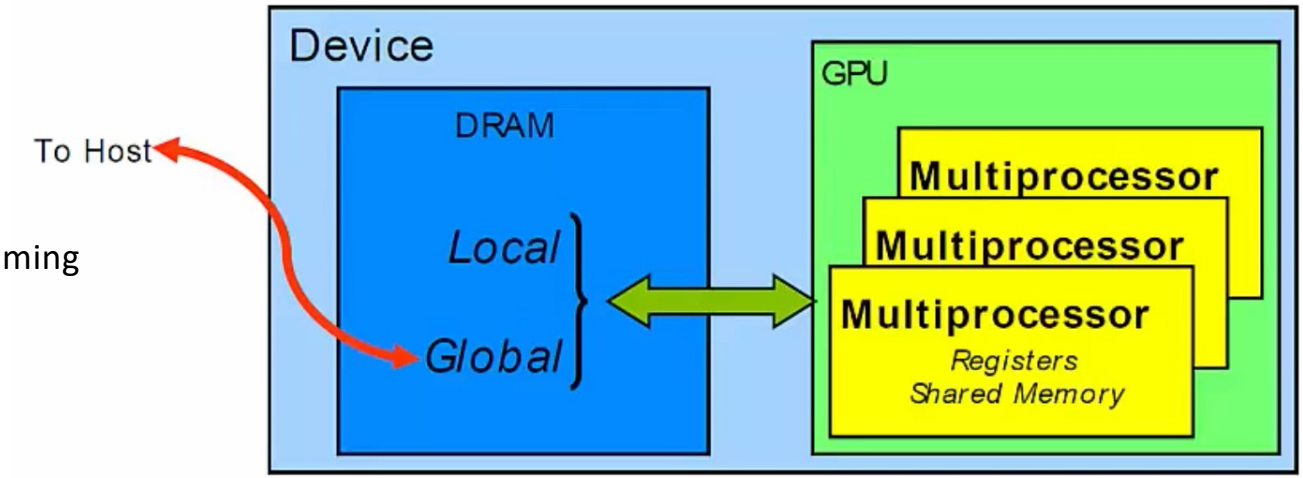
Grids ⇔ Global Memory

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

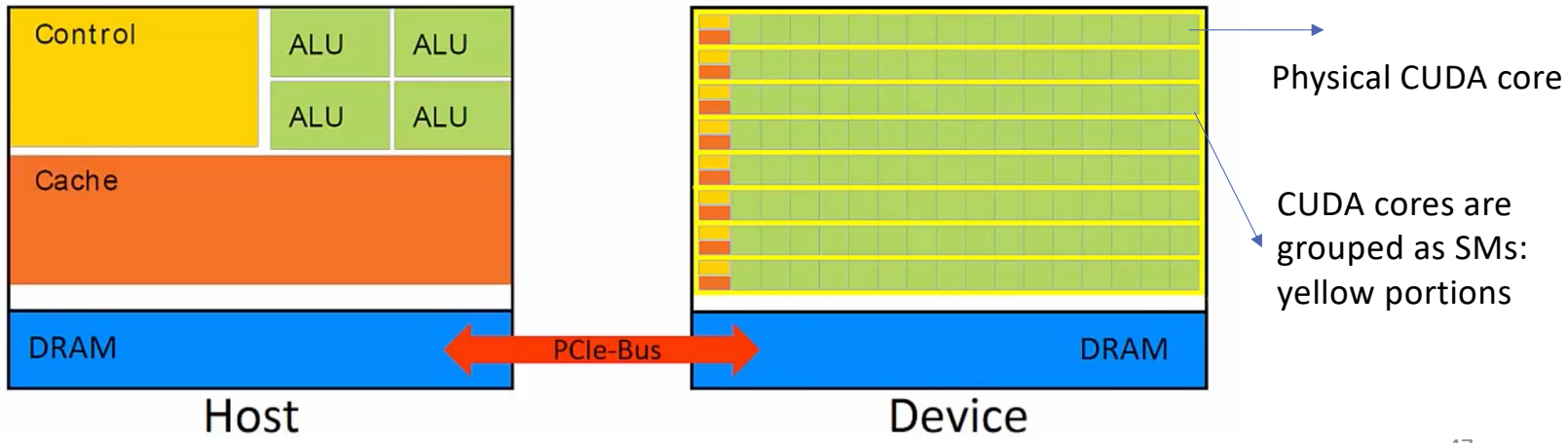


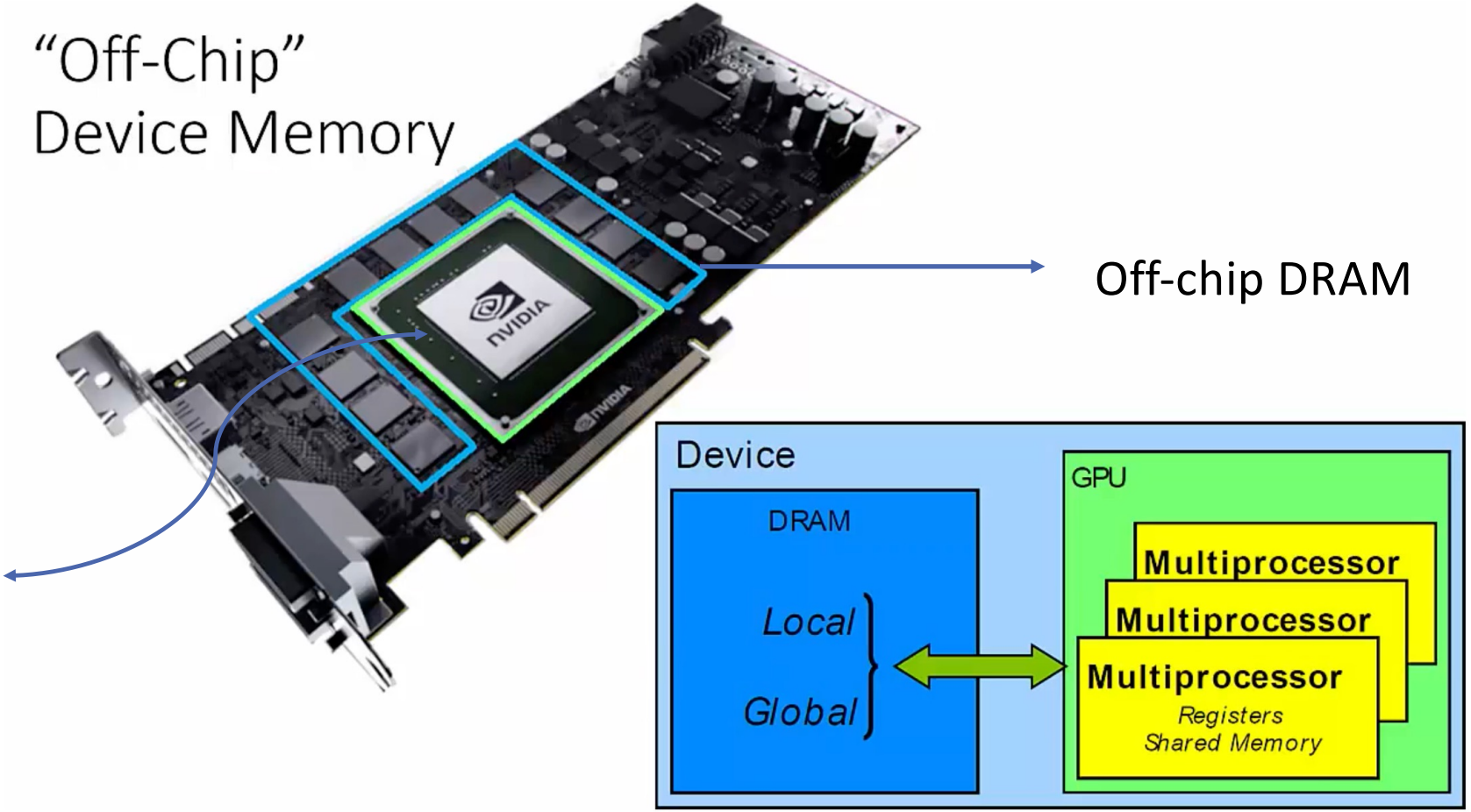
Memory Model

- 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

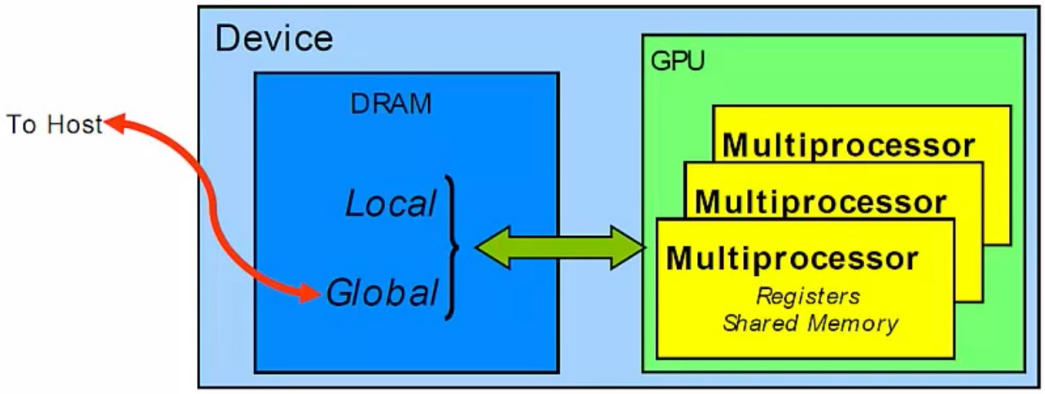


Term **local** refers to the scope of lifetime and not physical location




"Off-Chip"
Device Memory
Off-chip DRAM
GPU
Device
DRAM
Local
Global
GPU
Multiprocessor
Multiprocessor
Multiprocessor
Registers
Shared Memory
Off-chip DRAM
On-chip

Memory Speed



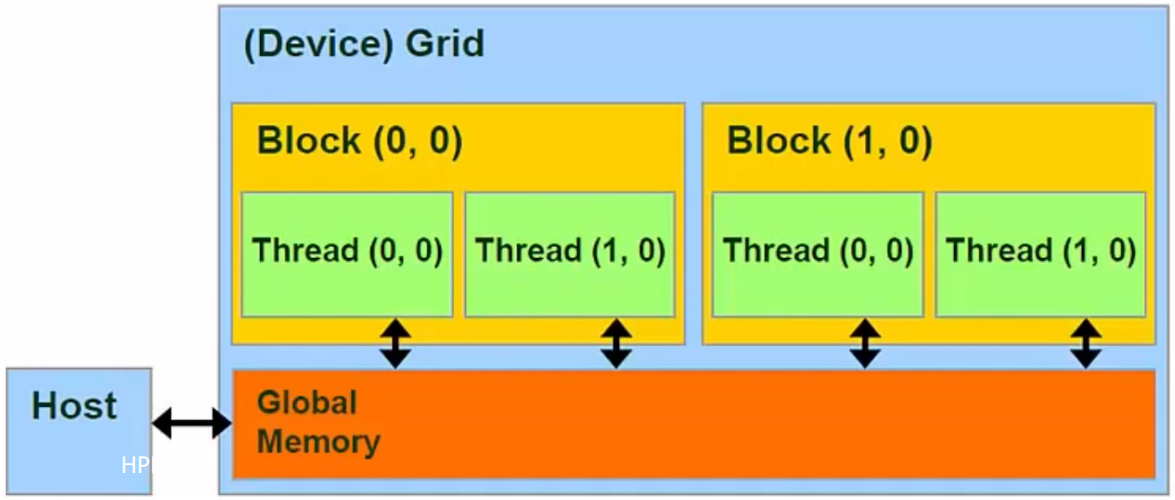
- Relative speed of memory spaces: $\underbrace{\text{"Memory Space"}}_{\substack{\text{"Bandwidth"}\\ \text{"Latency"}}}$

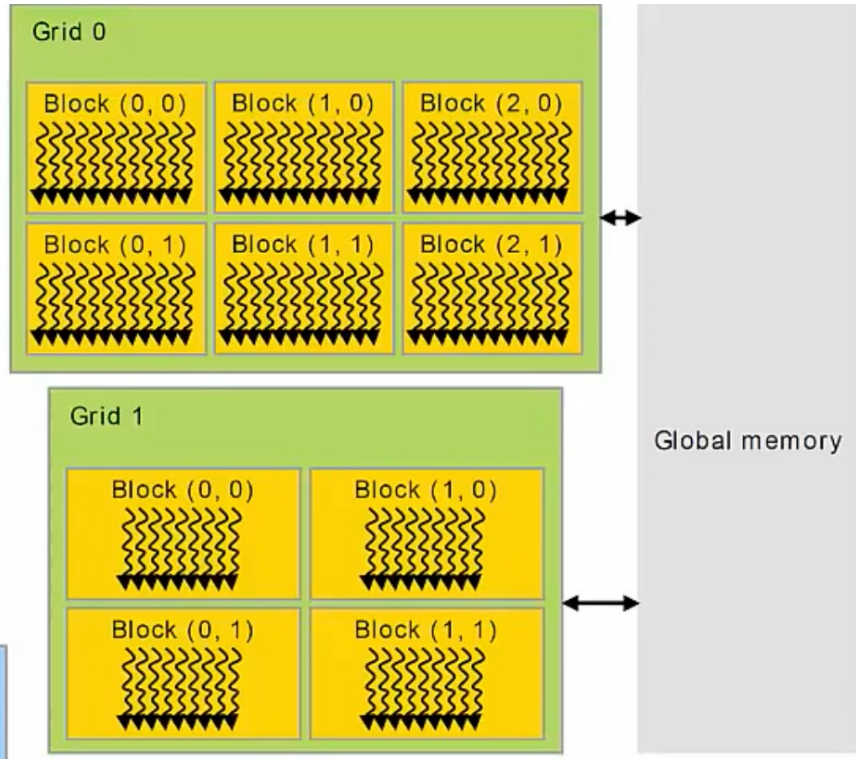
$$\underbrace{\text{Registers}}_{\substack{\sim 8\text{TB/s}\\ \sim 1\text{clock}}} < \underbrace{\text{Shared}}_{\substack{\sim 1.5\text{TB/s}\\ \sim 32\text{clocks}}} \ll \underbrace{\text{Local} \approx \text{Global}}_{\substack{\sim 200\text{GB/s}\\ \sim 800\text{clocks}}} \ll \underbrace{\text{Host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

