Software Performance Engineering

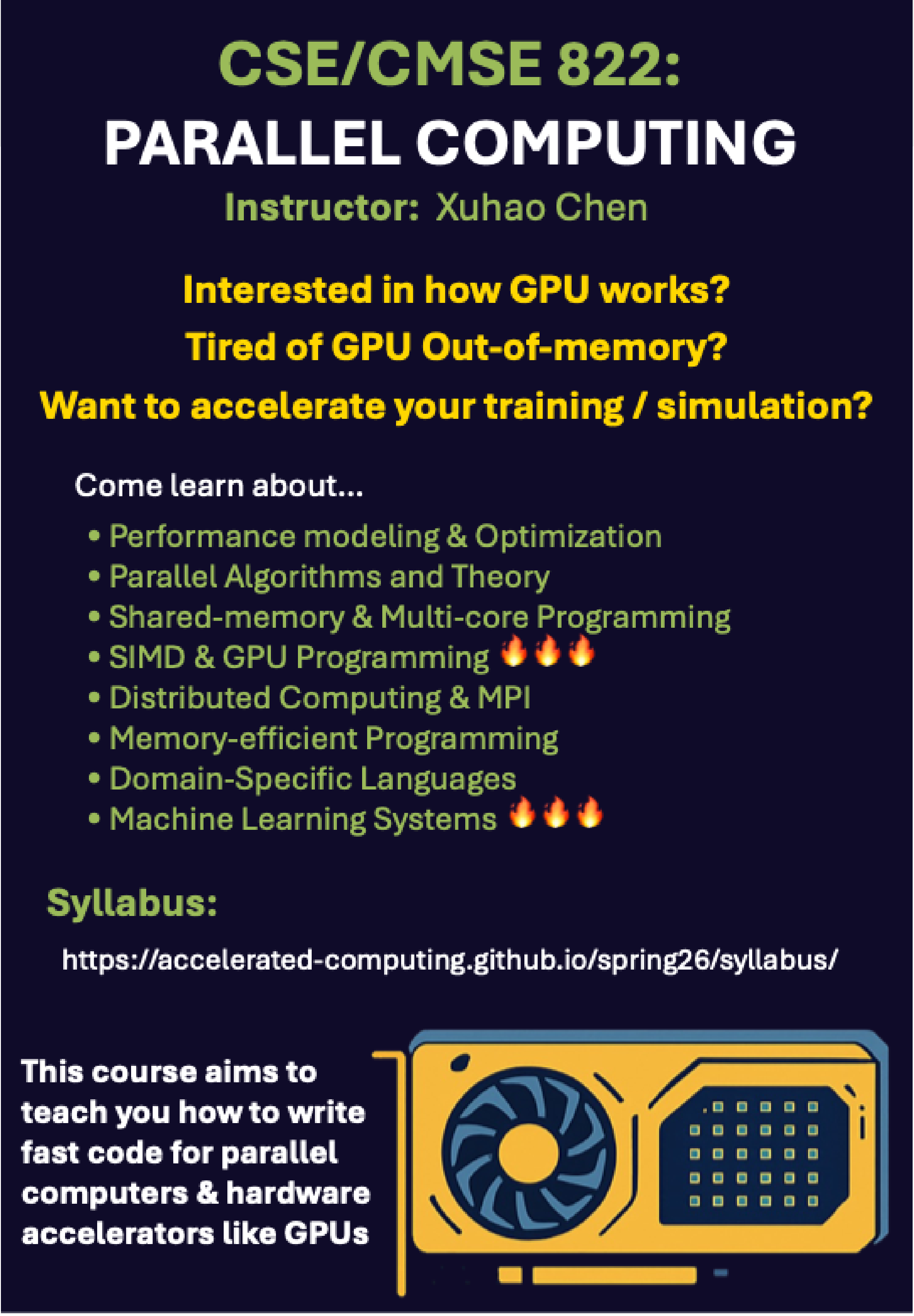
LECTURE 19
GPU PROGRAMMING

Xuhao Chen

November 6, 2025

CSE 822

- https://accelerated-computing.github.io/
- Spring 2026!
- Advanced GPU programming!
- Let me know if you’re in the waitlist



CSE/CMSE 822: PARALLEL COMPUTING

Instructor: Xuhao Chen

Interested in how GPU works?

Tired of GPU Out-of-memory?

Want to accelerate your training / simulation?

Come learn about…

- Performance modeling & Optimization
- Parallel Algorithms and Theory
- Shared-memory & Multi-core Programming
- SIMD & GPU Programming 🔥🔥🔥
- Distributed Computing & MPI
- Memory-efficient Programming
- Domain-Specific Languages
- Machine Learning Systems 🔥🔥🔥

Syllabus:

https://accelerated-computing.github.io/spring26/syllabus/

This course aims to teach you how to write fast code for parallel computers & hardware accelerators like GPUs

Outline

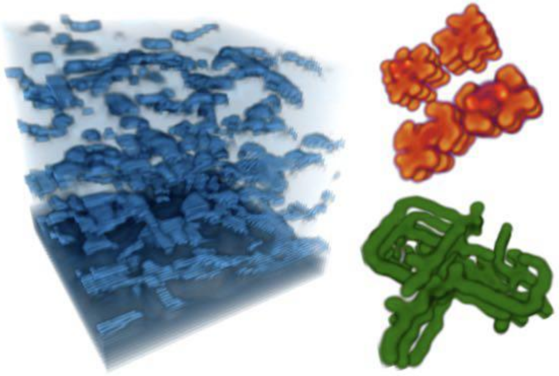
- GPU Architecture
  - Three major ideas that make GPU processing cores run fast
  - Closer look at real GPU designs
  - The GPU memory hierarchy: moving data to processors

- General Purpose GPU (GPGPU)

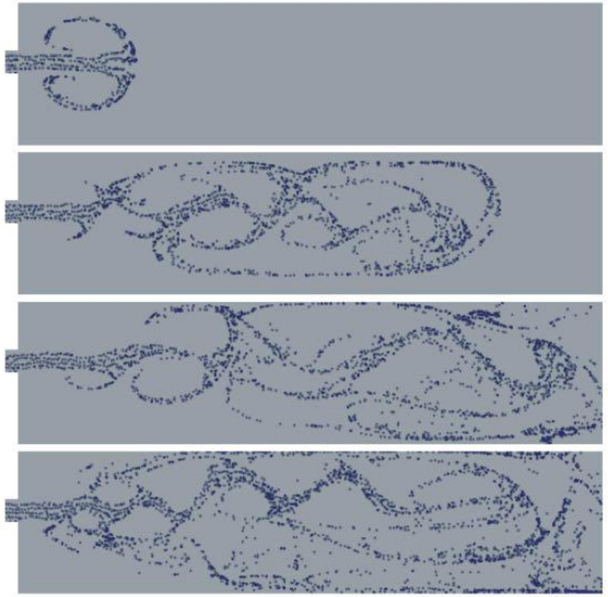
- GPU Programming Model

- Program a GPU with CUDA

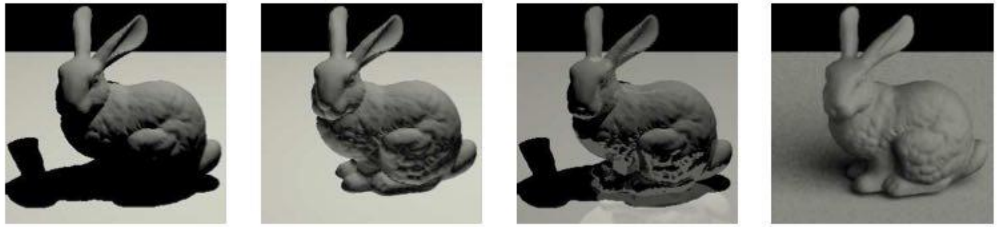
“GPGPU” 2002~2003



Coupled Map Lattice Simulation [Harris 02]



Sparse Matrix Solvers [Bolz 03]



Ray Tracing on Programmable Graphics Hardware [Purcell 02]

Brook stream programming language (2004)

- Stanford graphics lab research project*
- Abstract GPU hardware as data-parallel processor
- Brook compiler translates generic stream program into graphics commands (e.g. drawTriangles) and a set of shader programs

```
kernel void scale(float amount, float a<>, out float b<>) {
       b = amount * a;
}
float scale_amount;
float input_stream<1000>; // stream declaration
float output_stream<1000>; // stream declaration
// omitting stream element initialization...
// map kernel function onto streams
scale(scale_amount, input_stream, output_stream);
```

* Buck, Ian, et al. "Brook for GPUs: stream computing on graphics hardware." *ACM transactions on graphics (TOG)* 23.3 (2004): 777-786.

NVIDIA Tesla architecture (2007)

- First alternative, non-graphics-specific ("compute mode") interface to GPU hardware, e.g. GeForce 8800

- Let's say a user wants to run a non-graphics program on the GPU's programmable cores…
  - Application can allocate buffers in GPU memory and copy data to/from buffers
  - Application (via graphics driver) provides GPU a single kernel program binary
  - Application tells GPU to run the kernel in an SPMD fashion ("run N instances of this kernel") e.g. `launch(myKernel, N)`

- Interestingly, this is a far simpler operation than the graphics operation `drawPrimitives()`

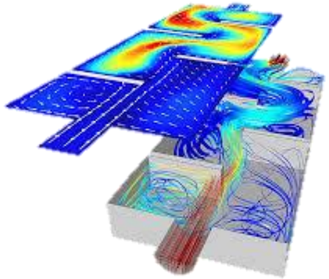
CUDA programming language

- Introduced in 2007 with NVIDIA Tesla architecture
- "C-like" language to express programs that run on GPUs using the compute-mode hardware interface
- Relatively low-level abstractions: closely match the capabilities/performance characteristics of modern GPUs

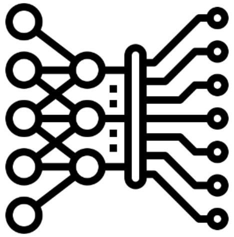
Compute Intensive Applications

- Bioinformatics

- Computational Fluid Dynamics



- Computational Chemistry
- AI & Machine Learning



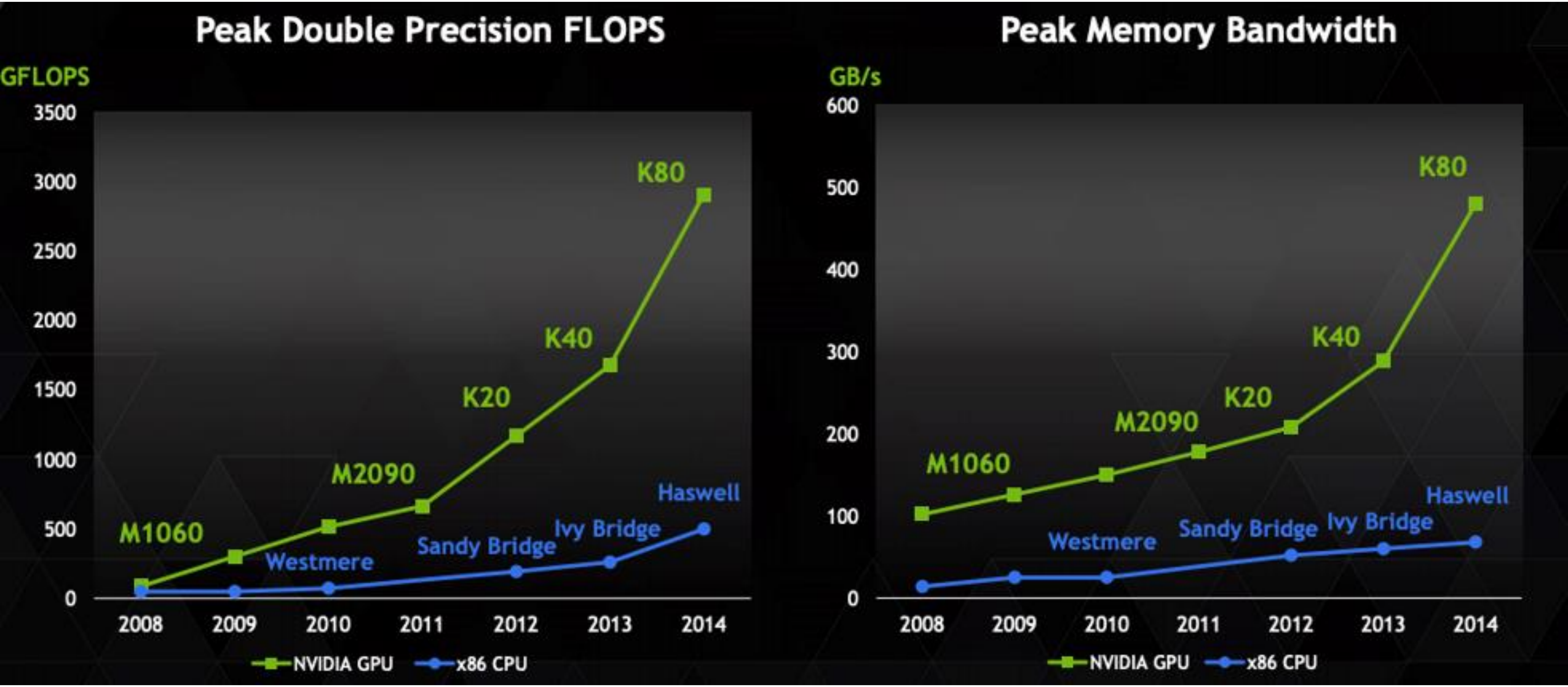
- Computer Vision
- Block Chain

Data Parallel Applications

- Computational Finance

- Data science, medical imaging,, weather and climate, ...

Why use GPGPU?



Why can't CPUs Scale?

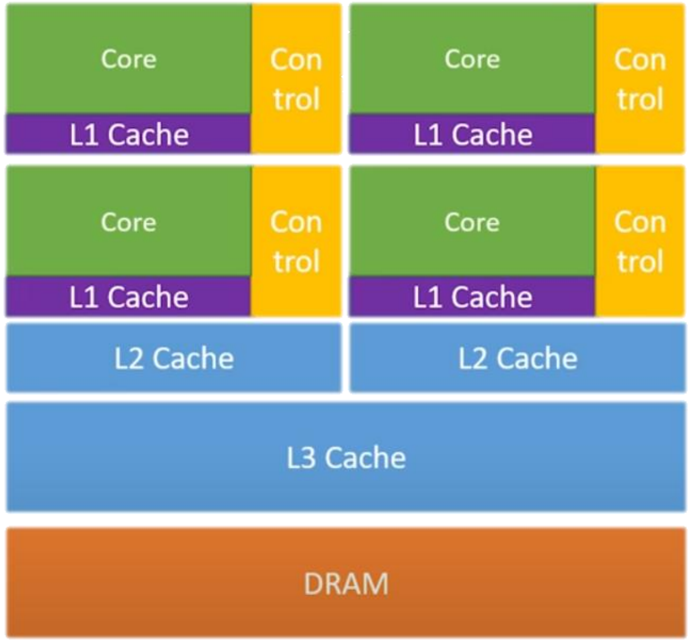
- # cores: Why Not Just Add More?
- SIMD width: Why Not Go Wider?
- Memory bandwidth: Why Not Boost It?

Why GPGPU?

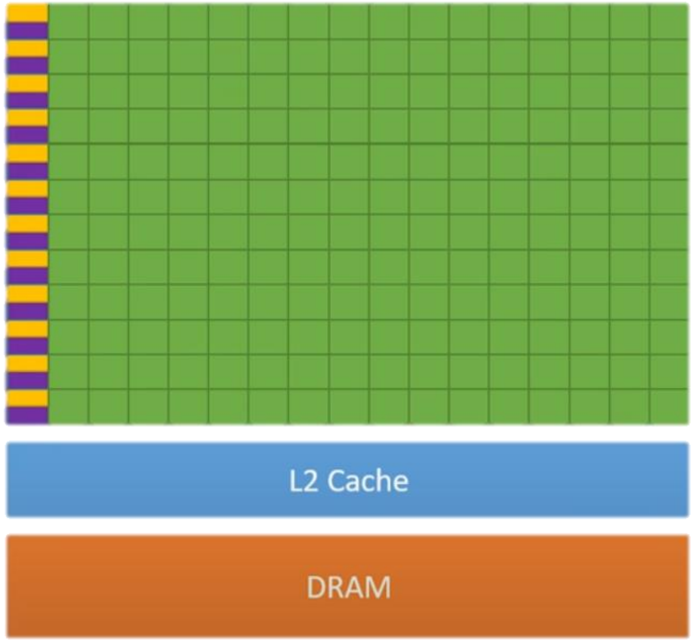
- **CPU**
  - ~10s cores
  - Low **latency**
  - Good for serial processing
  - Good for interactive tasks
  - Task parallelism

- **GPU**
  - 100s ~ 1000s cores
  - High **throughput**
  - Good for parallel processing
  - Good for big-data tasks
  - **Data** parallelism

Why GPU?



CPU



GPU

What’s Better?

- Scooter

- Sport car

What’s Better?

- Scooter

Deliver many packages within a reasonable timescale

- Sport car

Deliver a package as soon as possible

Why can't CPUs Scale?

- # cores: Why Not Just Add More?
- SIMD width: Why Not Go Wider?
- Memory bandwidth: Why Not Boost It?

- The short answer is: they could, but they shouldn’t.

