

Programmazione di Sistemi ~~Embedded e~~ Multicore

Teacher: Daniele De Sensi

Recap

Recap

- OpenMP loop scheduling
- OpenMP synchronization constructs (barrier, single, master, sections, etc...)

Extra Hours

- Do you have classes on Tuesdays 12.00-13.00?

Questions?

False sharing

False sharing

- **False sharing** : sharing cache lines without actually sharing data.
- How to fix it:
 - Pad the data
 - Change the mapping of data to threads/cores
 - Use private/local variables

Padding the data

- Original

```
double x[N];  
#pragma omp parallel for schedule(static, 1)  
    for( int i = 0; i < N; i++ )  
        x[ i ] = someFunc( x [ i ] );
```

- Padded:

```
double x[N][8];  
#pragma omp parallel for schedule(static, 1)  
    for( int i = 0; i < N; i++ )  
        x[ i ][ 0 ] = someFunc( x [ i ][ 0 ] );
```

- Can kill cache effectiveness.
- Wastes memory.

Data mapping change

```
double x[N];  
#pragma omp parallel for schedule(static, 8)  
    for( int i = 0; i < N; i++ )  
        x[ i ] = someFunc( x [ i ] );
```

Questions?

OpenMP + MPI

- MPI defines 4 levels of thread safety:
 - `MPI_THREAD_SINGLE`: One thread exists in program
 - `MPI_THREAD_FUNNELED`: only the master thread can make MPI calls. Master is one that calls `MPI_Init_thread()`
 - `MPI_THREAD_SERIALIZED`: Multithreaded, but only one thread can make MPI calls at a time
 - `MPI_THREAD_MULTIPLE`: Multithreaded and any thread can make MPI calls at any time
- Safest (easiest) to use `MPI_THREAD_FUNNELED`
 - Fits nicely with most OpenMP models
 - Expensive loops parallelized with OpenMP
 - Communication and MPI calls between loops

OpenMP/Pthreads + MPI

```
$ ./a.out 4
```

```
$ Time: 0.40 seconds
```

```
$ mpirun -n 1 ./a.out 4
```

```
$ Time: 1.17 seconds
```

Why?

Open MPI maps each process on a core. Thus, all the threads created by that process will run on the same core (i.e., 4 threads will run on the same core).

How to fix it?

```
$ mpirun --bind-to-none -n 1 ./a.out 4
```

```
$ Time: 0.40 seconds
```

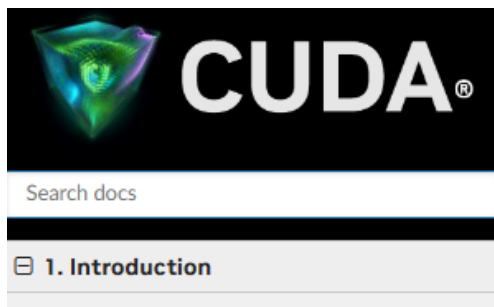
How to check how Open MPI is binding processes?

```
$ mpirun --report-bindings -n 1 ./a.out 4 1024 1024 1024
```

```
MCW rank 0 bound to socket 0[core 0[hwt 0-1]]:
```

```
[BB/../../../../../../../../][../../../../../../../../][../../../../../../../../][../../../../../../../../]
```

GPU Programming



[Home](#) » 1. Introduction

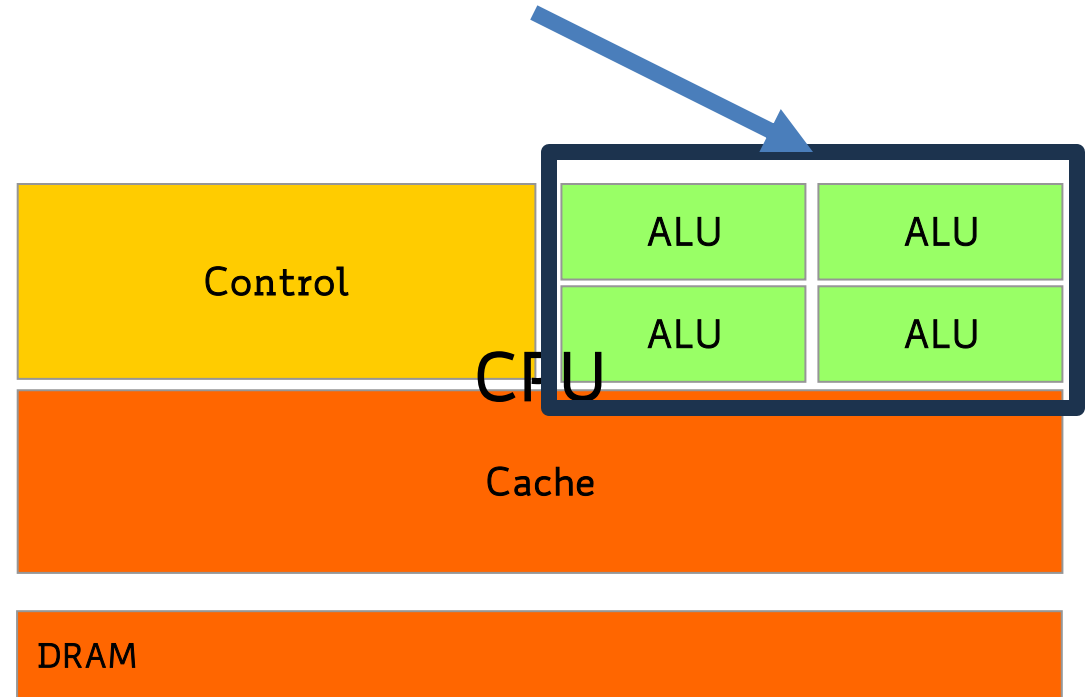
CUDA C++ Programming Guide

The programming guide to the CUDA model and interface.

