GPU Architectures

A CPU Perspective

BRAD BECKMANN

Goals

Data Parallelism: What is it, and how to exploit it?

- Workload characteristics

Parallel Execution Models

- MIMD, SIMD, **SIMT**

GPU Compute Programming Models

- Intro to OpenCL
- I will slowly introduce new terminology

Modern GPU Microarchitectures

- i.e., programmable GPU pipelines, not their fixed-function predecessors

Advanced Topics: (time permitting)

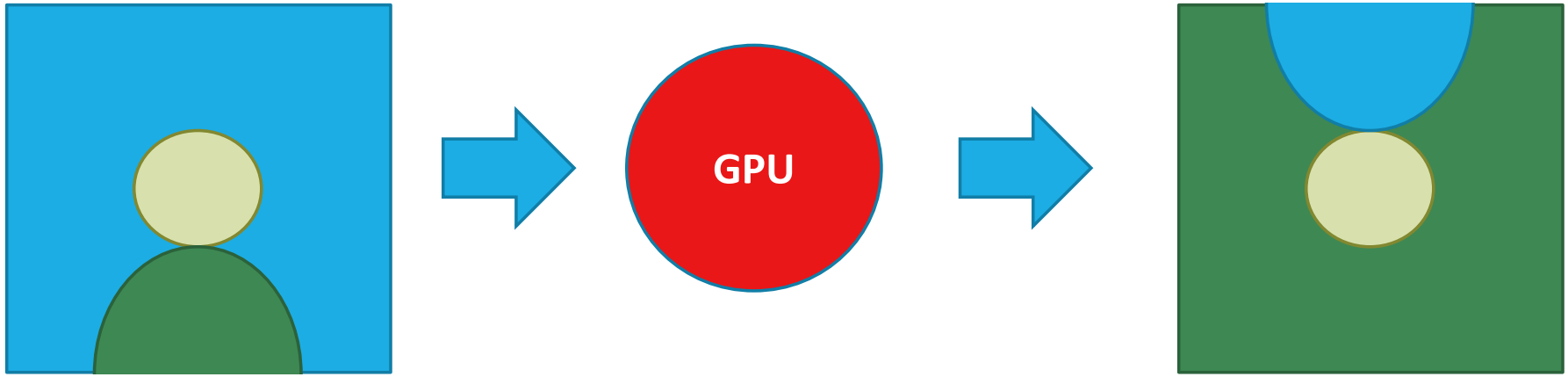
- **The Limits of GPUs:** What they can and cannot do
- **The Future of GPUs:** Where do we go from here?

