GPU Architectures and Basic CUDA Programming

Tong Shu

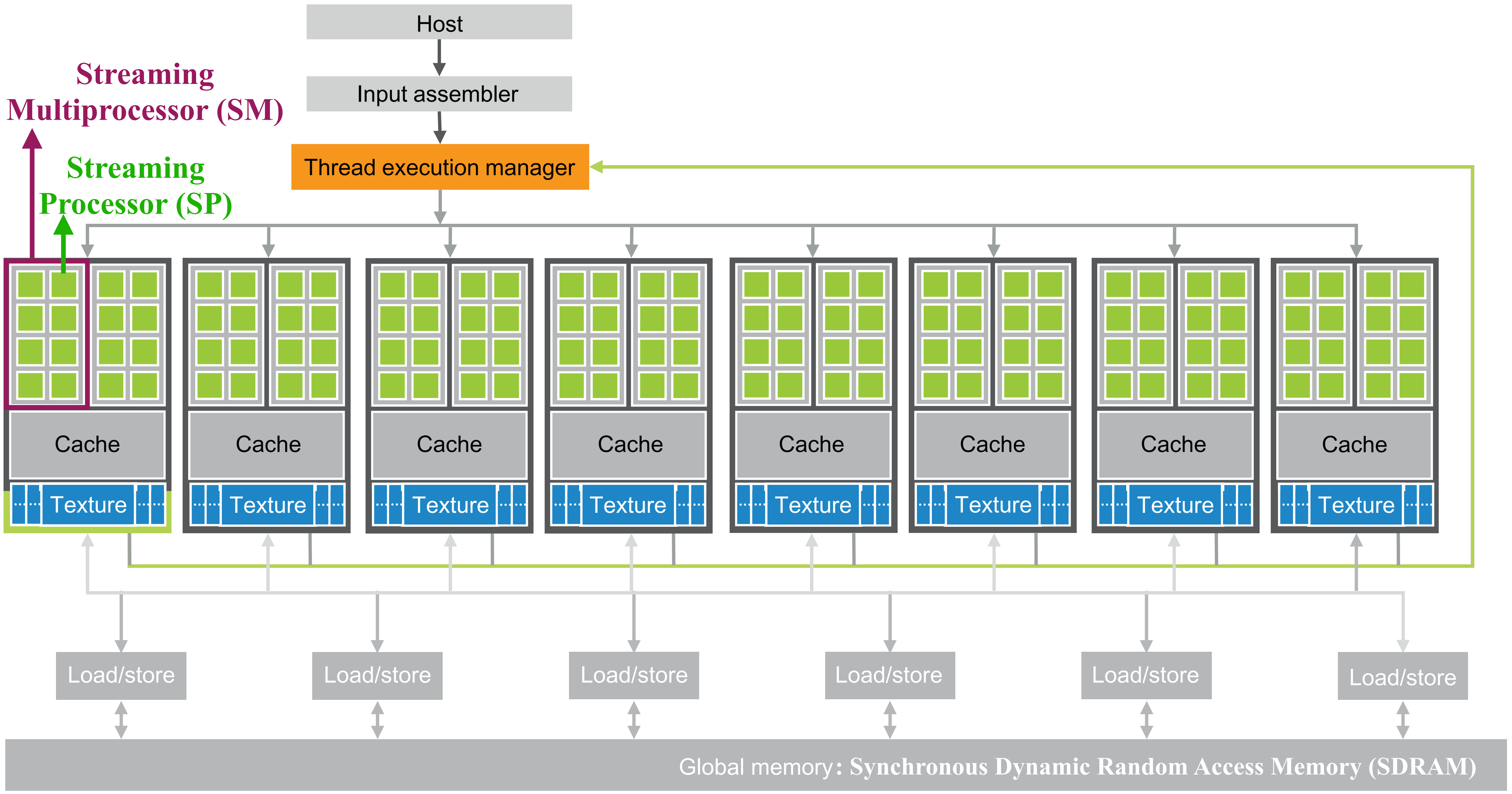
Day 3 — Part 2

DL-GPU Workshop 2023

07/26/2023

Architectures of State-of-the-Art GPUs

Architecture of a CUDA-capable GPU

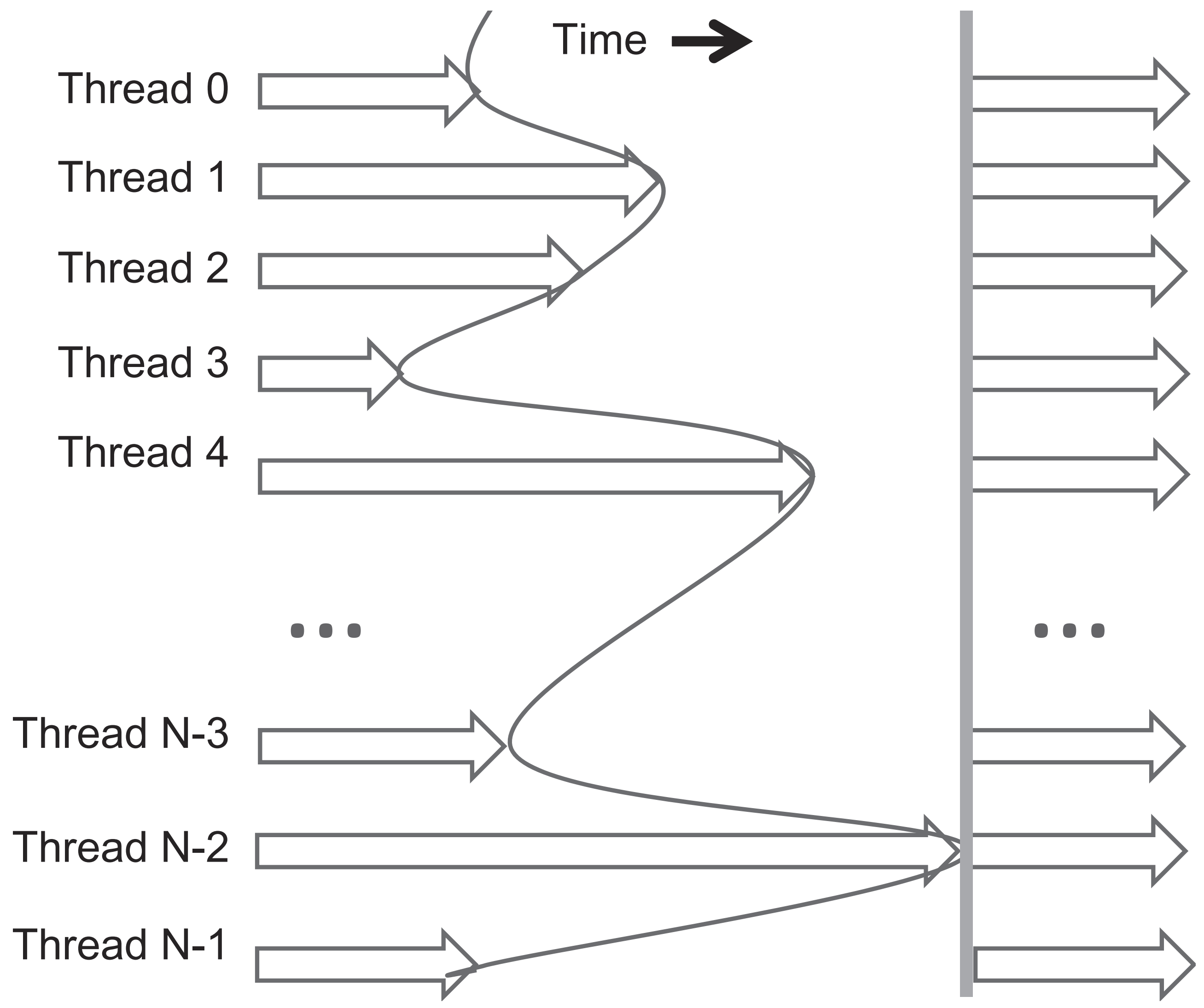

Host
Input assembler
Thread execution manager
Streaming Multiprocessor (SM)
Streaming Processor (SP)
Cache
Texture
Load/store
Global memory: Synchronous Dynamic Random Access Memory (SDRAM)

GPU SDRAMs differ from system DRAMs on the CPU motherboard because they are essentially the frame buffer memory used for graphics.

- For graphics applications: hold video images and texture information for 3D rendering.
- For computing: function as very high-bandwidth off-chip memory.

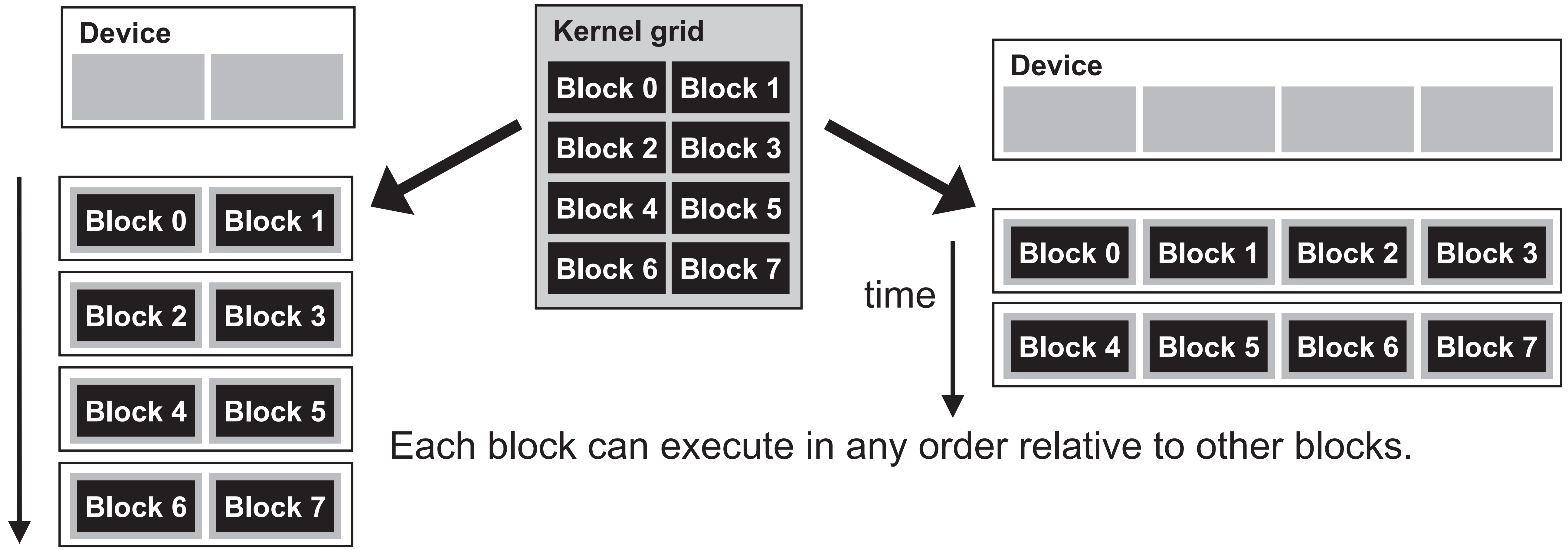
Thread Synchronization within a Block

- When a thread calls the barrier synchronization function __syncthreads(), it will be held at the calling location until every thread in the same block reaches the location.
  - For an if-then-else statement, if each path has a __syncthreads() statement, either all threads in a block execute the then-path or all of them execute the else-path.
- CUDA runtime systems assign the same execution resources to all threads in a block as a unit.
  - Ensure the temporal proximity of all threads in a block
  - Prevent excessive or indefinite waiting time during barrier synchronization



