EECS 570
Lecture 15
GPU programming

Winter 2025

Prof. Satish Narayanasamy

http://www.eecs.umich.edu/co4urses/eecs570/

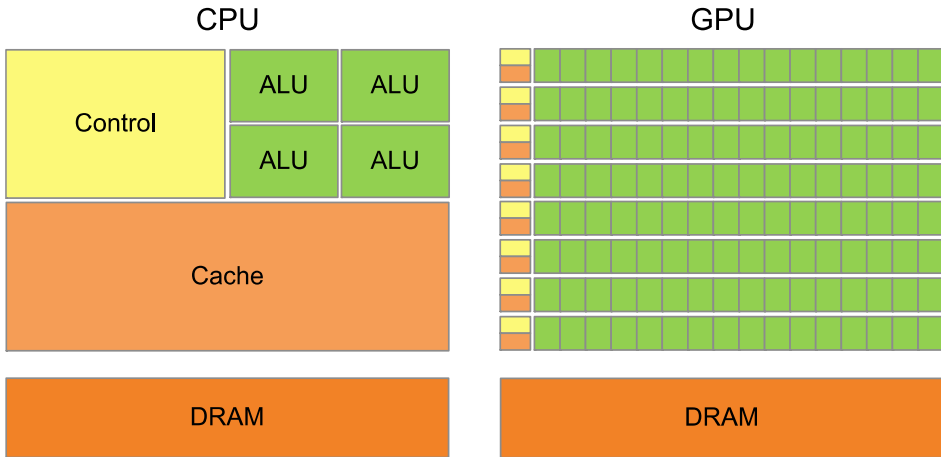
Slides adapted from instructional material with D. Kirk and W. Hwu, Programming Massively Parallel Processors: A Handson Approach, Third Edition.

Credits to Nikos Hardavellas (Northwestern), Reetu Das (UM)

Objective

- To learn the basic concept of data parallel computing
- To learn the basic features of the CUDA C programming interface

CPU vs. GPU summary



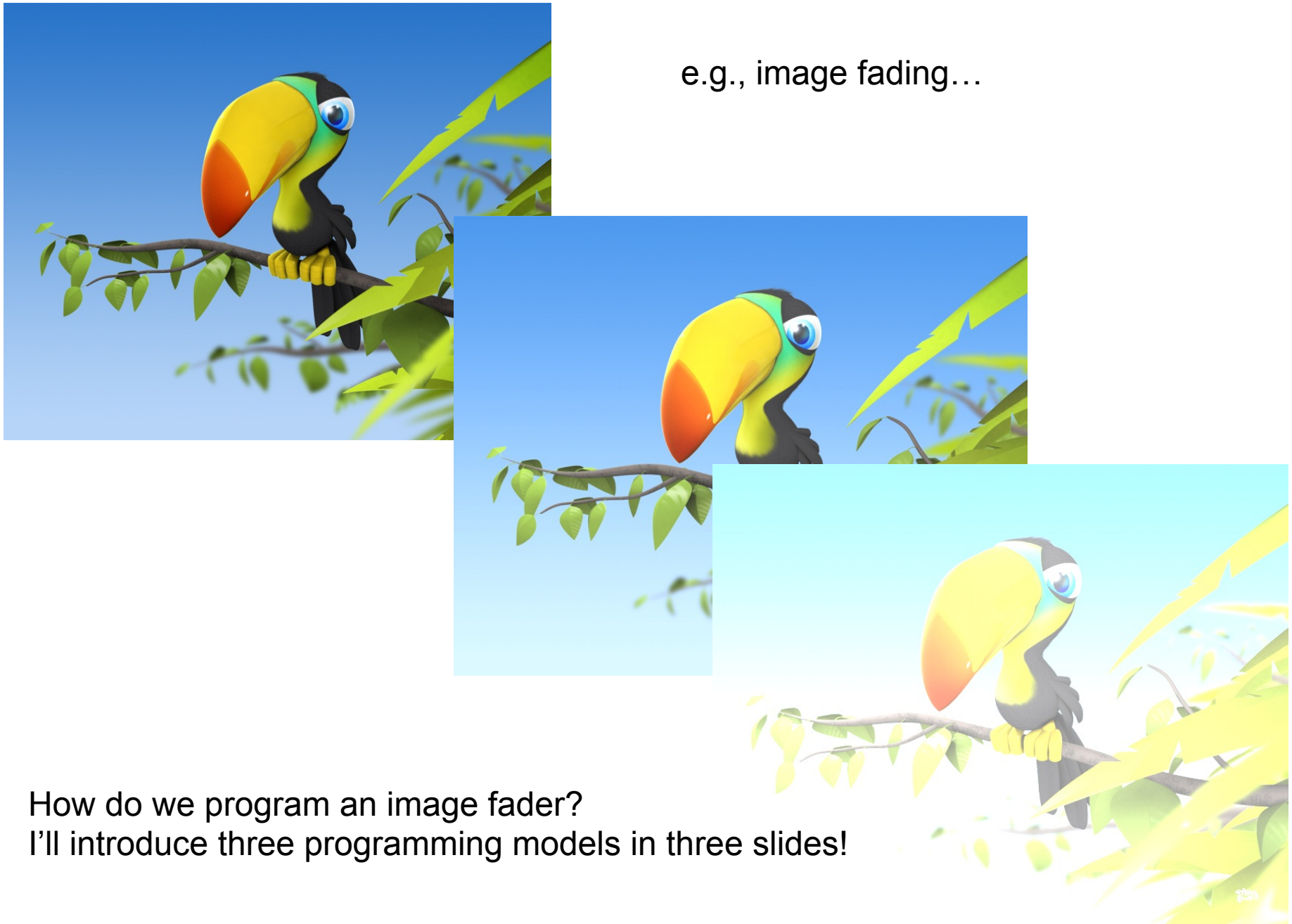
CPU:

- Handles sequential code well
  - Latency optimized: do all very fast
- Can't take advantage of massively parallel code
- Off-chip bandwidth lower – narrow pipes
- Lower peak computation capability

GPU:

- Requires massively parallel computation
  - Bandwidth optimized: do lots concurrently
- Handles some control flow
- Higher off-chip bandwidth – wide pipes
- Higher peak computation capability

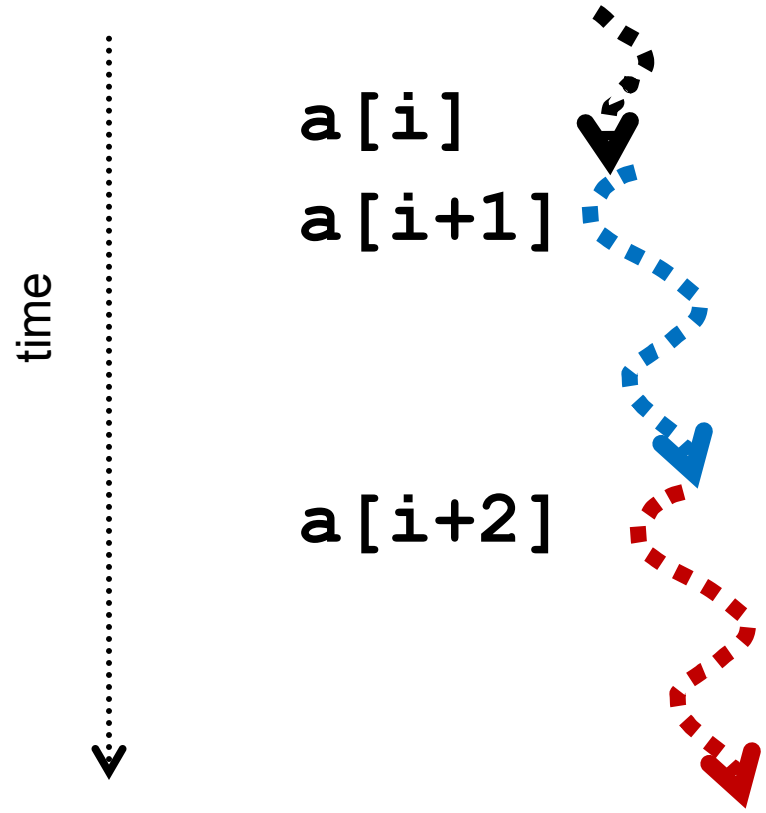
Some things are naturally parallel



e.g., image fading…

How do we program an image fader?
I'll introduce three programming models in three slides!

```
int a[N]; // a is image, N is large
for (i =0; i < N; i++){
   a[i] = a[i] * fade;
}
```

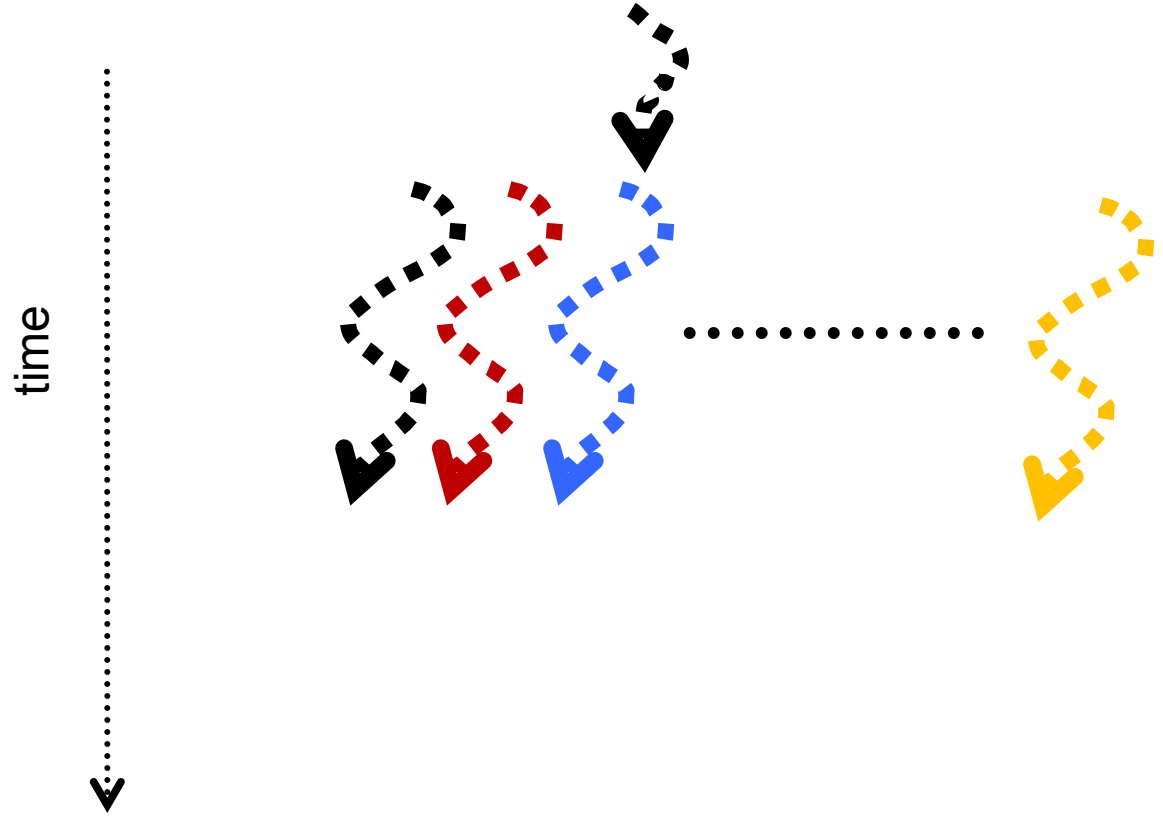


Flow of control / Thread
One instruction at the time
Optimizations possible at the machine level

