# CS302

# Introduction to GPUs and CUDA

**Spring 2025**

**Babak Falsafi, Arkaprava Basu**

**parsa.epfl.ch/course-info/cs302**

Some of the slides are from Derek R Hower, Adwait Jog, Wen-Mei Hwu, Steve Lumetta, Babak Falsafi, Andreas Moshovos, and from the companion material of the book “Programming Massively Parallel Processors”

# Where are We?

| M | T | W | T | F |
|---|---|---|---|---|
| 17-Feb | 18-Feb | 19-Feb | 20-Feb | 21-Feb |
| 24-Feb | 25-Feb | 26-Feb | 27-Feb | 28-Feb |
| 3-Mar | 4-Mar | 5-Mar | 6-Mar | 7-Mar |
| 10-Mar | 11-Mar | 12-Mar | 13-Mar | 14-Mar |
| 17-Mar | 18-Mar | 19-Mar | 20-Mar | 21-Mar |
| 24-Mar | 25-Mar | 26-Mar | 27-Mar | 28-Mar |
| 31-Mar | 1-Apr | 2-Apr | 3-Apr | 4-Apr |
| 7-Apr | 8-Apr | 9-Apr | 10-Apr | 11-Apr |
| 14-Apr | 15-Apr | 16-Apr | 17-Apr | 18-Apr |
| 21-Apr | 22-Apr | 23-Apr | 24-Apr | 25-Apr |
| 28-Apr | 29-Apr | 30-Apr | 1-May | 2-May |
| | 6-May | 7-May | 8-May | 9-May |
| 12-May | 13-May | 14-May | 15-May | 16-May |
| 19-May | 20-May | 21-May | 22-May | 23-May |
| 26-May | 27-May | 28-May | 29-May | 30-May |

- This week:
  - GPU hardware
  - CUDA programming basics
- Thursday exercise session:
  - GPU demo programming
  - Using the Izar cluster
- Next week:
  - Memory optimizations on GPUs

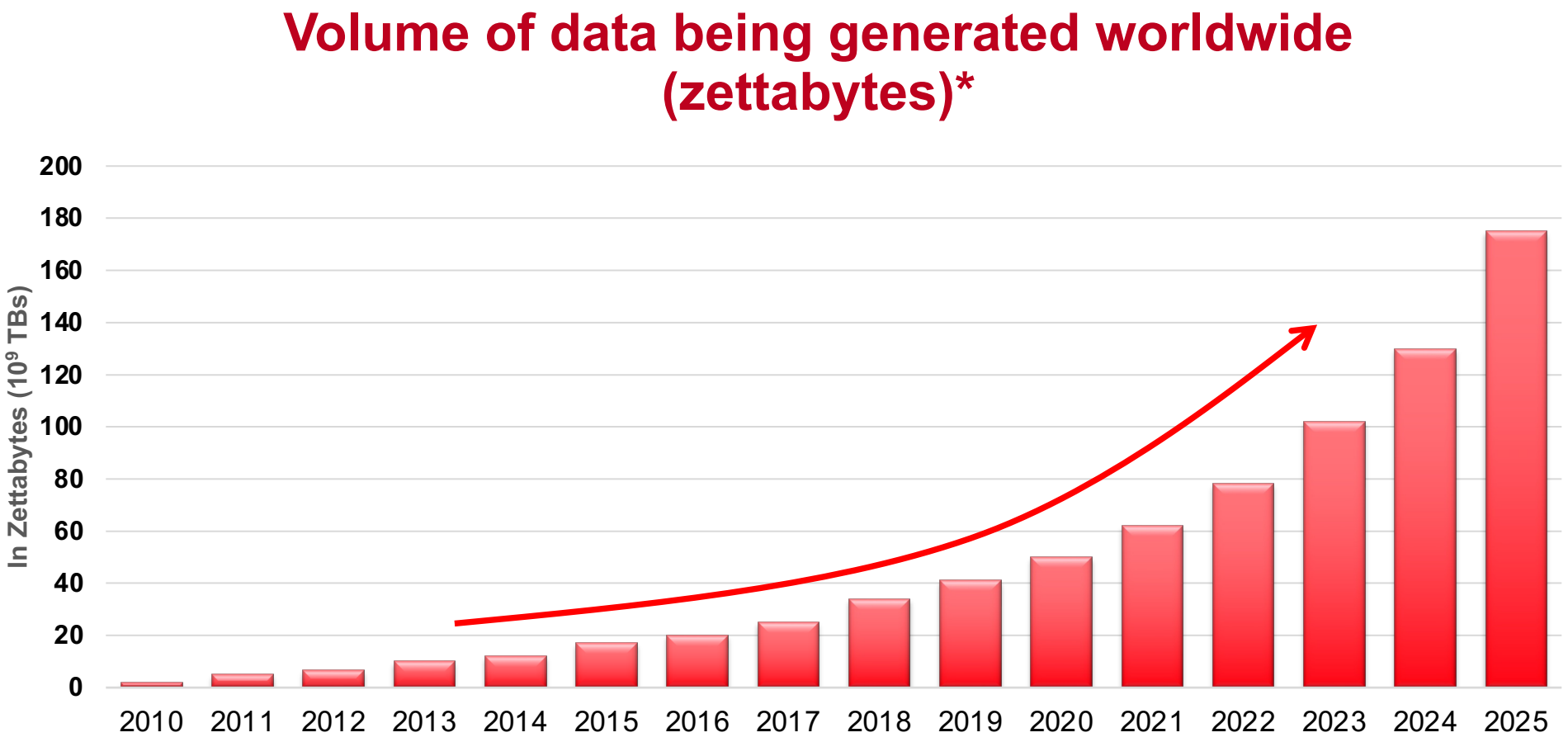
# Heads Up

- Assignment 3 released!
  - Implement the RMM algorithm on GPUs
  - Optimise the memory traffic of the algorithm
- Deadline to submit A3: Sunday June 1st, 2025 11:59 pm
- HW7 also released!
  - Deadline to submit: Sunday May 11th, 2025 11:59 pm

Even if you are not a video gamer!

# WHY CARE ABOUT A GPU?

# Need Big (Parallel) Compute to Draw Knowledge From Big Data




*https://www.statista.com/statistics/871513/worldwide-data-created/

# GPU: Accelerator For Parallel Computing

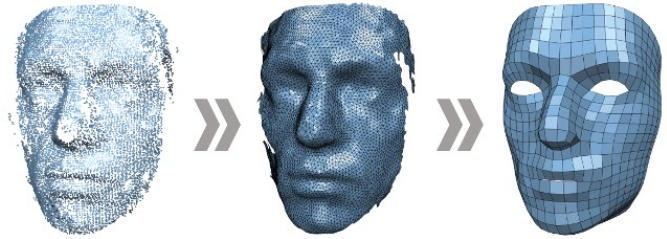
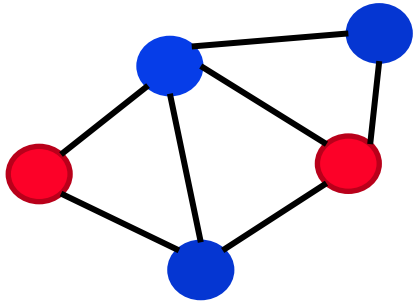
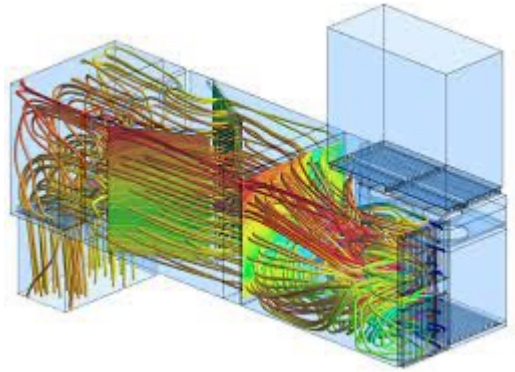


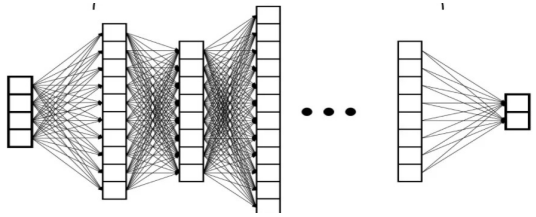
Image processing



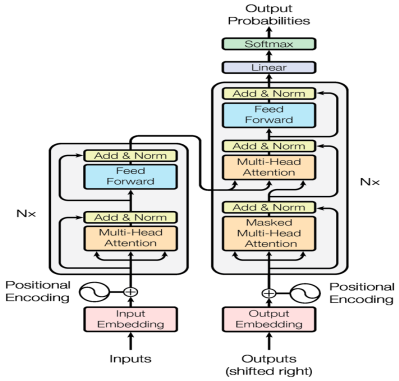
Graph processing

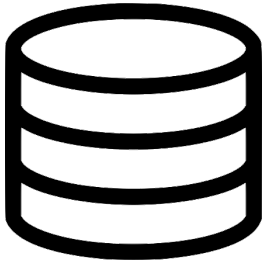


Scientific computing



Machine Learning

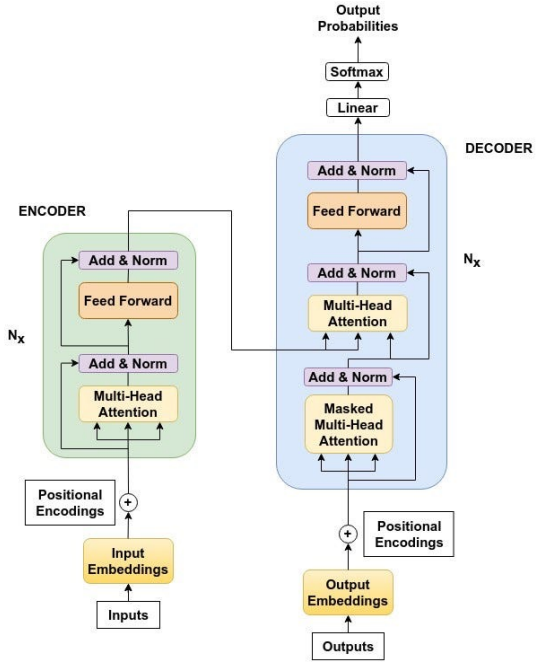


Data clustering, warehousing

A large swath of today's software relies on GPUs for their primary computing need

# GPUs Are Powering LLMs on Supercomputers




Elon Musk plans to scale the xAI supercomputer to a million GPUs — currently at over 100,000 H100 GPUs and counting

News By Anton Shilov published December 5, 2024

PCMag UK > News > Components > Graphics Cards

**Zuckerberg's Meta Is Spending Billions to Buy 350,000 Nvidia H100 GPUs**

In total, Meta will have the compute power equivalent to 600,000 Nvidia H100 GPUs to help it develop next-generation AI, says CEO Mark Zuckerberg.

By Michael Kan Jan 18, 2024

Nine out of the ten most powerful supercomputers in Top500 list rely on GPUs


The fastest supercomputer (El Capitan) today has 43,808 GPUs

# Review: Types of Parallelism

- Instruction Level Parallelism (ILP)
  - Example: Out-of-order processor
- Memory Level Parallelism  (MLP)
  - Example: Non-blocking caches
- Thread Level Parallelism (TLP)
  - Example: Simultaneous Multi-threading

- Data Level Parallelism (DLP)
  - Example: Single Instruction Multiple Data (SIMD), Vector

# Why GPUs for Data-Parallel Computing?

# Why GPUs for Data-Parallel Computing?

**Identical, parallel operation on pixels**

# Why GPUs for Data-Parallel Computing?

**Serendipity: ML/AI needs a lot of matrix and vector arithmetic → GPU boom**

| | | | | |
|---|---|---|---|---|
| 0 | 71 | 3 | 15 | 88 |
| 5 | 0 | 8 | 24 | 0 |
| 1 | 0 | 21 | 0 | 6 |
| 0 | 23 | 10 | 16 | 0 |
| 66 | 0 | 23 | 0 | 4 |
| 35 | 2 | 0 | 30 | 0 |
| 44 | 0 | 27 | 19 | 0 |
| 0 | 0 | 22 | 39 | 81 |

| | | | | |
|---|---|---|---|---|
| 0 | 33 | 72 | 4 | 13 |
| 9 | 0 | 66 | 25 | 8 |
| 36 | 22 | 0 | 13 | 15 |
| 0 | 23 | 17 | 5 | 0 |
| 21 | 0 | 1 | 0 | 12 |
| 6 | 97 | 0 | 45 | 23 |
| 4 | 0 | 65 | 0 | 43 |
| 0 | 55 | 42 | 9 | 0 |

**Identical, parallel operation on data items**

# Arrival of “General Purpose” GPUs (GPGPUs), a.k.a, CUDA

- Before 2007, GPUs could only be programmed using Graphics APIs
  - For computation one required to reformulate it as a graphics function on pixels
  - For example, use OpenGL, Direct3D for computation → Hard to program

- In 2007, NVIDIA introduced CUDA
  - A dialect of C/C++
  - General purpose programming/computation for GPUs
  - Extensions to the GPU architecture

**Hardware is useful only if it is (easily) programmable**

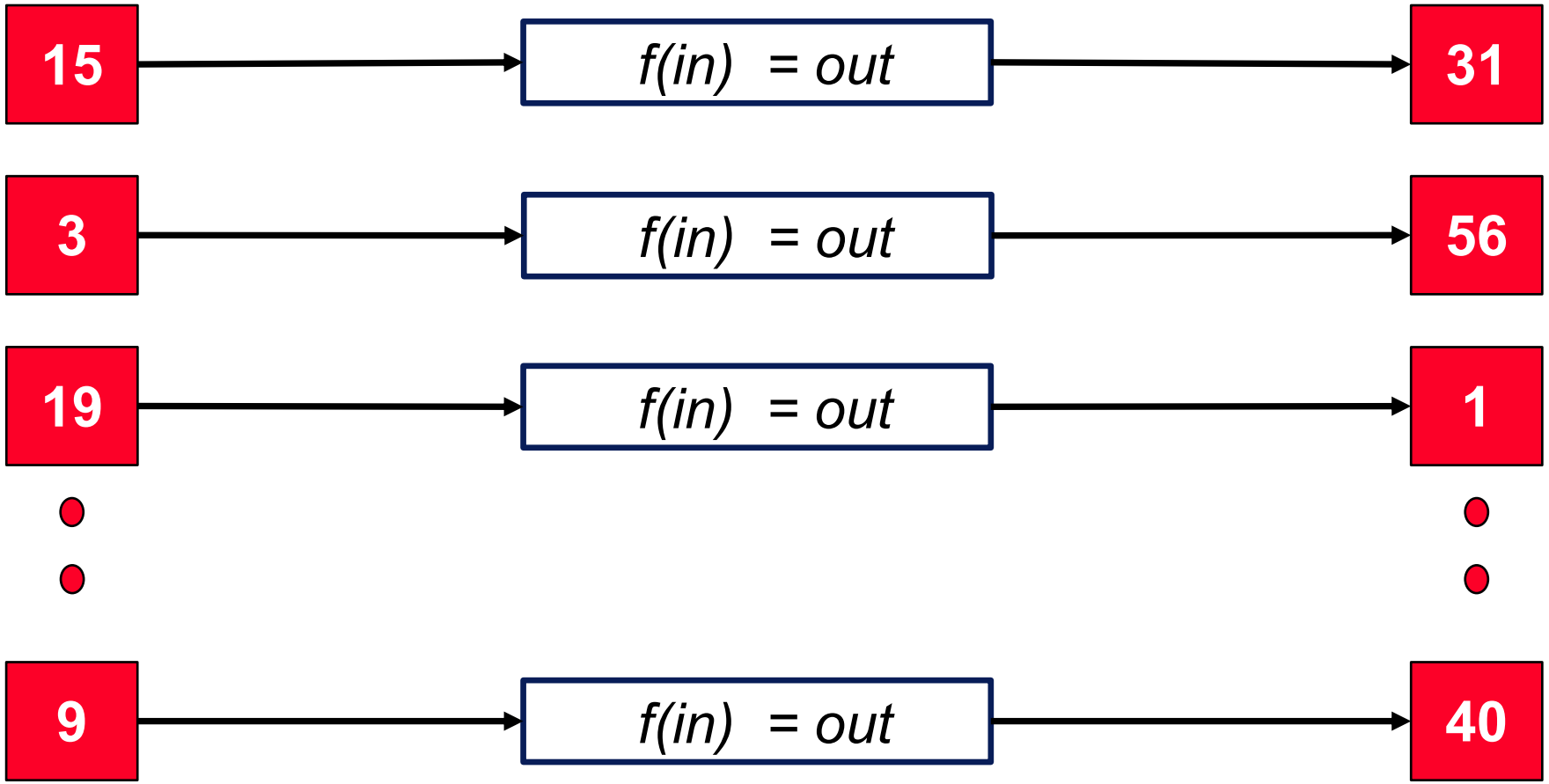
# Closer Look: GPU's Execution Model in Perspective

| | | | | |
|---|---|---|---|---|
| 0 | 71 | 3 | 15 | 88 |
| 5 | 0 | 8 | 24 | 0 |
| 1 | 0 | 21 | 0 | 6 |
| 0 | 23 | 10 | 16 | 0 |
| 66 | 0 | 23 | 0 | 4 |
| 35 | 2 | 0 | 30 | 0 |
| 44 | 0 | 27 | 19 | 0 |
| 0 | 0 | 22 | 39 | 81 |