CPUs: Latency Oriented Design

- High clock frequency
- Large caches
 - Convert long latency memory accesses to short latency cache accesses
- Sophisticated control
 - Branch prediction for reduced branch latency
 - Out-of-order execution
 - etc...
- Powerful ALU
 - Reduced operation latency

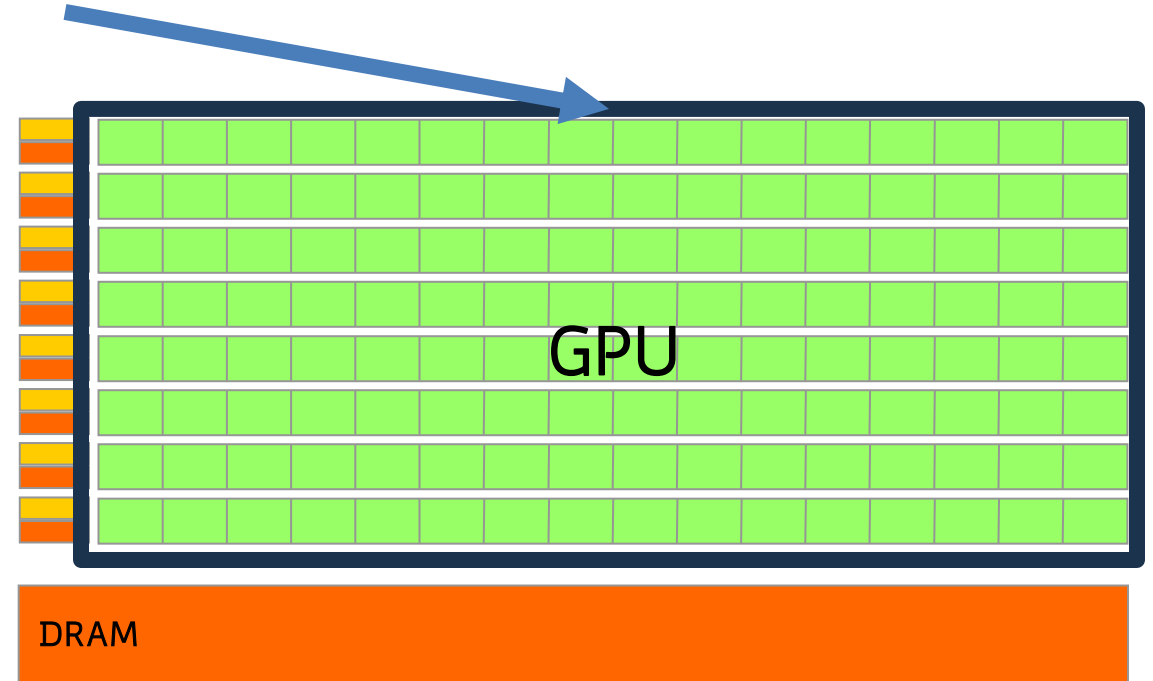
The actual computation is done here



GPUs: Throughput Oriented Design

The actual computation is done here

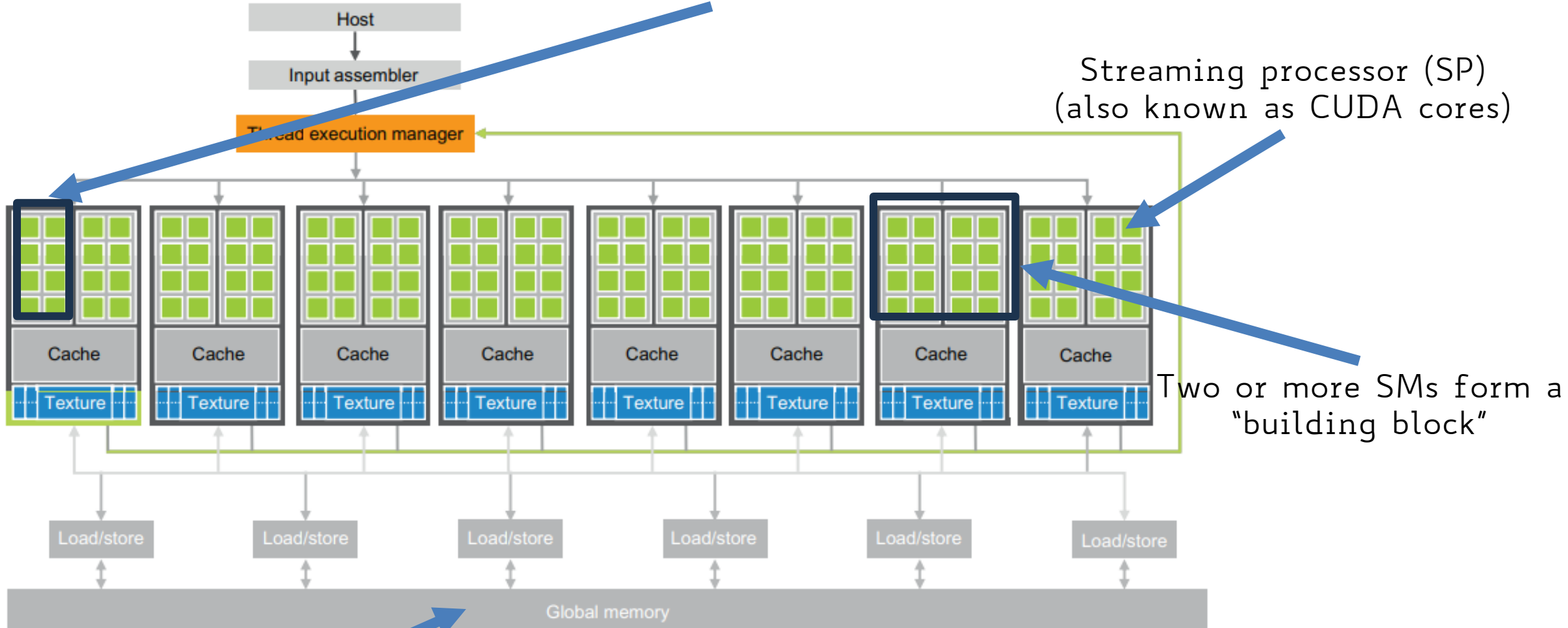
- Moderate clock frequency
- Small caches
 - To boost memory throughput
- Simple control
 - No branch prediction
 - In-order execution
- Energy efficient ALUs
 - Many, long latency but heavily pipelined for high throughput
- Require massive number of threads to tolerate latencies
- Use high bandwidth interfaces for memory and host data exchange