Data Parallel Execution on GPUs

Data Parallelism, Programming Models, SIMT

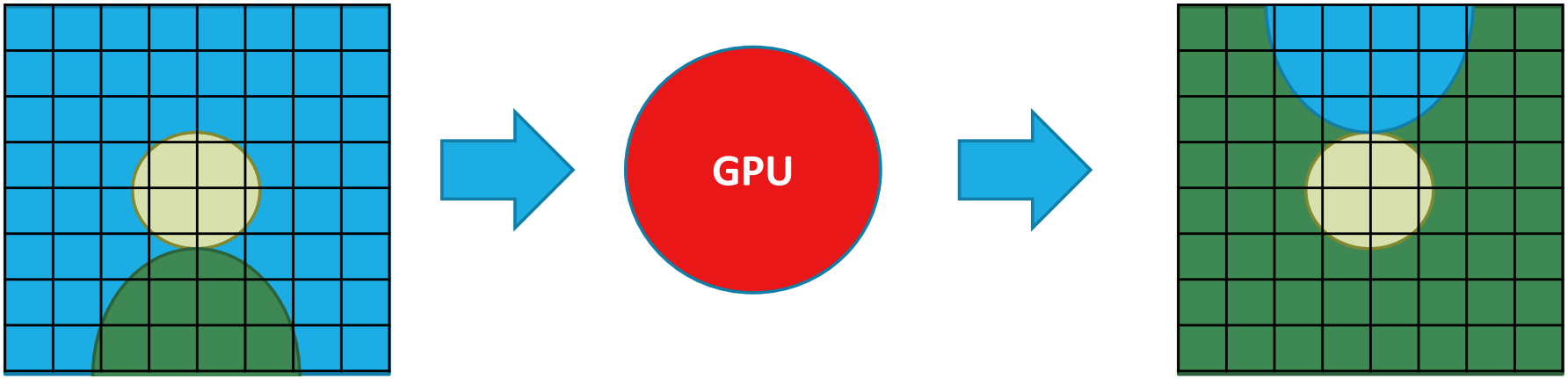
Graphics Workloads

Streaming computation



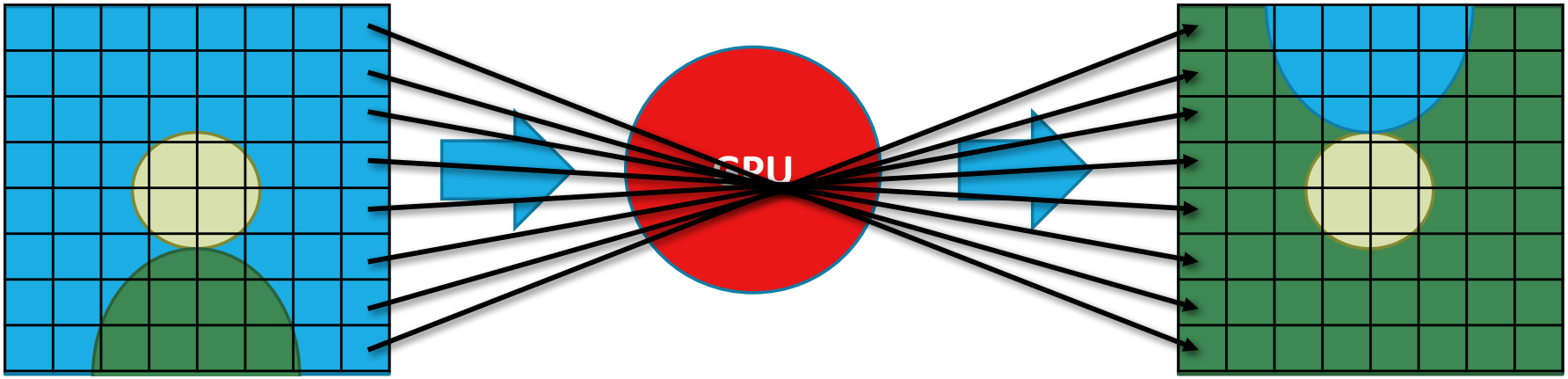
Graphics Workloads

Streaming computation ***on pixels***



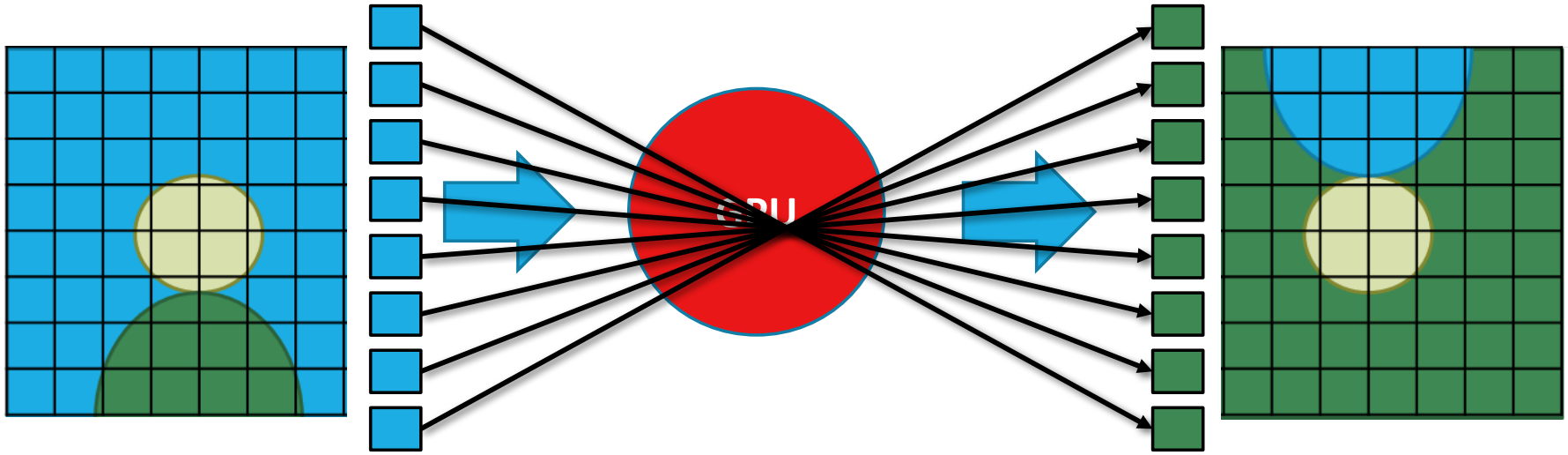
Graphics Workloads

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



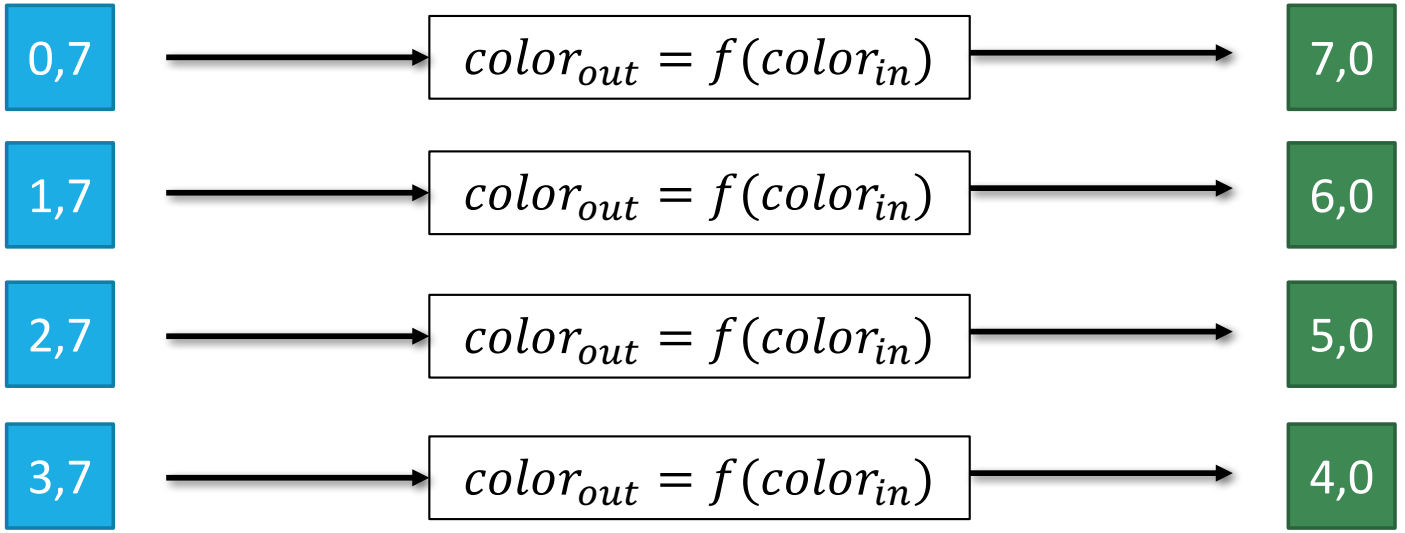
Graphics Workloads

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



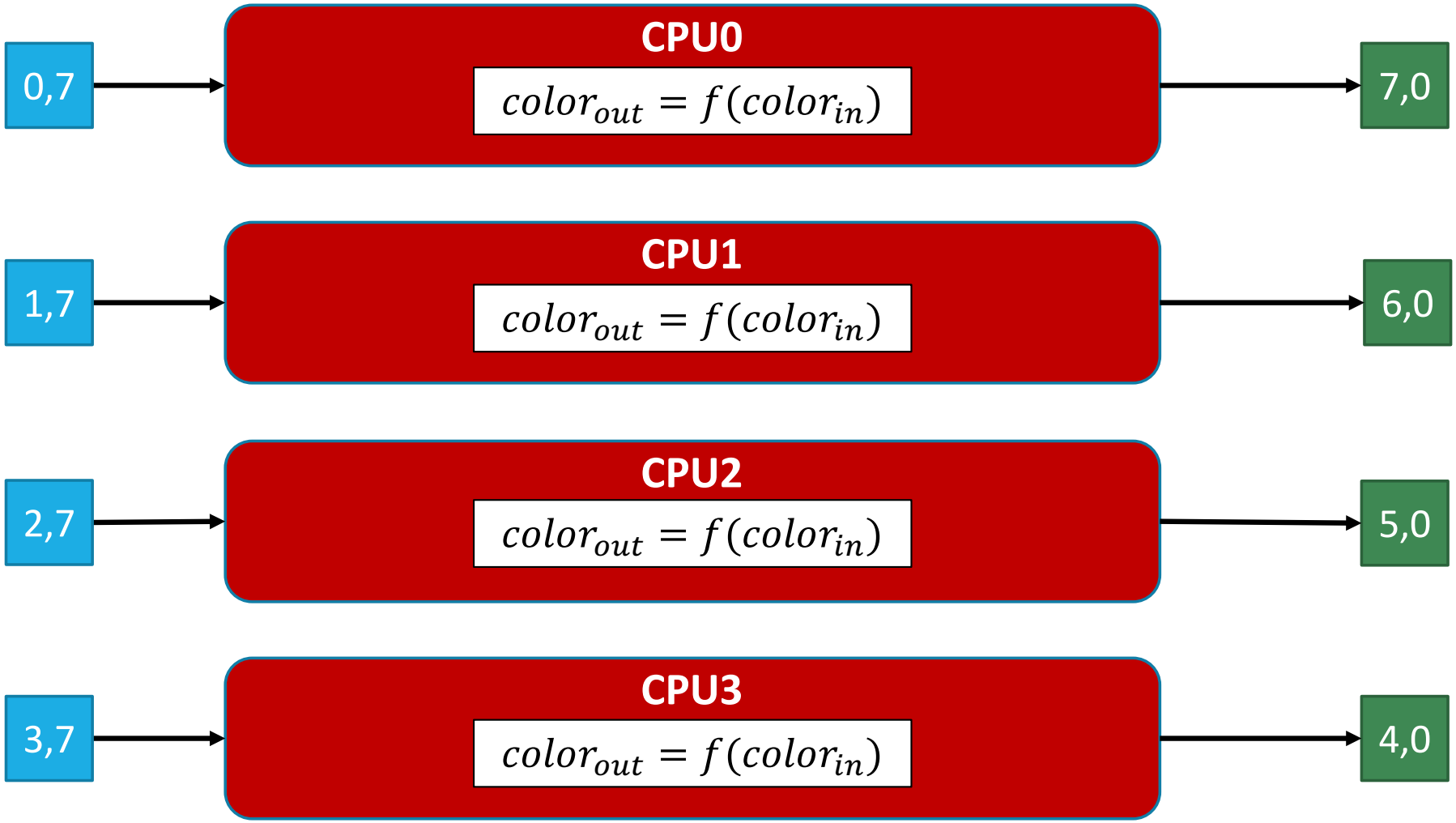
Generalize: Data Parallel Workloads

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



Naïve Approach

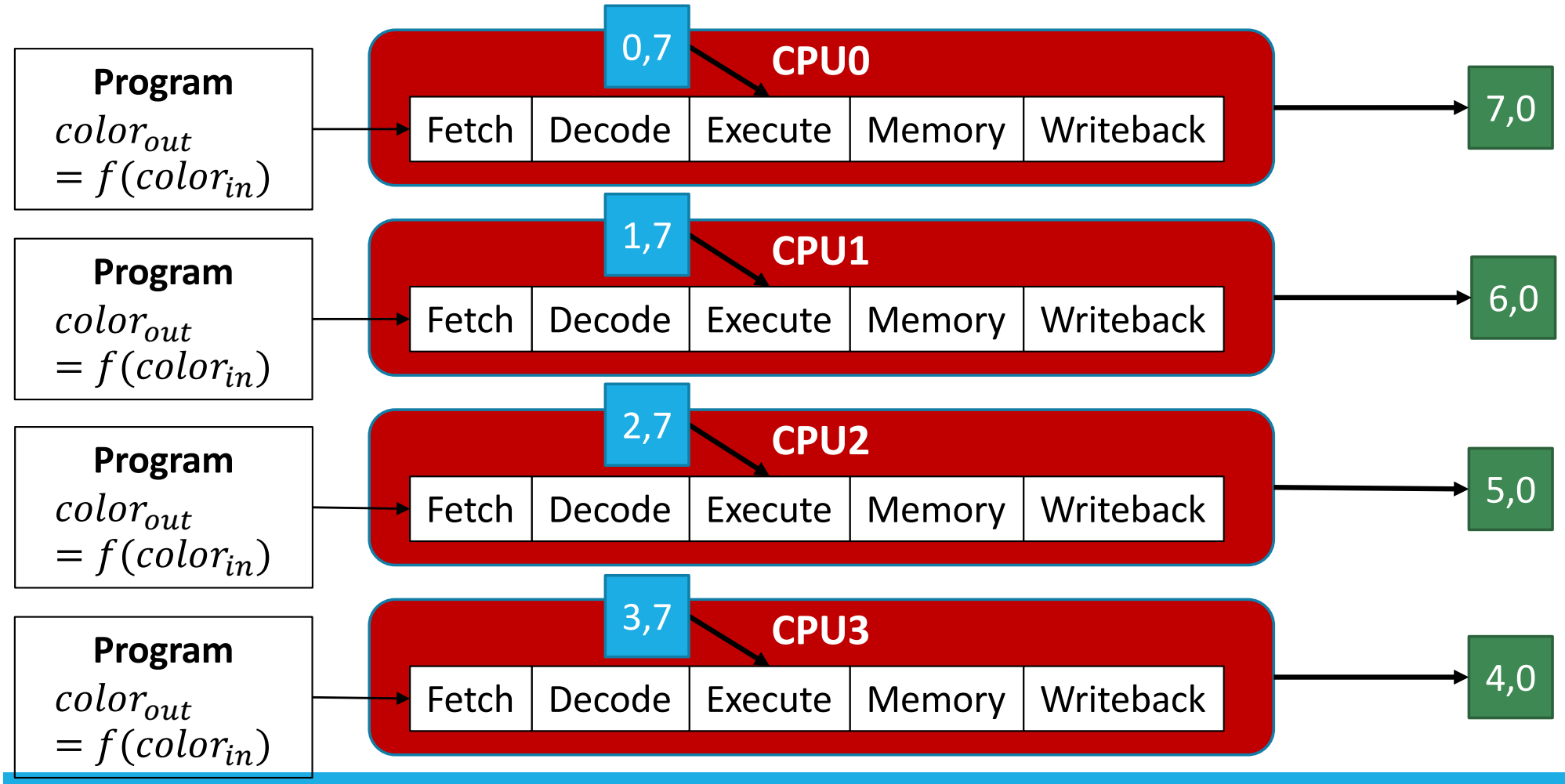
Split **independent** work over **multiple** processors



Data Parallelism: A MIMD Approach

Multiple **I**nstruction **M**ultiple **D**ata

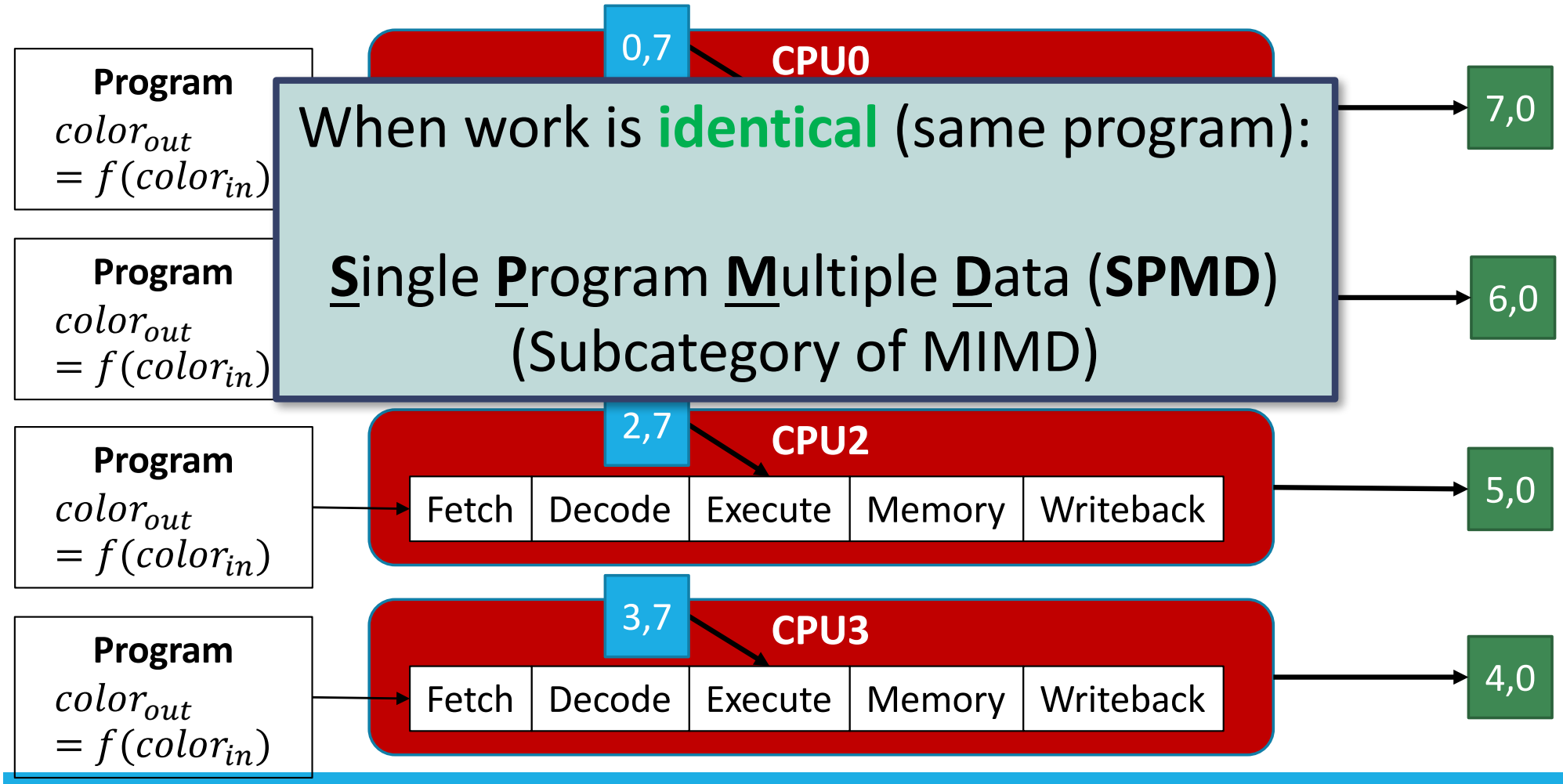
Split **independent** work over **multiple** processors



Data Parallelism: A MIMD Approach

Multiple **I**nstruction **M**ultiple **D**ata

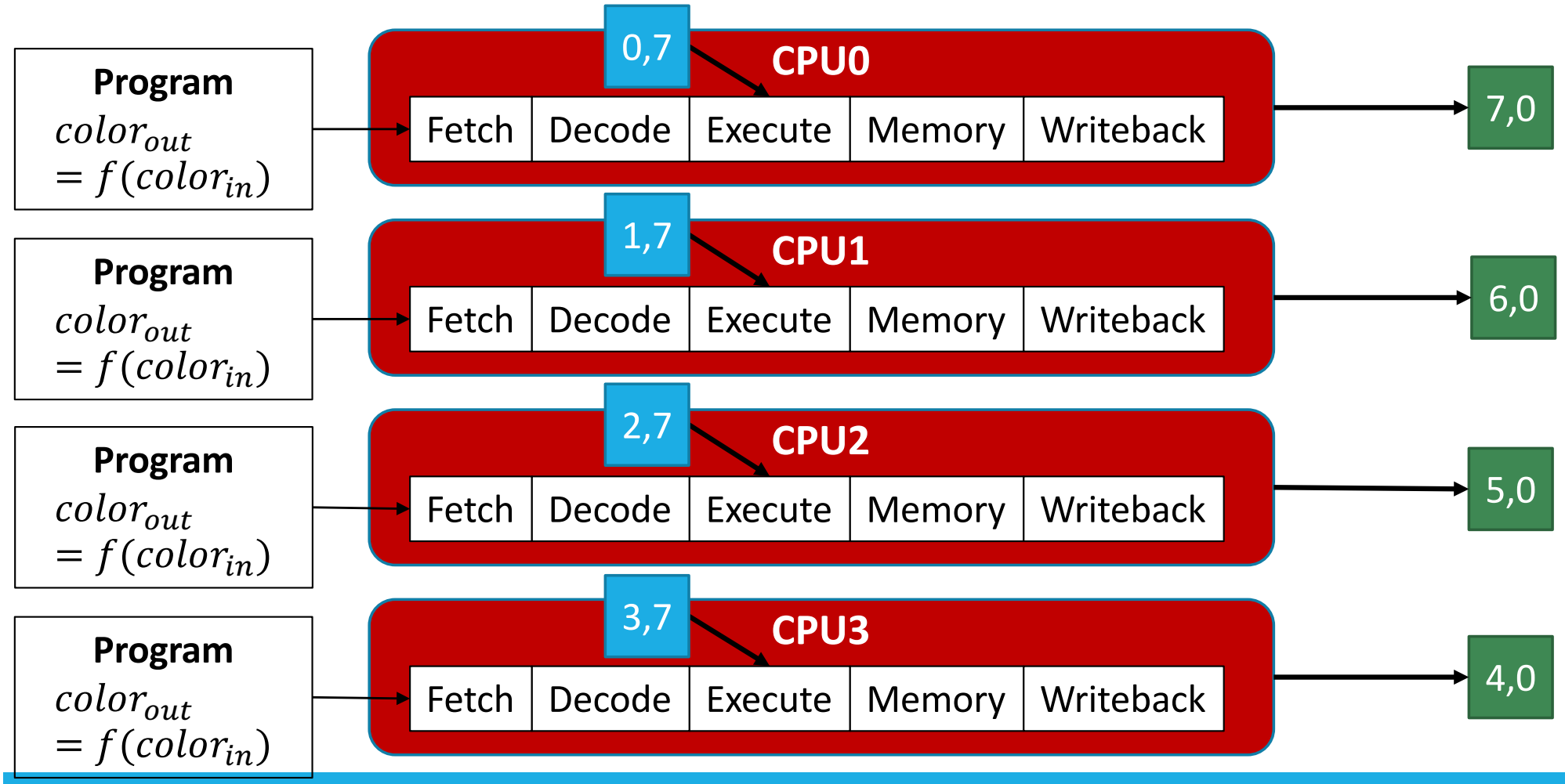
Split **independent** work over **multiple** processors



Data Parallelism: An SPMD Approach

Single **P**rogram **M**ultiple **D**ata

Split **identical, independent** work over **multiple** processors

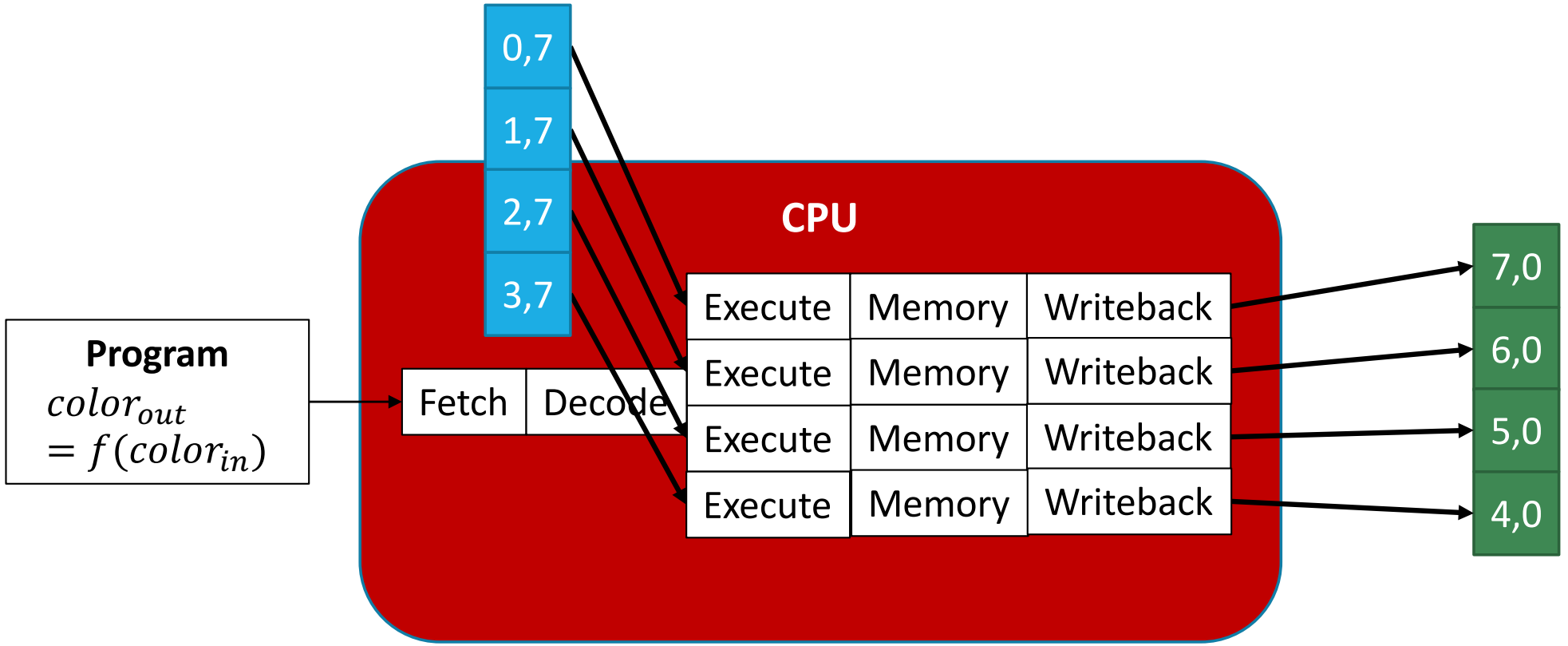


Data Parallelism: A SIMD Approach

Single **I**nstruction **M**ultiple **D**ata

Split **identical, independent** work over **multiple** execution units (lanes)

