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, Ndubuisi Osuji, Vishnu Ramadas, Kyle Roarty, Mingyuan Xiang, Bobbi Yogatama, & others

- Some slides based on content presented by these folks previously

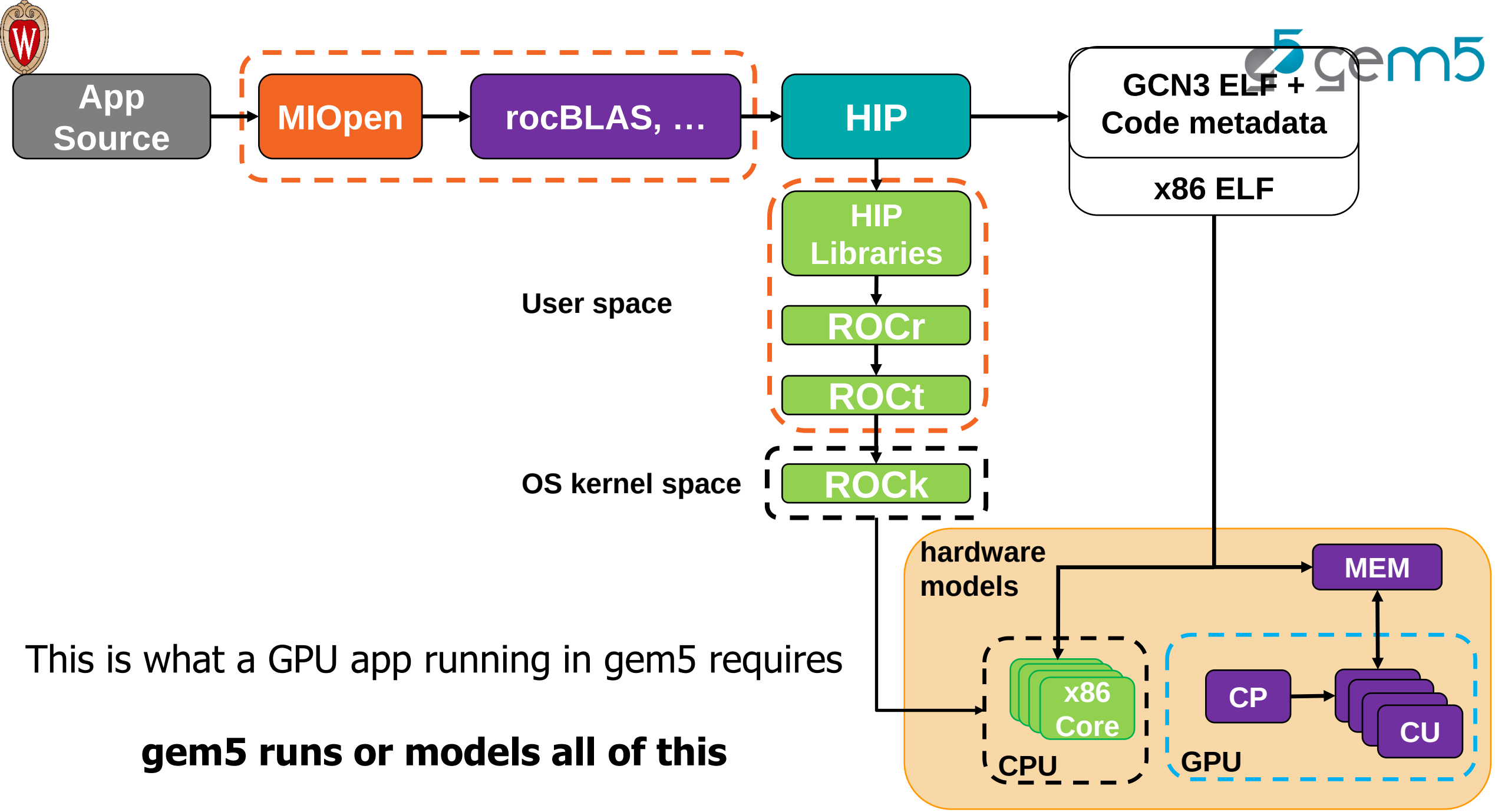
Compiling gem5 GPU Model

```
docker pull gcr.io/gem5-test/gcn-gpu:v22-1
git clone https://gem5.googlesource.com/public/gem5
cd gem5
docker run --volume $(pwd):$(pwd) -w $(pwd) gcr.io/gem5-test/gcn-gpu:v22-1 scons build/GCN3_X86/gem5.opt -j9
```

