Running (AMD) GPU experiments in gem5

Matthew D. Sinclair

University of Wisconsin-Madison, AMD Research

sinclair@cs.wisc.edu

Disclaimers

- #1: Currently gem5 only supports AMD GPUs
  - The concepts are similar to NVIDIA GPUs though

- #2: Currently gem5 only supports GPGPU workloads (no Vulkan, OpenGL)

Contributors

- AMD Research: Brad Beckmann, Alex Dutu, Tony Gutierrez, Michale LeBeane, Matthew Poremba, Brandon Potter, Sooraj Puthoor, & many more
- UW-Madison: Anushka Chandrashekar, Gaurav Jain, Charles Jamieson, Jing Li, Kyle Roarty, Mingyuan Xiang, Bobbi Yogatama, & others

- Some slides based on content presented by these folks previously

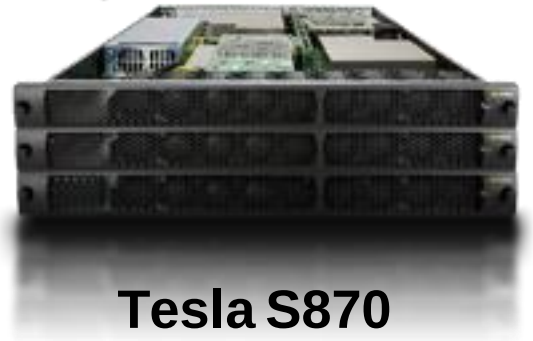
Compiling gem5 GPU Model

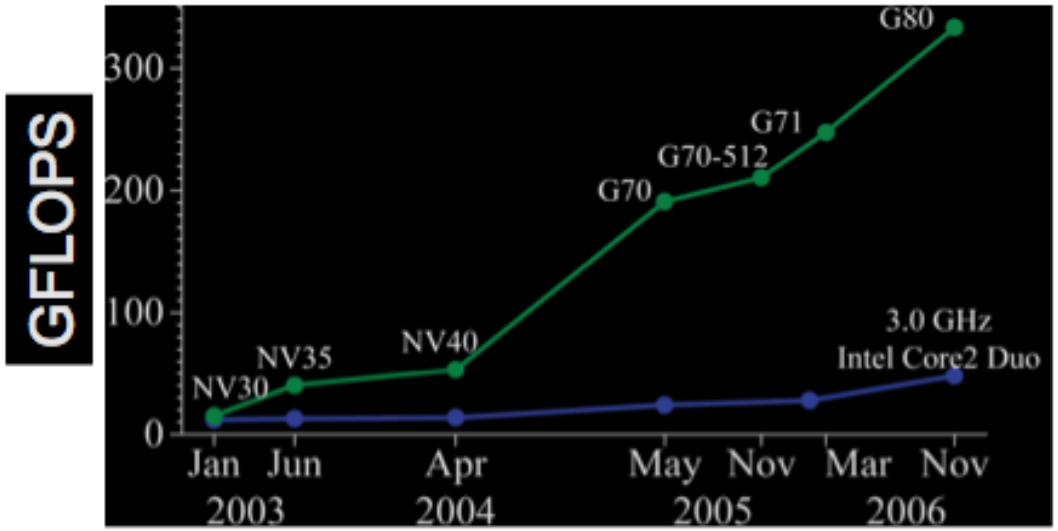
```
docker pull gcr.io/gem5-test/gcn-gpu:v22-0
cd gem5
docker run --rm --volume /var/lib/docker/codespacemount/workspace/:/workspaces -w `pwd` gcr.io/gem5-test/gcn-gpu:v22-0 scons build/GCN3_X86/gem5.opt –j17
```

- This will take ~20 minutes to compile – we'll come back to them
  - Commands also in 11-gpu/README.md

Graphics Processing Units (GPU)

- Killer app for parallelism: graphics (3D games)
- A quiet revolution and potential build-up
  - Calculation: 367 GFLOPS vs. 32 GFLOPS
  - Memory Bandwidth: 86.4 GB/s vs. 8.4 GB/s
  - Until recently, programmed through graphics API



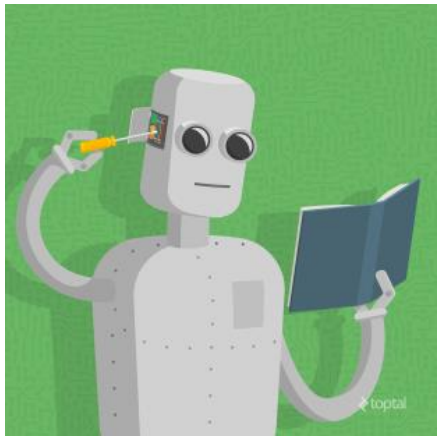


G80 = GeForce 8800 GTX
G71 = GeForce 7900 GTX
G70 = GeForce 7800 GTX
NV40 = GeForce 6800 Ultra
NV35 = GeForce FX 5950 Ultra
NV30 = GeForce FX 5800

  - GPU in every desktop, laptop, mobile device – massive volume and potential impact

GPU Evolution

- New killer app: Machine Learning
- … and crypto



Disclaimer: this talk will not teach you how to run crypto in gem5

Learning Outcomes

- By the end of this class attendees will be able to:
  - Understand the basics of GPU architecture and programming.
  - Understand the basics of how (AMD) GPUs are implemented in gem5.
  - Compile the gem5 GPU model (and describe how and why docker support is provided).
  - Run basic GPU tests on the (AMD) GPU model.
  - Compare and contrast the results of different register allocation schemes.
  - Identify what additional resources gem5-resources provides.

Outline

- Background: GPU Architecture & Programming Basics (20-30 minutes)
- Modeling & Using GPUs in gem5 (1 hour)
- Running GPU programs in gem5 (1 hour)

Flynn's Taxonomy

- 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)

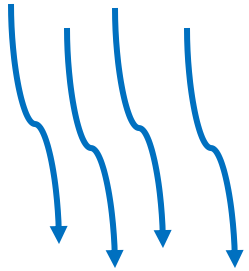
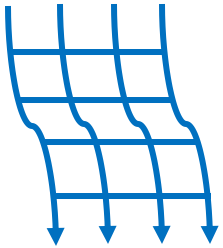
- SIMD (Single Instruction, Multiple Data):
  - Split identical, independent work over multiple execution units
  - More efficient: eliminate redundant fetch/decode vs. SPMD/MIMD
  - Use single PC and single register file

Flynn's Taxonomy (Cont.)

- SIMD's cousin: SIMT (Single Instruction, Multiple Thread)
  - Split identical, independent work over multiple lockstep 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

Execution Model Comparison

	MIMD/SPMD	SIMD/Vector	SIMT
			
Example	Multicore CPUs	x86 SSE/AVX	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

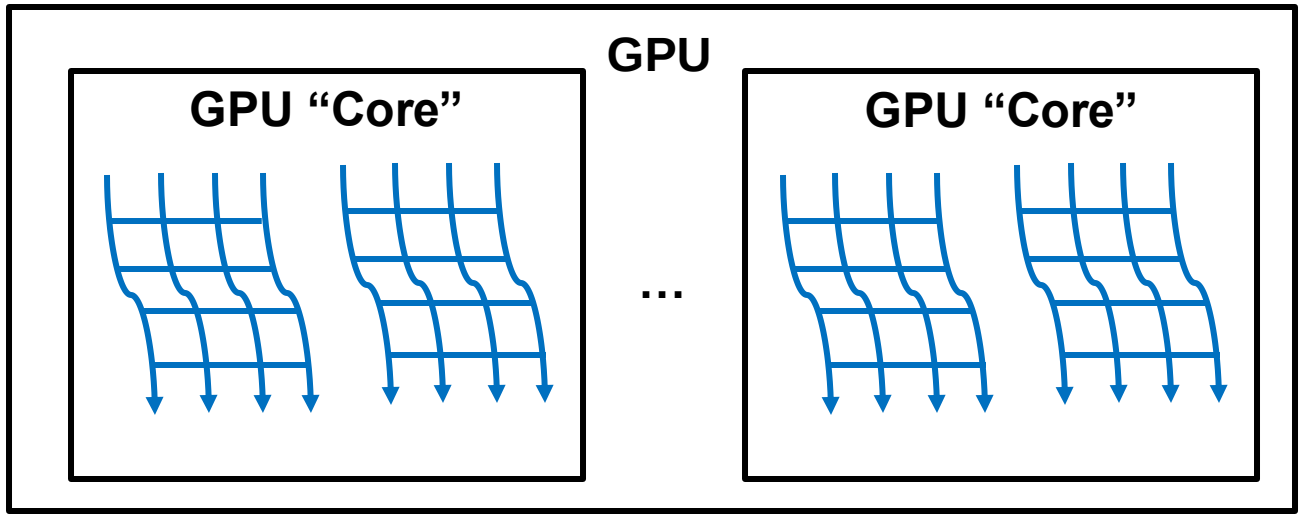
- GPUs optimized for streaming computations
  - Thus, we have a lot of streaming memory accesses

- DRAM: 100's of GPU cycles per memory access
  - How to hide this overhead & keep the GPU busy in the meantime?

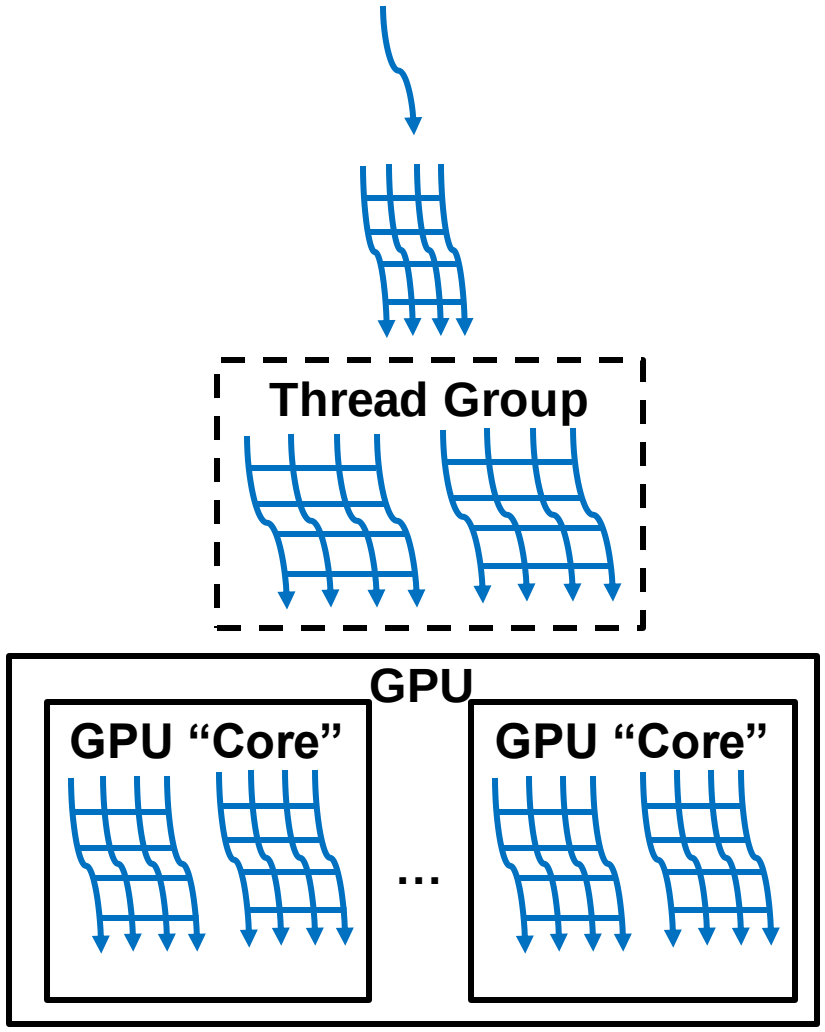
- Traditional CPU approaches:
  - Caches → Need spatial/temporal locality X
    - Streaming applications have little reuse
  - OOO/Dynamic Scheduling → Need ILP X
    - Too power hungry, diminishing returns for GPU applications
  - Multicore/Multithreading/SMT → need independent threads ✓

