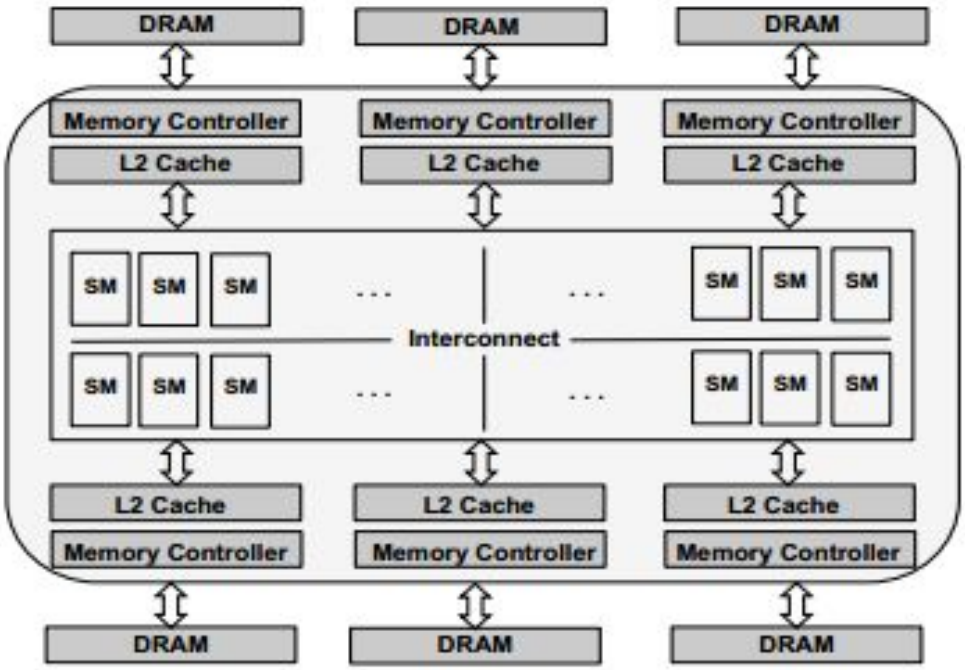
PROGRAMMING GPGPU ARCHITECTURE

Dr Noor Mahammad Sk

Goals

- Data Parallelism: What is it, and how to exploit it?
  - Workload characteristics
- Execution Models/ GPU Architectures
  - MIMD (SPMD), SIMD, SIMT
- GPU Programming Models
  - Terminology translations: CPU ?? AMD GPU ??Nvidia GPU
  - Intro to OpenCL
- Modern GPU Microarchitectures
  - i.e., programmable GPU pipelines, not their fixed-function predecessors

Architecture of GPU

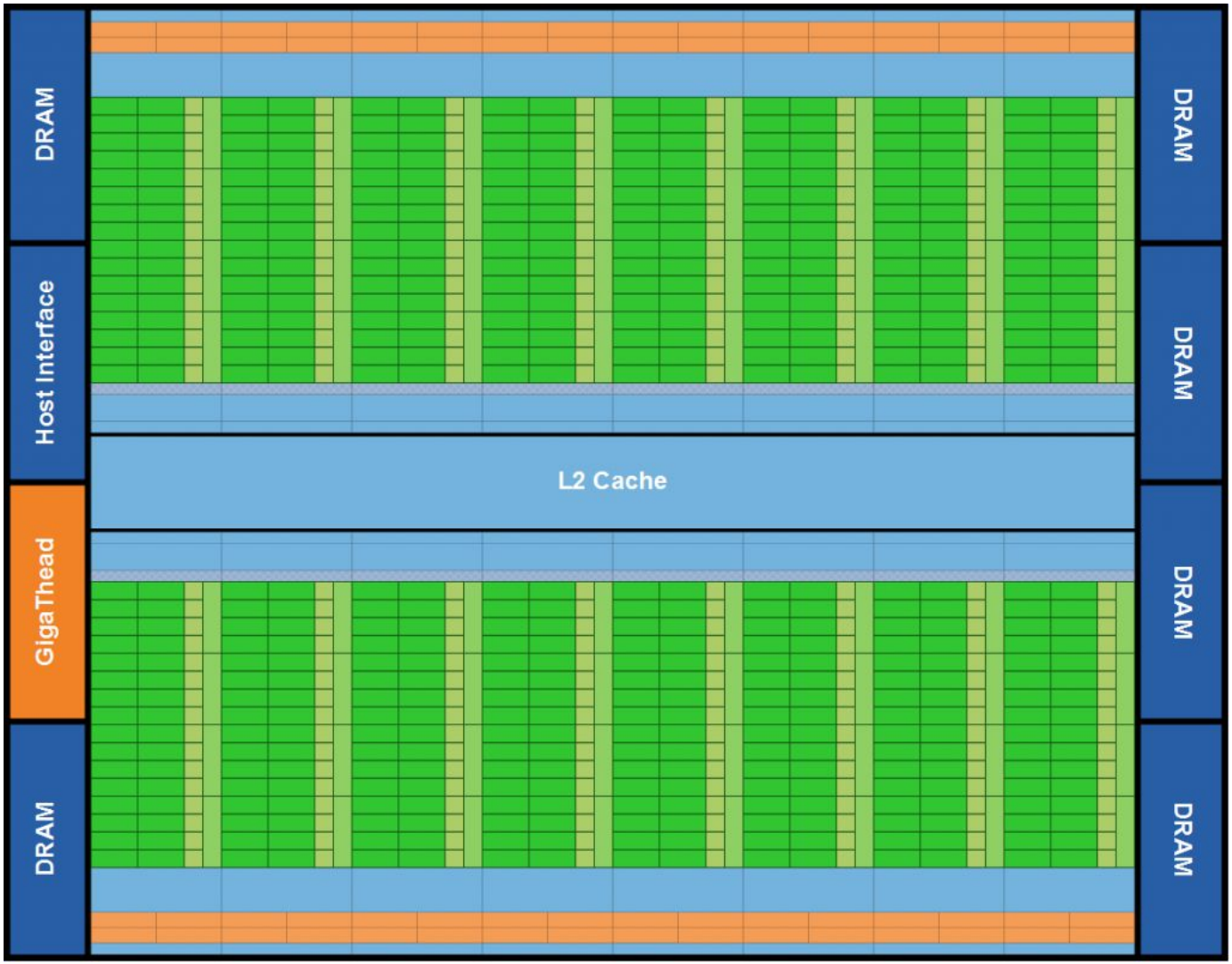


- Many streaming multiprocessors (SMs)
- Each SM typically having a SIMT width of 8 to 32
- Each SM is associated with
  - A private L1 Data Cache and
  - Read-only texture and
  - Constant caches
  - Along with a low latency shared memory (scratchpad memory)
- Every MC is associated with a slice of the shared L2 cache for faster access to the cached data.
- Both MC and L2 are on-chip.

Fermi based GPU

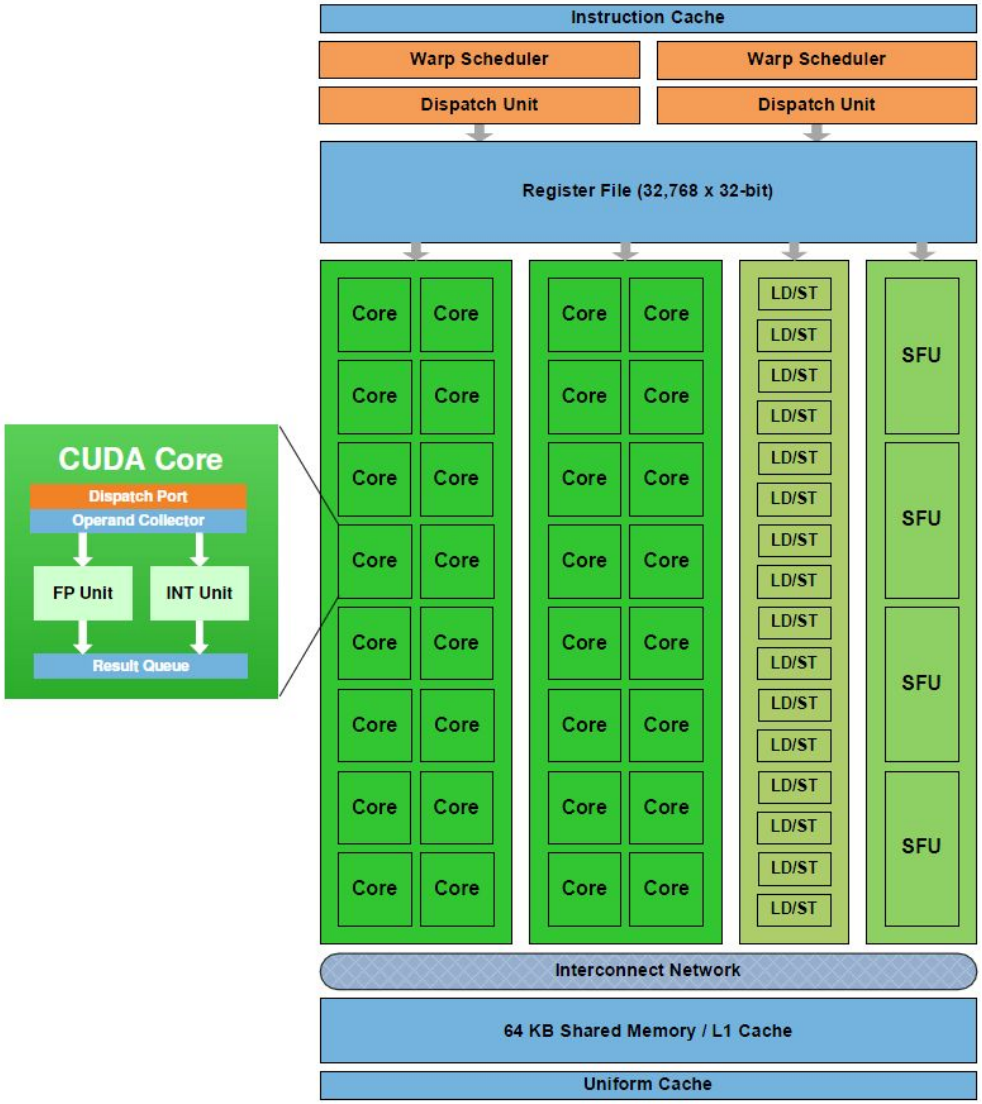
- Fermi based GPU has 512 CUDA cores.
- A CUDA core executes a floating point or integer instruction per clock for a thread.
- The 512 cores are organized in 16 Streaming Multiprocessors (SM) of 32 cores each.
- The SMs are supported by a L2, host interface, GigaThread scheduler and multiple DRAM interfaces.
- The GPU has six 64-bit memory partitions, for a 384-bit memory interface, supporting up to a total of 6GB of GDDR5 DRAM memory.
- A host interface connects the GPU to the CPU via PCI-Express.
- The GigaThread global scheduler distributes thread blocks to SM thread schedulers.