Global Memory

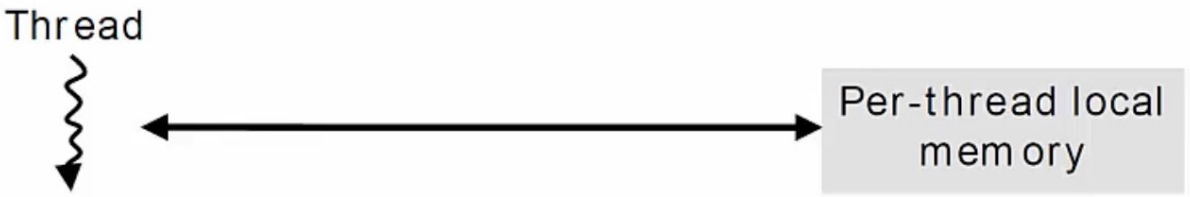
Accessed with

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

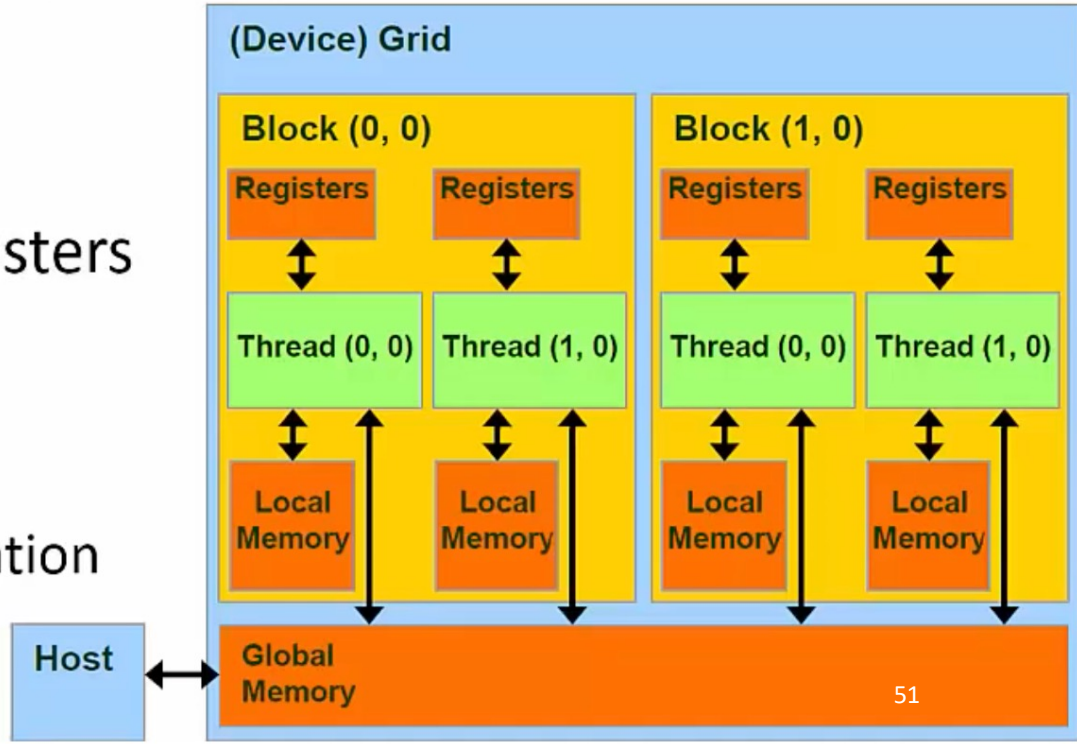




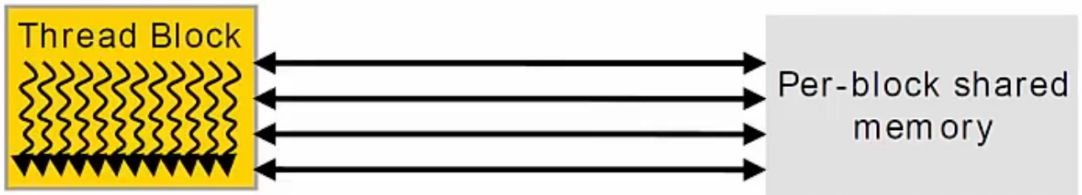
Registers and Local Memory



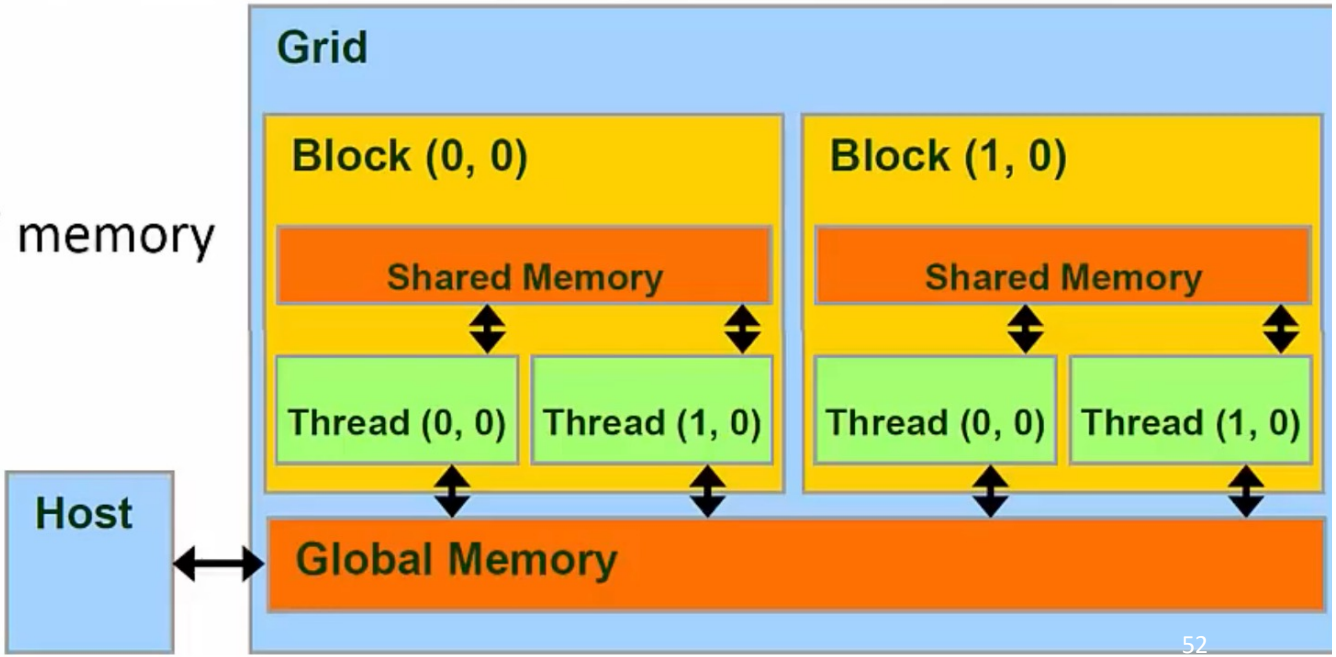
- Variables declared in a Kernel are stored in Registers
  - *On-Chip*
  - Fastest form of memory
- Arrays too large to fit into Registers spill over into Local memory
  - *Off-Chip*
  - Compiler controlled
  - "Local" refers to scope, not location
    - Local to each Thread



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

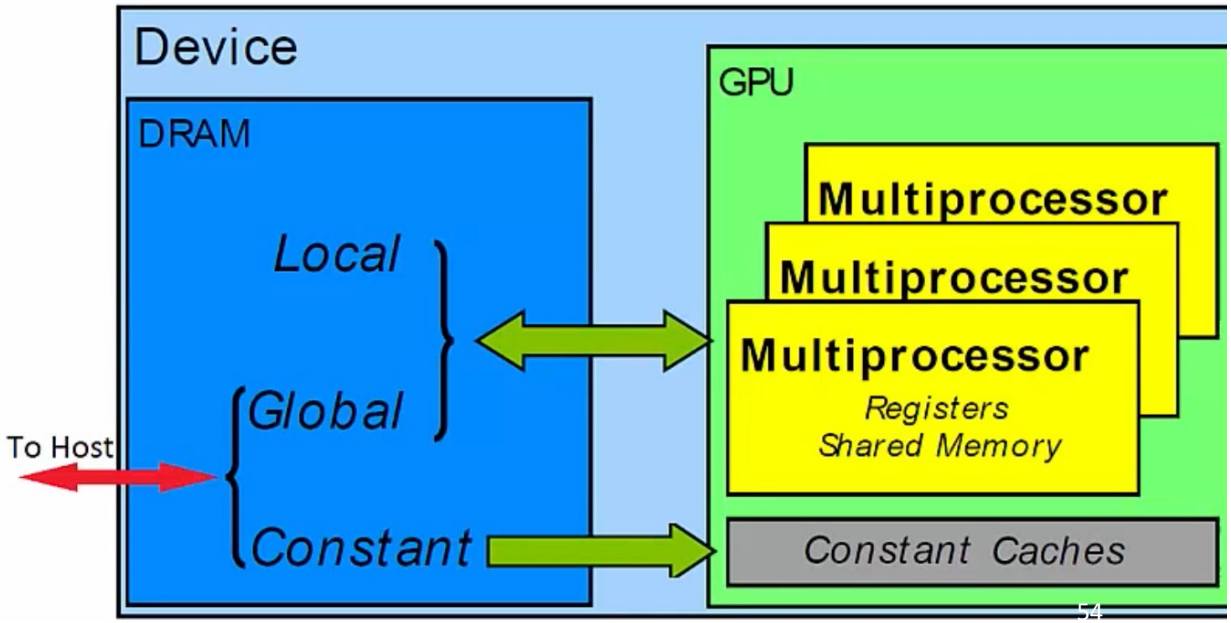


Using Shared Memory

```
__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

Shared Memory

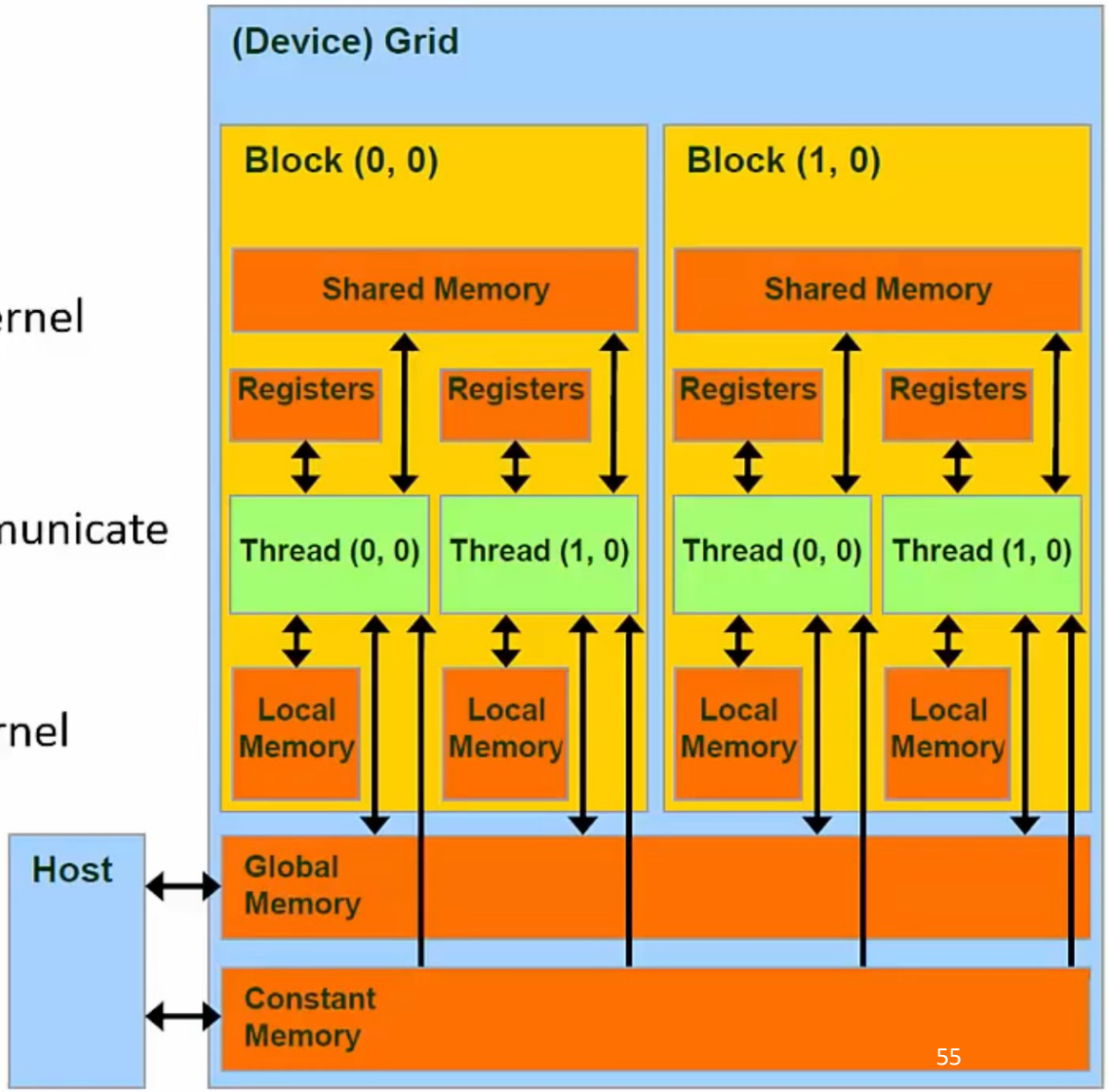
- Allows threads within a block to communicate

Constant

- Used for unchanging data through Kernel

Global Memory

- Stores data copied to and from Host



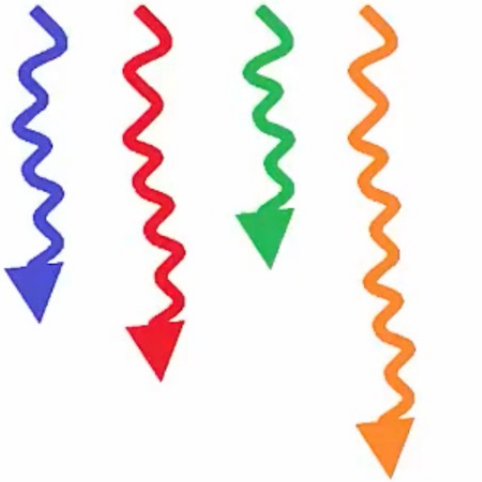
CUDA Thread Synchronization

Threads Execute in Parallel

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

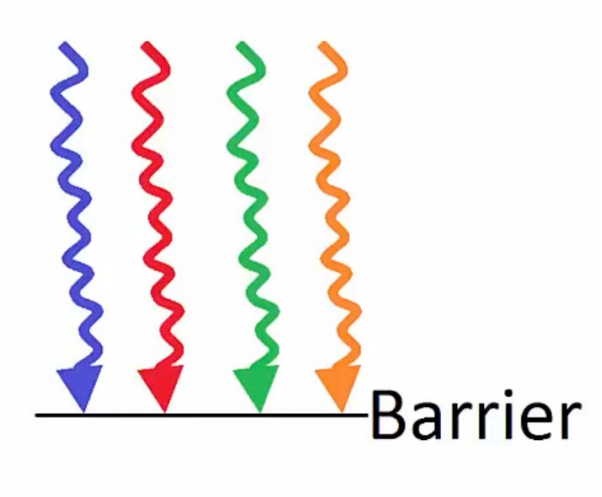
Thread Synchornization via Explicit Barrier

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



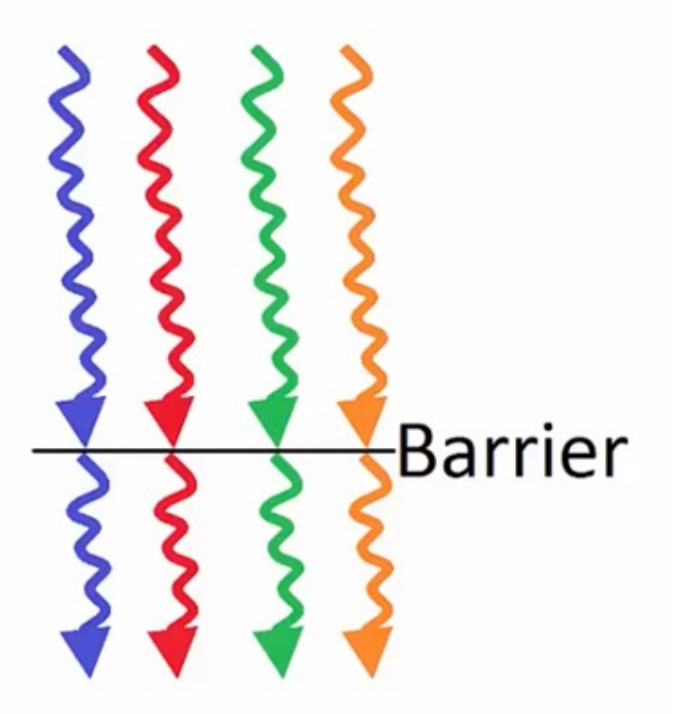
Thread Synchornization via Explicit Barrier

- 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

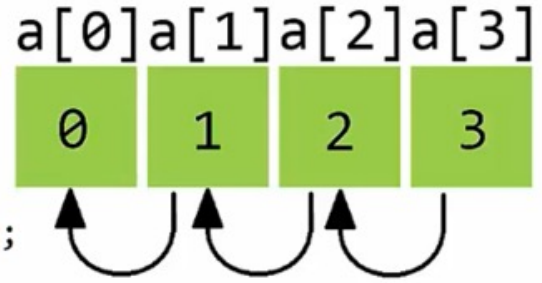
- Threads need to synchronize with one another to avoid race conditions
- A barrier is a point in the kernel where all the threads stop and wait on the others
- When all threads have reached the barrier, they can proceed
- Barriers can be implemented with:
  - __syncthreads();



syncthreads() Example

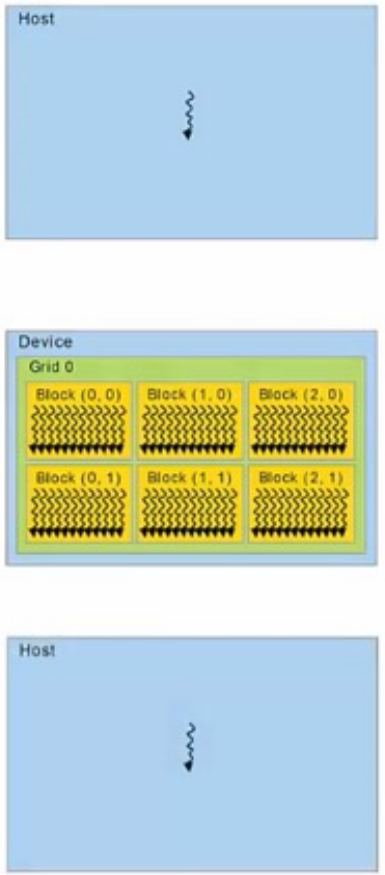
- Goal: Shift the contents of an array to the left by one elements

```
// 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;
    }
}
```

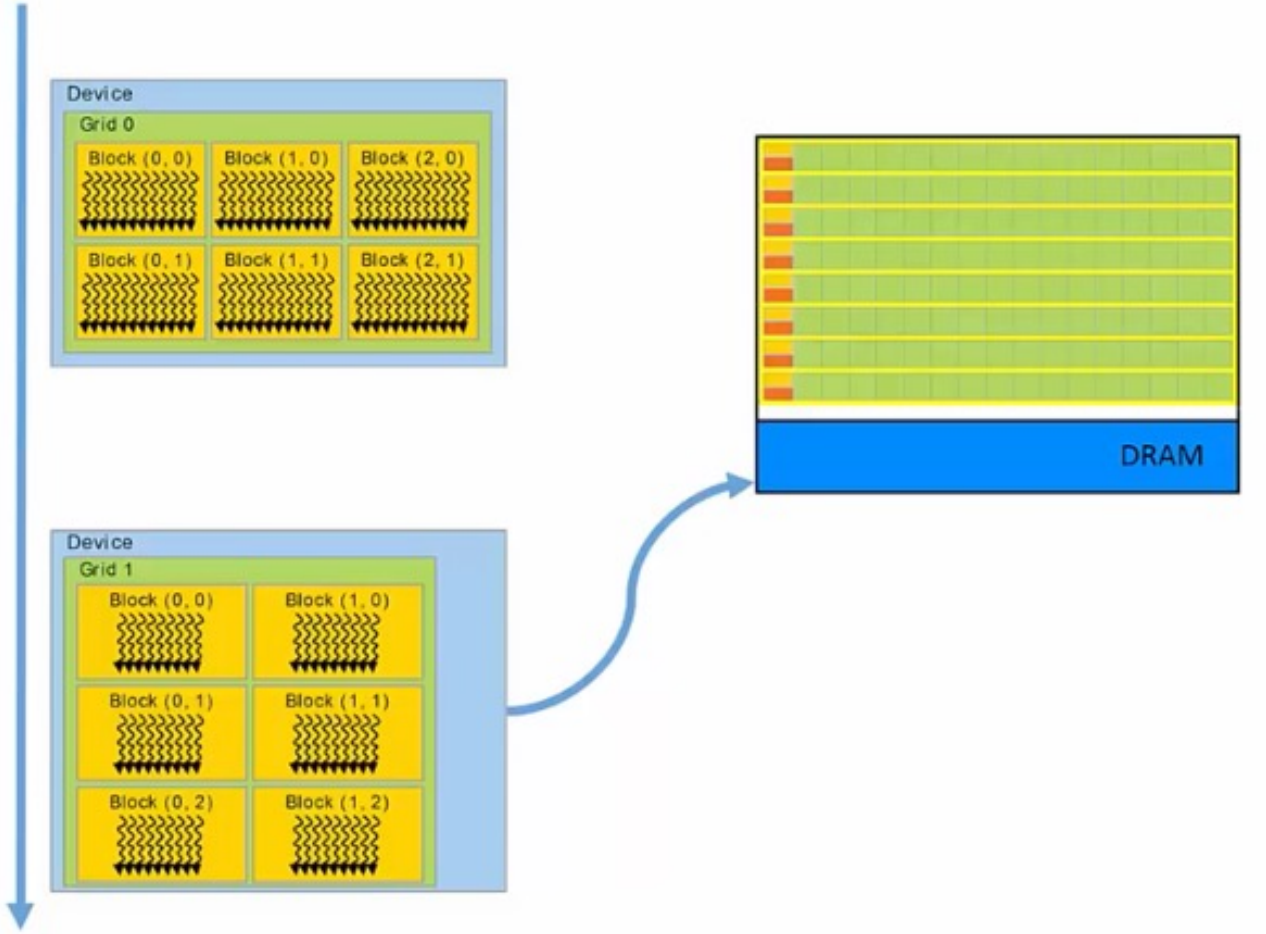


Implicit Barriers between Kernels