- This will take ~20 minutes to compile – we'll come back to them
  - Note: this is not currently working in codespace

Outline

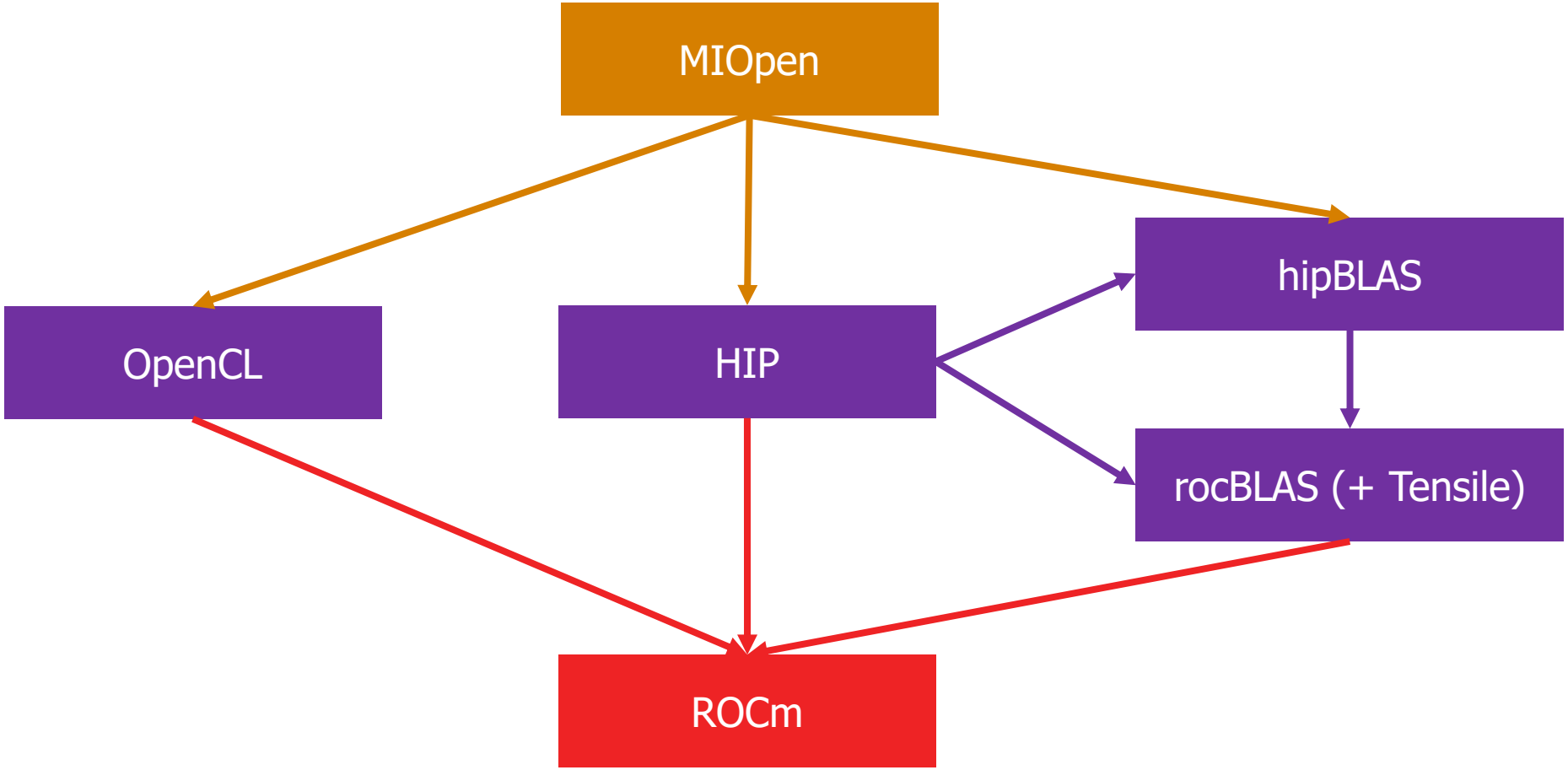
- **Modeling & Using GPUs in gem5**
  - **What libraries are required?**
  - What support is provided?
  - Where is GPU code?
  - How to compile GPU model in gem5?
  - Running SE mode GPU programs in gem5
  - GPUFS Primer