- CPUs are latency oriented, GPUs are throughput oriented

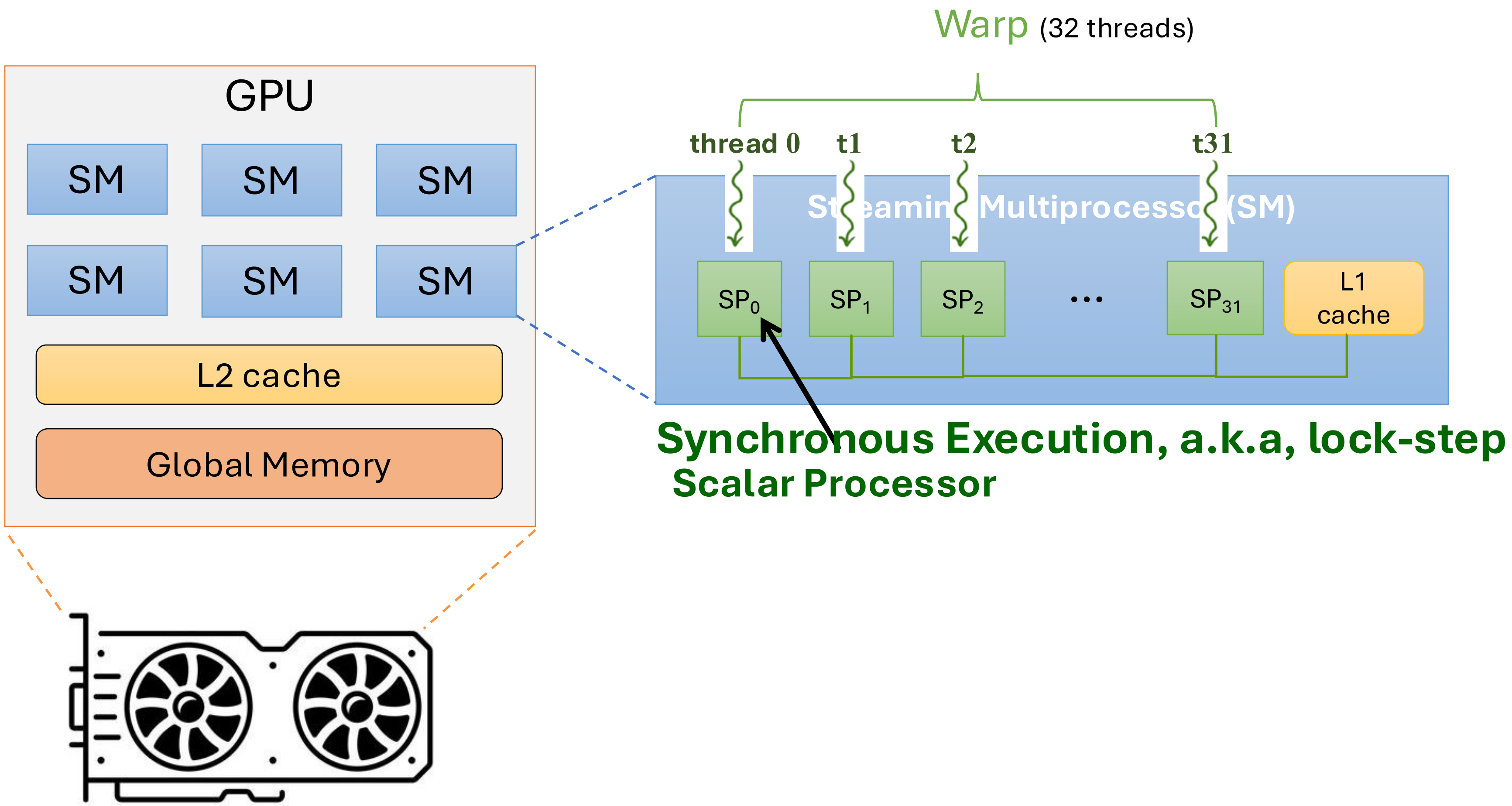
- Power, Heat, and Cost Constraints

GPU Architecture

SIMT: single-instruction multiple threads

In A100 GPU each SM has:

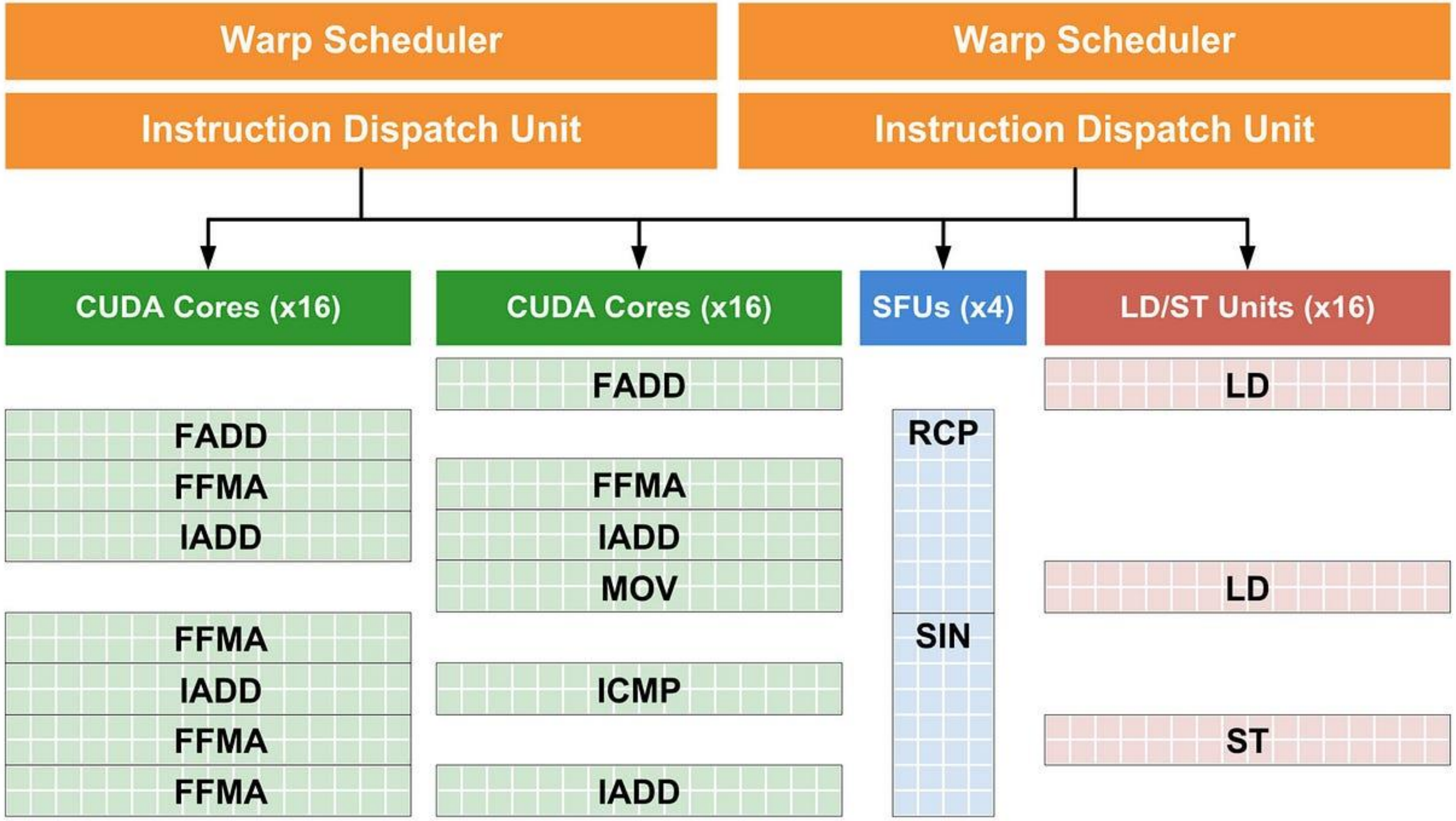
- 64 FP32 cores
- 32 FP64 cores
- 64 KB registers
- 192 KB L1 + shared mem

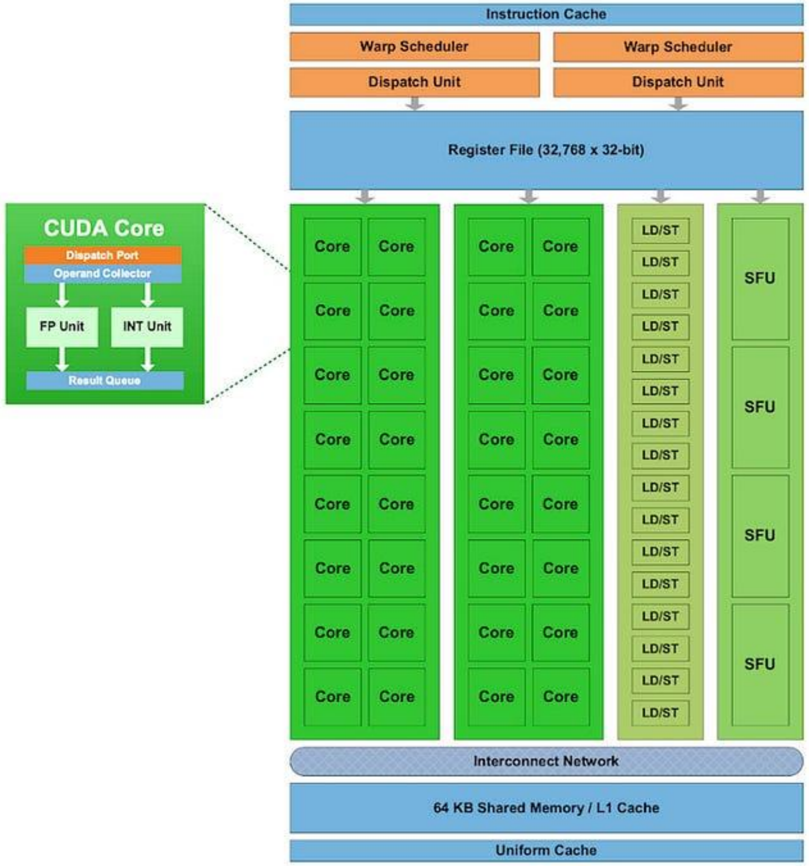


Synchronous Execution, a.k.a, lock-step

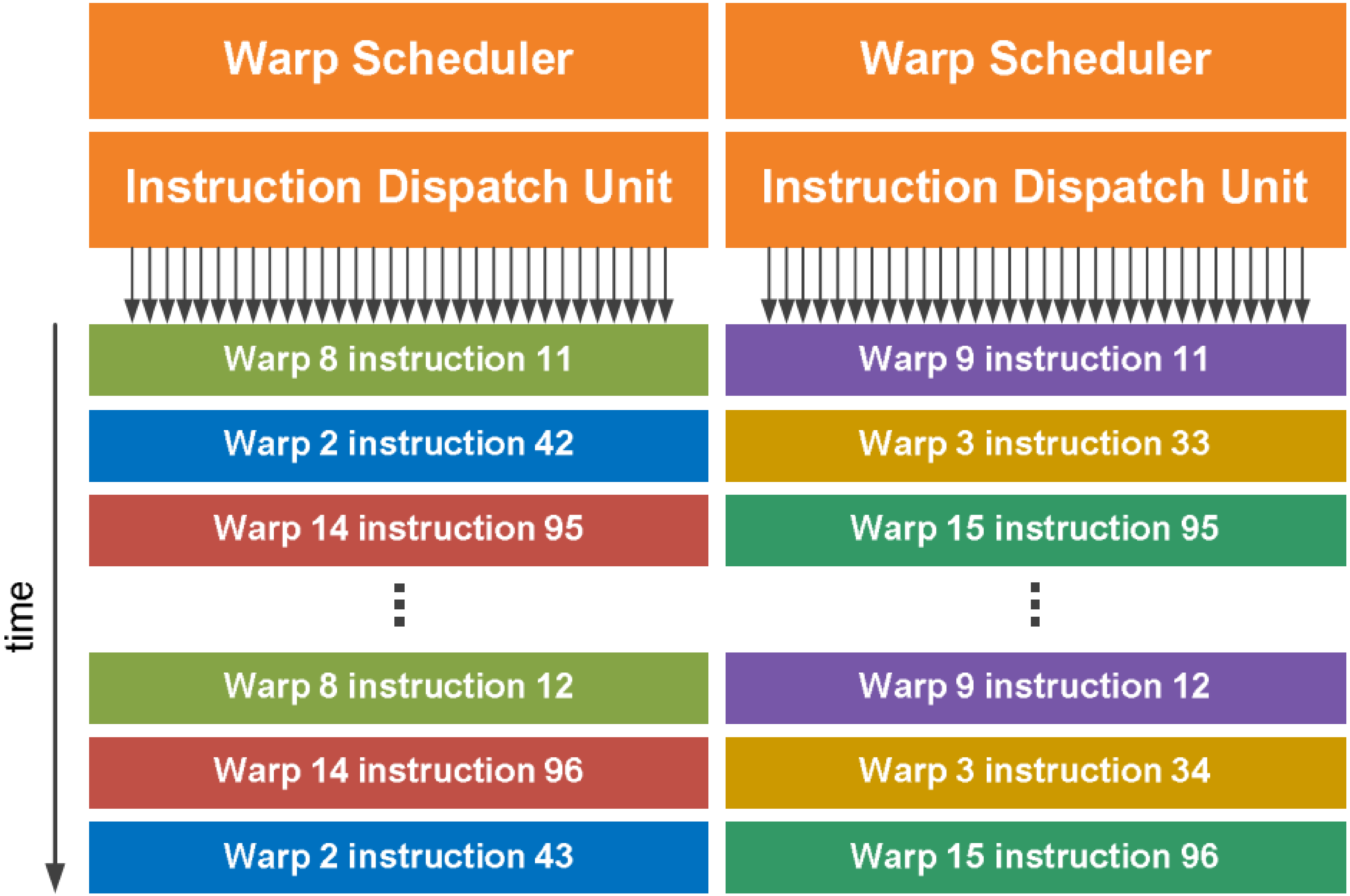
Scalar Processor

Warp Scheduler



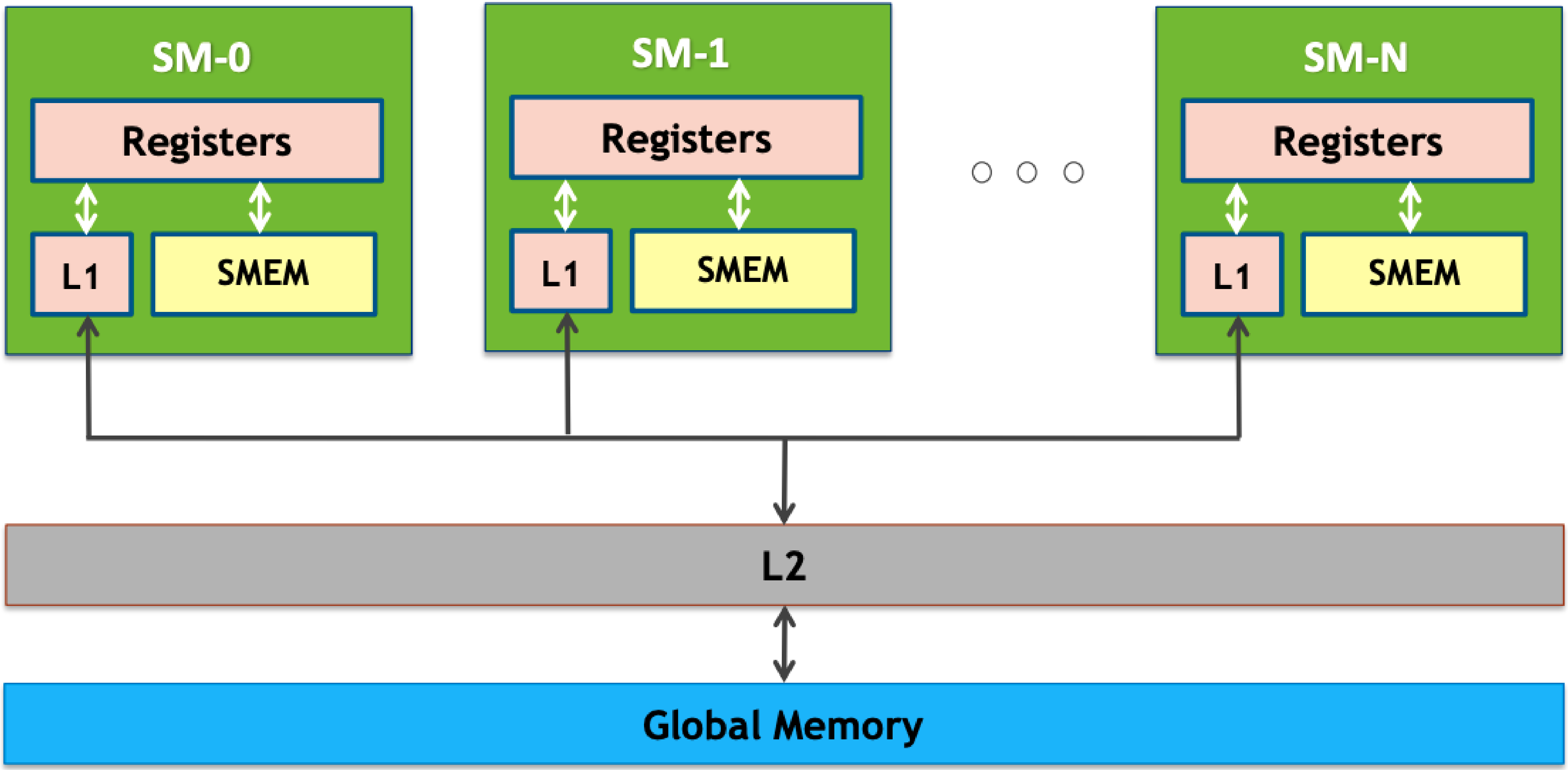


Warp Scheduler

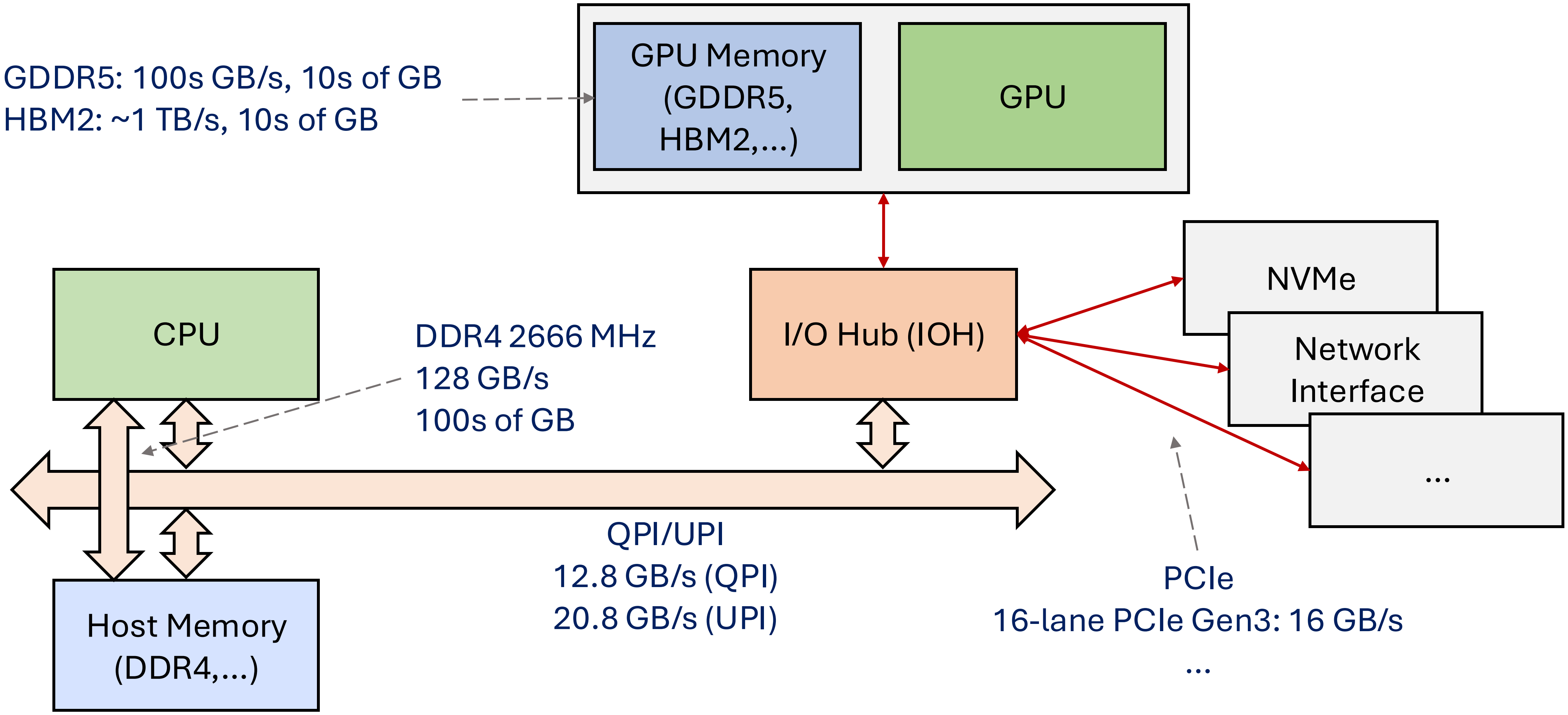


Warp Scheduler	Warp Scheduler
Instruction Dispatch Unit	Instruction Dispatch Unit
Warp 8 instruction 11	Warp 9 instruction 11
Warp 2 instruction 42	Warp 3 instruction 33
Warp 14 instruction 95	Warp 15 instruction 95
⋮	⋮
Warp 8 instruction 12	Warp 9 instruction 12
Warp 14 instruction 96	Warp 3 instruction 34
Warp 2 instruction 43	Warp 15 instruction 96

