GPGPU Programming with CUDA

Trung Nguyen, Ph.D.
Computational Scientist
Research Computing Center, University of Chicago

Credit: John Stone (ICTP 2015), Tim Warburton (ATPESC 2017),
Peng Wang, Stephen Jones (NVIDIA, GTC 2017)

What will you learn today?

1. General-purpose GPUs
   - Heterogeneous computing
   - GPU architecture
2. CUDA programming basics
   - Threads
   - Kernels
   - Memory hierarchy
   - Timing measurement
3. Read, build and run simple CUDA codes

There will be quizzes!

Hands-on Exercises

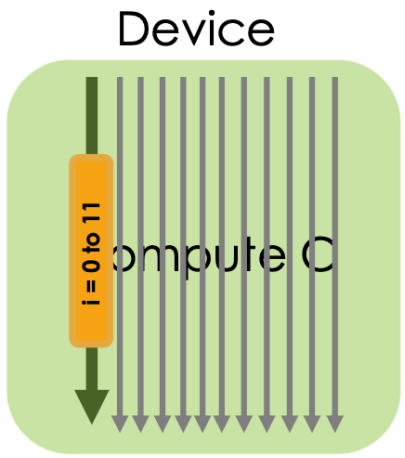
- Simple CUDA code git clone https://github.com/rcc-uchicago/GPU-computing
  - ex1-scale
  - ex2-vectorAdd
  - ex3-sharedmem
  - ex4-reduction
  - ex5-matMul
  - ex6-async
  - ex7-mpi
- Mini projects (optional):
  - Radial distribution functions
  - Histograms
  - Jacobi solver for Poisson's equation

GPU offload with OpenMP 4.0+

Swaroop Pophale, CSMD

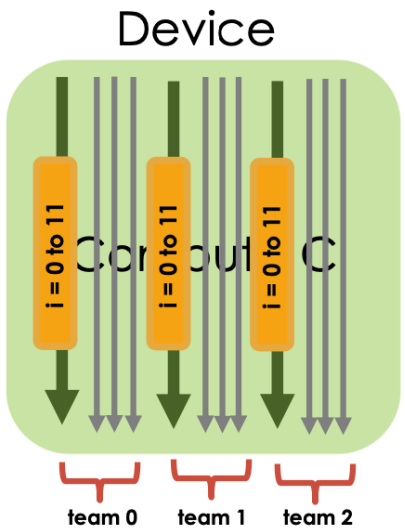
target

```
#pragma omp target
for (int i = 0; i < 12; ++i)
{
    C[i] = A[i] + B[i];
}
```



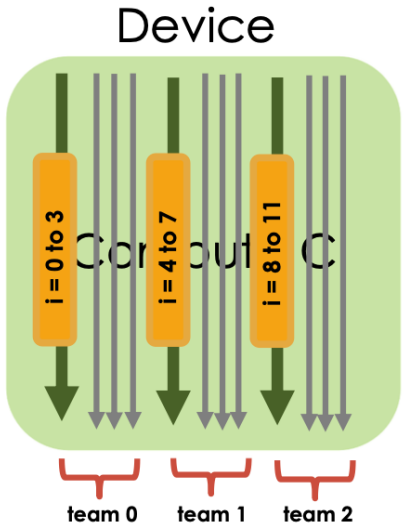
target teams

```
#pragma omp target teams
num_teams(3)
for (int i = 0; i < 12; ++i)
{
    C[i] = A[i] + B[i];
}
```



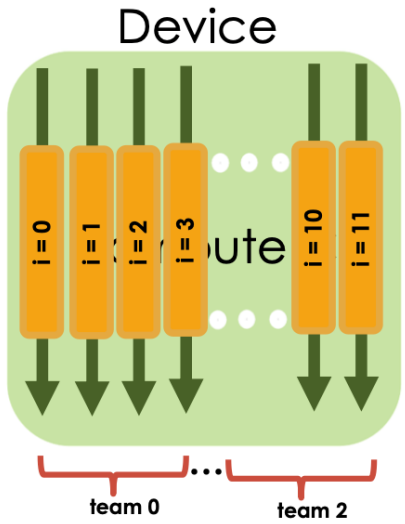
target teams distribute

```
#pragma omp target teams
distribute num_teams(3)
for (int i = 0; i < 12; ++i)
{
    C[i] = A[i] + B[i];
}
```



target teams distribute parallel

```
#pragma omp target teams
distribute parallel for
num_teams(3)
for (int i = 0; i < 12; ++i)
{
    C[i] = A[i] + B[i];
}
```

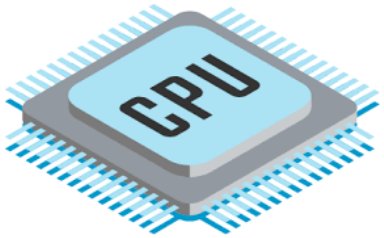


Part 1: General-Purpose GPUs

Traditional GPUs are designed for graphics

General-Purpose Graphics Processing Units (GPGPUs)

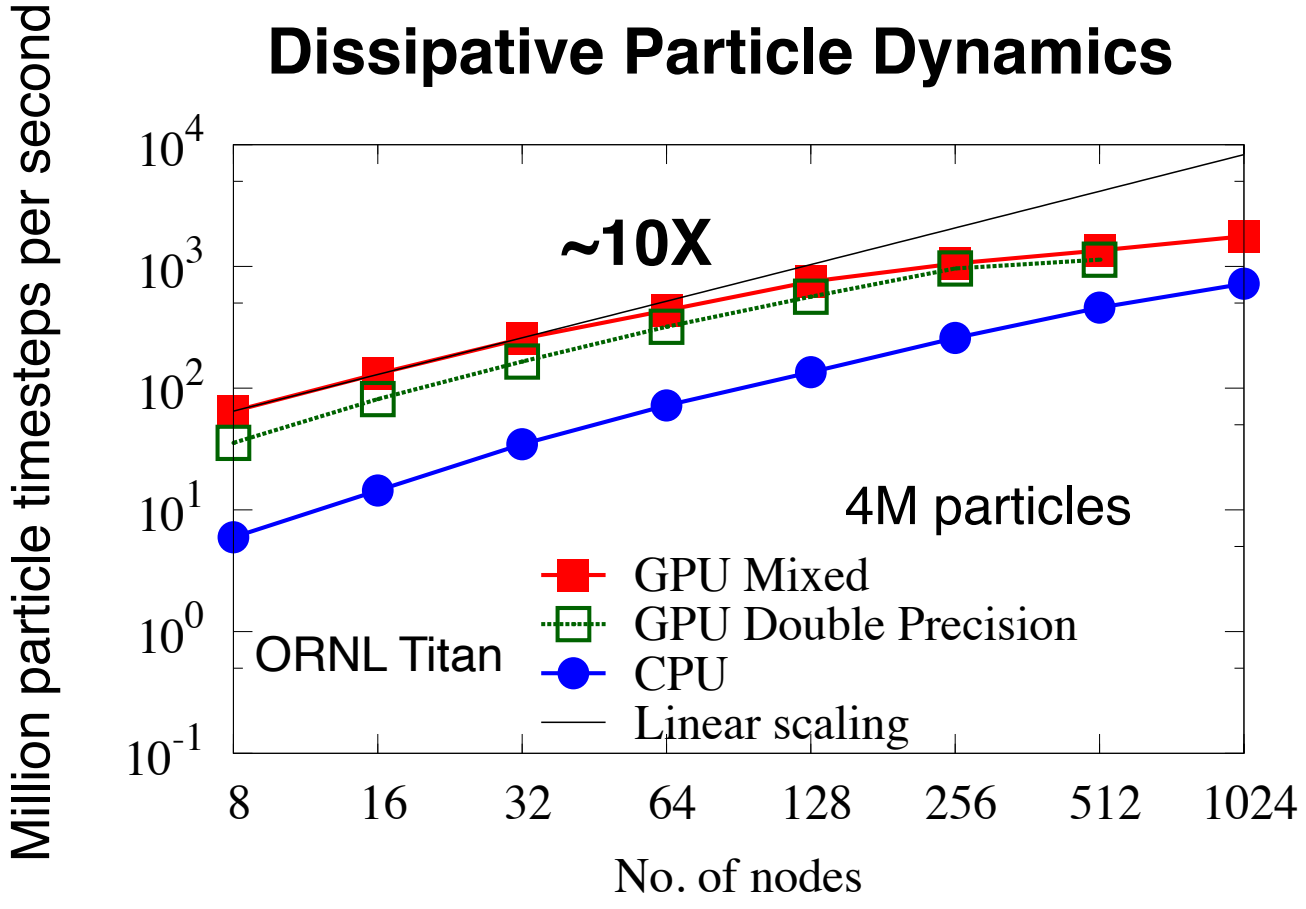
designed for data-parallel tasks



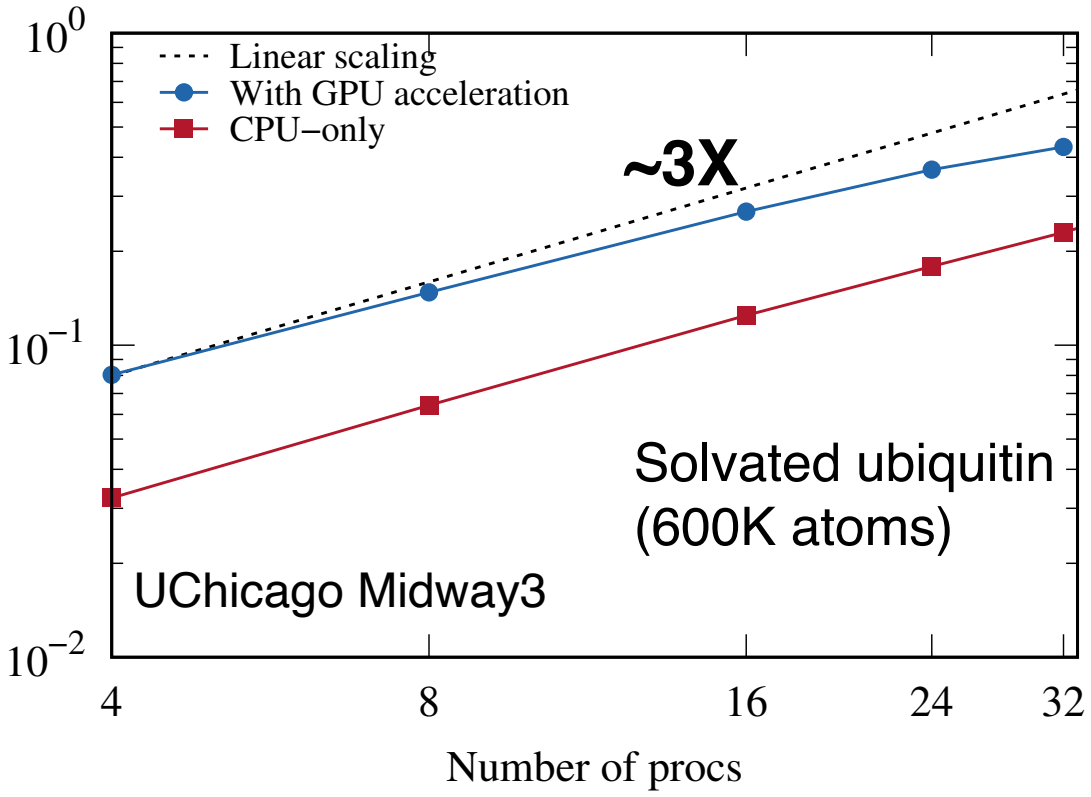
- ✓ Physical Modeling
- ✓ Materials Design
- ✓ Signal Processing
- ✓ Bitcoin Mining
- ✓ Machine Learning/AI

used as co-processors with CPUs for computing and data processing

Soft matter modeling: GPU package in LAMMPS



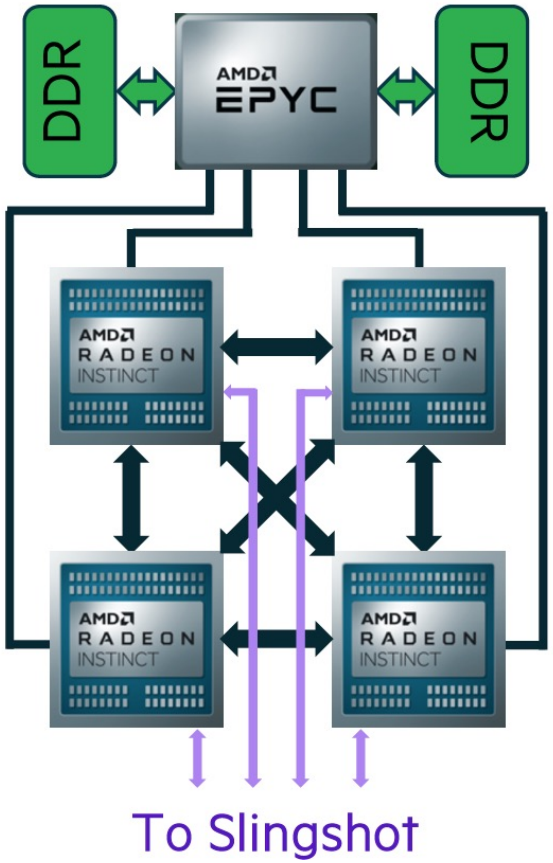
TDN, Plimpton, *Comput. Mat. Sci.*, 100, 173, 2015