Transparent Scalability

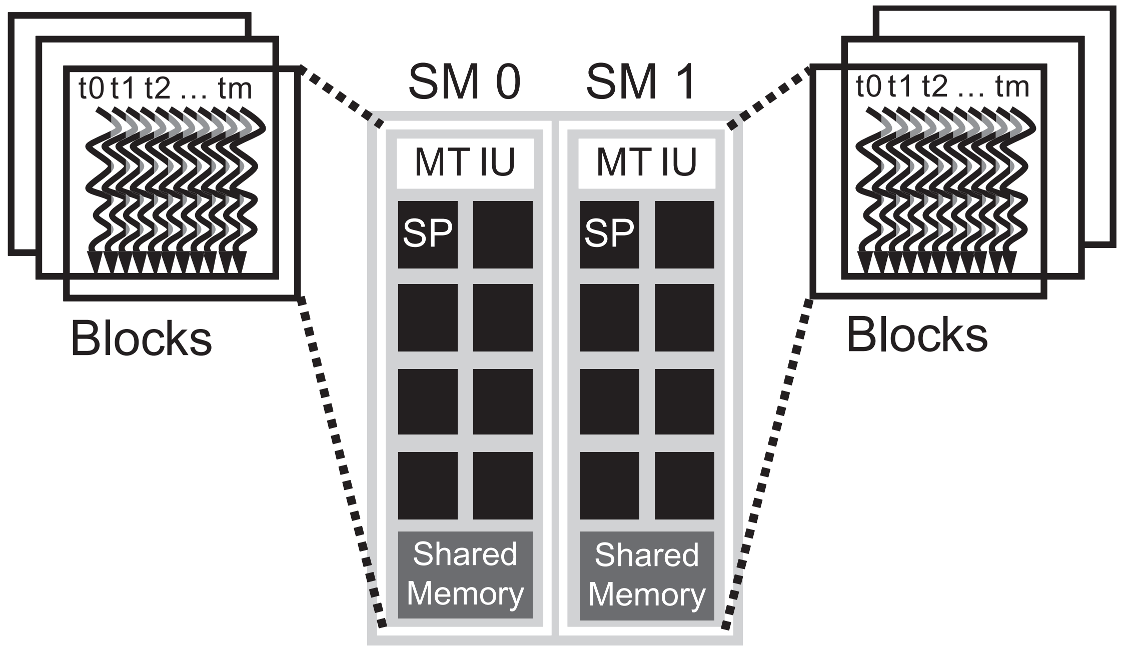
- The ability to execute the same application code on hardware with different numbers of execution resources is referred to as transparent scalability.



- To allow a kernel to maintain transparent scalability, the simple method for threads in different blocks to synchronize with each other is to terminate the kernel and start a new kernel for the activities after the synchronization point.

Resource Assignment

- Multiple blocks can be assigned to each streaming multiprocessor (SM).



- Limitations on
  - The number of SMs
  - The number of blocks that can be assigned to each SM
  - The number of blocks that can be actively executing in a CUDA device/GPU
  - The number of threads that can be assigned to an SM

- The runtime system maintains a list of blocks that need to execute and assigns new blocks to SMs when previously assigned blocks complete execution.

CUDA C API: Querying the Number of Devices

- __host__ __device__ cudaError_t cudaGetDeviceCount (int* count)
  - Parameters
    - count: Returns the number of compute-capable devices
  - Returns
    - cudaSuccess = 0
  - Description
    - Returns in *count the number of devices with compute capability greater or equal to 2.0 that are available for execution.
  - Example

```
int count;
cudaGetDeviceCount(&count);
```

CUDA C API: Querying Device Properties

- __host__ cudaError_t cudaGetDeviceProperties (cudaDeviceProp* prop, int device)
  - Parameters
    - prop: properties for the specified device
    - device: device number to get properties for
      - The CUDA runtime numbers all available devices in the system from 0 to count-1, where count is returned by function cudaGetDeviceCount.
  - Returns
    - cudaSuccess = 0
    - cudaErrorInvalidDevice = 101: The device ordinal supplied by the user does not correspond to a valid CUDA device.
  - Description: returns in *prop the properties of the device device.
  - Example

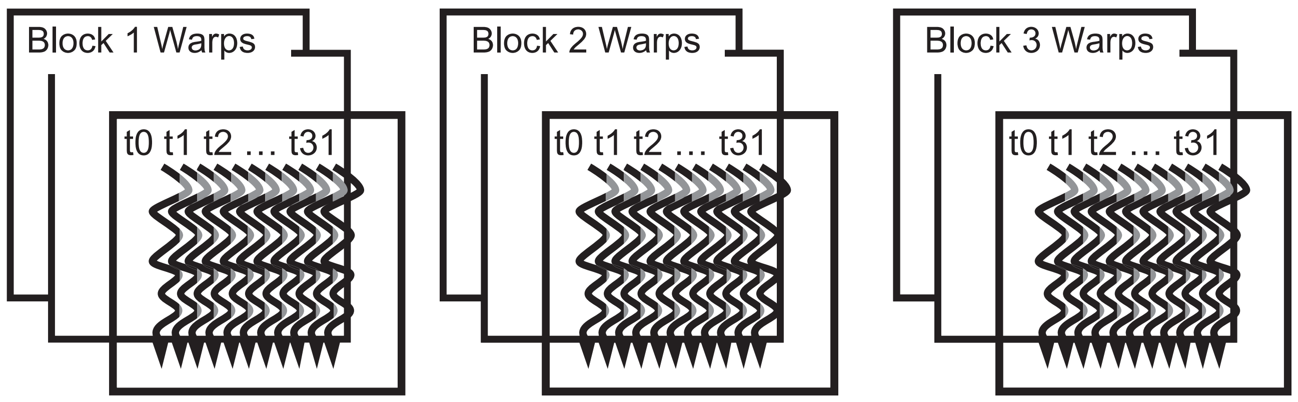
```
cudaDeviceProp prop;
for (int i = 0; i < count; i++) {
        cudaGetDeviceProperties(&prop, i);
        // Decide if device has sufficient resources and capabilities
}
```

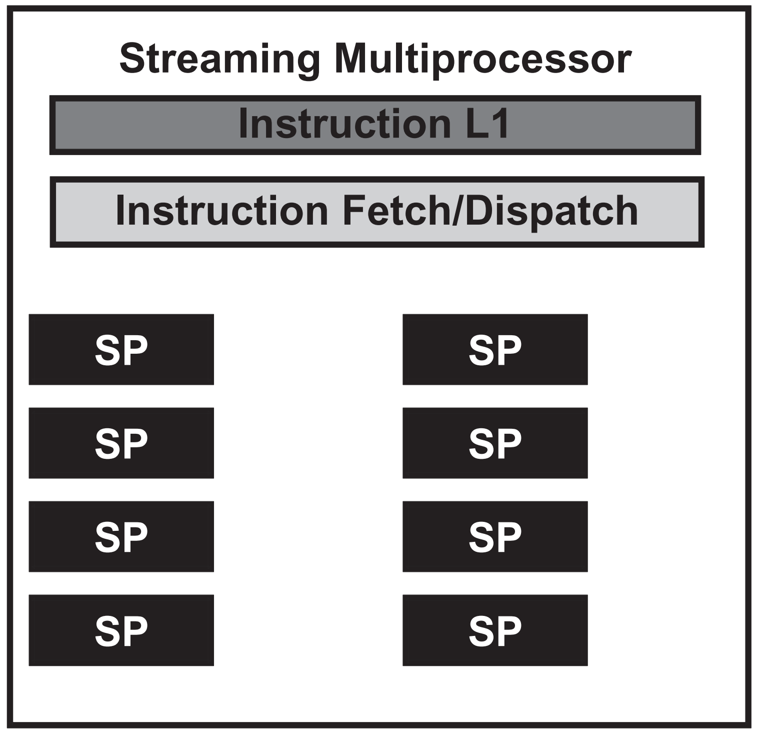
CUDA C Built-in Type: cudaDeviceProp

- C struct type with fields representing the properties of a CUDA device
  - Field prop.maxThreadsPerBlock with the type of int: the maximum number of threads allowed in a block in the queried device
  - Fields prop.maxThreadsDim[0], prop.maxThreadsDim[1], and prop.maxThreadsDim[2] with the type of int: the maximum number of threads allowed along the x, y, and z dimensions of a block
  - Fields prop.maxGridSize[0], prop.maxGridSize[1], and prop.maxGridSize[2] with the type of int: the maximum number of blocks allowed along the x, y, and z dimensions of a grid
  - Field prop.warpSize with the type of int: the warp size in threads
  - Field prop.multiProcessorCount with the type of int: the number of streaming multiprocessors on the device
  - Field prop.clockRate with the type of int: clock frequency in kilohertz

Warp

- A block is divided into 32 thread units, called warps.
- The size of warps is implementation-specific.
- The warp is the unit of thread scheduling in SMs.
- Each warp consists of 32 threads of consecutive threadIdx values.
- An SM execute all threads in a warp following 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.
- The hardware can execute instructions for a small subset of all warps in the SM.