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:v22-1 ⟵ For gem5 v22.1

gem5 GPU docker

Outline

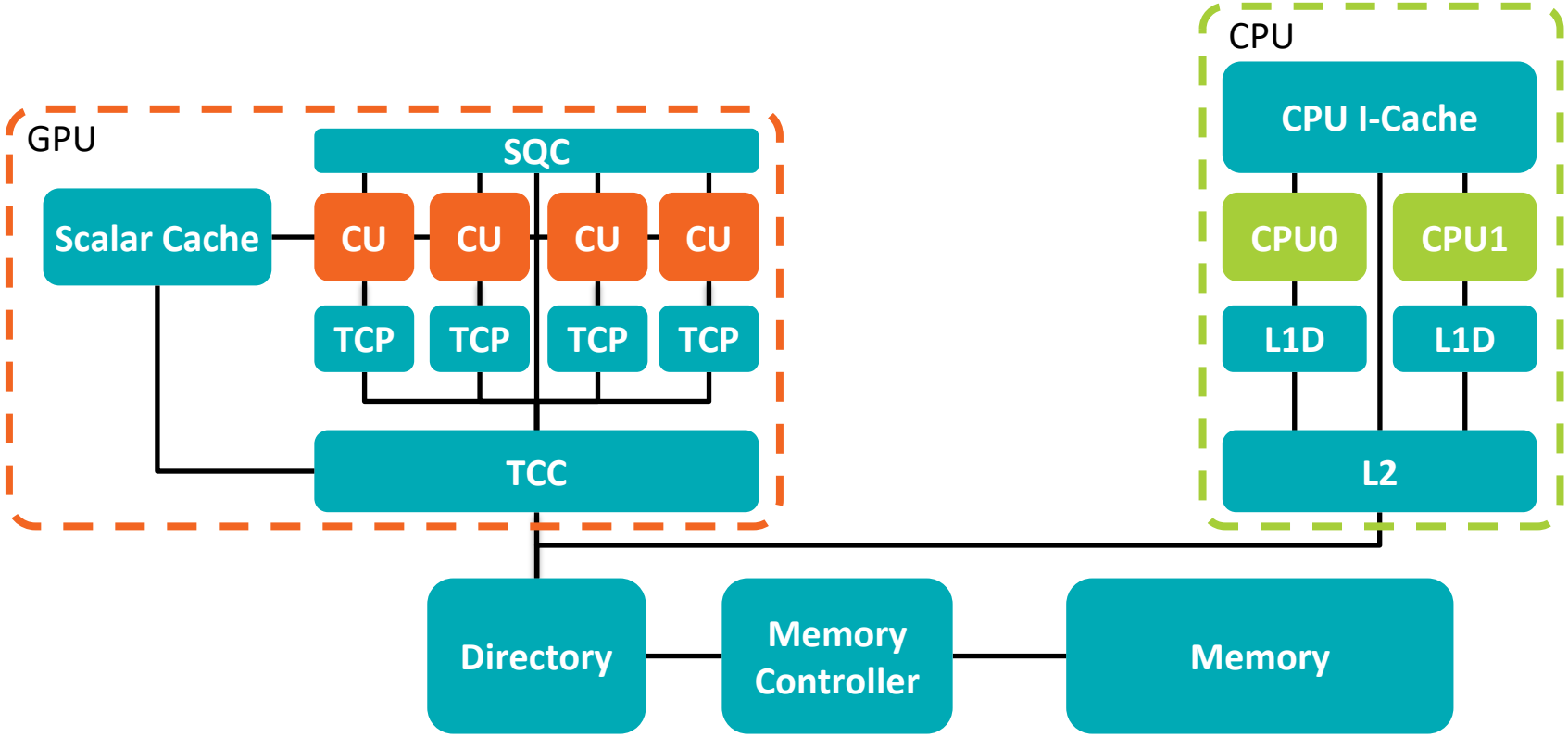
- **Modeling & Using GPUs in gem5**
  - What libraries are required?
  - **What support is provided?**
  - Where is GPU code?
  - How to compile GPU model in gem5?
  - Running GPU programs in gem5
  - GPUFS Primer

Current Support

- ROCm supported in gem5: ROCm v4.0
- SE mode vs. FS mode:
  - SE mode is well supported on stable – **today's main focus**
  - FS mode was just released on develop with 22.0, will **briefly 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
- **SE part will focus on GCN3 and gfx801 (most tested)**