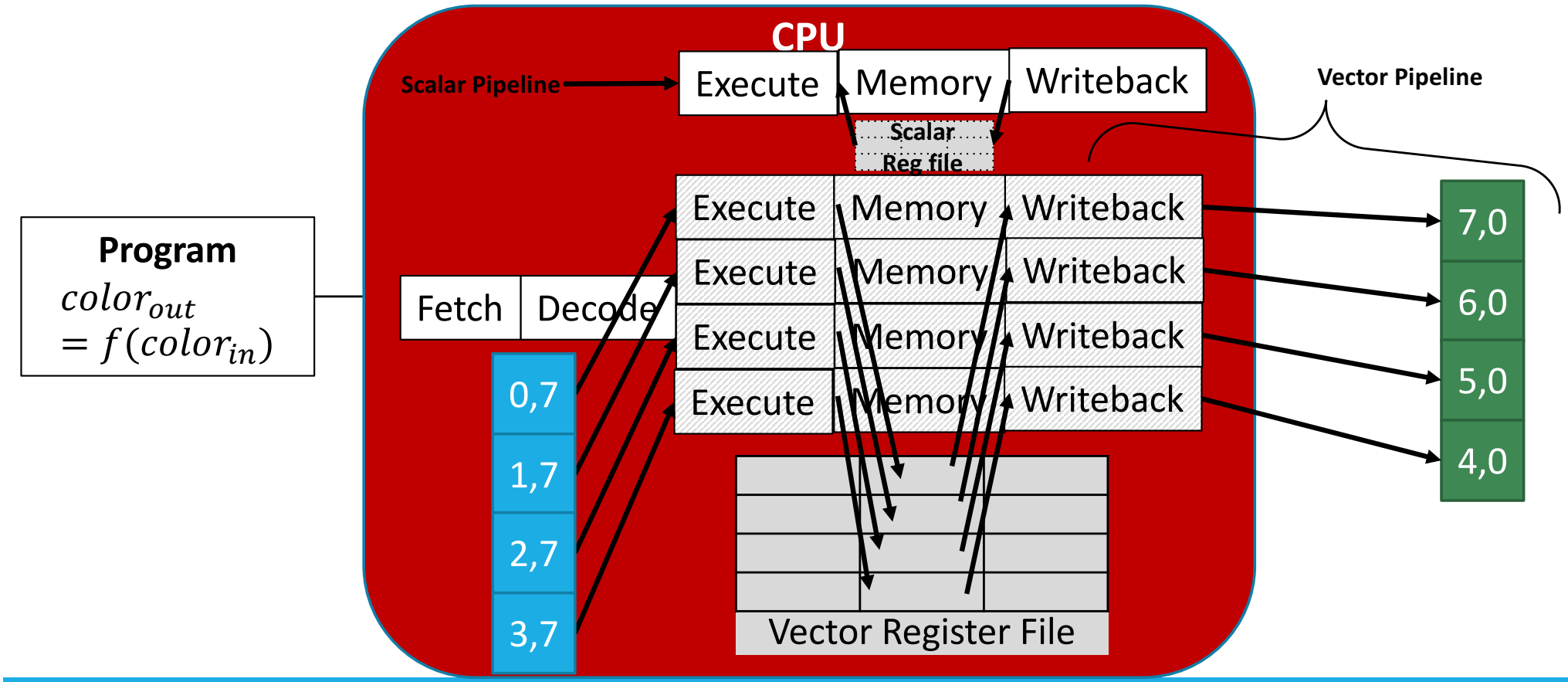
More efficient: Eliminate redundant fetch/decode



SIMD: A Closer Look

One Thread + **some** Data Parallel (vector) Ops → single PC, program explicitly manages scalar vs. vector instructions

ex. x86 SSE/AVX



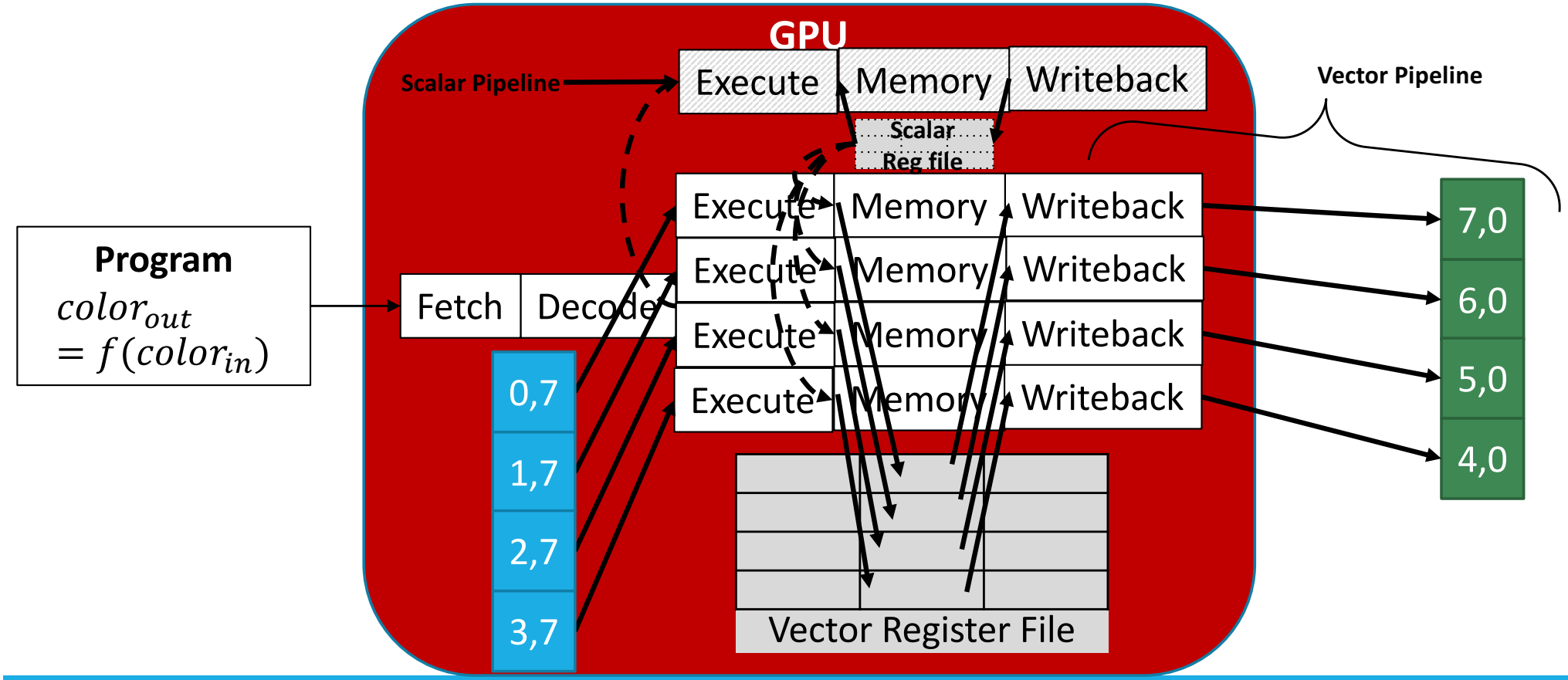
Data Parallelism: A SIMT Approach

Single **I**nstruction **M**ultiple **T**hread

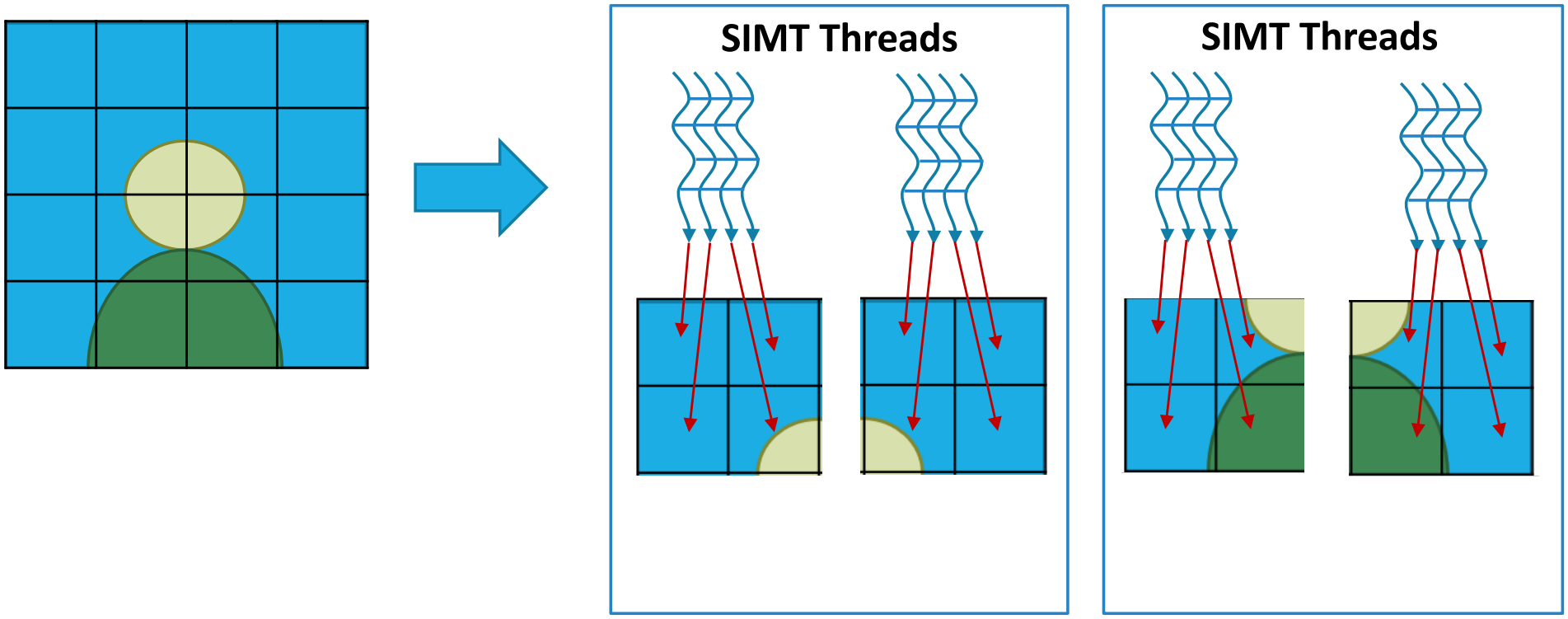
Split **identical, independent** work over **multiple *lockstep*** threads

Implicit data parallel → **complete** vector ops + **some** scalar ops

ex. HSAIL SMIT ISA compiled to AMD GCN ISA

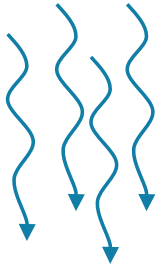


Threads to hardware



Data Parallel Execution Models

MIMD



Multiple **independent** threads

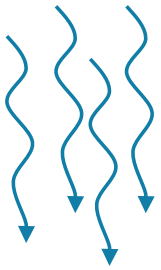
SIMD/Vector

One thread with wide execution datapath

SIMT

Multiple **lockstep** threads

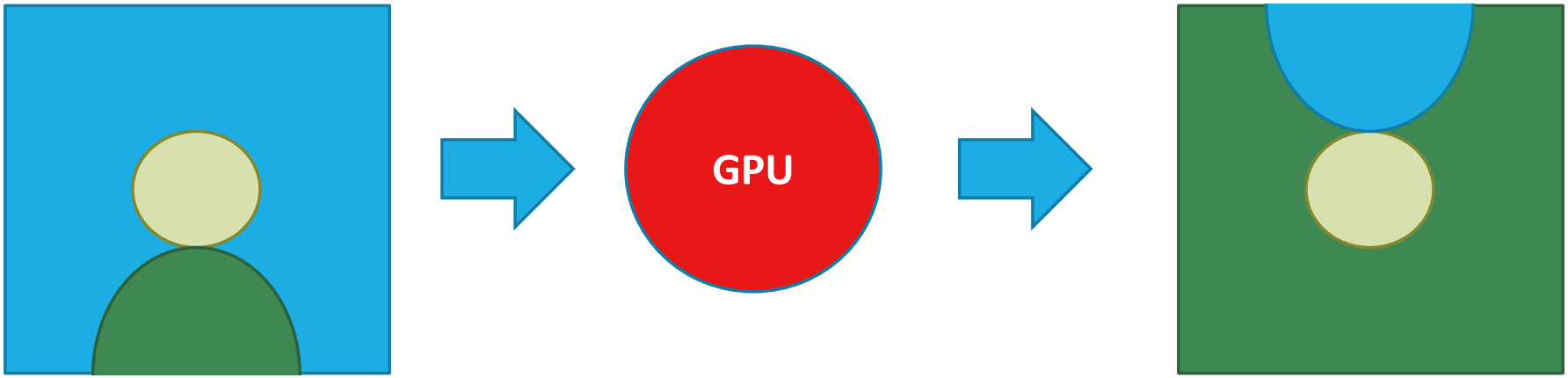
Execution Model Comparison

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

GPUs and Memory

Recall: GPUs perform ***Streaming*** computation →

Streaming memory access



DRAM latency: 100s-1000s of GPU cycles

How do we keep the GPU busy (*hide memory latency*)?