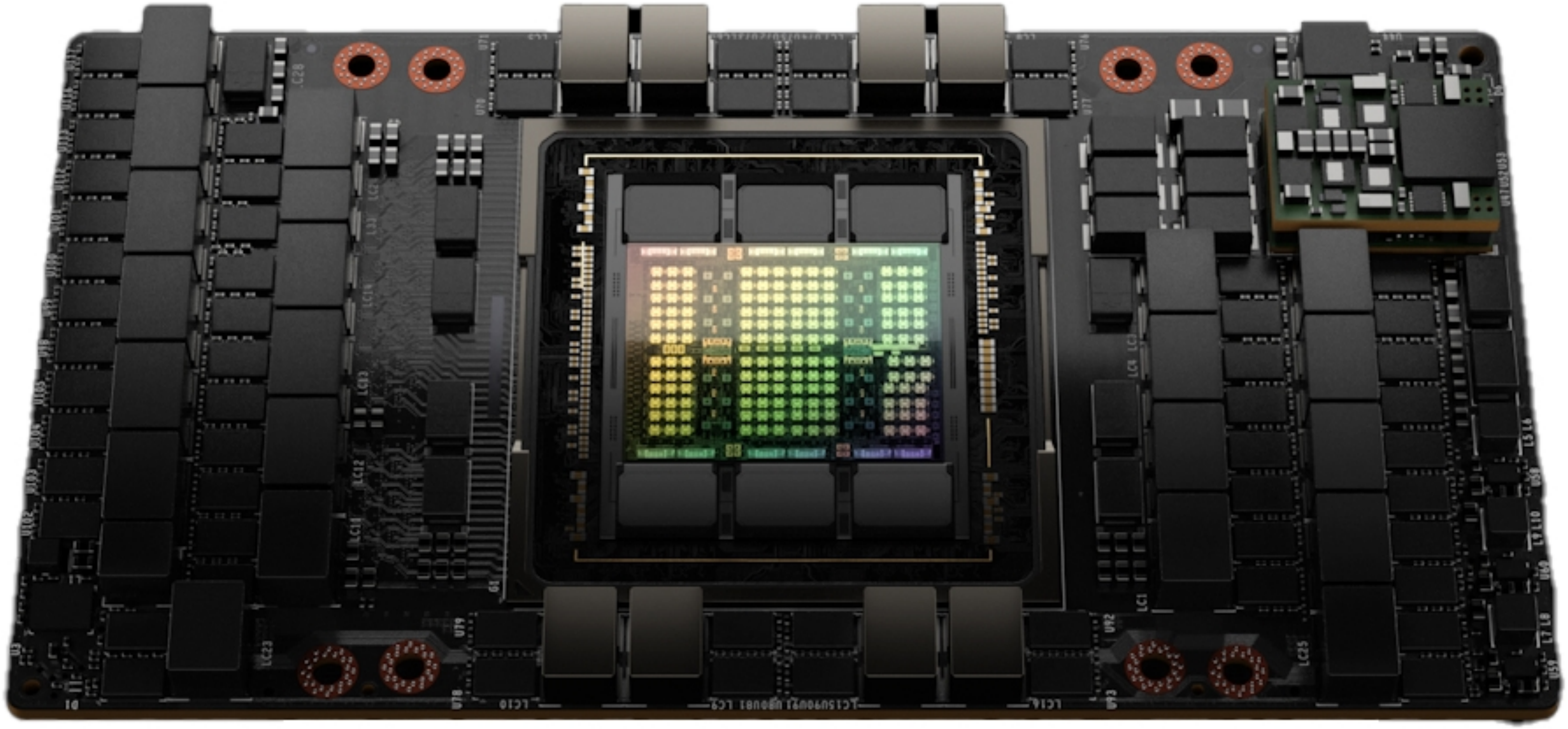


Thread Scheduling and Latency Tolerance

- Latency tolerance/hiding: the mechanism of filling the latency time of operations with work from other threads
  - When an instruction to be executed by a warp needs to wait for the result of a previously initiated long-latency operation, another resident warp that is no longer waiting for results is selected for execution.

- Zero-overhead thread scheduling
  - The selection of ready warps for execution avoids introducing idle or wasted time into the execution timeline

NVIDIA H100 GPU on new SXM5 Module



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

Hopper Architecture Features of Nvidia H100 GPU

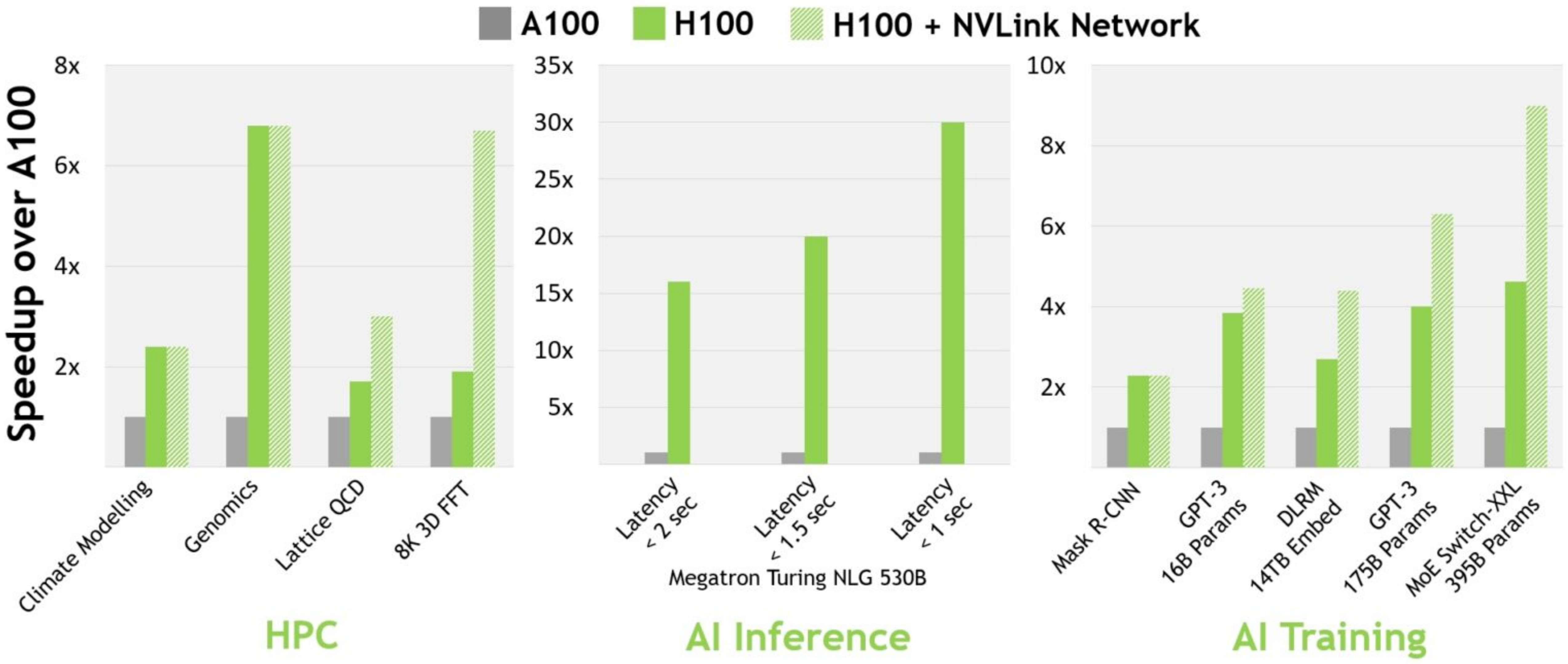
- The fourth-generation tensor cores perform faster matrix computations than ever before on an even broader array of AI and HPC tasks.

- A new transformer engine enables H100 to deliver up to 9x faster AI training and up to 30x faster AI inference speedups on large language models compared to Nvidia A100.

- The new NVLink network interconnect enables GPU-to-GPU communication among up to 256 GPUs across multiple compute nodes.

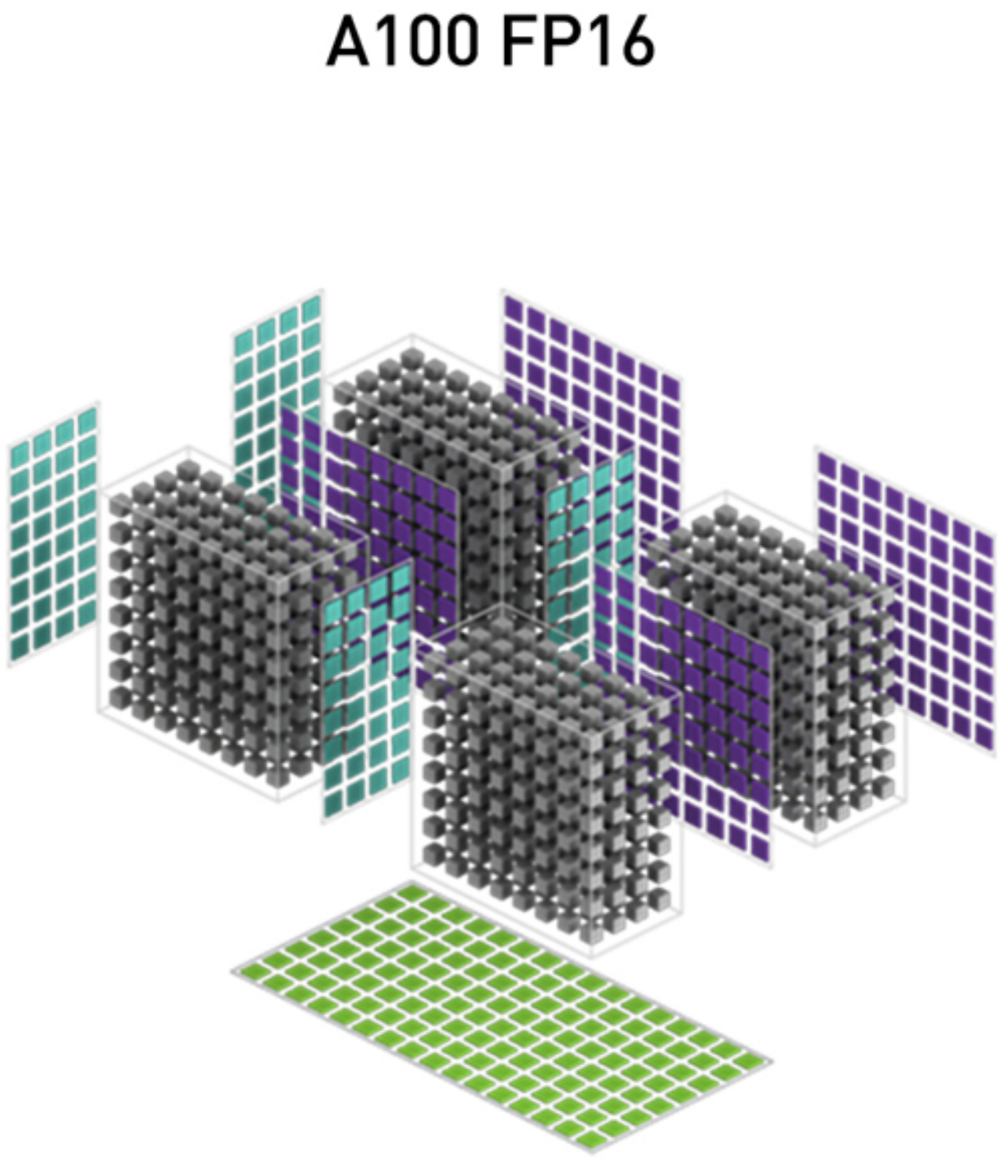
- Secure MIG partitions the GPU into isolated, right-size instances to maximize quality of service (QoS) for smaller workloads.