Fermi based GPU



- Fermi's 16 SM are positioned around a common L2 Cache.
- Each SM is a vertical rectangular strip that contain an orange portion (Scheduler and dispatch)
- A green portion (execution units), and light blue portions (Register file and L1 Cache)

Architecture of Fermi Streaming Multiprocessor

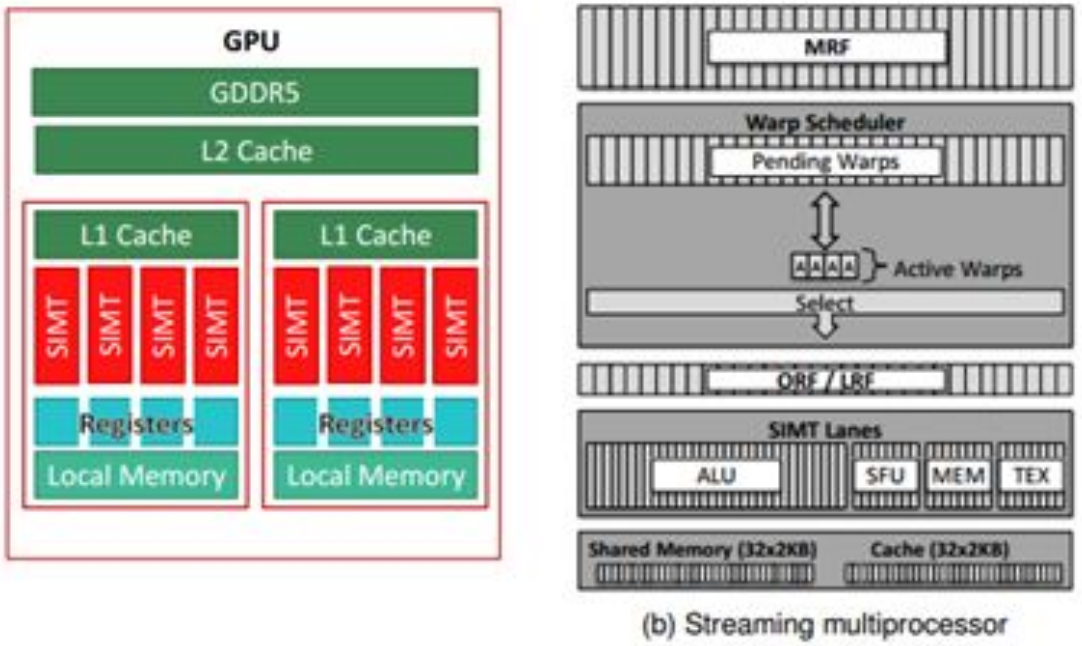


- For Fermi, a SM can have at most 8 Thread Blocks, 48 Warps (32 Threads/Warp) and 1536 Threads.

32 Streaming Processor:

- Each core has a fully pipelined integer arithmetic logic unit (ALU) and floating point unit (FPU).
- SP has no independent units of Fetch, Decode or Dispatch, and thus it can only receive the instructions issued by SM.
- Given that one SP has one ALU, thus we can use ALU to denote one SP or one Lane.
- Right figure shows an SM containing 32 SIMT lanes that each executes up to one thread instruction per cycle.
- The SM has up to 1024 resident threads, and a 32-entry, single issue, in-order warp scheduler selects one warp per cycle to issue an instruction.

Stream Multiprocessor



(b) Streaming multiprocessor

- ***16 Load/Store Units:***
- Each SM has 16 l/s units, allowing source and destination addresses to be calculated for 16 threads per clock.
- Supporting units load and store the data at each address to cache or DRAM.
- ***Four Special Function Units:***
- Each SFU executes one instruction per thread, per clock;
- A warp executes over eight clocks.
- The SFU pipeline is decoupled from the dispatch unit, allowing the dispatch unit to issue to other execution units while the SFU is occupied.

GigaThread Thread Scheduler

- One of the most important technologies of Fermi architecture is its two-level, distributed thread scheduler.
- At the chip level, a global work distribution engine schedules thread blocks to various SMs, while at the SM level, each warp scheduler distributes warps of 32 threads to its execution units.
- The first generation GigaThread engine introduced in G80 managed up to 12,288 threads in realtime.

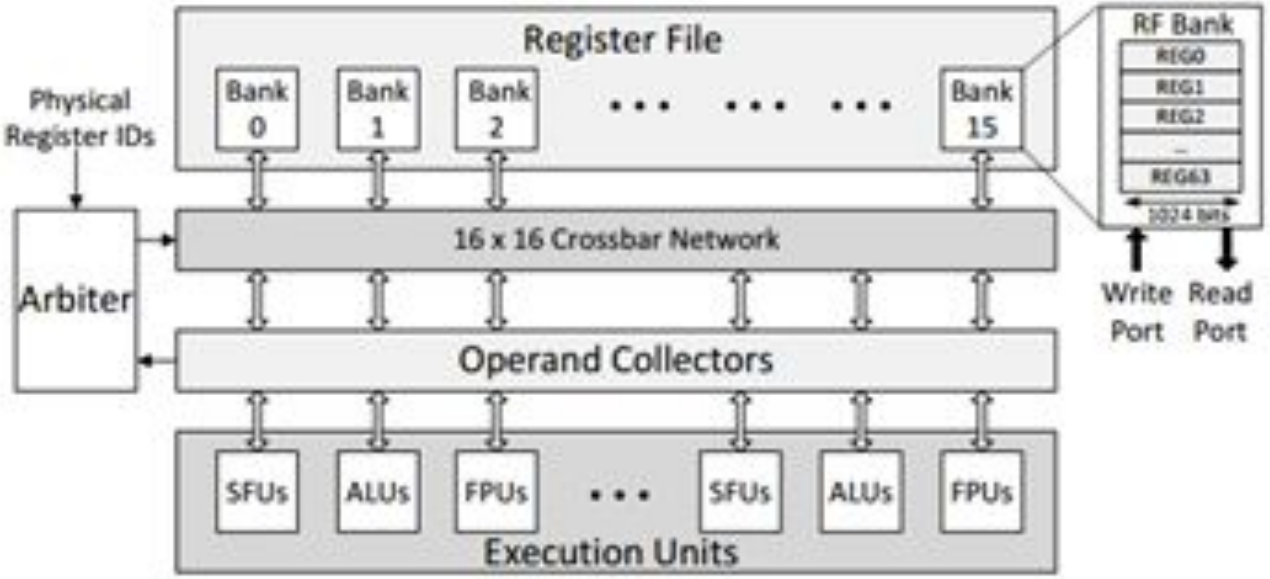
Dual Warp Scheduler

- Each SM features two warp schedulers and two instruction dispatch units, allowing two warps to be issued and executed concurrently.
- Fermi's dual warp scheduler selects two warps, and issues one instruction from each warp to a group of sixteen cores, sixteen l/s units, or four SFUs.
- Because warps execute independently, Fermi's scheduler does not need to check for dependencies from within the instruction stream.

32768 Registers

- Each SM has a large and unified register file that is shared across warps executing in the same SM.
- Register size is namely 128KB, and 21 registers/thread, 32768/1536, at full scheduler occupancy.
- The total register file capacity across the chip is 2MB, substantially exceeding the size of the L2 cache.
- GPUs adopt a multibanked register file architecture to avoid the area and power overhead of a multiported register file.
- Crossbar networks and operand collectors are used to transport operands form different banks to the appropriate SIMD execution lane.
- The register file can be modeled as 16 dual-ported (one read and one write port) banks, each of which provides 2048 logical 32-bit registers.
- Physically, these registers are accessed as 64, 1024-bit registers.
- Thus, each operand read from a register file bank provides operands for 32 SIMT threads.
- A crossorbar interconnection network is used to transfer operands from register file banks to operand collectors.
- Furthermore, in the case of bank conflicts, the arbiter is responsible for serializing the register file access, and the operand collectors are used to buffer the data already read from the register file.
- Each thread complete owns certain registers, and releases only when the thread block finishes.

Register File structure in Fermi Architecture



Register file structure in the Fermi architecture.