This is the predominant CPU model

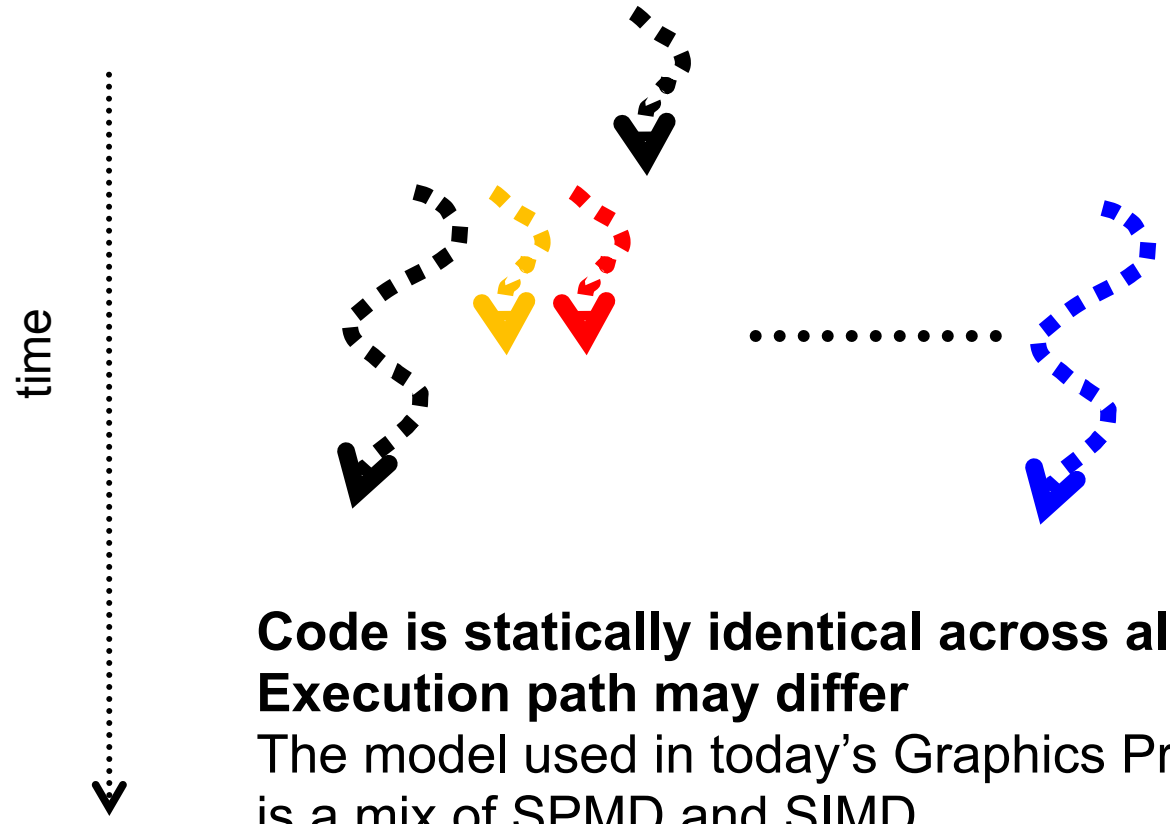
Lots of optimizations to shorten the time required for each individual operation

```
int a[N]; // N is large
for all elements do in parallel {
   a[i] = a[i] * fade;
}
```



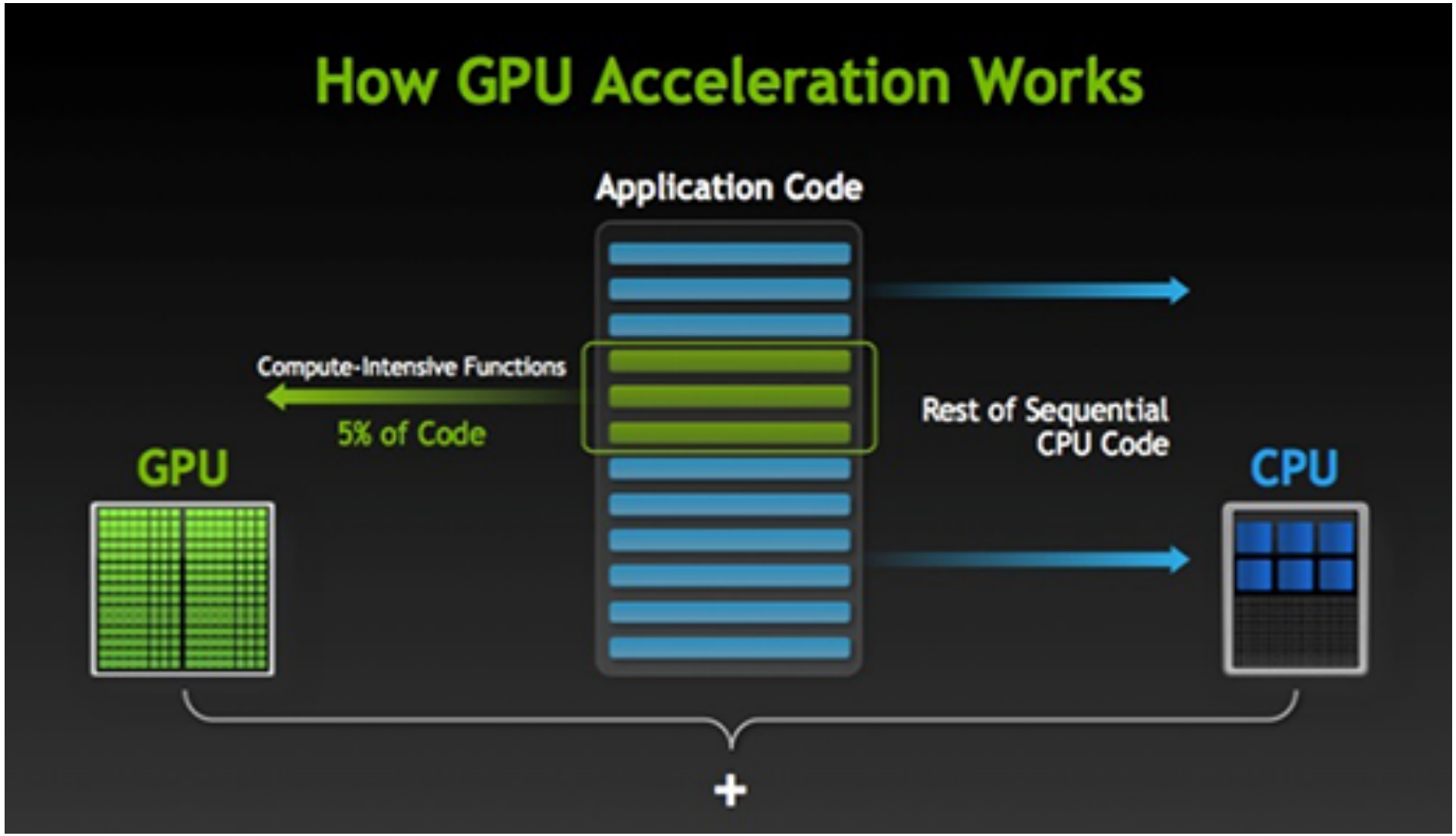
Most modern CPUs offer some limited support for SIMD

```
int a[N]; // N is large
for all elements do in parallel {
   if (a[i] > threshold) a[i]*= fade;
}
```



Code is statically identical across all threads
Execution path may differ
The model used in today's Graphics Processor Units (GPUs) is a mix of SPMD and SIMD

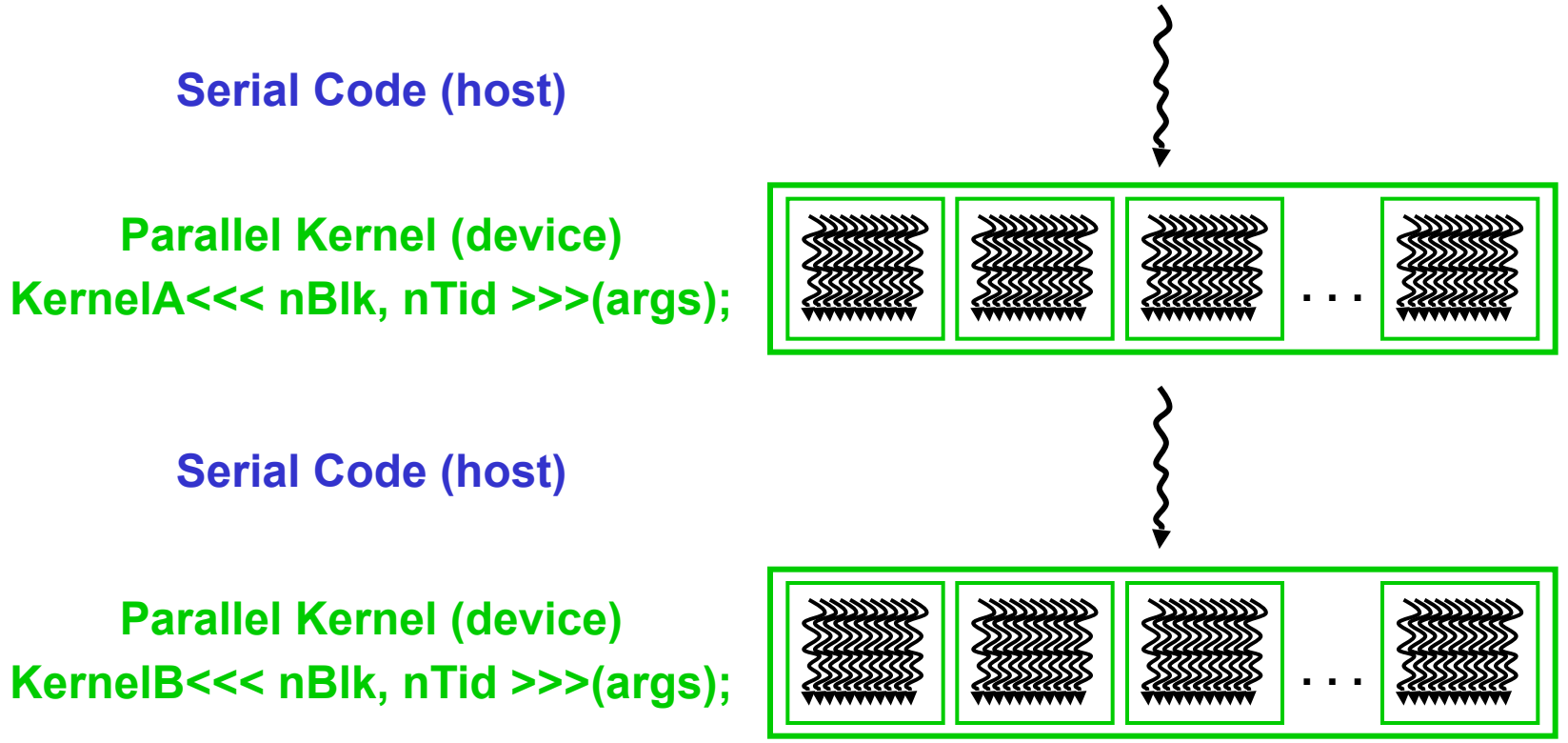
How GPU Acceleration Works



CUDA/OpenCL – Execution Model

Integrated host+device app C program

- Serial or modestly parallel parts in **host** C code
- Highly parallel parts in **device** SPMD kernel C code



```
__global__ void arradd (float *a, float f, int N)
 {
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  if (i < N) a[i] = a[i] + f;
}
```

GPU

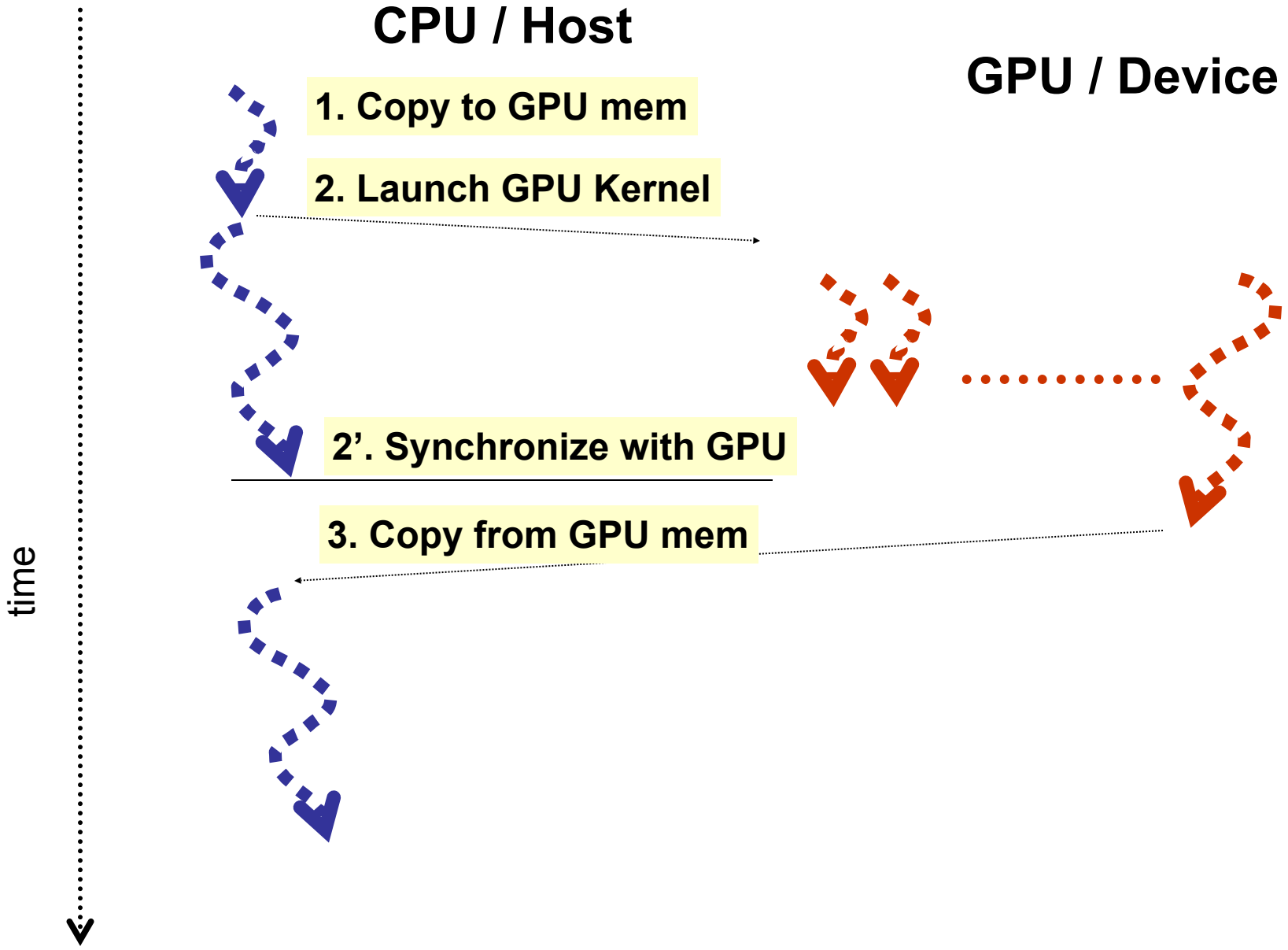
```
int main()
{
  float h_a[N]; /* allocate cpu container */
  for (int i=0; i < N; i++) h_a[i] = (float) i; /* initialize */

  float *d_a;
  cudaMalloc ((void **) &d_a, SIZE);


  cudaMemcpy (d_a, h_a, SIZE, cudaMemcpyHostToDevice));

  arradd <<< n_blocks, block_size >>> (d_a, 10.0, N);

  cudaThreadSynchronize ();
  cudaMemcpy (h_a, d_a, SIZE, cudaMemcpyDeviceToHost));
  CUDA_SAFE_CALL (cudaFree (d_a));
}
```