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

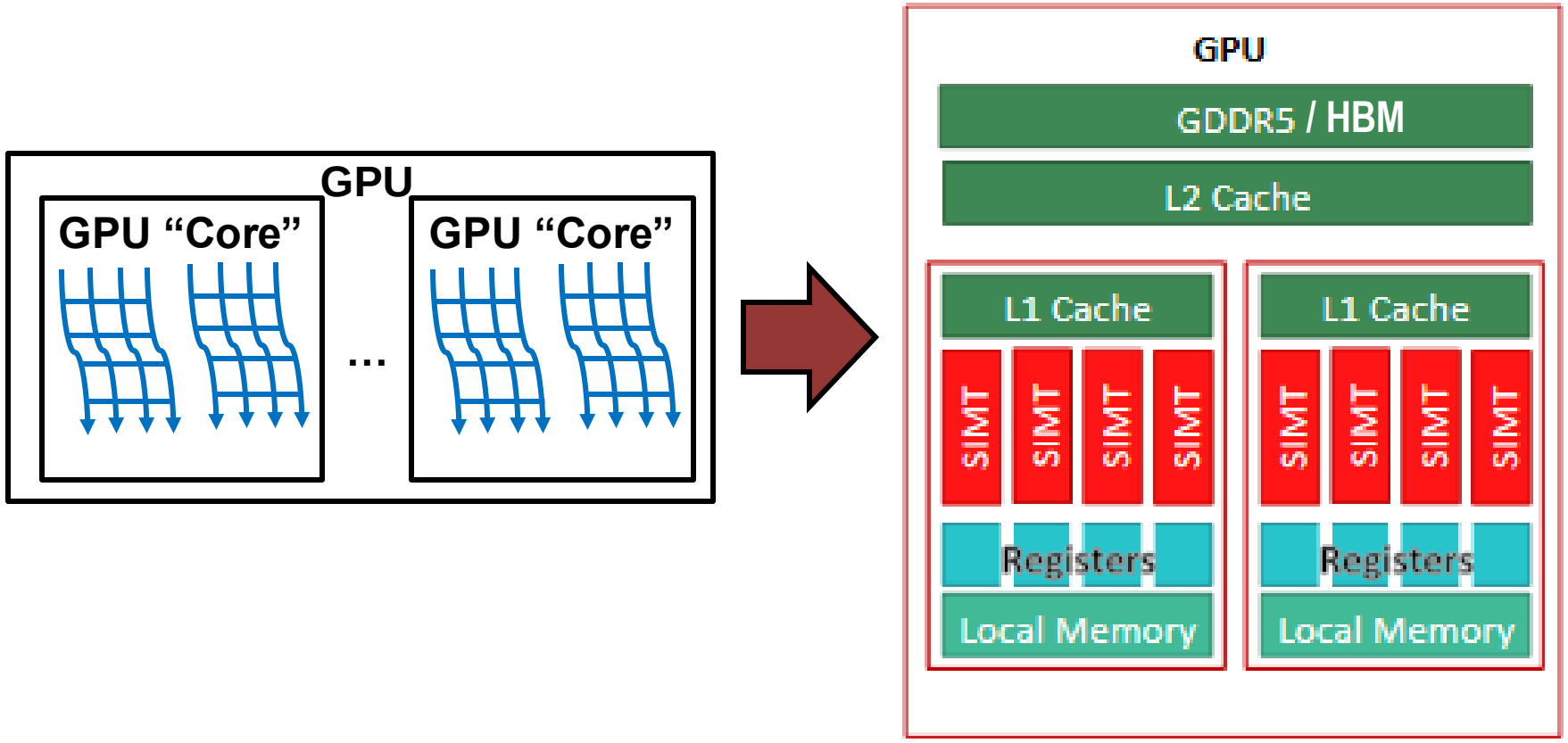


CUDA/HIP	OpenCL
Thread	Work-item
Warp	Wavefront
Thread Block/CTA	Workgroup
Grid (Kernel)	NDRange (Kernel)

Programming GPUs

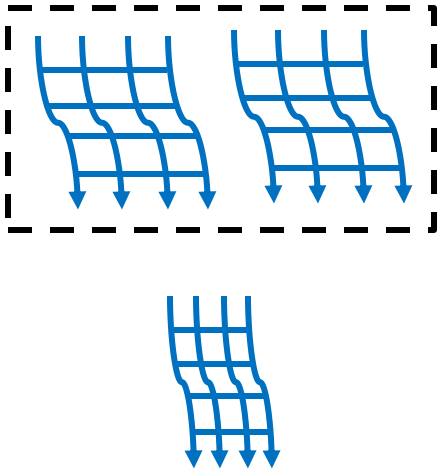
- Program it with CUDA, HIP, or OpenCL
  - CUDA = Compute Unified Device Architecture
    - NVIDIA's proprietary solution
  - OpenCL = Open Computing Language
    - Open, industrywide standard
  - HIP = Heterogeneous interface for portability
    - AMD's open solution, its successor to OpenCL
    - OpenCL partially supported inside HIP kernels
  - All: 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)

GPU Hardware Overview

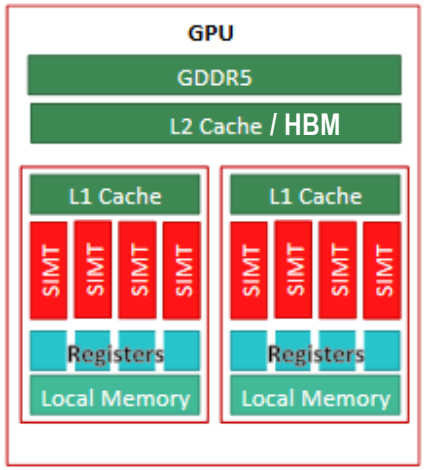


Compute Unit (CU) – The GPU "Core"

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



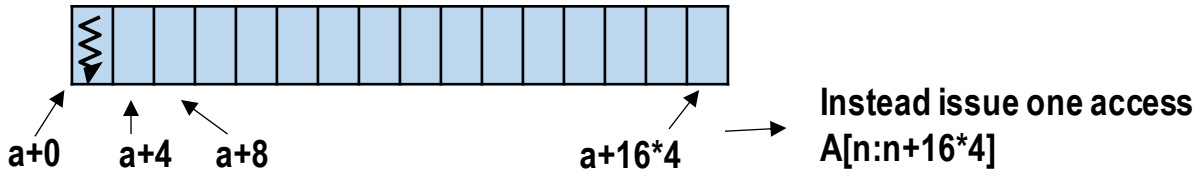
- SIMT unit: runs wavefronts/warps
  - Run the threads
  - AMD: size N (e.g., 10) wavefront instruction buffer
    - 4 cycles to execute one wavefront
    - Average: fetch and commit 1 wavefront/cycle

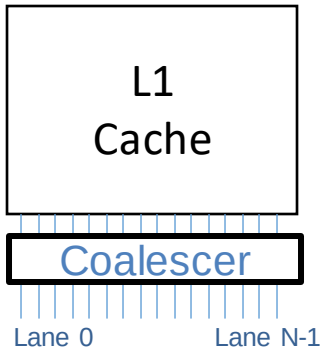


How do we do efficient memory access?

- Pseudo CUDA for contiguous access:

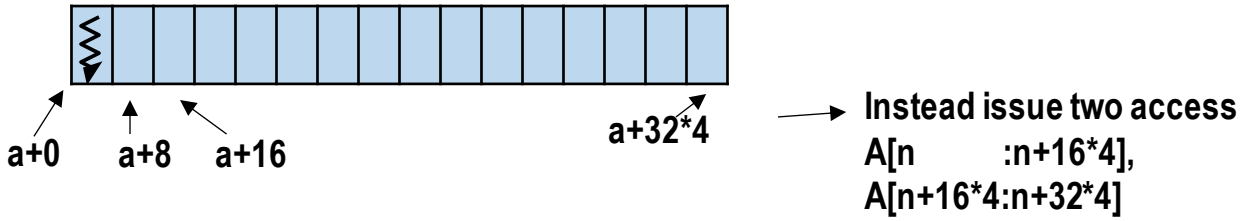
```
gpu void add(int *a, int *b, int *c) {
        c[tid] = a[tid] + b[tid];
}
```





- Pseudo CUDA for non-contiguous access:

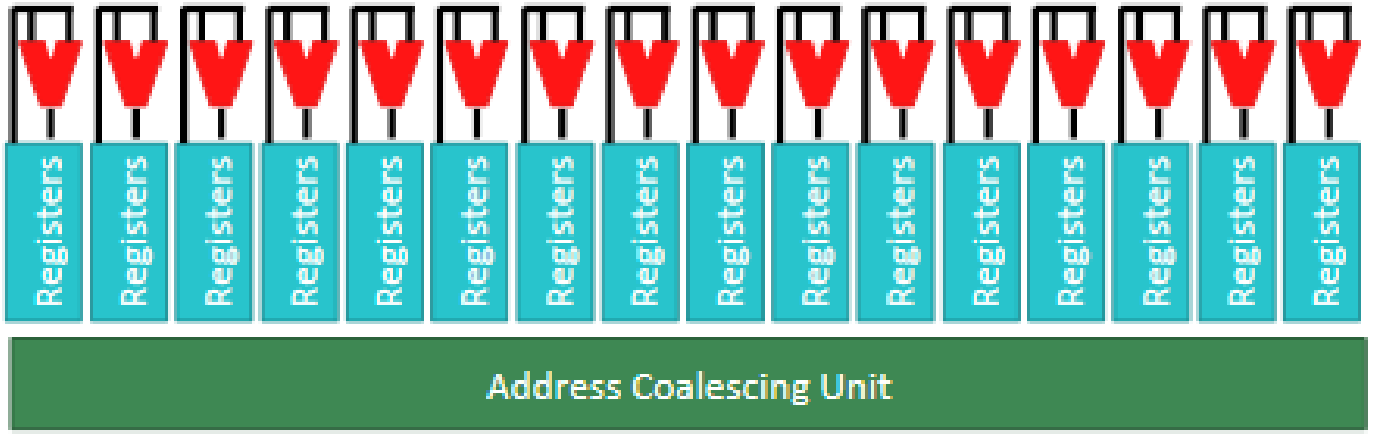
```
gpu void add(int *a, int *b, int *c) {
        c[tid] = a[tid*2] + b[tid];
}
```



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

How many ports should my L1 have?

