# CS/ECE 752 Advanced Computer Architecture I

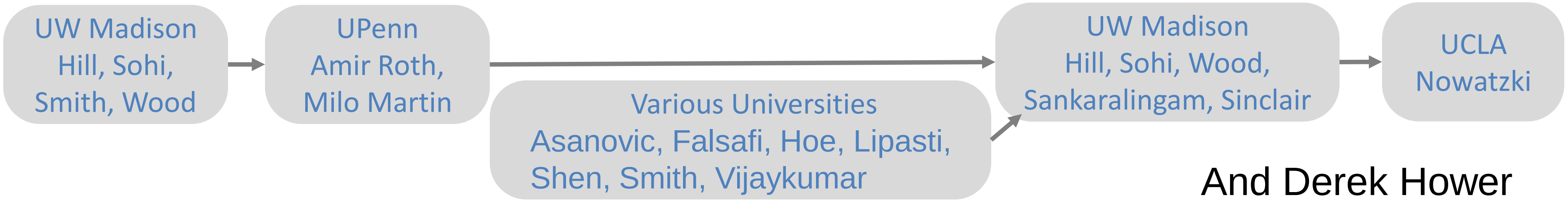
AN x64 PROCESSOR IS SCREAMING ALONG AT BILLIONS OF CYCLES PER SECOND TO RUN THE XNU KERNEL, WHICH IS FRANTICALLY WORKING THROUGH ALL THE POSIX-SPECIFIED ABSTRACTION TO CREATE THE DARWIN SYSTEM UNDERLYING OS X, WHICH IN TURN IS STRAINING ITSELF TO RUN FIREFOX AND ITS GECKO RENDERER, WHICH CREATES A FLASH OBJECT WHICH RENDERS DOZENS OF VIDEO FRAMES EVERY SECOND
BECAUSE I WANTED TO SEE A CAT JUMP INTO A BOX AND FALL OVER.
I AM A GOD.

Source: XKCD

Prof. Matthew D. Sinclair

## Vectors, Data-Level Parallelism, GPUs

Slide History/Attribution Diagram:

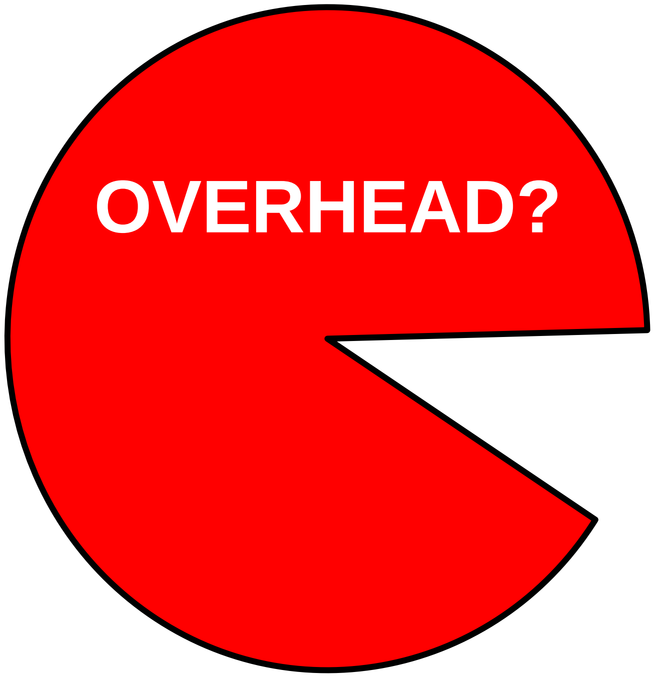


And Derek Hower

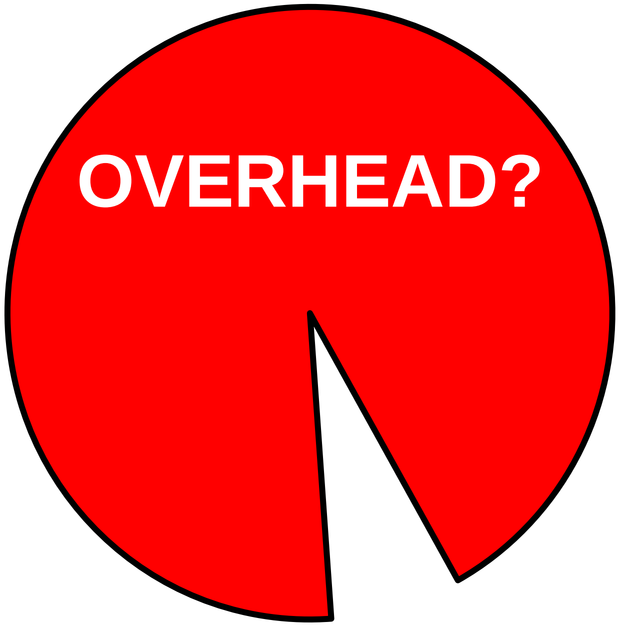
# Superscalar + Multicore

- **Are we done yet?**
- Consider two OOO processors:

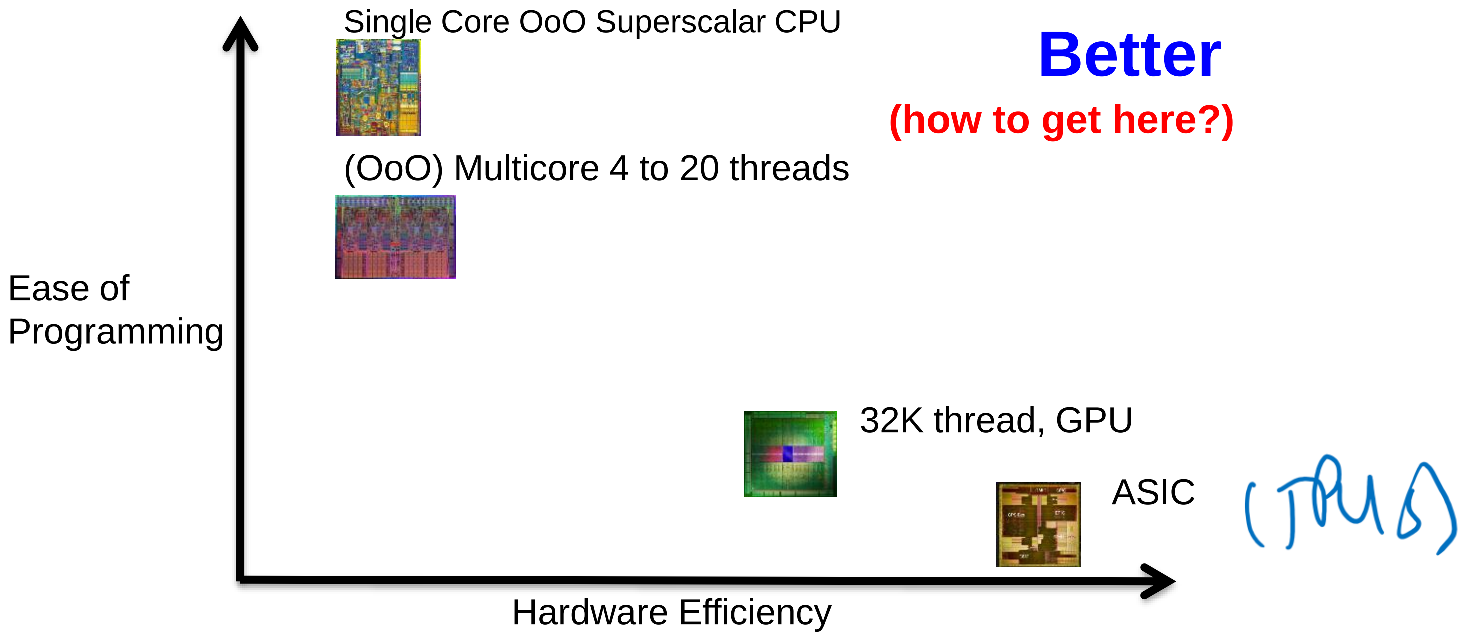
IBM Power 4
Power Breakdown



ARM Cortex A15
Power Breakdown
(only 3-way superscalar)



## Fundamental tradeoff: Programmability vs. Efficiency



**Less Power + Multicore -> More Performance**

# How to Compute This Fast?

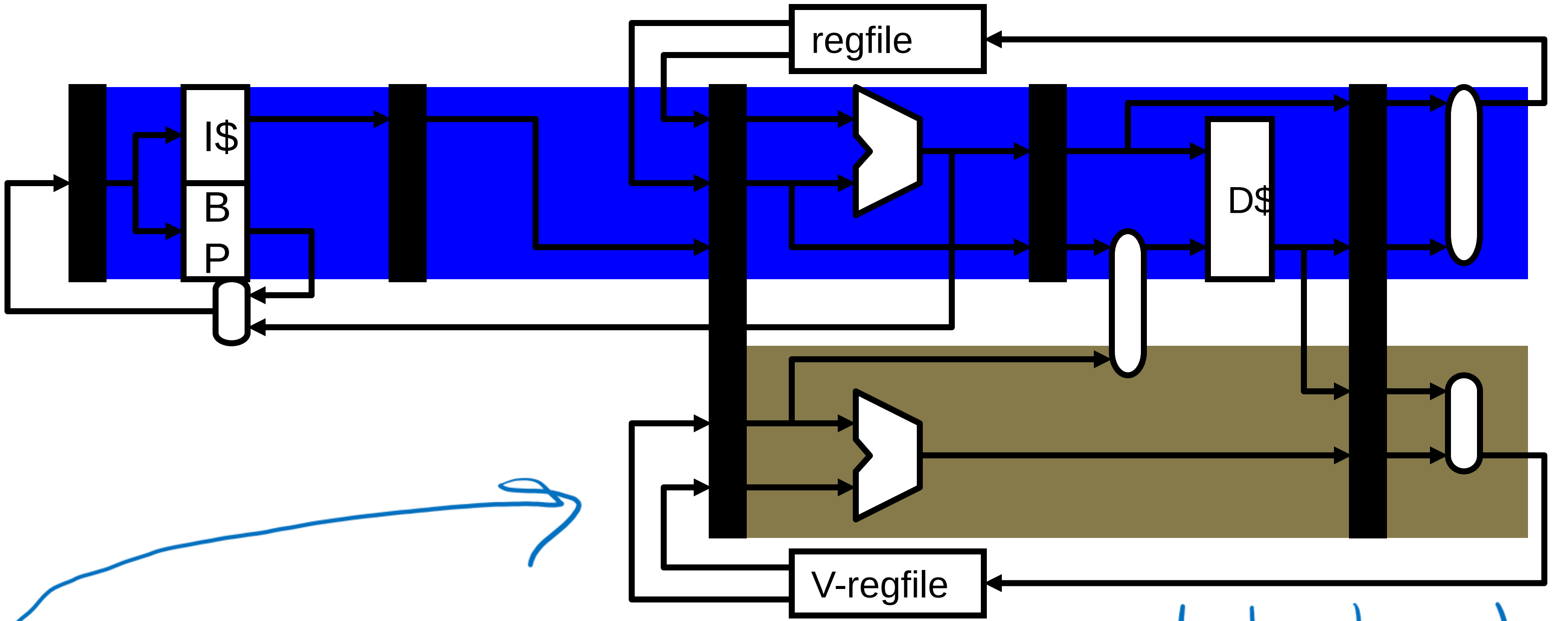
- Performing the **same** operations on **many** data items
  - Example: SAXPY

```
for (I = 0; I < 1024; I++) {
  Z[I] = A*X[I] + Y[I];
}
```

```
L1: ldf [X+r1]->f1     // I is in r1
    mulf f0,f1->f2     // A is in f0
    ldf [Y+r1]->f3
    addf f2,f3->f4
    stf f4->[Z+r1}
    addi r1,4->r1    ] branch (loop)
    blti r1,4096,L1    logic
```

- Instruction-level parallelism (ILP) - fine grained
  - Loop unrolling with static scheduling –or– dynamic scheduling
  - Wide-issue superscalar (non-)scaling limits benefits
- Thread-level parallelism (TLP) - coarse grained
  - Multicore
- Can we do some "medium grained" parallelism?

# Exploiting DLP With Vectors



data level parallelism

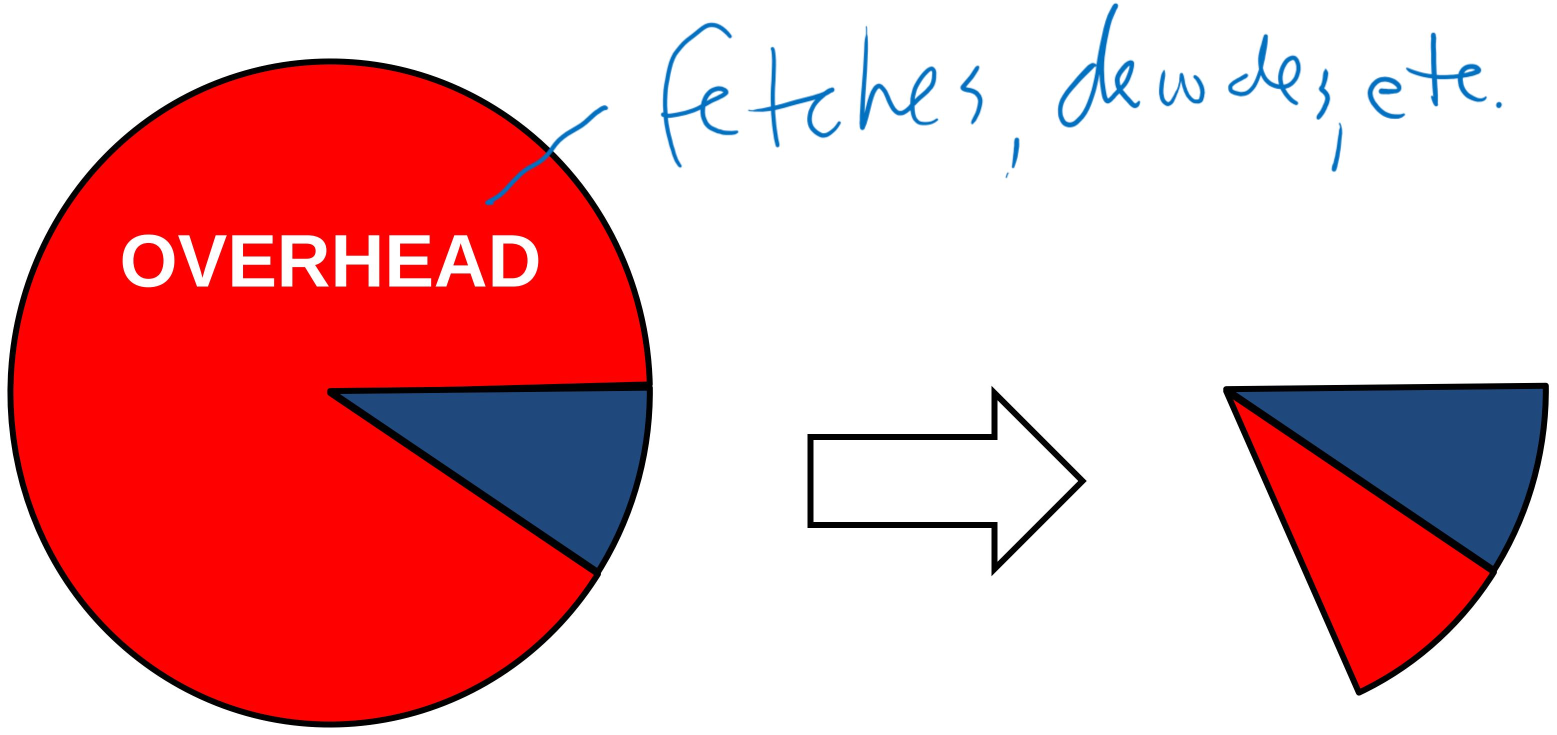
- One way to exploit DLP: **vectors**
  - Extend processor with **vector "data type"**
  - Vector: array of MVL Numbers
    - **Maximum vector length (MVL)**: anywhere between 2–64
    - ARM SVE, RISC-V RVV, X86 AVX roughly like this (backup slides)
  - **Vector register file**: 8–16 vector registers (**`v0–v15`**)

# Data-Level Parallelism

- **Data-level parallelism (DLP)**
  - Single operation repeated on multiple data elements
    - SIMD (**S**ingle-**I**nstruction, **M**ultiple-**D**ata)
  - Less general than ILP: parallel insns are all same operation
  - Exploit with **vectors**
- Old idea: Cray-1 supercomputer from late 1970s
  - Eight 64-entry x 64-bit floating point "vector registers"
    - 4096 bits (0.5KB) in each register! 4KB for vector register file
  - Special vector instructions to perform vector operations
    - Load vector, store vector (wide memory operation)
    - Vector+Vector or Vector+Scalar
      - addition, subtraction, multiply, etc.
    - In Cray-1, each instruction specifies 64 operations!
  - ALUs were expensive, so one operation per cycle (not parallel)

## Why so useful?

- A slight oversimplification:



- Not quite so simple for memory side…

## Why accelerators?

- Hardware acceleration is everywhere
  - Specialized chips for video/image encode/decoding
  - Machine learning specific accelerators (including autonomous agents)
  - Network accelerators
  - Cryptography
  - Bitcoin mining
  - Genomics
  - Database accelerators
  - …

You can design efficient hardware for lots of specific problems

## What is a programmable accelerator?

- However, these are not generally **programmable**!
- Many custom accelerators have knobs, configuration registers, etc. … that allow you to “program” them.
- **But you cannot run arbitrary code on them**
  - **They are not Turing Complete**

Today’s focus: GPUs that can execute (mostly) arbitrary code

## The rise of accelerators

- On the general-purpose front
  - CMOS compute frequency has reached its limits
  - ILP is mostly mined out
    - Branch predictors, caches, memory dependency prediction have done great things
    - However, these are energy-hungry operations
- For the time being, we are still getting more transistors
  - NVIDIA Volta V100: **21B transistors, 120 TFLOPS, 900 GB/s Memory BW**
  - Ampere, Hopper, etc. even larger
- Many important workloads are highly parallel
  - **Machine learning is one very popular example**

The future is in acceleration.

# General Purpose Graphics Processing Unit

Slide History/Attribution Diagram:

UW Madison Hill, Sohi, Smith, Wood
UPenn Amir Roth, Milo Martin
Various Universities Asanovic, Falsafi, Hoe, Lipasti, Shen, Smith, Vijaykumar
UW Madison Hill, Sohi, Wood, Sankaralingam, Sinclair
UCLA Nowatzki

## Announcements 11/12/24

- Canvas should be back to normal – let me know ASAP if this is not the case for you
- Midterm next Wednesday (in class)
  - Focuses on everything since Midterm 1
  - Monday: exam review, bring questions!
  - Posted: old exams, problems
- HW7 grading ongoing
  - Will post solutions soon to help with exam prep
- With some high likelihood, today will be "GPU only"
  - If so Chiplets + TPU Friday

## In-Class Assignment

- With a partner, answer the following questions:
  - Why have GPUs succeeded where other accelerators previously failed?
  - Do TensorCores have to be tightly coupled to the GPU pipeline?

- In 3 minutes we'll discuss as a class

## In-Class Assignment

- With a partner, answer the following questions:
  - Why have GPUs succeeded where other accelerators previously failed?

⇒ Fairly general, massive parallelism
⇒ killer app (*) (graphics)
⇒ business / luck / good timing
⇒ SW ecosystem (CUDA) eased adoption

  - Do TensorCores have to be tightly coupled to the GPU pipeline?

No - newer GPUs have asynch support
↳ tradeoff: tighter integr (x86, ARM, etc. vectors) can reduce overhead but ↑ contention

## GPU Agenda

- Why do we have GPUs?
- What is a GP GPU?
  - Hardware
  - Execution Model
  - Software Interface
- Sources of GPU Efficiency/Inefficiency
- **Very** Simple Programming Example

- Why still so much room to improve GPGPUs?