GPU Memory Hierarchy



System Architecture With a GPU (2019)



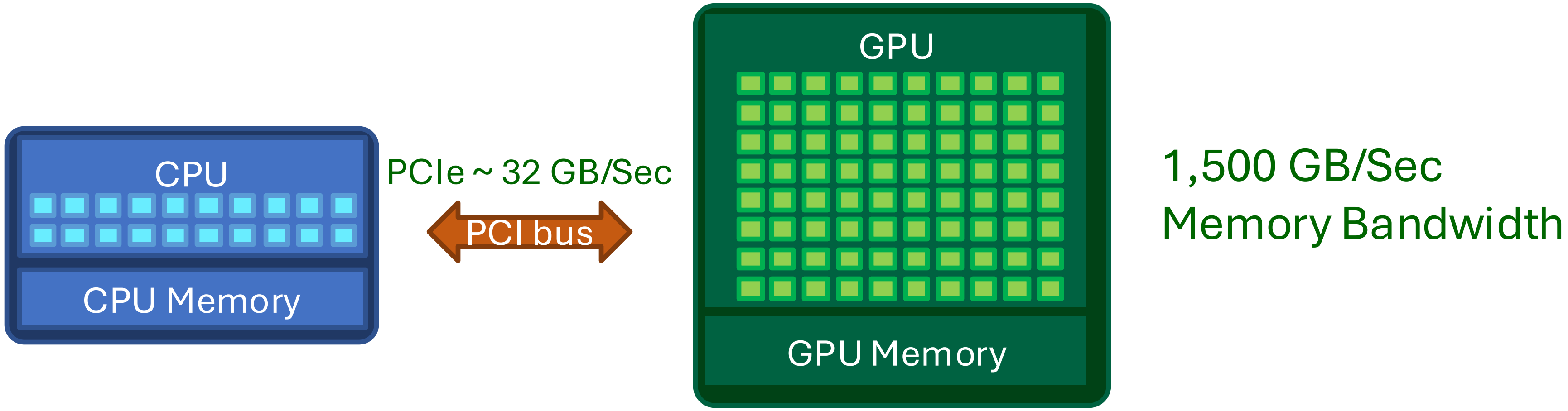
Outline

- GPU Architecture
  - Three major ideas that make GPU processing cores run fast
  - Closer look at real GPU designs
  - The GPU memory hierarchy: moving data to processors
- General Purpose GPU (GPGPU)
- GPU Programming Model
- Program a GPU with CUDA

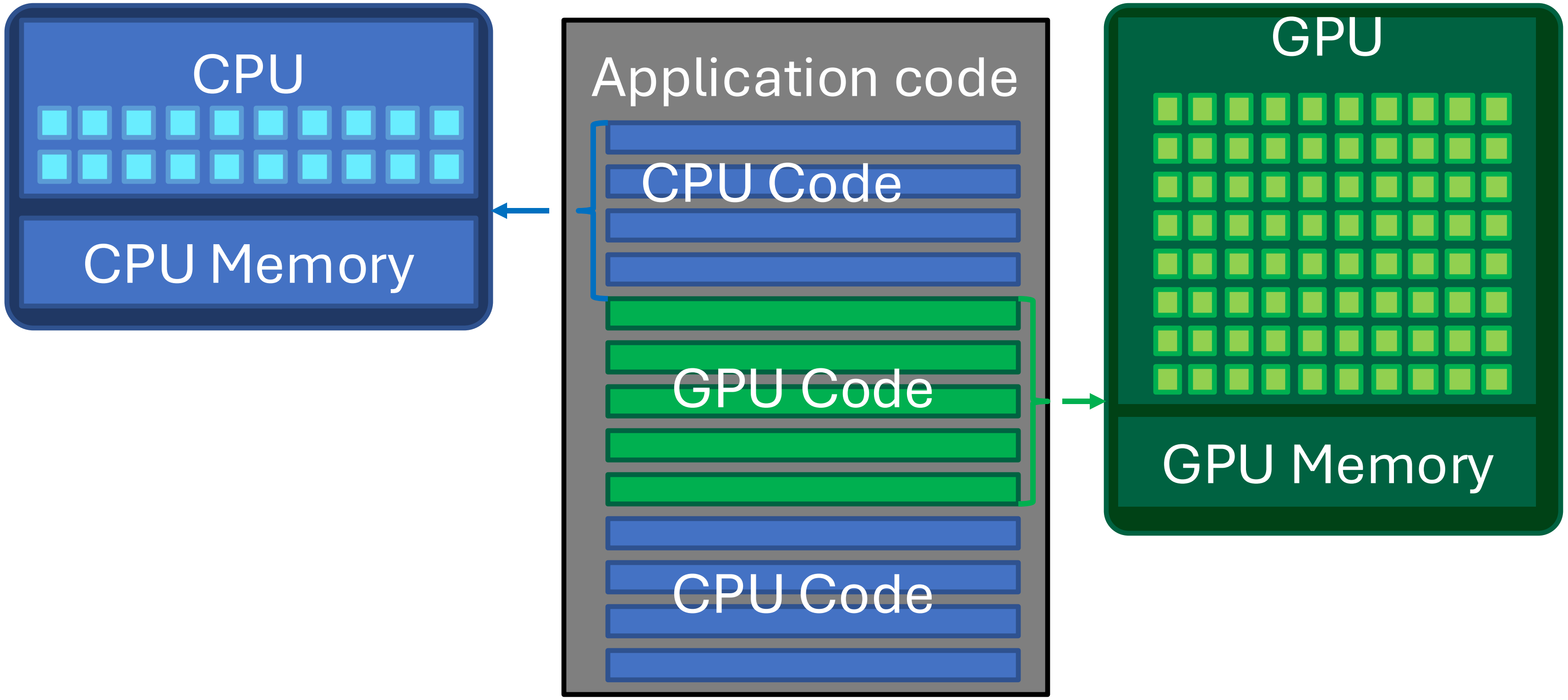
Heterogeneous Programming (CPU+GPU)

- CPU (Milan)
  - 64 cores, 2 threads per core
  - 4 NUMA domains per socket
  - 2xAVX2 (256 bit), so 2x4 in double precision
  - ~512 way parallelism (64*2*4)

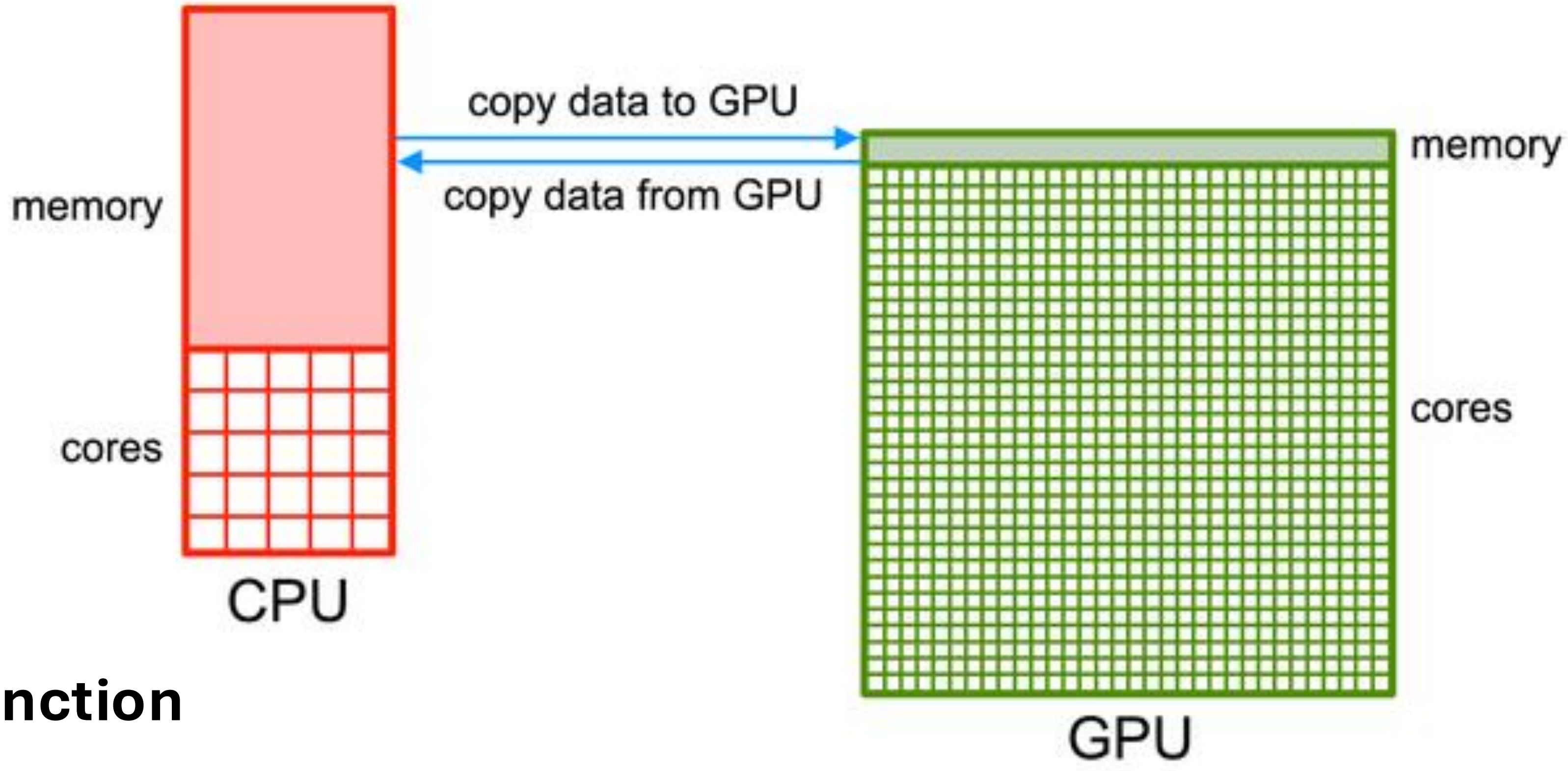
- GPU (A100)
  - 108 SM
  - 64 warps per SM (32 active at a time)
  - 32 threads per warp in double precision
  - ~200,000+ way parallelism (108*64*32) per GPU



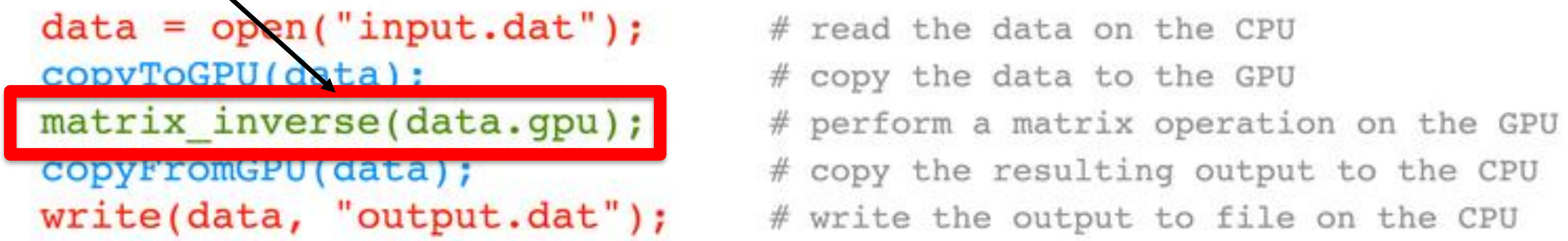
Heterogeneous Programming (CPU+GPU)



Data Movement

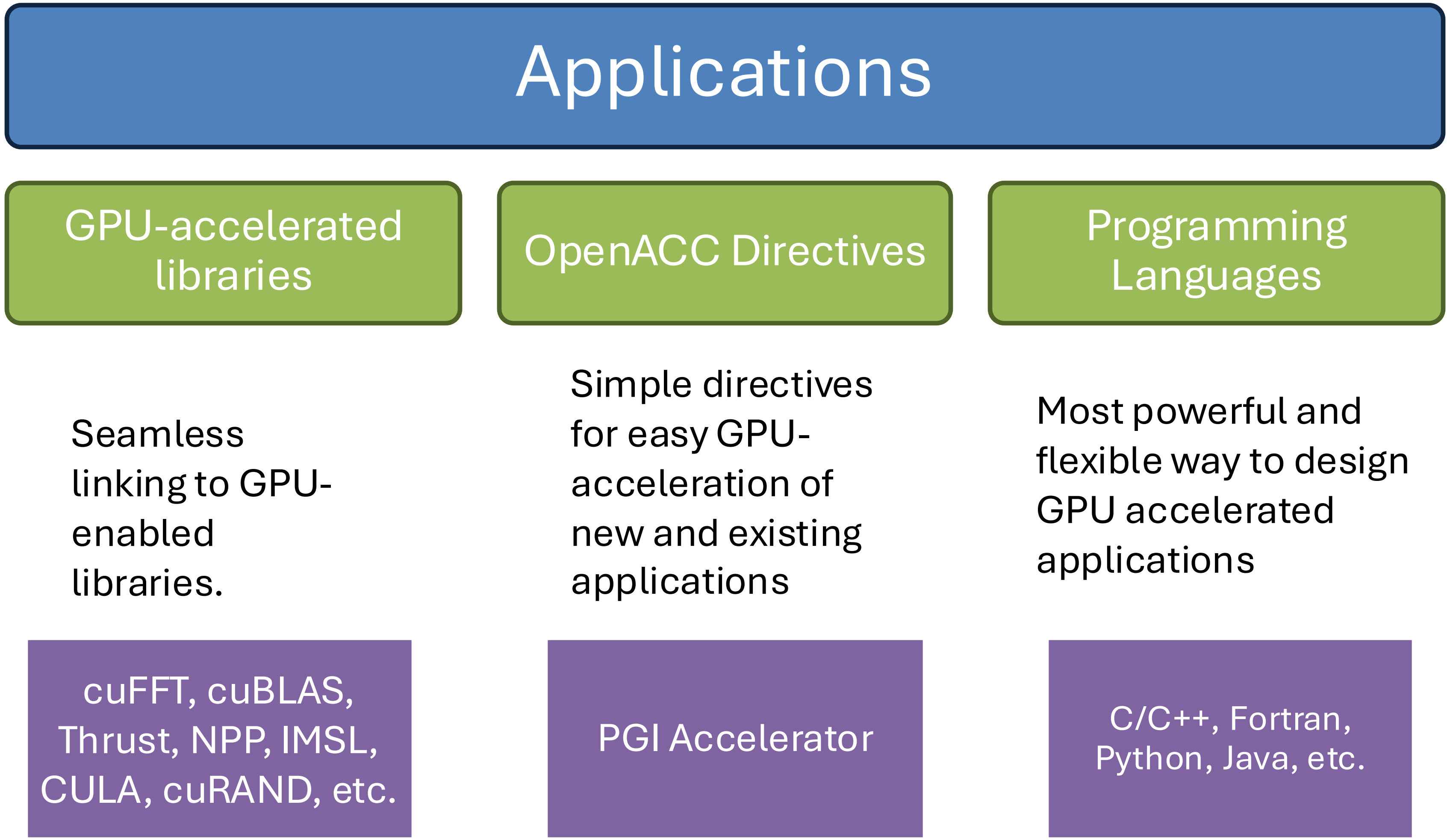


“kernel” function



```
data = open("input.dat");       # read the data on the CPU
copyToGPU(data);                # copy the data to the GPU
matrix_inverse(data.gpu);       # perform a matrix operation on the GPU
copyFromGPU(data);              # copy the resulting output to the CPU
write(data, "output.dat");      # write the output to file on the CPU
```

3 Ways of GPU Acceleration



Applications

GPU-accelerated libraries

Seamless linking to GPU-enabled libraries.

cuFFT, cuBLAS, Thrust, NPP, IMSL, CULA, cuRAND, etc.