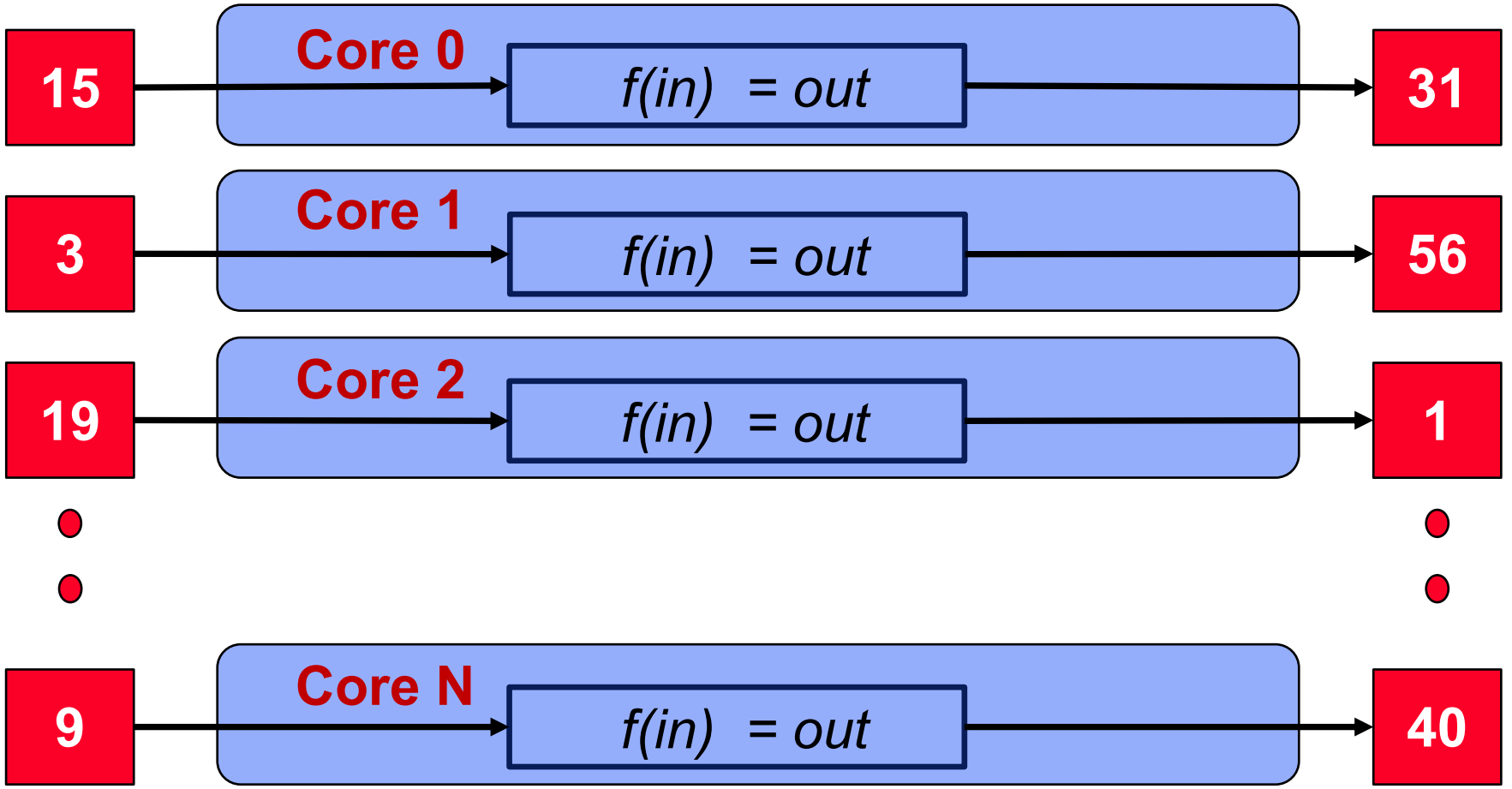
| | | | | |
|---|---|---|---|---|
| 0 | 33 | 72 | 4 | 13 |
| 9 | 0 | 66 | 25 | 8 |
| 36 | 22 | 0 | 13 | 15 |
| 0 | 23 | 17 | 5 | 0 |
| 21 | 0 | 1 | 0 | 12 |
| 6 | 97 | 0 | 45 | 23 |
| 4 | 0 | 65 | 0 | 43 |
| 0 | 55 | 42 | 9 | 0 |

**Identical, parallel operation on data items**

# Identical Computation on Different Data Items

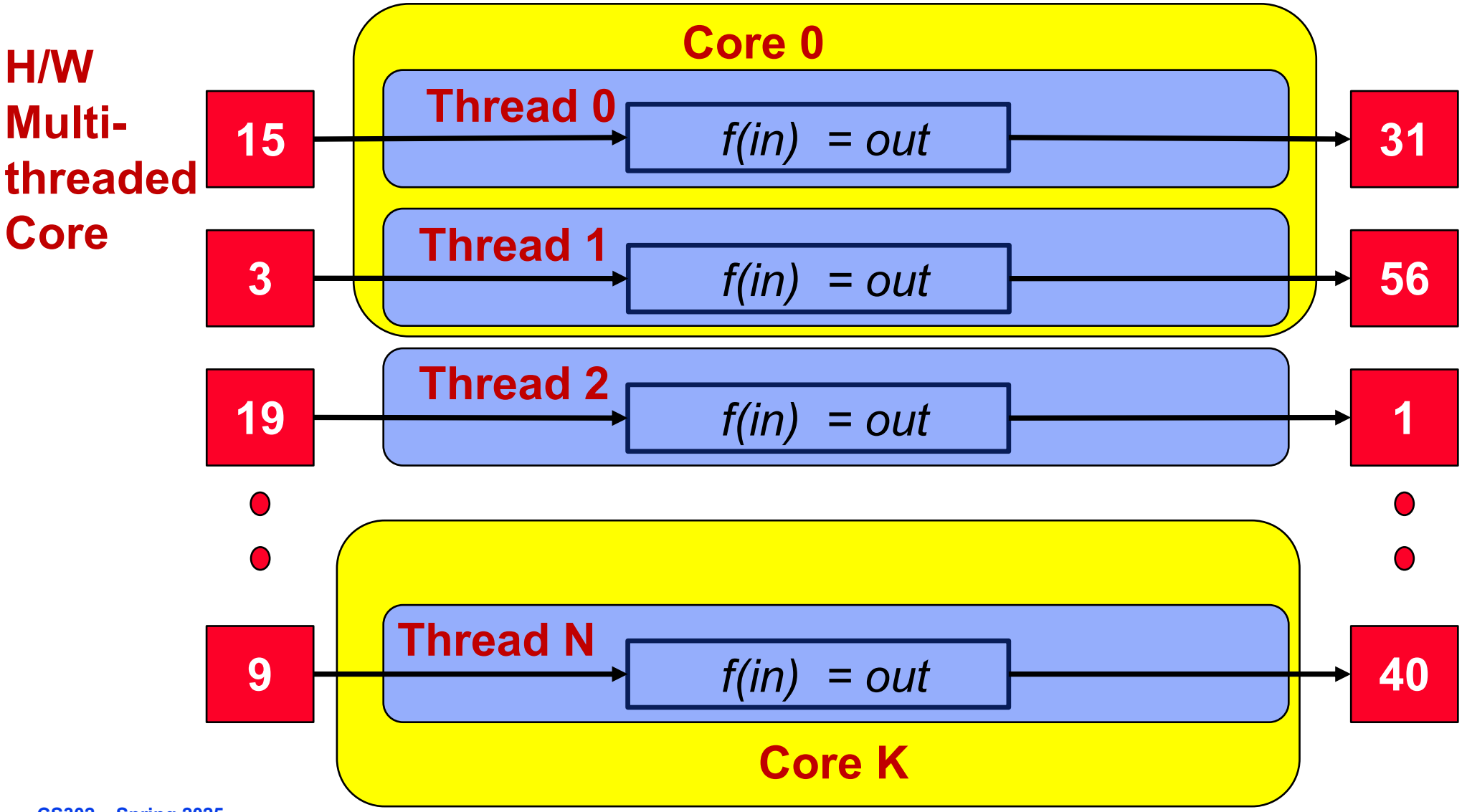


# Option (1): The Multi-Threading/Multi-Core/Multi-Processor Way



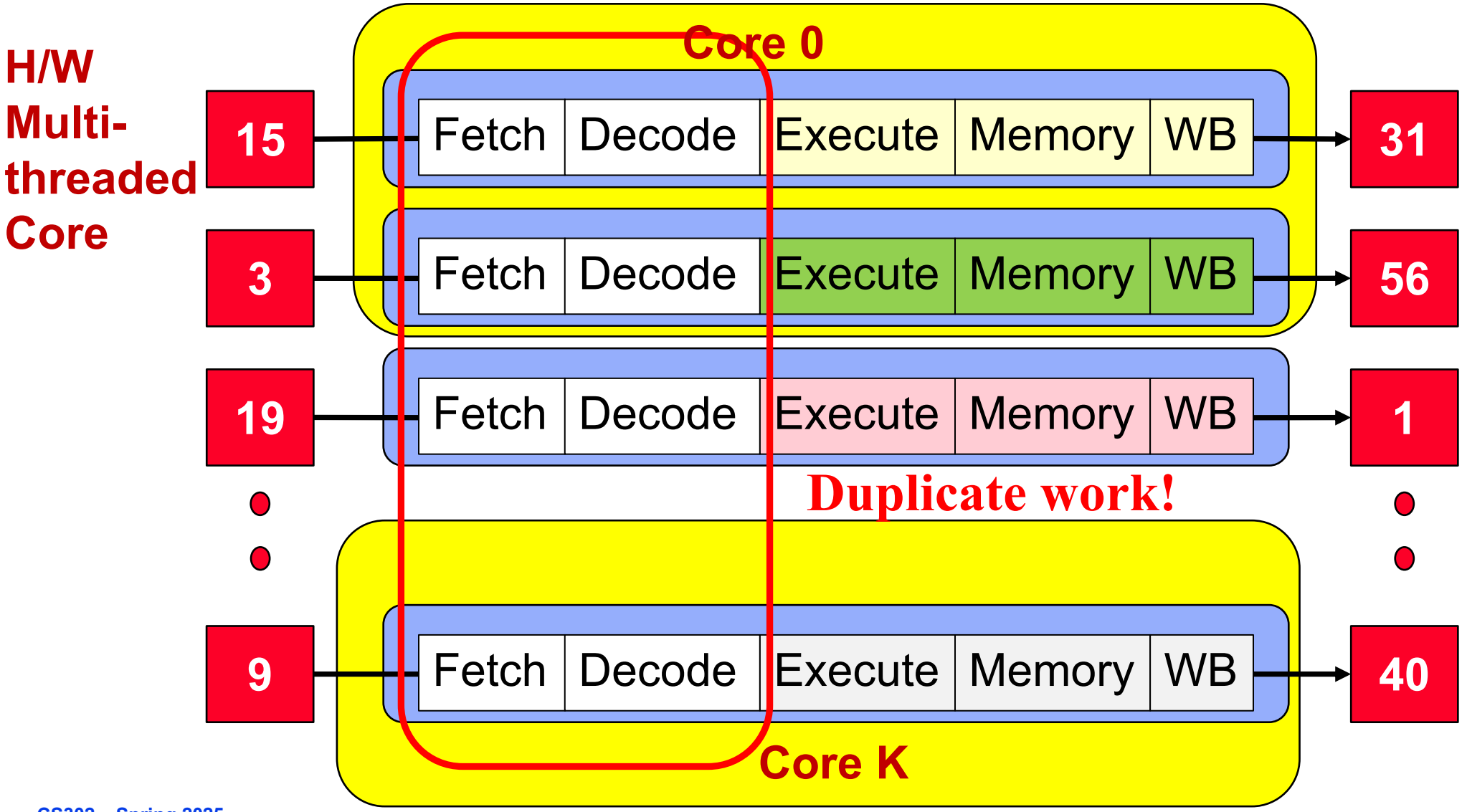

Identical, independent work on each CPU core → Unable to leverage "identical" instructions

# Option (1): The Multi-Threading/Multi-Core/Multi-Processor Way



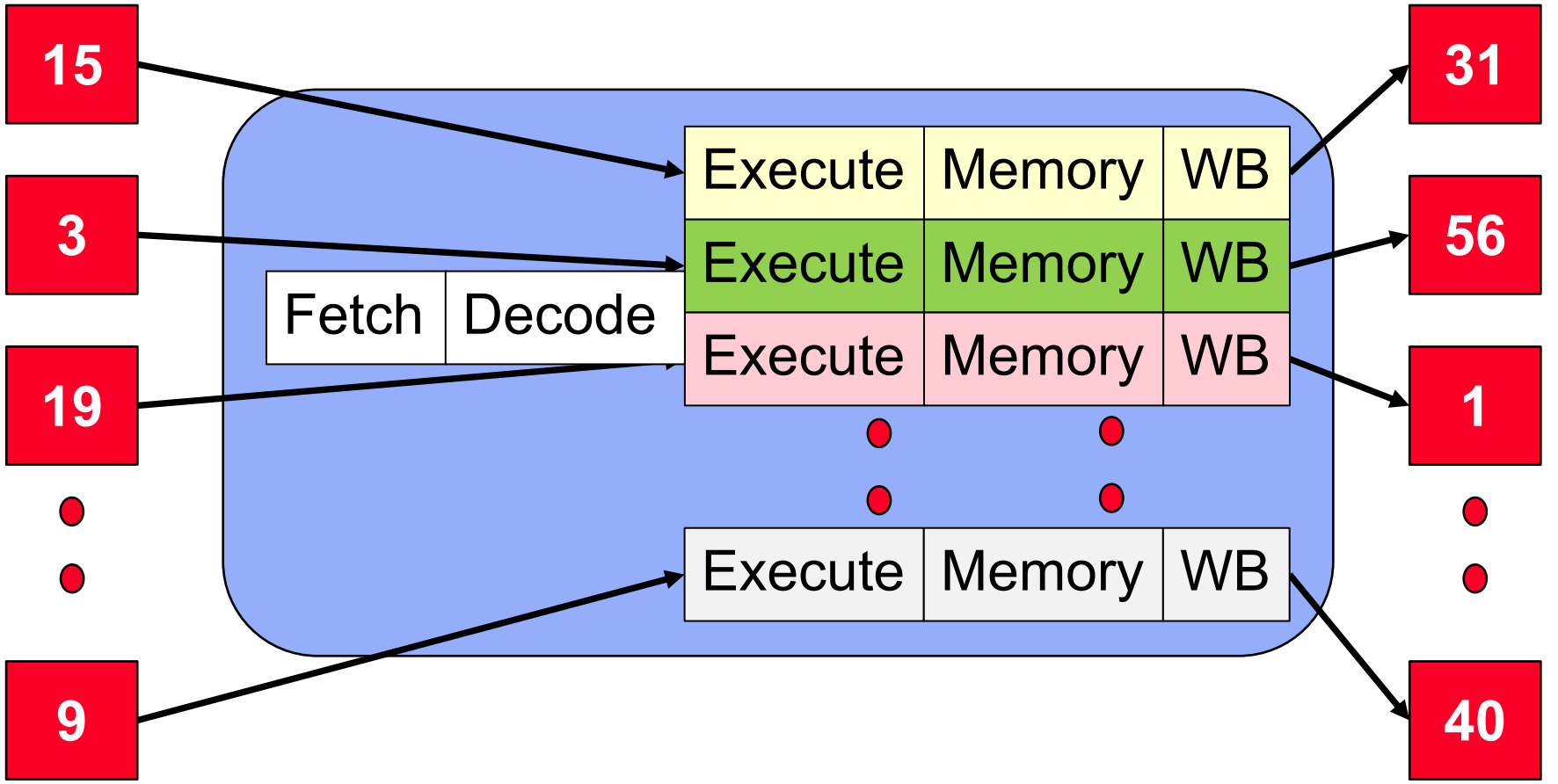

Identical, independent work on each CPU core → Unable to leverage "identical" instructions

# Option (1): The Multi-Threading/Multi-Core/Multi-Processor Way



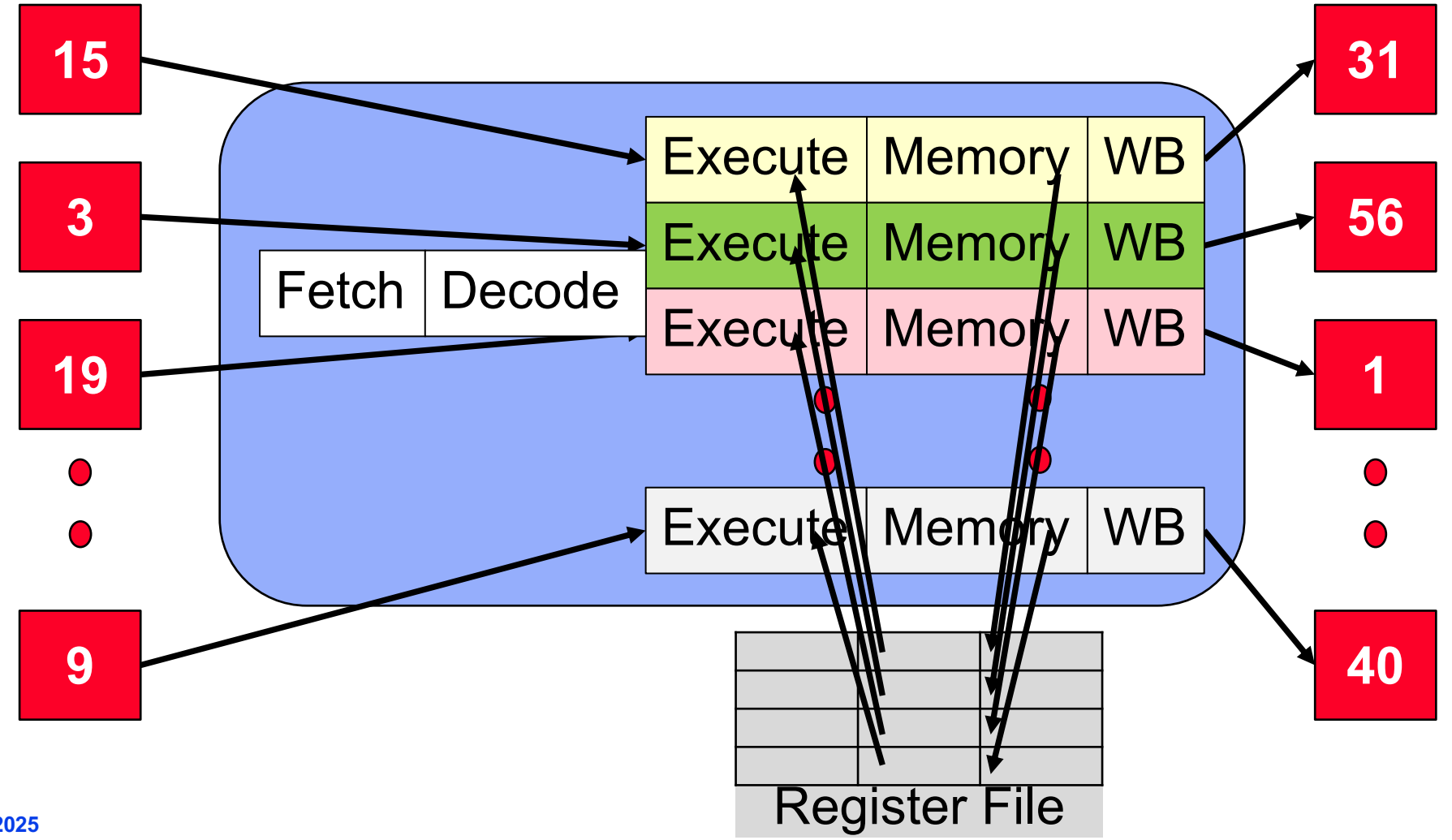

Identical, independent work on each CPU core → Unable to leverage "identical" instructions → Inefficient

