AMD

THE AMD gem5 APU SIMULATOR:
MODELING GPUS USING THE MACHINE ISA

TONY GUTIERREZ, SOORAJ PUTHOOR, TUAN TA*, MATT SINCLAIR,
AND BRAD BECKMANN
AMD RESEARCH, *CORNELL
JUNE 2, 2018

OBJECTIVES AND SCOPE

▲ Objectives
  - Introduce the Radeon Open Compute Platform (ROCm)
  - AMD's Graphics Core Next (GCN) architecture and GCN3 ISA
 - Describe the gem5-based APU simulator

▲ Scope
  - Emphasis on the GPU side of the simulator
    - APU (CPU+GPU) model, not discrete GPU
    - Covers GPU arch, GCN3 ISA, and HW-SW interfaces

▲ Why are we releasing our code?
  - Encourage AMD-relevant research
  - Modeling ISA and real system stack is important [1]
  - Enhance academic collaborations
    - Enable intern candidates to get experience before arriving
    - Enable interns to take their experience back to school

▲ Acknowledgement
  - AMD Research's gem5 team
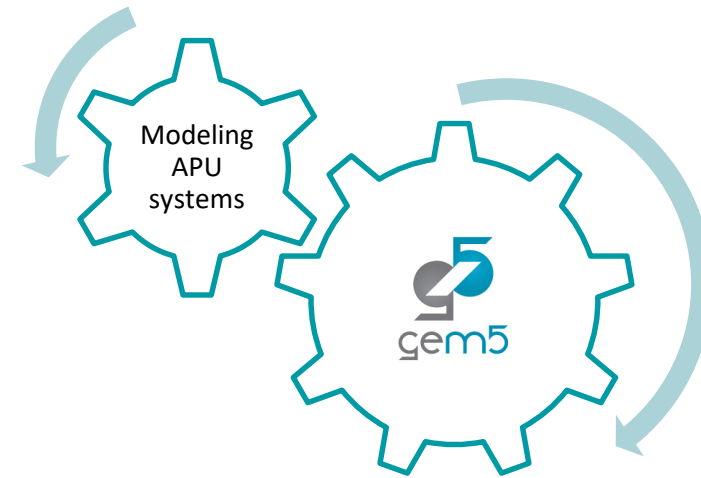
[1] Gutierrez *et al. Lost in Abstraction: Pitfalls of Analyzing GPUs at the Intermediate Language Level. HPCA*, 2018.

QUICK SURVEY

AMD

▲ Who is in our audience?

- – Graduate students
- – Faculty members
- – Working for government research labs
- – Working for industry

▲ Have you written an GPU program?

- – CUDA, OpenCL^{TM}, HIP, HC, C++ AMP, other languages

▲ Have you used these simulators?

- – GPGPU-Sim
- – Multi2Sim
- – gem5
  - – Our HSAIL-based APU model

▲ Are you familiar with our HPCA 2018 paper?

- – *Lost in Abstraction: Pitfalls of Analyzing GPUs at the Intermediate Language Level*

OUTLINE

AMD

Topic	Presenter	Time
Background	**Tony**	**8:00 – 8:15**
ROCm, GCN3 ISA, and GPU Arch	Tony	8:15 – 9:15
HSA Implementation in gem5	Sooraj	9:15 – 10:00
Break		10:00 – 10:30
Ruby and GPU Protocol Tester	Tuan	10:30 – 11:15
Demo and Workloads	Matt	11:15 – 11:50
Summary and Questions	All	11:50 – 12:00

BACKGROUND

AMD

▲ Overview of gem5
  – Source tree

▲ GPU terminology and system overview

▲ HSA standard and building blocks
  – Coherent shared virtual memory
  – User-level queues
  – Signals
  – etc.

OVERVIEW OF gem5

AMD

- Open-source modular platform for system architecture research
  - Integration of M5 (Univ. of Michigan) and GEMS (Univ. of Wisconsin)
  - Actively used in academia and industry

- Discrete-event simulation platform with numerous models
  - CPU models at various performance/accuracy trade-off points
    - Multiple ISAs: x86, ARM, Alpha, Power, SPARC, MIPS
  - Two memory system models: Ruby and "classic" (M5)
    - Including caches, DRAM controllers, interconnect, coherence protocols, etc.
  - I/O devices: disk, Ethernet, video, etc.
  - Full system or app-only (system-call emulation)

- Cycle-level modeling (not "cycle accurate")
  - Accurate enough to capture first-order performance effects
  - Flexible enough to allow prototyping new ideas reasonably quickly

- See http://www.gem5.org

- More information available from Jason Lowe-Power's tutorial
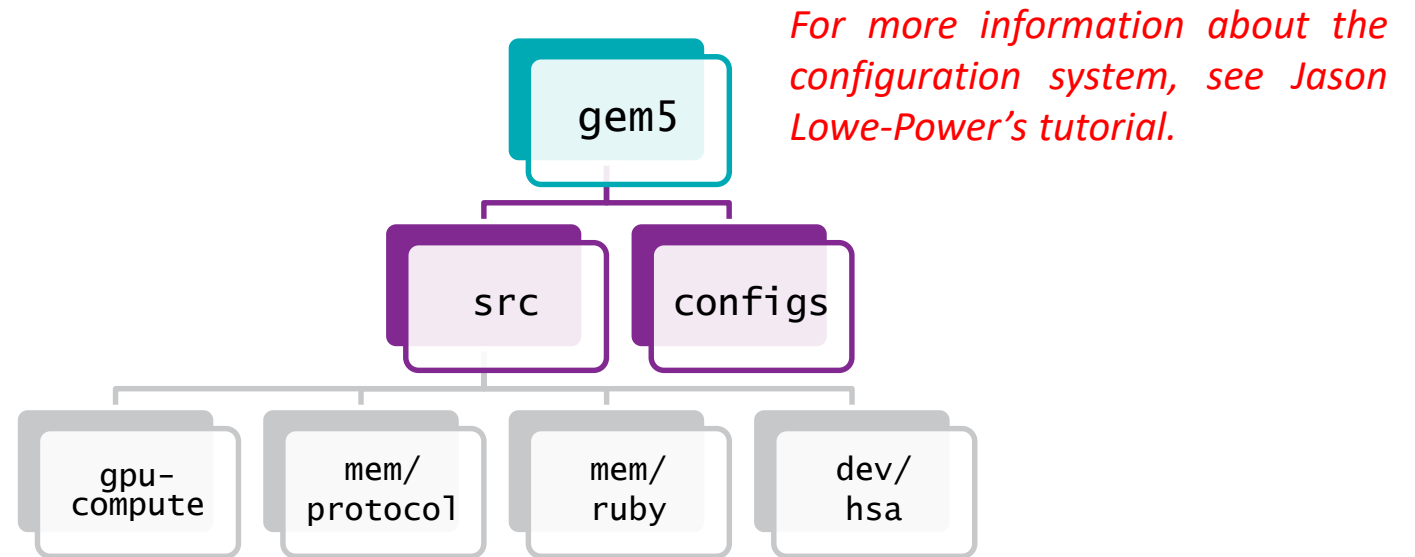 - http://learning.gem5.org/tutorial/

APU SIMULATOR CODE ORGANIZATION
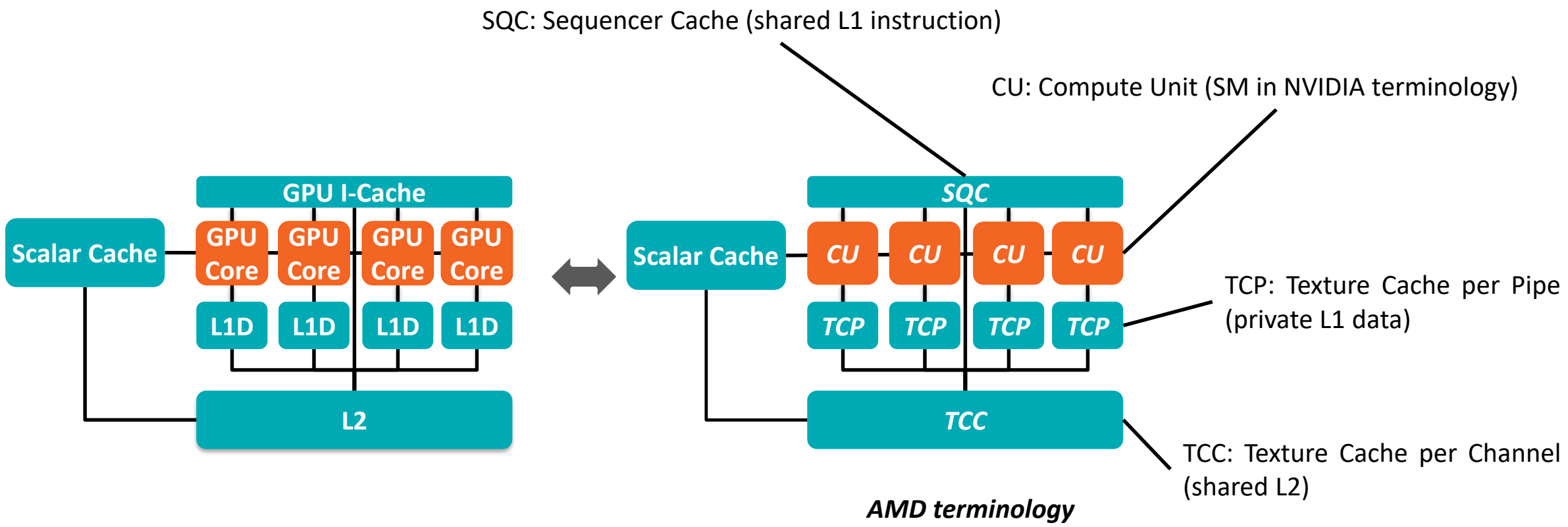
◢ Gem5 ← top-level directory
  – `src/`
    – `gpu-compute/` ← GPU core model
    – `mem/protocol/` ← APU memory model
    – `mem/ruby/` ← APU memory model
    – `dev/hsa/` ← HSA device models
  – `configs/`
    – `example/` ← `apu_se.py` sample script
    – `ruby/` ← APU protocol configs

For more information about the configuration system, see Jason Lowe-Power's tutorial.

For the remainder of this talk, files without a directory prefix are located in `src/gpu-compute/`

GPU TERMINOLOGY

AMD

SQC: Sequencer Cache (shared L1 instruction)

CU: Compute Unit (SM in NVIDIA terminology)

TCP: Texture Cache per Pipe (private L1 data)

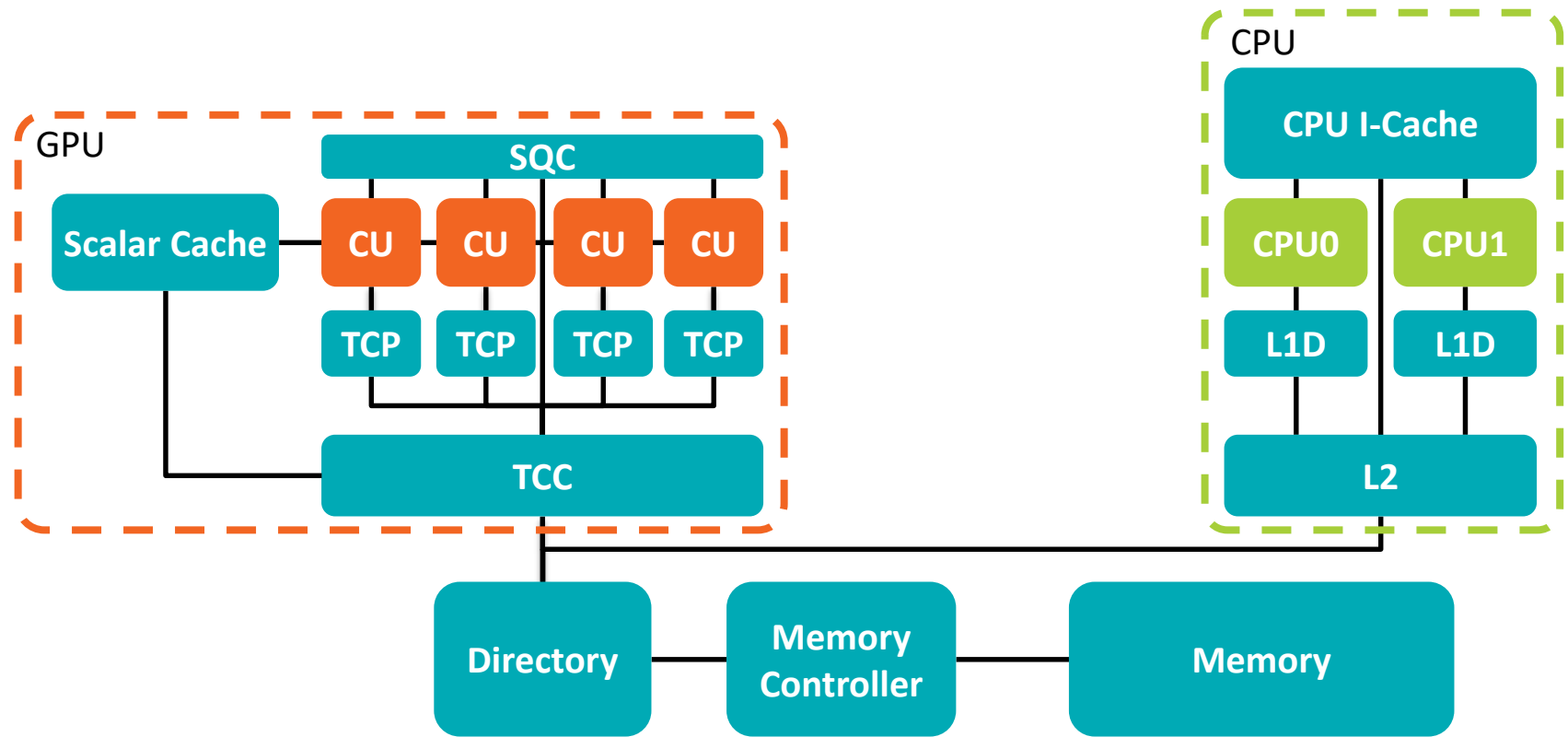TCC: Texture Cache per Channel (shared L2)

AMD terminology

Not shown (per GPU core):

LDS: Local Data Share (Shared memory in NVIDIA terminology, sometimes called "scratch pad")

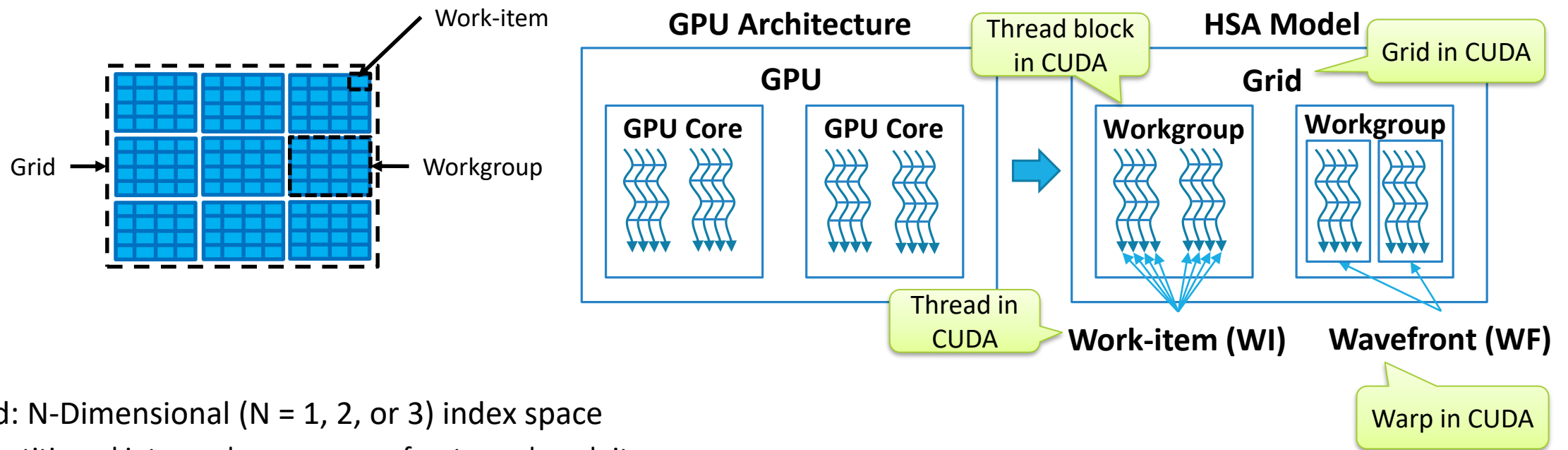# EXAMPLE APU SYSTEM

GPU + CPU CORE-PAIR WITH A SHARED DIRECTORY

AMD

AMD TERMINOLOGY IN A NUTSHELL

AMD

▲ Heterogeneous Systems Architecture (HSA) programming abstraction

- Standard for heterogeneous compute – supported by AMD hardware
- Light abstractions of parallel physical hardware
- Captures basic HSA and OpenCL constructs, plus much more

Work-item

Grid

Workgroup

GPU Architecture

GPU

GPU Core   **GPU Core**

Thread block in CUDA

HSA Model

Grid in CUDA

Grid

Workgroup   **Workgroup**

Thread in CUDA

Work-item (WI)   **Wavefront (WF)**

Warp in CUDA

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

- Partitioned into workgroups, wavefronts, and work-items

SPECIFICATION BUILDING BLOCKS

AMD

HSA Hardware Building Blocks

▲ **Shared Virtual Memory**
 – Single address space
 – Coherent
 – Pageable
 – Fast access from all components
 – Can share pointers

▲ **Architected User-Level Queues**

◢ **Signals**

◢ **Platform Atomics**

◢ **Defined Memory Model**

◢ **Context Switching**

Open-Source

HSA Platform System Arch Specification

Industry standard, architected requirements for how devices share memory and communicate with each other

HSA Software Building Blocks

◢ **HSA Runtime**
 – *Implemented by the ROCm runtime*
 – Create queues
 – Allocate memory
 – Device discovery

◢ **Multiple high-level compilers**
 – CLANG/LLVM
 – C++, HIP, OpenMP, OpenACC, Python

◢ **GCN3 Instruction Set Architecture**
 – Kernel state
 – ISA encodings
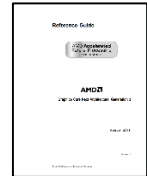 – Program flow control

Open-Source

HSA System Runtime Specification

Open-Source

GCN3 ISA Specification

Industry specifications to enable existing programming languages to target the GPU

http://hsafoundation.com
http://github.com/HSAFoundation

APU SIMULATION SUPPORT

AMD

HSA Hardware Building Blocks

- Shared virtual memory
  - Single address space
  - Coherent
  - Fast access from all components
  - Can share pointers
  - Pageable
- Architected user-level queues
  - Via architected queuing language (AQL)
- Signals
- Platform atomics
- Defined memory model
  - Acquire and release semantics as implemented by the compiler
  - Merging functional and timing models
- Context switching

HSA Software Building Blocks

- Radeon Open Compute platform (ROCm)
  - AMD's implementation of HSA principles
  - Create queues
  - Device discovery
  - AQL support
  - Allocate memory
- Machine ISA
  - GCN3
- Heterogeneous Compute Compiler (HCC)
  - CLANG/LLVM – direct to GCN3 ISA
  - C++, C++ AMP, HIP, OpenMP, OpenACC, Python

Legend
Included in this release
Work-in-progress / may be released
Longer term work

OUTLINE

AMD

Topic	Presenter	Time
Background	Tony	8:00 – 8:15
ROCm, GCN3 ISA, and GPU Arch	**Tony**	**8:15 – 9:15**
HSA Implementation in gem5	Sooraj	9:15 – 10:00
Break		10:00 – 10:30
Ruby and GPU Protocol Tester	Tuan	10:30 – 11:15
Demo and Workloads	Matt	11:15 – 11:50
Summary and Questions	All	11:50 – 12:00

HW-SW INTERFACES

◢ ROCm – high-level SW stack

◢ HW-SW interfaces

◢ Kernel launch flow

◢ GCN3 ISA overview

ARE YOU READY TO ROCm?

SW STACK AND HIGH-LEVEL SIMULATION FLOW

AMD

▲ HCC
- Clang front end and LLVM-based backend
  - Direct to ISA
  - Multi-ISA binary (x86 + GCN3)

▲ ROCm Stack
- HCC libraries
- Runtime layer – **ROCr**
- Thunk (user space driver) – **ROCt**
- Kernel fusion driver (KFD) – **ROCk**

▲ GPU is a HW-SW co-designed machine
- Command processor (CP) HW aids in implementing HSA standard
- Rich application binary interface (ABI)

▲ GPU directly executes GCN3 ISA
- Runtime ELF loaders for GCN3 binary

```
shader.[hh|cc]
compute_unit.[hh|cc]
gpu_command_processor.[hh|cc]
```

See https://rocm.github.io for documentation, source, and more.