OpenACC Directives

Simple directives for easy GPU-acceleration of new and existing applications

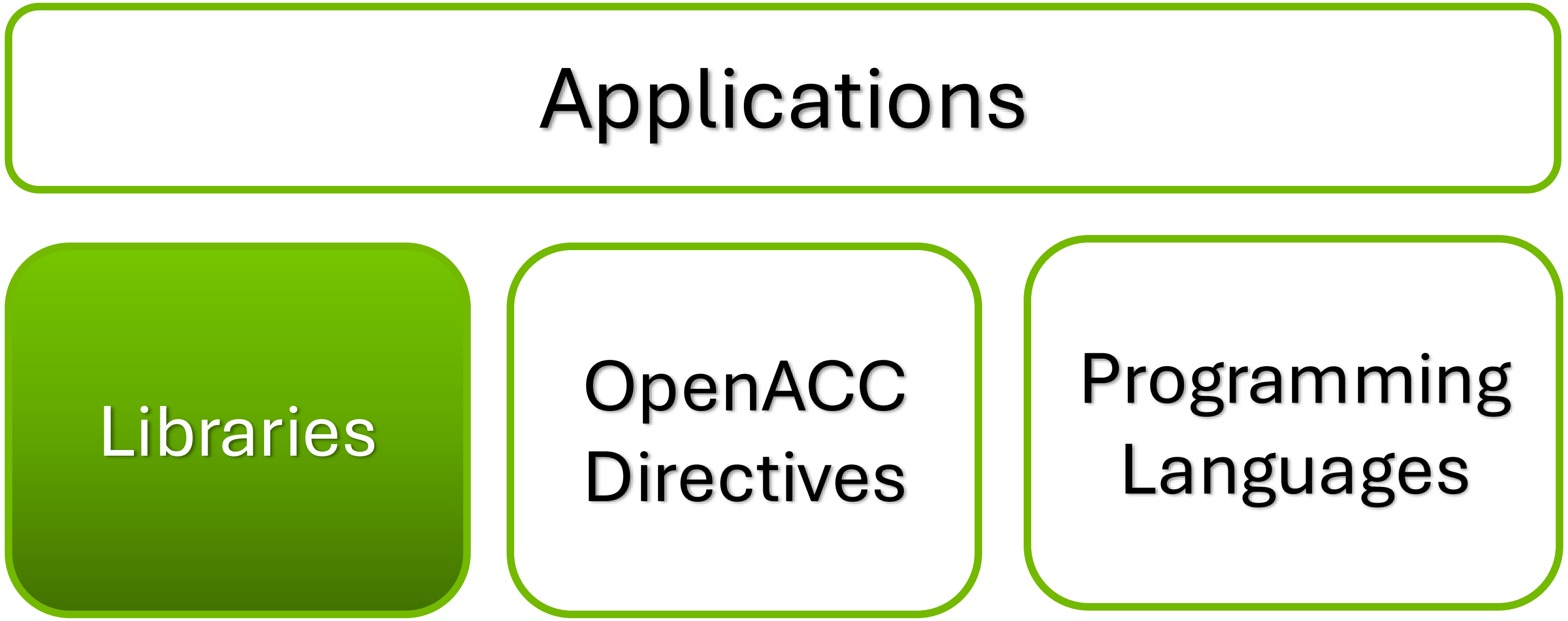
PGI Accelerator

Programming Languages

Most powerful and flexible way to design GPU accelerated applications

C/C++, Fortran, Python, Java, etc.

3 Ways of GPU Acceleration



Applications

Libraries

OpenACC Directives

Programming Languages

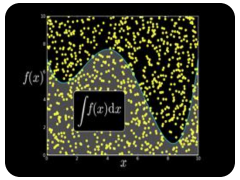
"Drop-in" Acceleration

Easily Accelerate Applications

Maximum Flexibility

GPU Accelerated Libraries

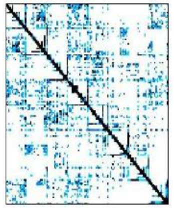
NVIDIA cuBLAS



NVIDIA cuRAND

NVIDIA NPP

OpenCV



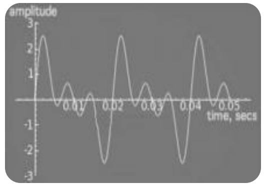
NVIDIA cuSPARSE

CUSP

Sparse Linear Algebra

Thrust

C++ STL Features for CUDA



NVIDIA cuFFT

Thrust: Rapid Parallel C++ Development

- Resembles C++ STL
- High-level interface
  - Enhances developer productivity
  - Enables performance portability between GPUs and multicore CPUs
- Flexible
  - CUDA, OpenMP, and TBB backends
  - Extensible and customizable
  - Integrates with existing software
- Open source

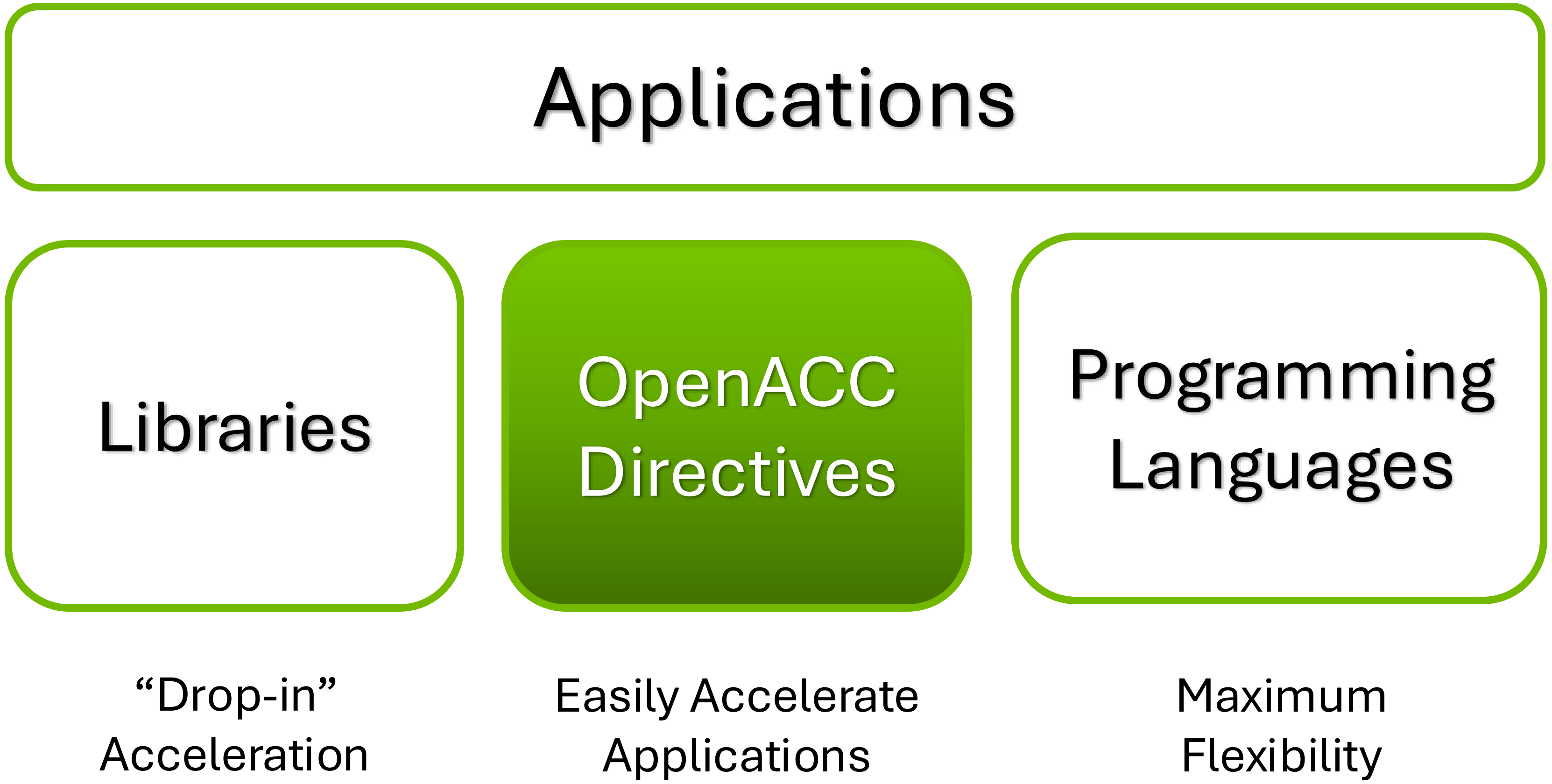
```
// generate 32M random numbers on host
thrust::host_vector<int> h_vec(32 << 20);
thrust::generate(h_vec.begin(),
                 h_vec.end(),
                 rand);

// transfer data to device (GPU)
thrust::device_vector<int> d_vec = h_vec;

// sort data on device
thrust::sort(d_vec.begin(), d_vec.end());

// transfer data back to host
thrust::copy(d_vec.begin(),
             d_vec.end(),
             h_vec.begin());
```

http://developer.nvidia.com/thrust or http://thrust.googlecode.com

Libraries: Easy, High-Quality Acceleration

- Ease of use: Using libraries enables GPU acceleration without in-depth knowledge of GPU programming
- “Drop-in”: Many GPU-accelerated libraries follow standard APIs, thus enabling acceleration with minimal code changes
- Quality: Libraries offer high-quality implementations of functions encountered in a broad range of applications
- Performance: NVIDIA libraries are tuned by experts

3 Ways of GPU Acceleration



Applications

Libraries

“Drop-in” Acceleration

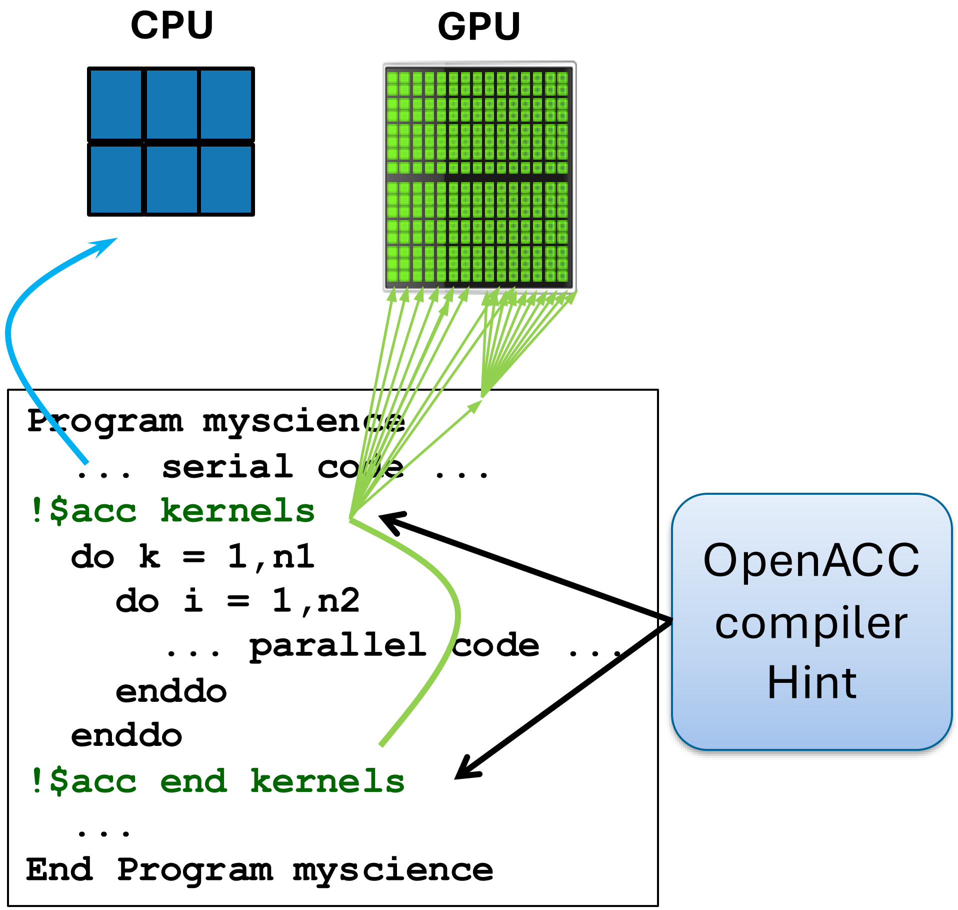
OpenACC Directives

Easily Accelerate Applications

Programming Languages

Maximum Flexibility

OpenACC Directives



```
Program myscience
  ... serial code ...
!$acc kernels
  do k = 1,n1
    do i = 1,n2
      ... parallel code ...
    enddo
  enddo
!$acc end kernels
  ...
End Program myscience
```

Your original
Fortran or C code

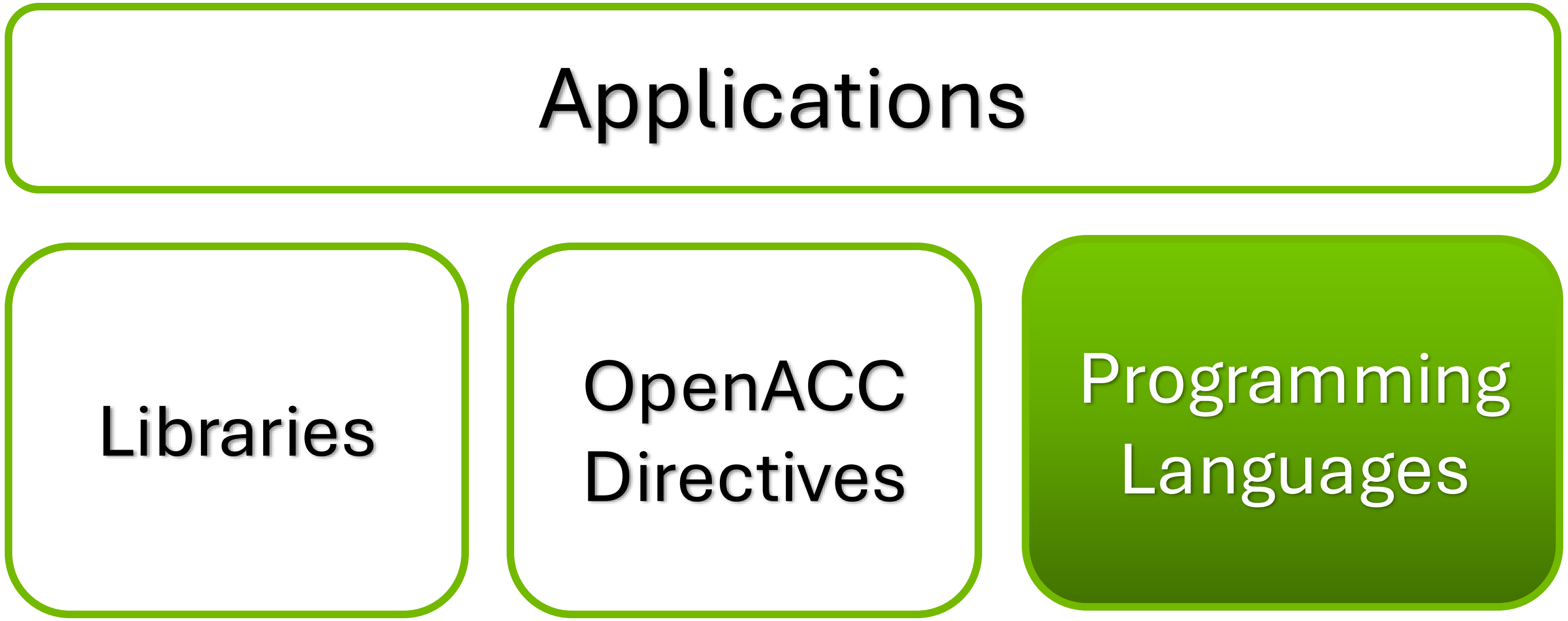
- Simple Compiler hints
- Compiler Parallelizes code
- Works on many-core GPUs & multicore CPUs

Easy

Open

Powerful

3 Ways of GPU Acceleration



Applications

Libraries

OpenACC Directives

Programming Languages

"Drop-in" Acceleration

Easily Accelerate Applications

Maximum Flexibility

Outline

- GPU Architecture
  - Three major ideas that make GPU processing cores run fast
  - Closer look at real GPU designs
  - The GPU memory hierarchy: moving data to processors
- General Purpose GPU (GPGPU)
- GPU Programming Model
- Program a GPU with CUDA

PROGRAM A GPU WITH CUDA

CONCEPTS

- Heterogeneous Computing
- Blocks
- Threads
- Indexing
- Shared memory
- __syncthreads()
- Asynchronous operation
- Handling errors
- Managing devices

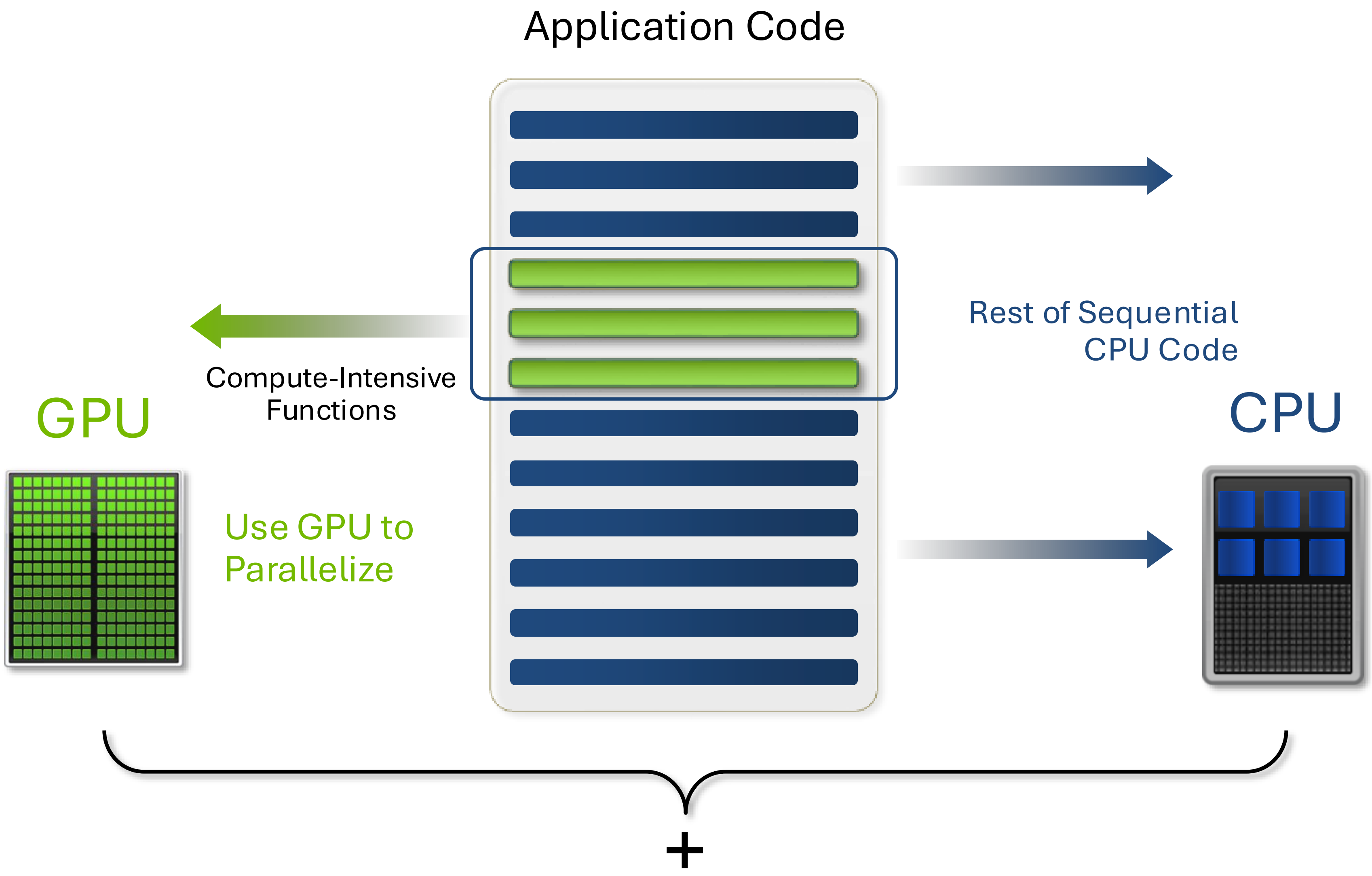
Heterogeneous Computing

- Terminology
  - Host: The CPU and its memory (host memory)
  - Device: The GPU and its memory (device memory)