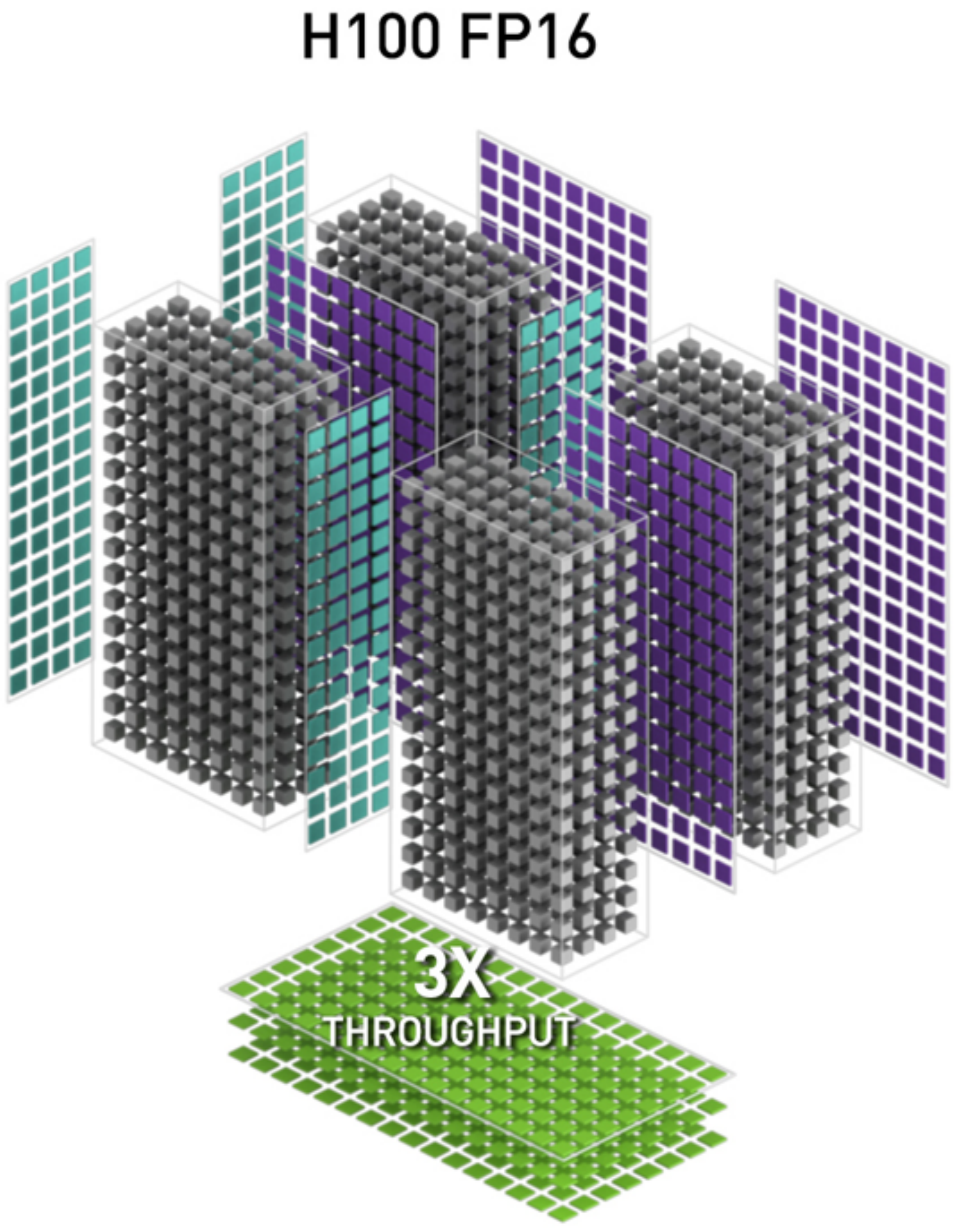
H100 v.s. A100 in HPC and AI



H100 SM Feature

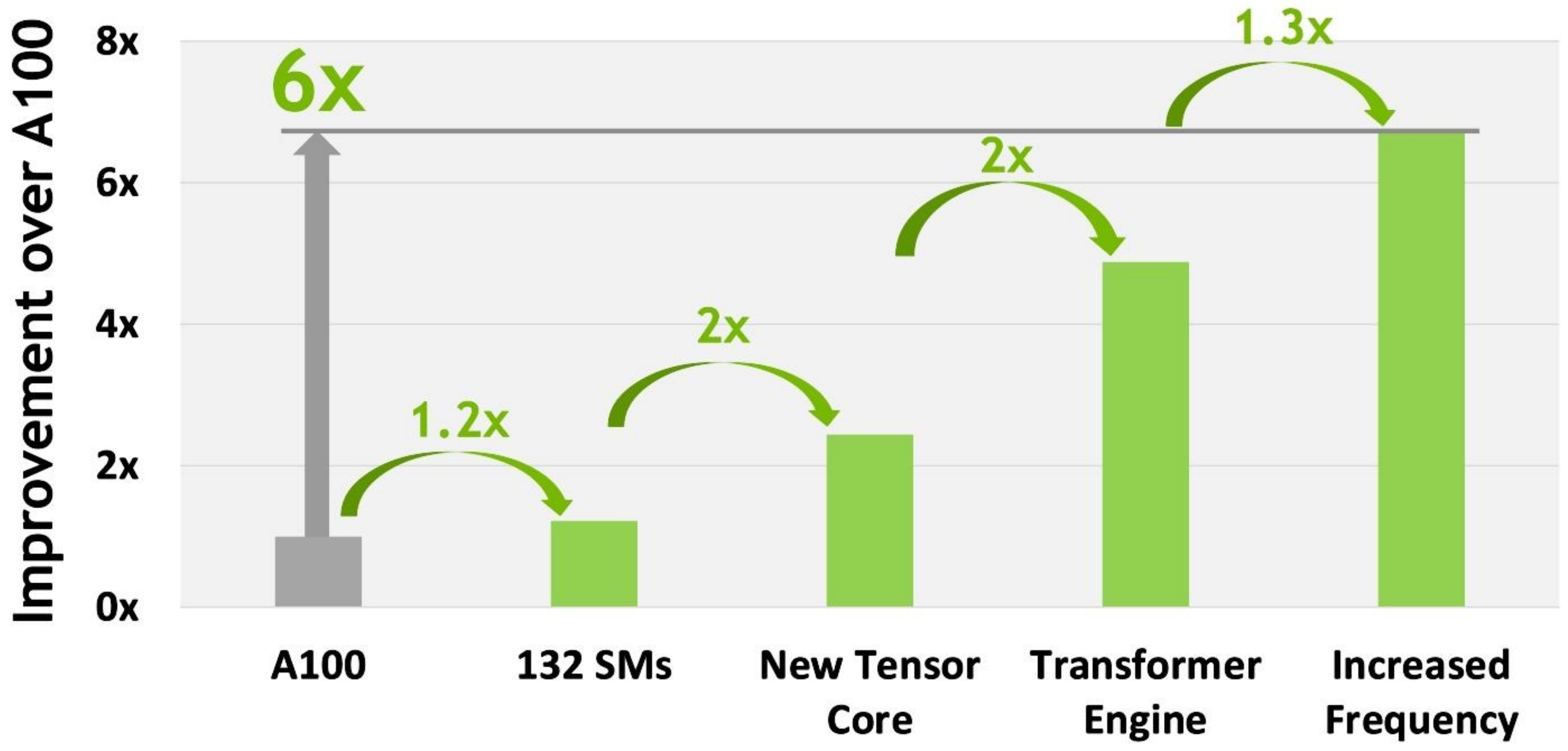
- H100 FP16 Tensor Core has 3x throughput compared to A100 FP16 Tensor Core