- Warp: 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 wavefronts (often size 64 threads)
- Common case:
  - All threads in warp/wavefront access same cache block(s)
- Addressing coalescing:
  - Dynamically combine addresses generated from each lane



  - Reduces in-flight memory requests, helps DRAM b/w, **important**

git remote add upstream https://github.com/gem5bootcamp/gem5-bootcamp-env

git fetch upstream

git reset –hard upstream/main

rebuild container

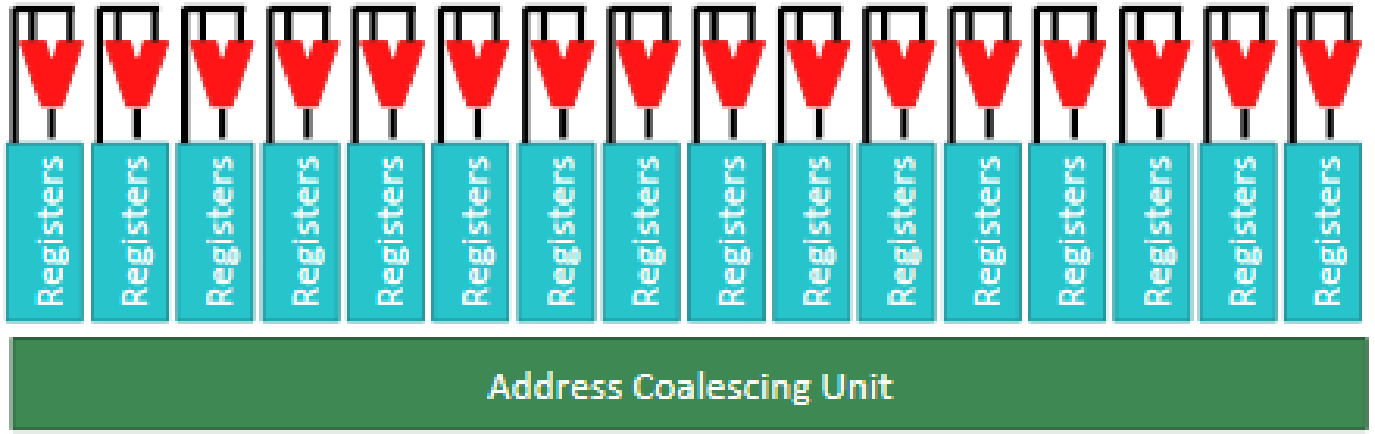
docker pull gcr.io/gem5-test/gcn-gpu:v22-0

cd gem5

docker run --rm --volume /var/lib/docker/codespacemount/workspace/:/workspaces -w `pwd` gcr.io/gem5-test/gcn-gpu:v22-0 scons build/GCN3_X86/gem5.opt –j17

SIMT Unit – A GPU Pipeline

- Similar to a wide CPU pipeline, except only fetch 1 instr.
- 16-wide physical ALU – why not 64?
- 64 KB register state/SIMD unit
  - Much bigger (~64X) than CPUs – why?
- Addressing coalescing key to good performance
  - Each thread potentially fetches a different piece of data
  - 64 separate addresses for AMD, 32 for NVIDIA (tradeoffs)



Address Coalescing

- 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**

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?

Caches to the Rescue?

- Comparison: Modern CPU and GPU caches

	CPU	GPU
L1 D$ capacity	64 KB	32 KB
Active threads/work-items sharing L1 D$	2	2560
L1 D$ capacity/thread	32 KB	12.8 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

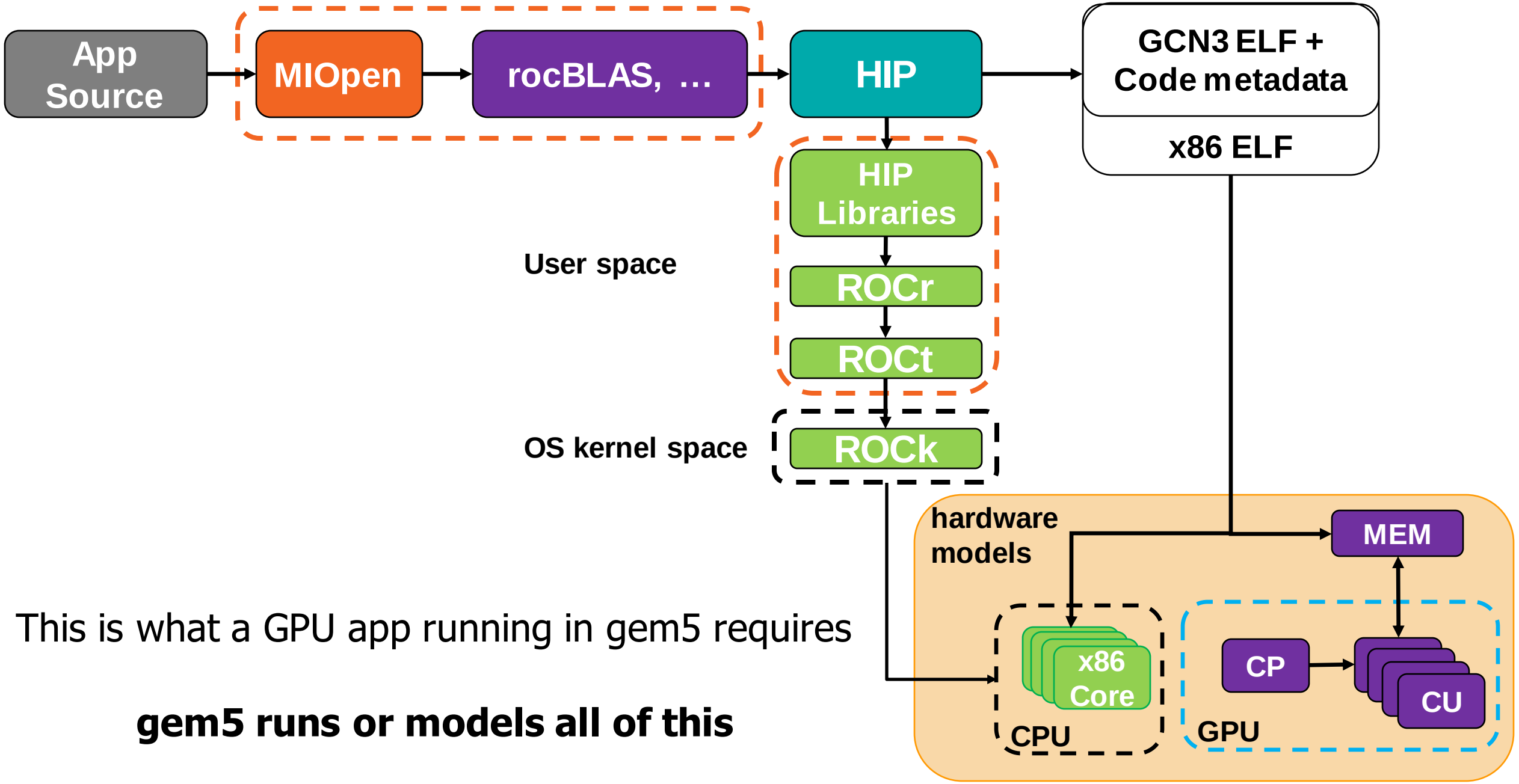
- L1 cache:
  - Coalesce requests to same cache block by different threads
  - Keep around long enough for all threads in warp/wavefront to hit
    - Once
  - Ultimate goal: **reduce number of requests sent to DRAM**

- 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

Outline

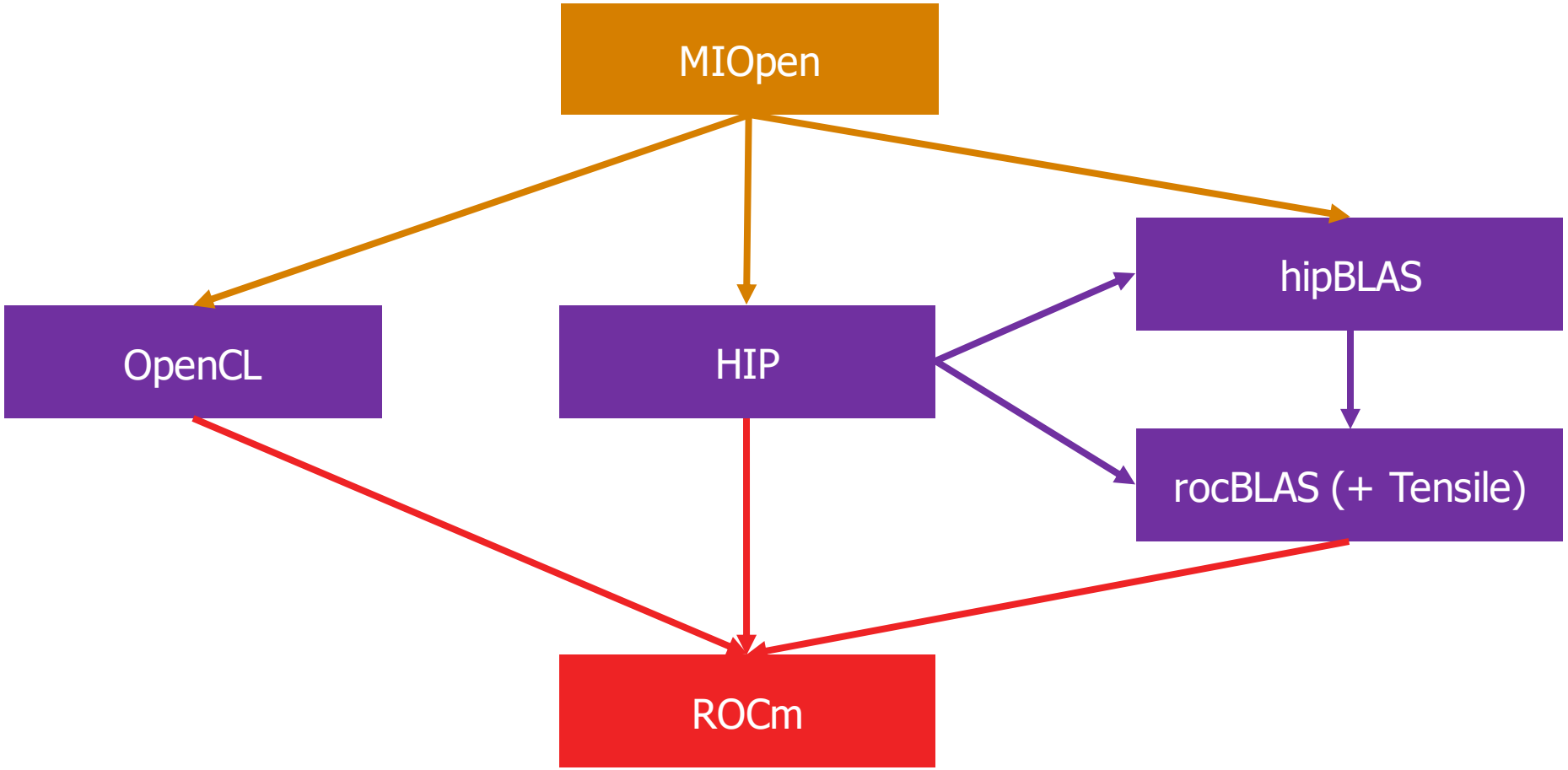
- Background: GPU Architecture & Programming Basics (20-30 minutes)
- **Modeling & Using GPUs in gem5 (1 hour)**
  - **What libraries are required?**
  - What support is provided?
  - Where is GPU code?
  - How to compile GPU model in gem5?
- Running GPU programs in gem5 (1 hour)



This is what a GPU app running in gem5 requires

gem5 runs or models all of this

Alternate View



Getting all of this installed correctly can be difficult!