Architecture of a CUDA-capable GPU

Streaming multiprocessor (SM)

(SPs on the same SM share control logic and instruction cache)



Streaming processor (SP)
(also known as CUDA cores)

Two or more SMs form a
"building block"

High-Bandwidth Memory

How many CUDA cores?
15,000 on NVIDIA H100

Applications Benefit from Both CPU and GPU

- CPUs for sequential parts where latency matters
 - CPUs can be 10+X faster than GPUs for sequential code
- GPUs for parallel parts where throughput matters
 - GPUs can be 10+X faster than CPUs for parallel code

CPU-GPU Architecture

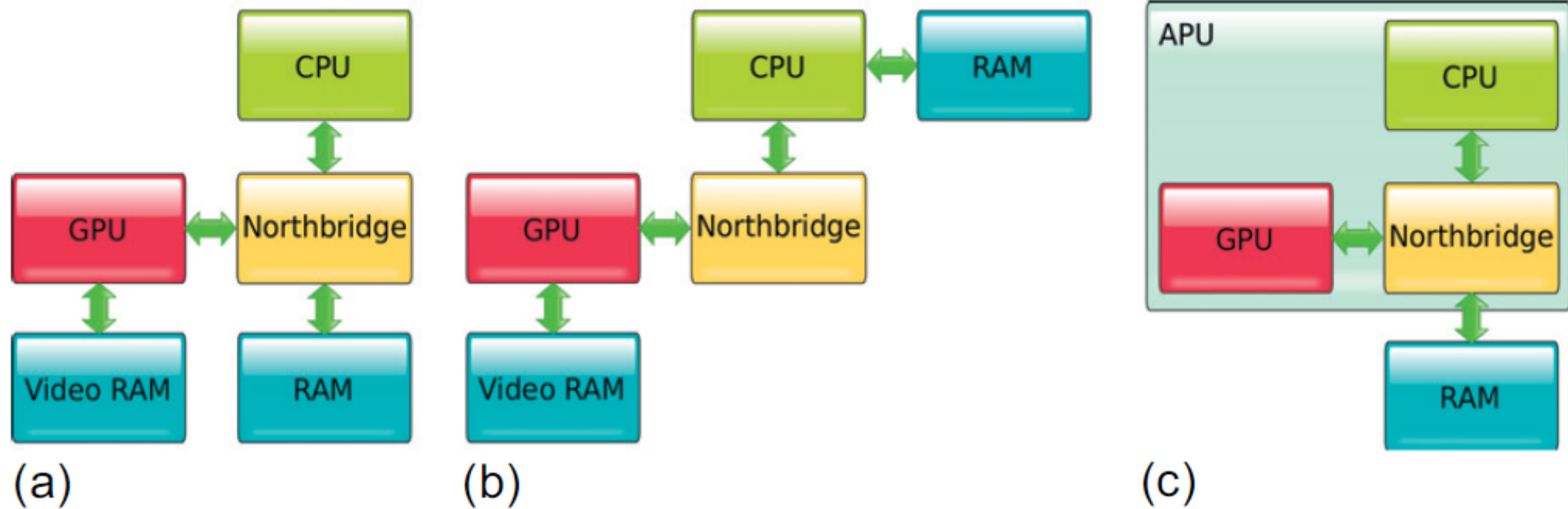


FIGURE 6.1

Existing architectures for CPU-GPU systems: (a) and (b) represent discrete GPU solutions, with a CPU-integrated memory controller in (b). Diagram (c) corresponds to integrated CPU-GPU solutions, such as the AMD's Accelerated Processing Unit (APU) chips.

Questions?

GPU programming caveat

- GPU program deployment has a characteristic that can be considered a major obstacle: GPU and host memories are typically disjoint, requiring explicit (or implicit, depending on the development platform) data transfer between the two.
- A second characteristic of GPU computation is that GPU devices may not adhere to the *same floating-point representation* and accuracy standards as typical CPUs.
 - Slightly different ways to do rounding and corner cases:
<https://docs.nvidia.com/cuda/floating-point/index.html>

GPU Software Development Platforms

- **CUDA** : Compute Unified Device Architecture. CUDA provides two sets of APIs (a low and a higher level one) and it is available freely for Windows, MacOS X and Linux operating systems. Major drawback : NVidia hardware only (even though now there are tools to run CUDA code on AMD GPUs).
- **HIP** : AMD's equivalent of CUDA. Tools provided to convert CUDA to HIP code.
- **OpenCL** : Open Computing Language is an open standard for writing programs that can execute across a variety of heterogeneous platforms that include GPUs, CPU, DSPs or other processors. OpenCL is supported by both NVidia and AMD. It is the primary development platform for AMD GPUs. OpenCL's programming model matches closely the one offered by CUDA.
- **OpenACC** : An open specification for an API that allows the use of compiler directives (e.g. `#pragma acc`, in a similar fashion to OpenMP) to automatically map computations to GPUs or multicore chips, according to a programmer's hints.
- Many more (OpenMP, Thrust, Data Parallel C++, C++ AMP) ...