CPU

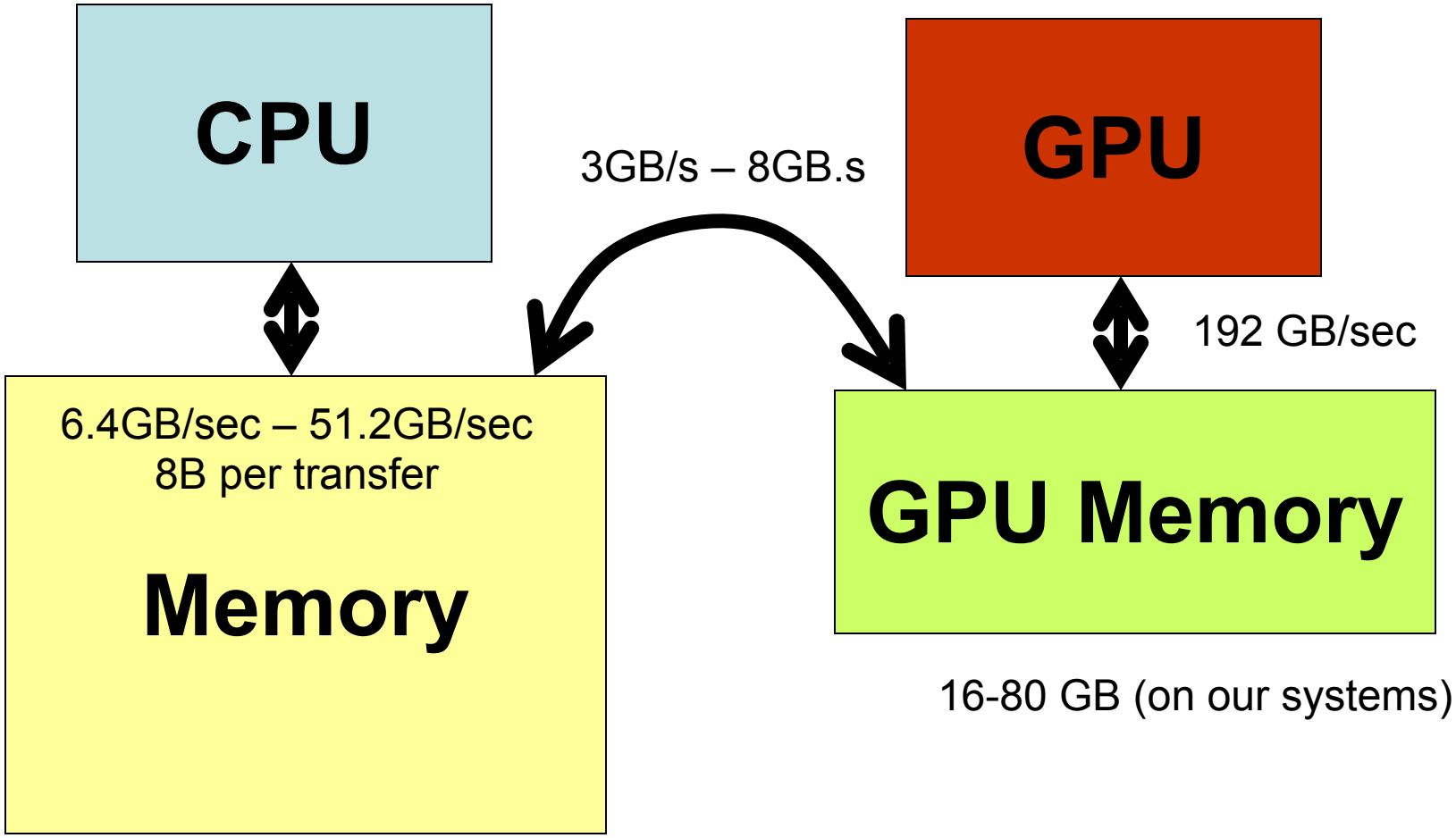
Why Use This Hierarchy?

- **Threads within a block** can share memory (good for local computations).
- **Blocks within a grid** execute independently (scales across multiple GPU cores).
- **Thread hierarchies** allow fine-grained control over memory access and performance optimization.