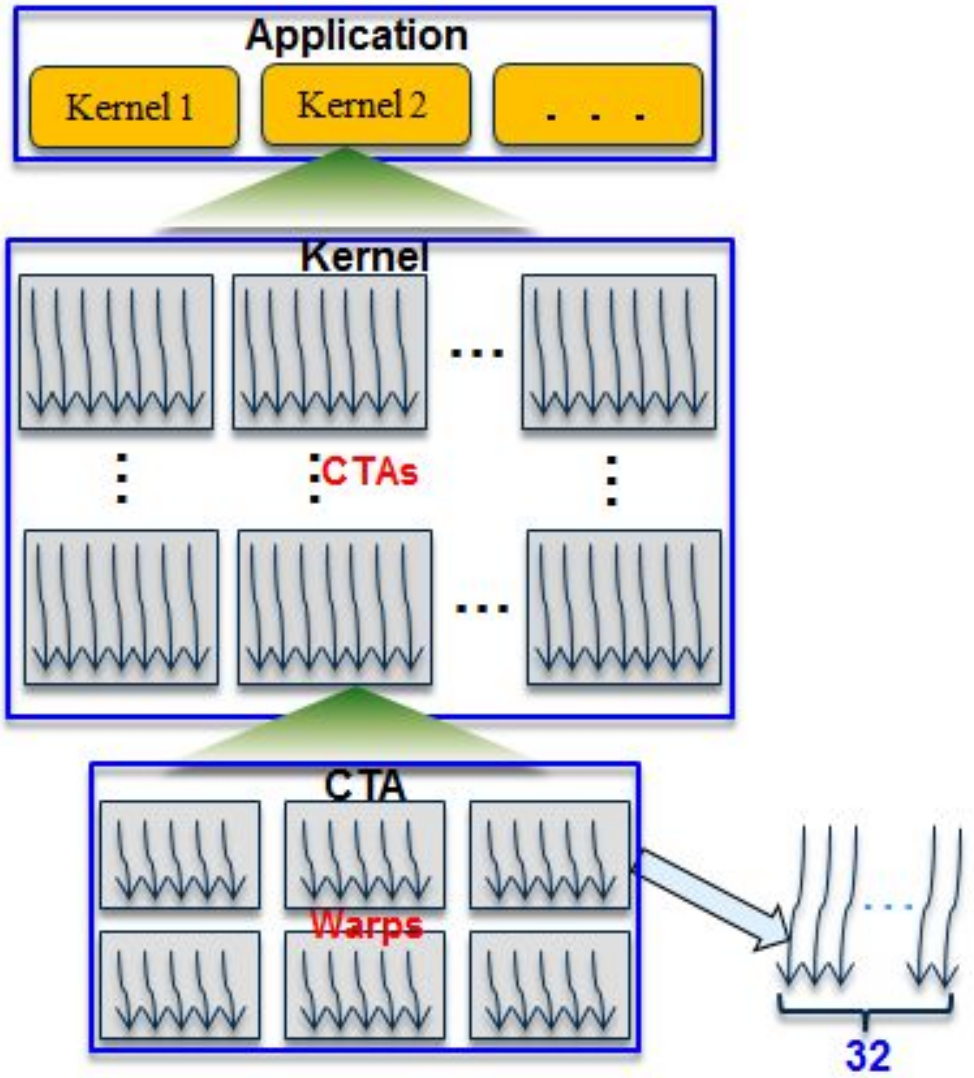
64 KB Configurable Shared Memory and L1 Cache

- on-chip shared memory enable threads within the same thread block to cooperate, facilitates extensive reuse of on-chip data, and greatly reduces off-chip traffic.
- On one hand, the shared memory is used by multiple thread blocks;
- on the other hand, each thread block exclusively uses part of the shared memory, which is shared by all thread of the block.

CUDA Hierarchy of Threads, Blocks and Grids

- A GPU executes one or more kernel grids;
- An SM executes one or more thread blocks; and
- CUDA cores and other execution units in the SM execute threads.

- A program may consist of one or more kernels, each consisting of one or more co-operative thread arrays (CTAs), and each CTA consists of multiple warps.

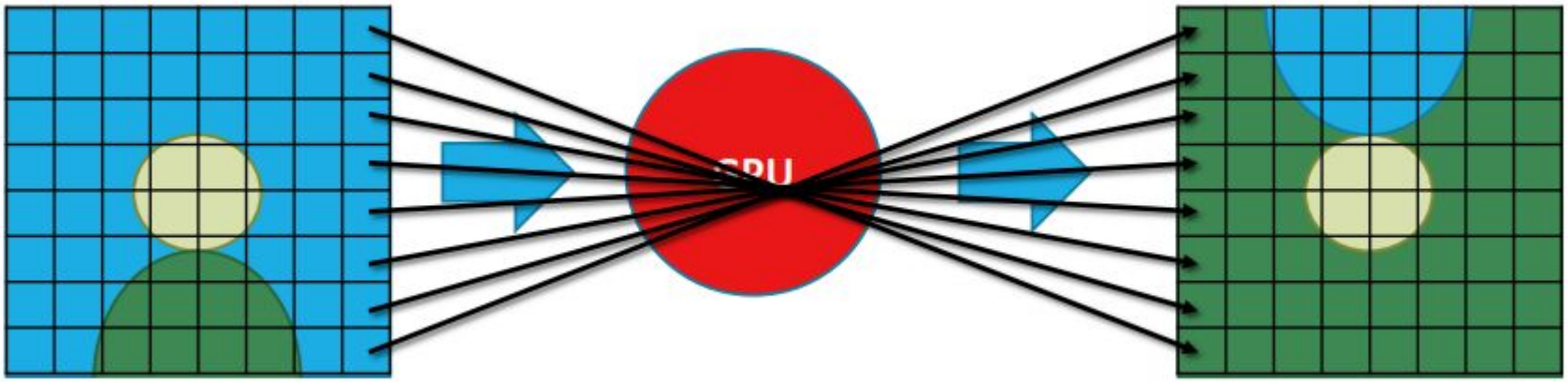

Application
Kernel 1
Kernel 2
. . .
Kernel
CTAs
CTA
Warps
32

Thread, Thread Block, Grid

- ***Thread:***
- Each thread within a thread block executes an instance of the kernel, and has a thread ID within its thread block, program counter, registers, per-thread private memory, inputs, and output results.
- ***Thread Block (also called Cooperative Thread Array, CTA):***
- A thread block is a set of concurrently executing threads that can cooperate among themselves through barrier synchronization and shared memory.
- A thread block has a block ID within its grid. (For Fermi, Max Thread/Block is 1024)
- ***Grid (Kernel):***
- A grid is an array of thread blocks that execute the same kernel, read inputs from global memory, write results to global memory, and synchronize between dependent kernel calls.

