

Introduction to High-Performance Machine Learning

Lecture 5 02/22/24

Dr. Kaoutar El Maghraoui

Dr. Parijat Dube

CUDA Basics



NVIDIA®
CUDA®

Performance Factors

Algorithms Performance

- Algorithm choice

Hyperparameters Performance

- Hyperparameters choice

Implementation Performance

- Implementation of the algorithms on top of a framework

Framework Performance

- Python, PyTorch, 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, PyTorch, CPython

Libraries Performance

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

Hardware Performance

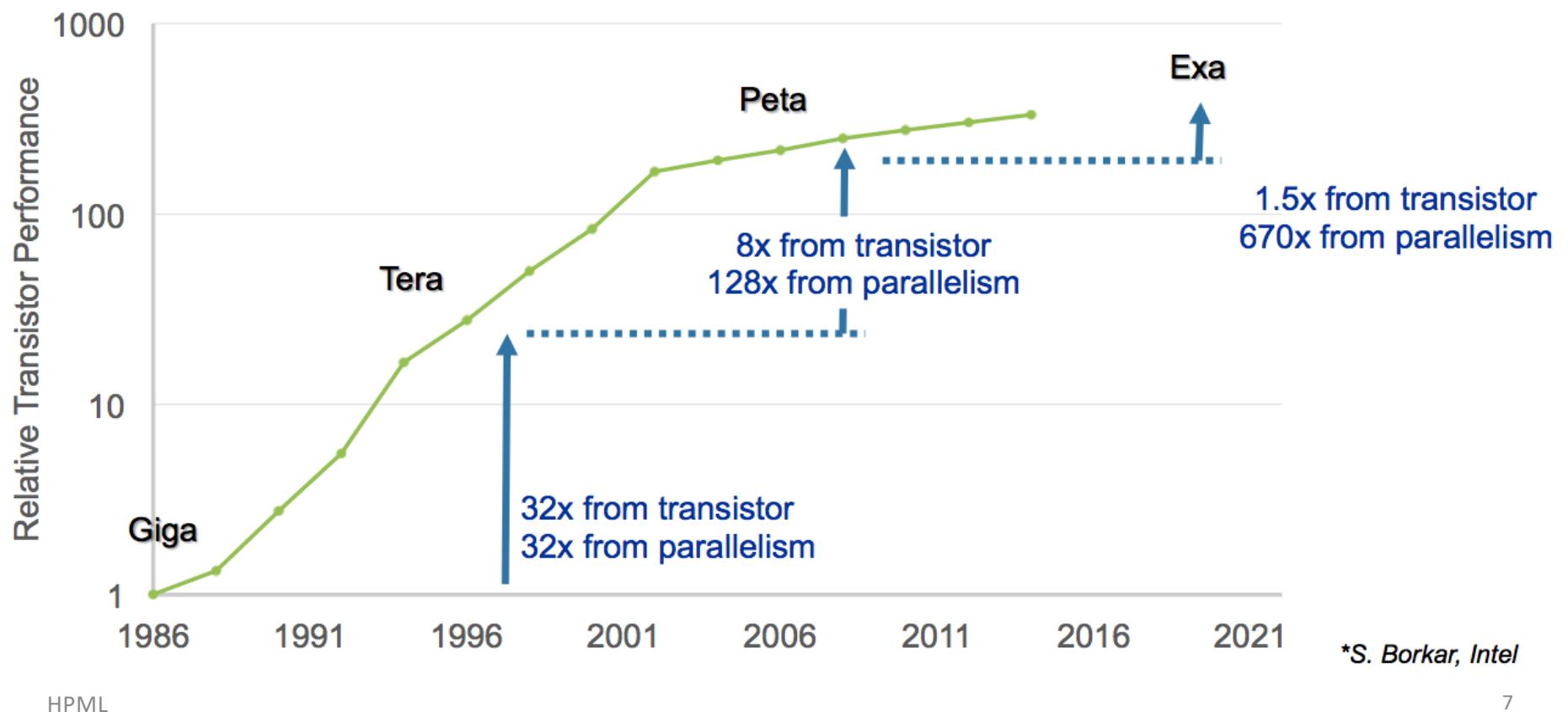
- CPU, DRAM, GPU, HBM, Tensor Units, Disk/Filesystem, Network

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 Warp Scheduling
- CUDA Context and Streams
- CUDA Pinned Memory
- CUDA Memory Access
- CUDA Compilation and Runtime:
 - CUDA Runtime, CUDA Driver, AoT and JIT compilation
- CUDA Hardware
- CUDA Profiling and Debugging
- cuDNN and cuBLAS