AMD's ROCm Stack

- ROCm == Radeon Open Compute
- ROCm stack
  - Runtime layer – ROCr
  - Thunk (user-space driver) – ROCt
  - Kernel fusion driver (KFD) – ROCk
  - MIOpen – machine intelligence (ML) library
  - rocBLAS – BLAS (e.g., GEMMs) library
  - HIP – GPU programming language (roughly: LLVM backend, clang front-end)
  - …
- gem5 simulates all of these except ROCk, which it emulates in SE mode

Creating Portable gem5 Resources

- Docker container
  - Properly installs ROCm software stack

- **Publicly Available!**
  - Integrated into gem5 repo: https://gem5.googlesource.com/
  - Added bmks & doc. in gem5-resources *[Bruce ISPASS '20 Best Paper Nom.]*
  - Used in continuous integration to ensure GPU support is stable
  - Strongly suggest building applications requiring ROCm with docker

- **All of our experiments today will assume this docker support**
  - docker pull gcr.io/gem5-test/gcn-gpu:v22-0 ← For gem5 v22.0

gem5 GPU docker

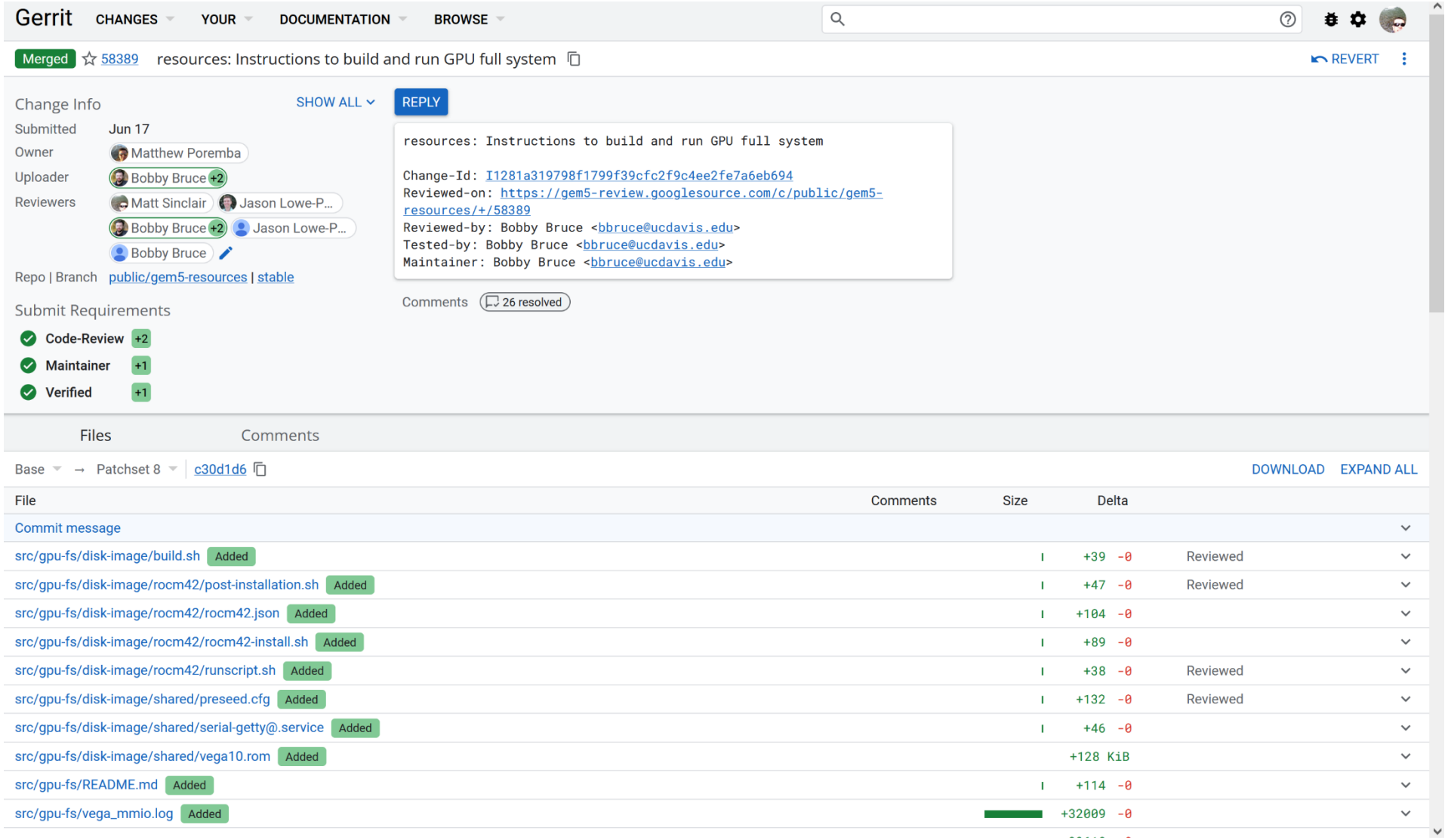
Outline

- Background: GPU Architecture & Programming Basics (20-30 minutes)
- **Modeling & Using GPUs in gem5 (1 hour)**
  - What libraries are required?
  - **What support is provided?**
  - Where is GPU code?
  - How to compile GPU model in gem5?
- Running GPU programs in gem5 (1 hour)

Current Support

- ROCm supported in gem5: ROCm v4.0
- SE mode vs. FS mode:
  - SE mode is well supported on stable – **today's focus**
  - FS mode was just released on develop with 22.0, but won't discuss today
- AMD GPU support
  - GCN3 (gfx801 – APU, gfx803 – dGPU)
  - Vega (gfx900 – dGPU, gfx902 – APU, partial support)
  - Vega is newer model than GCN3
  - If you want to run on the VEGA model in gem5, you need to compile for the appropriate gfx9* model
- Standard library: currently not supported – use apu_se.py and gpufs.py instead
- Currently only supports Ruby
- **Today we will focus on GCN3 and gfx801, because they're most tested**

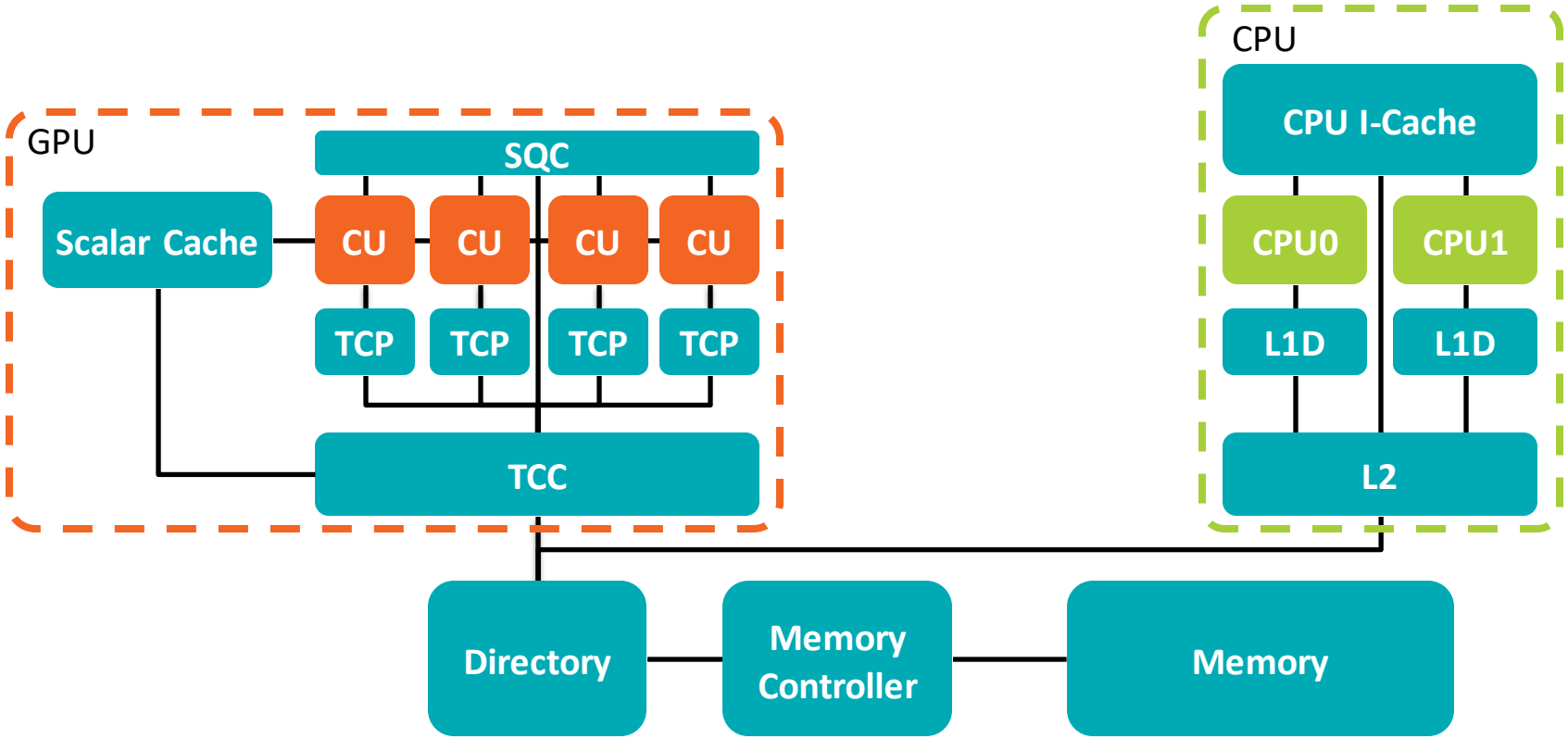
GPUFS Support



Simulates all driver calls + able to support newer ROCm versions “out of the box”

APU vs. dGPU

- APU = CPU+GPU have a single, unified address space
- dGPU = CPU and GPU have separate, discrete address spaces
- *Sidenote: SQC = GPU L1 I$, TCP = GPU L1 D$, TCC = unified GPU L2$*