Application source → HCC → GCN3 ELF + Code metadata / x86 ELF

User space: HCC Libraries → ROCr → ROCt

OS kernel space: ROCk

Runtime loader loads GCN3 ELF into memory

Hardware / **GPU:** MEM, Command Processor, CU

DETAILED VIEW OF KERNEL LAUNCH

GPU FRONTEND AND HW-SW INTERFACE

▲ User space SW talks to GPU via ioctl()
  - HCC/ROCr/ROCt are off-the-shelf ROCm
  - ROCk is emulated in gem5
    - Handles ioctl commands

▲ CP frontend
  - Two primary components:
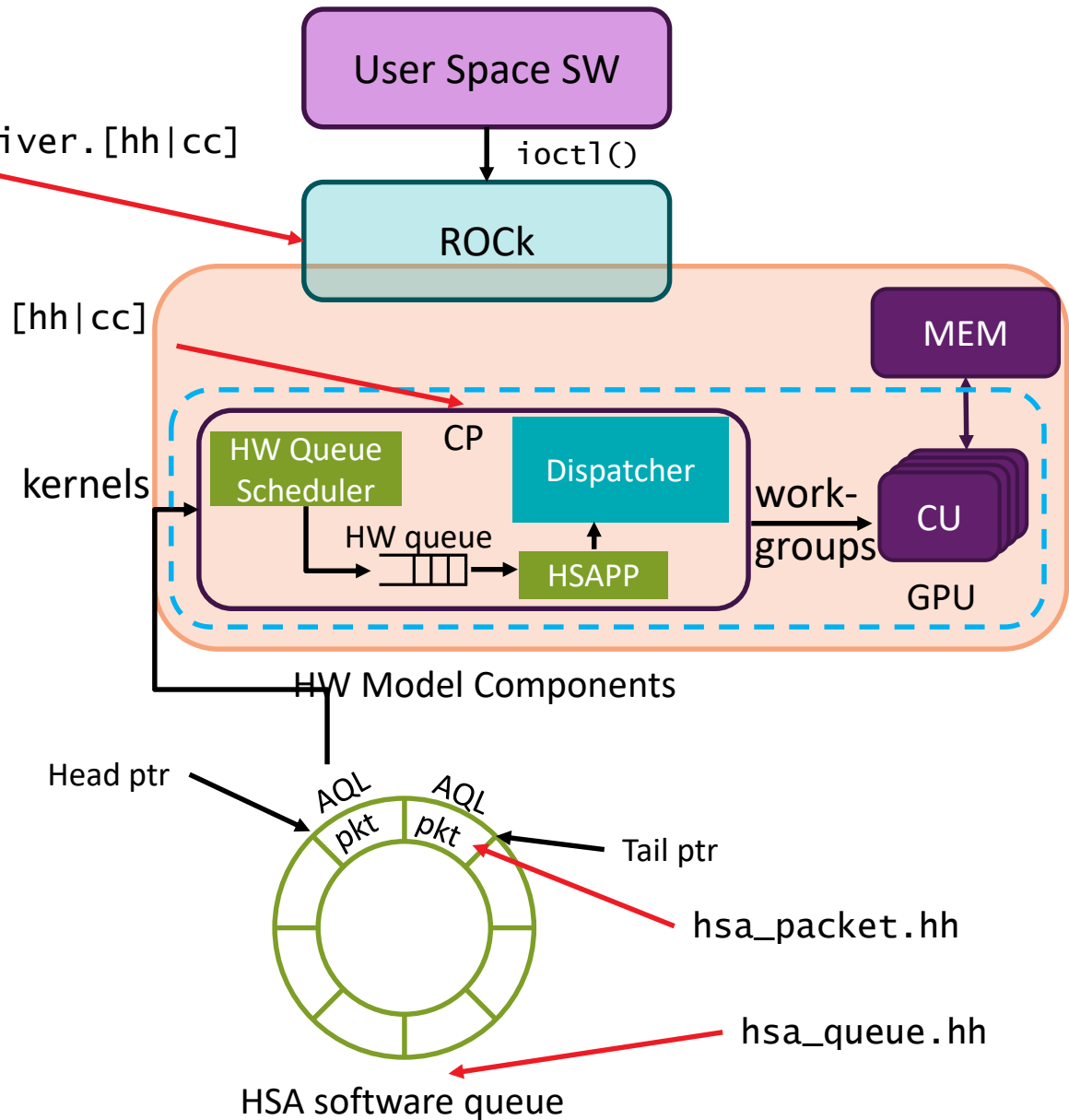   - HSA packet processor (HSAPP)
    - Workgroup dispatcher

▲ 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...

`gpu_compute_driver.[hh|cc]`

`dev/hsa/hsa_packet_processor.[hh|cc]`
`dev/hsa/hw_scheduler.[hh|cc]`

kernels

Head ptr

AQL pkt   AQL pkt   Tail ptr

`hsa_packet.hh`

`hsa_queue.hh`

HSA software queue

User Space SW

`ioctl()`

ROCk

MEM

CP

HW Queue Scheduler   Dispatcher

HW queue   HSAPP   work-groups   CU
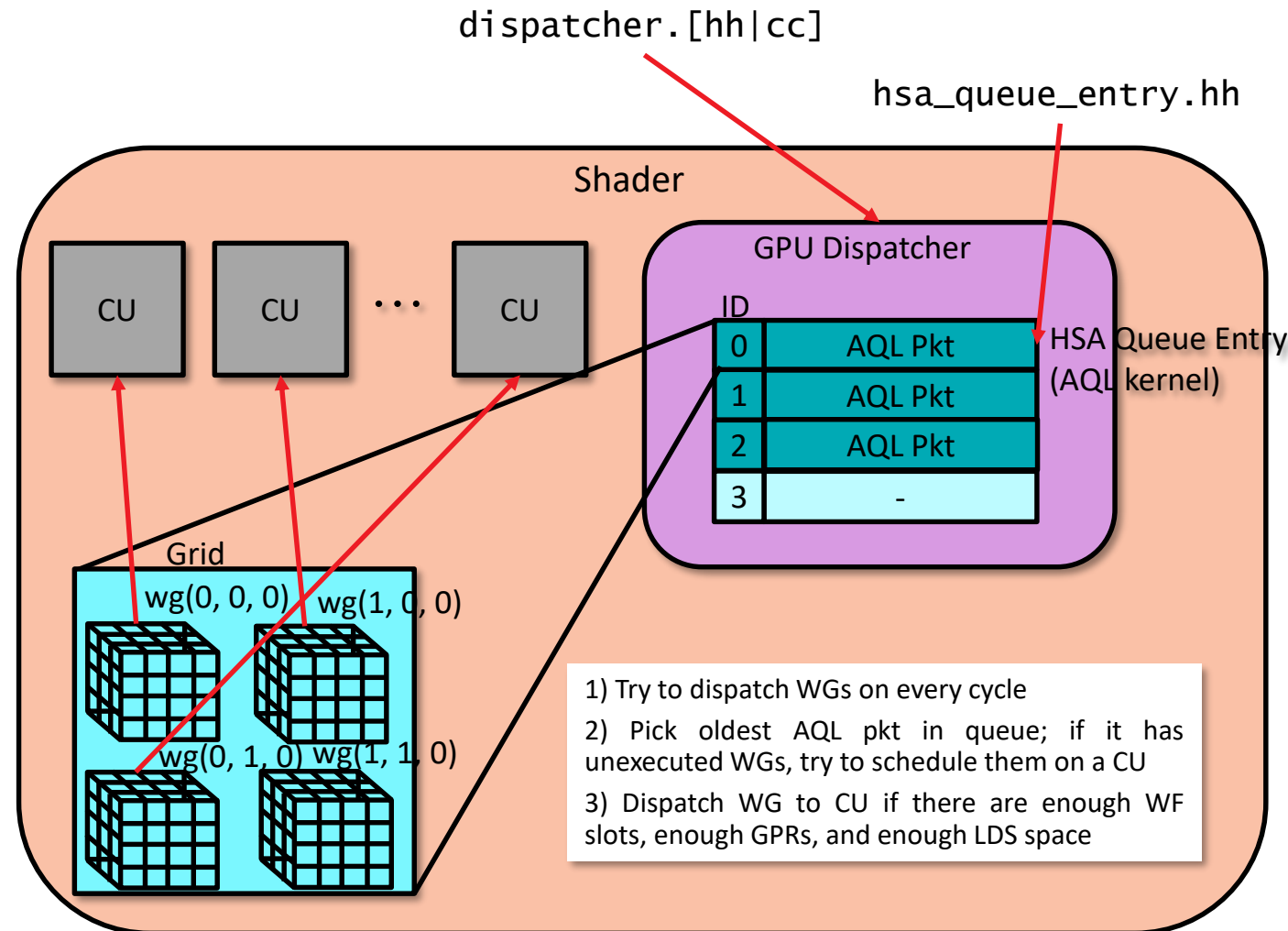
GPU

HW Model Components

DETAILED VIEW OF KERNEL LAUNCH

DISPATCHER WORKGROUP ASSIGNMENT

◤ 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

`dispatcher.[hh|cc]`

`hsa_queue_entry.hh`

Shader

GPU Dispatcher

CU CU · · · CU

ID

0	AQL Pkt
1	AQL Pkt
2	AQL Pkt
3	-

HSA Queue Entry (AQL kernel)

Grid

wg(0, 0, 0) wg(1, 0, 0)

wg(0, 1, 0) wg(1, 1, 0)

1) Try to dispatch WGs on every cycle

2) Pick oldest AQL pkt in queue; if it has unexecuted WGs, try to schedule them on a CU

3) Dispatch WG to CU if there are enough WF slots, enough GPRs, and enough LDS space

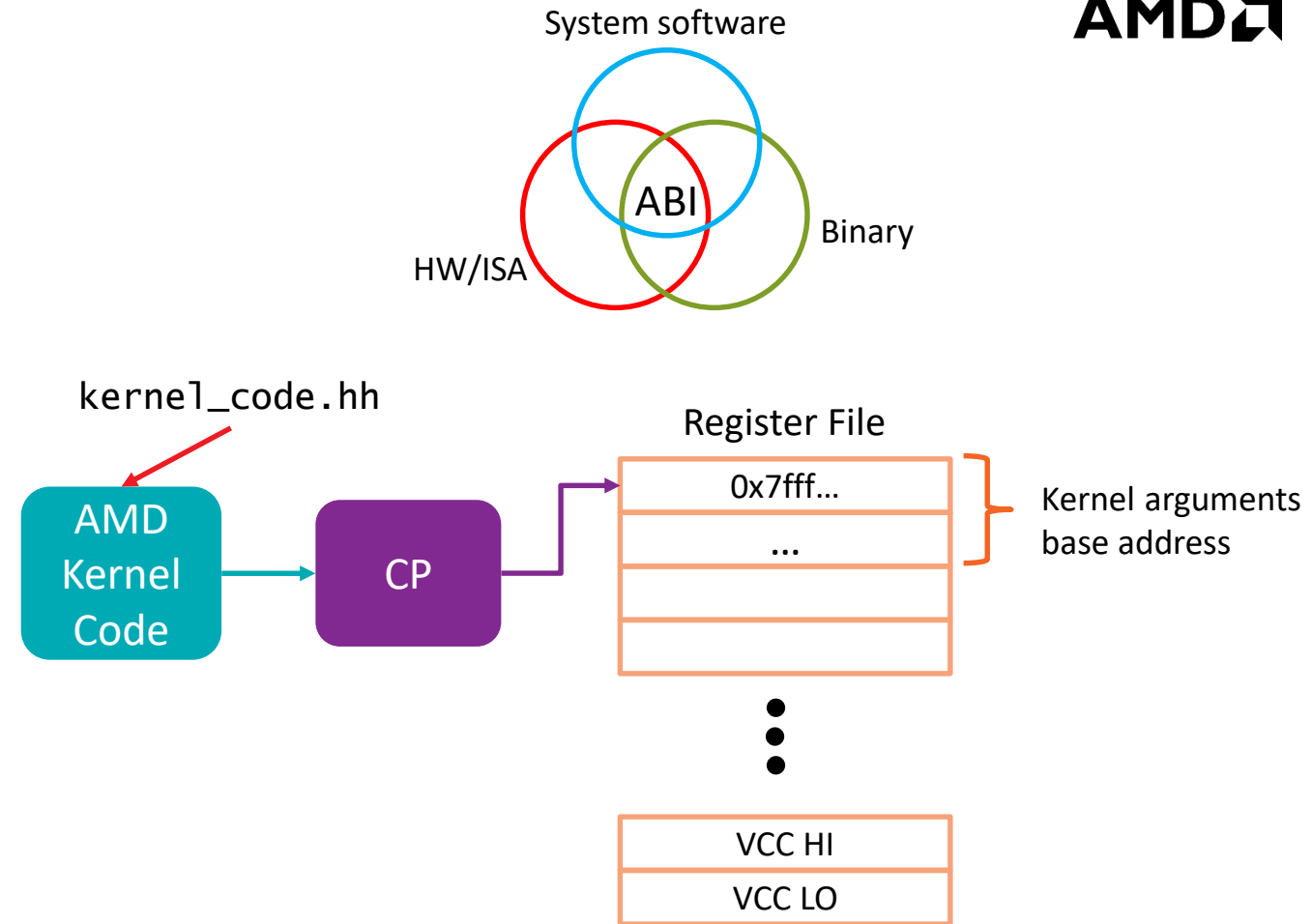# DETAILED VIEW OF KERNEL LAUNCH

GPU ABI INITIALIZATION

- ABI
  - Interface between the application binary (ELF) and other components (e.g., OS, ISA, and HW)
  - Function call convention, location of special values, etc.

- ABI state primarily maintained in register file
  - Some state maintained in the WF context for simplicity

- CP responsible for initializing register file state
  - Extracts metadata from code object (GPU ELF binary)
  - Initializes both scalar and vector registers
  - Kernel argument base address, vector condition codes (VCC), etc.

System software

ABI

HW/ISA

Binary

kernel_code.hh

AMD Kernel Code → CP → Register File

Register File

| 0x7fff... |
| ... |

Kernel arguments base address

VCC HI

VCC LO

GCN3 ISA application loading a kernel argument

```
# Load 2nd 64b kernel arg into s[2:3]
s_load_dwordx2 s[2:3], s[0:1], 0x08
```
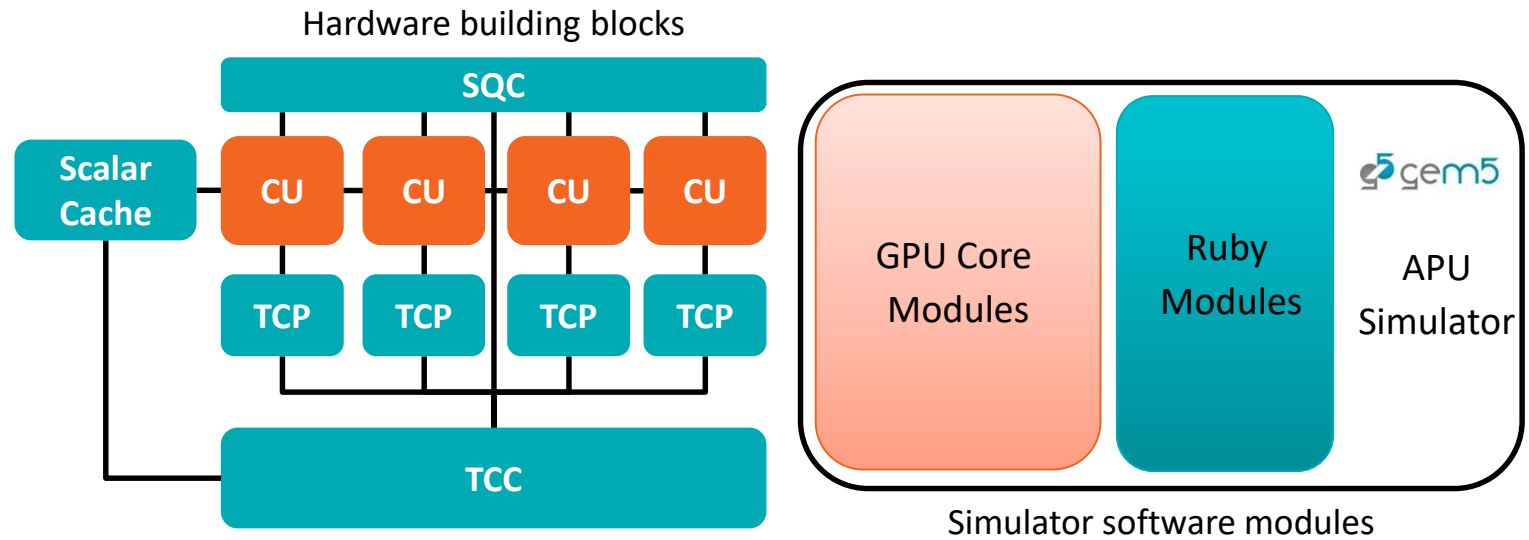
GPU MICROARCHITECTURE

◢ High-level gem5 and Ruby core models

◢ gem5 ISA/HW separation
  – Object oriented design
    – Modular, extensible…
    – Possible to support multiple-ISAs

◢ GCN microarchitecture

◢ gem5's conceptual pipeline and timing flow

◢ GCN3 ISA description

◢ gem5 compute unit implementation
  – Pipeline breakdown
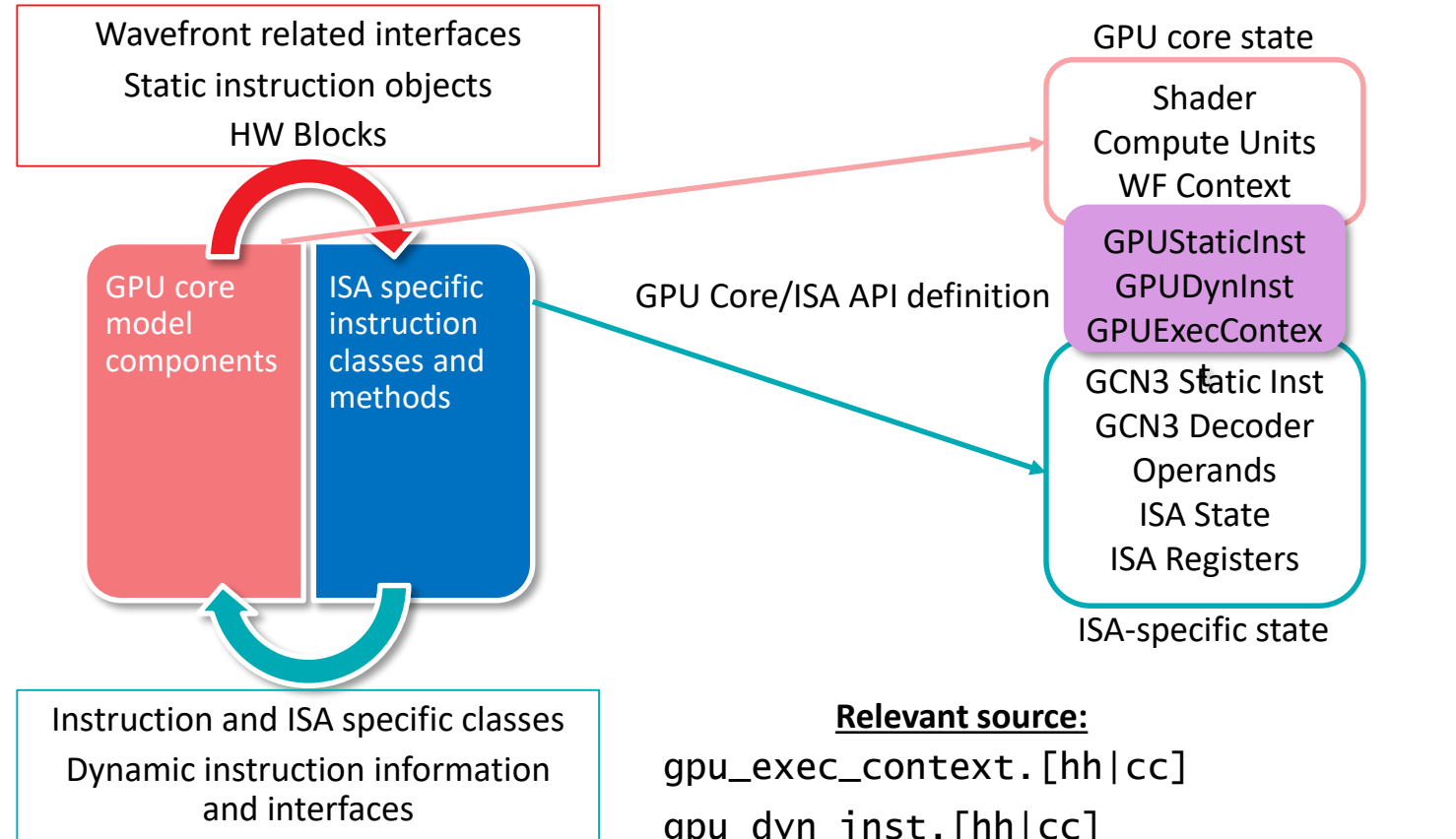
GPU CORE MODULES
GPU CORE MODULES VS. RUBY MODULES

- ▲ GPU core is the compute unit
  - Resources inside GPU Core
    - Instruction buffering, Registers, Vector ALUs
  - Resources outside GPU Core
    - TCP, TCC, SQC (Ruby based)

- ▲ Shader: object containing all GPU core models
  - The top-level view of the GPU in gem5
  - Contains multiple CUs
  - With other miscellaneous components