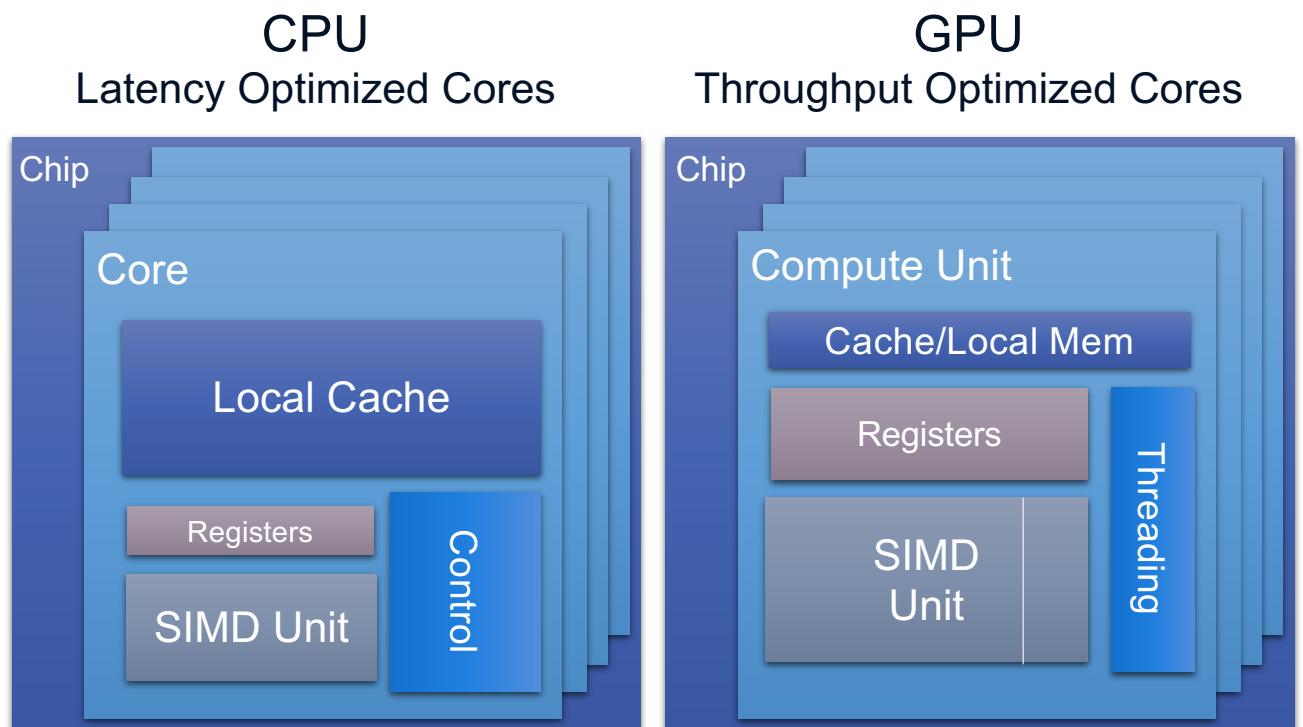
Heterogenous computing motivations

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>

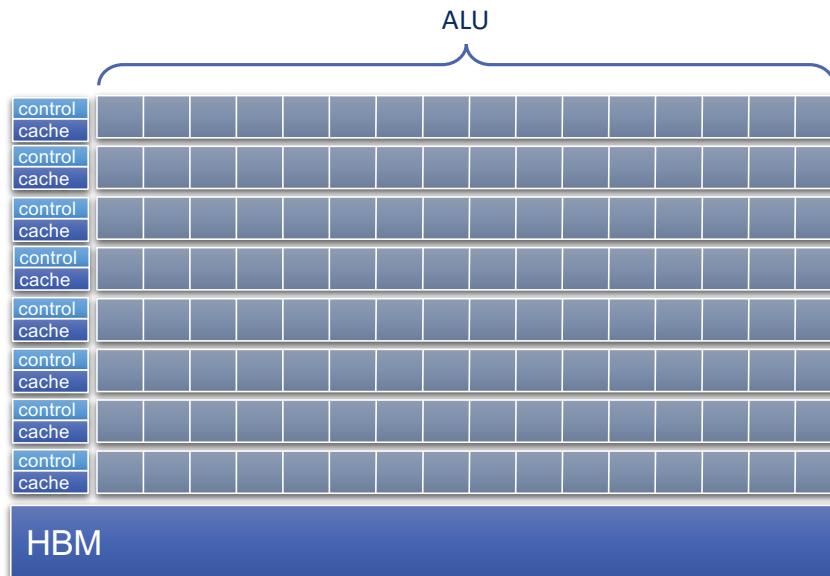
CPU Architecture



GPU: Throughput Optimized

- Many small **ALUs**
 - 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

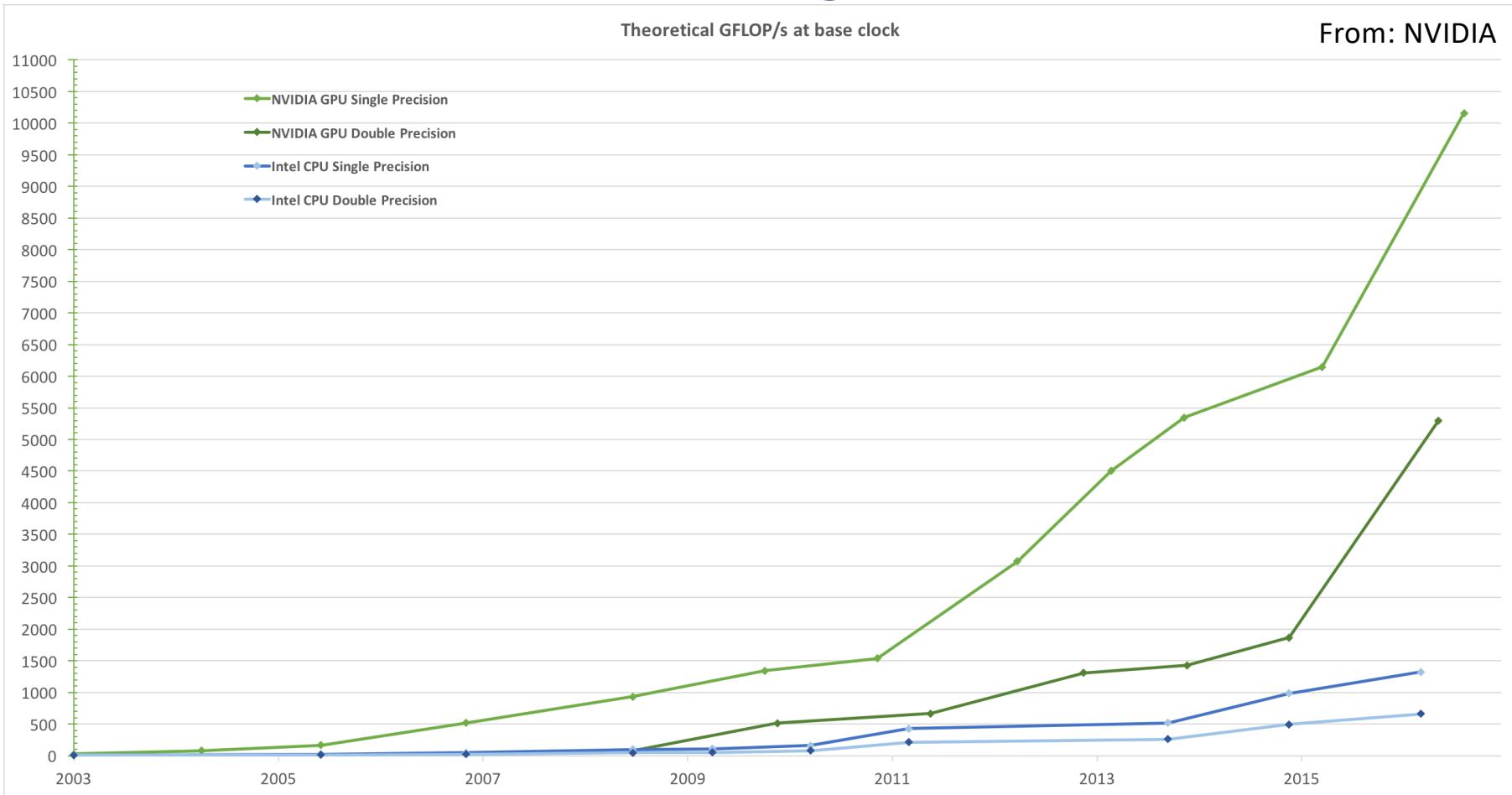
GPU Architecture



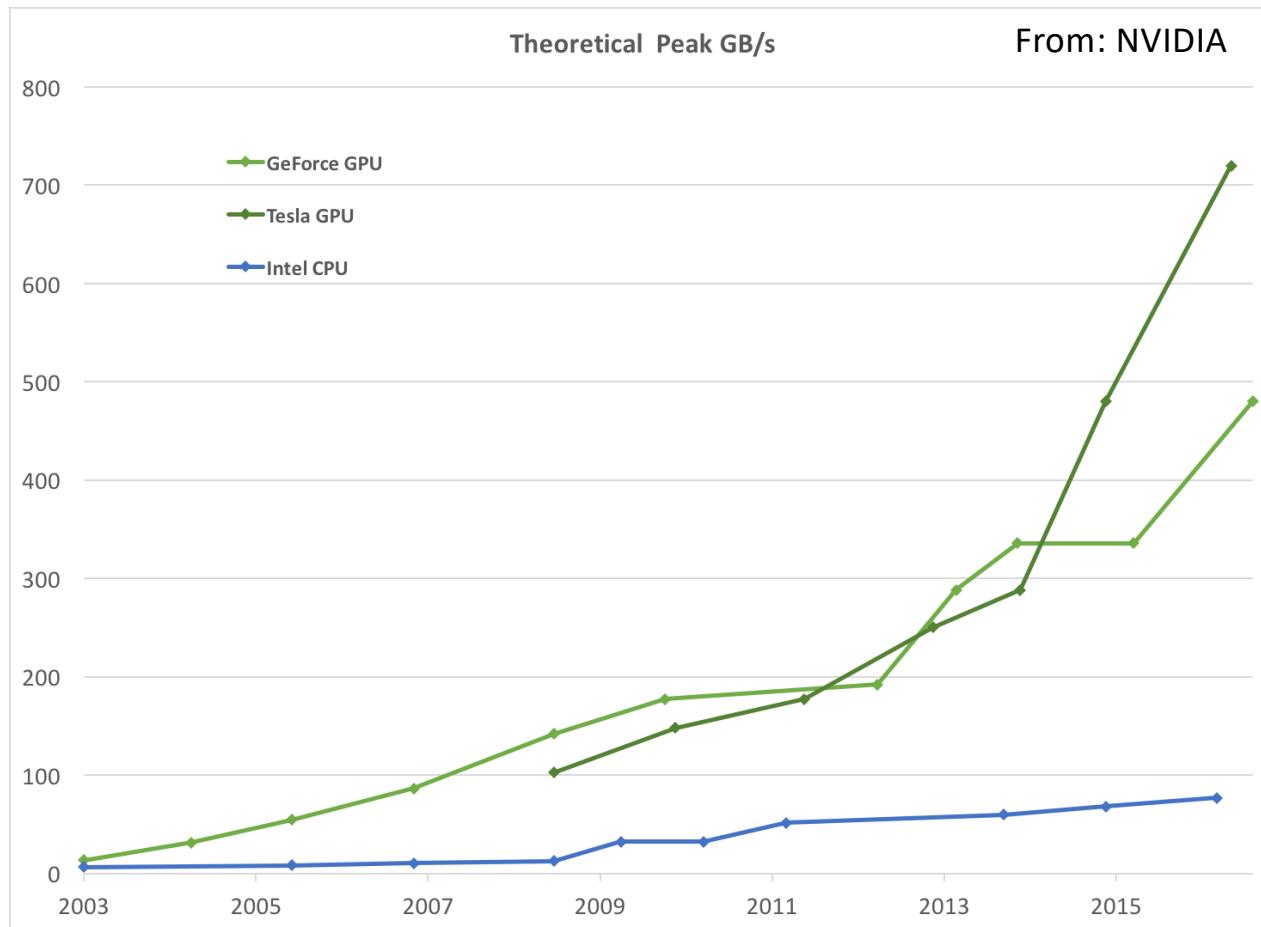
CPU/GPU Performance Comparison

- 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

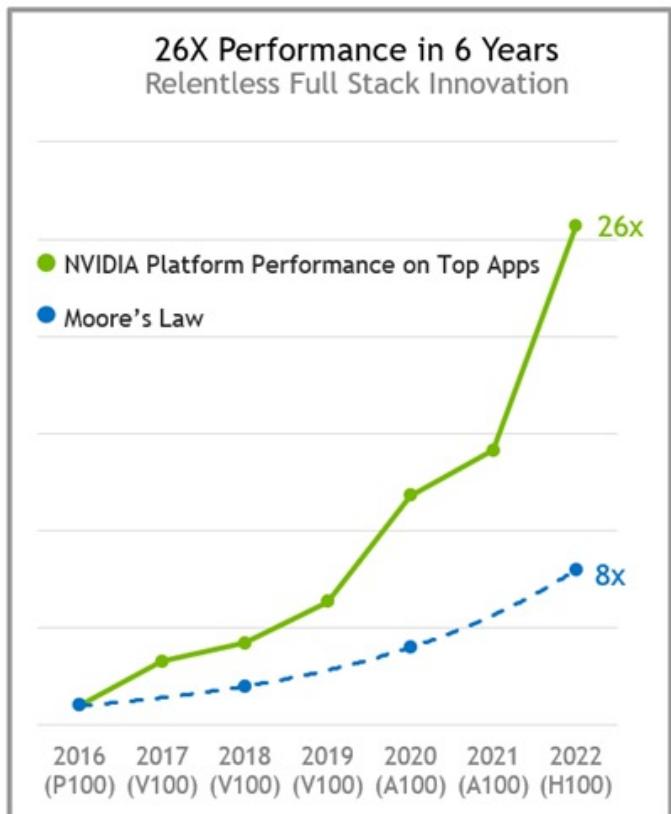
GPU Performance Advantage 1: FLOPS



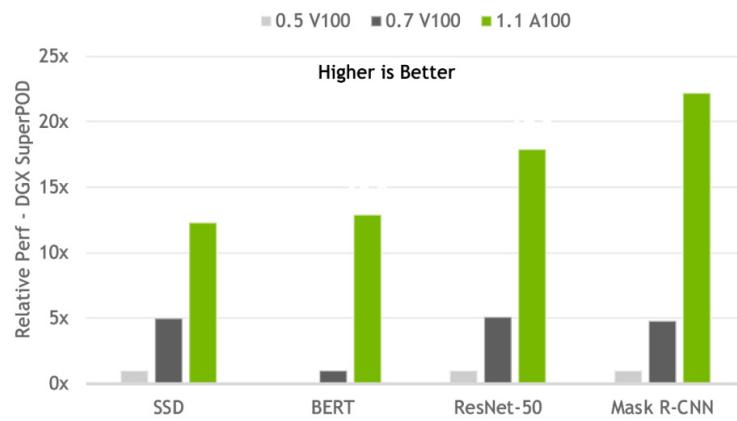
GPU Performance Advantage 2: Memory BW



NVIDIA GPU Performance Improvements Over the years



20X MORE PERFORMANCE IN 3 YEARS
NVIDIA AI Delivers Continuous Gains With SW and At-Scale Improvements



Results normalized for throughput due to higher accuracy requirements on latest round of MLPerf 1.1
MLPerf ID 0.5/0.7/1.1 comparison:
SSD: 0.5-23/0.7-53/1.1-2078 | BERT: 0.7-56/1.1-2083 | ResNet50v1.5: 0.5-17/0.7-55/1.1-2082 | Mask R-CNN: 0.5-22/0.7-48/1.1-2076 |
MLPerf name and logo are trademarks. See www.mlperf.org for more information.

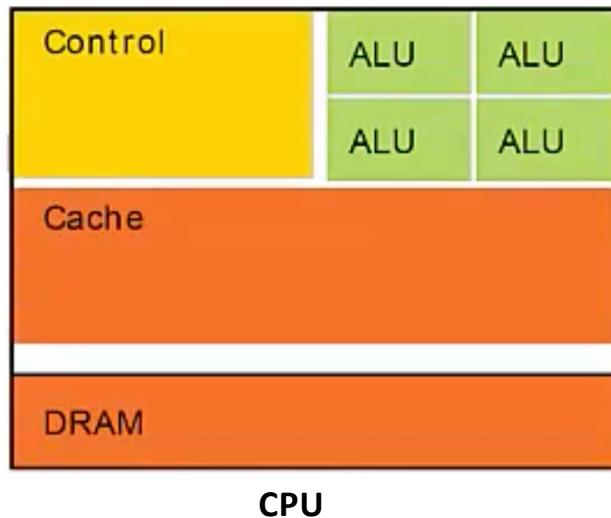
Source: <https://developer.nvidia.com/blog/fueling-high-performance-computing-with-full-stack-innovation/>

CPU vs. GPU Summary

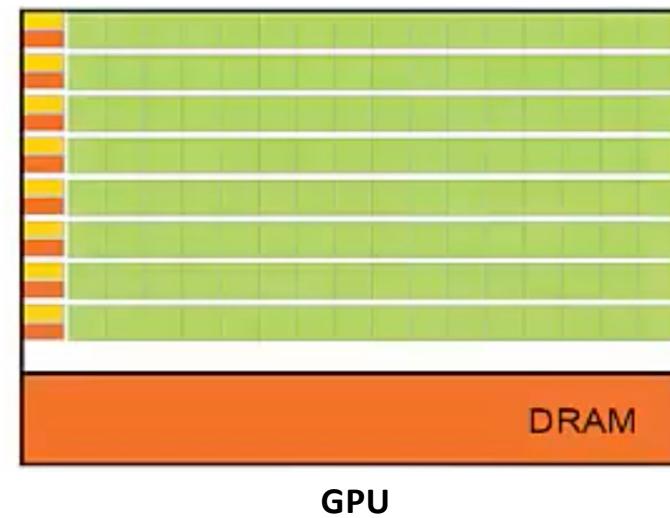
CPU	GPU
A smaller number of larger cores (up to 24)	A larger number (thousands) of smaller cores
Low latency	High throughput
Optimized for serial processing	Optimized for parallel processing
Designed for running complex programs	Designed for simple and repetitive calculations
Performs fewer instructions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

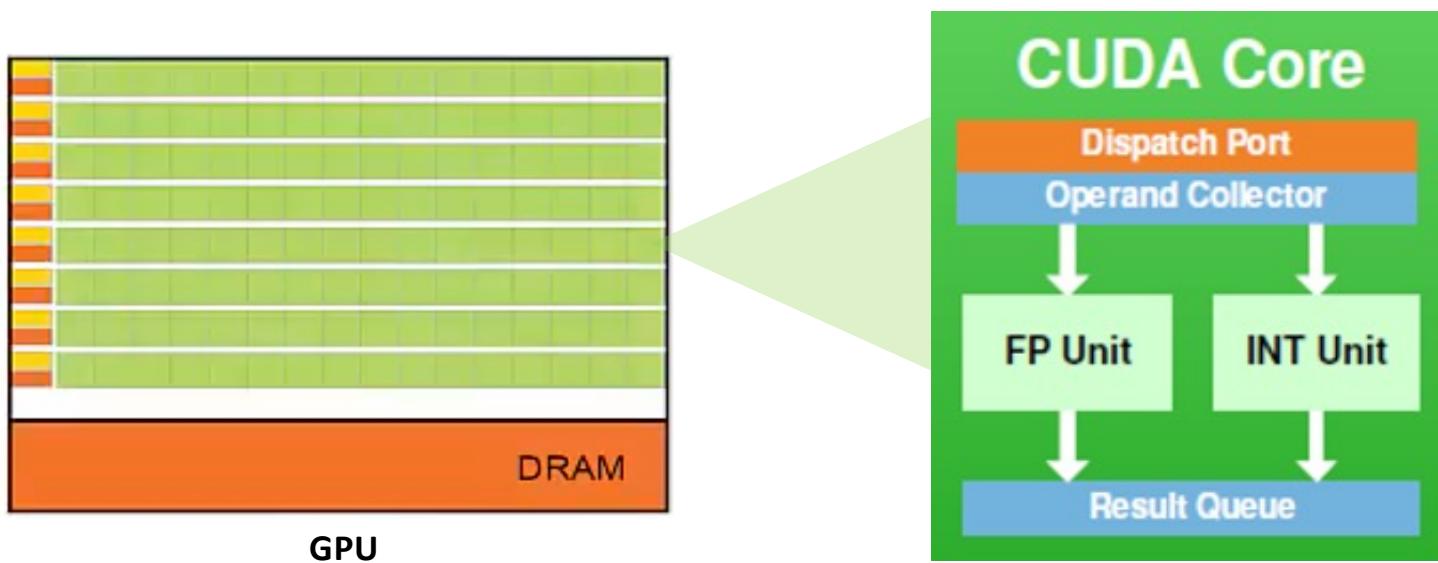


- CPUs have few strong cores
- CPU- Designed to minimize latency
 - Majority of the silicon area is dedicated to:
 - Advanced Control Logic
 - Large Cache



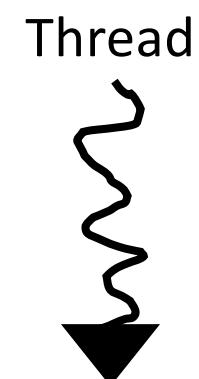
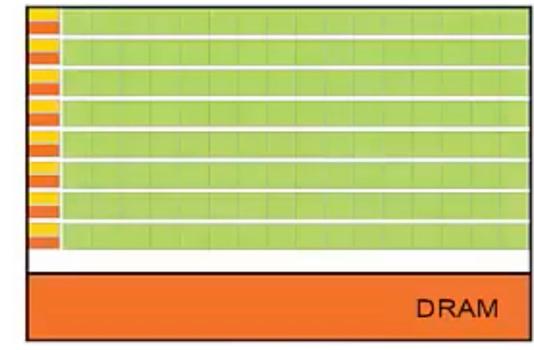
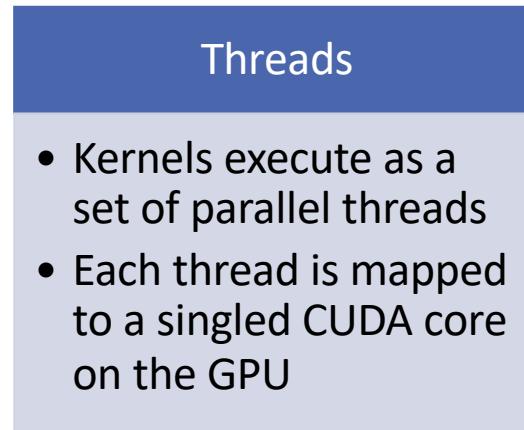
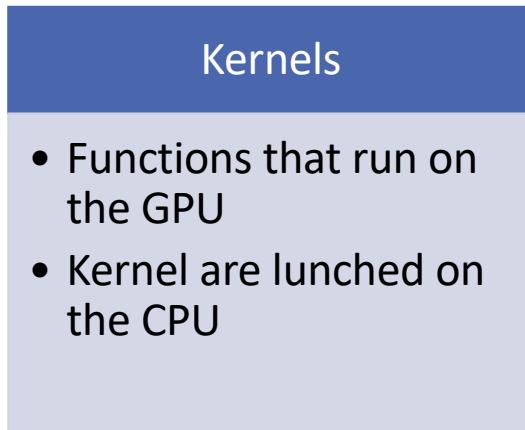
- GPUs have thousands of weaker cores
- GPU- Designed to maximize throughput
 - Majority of the silicon area is dedicated to:
 - Massive number of arithmetic units – CUDA cores

What is a CUDA Core

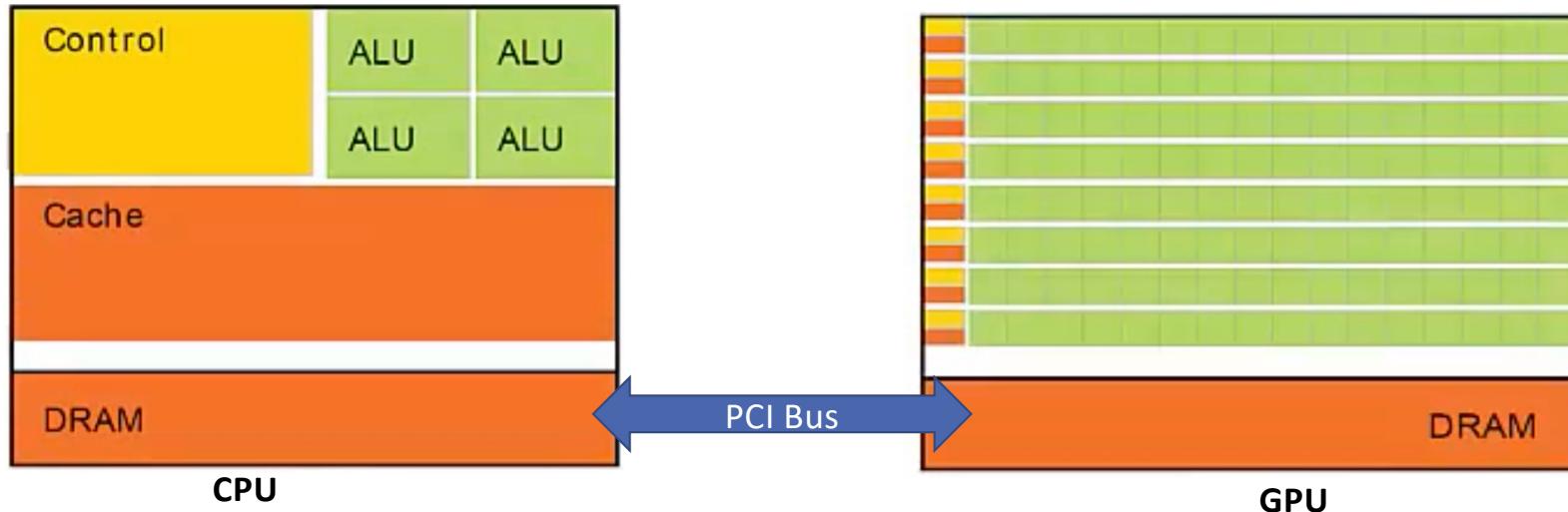


Key Concepts

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.



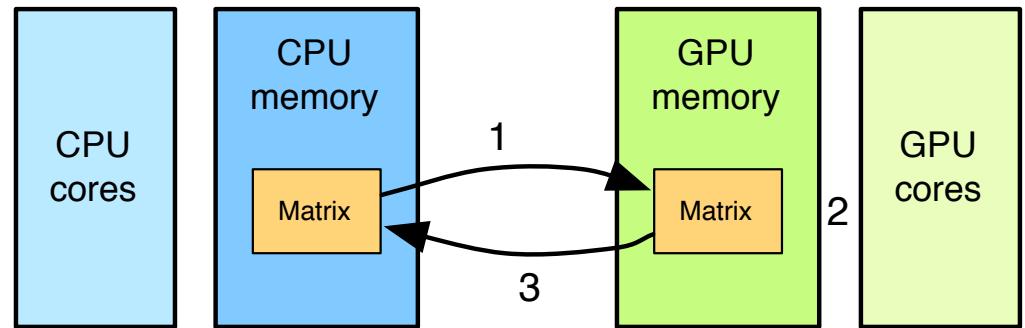
Key Concepts



Host	Device	Heterogenous	CUDA
<ul style="list-style-type: none">• CPU and its on chip memory	<ul style="list-style-type: none">• GPU + dedicated DRAM	<ul style="list-style-type: none">• Host + Device• Take the best of both worlds	<ul style="list-style-type: none">• C with extensions• Allows for heterogenous programming

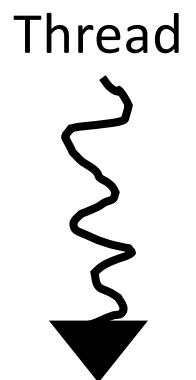
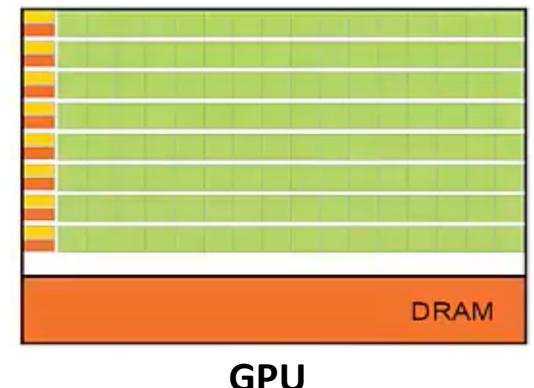
GPU Computing

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



CUDA Threads

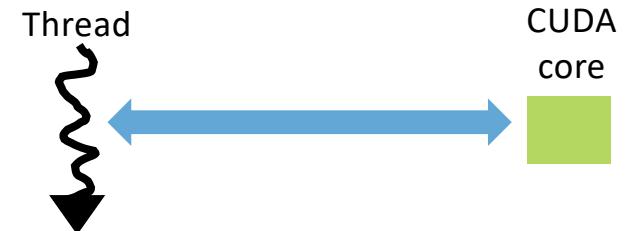
- Kernels execute as a **set of parallel threads**
- CUDA is designed to execute **thousands of threads**
- **CUDA Threads execute in a SIMD (Single Instruction Multiple Data) fashion**
 - NVIDIA calls this SIMT (Single Instruction Multiple Thread)
- **Threads are similar to data-parallel tasks**
 - Each thread performs the same operation on a subset of data
 - Threads execute independently
- **Threads do not execute at the same rate**
 - Different paths are taken in if/else statements
 - Different number of iterations in a loop, etc.