LAMMPS GitHub, PR #3599, Jan 2023

Hybrid computing nodes = Multicore CPUs + GPUs

from workstations, data centers, to supercomputers



ORNL

ANL

CINECA

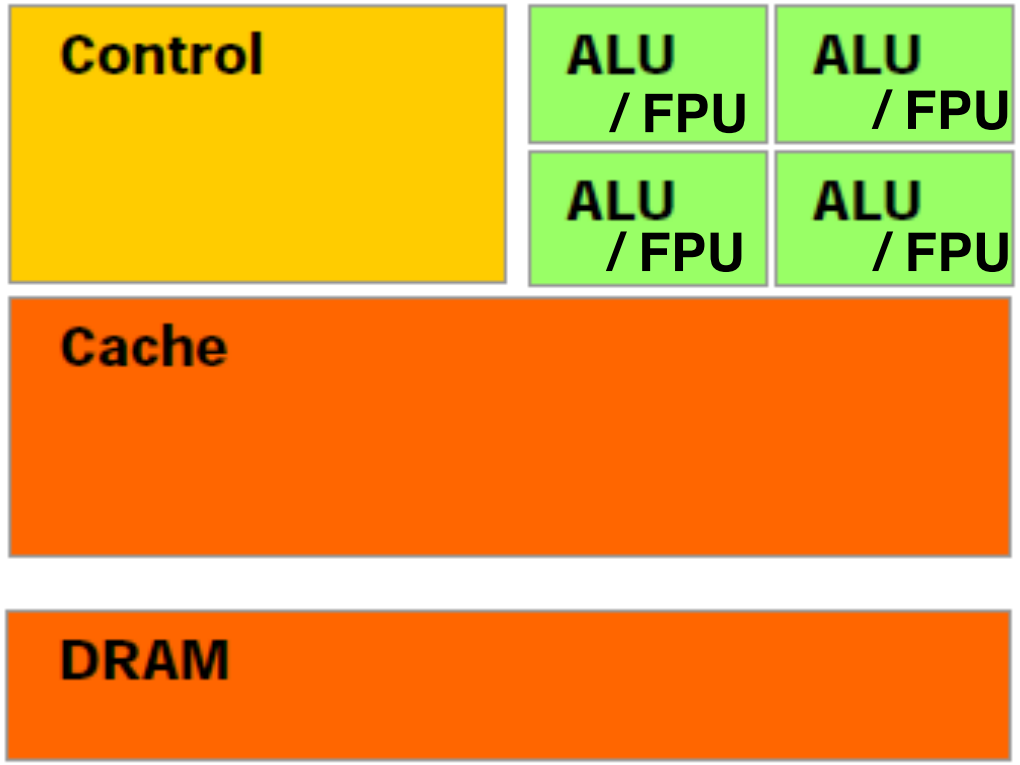
https://www.olcf.ornl.gov/frontier/

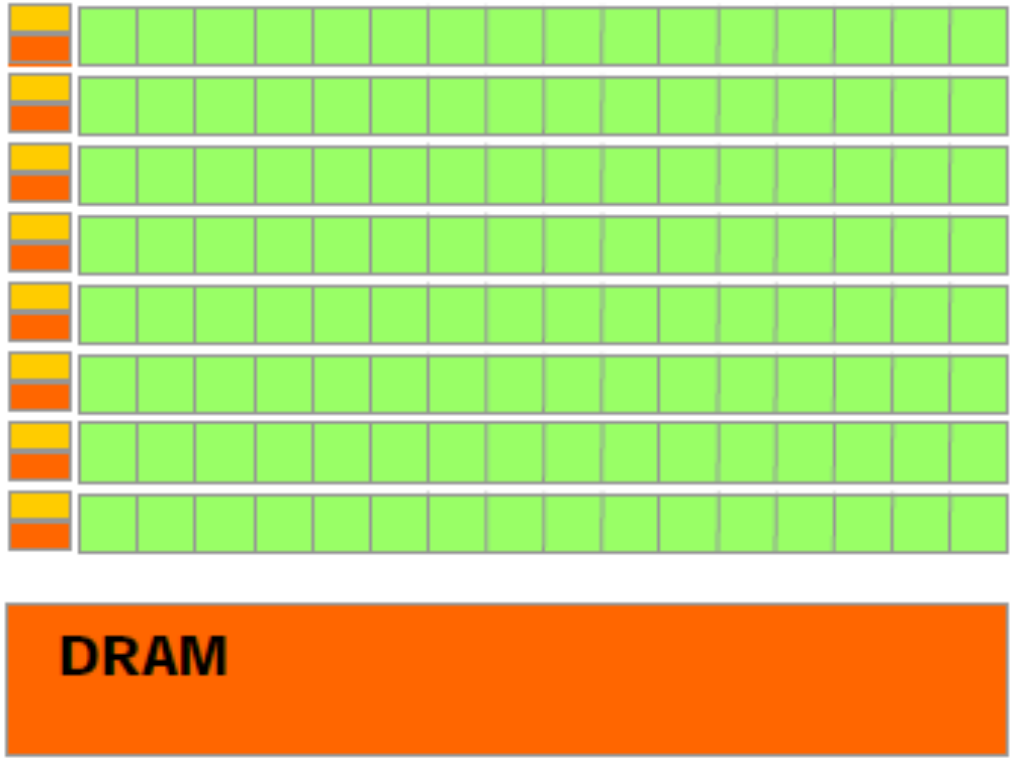
GPU architecture is devoted to massively parallel data processing

CPU: Focused on individual thread performance
- Large cache per core
- Dedicated units for logic and control

GPU: focused on high throughput
- a lot of ALUs and FPUs, few logic and control units
- thousands of active threads to hide memory latency

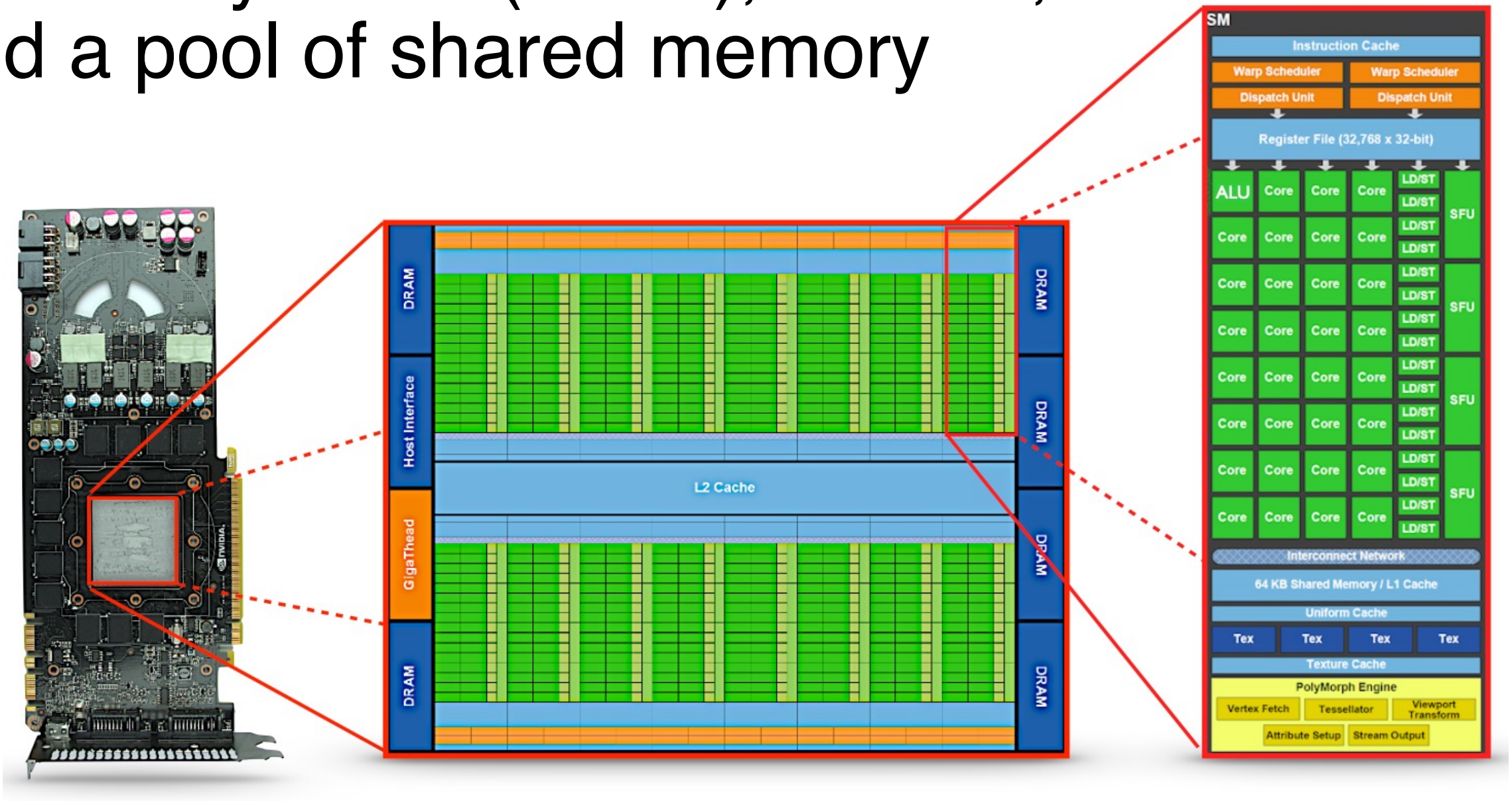
ALU = Arithmetic Logic Unit / FPU = Floating-Point Unit





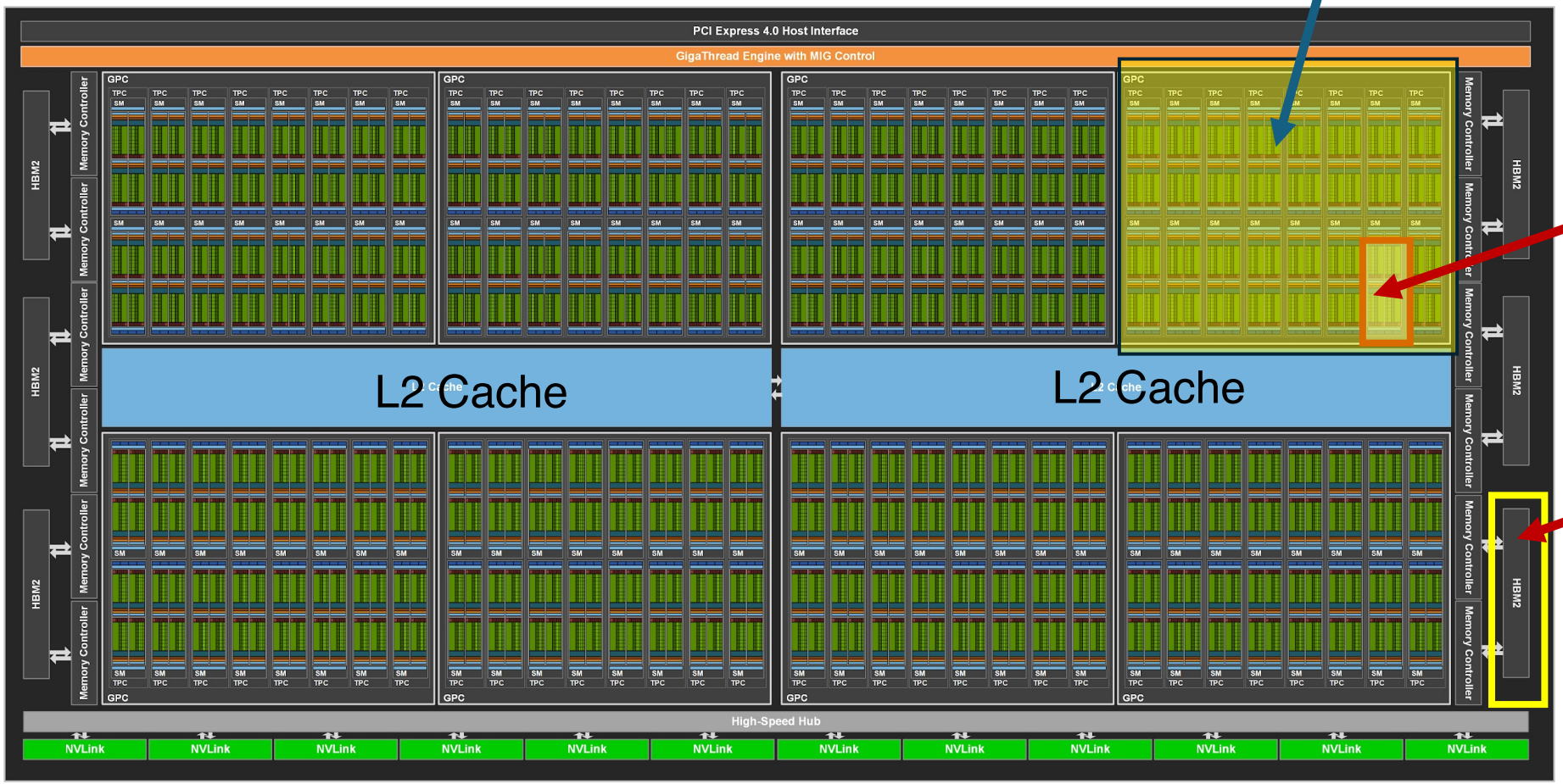
with many ALUs (cores), caches, and a pool of shared memory

NVIDIA terms ALUs/FPUs as “CUDA cores”



NVIDIA Ampere GPUs (2020)

Graphics Processor Cluster (GPC)



Streaming Multiprocessor (SM)

Memory

leonardo: A100 GPU, 8 GPCs, each with 16 SMs
galileo100: V100 GPU, 6 GPCs, each with 14 SMs

Streaming Multiprocessor (SM)