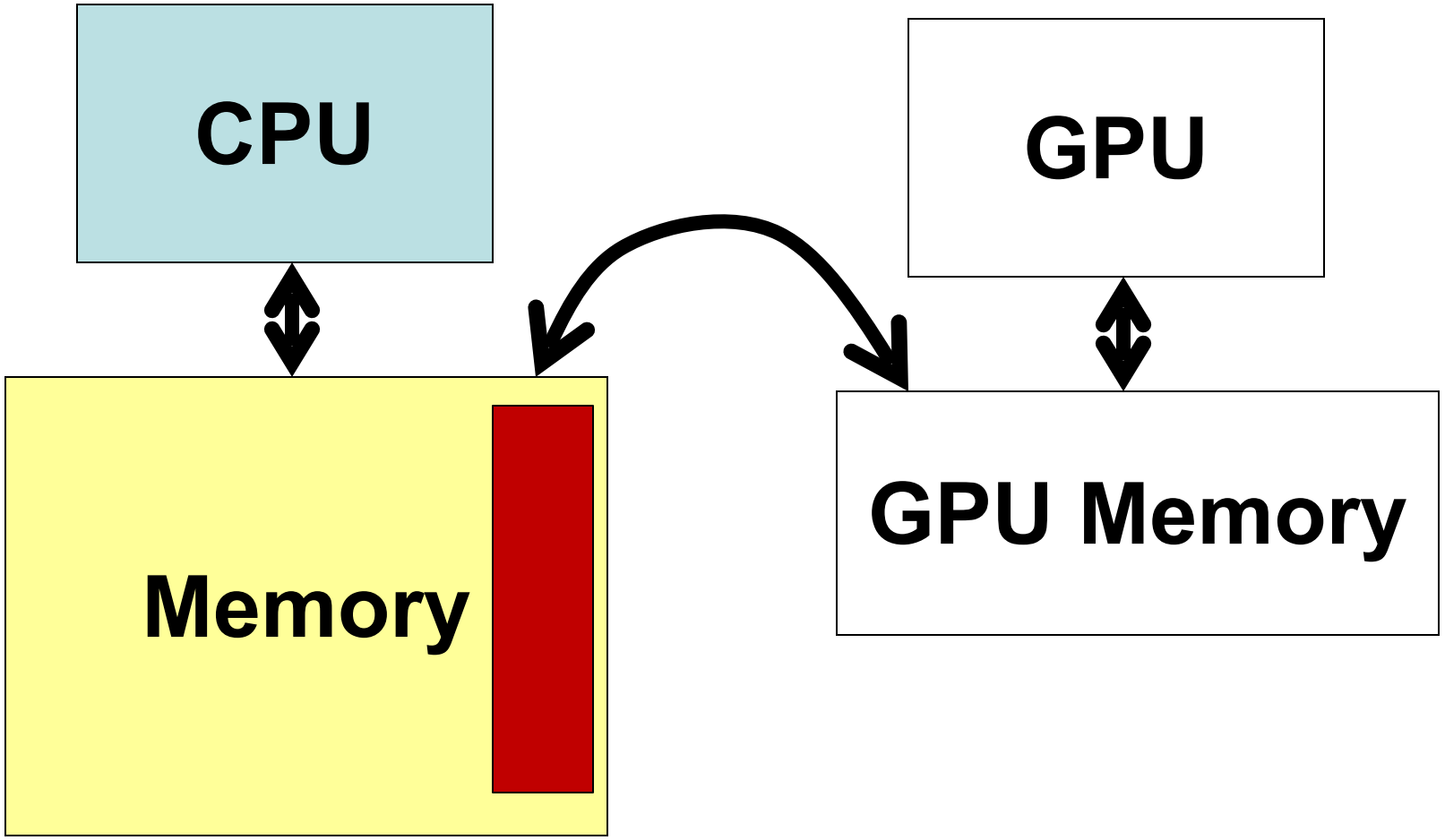
Execution Timeline



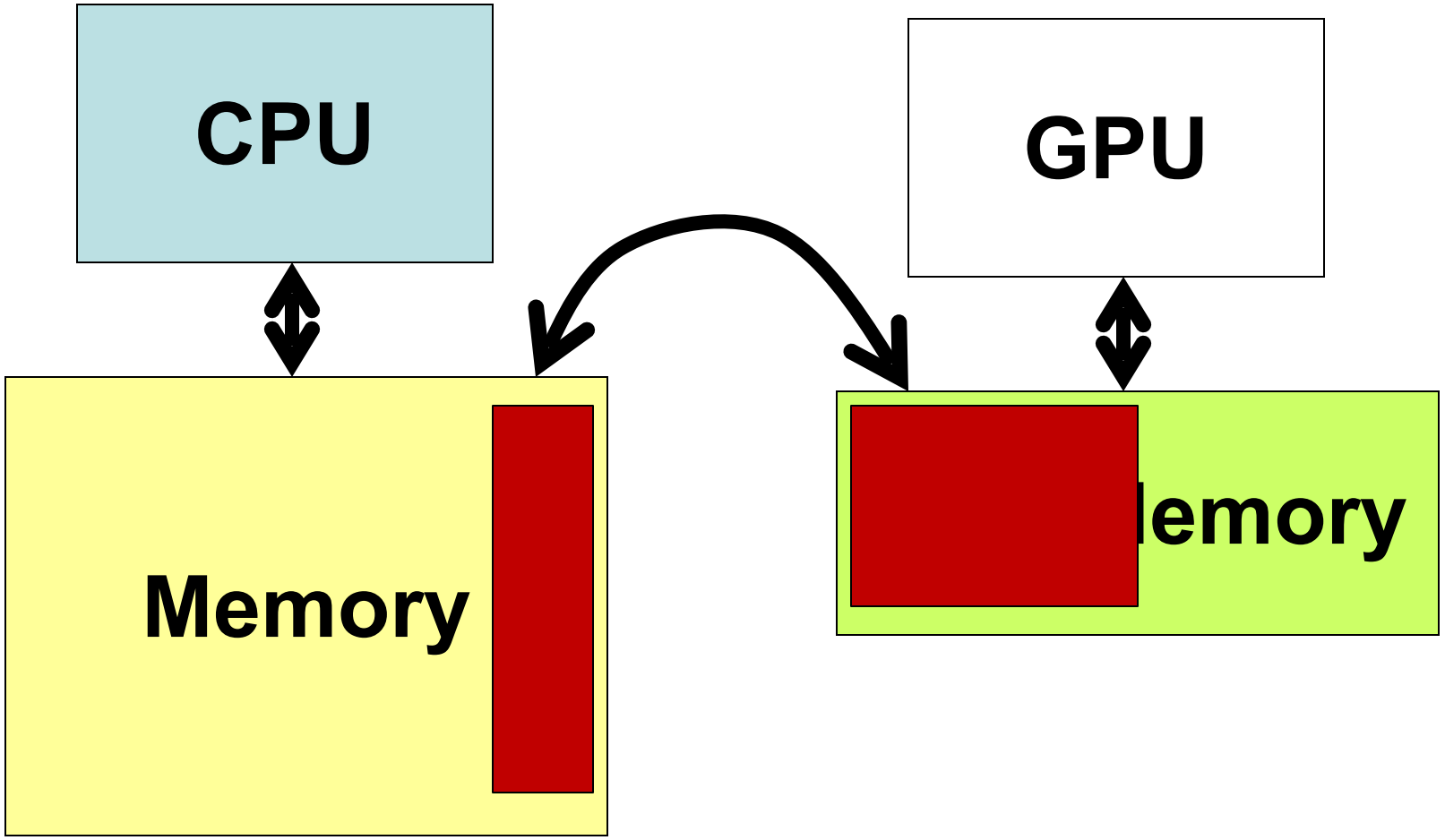
GPU as a co-processor



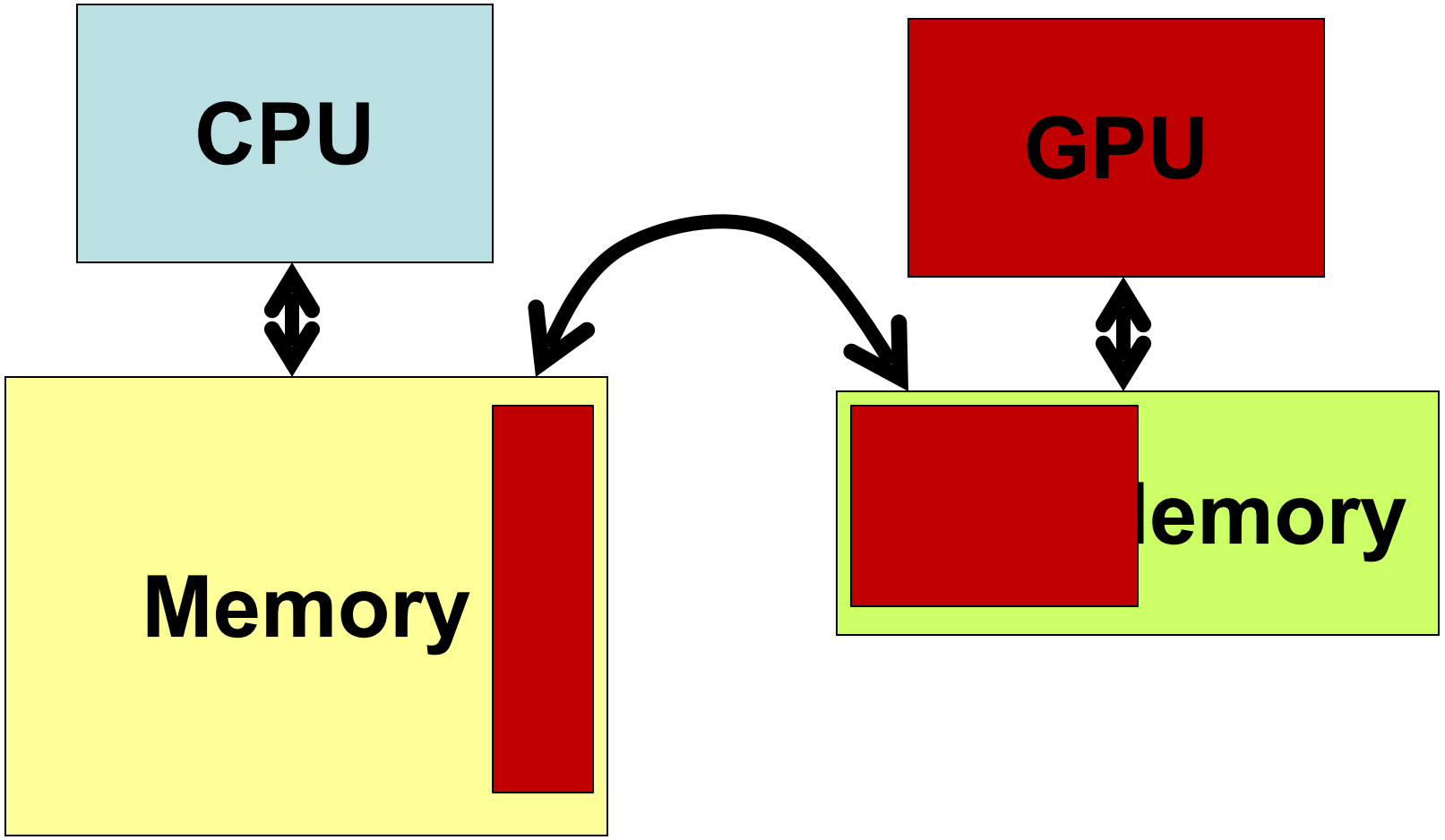
- First create data on CPU memory



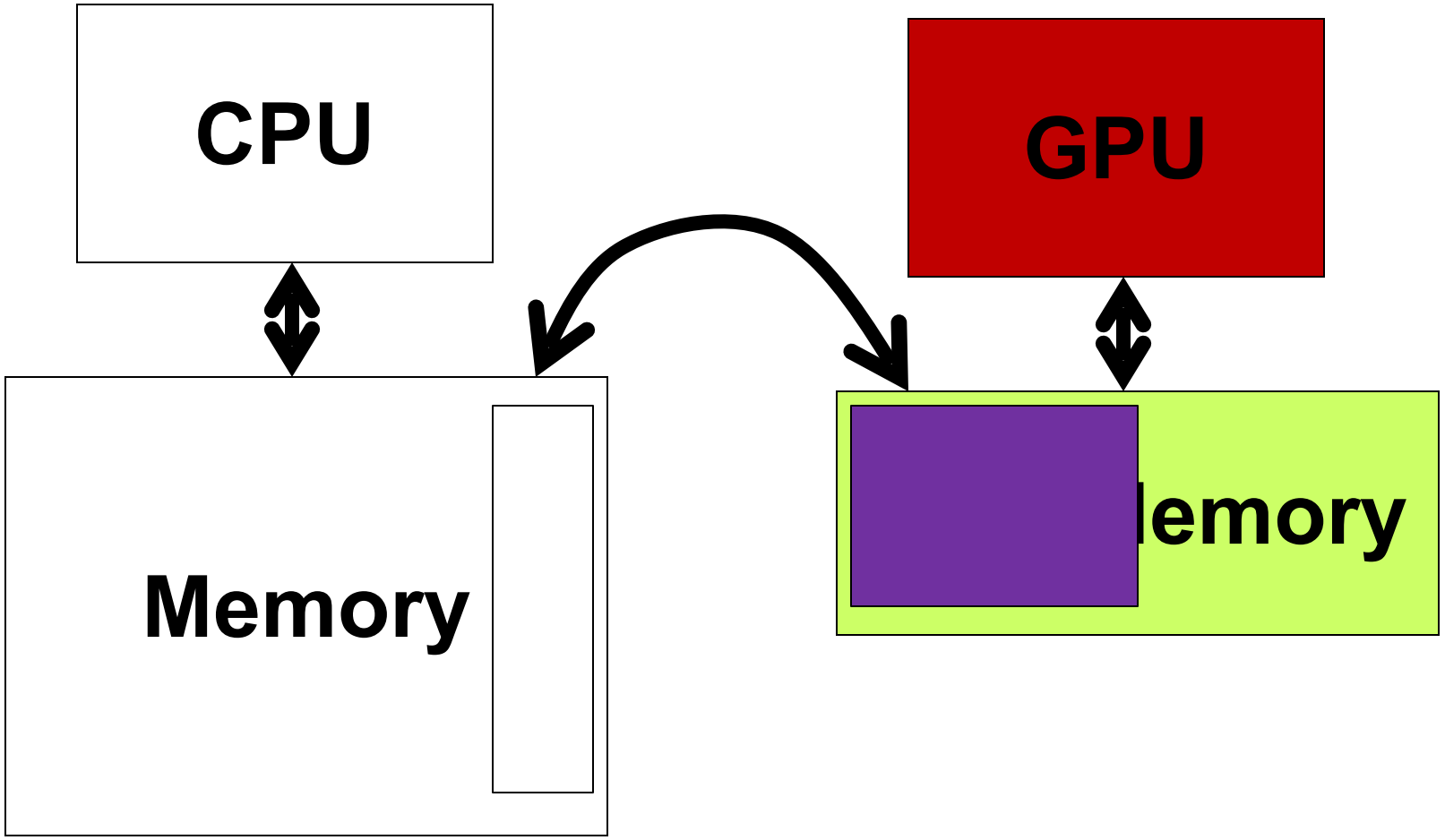
- Then Copy to GPU



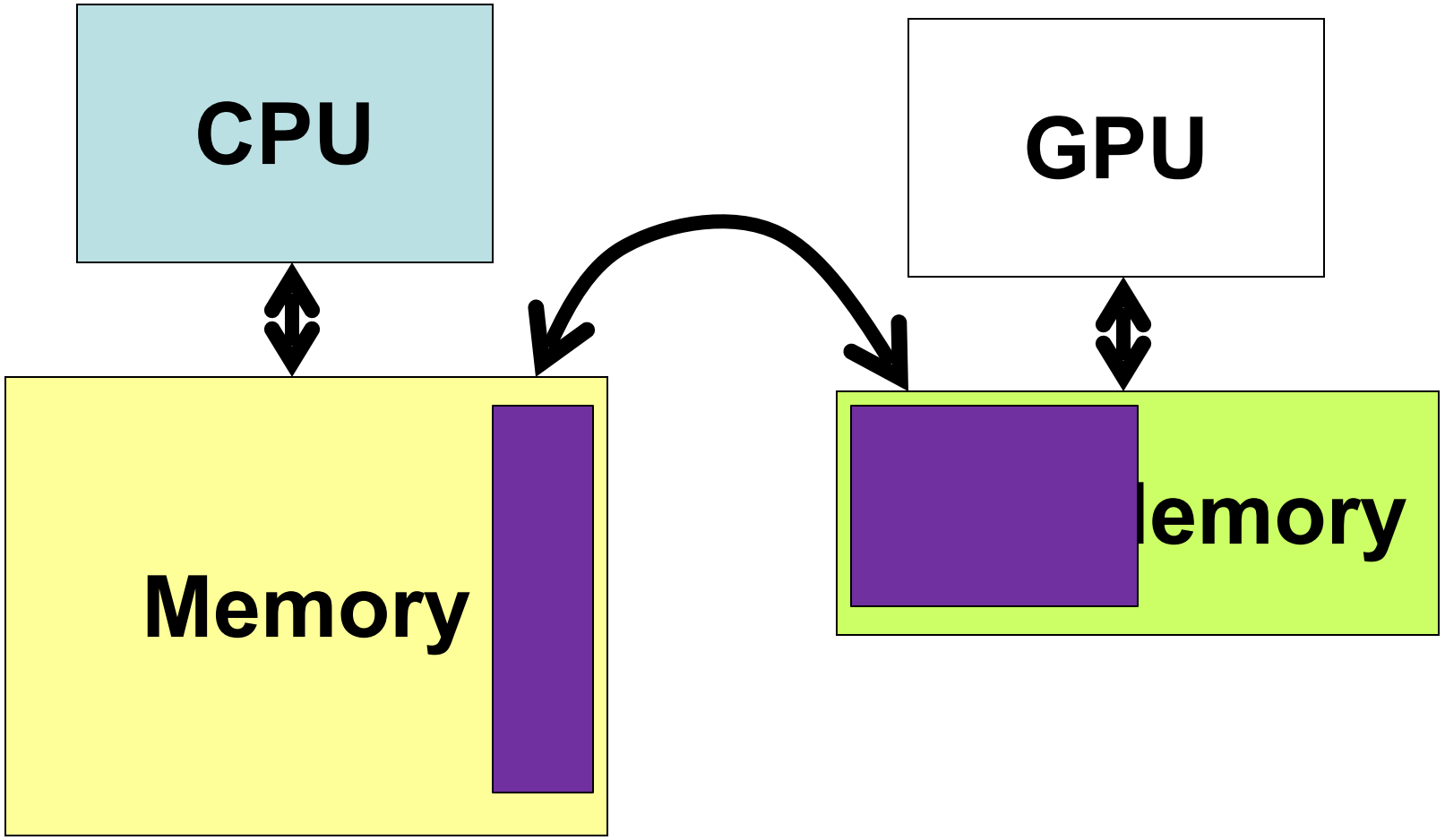
- GPU starts computation → runs a **kernel**
- CPU can also continue