Hiding Memory Latency

Options from the CPU world:

~~Caches~~ ✗
- Need spatial/temporal locality

~~OoO/Dynamic Scheduling~~ ✗
- Need ILP

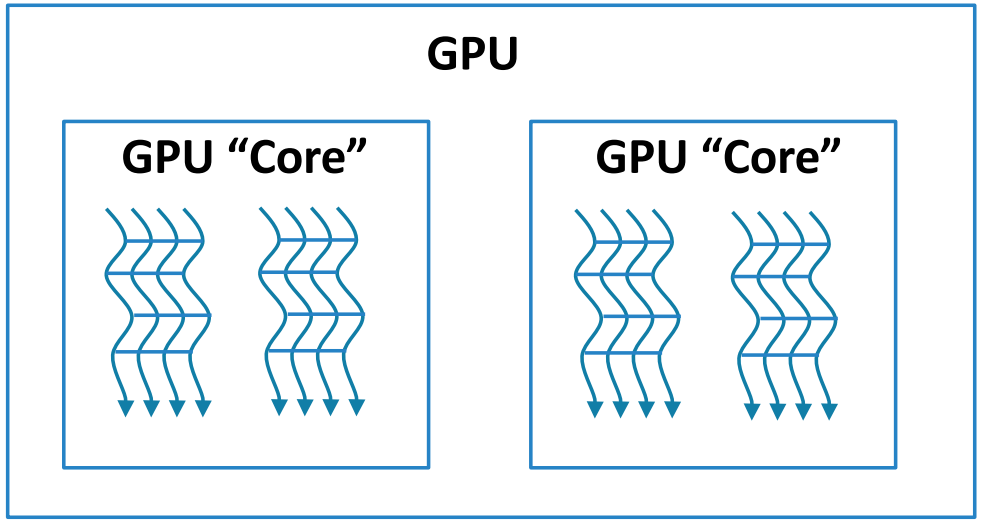
Multicore/Multithreading/SMT ✓
- Need TLP

Multicore Multithreaded SIMT

Many SIMT “threads” grouped together into GPU “Core”

- Note again GPUs support many more threads than CPUs
- The group hierarchy is exposed to programmers

Multiple GPU “Cores” in hardware



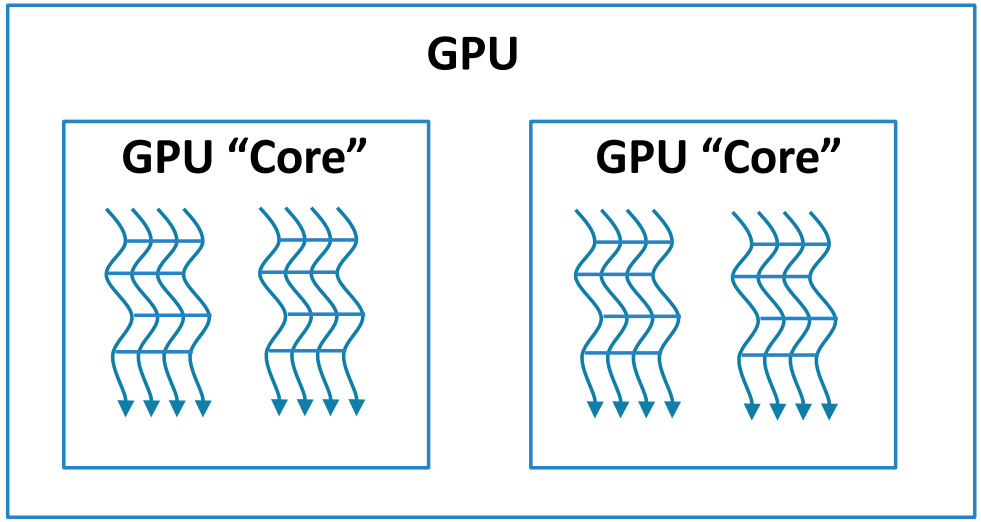
Multicore Multithreaded SIMT

Many SIMT “threads” grouped together into GPU “Core”

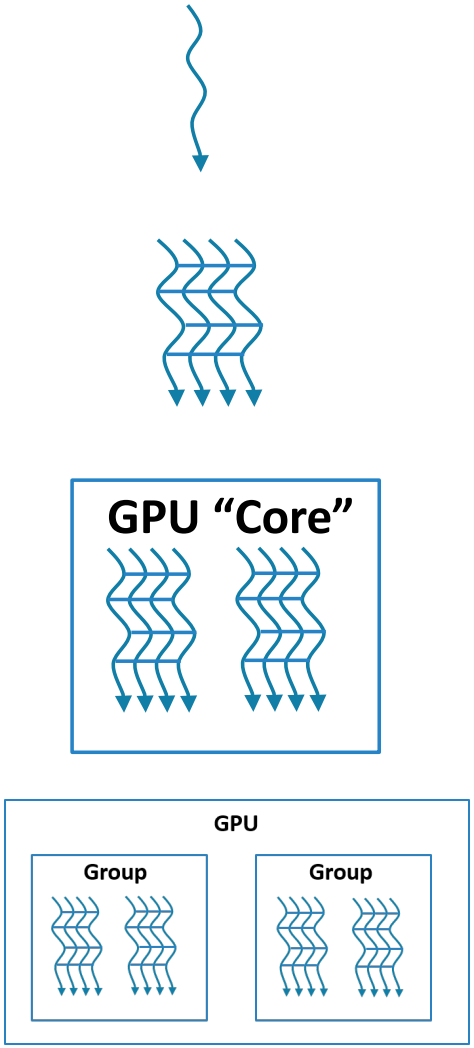
- Note again GPUs support many more threads than CPUs
- The group hierarchy is exposed to programmers

Multiple GPU “Cores” in hardware

This is a GPU Architecture (Whew!)



Generic GPU Hardware Terminology



Lane:
executes a
single thread

GPU Wave:
4 lanes
in lockstep

GPU Core:
supports multiple
SIMD Units

GPU Chip

SIMT Programming Languages

GPUs Going Beyond Graphics

Motivation

Traditionally GPUs were only graphic ASICs

GPU can be more computational efficient for other applications

- Computer vision
- Machine learning
- Bioinformatics
- Signal processing
- Numerical methods
- Network processing
- Finance
- Scientific computing

Each generation GPUs become easier to program

OpenCL

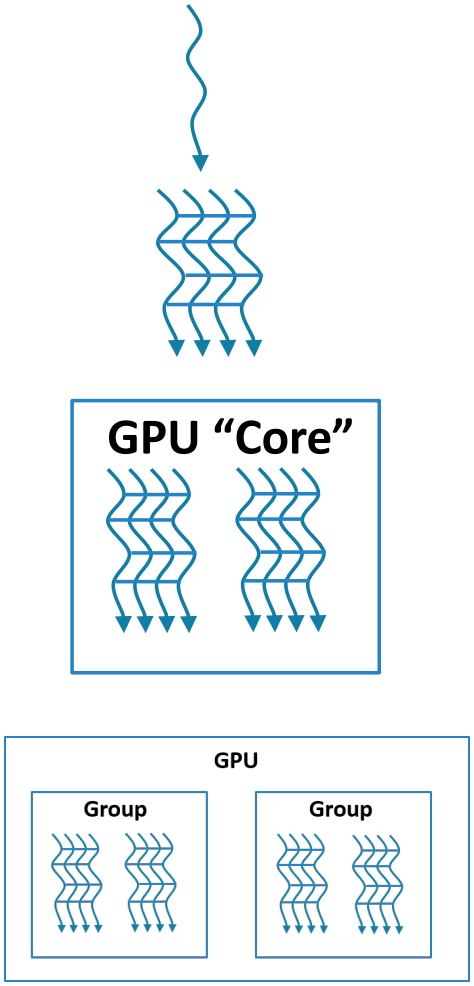
Early CPU languages were light abstractions of physical hardware

- E.g., C

Early GPU languages are light abstractions of physical hardware

- OpenCL and CUDA

How does OpenCL map to the HW

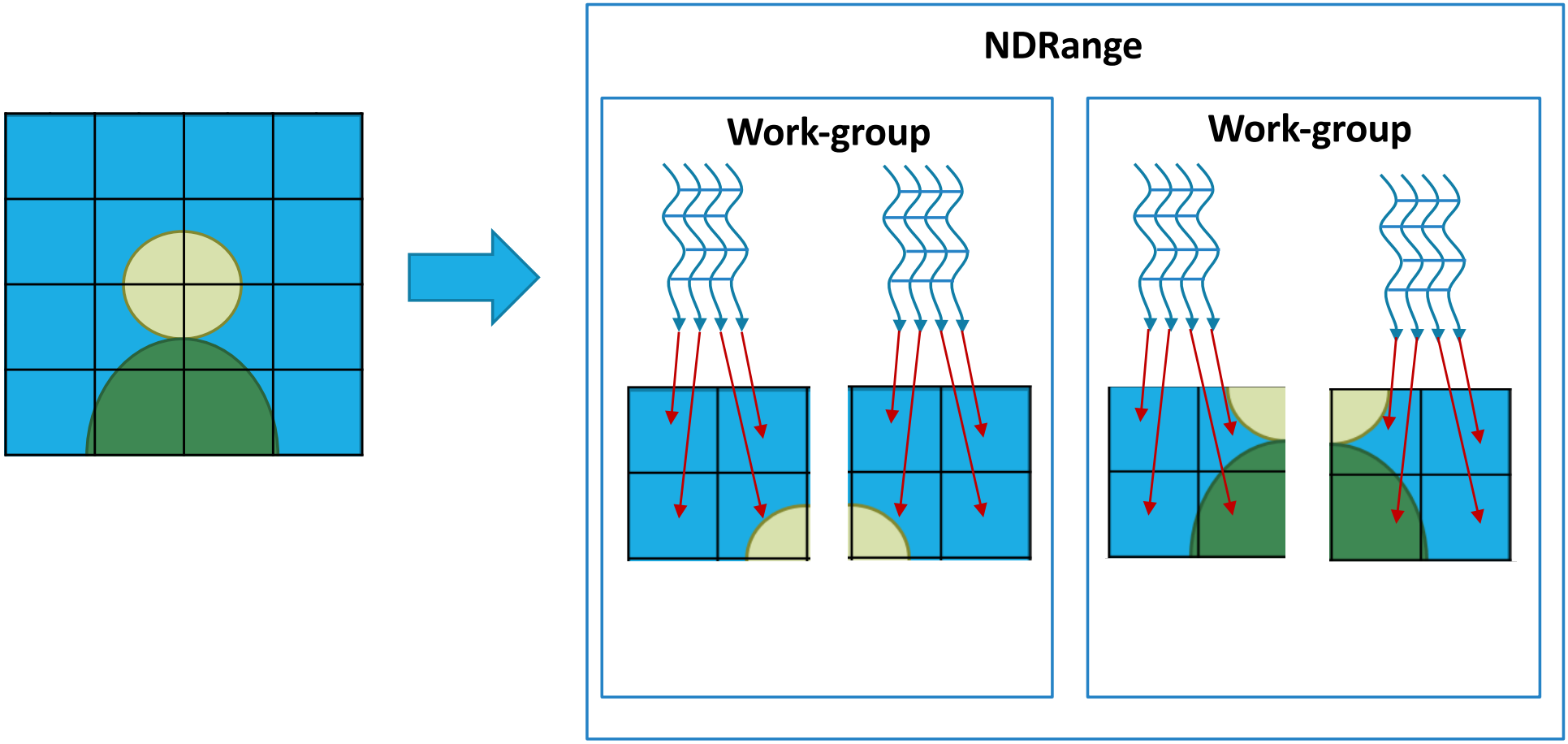


Hardware	OpenCL
Lane	Work-item
GPU Wave	Wavefront: 4 work-items in lock-step
GPU Core	Work-group: 1 or more wavefonts on the same GPU Core
GPU Chip	Device: 1 or more work-groups on the same GPU Chip