Threads Hierarchy

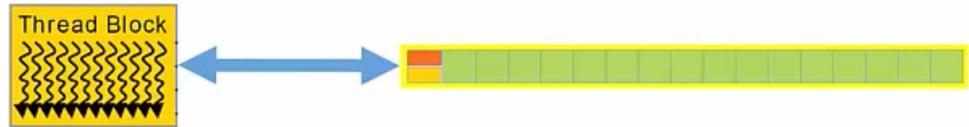
Threads

- Kernels execute as a set of Threads



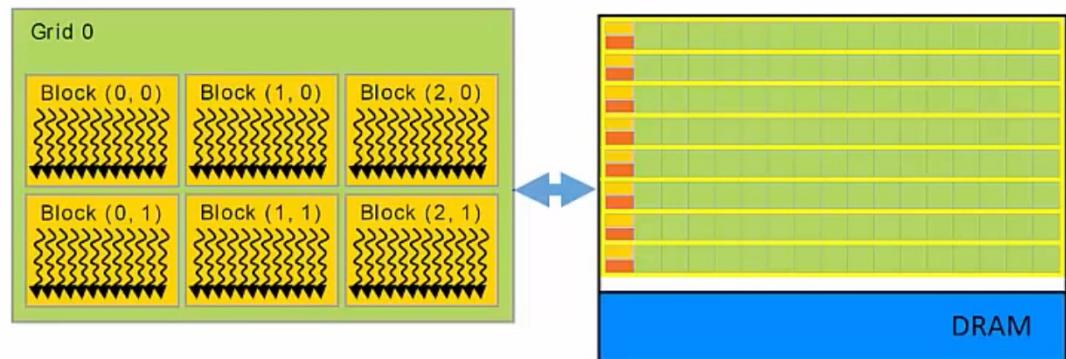
Blocks

- Threads are grouped into blocks



Grid

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

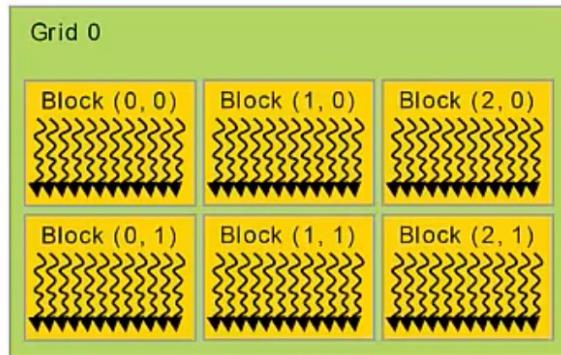


Thread \in Blocks \in Grids

Dimensions of Gids and Blocks

Grid dimension

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

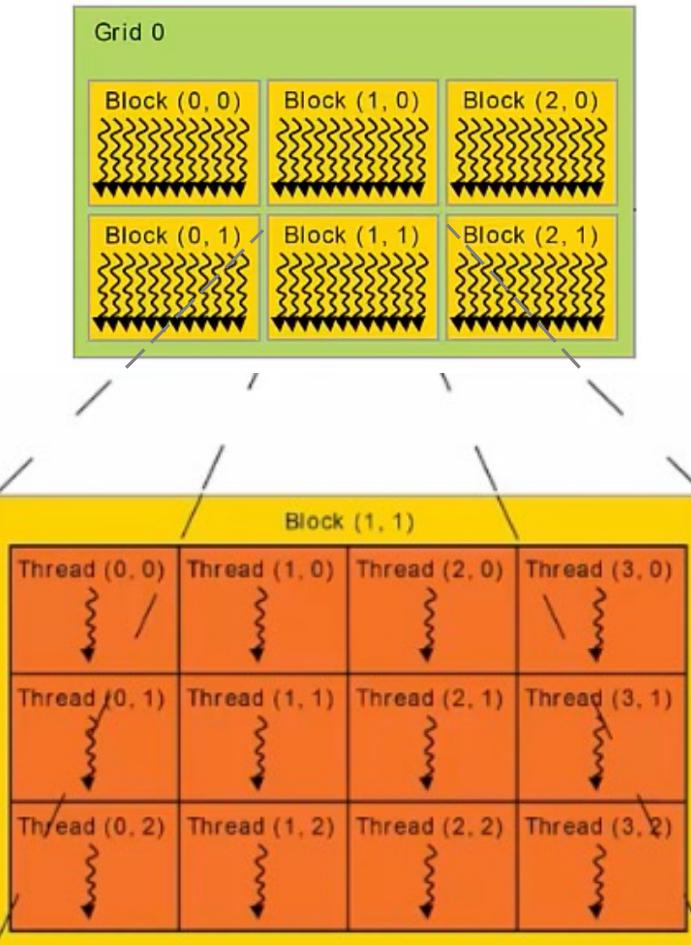


Grid dimension: 3 x 2
=> 6 block

Dimensions of Gids and Blocks

Grid dimension

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



Grid dimension

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

Grid dimension: 3 x 2

=> 6 block

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

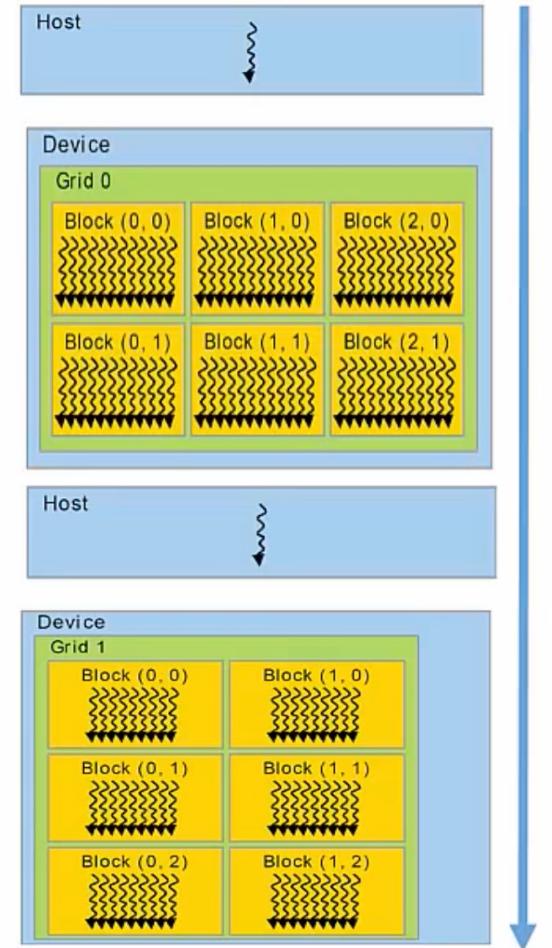
```
int main( void ) { // Host Code
    // Do sequential stuff

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

    // Do more sequential stuff

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

} return 0;
```



Kernel Launch

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

Configuration Parameters

<<<grid_size,block_size>>>

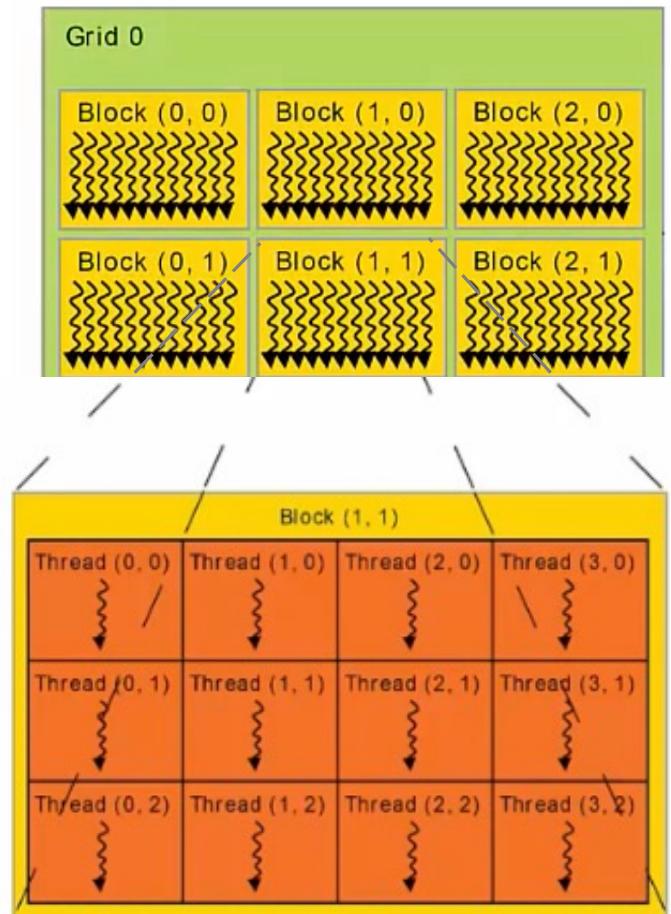
- dim3 is a CUDA data structure
- Default values are (1,1,1)

```
// Launch Kernel
```

```
KernelName<<< grid_size, block_size >>>( . . . );
```

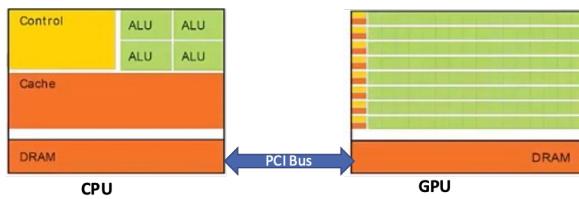
Kernel Launch Configuration Example

```
// 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 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(...);

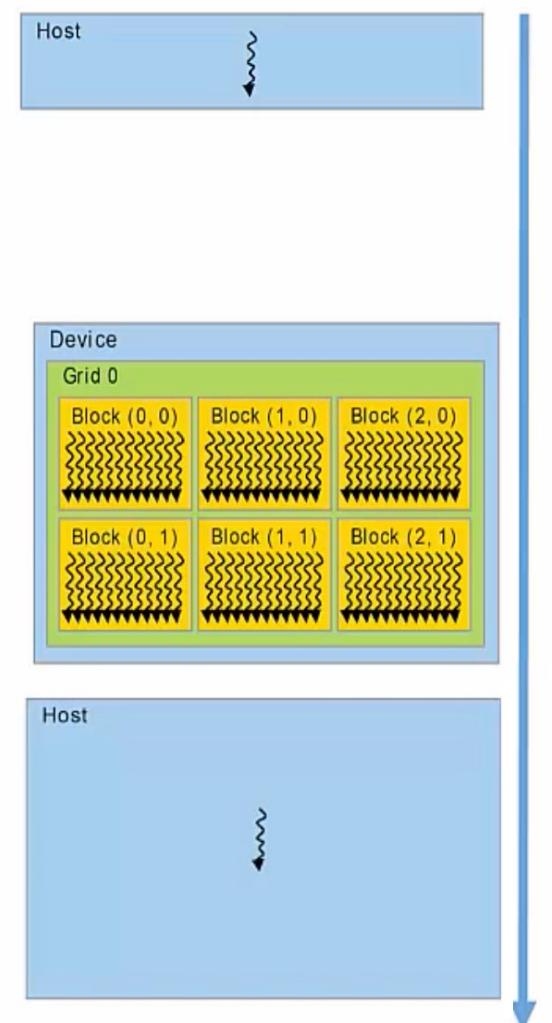
    // Copy data from Host to Device
    cudaMemcpy(...);

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

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

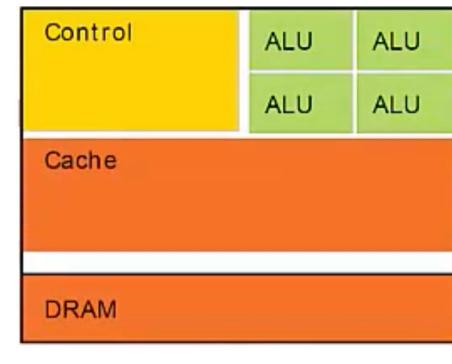
    // Do sequential stuff
    // ...
}

return 0;
```

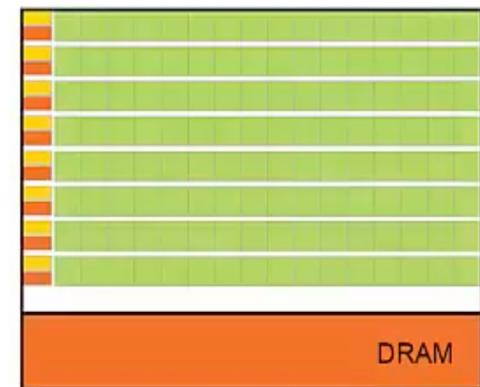


Allocation Device Memory

- 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

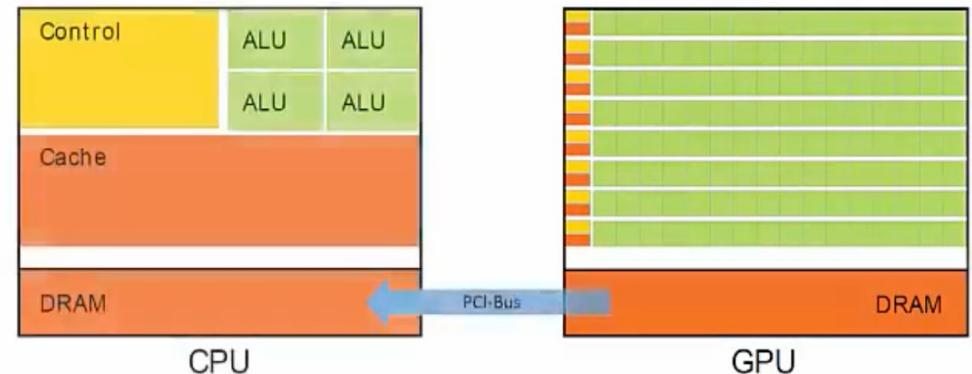


GPU

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;  
}
```

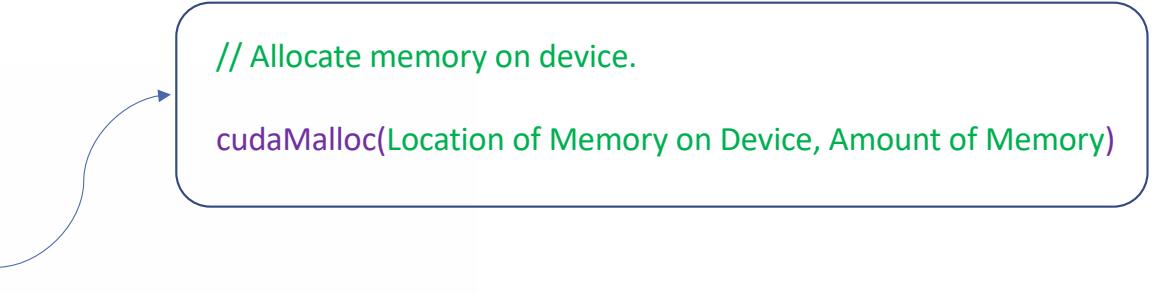
// Declare variable

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;  
}
```



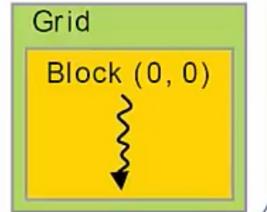
// Allocate memory on device.
cudaMalloc(Location of Memory on Device, Amount of Memory)