SIMD = Single Instruction, Multiple Data

32-wide SIMD cluster

4 SIMD clusters

32 ALU **32** FP32 8 FP64

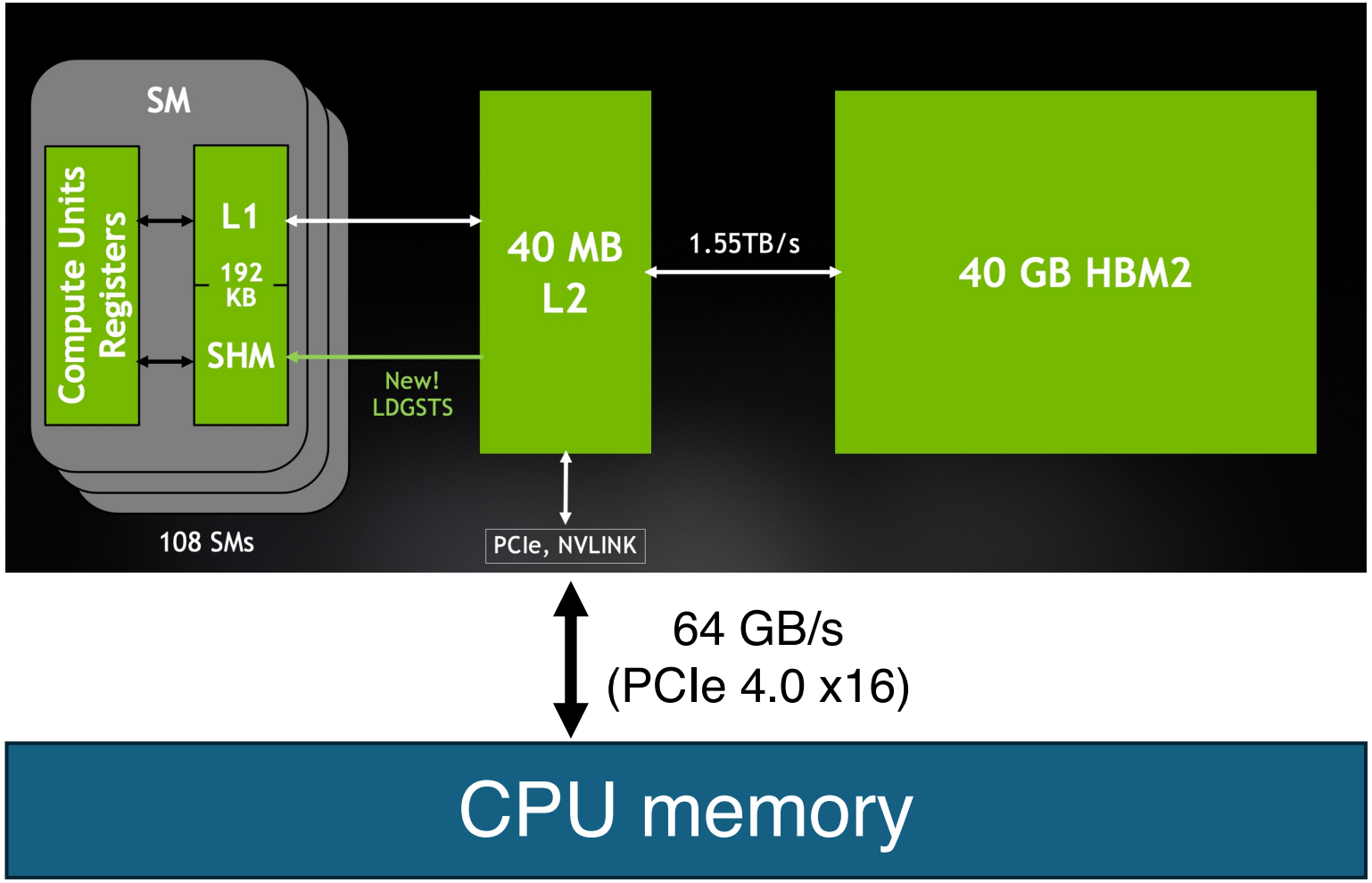
core clock rate 1.4 GHz
(1.4×10^9 clock cycles/sec)

Fundamental differences between GPUs vs CPUs

- The FPUs on each SM **ONLY** execute vector (SIMD) operations
  - CPU cores may execute serial, or vector operations
- The FPUs within a SM share big L1 cache
  - CPU register files are small (O(100)) vs 192K on A100
- Each SM is designed to switch between threads very quickly to hide stalls (due to memory and instruction latency).

Memory hierarchy: the closer to the cores, the faster

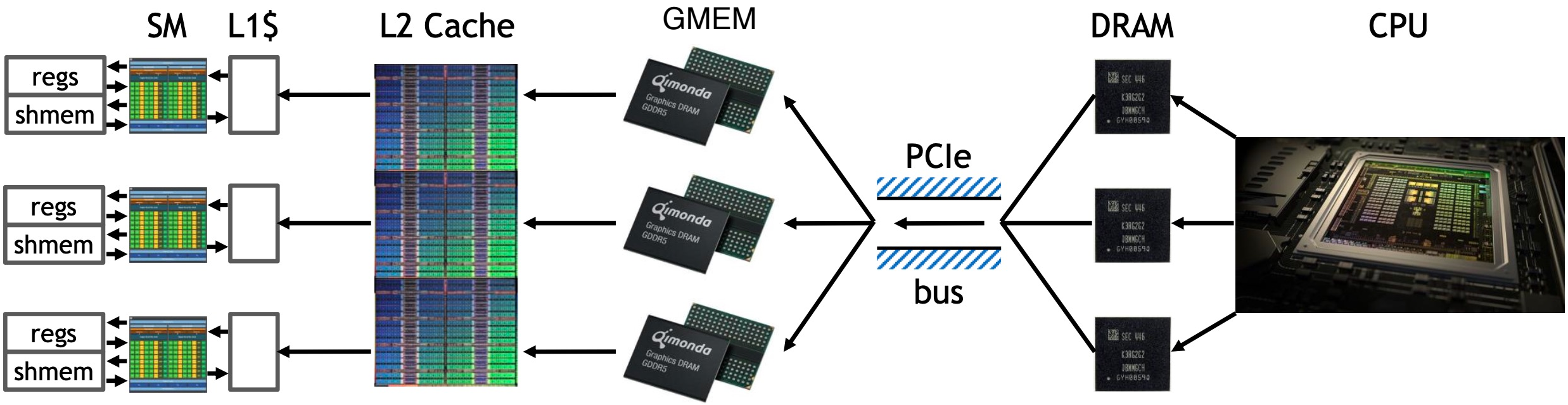
1. **On-chip**
   - **Registers** private to each thread (1 clock cycle)
   - **L1 cache + Shared memory** private to each block (30 cycles)
2. **Off-chip**
   - **L2 cache** (~200 cycles)
   - **Global memory** (~400 cycles)
   - 1.6 TB/s for GDDR5, or 6.4 TB/s for HBM2



Arithmetic operations:
- a*b+c (FMA): 4 cycles
- others 18-22 cycles

https://developer.nvidia.com/blog/cuda-refresher-cuda-programming-model
https://timdettmers.com/2023/01/30/which-gpu-for-deep-learning

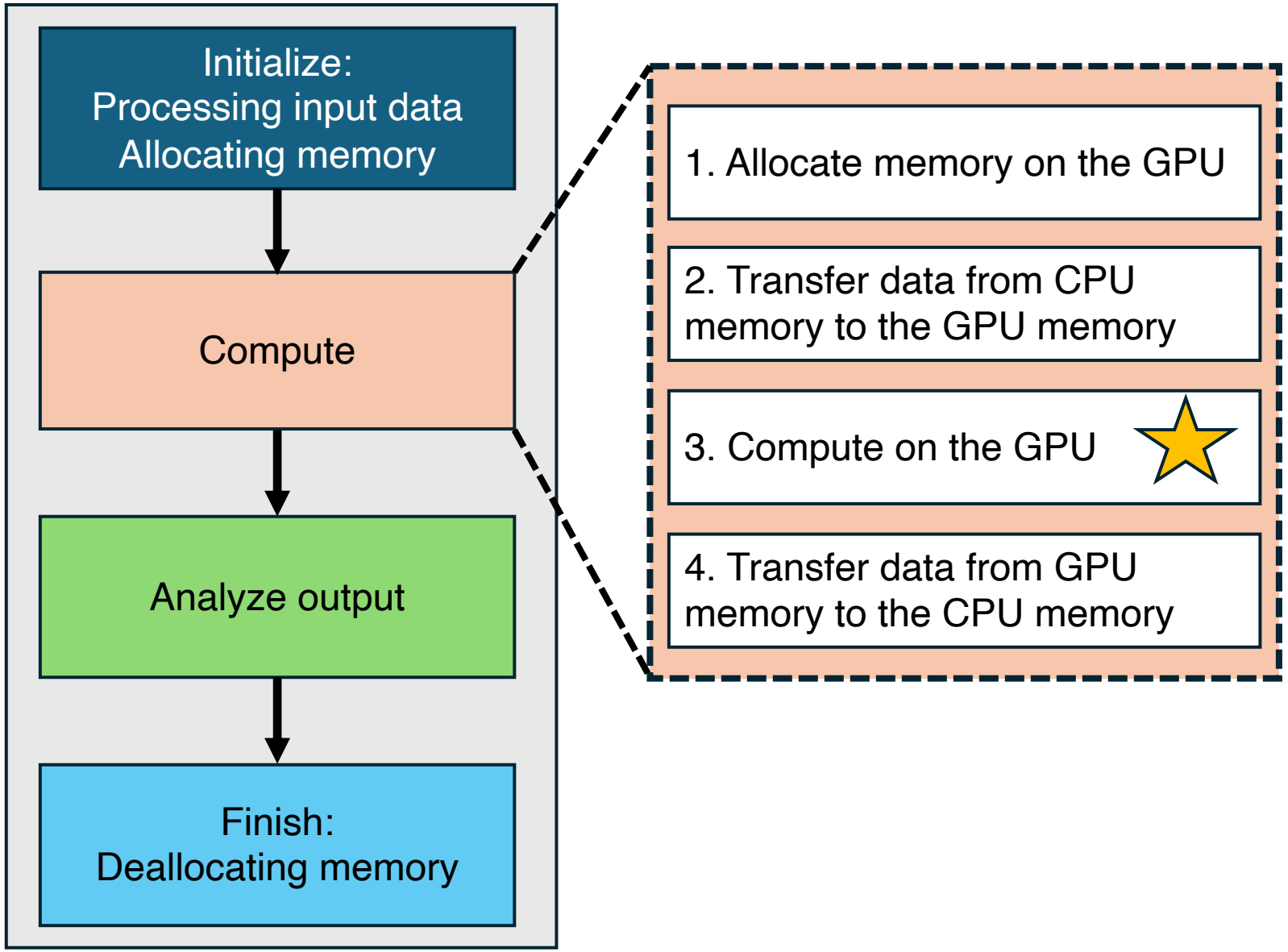
Memory bandwidths in perspectives



Stephen Jones, NVIDIA, GTC 2017

Typical workflow with GPU acceleration

a process or a thread on the CPU



Recall MPI and OpenMP lectures

Quiz

What is SIMD?

A) Similar Instances, More Data

B) Single Instruction, Multiple Data

C) Streaming Instruction, Multiple Devices

Quiz

Each CUDA core is

A) a streaming processor
B) a SIMD cluster
C) an ALU, or an FPU
D) none of the above

Quiz

What is the correct DECREASING order of bandwidth on the GPU?

A) L1/Register – Shared mem – L2 cache – Global mem

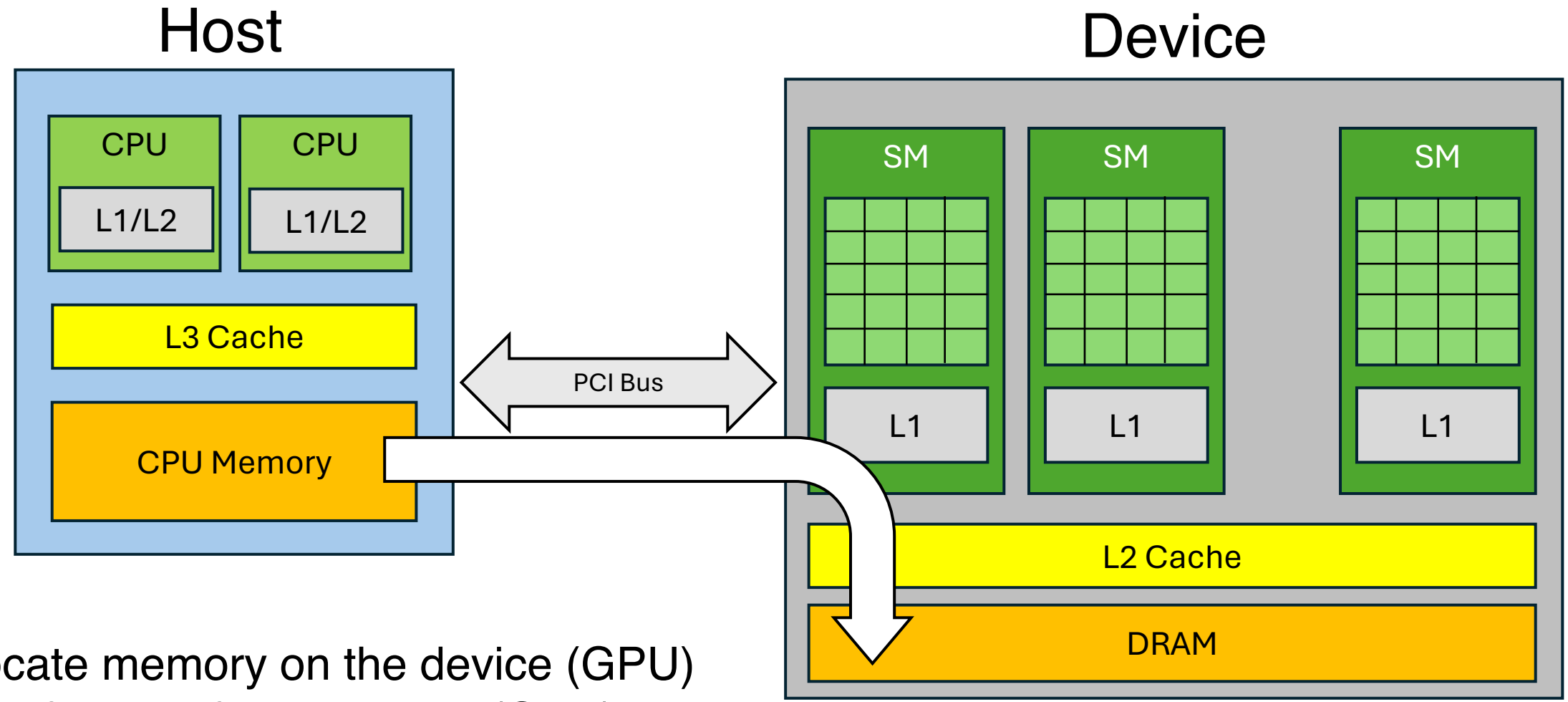
B) Global mem – Shared mem –- L1/Register – L2 cache