```
int main( void ) { // Host Code

  // Do sequential stuff

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

  // Force Host to wait on the
  // completion of the Kernel
  cudaDeviceSynchronize();

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

  // Do sequential stuff
  // ...

  return 0;
}
```



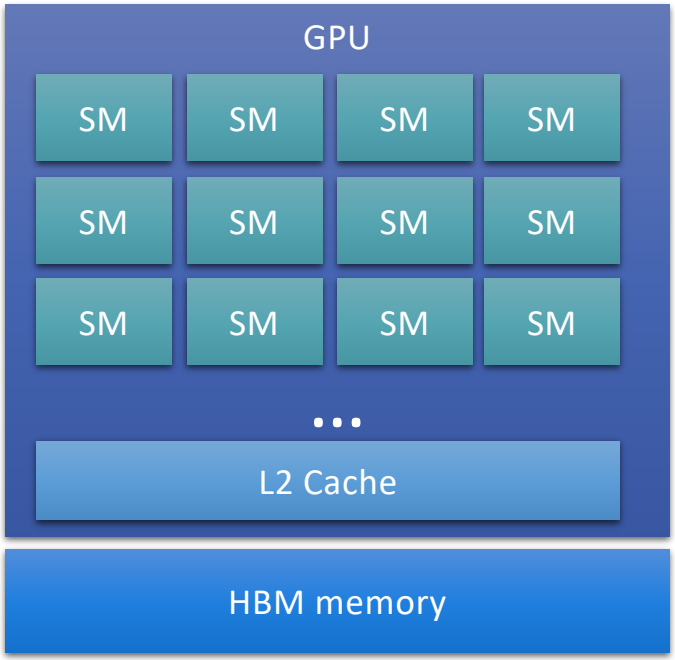
Implicit Barriers between Kernels



NVIDIA GPUs and CUDA

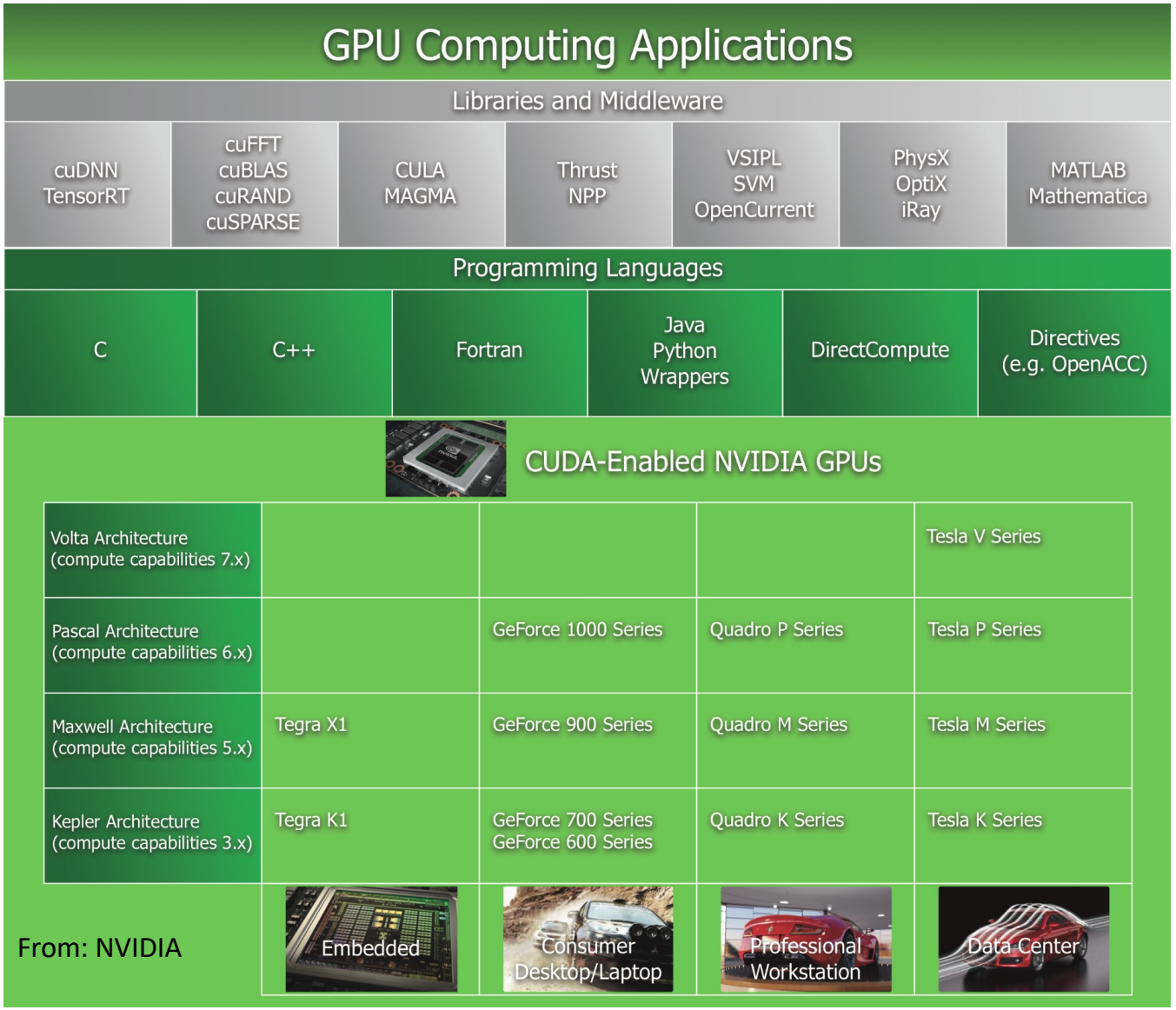
NVIDIA Volta Architecture (Tesla V100)

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



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



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

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

NVIDIA Hardware and Compute Capabilities

Feature Support	Compute Capability					
(Unlisted features are supported for all compute capabilities)	**3.5, 3.7, 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 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			
Warp vote functions (Warp Vote Functions)	Yes					
Memory fence functions (Memory Fence Functions)						
Synchronization functions (Synchronization Functions)						
Surface functions (Surface Functions)						
Unified Memory Programming (Unified Memory Programming)						
Dynamic Parallelism (CUDA Dynamic Parallelism)						
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

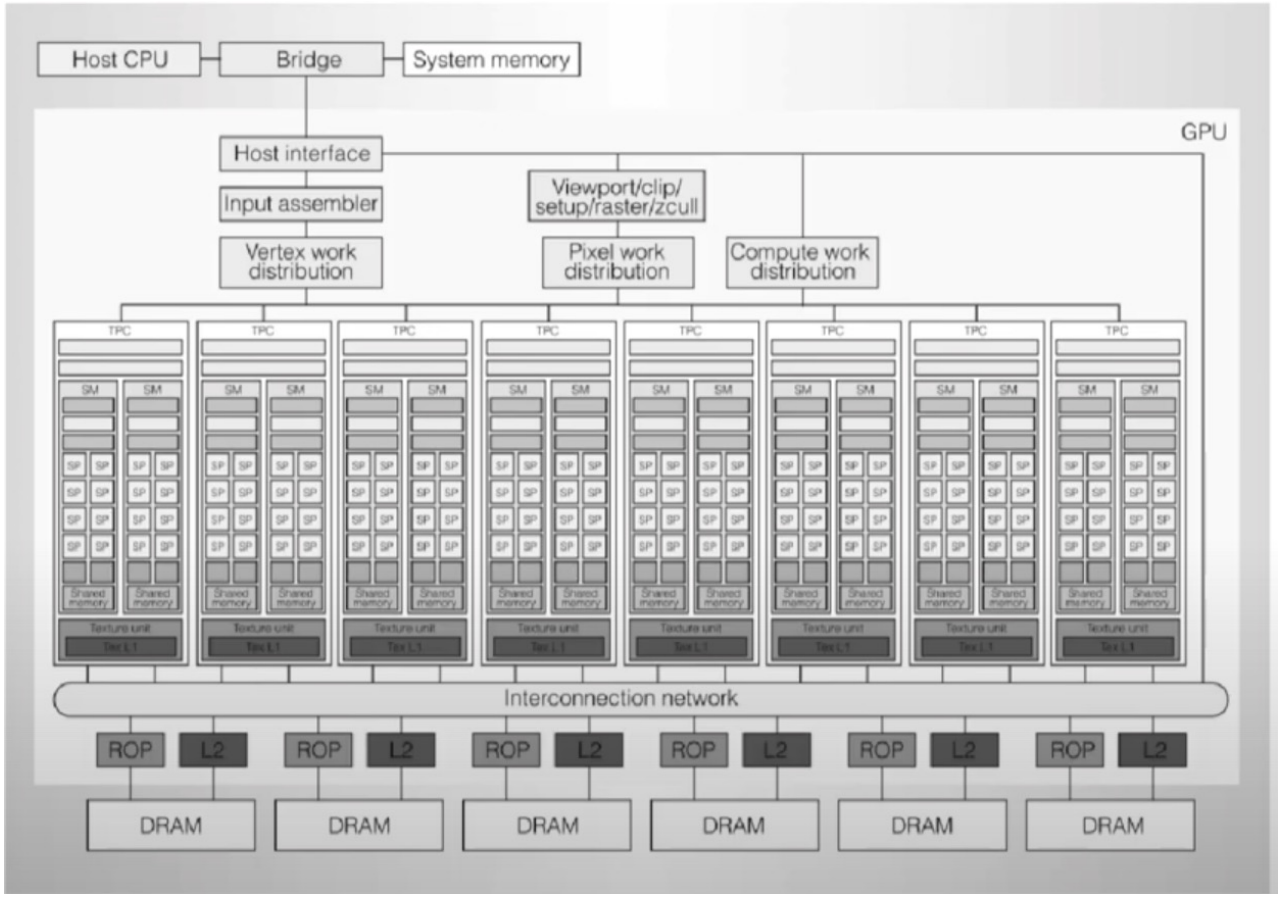
- CUDA Hardware
- CUDA Compilation and Runtime:
  - CUDA Runtime, CUDA Driver, AoT and JIT compilation
- CUDA Programming Model:
  - Grid, Block, Thread
  - UVM
- CUDA Warp Scheduling
- Context and Stream
- CUDA Profiling and Debugging

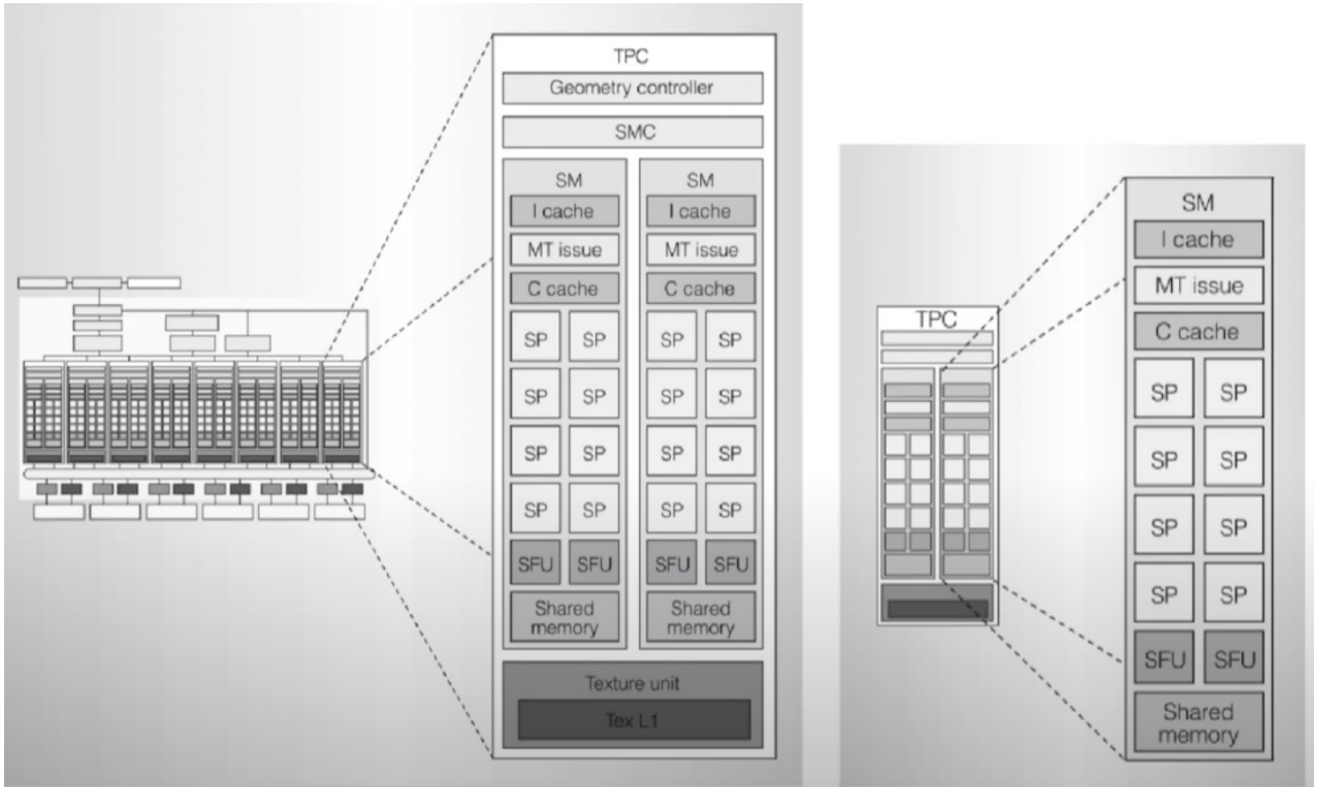
CUDA Hardware

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

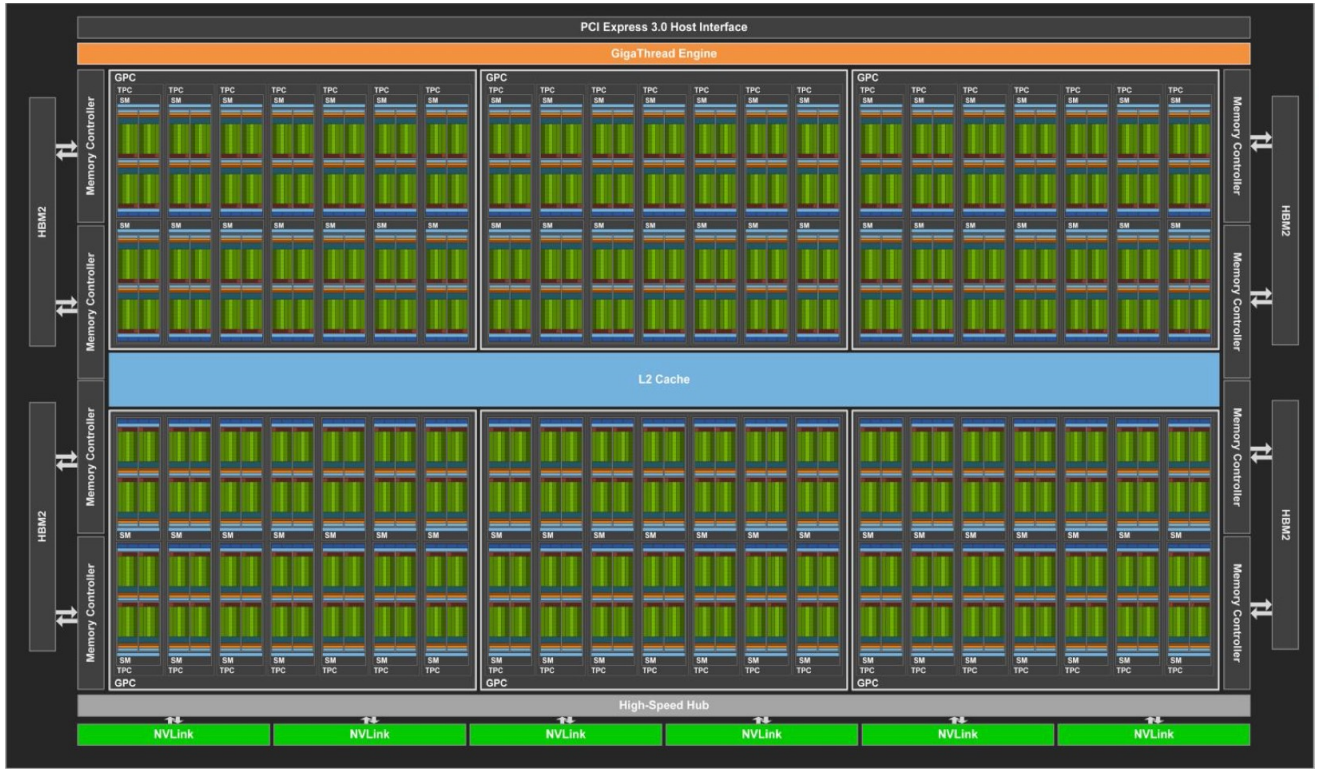
SF: Special Function Unit

Pascal GPU (2016)

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

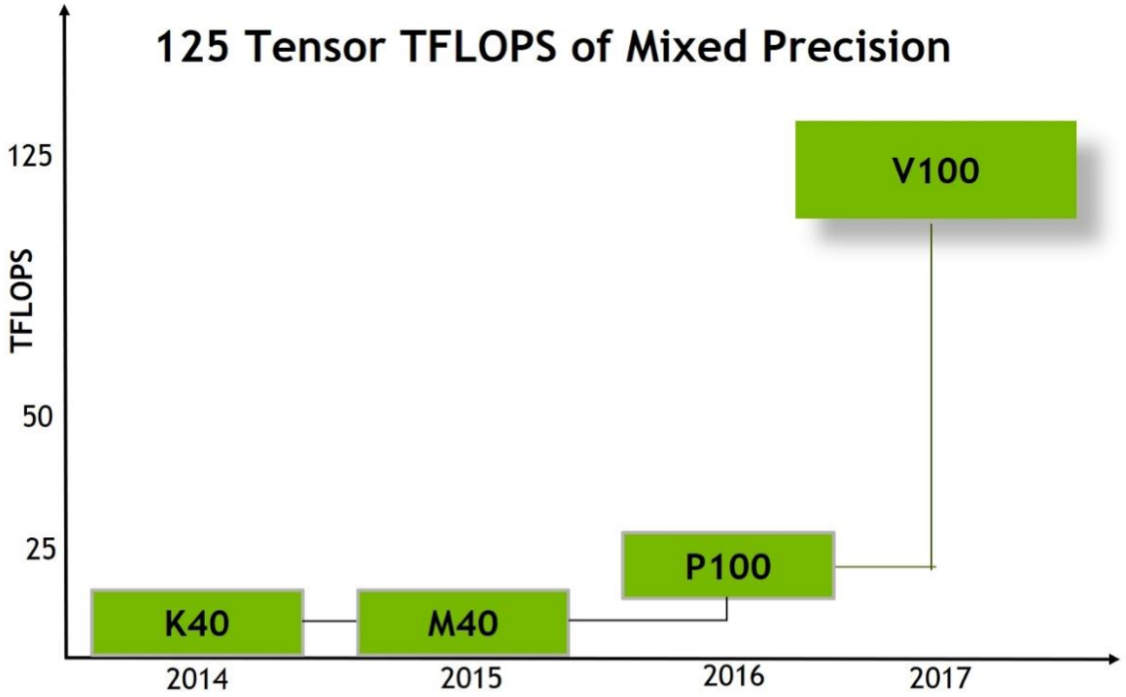
Pascal GPU (2016) - Closeup

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

Impact of Tensor Cores



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

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

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

Ampere GPU (2020) - SM

RT CORE: Ray Tracing

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

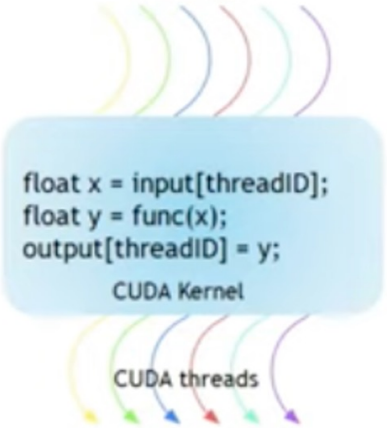
Summary/Comparisons

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	

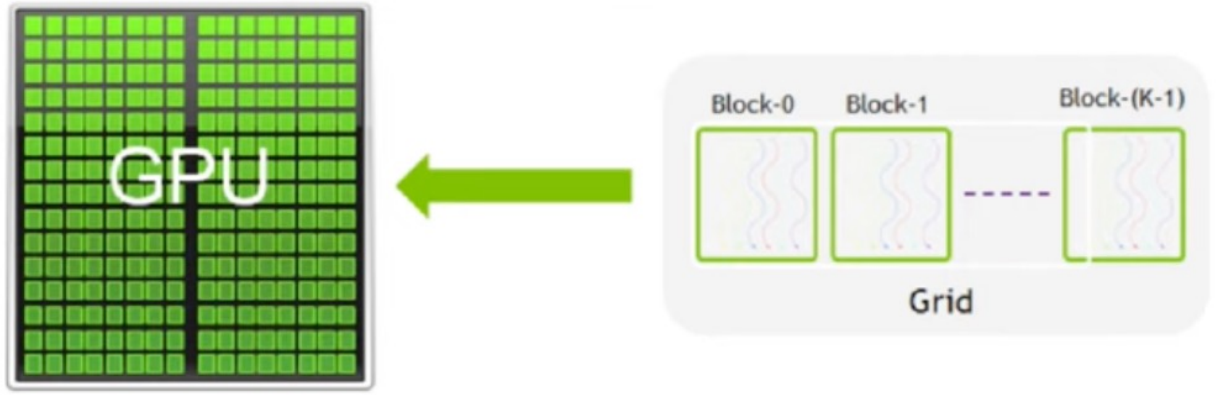
CUDA Programming Model

Part 2

Recap



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

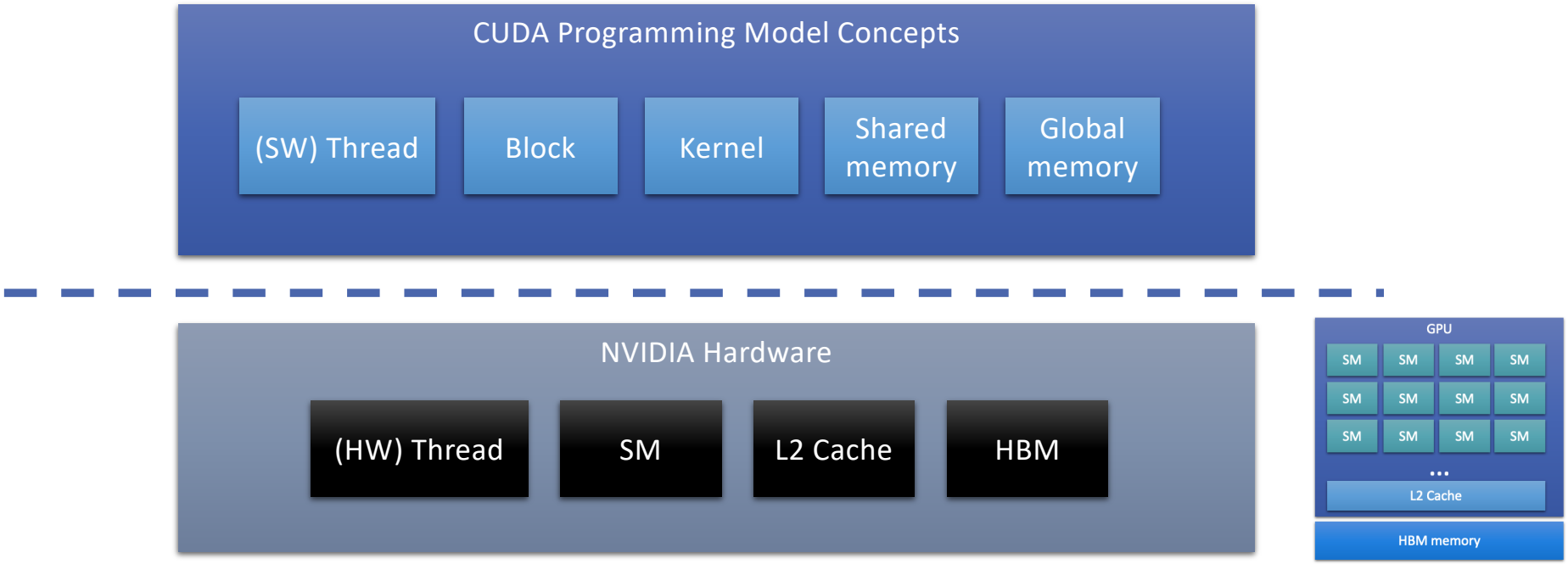


1. Kernel executes in a thread
2. Threads grouped into Thread Blocks (a.k.a. Blocks)
3. Blocks grouped into Grids

Kernel executed as a Grid of Blocks of Threads

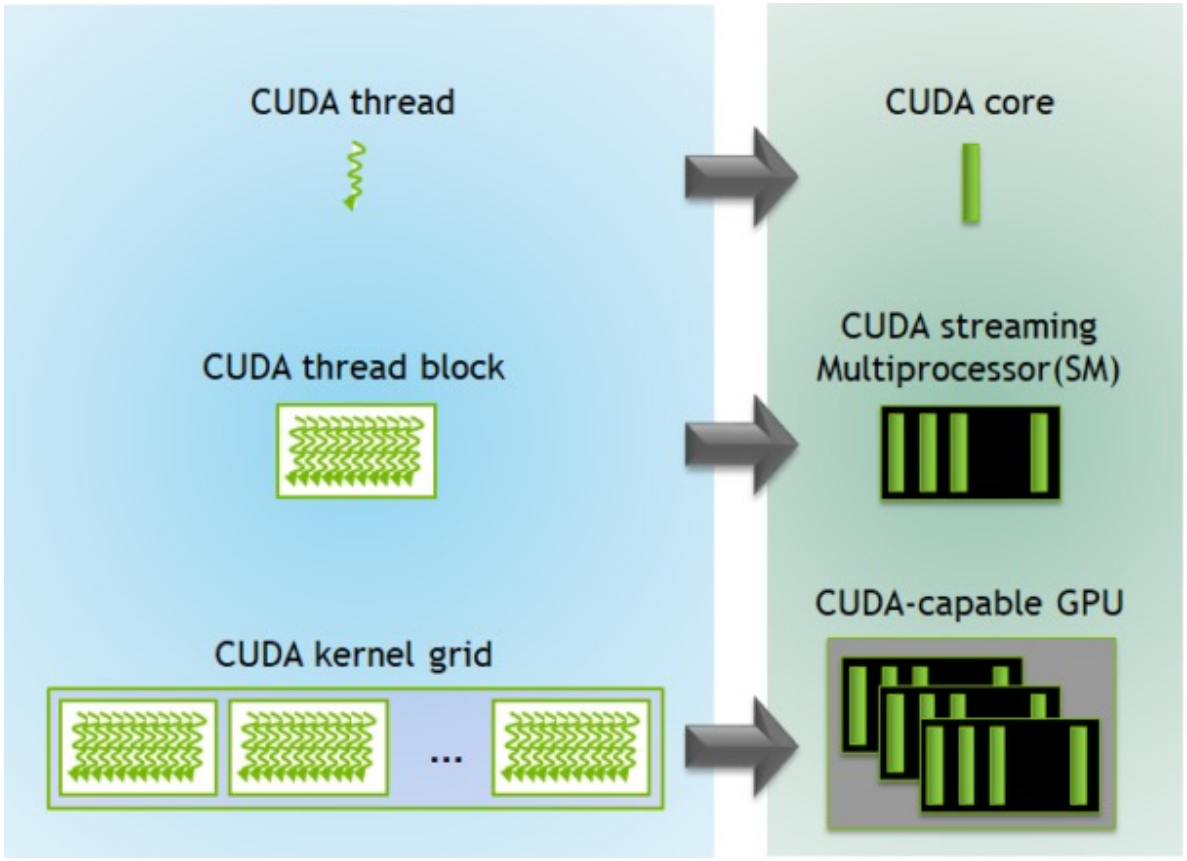
CUDA Programming model

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



HBM: High Bandwidth Memory
SM: Streaming Multiprocessor

Mapping to Hardware

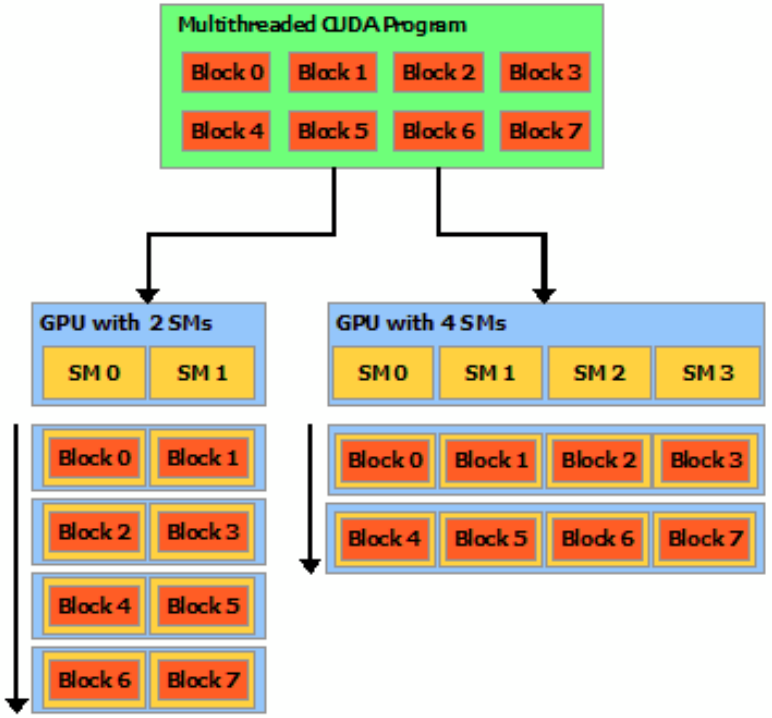


CUDA	Hardware
Thread	SP
Block	SM
Grid	GPU

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 increasing number of cores (SPs in SMs)
- Program using CUDA languages abstractions
- Divide into sub-problems, solved independently by thread blcoks
- Each block sones one sub-problem in parallel threads (running a kernel)
- CUDA runtime schedules blocks on multiprocessors
- Scheduling can be done in any order
  - Allows 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

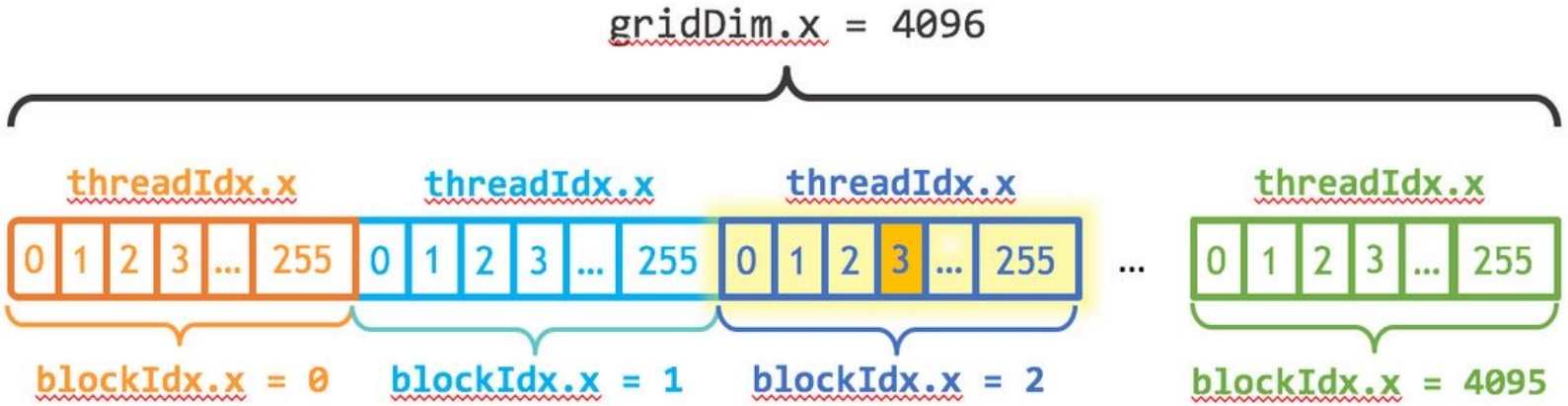
```
Int main() {
printf("Hello World!\n");
return 0;
}
```

- 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



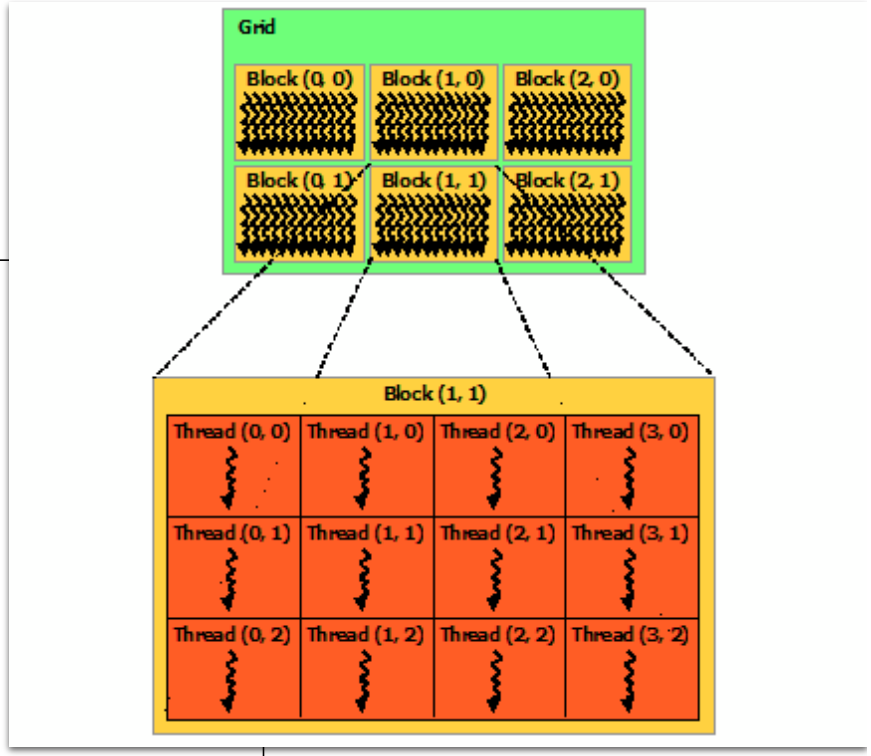