## What to Build?

- Key Properties:
    - Embarrassing Data Parallelism
        - Data structures are extremely regular
        - Very few dependences
    - Low Locality (or at least this is the idea)
- So maybe…
    - **SIMD** for low-overhead Data Parallelism
    - **Multithreading** to hide latency
    - **Multicore** (@ modest frequency) to scale up
- But SIMD is hard to program/compile for…
    - Wish we could have **flexibility of threading**, with the efficiency of SIMD

# Sun Niagara (Precursor to [Feehrer`13])

- Simple in-order pipeline
- Multithread to hide memory latency
- Multicore for parallelism
- **So can we just add SIMD?**

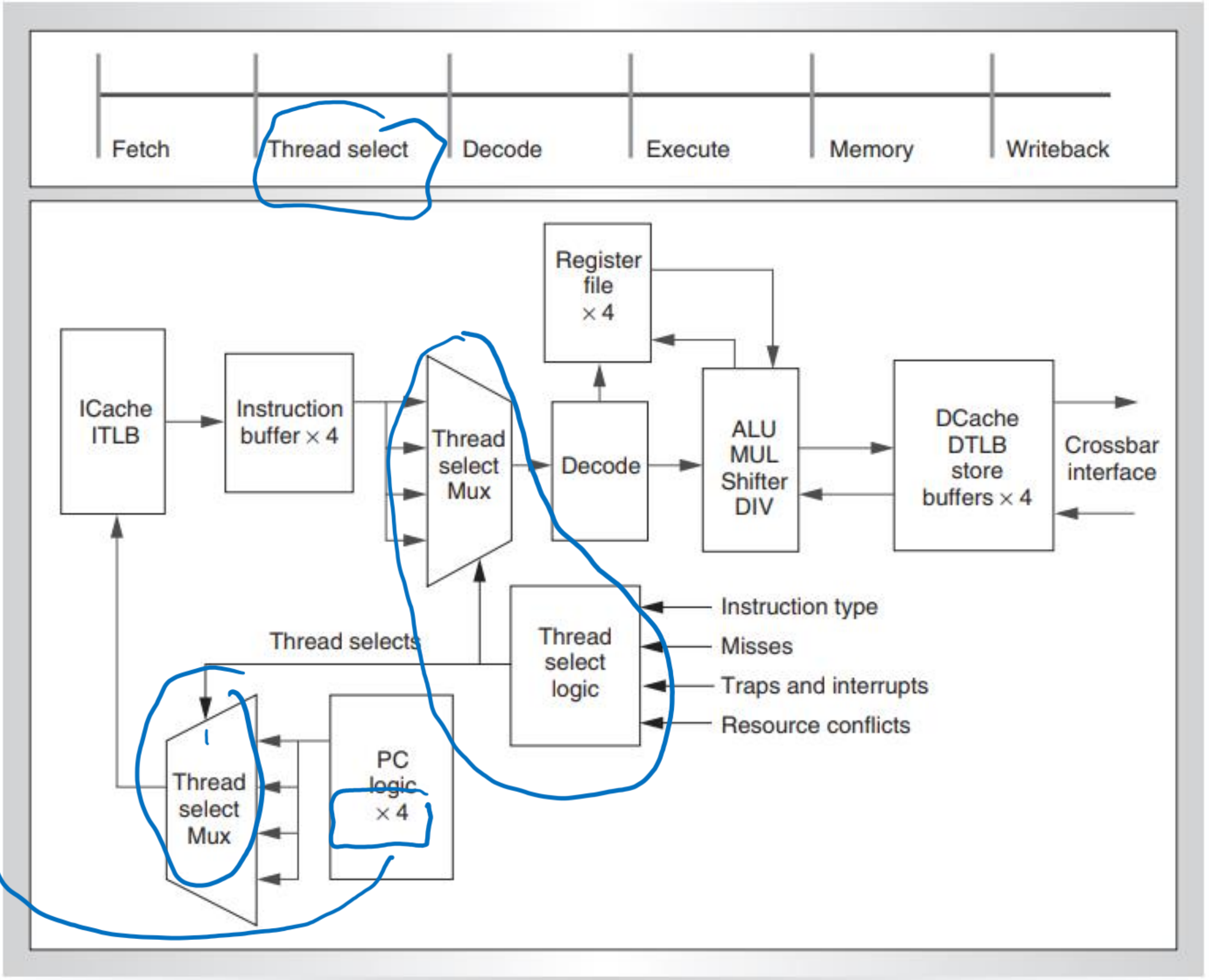
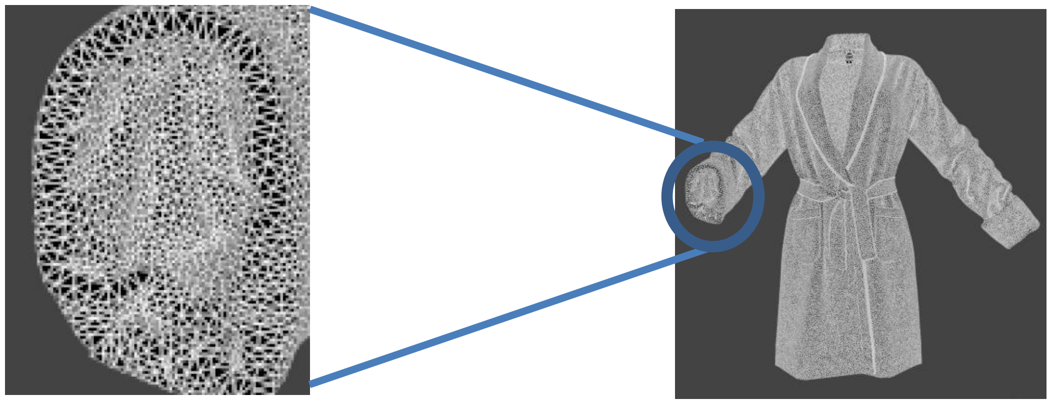


Figure 3. Sparc pipeline block diagram. Four threads share a six-stage single-issue pipeline with local instruction and data caches. Communication with the rest of the machine occurs through the crossbar interface.

# What **was** a GPU?

- GPU = Graphics Processing Unit
  - Accelerator for raster-based graphics (OpenGL, DirectX, Vulkan)
  - Highly programmable
  - Commodity hardware
  - 100's of ALUs; 10000's of concurrent threads

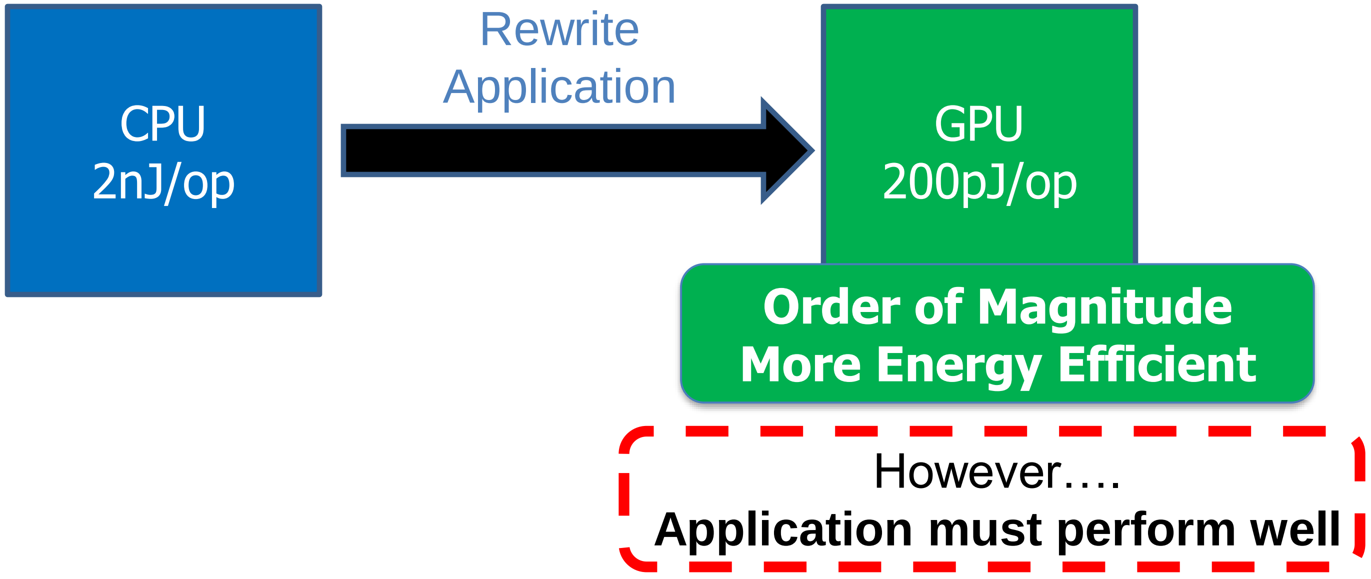


**Highly Parallel Operation**

**Requires Significant Memory Bandwidth**

## Why use a GPU for computing?

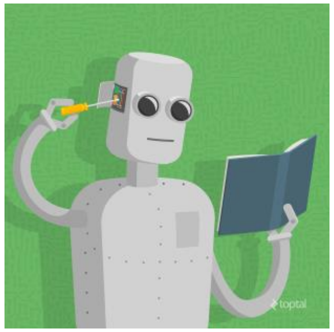
- GPU uses larger fraction of silicon for computation than CPU.
- At peak performance GPU uses order of magnitude less energy per operation than CPU.


CPU
2nJ/op
Rewrite Application
GPU
200pJ/op
Order of Magnitude More Energy Efficient
However….
Application must perform well

Amdahl's Law still applies

## Evolution of GPUs (Today GPUs are Ubiquitous)

- From being used for graphics
- To having a new killer application: Machine Learning
- … and crypto



Today the name GPU is not really meaningful.
Reality: highly parallel, highly programmable vector supercomputers.

## Exploring Parallelism

Many different definitions/types of parallelism

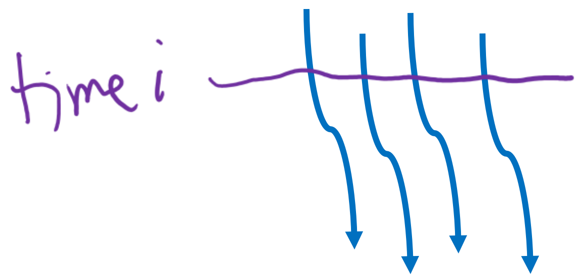
- Instruction Level Parallelism
  - Different machine instructions in the same thread can execute in parallel
- Task Level Parallelism
  - Higher level tasks can run concurrently
- Bit level Parallelism
  - In VHDL exploit the ability to do level bit-level computation in parallel (i.e. longer words, carry-lookahead adders)
- Data Level Parallelism
  - Identical computation just on different data
  - Single Instruction Multiple Data (SIMD) instructions exploit data parallelism
  - Single Program Multiple Data (SPMD) applications exploit data parallelism

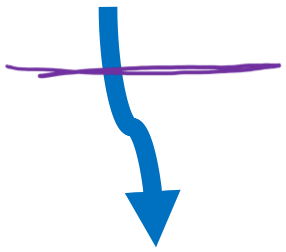
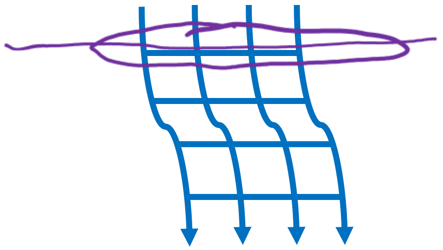
GPUs are designed to exploit DLP

## Why Data Parallelism?

- Easy to build efficient hardware to capture it
- The **regularity** in the computation can be exploited to reduce control hardware and make effective use of memory bandwidth

# Execution Model Comparison



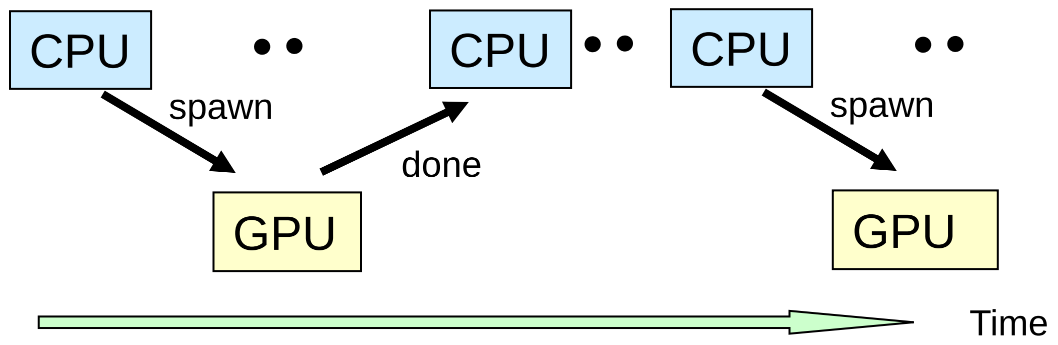
|  | MIMD/SPMD | SIMD/Vector | SIMT |
|---|---|---|---|
| **Example** | Multicore CPUs | x86 SSE/AVX<br>ARM SVE<br>RISC-V RVV | GPUs |
| **Pros** | More general: better support for TLP | Able to mix serial and parallel code | Easier to program, Scatter & Gather operations |
| **Cons** | Inefficient for data parallelism | Gather/Scatter implementations more complicated | Divergence kills performance |

## GPUs & Memory

- GPUs optimized for streaming computations (traditionally)
  - Thus, we have a lot of streaming memory accesses
- DRAM: 100's of GPU cycles per memory access
  - How to hide this overhead & keep the GPU busy in the meantime?
- Traditional CPU approaches:
  - Caches → Need spatial/temporal locality X
    - Streaming applications have little reuse
  - OOO/Dynamic Scheduling → Need ILP X
    - Too power hungry, diminishing returns for GPU applications
  - Multicore/Multithreading/SMT → need independent threads ✓

# GPGPU Programming Model

- CPU “Off-load” parallel kernels to GPU



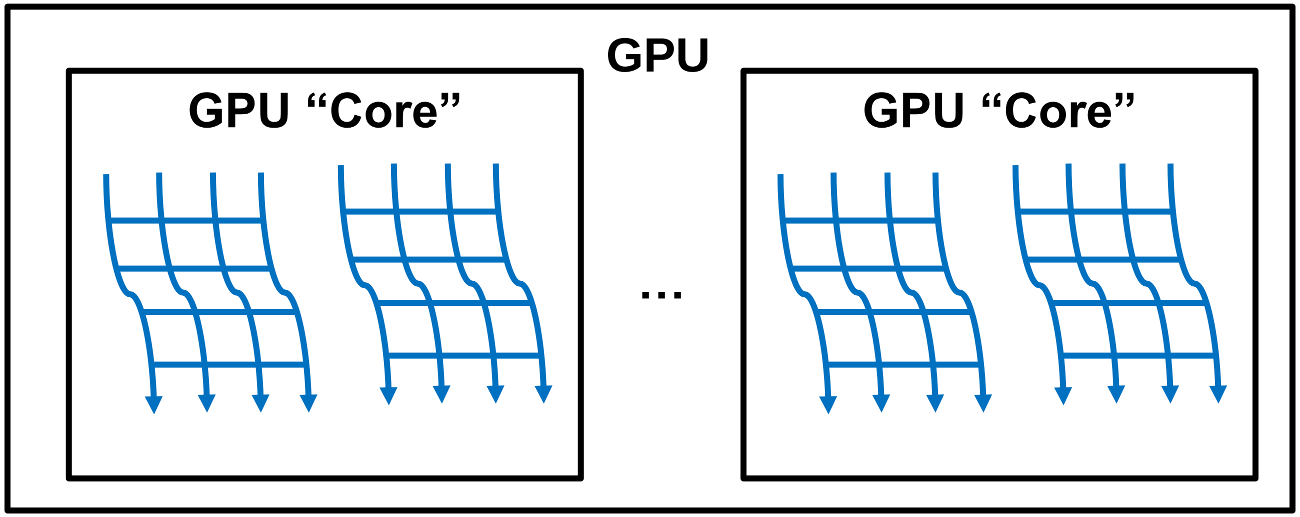
- Transfer data to GPU memory
- GPU HW spawns threads
- Need to transfer result data back to CPU main memory

# Programming GPUs (**CS/ECE/ME 759**)

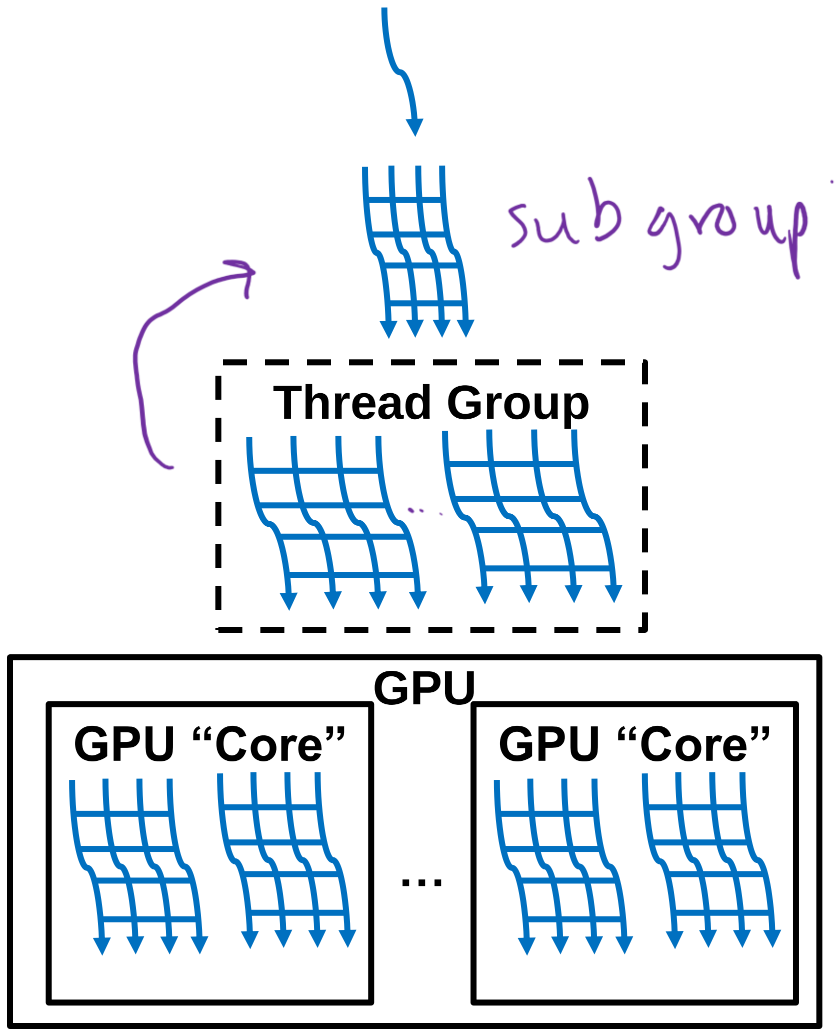
- Program it with CUDA, HIP, or OpenCL
  - CUDA = Compute Unified Device Architecture
    - NVIDIA’s proprietary solution
  - OpenCL = Open Computing Language
    - Open, industrywide standard
  - HIP = Heterogeneous interface for portability
    - AMD’s open solution, its successor to OpenCL
  - Extensions to C
  - Perform a “shader task” (a snippet of scalar computation) over many elements
  - Internally, GPU uses scatter/gather and vector mask operations
- Other solutions:
  - C++ AMP (Microsoft), OpenACC (extension to OpenMP)

# Multicore/Multithreading/SMT on GPUs

- Group SIMT “threads” together on a GPU “core”
- SIMT threads are grouped together for efficiency
  - Loose analogy: SIMT thread group ≈ one CPU SMT thread
  - Difference: GPU threads are **exposed** to the programmer
- Execute different SIMT thread groups simultaneously
  - On a single GPU “core” per-cycle SIMT thread groups swaps
  - Execute different SIMT thread groups on different GPU “cores”