Host: the CPU and its memory

Device: the GPU and its memory

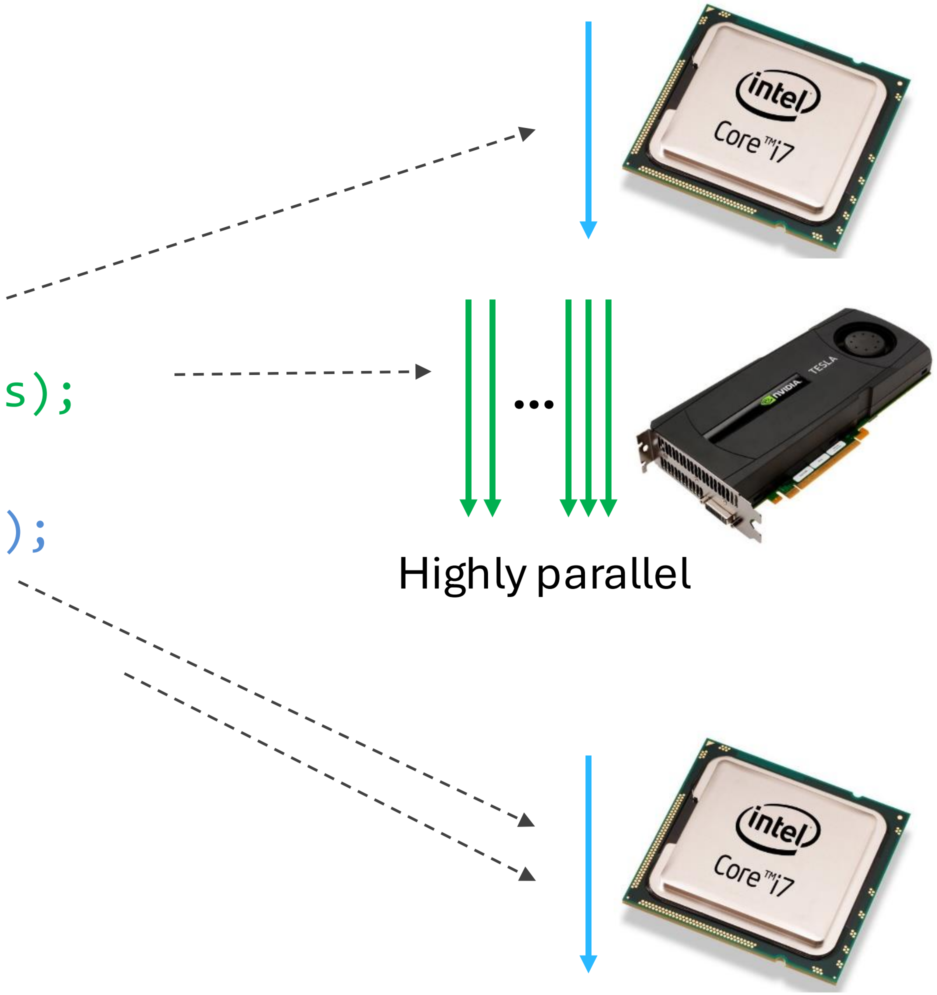
CPU-GPU Heterogeneous Computing



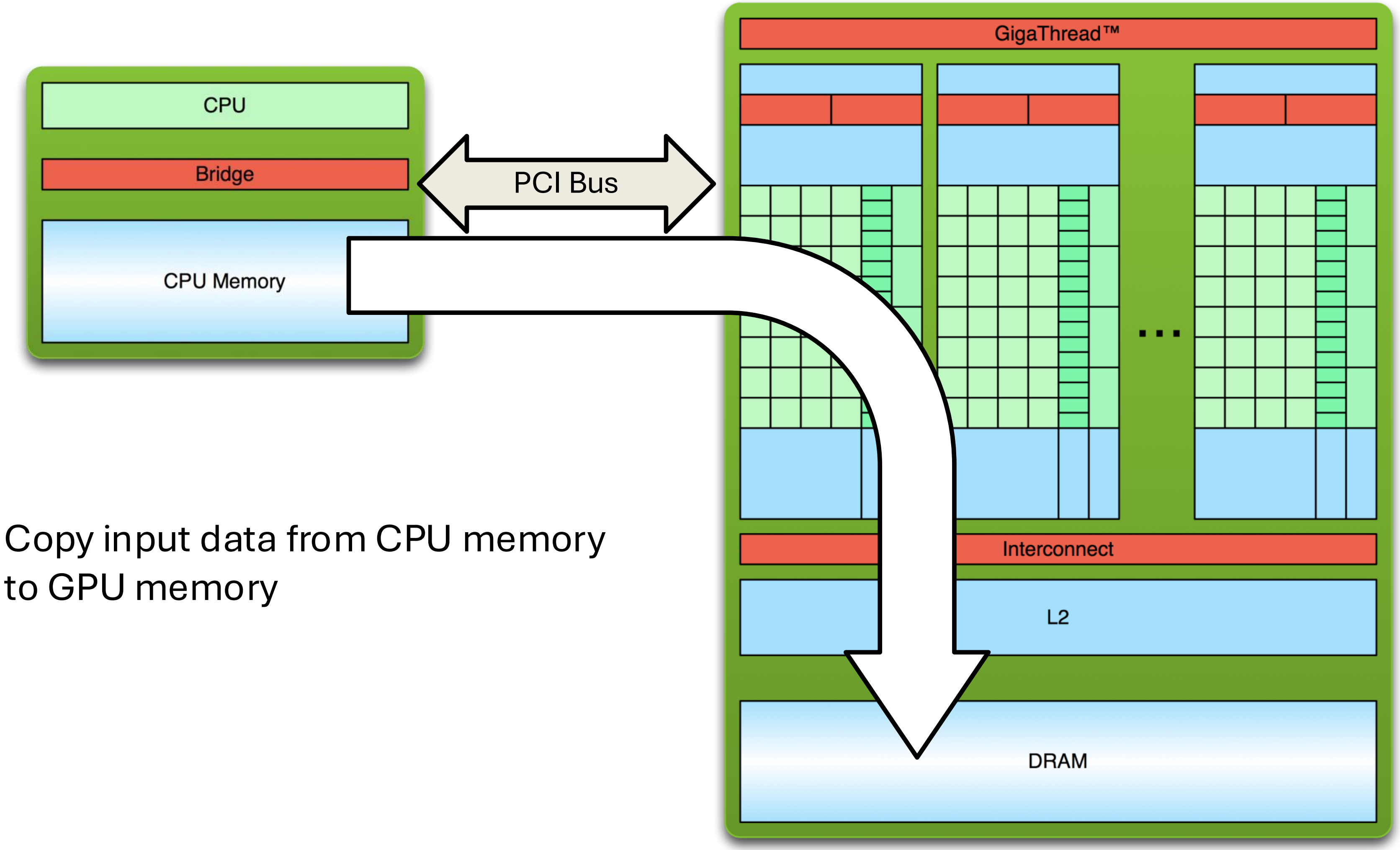
Heterogeneous Computing with CUDA

- CUDA Compute Unified Device Architecture

```
do_something_on_host();
kernel<<<nBlk, nTid>>>(args);
cudaDeviceSynchronize();
do_something_else_on_host();
```

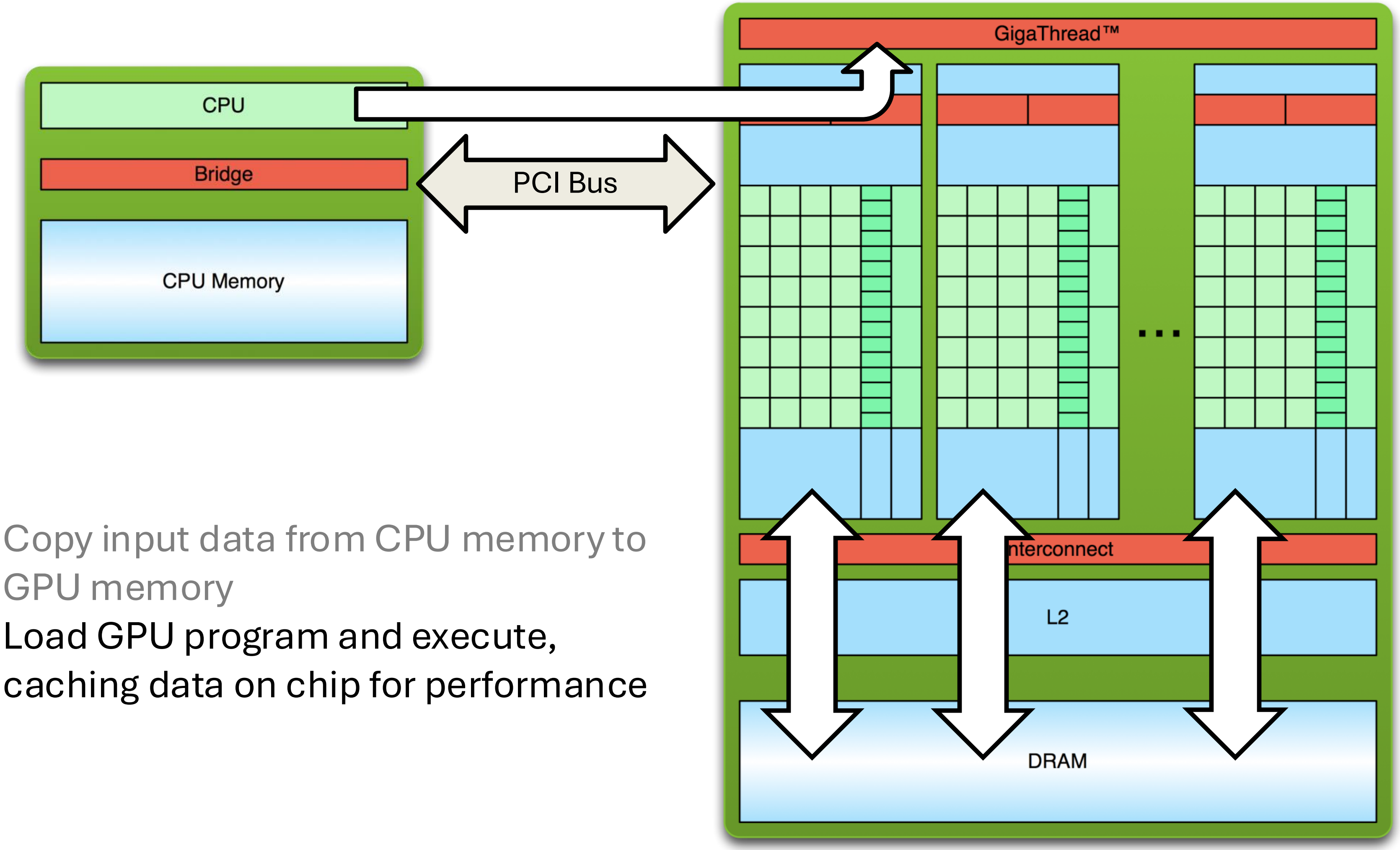


Simple Processing Flow



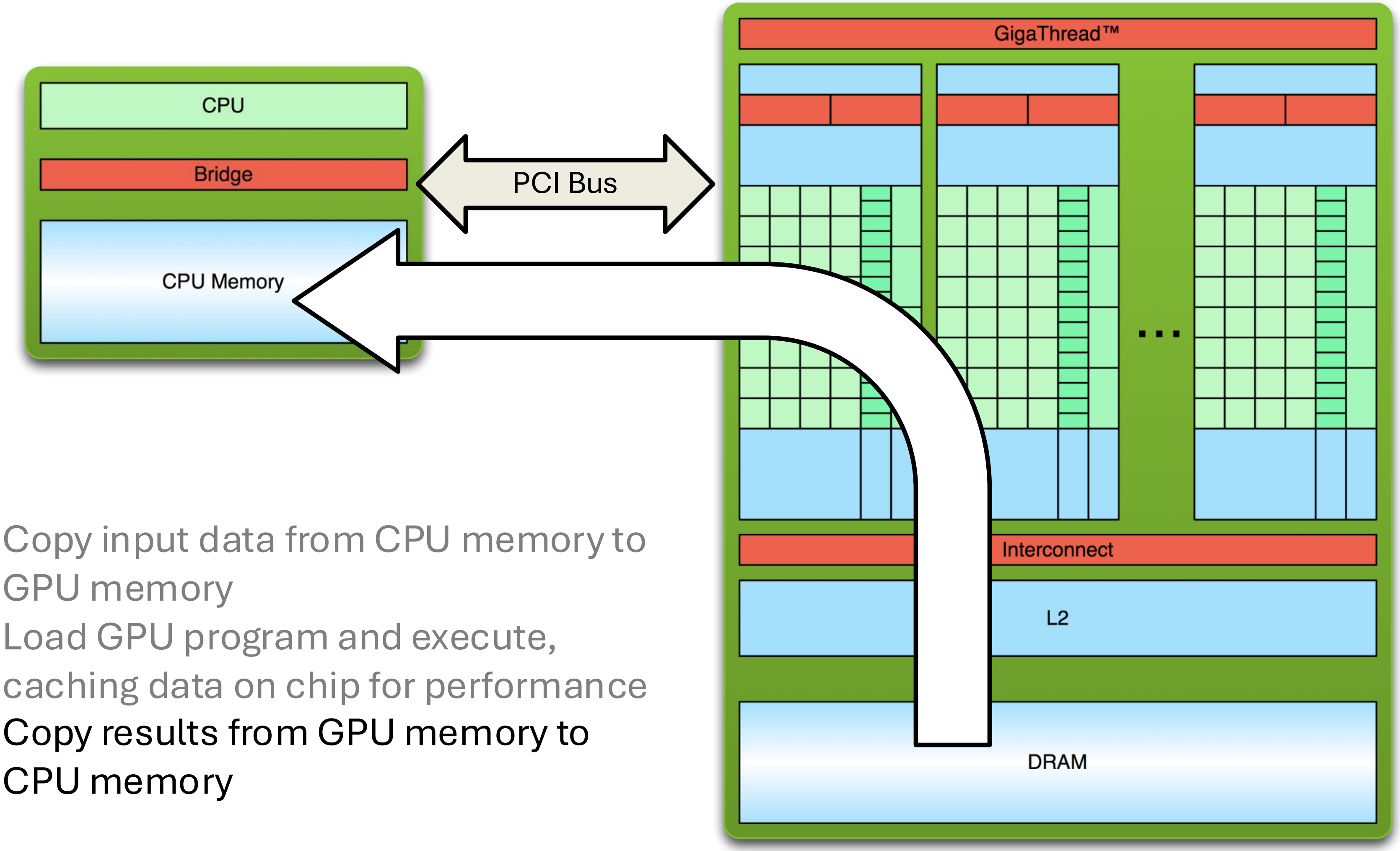
1. Copy input data from CPU memory to GPU memory

Simple Processing Flow



1. Copy input data from CPU memory to GPU memory
2. Load GPU program and execute, caching data on chip for performance

Simple Processing Flow



1. Copy input data from CPU memory to GPU memory
2. Load GPU program and execute, caching data on chip for performance
3. Copy results from GPU memory to CPU memory

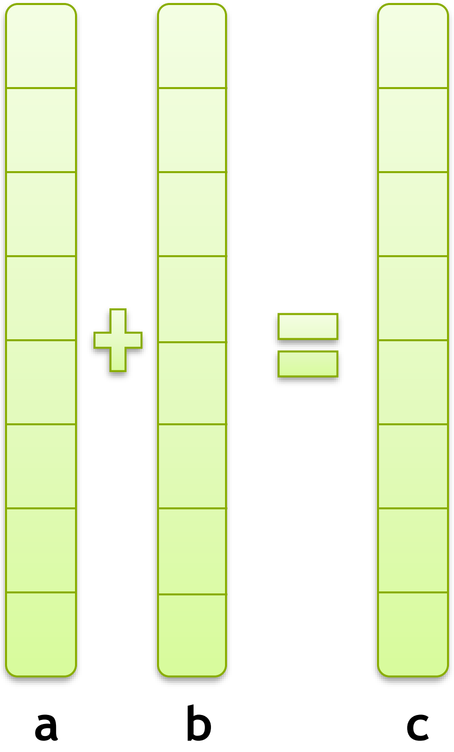
Heterogeneous Computing with CUDA C

- Let’s start with simply adding two integers

```
__global__ void add(int *a, int *b, int *c) {
       *c = *a + *b;
}
```

- Here `__global__` is a CUDA C/C++ keyword meaning
  - ▫ `add()` will execute on the device
  - ▫ `add()` will be called from the host

Vector Addition



Addition on the Device

- Note that we use pointers for the variables

```
__global__ void add(int *a, int *b, int *c) {
      *c = *a + *b;
}
```

- add() runs on the device, so a, b and c must point to device memory
- We need to allocate memory on the GPU

Memory Management

- Host and device memory are separate entities
  - Device pointers point to GPU memory
    - May be passed to/from host code
    - May not be dereferenced in host code
  - Host pointers point to CPU memory
    - May be passed to/from device code
    - May not be dereferenced in device code

- Simple CUDA API for handling device memory
  - `cudaMalloc()`, `cudaFree()`, `cudaMemcpy()`
  - Similar to the C equivalents `malloc()`, `free()`, `memcpy()`