- CPU and GPU Synchronize



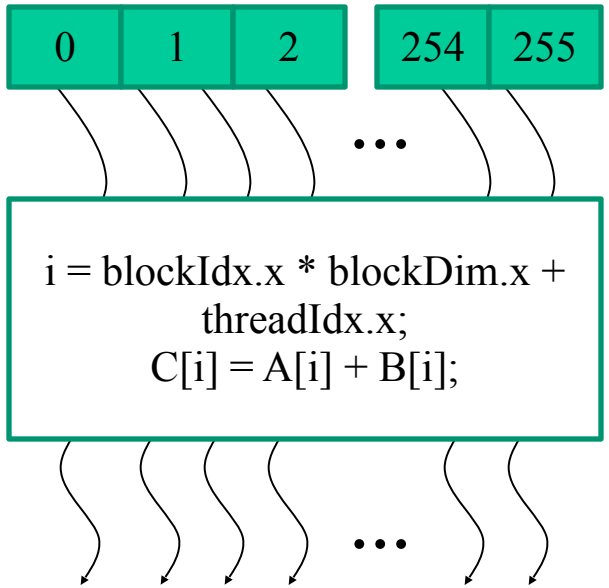
- Copy results back to CPU



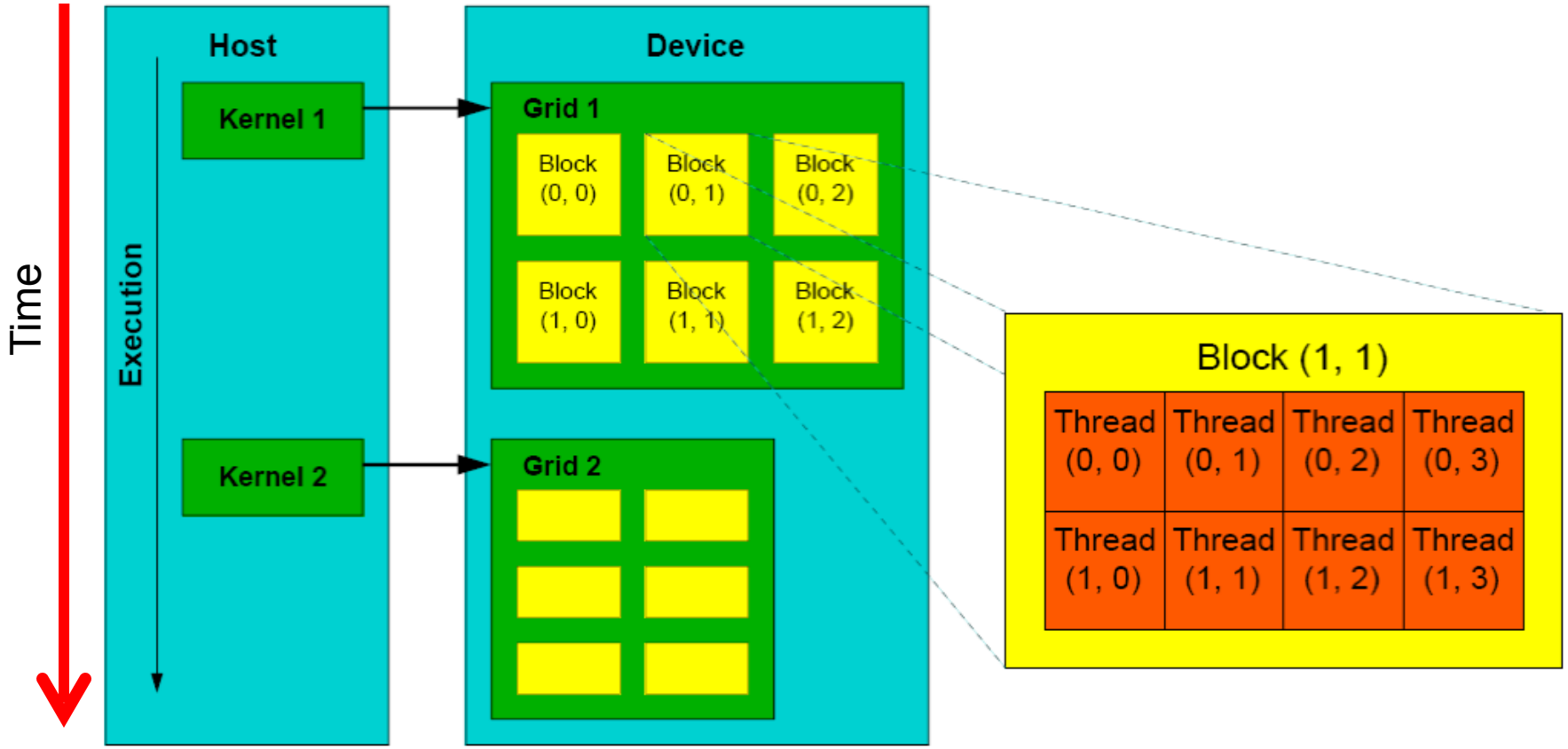
Arrays of Parallel Threads

A CUDA kernel is executed by a grid (array) of threads

- All **threads in a grid run the same kernel code** (SPMD)
- Each **thread has an index** that it uses to compute memory addresses and make control decisions



Programmers view of data and computation partitioning



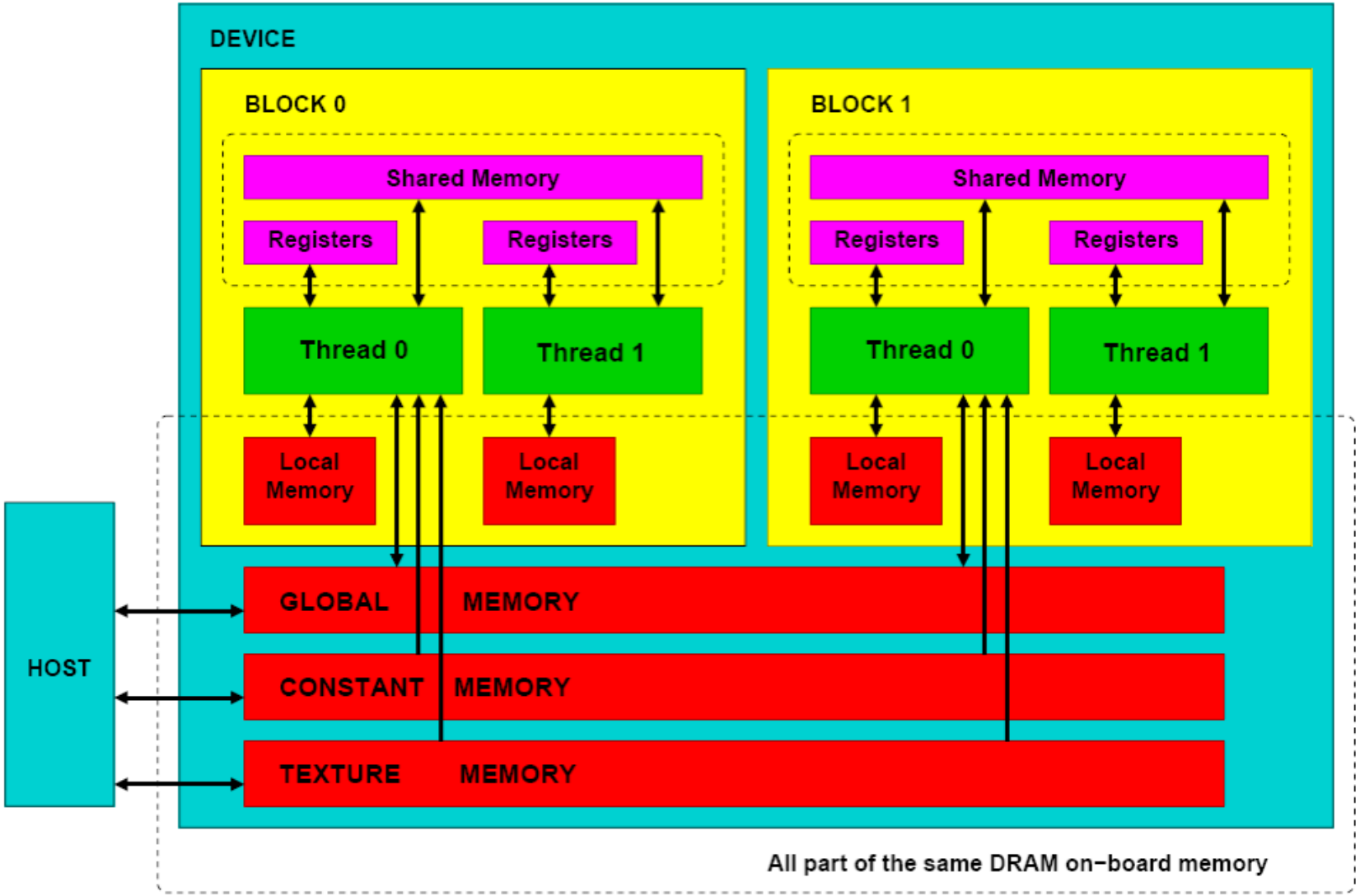
Why? Realities of integrated circuits:

need to cluster computation and storage to achieve high speeds

Philosophy is:

We'll tell you about the hardware – you figure out how to make the best of it

Programmer's view: Memory Model



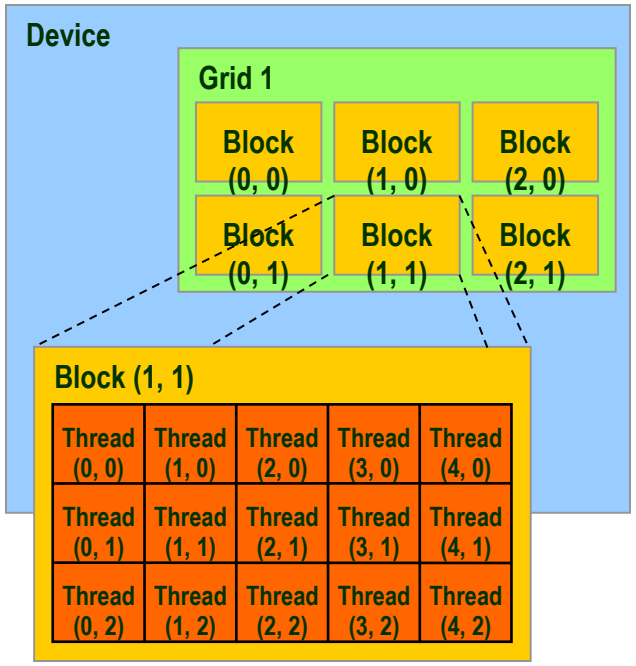
How is Texture Memory Different from Constant Memory?

Feature	Constant Memory	Texture Memory
Purpose	Storing small, frequently used constants	Storing spatially localized data (e.g., images, grids)
Access Type	Read-only from kernels	Read-only from kernels
Cached?	Yes, **constant cache**	Yes, **texture cache**
Access Pattern	Best for **all threads reading the same value**	Best for **spatially coherent access patterns**
Size Limit	**64 KB**	**Depends on global memory**
Binding	No need to bind, just declare as `__constant__`	Must bind memory using `cudaBindTexture()`
Interpolation Support	❌ No	✅ Yes, supports **hardware interpolation**
Best Use Cases	**Global constants** (e.g., physics constants, lookup tables)	**2D/3D images, scientific data, volumetric rendering**

- Grid of Blocks **1D, 2D, or 3D**
  - Max grid dimensions:
    2^31 - 1 x 65,535 x 65,535