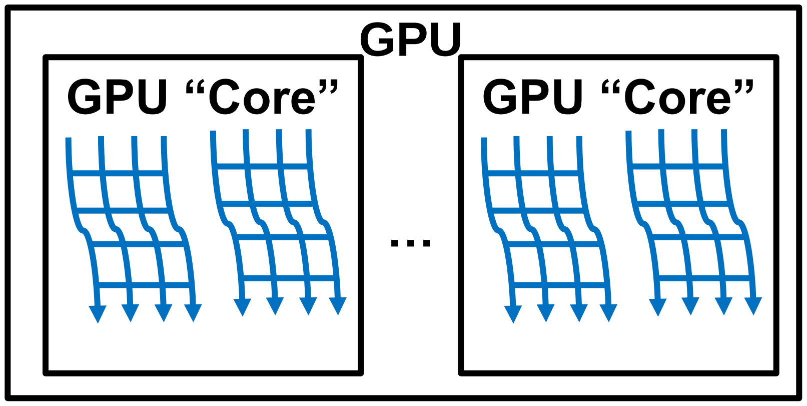
# GPU Component Names

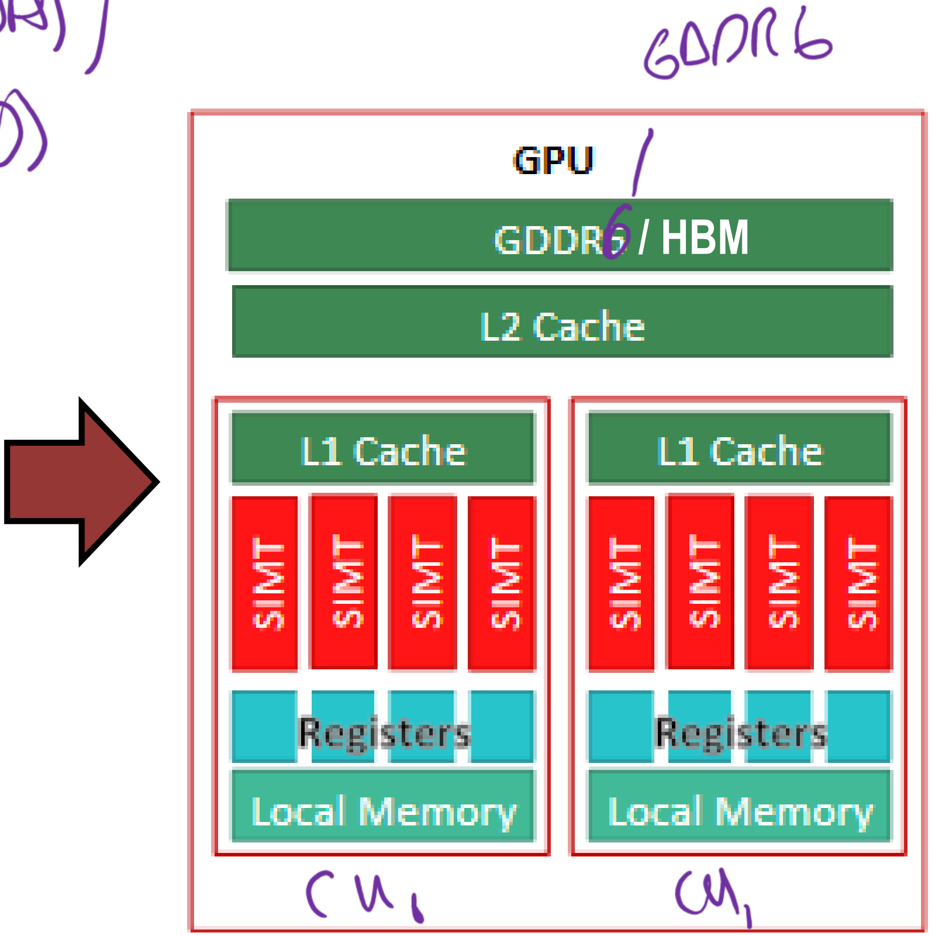


| CUDA/HIP | OpenCL |
|---|---|
| Thread | Work-item |
| Warp / wavefront | Wavefront |
| Thread Block/CTA<br>WG | Workgroup |
| Grid (Kernel) | NDRange (Kernel) |

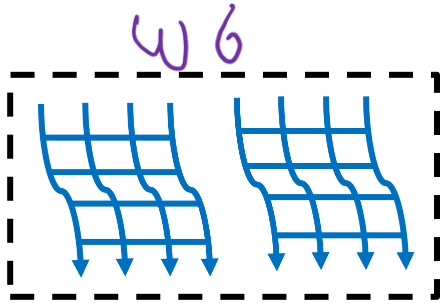
# GPU Hardware Overview

"core" = SM (CUDA) / CU (AMD)

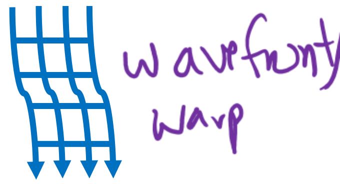




# Compute Unit (CU) – The GPU "Core"

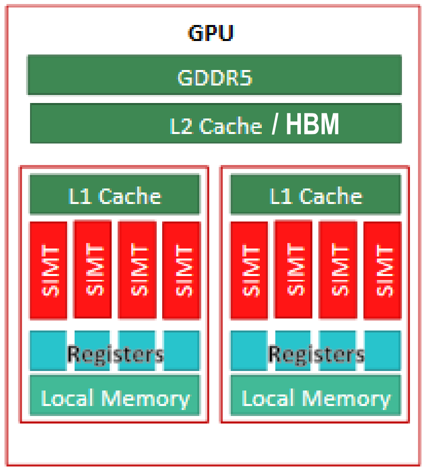


- Job: run thread blocks/workgroups
    - Contains multiple SIMT units (4 in picture below)
    - Each cycle, schedule one SIMT unit



- SIMT unit: runs wavefronts/warps
    - Run the threads
    - AMD: size N (e.g., 10) wavefront instruction buffer
        - 4 cycles to execute one wavefront
        - Average: fetch and commit 1 wavefront/cycle

throughput not latency



## Address Coalescing

warp – wavefront

- 32-64 memory requests issued per memory instruction

- Common case:
  - All threads in warp/wavefront access same cache block(s)
  - If not: **divergence**

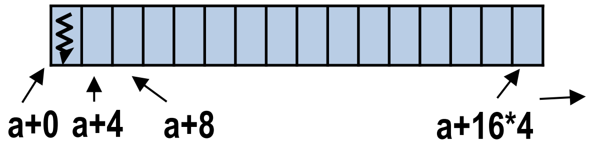
- Coalescing:
  - Merge many thread's requests into a single cache block request
  - Reduces number of in-flight memory requests
  - Helpful for reducing bandwidth to DRAM
  - **Very important for performance – let's see an example**

# Memory Accesses

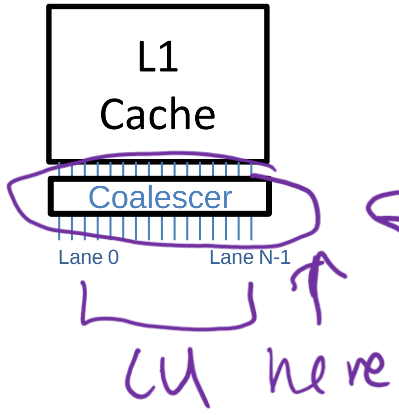
tid = thread id

- Pseudo code for contiguous access (coalesced):

```
gpu void add(int *a, int *b, int *c) {
        c[tid] = a[tid] + b[tid];
}
```

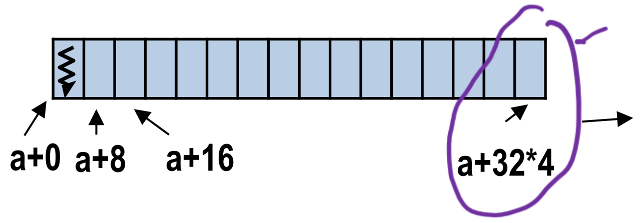


Instead issue one access
A[n:n+16*4]



- Pseudo code for non-contiguous access (divergence):

```
gpu void add(int *a, int *b, int *c) {
        c[tid] = a[tid*2] + b[tid];
}
```

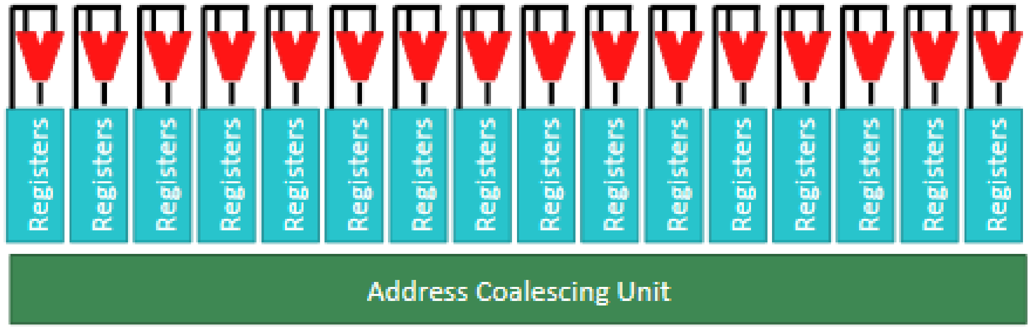


Instead issue two access
A[n :n+16*4],
A[n+16*4:n+32*4]

(hardware overhead to dynamically coalesce memory access…
and collect the operands)

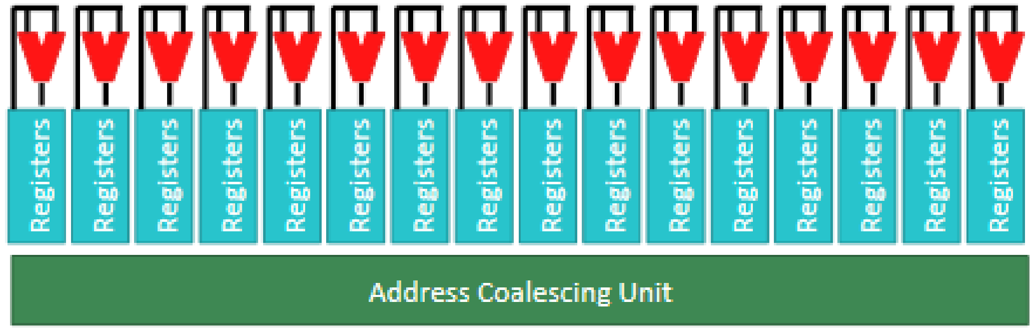
## SIMT Unit – A GPU Pipeline

- Similar to a wide CPU pipeline, except only fetch 1 instr.
- 16-wide physical ALU – specific to the AMD GPU
- 64 KB register state/SIMD unit – 4 SIMD units per CU
  - Much bigger (~64X) than CPUs
- Addressing coalescing key to good performance
  - Each thread potentially fetches a different piece of data
  - 64 separate addresses for AMD, 32 for NVIDIA (tradeoffs)



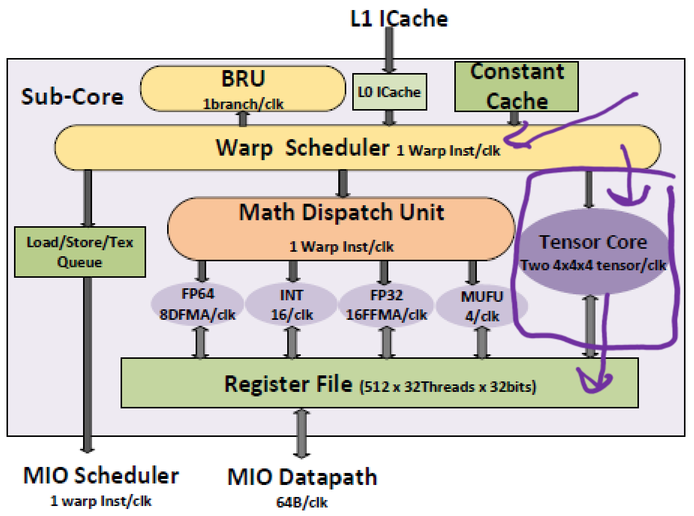
## L1 Caches

- Warp/Wavefront: 32 Threads, 32 Load/Store Ports to L1 Cache?
  - Non-starter, even banking doesn't solve the problem…
  - Should 32 cache misses cause 32 requests to memory!?
  - Aside: AMD hardware uses 64 threads per wavefronts
- Common case:
  - All threads in warp/wavefront access same cache block(s) – as shown on previous slide
- Addressing coalescing:
  - Dynamically combine addresses generated from each lane
  - Reduces in-flight memory requests, helps DRAM b/w, **important**



# Matrix Core Engines (AMD) / TensorCores (NVIDIA)

- Observation:
  - ML applications (and some others) can use reduced precision
  - Matrix multiplication operations (e.g., FMA) are common
- Solution:
  - Add specialized ALUs to SMs
- NVIDIA
  - Ampere: better sparsity support
  - Turing: FP16, INT8, INT4
  - Volta: FP16
- AMD
  - MI100 – MI300: similar INT/FP support



Volta SM Sub-Core [Choquette'18]

Lots of extra FLOPs for apps that can use them

## Memory System Optimizations

- GPUs are **throughput-oriented** processors
  - CPUs are **latency-oriented**

should sound familiar
(FGMT / SMT)

- Goal:
  - Hide the latency of memory accesses with many in-flight threads
  - Memory system needs must handle lots of overlapping requests

- But what if not enough threads to cover up the latency?

# Caches To The Rescue?

- Comparison: Modern CPU and GPU caches

40 x 64

(reconfig)

| | CPU | GPU |
|---|---|---|
| L1 D$ capacity | 64 KB | 16 KB ⟷ 128 KB |
| Active threads/work-items sharing L1 D$ | 2 | 2560 |
| L1 D$ capacity/thread | 32 KB | 6.4 bytes |
| Last level cache (LLC ) capacity | 8 MB | 4 MB |
| Active threads/work-items sharing LLC | 16 | 163840 |
| LLC capacity/thread | 0.5 MB | 25.6 bytes |

**GPU caches can't be used in the same way as CPU caches**

## GPU Caches

- Goal: maximize throughput, not latency (unlike CPUs)
  - Traditionally little temporal locality to exploit
  - Also little spatial locality, since coalescing logic handles most of it

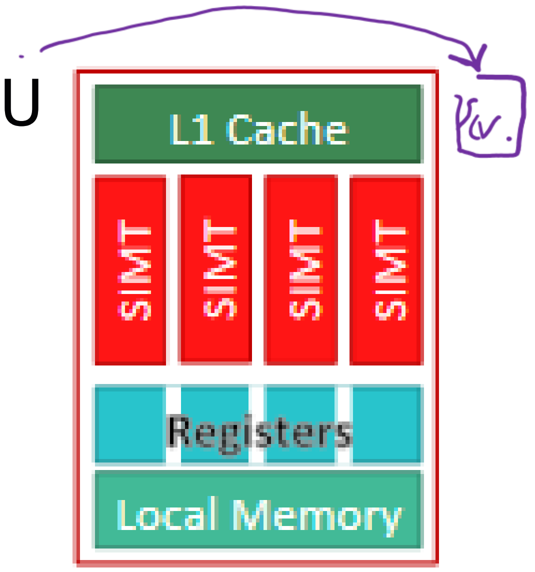
- L1 cache:
  - Coalesce requests to same cache block by different threads
  - Keep around long enough for all threads in warp/wavefront to hit
    - <u>Once</u>
  - **Ultimate goal**: **reduce number of requests sent to DRAM**

- L2 cache: DRAM staging buffer + some instruction reuse
  - **Ultimate goal**: **tolerate spikes in DRAM bandwidth**

- Use *specialized memories* (e.g., scratchpad, texture) for any temporal locality

# Scratchpad Memory (LDS/Shared Mem.)

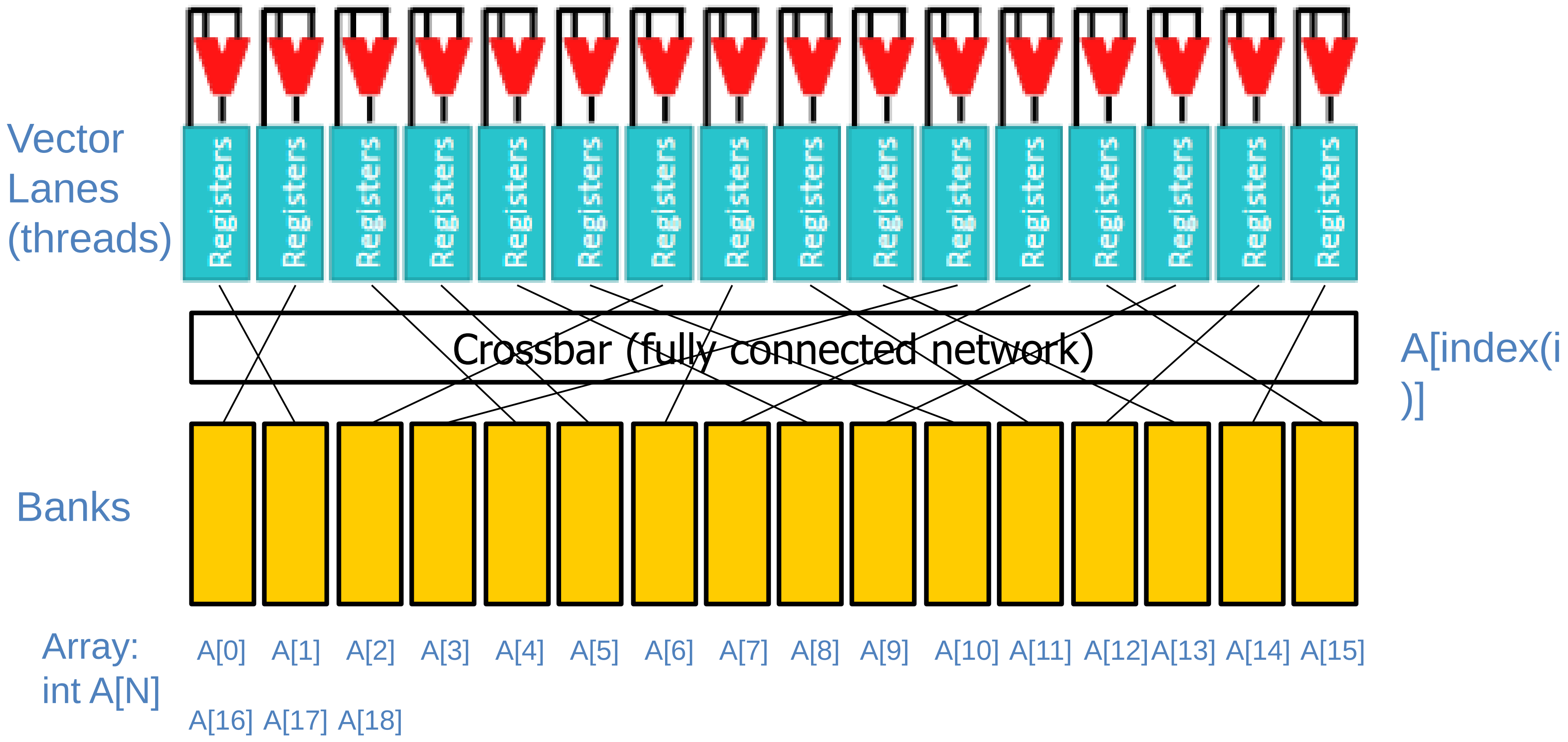
- GPUs also have scratchpads next to each CU
  - Usually co-located with L1 cache



- Scratchpad fundamentals:
  - Separate address space
  - Managed by software:
    - Renames address
    - Programmer manually manages capacity
      - Programmer brings data in when they need it
      - Programmer removes data when they're done with it
  - Each thread block/workgroup gets its own allocation
    - All warps/wavefronts in a thread block/workgroup share
    - Provides some isolation and security

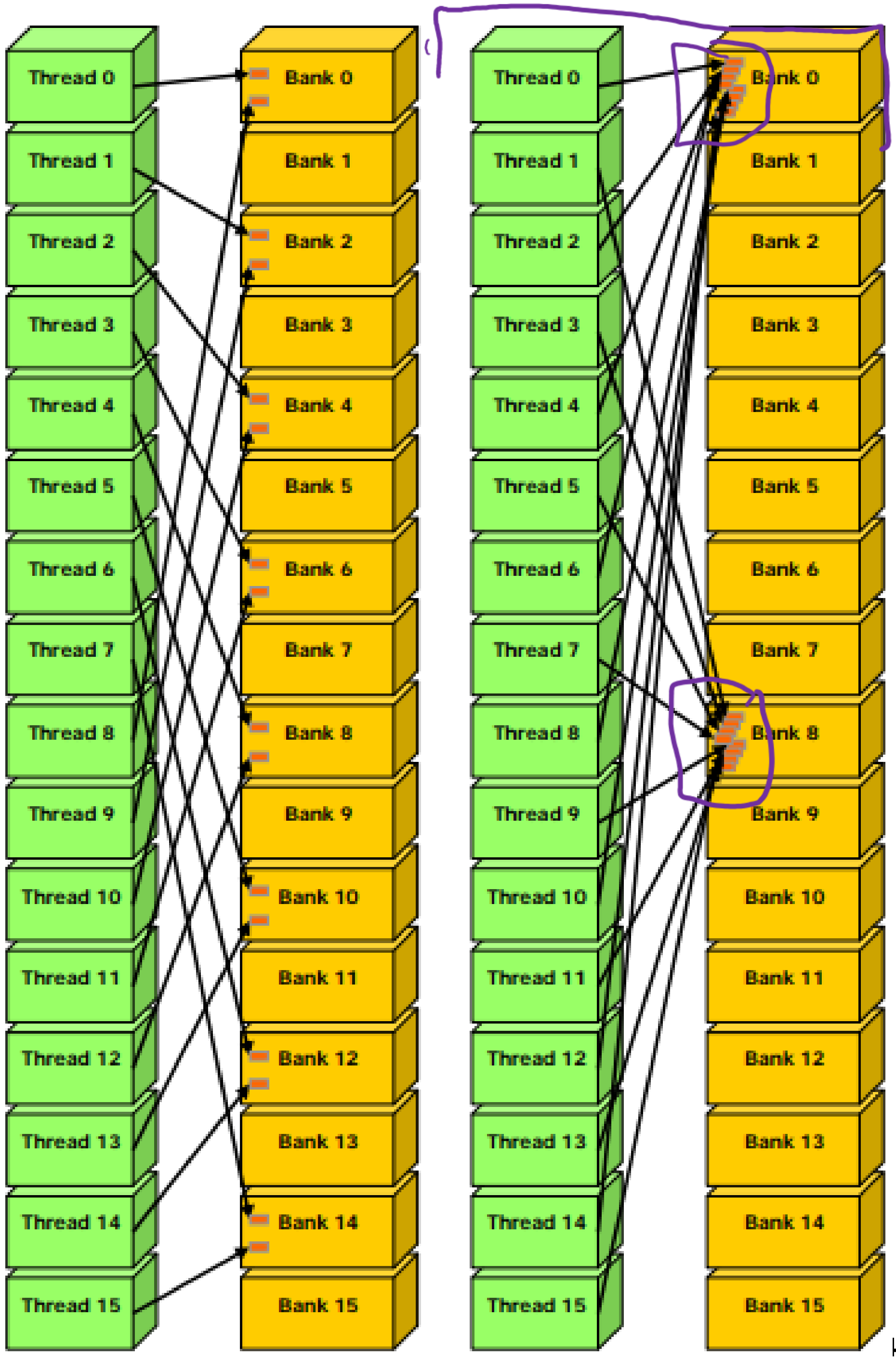
# Scratchpad Organization

- Banks divide the address space into chunks (corresponds to banks in hardware)
  - Stripe Data across it
- Threads can access different banks in parallel


Vector Lanes (threads)
Registers
Crossbar (fully connected network)
A[index(i)]
Banks
Array: int A[N]
A[0] A[1] A[2] A[3] A[4] A[5] A[6] A[7] A[8] A[9] A[10] A[11] A[12] A[13] A[14] A[15]
A[16] A[17] A[18]

## How does scratchpad deal with Conflicts?

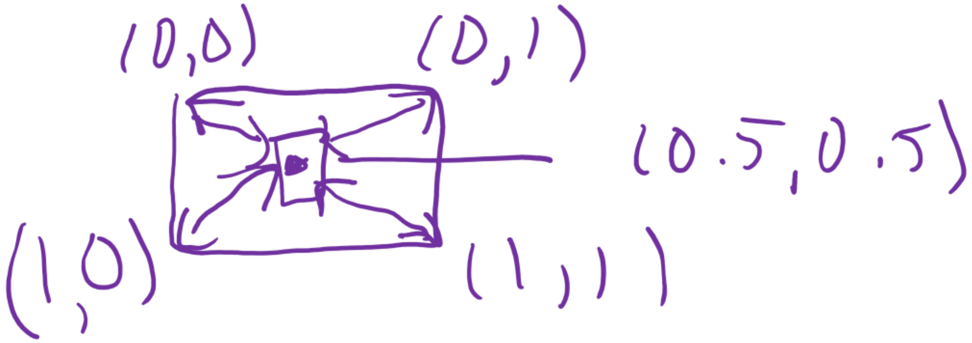
- Basic approach:
  1. Separate into non conflicting groups
  2. Service sequentially
- In contrast to cache, groups don't need to be the same sequential cache line



Left: Linear addressing with a stride of two 32-bit words causes 2-way bank conflicts.
Right: Linear addressing with a stride of eight 32-bit words causes 8-way bank conflicts.