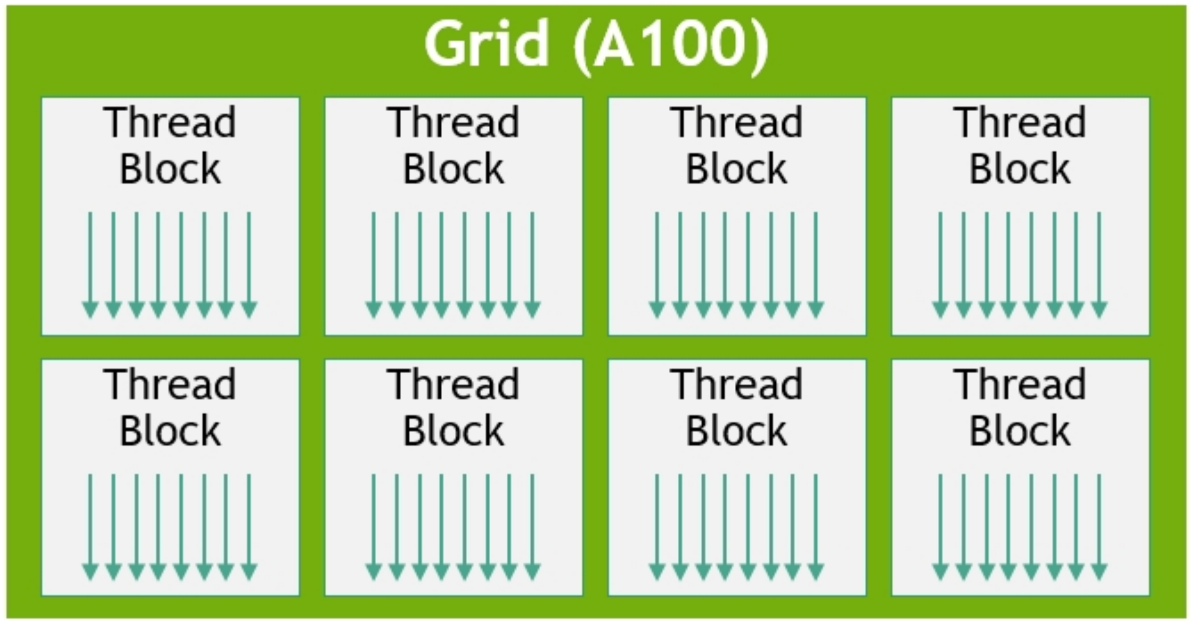
H100 SM Feature

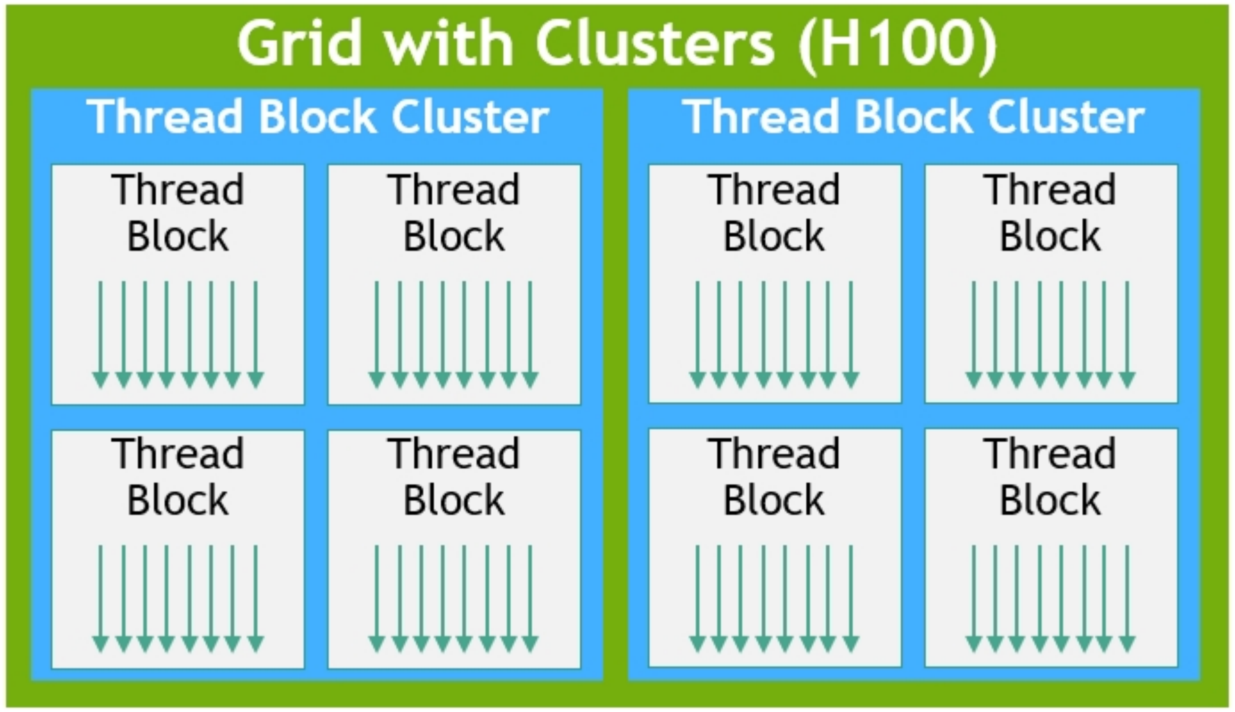
- H100 compute improvement
  - 132 SMs provide a 22% SM count increase over the A100 108 SMs
  - Each of the H100 SMs is 2x faster, due to the fourth-generation Tensor Core.
  - Within each Tensor Core, the new FP8 format and associated transformer engine provide another 2x improvement.
  - Increased clock frequencies in H100 deliver another approximately 1.3x performance improvement.



H100 SM Feature

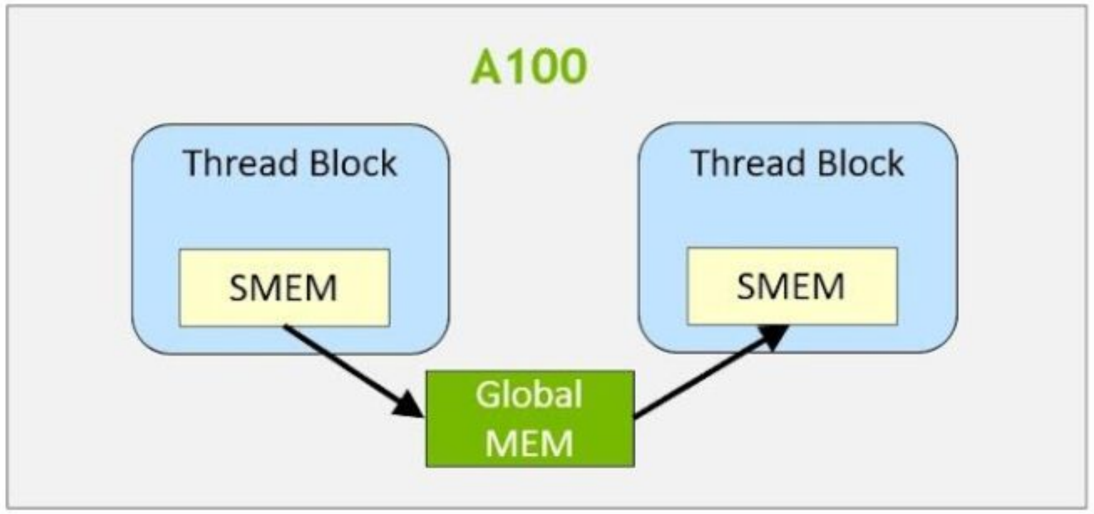
- Thread block clusters and grids with clusters
  - A cluster is a group of thread blocks that are guaranteed to be concurrently scheduled onto a group of SMs.
  - Clusters have hardware-accelerated barriers and new memory access collaboration capabilities.
  - A dedicated SM-to-SM network for SMs in a GPU processing cluster provides fast data sharing between threads in a cluster.
  - Cluster capabilities can be leveraged from the CUDA cooperative_groups API.