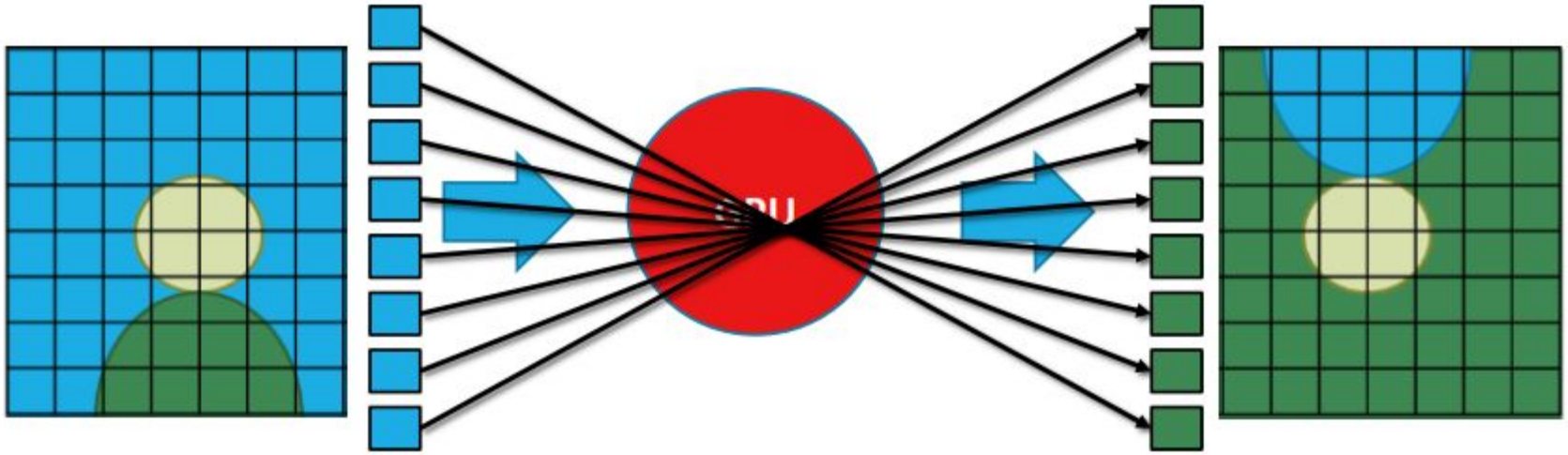
Graphics Workloads

- ***Identical, Streaming computation on pixels***



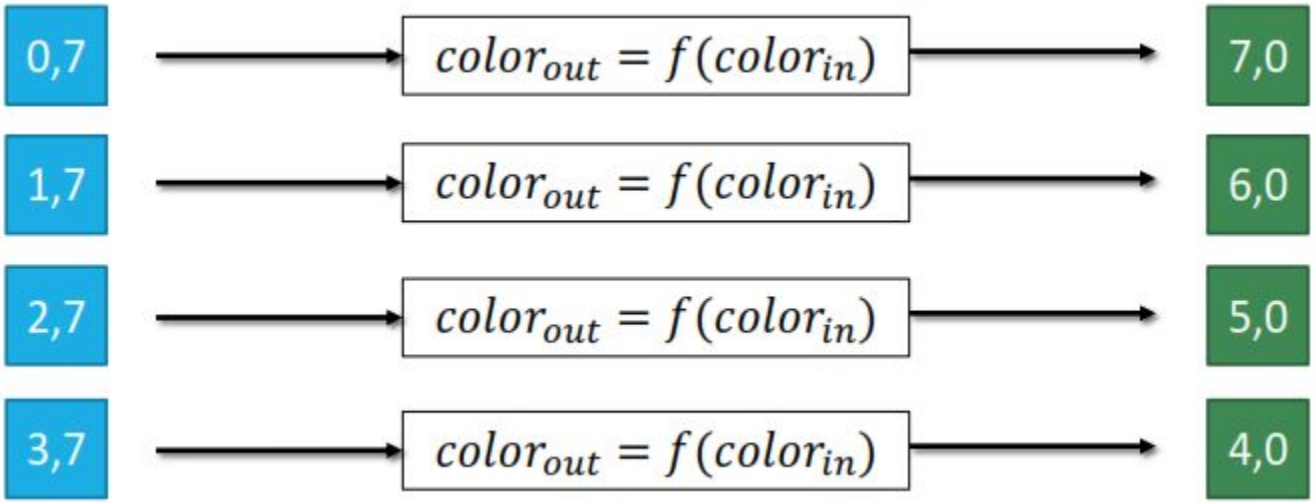
Graphics Workloads

- ***Identical, Independent, Streaming computation on pixels***



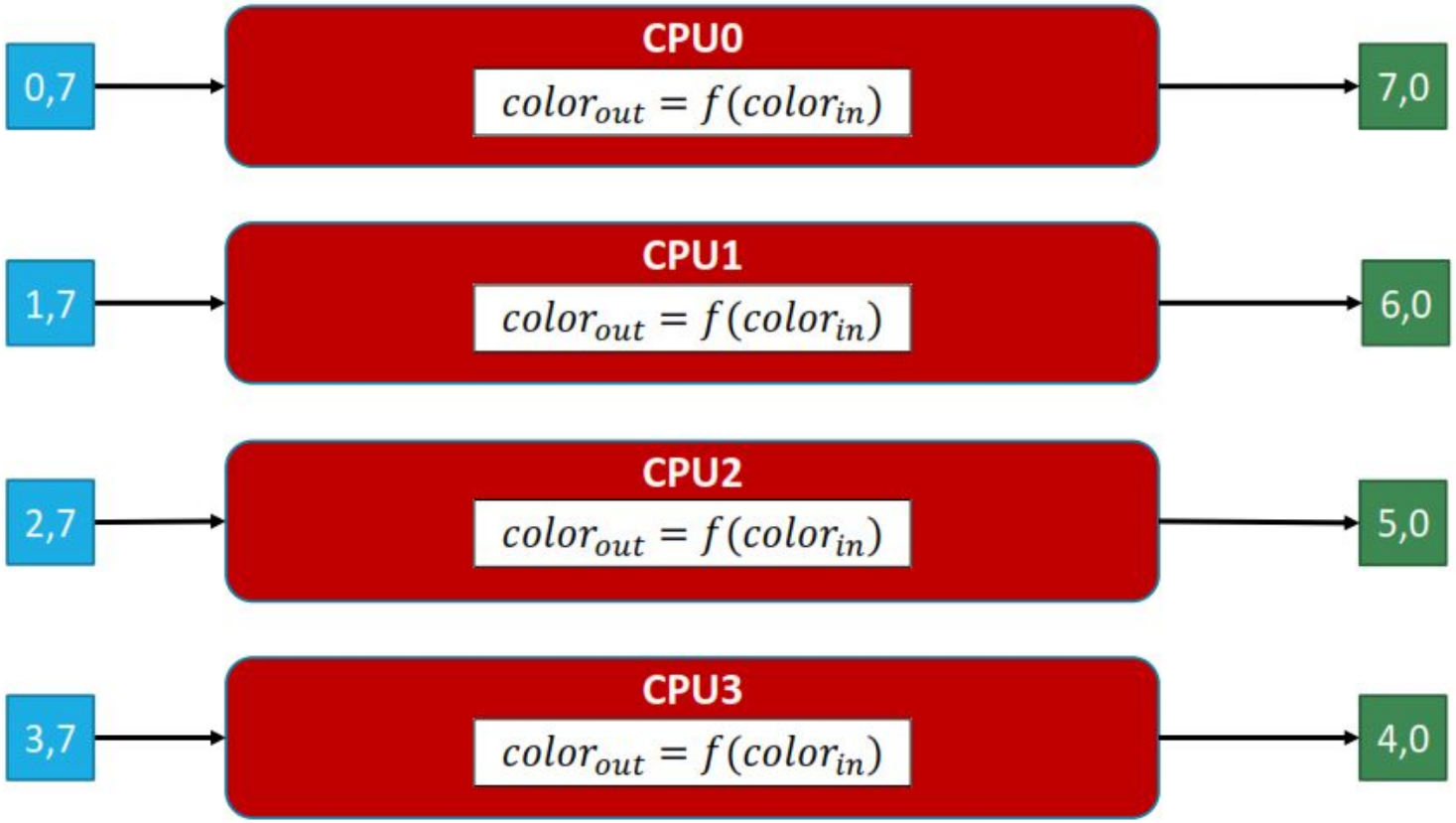
Generalize: Data Parallel Workloads

- ***Identical, Independent computation on multiple data inputs***



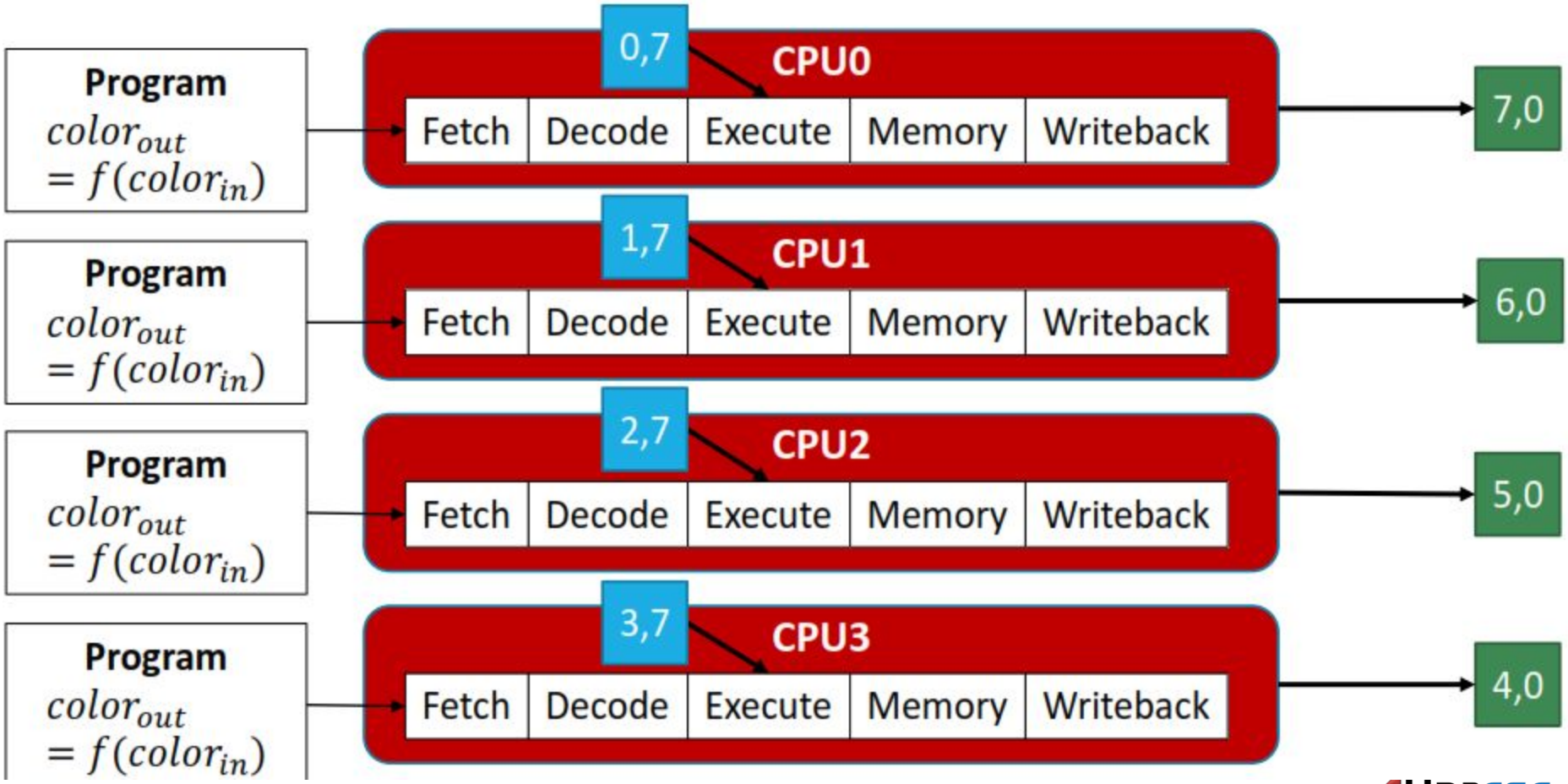
Naïve Approach

- Split independent work over **multiple** processors