# Other Memory Optimizations

- Read-only Memory/Constant Caches
  - Use for data that is guaranteed to be constant
- Texture Caches/Images/Samplers
  - Provides fast hardware 1D/2D/3D interpolation
  - Very useful for graphics
  - Before better caching for GPGPU, was used for compute apps



## Announcements 11/15/24

- Midterm next Wednesday (in class)
  - Focuses on everything since Midterm 1
  - Monday: exam review, bring questions!
  - Posted: old exams, problems
- HW7 grading ongoing
  - Will post solutions tonight to help with exam prep

## CPU/GPU Architectural Differences

**CPUs**

- Use caches and buffering in abundance.
- Few large cores.
- Much smaller BW.
- Fast synchronization.

**GPUs**

- More threads to hide latency to memory.
- Many small cores.
- Much higher BW.
- Slow/non-global synchronization.
- Special HW function units (transcendentals, textures)

# CPUs & GPUs have different characteristics.

**CPUs**

+ general-purpose (many types of apps)

+ multiple cores (compute in parallel).

+ fast response time for a single task.

- **Complexity** (few cores)

**GPUs**

+ designed to exploit data parallelism

+/- tradeoff single-thread performance for increased parallel processing

+ hide memory latencies.

+ more compute flops.

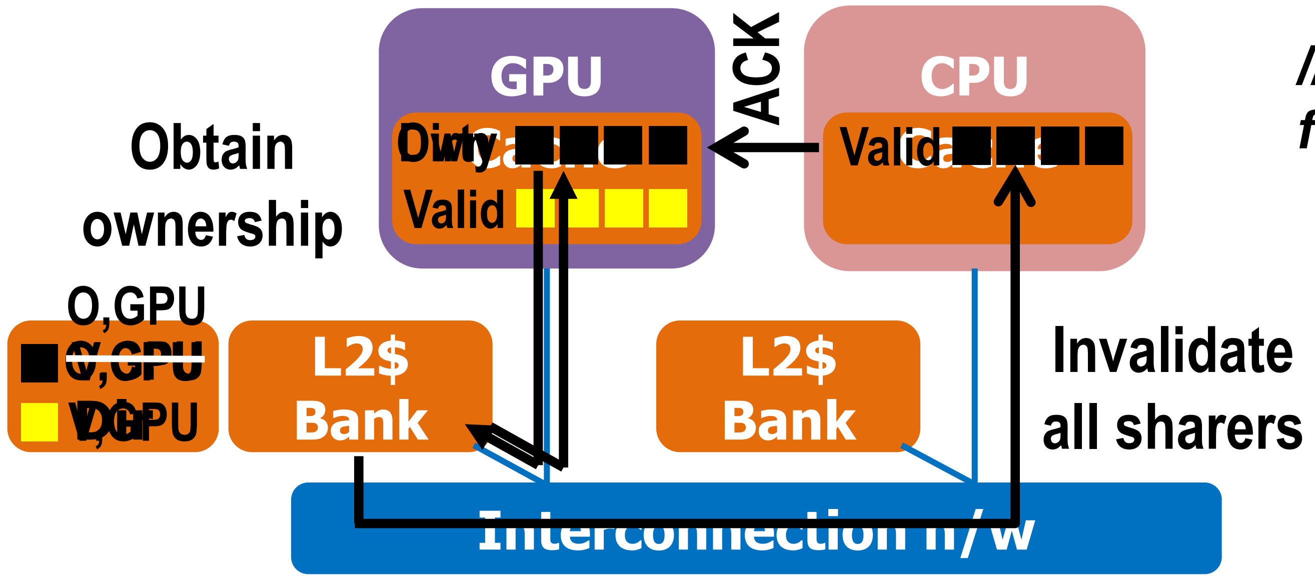
**- Limited by Amdahl's Law.**

## What’s “good” for executing on GPUs?

- (Traditionally)

- Abundant parallelism.
    - Single-threaded performance less important (MLP and TLP instead of ILP).
- Workloads that take advantage of “special features” (like textures).
- Workloads that require lots of bandwidth.
- Regular data access patterns

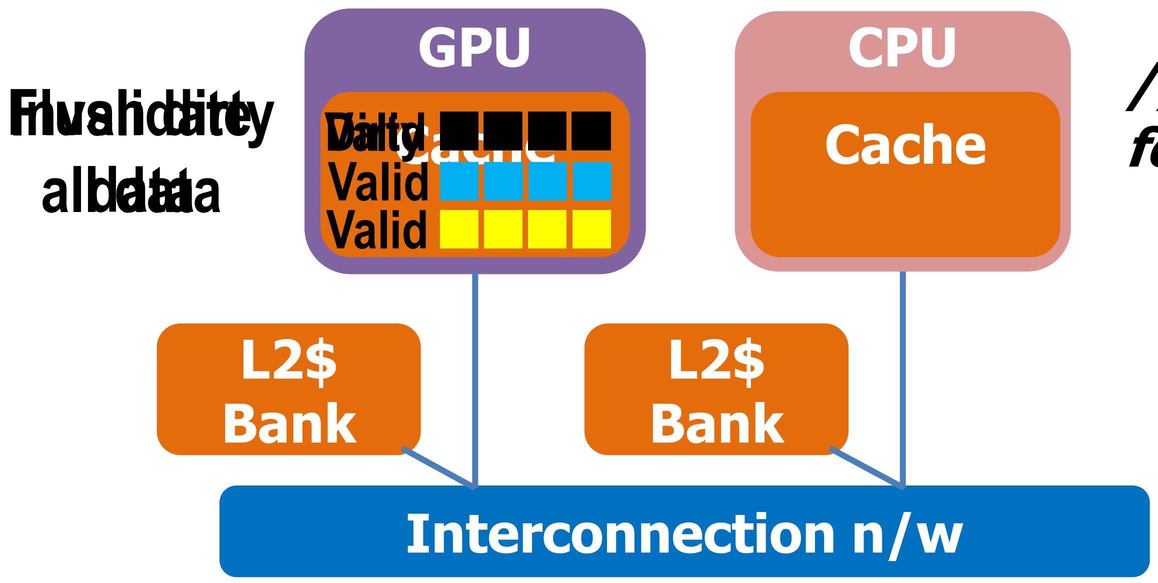
# CPU Coherence: MESI



```
// each thread
for i = r[tid]:r[tid+1]
    LOCK
    LD R1, A[i]; 🟨
    LD R2, B[i]; ■
    R3← Math(R1, R2);
    ST B[i], R3;
    UNLOCK
```

- Write miss: Get ownership, invalidate all sharers
- Read miss: Update sharers list
- Synchronization points are cheap
- BUT poor fit for GPUs:
    - Directory overhead, transient states, excessive traffic, indirection

# Traditional GPU Coherence



```
// unique per thread
for i = start:end
    LD R1, A[i];              🟨
    LD R2, B[i];              🟦
    R3← Math(R1, R2);
    ST C[i], R3;              ⬛
```

Each thread accesses independent data (no races)

No data reuse or data sharing

Coarse-grained synchronization

Optimized for streaming, data parallel applications

## GPU Memory Consistency Model

- Active area of research
- Tightly tied in with coherence protocol

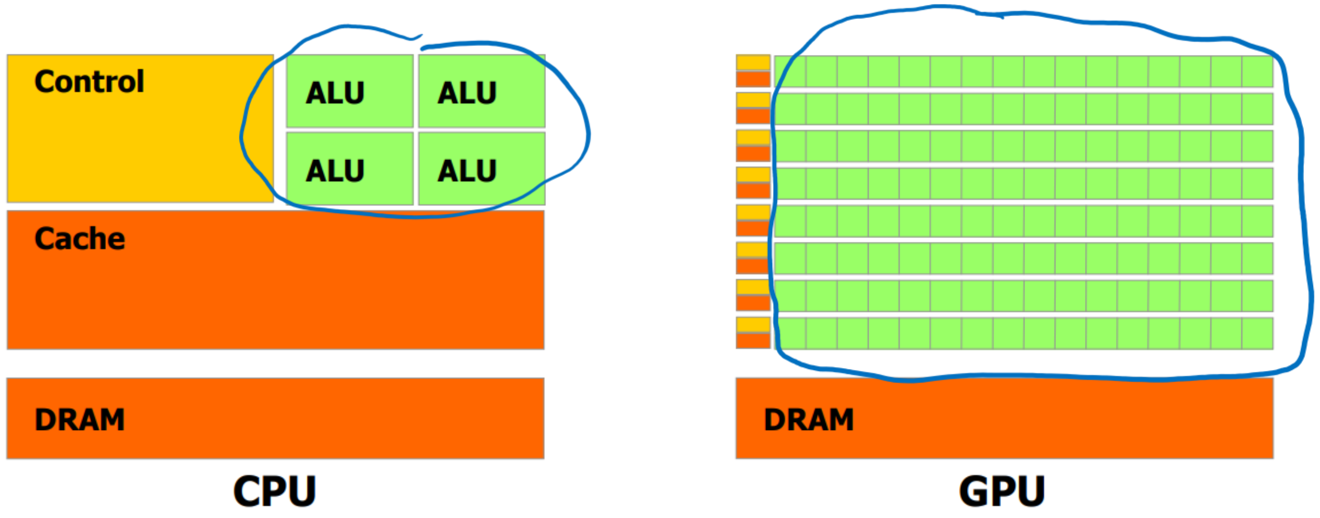
- Provides very weak guarantees
  - Respect program order within a single thread
  - Easy to design hardware
  - Programmers add ***fences*** to provide extra guarantees
    - Fence guarantee all previous accesses are visible before proceeding
    - … usually

- Most GPUs use a ***scoped*** memory consistency model
  - Only apply GPU fences locally if all users are local – less overhead
  - But more work for programmer

## Are GPUs awesome?
## … yes but…

# GPU's are more computationally dense right?

- **Conventional Wisdom:**
- GPUs use less cache, so more dense



- However, if you include register files….

| GPU | Register files + caches |
|---|---|
| NVIDIA GM204 GPU | 8.3 MB |
| AMD Hawaii GPU | 15.8 MB |
| Intel Core i7 CPU | 9.3 MB |

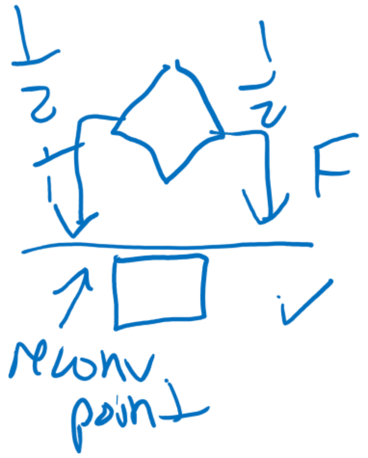
Did we really need that many threads???

# GPU Still have a lot of Overheads

- Memory Access:
  - Dynamic coalescing energy overheads
  - Cache thrashing from many threads ← (6.4 B/thr in $1)
  - Data needs to be laid out correctly (bank conflicts, communication, etc.)
- Control Flow: → NVIDIA
  - Hardware structures to track thread divergence (AMD: compiler)
- Operand Communication:
  - All communication between instructions goes through register files
- Scheduling Warps/Threads:
  - Dynamically decide which wards to execute
- Register File due to Multithreading
  - Each thread needs space in the register file for live values!

# Limits of GPUs

- SIMT Control Flow
  - Threads (warps/wavefronts) normally run in lockstep
  - But not all guaranteed to take same branch
  - Solution: reconvergence points … or use predication
  - Bad for performance and correctness
- Memory Divergence
  - Bank conflicts or cache misses for subset of threads delays warp
  - Data layout & partitioning important
  - Bad for perf
- Communication
  - Easy to communicate locally.  Expensive to communicate globally.
  - Active area of research



# Data Parallelism Recap

- Data Level Parallelism
  - "medium-grained" parallelism between ILP and TLP
  - Still one flow of execution (unlike TLP)
  - Compiler/programmer explicitly expresses it (unlike ILP)
- Hardware support: new "wide" instructions (SIMD)
  - Wide registers, perform multiple operations in parallel
- Trends
  - Wider: 64-bit (MMX, 1996), 128-bit (SSE2, 2000), 256-bit (AVX, 2011), 512-bit (Larrabee/Knights Corner)
  - More advanced and specialized instructions
- GPUs (**CS 758, CS/ECE/ME 759 cover in more detail**)
  - Embrace data parallelism via "SIMT" execution model
  - Becoming more programmable all the time
- Today's chips exploit parallelism at all levels: ILP, DLP, TLP

## Bonus

# Flynn's Taxonomy, Revisited

- Focus: Data parallel workloads
  - Independent, identical computation on multiple data inputs
- MIMD (Multiple Instruction, Multiple Data):
  - Split independent work over multiple processors
  - Subcategory: SPMD (Single Program, Multiple Data)
    - Only if work is identical (same program)
- SIMD (Single Instruction, Multiple Data):
  - Split identical, independent work over multiple execution units
  - More efficient: eliminate redundant fetch/decode vs. SPMD/MIMD
  - Use single PC and single register file

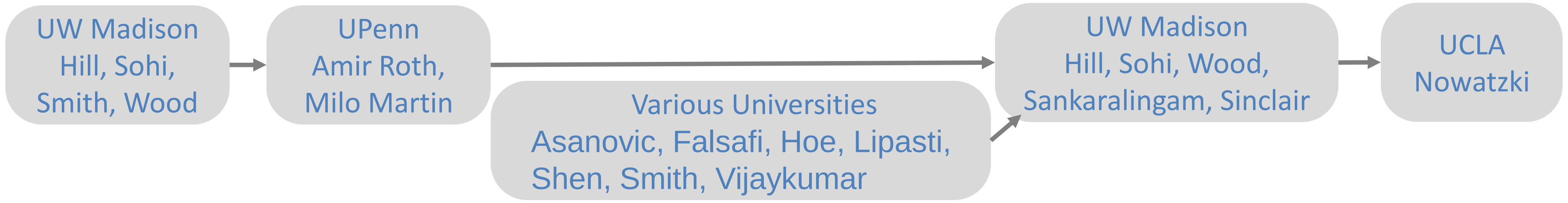
## Flynn’s Taxonomy, Revisited (Cont.)

- SIMD’s cousin: SIMT (Single Instruction, Multiple Thread)
  - Split identical, independent work over multiple <u>lockstep</u> threads
  - One PC for group of lockstep threads, but multiple register files
  - This is what GPUs do today
  - Work well for **streaming** applications

- Sidenote:
  - People use SIMT and SIMD somewhat interchangeably
  - They do have differences though

## Today's "CPU" Vectors / SIMD

Slide History/Attribution Diagram:


UW Madison Hill, Sohi, Smith, Wood
UPenn Amir Roth, Milo Martin
Various Universities Asanovic, Falsafi, Hoe, Lipasti, Shen, Smith, Vijaykumar
UW Madison Hill, Sohi, Wood, Sankaralingam, Sinclair
UCLA Nowatzki

## Example Vector ISA Extensions (SIMD)

- Extend ISA with floating point (FP) vector storage …
  - **Vector register**: fixed-size array of 32- or 64- bit FP elements
  - **Vector length**: For example: 4, 8, 16, 64, …
- … and example operations for vector length of 4
  - Load vector: **`ldf.v [X+r1]->v1`**

    **`ldf [X+r1+0]->`**$\mathtt{v1_0}$

    **`ldf [X+r1+1]->`**$\mathtt{v1_1}$

    **`ldf [X+r1+2]->`**$\mathtt{v1_2}$

    **`ldf [X+r1+3]->`**$\mathtt{v1_3}$
  - Add two vectors: **`addf.vv v1,v2->v3`**

    **`addf`** $\mathtt{v1_i, v2_i \to v3_i}$ **`(where i is 0,1,2,3)`**
  - Add vector to scalar: **`addf.vs v1,f2,v3`**

    **`addf`** $\mathtt{v1_i, f2 \to v3_i}$ **`(where i is 0,1,2,3)`**
- Today's vectors: short (128 or 256 bits), but fully parallel

## Example Use of Vectors – 4-wide

```
ldf [X+r1]->f1
mulf f0,f1->f2
ldf [Y+r1]->f3
addf f2,f3->f4
stf f4->[Z+r1]
addi r1,4->r1
blti r1,4096,L1
```

7x1024 instructions

$\Rightarrow$

```
ldf.v [X+r1]->v1
mulf.vs v1,f0->v2
ldf.v [Y+r1]->v3
addf.vv v2,v3->v4
stf.v v4,[Z+r1]
addi r1,16->r1
blti r1,4096,L1
```

7x256 instructions
(4x fewer instructions)

- Operations
  - Load vector: `ldf.v [X+r1]->v1`
  - Multiply vector to scalar: `mulf.vs v1,f2->v3`
  - Add two vectors: `addf.vv v1,v2->v3`
  - Store vector: `stf.v v1->[X+r1]`
- Performance?
  - Best case: 4x speedup
  - But, vector instructions don’t always have single-cycle throughput
    - Execution width (implementation) vs vector width (ISA)

## Vector Datapath & Implementation

- Vector insn. are just like normal insn… only "wider"
  - Single instruction fetch (no extra $N^2$ checks)
  - Wide register read & write (not multiple ports)
  - Wide execute: replicate floating point unit (same as superscalar)
  - Wide bypass (avoid $N^2$ bypass problem)
  - Wide cache read & write (single cache tag check)

- Execution width (implementation) vs vector width (ISA)
  - Example: Pentium 4 and "Core 1" executes vector ops at half width
  - "Core 2" executes them at full width

- Because they are just instructions…
  - …superscalar execution of vector instructions
  - Multiple n-wide vector instructions per cycle

## Intel’s SSE2/SSE3/SSE4…

- **Intel SSE2 (Streaming SIMD Extensions 2)** - 2001
  - 16 128bit floating point registers (`xmm0–xmm15`)
  - Each can be treated as 2x64b FP or 4x32b FP (“packed FP”)
    - Or 2x64b or 4x32b or 8x16b or 16x8b ints (“packed integer”)
    - Or 1x64b or 1x32b FP (just normal scalar floating point)
  - Original SSE: only 8 registers, no packed integer support

- Other vector extensions
  - AMD 3DNow!: 64b (2x32b)
  - PowerPC AltiVEC/VMX: 128b (2x64b or 4x32b)