# Option (2): Single Instruction Multiple Data (SIMD)




Eliminates redundant fetch and decode → Efficient

# Option (2): Single Instruction Multiple Data (SIMD)

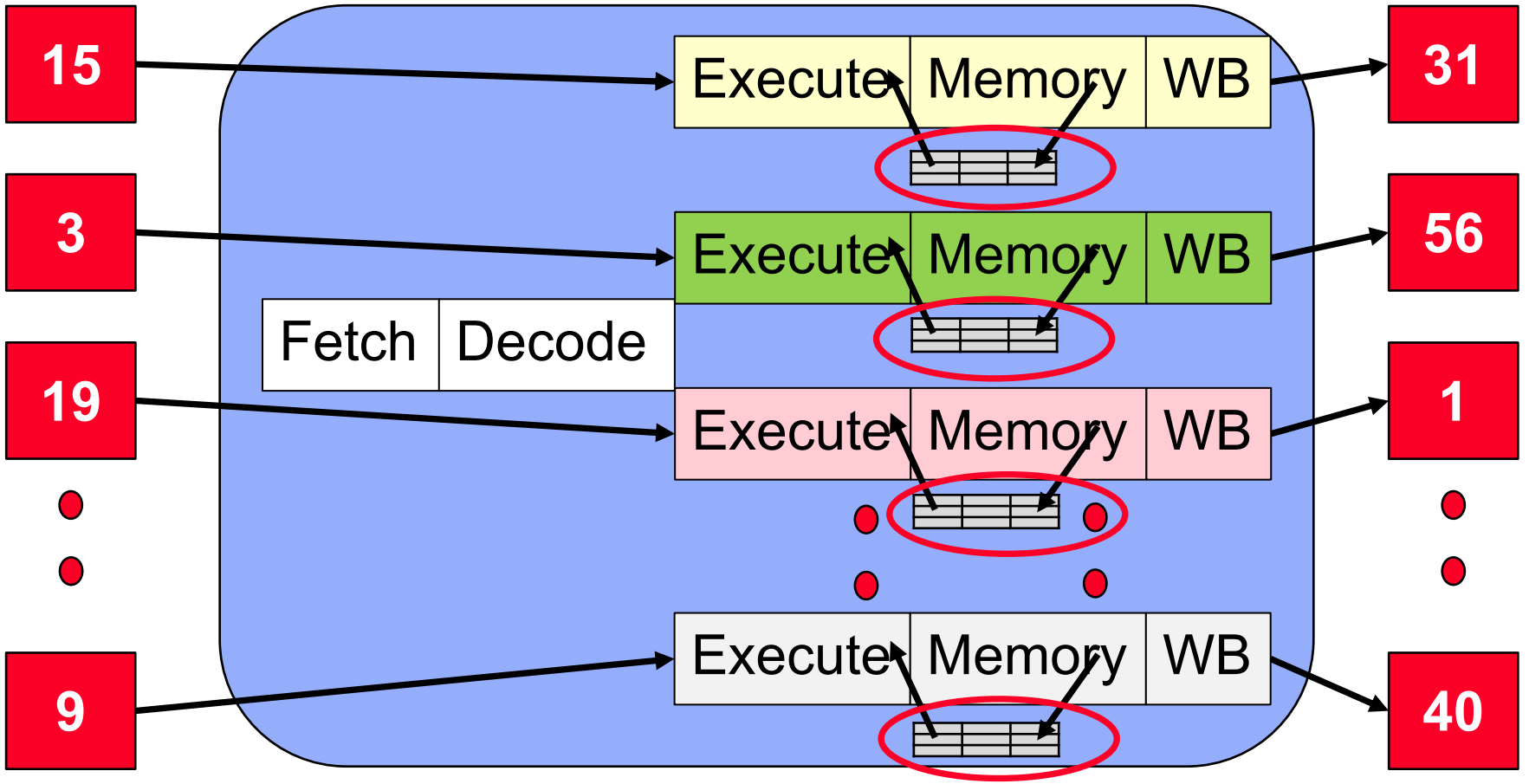



Eliminates redundant fetch and decode → Efficient

Single thread, parallel ops → Limited programm-ability

# Single Instruction Multiple Thread (SIMT) Execution Model

SIMT ≈ SIMD with H/W Multi-threading



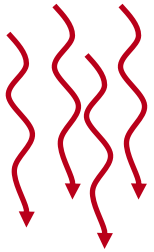

Independent thread context

Multiple threads execute in lockstep (almost always) → Efficient

Separate context → Better Programmability

# Summary of Execution Model

MIMD/SPMD



Good for TLP
but not for DLP

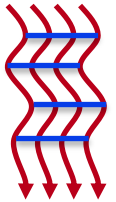
Easiest Programming

SIMD/Vector

Good for DLP
Limited Programmability

Harder to scale up

SIMT

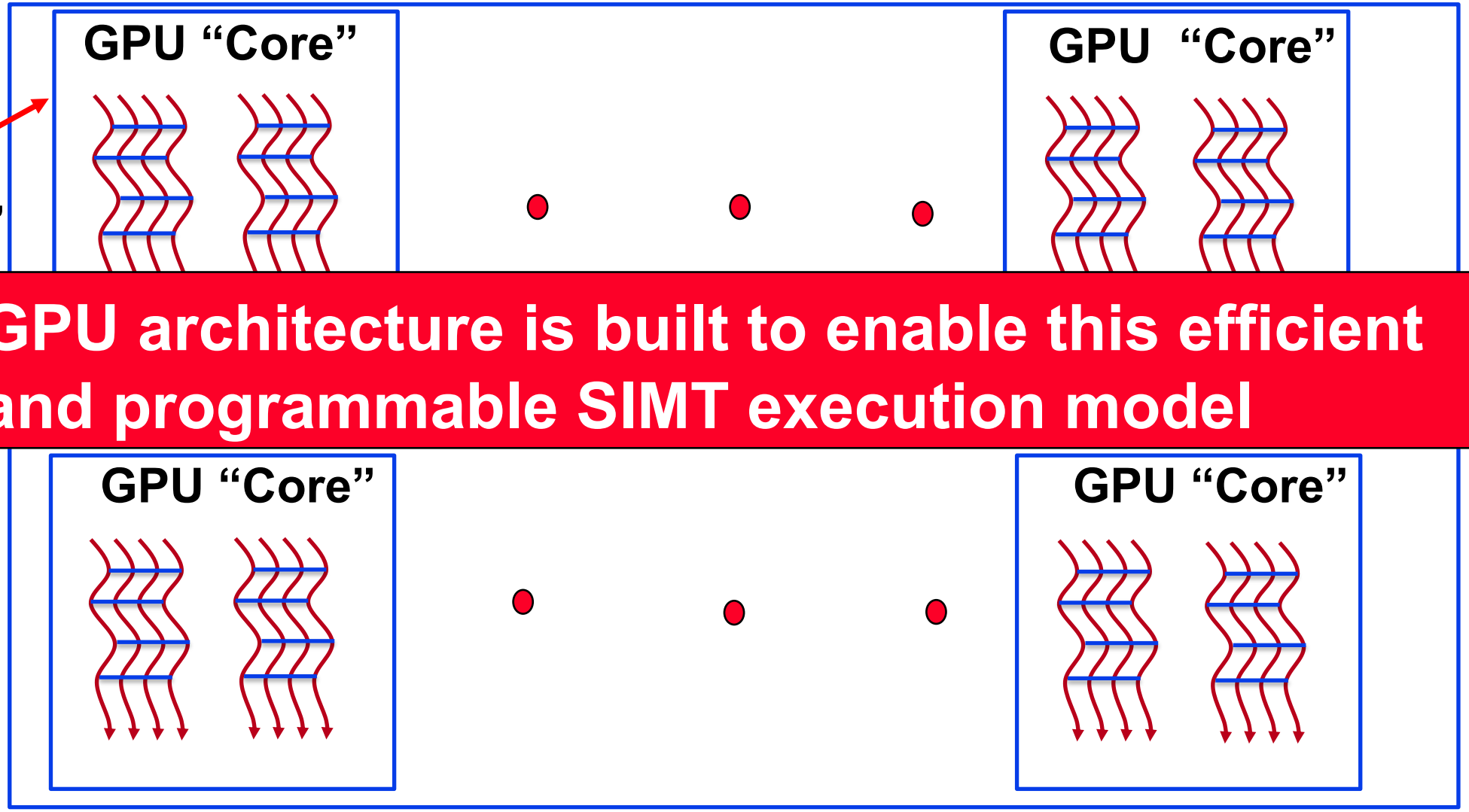


Good for DLP
Better Programmability

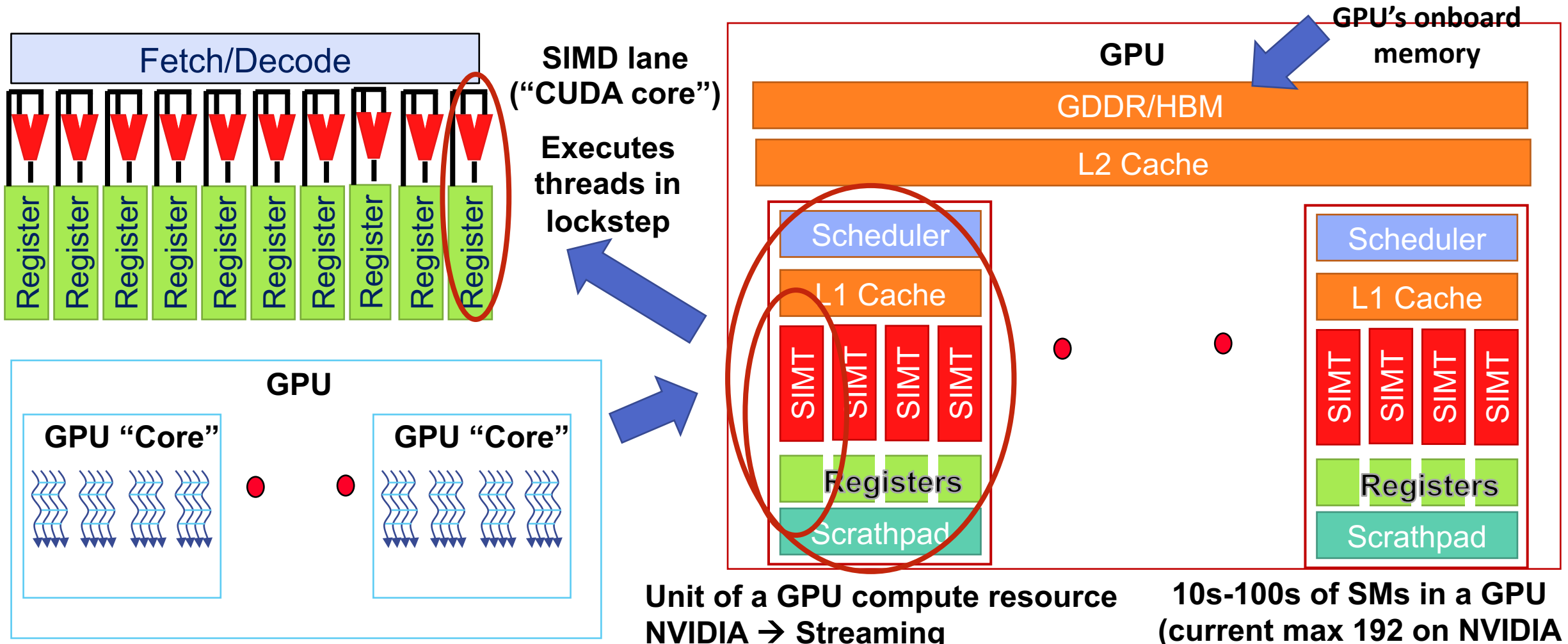
Easier to scale up

# A Conceptual View of a GPU: SIMT on Steroids!

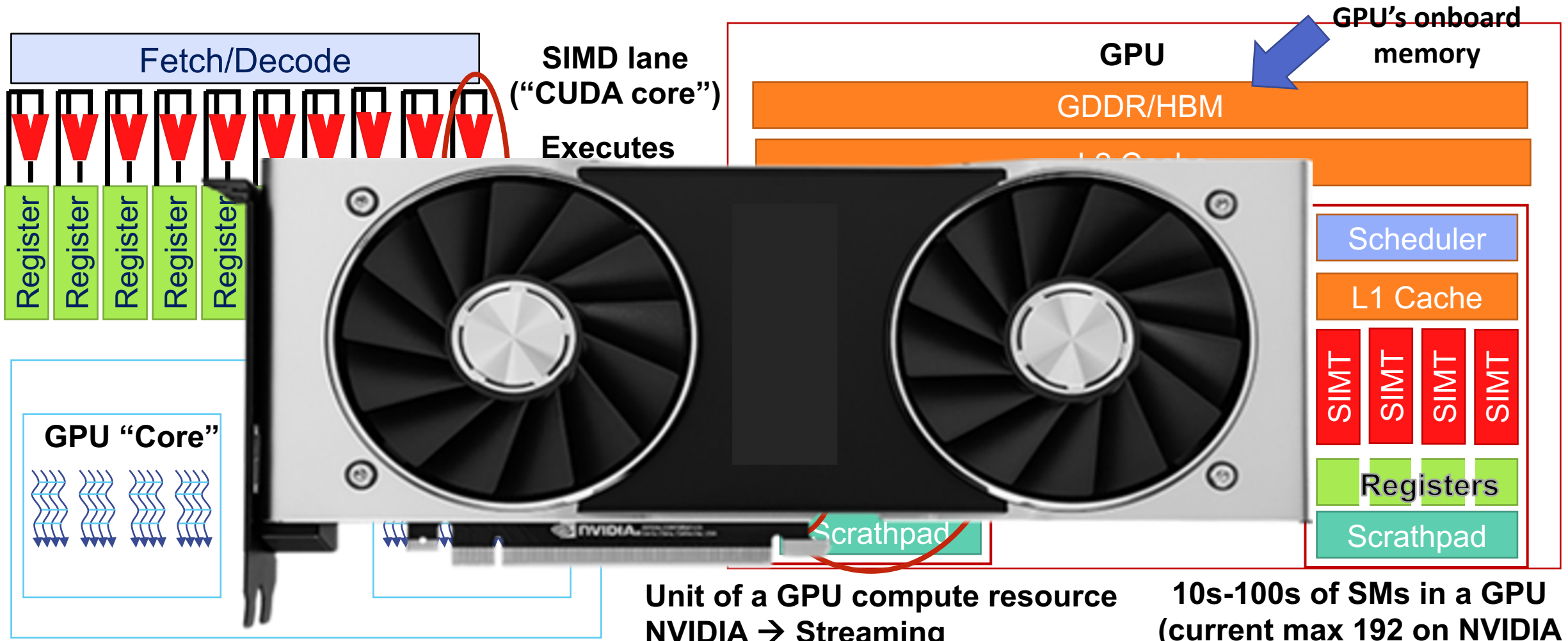
Conceptually, it's like a "core" of a multi-core CPU



# A More Realistic Picture of GPU Hardware



# A More Realistic Picture of GPU Hardware

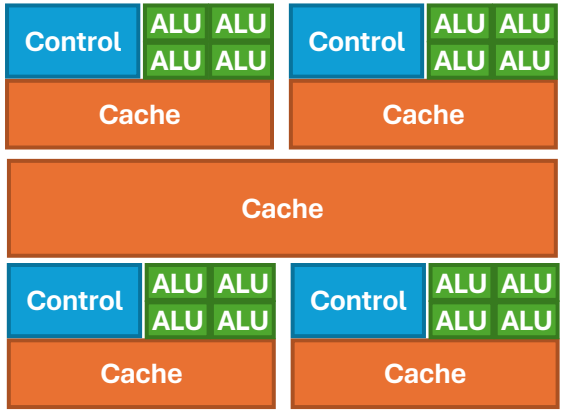



A typical SM can have 4 SIMT units with 128 total lanes

Unit of a GPU compute resource
NVIDIA → Streaming Multiprocessor (SM)
AMD → Compute Unit (CU)

10s-100s of SMs in a GPU (current max 192 on NVIDIA GPUs)

# CPU vs. GPU : A Bird's Eye View




**CPU: Latency-Oriented Design**

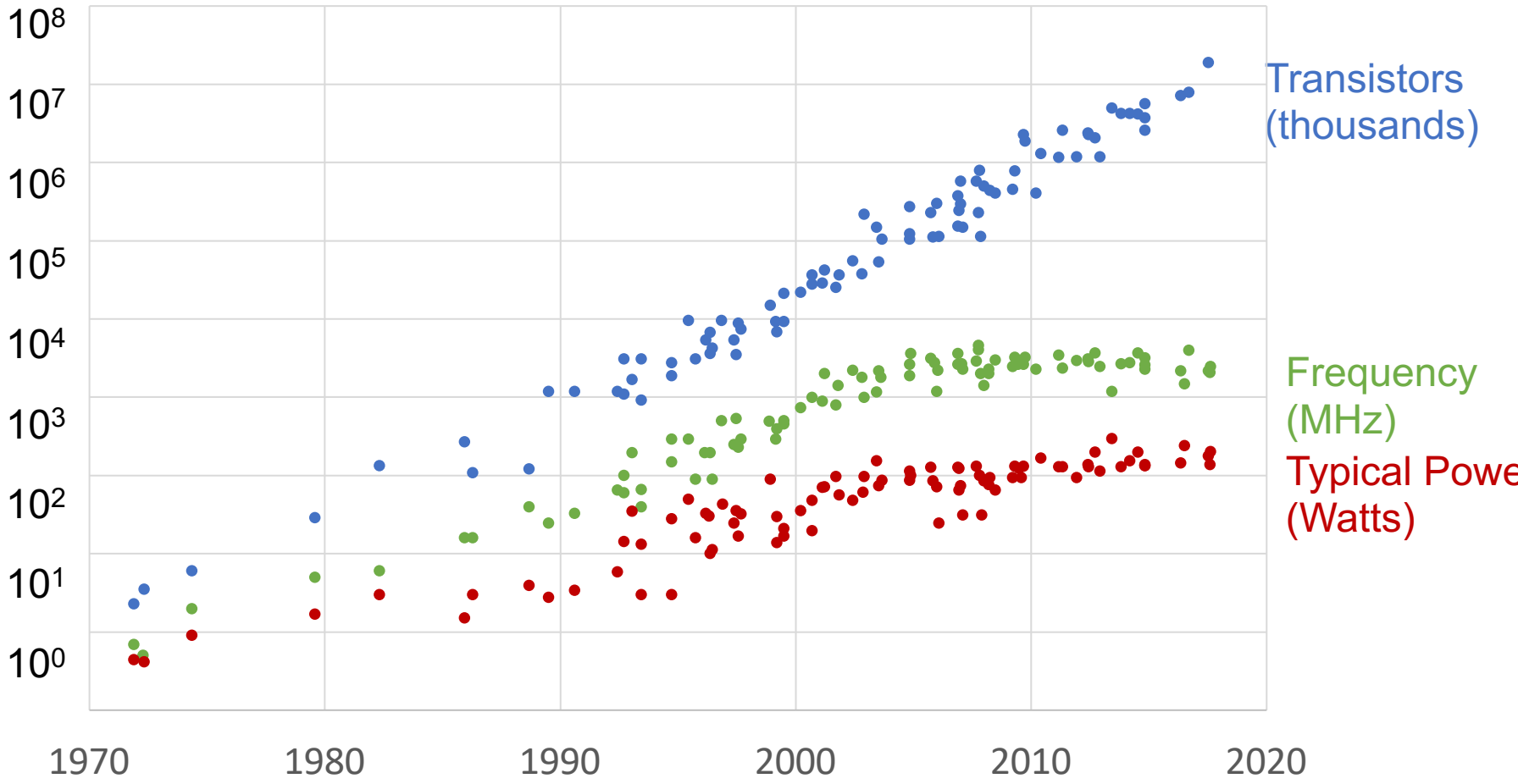
- A few, large ALUs
- Large caches
- Branch prediction, speculation, OoO
- High clock frequency
- Limited multithreading