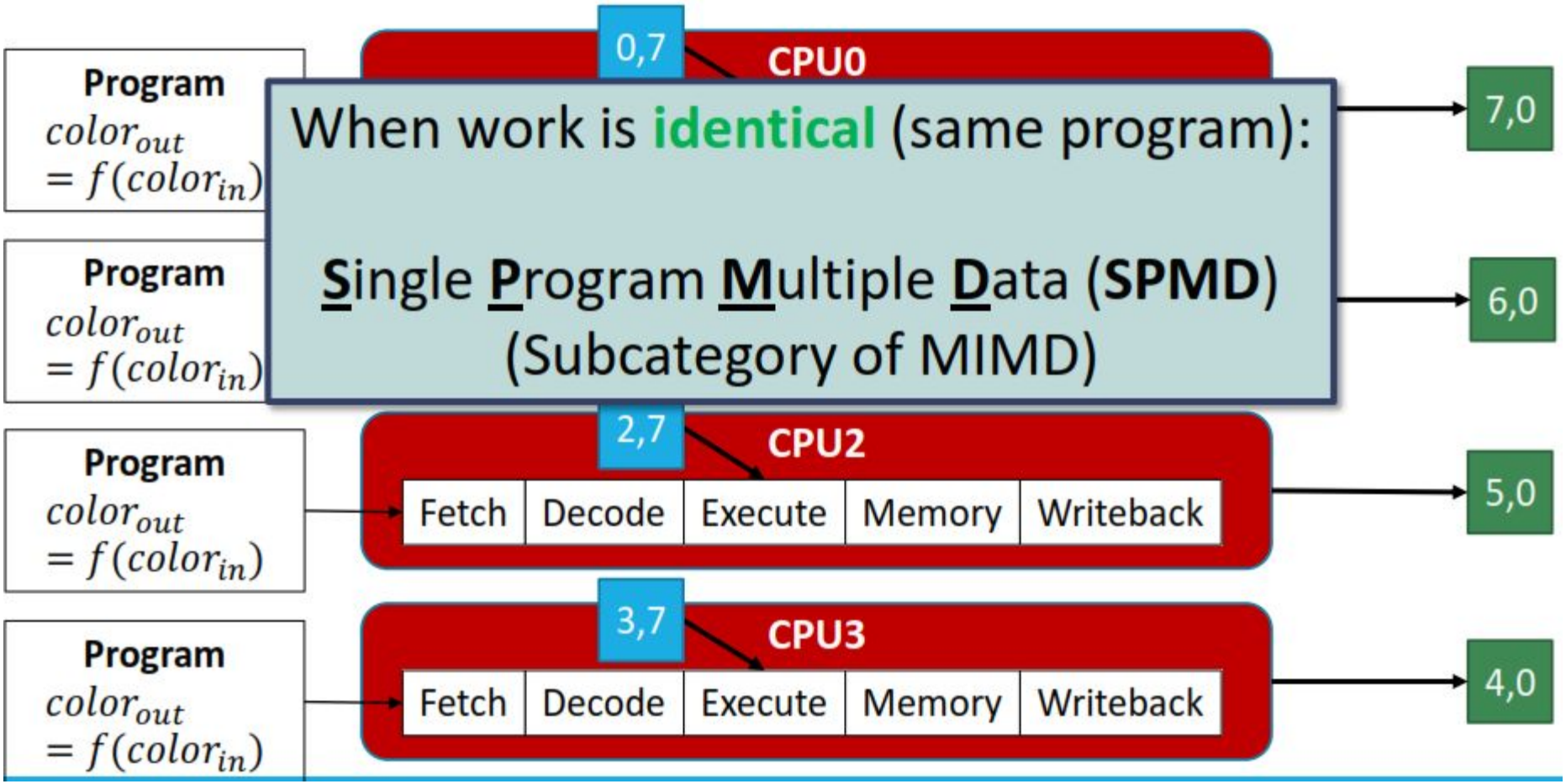
Data Parallelism: A MIMD Approach

- **Multiple Instruction Multiple Data**
- Split **independent** **work over** **multiple** **processors**



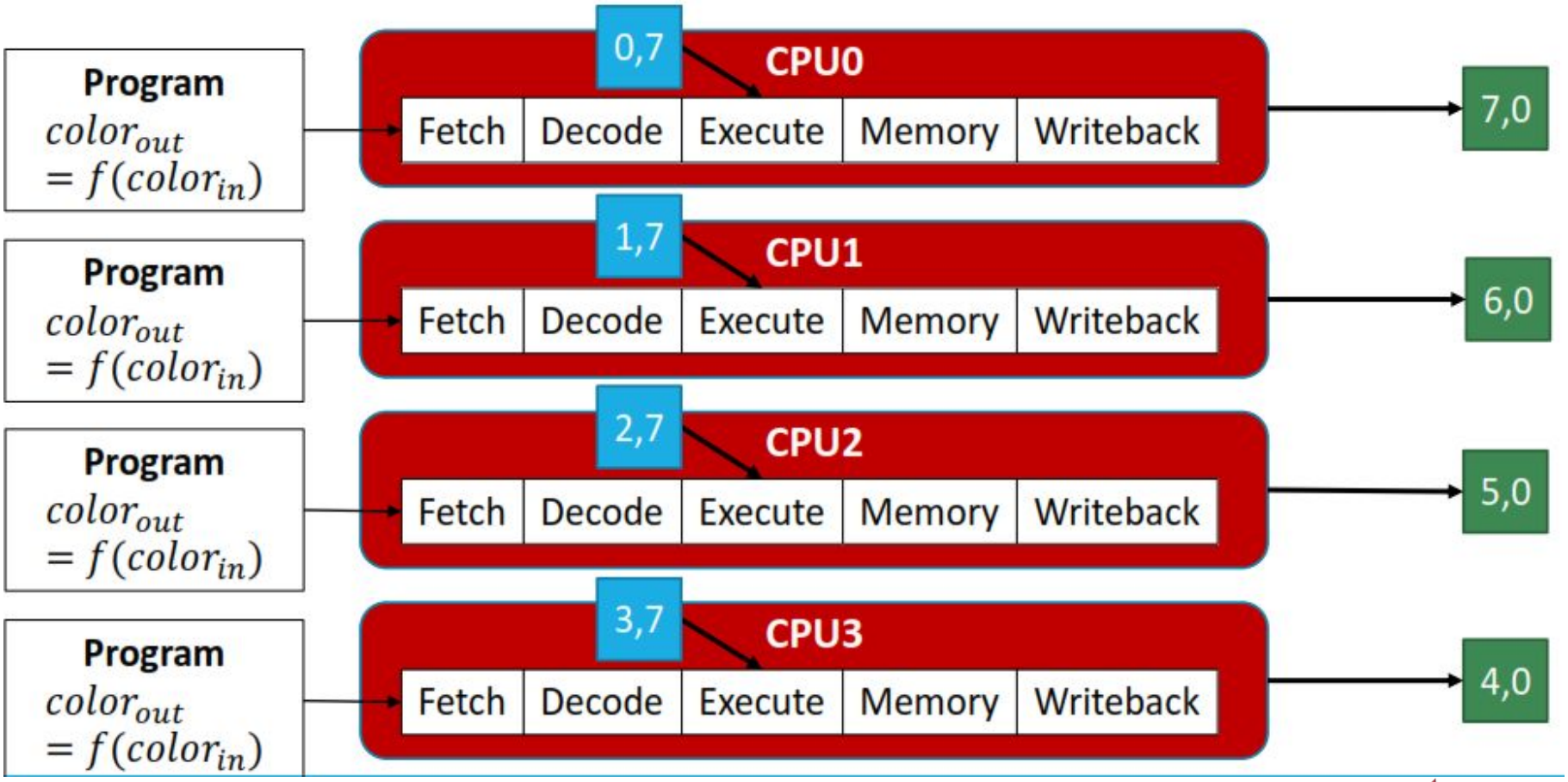
Data Parallelism: A MIMD Approach

- **Multiple Instruction Multiple Data**
- Split **independent work over multiple processors**



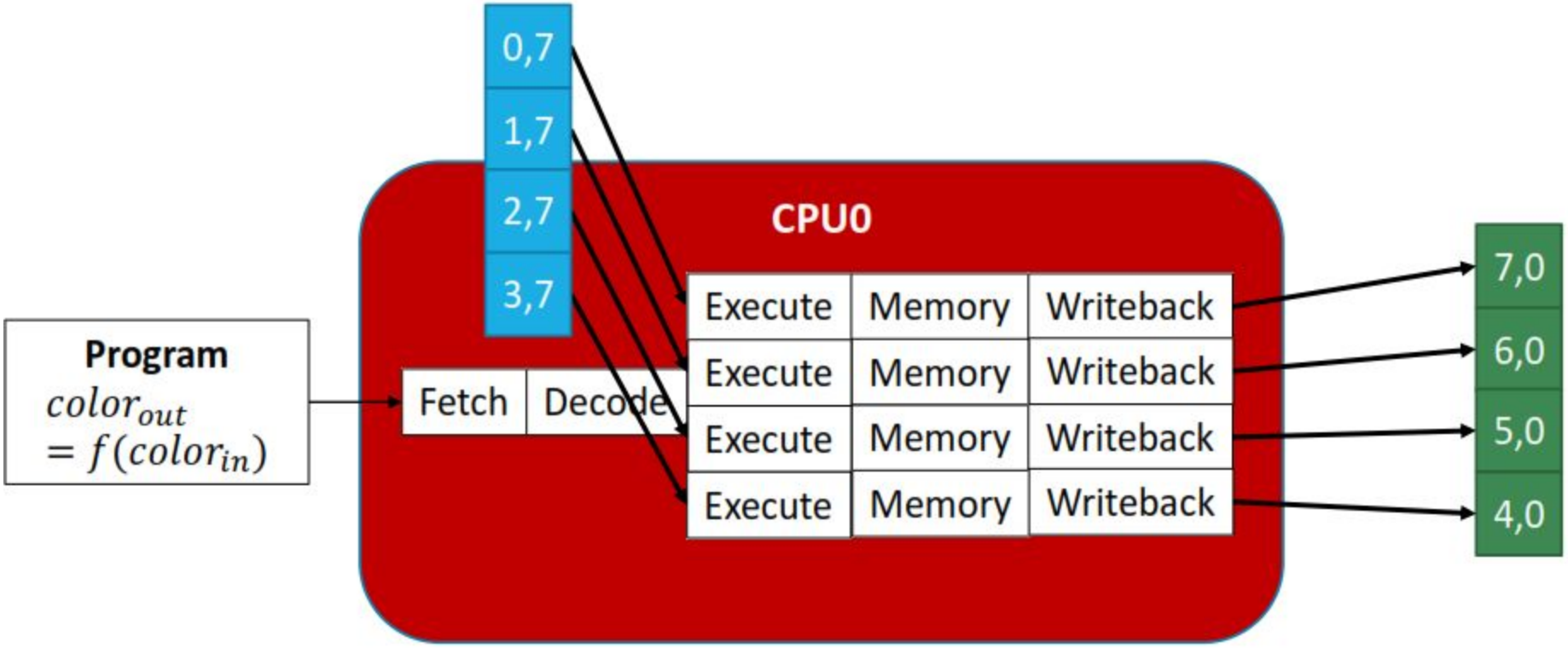
Data Parallelism: An SPMD Approach

- **Single Program Multiple Data**
- Split **identical, independent work over multiple processors**