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;  
}
```

Configuration Parameters <<<1,1>>>

Grid Dimension: $1 \times 1 \times 1$
⇒ 1 Block
Block Dimension: $1 \times 1 \times 1$
⇒ 1 Thread



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

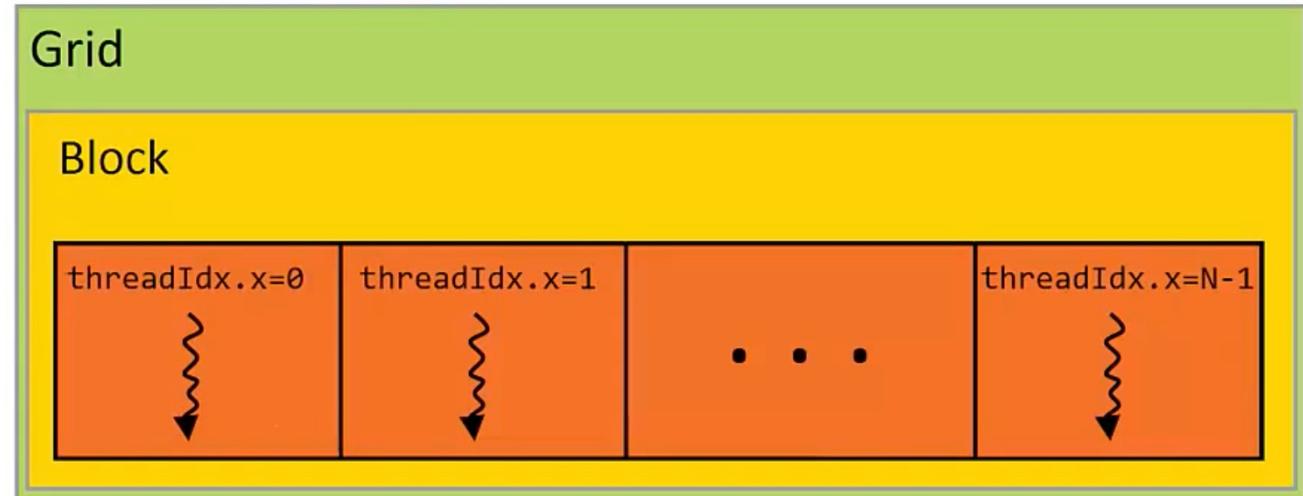
- Each Thread has its own thread index
 - Accessible within a Kernel through the built in `threadIdx` variable
- Thread Blocks can have as many as 3-dimensions, therefore there is a corresponding index for each dimension:

`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.

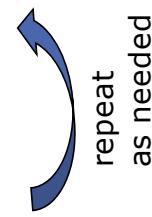
Traditional Program Structure in CUDA

- 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();

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**

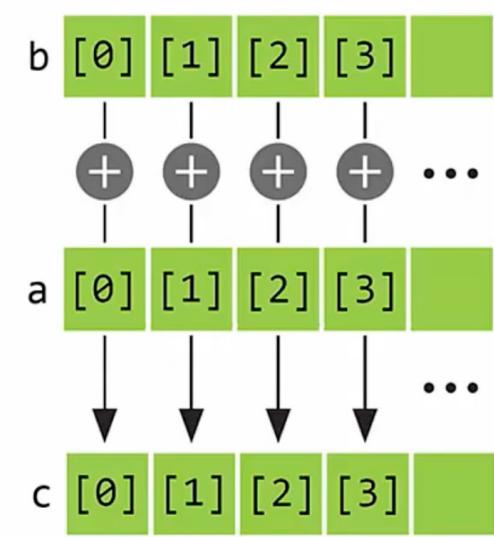
```
cudaDeviceSynchronize();
```

Vector Addition – A Very Parallel Problem

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

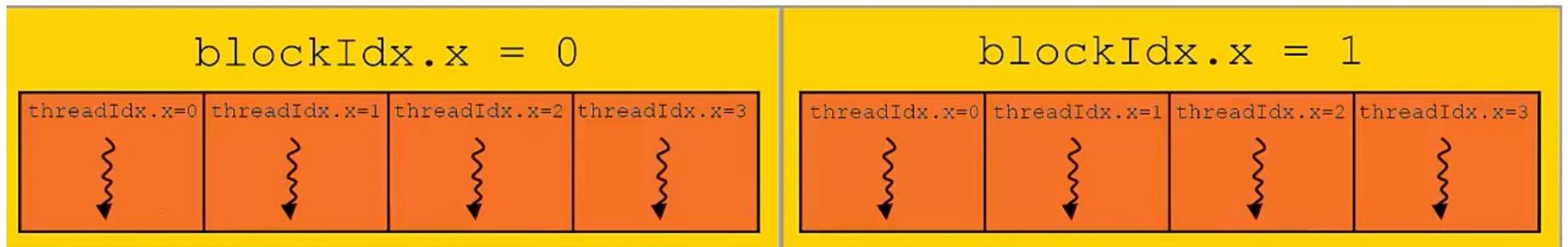
$$\mathbf{a} + \mathbf{b} = \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \\ \vdots \\ a_{N-1} \end{bmatrix} + \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \\ \vdots \\ b_{N-1} \end{bmatrix}$$

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

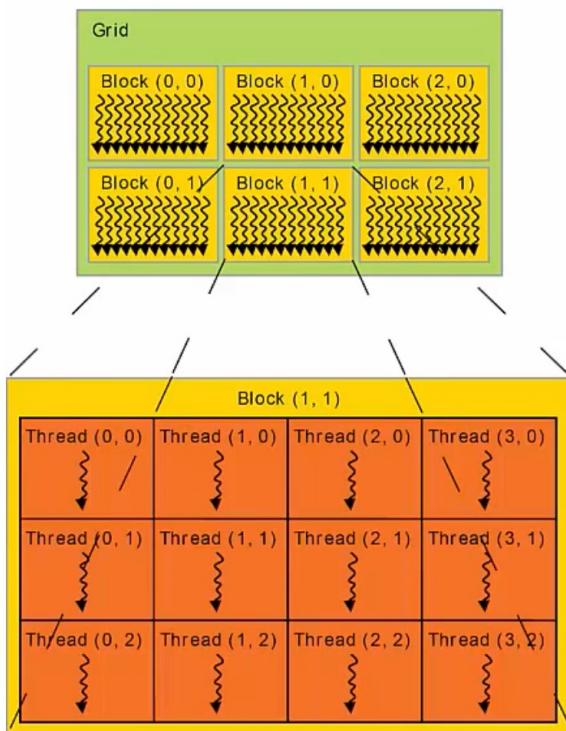


Thread Index

- 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;
```

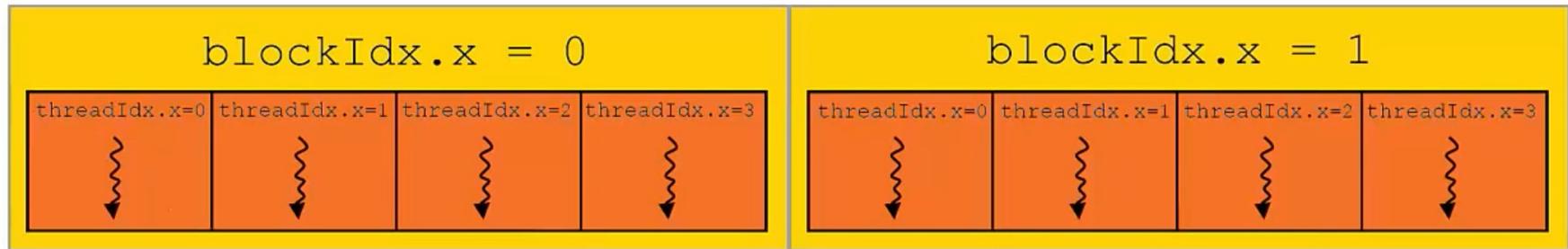
Index of a Thread

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

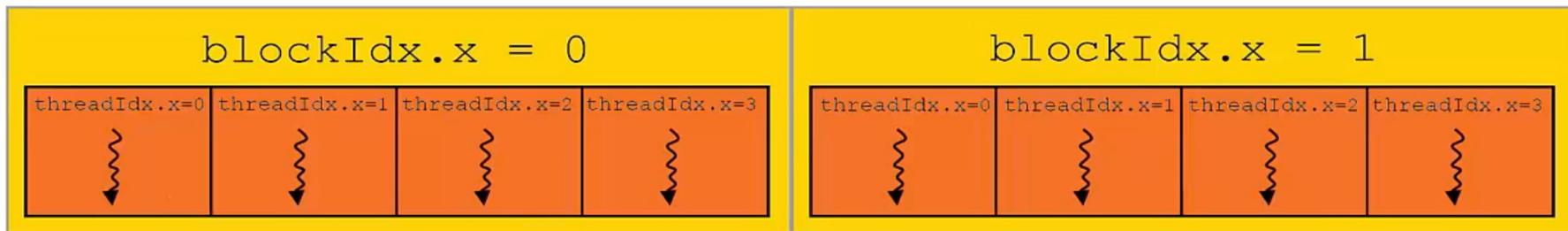
Indexing Within Grid

- **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;`

<code>i</code>	<code>threadIdx.x</code>	<code>blockIdx.x * blockDim.x</code>
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;  
}
```

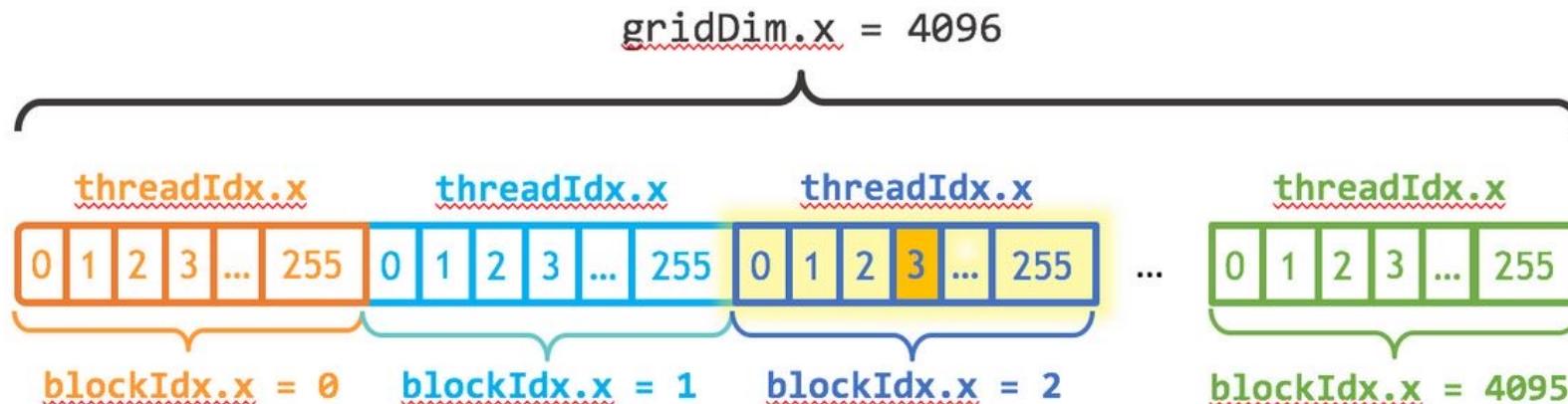
a: 0 0 0 0 1 1 1 1 2 2 2 2

```
_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`

Indexing and Memory Access

- Images are 2D data structures
 - height x width
 - $\text{Image}[j][i]$, where $0 \leq j < \text{height}$, and $0 \leq i < \text{width}$

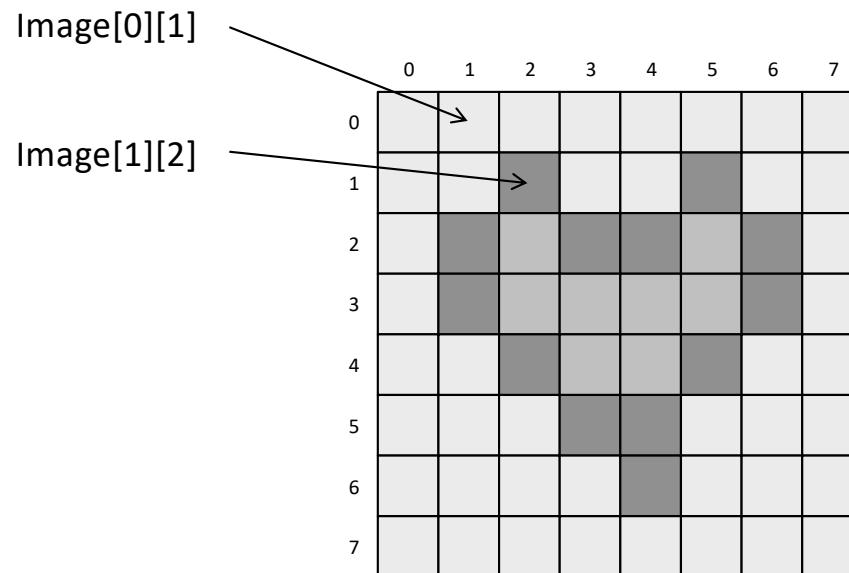
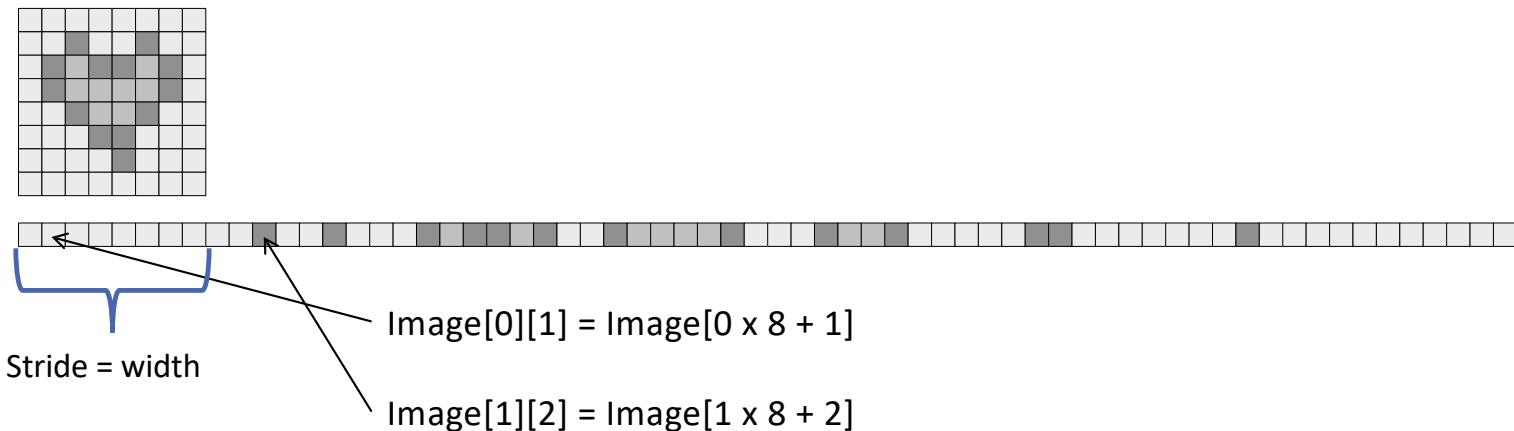


Image Layout in Memory

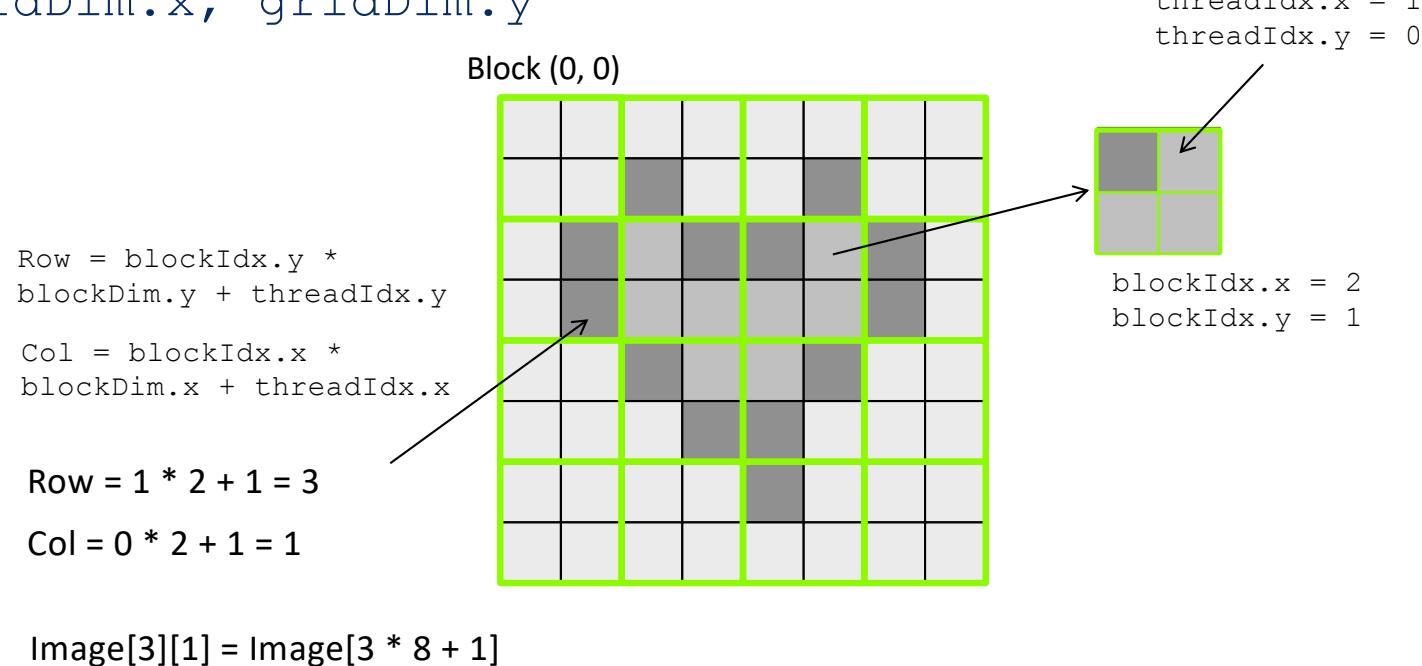
- Row-major layout
- $\text{Image}[j][i] = \text{Image}[j \times \text{width} + i]$