C) L1/Register – L2 cache – Global mem – Shared mem

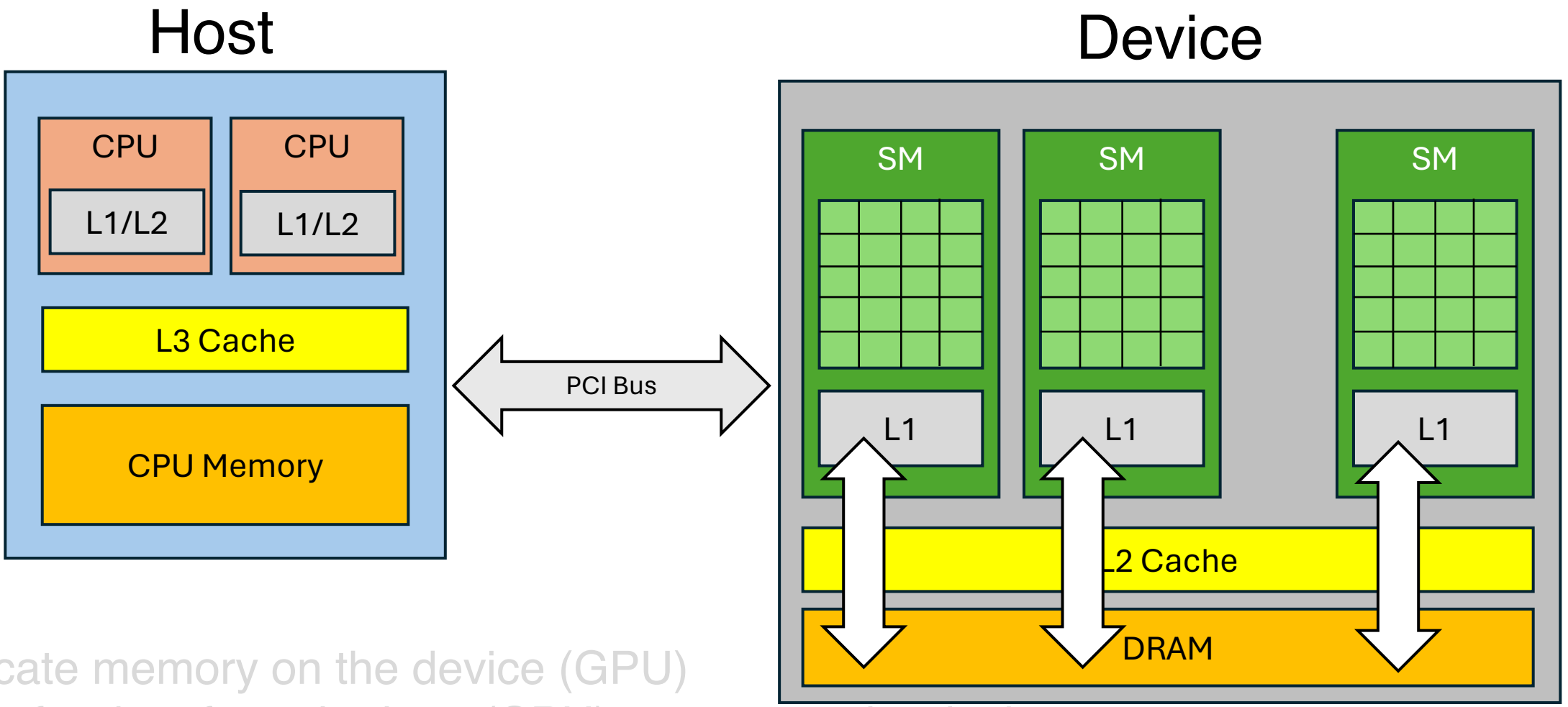
Key points to remember

1. Data transfers between CPU memory and GPU memory are always the bottlenecks in the whole workflow.
2. Global memory access is often the limiting factor in your kernel performance.
3. GPUs are of optimal use when all the SMs (and their ALUs) are busy computing, to hide memory latency.

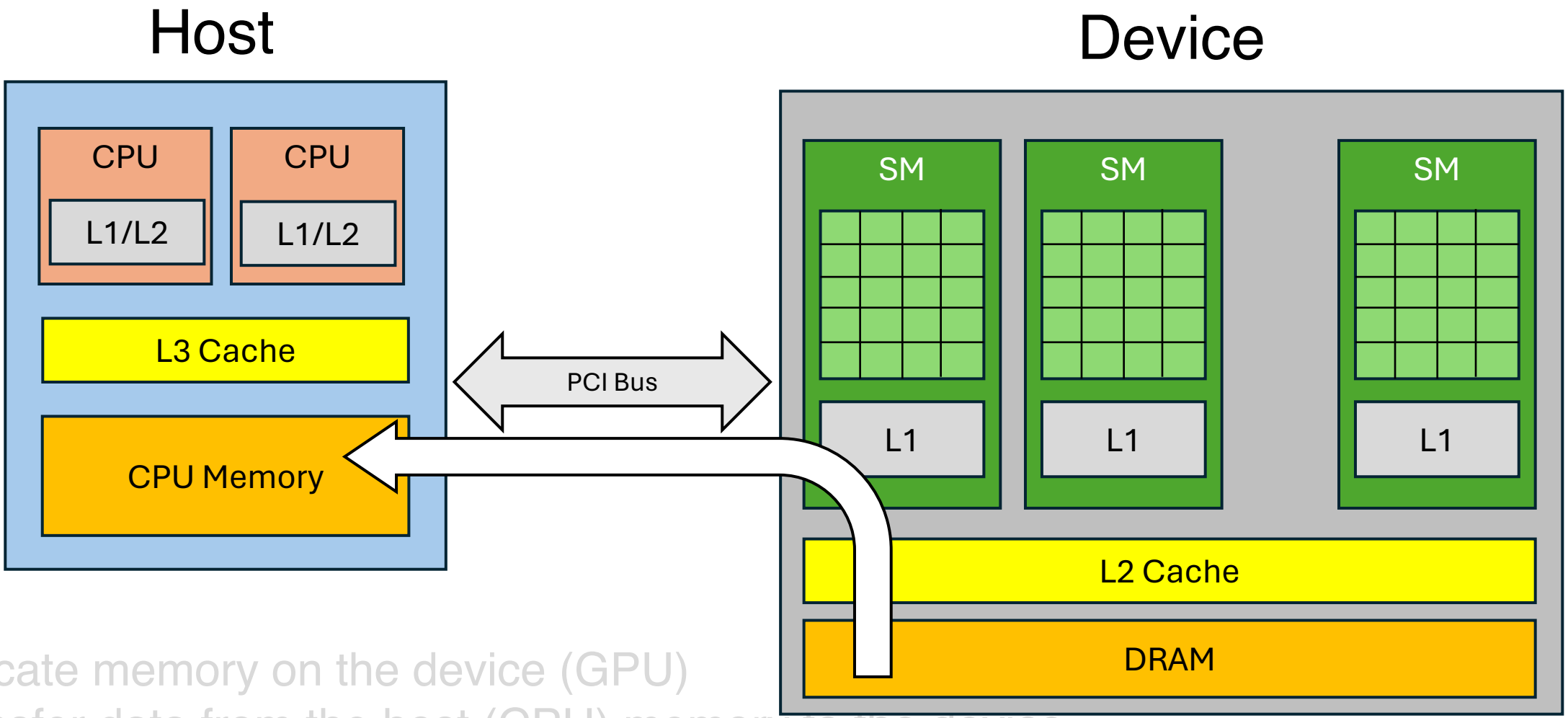
GPU computing step-by-step



1. Allocate memory on the device (GPU)
2. Transfer data from the host (CPU) memory to the device
3. Execute computation on the device
4. Copy data from the device memory back to the CPU memory

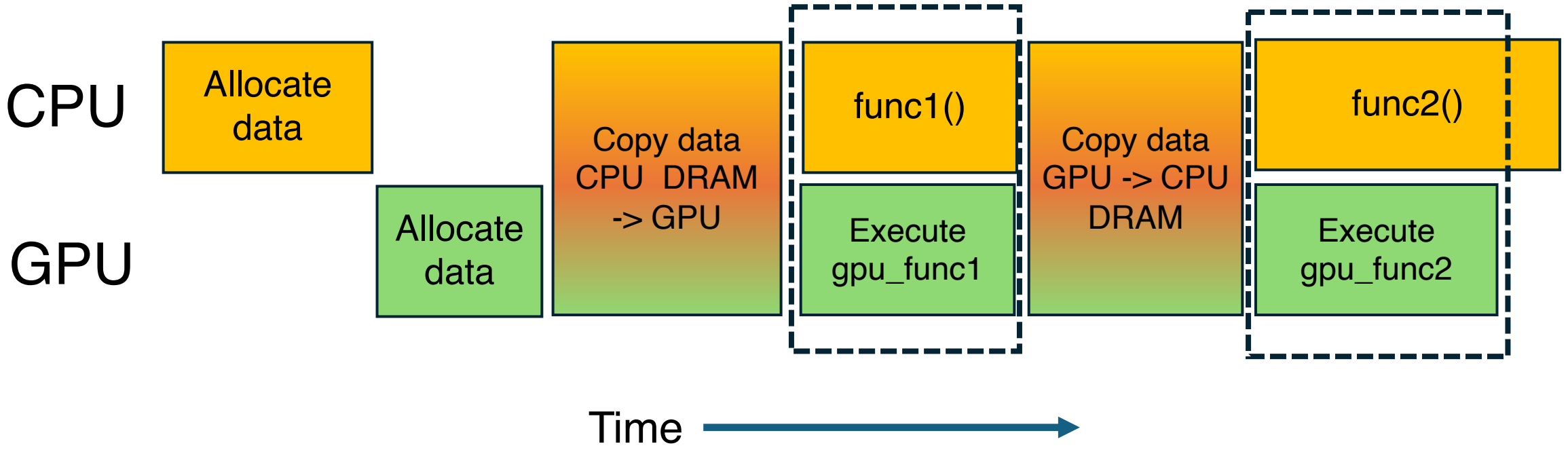


1. Allocate memory on the device (GPU)
2. Transfer data from the host (CPU) memory to the device
3. Execute computation on the device
4. Copy data from the device memory back to the CPU memory



1. Allocate memory on the device (GPU)
2. Transfer data from the host (CPU) memory to the device
3. Execute computation on the device
4. Copy data from the device memory back to the CPU memory

Executions on CPU and GPU may overlap



- Overlapping between CPU and GPU functions:
  - keep non-parallel data tasks on the CPU
  - regularize GPU workload (e.g. handling edge cases)

Key points to remember

1. Data transfers between CPU memory and GPU memory are always the bottlenecks in the whole workflow.
2. Global memory bandwidth is often the limiting factor.
3. GPUs are of optimal use when all the SMs (and their ALUs) are busy computing, to hide memory latency.

GPGPU programming frameworks

Libs/Compiler wrappers/Language extensions

Compiler directives

OpenCL™ NVIDIA CUDA kokkos SYCL™

OpenACC More Science, Less Programming OpenMP

NVIDIA GPUs

AMD GPUs

Intel GPUs

Key concepts shared among the frameworks

1. Single-Instruction Multiple-Data parallelism
2. Programming model
3. Memory management

Part 2: CUDA Programming Basics

What is NVIDIA CUDA?

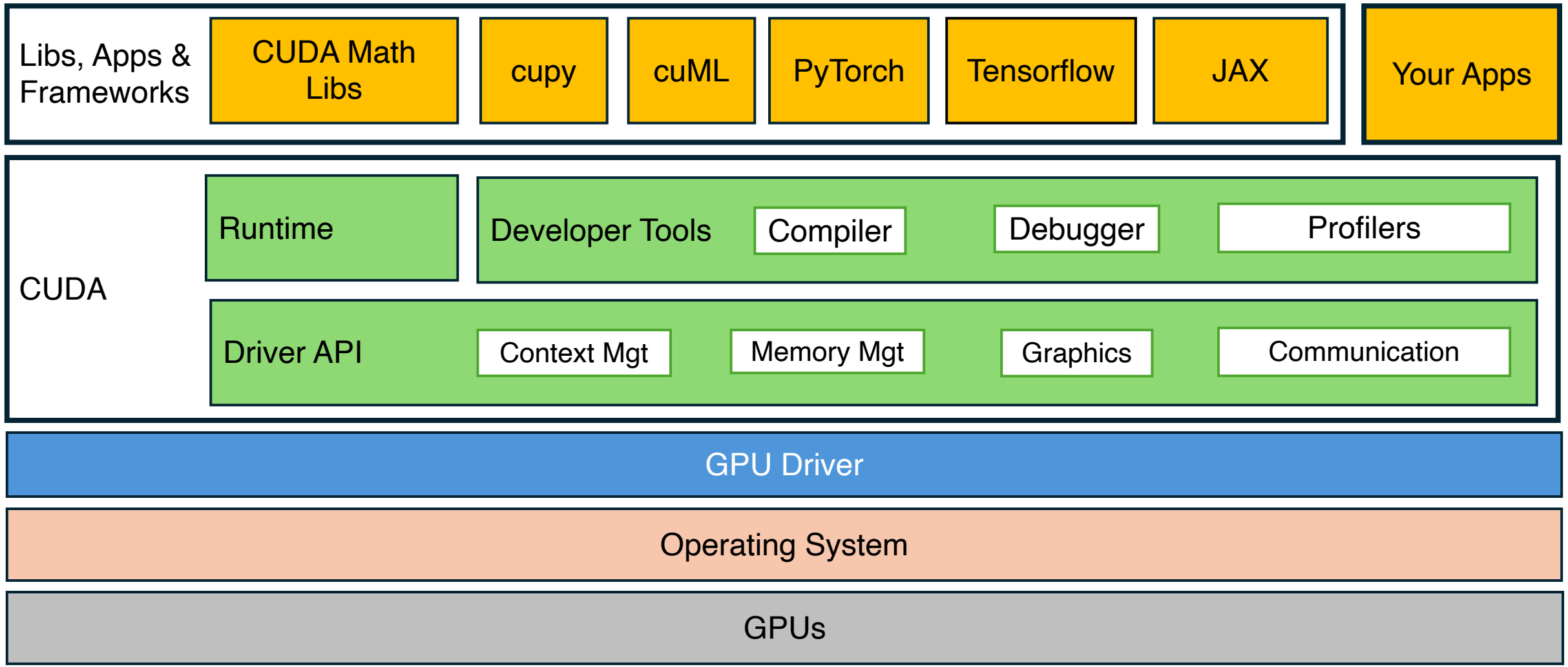
- CUDA = **Compute Unified Device Architecture**
  - first released in 2006, now version 12.4
  - CUDA Toolkit = a bundle of compilers, C/C++/Fortran API, debugger and profiler
  - free of charge, online documentation and tutorials
  - not easy to code, but suitable for learning GPU programming basics

GPU compute capability

Compute capability represents the features a GPU is capable of doing, so requires a minimum version of CUDA.

GPU	Compute Capability	CUDA minimum version
Pascal	6.0	8.0
Volta	7.x	9.0
Turing	7.5	10.0
Ampere	8.x	11.0
Hopper	9.0	11.8

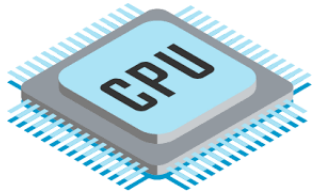
Software stack



https://blogs.nvidia.com/blog/what-is-cuda-2/

Programming Model: Host & Device

Host



DRAM

Device

Developer writes CUDA host and device codes to execute data-parallel tasks on the device.

Programming Model: Threads & Kernels

- A data-parallel task is expressed in terms of independent work items, called GPU threads.
- A kernel (i.e. a function) is a sequence of instructions to be executed by a GPU thread.
- Developer decides how many kernel instances are launched and mapped to the streaming multiprocessors on the GPUs.