Outline

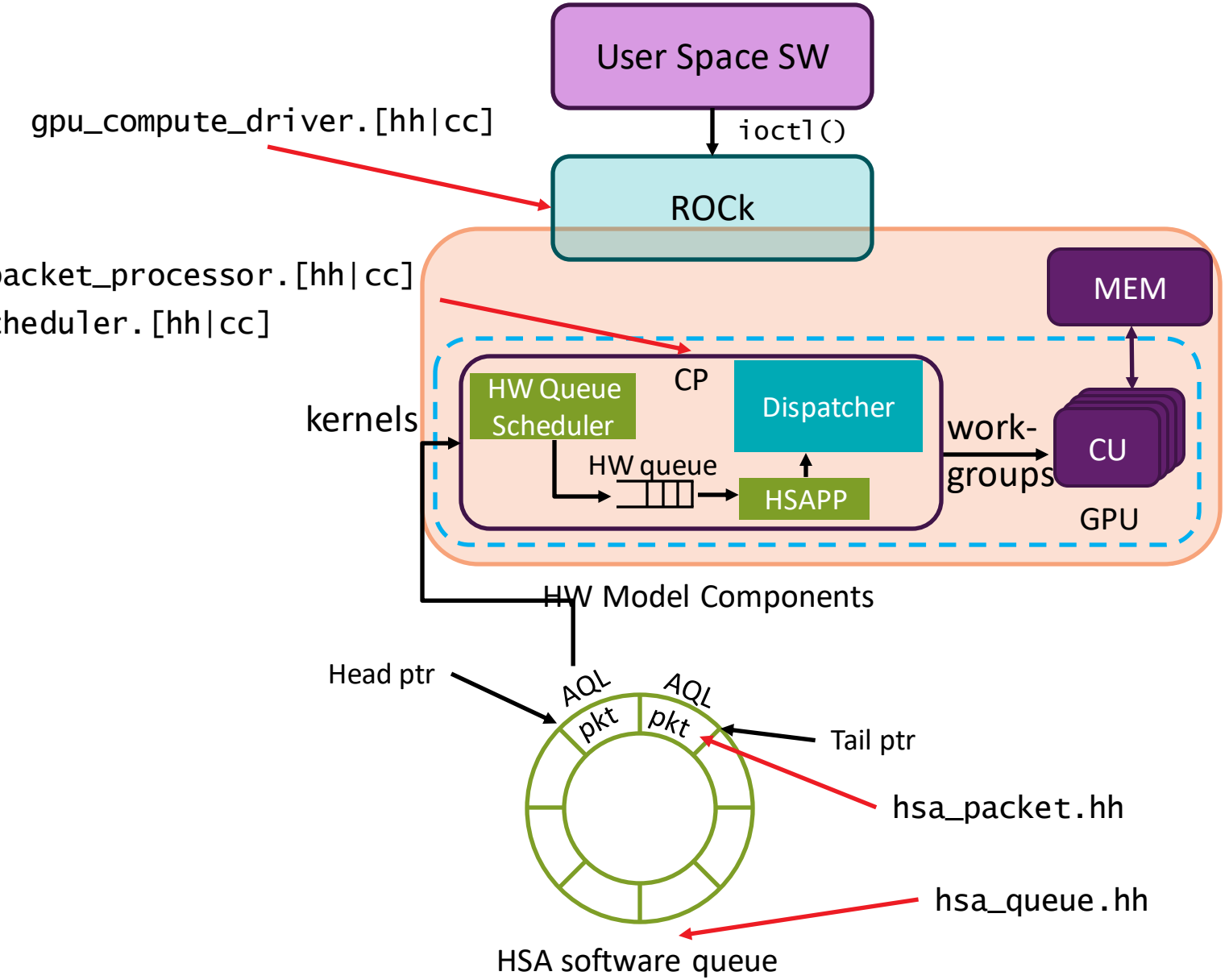
- Background: GPU Architecture & Programming Basics (20-30 minutes)
- **Modeling & Using GPUs in gem5 (1 hour)**
  - What libraries are required?
  - What support is provided?
  - **Where is GPU code?**
  - How to compile GPU model in gem5?
- Running GPU programs in gem5 (1 hour)

Key GPU Code Locations

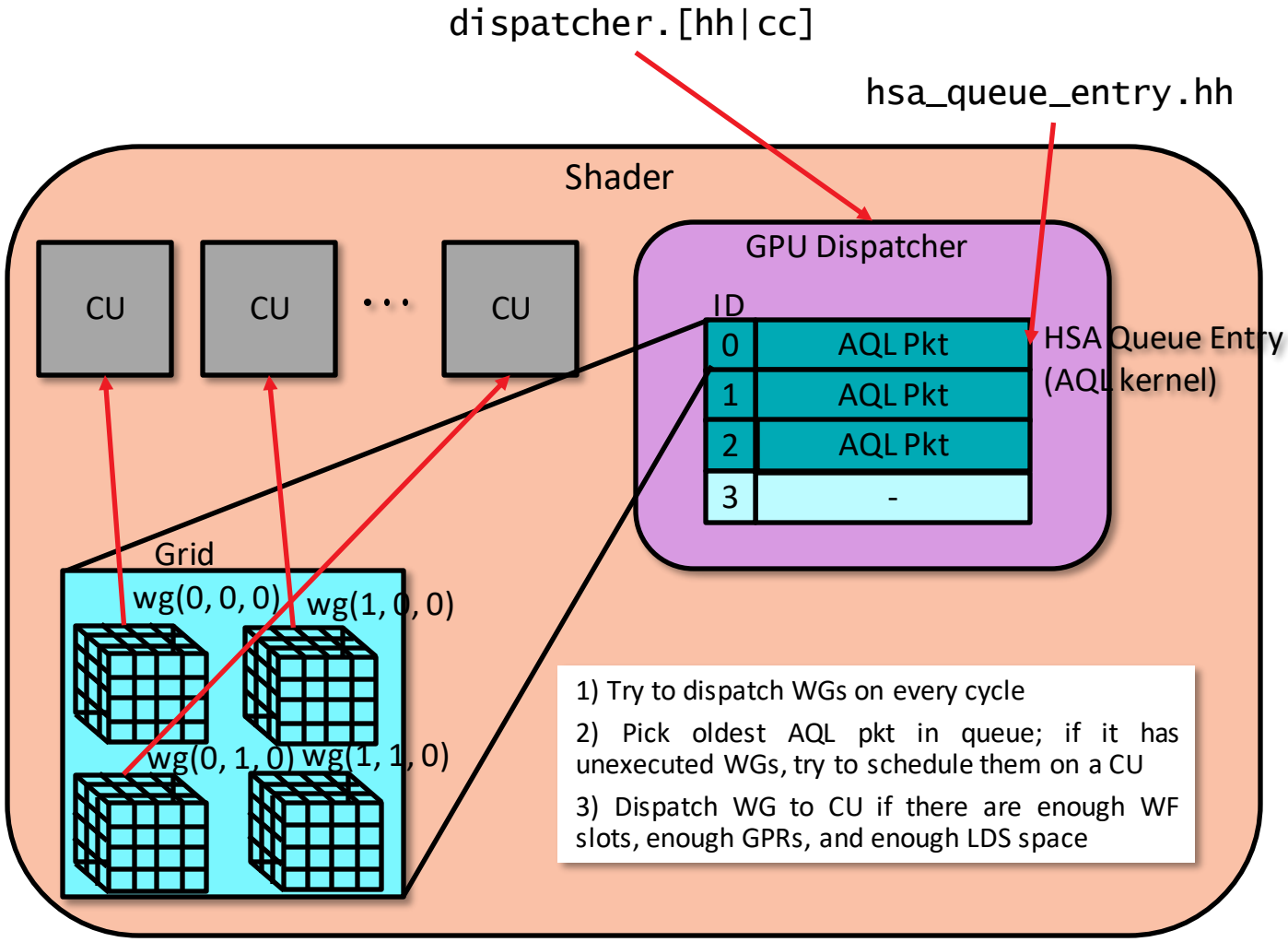
- Gem5 ← top-level directory
  - `src/`
    - `arch/amdgpu/`
      - `gcn3/` ← GCN3 specific code (e.g., GCN3 ISA)
      - `vega/` ← Vega specific code (e.g., Vega ISA)
    - `gpu-compute/` ← GPU core (CU) model
      - Instruction buffering, Registers, Vector ALUs
    - `mem/protocol/` ← APU memory model
    - `mem/ruby/` ← APU memory model
      - TCP, TCC, SQC (Ruby based)
    - `dev/hsa/` ← HSA device models
  - `configs/`
    - `example/` ← `apu_se.py` sample script (also gpufs.py script)
      - Connects multiple CUs, caches, etc. together to create overall GPU model
    - `ruby/` ← APU protocol configs

How does a GPU Kernel Actually Run?

- User space SW talks to GPU via ioctl()
  - ROCk is emulated in gem5 (SE mode only)
    - Handles ioctl commands
- CP (Command Proc) frontend
  - Two primary components:
    - HSA packet processor (HSAPP)
    - Workgroup dispatcher
- Runtime creates soft HSA queues
  - HSAPP maps them to hardware queues
  - HSAPP schedules active queues
- Runtime creates and enqueues AQL packets
  - Packets include:
    - Kernel resource requirements
    - Kernel size
    - Kernel code object pointer
    - More…

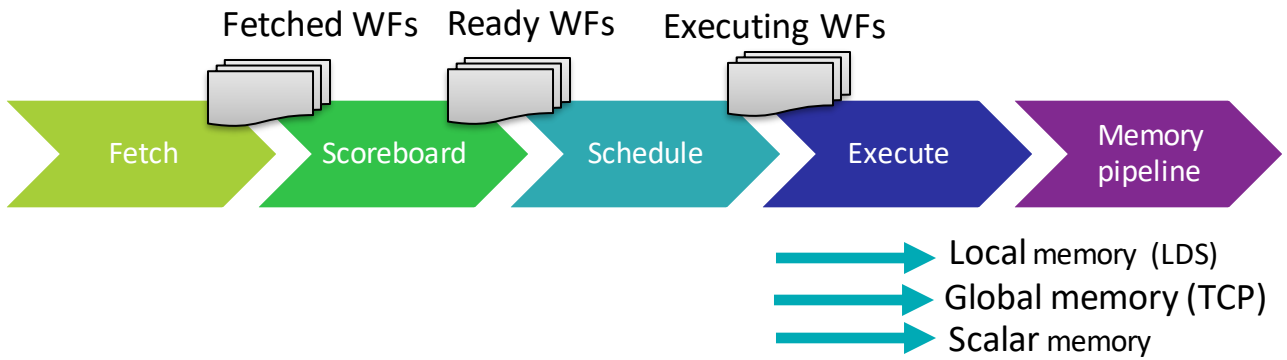


Dispatching Kernels to CUs



- Kernel dispatch is resource limited
  - WGs are scheduled to CUs
- Dispatcher tracks status of in-flight/pending kernels
  - If a WG from a kernel cannot be scheduled, it is enqueued until resources become available
  - When all WGs from a task have completed, the dispatcher frees CU resources and notifies the host

How does an instruction actually run through GPU?