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

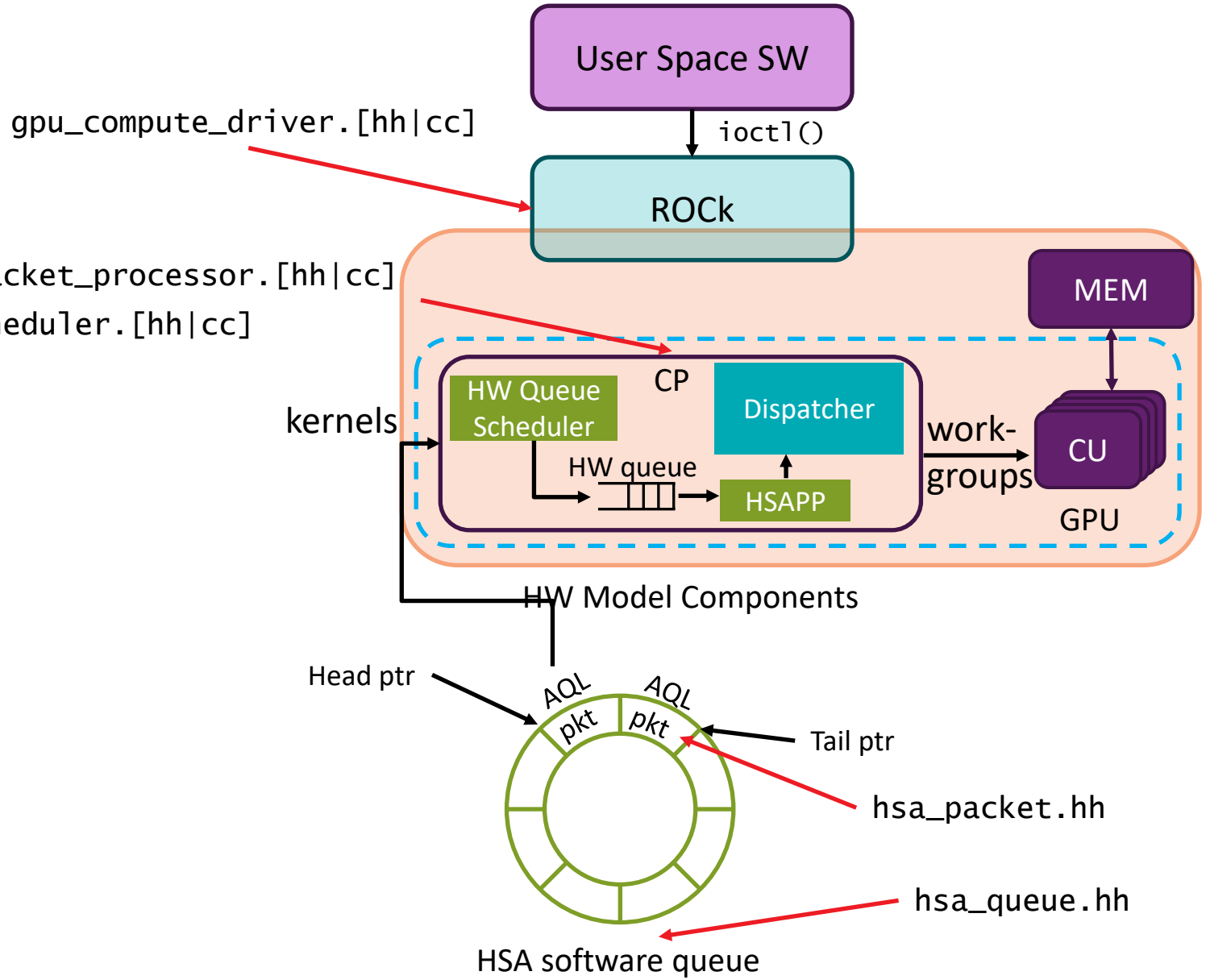
- **Modeling & Using GPUs in gem5**
  - What libraries are required?
  - What support is provided?
  - **Where is GPU code?**
  - How to compile GPU model in gem5?
  - Running SE mode GPU programs in gem5
  - GPUFS Primer