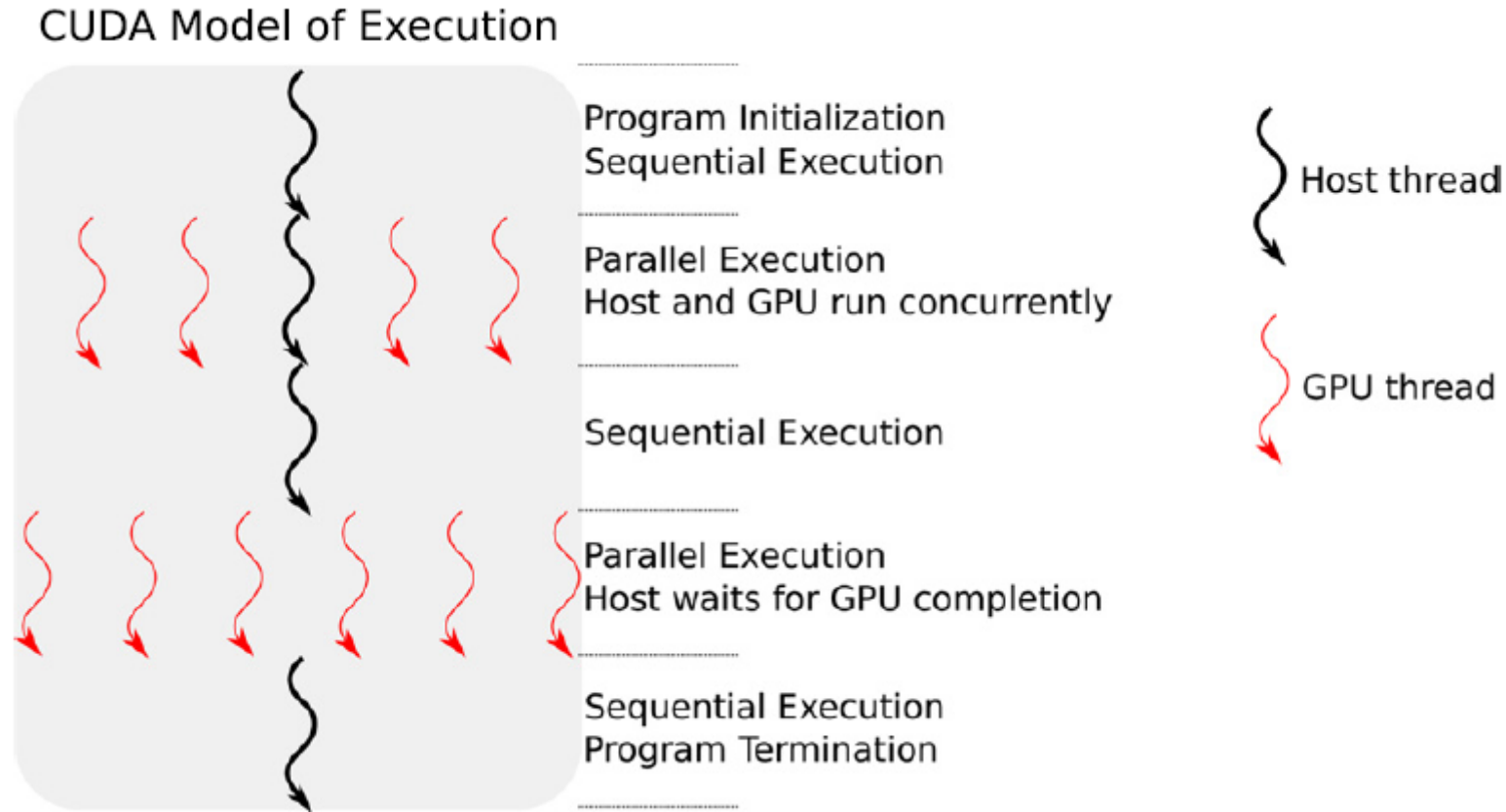
CUDA: Compute Unified Device Architecture

- It enables a **general-purpose programming model** on NVIDIA GPUs: before CUDA, GPU was programmed using transforming an algorithm in a sequence of image manipulation primitives
- It requires additional hardware to manage/interact with the host to provide generic computation
- Enables explicit GPU **memory management**
- The **GPU** is viewed as a **compute device** that:
 - Is a **co-processor** to the CPU (or host)
 - Has its own **DRAM** (global memory in CUDA parlance)
 - Runs many **threads** in parallel: thread creation/switching cost is few clock cycles!
- CUDA is a platform/programming model, not a programming language

CUDA Program Structure

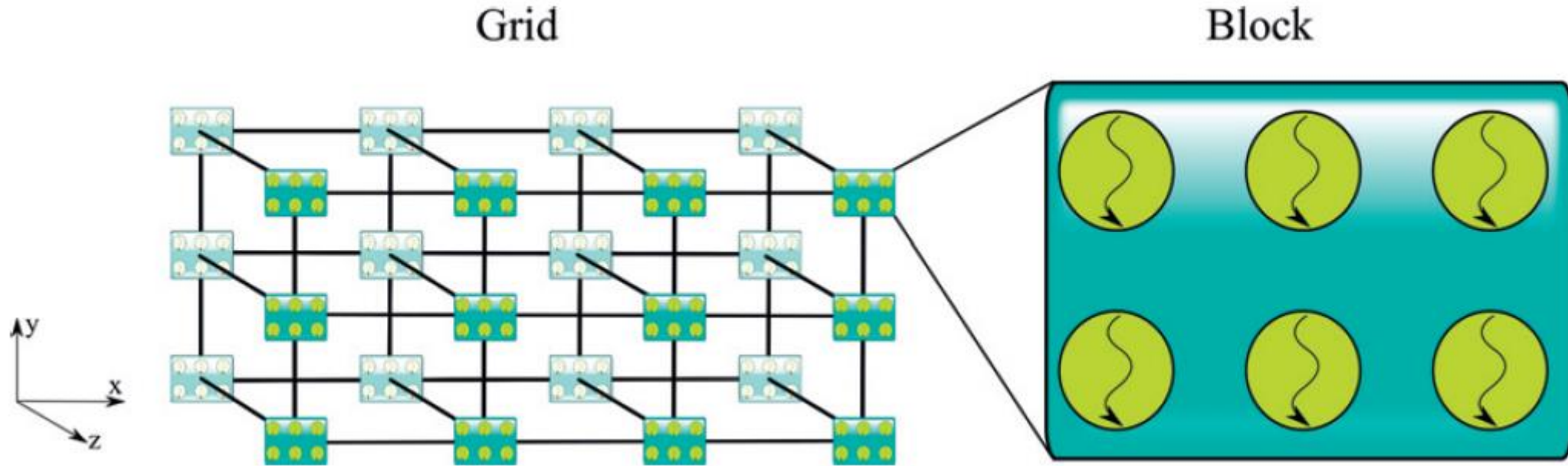
- Allocate GPU memory
- Transfer data from host to GPU memory
- Run CUDA kernel
- Copy results from GPU memory to host memory

CUDA Execution Model



In the vast majority of scenarios, the host is responsible for I/O operations, passing the input and subsequently collecting the output data from the memory space of the GPU.