**GPU: Throughput-Oriented Design**

- Sea of small ALUs
- Small, shallow cache hierarchy
- Simple control logic, in-order
- Massive multi-threading
- High memory bandwidth

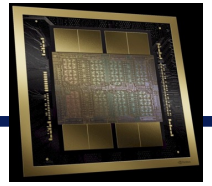
# Transistor Technology Trends Encourage GPU Architecture

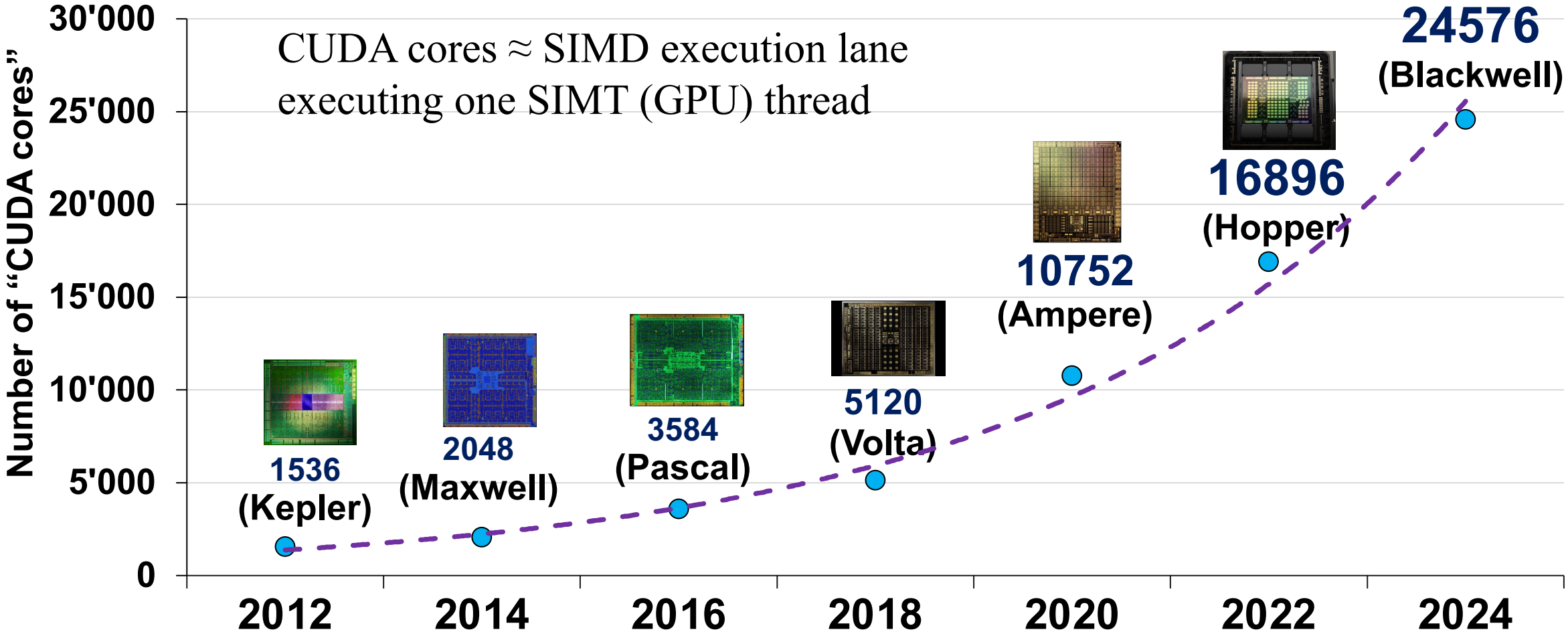
- Chip density increases
- Power wall limits frequency scaling
- Scaling frequency harder



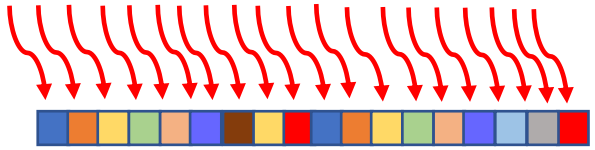

- GPUs turn transistor density into massive parallel compute

# Scale of Parallelism in Modern GPUs

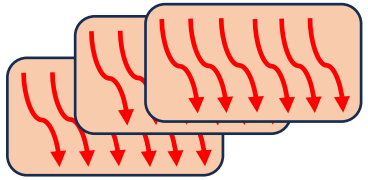




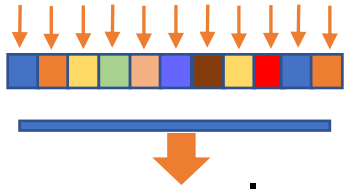
# Key GPU Characteristics at a Glance



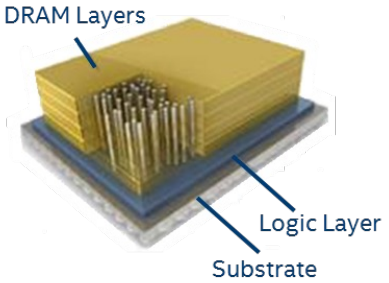
Single-instruction multiple thread (SIMT)



Many thread context → H/W context switch

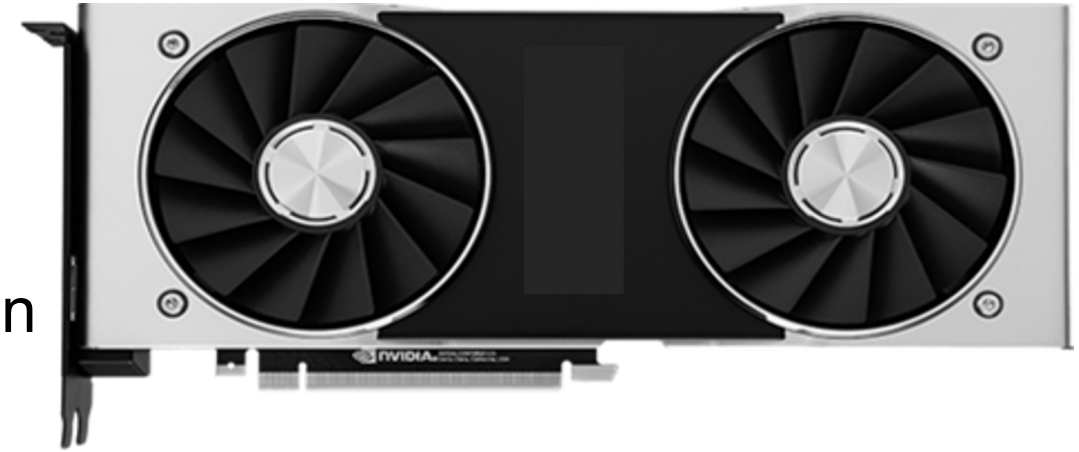


Memory coalescing




High bandwidth memory

The key enabler of efficient data-parallel execution



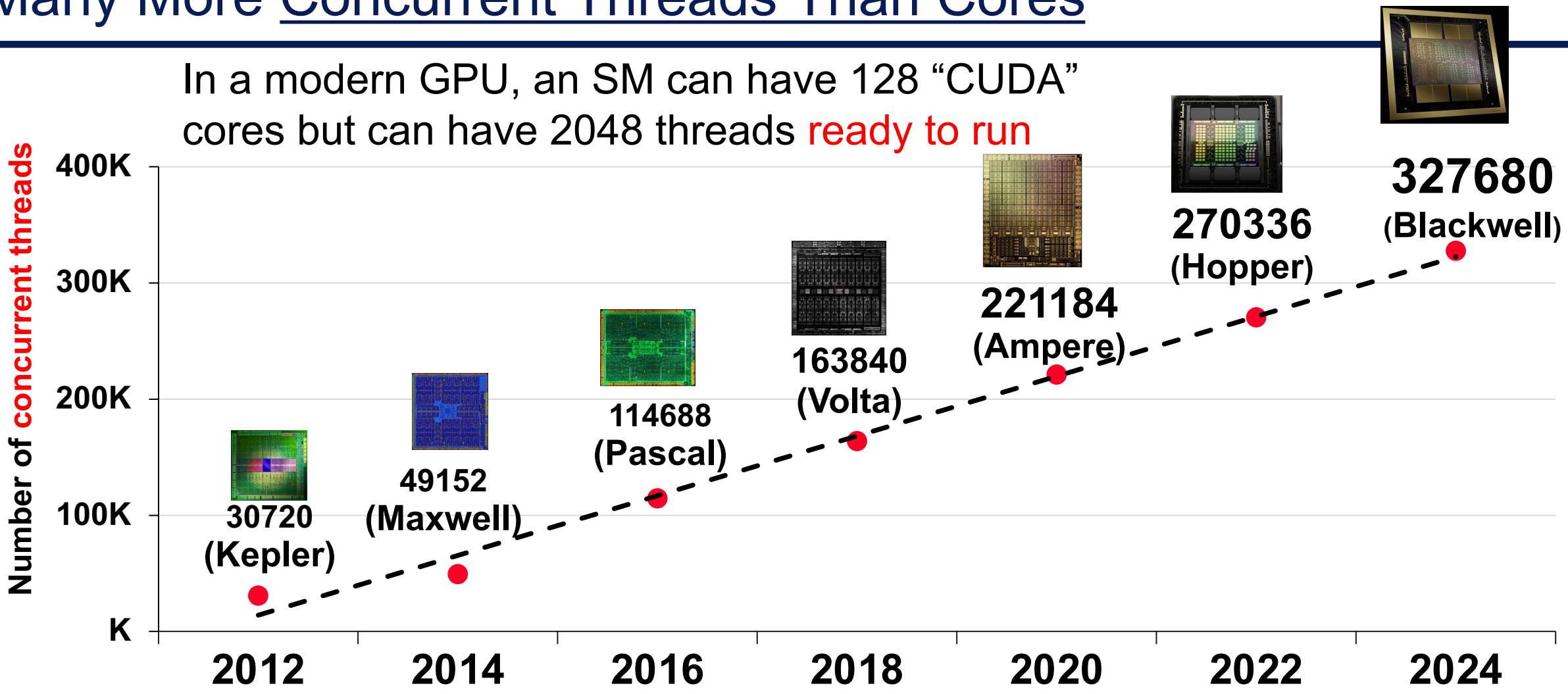
Feeding the large parallel compute with high bandwidth memory systems

(Next classes/weeks)

Effective latency hiding for high throughput

# Many More Concurrent Threads Than Cores

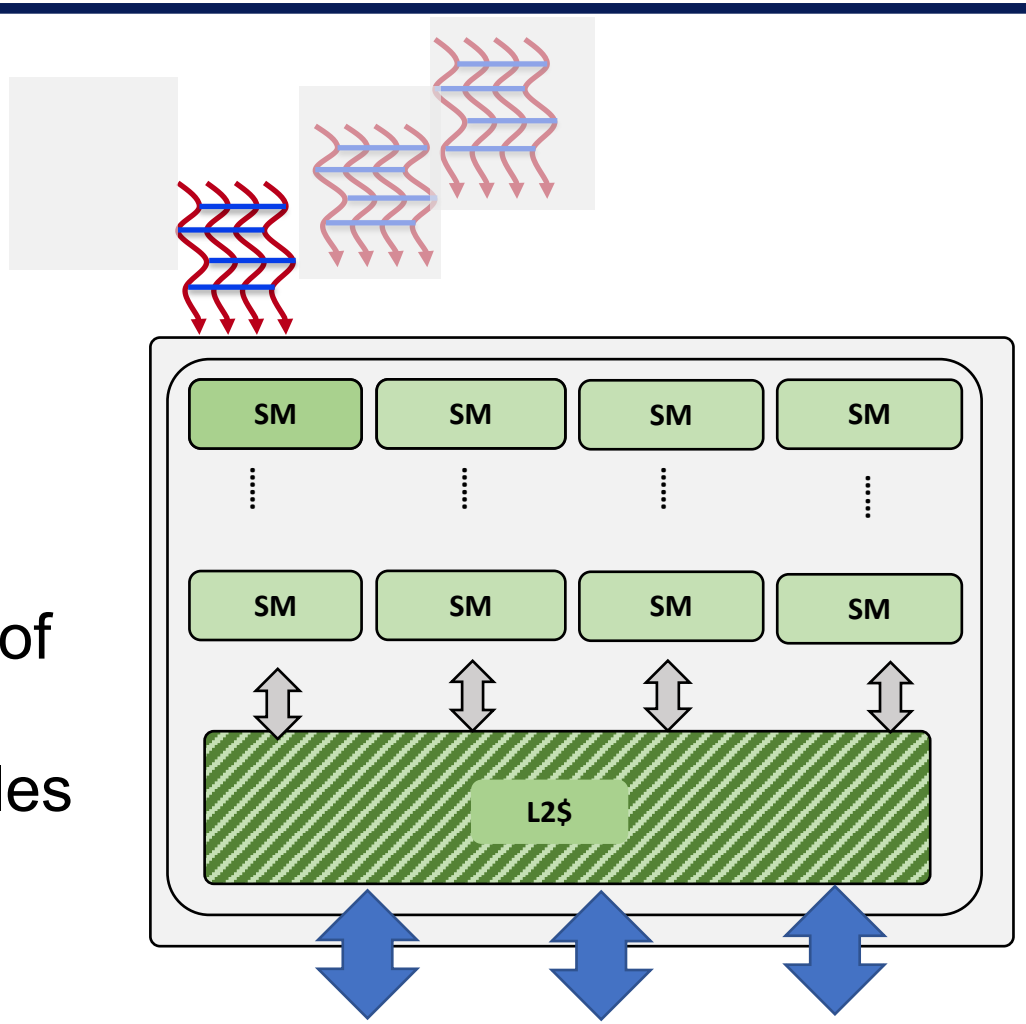
In a modern GPU, an SM can have 128 “CUDA” cores but can have 2048 threads ready to run




Key for hiding latency – keep many threads ready to run

# Hardware Context Switch for Hiding Latency

- Recall on the CPU:
  - OS does context switch of a thread when blocked/waiting
  - S/w context switch is slow (5-10 usec)
- On the GPU:
  - Hardware keeps many groups (SIMT) of threads ready to run
  - Upon long latency events, h/w schedules another group (SIMT) of thread
  - Many hardware registers
  - Scheduling policy baked into h/w



# Example: Commercial CPU vs. GPU

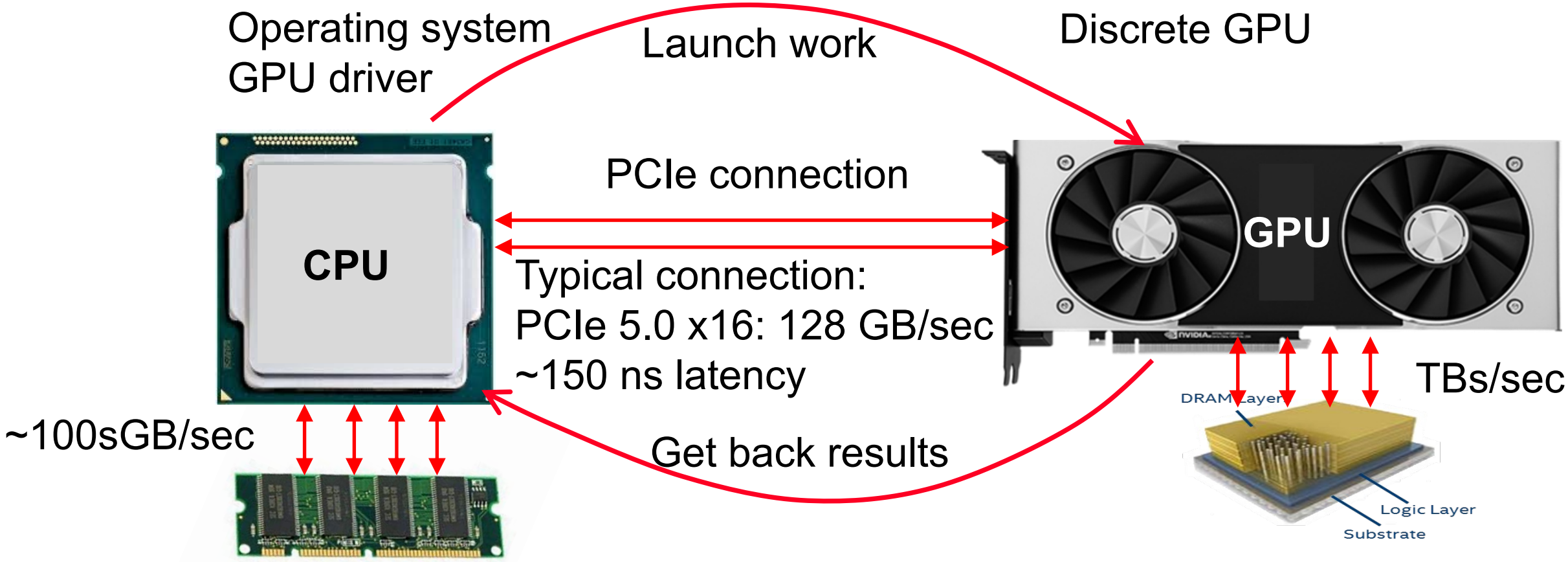
## AMD Epyc 5th gen (Turin)