Let’s look at a CPU code

```
void scale(float *x, float alpha, int N)
{
  for (int i = 0; i < N; i++) {
    x[i] = alpha * x[i];
  }
}
```

Work needs to be expressed into independent work items

- Thinking in parallel (recall OpenMP programming)
  all the N work items will be executed at the same time

Express the serial CPU code as groups of independent work items

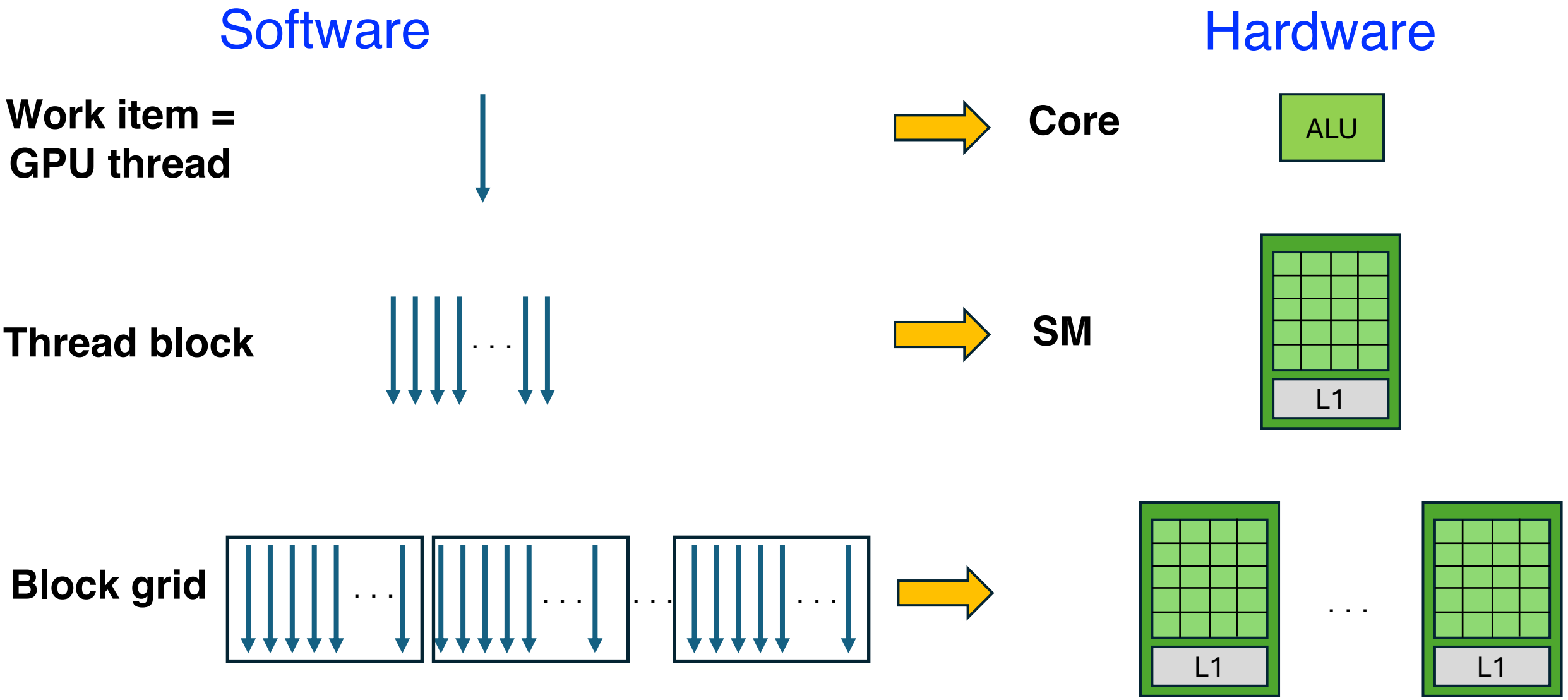
Let us divide the for loop over N into groups of work items, each working on a single element of the array

```
void scale(float *x, float alpha, int N)
{
  for (int b = 0; b < num_blocks; b++) {
    for (int t = 0; t < block_size; t++) {
      int tid = t + b * block_size;
      if (tid < N)
        x[tid] = alpha * x[tid];
    }
  }
}
```

a work item

`tid` is used for indexing the array `x`

CUDA maps thread hierarchy to hardware



Examples

- Scaling an array: ex1-scale
- Compile: with sm_80 for A100, sm_70 for V100
  - module load cuda
  - nvcc –arch=sm_80 scale.cu –o vec_scale

git clone https://github.com/Sera91/SMR3935-2024.git

Exercise #1: Scaling a vector by a scalar

```
int main(int argc, char** argv)
{
    float k = 10.0;

    // Print the vector length to be used, and compute its size
    int numElements = 50000;
    size_t size = numElements * sizeof(float);
    printf("Vector scaling of %d elements: k = %f\n", numElements, k);

    // Allocate the host input vector x
    float *h_x = (float *)malloc(size);

    // Initialize the host input vectors
    for (int i = 0; i < numElements; ++i) {
       h_x[i] = i;
    }
```

Allocate device array and copy data from the host

```
// Allocate the device input vector d_x
float *d_x = NULL;
cudaMalloc((void **)&d_x, size);

// Copy the host input vector h_x to the device input d_x
cudaMemcpy(d_x, h_x, size, cudaMemcpyHostToDevice);
```

CUDA functions

CUDA built-in constant indicating the direction of data transfer

CUDA functions for memory management

- Allocate data on the device
  cudaMalloc(&d_a, size_in_bytes)
- Allocate page-locked data on the host
  cudaMallocHost(&a, size_in_bytes)
- Copy data from host to device and vice versa
  cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice)
  cudaMemcpy(h_a, d_a, size, cudaMemcpyDeviceToHost)
- Free up the allocated arrays
  cudaFree(d_a)
  cudaFreeHost(a)

Host and device memory are separate entities

- **Host** pointers point to CPU memory
  - May be passed to/from device code
  - May *NOT* be dereferenced in device code
- **Device** pointers point to GPU memory
  - May be passed to/from host code
  - May *NOT* be dereferenced in host code
- Unified Virtual Memory*
  - needs page-locked host memory (cudaMallocHost)
  - make host-device transfers only when page faults occur

NVIDIA Introduction to CUDA C
*SYCL: Unified Shared Memory

Launch the kernel on the device

that is, put the kernel on the command queue on the device

```
// launch the kernel on the GPU
int threadsPerBlock = 256;
int blocksPerGrid = (numElements + threadsPerBlock-1)/threadsPerBlock;
scale<<< blocksPerGrid, threadsPerBlock >>>(d_x, k, numElements);
```

CUDA C extension

number of thread blocks

number of threads per block

- the total number of kernel instances to be executed = threadsPerBlock * blocksPerGrid
- each kernel instance is executed by a GPU thread, assigned to an ALU, on a SM.

Q1: Why choose threads per block = 256?

Q2: What if # of threads < numElements (like in OpenMP)? or >> numElements?

Kernels = functions to be executed on the device

CUDA C extension

kernel marker

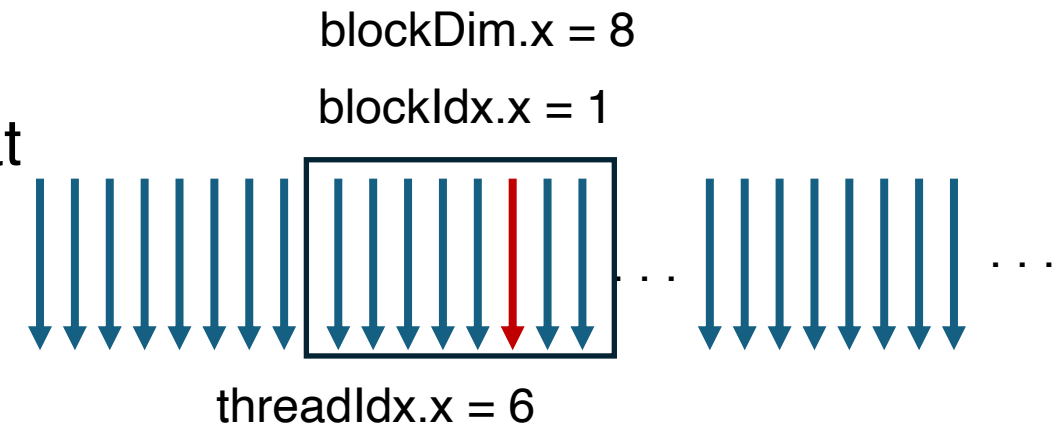
built-in variable index of a thread within a block

built-in variable indicates which block in a grid

built-in variable for the number of threads in a block

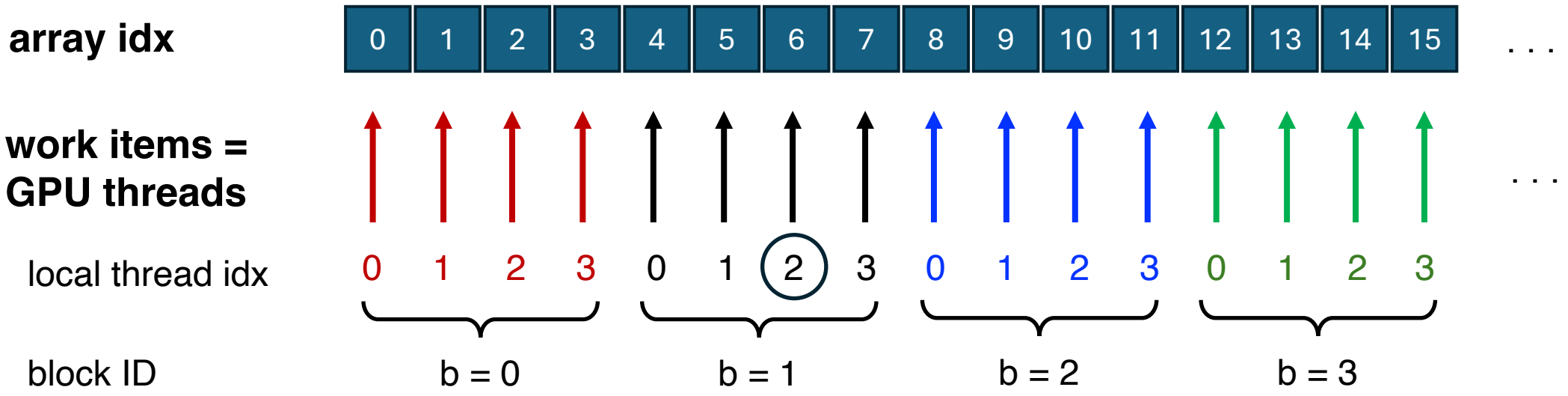
```
__global__ void scale(float* x, float alpha, const int N , float* y)
{
   int tid = threadIdx.x + blockIdx.x * blockDim.x;
   if (tid < N)
     x[tid] = alpha * x[tid];
}
```

- blockIdx, blockDim and threadIdx are available at runtime for a thread.



Indexing data with thread and block indices

`global thread_idx ← local thread idx + b * block_size`

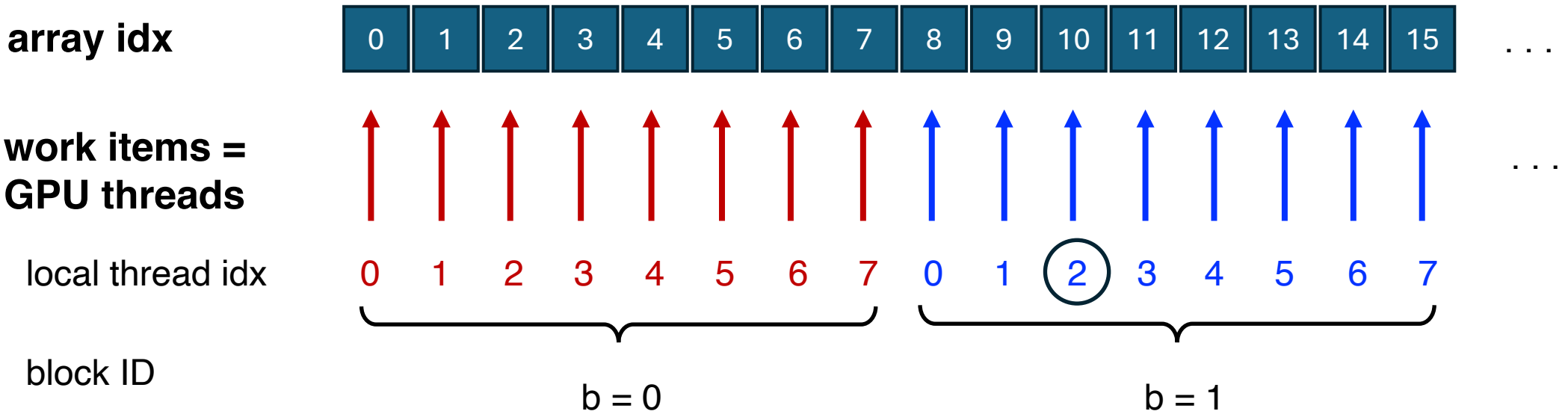


block_size = 4

(tid) global thread idx = 2 + 1 * 4 = 6

Indexing data with thread and block indices

`global thread_idx ← local thread idx + b * block_size`

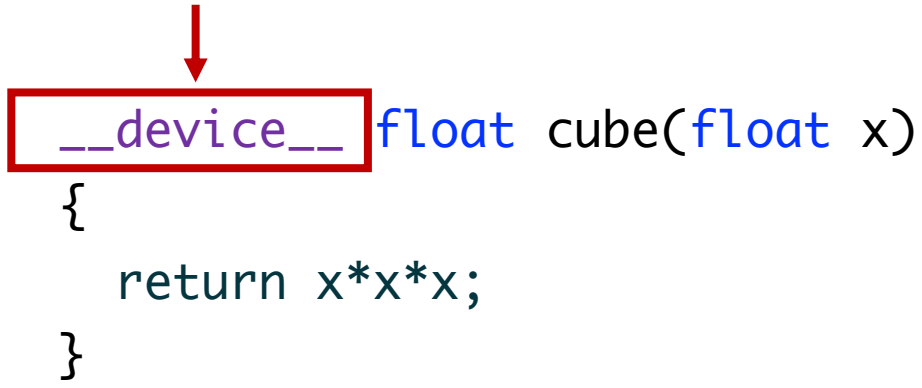


block_size = 256

tid global thread idx = 2 + 1 * 256 = 258

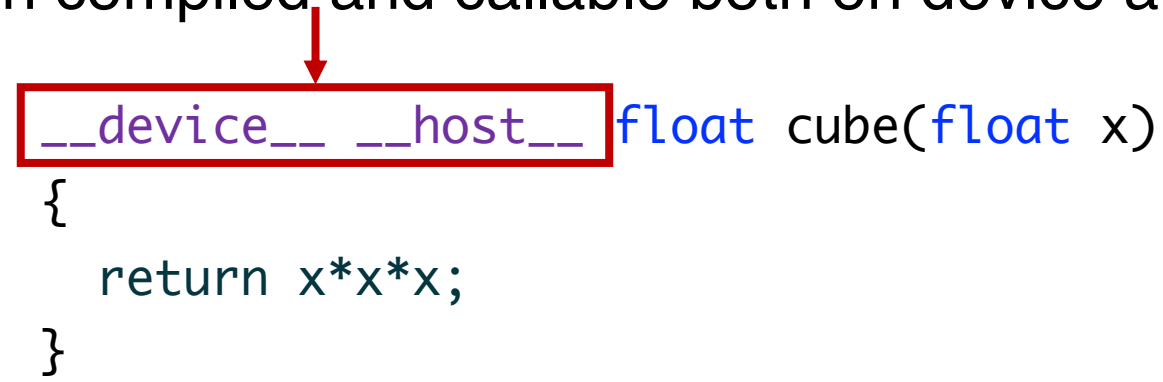
Kernel (function) modifiers

function complied as device code



```
__device__ float cube(float x)
{
  return x*x*x;
}
```

function compiled and callable both on device and host



```
__device__ __host__ float cube(float x)
{
  return x*x*x;
}
```

Bring output data back from GPU

```
// launch the kernel on the GPU
int threadsPerBlock = 256;
int blocksPerGrid = (numElements + threadsPerBlock-1)/threadsPerBlock;
scale<<< blocksPerGrid, threadsPerBlock >>>(d_x, k, numElements);

// Copy data from DEVICE to HOST
cudaMemcpy(h_a, d_a, N*sizeof(float), cudaMemcpyDeviceToHost);
```


built-in constant indicating the direction of transfer

Q: How do we know the scale() kernel complete before copying data back to host?