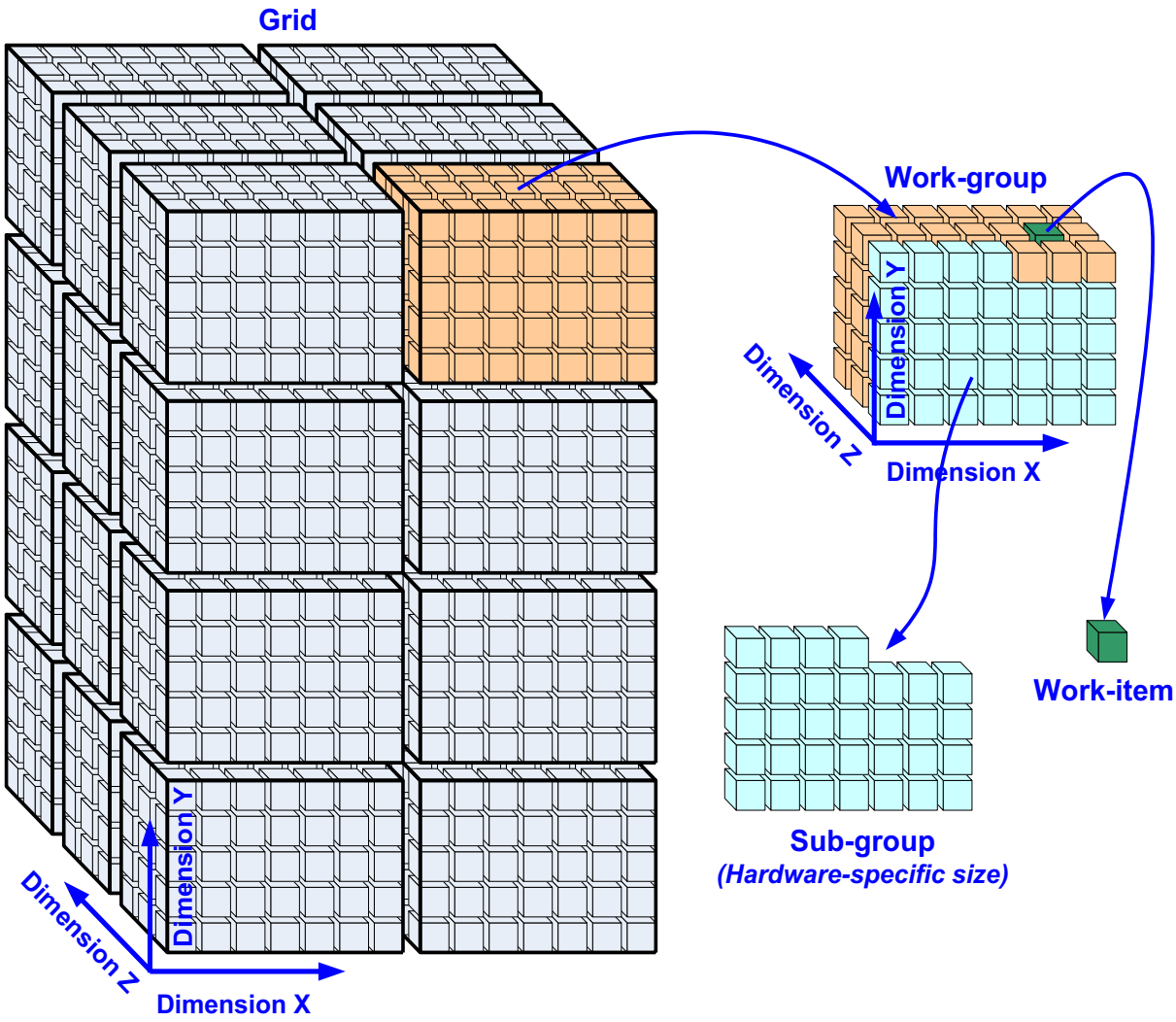
NDRange

N-Dimensional (N = 1, 2, or 3) index space

- Partitioned into work-groups, wavefronts, and work-items



OpenCL Execution Hierarchy

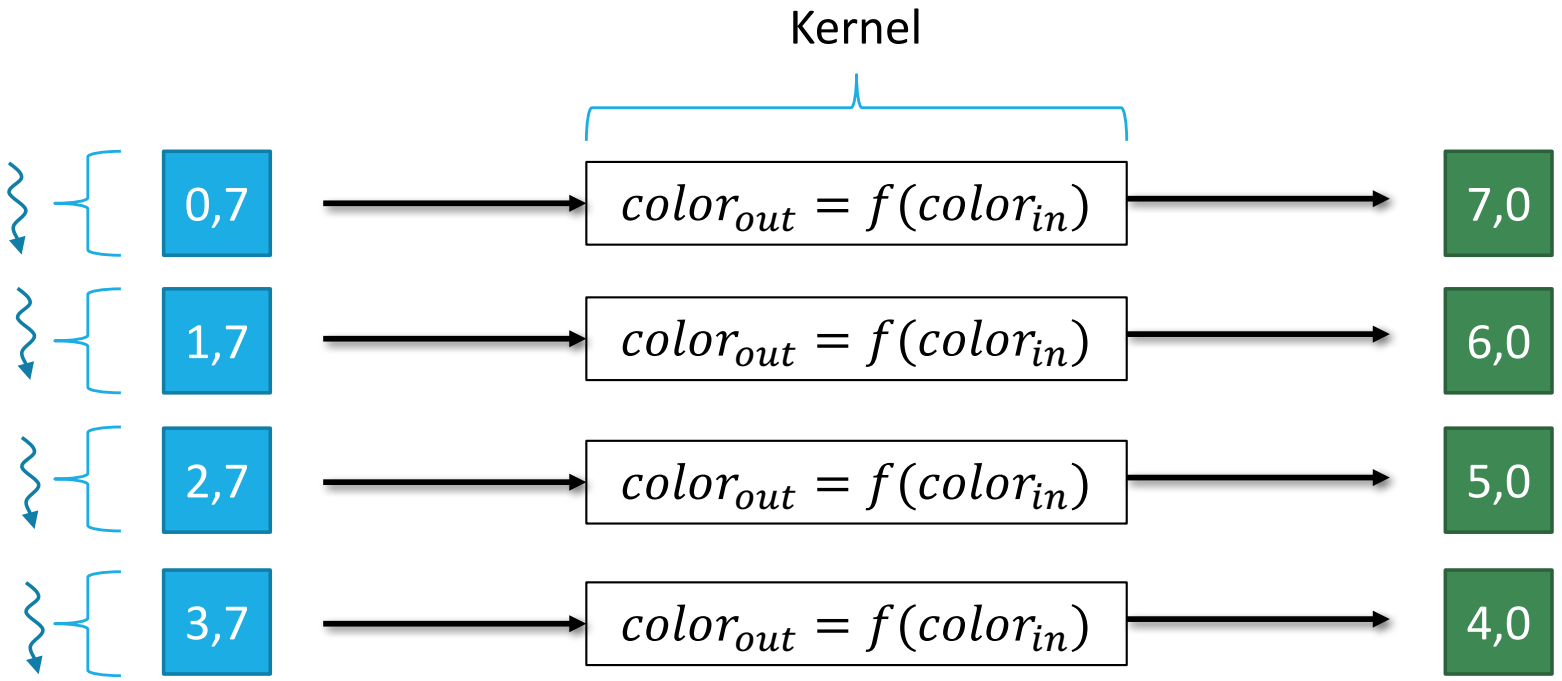


Kernel

Run an NDRange on a **kernel** (i.e., a function)

Same kernel executes for each work-item

- Maps well to SIMT

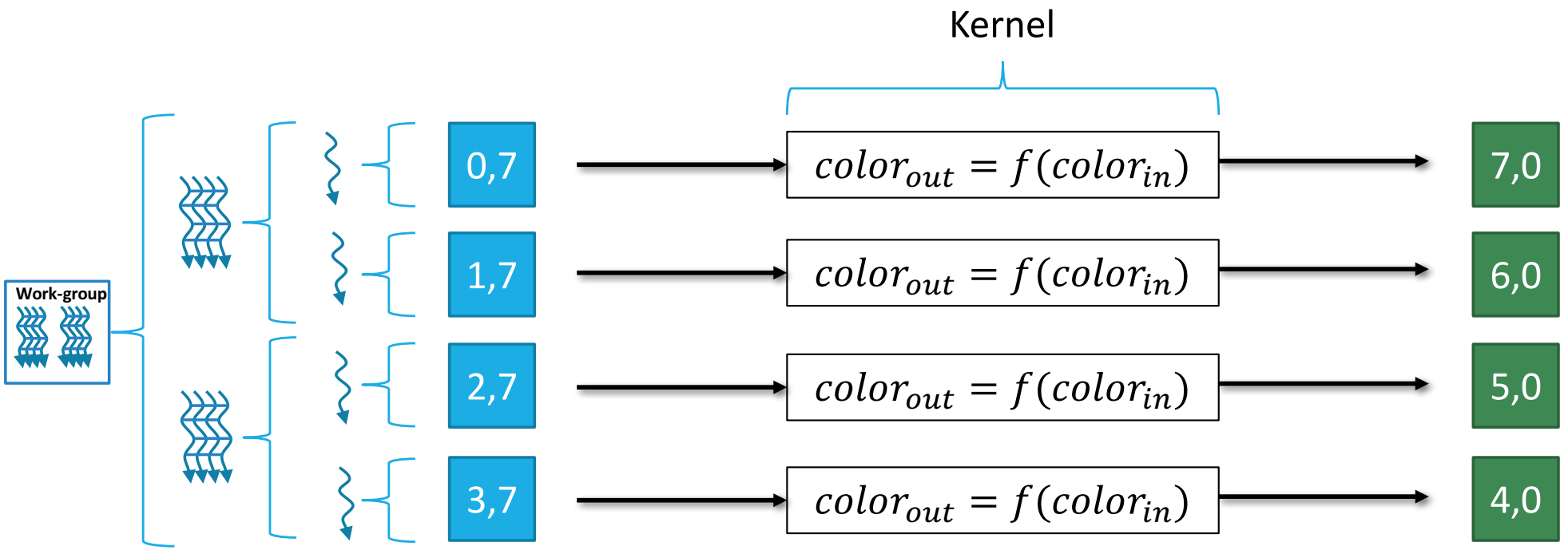


Kernel

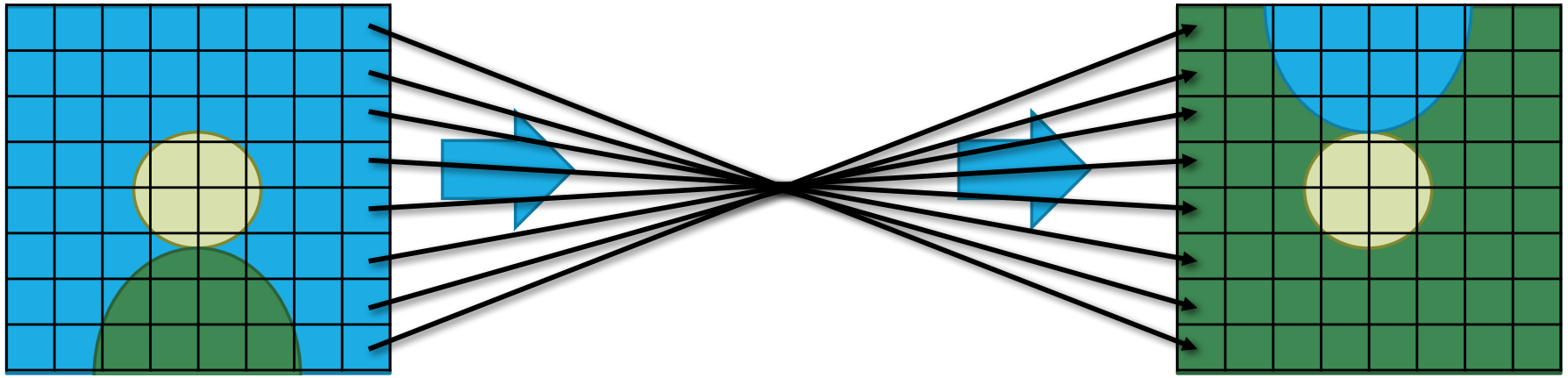
Run an NDRange on a **kernel** (i.e., a function)

Same kernel executes for each work-item

- Maps well to SIMT…**but beware of the execution hierarchy**



OpenCL Code



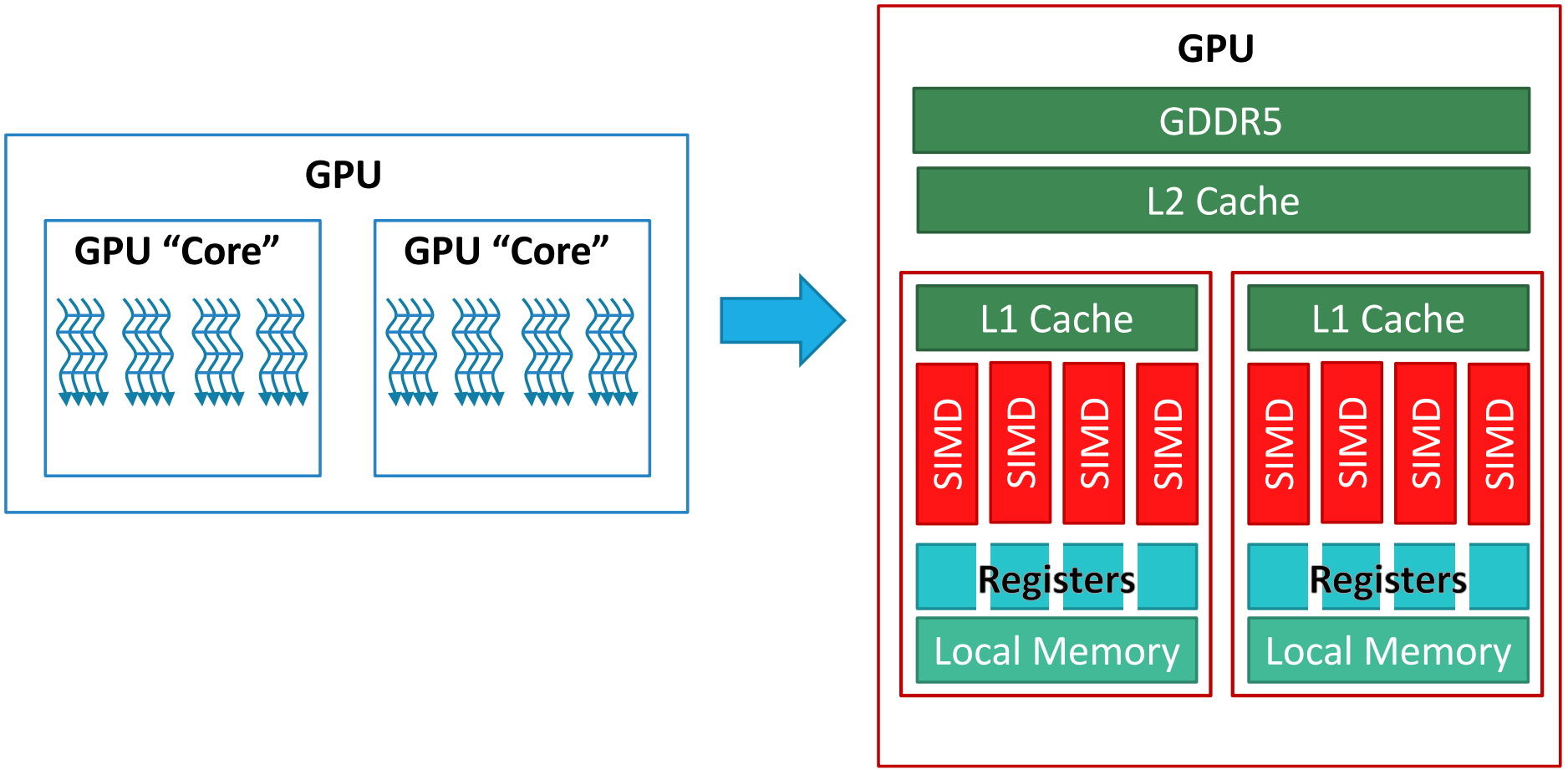
```
__kernel
void flip_and_recolor(__global float3 **in_image,
                      __global float3 **out_image,
                      int img_dim_x, int img_dim_y)
{
    int x = get_global_id(1); // get work-item id in dim 1
    int y = get_global_id(2); // get work-item id in dim 2

    out_image[img_dim_x - x][img_dim_y - y] =
       recolor(in_image[x][y]);
}
```

GPU Microarchitecture

AMD Graphics Core Next

GPU Hardware Overview



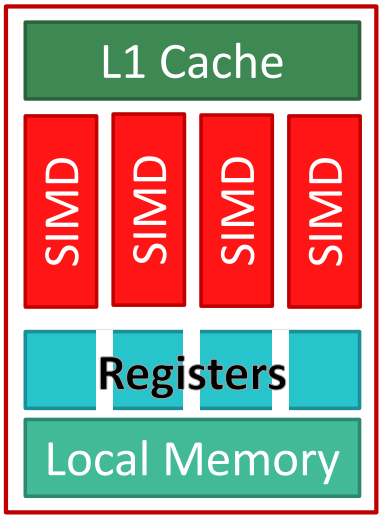
A GPU Core

GPU Core Hardware

- Contains 4 SIMD Units
- Picks one SIMD Unit per cycle for scheduling

SIMD Unit – Runs *Wavefronts*

- Each SIMD Unit has 10 wavefront instruction buffer
- Takes 4 cycles to execute one wavefront