- Pipeline stages
  - Fetch: fetch for dispatched WFs - `fetch_stage.[hh|cc]` and `fetch_unit.[hh|cc]`
  - Scoreboard: Check which WFs are ready - `scoreboard_check_stage.[hh|cc]`
  - Schedule: Select a WF from the ready pool - `schedule_stage.[hh|cc]`
  - Execute: Run WF on execution resource - `exec_stage.[hh|cc]`
  - Memory pipeline: Execute (local data store) LDS/global memory operation
    - `local_memory_pipeline.[hh|cc]`
    - `global_memory_pipeline.[hh|cc]`
    - `scalar_memory_pipeline.[hh|cc]`

Outline

- Background: GPU Architecture & Programming Basics (20-30 minutes)
- **Modeling & Using GPUs in gem5 (1 hour)**
  - Where is GPU code?
  - What libraries are required?
  - What support is provided?
  - **How to compile GPU model in gem5?**
- Running GPU programs in gem5 (1 hour)

Compiling gem5's GCN3 GPU model

```
cd gem5
docker run --rm --volume /var/lib/docker/codespacemount/workspace/:/workspaces -w `pwd` gcr.io/gem5-test/gcn-gpu:v22-0 scons build/GCN3_X86/gem5.opt –j17
```


Use the v22.0 gem5 docker we pulled earlier

Build the GCN3 model

Hopefully this has compiled for everyone already

Outline

- Background: GPU Architecture & Programming Basics (20-30 minutes)
- Modeling & Using GPUs in gem5 (1 hour)
- **Running GPU programs in gem5 (1 hour)**

Running Square

- What is square?
  - Simple vector addition program – each thread i does C[i] = A[i] + B[i]
  - Ideally suited to running on a GPU (perfectly parallel)
- Running:

base config script for running GPU models (in SE mode)

```
docker run --rm --volume
/var/lib/docker/codespacemount/workspace/:/workspaces -w `pwd`
gcr.io/gem5-test/gcn-gpu:v22-0 gem5/build/GCN3_X86/gem5.opt
gem5/configs/example/apu_se.py -n 3 -c
gem5-resources/src/gpu/square/bin/square
```

Path to square binary

3 threads because ROCm uses multiple processes

Should take < 5 minutes to run in gem5

Comparing register allocation schemes

- GPU models have support for multiple register allocation schemes
  - To specify: --reg-alloc-policy=[dynamic, simple] on command line
  - Simple policy: run 1 wavefront per CU at a time
    - Few stalls and contention
  - Dynamic policy: run up to max (40) wavefronts per CU at a time if registers are available
    - But more stalls and contention

- Your mission: run square with each policy, compare them!
  - Use –d to redirect output to a different folder (default: m5out)
  - Based on your results, which policy do you think runs by default?

GPU Stats

- GPU stats are different from CPU ones – specific counters for GPU

shaderActiveTicks: how long each CU was running this app

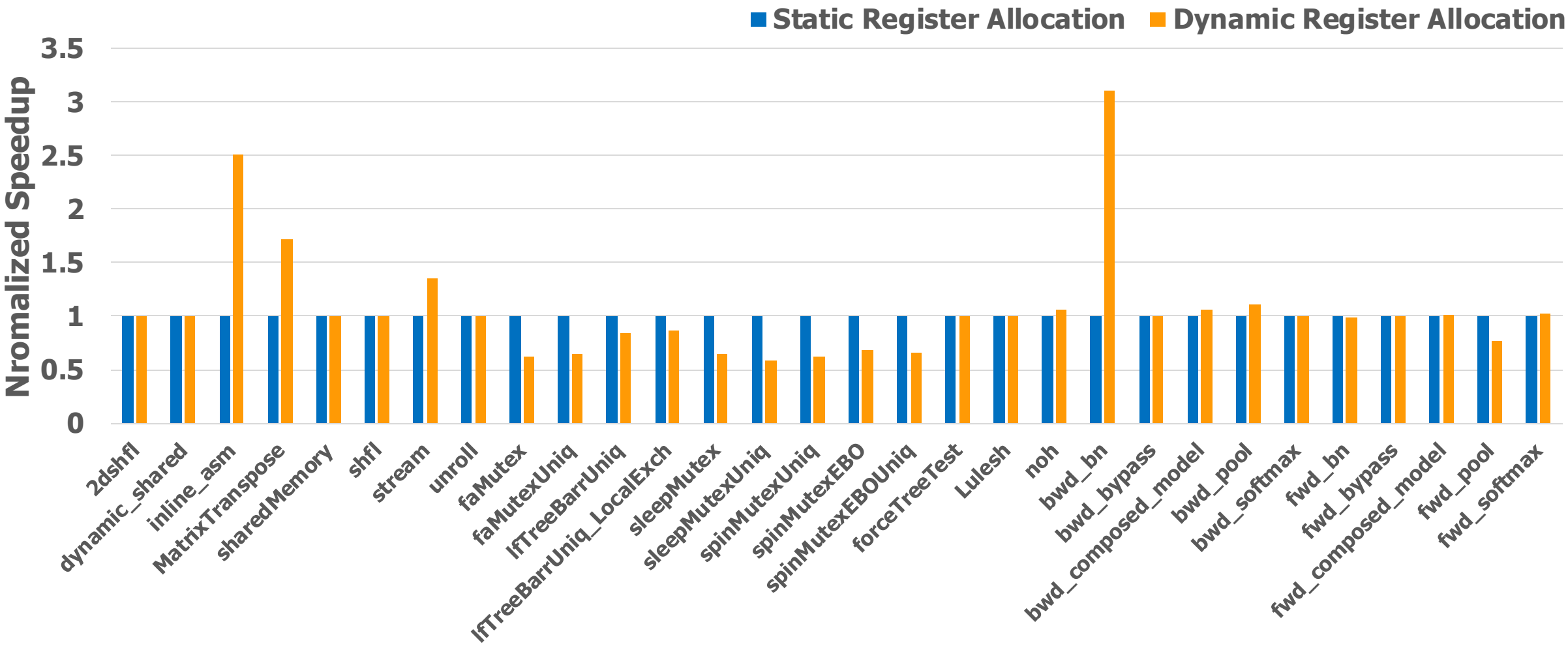
```
system.cpu3.gmToCompleteLatency::overflows            0                       # Ticks queued in GM pipes ordered response buffer (Unspecified)
system.cpu3.gmToCompleteLatency::min_value            0                       # Ticks queued in GM pipes ordered response buffer (Unspecified)
system.cpu3.gmToCompleteLatency::max_value            0                       # Ticks queued in GM pipes ordered response buffer (Unspecified)
system.cpu3.gmToCompleteLatency::total                0                     # Ticks queued in GM pipes ordered response buffer (Unspecified)
system.cpu3.coalsrLineAddresses::bucket_size            1                       # Number of cache lines for coalesced request (Unspecified)
system.cpu3.coalsrLineAddresses::min_bucket           0                       # Number of cache lines for coalesced request (Unspecified)
system.cpu3.coalsrLineAddresses::max_bucket          20                       # Number of cache lines for coalesced request (Unspecified)
system.cpu3.coalsrLineAddresses::samples        31250                     # Number of cache lines for coalesced request (Unspecified)
system.cpu3.coalsrLineAddresses::mean               0                     # Number of cache lines for coalesced request (Unspecified)
system.cpu3.coalsrLineAddresses::stdev              0                     # Number of cache lines for coalesced request (Unspecified)
system.cpu3.coalsrLineAddresses::underflows            0      0.00%      0.00% # Number of cache lines for coalesced request (Unspecified)
system.cpu3.coalsrLineAddresses          |       31250    100.00%    100.00% |           0      0.00%    100.00% |           0      0.00%    100.00%
 |           0      0.00%    100.00% |           0      0.00%    100.00% |           0      0.00%    100.00% |           0      0.00%    100.00% |
        0      0.00%    100.00% |           0      0.00%    100.00% |           0      0.00%    100.00% |           0      0.00%    100.00% |
     0      0.00%    100.00% |           0      0.00%    100.00% |           0      0.00%    100.00% |           0      0.00%    100.00% |
 0      0.00%    100.00% |           0      0.00%    100.00% |           0      0.00%    100.00% |           0      0.00%    100.00% |           0
    0.00%    100.00% |           0      0.00%    100.00% # Number of cache lines for coalesced request (Unspecified)
system.cpu3.coalsrLineAddresses::overflows            0      0.00%    100.00% # Number of cache lines for coalesced request (Unspecified)
system.cpu3.coalsrLineAddresses::min_value            0                       # Number of cache lines for coalesced request (Unspecified)
system.cpu3.coalsrLineAddresses::max_value            0                       # Number of cache lines for coalesced request (Unspecified)
system.cpu3.coalsrLineAddresses::total          31250                     # Number of cache lines for coalesced request (Unspecified)
system.cpu3.shaderActiveTicks              1151851499                     # Total ticks that any CU attached to this shader is active (Unspecified
)
system.cpu3.vectorInstSrcOperand::0            126518                     # vector instruction source operand distribution (Unspecified)
system.cpu3.vectorInstSrcOperand::1            103460                     # vector instruction source operand distribution (Unspecified)
system.cpu3.vectorInstSrcOperand::2            137288                     # vector instruction source operand distribution (Unspecified)
system.cpu3.vectorInstSrcOperand::3                 0                     # vector instruction source operand distribution (Unspecified)
system.cpu3.vectorInstDstOperand::0            128566                     # vector instruction destination operand distribution (Unspecified)
system.cpu3.vectorInstDstOperand::1            238700                     # vector instruction destination operand distribution (Unspecified)
system.cpu3.vectorInstDstOperand::2                 0                     # vector instruction destination operand distribution (Unspecified)
system.cpu3.vectorInstDstOperand::3                 0                     # vector instruction destination operand distribution (Unspecified)
system.cpu3.CUs0.vALUInsts                     62696                     # Number of vector ALU insts issued. (Unspecified)
system.cpu3.CUs0.vALUInstsPerWF            120.569231                     # The avg. number of vector ALU insts issued per-wavefront. (Unspecified
)
system.cpu3.CUs0.sALUInsts                     10016                     # Number of scalar ALU insts issued. (Unspecified)
system.cpu3.CUs0.sALUInstsPerWF             19.261538                     # The avg. number of scalar ALU insts issued per-wavefront. (Unspecified
)
system.cpu3.CUs0.instCyclesVALU                62696                     # Number of cycles needed to execute VALU insts. (Unspecified)
system.cpu3.CUs0.instCyclesSALU                10016                     # Number of cycles needed to execute SALU insts. (Unspecified)
system.cpu3.CUs0.threadCyclesVALU            4012544                     # Number of thread cycles used to execute vector ALU ops. Similar to ins
tCyclesVALU but multiplied by the number of active threads. (Unspecified)
system.cpu3.CUs0.vALUUtilization                100                     # Percentage of active vector ALU threads in a wave. (Unspecified)
system.cpu3.CUs0.ldsNoFlatInsts                    0                     # Number of LDS insts issued, not including FLAT accesses that resolve t
o LDS. (Unspecified)
system.cpu3.CUs0.ldsNoFlatInstsPerWF               0                     # The avg. number of LDS insts (not including FLAT accesses that resolve
 to LDS) per-wavefront. (Unspecified)
:
```

Comparing simple and dynamic register allocation

- Simple: 1151851499 ticks
- Dynamic: 1155814499 ticks

- Dynamic slightly (0.5%) worse!
  - Dependence tracking in gem5 GPU model is not perfect
    - Area where new research contributions are needed :)
  - Extra contention causes more stalls

Dynamic Register Allocation Not Always Better



We patched this with smarter dependence tracking, but other problems may exist

Running Multi-Kernel GPU Applications

- Many GPU applications (unlike square) run for multiple kernels
  - How to tell the stats for these different kernels apart?