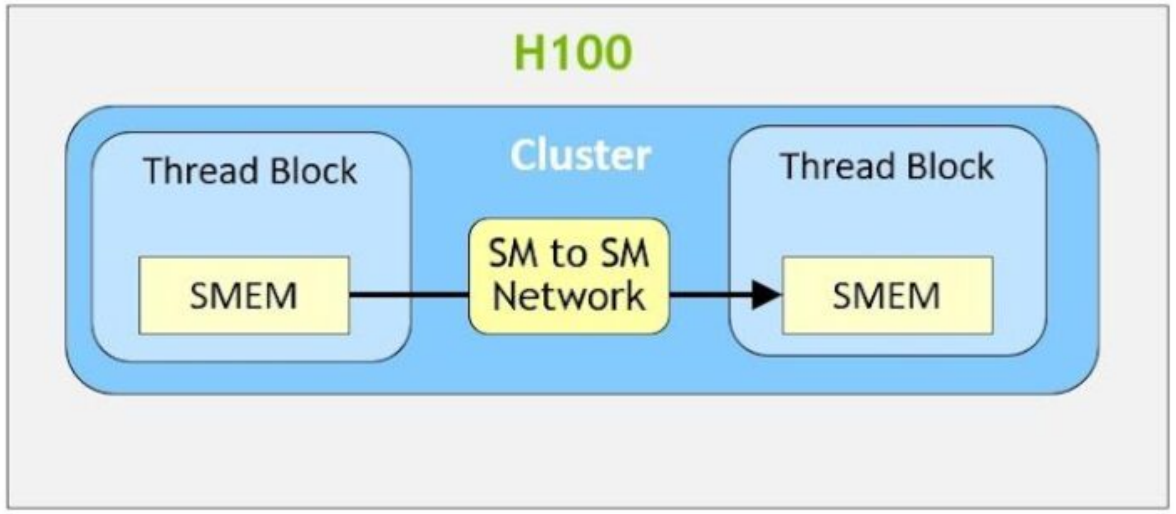




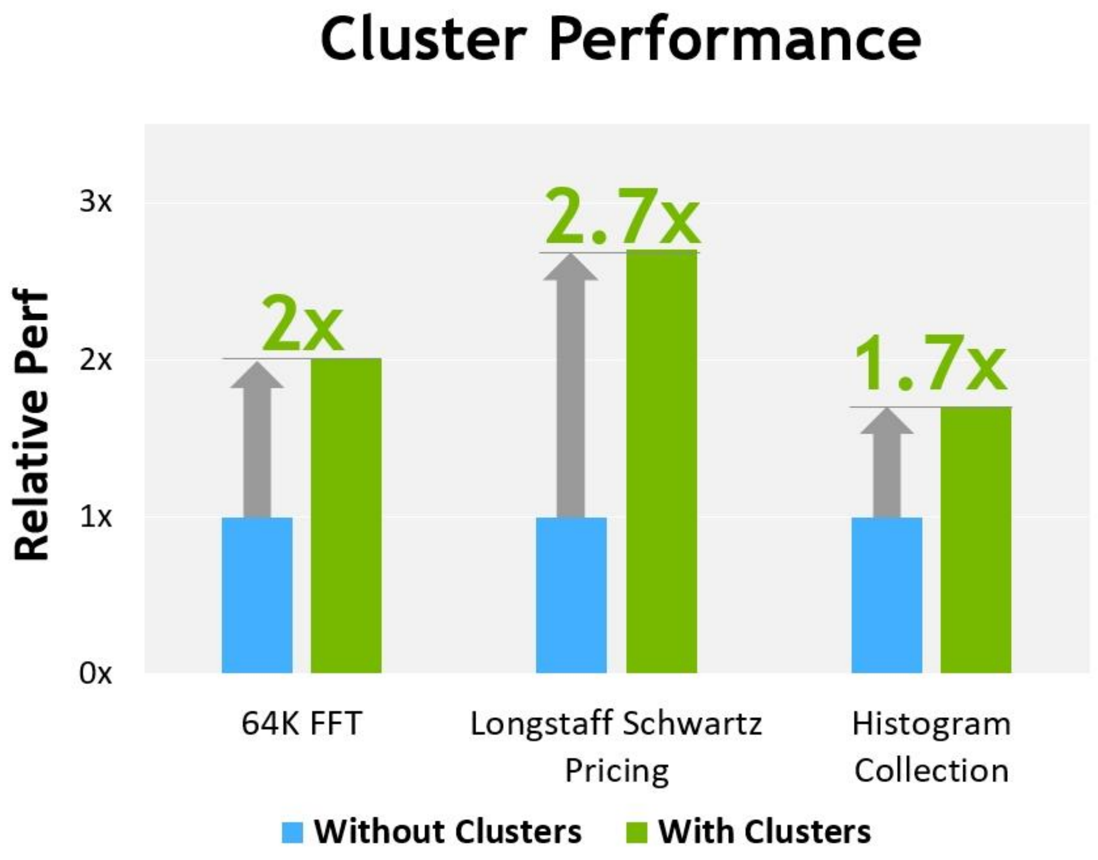
H100 SM Feature

- Distributed shared memory
  - Thread-block-to-thread-block data exchange (A100 vs. H100 with clusters)



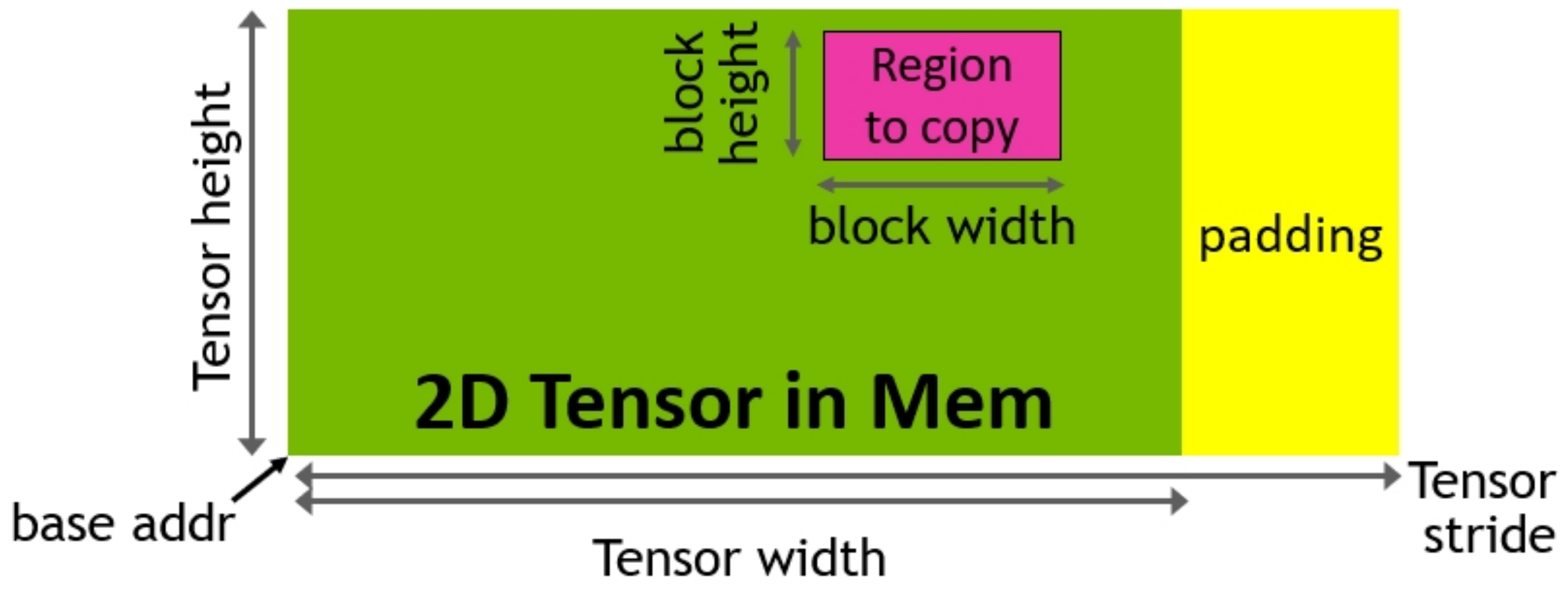


  - Cluster vs. non-cluster performance comparisons



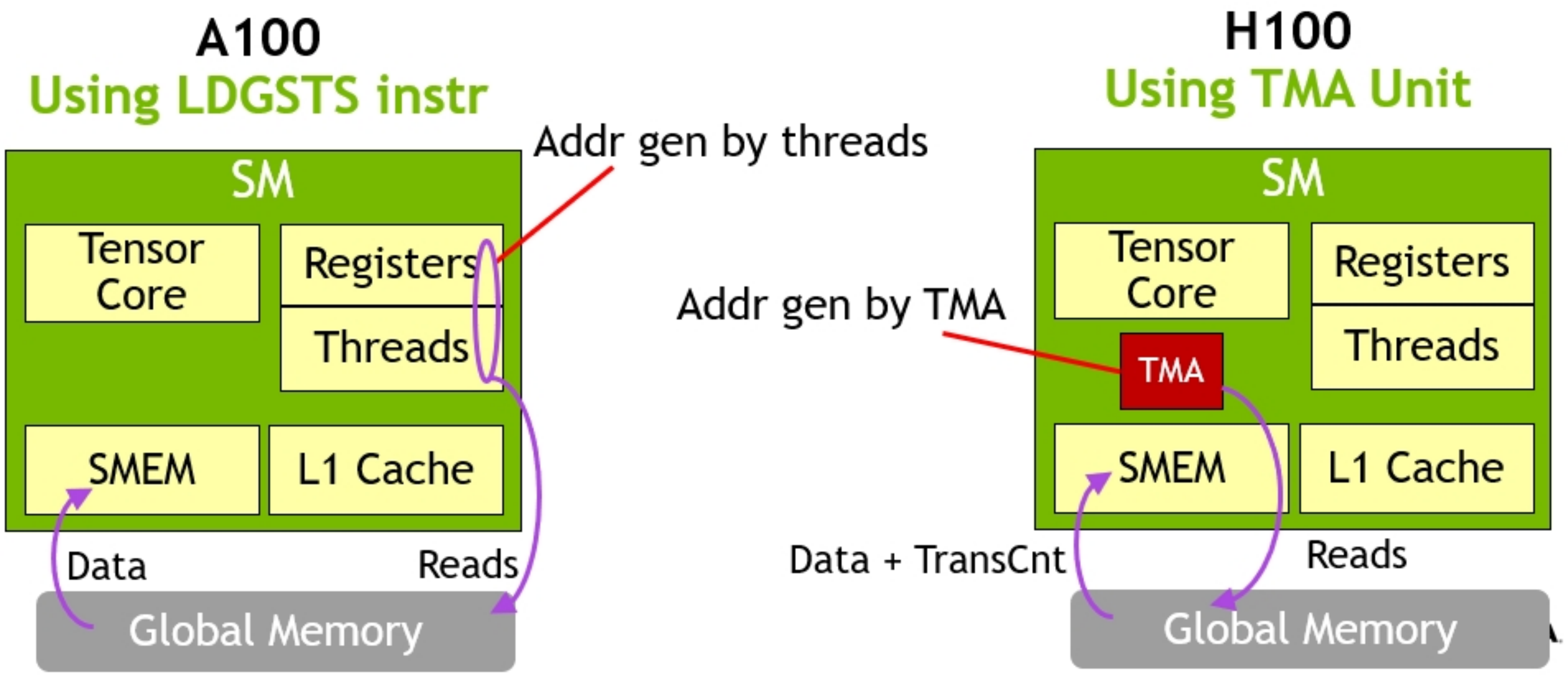
H100 SM Feature

- Tensor memory accelerator: efficiently transfers large blocks of data between global memory and shared memory.
  - TMA address generation through a copy descriptor that specifies data transfers using tensor dimensions and block coordinates instead of per-element addressing