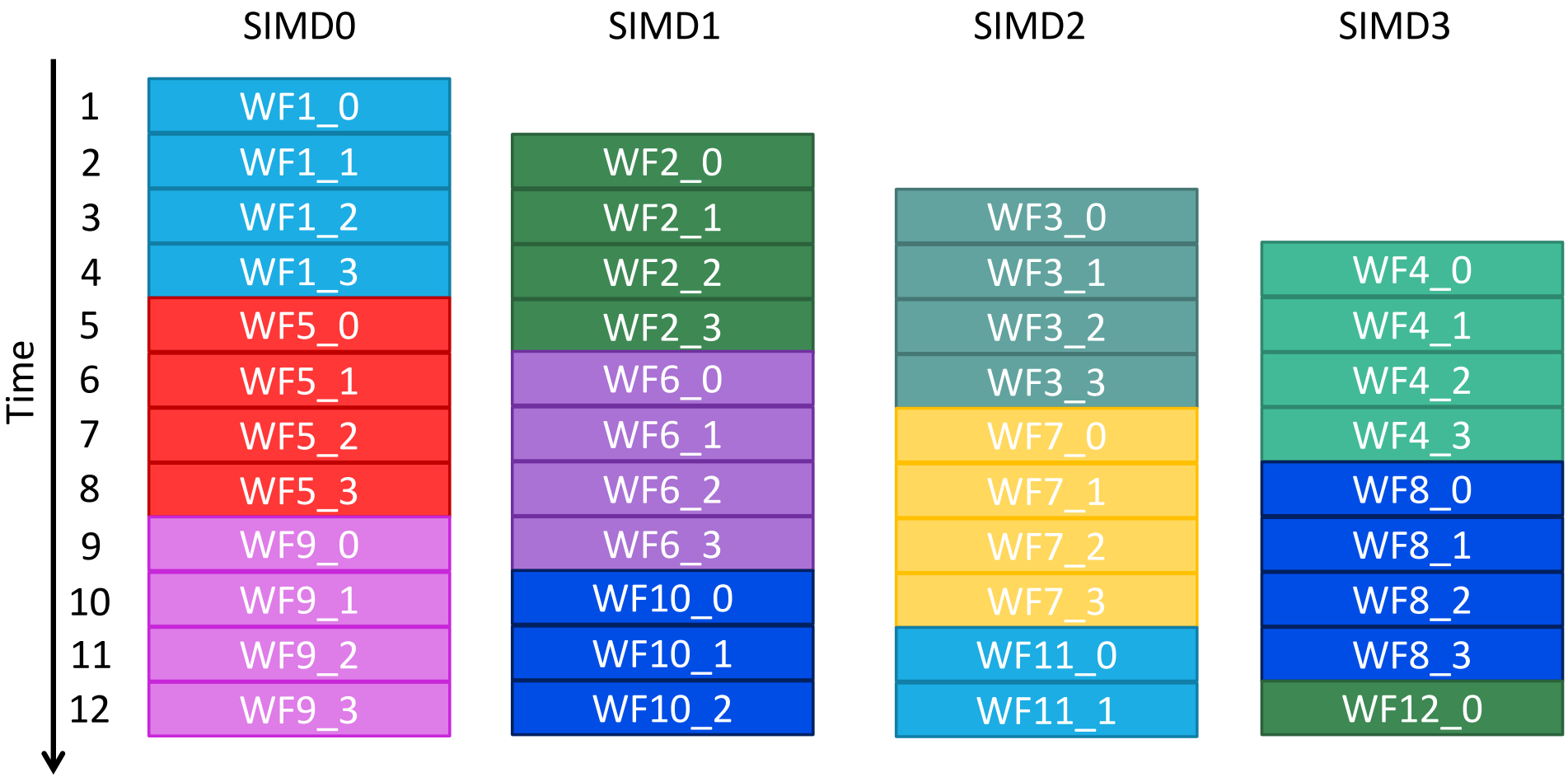


10 Wavefront x 4 SIMD Units =
40 Active Wavefronts / GPU Core

64 work-items / wavefront x 40 active wavefronts =
2560 Active Work-items / GPU Core

GPU Timing Diagram

On average: fetch & commit one  wavefront / cycle



SIMD Unit – A GPU Pipeline

Like a wide CPU pipeline – except one fetch for entire width

16-wide physical ALU

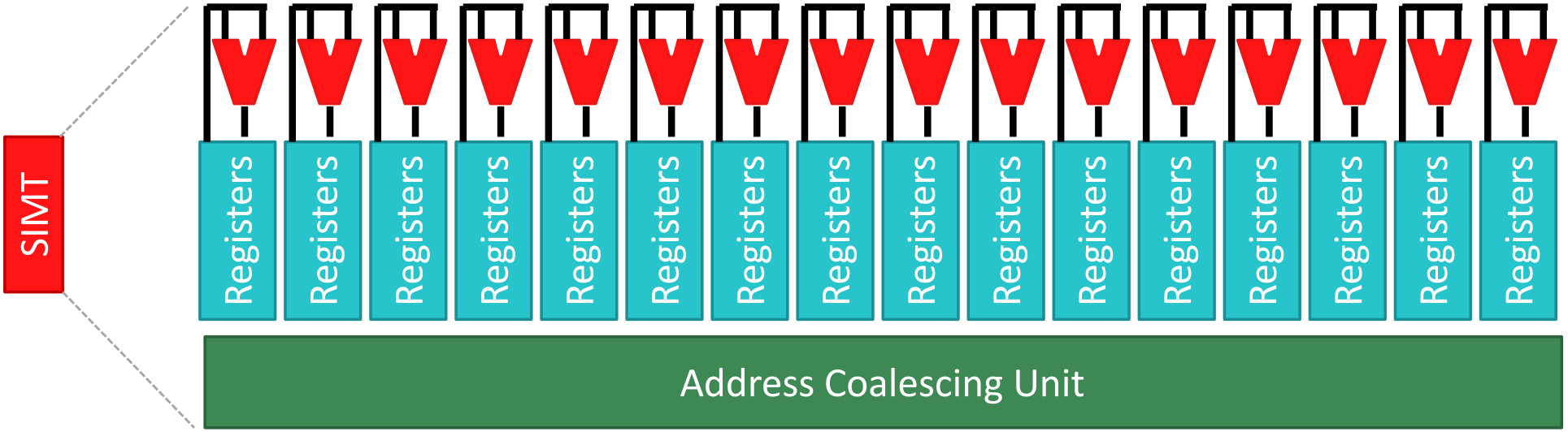
- Executes 64-wavefront over 4 cycles

64KB register state / SIMD Unit

- Compare to x86 (CPU): ~1KB of physical register file state (***~1/64 size***)

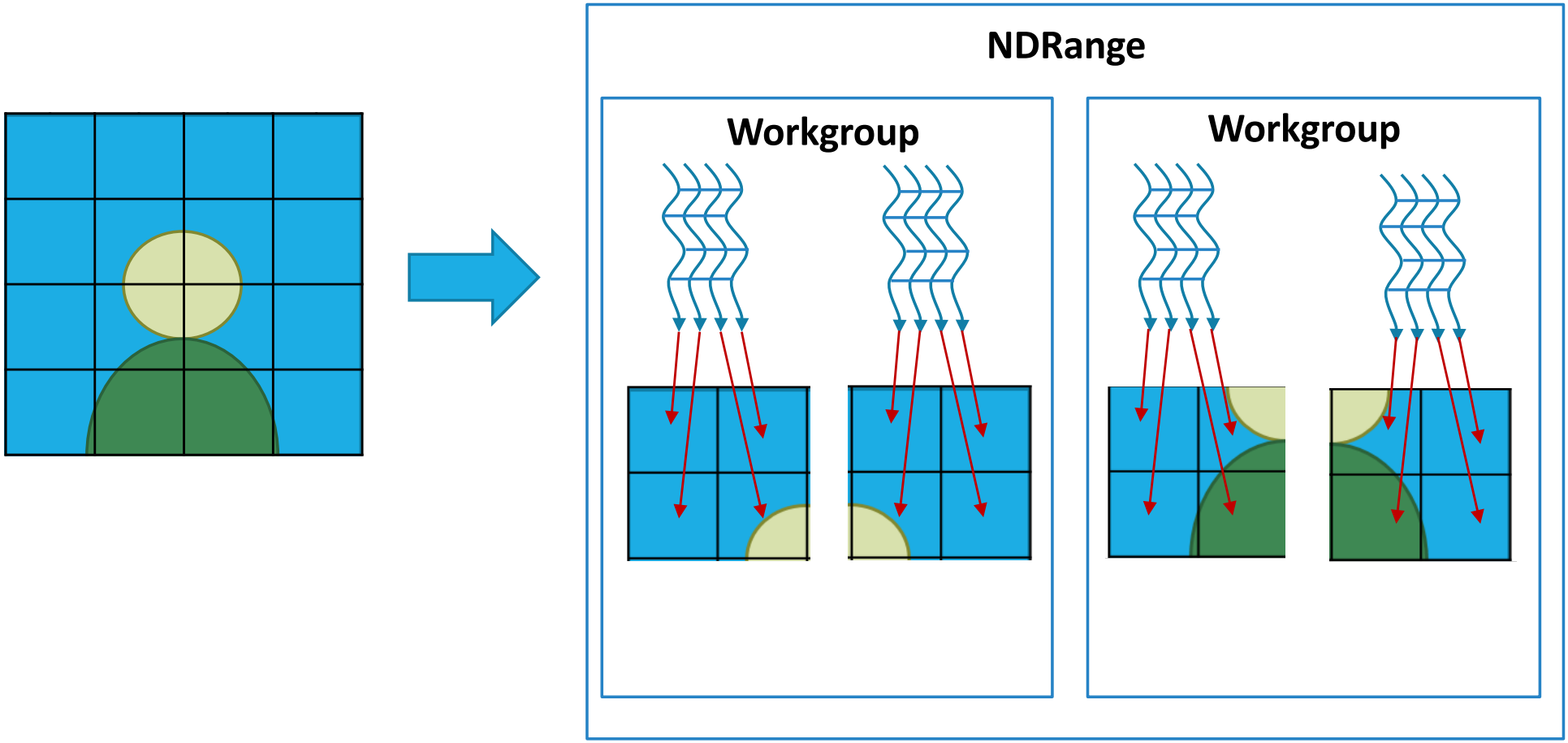
Address Coalescing Unit

- *A key to good memory performance*



Address Coalescing

Wavefront: Issue 64 memory requests



Address Coalescing

Wavefront: Issue 64 memory requests

Common case:

- work-items in same wavefront touch same cache block

Coalescing:

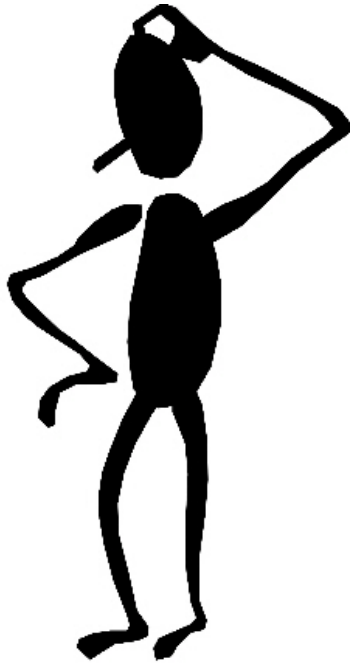
- Merge many work-items requests into single cache block request

Important for performance:

- Reduces bandwidth to DRAM

GPU Memory

GPUs have caches.



Not Your CPU's Cache

By the numbers: **Bulldozer – FX-8170** vs. **GCN – Radeon HD 7970**

	CPU (Bulldozer)	GPU (GCN)
L1 data cache capacity	16KB	16 KB
Active threads (work-items) sharing L1 D Cache	1	2560
L1 dcache capacity / thread	**16KB**	**6.4 bytes**
Last level cache (LLC) capacity	8MB	768KB
Active threads (work-items) sharing LLC	8	81,920
LLC capacity / thread	**1MB**	**9.6 bytes**

GPU Caches

Maximize throughput, not hide latency

- Not there for temporal locality
- It is barely there for spatial locality

L1 Cache: Coalesce requests to same cache block by different work-items

- i.e., streaming thread locality?
- Keep block around just long enough for each work-item to hit once
- Ultimate goal: **Reduce bandwidth to DRAM**

L2 Cache: DRAM staging buffer + some instruction reuse

- Ultimate goal: **Tolerate spikes in DRAM bandwidth**

If there is any spatial/temporal locality:

- Use local memory (scratchpad)

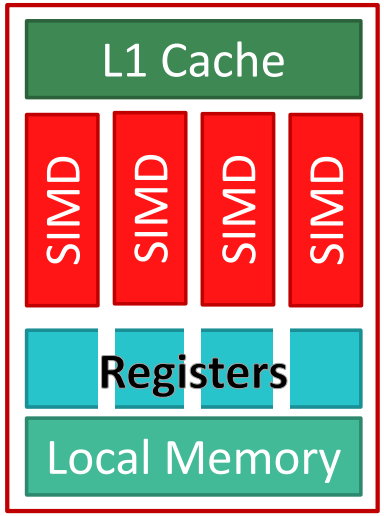
Scratchpad Memory

GPUs have scratchpads (Local Memory)

- Separate address space
- Managed by software:
  - Rename address
  - Manage capacity – manual fill/eviction