- One option: m5ops – dump_reset_stats between each kernel
- For this, we will use gem5-resources/src/gpu/pannotia/bc
  - BC already has support for m5_work_begin and m5_work_end (including in Makefile)
  - So you don't need to worry about adding this
  - We want to add a dump_resetstats after each kernel completes

Adding m5ops Steps

- Compile m5ops (for x86)

```
cd gem5/util/m5
docker run --rm --volume
/var/lib/docker/codespacemount/workspace/:/workspaces -w
`pwd` gcr.io/gem5-test/gcn-gpu:v22-0 scons
build/x86/out/m5
```

Adding m5ops Steps

- Add dump_reset_stats calls to BC + Compile BC:

```
cd gem5-resources/src/gpu/pannotia/bc
// add m5ops calls to BC.cpp
// change MAX_ITERS from 150 to 2 to speedup simulation
docker run --rm --volume
/var/lib/docker/codespacemount/workspace/:/workspaces -w `pwd`
gcr.io/gem5-test/gcn-gpu:v22-0 bash -c "export
GEM5_PATH=/workspaces/gem5-bootcamp-env/gem5 ; make gem5-
fusion"
```


For m5ops, BC requires path to GEM5

Adding m5ops Steps

- Now get input file and run in gem5:

```
cd $HOME
wget http://dist.gem5.org/dist/develop/datasets/pannotia/bc/1k_128k.gr
docker run --rm --volume /var/lib/docker/codespacemount/workspace/:/workspaces -w `pwd` gcr.io/gem5-test/gcn-gpu:v22-0 gem5/build/GCN3_X86/gem5.opt -d m5out-bc gem5/configs/example/apu_se.py -n 3 --mem-size=16GB --benchmark-root=gem5-resources/src/gpu/pannotia/bc/bin -c bc.gem5 --options="1k_128k.gr"
```

Should take ~30 minutes to run this (small) input file

Impact of m5ops

- Many more sets of stats – 1 per kernel
- Can see the difference in shaderActiveTicks (or other stats) across kernels
  - backtrack_kernel and bfs_kernel calls dominate (clean_1d and clean_bc are minor)
  - Certain kernel calls (even for the same kernel) are much longer than others (have more work)

```
system.cpu3.shaderActiveTicks                       0                       # Total ticks that any CU attached to this shader is active (U
specified)
system.cpu3.shaderActiveTicks                       0                       # Total ticks that any CU attached to this shader is active (U
specified)
system.cpu3.shaderActiveTicks                10144498                       # Total ticks that any CU attached to this shader is active (U
specified)
system.cpu3.shaderActiveTicks               138368499                       # Total ticks that any CU attached to this shader is active (U
specified)
system.cpu3.shaderActiveTicks               332231999                       # Total ticks that any CU attached to this shader is active (U
specified)
system.cpu3.shaderActiveTicks               157947999                       # Total ticks that any CU attached to this shader is active (U
specified)
system.cpu3.shaderActiveTicks                       0                       # Total ticks that any CU attached to this shader is active (U
specified)
system.cpu3.shaderActiveTicks                       0                       # Total ticks that any CU attached to this shader is active (U
specified)
system.cpu3.shaderActiveTicks              1063540997                       # Total ticks that any CU attached to this shader is active (U
```

Can contribute this change to BC to gem5-resources tomorrow!

gem5-Resources: lots of GPU workloads

The square test is used to test the GCN3-GPU model.

Compiling square, compiling the GCN3_X86 gem5, and running square on gem5 is dependent on the gcn-gpu docker image, built from the `util/dockerfiles/gcn-gpu/Dockerfile` on the gem5 stable branch.

Compiling Square

By default, square will build for all supported GPU types (gfx801, gfx803)

```
cd src/gpu/square
docker run --rm -v ${PWD}:${PWD} -w ${PWD} -u $UID:$GID gcr.io/gem5-test/gcn-gpu:v21-2 make
```

The compiled binary can be found in the `bin` directory.

Pre-built binary

A pre-built binary can be found at http://dist.gem5.org/dist/v21-2/test-progs/square/square.

Compiling GCN3_X86/gem5.opt

The test is run with the GCN3_X86 gem5 variant, compiled using the gcn-gpu docker image:

```
git clone https://gem5.googlesource.com/public/gem5
cd gem5
docker run -u $UID:$GID --volume $(pwd):$(pwd) -w $(pwd) gcr.io/gem5-test/gcn-gpu:v21-2 scons build/GCN3_X86/gem5.opt -j <num cores>
```

Running Square on GCN3_X86/gem5.opt

```
docker run -u $UID:$GID --volume $(pwd):$(pwd) -w $(pwd) gcr.io/gem5-test/gcn-gpu:v21-2 gem5/build/GCN3_X86/gem5.opt gem5/configs/example/apu_se.py -n 3 -c bin/square
```

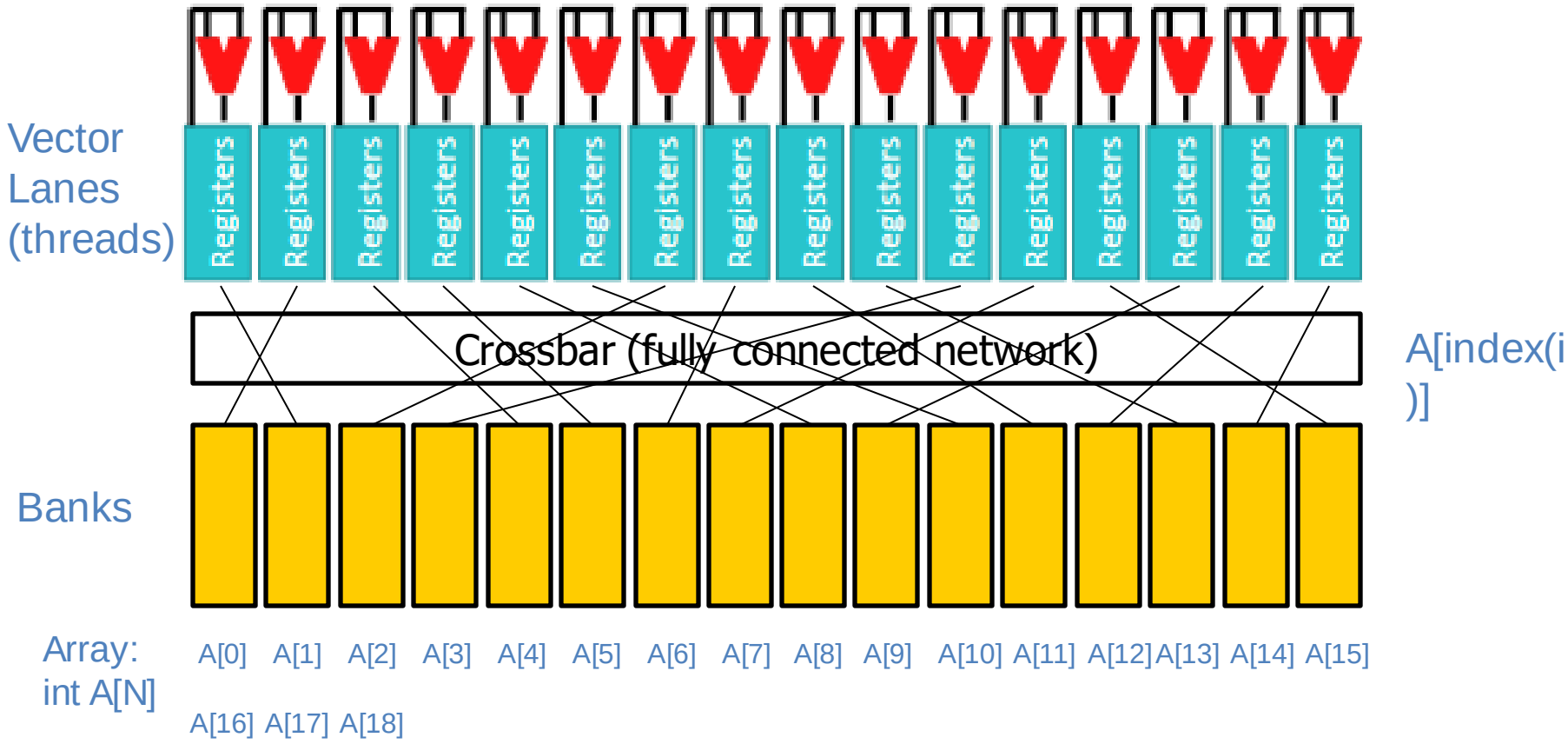
Utilize these to get started after the workshop!