```
index = blockIdx.x  * blockDim.x + threadIdx.x
index =    (2)      *   (256)    +     (3)      = 515
```

From: NVIDIA

CUDA - Example

- Block and Threads index

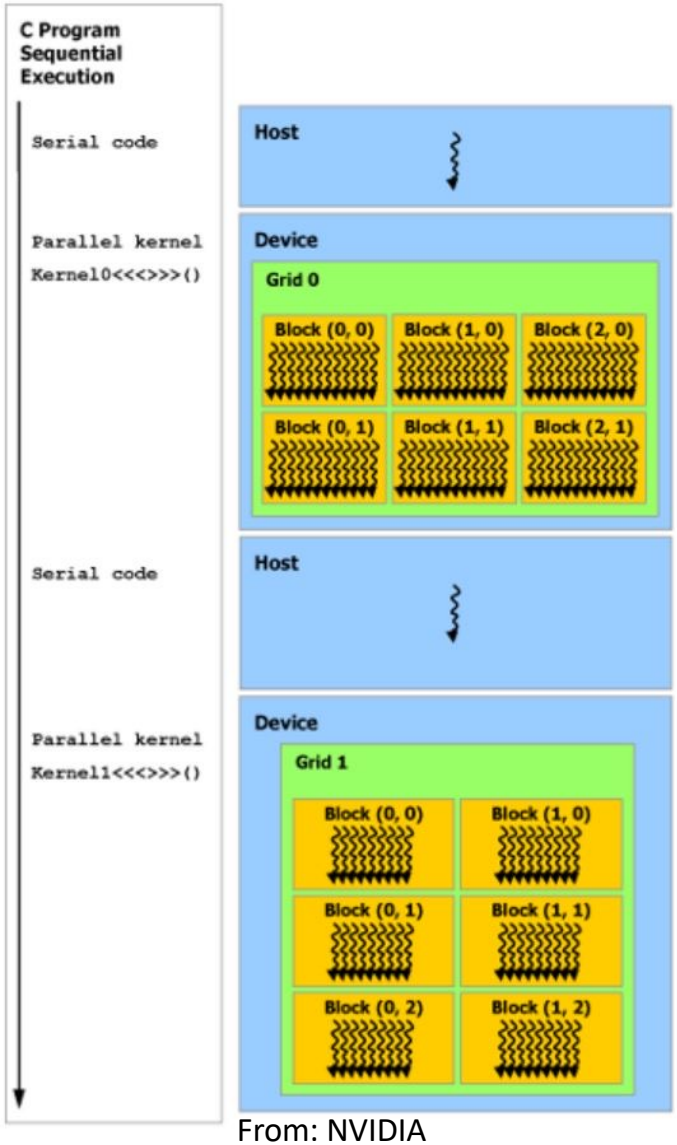
```
// Kernel definition
__global__ void MatAdd(float A[N][N], float B[N][N],
float C[N][N]) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;
    if (i < N && j < N)
        C[i][j] = A[i][j] + B[i][j];
}