Addition on the Device: `main()`

```
int main(void) {
    int a, b, c;              // host copies of a, b, c
    int *d_a, *d_b, *d_c;     // device copies of a, b, c
    int size = sizeof(int);

    // Allocate space for device copies of a, b, c
    cudaMalloc((void **)&d_a, size);
    cudaMalloc((void **)&d_b, size);
    cudaMalloc((void **)&d_c, size);

    // Setup input values
    a = 2;
    b = 7;
```

Vector Addition on the Device: `main()`

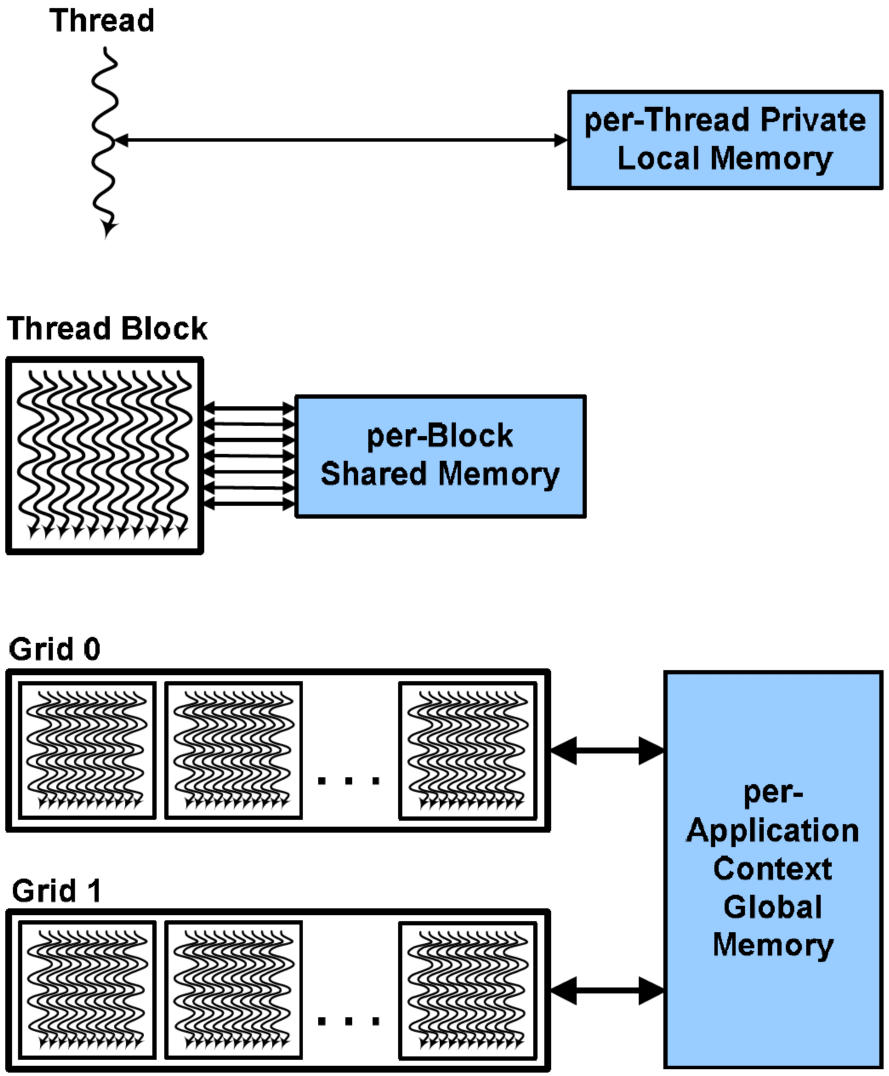
```
    // Copy inputs to device
    cudaMemcpy(d_a, &a, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_b, &b, size, cudaMemcpyHostToDevice);

    // Launch add() kernel on GPU
    add<<<1,1>>>(d_a, d_b, d_c);

    // Copy result back to host
    cudaMemcpy(&c, d_c, size, cudaMemcpyDeviceToHost);

    // Cleanup
    cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
    return 0;
}
```

CONCEPTS

- Heterogeneous Computing
- Blocks
- Threads
- Indexing
- Shared memory
- __syncthreads()
- Asynchronous operation
- Handling errors
- Managing devices

RUNNING IN PARALLEL

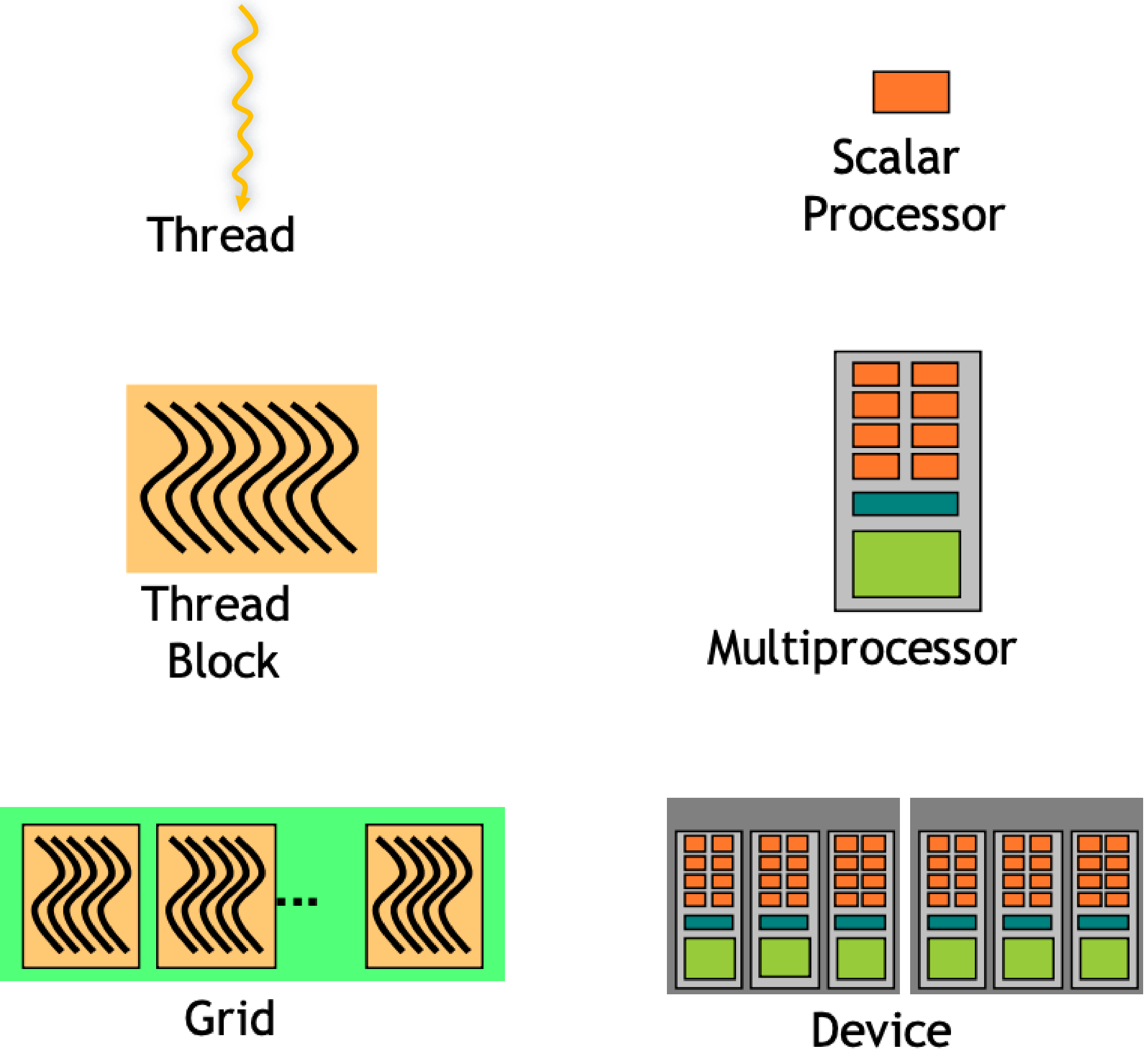
Multi-level Threading Model


Thread
per-Thread Private Local Memory
Thread Block
per-Block Shared Memory
Grid 0
. . .
Grid 1
. . .
per-Application Context Global Memory

CUDA Hierarchy of threads, blocks, and grids, with corresponding per-thread private, per-block shared, and per-application global memory spaces.

Execution Model

Software **Hardware**



Thread

Scalar Processor

Thread Block

Multiprocessor

Grid

Device

Threads are executed by scalar processors

Thread blocks are executed on multiprocessors

Thread blocks do not migrate

Several concurrent thread blocks can reside on one multiprocessor - limited by multiprocessor resources (shared memory and register file)

A kernel is launched as a grid of thread blocks

Moving to Parallel Execution

GPU computing is about massive parallelism

So how do we run code in parallel on the device?

```
add<<< 1, 1 >>>();

add<<< N, 1 >>>();
```

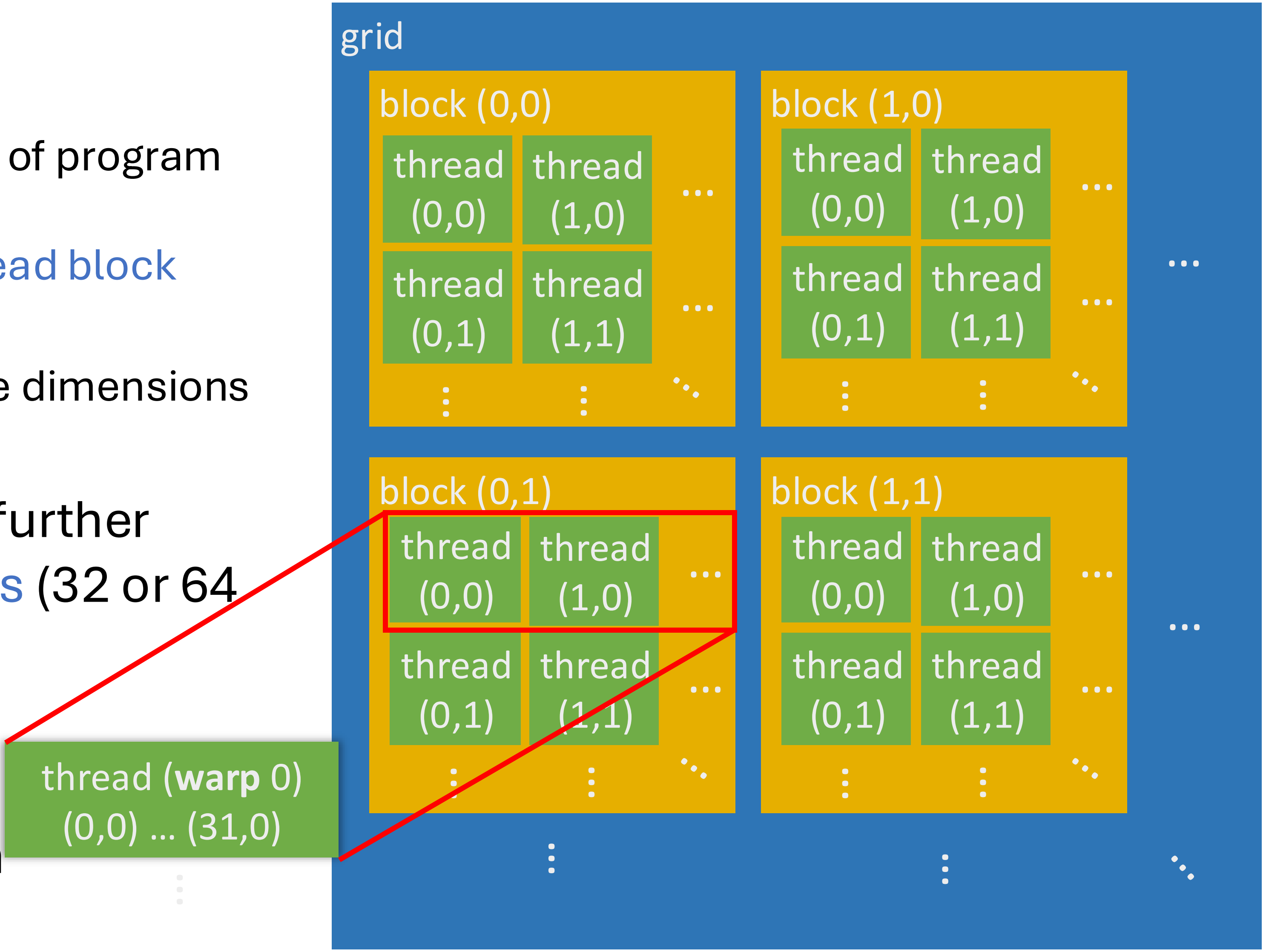
Instead of executing `add()` once, execute N times in parallel

Hierarchical Programming Model

- Multi-level hierarchy
  - The thread is the smallest unit of program execution
  - Threads are organized in a thread block
  - Blocks form a grid
  - Block and grid have up to three dimensions

- Threads within a block are further implicitly divided into warps (32 or 64 threads)

- Choose a certain level of granularity given a problem



Vector Addition on the Device

- With `add()` running in parallel we can do vector addition

- Each parallel invocation of `add()` is referred to as a block
  - The set of blocks is referred to as a grid
  - Each invocation can refer to its block index using `blockIdx.x`

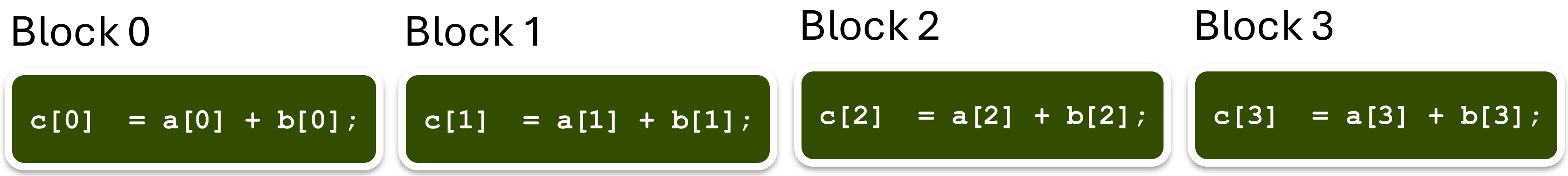
```
__global__ void add(int *a, int *b, int *c) {
    c[blockIdx.x] = a[blockIdx.x] + b[blockIdx.x];
}
```

- By using `blockIdx.x` to index into the array, each block handles a different index

Vector Addition on the Device

```
__global__ void add(int *a, int *b, int *c) {
    c[blockIdx.x] = a[blockIdx.x] + b[blockIdx.x];
}
```

- On the device, each block can execute in parallel:



Block 0

`c[0]  = a[0] + b[0];`

Block 1

`c[1]  = a[1] + b[1];`

Block 2

`c[2]  = a[2] + b[2];`

Block 3

`c[3]  = a[3] + b[3];`

Vector Addition on the Device: `add()`

- Returning to our parallelized **`add()`** kernel

```
__global__ void add(int *a, int *b, int *c) {
        c[blockIdx.x] = a[blockIdx.x] + b[blockIdx.x];
}
```

- Let’s take a look at main()…

Vector Addition on the Device: `main()`

```
#define N 512
int main(void) {
    int *a  *b  *c                // host copies of a, b, c
    int *d_a, *d_b, *d_c; // device copies of a, b, c
    int size = N * sizeof(int);

    // Alloc space for device copies of a, b, c
    cudaMalloc((void **)&d_a, size);
    cudaMalloc((void **)&d_b, size);
    cudaMalloc((void **)&d_c, size);

    // Alloc space for host copies of a, b, c and setup input values
    a = (int *)malloc(size); random_ints(a, N);
    b = (int *)malloc(size); random_ints(b, N);
    c = (int *)malloc(size);
```

Vector Addition on the Device: `main()`

```
    // Copy inputs to device
    cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);

    // Launch add() kernel on GPU with N blocks
    add<<<N,1>>>(d_a, d_b, d_c);

    // Copy result back to host
    cudaMemcpy(c, d_c, size, cudaMemcpyDeviceToHost);

    // Cleanup
    free(a); free(b); free(c);
    cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
    return 0;
}
```

Review (1 of 2)

- Difference between host and device

Host	CPU
Device	GPU

- Using `__global__` to declare a function as device code
  - Executes on the device
  - Called from the host

- Passing parameters from host code to a device function

Review (2 of 2)

- Basic device memory management
  - `cudaMalloc()`
  - `cudaMemcpy()`
  - `cudaFree()`

- Launching parallel kernels
  - Launch `N` copies of `add()` with `add<<<N,1>>>(…);`
  - Use `blockIdx.x` to access block index

CONCEPTS

- Heterogeneous Computing
- Blocks
- **Threads**
- Indexing
- Shared memory
- __syncthreads()
- Asynchronous operation
- Handling errors
- Managing devices

INTRODUCING THREADS

CUDA Threads

- Terminology: a block can be split into parallel threads

- Let’s change `add()` to use parallel threads instead of parallel blocks

```
__global__ void add(int *a, int *b, int *c) {
    c[threadIdx.x] = a[threadIdx.x] + b[threadIdx.x];
}
```

- We use `threadIdx.x` instead of `blockIdx.x`

- Need to make one change in `main()`…

Vector Addition Using Threads: `main()`

```
#define N 512
   int main(void) {
       int *a, *b, *c;             // host copies of a, b, c
       int *d_a, *d_b, *d_c;       // device copies of a, b, c
       int size = N * sizeof(int);

       // Alloc space for device copies of a, b, c
       cudaMalloc((void **)&d_a, size);
       cudaMalloc((void **)&d_b, size);
       cudaMalloc((void **)&d_c, size);

       // Alloc space for host copies of a, b, c and setup input values
       a = (int *)malloc(size); random_ints(a, N);
       b = (int *)malloc(size); random_ints(b, N);
       c = (int *)malloc(size);
```

Vector Addition Using Threads: main()

```
// Copy inputs to device
       cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);
       cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);

       // Launch add() kernel on GPU with N threads
       add<<<1,N>>>(d_a, d_b, d_c);

       // Copy result back to host
       cudaMemcpy(c, d_c, size, cudaMemcpyDeviceToHost);

       // Cleanup
       free(a); free(b); free(c);
       cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
       return 0;
   }
```