- 192 cores/356 threads
- ~150-200 registers/core
- Cache hierarchy:
  - 48KB (L1) /1MB (L2) per core
  - 384MB LLC
- Memory bandwidth:
  - 576 GB/sec
  - DDR5
- TDP 500 Watts

## NVIDIA Blackwell

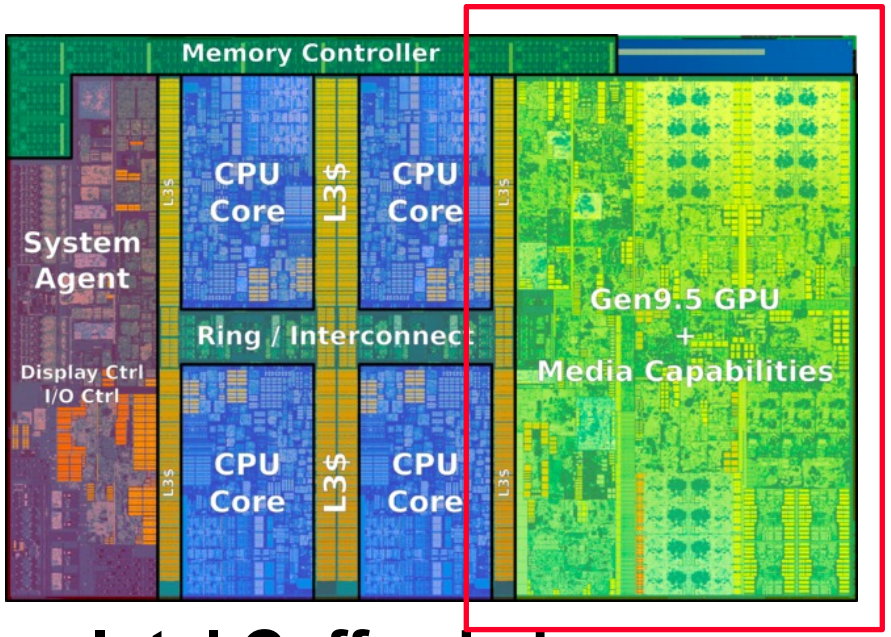
- 24,576 CUDA cores/ 192 SMs
- 64K registers per SM (256KB)
- Cache hierarchy:
  - 128KB (L1) per SM
  - 128MB L2
- Memory bandwidth
  - 8TB/sec
  - HBM3
- TDP 1000 Watts

# GPU Is a Co-Processor: Needs a Companion CPU




S/W running on CPU controls what runs on GPU, allocates GPU memory etc

# Tighter Integration: The Integrated GPU (iGPU)




**Intel Coffee Lake**

We will focus on discrete GPUs and not integrated GPUs

- GPU Integrated on-chip (SoC) with CPU
  - For example, Intel core i7
- Same architecture as discrete GPU
- But much smaller/less resources
- No onboard high-bandwidth memory, unlike discrete GPU
- Typically used for driving graphics in desktops/laptops

# PROGRAMMING THE GPU

# GPU Programming Languages

- **CUDA** – **C**ompute **U**nified **D**evice **A**rchitecture
  - Developed by Nvidia -- proprietary
  - First serious GPGPU language/environment
- **OpenCL** – **Open C**omputing **L**anguage
  - From makers of OpenGL
  - Wide industry support: AMD, Apple, Qualcomm, Nvidia, etc.
- **C++ AMP** – **C++ A**ccelerated **M**assive **P**arallelism
  - Microsoft
  - Much higher abstraction that CUDA/OpenCL
- **OpenACC** – **Open Acc**elerator
  - Like OpenMP for GPUs (semi-auto-parallelize serial code)
  - Much higher abstraction than CUDA/OpenCL

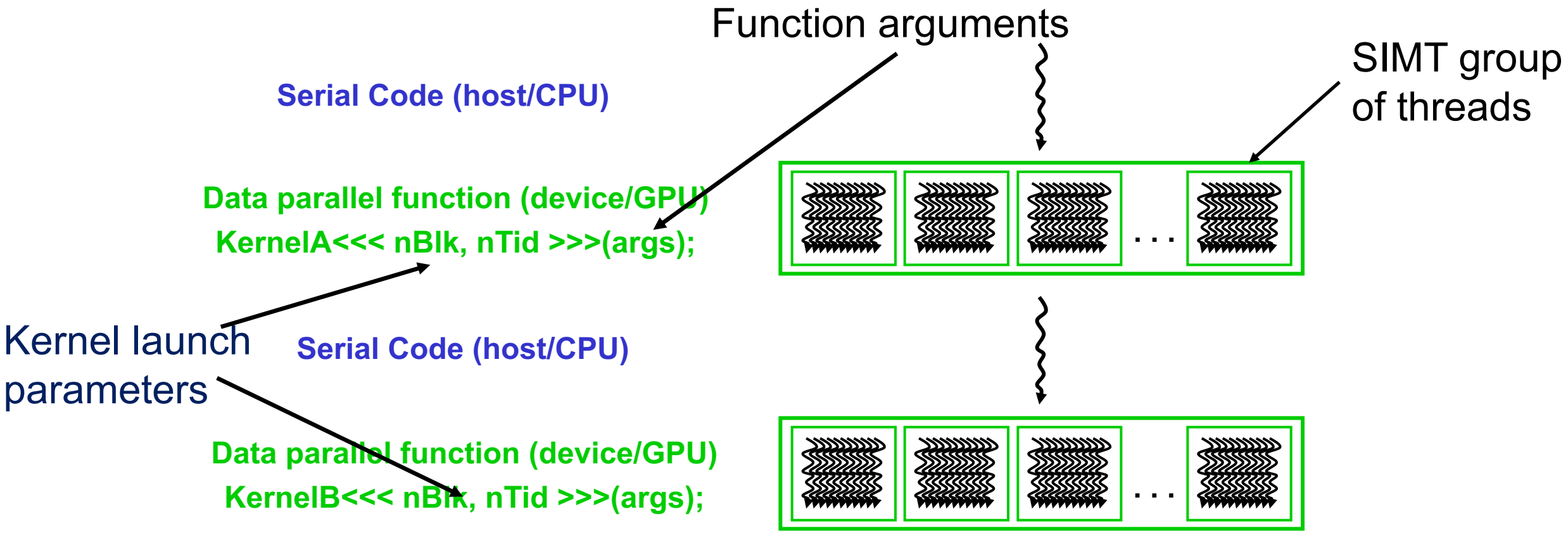
# GPU Programming Model

- There exists no “pure” GPU program
- Host or CPU program that launches work on GPU

- A *function* is accelerated on the GPU → a.k.a., GPU kernel (not to be confused with Linux kernel)

- GPU kernels follow the Single Program Multiple Thread/Data model
  - GPU kernel code specifies what *each* thread must do
  - Thread index specifies which data the thread must operate on
  - Kernel is launched by the CPU, specifying how many threads should execute the kernel

# Division of Labor Between CPU (Host) and the GPU (Device)



# Review: Don't Forget Amdahl's Law

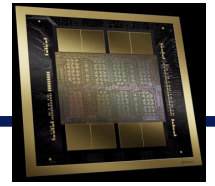
- Consider a CPU application before using GPU:
  - The sequential execution time is $100s$
  - The fraction of execution that is parallelizable is **80**%
  - GPU can speedup the parallelizable part by **100** ×

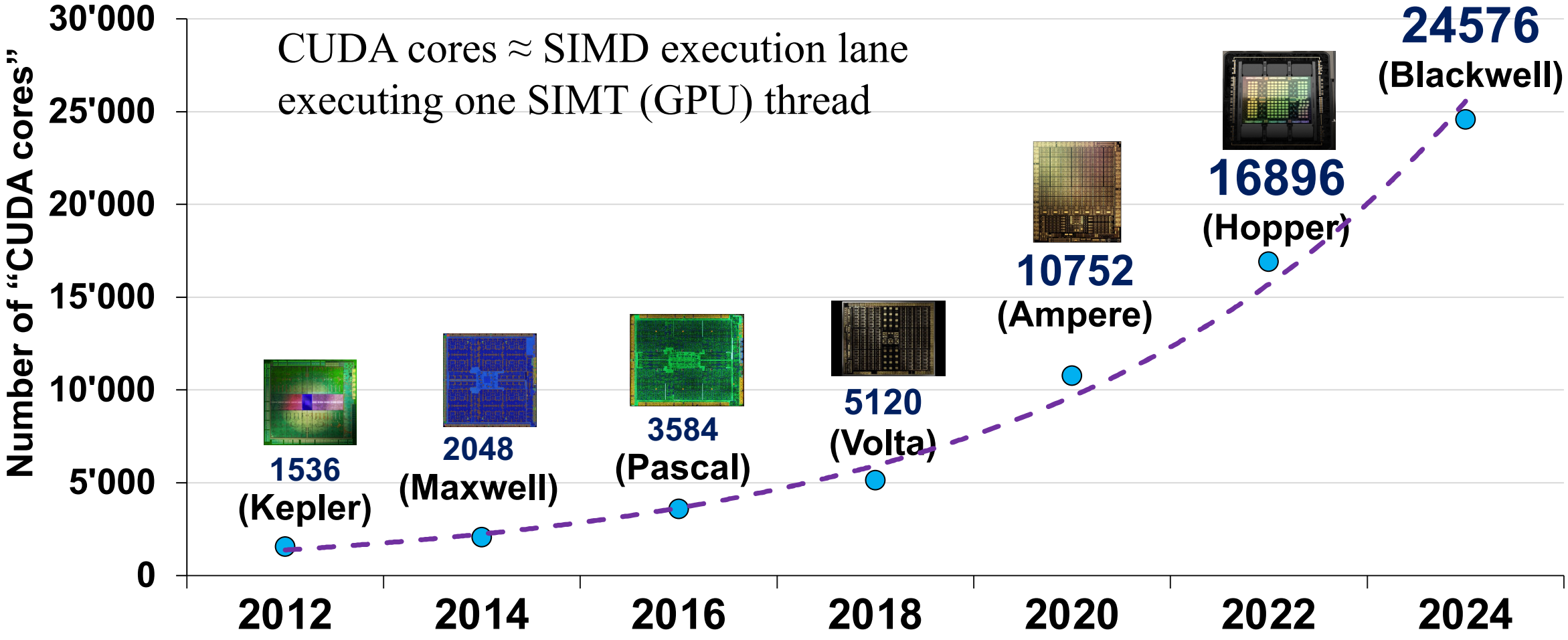
- What is the overall speedup of the application?

$$t_{parallel} = (1 - \mathbf{0.8}) * 100s + \frac{\mathbf{0.8} * 100s}{\mathbf{100}} = 20.8s$$

$$speedup = \frac{t_{sequential}}{t_{parallel}} = \frac{100s}{20.8s} = 4.81\times$$

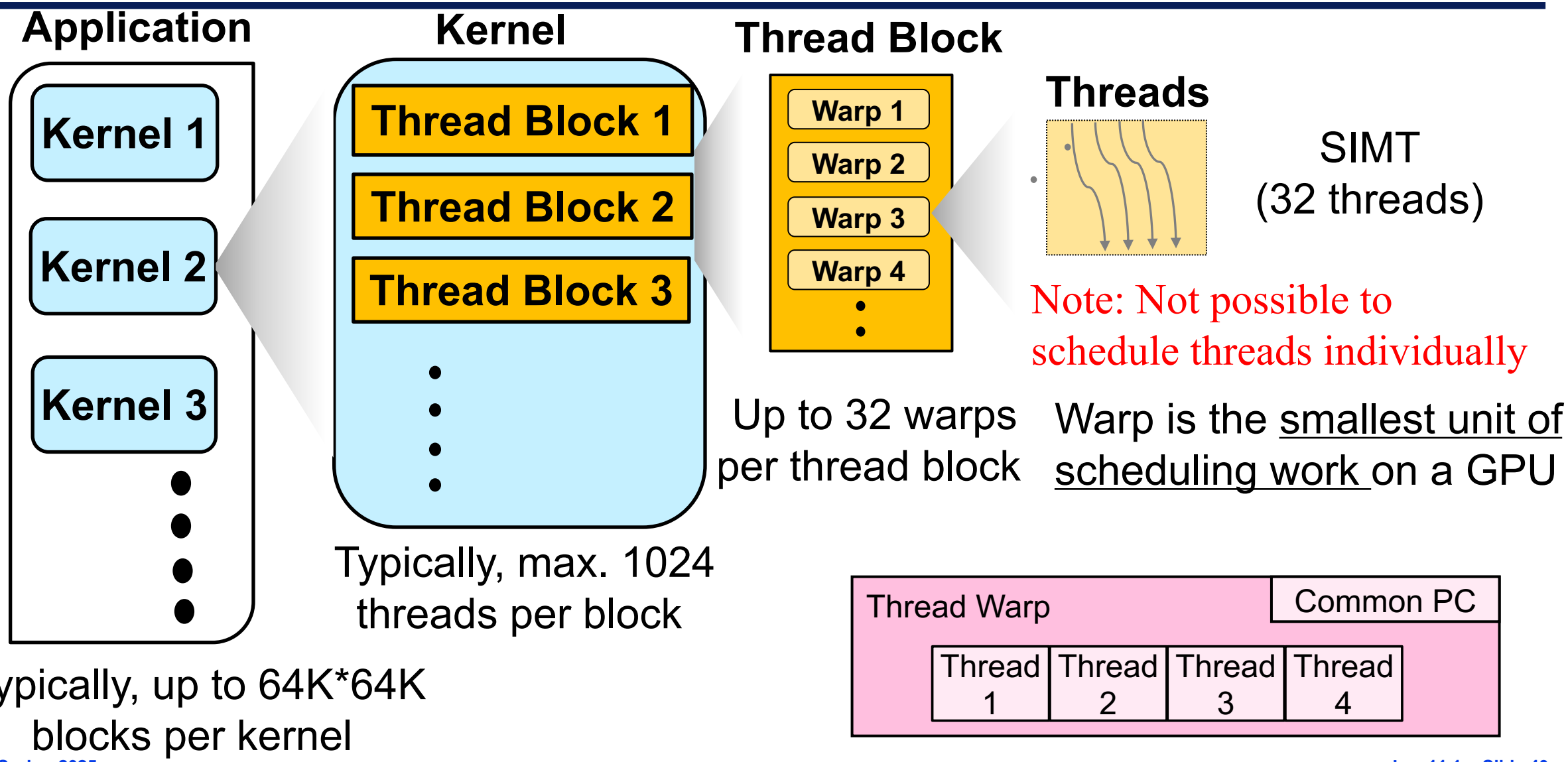
# Recall: Scale of Parallelism in Modern GPUs






Hierarchical execution and programming model for scaling parallelism

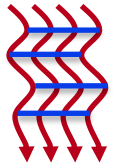
# The Hierarchy in GPU Thread Organization



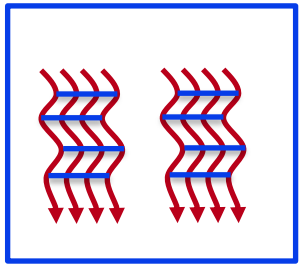
# Summary of Thread Hierarchy and Terminologies

A GPU thread runs on a SIMD lane (CUDA core)

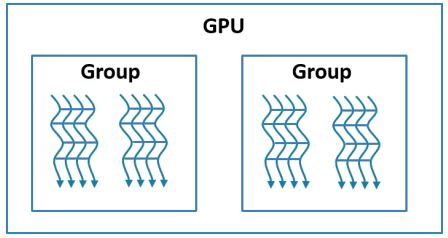
A SIMT thread group executing in lockstep



A group of SIMT thread groups executing on the same SM/CU

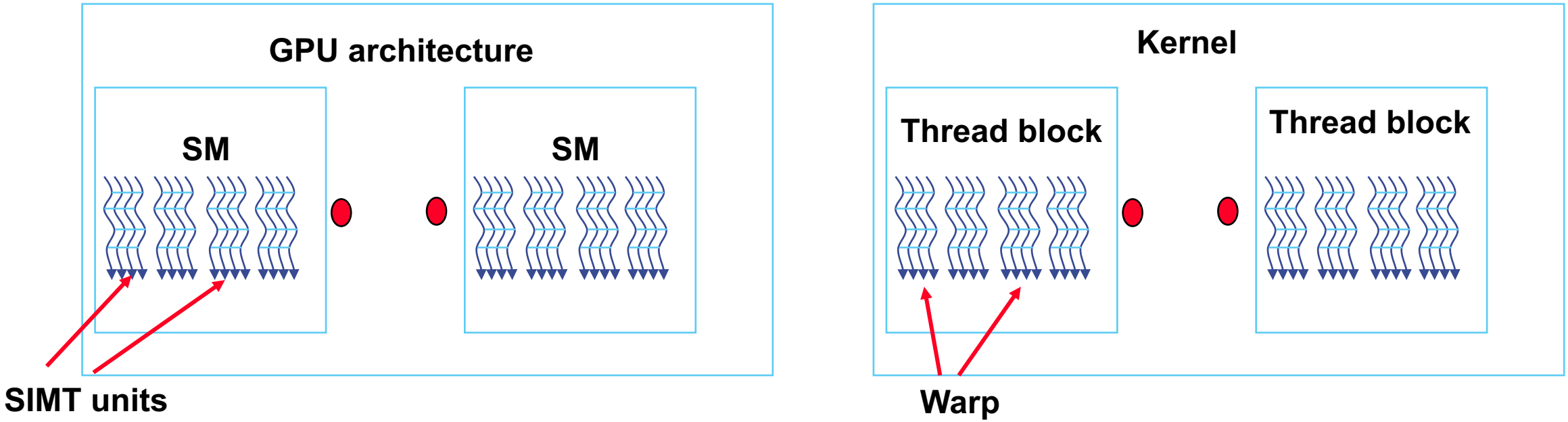


A group of a group of SIMT thread groups executing on GPU




| CUDA | OpenCL |
|---|---|
| Thread | Work-item |
| Warp | Wavefront |
| Block | Workgroup |
| Grid | NDRange |