- Looking forward for x86
  - Intel’s “Sandy Bridge” brings 256-bit vectors to x86
  - Intel’s “Knights Ferry” multicore added 512-bit vectors to x86

## Other Vector Instructions

- These target specific domains: e.g., image processing, crypto
  - Vector reduction (sum all elements of a vector)
  - Geometry processing: 4x4 translation/rotation matrices
  - Saturating (non-overflowing) subword add/sub: image processing
  - Byte asymmetric operations: blending and composition in graphics
  - Byte shuffle/permute: crypto
  - Population (bit) count: crypto
  - Max/min/argmax/argmin: video codec
  - Absolute differences: video codec
  - Multiply-accumulate: digital-signal processing
  - Special instructions for AES encryption
- More advanced (but in Intel's Larrabee/Knights Ferry)
  - Scatter/gather loads: indirect store (or load) from a vector of pointers
  - Vector mask: predication (conditional execution) of specific elements

## Using Vectors in Your Code

Slide History/Attribution Diagram:

## Using Vectors in Your Code

- Write in assembly
  - High performance, but painful
- Use "intrinsic" functions and data types
  - For example:  _mm_mul_ps() and  "__m128" datatype
- Use vector data types
  - typedef double v2df __attribute__ ((vector_size (16)));
- Use a library someone else wrote
  - Let them do the hard work
  - Matrix and linear algebra packages
- Let the compiler do it (automatic vectorization, with feedback)
  - GCC's "-ftree-vectorize" option, -ftree-vectorizer-verbose=**n**
  - Limited impact for C/C++ code (old, hard problem)

## SAXPY Example: Best Case

- Code

```
void saxpy(float* x, float* y,
           float* z, float a,
           int length) {
  for (int i = 0; i < length; i++) {
    z[i] = a*x[i] + y[i];
  }
}
```

- Scalar

```
.L3:
   movss (%rdi,%rax), %xmm1
   mulss %xmm0, %xmm1
   addss (%rsi,%rax), %xmm1
   movss %xmm1, (%rdx,%rax)
   addq  $4, %rax
   cmpq  %rcx, %rax
   jne   .L3
```

- Auto Vectorized

```
.L6:
   movaps (%rdi,%rax), %xmm1
   mulps %xmm2, %xmm1
   addps (%rsi,%rax), %xmm1
   movaps %xmm1, (%rdx,%rax)
   addq  $16, %rax
   incl  %r8d
   cmpl  %r8d, %r9d
   ja .L6
```

  - + Scalar loop to handle last few iterations (if length % 4 != 0)
  - “mulps”: multiply packed ‘single’

# SAXPY Example: Actual

- Code

```
void saxpy(float* x, float* y,
           float* z, float a,
           int length) {
  for (int i = 0; i < length; i++) {
    z[i] = a*x[i] + y[i];
  }
}
```

- Scalar

```
.L3:
   movss (%rdi,%rax), %xmm1
   mulss %xmm0, %xmm1
   addss (%rsi,%rax), %xmm1
   movss %xmm1, (%rdx,%rax)
   addq  $4, %rax
   cmpq  %rcx, %rax
   jne   .L3
```

- Auto Vectorized

```
.L8:
   movaps  %xmm3, %xmm1
   movaps  %xmm3, %xmm2
   movlps  (%rdi,%rax), %xmm1
   movlps  (%rsi,%rax), %xmm2
   movhps  8(%rdi,%rax), %xmm1
   movhps  8(%rsi,%rax), %xmm2
   mulps %xmm4, %xmm1
   incl  %r8d
   addps %xmm2, %xmm1
   movaps %xmm1, (%rdx,%rax)
   addq  $16, %rax
   cmpl  %r9d, %r8d
   jb .L8
```

  - + Explicit alignment test
  - + Explicit aliasing test

# Bridging "Best Case" and "Actual"

- Align arrays

```
typedef float afloat __attribute__ ((__aligned__(16)));
void saxpy(afloat* x,
           afloat* y,
           afloat* z,
           float a, int length) {
  for (int i = 0; i < length; i++) {
    z[i] = a*x[i] + y[i];
  }
}
```

- Avoid aliasing check

```
typedef float afloat __attribute__ ((__aligned__(16)));
void saxpy(afloat* __restrict__ x,
           afloat* __restrict__ y,
           afloat* __restrict__ z, float a, int length)
```

- Even with both, still has the "last few iterations" code

## Advanced vectorization….

- Simple model is limited
  - Wide compute
  - Wide load/store of consecutive addresses
  - Allows for “SOA” (structures of arrays) style parallelism
- Complex features:
  - **Vector masks**
    - Conditional execution on a per-element basis
    - Allows vectorization of conditionals
  - **Scatter/gather**
    - a[i] = b[y[i]]          b[y[i]] = a[i]
    - Helps with sparse matrices, “AOS” (array of structures) parallelism

# Vector Masks (Predication)

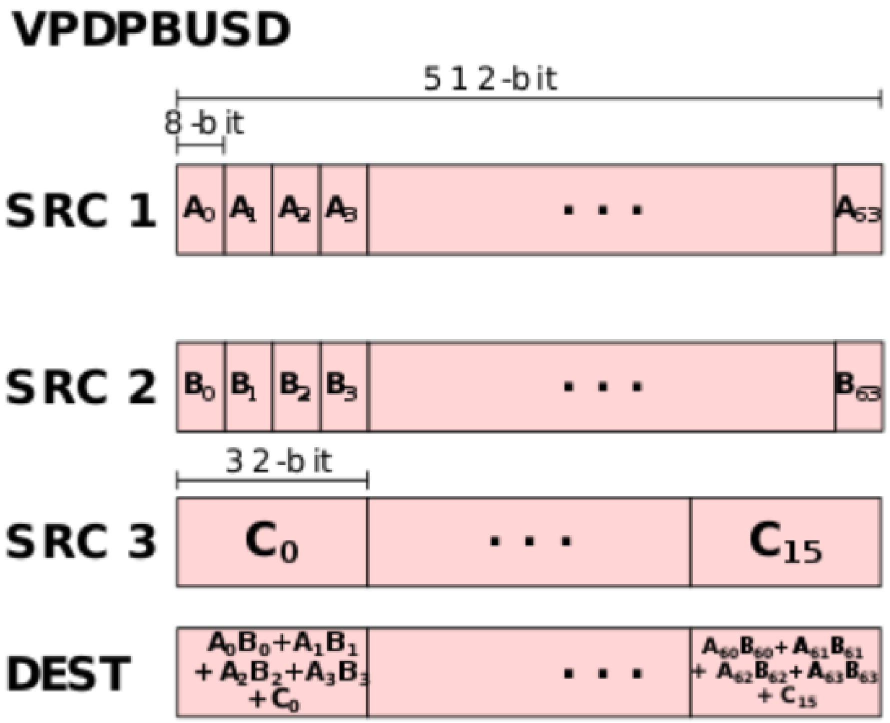
- Recall "cmov" prediction to avoid branches
- **Vector Masks**: 1 bit per vector element
  - Implicit predicate in all vector operations

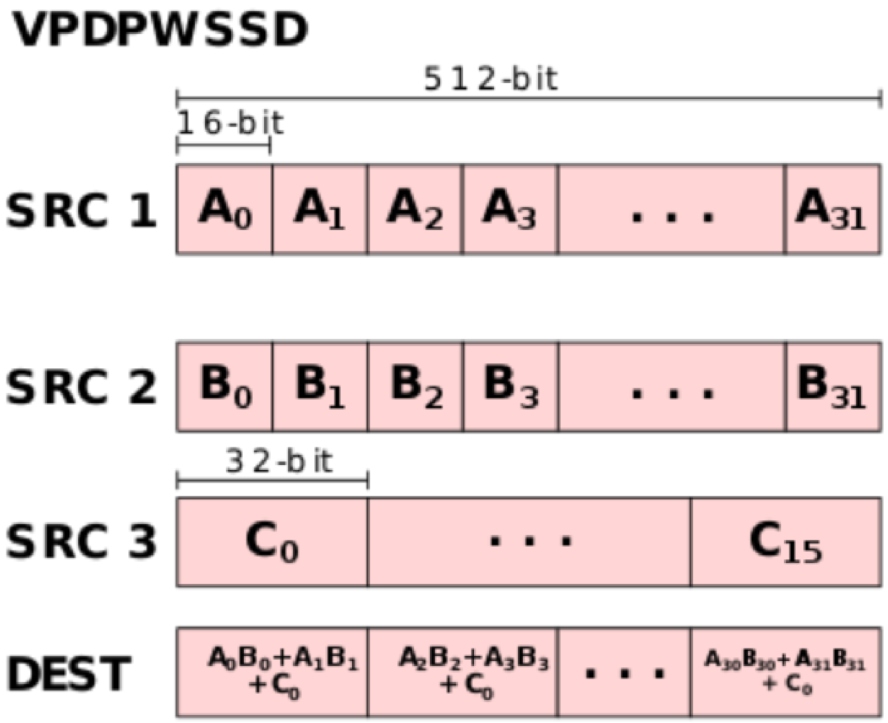
    `for (I=0; I<N; I++) if (maskI) { vop… }`
  - Usually stored in a "scalar" register (up to 64-bits)
  - Used to vectorize loops with conditionals in them

    `cmp_eq.v`, `cmp_lt.v`, etc.: sets vector predicates

```
for (I=0; I<32; I++)
   if (X[I] != 0.0) Z[I] = A/X[I];

ldf.v [X+r1] -> v1
cmp_ne.v v1,f0 -> r2        // 0.0 is in f0
divf.sv {r2} v1,f1 -> v2   // A is in f1
stf.v {r2} v2 -> [Z+r1]
```

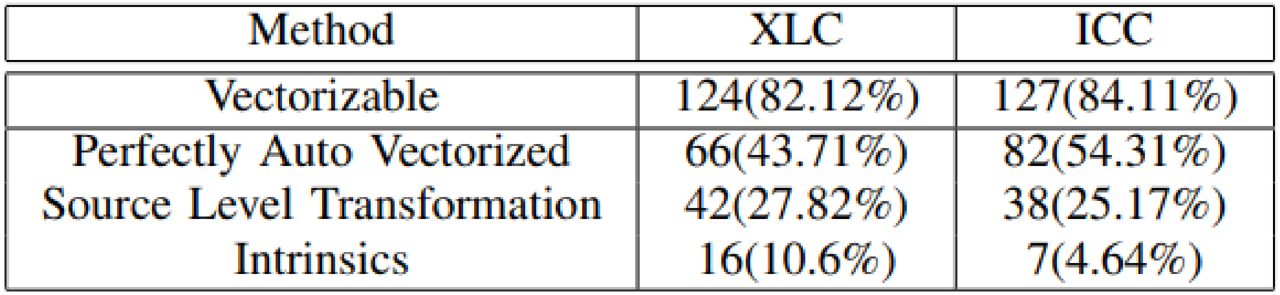
# Vector Neural Network Instructions (VNNI) Examples


VPDPBUSD
512-bit
8-bit
SRC 1
$A_0$ $A_1$ $A_2$ $A_3$ . . . $A_{63}$
SRC 2
$B_0$ $B_1$ $B_2$ $B_3$ . . . $B_{63}$
32-bit
SRC 3
$C_0$ . . . $C_{15}$
DEST
$A_0B_0+A_1B_1+A_2B_2+A_3B_3+C_0$
. . .
$A_{60}B_{60}+A_{61}B_{61}+A_{62}B_{62}+A_{63}B_{63}+C_{15}$



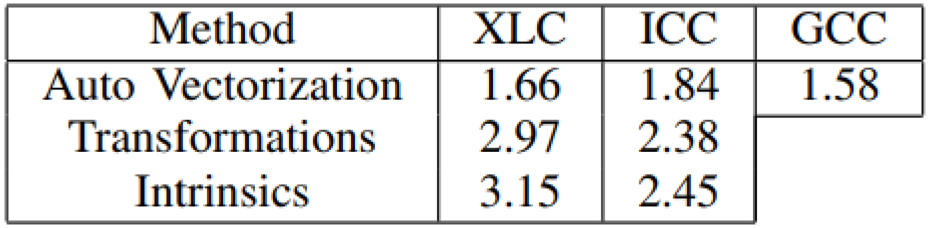
- Similar to TensorOps in NVIDIA GPUs

## An Evaluation of Vectorizing Compilers (Maleki et. al. 2011)



| Method | XLC | ICC |
|---|---|---|
| Vectorizable | 124(82.12%) | 127(84.11%) |
| Perfectly Auto Vectorized | 66(43.71%) | 82(54.31%) |
| Source Level Transformation | 42(27.82%) | 38(25.17%) |
| Intrinsics | 16(10.6%) | 7(4.64%) |

TABLE IV
LOOP CLASSIFICATION BASED ON THE METHOD USED TO ACHIEVE THE BEST SPEEDUP.



| Method | XLC | ICC | GCC |
|---|---|---|---|
| Auto Vectorization | 1.66 | 1.84 | 1.58 |
| Transformations | 2.97 | 2.38 | |
| Intrinsics | 3.15 | 2.45 | |

TABLE VIII
THE AVERAGE SPEEDUPS OBTAINED BY EACH METHOD.

TSVC Benchmarks: Pretty Simple Loop Codes (originally Fortran)

# Scatter Stores & Gather Loads

- How to vectorize:

```
for(int i = 1, i<N, i++) {
    int bucket = val[i] / scalefactor;
    found[bucket] = 1;
```

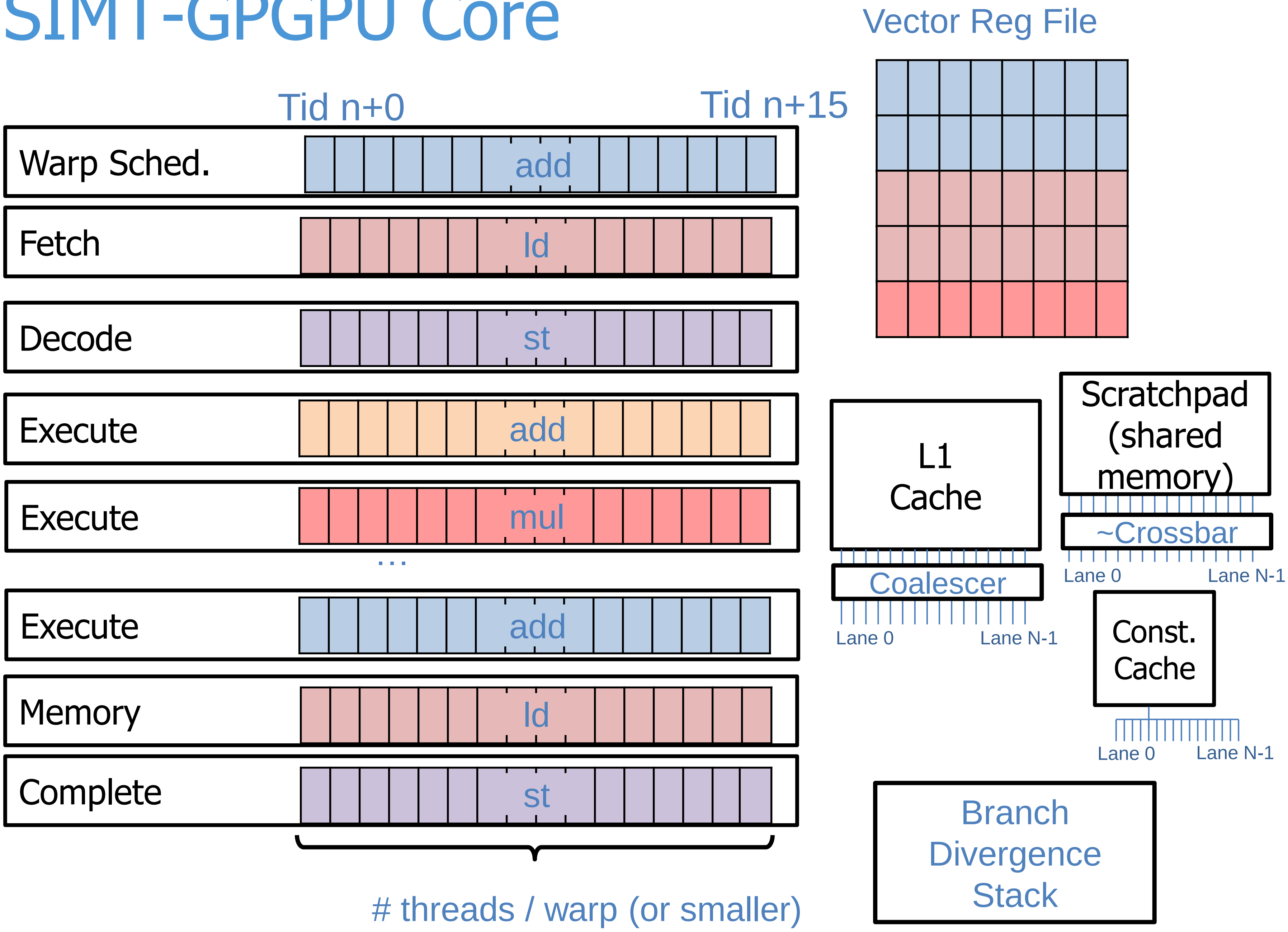
  - Easy to vectorize the divide, but what about the load/store?
- Solution: hardware support for vector "scatter stores"
    - `stf.v v2->[r1+v1]`
  - Each address calculated from $r1+v1_i$

$stf\ v2_0 \text{->} [r1+v1_0]$, $stf\ v2_1 \text{->} [r1+v1_1]$,

$stf\ v2_2 \text{->} [r1+v1_2]$, $stf\ v2_3 \text{->} [r1+v1_3]$

- Vector "gather loads" defined analogously
  - `ldf.v [r1+v1]->v2`
- Scatter/gathers slower than regular vector load/store ops
  - Still provides a throughput advantage over non-vector version

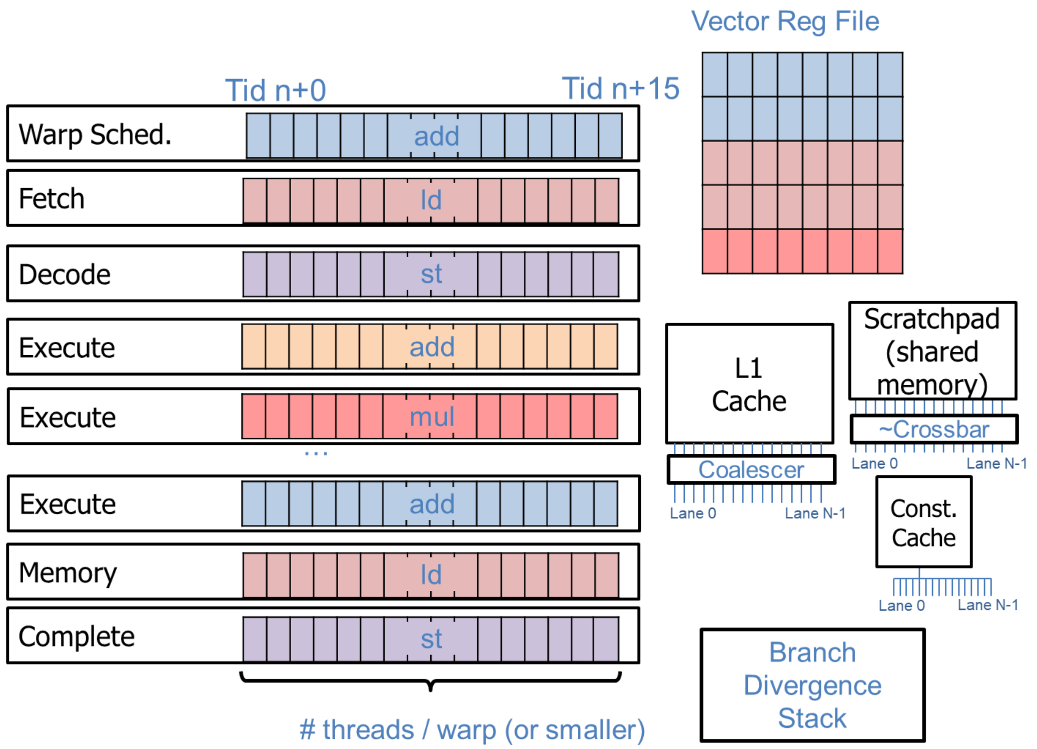
# SIMT-GPGPU Core



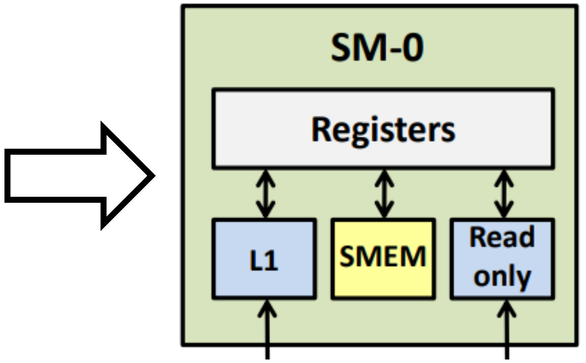
## Scaling up?