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

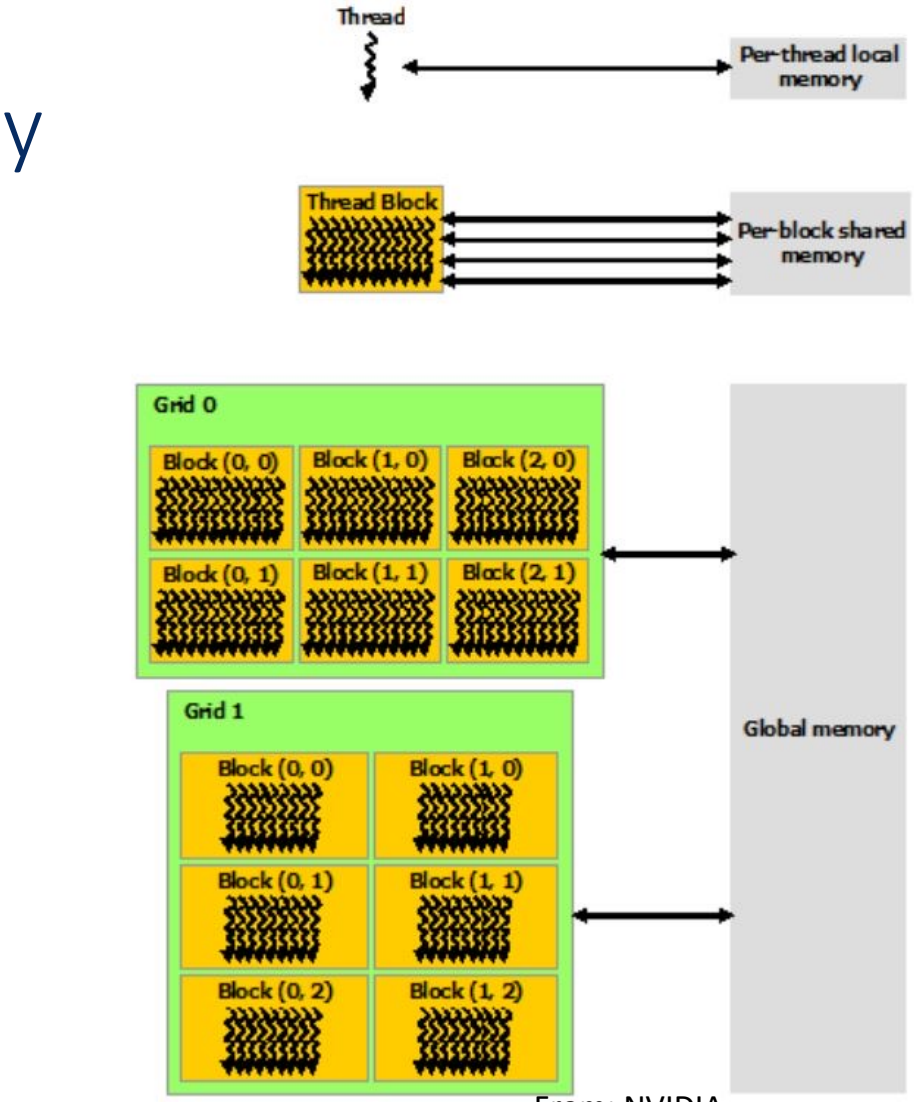
- A CUDA program starts as a sequential C/C++/Fortran process in the Host
- With a kernel launch:
  - Parallel code is executed inside the device (GPU)
  - SIMT Parallelism
    - Single Instruction Multiple Threads
- At the exit of a kernel launch the execution becomes sequential again



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 no 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

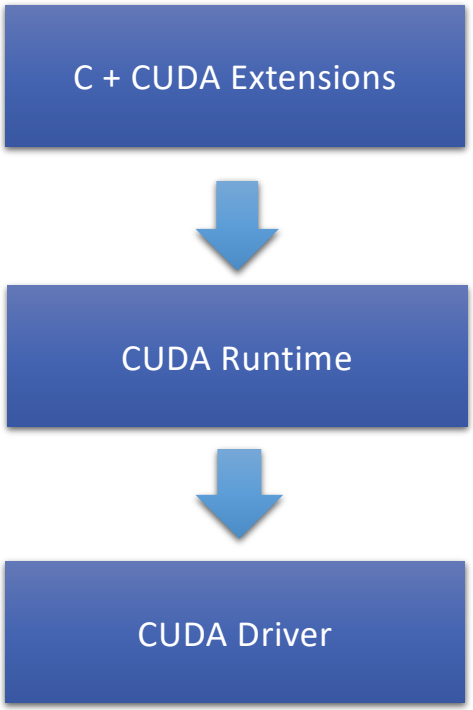
From: NVIDIA

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

CUDA Compilation and Runtime

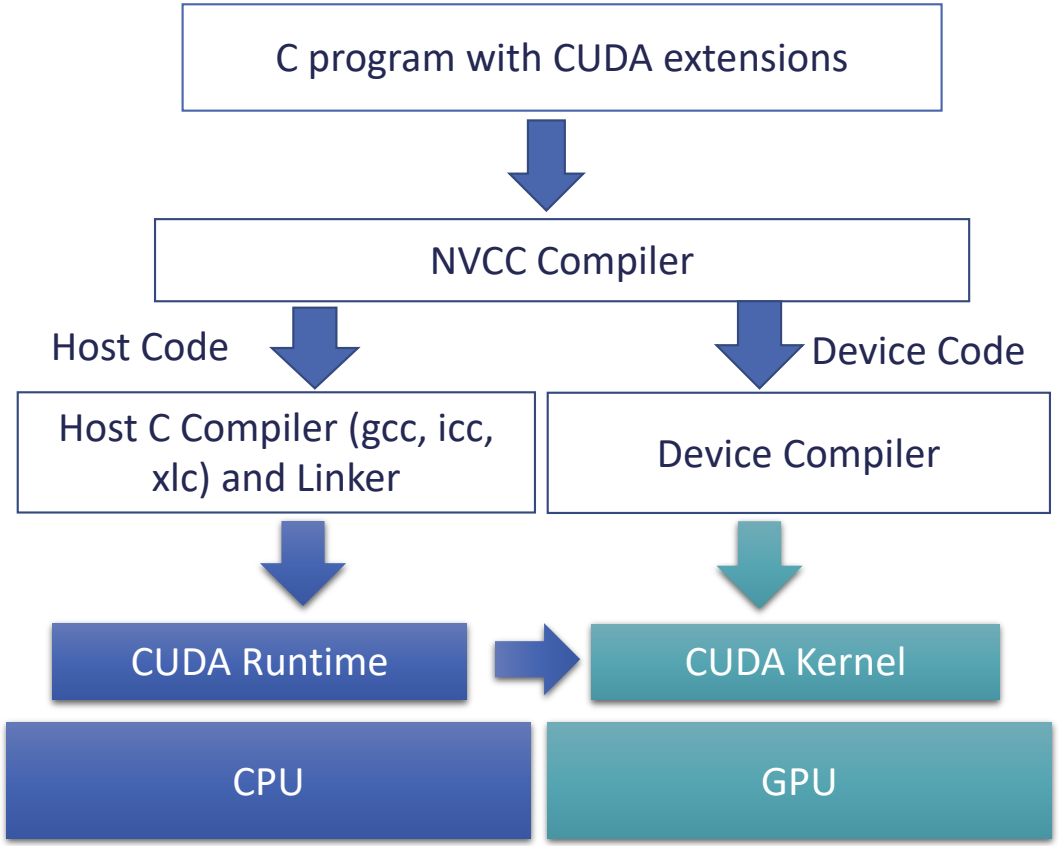
CUDA C Extension Runtime and Driver

- There are three layers for a CUDA program:
  1. CUDA C extension:
     - Developers write code using this extension
     - Ex. The <<<>>> kernel construct or the threadIdx variable
  2. CUDA Runtime API
     - NVCC compiler produces code that calls the runtime
     - Functions preceded by *cudaXXX*
     - Library: *cudart.dll or cudart.so*
  3. CUDA Driver API
     - The runtime uses the CUDA driver APIs
     - Library: *cuda.dll or cuda.so*
     - Functions preceded by cuXXX
     - Uses PTX code to execute on the GPU
- Parallel Thread Execution (PTX or NVPTX) is a low-level parallel thread execution virtual machine and instruction set architecture used in Nvidia's CUDA programming environment.
  - PTX is stable across multiple GPU generations
- The runtime API is a wrapper/helper of the driver API.



CUDA Compilation and Runtime

- NVCC compiler divides code in two parts:
  - Host C/C++ compiler
  - Device compiler
- Host C/C++ code is passed to the host compiler
- Host C code:
  - Contains calls to **CUDA runtime** and pass the **kernel code** as argument
- Kernel code is executed on the GPU



Virtual vs. real architecture when compiling

- A virtual GPU is defined entirely by the set of capabilities, or features, that it provides to the application
- PTX code (text format) can be considered as assembly for a virtual GPU architecture
- nvcc compilation command always uses two architectures: a compute architecture to specify the virtual intermediate architecture, plus a real GPU architecture to specify the intended processor to execute on

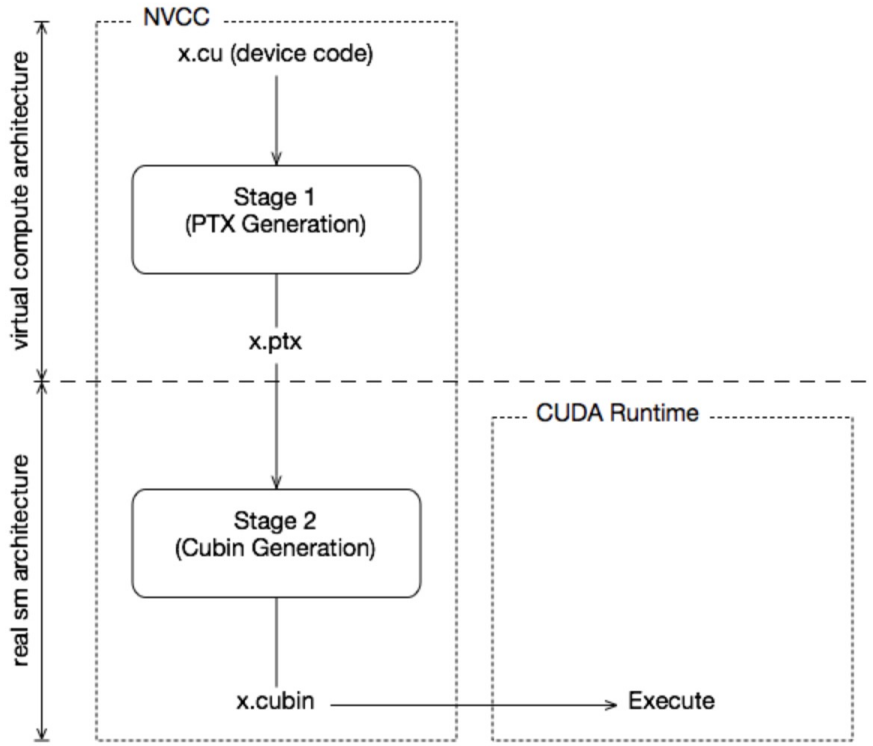
```
nvcc x.cu --gpu-architecture=compute_50 --gpu-code=sm_50,sm_52
```

- The real architecture must be an implementation (someway or another) of the virtual architecture

What architecture to specify when compiling?

- Specify the lowest virtual architecture that has a sufficient feature set to enable the program to be executed on the widest range of physical architectures
- Chosen virtual architecture is more of a statement on the GPU capabilities that the application requires
- Using a smallest virtual architecture still allows a widest range of actual architectures for the second nvcc stage (PTX→cubin)
- Specifying a virtual architecture that provides features unused by the application unnecessarily restricts the set of possible GPUs that can be specified in the second nvcc stage.

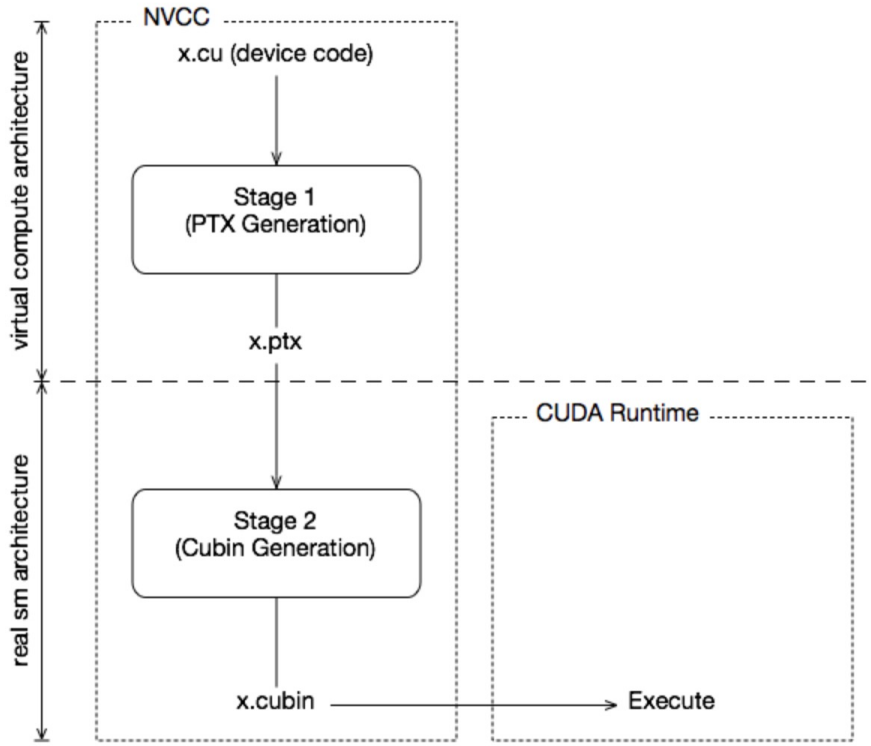
Two staged compilation with Virtual and Real architectures



- nvcc compiler driver (similar to gcc)
- Off-line Compilation
  - Host Code
    - Modified to run kernels
    - Compiled to (x86) binary
  - Device Code
    - Compiled to PTX (Parallel Thread Execution)
- Just-in-Time Compilation
  - Compile PTX into native GPU instructions
  - Allows forward compatibility

https://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/index.html#virtual-architecture-feature-list

Two staged compilation with Virtual and Real architectures



TIPS:
Maximize portability of code: Choose *virtual* arch as *low* as possible when compiling

Get the optimal performance on a GPU: r*eal* arch should be chosen as *high* as possible

https://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/index.html#virtual-architecture-feature-list

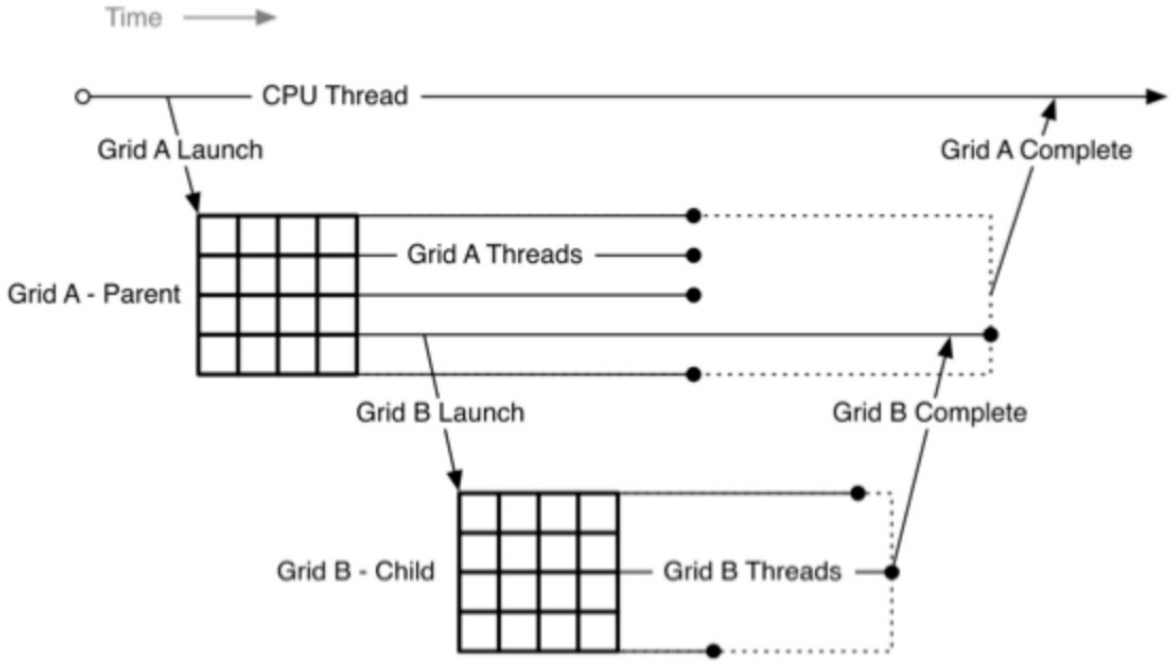
NVIDIA Virtual Architectures Feature List

`compute_35`, and `compute_37`	Kepler support Unified memory programming Dynamic parallelism support
`compute_50`, `compute_52`, and `compute_53`	+ Maxwell support
`compute_60`, `compute_61`, and `compute_62`	+ Pascal support
`compute_70` and `compute_72`	+ Volta support
`compute_75`	+ Turing support
`compute_80`, `compute_86` and `compute_87`	+ NVIDIA Ampere GPU architecture support
`compute_89`	+ Ada support
`compute_90`	+ Hopper support

https://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/index.html#virtual-architecture-feature-list

Dynamic Parallelism

CUDA Dynamic Parallelism: a *parent grid* launches kernels called *child grids*.