Indexing and Memory Access: 2D Grid

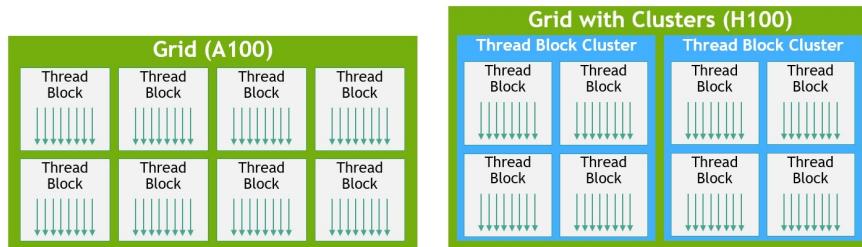
- 2D blocks

- `gridDim.x, gridDim.y`



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)



NVIDIA H100: Thread Block Clusters

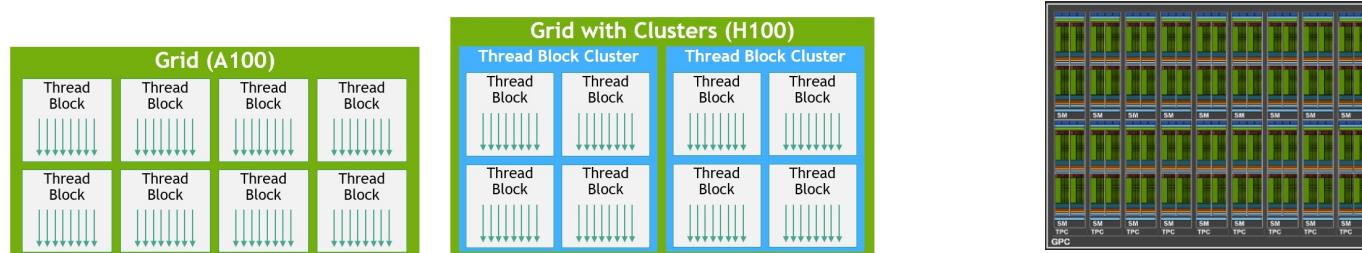
- GPUs grow beyond 100 GPU cores (SMs): a new level in the software hierarchy can improve execution efficiency
 - Programmatic control of locality at a granularity larger than a single thread block on a single SM



<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

CUDA Memory Model

Thread Memory Correspondence

Threads \Leftrightarrow Local Memory (and Registers)

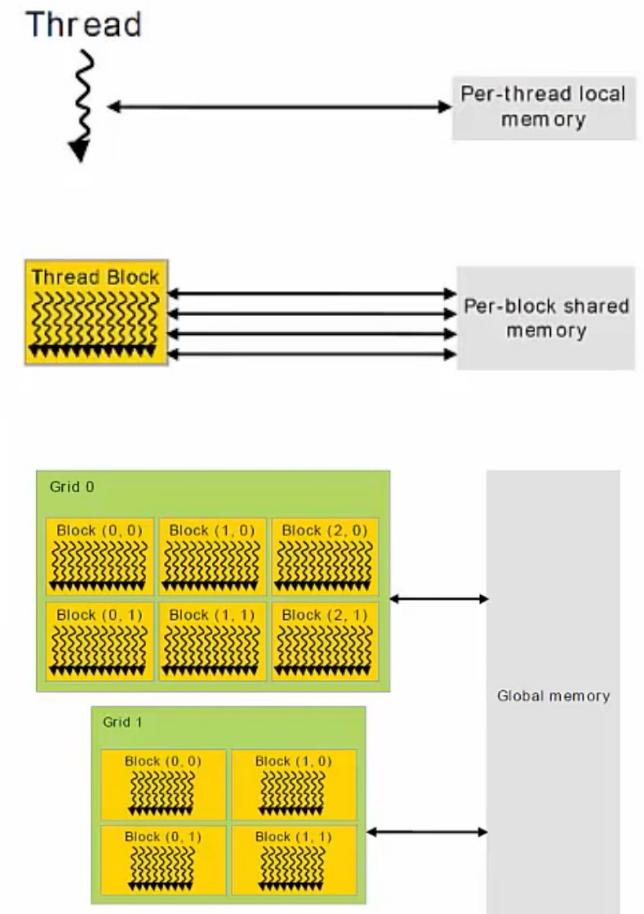
- Scope: Private to its corresponding Thread
- Lifetime: Thread

Blocks \Leftrightarrow Shared Memory

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

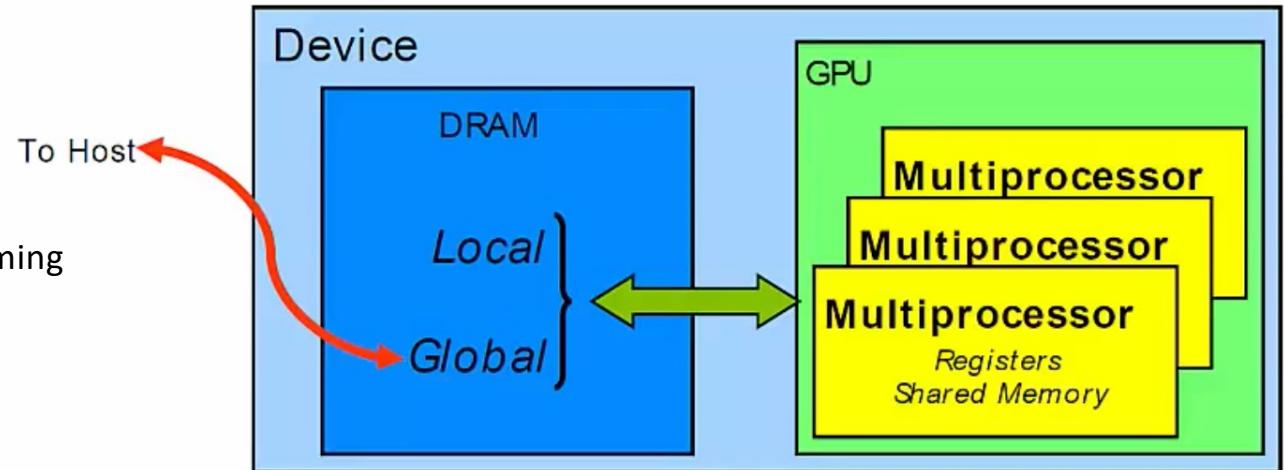
Grids \Leftrightarrow Global Memory

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

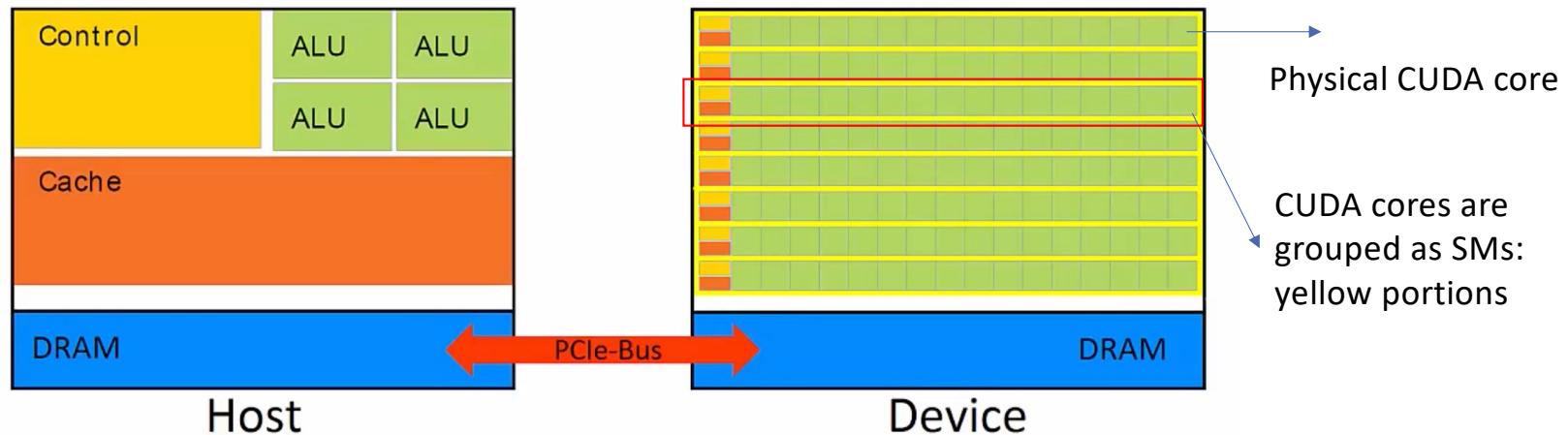


Memory Model

- 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**

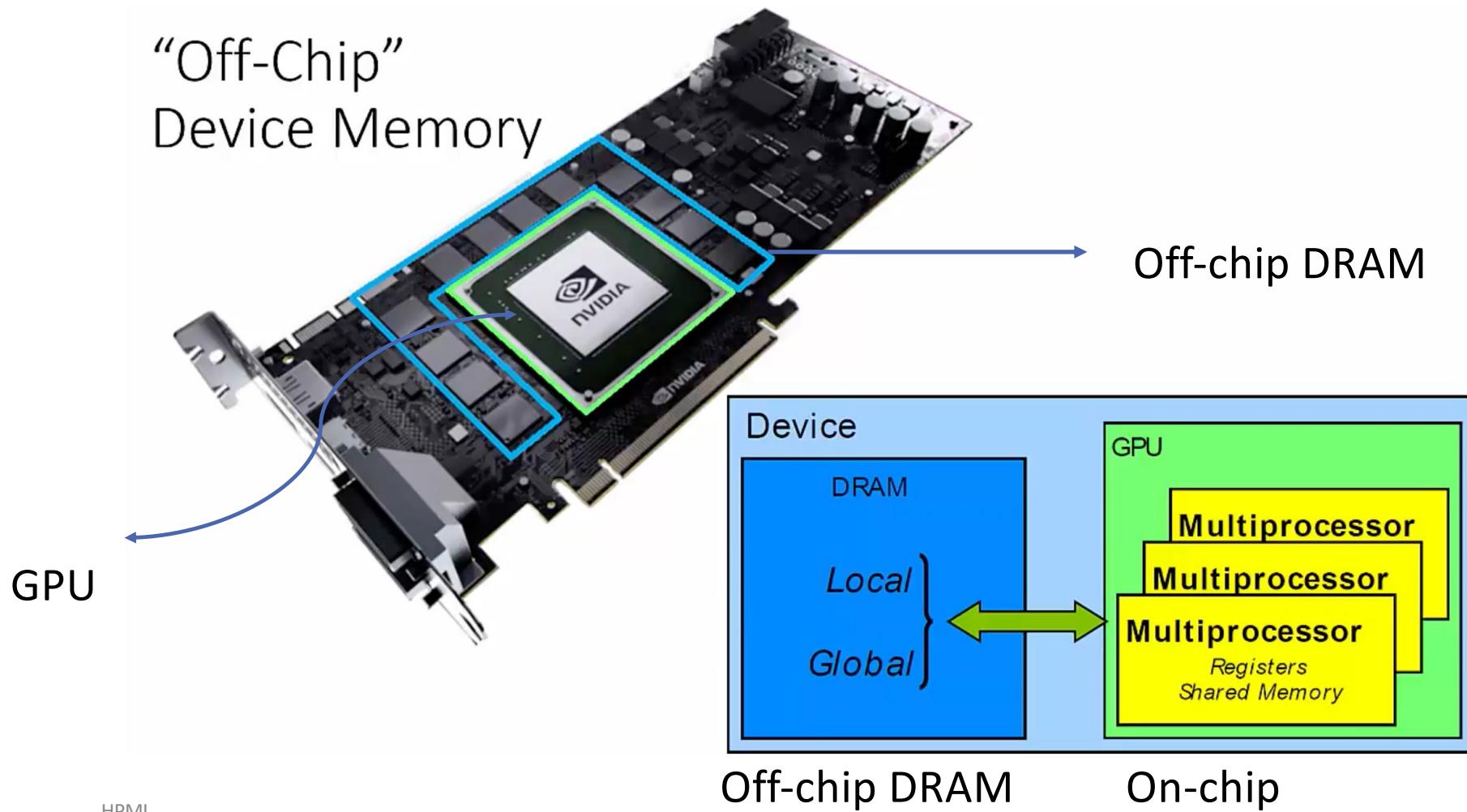


Term **local** refers to the scope of lifetime and not physical location

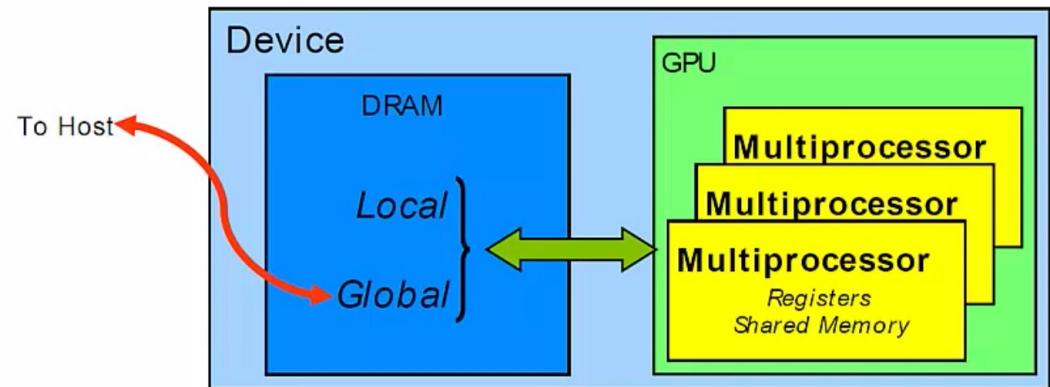


Streaming Multiprocessors

- 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.



Memory Speed



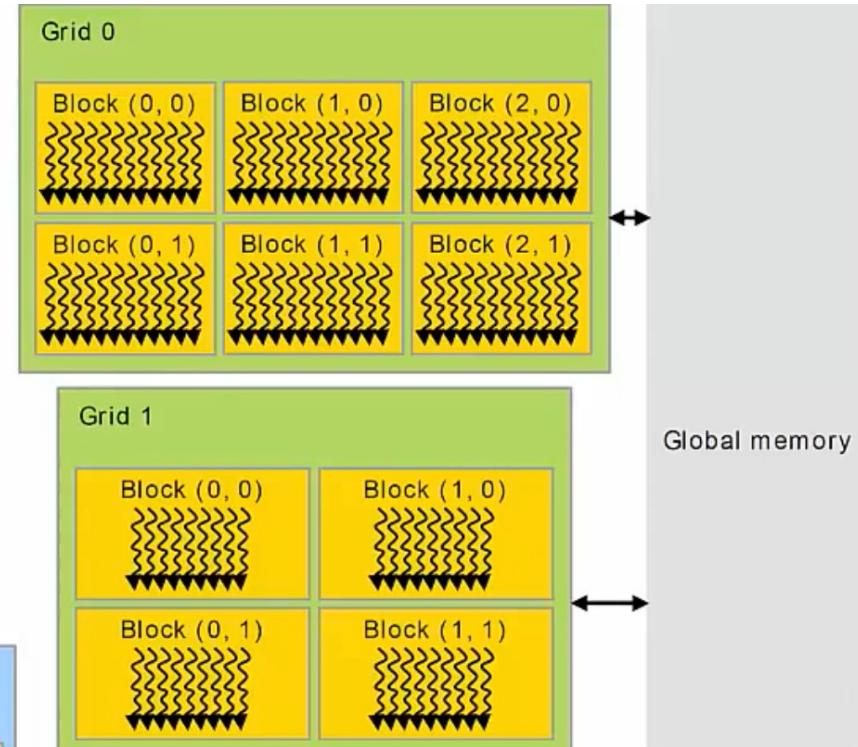
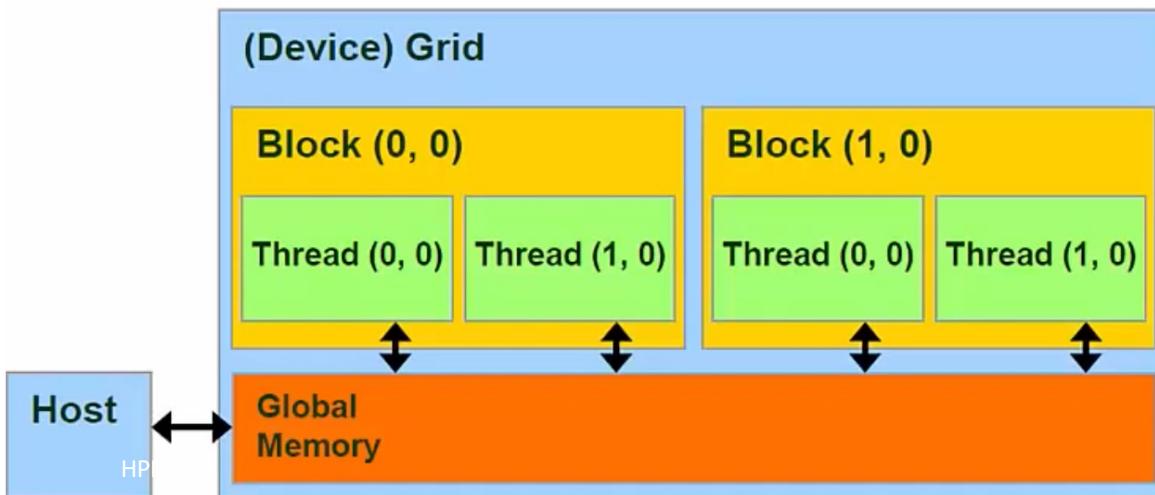
- Relative speed of memory spaces:
$$\frac{\text{"Memory Space"}}{\text{"Bandwidth"} \quad \text{"Latency"}}$$