Key GPU Code Locations

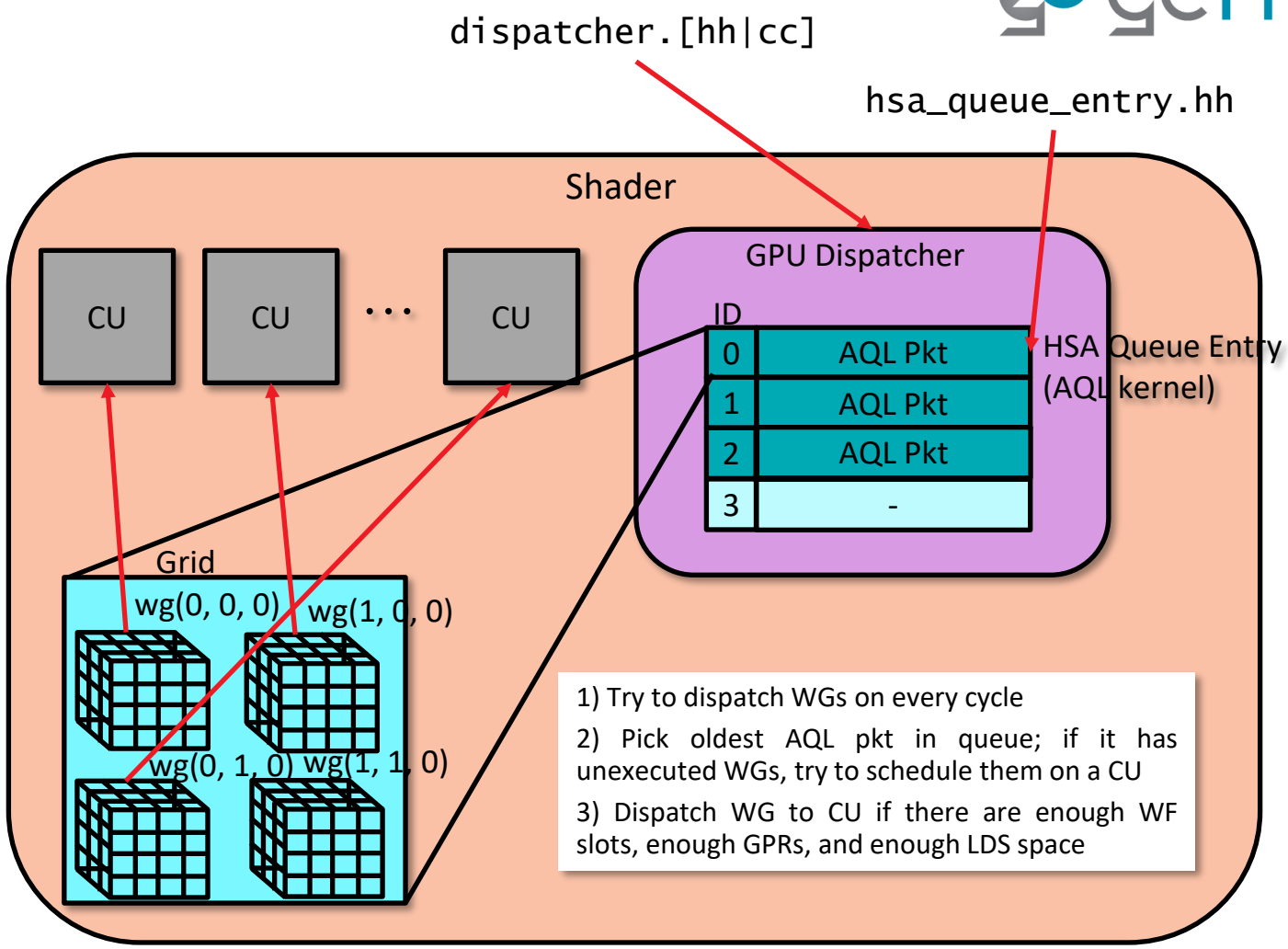
- Gem5 ← top-level directory
  - `src/`
    - `arch/amdgpu/`
      - `gcn3/` ← GCN3 specific code (e.g., GCN3 ISA)
      - `vega/` ← Vega specific code (e.g., Vega ISA)
    - `gpu-compute/` ← GPU core (CU) model
      - Instruction buffering, Registers, Vector ALUs
    - `mem/protocol/` ← APU memory 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…