# Mapping the Thread Hierarchy onto the Hardware

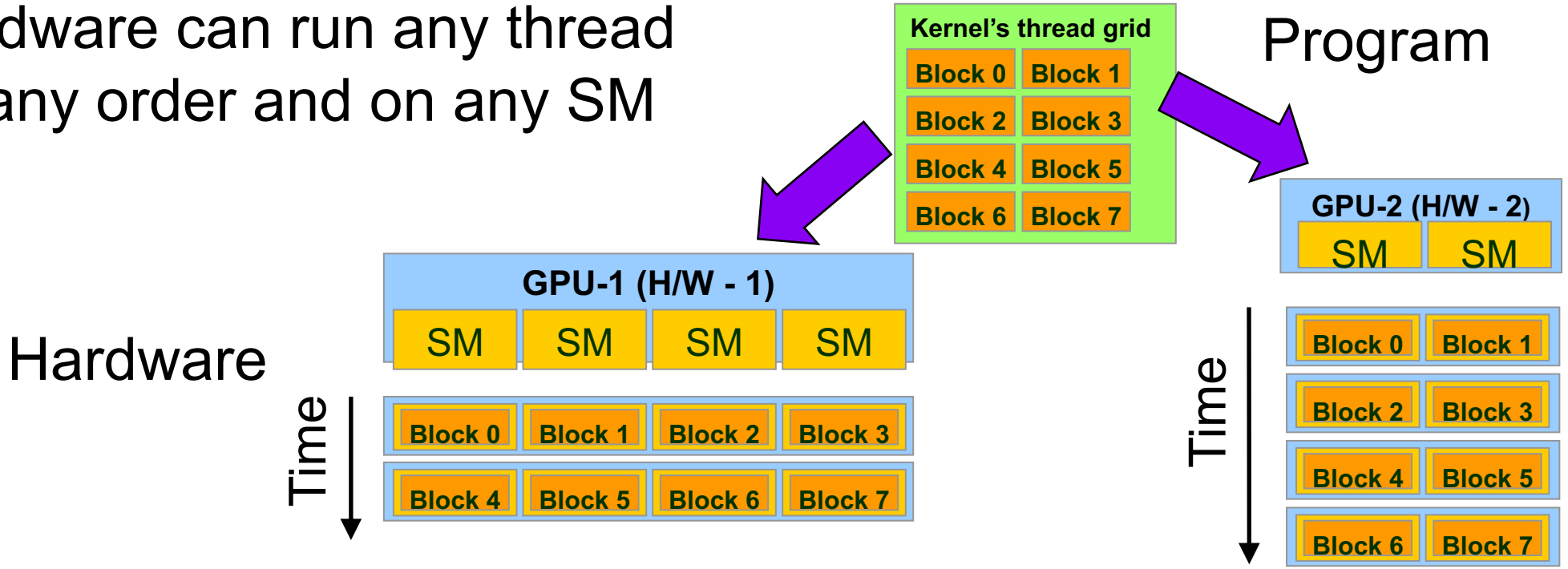



## Scheduling rules in a CUDA/GPU

1) All warps of a thread block are scheduled on the same SM
2) Different thread blocks can execute on the same or different SMs
3) Different thread blocks can execute concurrently; no ordering

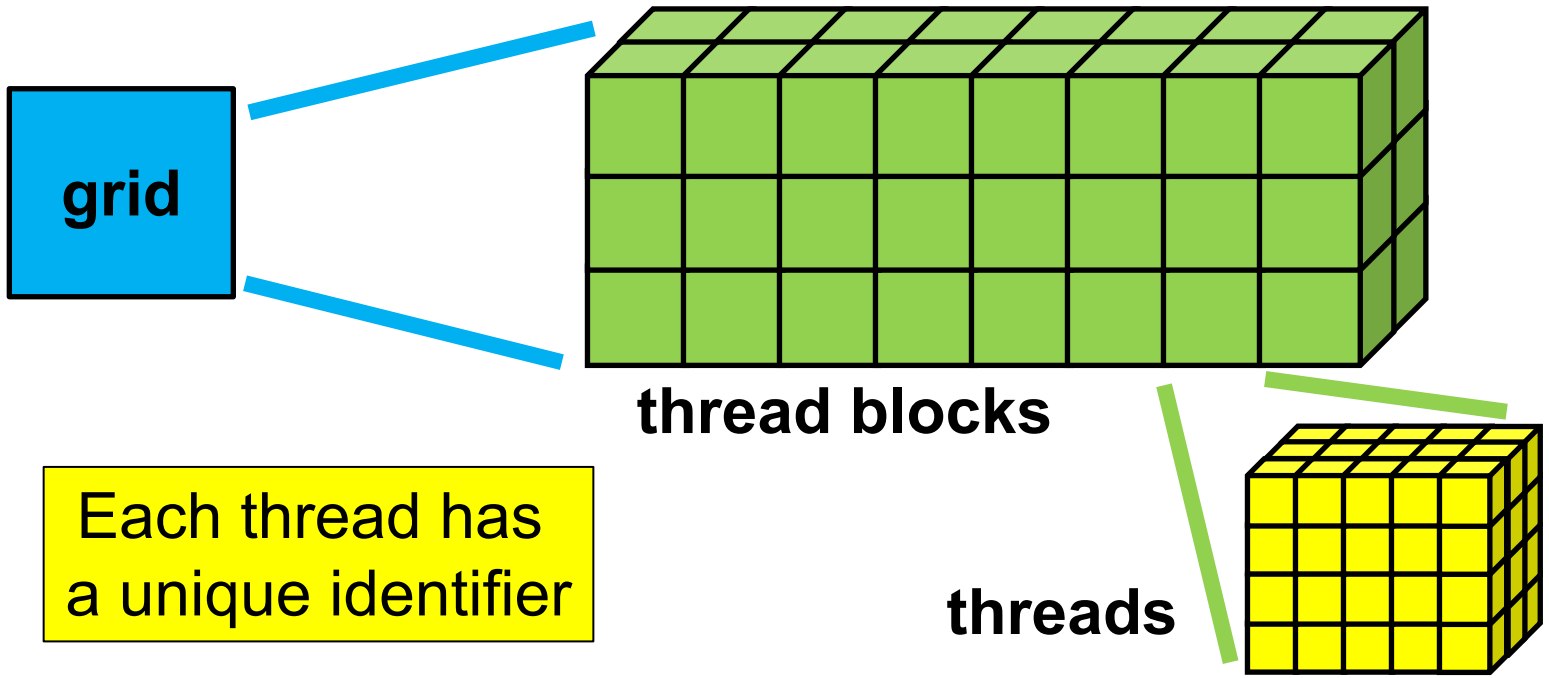
# Advantages of Breaking the Set of Threads into Thread Blocks

GPU hardware can run any thread block in any order and on any SM




- Decoupling hardware resources in a given GPU from kernel threads
- Threads within a thread block can collaborate more efficiently than those from different thread blocks → helps scaling up !

# Logical Thread Hierarchy in CUDA




Each thread has a unique identifier

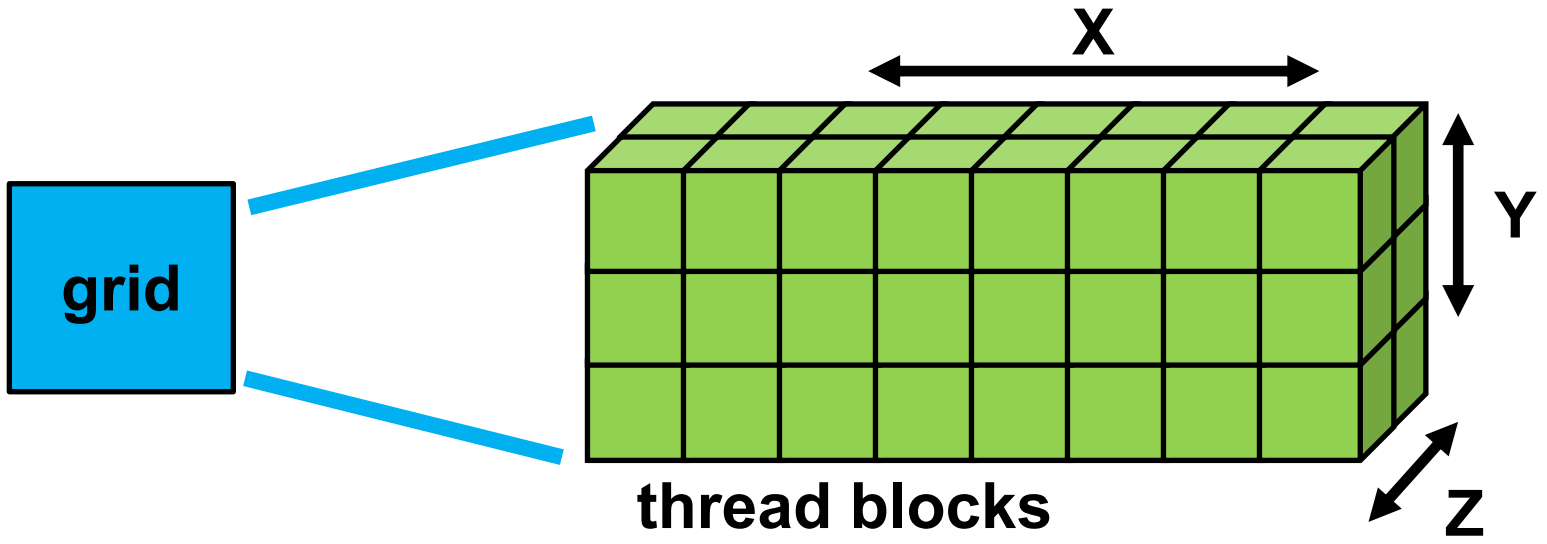
- A kernel is launched with a grid of threads
- A grid is a 3D array of thread blocks and threads

Recall that kernel follows SPMD

Must uniquely identify each thread and thread block

Each thread computes on different data items based on its identity

# Number of Thread Blocks in a Grid (*gridDim*)



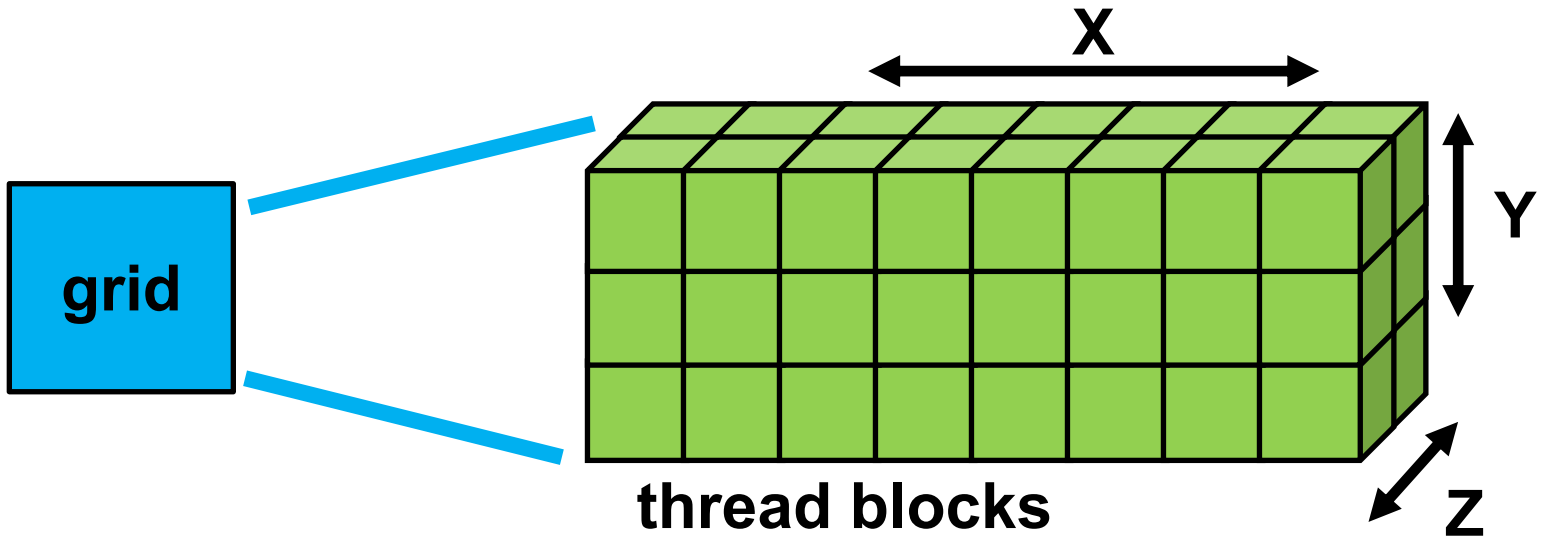

For 2D (and 1D grids), simply use grid dimension 1 for Z (and Y).

- gridDim.x …
- gridDim.y …
- gridDim.z …

**Intrinsic variables:**
Available to each thread automatically → No need to allocate/initialize.

H/W returns the correct value for the calling thread

# Uniquely Identifying Thread block

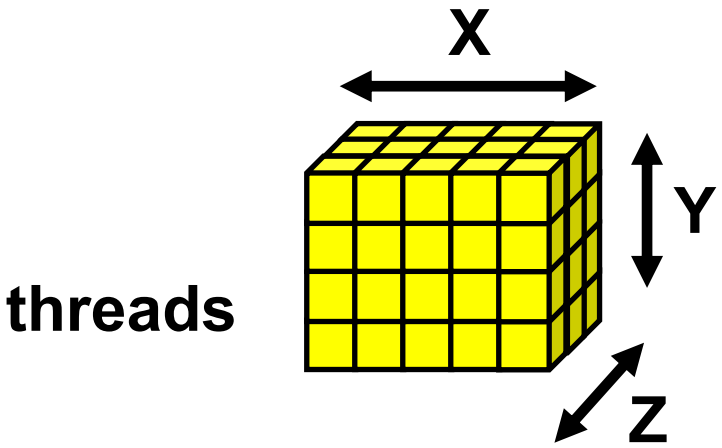



For 2D (and 1D grids), simply use grid dimension 1 for Z (and Y).

Each thead block has a unique index tuple

- blockIdx.x [from 0 to (gridDim.x – 1) ]
- blockIdx.y [from 0 to (gridDim.y – 1) ]
- blockIdx.z [from 0 to (gridDim.z – 1) ]

# Finding the Number of Threads in a Thread Block



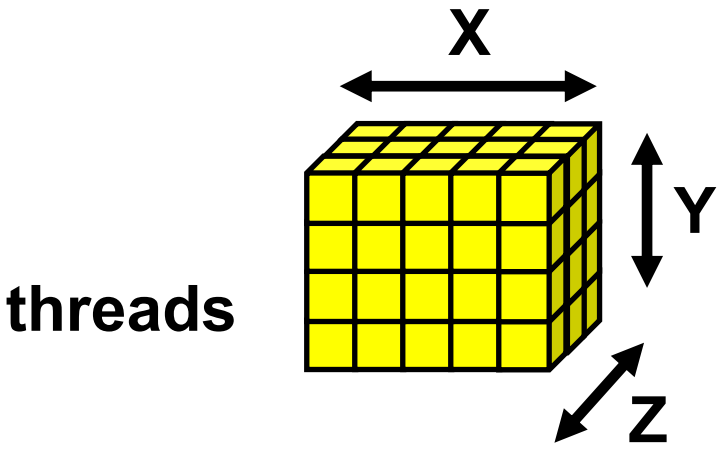

- The number of threads in each dimension in a thread block
  - blockDim.x …
  - blockDim.y …
  - blockDim.z …

For 2D (and 1D grids), simply use grid dimension 1 for Z (and Y).

# Finding the Number of Threads in a Thread Block

Why a 3D grid instead of a linear array of threads?

- Historical reason stemming from graphics
- Easier for image processing, PDEs on volumes

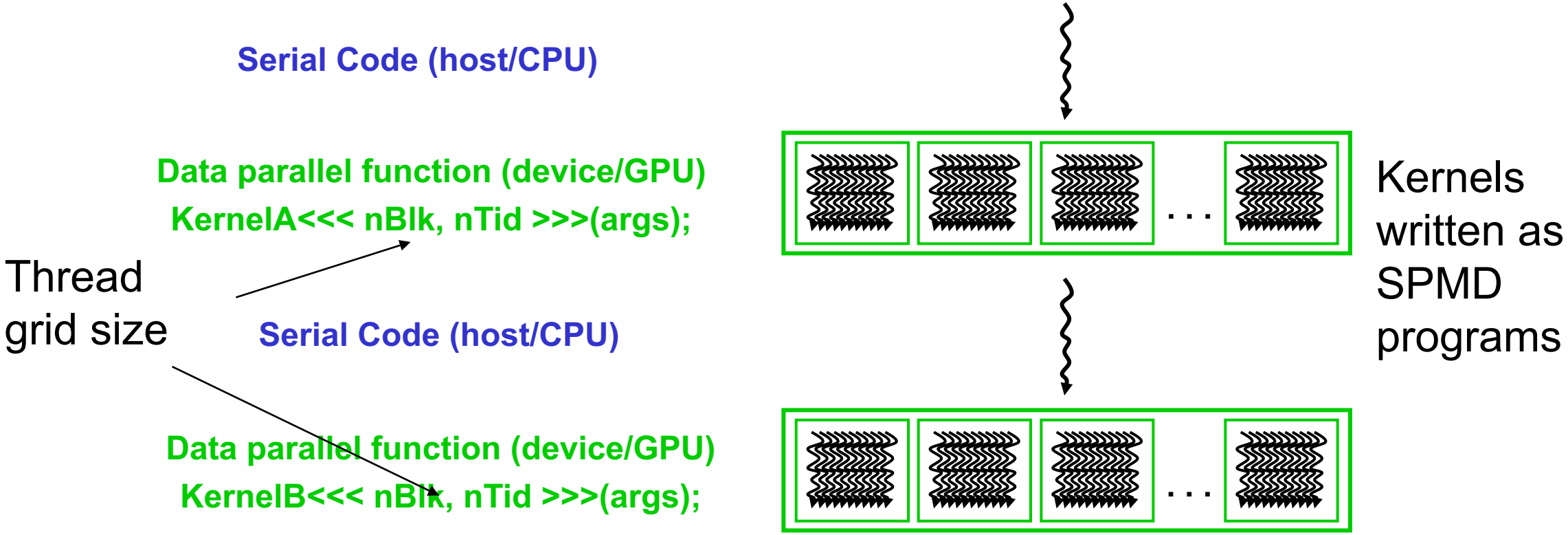



Each thread has a unique index tuple

- threadIdx.x (from 0 to (blockDim.x – 1) )
- threadIdx.y (from 0 to (blockDim.y – 1) )
- threadIdx.z (from 0 to (blockDim.z – 1) )