CUDA Execution Model



- CUDA organizes the threads in a 6-D structure (lower dimensions are also possible).
- Each thread has a position in a 1D, 2D, or 3D *block*
- Each block has a position in a 1D, 2D or 3D *grid*
- Each thread is aware of its position in the overall structure, via a set of **intrinsic variables**/structures. With this information a thread can map its position to the subset of data that it is assigned to.






CUDA compute capability

- The sizes of blocks and grids are determined by the *capability*, which determines what each generation of GPUs is capable of
- The *compute capability* of a device is represented by a version number, also sometimes called its "SM version".
- This version number identifies the features supported by the GPU hardware and is used by applications at runtime to determine which hardware features and/or instructions are available on the present GPU.

Table 6.1 Compute Capabilities and Associated Limits on Block and Grid sizes

Item	Compute Capability			
	1.x	2.x	3.x	5.x
Max. number of grid dimensions	2	3		
Grid maximum x-dimension	$2^{16} - 1$		$2^{31} - 1$	
Grid maximum y/z-dimension	$2^{16} - 1$			
Max. number of block dimensions	3			
Block max. x/y-dimension	512	1024		
Block max. z-dimension	64			
Max. threads per block	512	1024		
GPU example (GTX family chips)	8800	480	780	980

CUDA: Nvidia architectures

 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
	 Embedded	 Consumer Desktop/Laptop	 Professional Workstation	 Data Center

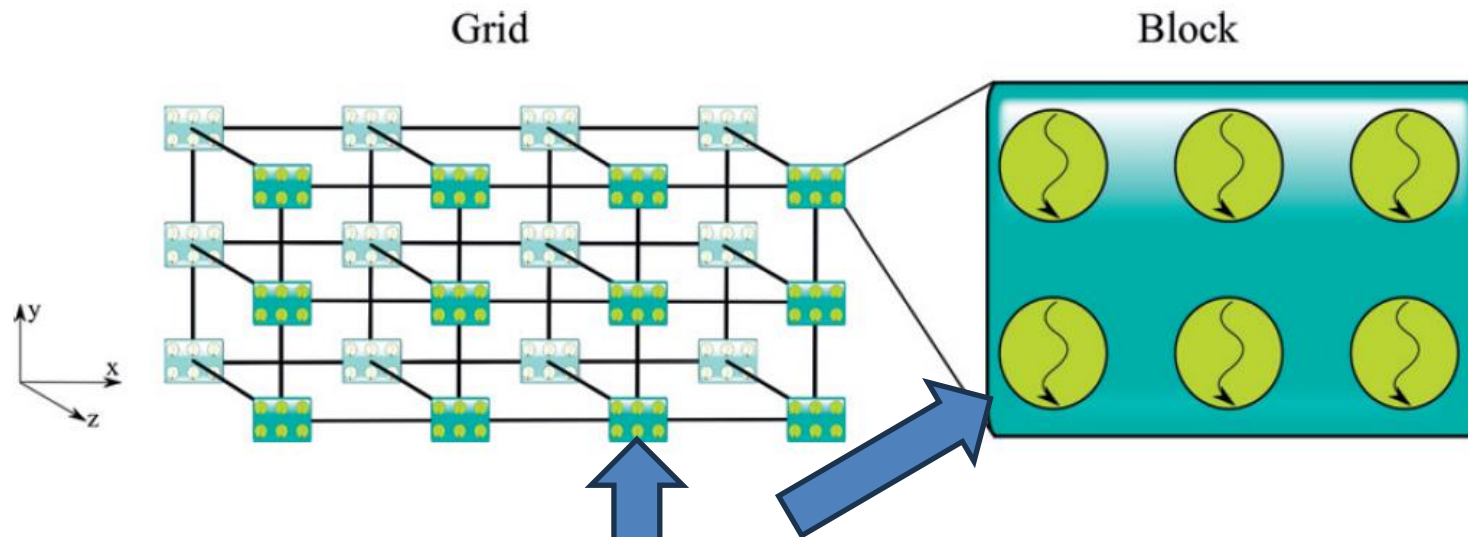
CUDA compute capability

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
Atomic functions operating on 32/64-bit integer values in global and shared memory	Yes				
Atomic addition operating on 32-bit floating point values in global and shared memory	Yes				
Atomic addition operating on 64-bit floating point values in global memory and shared memory	No		Yes		
Warp vote, Memory Fence, Synchronization functions, Unified Memory Programming , 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	No			Yes	
Hardware-accelerated async-copy	No				Yes
L2 Cache Residency Management	No				Yes

Questions?

How to write a program?

- You must specify a function that is going to be executed by all the threads (SIMD/SPMD/SIMT)
- This function is called *kernel*
- You must specify how threads are arranged in the grid/blocks



```
dim3 block(3,2);  
dim3 grid(4,3,2);  
foo<<<grid, block>>>();
```

dim3

```
dim3 block(3,2);  
dim3 grid(4,3,2);  
foo<<<grid, block>>>();
```

- dim3 is a vector of int
- Every non-specified component is set to 1
- Every component accessible to x, y, z fields (we will see it later)
- Shall I specify a 1D, 2D, or 3D grid/block? Depends on the problem: are you working on a 1D, 2D, ..., 6D domain?

How to specify grid/block size

```
dim3 b(3,3,3);  
dim3 g(20,100);  
foo<<<g, b>>>(); // Run a 20x100 grid made of 3x3x3 blocks  
foo<<<10, b>>>(); // Run a 10-block grid, each block made by 3x3x3 ←  
    threads  
foo<<<g, 256>>>(); // Run a 20x100 grid, made of 256 threads  
foo<<<g, 2048>>>(); // An invalid example: maximum block size is ←  
    1024 threads even for compute capability 5.x  
foo<<<5, g>>>(); // Another invalid example, that specifies a ←  
    block size of 20x100=2000 threads
```