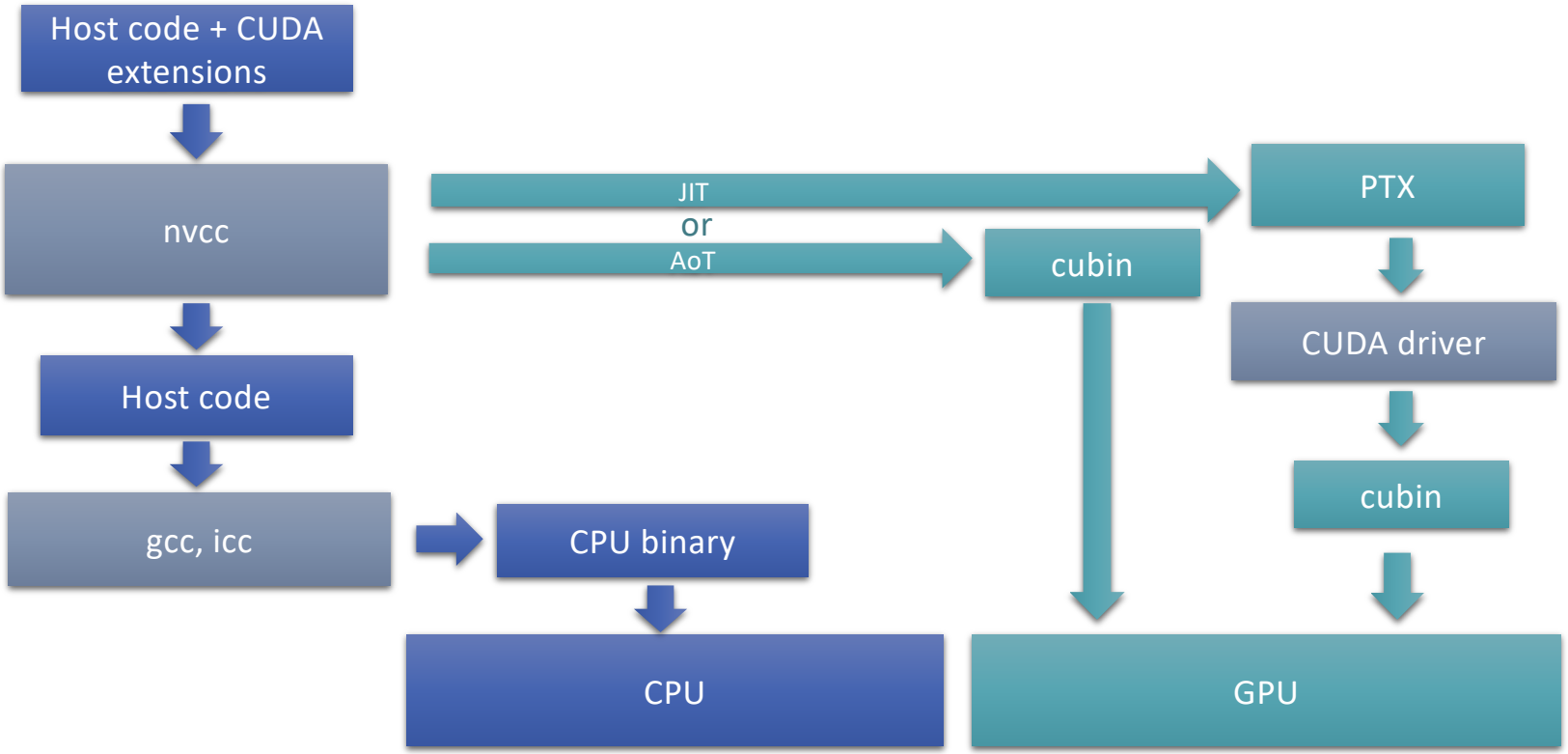
https://developer.nvidia.com/blog/cuda-dynamic-parallelism-api-principles/

CUDA Just-in-time compilation

- Two approaches to compiling CUDA
- CUDA Ahead of Time/Offline Compilation:
  - Compilation flag code (example: --code=sm_60 for compute capability 6.0)
  - Directly compile into CUDA binary: **cubin**
- CUDA JIT:
  - Compilation flag arch (example: --arch=compute_60 for compute capability 6.0)
  1. Compile kernels in **PTX** (CUDA assembly)
  2. PTX code is compiled by the **CUDA Driver**

What is the advantage of JIT for CUDA applications like games?

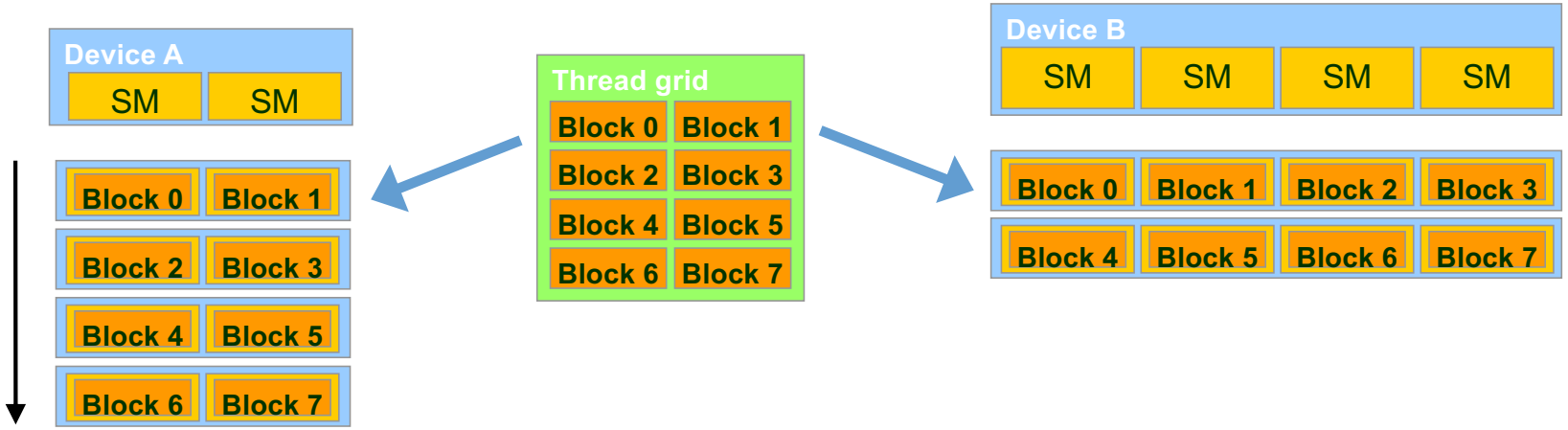
CUDA Compilation Schema



http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#compilation-with-nvcc

CUDA Block and Warp Scheduling

CUDA Transparent Scalability



- Each block can execute in any order relative to others.
- Hardware is free to assign blocks to any processor at any time
  - A kernel scales to any number of SMs

CUDA – Grids and Blocks maximum sizes

- Grid, and Block sizes have been the same for all Compute capabilities
- Newer GPUs can have more SMs

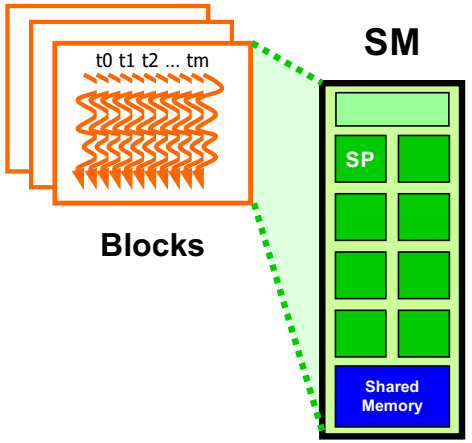
	Compute Capability										
Technical Specifications	**3.0**	**3.2**	**3.5**	**3.7**	**5.0**	**5.2**	**5.3**	**6.0**	**6.1**	**6.2**	**7.0**
Maximum number of resident grids per device (Concurrent Kernel Execution)	16	4	32				16	128	32	16	128
Maximum dimensionality of grid of thread blocks	3										
Maximum x-dimension of a grid of thread blocks	$2^{31}-1$										
Maximum y- or z-dimension of a grid of thread blocks	65535										
Maximum dimensionality of thread block	3										
Maximum x- or y-dimension of a block	1024										
Maximum z-dimension of a block	64										
Maximum number of threads per block	1024										
Warp size	32										

From: NVIDIA

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

Example: Executing Thread Blocks on Fermi

- Threads are assigned to Streaming Multiprocessors (SM) in block granularity
  - Fermi's compute capability:
    - See: https://en.wikipedia.org/wiki/CUDA
    - Up to **8** blocks to each SM as resource allows
  - Example: A Fermi SM can take up to **1536** threads
    - Could be 256 (threads/block) * 6 blocks
    - Or 512 (threads/block) * 3 blocks, etc.
  - Recent GPUs (Maxwell, Pascal, Volta)
    - resident blocks per SM: 32
- SM maintains thread/block idx #s
- SM manages/schedules thread execution

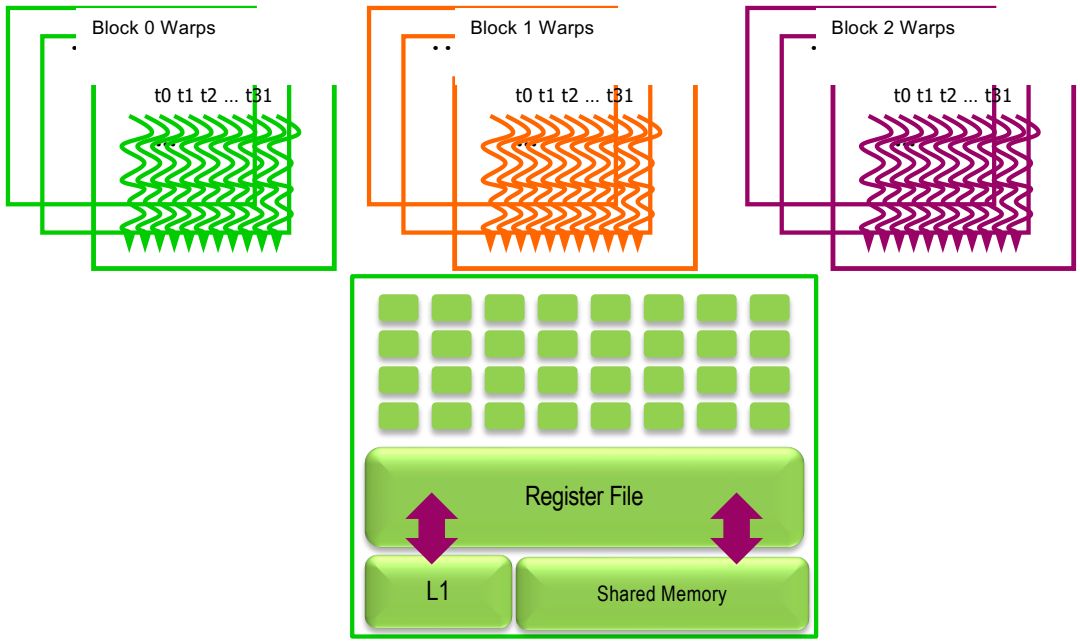


Warps as Scheduling Units

- Each Block is executed as 32-thread Warps
  - An implementation decision, not part of the CUDA programming model
  - Warps are scheduling units in SM
  - Threads in a warp execute in SIMD
  - Future GPUs may have different number of threads in each warp

Warp Example

- If 3 blocks are assigned to an SM and each block has 256 threads, how many Warps are there in an SM?
- Each Block is divided into 256/32 = 8 Warps
- There are 8 * 3 = 24 Warps



Example: Thread Scheduling (Cont.)

- SM implements zero-overhead warp scheduling
  - Warps whose **next instruction has its operands ready** for consumption are eligible for execution
  - Eligible Warps are selected for execution based on an optimized scheduling policy
  - All threads in a warp execute the **same instruction** when selected

SM Occupancy for performance

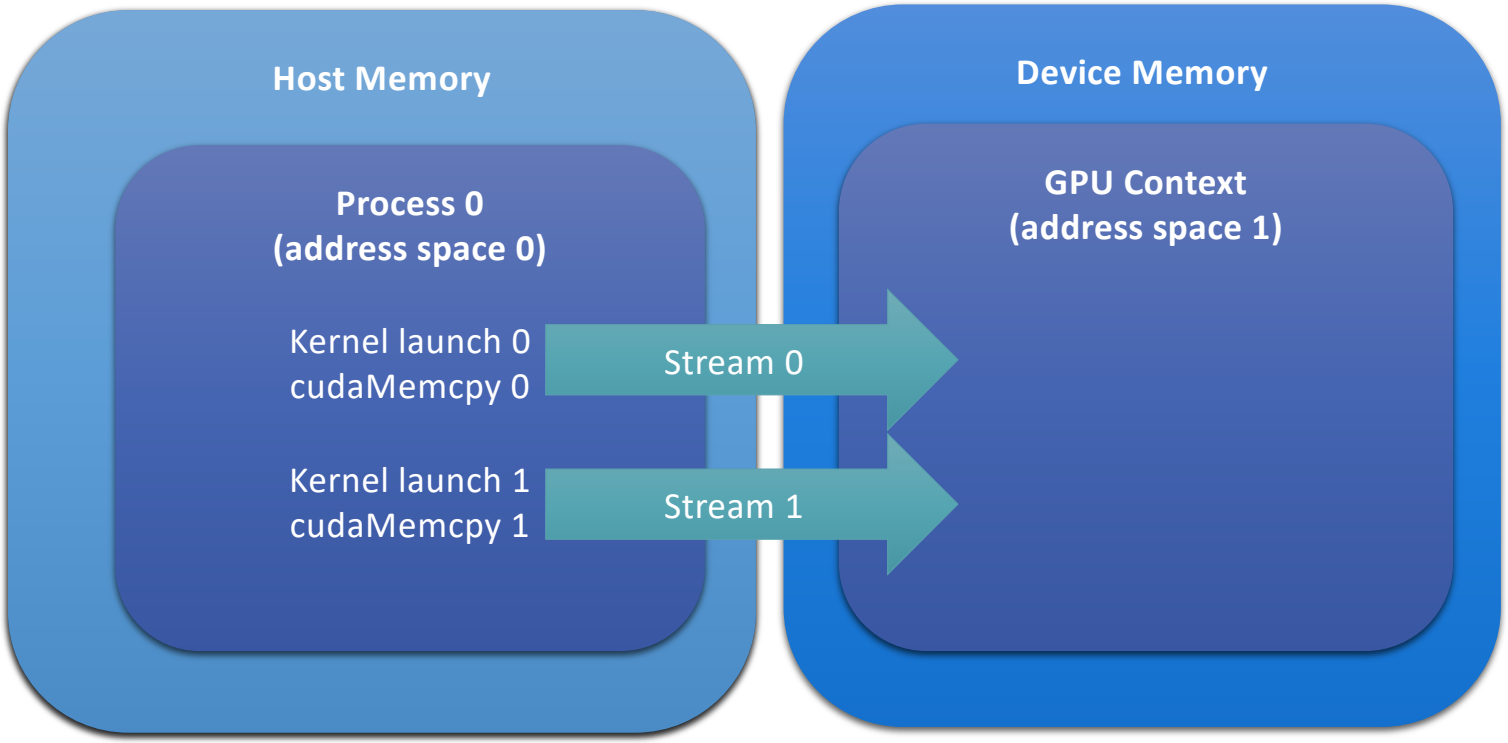
- Always try to achieve maximum **SM occupancy:**
  - #Active-Warps / #Max-Active-Warps
  - Approach: increase block size until it can fit the SM
- For Matrix Multiplication using multiple blocks, should I use 8X8, 16X16 or 32X32 blocks for Fermi?
  - For 8X8, we have 64 threads per Block. Since each SM can take up to 1536 threads, which translates to **24 Blocks** (1536/64). However, **each SM can only take up to 8 Blocks**, only 512 **(8*64**) threads will go into each SM!
  - For 16X16, we have 256 threads per Block. Since each SM can take up to 1536 threads, it can take up to 6 Blocks (1536/256) and achieve full capacity unless other resource considerations overrule.
  - For 32X32, we would have 1024 threads per Block. Only one block can fit into an SM for Fermi. Using only 2/3 of the thread capacity of an SM.
- https://docs.nvidia.com/gameworks/content/developertools/desktop/analysis/report/cudaexperiments/kernellevel/achievedoccupancy.htm

CUDA Context and Streams

CUDA Context

- Each process has a unique context
- Only a single context can be active on a device at a time
- Multiple processes on a single GPU could not operate concurrently
- Best practice would be to create one CUDA context per device
- Device memory allocated in context A cannot be accessed by context B.