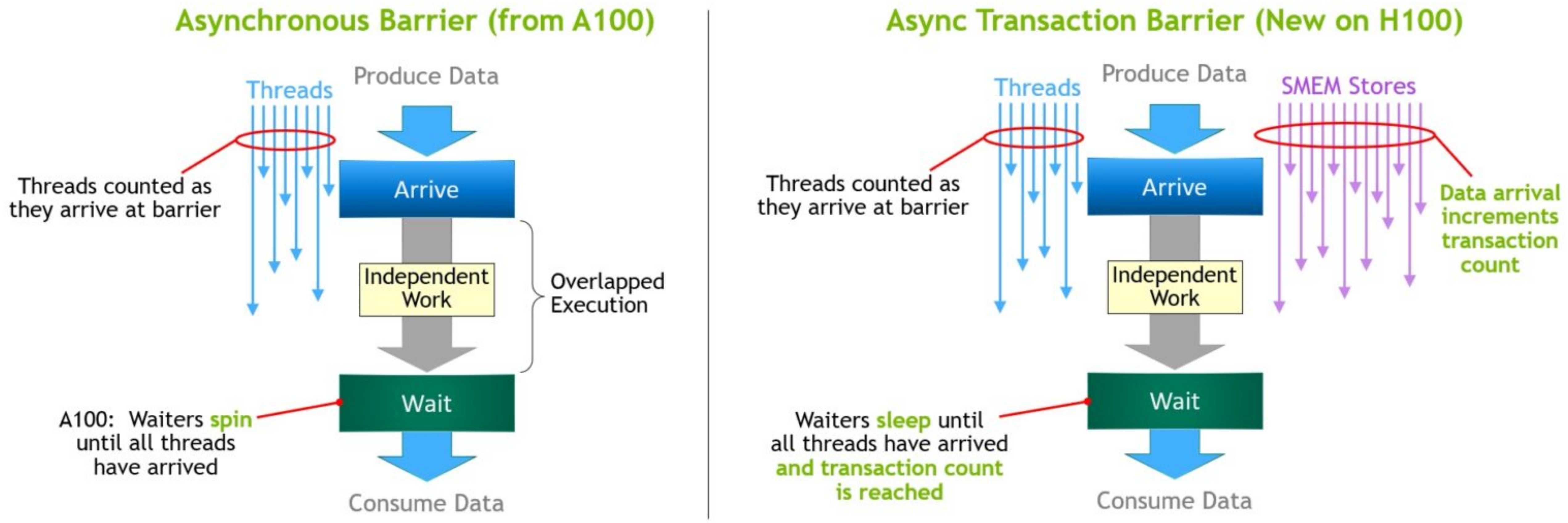
Tensor Memory Accelerator

- A single thread in a warp is elected to issue an asynchronous TMA operation (cuda::memcpy_async) to copy a tensor.
- Multiple threads can wait on a shared memory-based asynchronous barrier cuda::barrier for the completion of the data transfer.
- Advantage: it frees the threads to execute other independent work.
- Asynchronous memory copy with TMA on H100 vs. LoadGlobalStoreShared (LDGSTS) on A100



H100 SM Feature

- Asynchronous barrier in A100
  - First, threads signal Arrive when they are done producing their portion of the shared data. This Arrive is non-blocking so that the threads are free to execute other independent work.
  - Eventually, the threads need the data produced by all the other threads. At this point, they do a Wait, which blocks them until every thread has signaled Arrive.
- Asynchronous transaction barrier in H100



Asynchronous Barrier (from A100)
Threads
Produce Data
Threads counted as they arrive at barrier
Arrive
Independent Work
Overlapped Execution
Wait
A100: Waiters spin until all threads have arrived
Consume Data
Async Transaction Barrier (New on H100)
Threads
Produce Data
SMEM Stores
Threads counted as they arrive at barrier
Arrive
Data arrival increments transaction count
Independent Work
Wait
Waiters sleep until all threads have arrived and transaction count is reached
Consume Data

H100 SM Feature

- Fourth-generation NVLink
  - A100 includes 12 third-generation NVLink links to provide 600 GB/sec total bandwidth.
  - H100 includes 18 fourth-generation NVLink links to provide 900 GB/sec total bandwidth.

- Fourth-generation NVLink Network
  - H100 introduces the new NVLink Network interconnect, a scalable version of NVLink that enables GPU-to-GPU communication among up to 256 GPUs across multiple compute nodes.

- Third-generation NVSwitch
  - NVLink Switch System supports up to 256 GPUs. The connected nodes can deliver 57.6 TBs of all-to-all bandwidth and can supply an incredible one exaFLOP of FP8 sparse AI compute.

Memory and Data Locality in CUDA Programming

Memory and Data Locality

- Compute-to-global-memory-access ratio
  - The number of floating-point calculation performed for each access to the global memory within a region of a program

- Example
  - In a high-end device, the global memory bandwidth is around 1 TB/s.
  - 1000/4 = 250 giga single-precision operands per second can be loaded.
  - 250 giga floating-point operations per second (GFLOPS)
  - 2% of 12 TFLOPS for these high-end devices

```
for (int blurRow = -BLUR_SIZE; blurRow < BLUR_SIZE + 1; ++blurRow) {
    for (int blurCol = -BLUR_SIZE; blurCol < BLUR_SIZE + 1; ++blurCol) {
        int curRow = row + blurRow;
        int curCol = col + blurCol;
        if (curRow >= 0 && curRow <= h - 1 && curCol >= 0 && curCol <= w - 1) {
            sumPixRed += d_in[curRow * w + curCol].r;
            sumPixGreen += d_in[curRow * w + curCol].g;
            sumPixBlue += d_in[curRow * w + curCol].b;
            numPix++;
        }
    }
}
```

Compute-to-global-memory ratio = 1.0

Compute-bound vs. Memory-bound

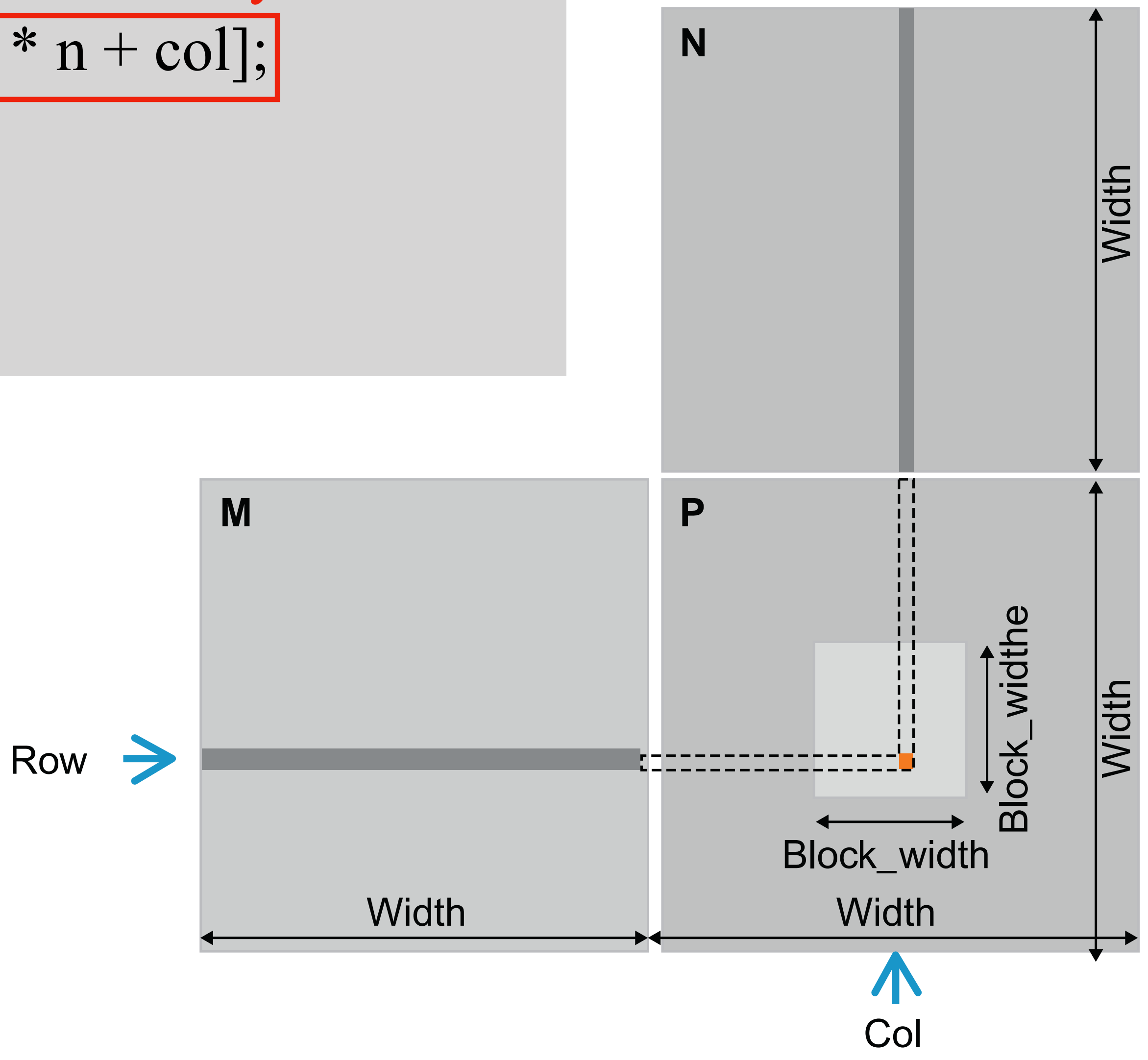
- Compute-bound
  - The time for it to complete a task is determined principally by the speed of the processor.
  - E.g. Processor utilization is high, perhaps at 100% usage.