Hello World in CUDA

Can be called from the host or the device
Must run on the device

A kernel is always declared as void
Computed result must be copied explicitly from GPU to host memory

```
// File : hello.cu
#include <stdio.h>
#include <cuda.h>

__global__ void hello()
{
    printf("Hello world\n");
}

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

printf in device
code is supported
by CC 2.0 and above

Blocks until the CUDA kernel terminates
(like a barrier, kernel launch is asynchronous)

Compile for
CC 2.0

- To compile and run:
\$ nvcc -arch=sm_20 hello.cu -o hello
\$./hello

Function decorations

- `__global__` : can be called from the host or the GPU and executed on the device/GPU. In CC 3.5 and above, the device can also call `__global__` functions
- `__device__` : A function that runs on the GPU and can only be called from within a kernel (i.e., from the GPU)
- `__host__` : A function that can only run on the host. The `__host__` qualifier is typically omitted, unless used in combination with `__device__` to indicate that the function can run on both the host and the device. Such a scenario implies the generation of two compiled codes for the function. Can you guess why?

How to get the thread position in the grid/block

Remember that threads are arranged in a 6D space

- `blockDim`: Contains the size of each block, e.g., (B_x, B_y, B_z) .
- `gridDim`: Contains the size of the grid, in blocks, e.g., (G_x, G_y, G_z) .
- `threadIdx`: The (x, y, z) position of the thread within a block, with $x \in [0, B_x - 1]$, $y \in [0, B_y - 1]$, and $z \in [0, B_z - 1]$.
- `blockIdx`: The (b_x, b_y, b_z) position of a thread's block within the grid, with $b_x \in [0, G_x - 1]$, $b_y \in [0, G_y - 1]$, and $b_z \in [0, G_z - 1]$.

How to get a unique thread id?

- Different threads might have the same threadIdx but be on different blocks
- I need to combine threadIdx and blockIdx to get a unique identifier

```
int myID = ( blockIdx.z * gridDim.x * gridDim.y +  
            blockIdx.y * gridDim.x +  
            blockIdx.x ) * blockDim.x * blockDim.y * blockDim.z +  
            threadIdx.z * blockDim.x * blockDim.y +  
            threadIdx.y * blockDim.x +  
            threadIdx.x;
```

- Often threads are arranged in fewer than 6 dimensions (i.e., some of those dimensions will be equal to 1 and the corresponding coordinates to 0)
- E.g., to get the ID for the hello world case (threads were arranged in 1 block of 10 threads):

```
int myID = ( blockIdx.z * gridDim.x * gridDim.y +  
            blockIdx.y * gridDim.x +  
            blockIdx.x ) * blockDim.x +  
            threadIdx.x;
```

Hello World in CUDA

```
// File : hello.cu
#include <stdio.h>
#include <cuda.h>

__global__ void hello()
{
    printf("Hello world\n");
}

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

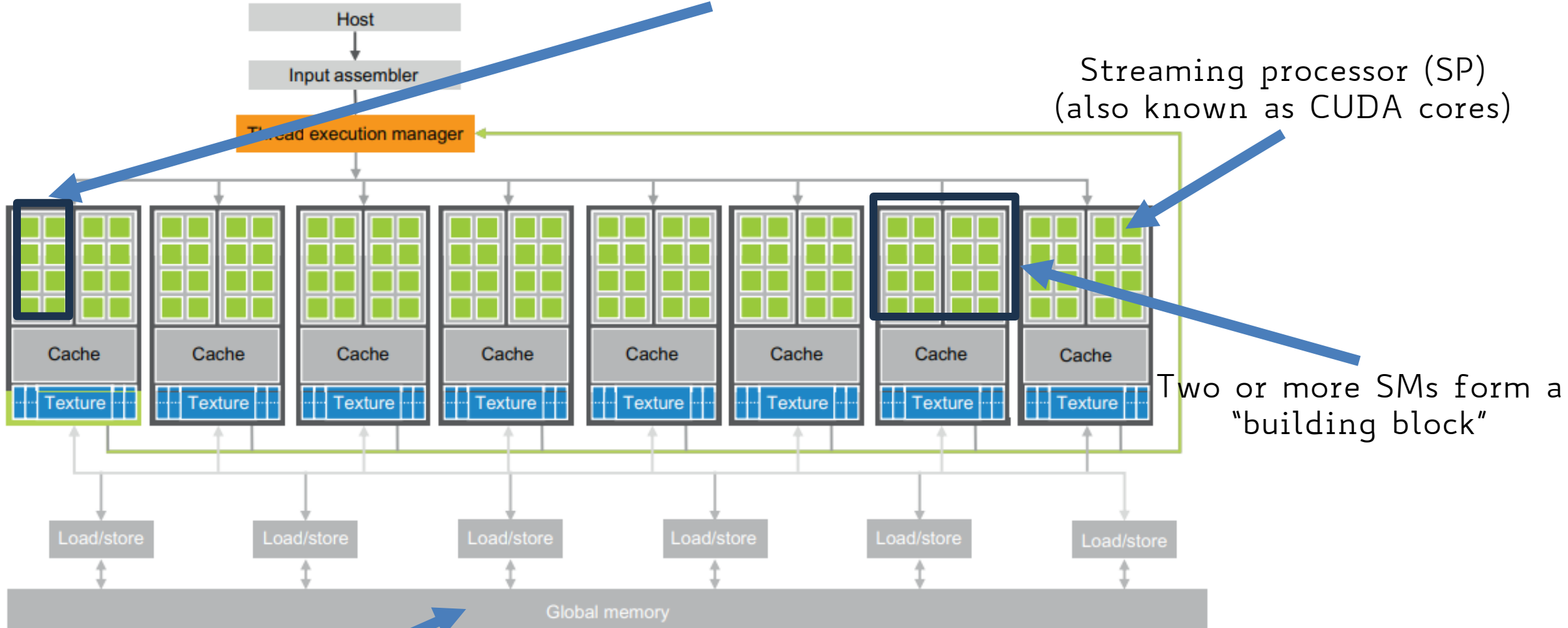
Detrimental for performance
(GPU should not do I/O)
Only use it for debugging purposes

Questions?