Allocated to a workgroup

- i.e., shared by wavefronts in workgroup



Example System: Radeon Fury X

High-end part from last year

64 Compute Units:

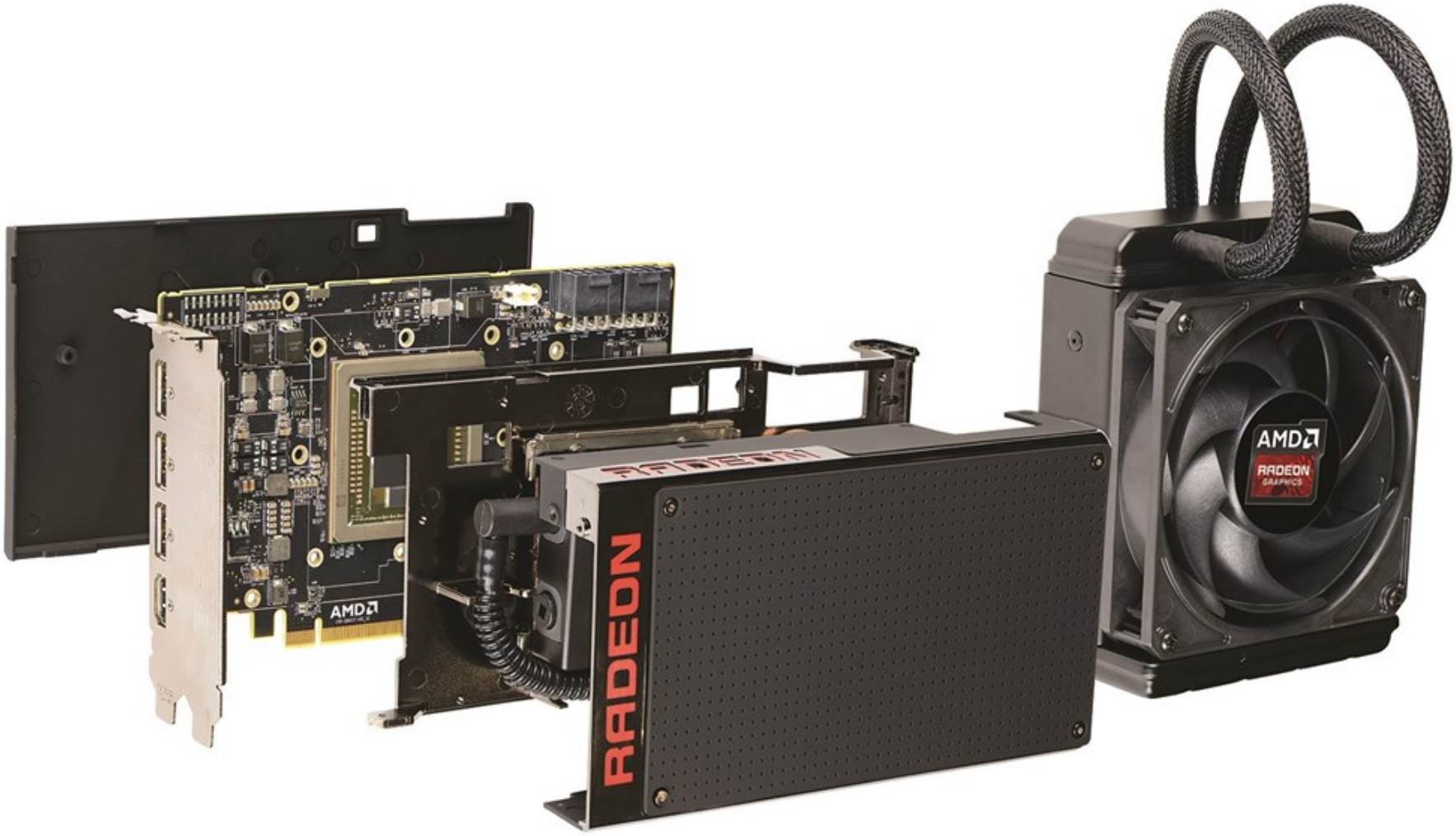
- 163,840 Active work-items
- 64 GPU Cores * 4 SIMT Units * 16 ALUs = 4096 Max FP ops/cycle
- 512 GB/s Max memory bandwidth (4 GB of HBM)

1 GHz engine clock

- 8.6 TFLOPS single precision (accounting trickery: FMA)

275W Max Power (Chip)

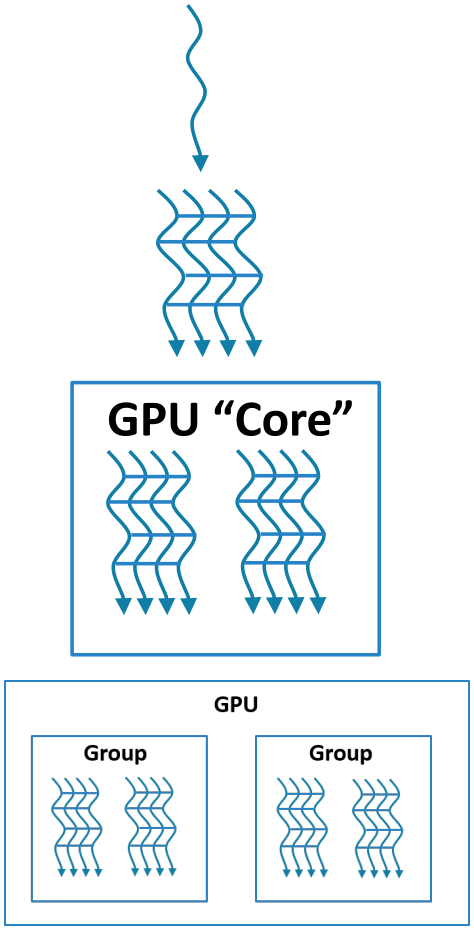
Radeon R9 Fury X - Cooking



A Rose by Any Other Name…

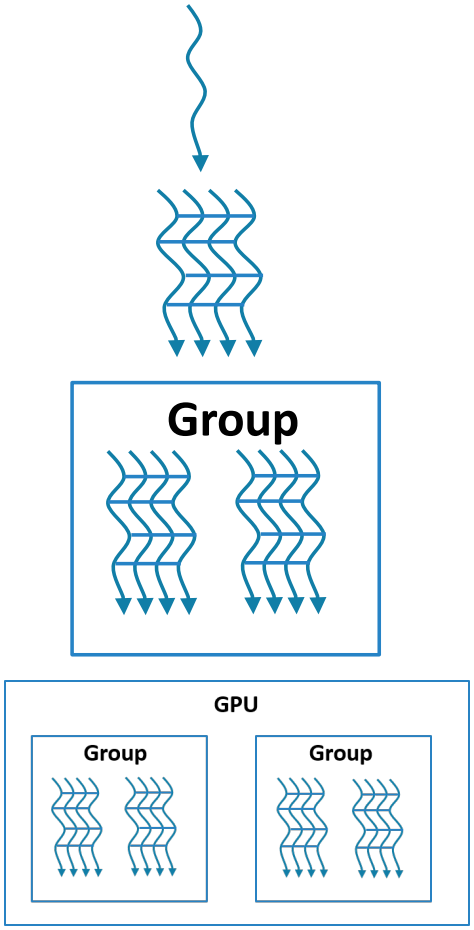
The GPU Decoder Ring

Terminology Headaches #2-5



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

Terminology Headaches #6-9



CUDA/Nvidia	OpenCL/AMD	Henn&Patt
Thread	Work-item	Sequence of SIMD Lane Operations
Warp	Wavefront	Thread of SIMD Instructions
Block	Work-group	Body of vectorized loop
Grid	NDRange	Vectorized loop

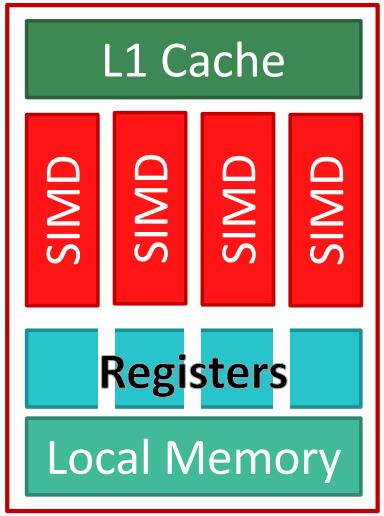
Terminology Headache #10

GPUs have scratchpads (Local Memory)

- Separate address space
- Managed by software:
  - Rename address
  - Manage capacity – manual fill/eviction

Allocated to a workgroup

- i.e., shared by wavefronts in workgroup



Nvidia calls 'Local Memory' or 'Shared Memory'.
AMD calls it 'Group Memory' or 'LDS'.

Recap

Data Parallelism: Identical, Independent work over multiple data inputs

- GPU version: Add streaming access pattern

Data Parallel Execution Models: MIMD, SIMD, SIMT

GPU Execution Model: Multicore Multithreaded SIMT

OpenCL Programming Model

- NDRange over workgroup/wavefront

Modern GPU Microarchitecture: AMD Graphics Core Next (GCN)

- Compute Unit ("GPU Core"): 4 SIMD Units
- SIMD Unit ("GPU Pipeline"): 16-wide ALU pipe (16x4 execution)
- Memory: designed to *stream*

GPUs: Massively multithread. Efficient throughput-oriented design.

Advanced Topics

GPU Limitations, Future of GPGPU

Choose Your Own Adventure!

SIMT Control Flow & Branch Divergence

Memory Divergence

When GPUs talk

- Wavefront communication
- GPU coherence
- GPU consistency

Future of GPUs: What's next?

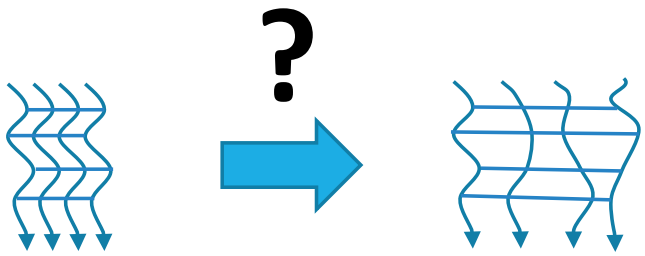
- Task-based programming

SIMT Control Flow

Consider SIMT conditional branch:

- One PC
- Multiple data (i.e., multiple conditions)

```
if (x <= 0)
    y = 0;
else
    y = x;
```

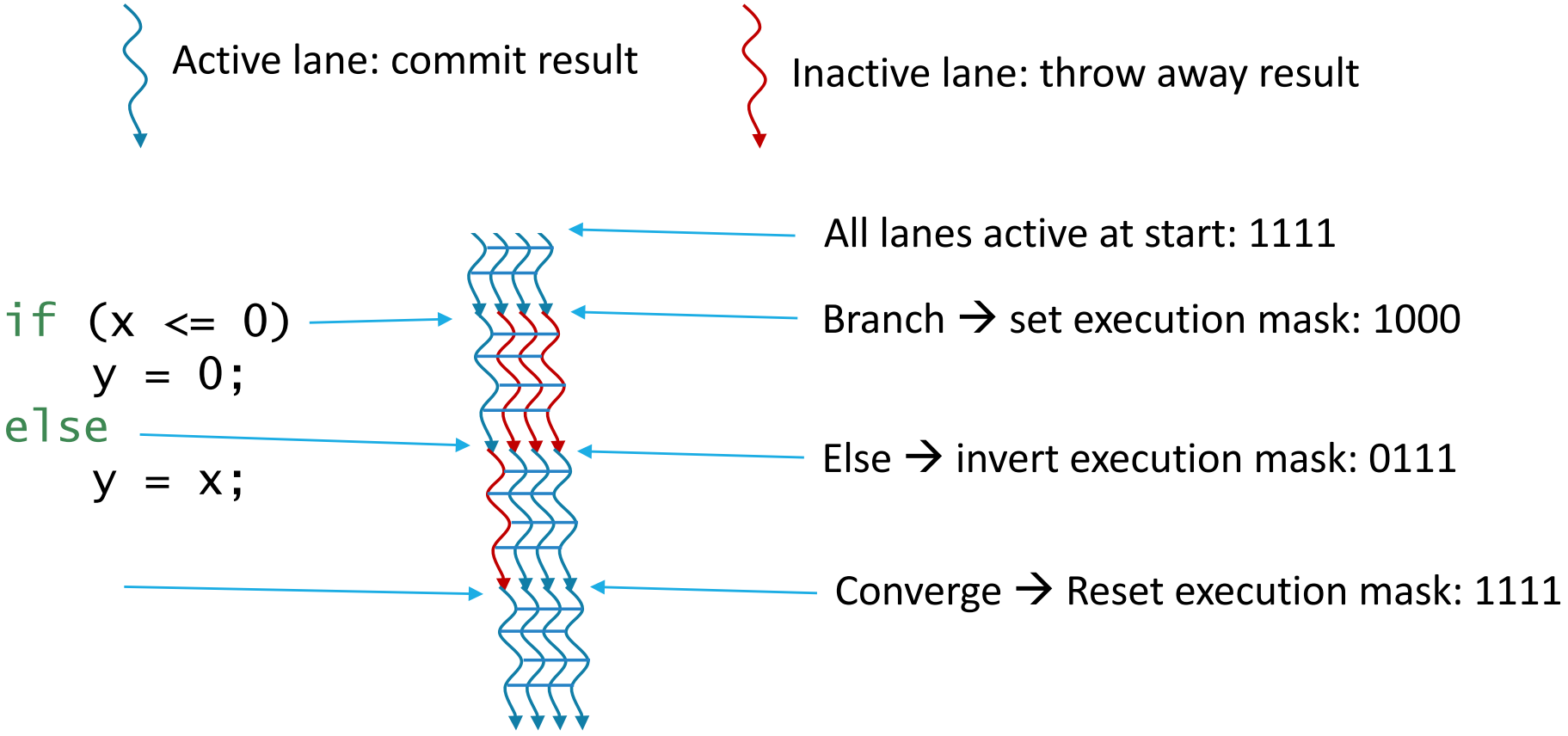


SIMT Control Flow

Work-items in wavefront run in lockstep

- *Don't all have to commit*

Branching through **predication**

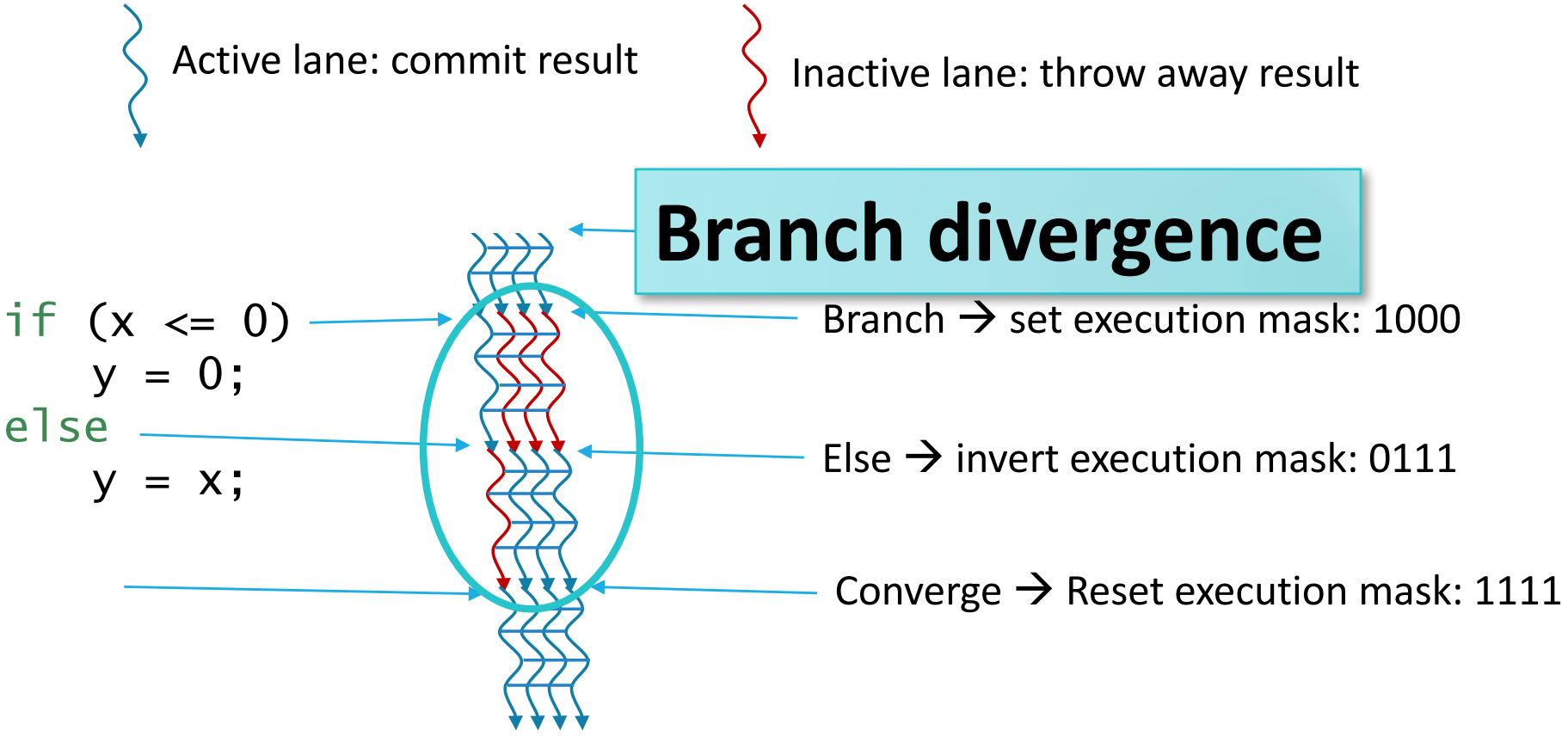


SIMT Control Flow

Work-items in wavefront run in lockstep

- *Don't all have to commit*

Branching through **predication**



Branch Divergence

When control flow *diverges*, ***all lanes take all paths***

Divergence Kills Performance

Beware!