- Need to scale up past the design I'm showing here…
- Can't naively make the vector-width longer
  - (bad for programmability, not all programs have <32 vector width)
- So, we go multicore!

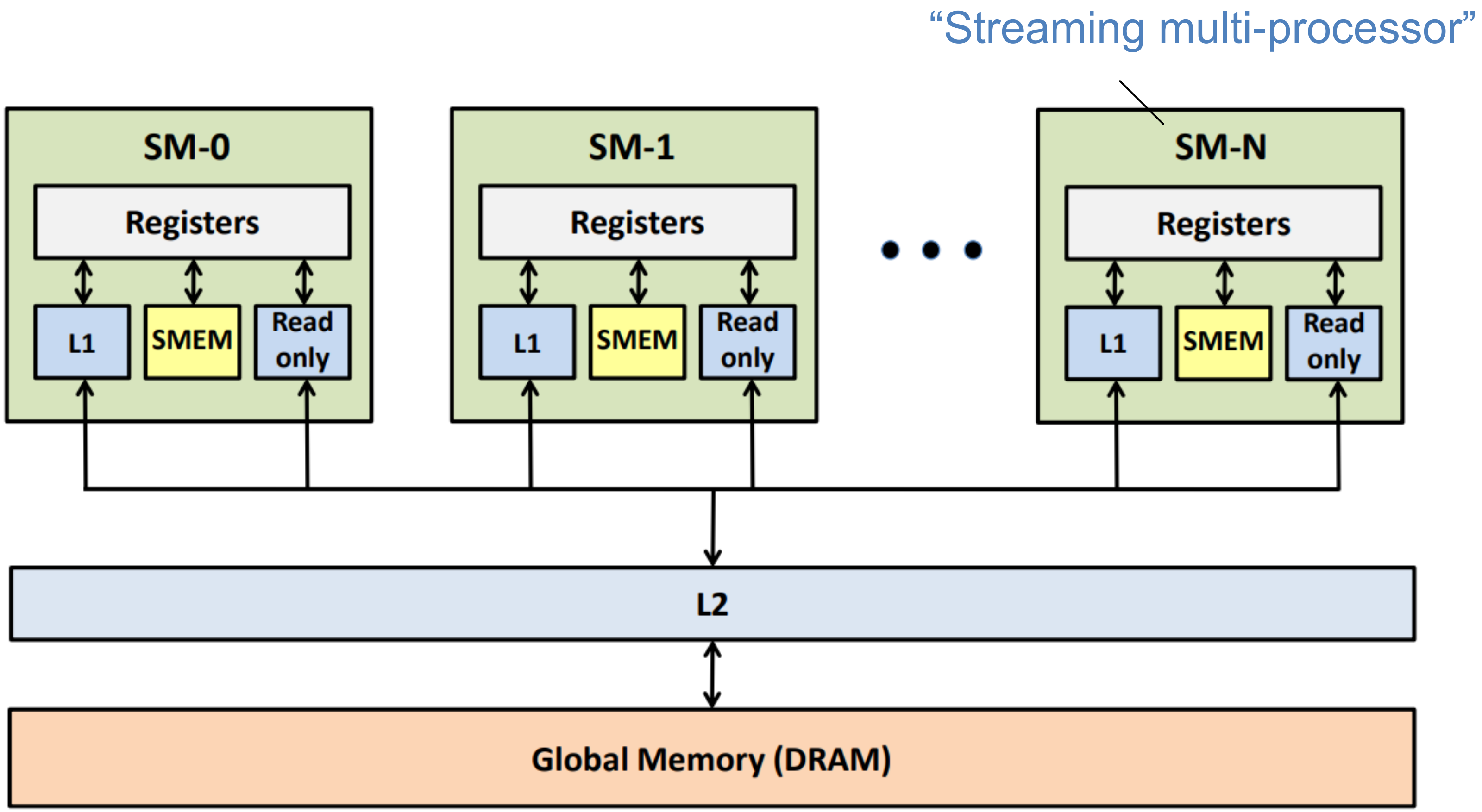
# SIMT-GPGPU Core



“Streaming multi-processor”



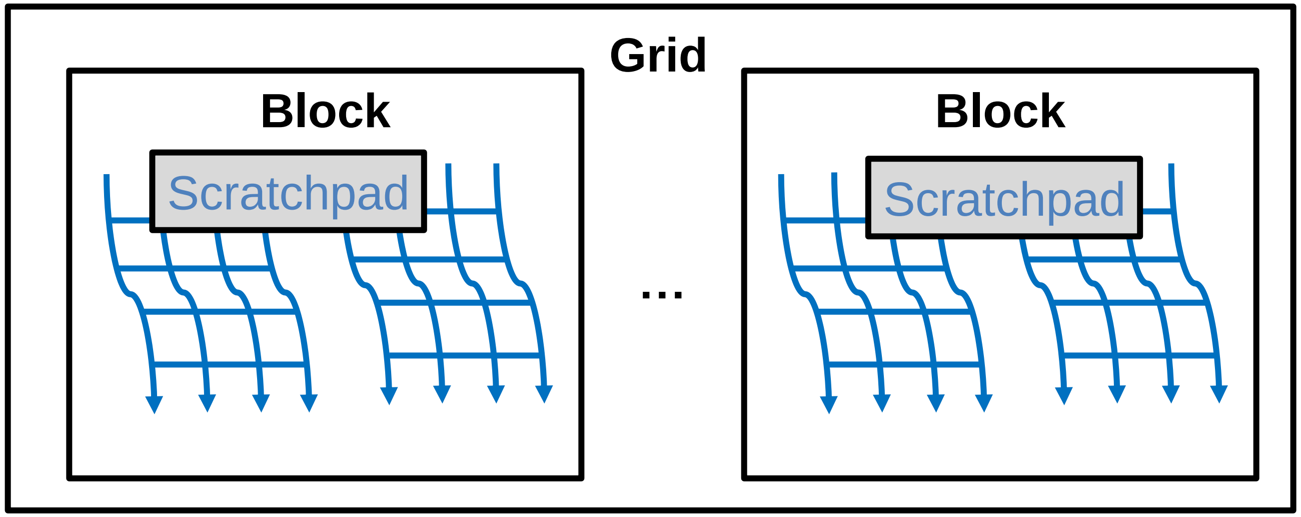
# GPU from Top Level



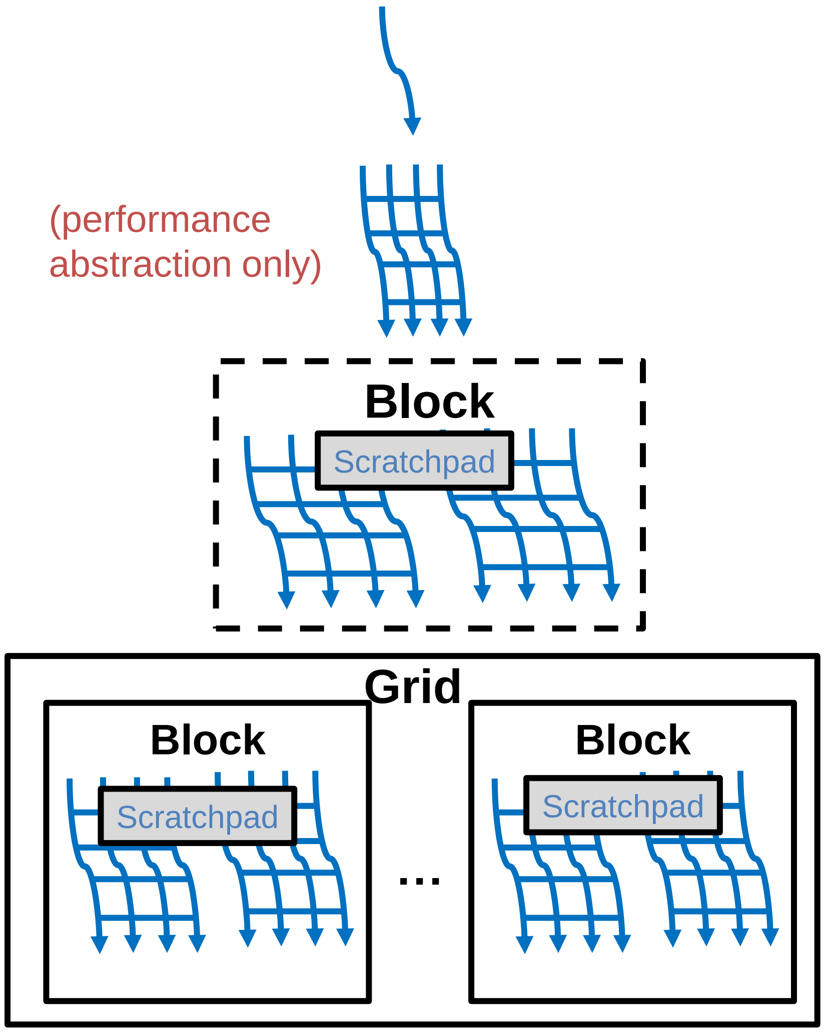
## Programming Abstractions when we scale up…

## Who should share a scratchpad?

- All cores share one: Good for programmer, but …
  - How to attain low latency and high bandwidth if shared?
- Each Warp gets its own: limited sharing potential
- **What to do? Add another level of the hierarchy:**
  - Block: Group of threads that can access the same shared **SPAD**
  - Multiple blocks may be mapped to one GPGPU Core
  - But each block can only be mapped to one particular core



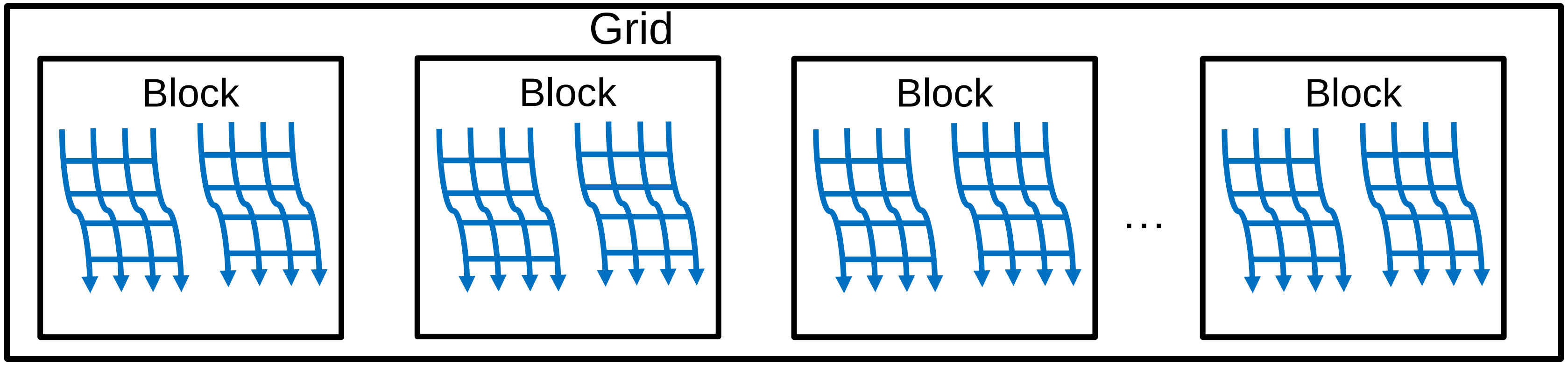
# GPU SIMT Model Terminology



| NVIDIA CUDA/HIP | AMD OpenCL |
|---|---|
| Thread | Work-item |
| Warp | Wavefront |
| Thread Block | Workgroup |
| Grid (Kernel) | NDRange (Kernel) |

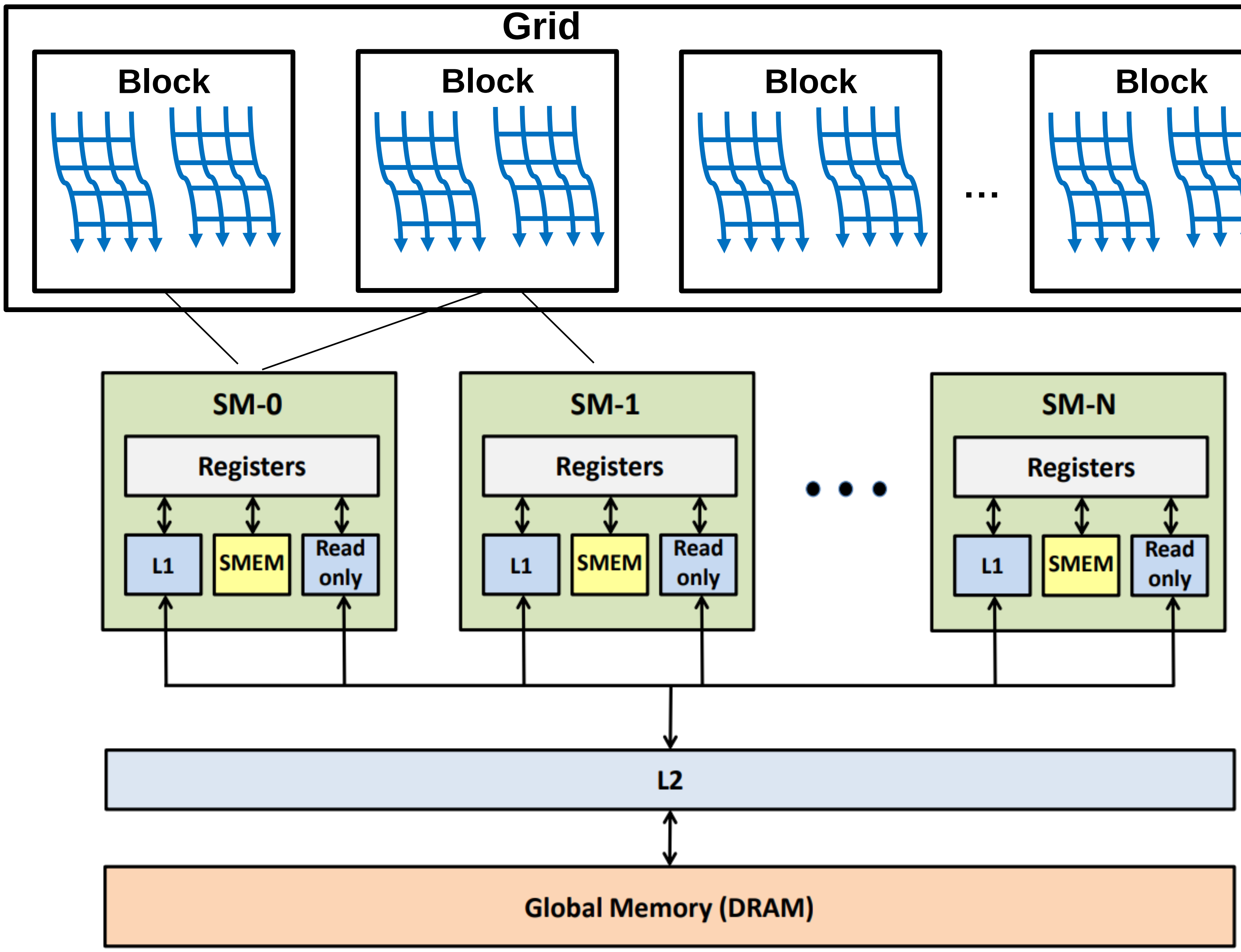
# Synchronization & Cache Coherence

- Fun Fact: No prescribed way to synchronize between Blocks



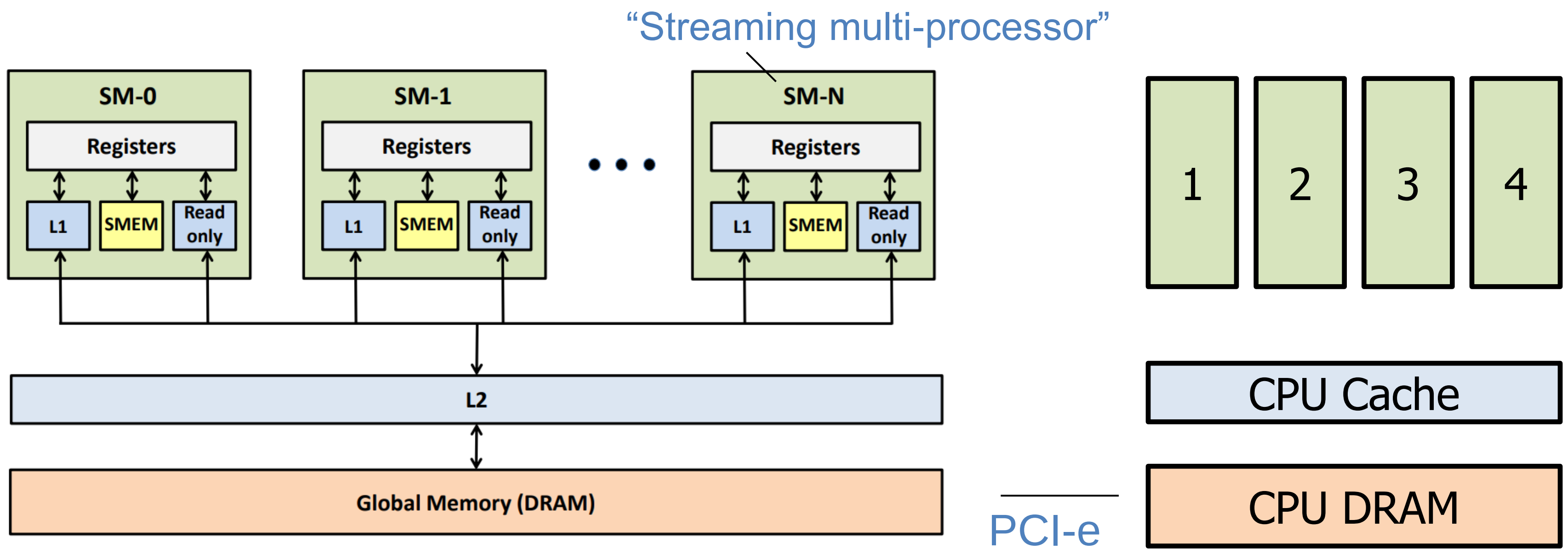
  - __syncthreads(): block-wide barrier

# Core complete, here's the Overall Picture



"Streaming multi-processor"

# Typical CPU/GPU Integration



## Memory System Optimizations

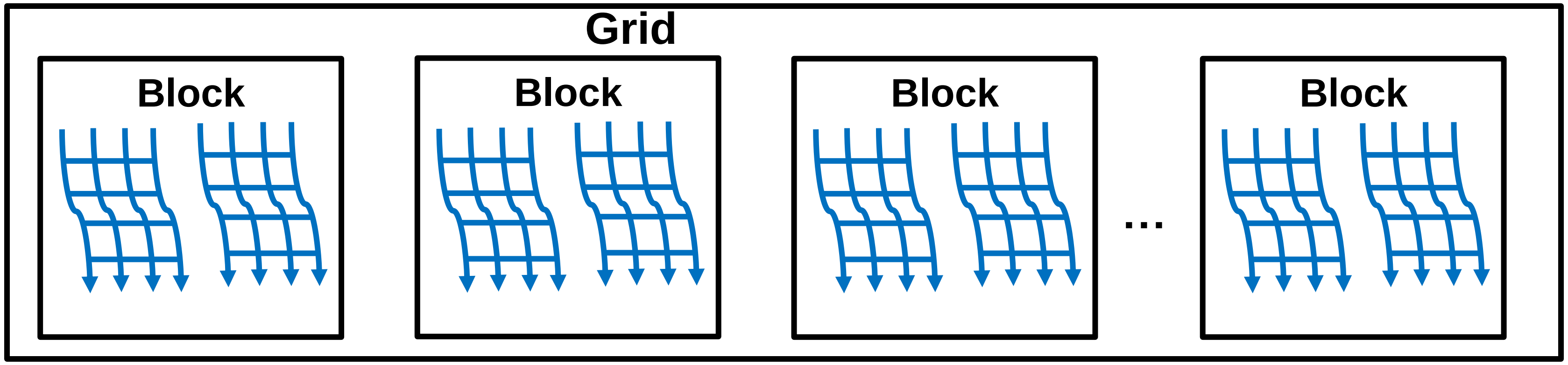
- GPUs are **throughput-oriented** processors
  - CPUs are **latency-oriented**

- Goal:
  - Hide the latency of memory accesses with many in-flight threads
  - Memory system needs must handle lots of overlapping requests

- But what if not enough threads to cover up the latency?