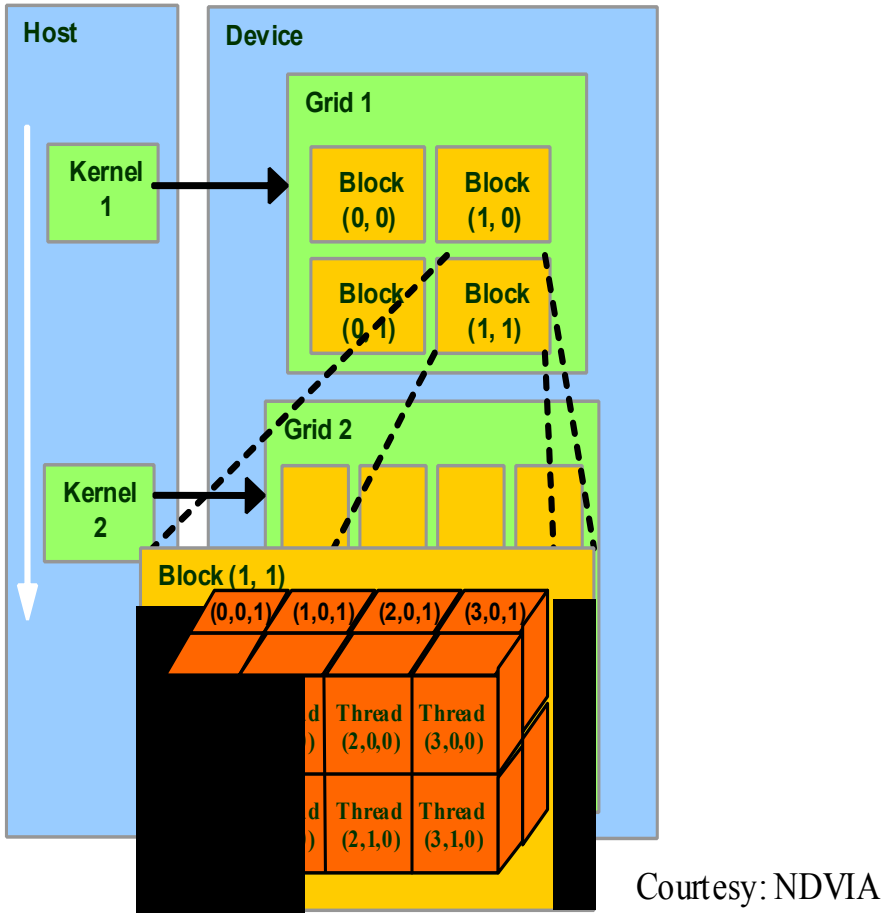
- Block of Threads: **1D, 2D, or 3D**
  - Max number of threads: **1024**
  - Max block dimension:
    1024 x 1024 x 64

- Limits apply to Compute Capability
  - **A100 = 8.0**
  - H100 = 9.0



- Threads and blocks have IDs
  - So each thread can decide what data to work on
  - Block ID: 1D, 2D, or 3D
  - Thread ID: 1D, 2D, or 3D
  - **Combination is unique**

- *Simplifies memory addressing when processing multidimensional data*
  - ***Convenience, not necessity***



Courtesy: NDVIA

- IDs and dimensions are accessible through predefined “variables”, e.g., blockDim.x and threadIdx.x

1. The CUDA Execution Model

CUDA organizes parallel computation using a hierarchy:

1. **Threads** – The smallest unit of execution.
2. **Thread Blocks** – A group of threads that execute together and share resources.
3. **Grid of Blocks** – A collection of thread blocks that form the entire computation.

Each level allows the GPU to scale execution across thousands of cores efficiently.

2. CUDA Thread Organization

(a) Threads and Thread Blocks

- Each **thread** executes an instance of a kernel (a CUDA function that runs on the GPU).
- A **thread block** is a collection of threads that execute together. Threads within a block:
  - Share **shared memory**.
  - Can synchronize using `__syncthreads()`.
  - Are identified by a **thread index** (`threadIdx`).

(b) Grids and Blocks

- A **grid** is a collection of **blocks**.
- Each block is identified using `blockIdx`.
- Blocks themselves contain multiple **threads**.

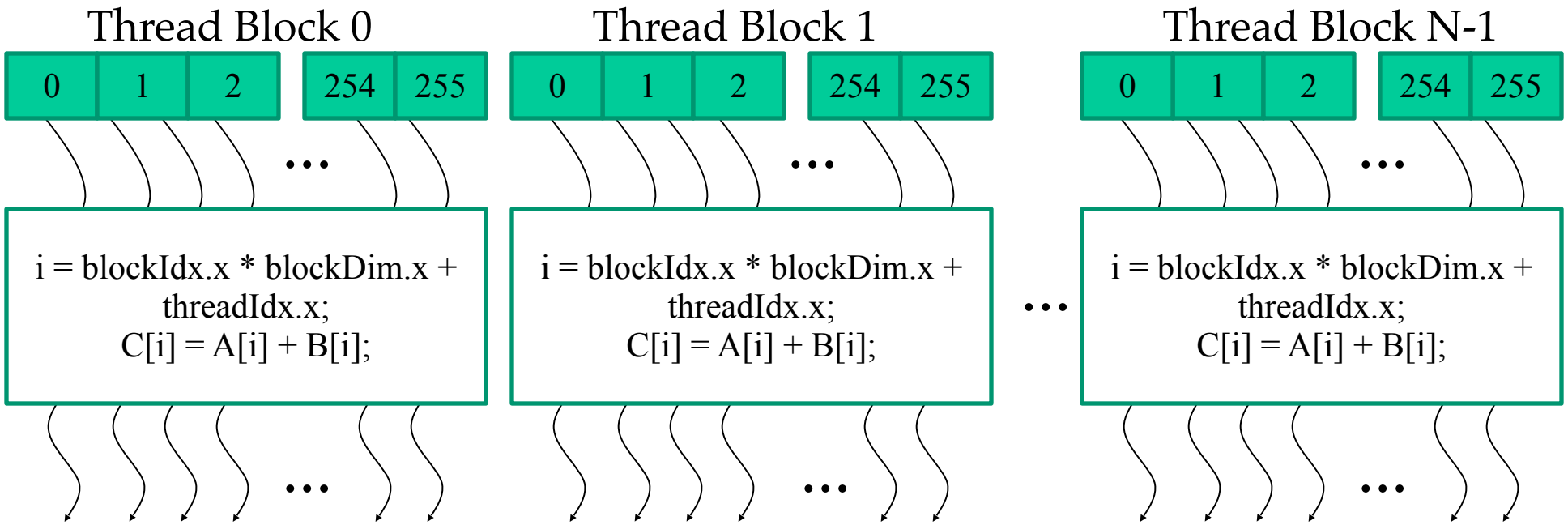
3. How CUDA Exposes This to Programmers

CUDA provides built-in variables to help programmers manage threads and blocks:

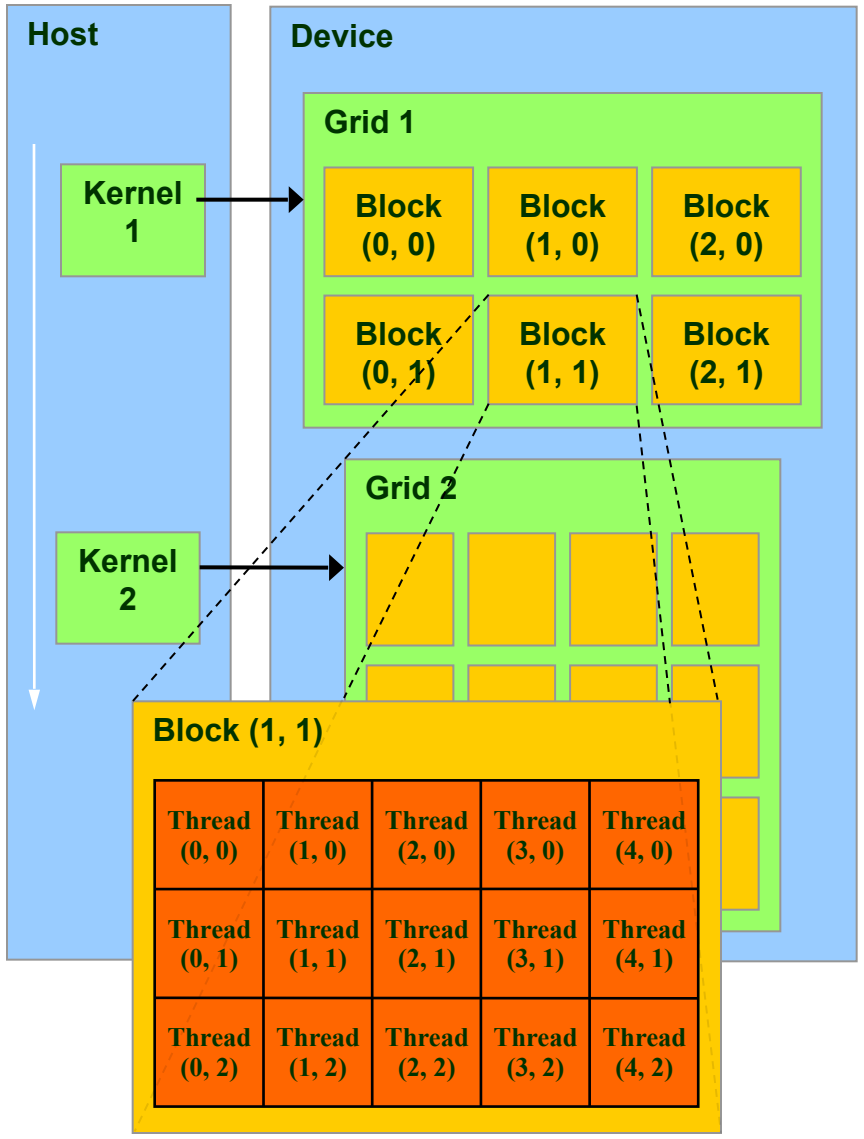
Variable	Description
`threadIdx.x`	Thread index within a block (1D)
`threadIdx.y`	Thread index in 2D
`threadIdx.z`	Thread index in 3D
`blockIdx.x`	Block index within a grid (1D)
`blockIdx.y`	Block index in 2D
`blockIdx.z`	Block index in 3D
`blockDim.x`	Number of threads per block in 1D
`blockDim.y`	Number of threads per block in 2D
`blockDim.z`	Number of threads per block in 3D
`gridDim.x`	Number of blocks in a grid (1D)
`gridDim.y`	Number of blocks in a grid (2D)
`gridDim.z`	Number of blocks in a grid (3D)

Thread Blocks: Scalable Cooperation

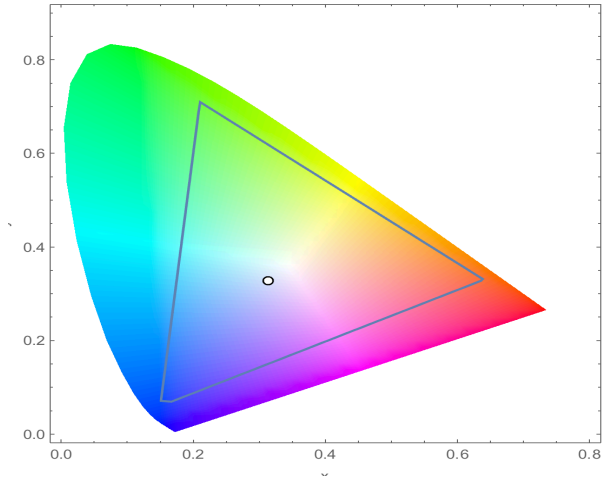
- Divide thread array into multiple blocks
  - Threads within a block cooperate via **shared memory, atomic operations** and **barrier synchronization**
  - Threads in different blocks cannot cooperate



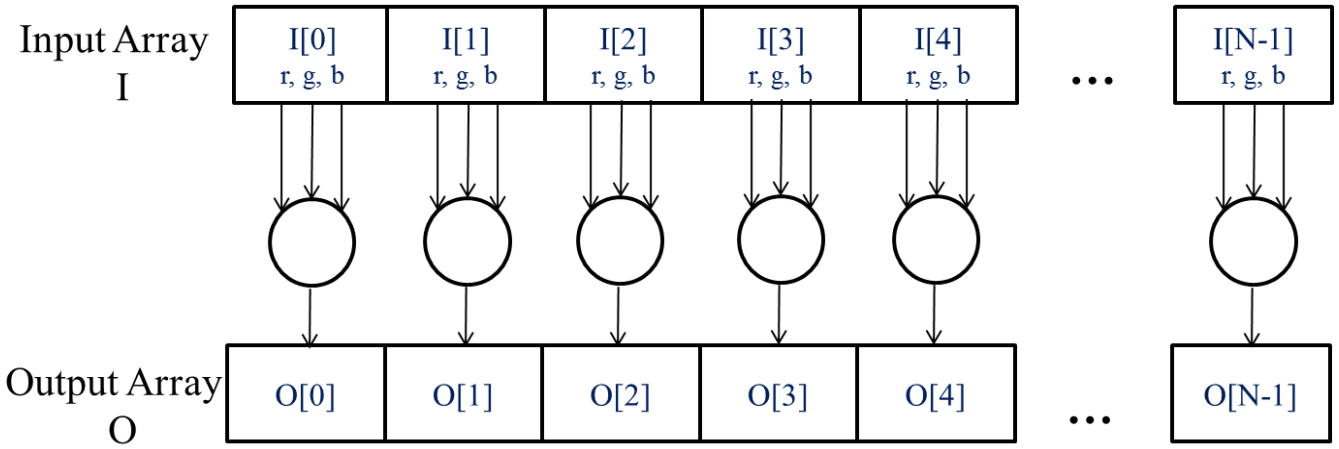
- A kernel is executed as a **grid of thread blocks**
  - All thread blocks **share** the same data memory space
    - But **cannot** communicate through it
- A **thread block:**
  - Threads that can cooperate with each other by:
    - Synchronizing their execution, for hazard-free shared memory accesses
    - Efficiently sharing data through a low latency shared memory
- Two threads from two different blocks cannot cooperate