“kernel” function

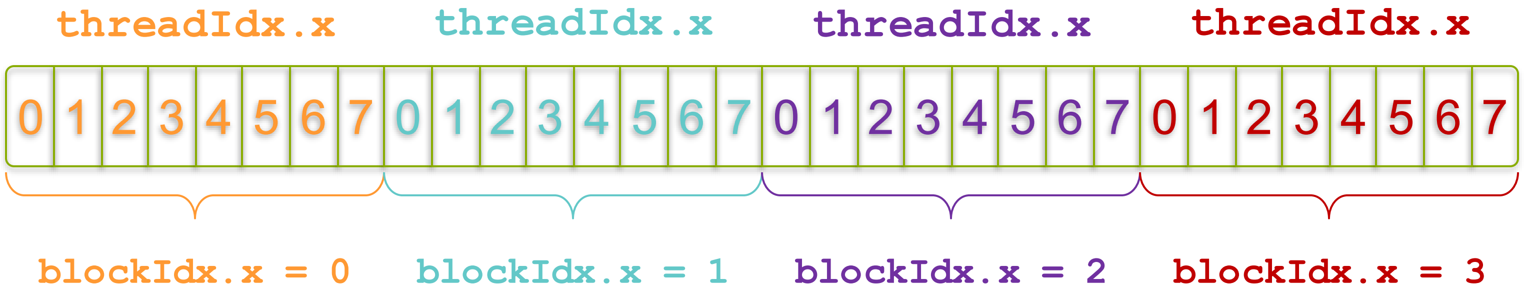
CONCEPTS

- Heterogeneous Computing
- Blocks
- Threads
- Indexing
- Shared memory
- __syncthreads()
- Asynchronous operation
- Handling errors
- Managing devices

COMBINING THREADS AND BLOCKS

Indexing Arrays with Blocks and Threads

- No longer as simple as using `blockIdx.x` and `threadIdx.x`
  - Consider indexing an array with one element per thread (8 threads/block)



threadIdx.x threadIdx.x threadIdx.x threadIdx.x

0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7

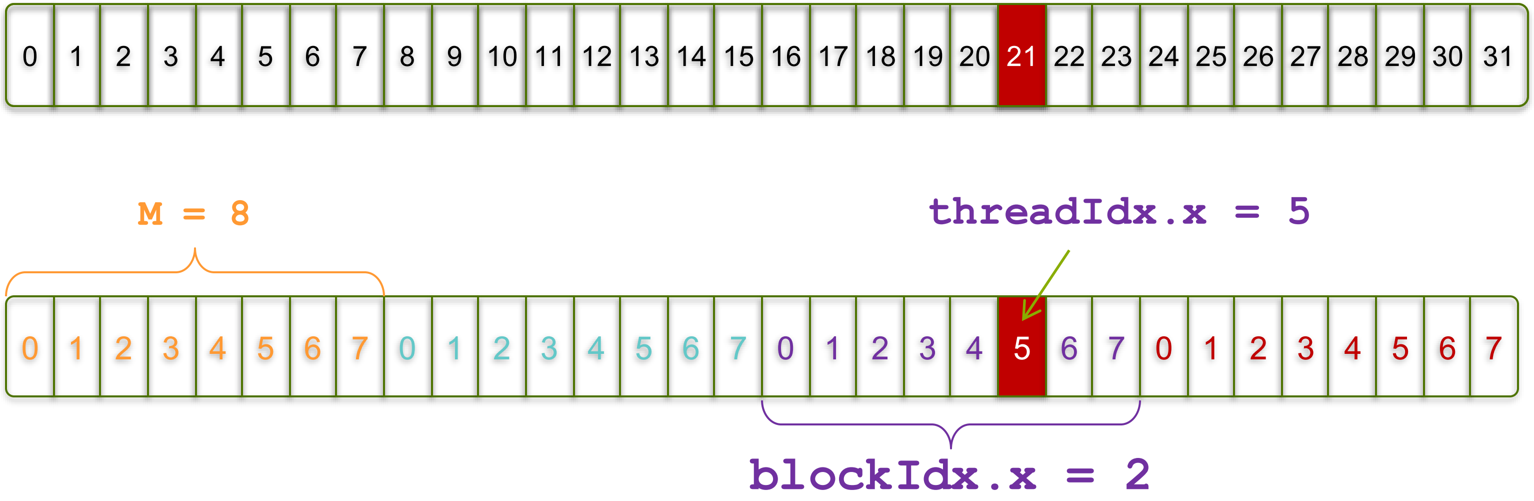
blockIdx.x = 0 blockIdx.x = 1 blockIdx.x = 2 blockIdx.x = 3

- With M threads/block a unique index for each thread is given by

```
int index = threadIdx.x + blockIdx.x * M;
```

Indexing Arrays: Example

- Which thread will operate on the red element?



0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31

M = 8

threadIdx.x = 5

0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7

blockIdx.x = 2

```
int index = threadIdx.x + blockIdx.x * M;
          =       5     +     2      * 8;
          = 21;
```

Vector Addition with Blocks and Threads

- Use the built-in variable `blockDim.x` for threads per block

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

- Combined version of `add()` to use parallel threads and parallel blocks

```
__global__ void add(int *a, int *b, int *c) {
    int index = threadIdx.x + blockIdx.x * blockDim.x;
    c[index] = a[index] + b[index];
}
```

- What changes need to be made in `main()`?

Addition with Blocks and Threads: `main()`

```
#define N (2048*2048)
#define THREADS_PER_BLOCK 512
int main(void) {
    int *a, *b, *c;                      // host copies of a, b, c
    int *d_a, *d_b, *d_c;        // device copies of a, b, c
    int size = N * sizeof(int);

    // Alloc space for device copies of a, b, c
    cudaMalloc((void **)&d_a, size);
    cudaMalloc((void **)&d_b, size);
    cudaMalloc((void **)&d_c, size);

    // Alloc space for host copies of a, b, c and setup input values
    a = (int *)malloc(size); random_ints(a, N);
    b = (int *)malloc(size); random_ints(b, N);
    c = (int *)malloc(size);
```

Addition with Blocks and Threads: `main()`

```
// Copy inputs to device
       cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);
       cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);

       // Launch add() kernel on GPU
       add<<<N/THREADS_PER_BLOCK, THREADS_PER_BLOCK>>>(d_a, d_b, d_c);

       // Copy result back to host
       cudaMemcpy(c, d_c, size, cudaMemcpyDeviceToHost);

       // Cleanup
       free(a); free(b); free(c);
       cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
       return 0;
   }
```

Handling Arbitrary Vector Sizes

- Typical problems are not friendly multiples of `blockDim.x`

- Avoid accessing beyond the end of the arrays:

```
__global__ void add(int *a, int *b, int *c, int n) {
    int index = threadIdx.x + blockIdx.x * blockDim.x;
    if (index < n)
        c[index] = a[index] + b[index];
}
```

Check bound

- Update the kernel launch:

```
add<<<(N-1) / M + 1 M>>>(d_a, d_b, d_c, N);
```

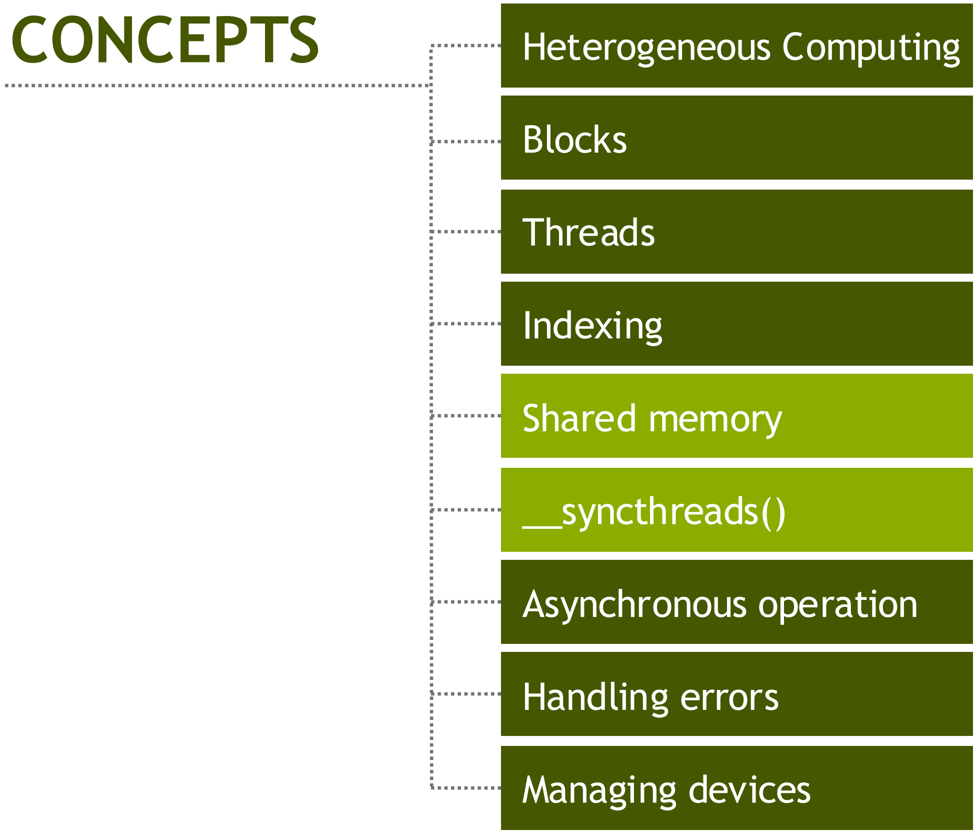
?

Why Bother with Threads?

- Threads seem unnecessary
  - They add a level of complexity
  - What do we gain?

- Unlike parallel blocks, threads have mechanisms to:
  - Communicate
  - Synchronize

- To look closer, we need a new example…

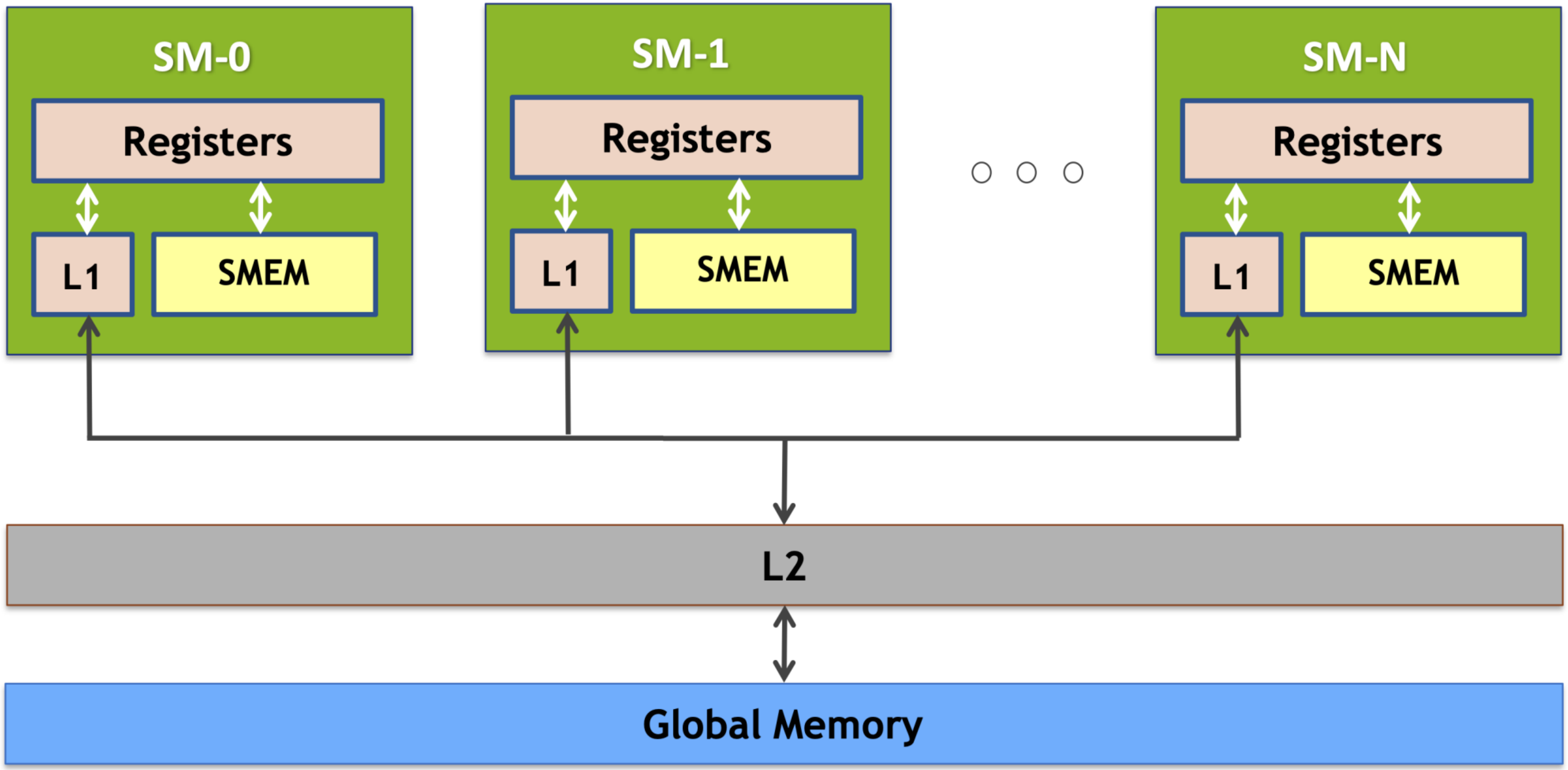


CONCEPTS

- Heterogeneous Computing
- Blocks
- Threads
- Indexing
- Shared memory
- __syncthreads()
- Asynchronous operation
- Handling errors
- Managing devices