# Synchronization & Cache Coherence

- Fun Fact: No prescribed way to synchronize between Blocks
  - __syncthreads(): block-wide barrier



- Not really supposed to communicate between blocks… so GPUs neglect hardware coherence between cores…
- Coherence becomes software's job:
  - __threadfence_system: "all writes to all memory made by the calling thread before the call to __threadfence_system() are observed by all threads … as occurring before all writes to all memory made by the calling thread after the call"

## GPU Memory Consistency Model

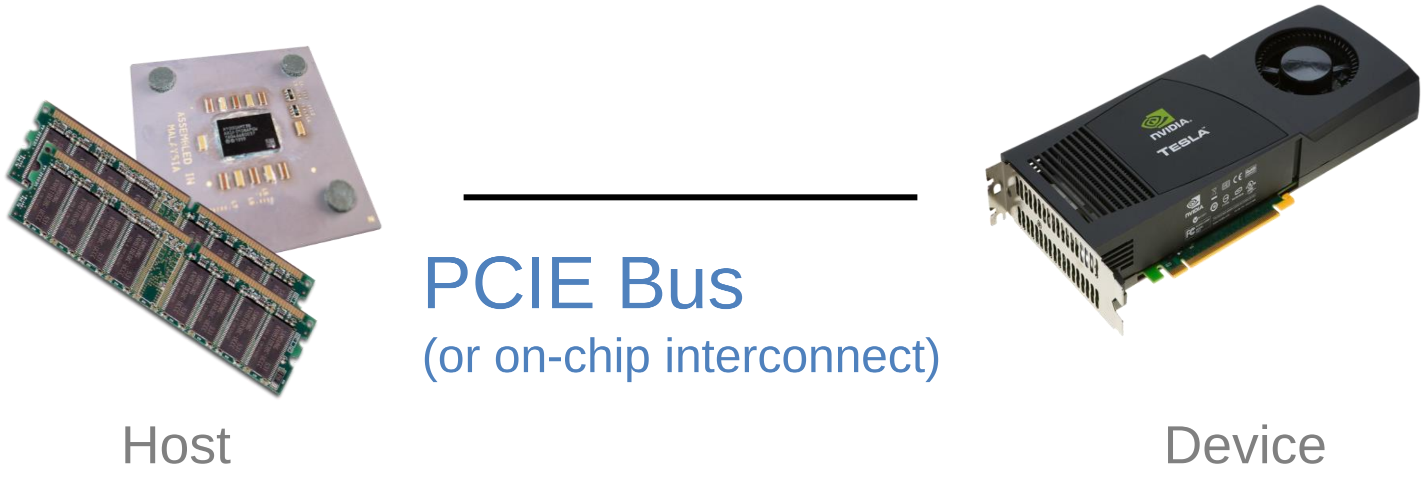
- Active area of research
- Tightly tied in with coherence protocol

- Provides very weak guarantees
  - Respect program order within a single thread
  - Easy to design hardware
  - Programmers add ***fences*** to provide extra guarantees
    - Fence guarantee all previous accesses are visible before proceeding
    - … usually

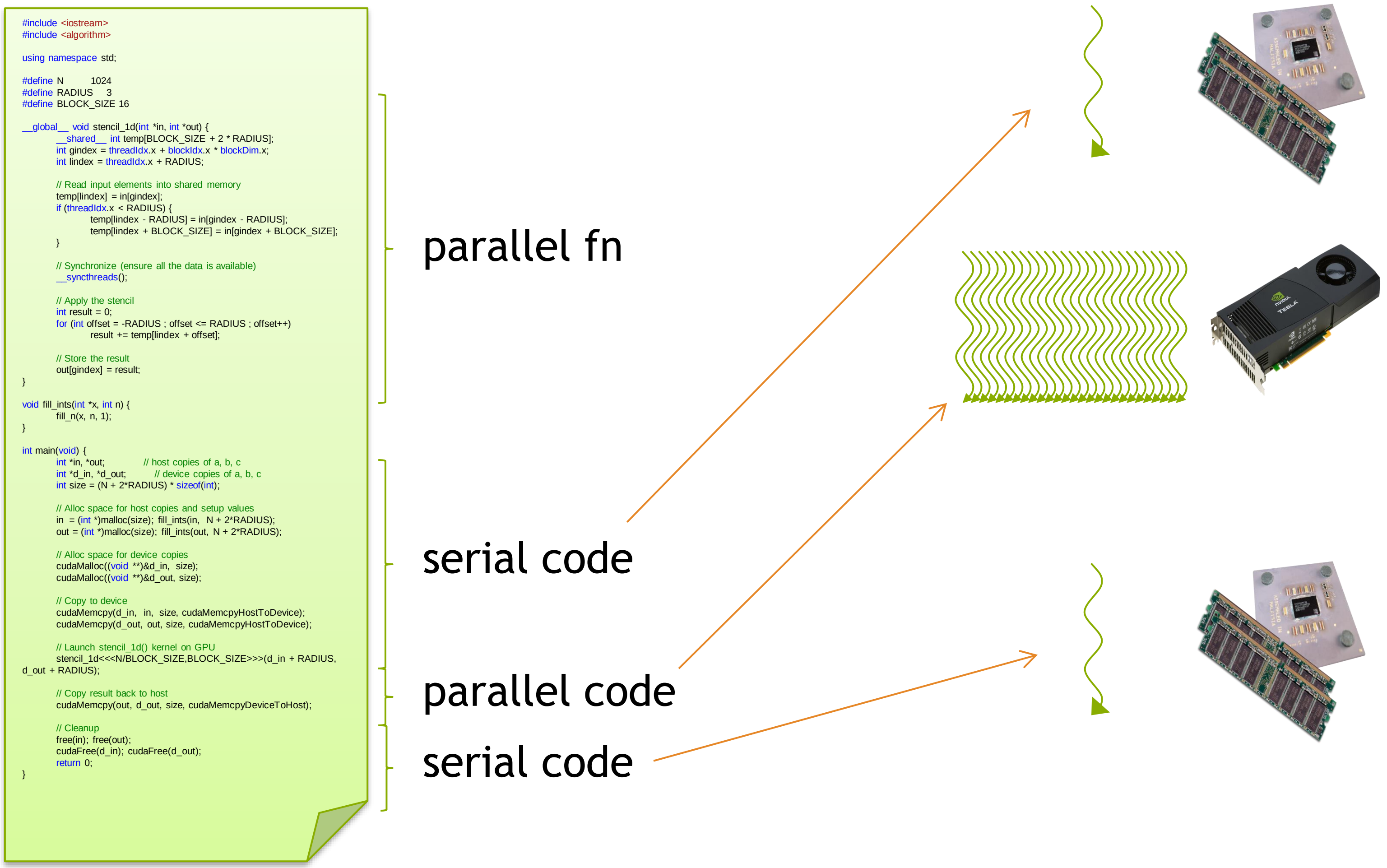
- Most GPUs use a ***scoped*** memory consistency model
  - Only apply GPU fences locally if all users are local – less overhead
  - But more work for programmer

# Accelerator Model of Computing

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

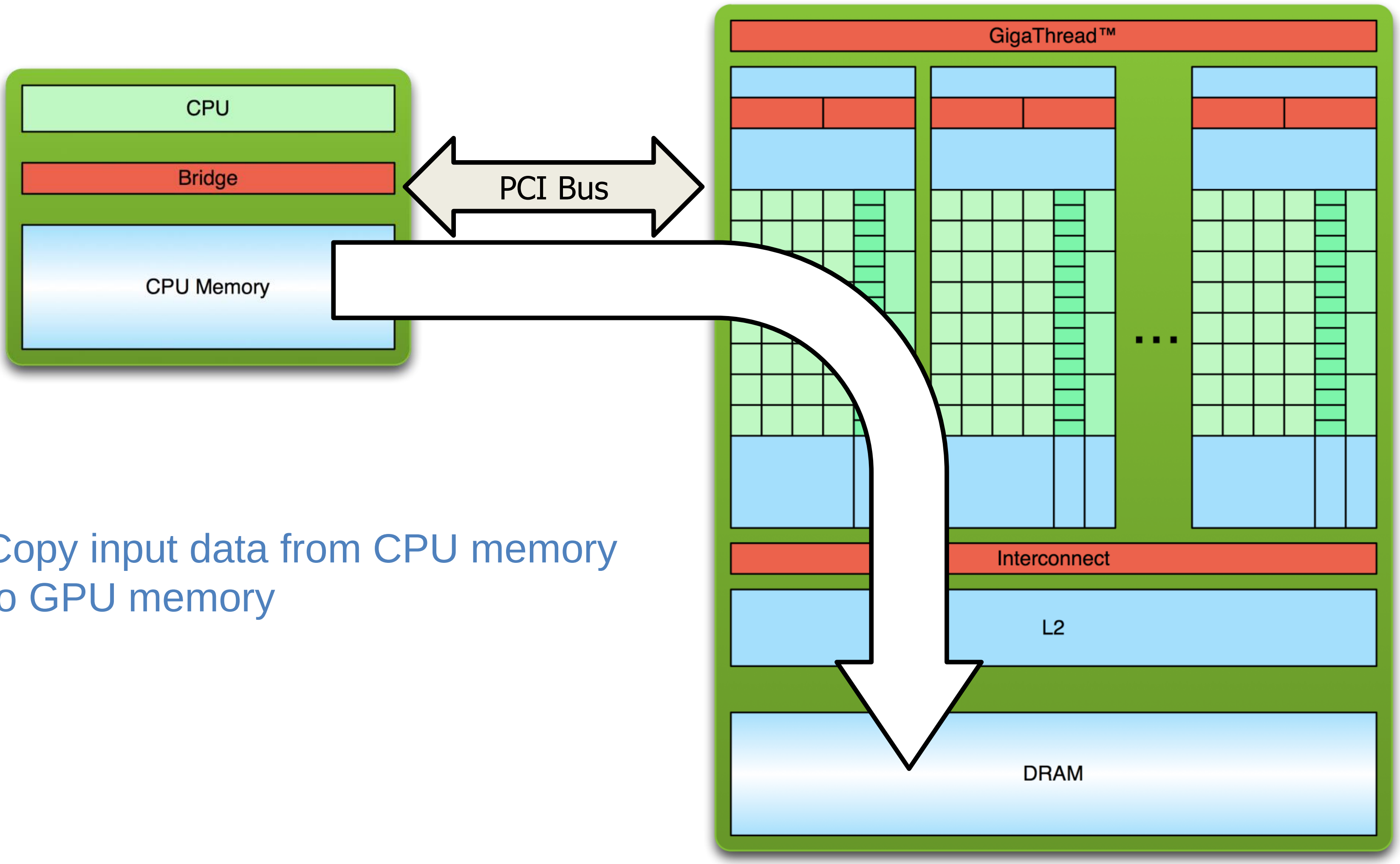


# Heterogeneous Computing



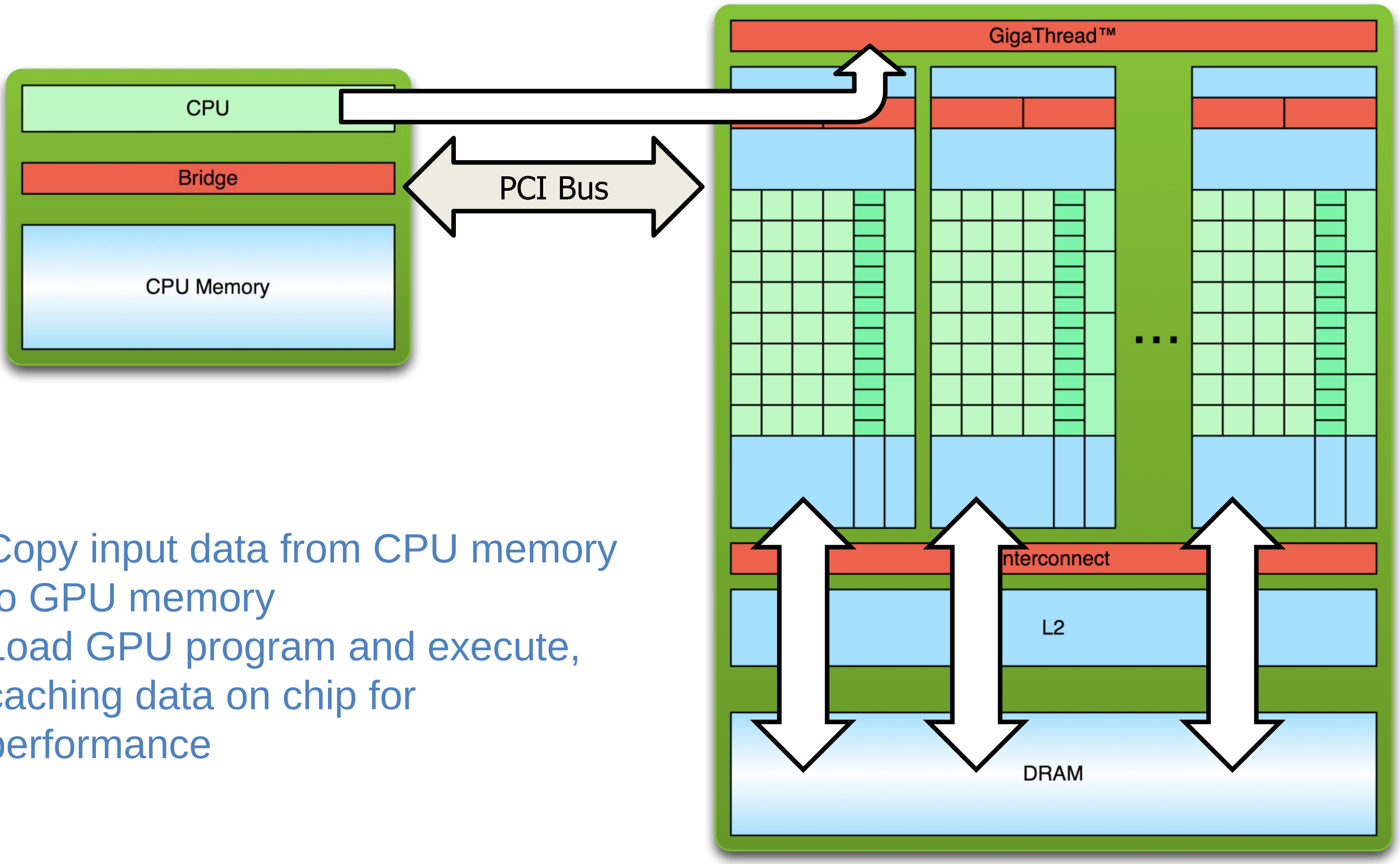
```
#include <iostream>
#include <algorithm>

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



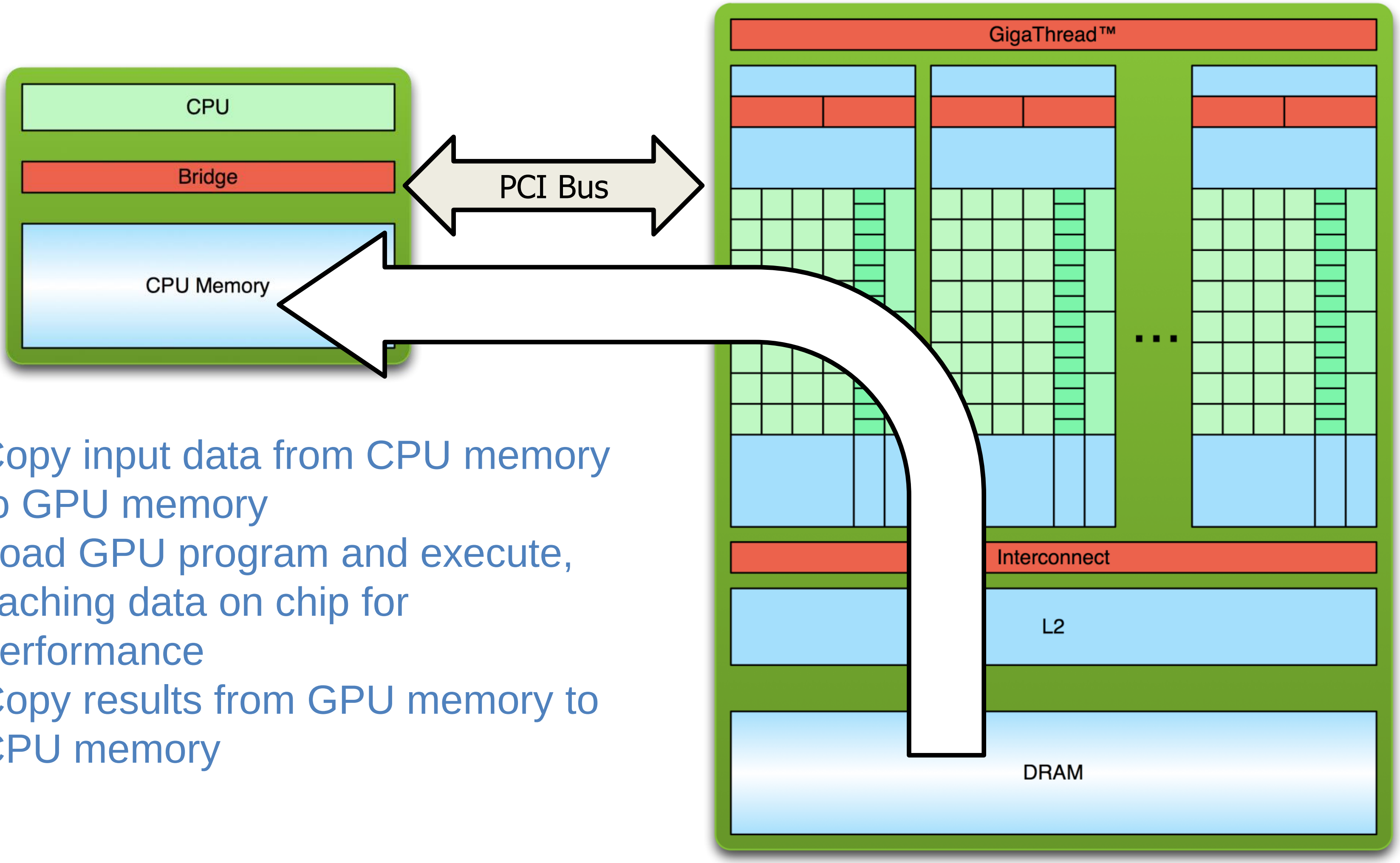
1. Copy input data from CPU memory to GPU memory

# Simple Processing Flow



1. Copy input data from CPU memory to GPU memory
2. Load GPU 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

# (non-unified) Memory Management

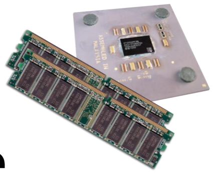
- Old model: Explicit memory management
- Host and device memory are separate entities
  - *Device* pointers point to GPU memory

    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

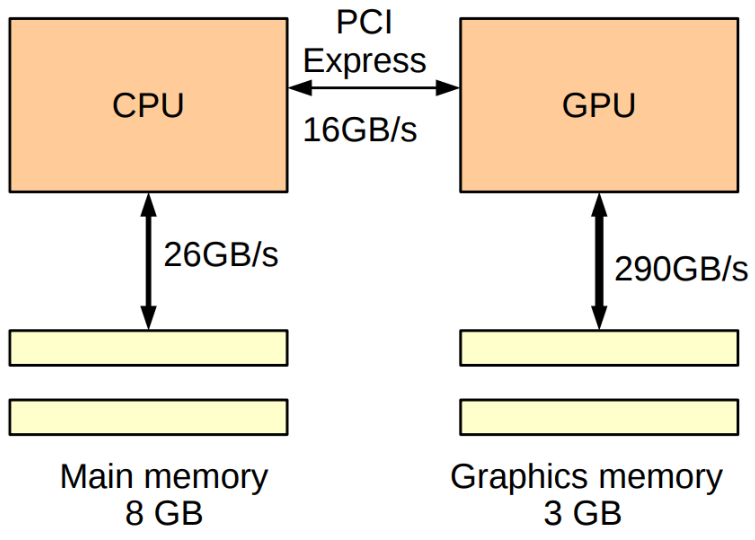
    May *not* be dereferenced in device code



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

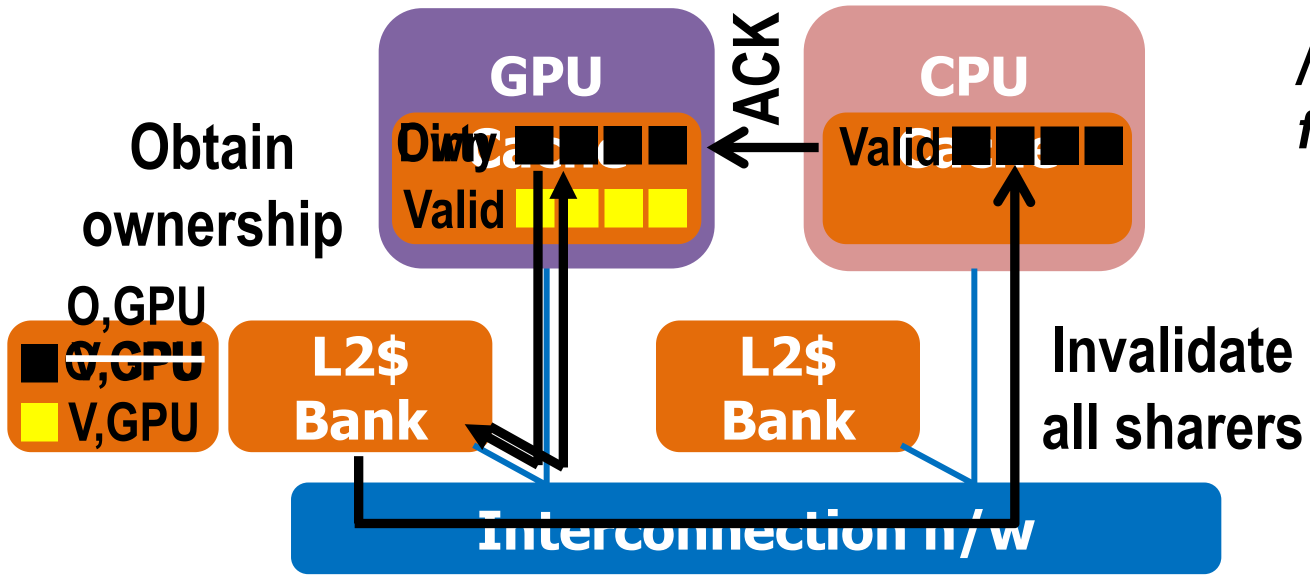
## Consequences of CPU/GPU Integration

- Software Managed Access:
  - cuda_malloc – allocate device memory
  - cuda_memcpy – transfer data back and forth
- Unified Memory:
  - cudaMallocManaged -- allocate transparently managed memory
  - Page-faults cause data transfer
  - GPU transfers CPU-GPU memory as it accesses it



Ex: Intel Core i7 4770, Nvidia GeForce GTX 780

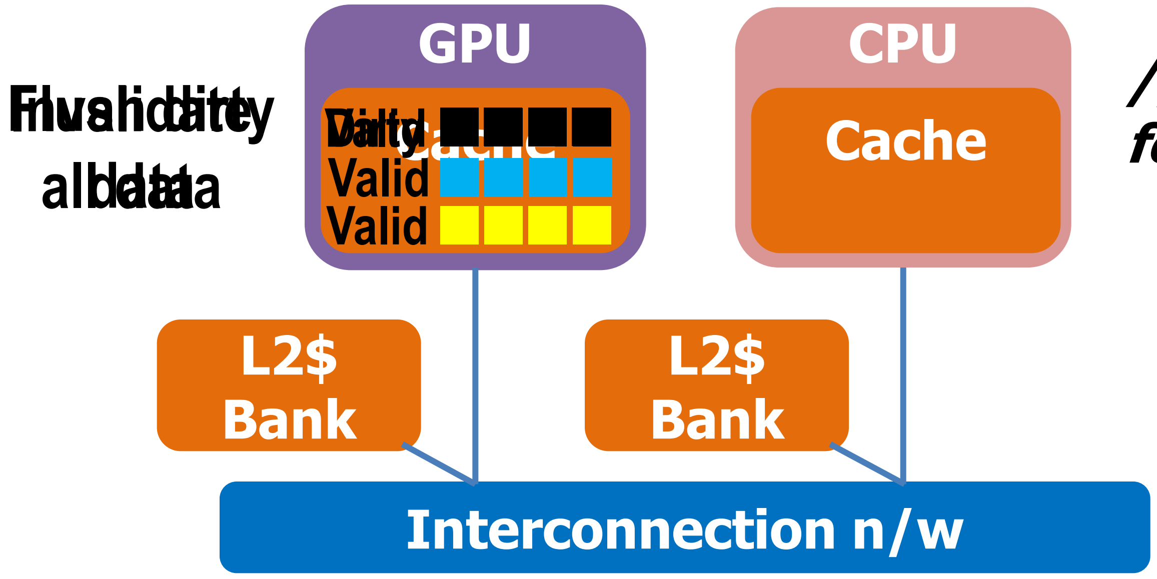
# CPU Coherence: MESI