Hardware building blocks

Simulator software modules

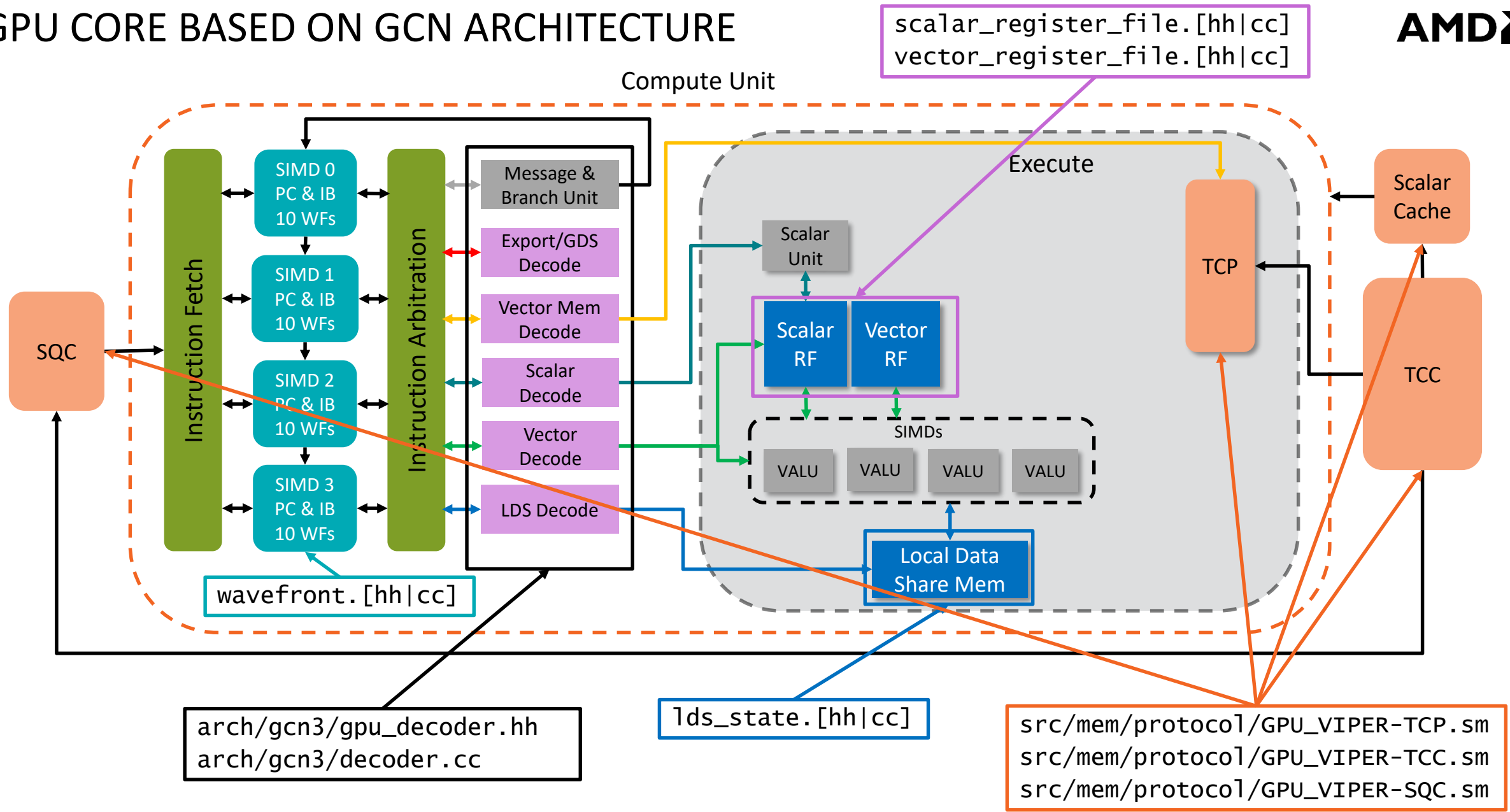# ISA DESCRIPTION/MICROARCHITECTURE SEPARATION

AMD

- ◢ GPUStaticInst & GPUDynInst
  - Architecture-specific code src/arch/
    - Base instruction classes
    - ISA decoder
    - ISA state, etc.
  - Define API for instruction execution
    - e.g., execute() – perform instruction execution
  - Implemented by ISA-specific instruction classes

- ◢ GPUExecContext & GPUISA
  - Define API for accessing ISA state

- ◢ Object-oriented design
  - Base classes define the API
  - Model uses base class pointers
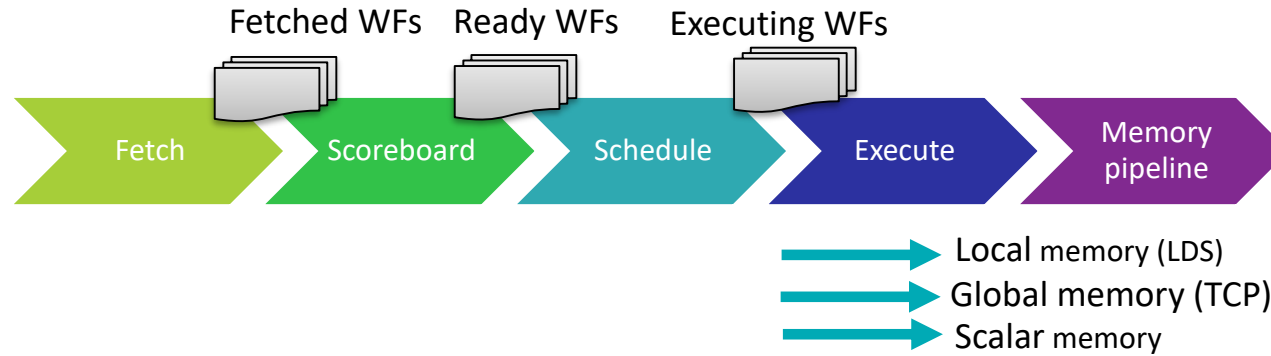 - Configuration system instantiates ISA-specific objects

Wavefront related interfaces
Static instruction objects
HW Blocks

GPU core model components

ISA specific instruction classes and methods

Instruction and ISA specific classes
Dynamic instruction information and interfaces

GPU core state

Shader
Compute Units
WF Context

GPUStaticInst
GPUDynInst
GPUExecContex t

GPU Core/ISA API definition

GCN3 Static Inst
GCN3 Decoder
Operands
ISA State
ISA Registers

ISA-specific state

Relevant source:
`gpu_exec_context.[hh|cc]`
`gpu_dyn_inst.[hh|cc]`
`src/arch/gcn3/gpu_static_inst.hh|cc]`
`src/arch/gcn3/gpu_isa.hh,`
`src/arch/gcn3/isa.cc`
`src/arch/gcn3/operand.hh`
`src/arch/gcn3/registers.[hh|cc]`

GPU CORE BASED ON GCN ARCHITECTURE

AMD

Compute Unit

Execute

scalar_register_file.[hh|cc]
vector_register_file.[hh|cc]

SIMD 0 PC & IB 10 WFs

SIMD 1 PC & IB 10 WFs

SIMD 2 PC & IB 10 WFs

SIMD 3 PC & IB 10 WFs

Instruction Fetch

Instruction Arbitration

Message & Branch Unit

Export/GDS Decode

Vector Mem Decode

Scalar Decode

Vector Decode

LDS Decode

Scalar Unit

Scalar RF

Vector RF

SIMDs

VALU VALU VALU VALU

Local Data Share Mem

TCP

Scalar Cache

TCC

SQC

wavefront.[hh|cc]

arch/gcn3/gpu_decoder.hh
arch/gcn3/decoder.cc

lds_state.[hh|cc]

src/mem/protocol/GPU_VIPER-TCP.sm
src/mem/protocol/GPU_VIPER-TCC.sm
src/mem/protocol/GPU_VIPER-SQC.sm

More details available here: GCN Architecture Whitepaper www.amd.com/Documents/GCN_Architecture_whitepaper.pdf

GPU CORE TIMING
CONCEPTUAL TIMING STAGES

AMD

Fetched WFs   Ready WFs   Executing WFs

Fetch → Scoreboard → Schedule → Execute → Memory pipeline

→ Local memory (LDS)
→ Global memory (TCP)
→ Scalar memory

◢ Execute-in-execute philosophy

◢ 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 LDS/global memory operation
    – `local_memory_pipeline.[hh|cc]`
    – `global_memory_pipeline.[hh|cc]`
    – `scalar_memory_pipeline.[hh|cc]`

GCN3 GPU ISA

`arch/gcn3/insts/op_encodings.[hh|cc]`

`arch/gcn3/insts/instructions.[hh|cc]`

▲ **Vector and scalar instructions**
- Single instruction stream
- <u>Not really "SIMT"</u>
- Divergence handled by scalar unit
  - Can directly modify execution mask
  - Jump over basic blocks when EXEC = 0

▲ **Instructions broken down by OP type**
- Op types map to different functional units in CU
- The CU can issue one instruction to each unit in the same cycle
  - Export/GDS not supported in gem5

Op Type	Functional Unit	Usage
SOPP	Branch/Message Unit	Branching, NOPs, Barriers, waitcnts, messaging
SOPC/SOPK/SOP1/SOP2	Scalar ALU	General scalar computation/divergence handling
SMEM	Scalar Memory	Scalar memory access, cache maintenance
VOPC/VOP1/VOP2/VOP3	SIMD Unit	General SIMD computation
DS	LDS	Private scratch pad memory
MUBUF	Vector Memory	Accessing vector memory, cache maintenance
FLAT	LDS/Vector Memory	Accessing vector memory, may resolve to LDS or system memory
VINTRP/MTBUF/MIMG/EXP	Varies	Primarily used by graphics. Not currently modeled in gem5; however, the infrastructure to do so is present.

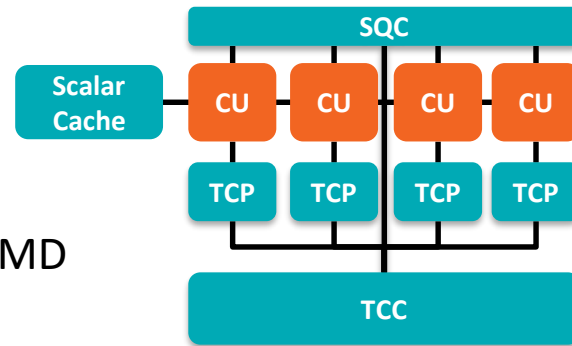Full GCN3 spec available at: https://gpuopen.com/compute-product/amd-gcn3-isa-architecture-manual/

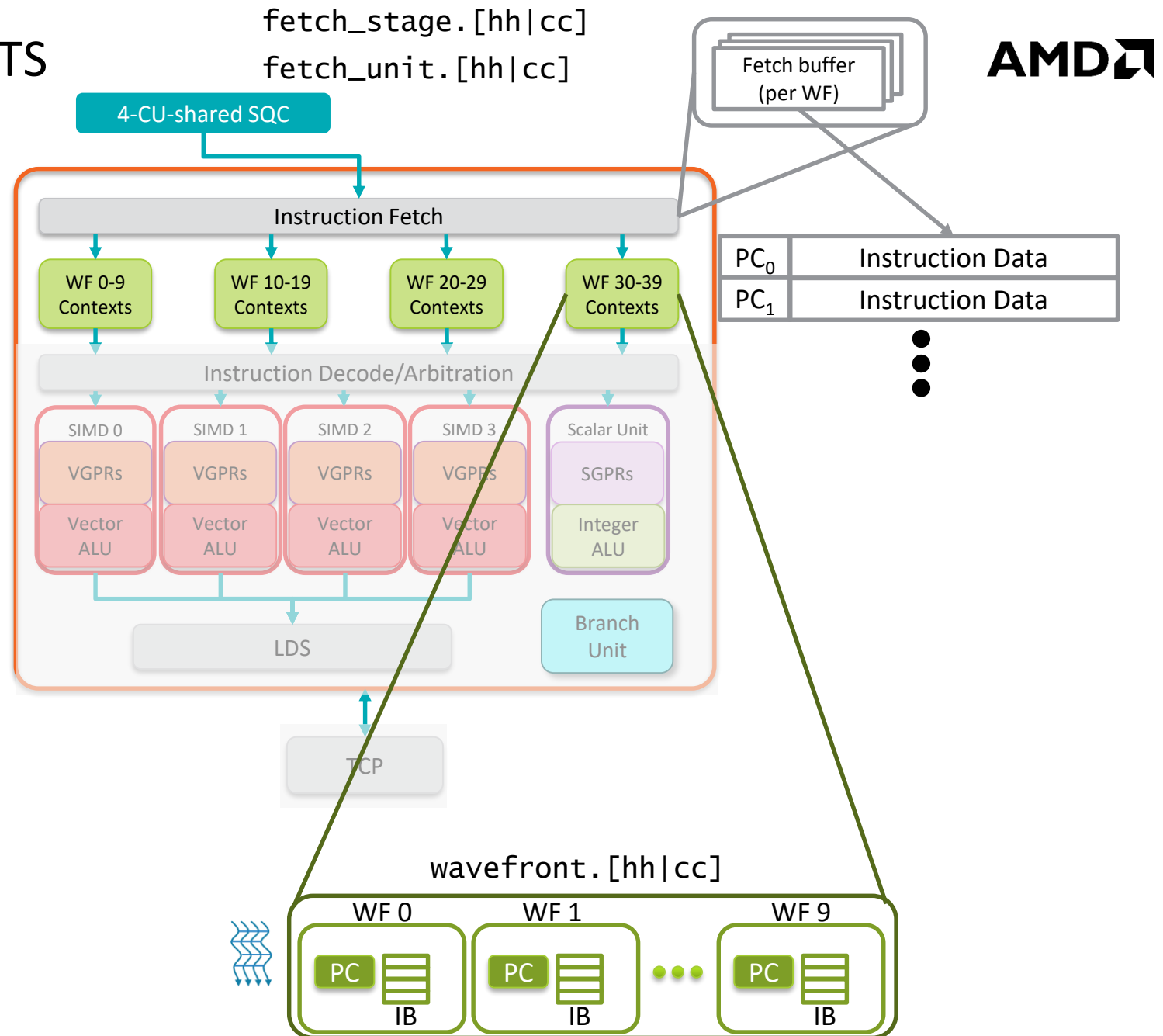# GPU CORE MODULE INTERNALS

SHARED VS. PRIVATE STRUCTURES

◢ Compute unit
- Four 16-wide SIMD units
- SIMD hosts WFs
- **Private** resources to each SIMD
  - Instruction buffering
  - Registers
  - Vector ALUs
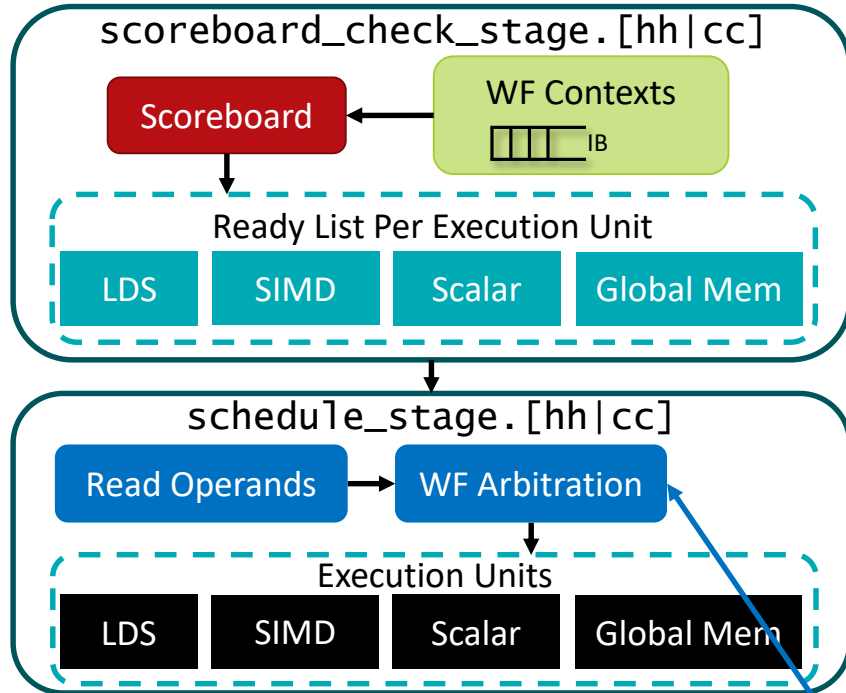- **Shared** resources
  - Fetch and decode
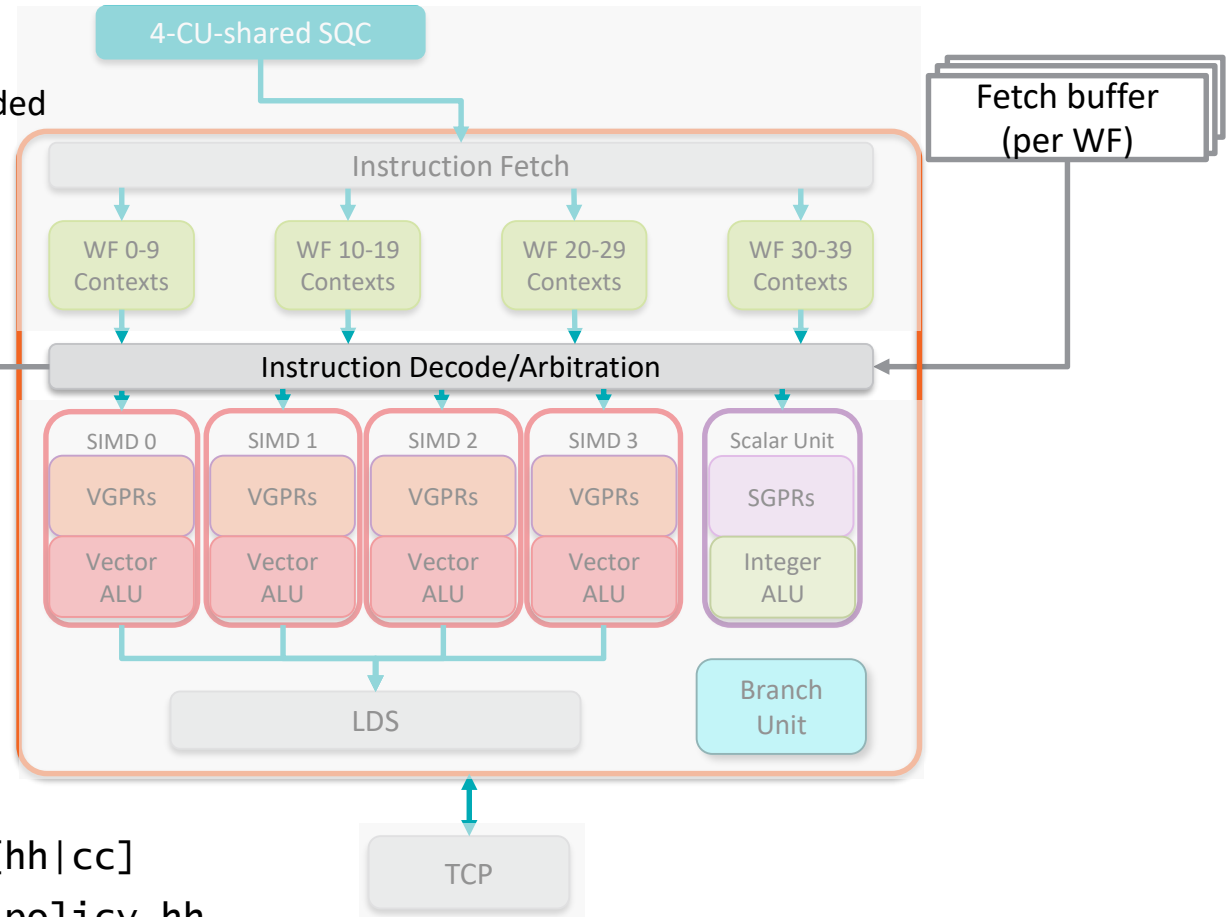 - TCP
  - LDS
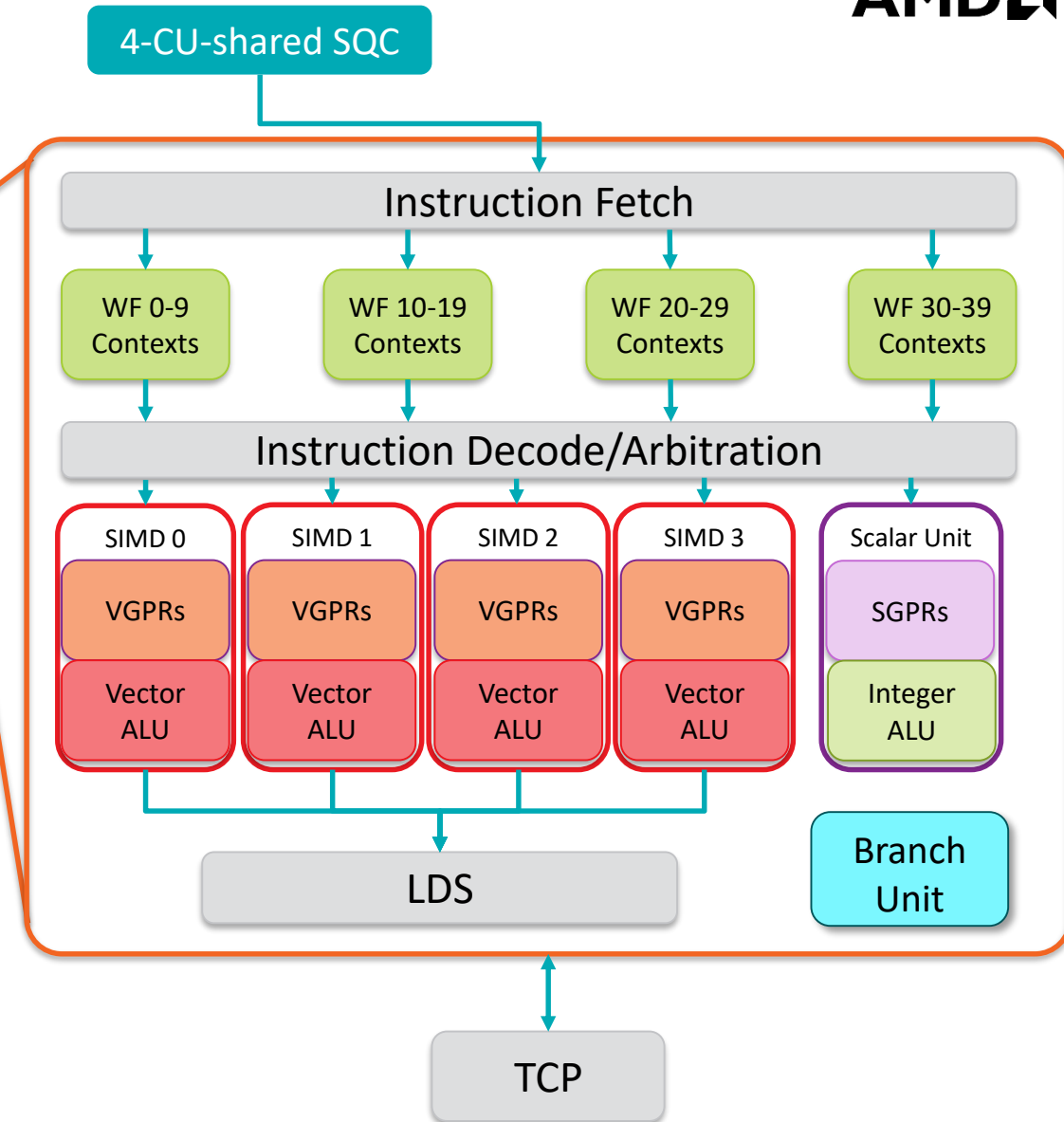
FETCH AND WAVEFRONT CONTEXTS

```
fetch_stage.[hh|cc]
fetch_unit.[hh|cc]
```