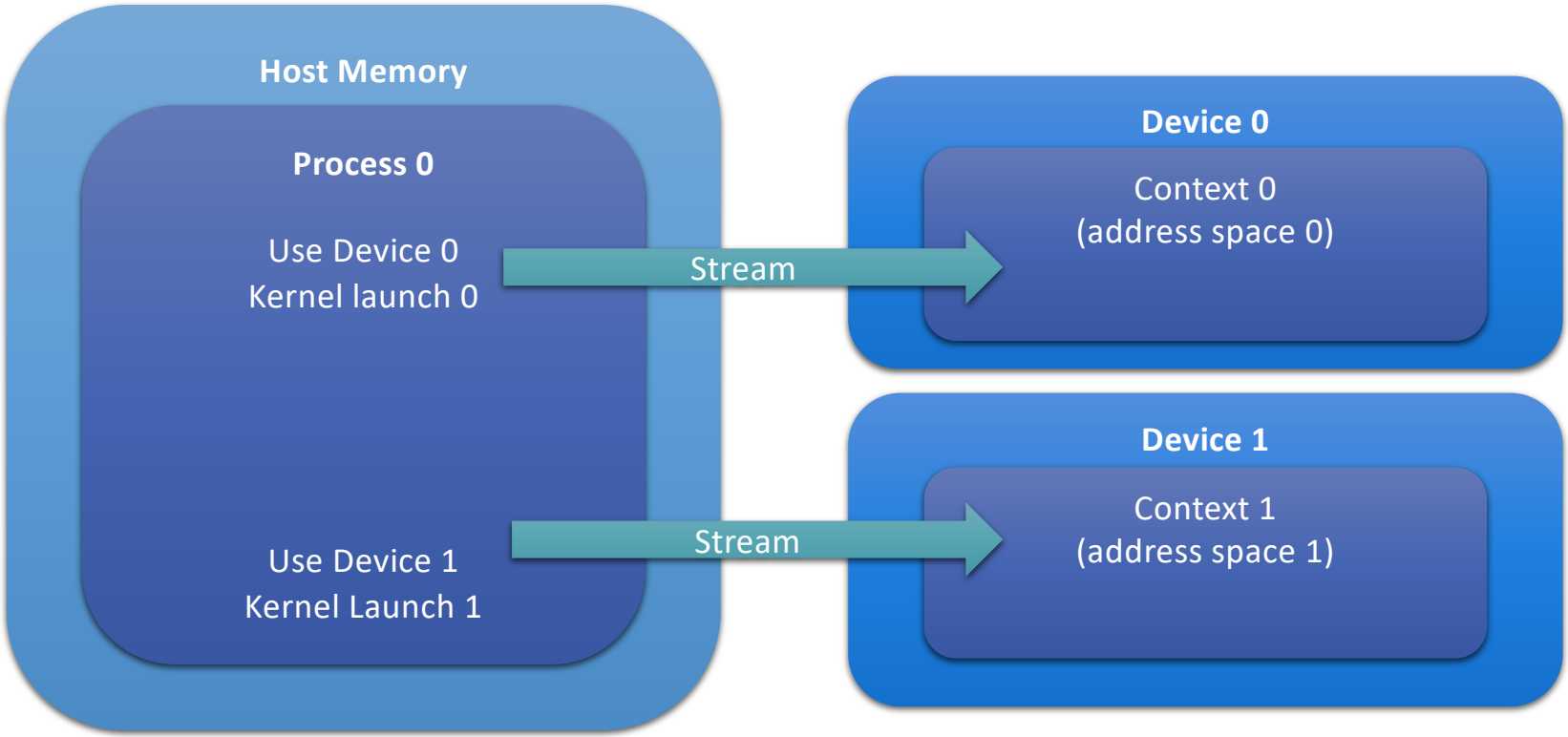
CUDA Context and Streams



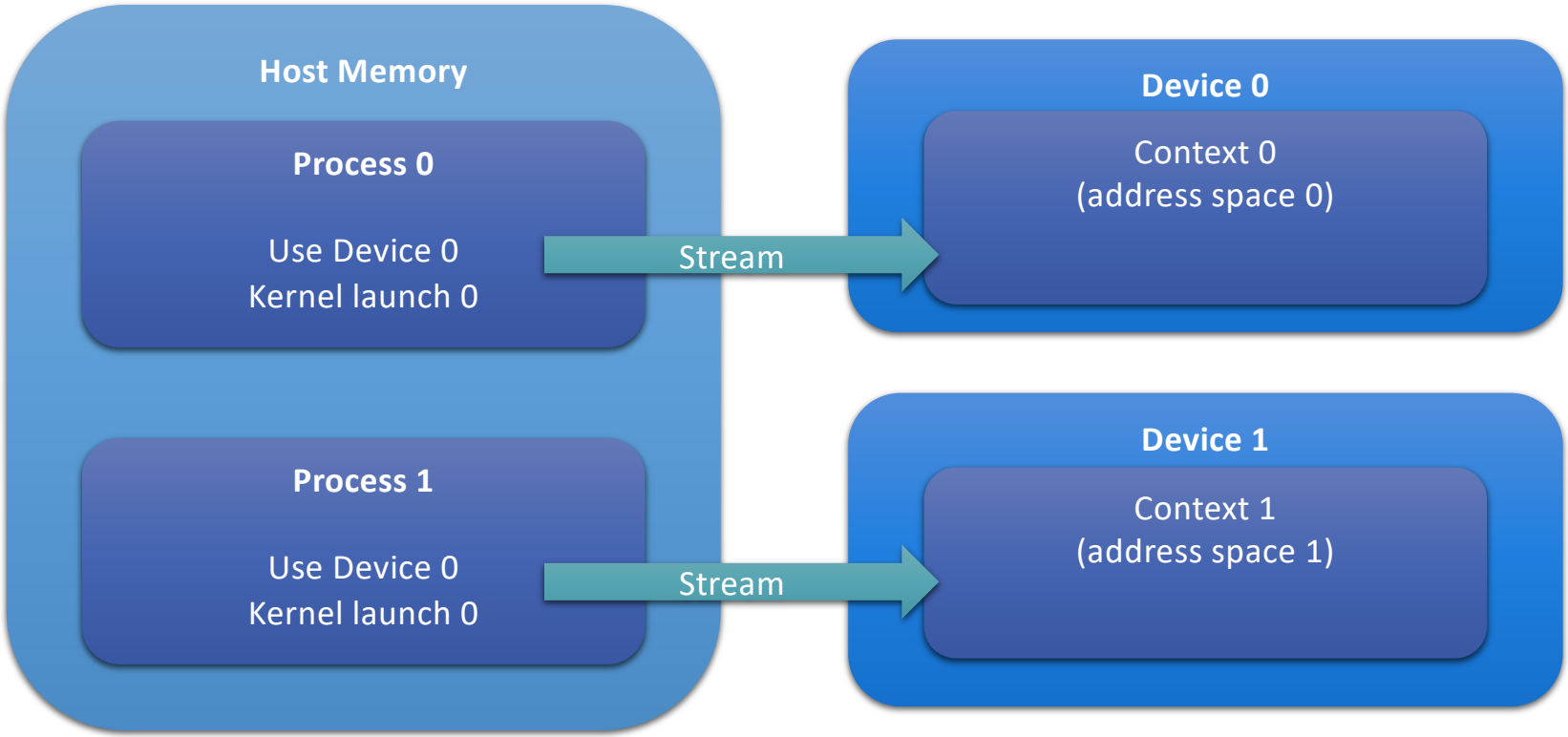
CUDA Streams

- A stream is a queue of device work
  - The host places work in the queue and continues on immediately
  - Device schedules work from streams when resources are free
- CUDA operations are placed within a stream
  - e.g. Kernel launches, memory copies
- Operations within the same stream are ordered (FIFO) and cannot overlap
- Operations in different streams are unordered and can overlap
- Streams are valid only in the context in which they were created
- By default CUDA uses one primary stream but more can be created.

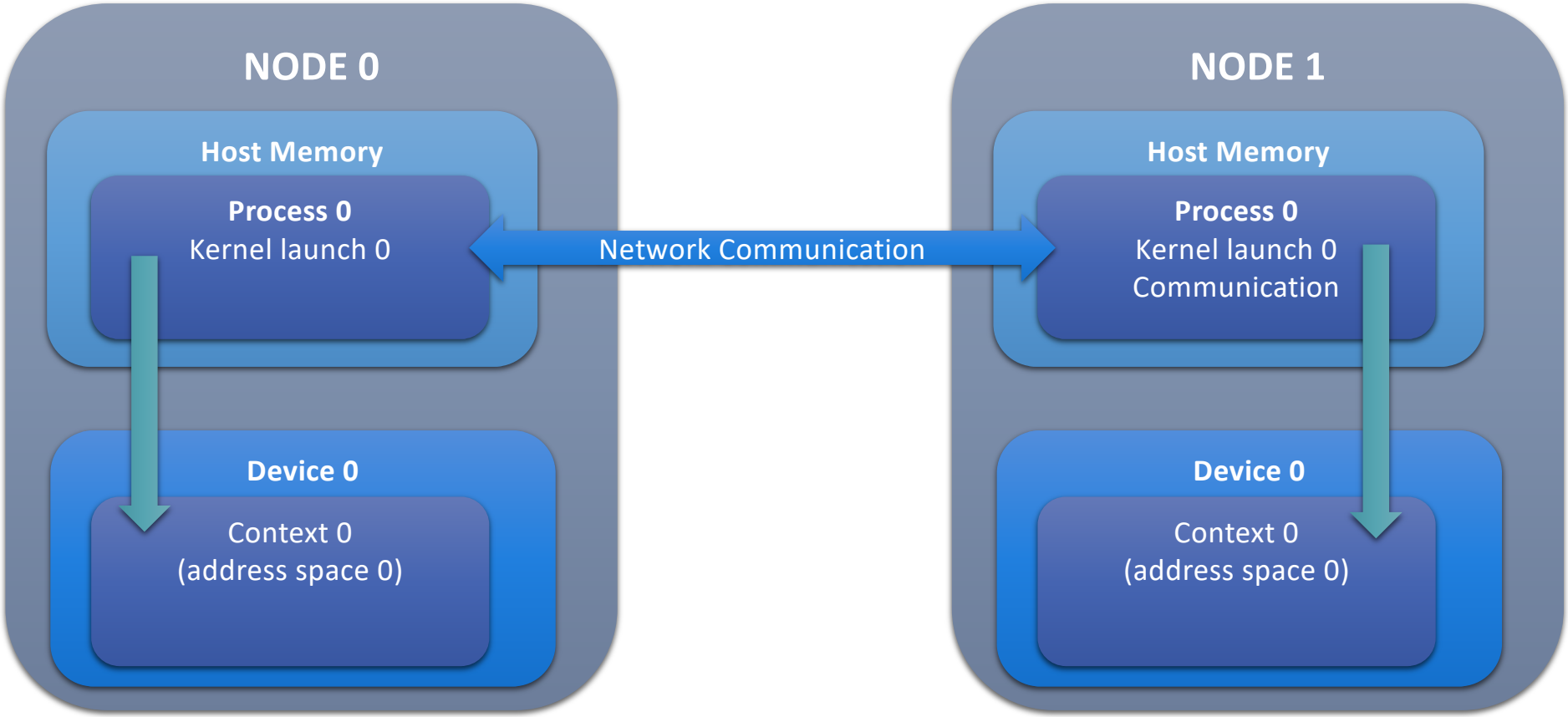
Multi-GPUs with 1 Process



Multi-GPUs with Multiple Processes

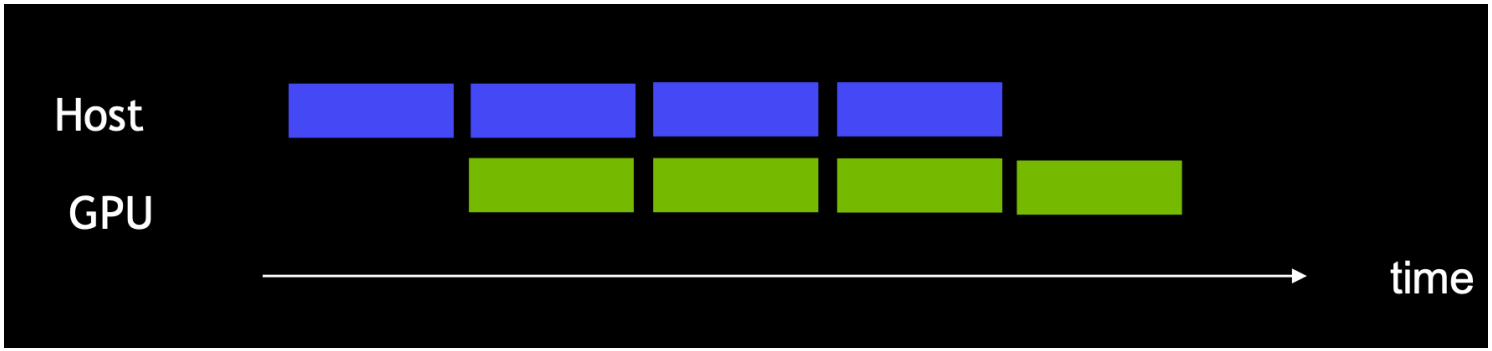


Multi-GPU – Multiple processes - Distributed



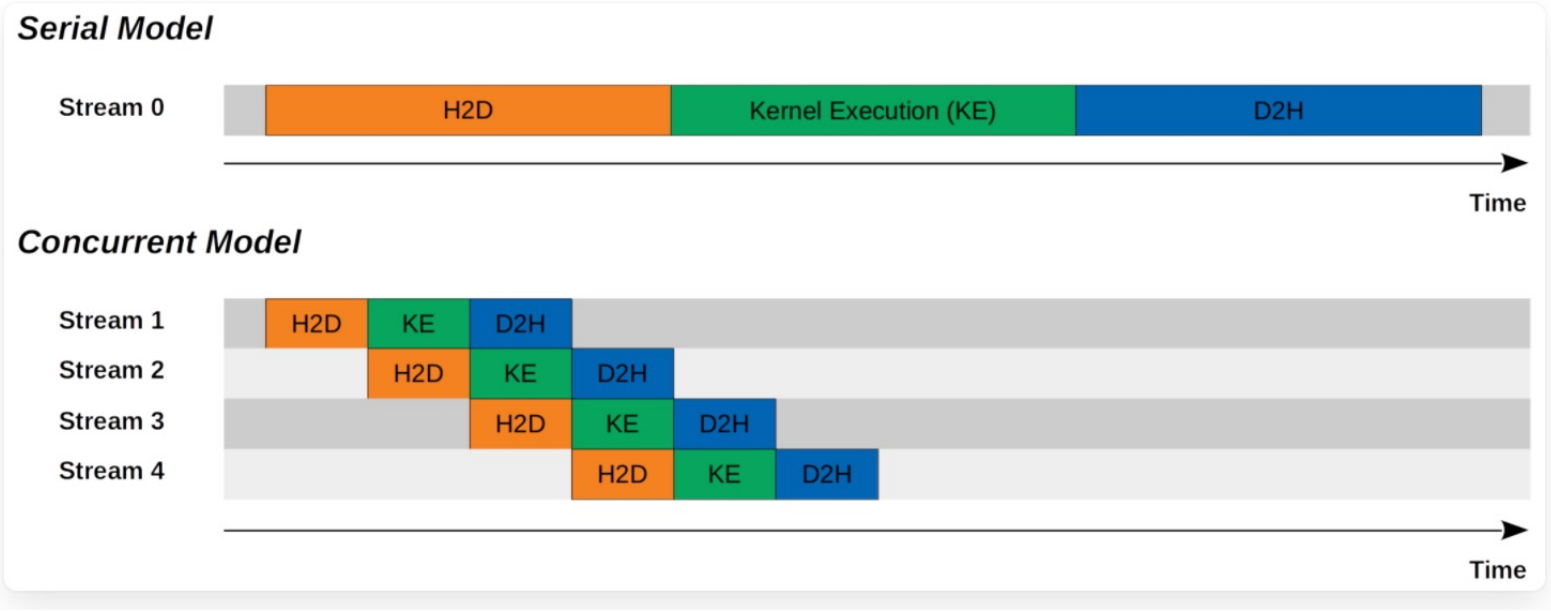
Synchronicity in CUDA

- All CUDA calls are either synchronous or asynchronous w.r.t the host
  - Synchronous: enqueue work and wait for completion
  - Asynchronous: enqueue work and return immediately
- Kernel Launches are asynchronous



CUDA Streams: Best Practices and Common Pitfalls

Serial vs concurrent execution with Streams



https://leimao.github.io/blog/CUDA-Stream/

torch.cuda.Stream

- *torch.cuda.Stream() c*reates a stream
  - i.e. a linear sequence of execution that belongs to a specific device
- Completion can be verified with *.query()*
- Without explicit stream, default stream is used: pytorch handles all synchronization (such as *synchronize()* or *wait_stream()*) with data
- **With explicit streams: user is expected to do synchronization**

```
cuda = torch.device('cuda')
s = torch.cuda.Stream()  # Create a new stream.
A = torch.empty((100, 100), device=cuda).normal_(0.0, 1.0)
with torch.cuda.stream(s):
   # sum() may start execution before normal_() finishes!!!
   B = torch.sum(A)
```

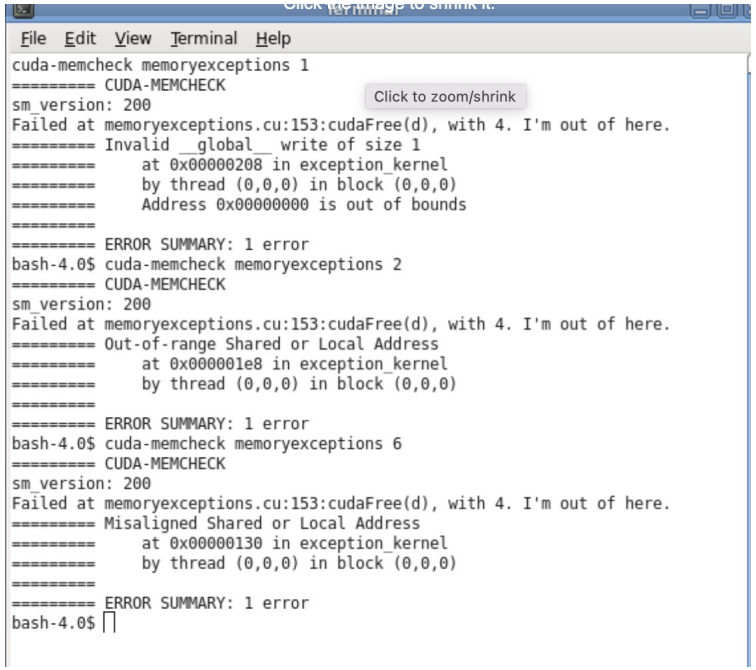
Missing a synchronization!