Compiling CUDA code with nvcc

module load cuda
Single source .cu file (containing both host code and device code)

```
nvcc -arch=sm_80 scale.cu -o myapp
```

with optimization

```
nvcc -O2 -use_fast_math -arch=sm_80 main.cu -o myapp
```

or with for debugging

```
nvcc -g -G -arch=sm_80 main.cu -o myapp
```

Compiling separate host and device source files

- Compile host code with a host compiler (e.g., g++)

```
g++ -c source.cpp –o source.o
```

- Compile device code with the CUDA compiler nvcc

```
nvcc –c kernels.cu –o kernels.o \
  -gencode arch=compute_60,code=sm_60 \
  -gencode arch=compute_70,code=\"compute_70,sm_70\"
```

- Link the generated objects with a linker on the host:

```
g++ -o my_app source.o kernels.o –L/usr/local/cuda/lib –lcudart -lcuda
```

Run the binary on a compute node with GPUs

- Request a GPU node

```
srun --nodes=1 --ntasks-per-node=4 --cpus-per-task=1 --time 00:20:00
--gres=gpu:1 --mem=8000MB -A tra24_ictp_np -p boost_usr_prod --pty
/bin/bash
```

- Make sure you have loaded the module cuda

```
module list
nvcc
module load cuda
module show cuda
nvcc
```

- Run the application

```
./myapp
```

Combining vectorAdd and scale

Implement A+B -> C and then scale C by a scalar

- Option 1: modify the vectorAdd kernel
- Option 2: implement another kernel for scaling after vectorAdd

CUDA Programming Guide:

https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

1. Allocate h_A, h_B, h_C
2. Init h_A, h_B
3. Allocate d_A, d_B, d_C
4. cudaMemcpy(from CPU to GPU) for (h_A, d_A) and for (h_B, d_B)
5. vectorAdd(d_A, d_B, d_C)
6. scale(d_C, k)
7. cudaMemcpy(from GPU to CPU) for (d_C, h_C)
8. scale_cpu(h_C, k)
9. cudaMemcpy(from CPU to GPU) for (h_C, d_C)
10. reverse(d_C)
11. cudaMemcpy(from GPU to CPU) for (d_C, h_C)

Timing on host using clock

```
#include <time.h>

clock_t start, stop;
float elapsed;

. . .
start = clock();
kernel<<<grid, threads>>>(..);

cudaDeviceSynchronize();

end = clock();
elapsed = (stop - start)/CLOCKS_PER_SEC;
. . .
```

set a “time stamp” on the host

wait until all the tasks on the device are complete

Timing on device using cudaEvent

```
cudaEvent_t start, stop;
float time;
cudaEventCreate(&start);
cudaEventCreate(&stop);
. . .
cudaEventRecord(start, 0);
kernel<<<grid, threads>>>(..);
cudaEventRecord(stop,0);
cudaEventSynchronize(stop);

cudaEventElapsedTime(&time, start, stop);
. . .
cudaEventDestroy(start);
cudaEventDestroy(stop);
```

create the events

set a “time stamp” using event start on stream 0

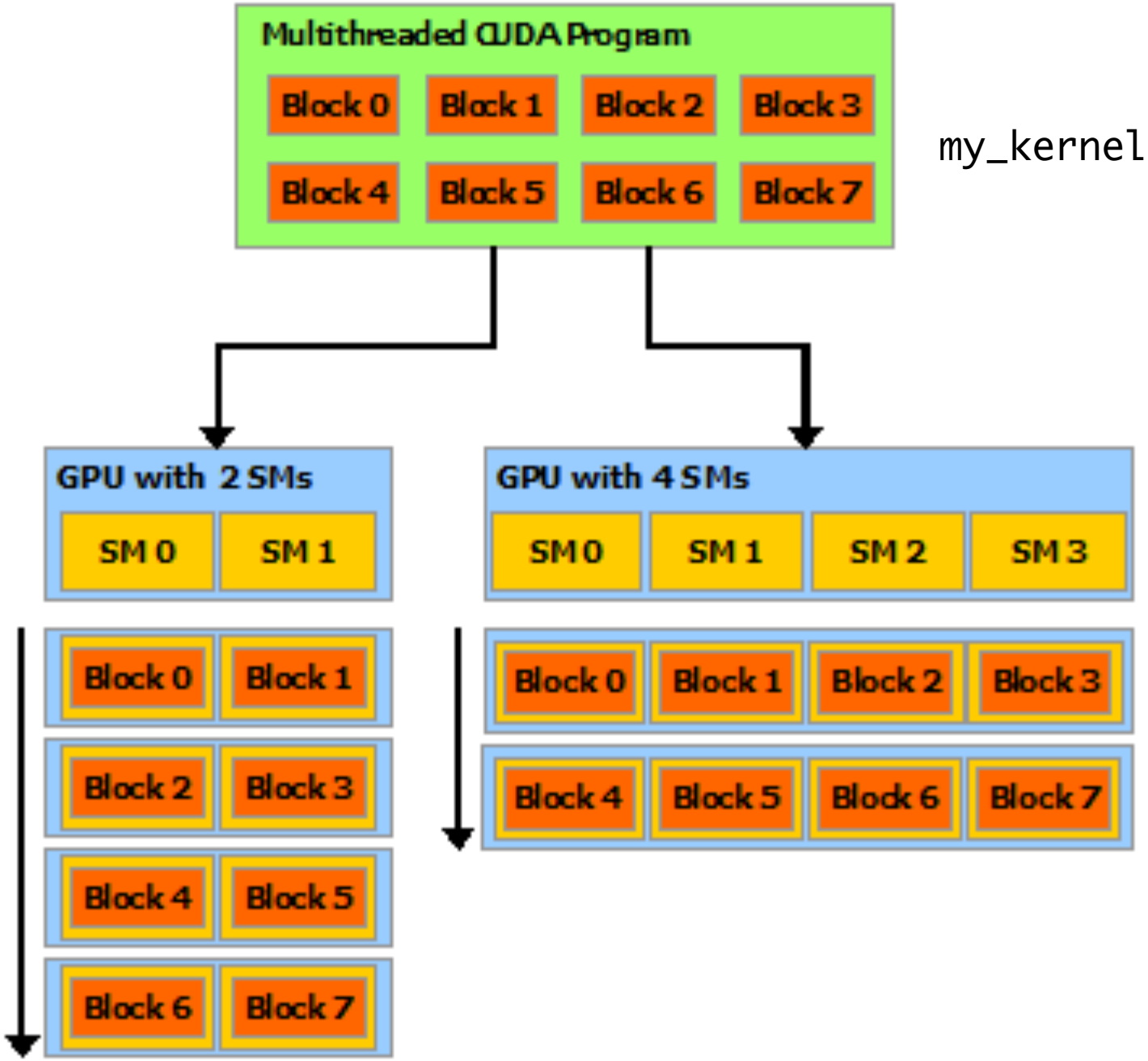
set a “time stamp” using event end on stream 0

wait until all the tasks before the last record of end are complete

destroy the events

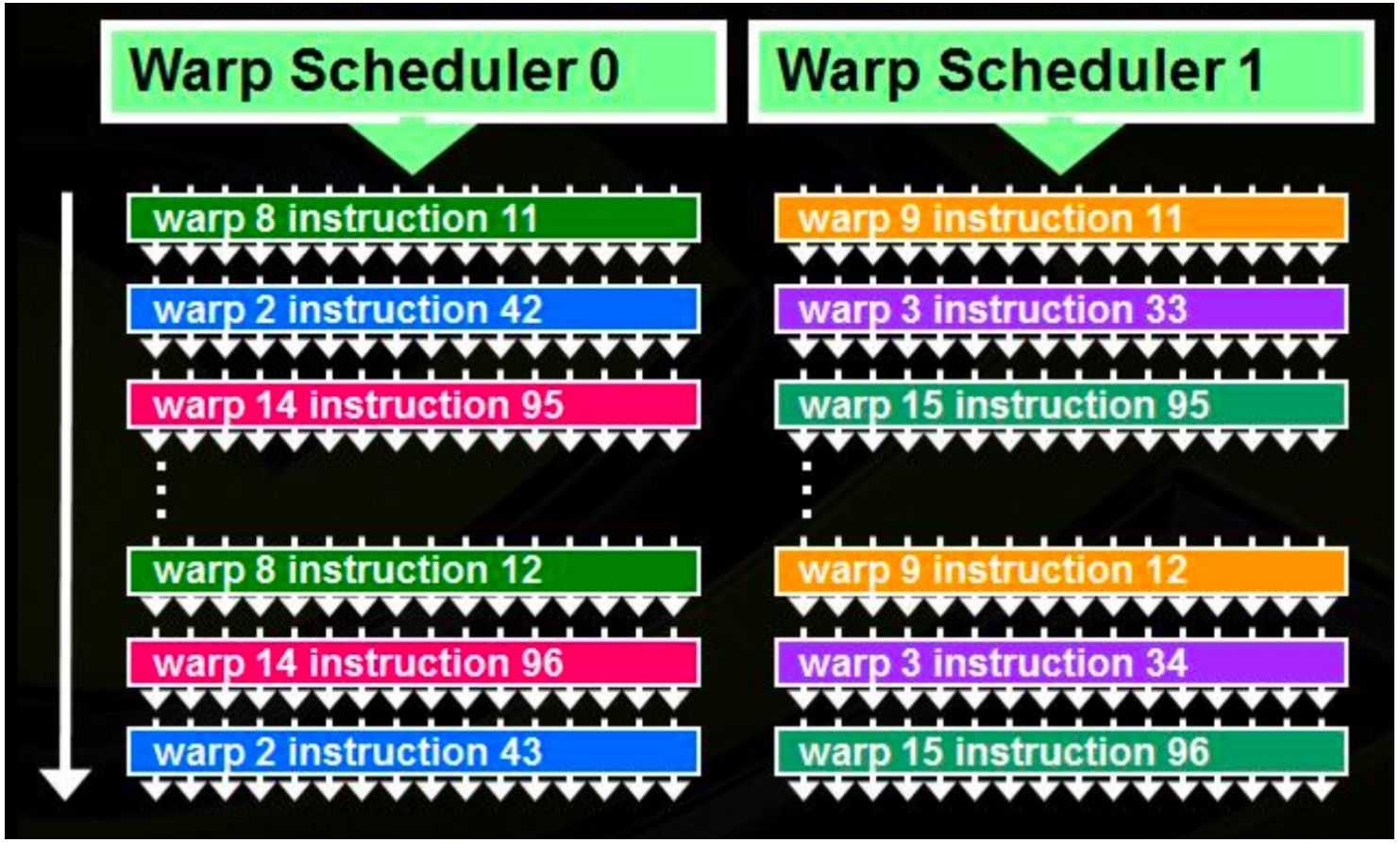
Thread block execution

- Each thread block is assigned entirely for a given SM
  - Thread block will be executed until completion.
  - A SM may be given multiple thread blocks
- A thread block is divided in units of 32-thread “warps”, like 32-wide vector SIMD on a CPU
  - The order of warps in a block being executed is given by the warp schedulers on the SM.
  - A SM can execute warps from multiple thread blocks