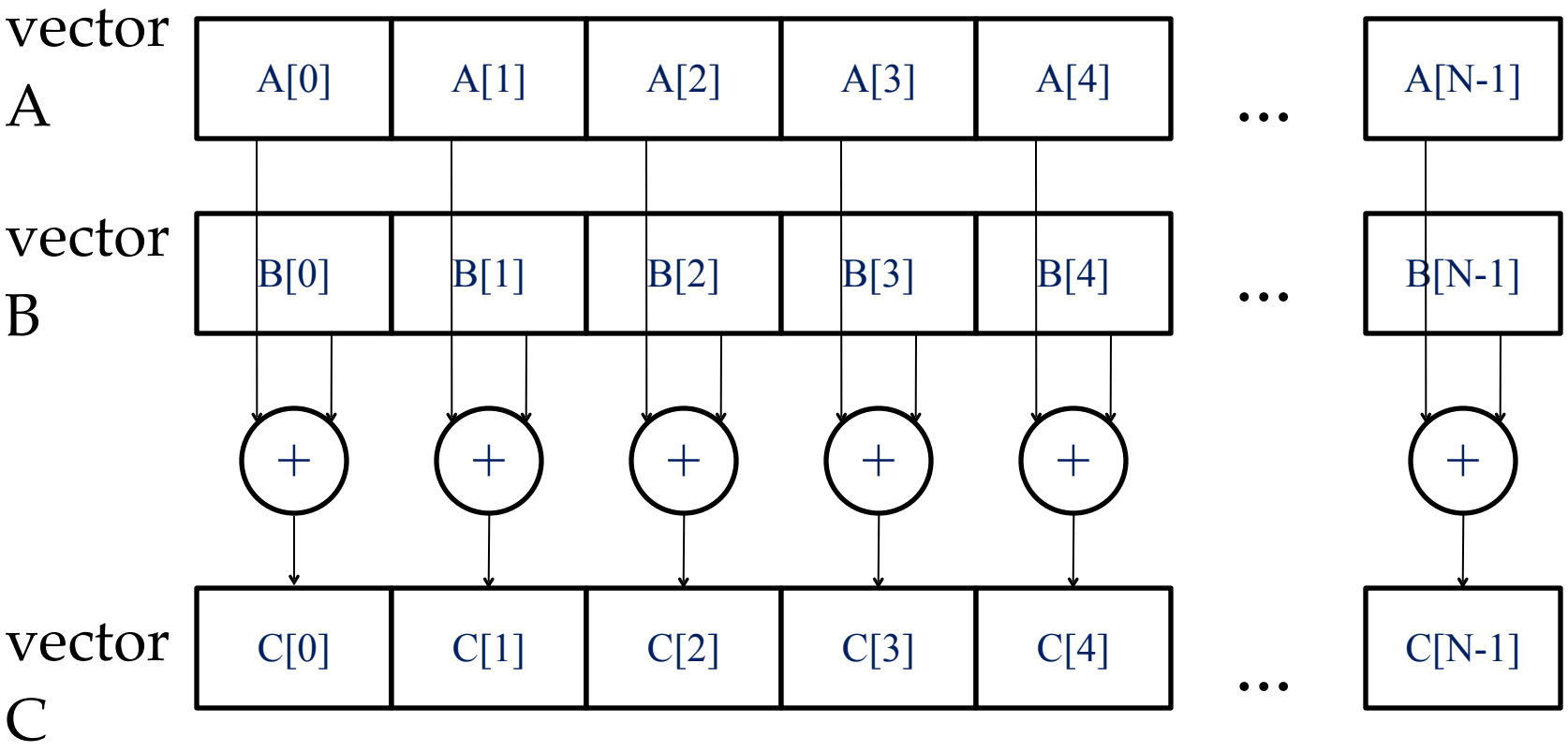
Conversion of a color image to grey–scale image



The pixels can be calculated independently of each other



Vector Addition – Conceptual View



Vector Addition – Traditional C Code

```
// Compute vector sum C = A+B
void vecAdd(float* A, float* B, float* C, int n)
{
  for (i = 0, i < n, i++)
    C[i] = A[i] + B[i];
}

int main()
{
    // Memory allocation for A_h, B_h, and C_h
    // I/O to read A_h and B_h, N elements
    …
    vecAdd(A_h, B_h, C_h, N);
}
```

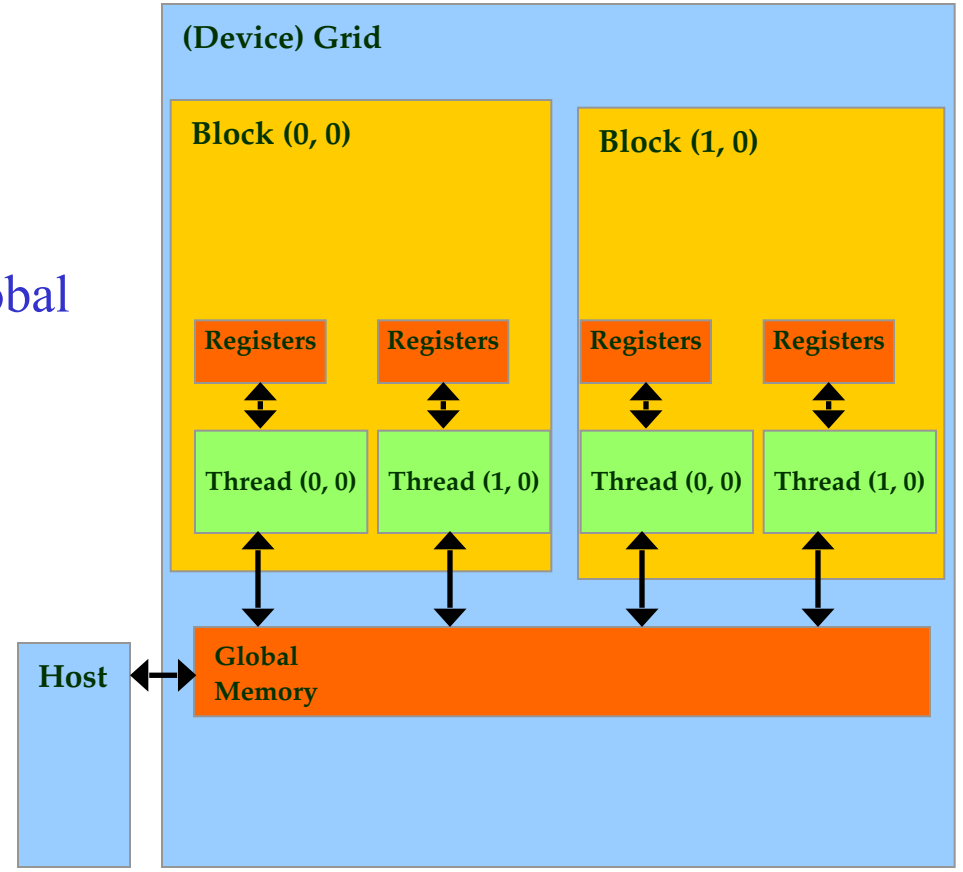
Heterogeneous Computing vecAdd Host Code

```
#include <cuda.h>
void vecAdd(float* A, float* B, float* C, int n)
{
    int size = n* sizeof(float);
   float* A_d, B_d, C_d;
   …
1. // Allocate device memory for A, B, and C
   // copy A and B to device memory

2. // Kernel launch code – to have the device
   // to perform the actual vector addition

3. // copy C from the device memory
   // Free device vectors
}
```

Partial Overview of CUDA Memories

- Device code can:
  - R/W per-thread registers
  - R/W per-grid global memory

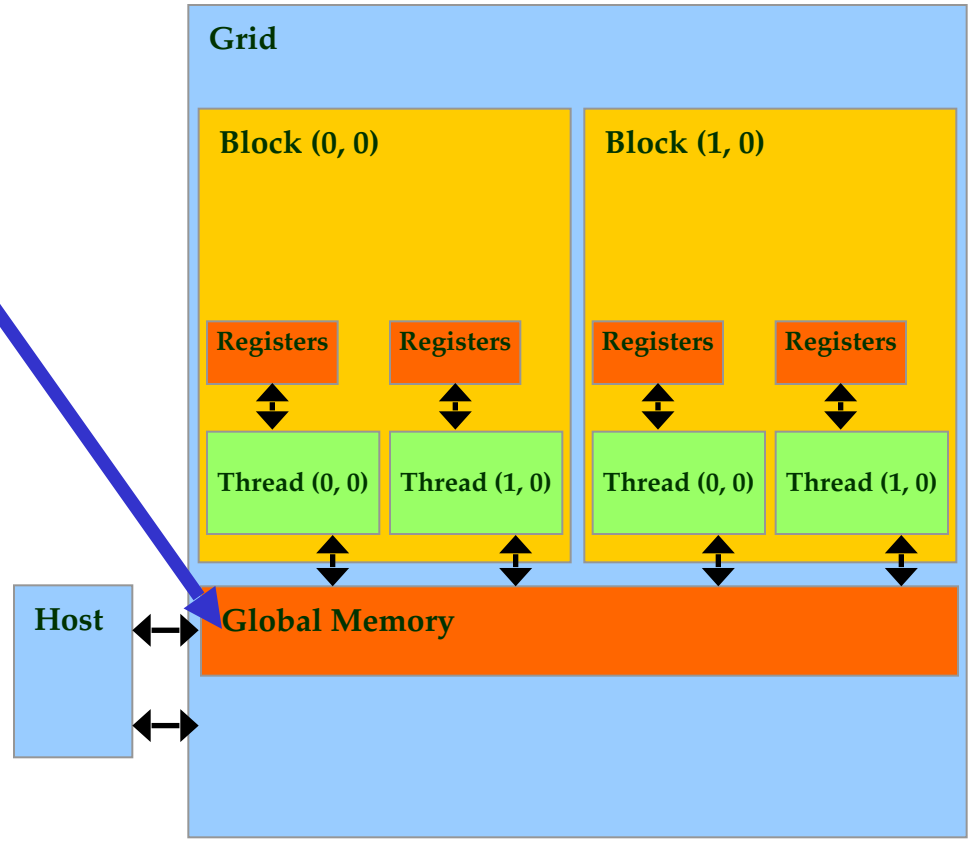
- Host code can
  - Transfer data to/from per grid global memory



- Each thread can:
  - R/W per-thread registers
  - R/W per-thread local memory
  - R/W per-block shared memory
  - R/W per-grid global memory
  - Read only per-grid constant memory
  - Read only per-grid texture memory
- The host can R/W:
  - global, constant, and texture memories

CUDA Device Memory Management API functions

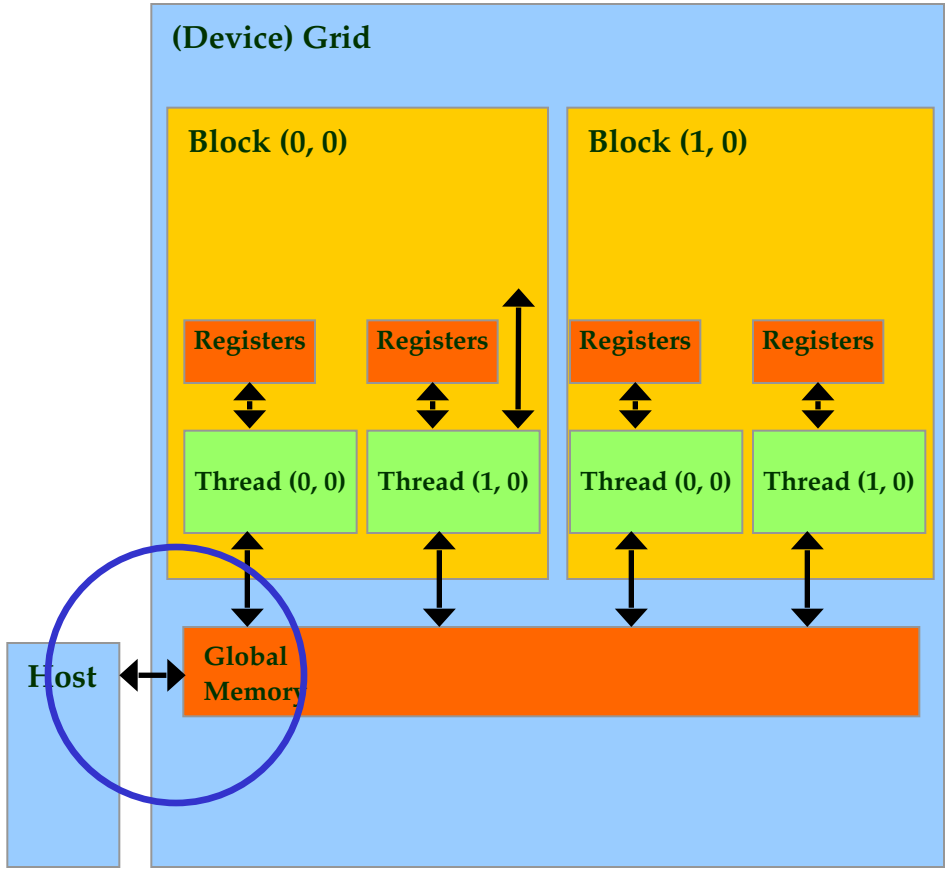
- cudaMalloc()
  - Allocates object in the device global memory
  - Two parameters
    - **Address of a pointe**r to the allocated object
    - **Size of** of allocated object in terms of bytes
- cudaFree()
  - Frees object from device global memory
    - **Pointer** to freed object