Fetch buffer (per WF)

4-CU-shared SQC

- ◤ SQC shared by 4 CUs
  - – Number of SQCs and CUs are configurable

- ◤ Fetch
  - – Shared and arbitrated between SIMDs in a CU
  - – Fetch to each SIMD unit
  - – Buffers fetched cache lines per WF

- ◤ WF Contexts
  - – 10 WFs per SIMD, 40 per CU
    - – PC and decoded instruction buffers (IB)
    - – Register file and LDS allocation
  - – ¼ of WF executes each cycle
    - – 4 cycles needed to fully execute single SIMD instruction

Instruction Fetch

| WF 0-9 Contexts | WF 10-19 Contexts | WF 20-29 Contexts | WF 30-39 Contexts |

| PC$_0$ | Instruction Data |
| PC$_1$ | Instruction Data |

Instruction Decode/Arbitration

SIMD 0	SIMD 1	SIMD 2	SIMD 3	Scalar Unit
VGPRs	VGPRs	VGPRs	VGPRs	SGPRs
Vector ALU	Vector ALU	Vector ALU	Vector ALU	Integer ALU

LDS

Branch Unit

TCP

`wavefront.[hh|cc]`

| WF 0 | WF 1 | WF 9 |
| PC IB | PC IB | PC IB |

DECODE AND ISSUE


```
scoreboard_check_stage.[hh|cc]
```

Scoreboard

WF Contexts
IB

Ready List Per Execution Unit
LDS | SIMD | Scalar | Global Mem

```
schedule_stage.[hh|cc]
```

Read Operands → WF Arbitration

Execution Units
LDS | SIMD | Scalar | Global Mem

Instruction data decoded into instruction buffer

4-CU-shared SQC

Instruction Fetch

WF 0-9 Contexts | WF 10-19 Contexts | WF 20-29 Contexts | WF 30-39 Contexts

Instruction Decode/Arbitration

SIMD 0 — VGPRs — Vector ALU
SIMD 1 — VGPRs — Vector ALU
SIMD 2 — VGPRs — Vector ALU
SIMD 3 — VGPRs — Vector ALU
Scalar Unit — SGPRs — Integer ALU

LDS

Branch Unit

TCP

Fetch buffer (per WF)

```
scheduler.[hh|cc]
scheduling_policy.hh
```

◢ Instructions are decoded out of fetch buffers

◢ Instruction arbitration
- Can issue to each functional unit each cycle
- Finds ready WFs
- Scheduling policy dictates which WFs have priority
  - Oldest first, easy to add others

REGISTER FILES

```
vector_register_file.[hh|cc]
scalar_register_file.[hh|cc]
simple_pool_manager.[hh|cc]
static_register_manager_policy.[hh|cc]
```

▲ General Purpose Registers (GPRs)
  – Vector registers (VGPR) partitioned per SIMD
    – Configurable size
    – Because each SIMD executes independent WF
  – 32-bit wide
    – Combine adjacent VGPRs for 64-bit or 128-bit data
  – Each WF also has scalar general purpose registers (SGPRs)

▲ Register Allocation Done by a Simple Pool Manager
  – *Only allows one WG at a time*
  – Statically mapped virtual → physical register index
    – <base, limit> pair of registers specify GPR allocation
  – Modular design – more advanced pool managers can be swapped into the VRF seamlessly
  – Simple timing model with constant delay

SIMD 0

WF0 – <V0, N>
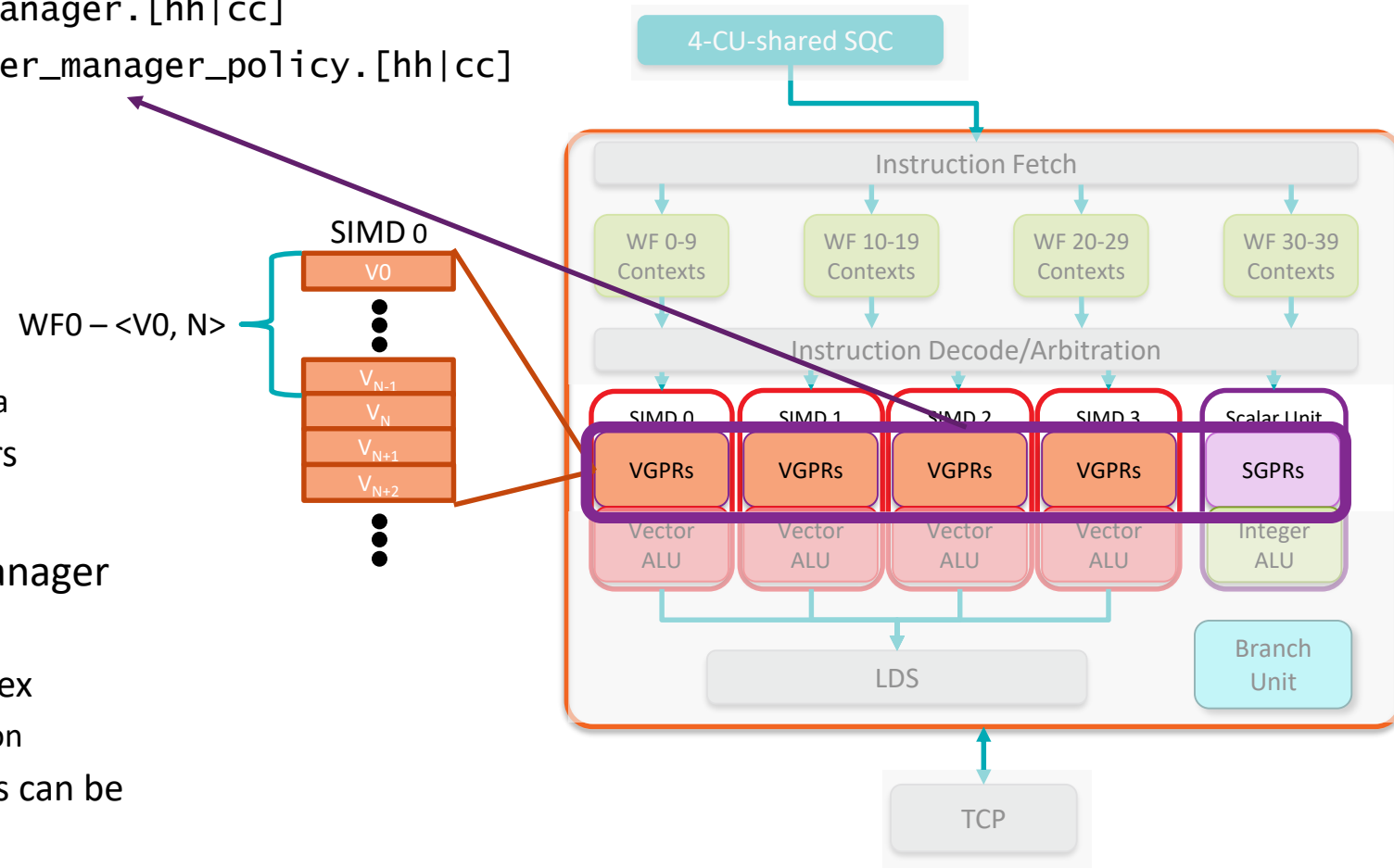
This is an area where gem5 user contributions would be extremely valuable.

VECTOR ALUs

AMD

- 16-lane vector pipeline per SIMD
  - Each lane has a set of functional units
  - One work-item per lane

- 4 cycles to execute a WF for all 64 work-items
  - In gem5, 64 work-items are executed in one tick and ticks are multiplied by 4

- SIMD execution may take longer if work-items in WF have dissimilar behaviors
  - Example 1: Branch (or spatial) divergence
    - Branches executed through predication
    - When control flow diverges, all lanes take all paths
  - Example 2: Memory (or temporal) divergence
    - Longer access latency by one work-item stalls entire WF

compute_unit.[hh|cc]
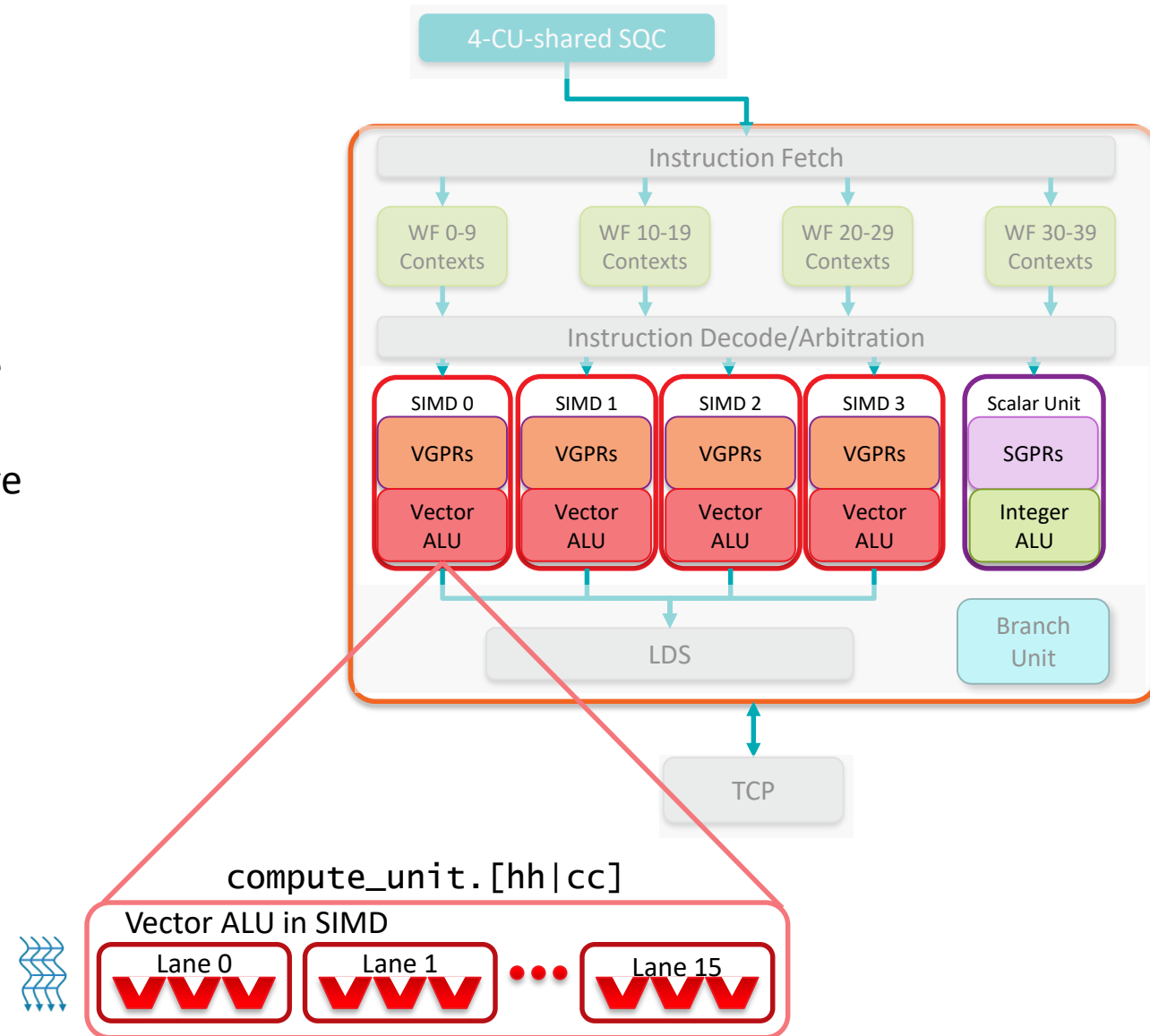
GPU CORE TIMING
HANDLING MEMORY INSTRUCTIONS

```
gpu_dyn_inst.[hh|cc]
arch/gcn3/instructions.[hh|cc]
arch/gcn3/insts/op_encodings[hh|cc]
```

GPU dynamic memory instruction

AMD

| Create packet | Increment waitcnt | Global/LDS operation | Writeback |

- ◣ Memory instructions generate memory requests
  - Part of GPU instruction definition (ISA-specific)

- ◣ Three phases
  - *execute()*
    - Read operands, calculate address, increment wait count, and issue to appropriate memory pipe
  - *initiateAcc()*
    - Issue request to memory system
  - *completeAcc()*
    - For loads write back data. Stores do nothing.

- ◣ New machine ISAs can use this capability to support their own memory instructions

- ◣ Individual stages contribute to the memory instruction timing
  - Additionally memory end timing handled by Ruby and memory technology parameters

- ◣ Memory dependencies are preserved using **waitcnts**

Example GCN3 code:

```
flat_load_dword v4, v[4:5]      ①
flat_load_dword v16, v[8:9]
flat_load_dword v23, v[14:15]
flat_load_dword v10, v[10:11]
s_waitcnt       cnt(3)          ②
v_ashrrev       v5, 31, v4      ③
```

```
local_memory_pipeline.[hh|cc]
global_memory_pipeline.[hh|cc]
scalar_memory_pipeline.[hh|cc]
```

1. flat_load writes v4
2. Waitcnt specifies wait count value must be ≤ 3
3. Arithmetic shift later reads from v4
4. Waitcnt waits until at least #1 is finished

VECTOR MEMORY EXECUTION

- **In gem5:**
  - Address calculation: `arch/gcn3/insts/op_encodings.hh`
  - Address coalescing
    - `mem/ruby/system/GPUCoalescer.[hh|cc]`
    - `mem/ruby/system/VIPERCoalescer.[hh|cc]`
    - `mem/protocol/GPU_VIPER-TCP.sm`
    - `mem/protocol/GPU_VIPER-TCC.sm`

- **LDS**
  - User-managed address space
  - Scratchpad for WFs in workgroup
  - Used for data sharing and synchronization within workgroup
  - Cleared when workgroup completes
  - In gem5, functional model with a pointer per workgroup
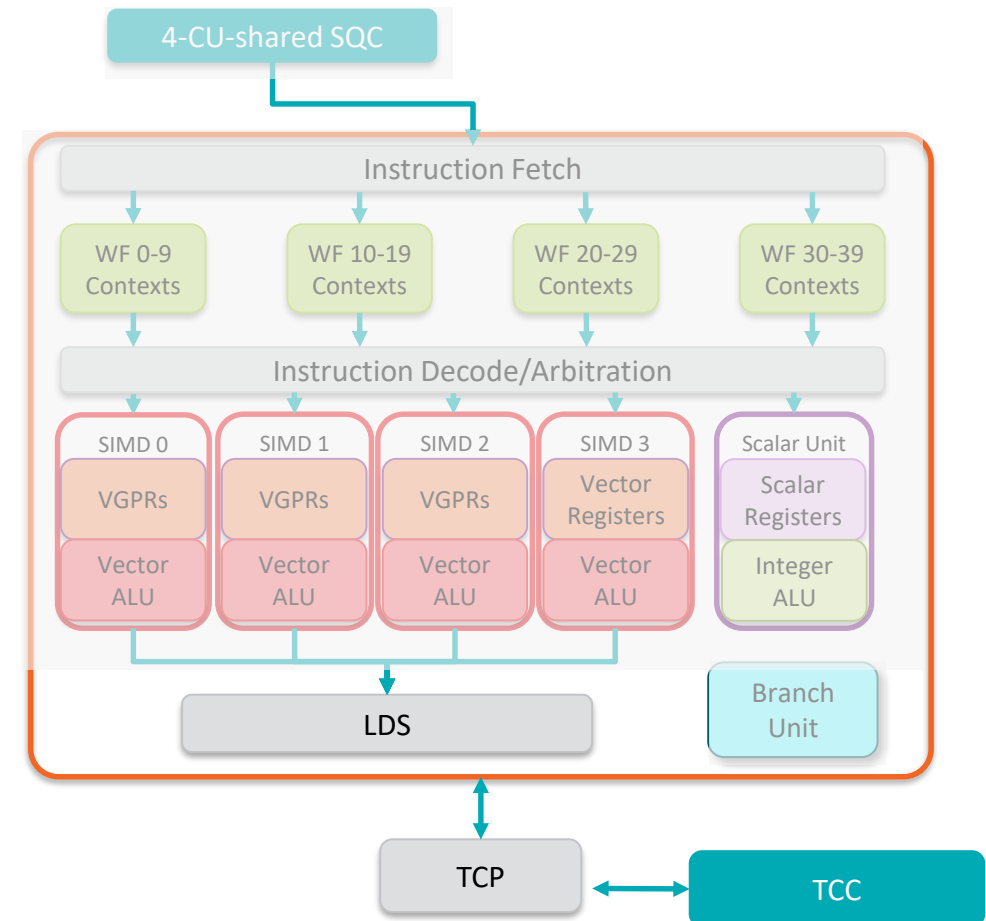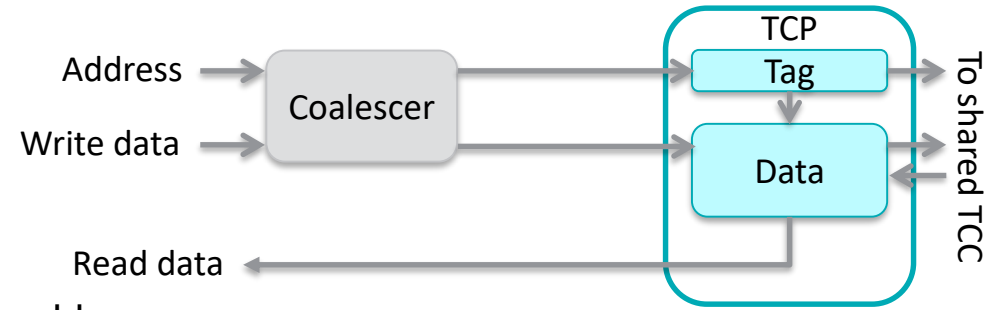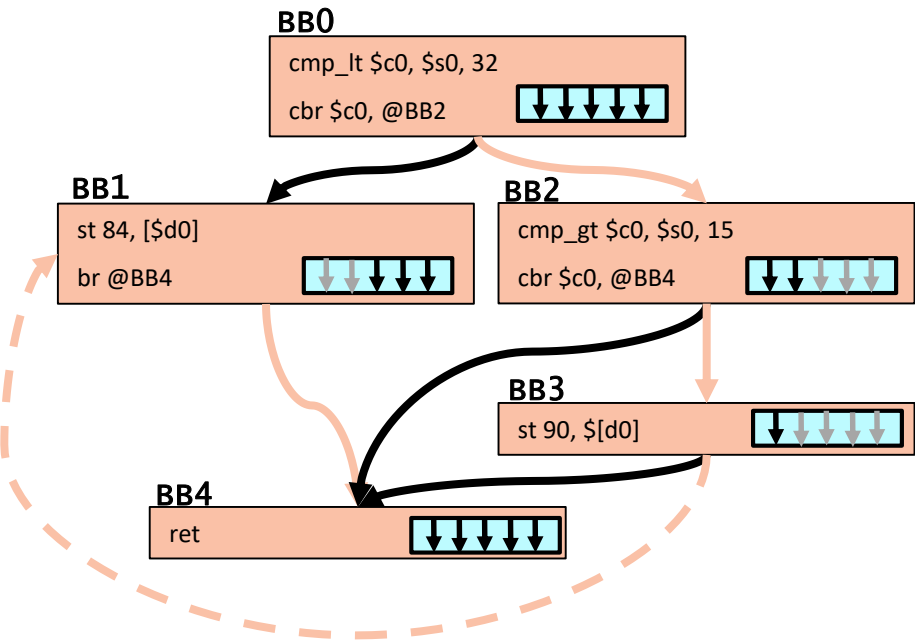
CONTROL FLOW DIVERGENCE

SIMT VS. VECTOR EXECUTION MODEL

Source code:

```
if (i > 31) {
    *x = 84;
} else if (i < 16) {
    *x = 90;
}
```

HSAIL

BB0

| cmp_lt $c0, $s0, 32 |
| cbr $c0, @BB2 |

Execute taken path
first & flush IB

Instruction buffer

| cmp |
| cbr |
| ret |
| br |

Flushed!