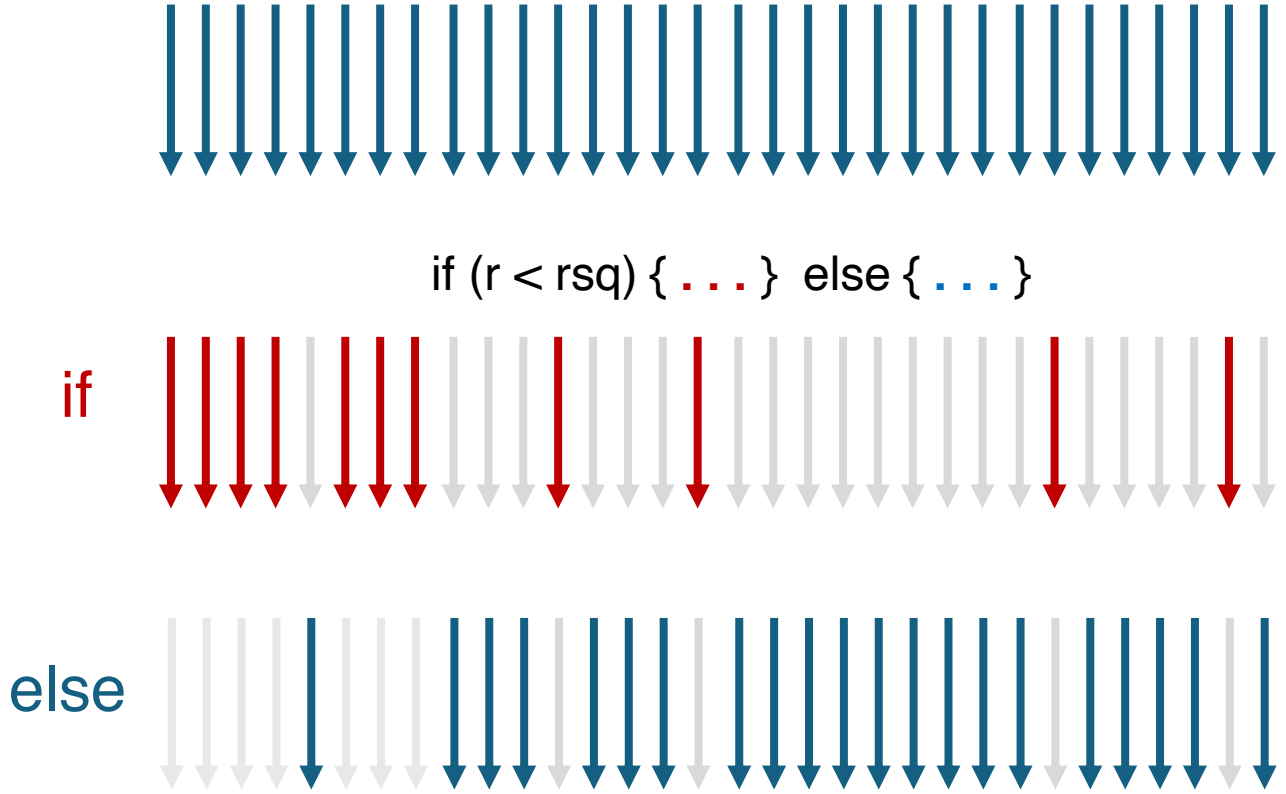

```
my_kernel<<< 8, 256 >>> (arg1, arg2)
```

Warps are executed out of order on a SM



Thread execution within a warp

- All threads in a warp execute the same instruction in lock-step
- Each thread (assigned to a kernel instance) is processed by a SMID lane (that is, an ALU/a FPU)
- A branch taken by a thread has to be taken by all threads (active and predicated threads)
  - warp divergence



Warp Divergence

- Branch divergence: when not all threads take the same branch, the entire warp has to execute both sides of the branch
- GPU blocks memory writes from disabled threads in the “if then” branch, then inverts all thread enable states and runs the “else” branch.
  - On GPUs, we get fast hardware-based implementation of thread predication/masking
- GPU hardware detects warp reconvergence and then runs with all threads enabled.

Quiz

1. How many threads can be launched on the GPU?
2. What is the number of threads in a warp?
3. What does the command `kernel<<< 16, 512 >>>` mean to do?
4. How to allocate memory on the device?
5. How to copy memory data from the host to device and back?

Exercise 2: Adding two vectors (vectorAdd)

Experiment:

- Varying the vector length to see the GPU performance vs CPU
- Varying the block size to see the GPU performance change

Coalesced memory access

Recall Axel's lecture: CPU cache (similar idea applied here)

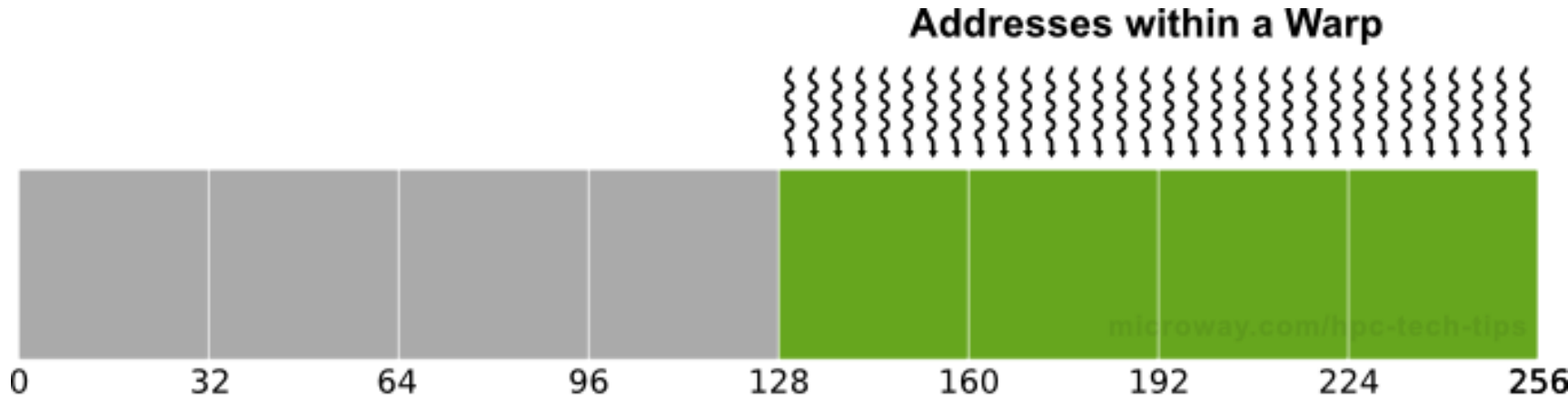
- When consecutive threads access (read/write) to aligned contiguous memory addresses
  - GPU combines (coalesces) these reads and writes into single transactions to cached memory (L1 or L2) instead of fetching from global memory
  - L1/shared memory transactions: 128-byte alignment
  - L2 transactions: 32-byte alignment

Coalesced memory access

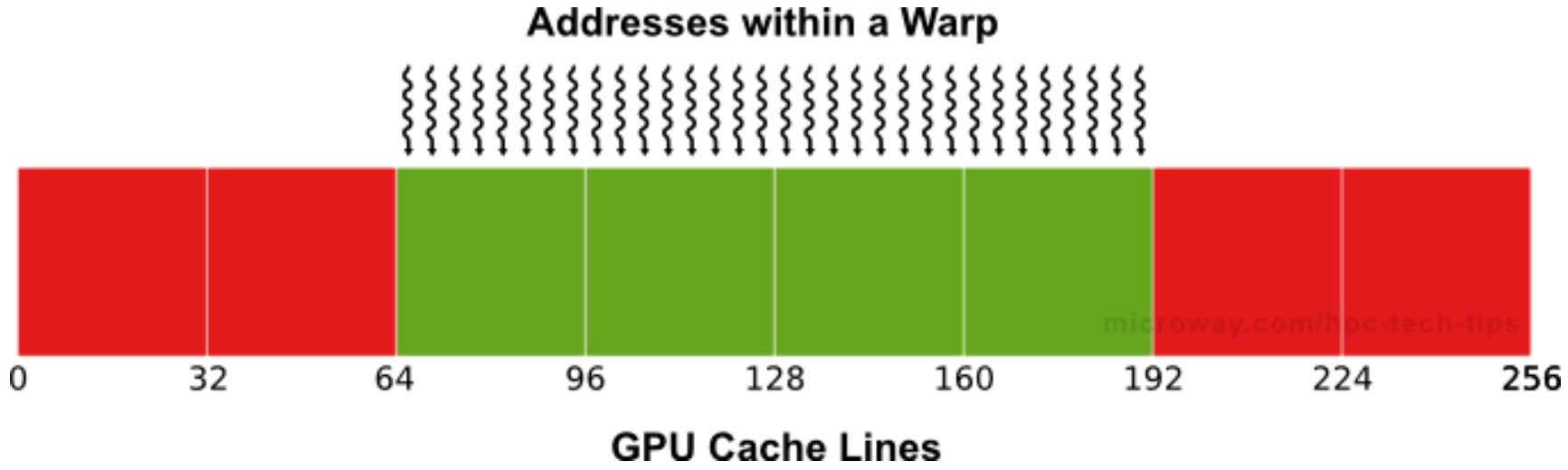
- If all the (32) threads in a warp read from a **contiguous** region of 32 items of 4, 8, or 16 bytes in size (all in 128 bytes, i.e. a L1 cache line), we have a coalesced access.
  - Special case: Multiple threads reading the same data are handled by a hardware broadcast.

Aligned memory accesses

Aligned memory access:
need to load 1 cache line

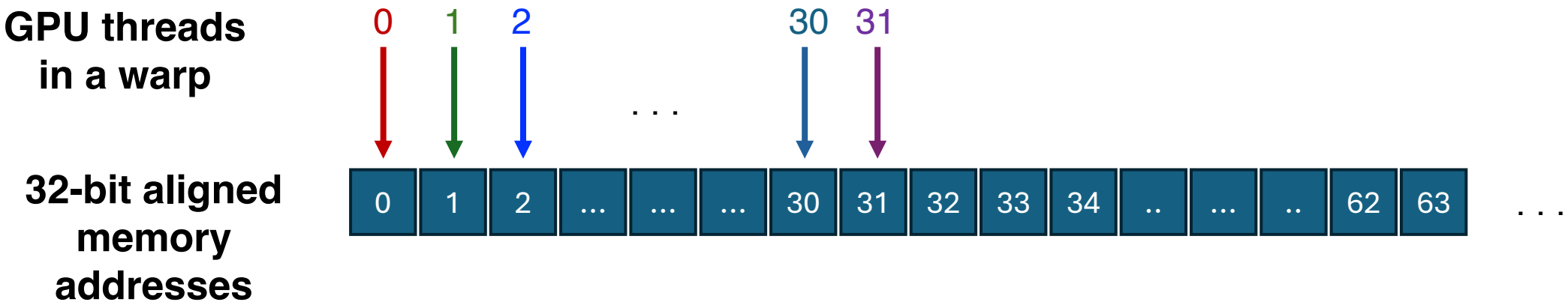


Mis-aligned memory access:
need to load 2 cache lines



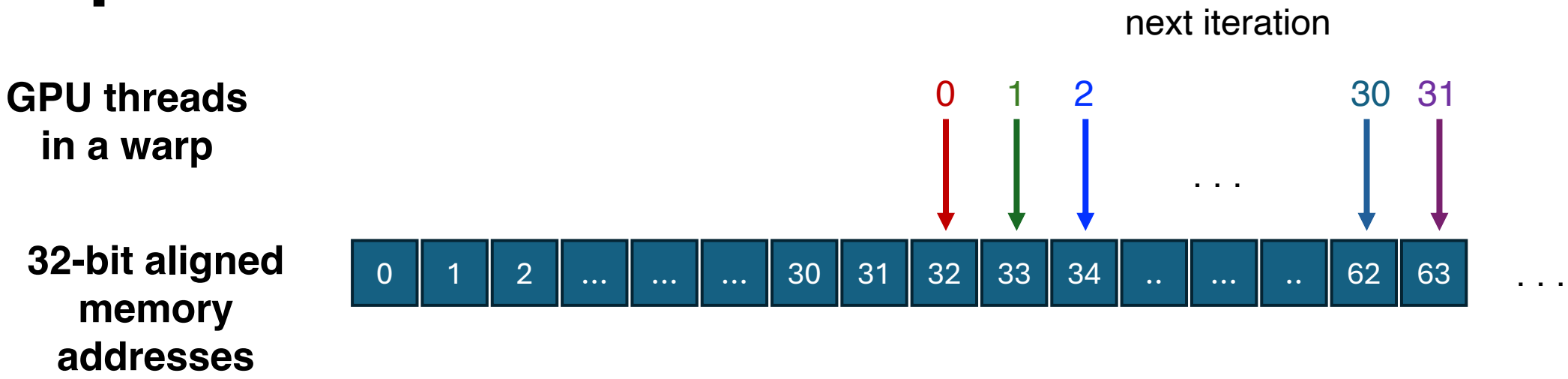
https://www.microway.com/hpc-tech-tips/gpu-shared-memory-performance-optimization/

Ideal case: accessing 32 aligned consecutive **fp32** numbers



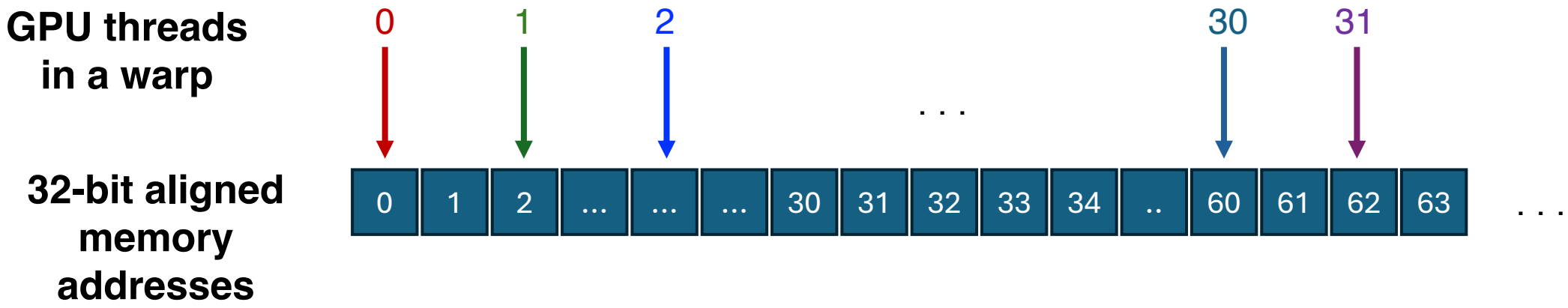
- 1x L1 transaction: 128 bytes needed / 128 bytes transferred
- 4x L2 transactions: 128 bytes needed / 128 bytes transferred

Ideal case: accessing 32 aligned consecutive **fp32** numbers



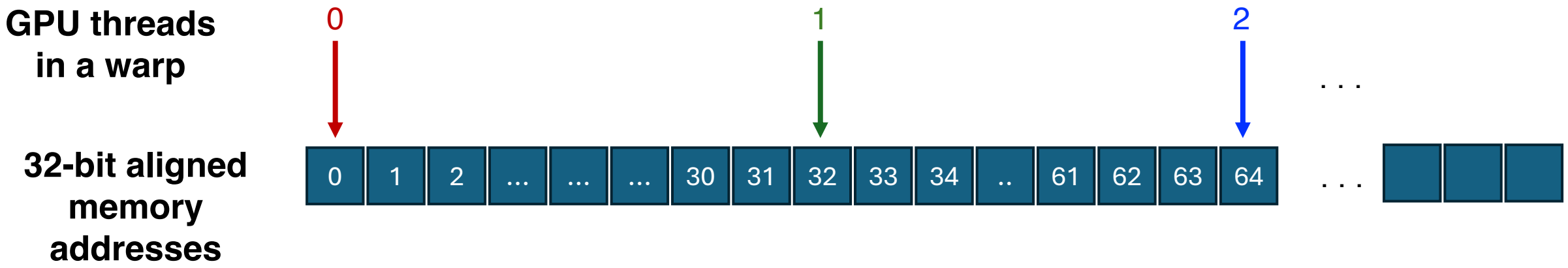
- 1x L1 transaction: 128 bytes needed / 128 bytes transferred
- 4x L2 transactions: 128 bytes needed / 128 bytes transferred