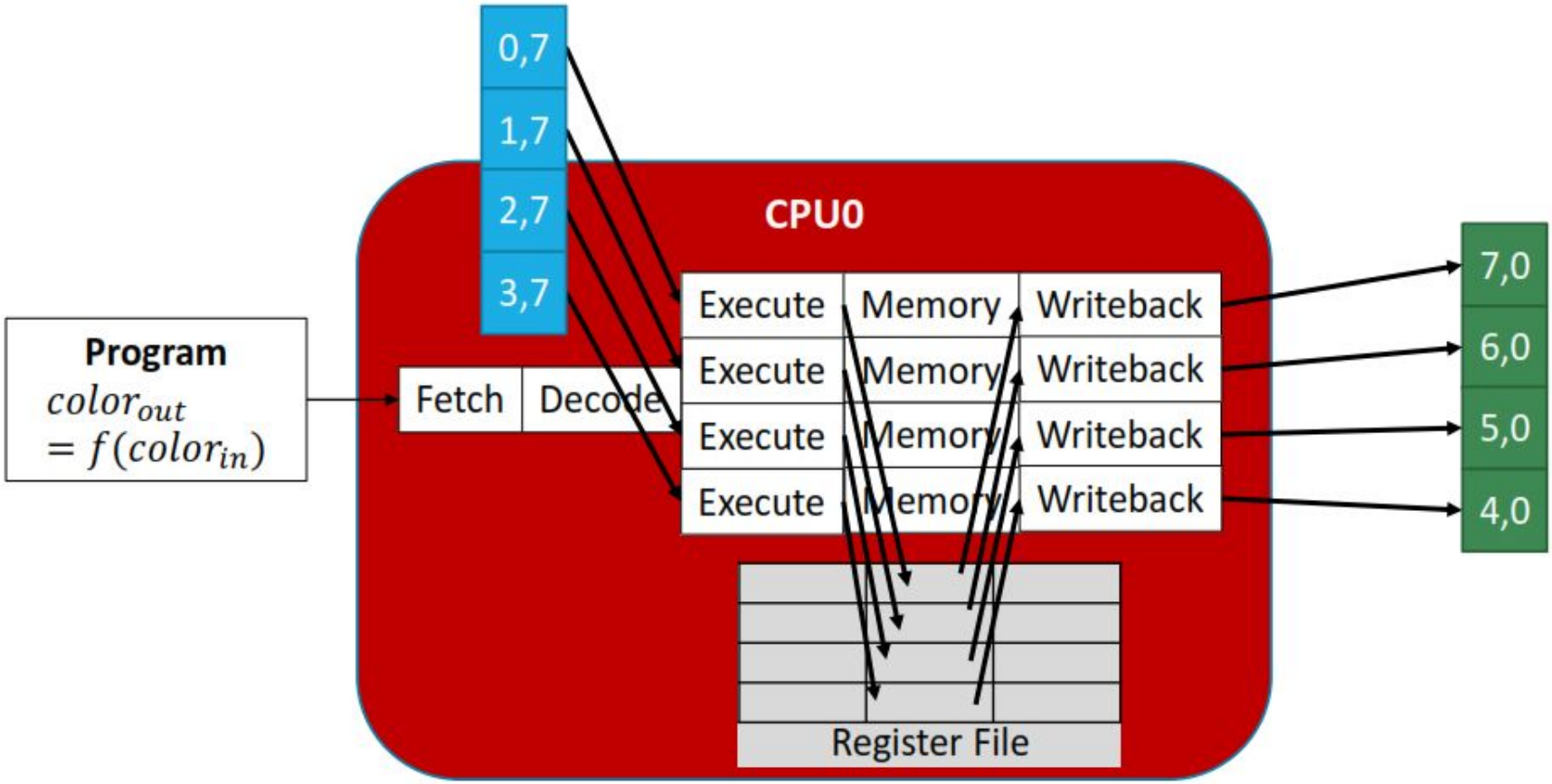
Data Parallelism: A SIMD Approach

- **Single Instruction Multiple Data**
- Split **identical, independent work over multiple execution units (lanes)**
- More efficient: Eliminate redundant fetch/decode



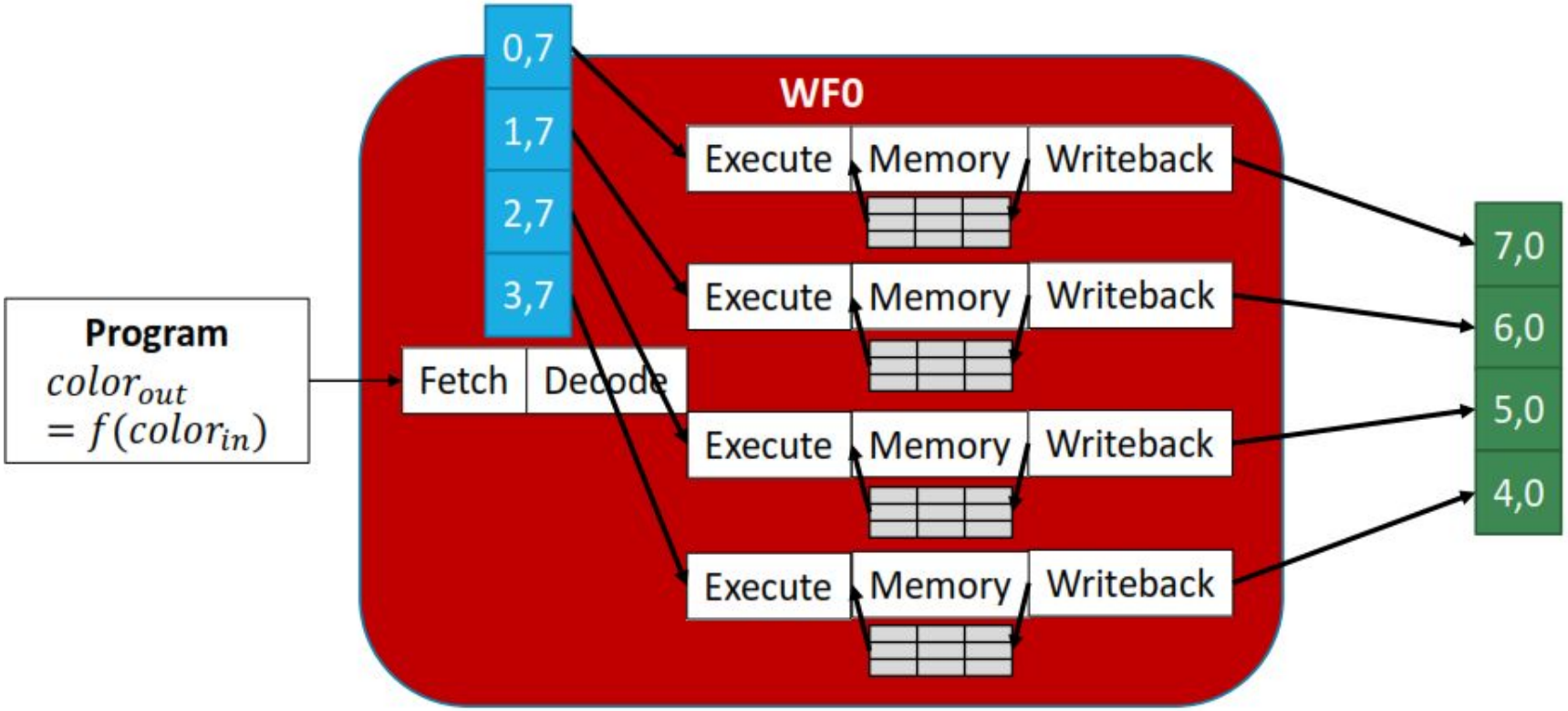
SIMD: A Closer Look

- One Thread + Data Parallel Ops ⍰ Single PC, single register file



Data Parallelism: A SIMT Approach

- **Single Instruction Multiple Thread**
- Split **identical, independent work over multiple *lockstep threads***
- Multiple Threads + Scalar Ops ⇨ One PC, Multiple register files