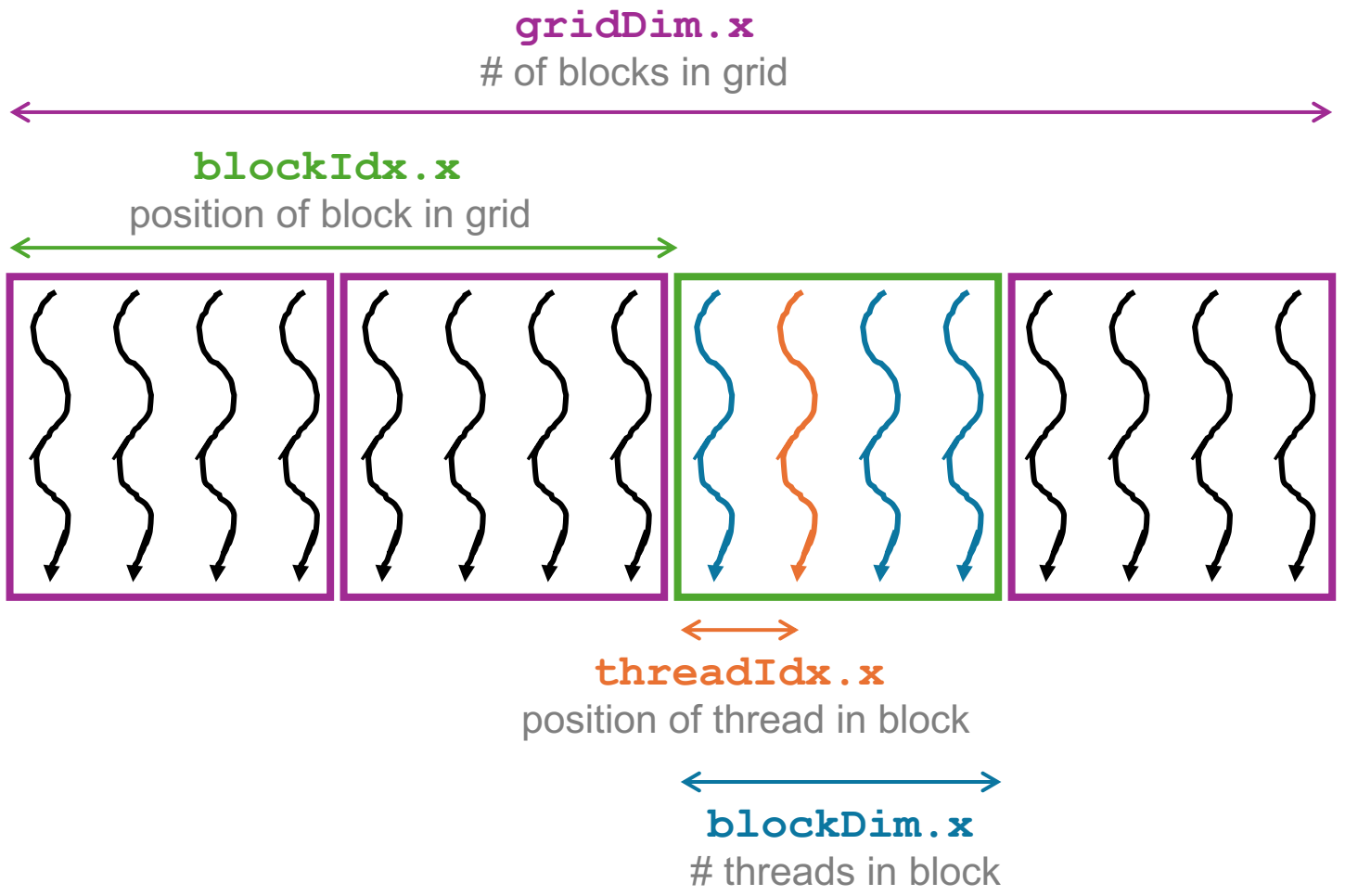
threadIdx tuple is unique *within* a threadblock

# Review: What Have We Learnt So Far?

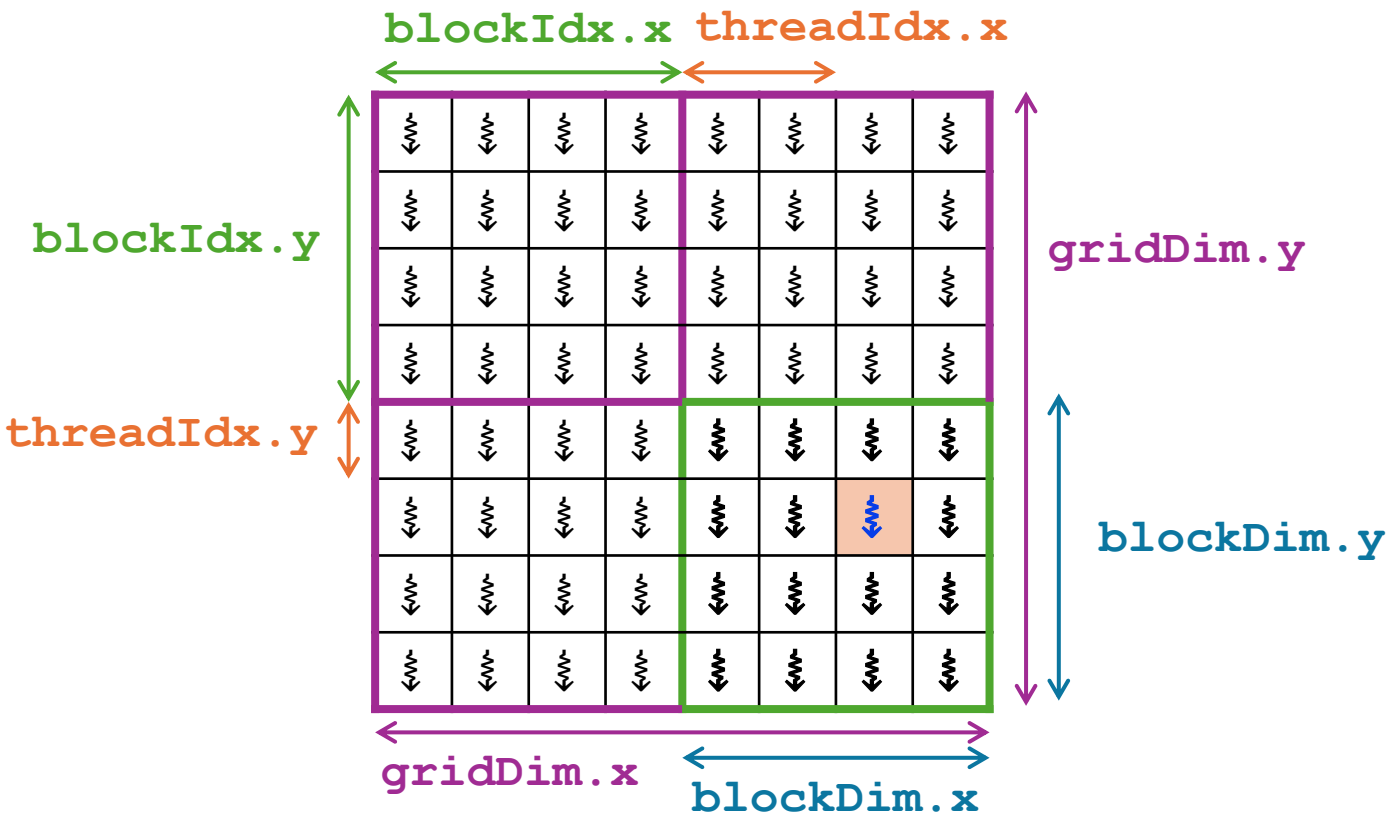



- Next: The life cycle of a GPU-accelerated program with an example

# Visualization of Thread Organization and Indexing in 1D

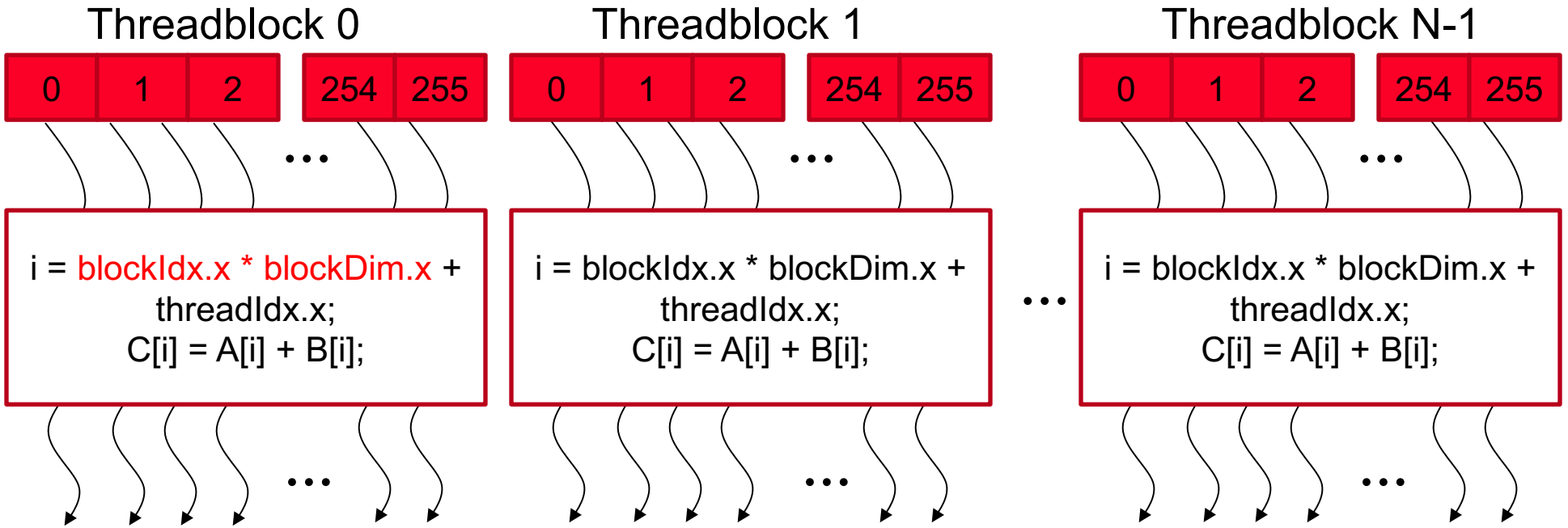


# Visualization of Thread Organization and Indexing in 2D

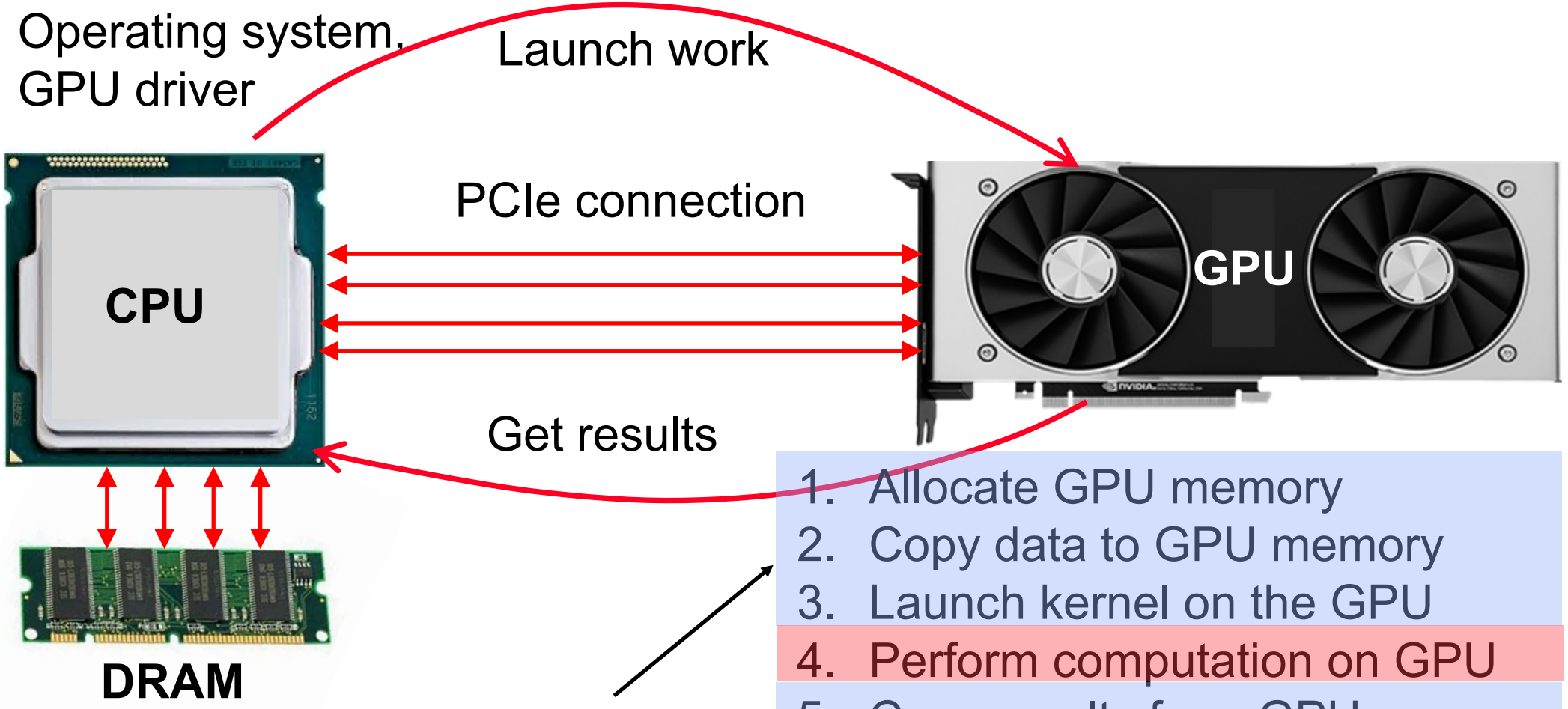


# Identifying Data Item That Each Thread Must Work On

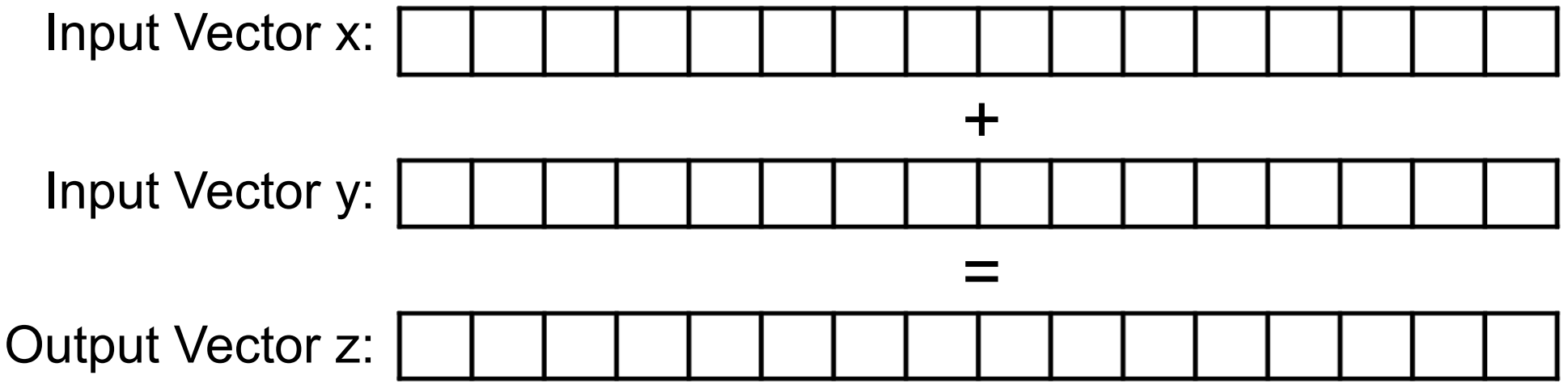
- Recall: GPU kernel specifies program (instructions) that each thread must execute
- Each thread will execute the same program on different data items based on its unique identity



# Key Steps in the Lifecycle of a GPU-Accelerated Program



# Example Program for Walkthrough: Vectoradd




```
void vecadd(float* x, float* y, float* z, int N) {
    for(unsigned int i = 0; i < N; ++i) {
        z[i] = x[i] + y[i];
    }
}
```

Sequential C Code

# APIs for (1) GPU Memory Allocation and (6) De-Allocation

- Allocating memory:

  `cudaError_t cudaMalloc(void **devPtr, size_t size)`

  - `devPtr`: Pointer to a pointer to allocated device (GPU) memory
  - `size`: Requested allocation size in bytes

- Deallocating memory:

  `cudaError_t cudaFree(void *devPtr)`

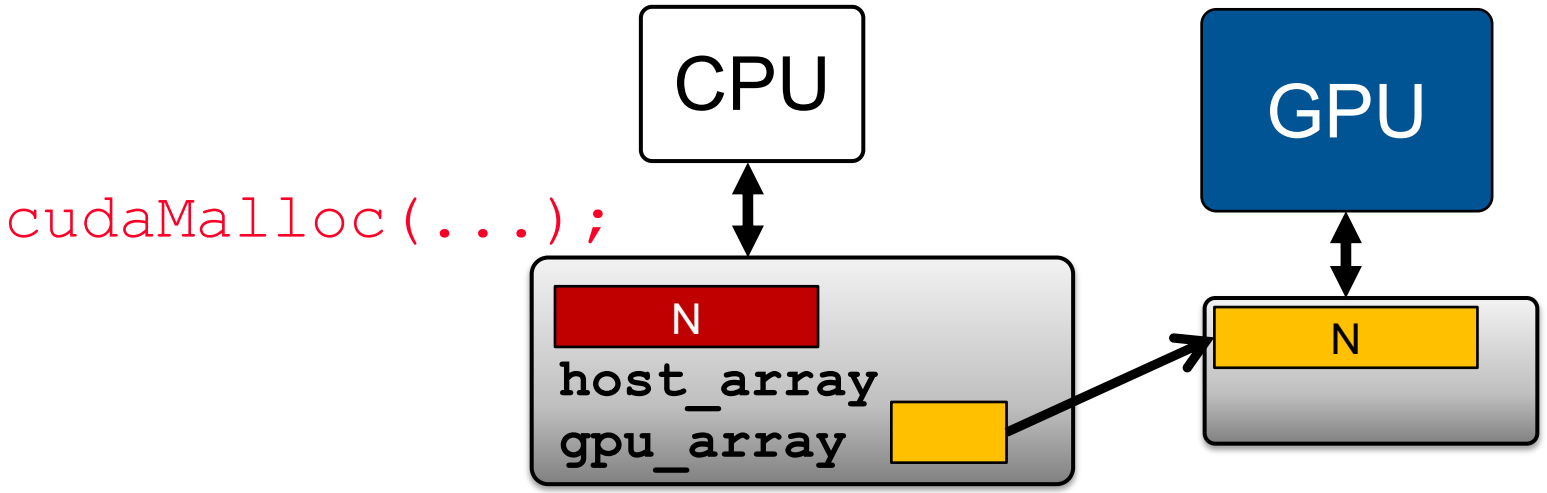
  - `devPtr`: Pointer to device memory to free