Ideal case: accessing 32 aligned consecutive **fp64** numbers



- 2x L1 transaction: 256 bytes needed / 2 x 128 = 256 bytes transferred
- 8x L2 transactions: 256 bytes needed / 8 x 32 = 128 bytes transferred

Worst case: accessing 32 **fp64** numbers with a stride of 128 bytes (16x fp64)



- 32x L1 transactions: 256 bytes needed / 32 x 128 = 4096 bytes transferred
- 32x L2 transactions: 256 bytes needed / 32 x 32 = 1024 bytes transferred

Optimal data layout depends on access patterns

Array of Structures

```
typedef struct {
  float x;
  float y;
  float z;
} myvec;

myvec aos[1024];
```

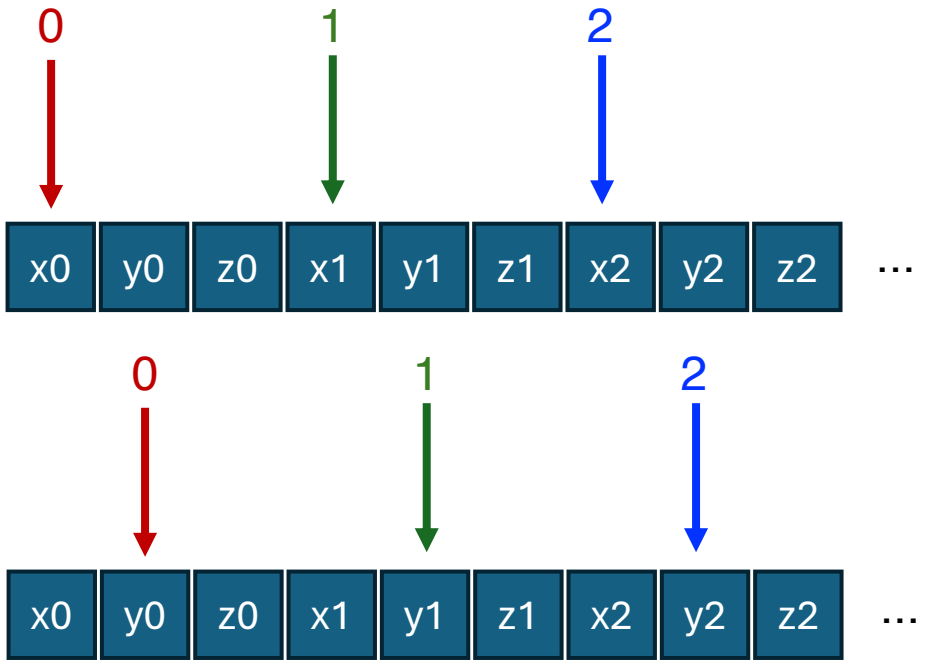
Structure of Arrays

```
typedef struct {
  float x[1024];
  float y[1024];
  float z[1024];
} myvecs;

myvecs soa;
```

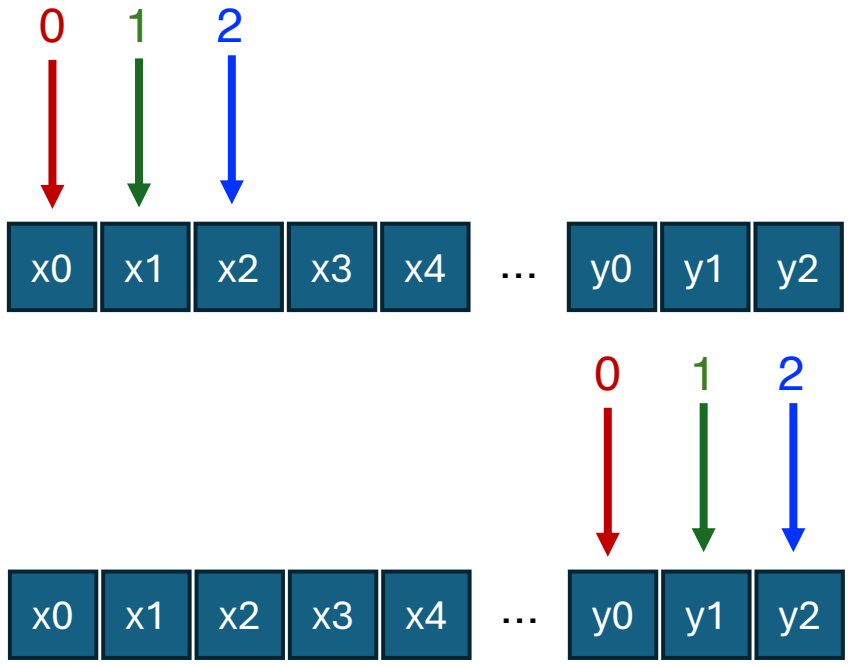
Accessing individual components

```
aos[threadIdx.x].x = ...;
aos[threadIdx.x].y = ...;
```

```
soa.x[threadIdx.x] = ...;
soa.y[threadIdx.x] = ...;
```



Uncoalesced

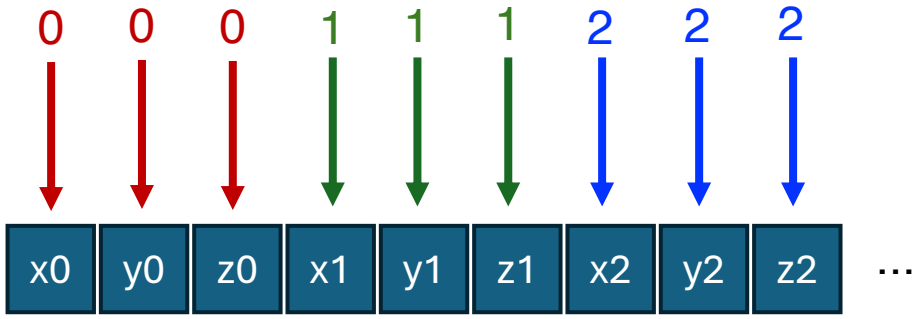


Coalesced

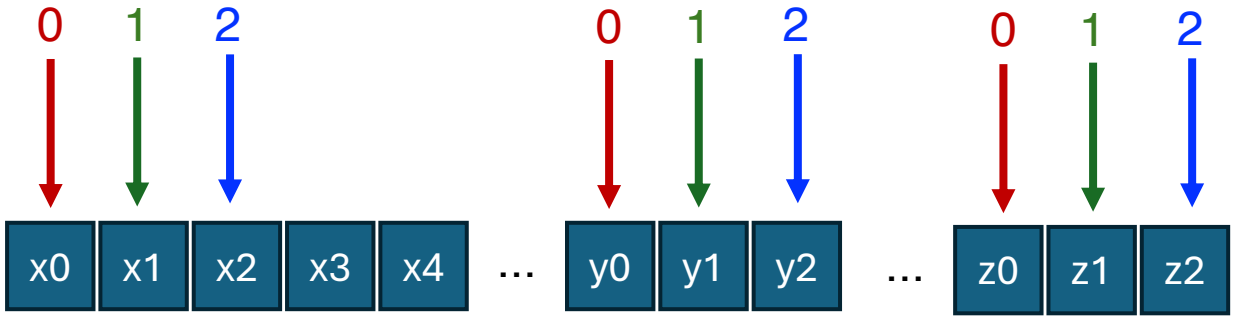
Accessing all the components at once

```
t = aos[threadIdx.x].x
+ aos[threadIdx.x].y
+ aos[threadIdx.x].z;
```

```
t = soa.x[threadIdx.x]
+ soa.y[threadIdx.x]
+ soa.z[threadIdx.x];
```



Coalesced



Uncoalesced

Porting CPU codes to GPU often involves converting b/w AOS and SOA

Communication between GPU threads in a block

- Threads can write to/read from **shared memory** and global memory.
- Using barrier synchronizations: __syncthreads()

```
__global__ void my_kernel(float* input, float* output, int num_elements)
{
   extern __shared__ float _sdata[];
   int tid = blockIdx.x * blockDim.x + threadIdx.x;

   _sdata[threadIdx.x] = input[tid];

   __syncthreads(); // wait for all threads in the block to reach here

   output[tid] = _sdata[threadIdx.x];
}
```

Shared memory allocation at kernel launch

Default: 48 KB per block

```
my_kernel<<< num_blocks, block_size >>> (input, output,
num_elements);
```

Dynamic allocation:

```
int shared_mem_size = 4*block_size * sizeof(float);
my_kernel<<< num_blocks, block_size, shared_mem_size >>>
(input, output, num_elements);
```

Threads in the same warp can communicate with each other using warp-level primitives

- exchange registers between threads (lanes) without going to shared memory: __shufl_down_sync(), __shufl_down_xor(),

```
#define FULL_MASK 0xffffffff // all 32 threads in the warp
for (int offset = 16; offset > 0; offset /= 2)
    val += __shfl_down_sync(FULL_MASK, val, offset);
```

Communicate between blocks: Atomic operations

- Multiple threads may write to the same memory address
  - Similar issues to multithreading with OpenMP: race condition
  - Atomic operations serialize executations
- CUDA functions for 32-bit and 64-bit
  - atomicAdd
  - atomicSub
  - atomicMax/Min
  - atomicInc/Dec
  - ...
- If used wisely, could offer good speedup vs non-GPU

Error checking with CUDA runtime functions

```
cudaError_t status;
status = cudaMalloc(&d_a, memSize);
if (status != cudaSuccess) {
   printf("Error in device allocation: Error code %d Reason: %s\n",
      status, cudaGetErrorString(status));
}
```

Details of the error code are given in
$CUDA_HOME/include/cuda.h

Kernel execution failure on the device

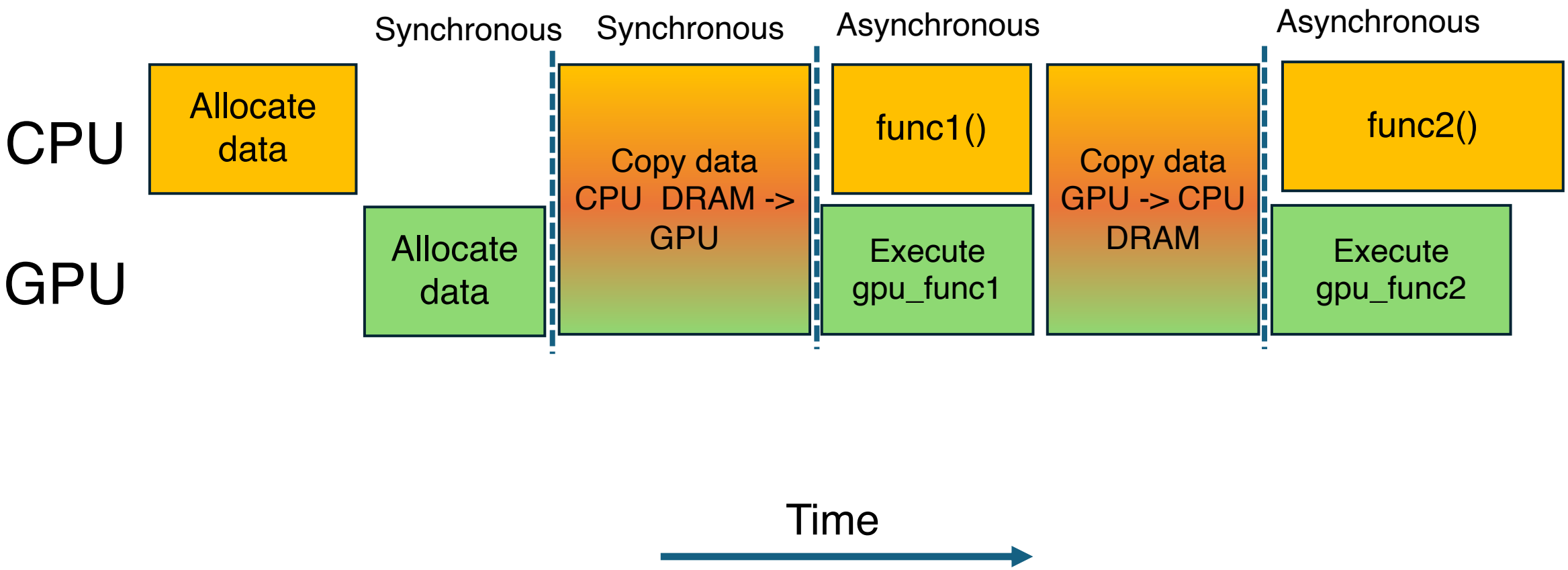
```
cudaError_t status;
myKernel<<< num_blocks, block_size >>>(arg1, arg2);
status = cudaGetLastError();
if (status != cudaSuccess) {

     printf("Error in Kernel execution: Error code %d Reason: %s\n",
       status, cudaGetErrorString(status) );
}
```

Details of the error code are given in
$CUDA_HOME/include/cuda.h

Exercises

- Thread communication within a block
  - ex3: reduction
  - ex4: scalar product of 2 vectors
- More on shared memory access:
  - ex4: transpose (afternoon)
  - ex5: matMul (afternoon)

CUDA function calls sync/async wrt host



CPU and GPU interactions

All kernel launches are asynchronous (i.e. non-blocking with respect to the host)

- control returns to CPU immediately
- kernel starts executing once all previous CUDA calls on the same stream have completed.

Memory copies (cudaMemcpy) are synchronous

- control returns to CPU once the copy is complete
- copy starts once all previous CUDA calls have completed

Asynchronous mem copies are available (tomorrow):

- on user streams, with pinned host memory
- ability to overlap mem copies and kernel execution

Exercises

- Shared memory access:
  - ex3-sharedmem: jacobi, stencil

Take-home messages

1. GPUs need a lot of active threads to keep all the cores busy.
2. Data transfers between GPU memory and CPU memory are always the bottleneck.
3. Memory accesses on the GPU need to be coalesced.

Summary

- GPU architecture and programming concepts
  - Streaming processors, CUDA cores and memory hierachy
- CUDA programming models
  - Kernels, threads, blocks and grid
  - Memory hierarchy
  - Coalesced memory access