Data Parallel Execution Models

MIMD/SPMD

Multiple **independent** threads

SIMD/Vector

One thread with wide execution datapath

SIMT

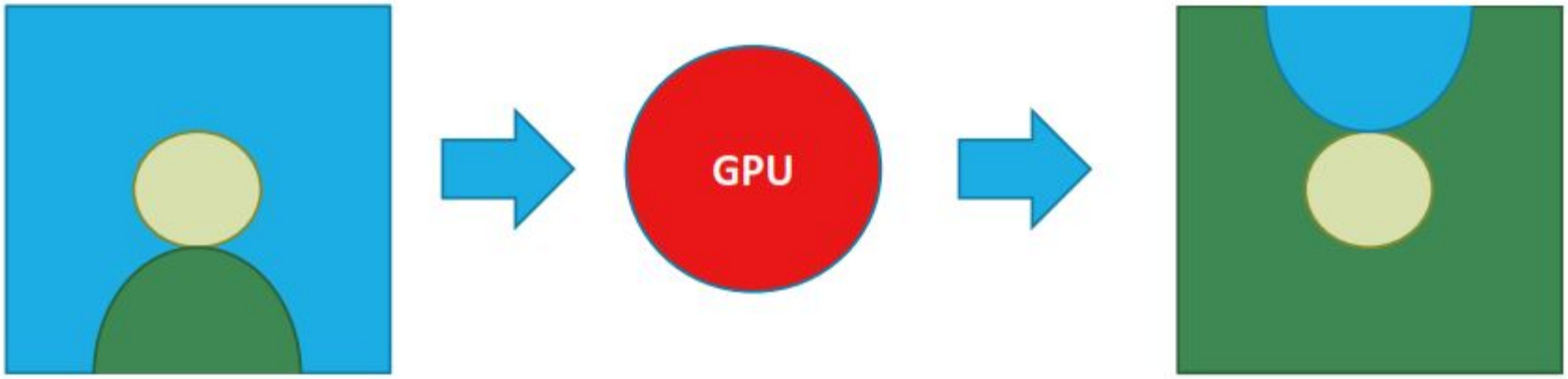
Multiple **lockstep** threads

Execution Model Comparison

	MIMD/SPMD	SIMD/Vector	SIMT
			
Example Architecture	Multicore CPUs	x86 SSE/AVX	GPUs
Pros	More general: supports TLP	Can mix sequential & parallel code	Easier to program Gather/Scatter operations
Cons	Inefficient for data parallelism	Gather/Scatter can be awkward	Divergence kills performance

GPUs and Memory

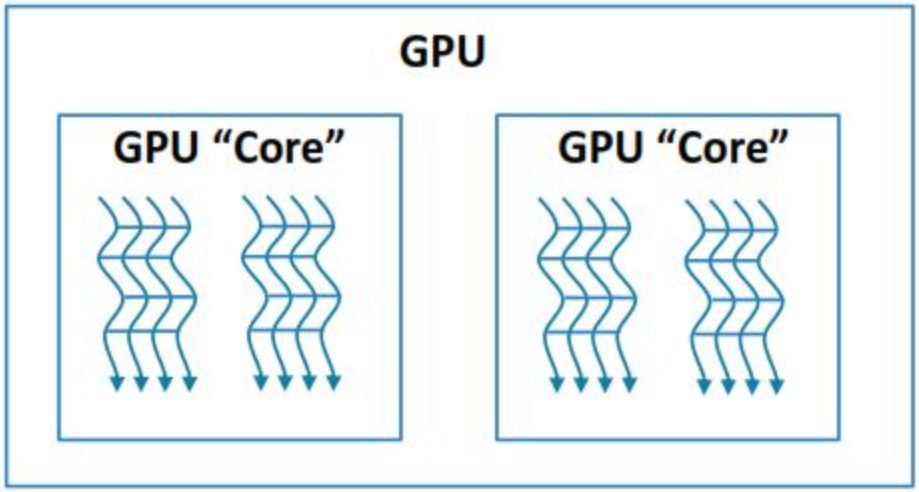
- Recall: GPUs perform ***Streaming computation***
  Streaming memory access



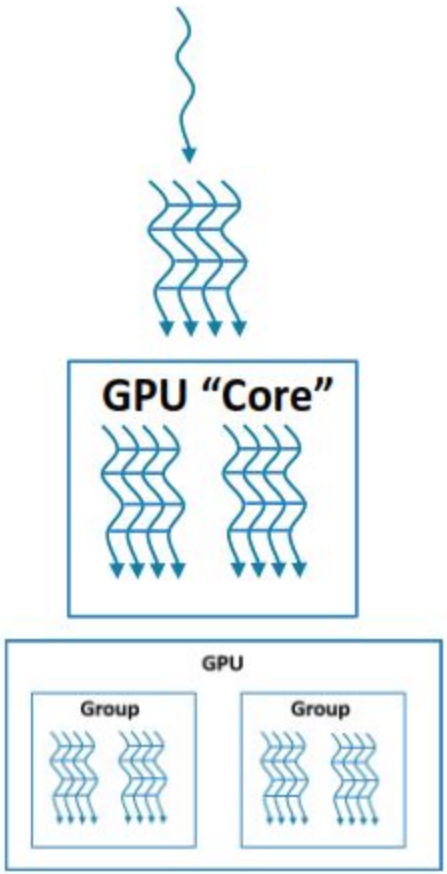
DRAM latency: 100s of GPU cycles
How do we keep the GPU busy (*hide memory latency)?*

Multicore Multithreaded SIMT

- Many SIMT “threads” grouped together into GPU “Core”
- SIMT threads in a group == SMT threads in a CPU core
- ◦ Unlike CPU, groups are exposed to programmers
- Multiple GPU “Cores”



Terminology



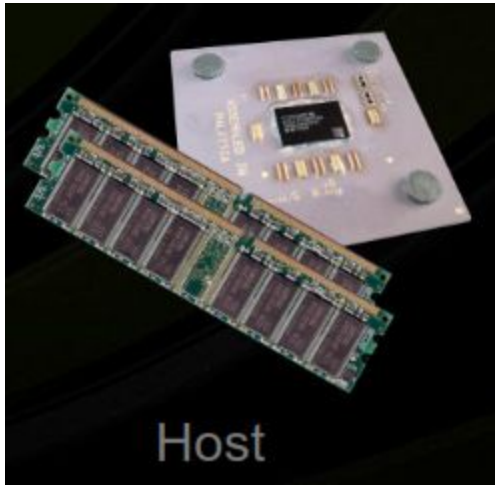
Nvidia/CUDA	AMD/OpenCL	Derek's CPU Analogy
CUDA Processor	Processing Element	Lane
CUDA Core	SIMD Unit	Pipeline
Streaming Multiprocessor	Compute Unit	Core
GPU Device	GPU Device	Device

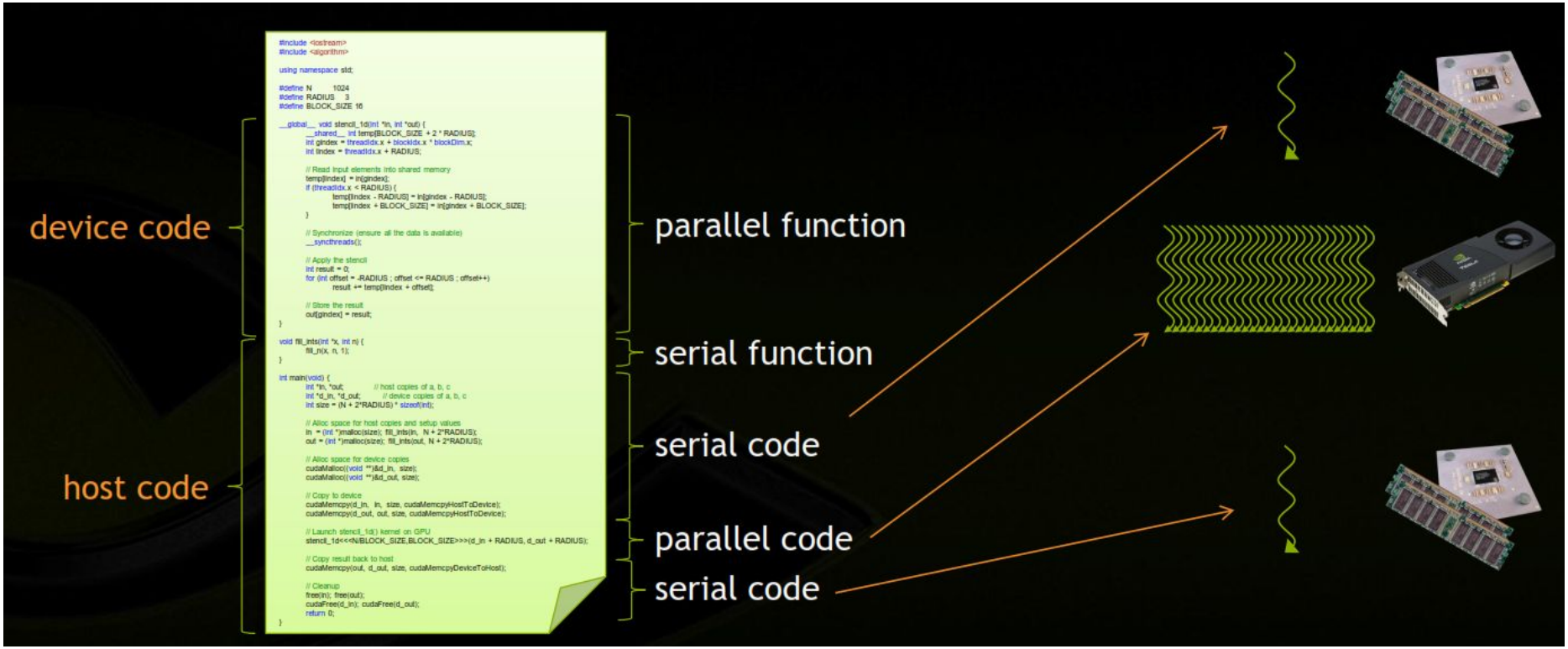
GPU Programming Models

- **CUDA – Compute Unified Device Architecture**
  - Developed by Nvidia -- proprietary
  - First serious GPGPU language/environment
- **OpenCL – Open Computing Language**
  - From makers of OpenGL
  - Wide industry support: AMD, Apple, Qualcomm, Nvidia, etc.
- **C++ AMP – C++ Accelerated Massive Parallelism**
  - Microsoft
  - Much higher abstraction that CUDA/OpenCL
- **OpenACC – Open Accelerator**
  - Like OpenMP for GPUs (semi-auto-parallelize serial code)
  - Much higher abstraction than CUDA/OpenCL