BACKUP

gem5

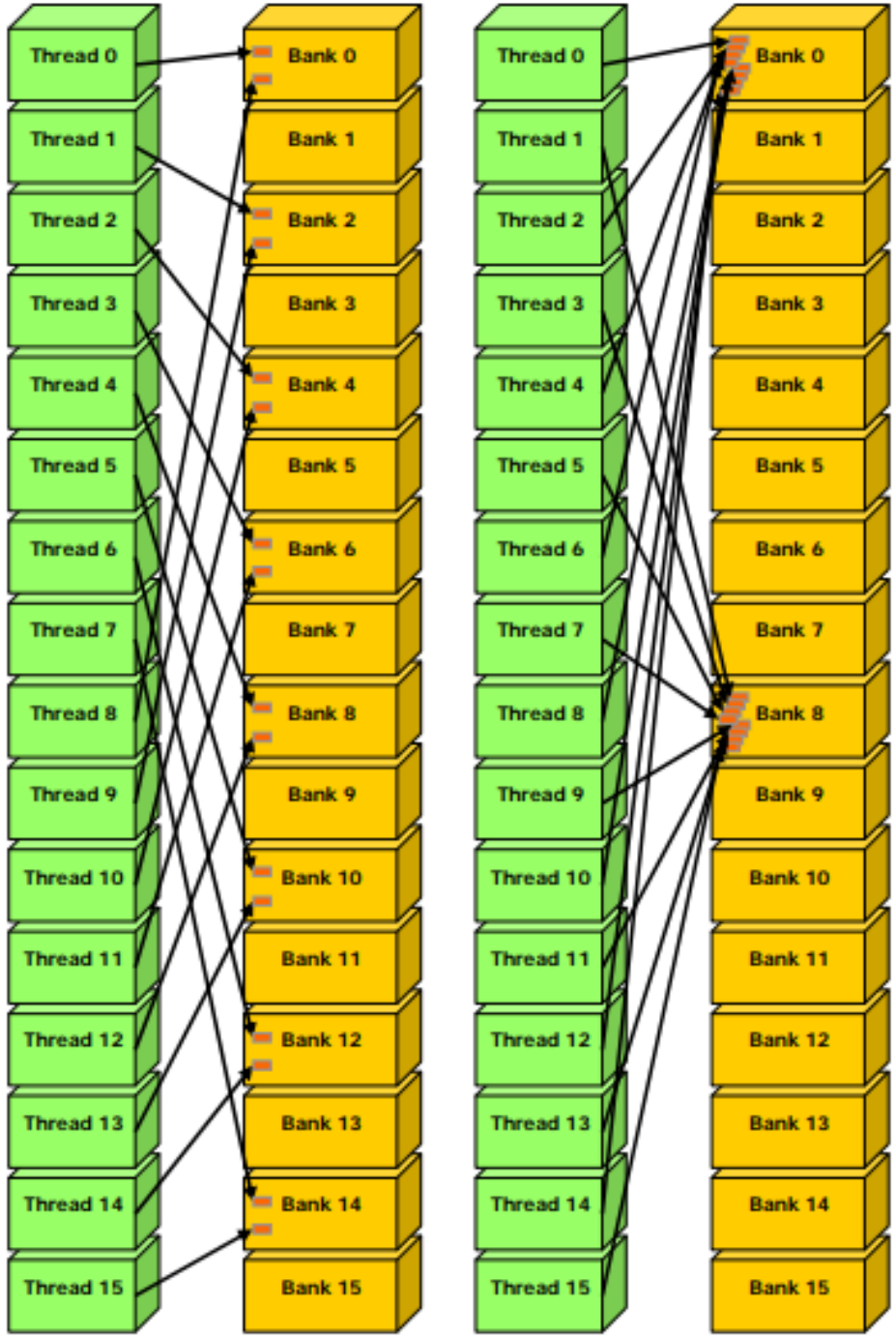
Scratchpad Organization

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



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

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

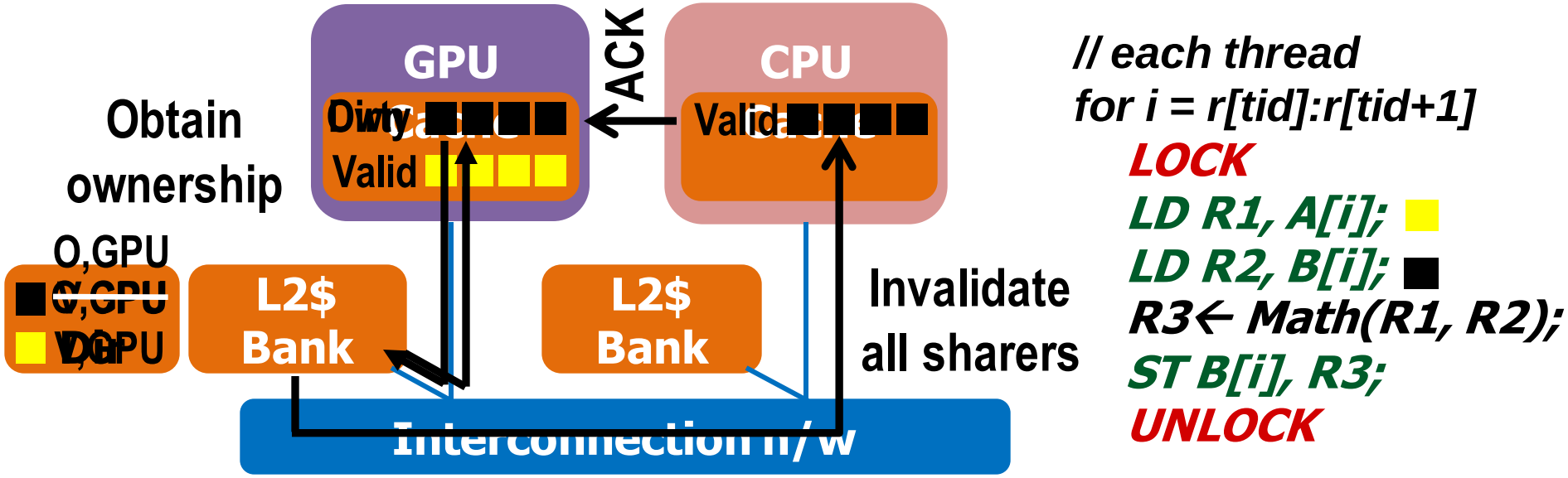
+ designed to exploit data parallelism
+/- tradeoff single-thread performance for increased parallel processing
+ hide memory latencies.
+ more compute flops.
- Limited by Amdahl's Law.

What's "good" for executing on GPUs?

- (Traditionally)

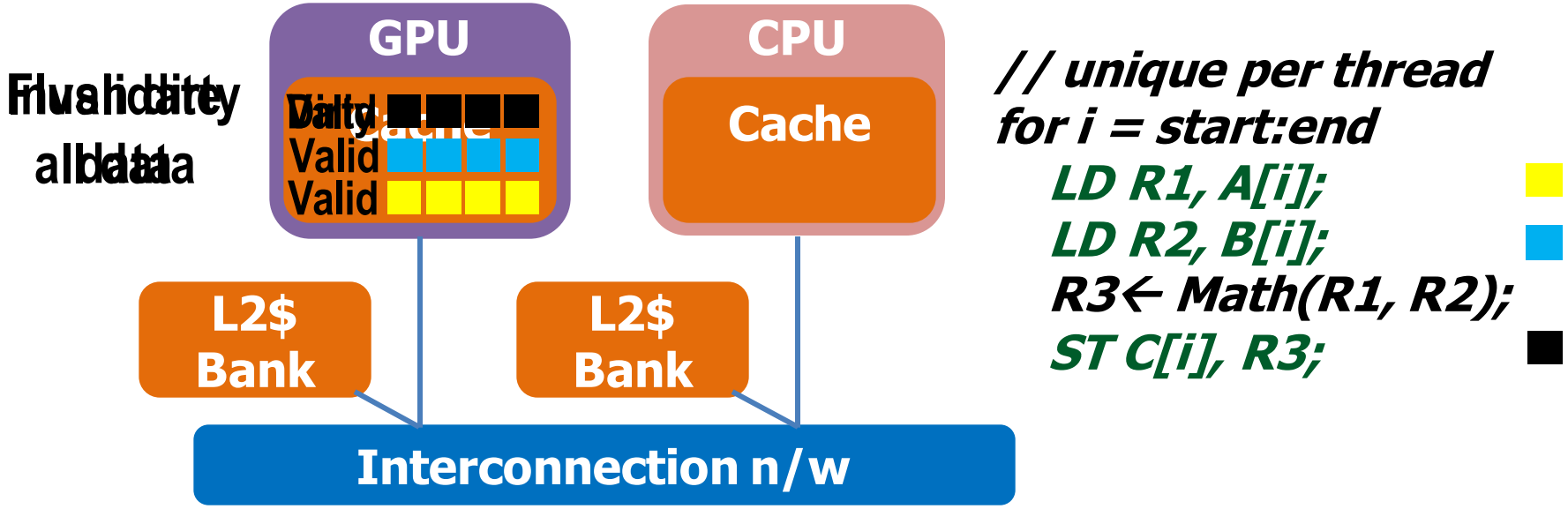
- Abundant parallelism.
  - Single-threaded performance less important (MLP and TLP instead of ILP).
- Workloads that take advantage of "special features" (like textures).
- Workloads that require lots of bandwidth.
- Regular data access patterns

CPU Coherence: MESI



- 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



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

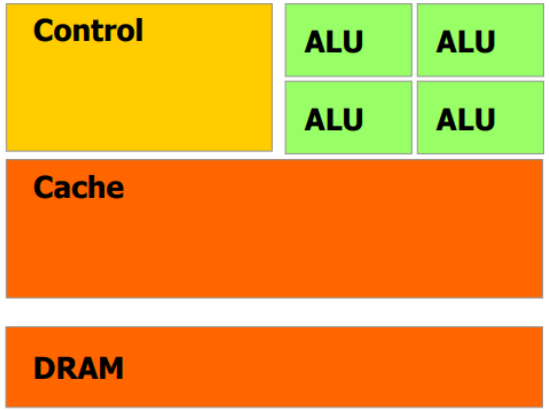
- Provides very weak guarantees
  - Respect program order within a single thread
  - Easy to design hardware
  - Programmers add ***fences*** to provide extra guarantees
    - Fence guarantee all previous accesses are visible before proceeding
    - … usually

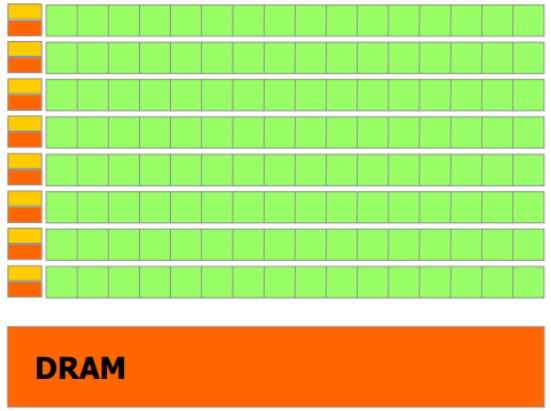
- Most GPUs use a ***scoped*** memory consistency model
  - Only apply GPU fences locally if all users are local – less overhead
  - But more work for programmer

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
  - Data needs to be laid out correctly (bank conflicts, communication, etc.)
- Control Flow:
  - Hardware structures to track thread divergence
- 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

Example Slide

```
1  from gem5.components.boards.simple_board import SimpleBoard
2  from gem5.components.cachehierarchies.classic.no_cache import NoCache
3  from gem5.components.memory.single_channel import SingleChannelDDR3_1600
4  from gem5.components.processors.simple_processor import SimpleProcessor
5  from gem5.components.processors.cpu_types import CPUTypes
6  from gem5.resources.resource import Resource
7  from gem5.simulate.simulator import Simulator
8
9  """
10 Instructions for generating this code will largely follow the tutorial outlined
11 in https://www.gem5.org/documentation/gem5-stdlib/hello-world-tutorial
12 """
13
14 # Obtain the components.
15 cache_hierarchy = NoCache()
16 memory = SingleChannelDDR3_1600("1GiB")
17 processor = SimpleProcessor(cpu_type=CPUTypes.ATOMIC, num_cores=1)
18
19 #Add them to the board.
20 board = SimpleBoard(
21     clk_freq="3GHz",
22     processor=processor,
23     memory=memory,
24     cache_hierarchy=cache_hierarchy,
25 )
```

- Code should include line numbers for easy referencing.
- No "dark mode" code examples. Dark text on light background is best.
- The font we are using is called "Neuzeit". Install here: https://dl.freefontsfamily.com/download/Neuzeit-Font
- Text color is "Aqua".