Host-Device Data Transfer API functions

cudaMemcpy()

- memory data transfer
- Requires four parameters
  - Pointer to destination
  - Pointer to source
  - Number of bytes copied
  - Type/Direction of transfer
- Transfer to device is synchronous



Copy Initialized CPU data to GPU

```
float *da;
float *ha;

cudaMemCpy ((void *) da,                    // DESTINATION
            (void *) ha,                    // SOURCE
            sizeof (float) * N,             // #bytes
            cudaMemcpyHostToDevice);        // DIRECTION
```

- The host initiates all transfers:
- **cudaMemcpy**(void *dst, void *src,
  size_t nbytes,
  enum cudaMemcpyKind direction)

- enum cudaMemcpyKind
  - cudaMemcpy**Host**To**Device**
  - cudaMemcpy**Device**To**Host**
  - cudaMemcpy**Device**To**Device**

```
void vecAdd(float* A, float* B, float* C, int n)
{
    int size = n * sizeof(float);
     float* A_d, B_d, C_d;

1. // Transfer A and B to device memory
     cudaMalloc((void **) &A_d, size);
     cudaMemcpy(A_d, A, size, cudaMemcpyHostToDevice);
     cudaMalloc((void **) &B_d, size);
     cudaMemcpy(B_d, B, size, cudaMemcpyHostToDevice);

    // Allocate device memory for
     cudaMalloc((void **) &C_d, size);

2. // Kernel invocation code – to be shown later
      …
3. // Transfer C from device to host
     cudaMemcpy(C, C_d, size, cudaMemcpyDeviceToHost);
    // Free device memory for A, B, C
     cudaFree(A_d); cudaFree(B_d); cudaFree (C_d);
}
```

Example: Vector Addition Kernel

Device Code

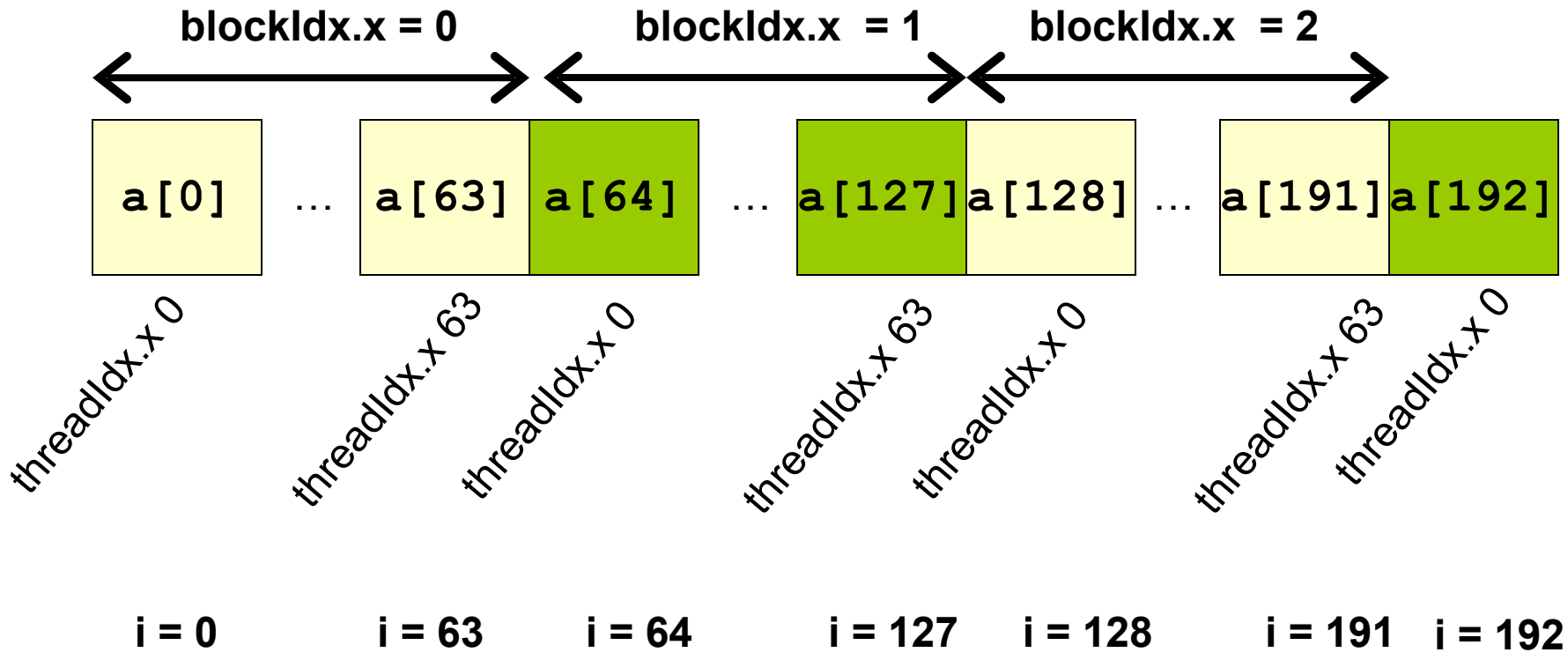
```
// Compute vector sum C = A+B
// Each thread performs one pair-wise addition
__global__
void vecAddKernel(float* A_d, float* B_d, float* C_d, int n)
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    if(i<n) C_d[i] = A_d[i] + B_d[i];
}

int vectAdd(float* A, float* B, float* C, int n)
{
        // A_d, B_d, C_d allocations and copies omitted
    // Run ceil(n/256) blocks of 256 threads each
    vecAddKernel<<<ceil(n/256.0), 256>>>(A_d, B_d, C_d, n);
}
```

```
int i = threadIdx.x + blockDim.x * blockIdx.x;
```

- **BlockIdx:** Unique Block ID.
  - Numerically asceding: 0, 1, …
- **BlockDim:** Dimensions of Block = how many threads it has
  - BlockDim.x, BlockDim.y, BlockDim.z
  - Unused dimensions default to 0
- **ThreadIdx:** Unique per Block Index
  - 0, 1, …
  - Per Block

Array Index Calculation Example

```
int i = blockIdx.x * blockDim.x + threadIdx.x;
```



Assuming blockDim.x = 64

- **1D Grid / 1D Blocks:**

```
UniqueBlockIndex = blockIdx.x;
UniqueThreadIndex = blockIdx.x * blockDim.x + threadIdx.x;
```

- **1D Grid / 2D Blocks:**

```
UniqueBlockIndex = blockIdx.x;
UniqueThreadIndex = blockIdx.x * blockDim.x * blockDim.y + threadIdx.y *
blockDim.x + threadIdx.x;
```

- **1D Grid / 3D Blocks:**

```
UniqueBockIndex = blockIdx.x;
UniqueThreadIndex = blockIdx.x * blockDim.x * blockDim.y * blockDim.z +
threadIdx.z * blockDim.y * blockDim.x + threadIdx.y * blockDim.x +
threadIdx.x;
```

- More options: https://www.eecs.umich.edu/courses/eecs498-APP/resources/materials/CUDA-Thread-Indexing-Cheatsheet.pdf

- **2D Grid / 1D Blocks:**

```
UniqueBlockIndex = blockIdx.y * gridDim.x + blockIdx.x;
UniqueThreadIndex = UniqueBlockIndex * blockDim.x + threadIdx.x;
```

- **2D Grid / 2D Blocks:**

```
UniqueBlockIndex = blockIdx.y * gridDim.x + blockIdx.x;
UniqueThreadIndex =UniqueBlockIndex * blockDim.y * blockDim.x + threadIdx.y *
blockDim.x + threadIdx.x;
```

- **2D Grid / 3D Blocks:**

```
UniqueBlockIndex = blockIdx.y * gridDim.x + blockIdx.x;
UniqueThreadIndex = UniqueBlockIndex * blockDim.z * blockDim.y * blockDim.x +
threadIdx.z * blockDim.y * blockDim.z + threadIdx.y * blockDim.x +
threadIdx.x;
```

- **UniqueThreadIndex means unique per grid.**

Example: Vector Addition Kernel - Launch

```
// Compute vector sum C = A+B
// Each thread performs one pair-wise addition
__global__
void vecAddkernel(float* A_d, float* B_d, float* C_d, int n)
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    if(i<n) C_d[i] = A_d[i] + B_d[i];
}

int vecAdd(float* A, float* B, float* C, int n)
{
 // A_d, B_d, C_d allocations and copies omitted
 // Run ceil(n/256) blocks of 256 threads each
  vecAddKernnel<<<ceil(n/256.0),256>>>(A_d, B_d, C_d, n);
}
```

Host Code

CUDA supports **1D, 2D, and 3D** thread/block structures. For example:

cpp Copy Edit

```cpp
dim3 blocks(2, 2);    // 2x2 grid of blocks
dim3 threads(4, 4);   // Each block has 4x4 threads
kernelExample<<<blocks, threads>>>();
```

Threads are then identified as:

cpp Copy Edit

```cpp
int x = threadIdx.x + blockIdx.x * blockDim.x;
int y = threadIdx.y + blockIdx.y * blockDim.y;
```

- A kernel function must be called with an execution configuration:

```
dim3   DimGrid(100, 50);     // 5000 thread blocks
dim3   DimBlock(4, 8, 8);    // 256 threads per block
size_t SharedMemBytes = 64; // 64 bytes of shared memory
KernelFunc<<< DimGrid, DimBlock, SharedMemBytes >>>(...);
```

```
void vecAdd(float* A, float* B, float* C, int n)
{
    int size = n * sizeof(float);
     float* A_d, B_d, C_d;

1. // Transfer A and B to device memory
     cudaMalloc((void **) &A_d, size);
     cudaMemcpy(A_d, A, size, cudaMemcpyHostToDevice);
     cudaMalloc((void **) &B_d, size);
     cudaMemcpy(B_d, B, size, cudaMemcpyHostToDevice);

    // Allocate device memory for
     cudaMalloc((void **) &C_d, size);

2. // Kernel invocation code – to be shown later
      …
3. // Transfer C from device to host
     cudaMemcpy(C, C_d, size, cudaMemcpyDeviceToHost);
    // Free device memory for A, B, C
     cudaFree(A_d); cudaFree(B_d); cudaFree (C_d);
}
```

GPU Software Stack Overview

1. **Application Layer (High-Level Code)**
    - CUDA C/C++, Fortran, Python (with libraries like Numba, TensorFlow, PyTorch)
    - OpenCL, HIP (AMD's CUDA alternative)
    - DirectCompute, Vulkan, OpenGL Compute Shaders
2. **Compiler & Intermediate Representation (IR)**
    - **NVCC (NVIDIA CUDA Compiler)**: Compiles CUDA code into PTX
    - **PTX (Parallel Thread Execution) Assembly**:
        - A virtual instruction set that abstracts away GPU hardware details.
        - Allows compatibility across different NVIDIA GPU architectures.
        - Can be JIT (Just-In-Time) compiled to **SASS** (machine code) by the CUDA driver.
    - **LLVM-based Compiler (for OpenCL & HIP)**

3. **Driver & Runtime Layer**
    - **CUDA Runtime API** (e.g., `cudaMalloc`, `cudaMemcpy`, `cudaLaunchKernel`)
    - **CUDA Driver API** (lower-level, more explicit control)
    - **NVIDIA GPU Drivers** (convert PTX to SASS and manage hardware execution)
    - OpenCL Runtime (for OpenCL-based GPU programs)
4. **Hardware Execution Layer**
    - **SASS (Streaming Assembly)**
        - Machine code specific to GPU architecture (e.g., Ampere, Ada, Hopper).
        - Produced from PTX by JIT compilation in the driver.
    - **GPU Hardware (SMs, Tensor Cores, Memory Controllers)**
        - Executes the final machine instructions.

Compiling A CUDA Program