Registers < Shared < Local \approx Global < Host (PCIe)
~8TB/s ~1.5TB/s ~200GB/s ~5GB/s
~1clock ~32clocks ~800clocks

Global Memory

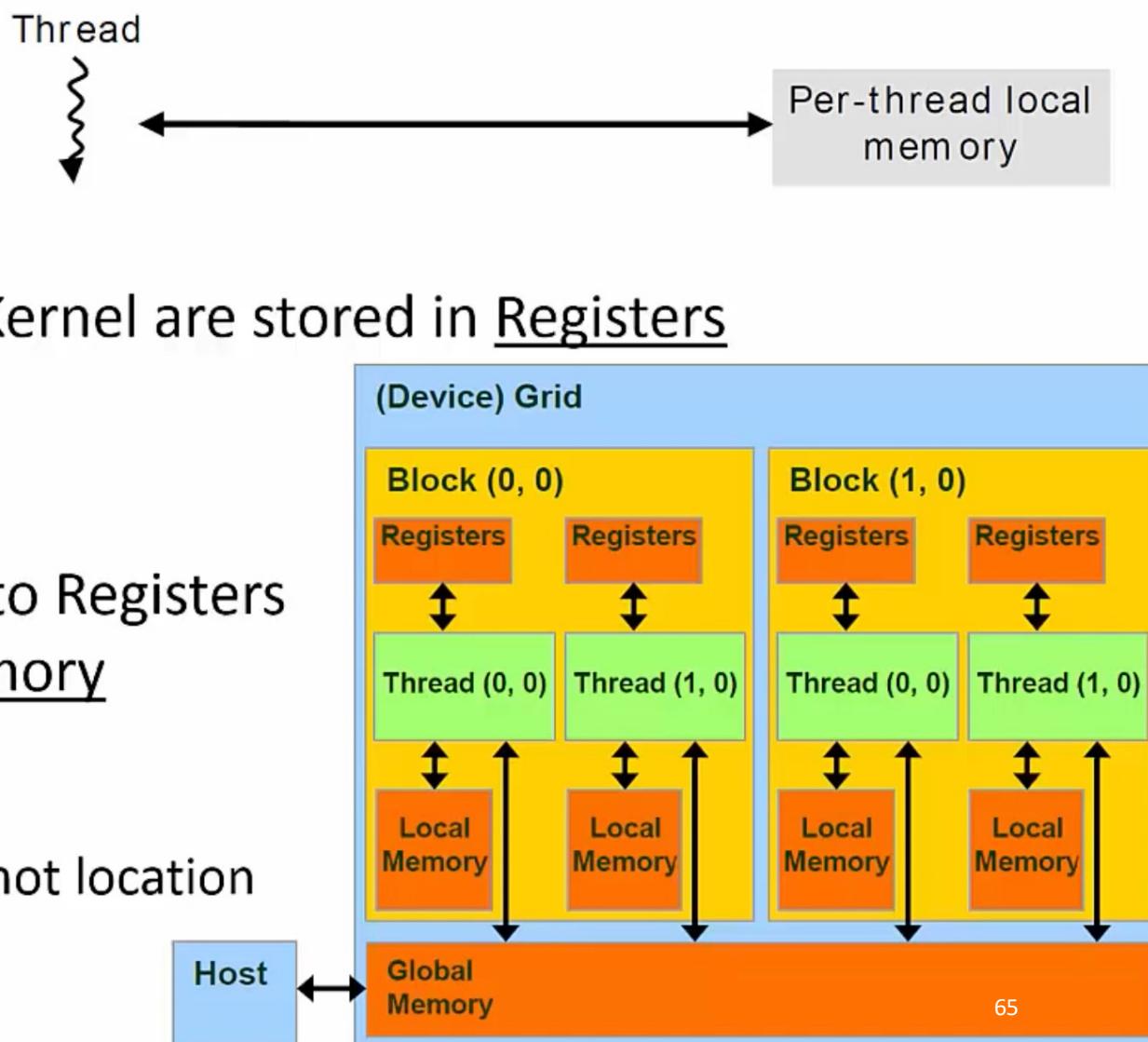
Accessed with

- `cudaMalloc()`
- `cudaMemset()`
- `cudaMemcpy()`
- `cudaFree()`

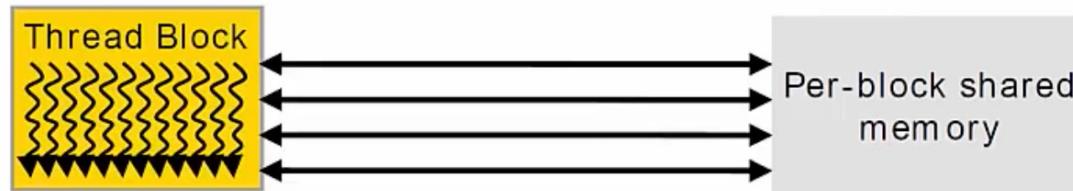


Registers and Local Memory

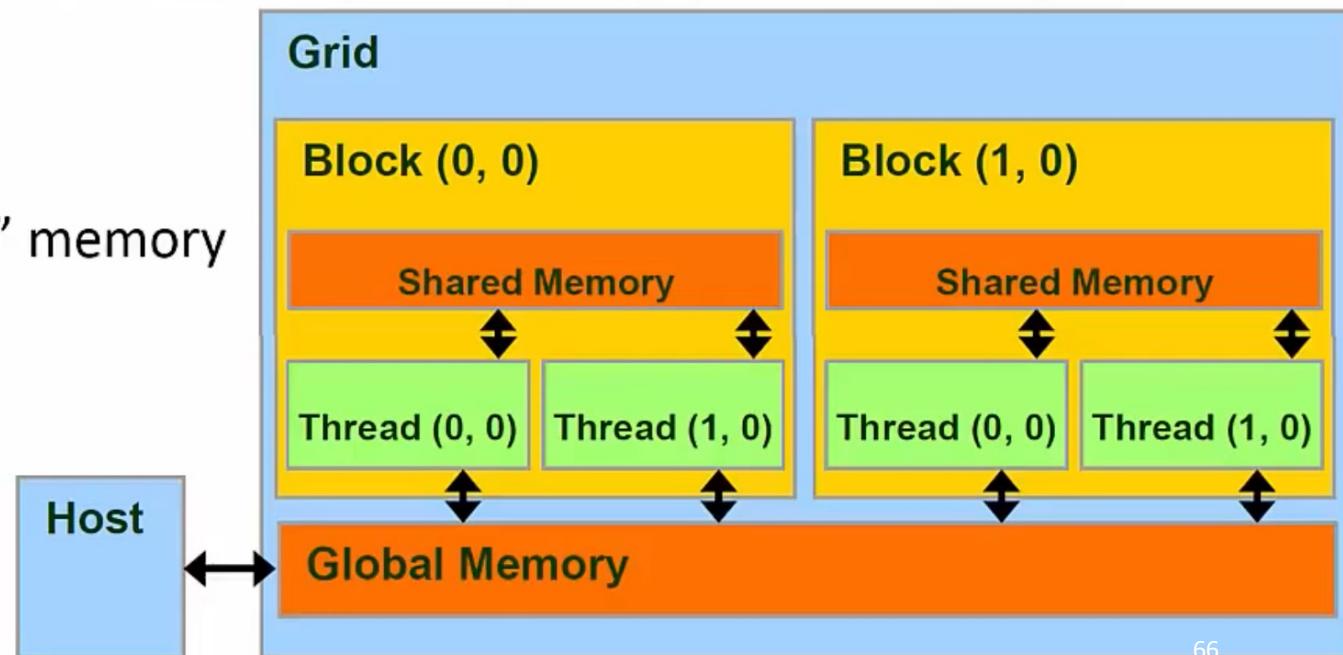
- Variables declared in a Kernel are stored in Registers
 - *On-Chip*
 - Fastest form of memory
- Arrays too large to fit into Registers spill over into Local memory
 - *Off-Chip*
 - Compiler controlled
 - “Local” refers to scope, not location
 - Local to each Thread



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;

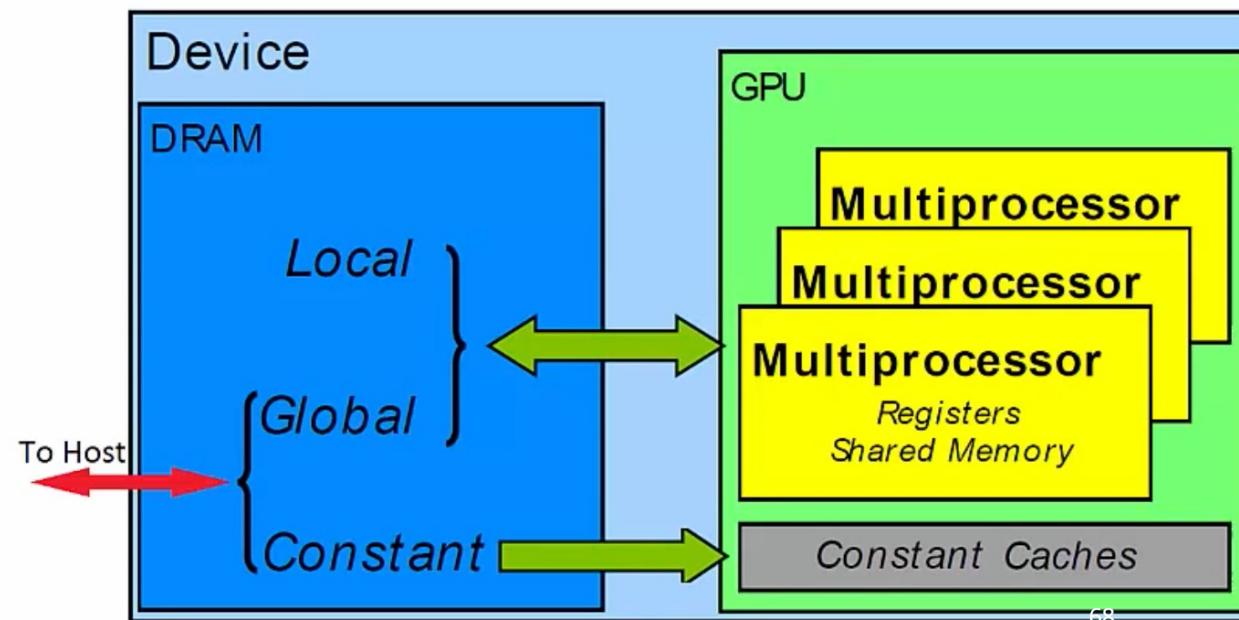
    // Allocate a shared array
    __shared__ int shared_array[N];

    // 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

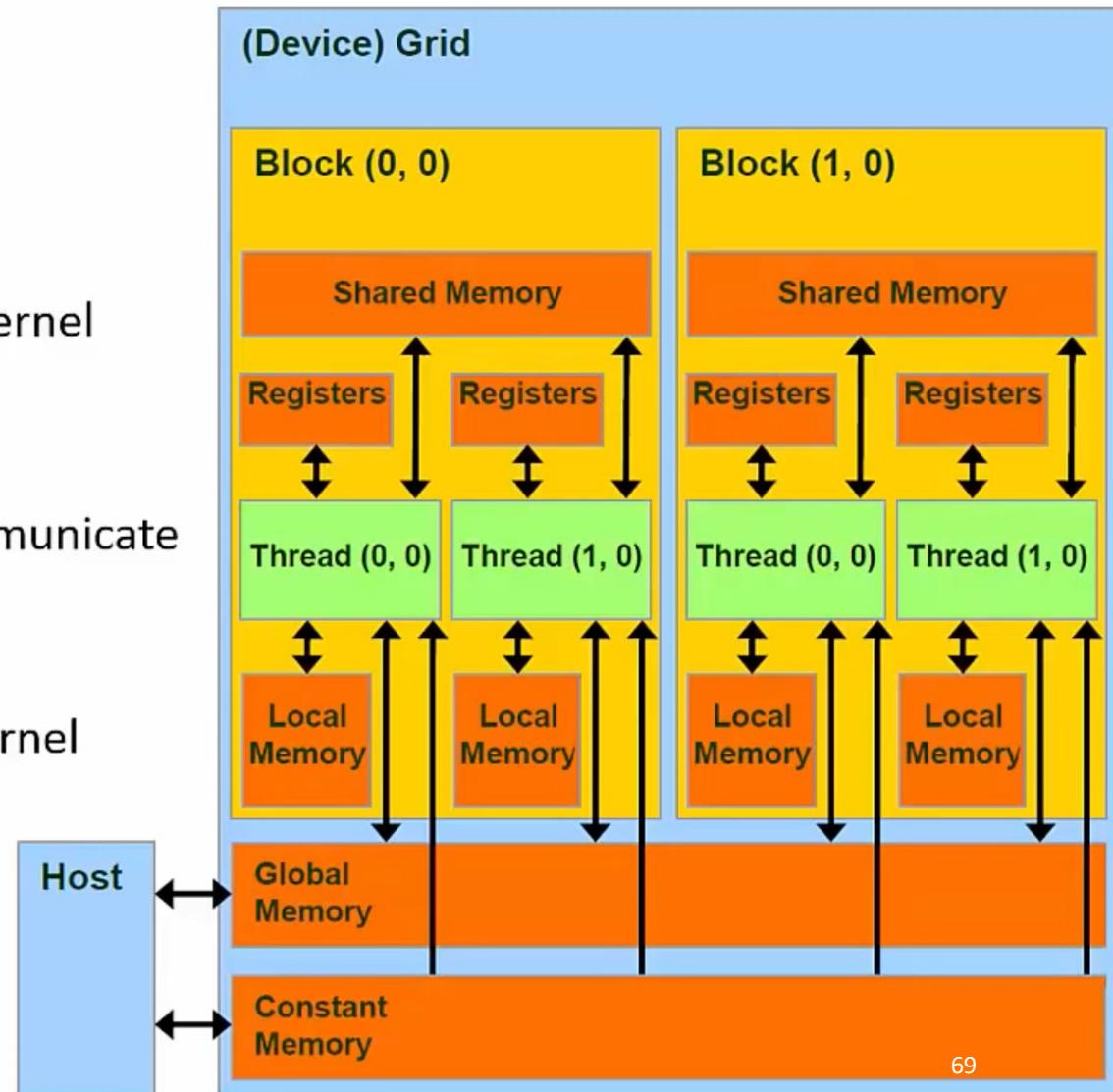
- Allows threads within a block to communicate

Constant

- Used for unchanging data through Kernel

Global Memory

- Stores data copied to and from Host



Registers Usage Best Practices

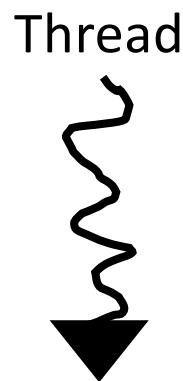
Here are some tips to help you manage register usage effectively:

1. **Understand Register Usage:** Each CUDA kernel invocation uses a certain number of registers. You can find out the number of registers used by your kernel by using the CUDA Compiler (nvcc) with the `-Xptxas="-v"` flag. This will give you information about the register usage per thread.
2. **Optimize Variable Usage:**
 - Minimize the number of local variables.
 - Use simpler data types when possible (e.g., float instead of double).
 - Reuse variables instead of declaring new ones for different purposes.
3. **Inline Functions:** If you use device functions, consider inlining them to reduce register usage. This can be done using the `__forceinline__` qualifier.
4. **Loop Unrolling:** Sometimes, unrolling loops can reduce the number of registers used, but this could also increase register usage in other cases. You need to test and profile to see the effect.
5. **Limit Thread Usage:** The more threads you have per block, the fewer registers are available per thread. If you're hitting register limits, consider using fewer threads per block.
6. **Compiler Optimization Flags:** Use compiler optimization flags to control register usage. For example, `--maxrregcount=N` limits the number of registers that nvcc allows for each kernel.
7. **Shared Memory:** Consider using shared memory for some variables that do not need to be in registers. However, this might impact performance due to the slower access times compared to registers.
8. **Profiling and Optimization Tools:** Utilize CUDA profiling tools like nvprof or NVIDIA Nsight to understand how your kernel uses registers and to identify bottlenecks. These tools can provide insights that guide your optimization efforts.
9. **Architecture Considerations:** Different GPU architectures have different numbers of registers. Keep in mind the target GPU architecture when optimizing register usage.
10. **Manual Register Allocation:** In some advanced cases, you can manually manage registers through assembly-level programming using PTX, but this requires deep knowledge of GPU architecture and should be done with caution.