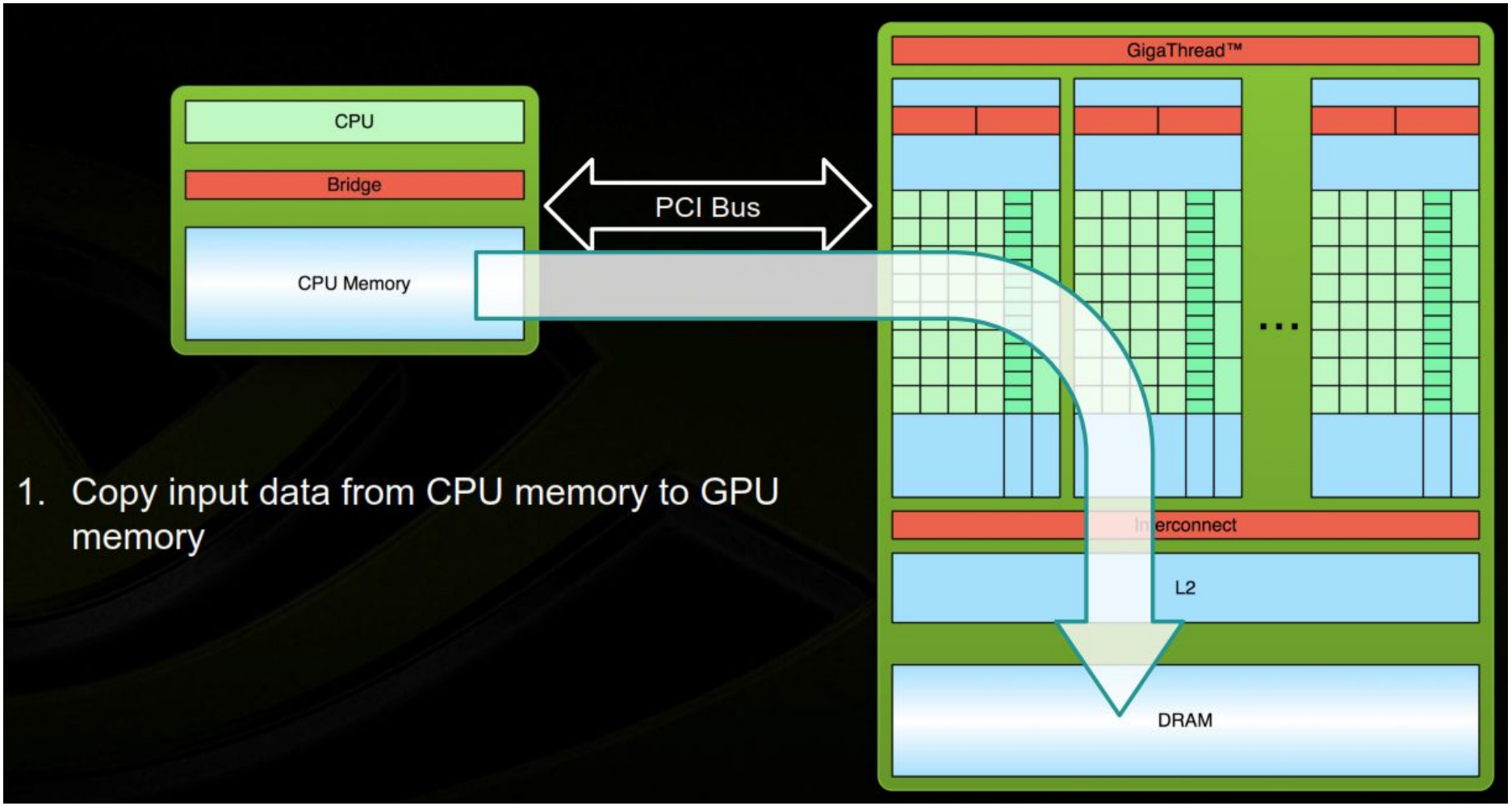
Terminology

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



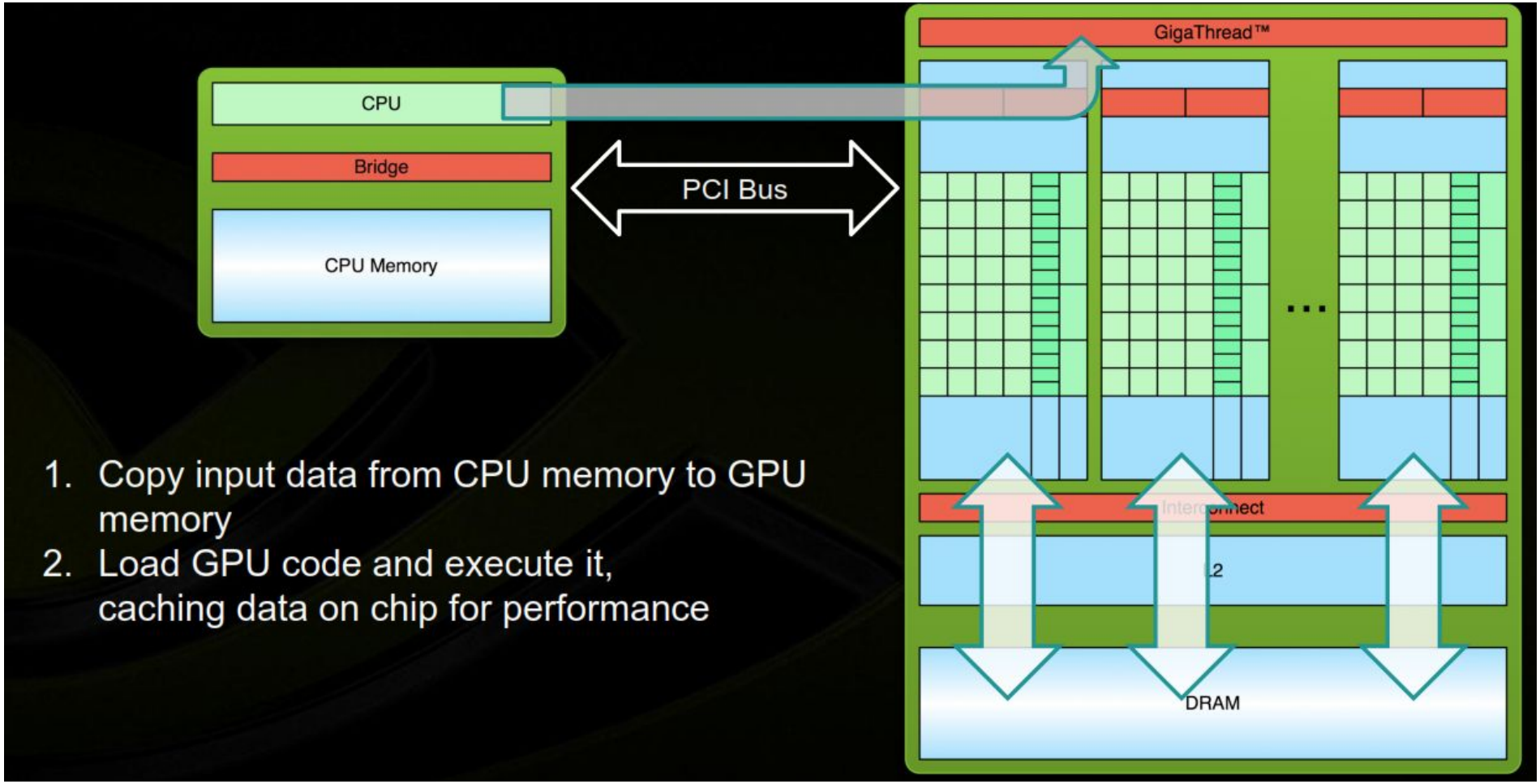

device code
host code
parallel function
serial function
serial code
parallel code
serial code
#include <iostream>
#include <algorithm>
using namespace std;
#define N 1024
#define RADIUS 3
#define BLOCK_SIZE 16
__global__ void stencil_1d(int *in, int *out) {
__shared__ int temp[BLOCK_SIZE + 2 * RADIUS];
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();
// Apply the stencil
int result = 0;
for (int offset = -RADIUS ; offset <= RADIUS ; offset++)
result += temp[lindex + offset];
// Store the result
out[gindex] = result;
}
void fill_ints(int *x, int n) {
fill_n(x, n, 1);
}
int main(void) {
int *in, *out; // host copies of a, b, c
int *d_in, *d_out; // device copies of a, b, c
int size = (N + 2*RADIUS) * sizeof(int);
// Alloc space for host copies and setup values
in = (int *)malloc(size); fill_ints(in, N + 2*RADIUS);
out = (int *)malloc(size); fill_ints(out, N + 2*RADIUS);
// Alloc space for device copies
cudaMalloc((void **)&d_in, size);
cudaMalloc((void **)&d_out, size);
// Copy to device
cudaMemcpy(d_in, in, size, cudaMemcpyHostToDevice);
cudaMemcpy(d_out, out, size, cudaMemcpyHostToDevice);
// Launch stencil_1d() kernel on GPU
stencil_1d<<<N/BLOCK_SIZE,BLOCK_SIZE>>>(d_in + RADIUS, d_out + RADIUS);
// Copy result back to host
cudaMemcpy(out, d_out, size, cudaMemcpyDeviceToHost);
// Cleanup
free(in); free(out);
cudaFree(d_in); cudaFree(d_out);
return 0;
}

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 code and execute it, caching data on chip for performance

Simple Processing Flow