Architecture of a CUDA-capable GPU

Streaming multiprocessor (SM)

(SPs on the same SM share control logic and instruction cache)



High-Bandwidth Memory

How many CUDA cores?
15,000 on NVIDIA H100

Thread Scheduling

- Each thread runs on a streaming processor (CUDA core)
- Cores on the same SM share the control unit (i.e., they must **synchronously** execute the same instruction)
- Different SMs can run different kernels
- Each block runs on an SM (i.e., I can't have a block spanning over multiple SMs, but I can have more blocks running on the same SM)
- Once a block is fully executed, the SM will run the next one
- Not all the threads in a group run concurrently

Warps

- They are executed in groups called *warps* (in current GPUs, the size of a warp is 32 – might change in the future, check the *warpSize* variable)
- Thread in a block are split into warps according to their intra-block ID (i.e., the first 32 threads in a block belong to the same warp, the next 32 threads to a different warp, etc...)
- All threads in a warp executed according to the Single Instruction, Multiple Data (SIMD) model—i.e., at any instant in time, one instruction is fetched and executed for all threads in the warp. Consequently, all threads in a warp will always have the same execution timing.
- Several warp schedulers (e.g., 4) can be present on each SM. I.e., multiple (e.g., 4) warps can run at the same time, each possibly following a different execution path

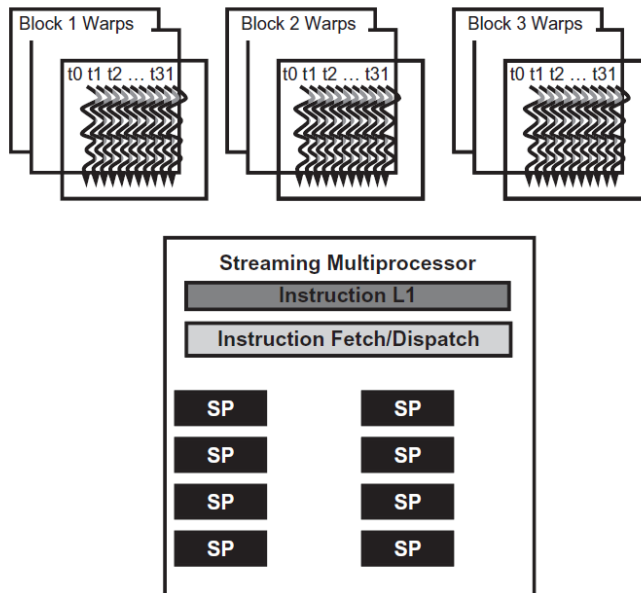
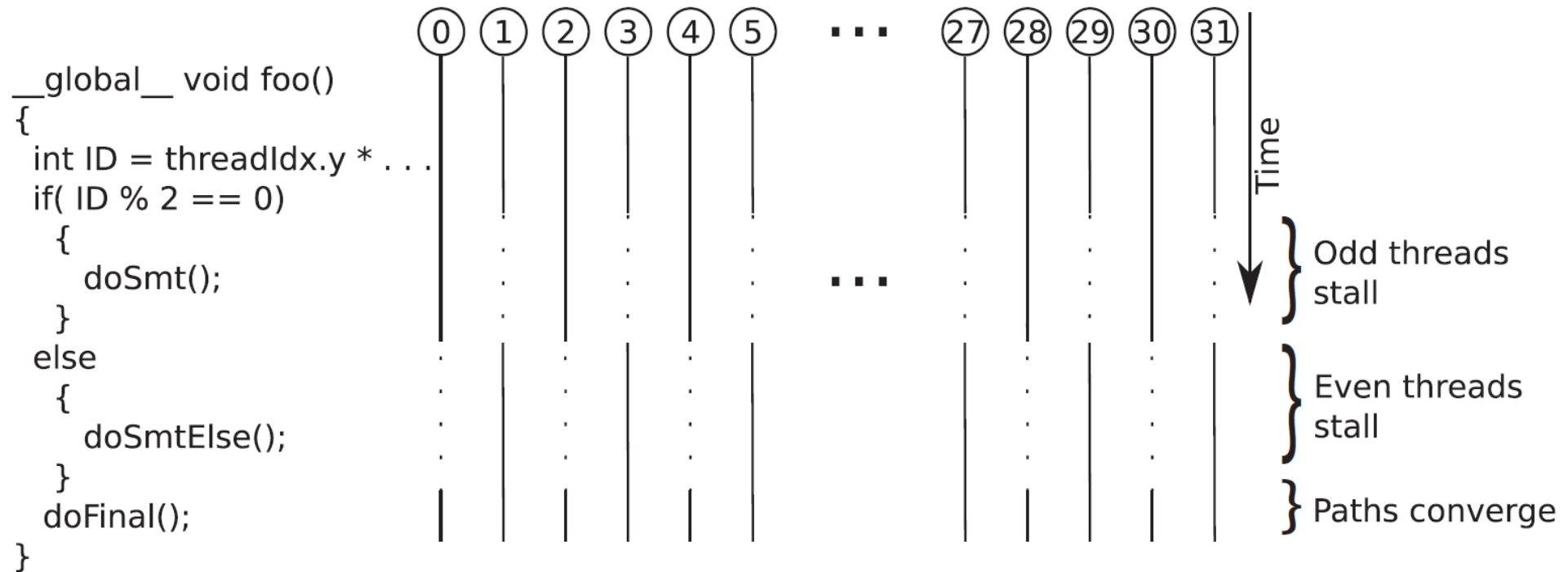


FIGURE 3.13

Blocks are partitioned into warps for thread scheduling.

Warp Divergence

- All threads in a warp executed according to the Single Instruction, Multiple Data (SIMD) model—i.e., at any instant in time, one instruction is fetched and executed for all threads in the warp. Consequently, all threads in a warp will always have the same execution timing.
- What happens if the result of a conditional operation leads them to different paths?
- All the divergent paths are evaluated (if threads branch into them) in sequence until the paths merge again.
- The threads that do not follow the path currently being executed are stalled.



Context Switching

- Usually a SM has more *resident* blocks/warps than what it is able to concurrently run
- Each SM can switch seamlessly between warps
- Each thread has its own private execution context that is maintained on-chip (i.e., context switch comes for free)
- When an instruction to be executed by a warp needs to wait for the result of a previously initiated long-latency operation, the warp is not selected for execution (e.g., memory read, long latency floating-point operations).
- Instead, another resident warp that is no longer waiting for results will be selected for execution.
- This mechanism of filling the latency time of operations with work from other threads is often called "latency tolerance" or "latency hiding"
- Given a sufficient number of warps, the hardware will likely find a warp to execute at any point in time, thus making full use of the execution hardware in spite of these long-latency operations.
- With warp scheduling, the long waiting time of warp instructions is "hidden" by executing instructions from other warps.
- This ability to tolerate long-latency operations is the main reason GPUs do not dedicate nearly as much chip area to cache memories and branch prediction mechanisms as do CPUs.