BB1

| st 84, [$d0] |
| br @BB4 |

BB2

| cmp_gt $c0, $s0, 15 |
| cbr $c0, @BB4 |

Branch over
BB2 & BB3,
flush IB

Fall through to BB3

BB3

| st 90, $[d0] |

BB4

| ret |
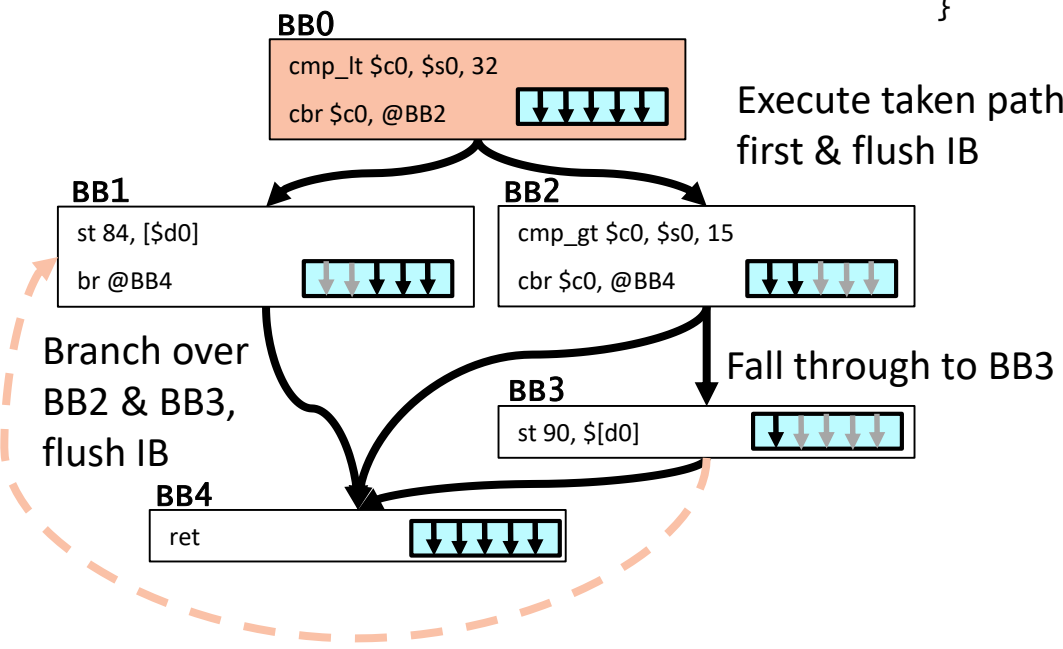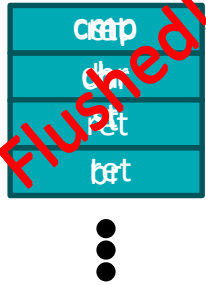
Reconvergence point reached, HW
initiated jump to divergent path

CONTROL FLOW DIVERGENCE
SIMT VS. VECTOR EXECUTION MODEL

AMD

Source code:
```
if (i > 31) {
    *x = 84;
} else if (i < 16) {
    *x = 90;
}
```

HSAIL

BB0
```
cmp_lt $c0, $s0, 32
cbr $c0, @BB2
```

BB1
```
st 84, [$d0]
br @BB4
```

BB2
```
cmp_gt $c0, $s0, 15
cbr $c0, @BB4
```

BB3
```
st 90, $[d0]
```

BB4
```
ret
```

BB0
```
cmp_le vcc, 32, v0
s_load s[0:1], s[6:7], 0x0
s_and_saveexec s[2:3], vcc
v_mov v0, 0x00000054
s_cbranch_execz @BB2
```

BB1
```
s_waitcnt lgkmcnt(0)
v_mov v[1:2], s[0:1]
flat_store v[1:2], v0
```

BB2
```
s_andn2 exec, s[2:3], exec
s_cbranch_execz @BB5
```

BB3
```
v_cmp_ge vcc, 15, v0
s_and_saveexec s[4:5], vcc
v_mov v0, 0x0000005a
s_cbranch_execz @BB5
```

BB4
```
s_waitcnt lgkmcnt(0)
v_mov v[1:2], s[0:1]
flat_store v[1:2], v0
```

BB5
```
s_endpgm
```

Instruction buffer

cmp
s_load
s_and_saveexec
v_mov

Branches are optimizations for case when EXEC = 0 for a BB

OUTLINE

AMD

Topic	Presenter	Time
Background	Tony	8:00 – 8:15
ROCm, GCN3 ISA, and GPU Arch	Tony	8:15 – 9:15
HSA Implementation in gem5	**Sooraj**	**9:15 – 10:00**
Break		10:00 – 10:30
Ruby and GPU Protocol Tester	Tuan	10:30 – 11:15
Demo and Workloads	Matt	11:15 – 11:50
Summary and Questions	All	11:50 – 12:00

WHAT IS HSA?

Heterogeneous System Architecture

Processor design that makes it easy to harness the entire computing power of GPUs for faster and more power-efficient devices, including personal computers, tablets, smartphones, and servers
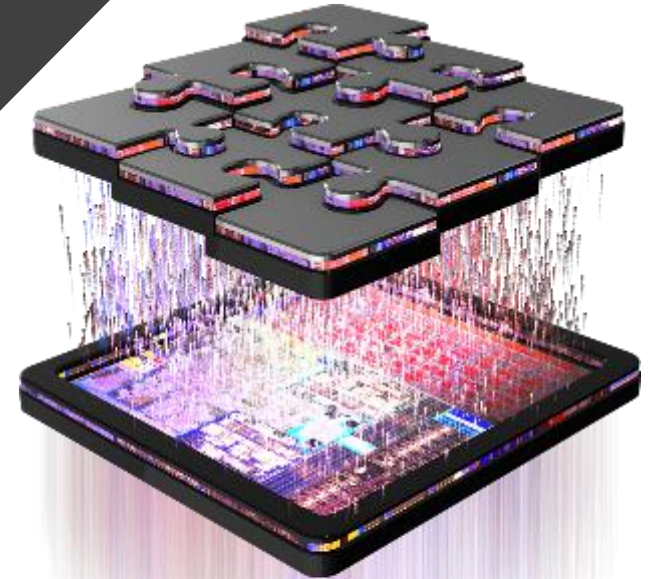
Bringing GPU performance to a wide variety of applications

KEY FEATURES OF HSA

AMD

hUMA — Heterogeneous Unified Memory Architecture

hComm — Heterogeneous Communication via Signals and Atomics

hQ — Heterogeneous Queuing

TRADITIONAL DISCRETE GPU
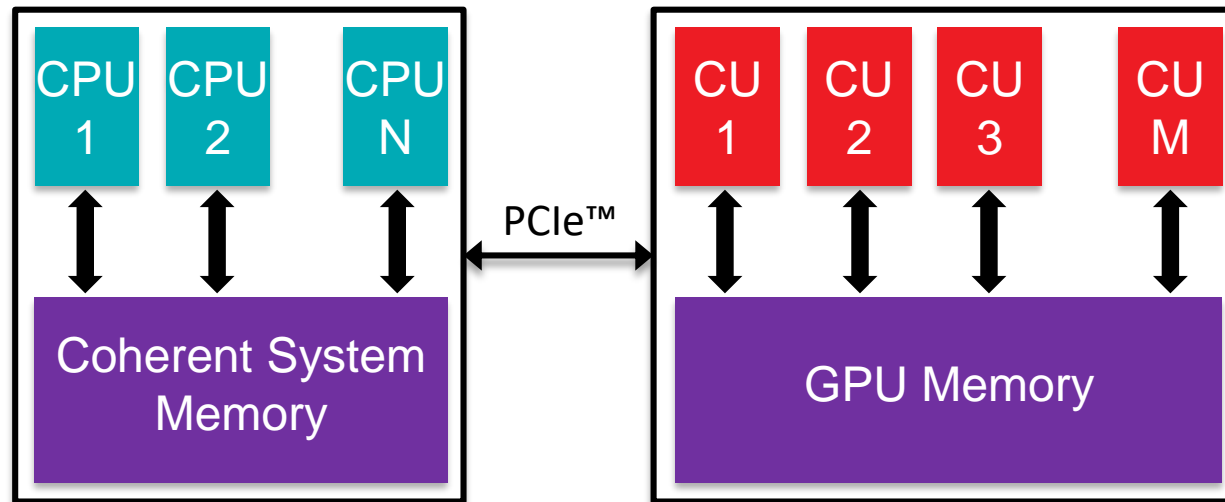
- ◢ Separate memory
- ◢ Separate addr space
  - No pointer-based data structures
- ◢ Explicit data copying
  - High latency
  - Low bandwidth
- ◢ Need lots of compute on GPU to amortize copy overhead
- ◢ Very limited GPU memory capacity

hUMA UNIFIED MEMORY
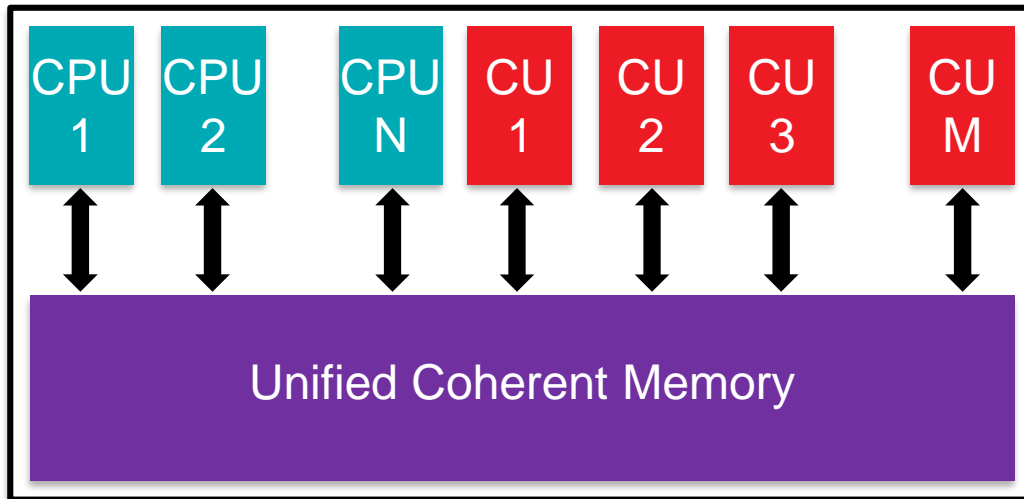
- ◢ Unified address space
  - GPU uses user virtual addresses
  - Fully coherent
- ◢ No explicit copying
  - Data movement on demand
- ◢ Pointer-based data structures shared across CPU & GPU
- ◢ Pageable virtual addresses
  - No GPU capacity constraints

HSA ATOMICS

- Atomic memory updates fundamental to efficient thread synchronization
  - Implement primitives like mutexes, semaphores, histograms, …, previously only implemented on CPU
- HSA supports 32bit or 64bit values for atomic ops
  - CAS, SWAP, add, increment, sub, decrement, …
  - and other common arithmetic and logic atomic ops
- On PCI-Express system, atomics map to PCI-E atomics
- HSA specifies a well-defined "SC for HRF" memory model
  - A variant of "Release Consistency" model
  - Acquire: pull latest data (to me)
  - Release: push latest data (to others)
  - Compatible with C++11, Java, OpenCL, and .NET memory models
  - Details: "HSA Platform System Arch Specification", http://hsafoundation.com
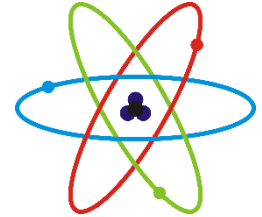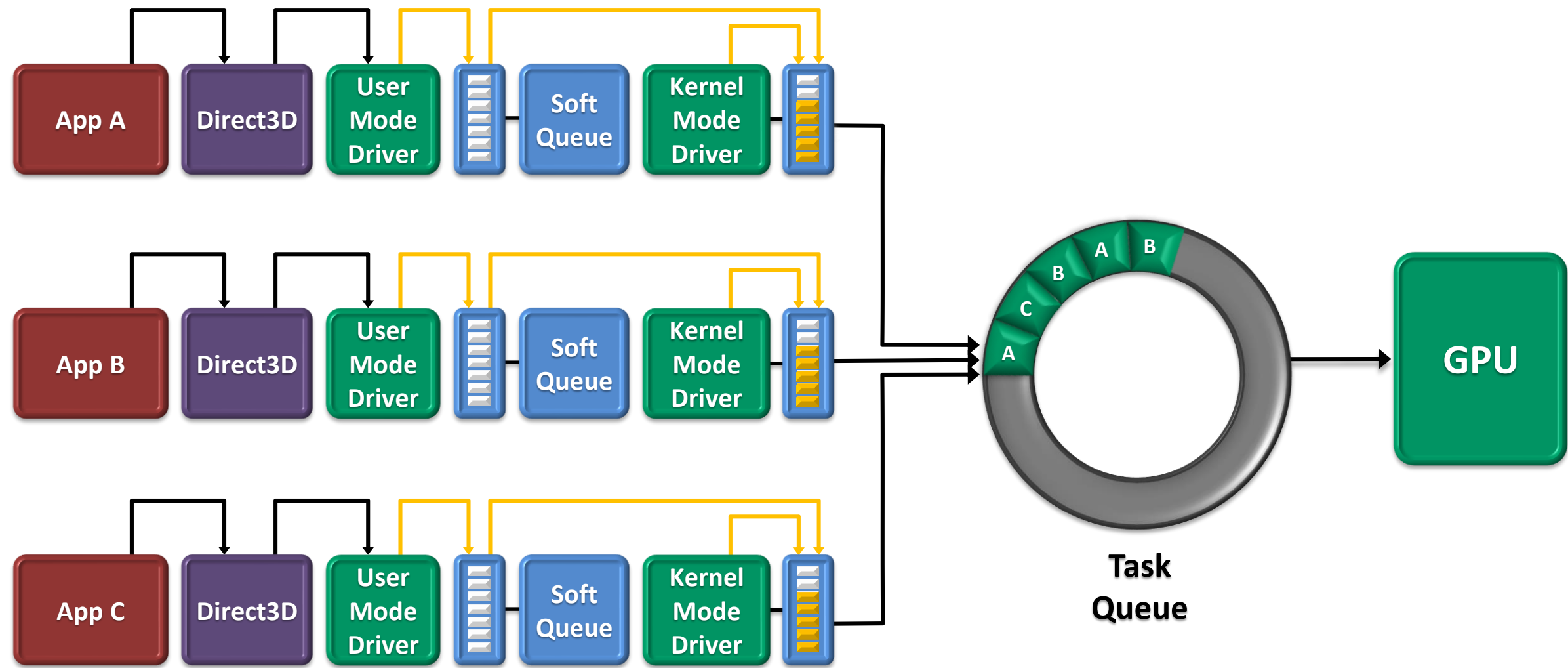
THE HSA SIGNALS INFRASTRUCTURE

▲ Hardware-assisted signaling and synchronization primitives

– Memory semantics, equivalent to atomics

– e.g., 32bit or 64bit value, content updated atomically

– Threads can wait on a value

– Power-efficient synchronization between CPU and GPU threads

▲ Allows one-to-one, many-to-one, and one-to-many signaling

– Used by system software, runtime, and application SW

– Infrastructure to build higher-level synchronization primitives like mutexes, semaphores, etc.

▲ Updating the value of a signal is equivalent to sending the signal

– Release semantics: push data to others

▲ Waiting on a signal is also permitted

– Via a wait instruction or via a runtime API
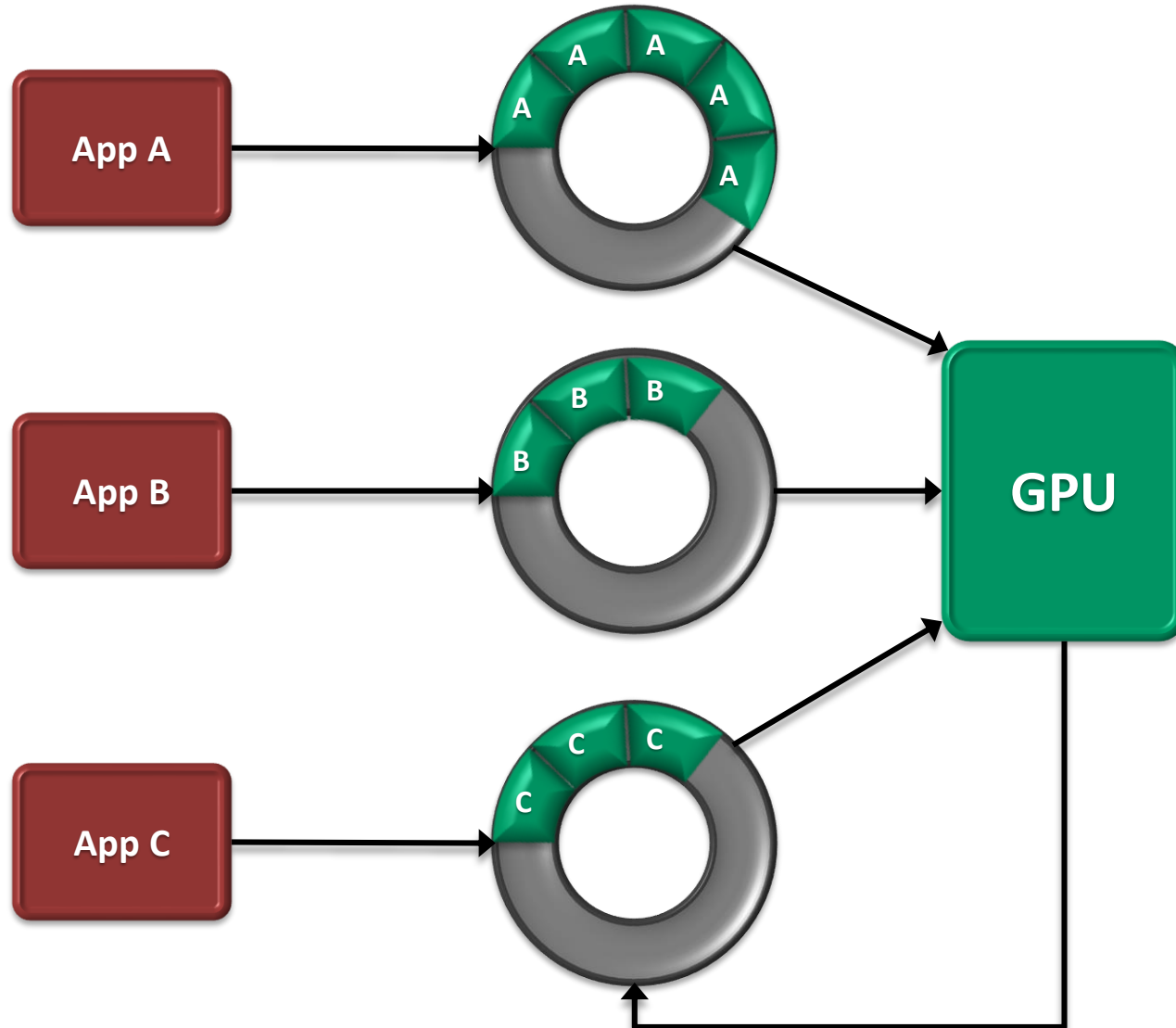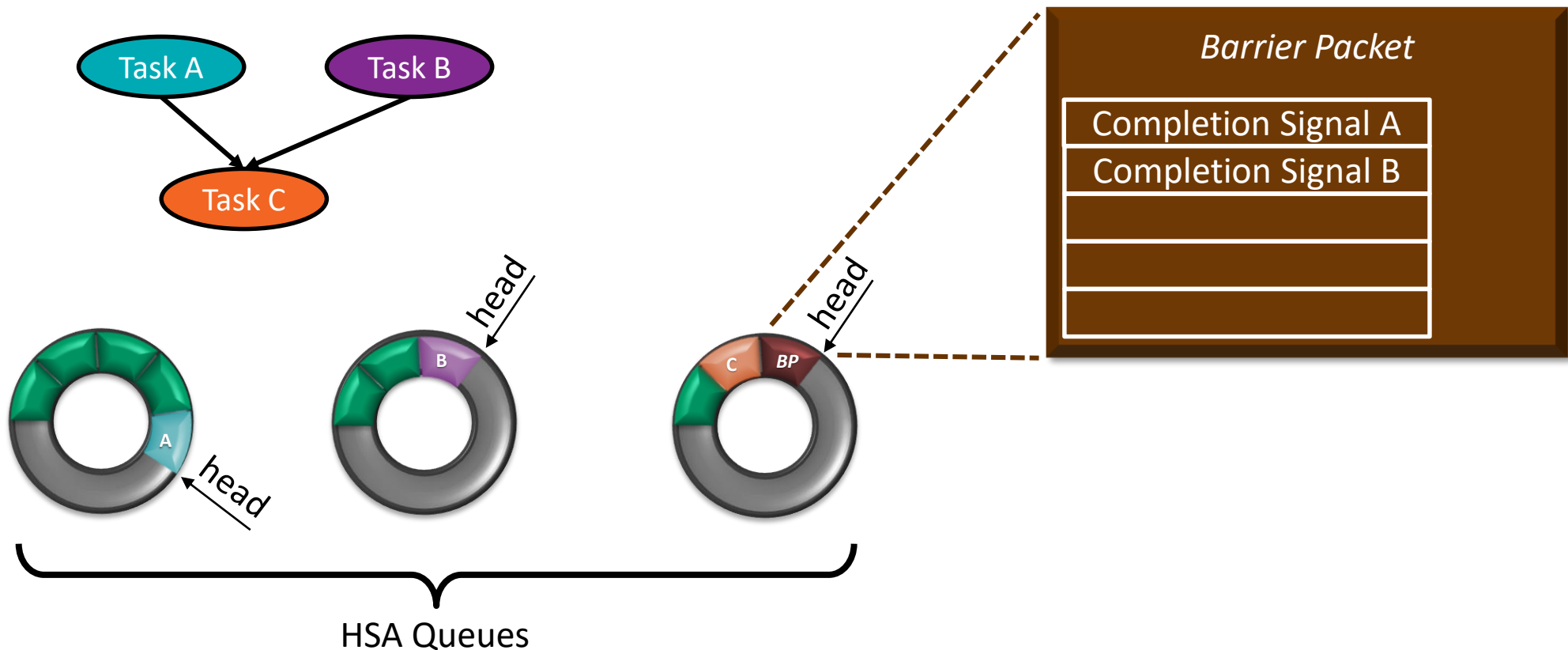
– Acquire semantics: pull data from others
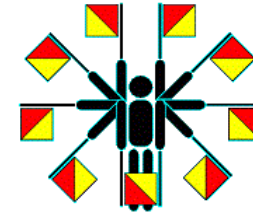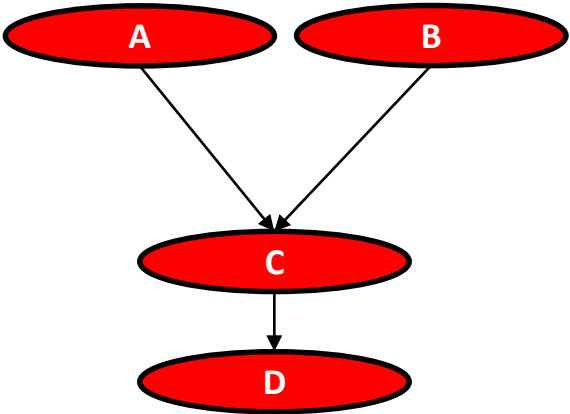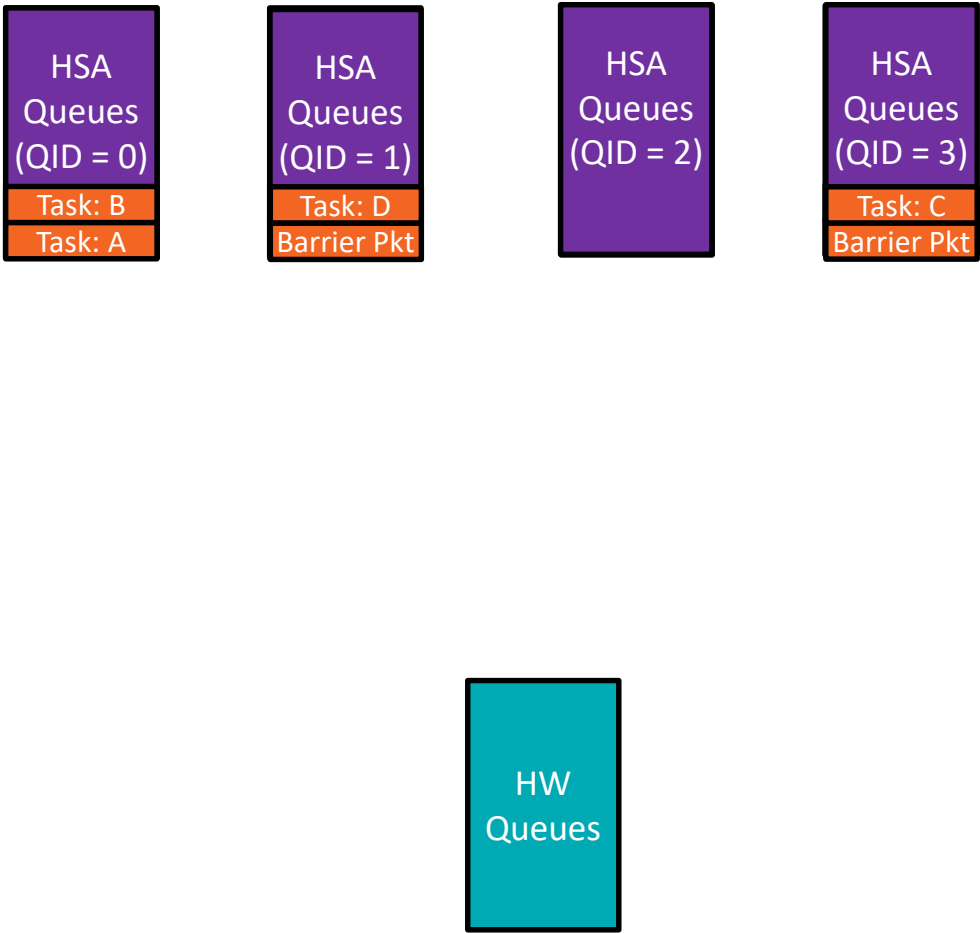
hQ COMMAND AND DISPATCH FLOW