CUDA Thread Synchronization

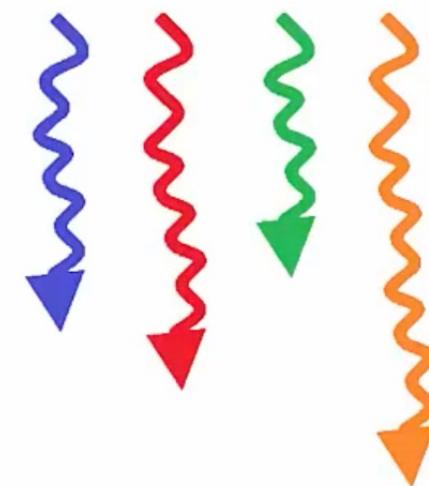
Threads Execute in Parallel

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



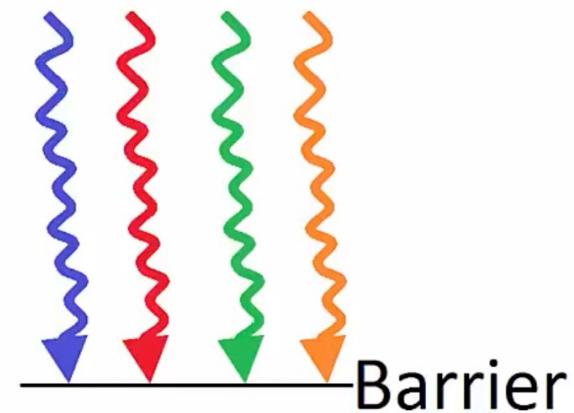
Thread Synchronization via Explicit Barrier

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



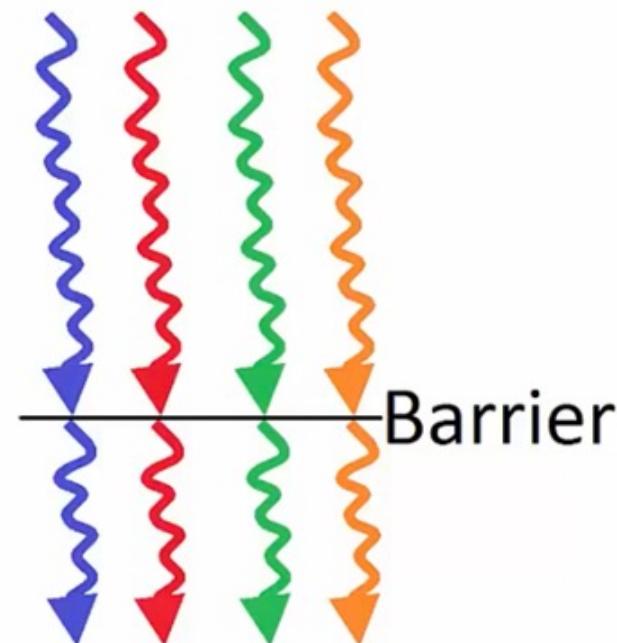
Thread Synchronization via Explicit Barrier

- 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 Synchronization via Explicit Barrier

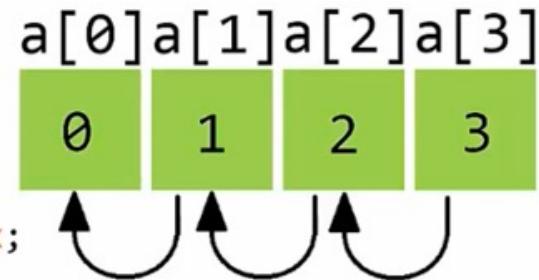
- Threads need to synchronize with one another to avoid race conditions
- A barrier is a point in the kernel where all the threads stop and wait on the others
- When all threads have reached the barrier, they can proceed
- Barriers can be implemented with:
 - `__syncthreads();`



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

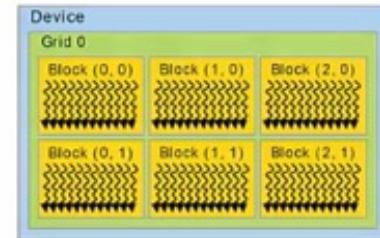
    // Launch Kernel
    kernel_0<<<grid_sz0,blk_sz0>>>(...);

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

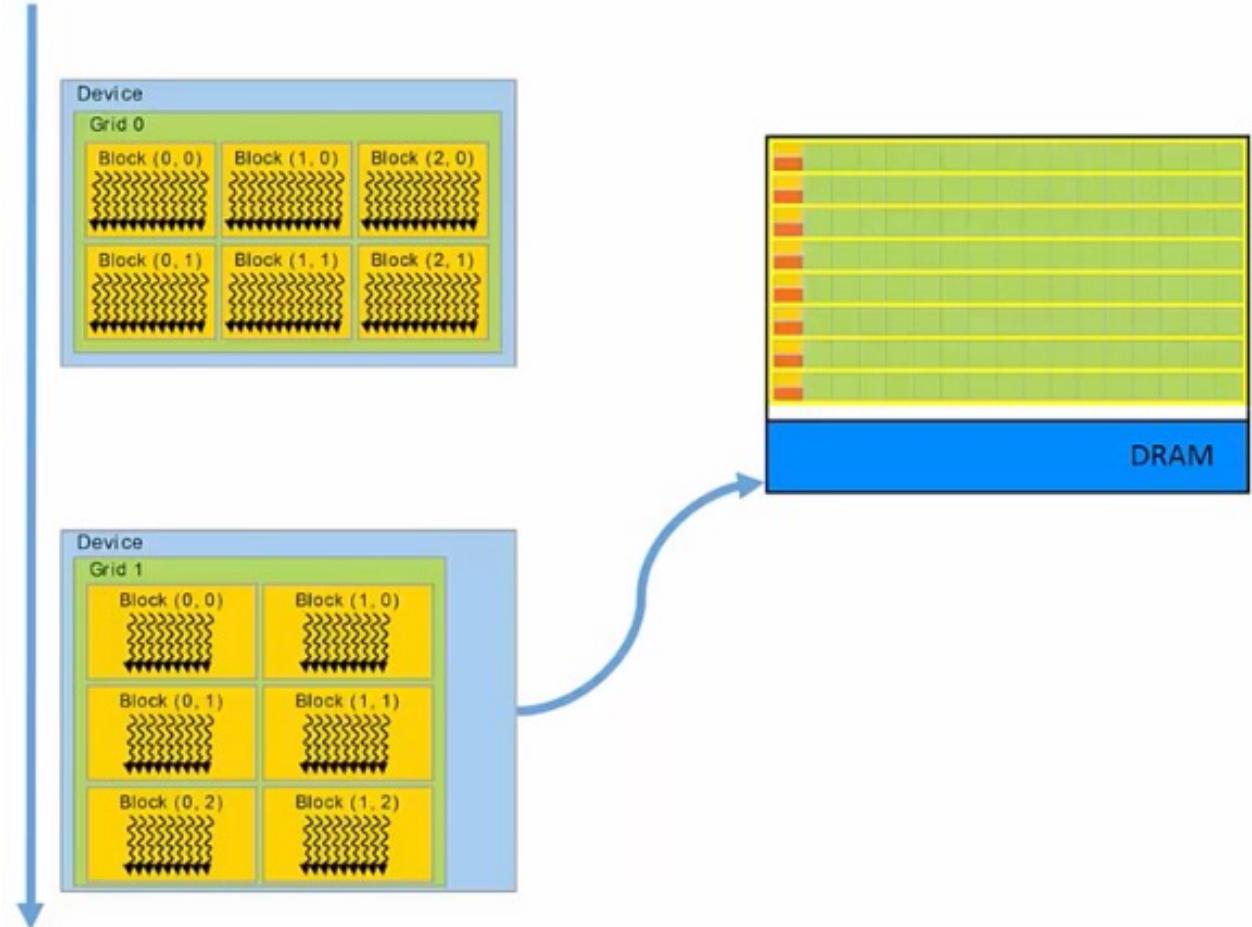
    // Copy data from Device to Host
    cudaMemcpy(...);

    // Do sequential stuff
    // ...
}

return 0;
}
```



Implicit Barriers between Kernels



NVIDIA GPUs and CUDA

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>

GPU Computing Applications						
Libraries and Middleware						
cuDNN TensorRT	cuFFT cuBLAS cuRAND cuSPARSE	CULA MAGMA	Thrust NPP	VSIPL SVM OpenCurrent	PhysX OptiX iRay	MATLAB Mathematica
Programming Languages						
C	C++	Fortran	Java Python Wrappers	DirectCompute	Directives (e.g. OpenACC)	
CUDA-Enabled NVIDIA GPUs						
NVIDIA Ampere Architecture (compute capabilities 8.x)					Tesla A Series	
NVIDIA Turing Architecture (compute capabilities 7.x)			GeForce 2000 Series	Quadro RTX Series	Tesla T Series	
NVIDIA Volta Architecture (compute capabilities 7.x)	DRIVE/JETSON AGX Xavier			Quadro GV Series	Tesla V Series	
NVIDIA Pascal Architecture (compute capabilities 6.x)	Tegra X2		GeForce 1000 Series	Quadro P Series	Tesla P Series	
From: NVIDIA		Embedded	Consumer Desktop/Laptop	Professional Workstation	Data Center	

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

NVIDIA Hardware and Compute Capabilities

Feature Support	Compute Capability									
	5.0, 5.2	5.3	6.x	7.x	8.x	9.0				
(Unlisted features are supported for all compute capabilities)	Yes									
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 <code>memcpy_async</code> (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>

NVIDIA Micro-Architectures

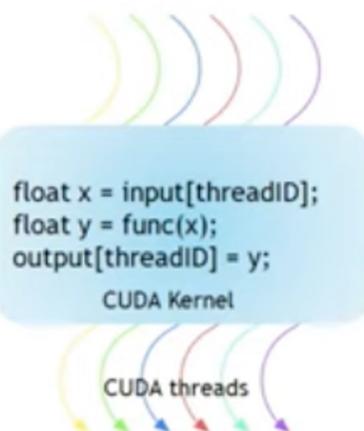
Year	μ Arch	Series	Process	Xistors	Notes
1999	—	GeForce 256	220 nm		First GPU
2008	Tesla	GeForce 8 (8xxx)	65 nm	210 M	CUDA appears
2008		GeForce 9 (9xxx)		1.4 B	
2009		GeForce 200	40 nm		
2010	Fermi	GeForce 400	40 nm		
2011		GeForce 500		3.0 B	
2012	Kepler	GeForce 600	28 nm	7.1 B	
2014		GeForce 700		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	GeForce 30A	7 nm	28.3 B	
		Hopper			

Lesson Outline

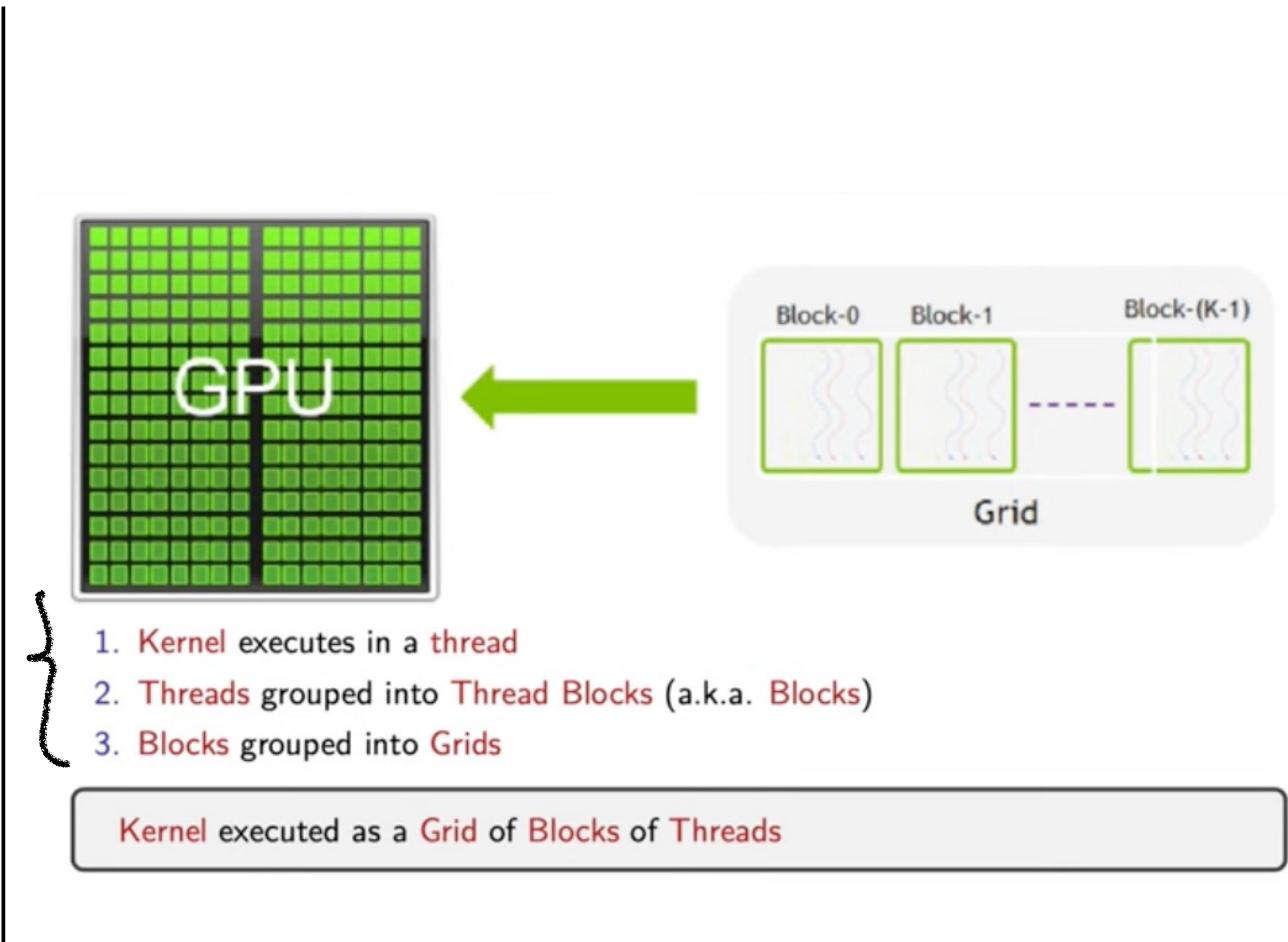
- Heterogenous architectures motivations
- Hierarchy of Computations:
 - Threads
 - Blocks
 - Grids
- Corresponding Memory Spaces
 - Local
 - Shared
 - Global
- Synchronization Primitives
 - Implicit Barriers
 - Thread Synchronization
- NVIDIA GPUs and CUDA:
 - Compute capability
- CUDA Hardware
- CUDA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

Recap

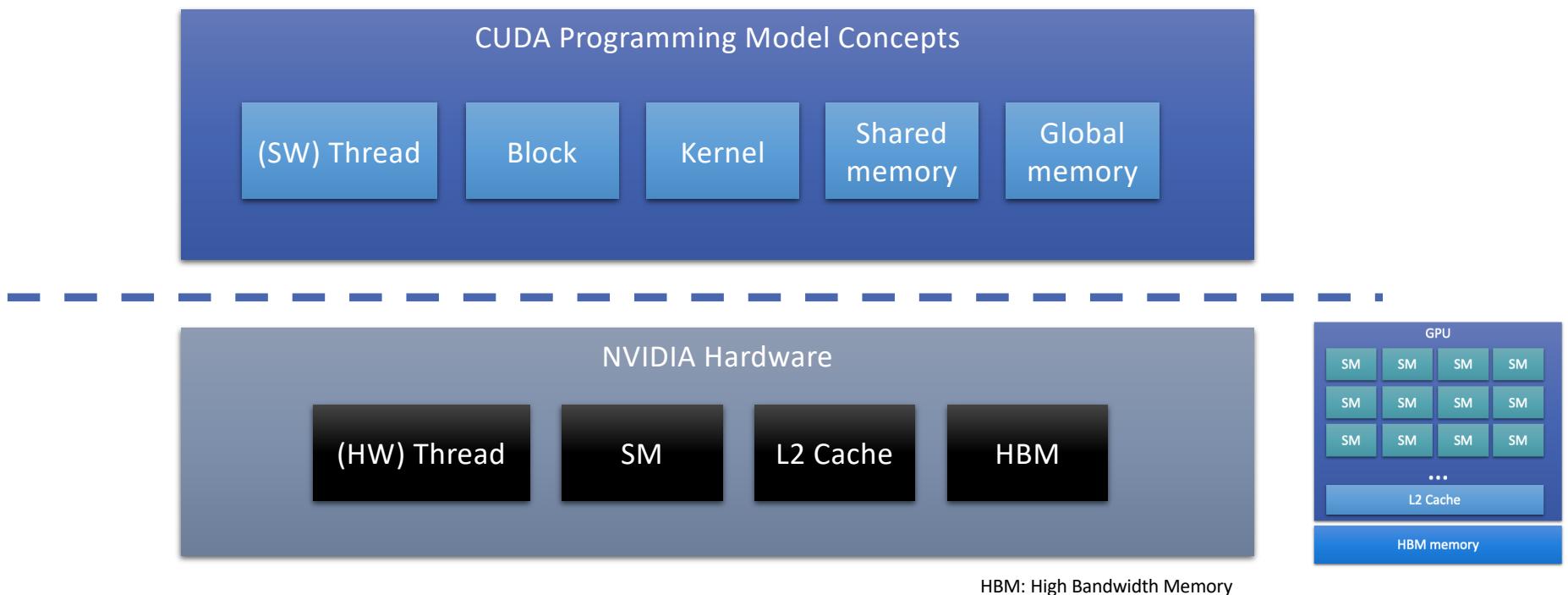


- Kernel: fundamental unit of concurrency
- Executed K times in parallel
- On K different CUDA Threads

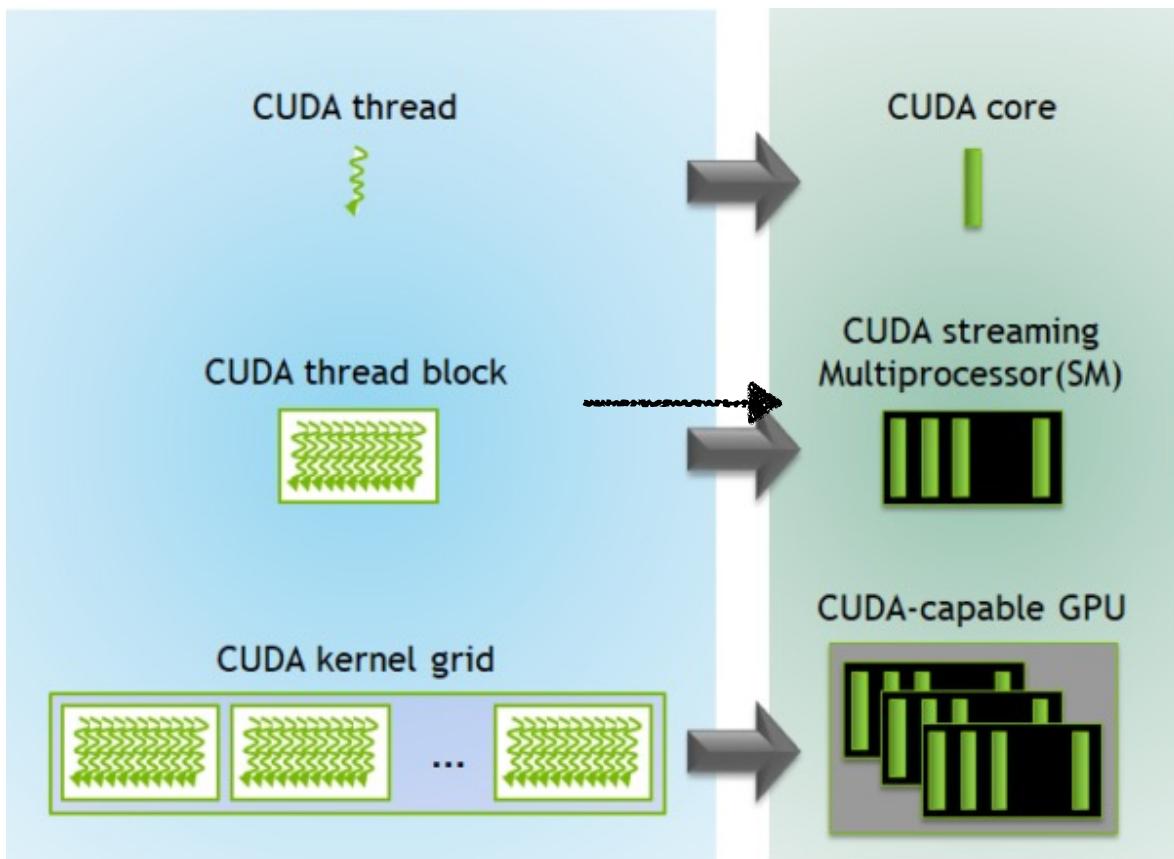


CUDA Programming model

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



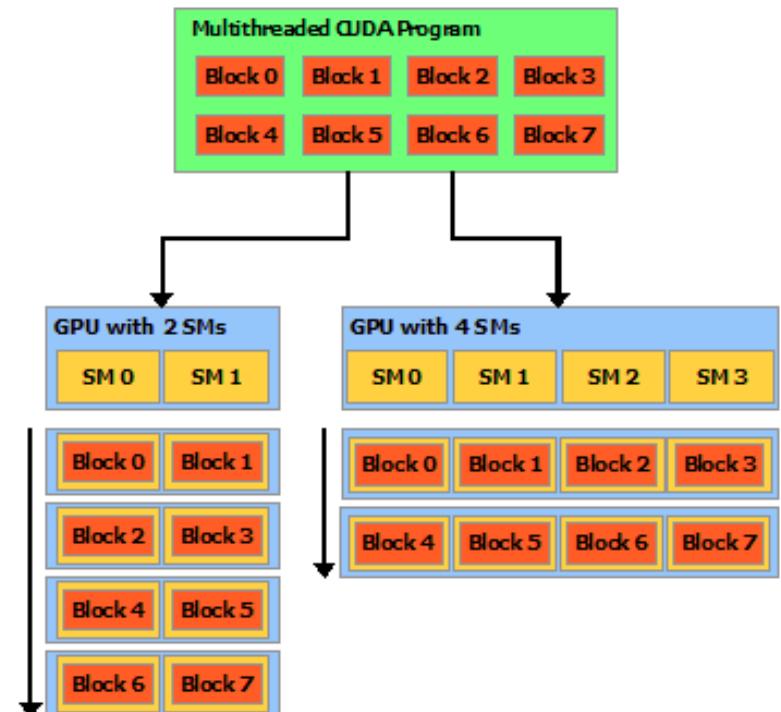
Mapping to Hardware



CUDA	Hardware
Thread	SP
Block	SM
Grid	GPU

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

SM - streaming multiprocessor

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);
    ...
}
```