- I/O bound
  - The time it takes to complete a computation is determined principally by the period spent waiting for input/output operations to be completed.
  - Memory-bound programs
    - Programs whose execution speed is limited by memory access throughput

Kernel Function for Matrix Multiplication

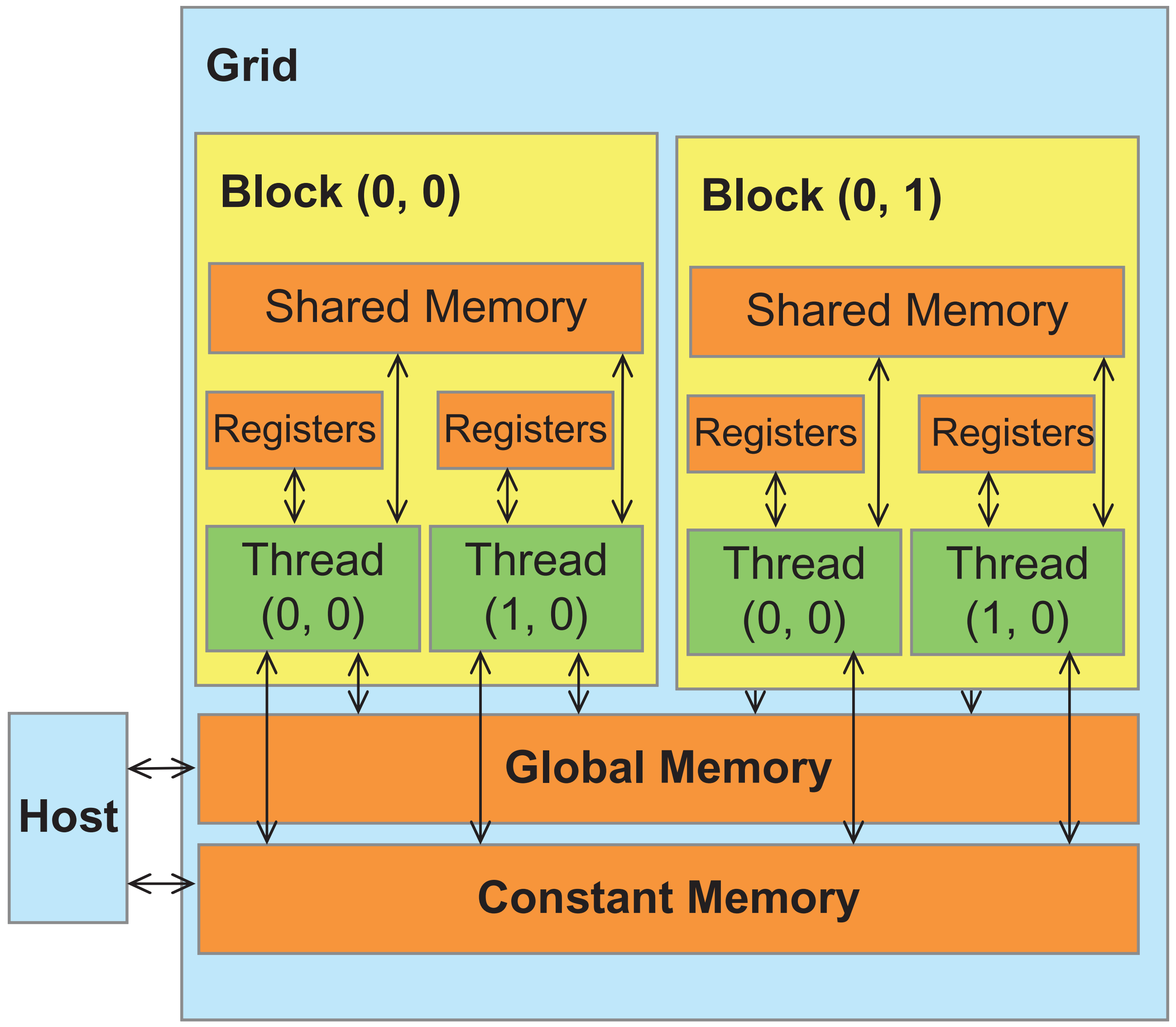
```
// Each thread produces one output matrix element.
__global__ void matMulKernel(float* d_C, float* d_A, float* d_B, int n) {
        // Calculate the column index of the element in d_C
        int col = blockIdx.x * blockDim.x + threadIdx.x;
        // Calculate the row index of the element in d_C
        int row = blockIdx.y * blockDim.y + threadIdx.y;

        if (col < n && row < n) {
                float dotProc = 0;
                for (int k = 0; k < n; k++) {
                        dotProc += d_A[row * n + k] * d_B[k * n + col];
                }
                d_C[row * n + col] = dotProc;
        }
}
```

Compute-to-global-memory ratio = 1.0



CUDA Device Memory Model

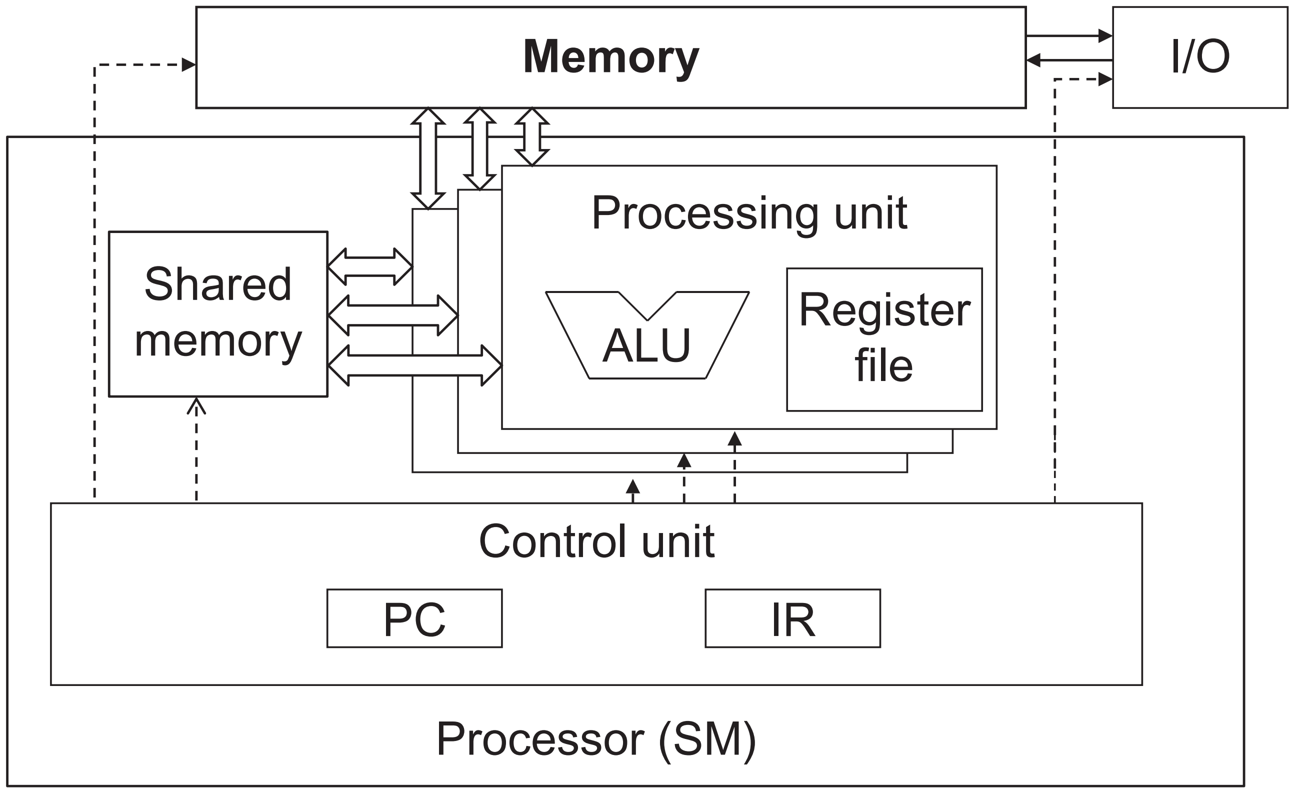
- Device code can
  - Read and write per-thread on-chip registers
    - Bandwidth of registers is at least two orders of magnitude higher than that of the global memory
  - Read and write per-block on-chip shared memory
  - Read and write per-thread off-chip local memory (cached)
  - Read only per-grid off-chip constant memory (cached)
  - Read and write per-grid off-chip global memory (DRAM)



- Host code can
  - Transfer data to/from per-grid global and constant memories

Fewer Instructions for Access to Registers than Global Memory

- Arithmetic instructions have "built-in" register operands.
- Example: fadd r1, r2, r3
  - r2 and r3 are the register numbers that specify the location in the register file where the input operand values can be found.
  - r1 specifies the location for storing the floating-point addition result value.



- If an operand value is in the global memory, the processor needs to perform a memory load operation to make the operand value available to the arithmetic and logic unit (ALU) first.
- Example: assembly language

```
load r2, r4, offset
fadd r1, r2, r3
```

  - Add an offset value to the content of r4 to form an address for the operand value.
  - Access the global memory and place the operand value into register r2.

On-chip Shared Memory is Scratchpad Memory

- Latency and bandwidth
  - Accessed with much lower latency and much higher throughput than the global memory.
  - Accessed with longer latency and lower bandwidth than on-chip registers
    - Because the processor needs to perform a memory load operation when accessing data in the shared memory

- Data sharing
  - Variables in the shared memory are accessible by all threads in a block
  - Register data are private to a thread.

Thread

- Control unit
    - Maintains a program counter (PC), which contains the memory address of the next instruction to be executed.
    - Uses the PC to fetch an instruction into the instruction register (IR) in each "instruction cycle"

- Context-switching
    - Multiple threads can time-share a processor by taking turns to make progress.
    - The execution of a thread can be suspended and then correctly resumed later by saving and restoring the PC value and the contents of registers/memory.

- Single-Instruction, Multiple-Data design
    - Multiple processing units share a PC and IR, and allow multiple threads to make simultaneous progress.
    - All threads make simultaneous progress by executing the same instruction.

Scope vs. Lifetime

- Scope of a variable (a textual or spatial concept in the compiling period)
  - Serial programming: the range of statements in which the variable is visible
    - A variable is visible in a statement if it can be referenced or assigned in that statement.
  - Parallel programming: the range of threads that can access the variable
    - A single thread only
    - All threads of a block
    - All threads of a grid

- Lifetime of a variable (a temporal concept in run-time)
  - The portion of the program execution duration when the variable is available for use (has valid memory)
    - Within a kernel execution
    - Throughout the entire application

CUDA Variable Types

Variable declaration	Memory	Scope	Lifetime
Automatic scalar variables	Register	Thread	Kernel
Automatic array variables	Local	Thread	Kernel
__device__ __shared__ int SharedVar;	Shared	Block	Kernel
__device__ __constant__ int ConstVar;	Constant	Grid	Application
__device__ int GlobalVar;	Global	Grid	Application

CUDA Variable Types

- Automatic scalar variables
  - Scalar variables: variables that are not arrays or matrices
- Automatic array variables
  - Rarely used in kernel/device functions
- Shared variables
  - Reside within a kernel/device function
  - Hold global memory data heavily used in a kernel execution phase
- Constant variables
  - Declaration is outside any function body.
  - Provide input values to kernel functions
  - Stored in the global memory but cached for efficient access
  - The total size in an application is limited to 65,536 bytes.
- Global variables
  - The way to synchronize between threads from different blocks or to ensure data consistency across threads when accessing global memory is by terminating the current kernel execution.
  - Pass information from one kernel invocation to another kernel invocation

Question & Answer