▲ User-mode application talks directly to the hardware
  - HSA Architected Queuing Language (AQL) defines vendor-independent format
  - No system call
  - No kernel driver involvement

▲ Hardware scheduling

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

▲ Device enqueue: GPU kernels can self-enqueue additional tasks (dynamic parallelism)
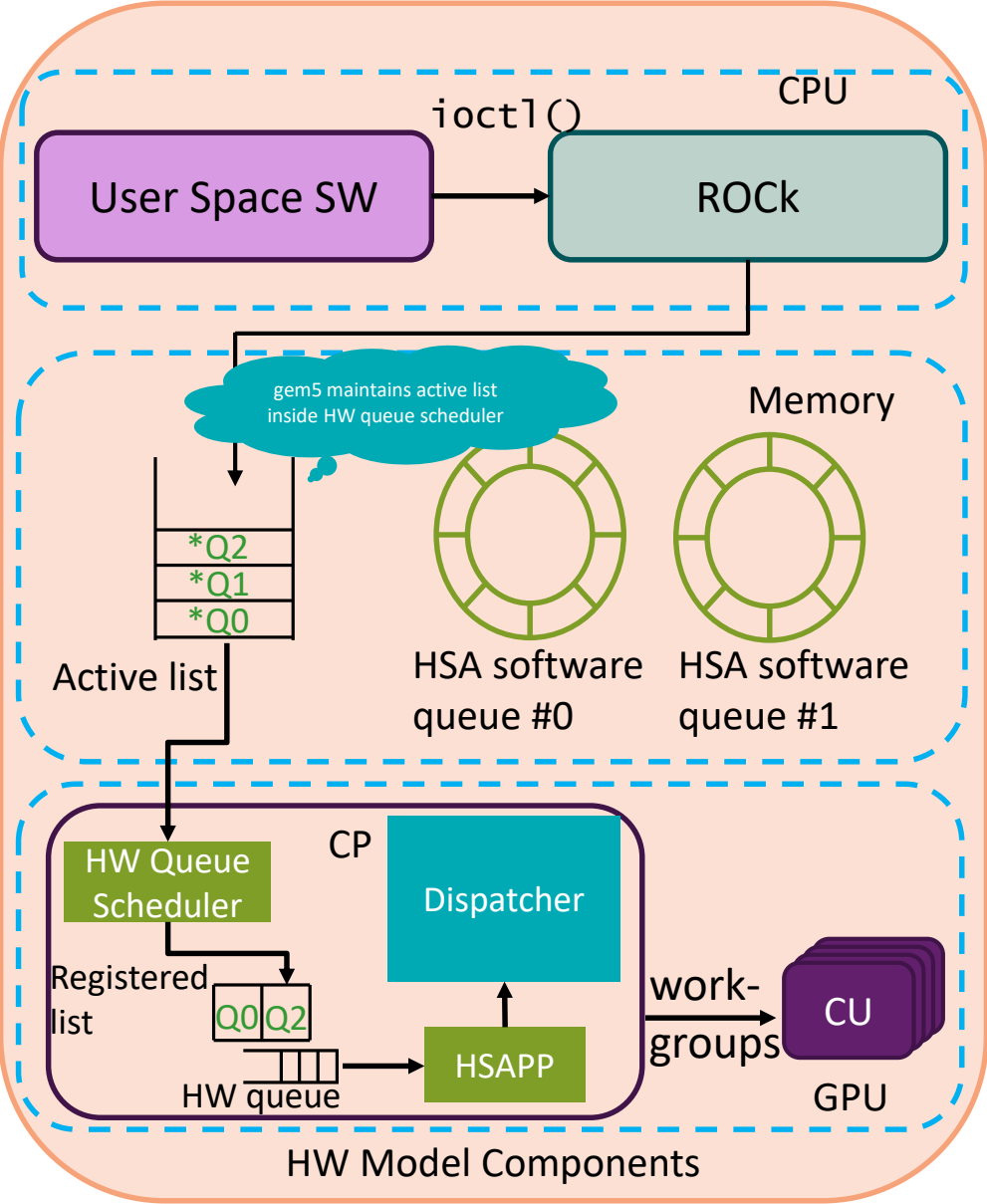
NATIVE SUPPORT FOR DATA-DEPENDENT TASKS

EXPOSE DIRECTED ACYCLIC GRAPHS (DAG) TO HARDWARE

AMD◢

HSA Queues (QID = 0)	HSA Queues (QID = 1)	HSA Queues (QID = 2)	HSA Queues (QID = 3)
Task: B	Task: D		Task: C
Task: A	Barrier Pkt		Barrier Pkt

HW Queues

Sooraj Puthoor, Xulong Tang, Joseph Gross, and Bradford M. Beckmann. 2018. Oversubscribed Command Queues in GPUs. *In Proceedings of the 11th Workshop on General Purpose GPUs* (GPGPU-11). February 2018.

Component	Source file
driver	gpu-compute/gpu_compute_driver.cc dev/hsa/kfd_ioctl.h
hardware scheduler	dev/hsa/hw_scheduler.[hh|cc]
packet processor	dev/hsa/hsa_packet_processor.[hh|cc]
dispatcher	gpu-compute/dispatcher.[hh|cc]

# DOORBELL PAGES AND EVENT PAGES

1. `HSADriver::mmap(mmap args)`
   - runtime calls mmap on the driver
   - mmap offset distinguishes event page vs doorbell page mmap

2. driver allocates doorbell page and returns the page address
   - For doorbells -> Driver maps the page address to PIO address in the PT
   - For events -> Event pages are user pages

3. runtime allocates doorbell address for each queue based on *queue ID*.
   - Event pages are currently unused; model relies on functional interfaces for event notification.
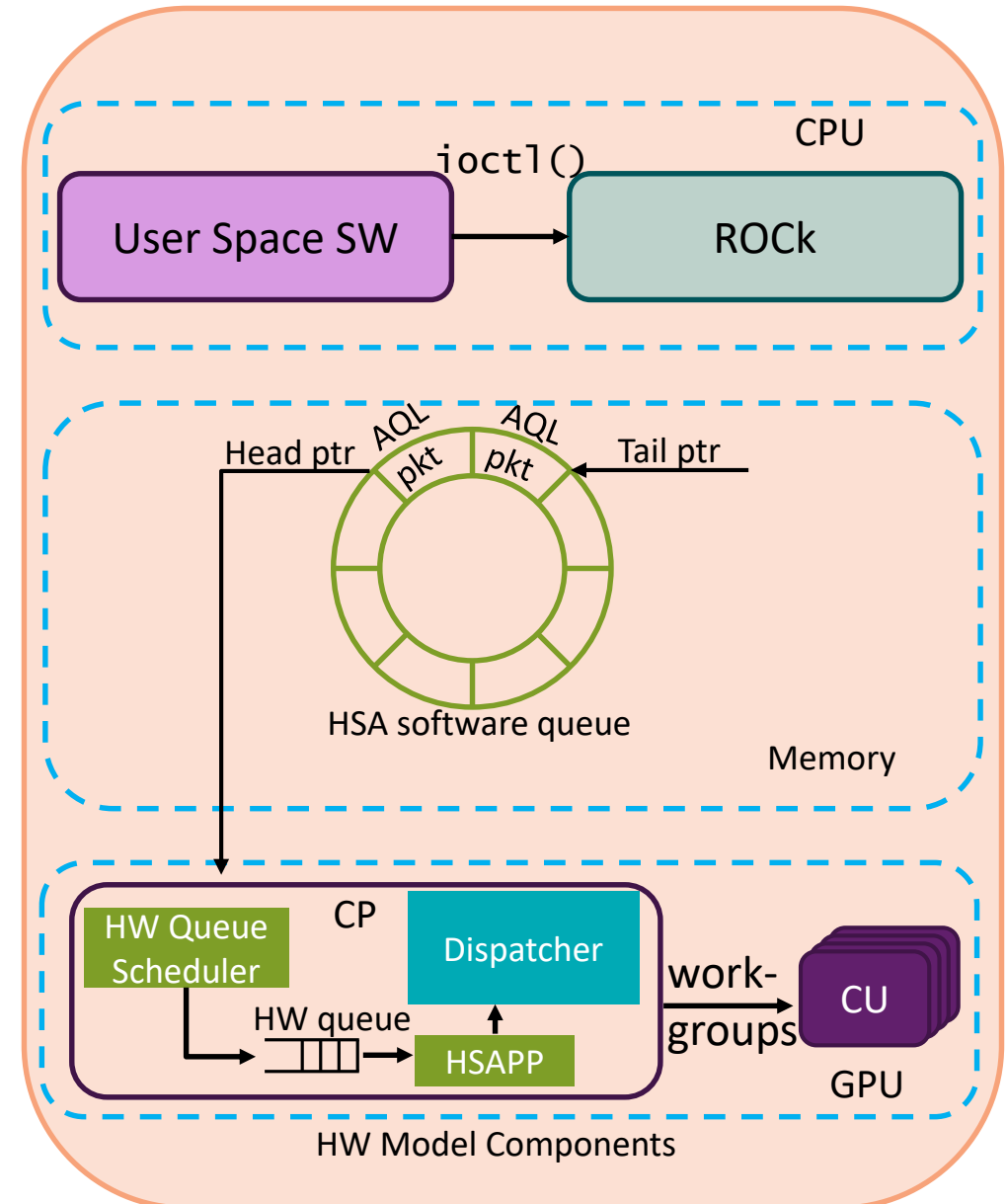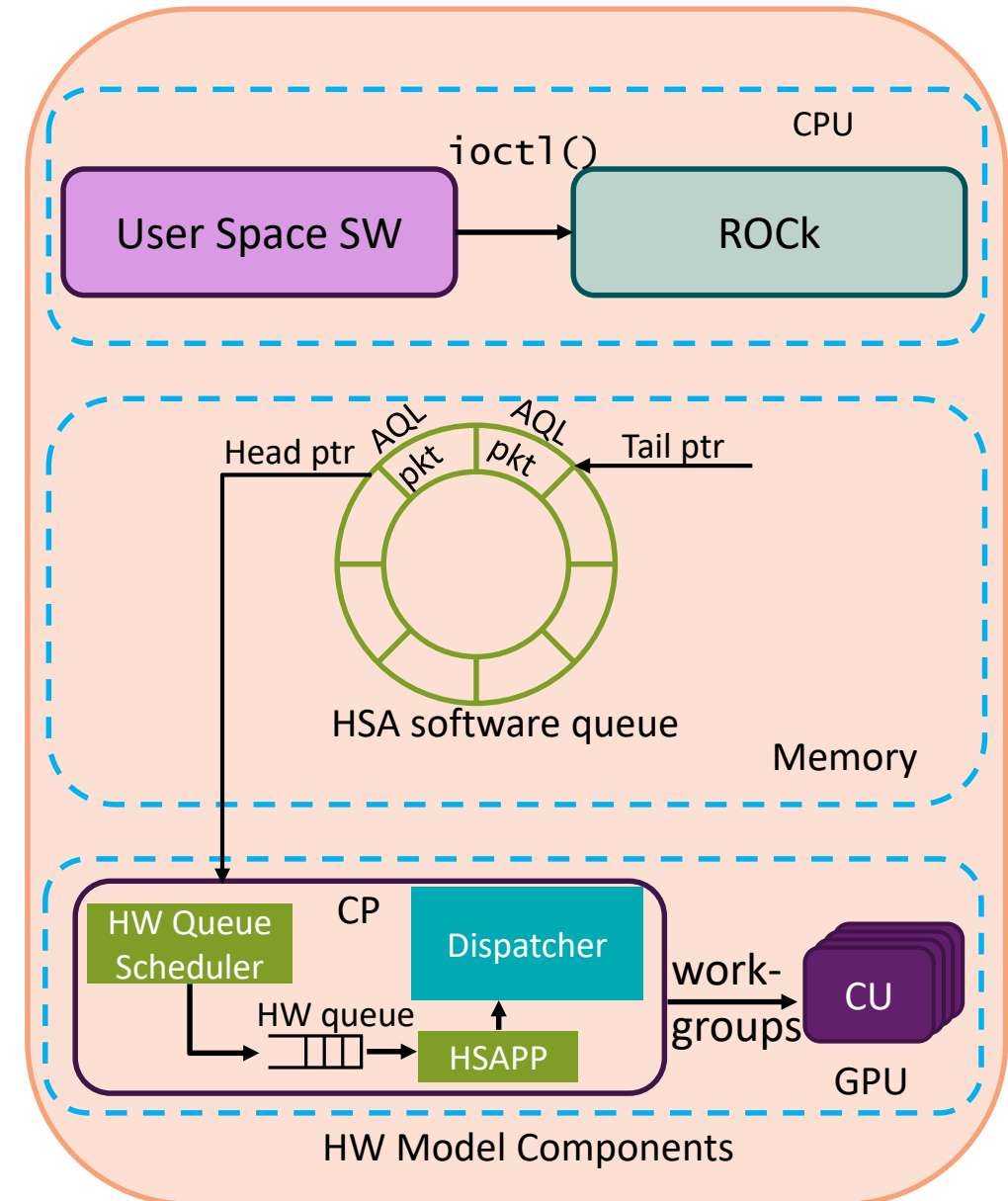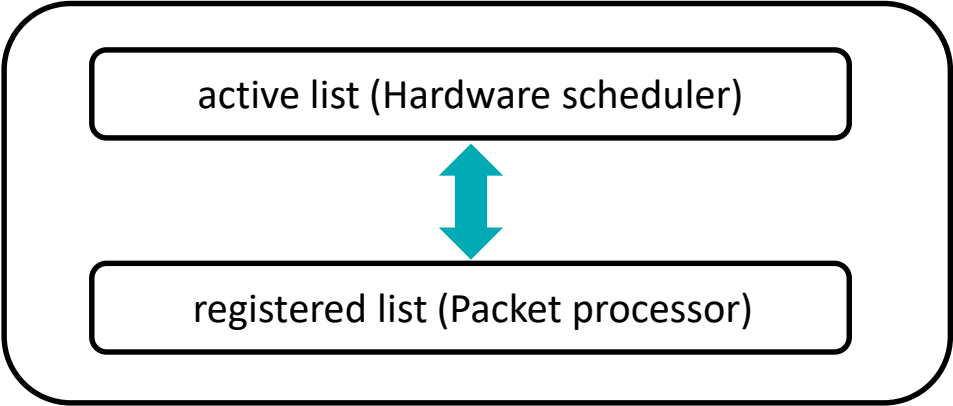
HSA QUEUE CREATION

1. `hsa_queue_create(some args)`

2. `GPUComputeDriver::ioctl(tc, AMDKFD_IOC_CREATE_QUEUE)`

3. `HWScheduler::registerNewQueue(some args)`

QUEUE SWAPPING LOGIC

AMD

active list (Hardware scheduler)

⇅

registered list (Packet processor)

```
// Wakeup every wakeupDelay ticks
1. HWScheduler::wakeup()
```

```
2. HWScheduler::contextSwitchQ ()
```
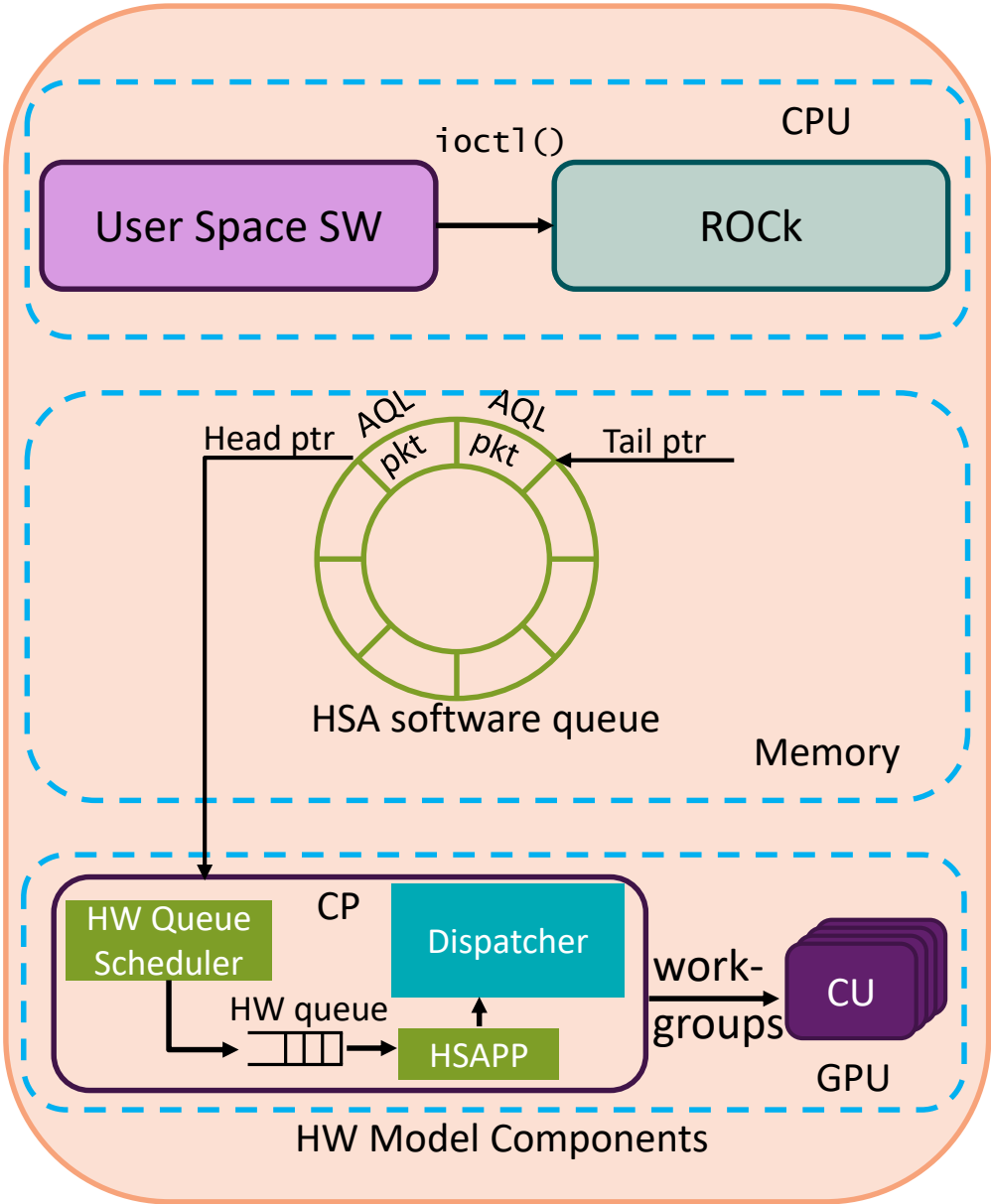
HSA SIGNAL CREATION