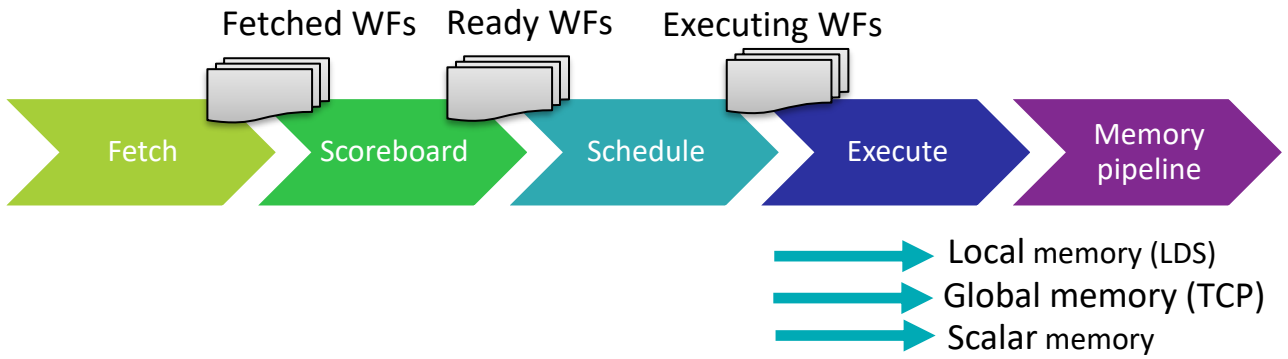


Dispatching Kernels to CUs



- 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

- **Modeling & Using GPUs in gem5**
  - Where is GPU code?
  - What libraries are required?
  - What support is provided?
  - **How to compile GPU model in gem5?**
  - Running GPU programs in gem5
  - GPUFS Primer

Compiling gem5's GCN3 GPU model

```
cd gem5
docker run --volume $(pwd):$(pwd) -w $(pwd) gcr.io/gem5-test/gcn-gpu:v22-1 scons build/GCN3_X86/gem5.opt -j9
```

Use the v22.1 gem5 docker we pulled earlier

Build the GCN3 model

Hopefully this has compiled for everyone already

Outline

- **Modeling & Using GPUs in gem5**
  - Where is GPU code?
  - What libraries are required?
  - What support is provided?
  - How to compile GPU model in gem5?
  - **Running GPU programs in gem5**
  - GPUFS Primer

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:

```
cd .. ; mkdir –p bin
wget http://dist.gem5.org/dist/v22-1/test-progs/square/square
docker run --volume $(pwd):$(pwd) -w $(pwd) gcr.io/gem5-test/gcn-gpu:v22-1 gem5/build/GCN3_X86/gem5.opt gem5/configs/example/apu_se.py -n 3 -c bin/square
```