Example 1

- A CUDA device allows up to 8 blocks and 1024 threads per SM, and 512 threads per block
- Shall we use 8x8, 16x16, or 32x32 thread blocks?
- 8x8 blocks:
 - We would have 64 threads per block
 - To fill the 1024 threads we can have for each SM, we would need $1024/64 = 16$ blocks
 - However, we can have at most 8 blocks per SM, thus we would end with only $64 \times 8 = 512$ threads per SM
 - We are not fully utilizing the resources. Most likely, at some time, the scheduler might not find threads to schedule when some thread is waiting for long-latency operations
- 16x16
 - We would have 256 threads per block
 - To fill the 1024 threads we can have for each SM, we would need $1024/256 = 4$ blocks
 - This would allow to have 1024 threads on the SM, so there will be a lot of opportunities for latency hiding
- 32x32
 - We would have 1024 threads per block, which is higher than the 512 threads per block we can have
- In practice, it is more complicated than this, the usage of other resources as registers and shared memory must be taken into account (we will see how)

Example 2

- A CUDA device allows up to 8 blocks and 1536 threads per SM, and 1024 threads per block
- Shall we use 8x8, 16x16, or 32x32 thread blocks
- 8x8 blocks:
 - We would have 64 threads per block
 - To fill the 1536 threads we can have for each SM, we would need $1536/64 = 24$ blocks
 - Only 512 threads (8x8x8) would go into each SM
- 16x16
 - We would have 256 threads per block
 - To fill the 1536 threads we can have for each SM, we would need $1536/256 = 6$ blocks
 - We achieve full capacity (unless other resources constraints come into play)
- 32x32
 - We would have 1024 threads per block. Only one block can fit (two would bring the number of threads to 2048, which is higher than 1536)
 - Thus, we would use only 2/3 of the thread capacity of the SM (1024 out of 1536)

Example 3

- A grid of 4x5x3 blocks, each made of 100 threads
- The GPU has 16 SMs
- Thus, we have 4x5x3=60 blocks, that need to be distributed over 60 16SMs
 - Let's assume that they are distributed round-robin
 - 12 SMs will receive 4 blocks, and 6 SMs will receive 3 blocks
 - Inefficient! While the first 12 SMs process the last block, the other 6 SMs are idle
- A block contains 100 threads, which are divided into $100/32 = 4$ warps
 - The first three warps have 32 threads, and the last one have 4 threads (32+32+32+4)
 - Assume we can only schedule a warp at a time (e.g., because we have 32 CUDA cores per SM)
 - The last warp would only use 4 out of the 32 available cores (87.5% of the cores on each SM will be unused)
- Take home message: pay attention to how you define the grid and block sizes (we will come back to this again)

Device properties

Example : Listing all the GPUs in a system

```
int deviceCount = 0;
cudaGetDeviceCount(&deviceCount);
if(deviceCount == 0)
    printf("No CUDA compatible GPU exists.\n");
else
{
    cudaDeviceProp pr;
    for(int i=0;i<deviceCount;i++)
    {
        cudaGetDeviceProperties(&pr, i);
        printf("Dev #%i is %s\n", i, pr.name);
    }
}
```


Device Properties

```
struct cudaDeviceProp{
    char name[256]; // A string identifying the device
    int major;      // Compute capability major number
    int minor;      // Compute capability minor number
    int maxGridSize [3];
    int maxThreadsDim [3];
    int maxThreadsPerBlock;
    int maxThreadsPerMultiProcessor;
    int multiProcessorCount;
    int regsPerBlock; // Number of registers per block
    size_t sharedMemPerBlock;
    size_t totalGlobalMem;
    int warpSize;
    . . .
};
```

Memory Hierarchy

Memory Accesses

- Data allocated on host memory is not visible from the GPU, and viceversa

```
int *mydata = new int[N];  
. . . // populating the array  
foo<<<grid, block>>>(mydata, N);    // NOT POSSIBLE!
```

- Instead, it must explicitly be copied from/to host to GPU

Memory Allocation and Copy

```
// Allocate memory on the device.
cudaError_t cudaMalloc ( void** devPtr, // Host pointer address,
                                // where the address of
                                // the allocated device
                                // memory will be stored
                                size_t size ) // Size in bytes of the
                                                // requested memory block

// Frees memory on the device.
cudaError_t cudaFree ( void* devPtr ); // Parameter is the host
                                         // pointer address, returned
                                         // by cudaMalloc

// Copies data between host and device.
cudaError_t cudaMemcpy ( void* dst, // Destination block address
                        const void* src, // Source block address
                        size_t count, // Size in bytes
                        cudaMemcpyKind kind ) // Direction of copy.
```

`cudaError_t` is an enumerated type. If a CUDA function returns anything other than `cudaSuccess (0)`, an error has occurred.

Memory Copy Kind

The `cudaMemcpyKind` parameter of `cudaMemcpy` is also an enumerated type. The `kind` parameter can take one of the following values:

- `cudaMemcpyHostToHost = 0`, Host to Host
- `cudaMemcpyHostToDevice = 1`, Host to Device
- `cudaMemcpyDeviceToHost = 2`, Device to Host
- `cudaMemcpyDeviceToDevice = 3`, Device to Device (for multi-GPU configurations)
- `cudaMemcpyDefault = 4`, used when Unified Virtual Address space capability is available (see [Section 6.7](#))

Example: Vector Addition

```
void vecAdd(float* h_A, float* h_B, float* h_C, int n)
{
    int size = n * sizeof(float);
    float *d_A, *d_B, *d_C;

    cudaMalloc((void **) &d_A, size);
cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
    cudaMalloc((void **) &d_B, size);
cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);

    cudaMalloc((void **) &d_C, size);

    // Kernel invocation code - to be shown later
    ...

cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);

    // Free device memory for A, B, C
    cudaFree(d_A); cudaFree(d_B); cudaFree (d_C);
}
```

FIGURE 2.10

A more complete version of `vecAdd()`.