- Return type: `cudaError_t`
  - Helps with error checking

# APIs for (1) GPU Memory Allocation and (6) De-Allocation

```
int main(...)
{
    int host_array[N];
    int* gpu_array;
    cudaMalloc( (void**)&gpu_array, SIZE_N);
}
```

A CUDA runtime routine that ultimate calls the GPU driver running on CPU to allocate GPU memory



# How Does `gpu_array` Pointer Work?

- Disclaimer: It is **not** a regular C pointer!
  - If you dereference it, the behavior is undefined

- The host **pointer** is on the CPU's stack
  - But, it serves as a simple name for the CUDA library
  - When calling `cudaMalloc(…)`, device driver tells the GPU to "please allocate `SIZE_N` bytes of memory"

# API for (2) Copying Data to and from (6) GPU Memory

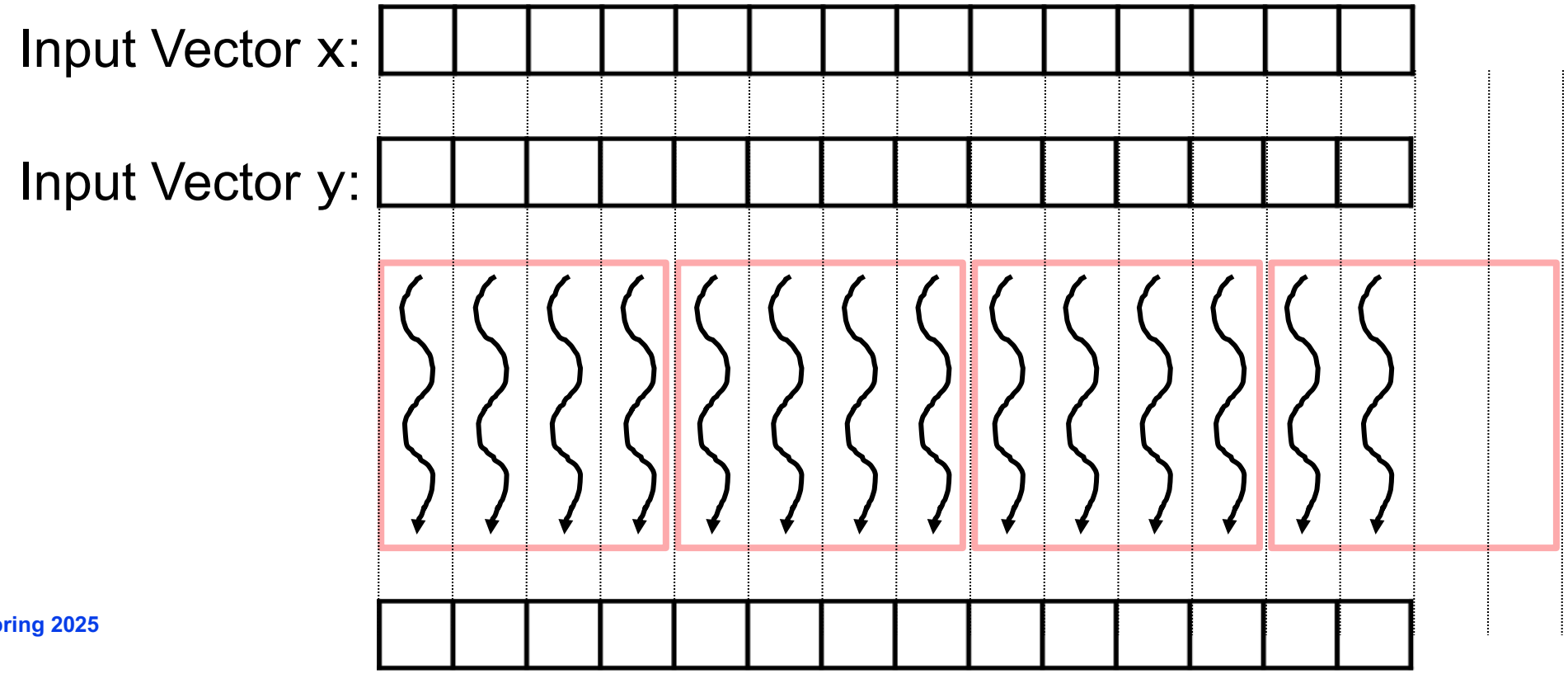
- **`cudaError_t cudaMemcpy(void *dst, const void *src, size_t count, enum cudaMemcpyKind kind)`**
  - `dst`: Destination memory address
  - `src`: Source memory address
  - `count`: Size in bytes to copy
  - `kind`: Type of transfer
    - `cudaMemcpyHostToHost`
    - `cudaMemcpyHostToDevice`
    - `cudaMemcpyDeviceToHost`
    - `cudaMemcpyDeviceToDevice`
  - cudaMemcpy is a CUDA runtime routine that ultimately invokes the GPU driver

# Vector Addition (CPU/Host Side Code)

```
void vecadd(float* x, float* y, float* z, int N) {

    // Allocate GPU memory
    float *x_d, *y_d, *z_d;
    const unsigned int numThreadsPerBlock, numBlocks;
    cudaMalloc((void**) &x_d, N*sizeof(float));
    cudaMalloc((void**) &y_d, N*sizeof(float));      // Step (1)
    cudaMalloc((void**) &z_d, N*sizeof(float));

    // Copy data to GPU memory
    cudaMemcpy(x_d, x, N*sizeof(float), cudaMemcpyHostToDevice);   // Step (2)
    cudaMemcpy(y_d, y, N*sizeof(float), cudaMemcpyHostToDevice);

    // Perform computation on GPU
     numThreadsPerBlock = 512;
     numBlocks = (N + numThreadsPerBlock - 1)/numThreadsPerBlock;
     vecadd_kernel <<< numBlocks, numThreadsPerBlock >>> (x_d, y_d, z_d, N);  //Step (3). Note warp is not specified

    // Copy data from GPU memory
    cudaMemcpy(z, z_d, N*sizeof(float), cudaMemcpyDeviceToHost);     //Step (5)

    // Deallocate GPU memory
    cudaFree(x_d);
    cudaFree(y_d);      //Step (6)
    cudaFree(z_d);

}
```

# Vector Add (Device/GPU Code)

```
//Step (4)
__global__ void vecadd_kernel(float* x, float* y, float* z, int N) {
    int i = blockDim.x*blockIdx.x + threadIdx.x;
    if (i < N)
      z[i] = x[i] + y[i];
}
```



# Keywords for Function Declaration

__host__ keyword can be added for functions that can be executed on both CPU and GPU

| Keyword | Callable From | Executed On |
|---|---|---|
| __host__ (default) | Host | Host |
| __global__ | Host (or Device) | Device |
| __device__ | Device | Device |

```
__host__ __device__ float f(float a, float b) {
    return a + b;
}
__global__ void vecadd_kernel(float* x, float* y, float* z, int N) {
    int i = blockDim.x*blockIdx.x + threadIdx.x;
    if (i < N) {
        z[i] = f(x[i], y[i]);
    }
}
```

# How to Decide Thread Block Size?

- Recall that a thread block can have *up to* 1024 threads
  - Programmer has the option of choosing from a range
- A key step in GPU programming
  - Dividing up the work between thread blocks
- For max perf., GPU should be fully occupied → leverage as much parallelism as the hardware allows
  - Every cycle should have a warp ready to issue
  - Requires you to think about your kernel's operations
    - Which ones are long latency?
    - How many warps can be swapped in while the current warp is waiting?

# Occupancy Example

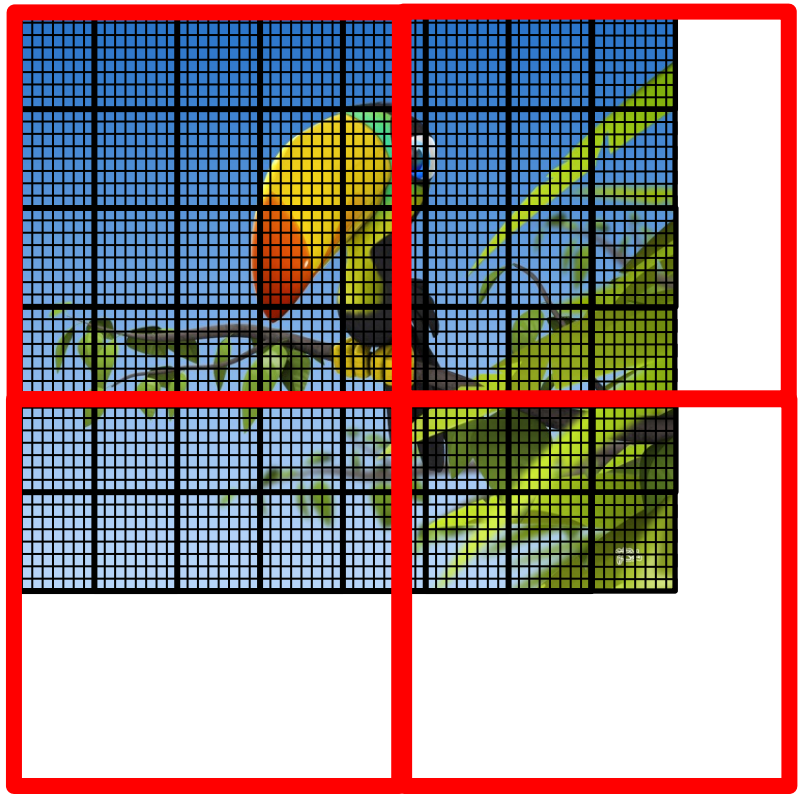
- Assume that we want to compute on an image of size  64x48
  - One thread per pixel means we need ~3k threads
  - But, how to arrange them?

- Example resources available in a GPU and constraints
  - 15 SMs, max 64 warps per SM
  - 2048 threads max per SM
  - 16 thread blocks max per SM
  - 1 warp is 32 threads
  - 960 concurrently scheduled warps/GPU
    - You can launch more, but they will serialize

Whichever limit hits first constraints parallelism

# Occupancy Example: Big Blocks

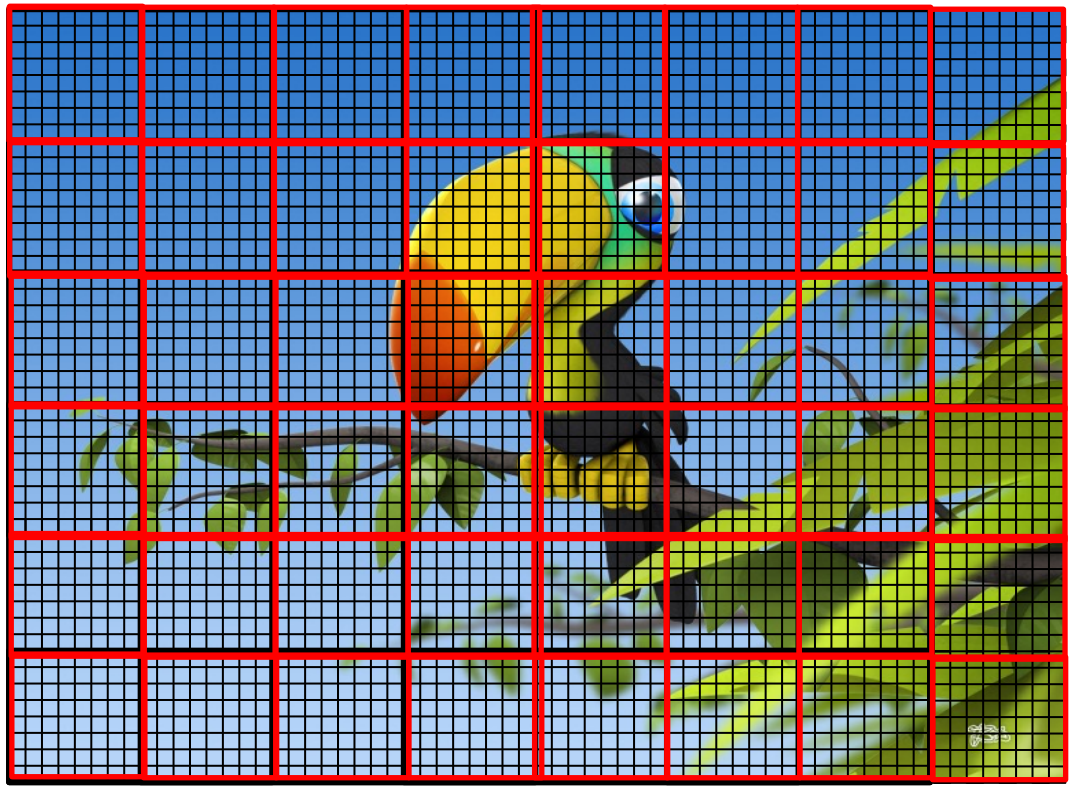
- 4 large thread blocks of 1024 threads each
  - Only 4 of 15 SMs occupied
  - Absolute max occupancy is 26%, not using all HW



**Why?**

# Occupancy Example: Smaller Blocks

- 48 thread blocks of 64 threads each
  - SMs have 3 or 4 threadblocks, each with 2 warps



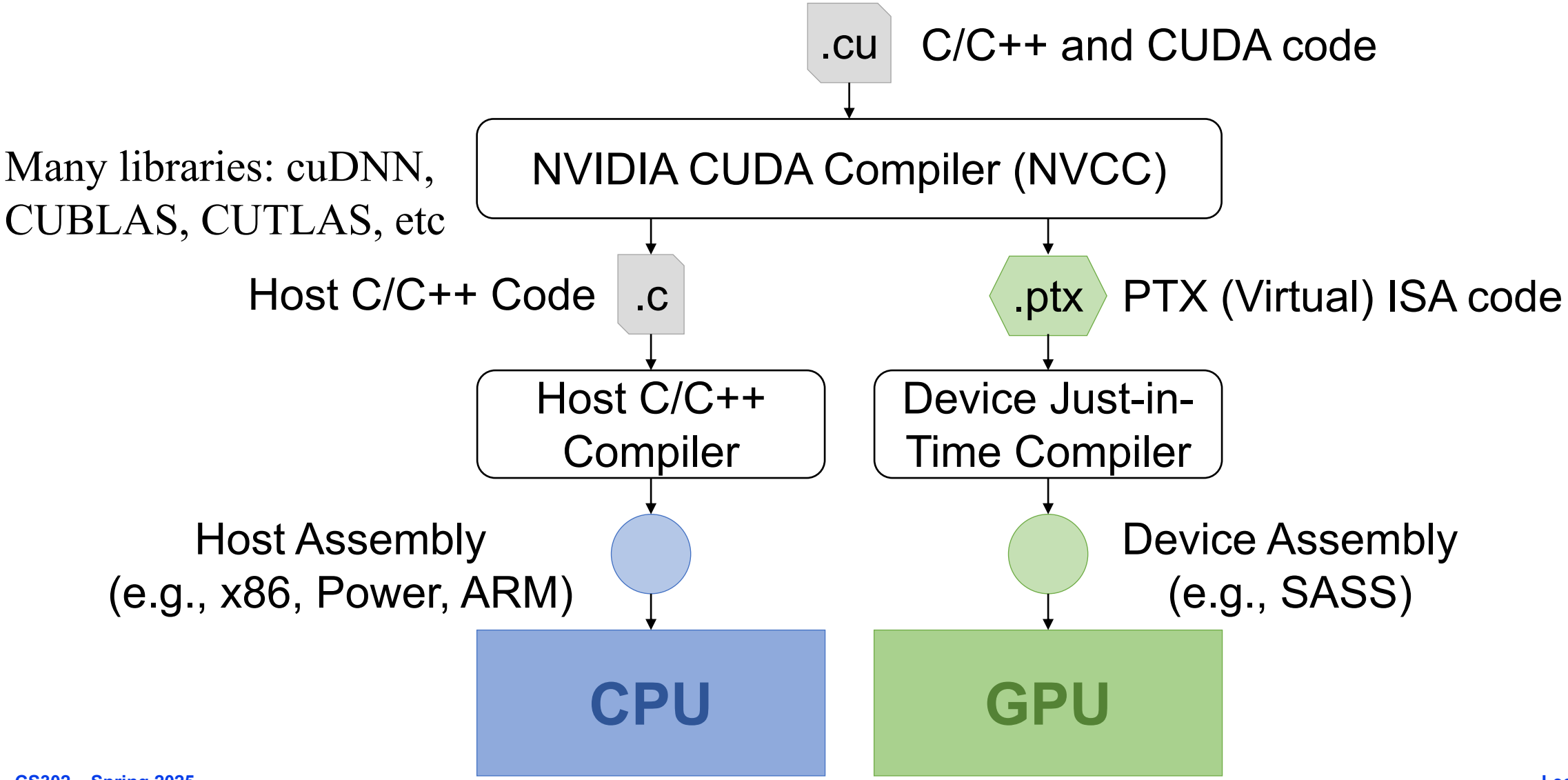
# Another Example (2): Occupancy in an SM

- Constraints:
  - 15 SMs, max 64 warps per SM
  - 2048 threads max per SM
  - 16 thread blocks max per SM
  - 1 warp is 32 threads
- Assume that there are many threads available to run
- Goal: Schedule as many threads as possible in each SM
- If thread block set to 768 threads, each SM can accommodate only two thread blocks → Maximum 1536 threads in each SM 75% (1536/2048) max occupancy

# Another Example (3): Occupancy in an SM

- Constraints:
  - 15 SMs, max 64 warps per SM
  - 2048 threads max per SM
  - 16 thread blocks max per SM
  - 64K register per SM
  - 1 warp is 32 threads
- Assume that there are many threads available to run
- Goal: Schedule as many threads as possible in each SM
- If each thread uses 64 registers, up to 1024 threads can be scheduled in each SM

# CUDA Compilation Flow



# Summary: GPUs Are Accelerators for Massive Data-Parallelism

- Originally designed for graphics but emerged as a platform of choice for data-parallel computing
- GPU follows SIMT execution model while the GPU kernels are specified in SPMD fashion
- The host (CPU) code is responsible for allocating memory in GPU, copying data to GPU memory, launching kernels with a grid of threads, copying results back and de-allocate GPU memory