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

3 threads because ROCm uses multiple processes

Path to square binary

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

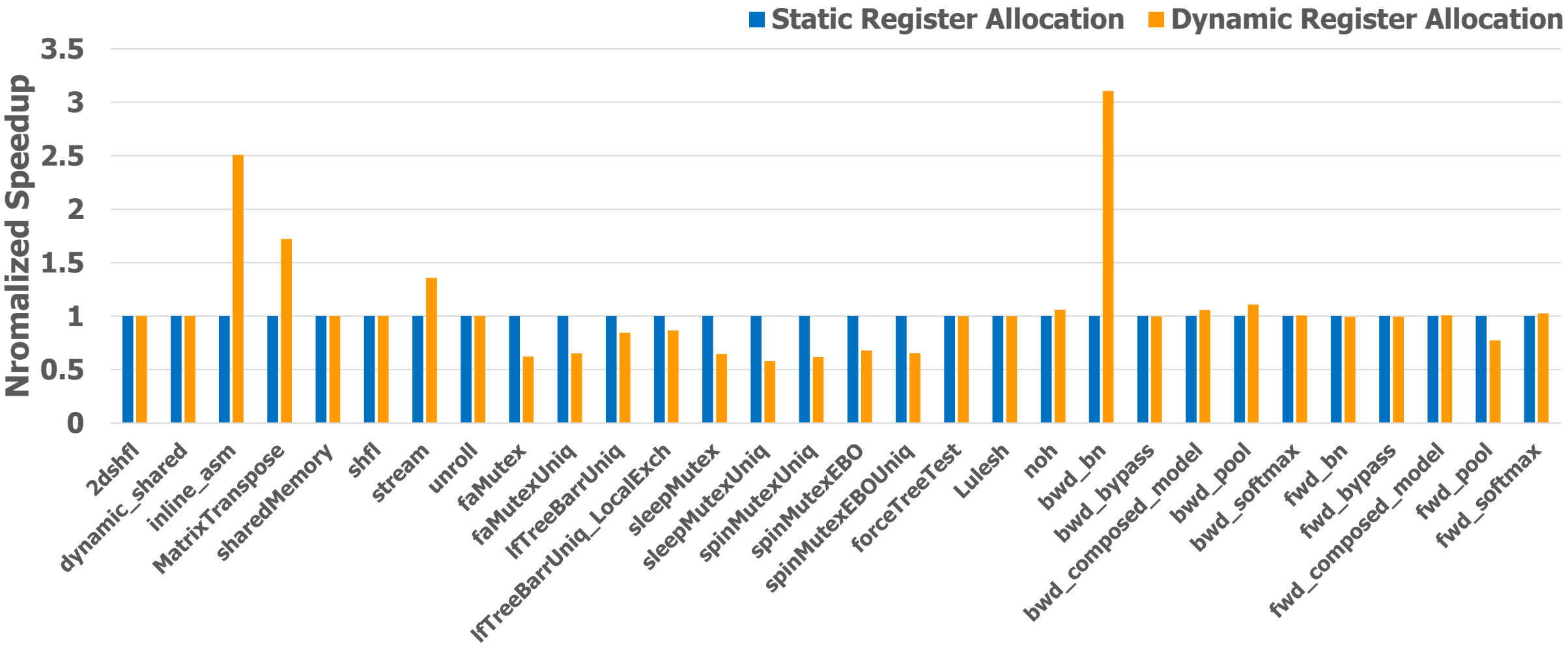
```
system.cpu3.gmToCompleteLatency::overflows            0                       # Ticks queued in GM pipes ordered response buffer (Unspecified)
system.cpu3.gmToCompleteLatency::min_value            0                       # Ticks queued in GM pipes ordered response buffer (Unspecified)
system.cpu3.gmToCompleteLatency::max_value            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

gem5-Resources: lots of GPU workloads

https://resources.gem5.org/resources/square

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!

Outline

- **Modeling & Using GPUs in gem5**
  - Where is GPU code?
  - What libraries are required?
  - What support is provided?
  - How to compile GPU model in gem5?
  - Running GPU programs in gem5
  - **GPUFS Primer**

GPUFS (Full System) Simulation

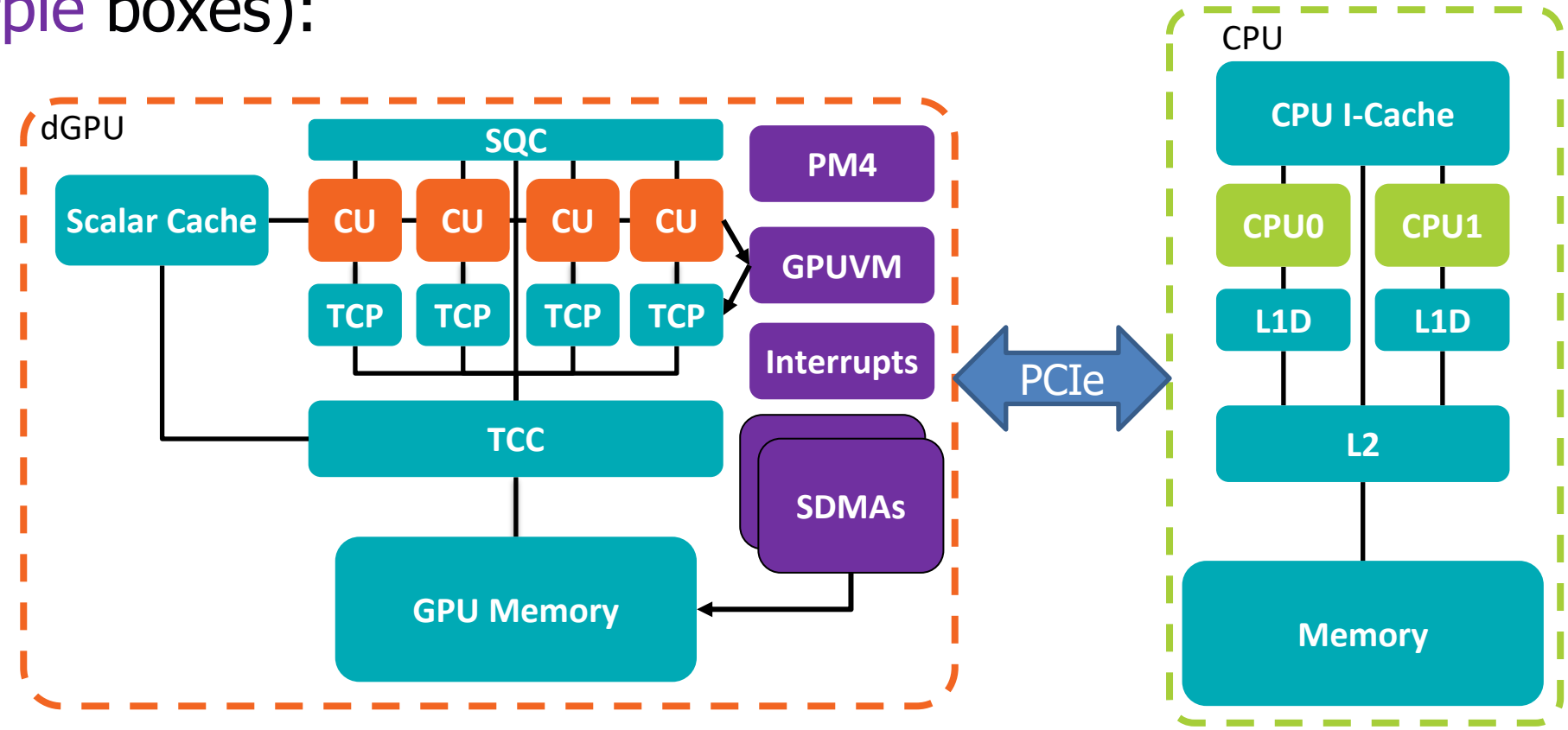
- Now can simulate GPU apps in Full System mode too ("GPUFS")
  - **Caveat**: As of gem5 22.1 only X86 KVM CPU is supported
  - Thus gem5 host machine must be X86 with KVM support
  - Support for other models is in progress.
- Main GPUFS differences vs. SE mode:
  - ROCk (Linux kernel driver) is simulated instead of emulated
  - GPU DMA engines and packet processors are modeled in GPUFS
  - Virtual memory support is available in GPUFS

GPUFS use cases

- ROCk (Linux kernel driver) is simulated instead of emulated
  - Linux driver can be modified to simulate various changes
    For example: Experimenting with AMD GPU's flexible page sizes
- GPU DMA engines and packet processors are modeled in GPUFS
  - GPU memcpy calls can be performed functionally to decrease simulation time
  - Can introduce new DMA packet types
  - For example: New packets for data placement
- Virtual memory support is available in GPUFS
  - Can modify virtual memory in driver to test new GPU uses
  - For example: Page fault handling when GPU footprint > dGPU memory size

GPUFS Simulated System Changes

- GPU Virtual Memory (GPUVM), DMA engines (SDMA), PM4 packet processor, host data bypass path, and interrupt handler are added (purple boxes):



Creating GPUFS Resources

- Docker Container
  - Contains an installation of ROCm software stack