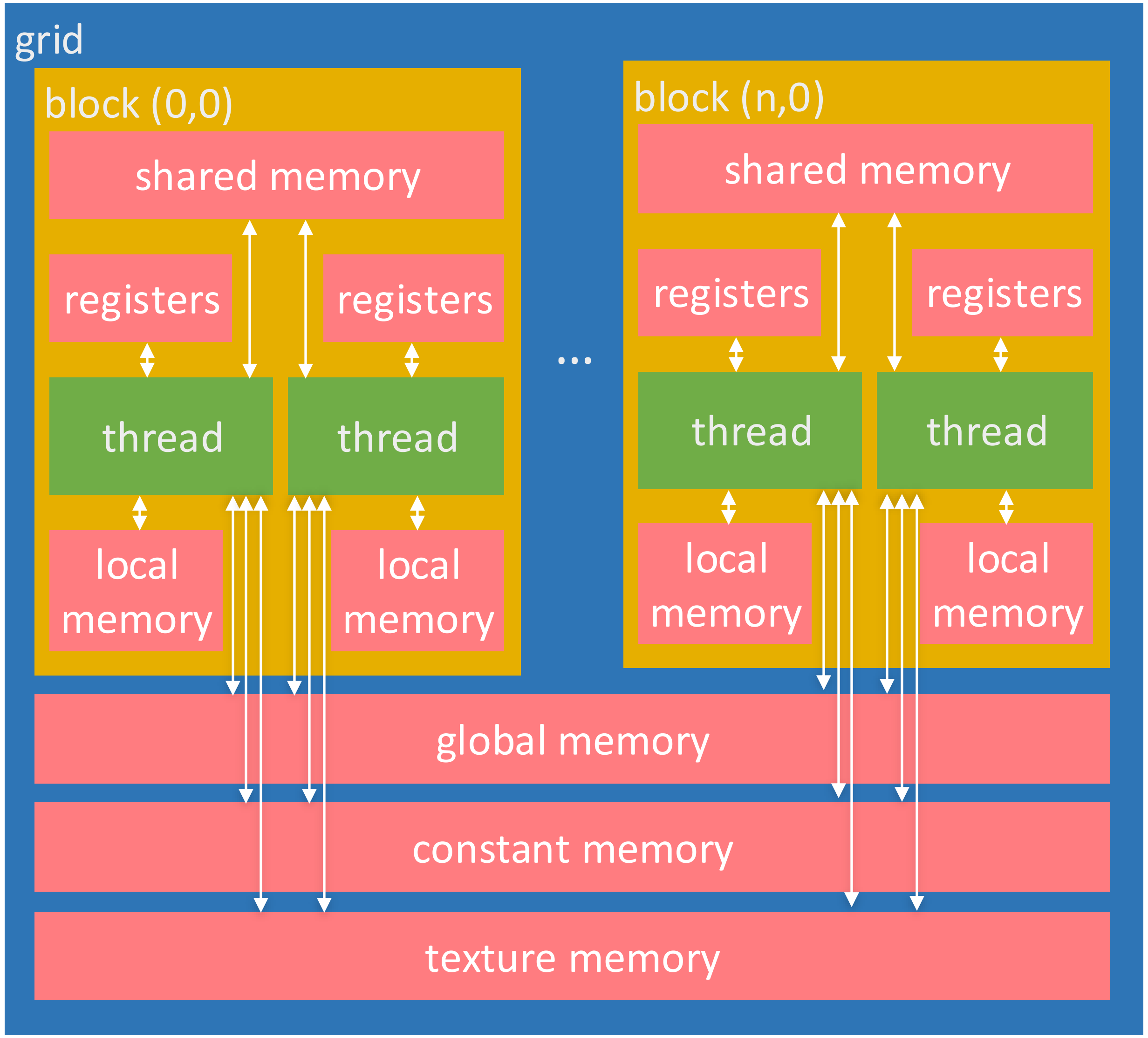
COOPERATING THREADS

GPU memory hierarchy



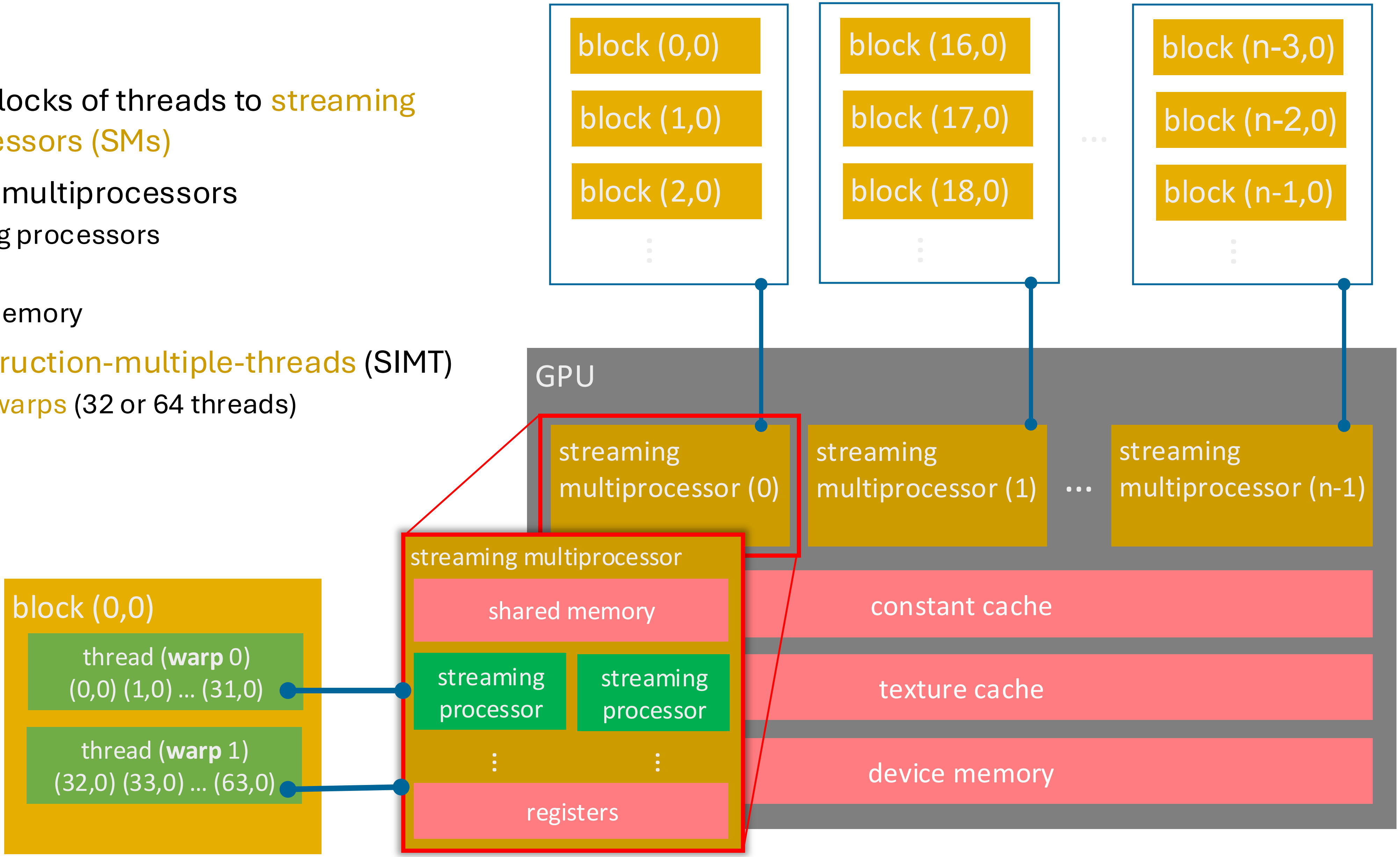
Memory Model

- Define memory hierarchy
- Registers (VGPR, SGPR)
  - Local variables per thread
  - The fastest memory
- Local memory
  - local variables per thread
  - Slow off-chip memory (VRAM)
  - Register spills if not enough registers
- Shared memory (Local data share)
  - Shared between threads in a block
  - Faster on-chip memory
- Global memory
  - Shared with all blocks
  - Largest memory
  - Slow off-chip memory (VRAM)
- Texture and constant memory
  - Cached differently, on off-chip memory



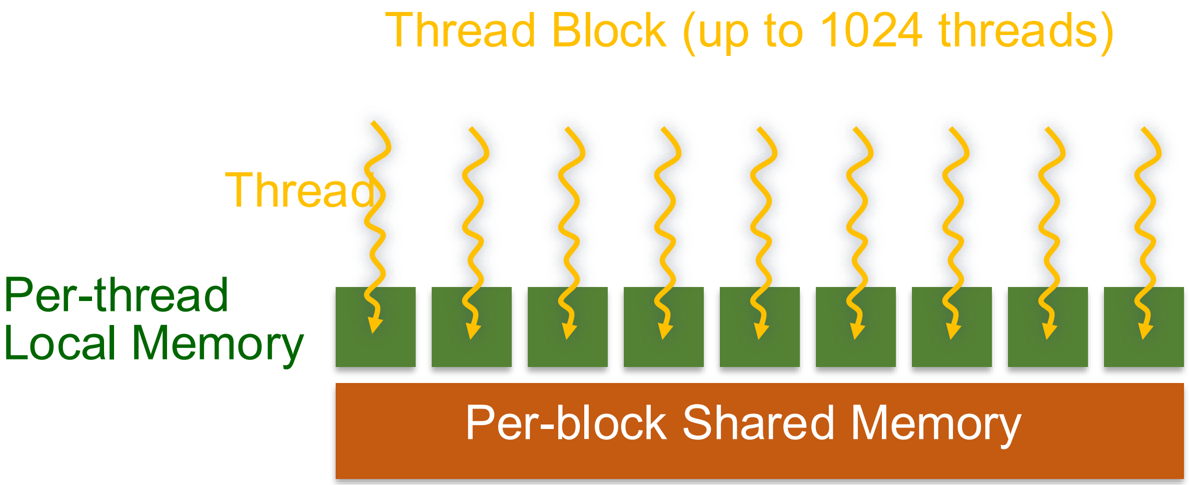
Execution Model

- Mapping blocks of threads to streaming multiprocessors (SMs)
- Streaming multiprocessors
  - Streaming processors
  - Registers
  - Shared memory
- Single-instruction-multiple-threads (SIMT)
  - Process warps (32 or 64 threads)



Shared (within a block) Memory

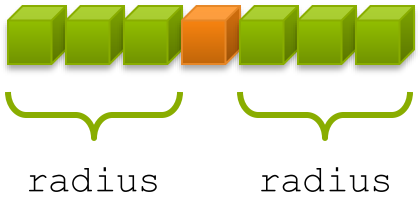
- Declare using __shared__, allocated per block
- Fast on-chip memory, user-managed
- Not visible to threads in other blocks



1D Stencil

- Consider applying a 1D stencil to a 1D array of elements
  - Each output element is the sum of input elements within a radius

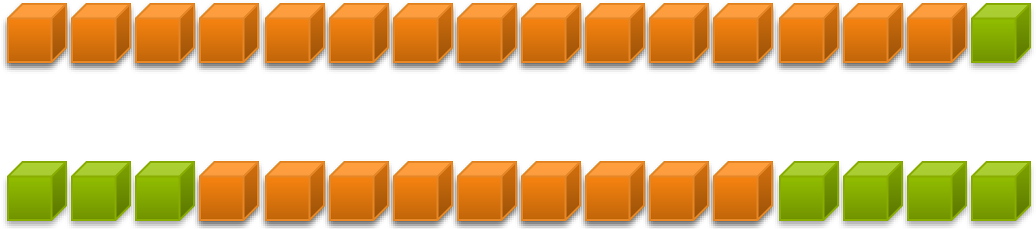
- If radius is 3, then each output element is the sum of 7 input elements:



Implementing Within a Block

- Each thread processes one output element
  - blockDim.x elements per block

- Input elements are read several times
  - With radius 3, each input element is read seven times



© 2025–2026 by the Xuhao Chen

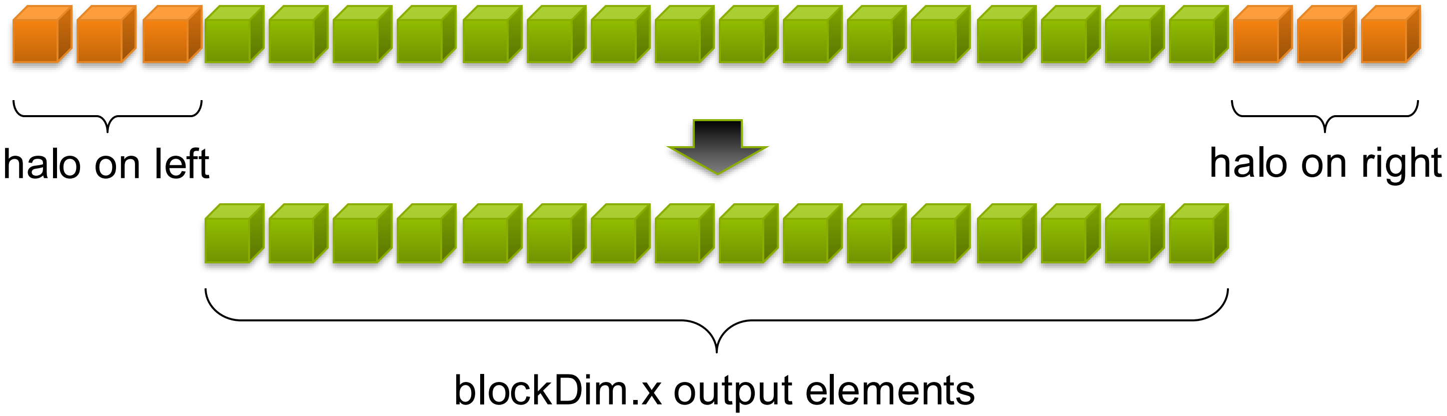
Sharing Data Between Threads

- Terminology: within a block, threads share data via shared memory
- Extremely fast on-chip memory, user-managed
- Declare using `__shared__`, allocated per block
- Data is not visible to threads in other blocks

Implementing With Shared Memory

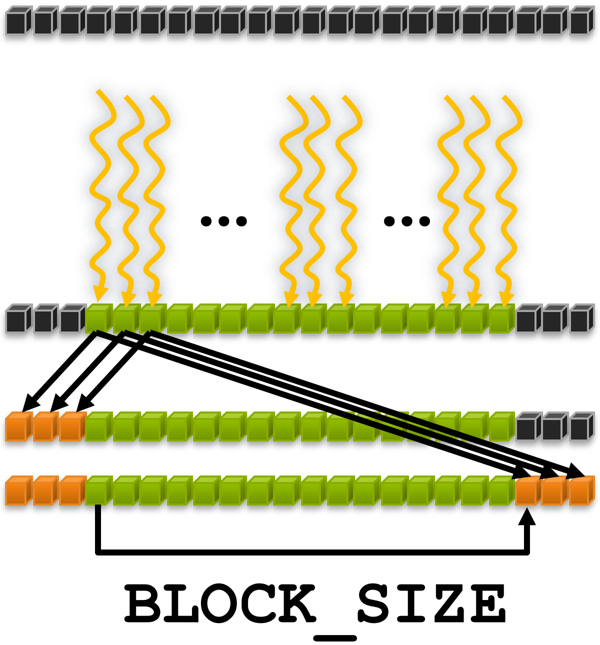
- Cache data in shared memory (N = `blockDim.x`)
  - Read (N + 2 * radius) input elements from global memory to shared memory
  - Compute N output elements
  - Write N output elements to global memory

  - Each block needs a halo of radius elements at each boundary



Stencil Kernel

```
__global__ void stencil_1d(int *in, int *out) {
  __shared__ int temp[BLOCK_SIZE + 2 * RADIUS];
  int gindex = threadIdx.x + blockIdx.x * blockDim.x;
  int lindex = threadIdx.x + RADIUS;

  // Read input elements into shared memory
  temp[lindex] = in[gindex];
  if (threadIdx.x < RADIUS) {
    temp[lindex - RADIUS] = in[gindex - RADIUS];
    temp[lindex + BLOCK_SIZE] =
      in[gindex + BLOCK_SIZE];
  }
```



Stencil Kernel

```
    // Apply the stencil
    int result = 0;
    for (int offset = -RADIUS ; offset <= RADIUS ; offset++)
      result += temp[lindex + offset];

    // Store the result
    out[gindex] = result;
}
```

Any problem?

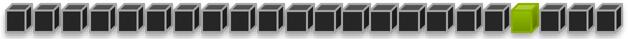
Data Race!

- The stencil example will not work…

- Suppose thread 15 reads the halo before thread 0 has fetched it…

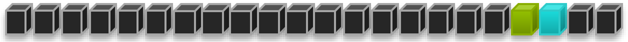
```
temp[lindex] = in[gindex];
if (threadIdx.x < RADIUS) {
  temp[lindex - RADIUS = in[gindex - RADIUS];
  temp[lindex + BLOCK_SIZE] = in[gindex + BLOCK_SIZE];
}

int result = 0;
result += temp[lindex + 1];
```

Store at temp[18]



Skipped, threadIdx > RADIUS

Load from temp[19]



__syncthreads()

- `void __syncthreads();`

- Synchronizes all threads within a block
  - Used to prevent RAW / WAR / WAW hazards

- All threads must reach the barrier
  - In conditional code, the condition must be uniform across the block

Stencil Kernel

```
__global__ void stencil_1d(int *in, int *out) {
    __shared__ int temp[BLOCK_SIZE + 2 * RADIUS];
    int gindex = threadIdx.x + blockIdx.x * blockDim.x;
    int lindex = threadIdx.x + radius;

    // Read input elements into shared memory
    temp[lindex] = in[gindex];
    if (threadIdx.x < RADIUS) {
        temp[lindex - RADIUS] = in[gindex - RADIUS];
        temp[lindex + BLOCK_SIZE] = in[gindex + BLOCK_SIZE];
    }

    // Synchronize (ensure all the data is available)
    __syncthreads();
```

Stencil Kernel

```
    // Apply the stencil
    int result = 0;
    for (int offset = -RADIUS ; offset <= RADIUS ; offset++)
        result += temp[lindex + offset];

    // Store the result
    out[gindex] = result;
}
```

Review (1 of 2)

- Launching parallel threads
  - Launch `N` blocks with `M` threads per block with `kernel<<<N,M>>>(…);`
  - Use `blockIdx.x` to access block index within grid
  - Use `threadIdx.x` to access thread index within block

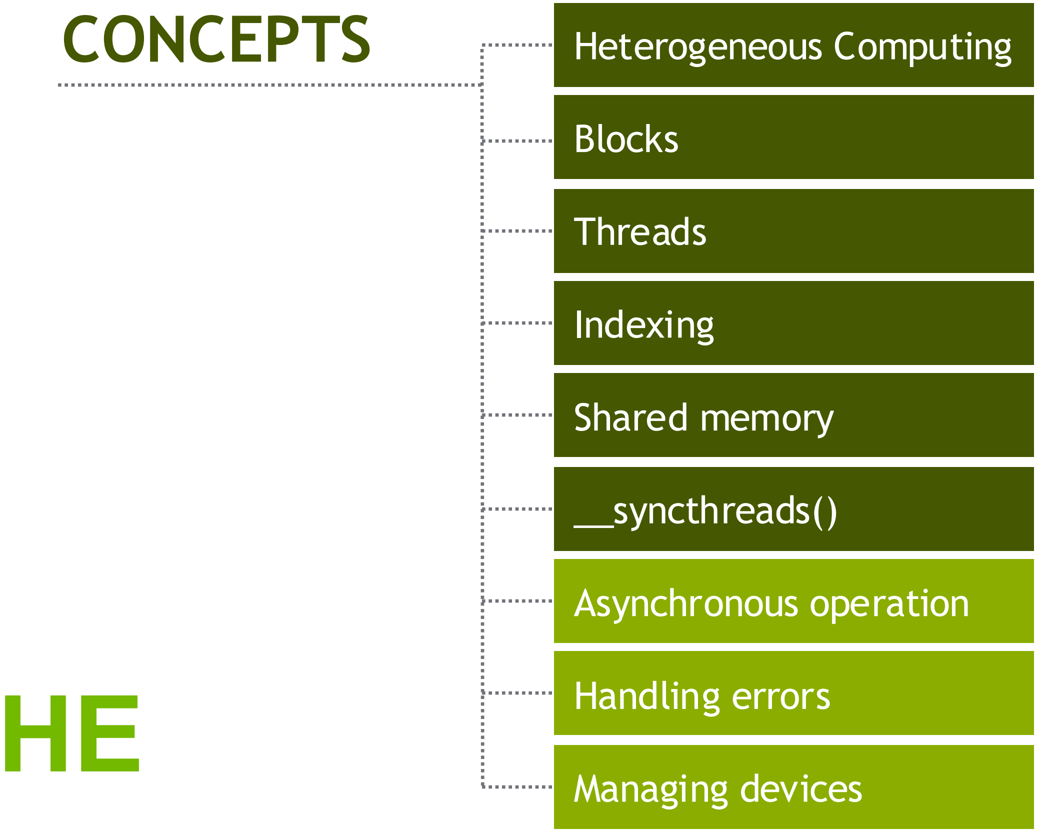
- Allocate elements to threads:

```
int index = threadIdx.x + blockIdx.x * blockDim.x
```

Review (2 of 2)

- Use `__shared__` to declare a variable/array in shared memory
  - Data is shared between threads in a block
  - Not visible to threads in other blocks

- Use `__syncthreads()` as a barrier
  - Use to prevent data hazards



CONCEPTS

Heterogeneous Computing

Blocks

Threads

Indexing

Shared memory

__syncthreads()

Asynchronous operation

Handling errors

Managing devices

MANAGING THE DEVICE

Coordinating Host & Device

- Kernel launches are asynchronous
  - Control returns to the CPU immediately

- CPU needs to synchronize before consuming the results

cudaMemcpy()	Blocks the CPU until the copy is complete Copy begins when all preceding CUDA calls have completed
cudaMemcpyAsync()	Asynchronous, does not block the CPU
cudaDeviceSynchronize()	Blocks the CPU until all preceding CUDA calls have completed

Reporting Errors

- All CUDA API calls return an error code (`cudaError_t`)
  - Error in the API call itself OR
  - Error in an earlier asynchronous operation (e.g. kernel)

- Get the error code for the last error:

```
cudaError_t cudaGetLastError(void)
```

- Get a string to describe the error:

```
char *cudaGetErrorString(cudaError_t)

printf("%s\n", cudaGetErrorString(cudaGetLastError()));
```

Device Management

- Application can query and select GPUs

```
cudaGetDeviceCount(int *count)
cudaSetDevice(int device)
cudaGetDevice(int *device)
cudaGetDeviceProperties(cudaDeviceProp *prop, int device)
```

- Multiple threads can share a device

- A single thread can manage multiple devices

  `cudaSetDevice(i)` to select current device

  `cudaMemcpy(…)` for peer-to-peer copies†

† requires OS and device support

CUDA Summary

- GPU Architecture for Data Parallelism
- GPU Programming Model: SIMT

- Write and launch CUDA C/C++ kernels
  - `__global__`, `blockIdx.x`, `threadIdx.x`, `<<<>>>`

- Manage GPU memory
  - `cudaMalloc()`, `cudaMemcpy()`, `cudaFree()`

- Manage communication and synchronization
  - `__shared__`, `__syncthreads()`
  - `cudaMemcpy()` vs. `cudaMemcpyAsync()`
  - `cudaDeviceSynchronize()`