1. `hsa_signal_create(some args)`

2. `GPUComputeDriver::ioctl(tc, AMDKFD_IOC_CREATE_EVENT)`

3. `hsa_signal_wait_scacquire(some args)`

4. `GPUComputeDriver::ioctl(tc, AMDKFD_IOC_WAIT_EVENTS)`

5. `GPUComputeDriver::ioctl(tc, AMDKFD_IOC_SET_EVENT)` //or
5. `signalWakeupEvent(event ID)`
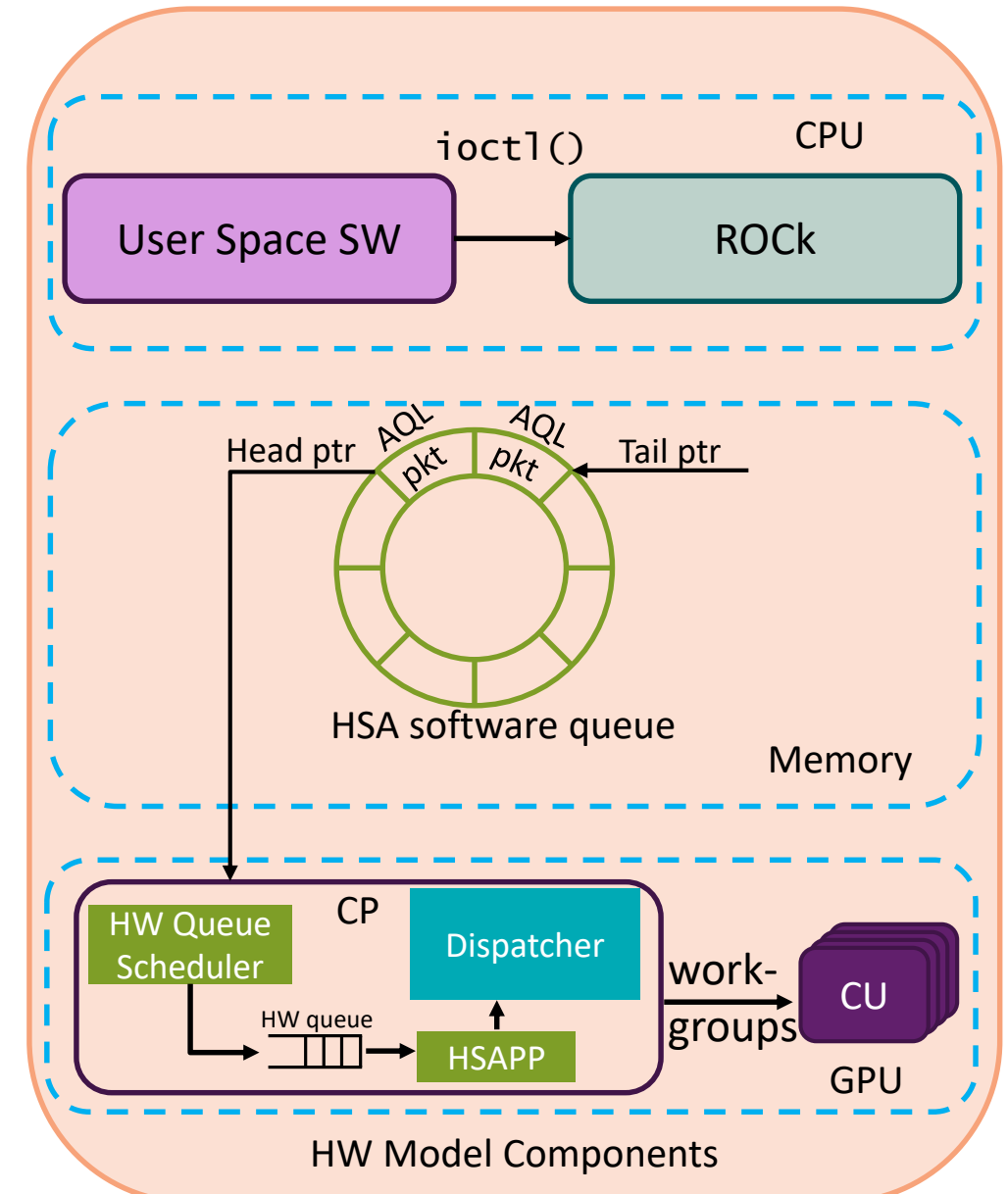
HSA DOORBELLS

1. `hsa_signal_store_screlease(queue->doorbell_signal, packet_id);`

2. `HSAPacketProcessor::write(some args)`

3. `HWScheduler::write(Addr db_addr, uint32_t doorbell_reg)`

4. `HSAPacketProcessor::getCommandsFromHost(some args)`

OUTLINE

AMD

Topic	Presenter	Time
Background	Tony	8:00 – 8:15
ROCm, GCN3 ISA, and GPU Arch	Tony	8:15 – 9:15
HSA Implementation in gem5	Sooraj	9:15 – 10:00
Break		**10:00 – 10:30**
Ruby and GPU Protocol Tester	Tuan	10:30 – 11:15
Demo and Workloads	Matt	11:15 – 11:50
Summary and Questions	All	11:50 – 12:00

OUTLINE

AMD

Topic	Presenter	Time
Background	Tony	8:00 – 8:15
ROCm, GCN3 ISA, and GPU Arch	Tony	8:15 – 9:15
HSA Implementation in gem5	Sooraj	9:15 – 10:00
Break		10:00 – 10:30
Ruby and GPU Protocol Tester	**Tuan**	**10:30 – 11:15**
Demo and Workloads	Matt	11:15 – 11:50
Summary and Questions	All	11:50 – 12:00

RUBY MEMORY CONTRIBUTIONS
OUTLINE

◢ Ruby Background

◢ CU – Memory Interface

◢ GPU VIPER Protocol

◢ GPU SLICC Protocol Tester

RUBY MEMORY CONTRIBUTIONS

OUTLINE

- ◢ **Ruby Background**
- ◢ CU – Memory Interface
- ◢ GPU VIPER Protocol
- ◢ GPU SLICC Protocol Tester

AMD

◢ Flexible Memory System
  – Rich configuration
    – Simulate combination of caches, coherence, interconnect, etc.
  – Rapid prototyping
    – Domain-Specific Language (SLICC) for coherence protocols
    – Modular components

◢ Detailed statistics
  – Latency distributions for requests
  – Generated state transitions, network utilization, etc.

◢ Detailed component simulation
  – Network (fixed/flexible Garnet pipelines and simple)
  – Caches (pluggable replacement policies)
  – Memory (shared memory controllers between Classic and Ruby)

◢ gem5 Ruby tutorial: http://learning.gem5.org/book/part3/index.html

◢ Our GCN3 GPU model only works with Ruby memory

RUBY MEMORY CONTRIBUTIONS
OUTLINE

- Ruby Background
- CU – Memory Interface
- GPU VIPER Protocol
- GPU SLICC Protocol Tester

CU - MEMORY INTERFACE

PRIMARY GCN3 MEMORY OPERATIONS

◢ Scalar memory operations (to scalar cache)
  – S_LOAD
  – S_STORE

◢ Vector memory operations (to TCP)
  – BUFFER_LOAD
  – BUFFER_STORE
  – BUFFER_ATOMIC

◢ Cache-wide Synchronization Operation
  – BUFFER_WBINVL1: write back and invalidate TCP cache

CU - MEMORY INTERFACE
GPU MEMORY COALESCING – CRITICAL FOR PERFORMANCE
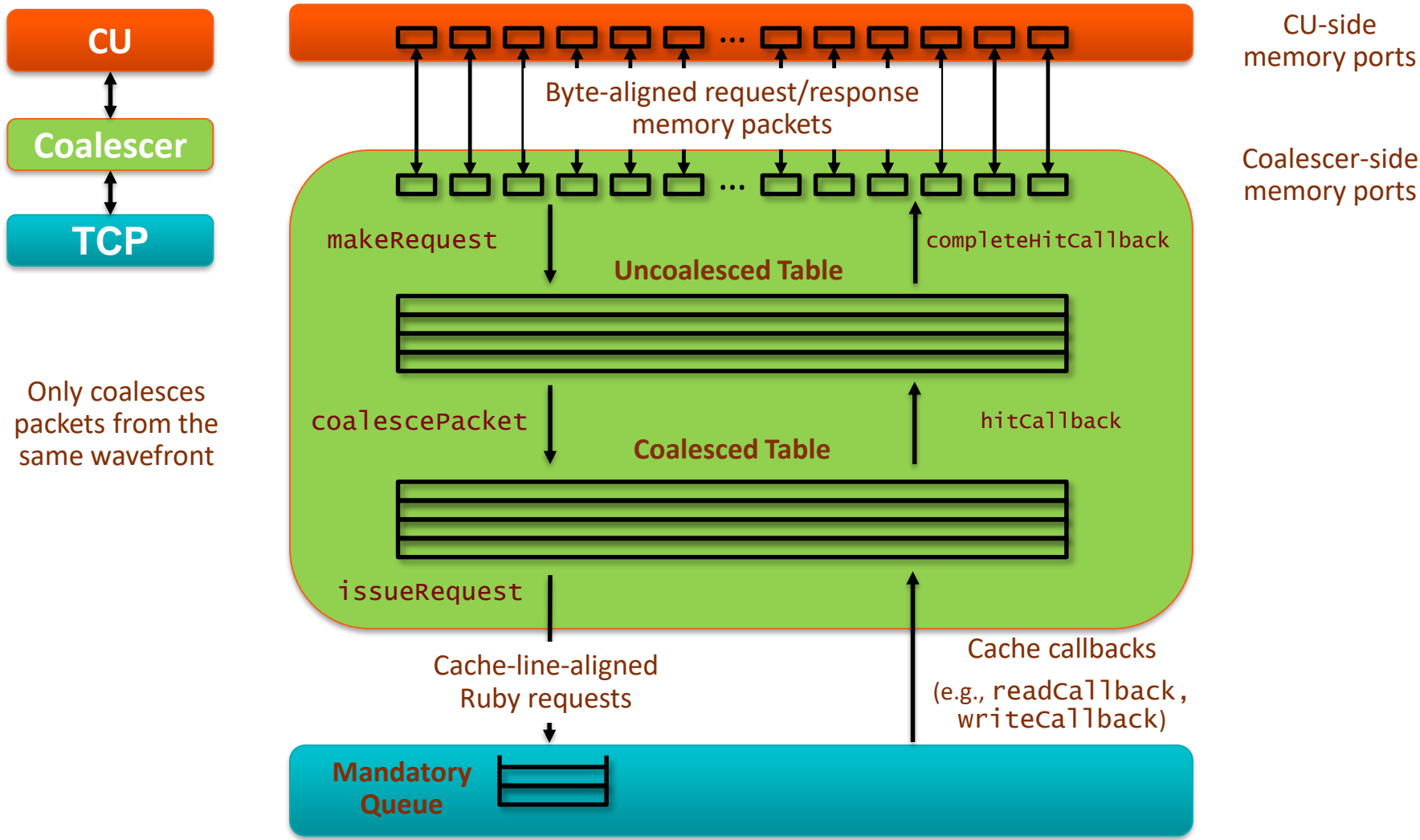
RUBY MEMORY CONTRIBUTIONS
OUTLINE

◢ Ruby Background

◢ CU – Memory Interface

◢ **GPU VIPER Protocol**

◢ GPU SLICC Protocol Tester

HIGH-LEVEL STRUCTURE

AMD

MOESI_AMD_Base-CorePair.sm

GPU

CPU

SQC (Scalar)   SQC (Inst)

GPU_VIPER-SQC.sm

L1I

CU   CU   CU   CU

CPU0   CPU1

GPU_VIPER-TCP.sm

TCP   TCP   TCP   TCP

GPU_VIPER-TCC.sm

L1D   L1D

TCC

L2

Stateless Directory   Memory Controller   Memory

MOESI_AMD_Base-dir.sm
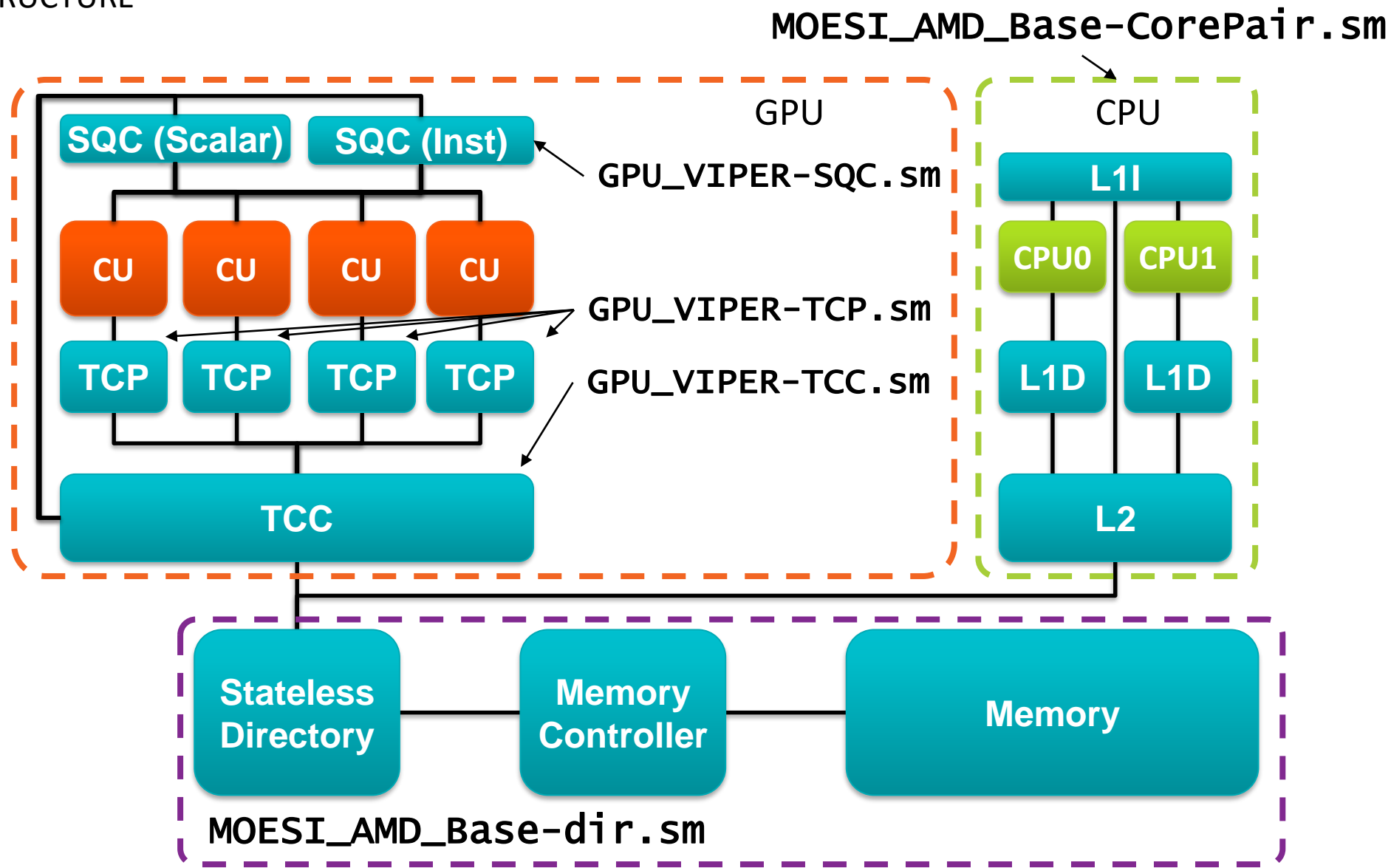
GPU VIPER PROTOCOL

WRITE-THROUGH COHERENCE

◢ GPU write-through protocol
  - Store & Atomic requests are written through TCP and TCC
  - TCP performs Store & Atomic requests immediately (i.e., no stalling for exclusive permissions)
  - For Store requests, TCP calls back immediately (i.e., `writeCallback`) without waiting for ACKs from memory
  - TCP does another callback (i.e., `writeCompleteCallback`) when receiving store-complete ACKs from memory
  - Atomic requests are performed in memory

◢ Data coherence in TCP
  - TCP-INV requests (e.g., issued by `BUFFER_WBINVL1`) to invalidate entire TCP

◢ Data coherence between TCC and CPU caches
  - Data coherence is maintained through a stateless directory (`MOESI_AMD_Base-dir.sm`)

◢ Support single coherent address space between CPU and GPU

GPU VIPER PROTOCOL

RELEASE CONSISTENCY SUPPORT

AMD

◢ Release consistency
  – Hower *et al.* [ASPLOS 2014]
  – HSA System Arch Specification [hsafoundation.com]

◢ Acquire fence
  – Issue TCP-INV to invalidate all stale data in TCP

◢ Release fence
  – Wait for all outstanding store requests to commit globally (i.e., `writeCompleteCallback`)
  – Wait for all atomic requests (i.e., `atomicCallback`)

62 | THE AMD gem5 APU SIMULATOR | JUNE 2, 2018 | ISCA 2018 TUTORIAL |

RUBY MEMORY CONTRIBUTIONS

OUTLINE

AMD

◢ Ruby Background

◢ CU – Memory Interface

◢ GPU VIPER Protocol

◢ GPU SLICC Protocol Tester

GPU SLICC PROTOCOL TESTER

MOTIVATION

◢ Developing a functionally correct SLICC protocol is challenging

◢ We need an effective tool to verify a protocol

Desired Features	GPU Protocol Tester
GPU protocol compatibility	✓
Precise validation	✓
Wide bug coverage	✓
Fast bug detection	✓
Intuitive bug report	✓

GPU SLICC PROTOCOL TESTER

HIGH-LEVEL STRUCTURE – RUBY INTERFACE

Compute Cores

Ruby Ports

Ruby Memory

CPU core

GPU core

Sequencer

Sequencer

Sequencer

Sequencer

Coalescer

CPU Inst$

CPU Data$

GPU Inst$

GPU Scalar$

GPU Data$

The rest of cache hierarchy and memory

AMD
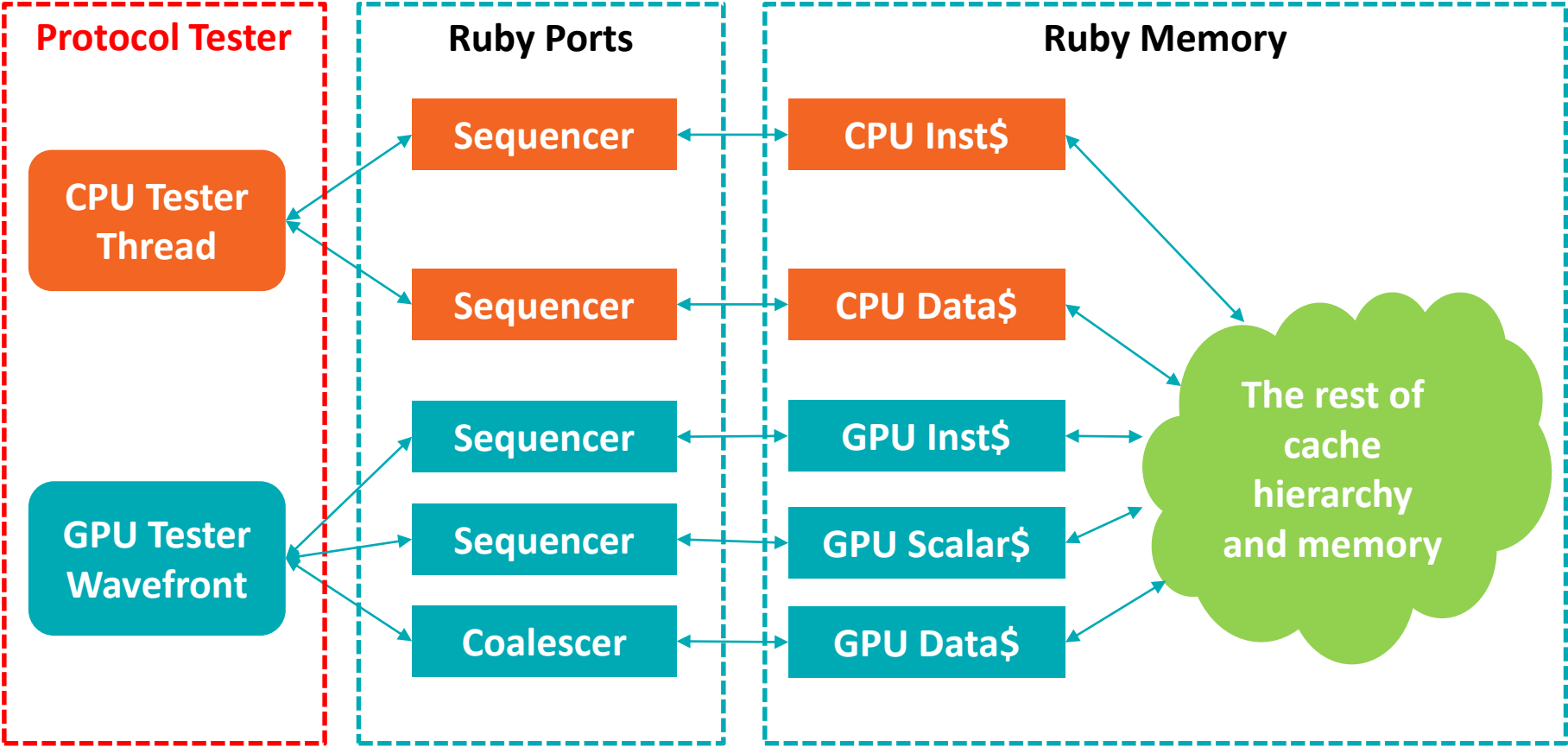
◢ Only support data caches for now

GPU SLICC PROTOCOL TESTER

AMD

HIGH-LEVEL STRUCTURE – TESTER THREAD

◢ Issue sequences (a.k.a., episodes) of memory operations (i.e., Load, Store, Atomic and synchronization operations) with respect to release consistency