  - Used to **build** applications to run in full system simulations
- Publicly Available!
  - Integrated into gem5 repo: https://gem5.googlesource.com/
- Added bmks & doc. in gem5-resources *[Bruce ISPASS '20 Best Paper Nom.]*
- Strongly suggest building applications requiring ROCm with docker
- Disk Image & Linux Kernel
- Contains a version of Linux and ROCm to be used for Full System **simulation**
- http://dist.gem5.org/dist/v22-1/images/x86/ubuntu-18-04/x86-gpu-fs-20220512.img.gz
  http://dist.gem5.org/dist/v22-1/kernels/x86/static/vmlinux-5.4.0-105-generic
- Disks can also be created manually for more recent versions

Current GPUFS Support

- ROCm supported in gem5: ROCm v4.3
  - ROCm 5.0 and ROCm 5.4 have also been tested
  - Currently these disk images need to be created manually (vs. using packer / downloading image)
- Full System AMD GPU support
  - Vega (gfx900 – dGPU)
  - Only officially supported gfx90x GPUs can be run in gem5 Full System with real driver
- Standard library currently not supported – use configs/example/gpufs/vega10_kvm.py
- Currently only supports Ruby
- **GPUFS is only supported on Vega with dGPU devices**

Compiling gem5's Vega GPU Model

- Full System GPU model is built similar to other ISAs:
  - `scons -j17 build/VEGA_X86/gem5.opt`

- Do not need to build gem5 using docker!

Building applications to run in GPUFS

- A docker image with ROCm stack and compilers is provided to build **apps**
  - `docker pull gcr.io/gem5-test/gpu-fs:v22-1`

- Example: Building square in gem5-resources repository
  - `cd gem5-resources/src/gpu/square`
  - `docker run --rm -v $PWD:$PWD -w $PWD gcr.io/gem5-test/gpu-fs:v22-1 make`

Running square in GPUFS

- Note: Currently on the X86 KVM CPU can be used
  - In the future other CPU models will be supported

- Running:
  - `build/VEGA_X86/gem5.opt configs/example/gpufs/vega10_kvm.py --app` **`gem5-resources/src/gpu/square/bin/square`** `--disk-image=/path/to/disk –kernel=/path/to/kernel --gpu-mmio-trace=gem5-resources/src/gpu-fs/vega_mmio.log`

- Note: All files passed to command lines are **inputs** and must be valid
  - This requires that you have built the disk image and kernel (**takes too long for tutorial**)

Comparing register allocation schemes

- Similar to SE mode specify --reg-alloc-policy on command line
  - In general commands are the same as SE mode but without running through docker
- 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?