The diagram shows a hand-drawn red circle around the `__global__` keyword in the kernel definition. There are also red arrows pointing from the text "will run on single thread" and the labels "blocks" and "thread" to the `VecAdd` kernel invocation and the `threadIdx.x` variable respectively.

will run on single thread

blocks thread

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

CUDA functions definition

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

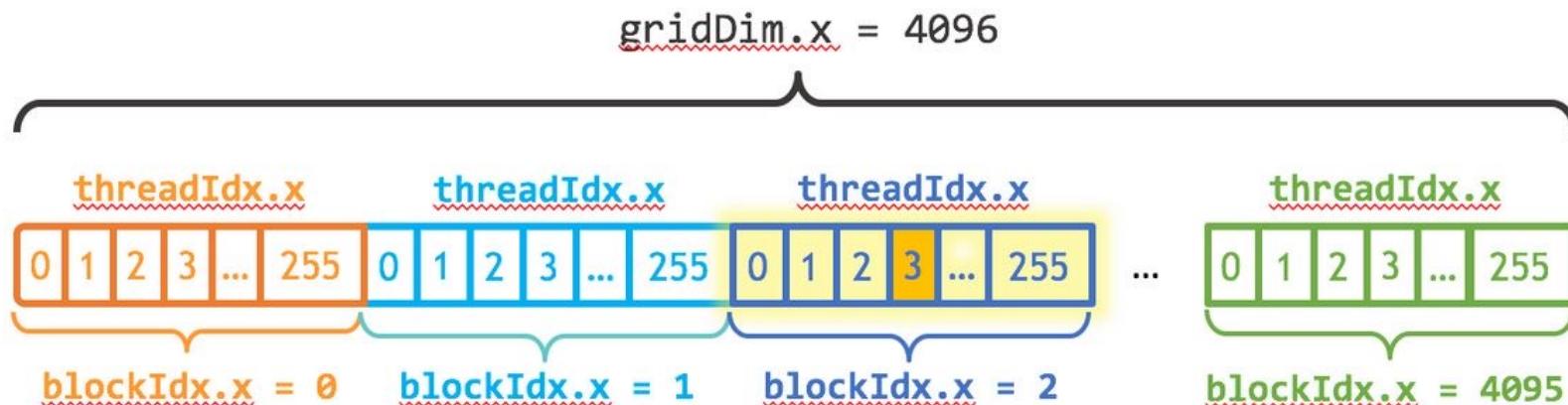
```
Int main() {  
    printf("Hello World!\n");  
    return 0;  
}
```

- 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`

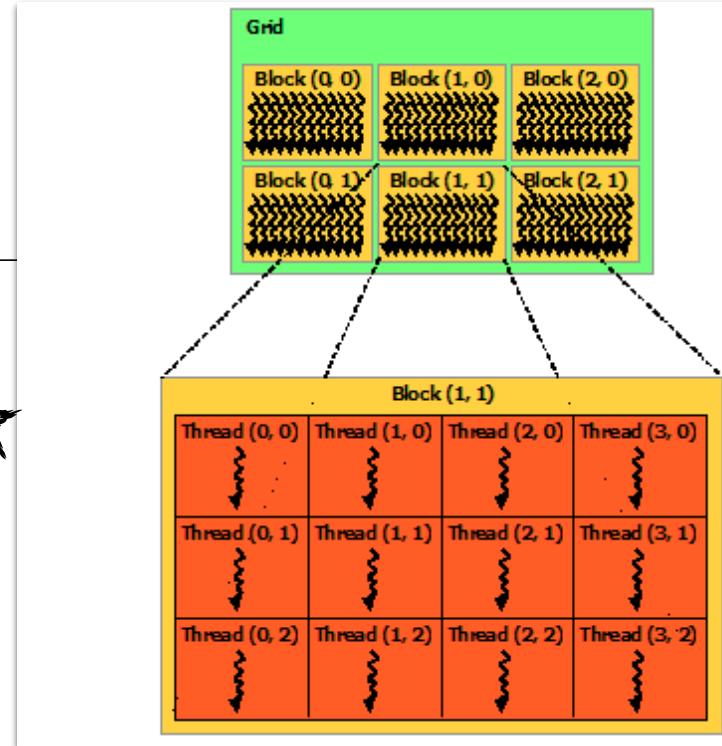
CUDA - Example

- Block and Threads index

let N=256

```
// 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];
}

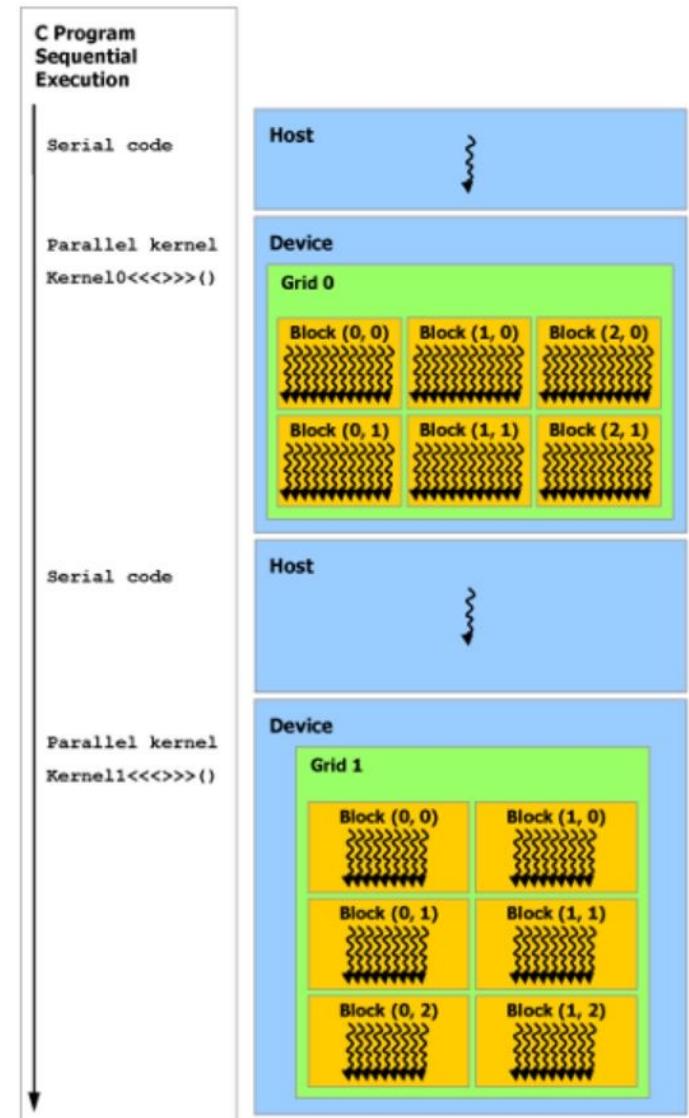
int main() {
    ...
    // Kernel invocation
    dim3 threadsPerBlock(16, 16);
    dim3 numBlocks(N / threadsPerBlock.x, N / threadsPerBlock.y);
    MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
    ...
}
```



From: NVIDIA

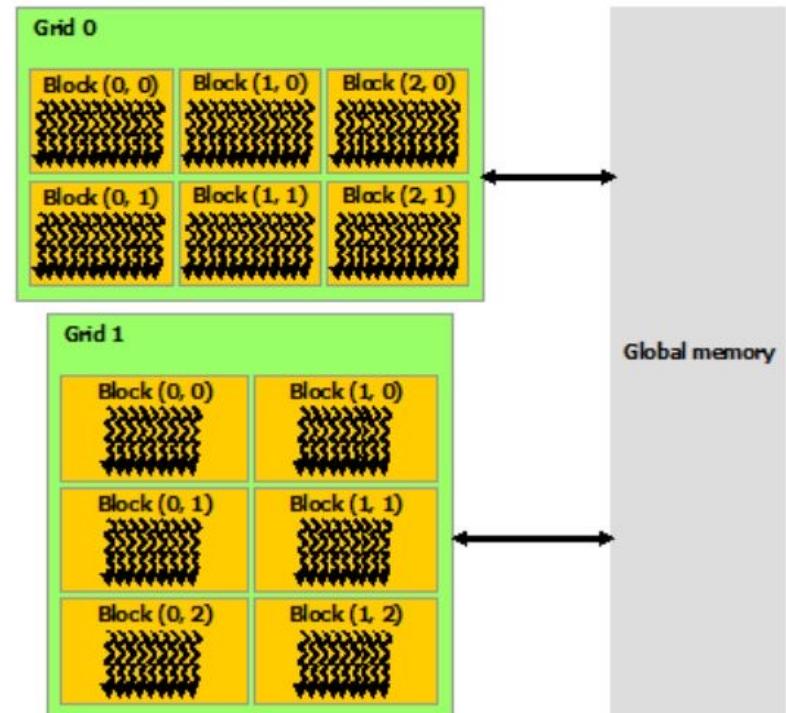
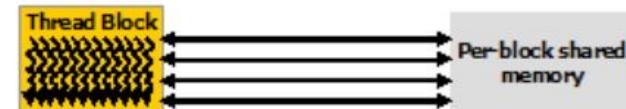
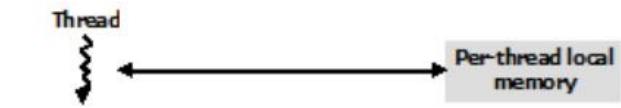
CUDA – Serial and Parallel code

- 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



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



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__                                     one thread processing multiple data locations
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;                           In a queue, Memcopy will run only after kernel has run
    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;
    }
```

one block can have max 1024 threads

```
// 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);
                                            to ensure that we have sufficient blocks
// 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;
}
```

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; #blocks
    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

No need of cudaMemcpy

```
#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;
}
```

CUDA Example Performance Results

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

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

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 Warp Scheduling
- CUDA Context and Streams
- CUDA Pinned Memory
- CUDA Memory Access
- CUDA Compilation and Runtime:
 - CUDA Runtime, CUDA Driver, AoT and JIT compilation
- CUDA Hardware
- CUDA Profiling and Debugging
- cuDNN and cuBLAS

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