```
// each thread
for i = r[tid]:r[tid+1]
    LOCK
    LD R1, A[i]; 🟨
    LD R2, B[i]; ■
    R3← Math(R1, R2);
    ST B[i], R3;
    UNLOCK
```

- Write miss: Get ownership, invalidate all sharers
- Read miss: Update sharers list
- Synchronization points are cheap
- BUT poor fit for GPUs:
    - Directory overhead, transient states, excessive traffic, indirection

# Traditional GPU Coherence



```
// unique per thread
for i = start:end
    LD R1, A[i];          🟨
    LD R2, B[i];          🟦
    R3← Math(R1, R2);
    ST C[i], R3;          ⬛
```

Each thread accesses independent data (no races)

No data reuse or data sharing

Coarse-grained synchronization

Optimized for streaming, data parallel applications

# Scalar SAXPY Performance

```
ldf X(r1),f1
mulf f0,f1,f2
ldf Y(r1),f3
addf f2,f3,f4
stf f4,Z(r1)
addi r1,4,r1
slti r1,400,r2
bne Loop
```

- Scalar version
  - 5-cycle **`mulf`**, 2-cycle **`addf`**, 1 cycle others
  - 100 iters * 11 cycles/iter = 1100 cycles

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| `ldf X(r1),f1` | F | D | X | M | W | | | | | | | | | | | | | | |
| `mulf f0,f1,f2` | | F | D | d* | E* | E* | E* | E* | E* | W | | | | | | | | | |
| `ldf Y(r1),f3` | | | F | p* | D | X | M | W | | | | | | | | | | | |
| `addf f2,f3,f4` | | | | | F | D | d* | d* | d* | E+ | E+ | W | | | | | | | |
| `stf f4,Z(r1)` | | | | | | F | p* | p* | p* | D | X | M | W | | | | | | |
| `addi r1,4,r1` | | | | | | | | | | F | D | X | M | W | | | | | |
| `blt r1,r2,0` | | | | | | | | | | | F | D | X | M | W | | | | |
| `ldf X(r1),f1` | | | | | | | | | | | | F | D | X | M | W | | | |

# Vector SAXPY Performance

```
ldf.v X(r1),v1
mulf.vs v1,f0,v2
ldf.v Y(r1),v3
addf.vv v2,v3,v4
stf.v v4,Z(r1)
addi r1,16,r1
slti r1,400,r2
bne r2,Loop
```

- Vector version
  - 4 element vectors
  - 25 iters * 11 insns/iteration * = 275 cycles
  - + Factor of 4 speedup

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| `ldf.v X(r1),v1` | F | D | X | M | W | | | | | | | | | | | | | | |
| `mulf.vv v1,f0,v2` | | F | D | d* | E* | E* | E* | E* | E* | W | | | | | | | | | |
| `ldf.v Y(r1),v3` | | | F | p* | D | X | M | W | | | | | | | | | | | |
| `addf.vv v2,v3,v4` | | | | | F | D | d* | d* | d* | E+ | E+ | W | | | | | | | |
| `stf.v f4,Z(r1)` | | | | | | F | p* | p* | p* | D | X | M | W | | | | | | |
| `addi r1,4,r1` | | | | | | | | | | F | D | X | M | W | | | | | |
| `blt r1,r2,0` | | | | | | | | | | | F | D | X | M | W | | | | |
| `ldf X(r1),f1` | | | | | | | | | | | | F | D | X | M | W | | | |

# Not So Fast

- A processor with 32-element vectors
  - 1 Kb (32 * 32) to cache? 32 FP multipliers?
- No: vector load/store/arithmetic units are **pipelined**
  - Processors have L (1 or 2) of each type of functional unit
    - L is called number of vector **lanes**
  - Micro-code streams vectors through units M data elements at once
- Pipelined vector insn timing
  - $T_{vector} = T_{scalar} + (MVL / L) – 1$
  - Example: 64-element vectors, 10-cycle multiply, 2 lanes
  - $T_{mulf.vv} = 10 + (64 / 2) – 1 = 41$

  + Not bad for a loop with 64 10-cycle multiplies

# Pipelined Vector SAXPY Performance

```
ldf.v X(r1),v1
mulf.vs v1,f0,v2
ldf.v Y(r1),v3
addf.vv v2,v3,v4
stf.v v4,Z(r1)
addi r1,16,r1
slti r1,400,r2
bne r2,Loop
```

- Vector version
  - 4-element vectors, 1 lane
  - 4-cycle **`ldf.v/stf.v`**
  - 8-cycle **`mulf.sv`**, 5-cycle **`addf.vv`**
  - 25 iters * 20 cycles/iter = 500 cycles
  - Factor of 2.2 speedup

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| `ldf.v X(r1),v1` | F | D | X | **M** | M | M | M | W | | | | | | | | | | | |
| `mulf.sv v1,f0,v2` | | F | D | d* | d* | d* | d* | **E*** | **E*** | **E*** | **E*** | **E*** | E* | E* | E* | W | | | |
| `ldf.v Y(r1),v3` | | | F | p* | p* | p* | p* | D | X | **M** | M | M | M | W | | | | | |
| `addf.vv v2,v3,v4` | | | | | | | | F | D | d* | d* | d* | d* | d* | d* | **E+** | **E+** | E+ | E+ |
| `stf.v f4,Z(r1)` | | | | | | | | | F | p* | p* | p* | p* | p* | p* | D | X | d* | d* |
| `addi r1,4,r1` | | | | | | | | | | | | | | | | F | D | p* | p* |
| `blt r1,r2,0` | | | | | | | | | | | | | | | | | F | p* | p* |
| `ldf.v X(r1),f1` | | | | | | | | | | | | | | | | | | | |

## Not So Slow

- For a given vector operation
  - All MVL results complete after $T_{scalar} + (MVL / L) - 1$
  - First M results (e.g., `v1[0]` and `v1[1]`) ready after $T_{scalar}$
  - Start dependent vector operation as soon as those are ready

- **Chaining**: pipelined vector forwarding
  - $T_{vector1} = T_{scalar1} + (MVL / L) - 1$
  - $T_{vector2} = T_{scalar2} + (MVL / L) - 1$
  - $T_{vector1} + T_{vector2} = T_{scalar1} + T_{scalar2} + (MVL / L) - 1$

# Chained Vector SAXPY Performance

```
ldf.v X(r1),v1
mulf.vs v1,f0,v2
ldf.v Y(r1),v3
addf.vv v2,v3,v4
stf.v v4,Z(r1)
addi r1,16,r1
slti r1,400,r2
bne r2,Loop
```

- Vector version
  - 1 lane
  - 4-cycle **ldf.v/stf.v**
  - 8-cycle **mulf.sv**, 5-cycle **addf.vv**
  - 25 iters * 11 cycles/iter = 275 cycles
  - + Factor of 4 speedup again

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| ldf.v X(r1),v1 | F | D | X | **M** | M | M | M | W | | | | | | | | | | | |
| mulf.vv v1,f0,v2 | | F | D | d* | **E*** | **E*** | **E*** | **E*** | **E*** | E* | E* | E* | W | | | | | | |
| ldf.v Y(r1),v3 | | | F | p* | D | X | s* | **M** | M | M | M | W | | | | | | | |
| addf.vv v2,v3,v4 | | | | | F | D | p* | d* | d* | **E+** | **E+** | E+ | E+ | E+ | W | | | | |
| stf.v f4,Z(r1) | | | | | | F | p* | p* | p* | D | X | **M** | M | M | M | W | | | |
| addi r1,4,r1 | | | | | | | | | | F | D | X | M | W | | | | | |
| blt r1,r2,0 | | | | | | | | | | | F | D | X | M | W | | | | |
| ldf.v X(r1),f1 | | | | | | | | | | | | F | D | X | M | M | M | M | W |

# Vector Performance

- Where does it come from?
  - + Fewer loop control insns: **`addi`**, **`blt`**, etc.
    - Vector insns contain implicit loop control
  - + RAW stalls taken only once, on "first iteration"
    - Vector pipelines hide stalls of "subsequent iterations"

- How does it change with vector length?
  - + Theoretically increases, think of $T_{vector}/MVL$
    - $T_{vector} = T_{scalar} + (MVL / L) - 1$
    - $MVL = 1 \rightarrow (T_{vector}/MVL) = T_{scalar}$
    - $MVL = 1000 \rightarrow (T_{vector}/MVL) = 1$
  - – But vector regfile becomes larger and slower

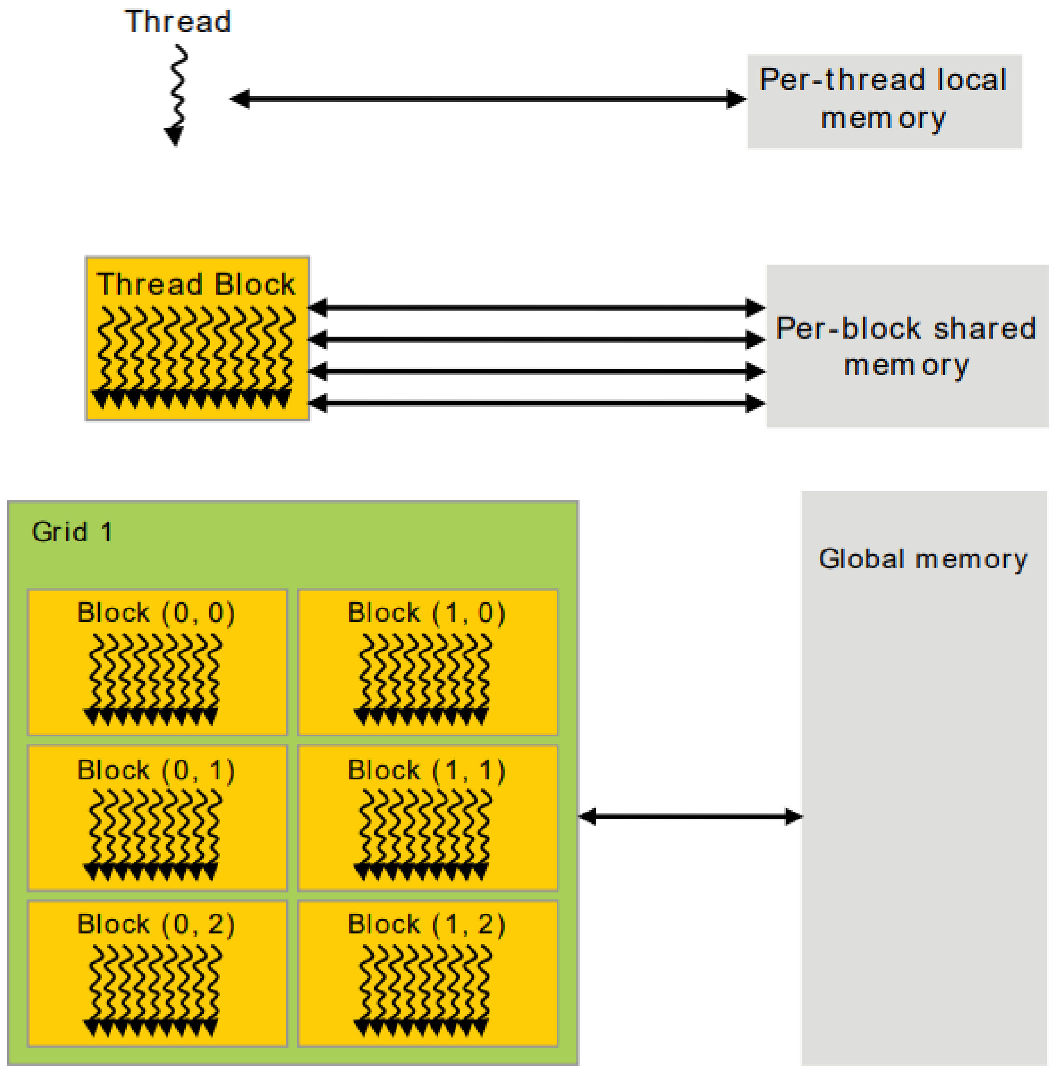
# Amdahl's Law

- **Amdahl's law**: the law of diminishing returns
  - $\text{speedup}_{\text{total}} = 1 / [\%_{\text{vector}} / \text{speedup}_{\text{vector}} + (1–\%_{\text{vector}})]$
  - Speedup due to vectorization limited by **non-vector portion**
  - In general: optimization speedup limited by unoptimized portion

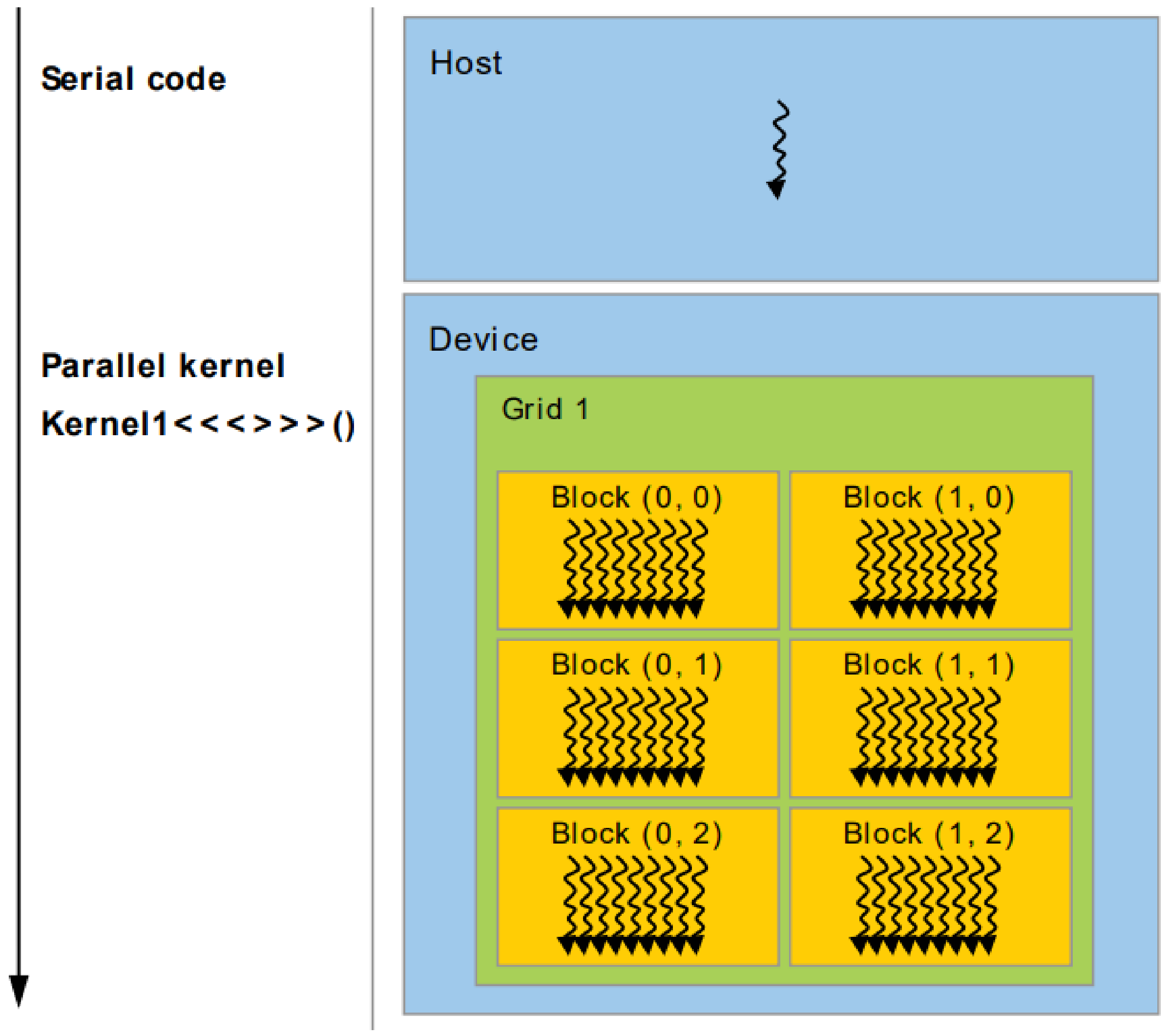
  - Example: $\%_{\text{opt}} = 90\%$
    - $\text{speedup}_{\text{opt}} = 10 \rightarrow \text{speedup}_{\text{total}} = 1 / [0.9/10 + 0.1] = 5.3$
    - $\text{speedup}_{\text{opt}} = 100 \rightarrow \text{speedup}_{\text{total}} = 1 / [0.9/100 + 0.1] = 9.1$
    - $\text{Speedup}_{\text{opt}} = \infty \rightarrow \text{speedup}_{\text{total}} = 1 / [0.9/\infty + 0.1] = 10$

  - CRAY-1 rocked because it had fastest vector unit …
  - … and the fastest scalar unit

# CUDA Threading / Memory Abstractions



# GPU Execution Model



Serial code
Parallel kernel
Kernel1<<<>>>()
Host
Device
Grid 1
Block (0, 0)
Block (1, 0)
Block (0, 1)
Block (1, 1)
Block (0, 2)
Block (1, 2)