◢ CPU tester thread
  – Load, Store, and Atomic operations are scalar
  – Acquire and release fence are no-ops

◢ GPU Tester Wavefront
  – Load, Store, and Atomic are vector operations (i.e., issued by multiple lanes in a wavefront)
  – Acquire fence issues TCP-INV
  – Release fence waits for all outstanding Store and Atomic operations to complete (i.e., through `writeCompleteCallback` and `atomicCallback`)

GPU SLICC PROTOCOL TESTER

HIGH-LEVEL STRUCTURE - EPISODE

◢ Think of an episode as a critical section

CPU Tester Thread

GPU Tester Wavefront (4 lanes)

Acquire Lock

ATOMIC (A)
ACQUIRE
LD (X)
...
ST (Y)
RELEASE
ATOMIC (A)

AT (A0)	AT (A1)	AT (A2)	AT (A3)
ACQUIRE			
LD (X0)	LD (X1)	LD (X2)	LD (X3)
...			
ST (Y0)	ST (Y1)	ST (Y2)	ST (Y3)
RELEASE			
AT (A0)	AT (A1)	AT (A2)	AT (A3)

Mix of LDs & STs

Release Lock
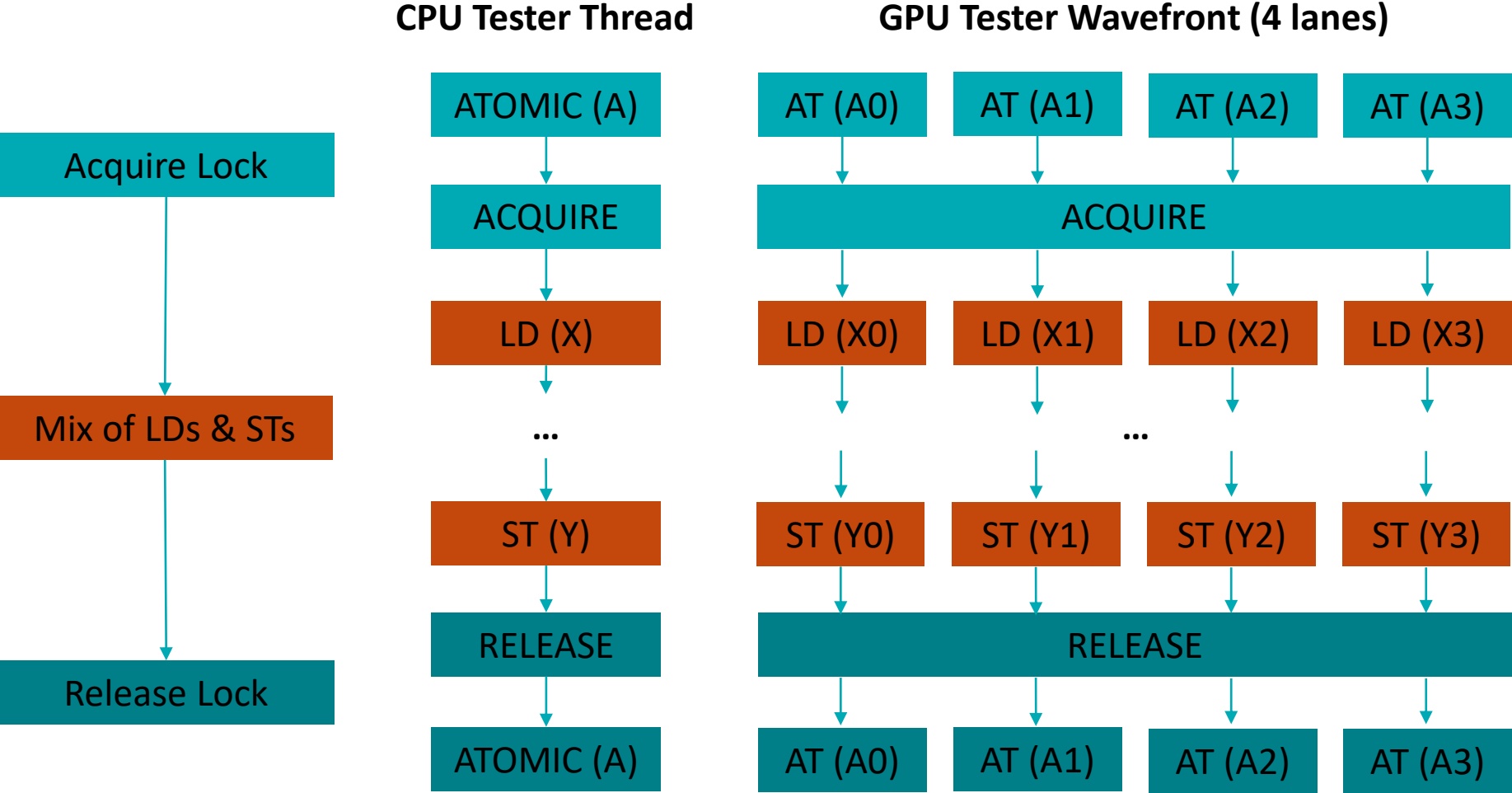
GPU SLICC PROTOCOL TESTER

HIGH-LEVEL STRUCTURE - EXECUTION FLOW PER THREAD

Issue a new episode

- ▶ Randomly generate a series of Load, Store and Atomic operations to be issued
- ▶ For Stores, reserve variables

Execute the episode

- ▶ Issue requests across all lanes
- ▶ Check dependencies between requests
- ▶ Receive and validate data responses (for Load and Atomic operations)
- ▶ Track per-address last writers after each store operation

Retire the episode

- ▶ Release all reserved variables in this episode
- ▶ Check for completion

DATA RACE FREE (DRF) REQUEST STREAM

▲ Random GPU Ruby tester does **NOT** cover data-race scenarios

Eps 0

- Atomic_Acq (A)
- LD (X) = 0
- ST: 1 -> Y
- Atomic_Rel (A)

Eps 1 (Invalid)

- Atomic_Acq (A)
- LD (Y) = 0 or 1?
- ST: 2 -> X
- Atomic_Rel (A)

Eps 2

- Atomic_Acq (A)
- LD (Y) = 1
- ST: 1 -> X
- Atomic_Rel (A)

happened-before

race

no race

A: Atomic variable

X & Y: Non-atomic variables associated with A

Time
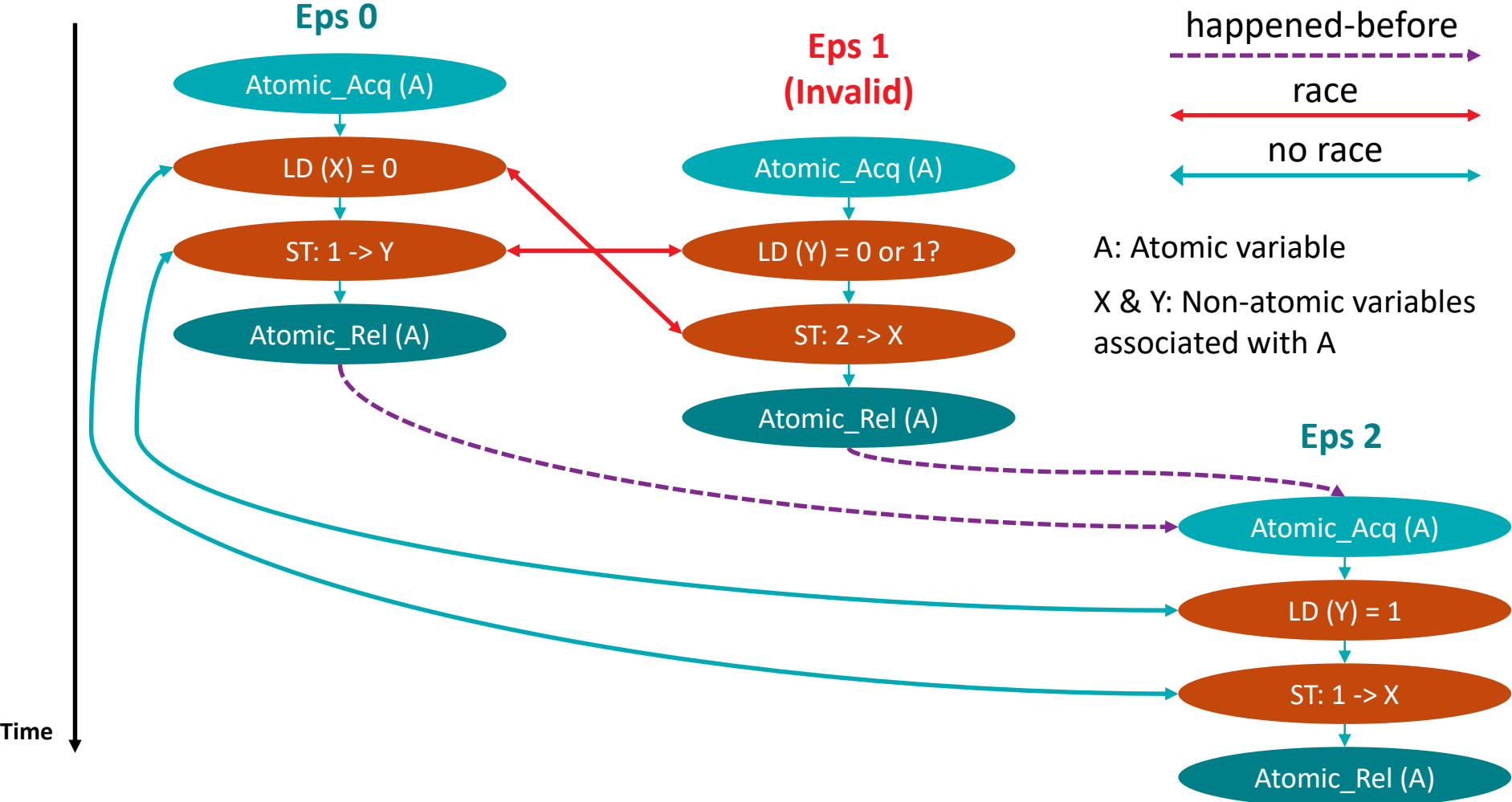
GPU SLICC PROTOCOL TESTER

HIGH LEVEL STRUCTURE – ADDRESS RANGE

AMD

Atomic Variables   **Non-Atomic Variables**

Variables | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |

Randomly mapped

Addresses | 0x00 | 0x04 | 0x08 | 0x0c | 0x10 | 0x14 | 0x18 | 0x1c | 0x20 | 0x24 |

Line 0   **Line 1**   **Line 2**   **Line 3**   **Line 5**
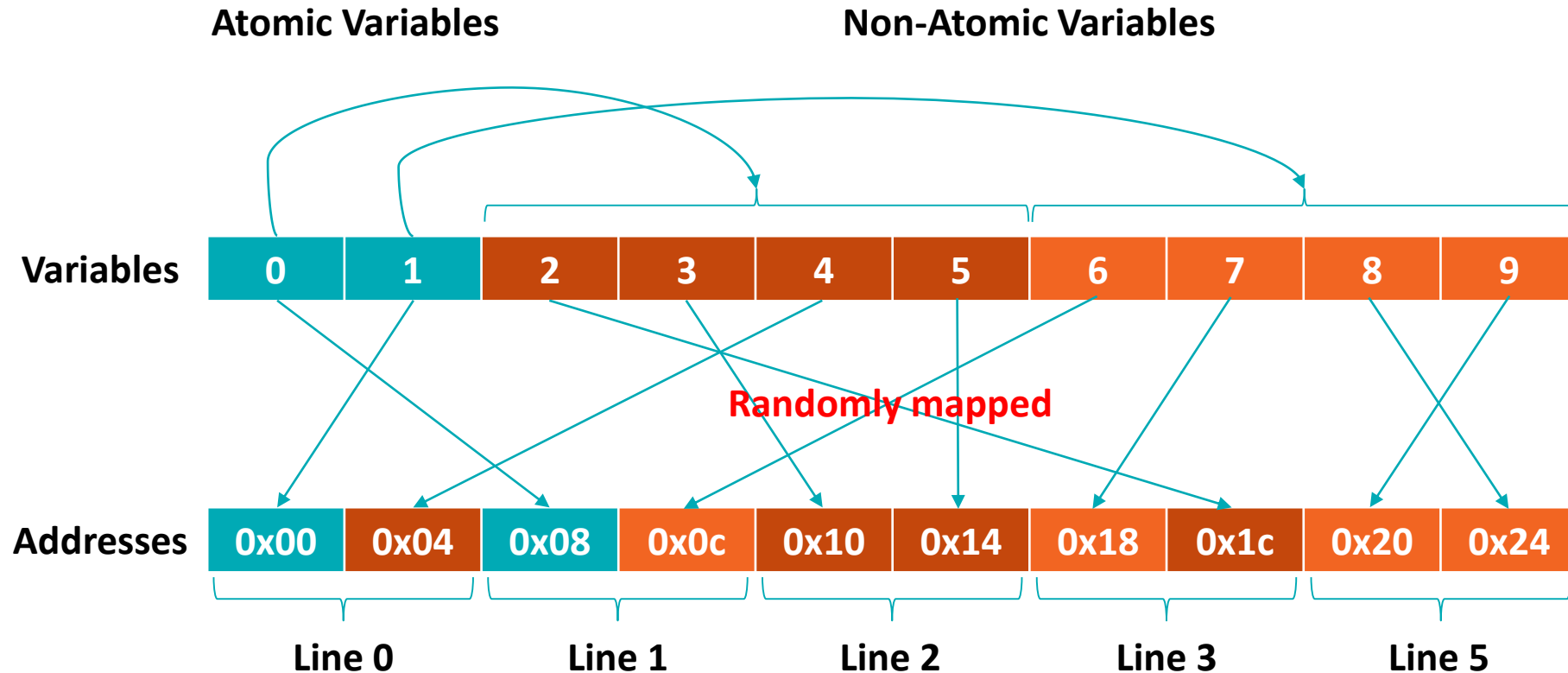
◣ Random mapping → false sharing

◣ An episode can only reserve and issue load/store operations to non-atomic variables associated with the atomic variable it acquires

◢ Track per-variable last writers

- – Who? (i.e., Lane ID, Wavefront ID, CU ID, and Episode ID)

- – What value was written?

- – When did the store operation happen?

◢ For a Load (X), check if return value of X is equal to value written by the last writer to X. The last writer of X could be

- – Either the last ST → X in the same episode, or

- – The last ST → X in a previous episode

◢ For an Atomic (A)

- – Use atomic_inc: increment A by 1 atomically

- – Track how many atomics have been issued so far. Let's say N.

- – Return value must be unique in the range [0..N-1] inclusively

GPU SLICC PROTOCOL TESTER

FAILURE REPORT

AMD

▲ Forward progress check
- Report all outstanding requests that have been pending for more than a certain threshold
- Report their target addresses, lane IDs, CU IDs, CPU IDs, and episode IDs

▲ Inconsistency between a Reader and its last Writer (the most common failure!)

```
warn: GpuWavefront(Thread ID = 35, CU ID = 4): Loaded value is not consistent with the last stored value
    Thread 35
    Episode 727
    Lane ID 2
    Address [0x52860, line 0x52840]
    Loaded value 16
    Last writer (Thread ID 35, CU ID 4, Episode ID 727, Value 17, Tick 8454382)
```

▲ Unexpected atomic return
- Report all expected return values
- Report address, lane ID, CU ID, and episode ID

▲ The report really helps trace protocol bugs down **quickly** and **precisely**!

GPU SLICC PROTOCOL TESTER

AMD

TESTING OPTIONS AND SCENARIOS

◢ Available configurations

- – Cache size

- – Address range - # of atomic and normal variables

- – System size - # CUs and #CPUs

- – Episode length - # of LDs and STs per episode

- – Test length - # of episodes

◢ Example testing scenarios

- – Small cache size + Wide address range
  - – Working set is much larger than the cache size
  - – Likely to expose bugs related to cache replacement
- – Large cache size + Small address range
  - – Likely to expose bugs related to cache coherence (e.g., values are not passed between private caches correctly)
- – Long episode length
  - – Conflicting accesses in the same episode

OUTLINE

AMD

Topic	Presenter	Time
Background	Tony	8:00 – 8:15
ROCm, GCN3 ISA, and GPU Arch	Tony	8:15 – 9:15
HSA Implementation in gem5	Sooraj	9:15 – 10:00
Break		10:00 – 10:30
Ruby and GPU Protocol Tester	Tuan	10:30 – 11:15
Demo and Workloads	**Matt**	**11:15 – 11:50**
Summary and Questions	All	11:50 – 12:00

HIPIFY'ING CUDA BENCHMARKS

◢ HIP code is easy to write if you have experience with CUDA

◢ Hipify-perl can automatically convert most CUDA code to HIP code
  – Works well for simple applications (e.g., no library calls)
  – Perl regular expression replacement script
  – Very simple, easy to use (as long as no library calls), fast

◢ Useful HIP Information:
  – <u>Basic Intro to HIP</u>
  – <u>Basic HIP Tutorial</u>
  – <u>HIP Porting Guide</u>
  – <u>Platform-Aware Coding</u>

OUTLINE

Topic	Presenter	Time
Background	Tony	8:00 – 8:15
ROCm, GCN3 ISA, and GPU Arch	Tony	8:15 – 9:15
HSA Implementation in gem5	Sooraj	9:15 – 10:00
Break		10:00 – 10:30
Ruby and GPU Protocol Tester	Tuan	10:30 – 11:15
Demo and Workloads	Matt	11:15 – 11:50
Summary and Questions	**All**	**11:50 – 12:00**

COMPARISON TO OTHER GPU SIMULATORS

◢ **GPGPU-Sim**

– Primarily focused on running Nvidia PTX instructions and CUDA applications

– Functional CPU model, oriented to model discrete GPU systems

– Wisconsin's gem5-gpu added gem5 timing CPU models

– And a Ruby memory system protocol

– Differences from gem5-GPU:

– GCN3 instructions and ROCm software stack

– Unified under the gem5 source control repo

◢ **Multi2Sim**

– Supports multiple ISAs including AMD Southern Island's Machine ISA

– Limited instruction support

– No transient states in coherence protocol

◢ This is very different than the gem5 NoMALI emulated GPU device

OBVIOUS IMPROVEMENTS

AMD

▲ Other GPU ISAs

▲ Complete IOMMU model

▲ Add graphics functionality
  – Currently compute only

▲ Better register model
  – Simple register pool manager: one WG per CU

▲ Remove backing store for memory data
  – When running applications, the data from the caches is not used

SUMMARY

AMD

◢ Covered a very high-level overview of:
– Introduction to the gem5 APU simulator
– Mapping between APU system and gem5 APU simulator

◢ Topics discussed
– HSA and GCN Background
– Compilation and Simulation Flow
– GPU Core modules
– GPU memory system model in Ruby
– GPU protocol tester
– Comparisons/Improvements

◢ Much more detail in the gem5 source code

◢ **Please contribute back to this community tool!**

THANK YOU

AMD

MANY CONTRIBUTORS OVER THE PAST 8+ YEARS

Alex Dutu	David Roberts	John Alsop	Matt Poremba	Si Li
Ali Jafri	Derek Hower	John Kalamatianos	Matt Sinclair	Sooraj Puthoor
Arka Basu	Dmitri Yudanov	Kishore Punniyamurthy	Michael LeBeane	Steve Reinhardt
Ayse Yilmazer	Eric Van Tassell	Kunal Korgaonkar	Mike Chu	Tanmay Gangwani
Binh Pham	Gagan Sachdev	Lisa Hsu	Myrto Papadopoulou	Tim Rogers
Blake Hechtman	James Wang	Manish Arora	Monir Mozumder	Tony Gutierrez
Brad Beckmann	Jason Power	Marc Orr	Nagesh Lakshminarayana	Tsung Tai Yeh
Brandon Potter	Joel Hestness	Mario Mendez-Lojo	Nilay Vaish	Tushar Krishna
Can Hankendi	Jieming Yin	**Mark Leather**	Onur Kayiran	Xianwei Zhang
David Hashe	Joe Gross	Mark Wilkening	Srikant Bharadwaj	Yatin Manerkar
		Martin Brown	Shrikanth Ganapathy	Yasuko Eckert

DISCLAIMER & ATTRIBUTION

AMD

The information presented in this document is for informational purposes only and may contain technical inaccuracies, omissions and typographical errors.

The information contained herein is subject to change and may be rendered inaccurate for many reasons, including but not limited to product and roadmap changes, component and motherboard version changes, new model and/or product releases, product differences between differing manufacturers, software changes, BIOS flashes, firmware upgrades, or the like. AMD assumes no obligation to update or otherwise correct or revise this information. However, AMD reserves the right to revise this information and to make changes from time to time to the content hereof without obligation of AMD to notify any person of such revisions or changes.

AMD MAKES NO REPRESENTATIONS OR WARRANTIES WITH RESPECT TO THE CONTENTS HEREOF AND ASSUMES NO RESPONSIBILITY FOR ANY INACCURACIES, ERRORS OR OMISSIONS THAT MAY APPEAR IN THIS INFORMATION.

AMD SPECIFICALLY DISCLAIMS ANY IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE. IN NO EVENT WILL AMD BE LIABLE TO ANY PERSON FOR ANY DIRECT, INDIRECT, SPECIAL OR OTHER CONSEQUENTIAL DAMAGES ARISING FROM THE USE OF ANY INFORMATION CONTAINED HEREIN, EVEN IF AMD IS EXPRESSLY ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.