CUDA Profiling and Debugging

CUDA-MEMCHECK

- Memory debugging tool
  - No recompilation necessary

```
$ cuda-memcheck ./exe
```



- Can detect the following errors
  - Memory leaks
  - Memory errors (OOB, misaligned access, illegal instruction, etc)
  - Race conditions
  - Illegal Barriers
  - Uninitialized Memory
- For line numbers use the following compiler flags:

```
-Xcompiler-rdynamic-lineinfo
```

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

NVPROF

- Command Line Profiler
  - Compute time in each kernel
  - Compute memory transfer time
  - Collect metrics and events
  - Support complex process hierarchy's
  - Collect profiles for NVIDIA Visual Profiler
  - No need to recompile

https://developer.nvidia.com/blog/cuda-pro-tip-nvprof-your-handy-universal-gpu-profiler/

CUDA-GDB (like gdb)

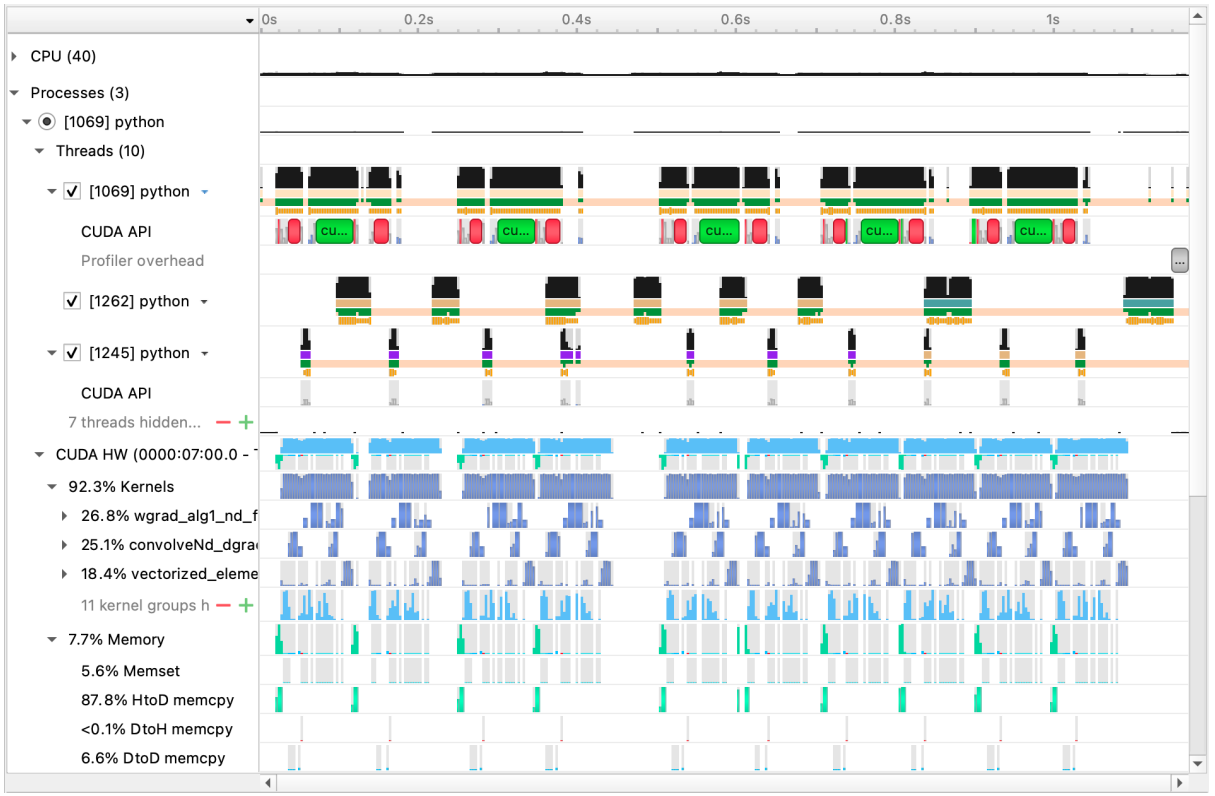
- *cuda-gdb* is an extension of GDB
  - Provides seamless debugging of CUDA and CPU code

```
$ cuda-gdb --args ./exe
```

  - Step-by-step execution, Breakpoints, etc.
- Works on Linux and Macintosh
  - For a Windows debugger use NSIGHT Visual Studio Edition
- http://docs.nvidia.com/cuda/cuda-gdb

PROFILING CODE

Using NVIDIA Nsight Systems



Using a profiler is an essential step in optimizing any code

Nsight Systems timeline provides a high-level view of your workload and helps you identify bottlenecks:

- I/O, data input pipeline
- Compute
- Scheduling (e.g. unexpected synchronization)

To generate a profile:

```
nsys profile -o myprofile python train.py
```

```
nsys profile -o myprofile -t cuda,nvtx python train.py
```

PROFILING CODE

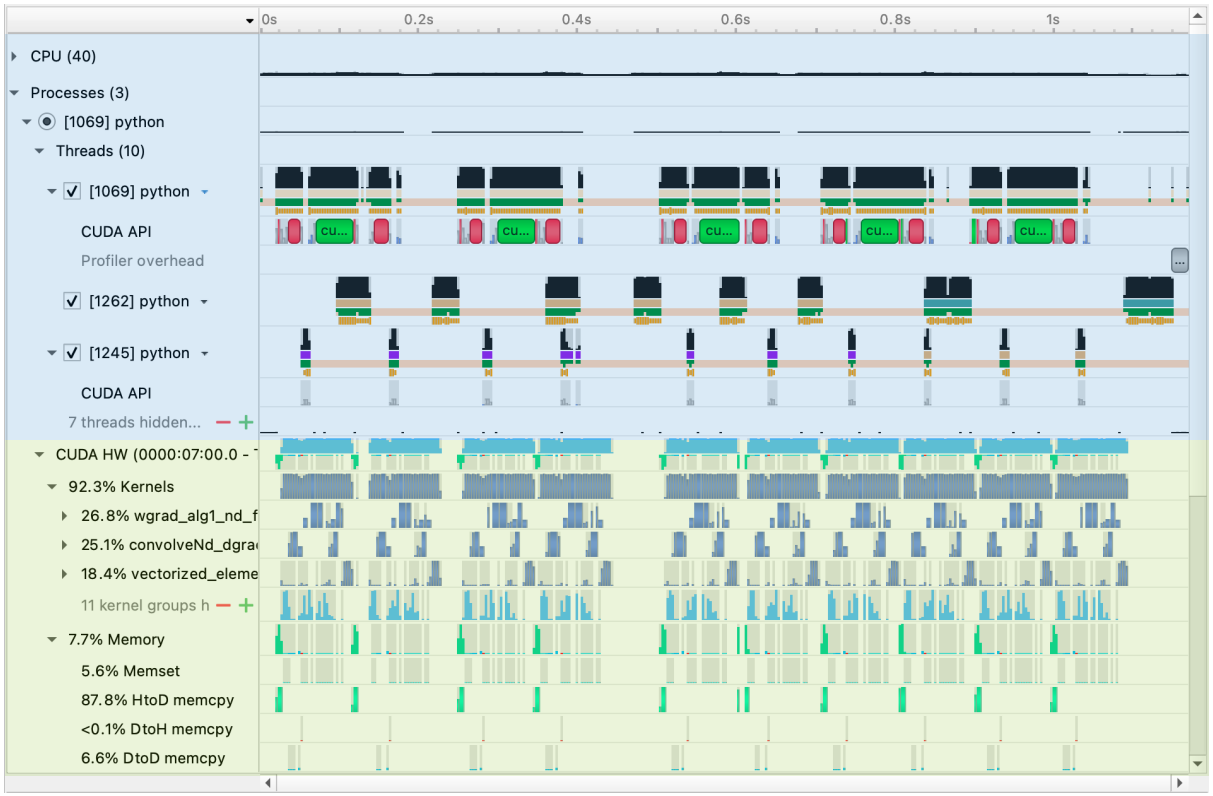
Using NVIDIA Nsight Systems

On the right is a sample of the Nsight Systems GUI view of a profile

Nsight Systems can show information about:

- CUDA API calls on CPU threads
- OS calls (if OSRT profiling enabled)
- GPU kernels and memcopies running on the GPU

Provides tons of information, but can benefit from adding additional context



PROFILING CODE

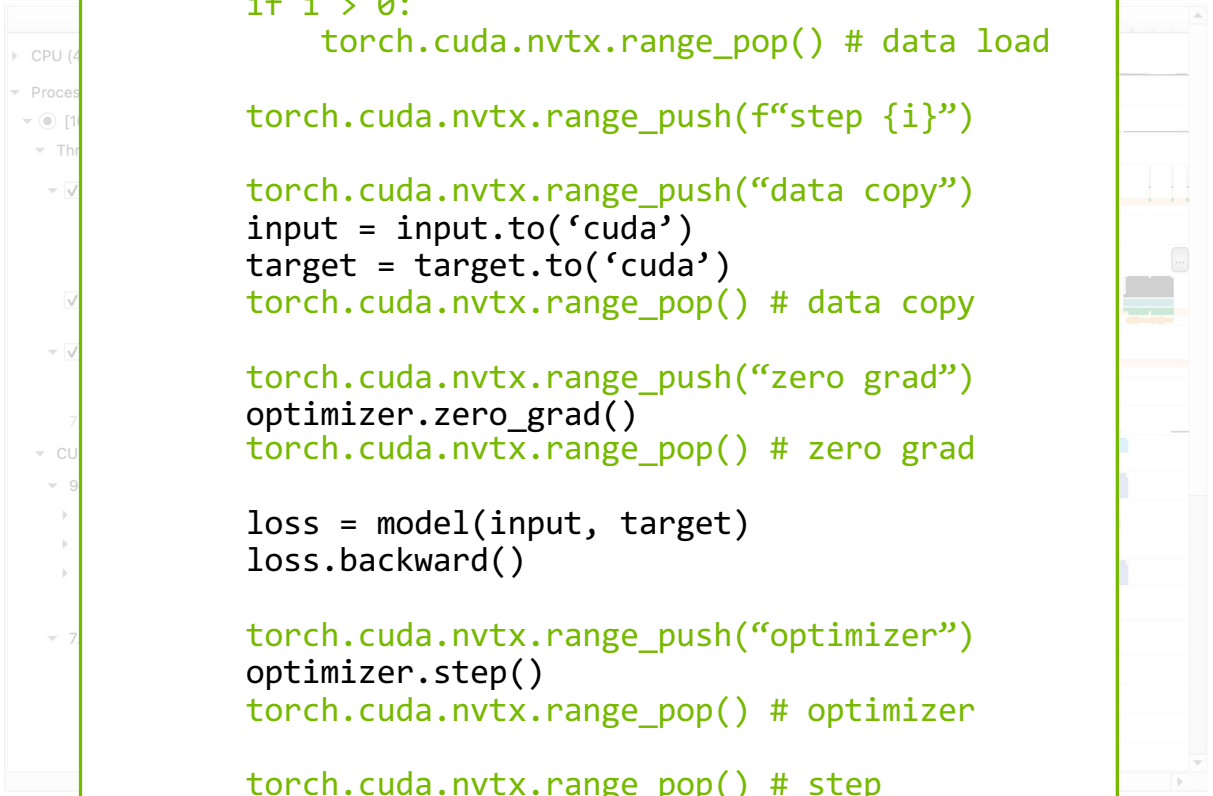
Adding NVTX Annotations

Can use NVTX ranges to annotate profiles:

```
torch.cuda.nvtx.range_push(“foo”)
torch.cuda.nvtx.range_pop()
torch.autograd.profiler.emit_nvtx()
```

NVTX ranges can be used to annotate:

- large general code regions (training step/epoch, file I/O, etc)
- targeted code locations suspected of leading to GPU idle time



```
with torch.autograd.profiler.emit_nvtx():
  for i, (input, target) in enumerate(data_loader):
        if i > 0:
            torch.cuda.nvtx.range_pop() # data load

        torch.cuda.nvtx.range_push(f“step {i}”)

        torch.cuda.nvtx.range_push(“data copy”)
        input = input.to(‘cuda’)
        target = target.to(‘cuda’)
        torch.cuda.nvtx.range_pop() # data copy

        torch.cuda.nvtx.range_push(“zero grad”)
        optimizer.zero_grad()
        torch.cuda.nvtx.range_pop() # zero grad

        loss = model(input, target)
        loss.backward()

        torch.cuda.nvtx.range_push(“optimizer”)
        optimizer.step()
        torch.cuda.nvtx.range_pop() # optimizer

        torch.cuda.nvtx.range_pop() # step

        if i < len(data_loader)-2:
            torch.cuda.nvtx.range_push(“data load”)
```

PROFILING CODE

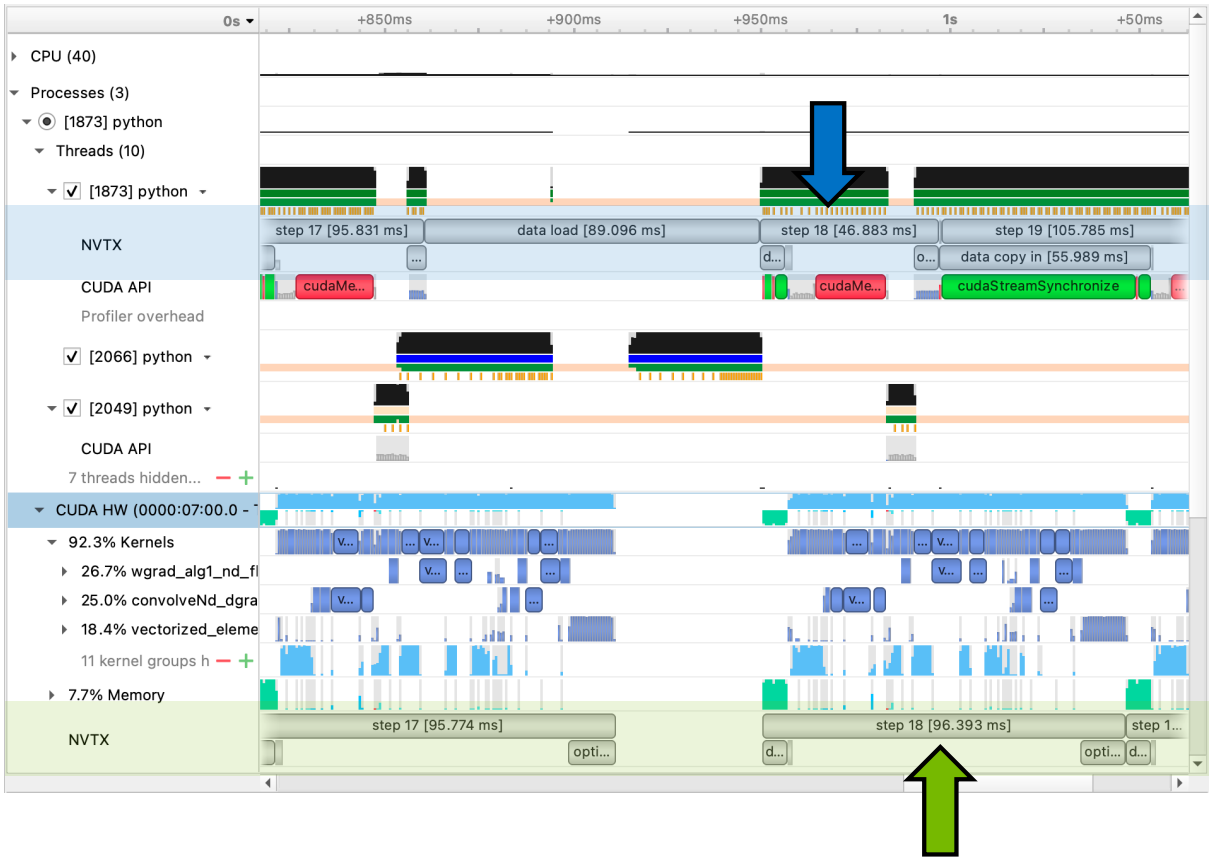
Profile Example with NVTX

Here is what the profile looks like with NVTX ranges added.

With these ranges annotated, there is a lot more context, enabling you to better focus your optimization efforts.

Ranges under CPU threads span CPU activity only (API calls/kernel launches, Python work)

Ranges under GPU span actual kernel and memcopy durations



PROFILING CODE

Profile Example with NVTX

With profiling in place, you can start addressing observed issues and track performance improvements.

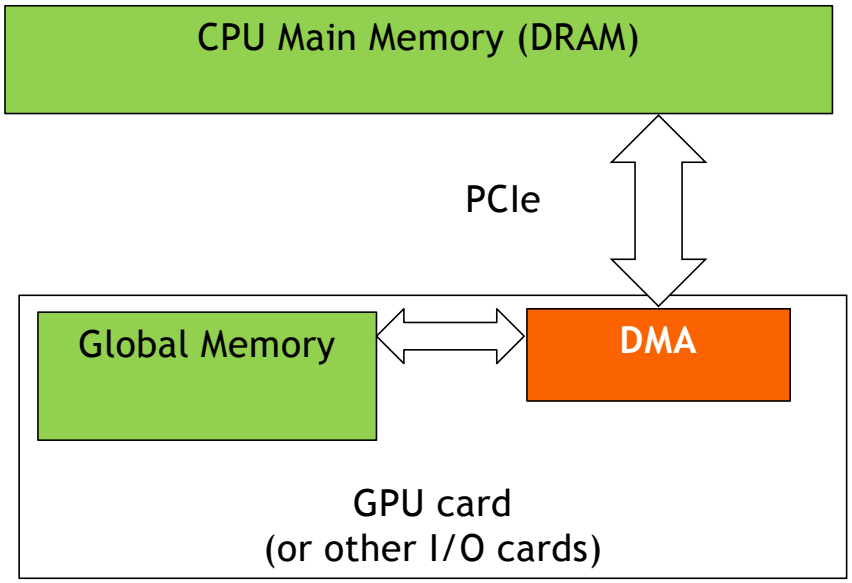
RESOURCES

- NVIDIA NSight Systems user guide (link)
- NVTX Documentation (link)

CUDA Pinned (page-locked) Memory

CPU-GPU Data Transfer using DMA

- *cudaMemcpy()* uses DMA for better performance:
  - Frees CPU for other tasks
  - Hardware unit specialized to transfer a number of bytes requested by OS
  - Between physical memory address space regions (some can be mapped I/O memory locations)
  - Uses system interconnect, typically PCIe in today's systems



Virtual Memory Management

- Virtual memory management:
  - Many virtual memory spaces mapped into a single physical memory
  - Virtual addresses (pointer values) are translated into physical addresses
- Not all variables and data structures are always in the physical memory
  - Each virtual address space is divided into pages that are mapped into and out of the physical memory
  - Virtual memory pages can be mapped out of the physical memory (page-out) to make room
  - Whether or not a variable is in the physical memory is checked at address translation time

Data Transfer and Virtual Memory

- DMA uses physical addresses
  - When `cudaMemcpy()` copies an array, it is implemented as one or more DMA transfers
  - Address is translated and page presence checked for the entire source and destination regions at the beginning of each DMA transfer
  - No address translation for the rest of the same DMA transfer so that high efficiency can be achieved

- The OS could accidentally page-out the data that is being read or written by a DMA and page-in another virtual page into the same physical location

Pinned (page-locked) Memory and DMA Data Transfer

- **Pinned memory** uses virtual memory pages that are specially marked so that they **cannot be paged out**
- Allocated with a special system API function call
- a.k.a. Page Locked Memory, Locked Pages, etc.
- CPU memory that serve as the source or destination of a DMA transfer must be allocated as pinned memory

CUDA data transfer uses pinned memory.

- The DMA used by *cudaMemcpy()* requires that any source or destination in the host memory is allocated as pinned memory
- If a source or destination of a *cudaMemcpy()* in the host memory is not allocated in pinned memory, it needs to be first copied to a pinned memory – extra overhead
- *cudaMemcpy()* is faster if the host memory source or destination is allocated in pinned memory since no extra copy is needed

Allocate/Free Pinned Memory

- *cudaHostAlloc(),* three parameters
  - Address of pointer to the allocated memory
  - Size of the allocated memory in bytes
  - Option – use cudaHostAllocDefault for now

- cudaFreeHost(), one parameter
  - Pointer to the memory to be freed

Using Pinned Memory in CUDA

- Use the allocated pinned memory and its pointer the same way as those returned by malloc()
- The only difference is that the allocated memory cannot be paged by the OS
- The cudaMemcpy() function should be about 2X faster with pinned memory
- Pinned memory is a limited resource
  - over-subscription can have serious consequences

Pinned memory allocation Example

```
int main()
{
   float *h_A, *h_B, *h_C;
…
   // Allocate pinned memory
   cudaHostAlloc((void **) &h_A, N* sizeof(float), cudaHostAllocDefault);
   cudaHostAlloc((void **) &h_B, N* sizeof(float), cudaHostAllocDefault);
   cudaHostAlloc((void **) &h_C, N* sizeof(float), cudaHostAllocDefault);
…
   // cudaMemcpy() runs faster
}
```

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 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

Part of the material have been adopted from Josh Holloway (CUDA Teaching Center)