Divergence isn't just a performance problem:

```
__global int lock = 0;

void mutex_lock(…)
{
…
    // acquire lock
    while (test&set(lock, 1) == false) {
        // spin
    }
    return;
}
```

Beware!

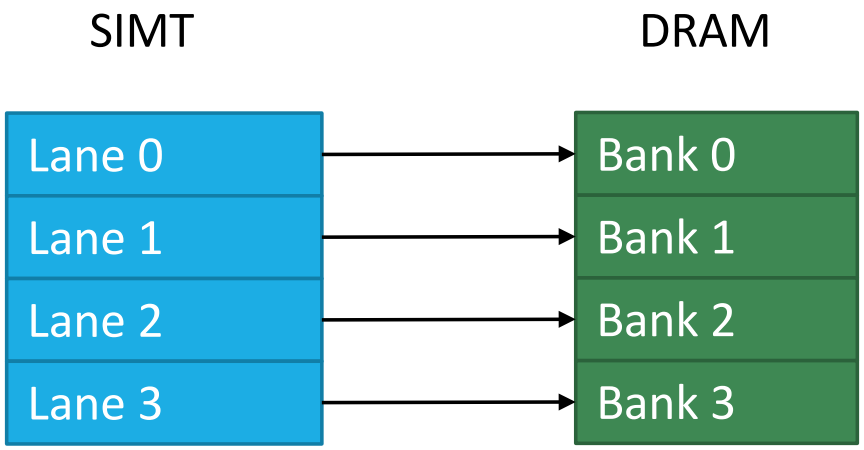
Divergence isn't just a performance problem:

```
__global int lock = 0;

void mutex_lock(…)
{
    // acquire lock
    while (test&set(lock, 1) == false) {
        // spin
    }
    return;
}
```

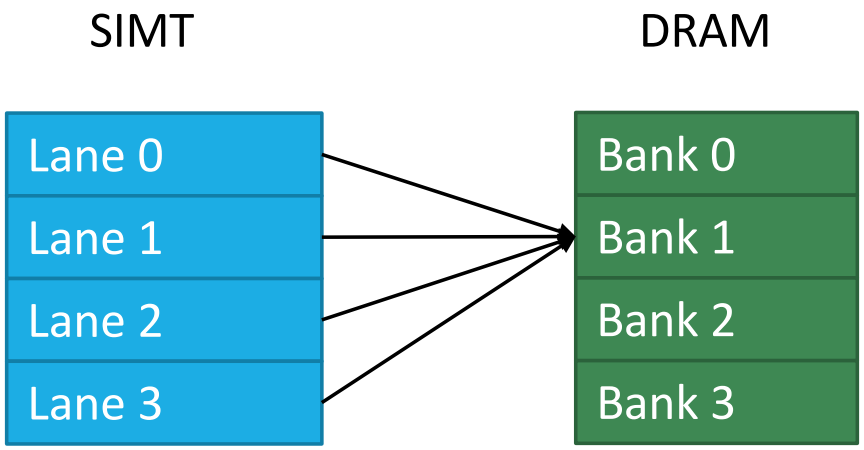
Deadlock: work-items can't enter mutex together!

Memory Bandwidth



✓ -- Parallel Access

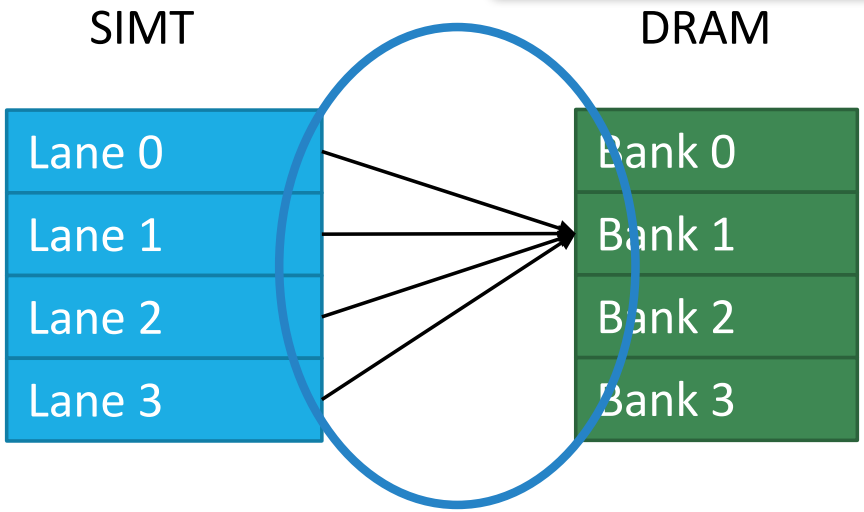
Memory Bandwidth



✘ -- Sequential Access

Memory Bandwidth

Memory divergence



✘ -- Sequential Access

Memory Divergence

One work-item stalls → entire wavefront must stall

- Cause: Bank conflicts, cache misses

Data layout & partitioning is important

Memory Divergence

One work-item stalls → entire wavefront must stall

- Cause: Bank conflicts, cache misses

Data layout & partitioning is important

Divergence Kills Performance

Communication and Synchronization

Work-items can communicate with:

- Work-items in same wavefront
  - No special sync needed…they are lockstep!
- Work-items in different wavefront, same workgroup (local)
  - Local barrier
- Work-items in different wavefront, different workgroup (global)
  - OpenCL 1.x: Nope
  - OpenCL 2.0: Yes, similar synchronization operations to CPUs
  - CUDA x: Incrementally getting better

GPU Consistency Models (circa 2013)

Very weak guarantee:

- Program order respected within single work-item
- All other bets are off

Safety net:

- Fence – "make sure all previous accesses are visible before proceeding"
- Built-in barriers are also fences

A wrench:

- GPU fences are *scoped* – only apply to subset of work-items in system
  - E.g., local barrier

Take-away: confusion abounded

Read Hower et al. ASPLOS 2014

GPU Coherence

Notice: GPU consistency model does not require strong coherence

- i.e., no Single Writer, Multiple Reader invariant
- i.e., no read-for-ownership

Don’t get caught up in the historical CPU definition of coherence

GPU coherence implementations:

- Nvidia: disable private caches
- AMD: flush/invalidate cache at fences

Read Hechtman et al. HPCA 2014

GPU Architecture Research

The future is GPU compute

- Fundamentally more efficient than CPUs
- Simplifying GPU programmability is the challenge

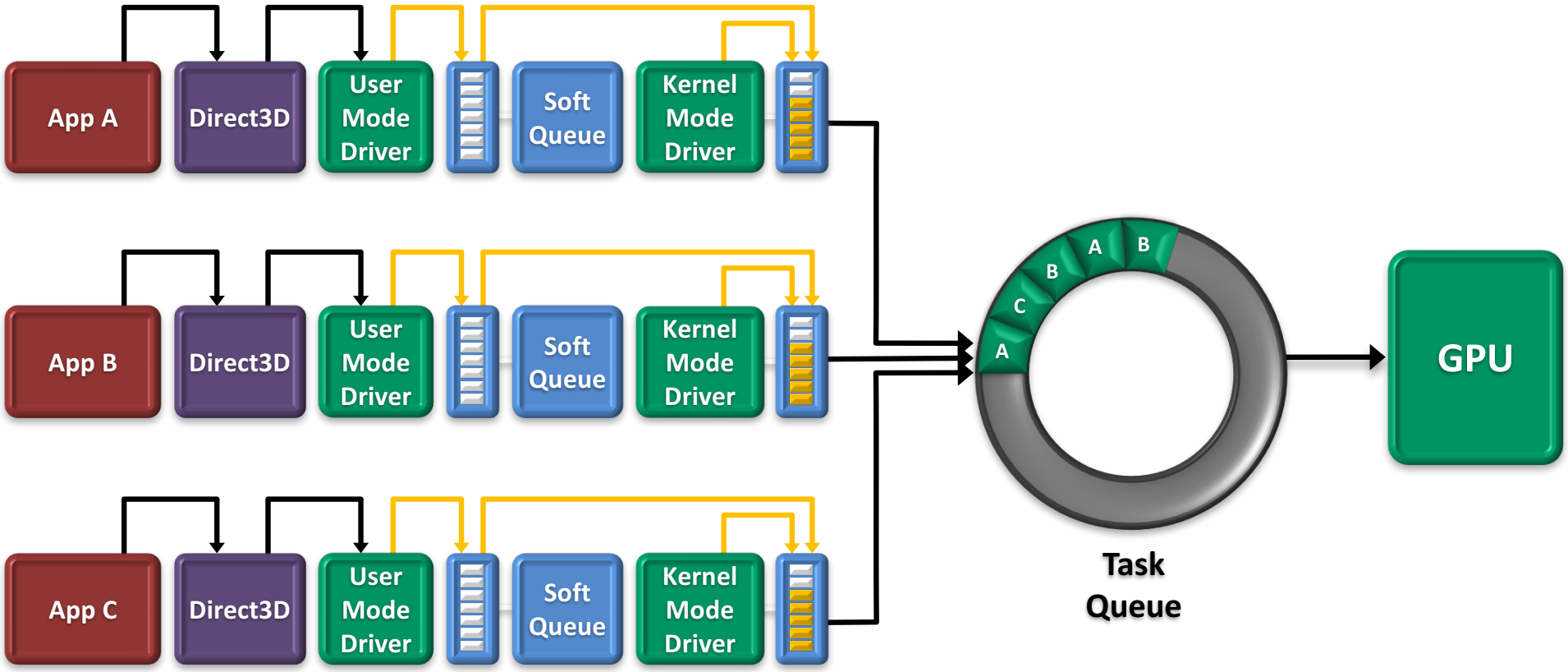
Blending with CPU architecture:

- Dynamic scheduling / dynamic wavefront re-org
- Work-items have more locality than we think

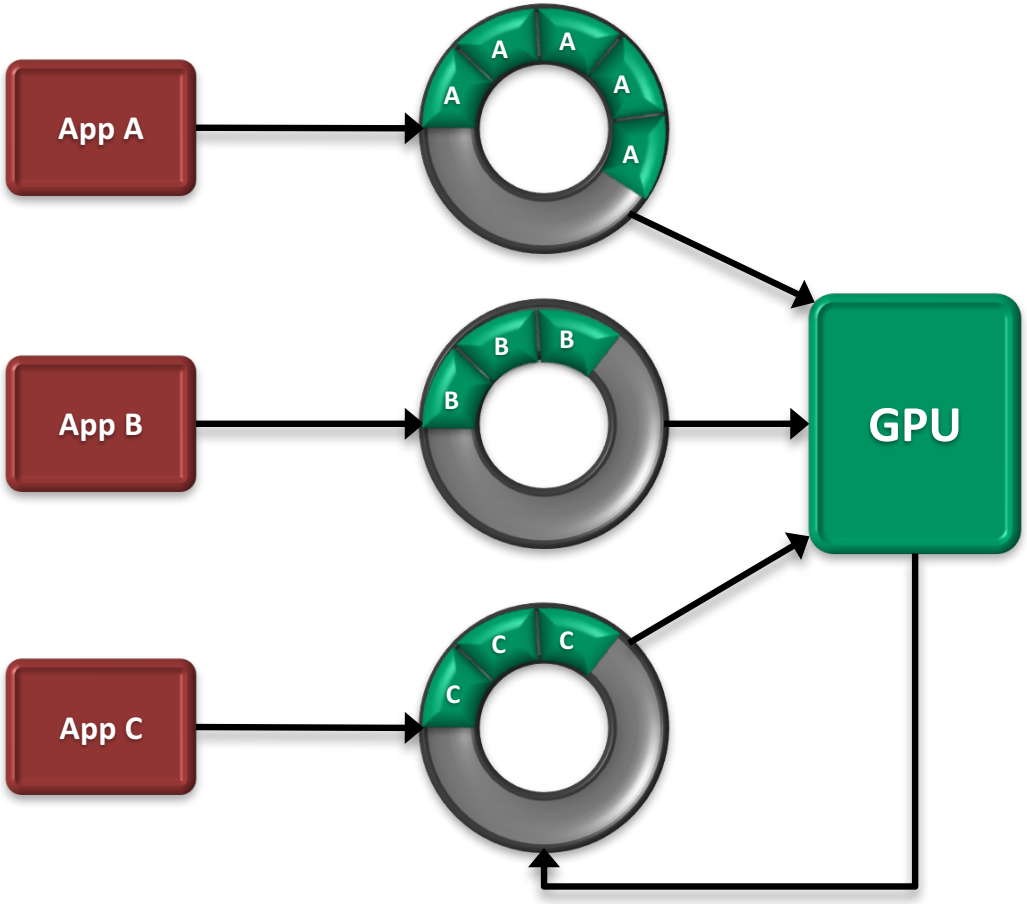
Tighter integration with CPU on SOC:

- Fast kernel launch
  - Exploit fine-grained parallel region: Remember Amdahl's law
- Common shared memory

Traditional Command and Dispatch Flow



hQ Command and Dispatch Flow



User-mode application talks directly to the hardware

- HSA Architected Queuing Language (AQL) defines vendor-independent format
- No system call
- No kernel driver involvement

Hardware scheduling

Greatly reduced dispatch overhead
→ less overhead to amortize
→ profitable to offload smaller tasks

GPU kernels can self-enqueue additional tasks for dynamic parallelism