
# Running ML Workloads in PyTorch/TensorFlow in gem5

Matthew Poremba, Vishnu Ramadas, and Matthew Sinclair

University of Wisconsin-Madison

vramadas@wisc.edu

# Disclaimers

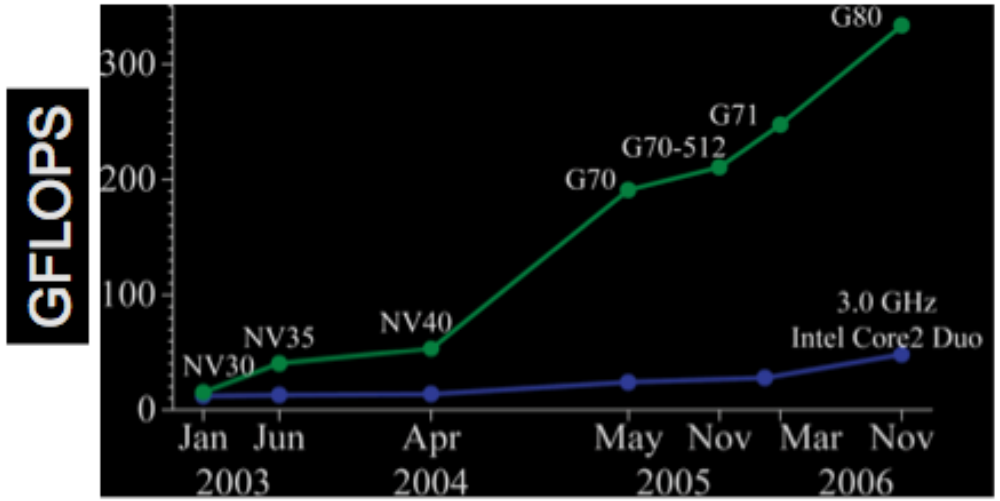
1. Currently gem5 only supports AMD's GPU ISA
   - The concepts are similar to NVIDIA GPUs
2. Currently gem5 only supports GPGPU workloads (no Vulkan, OpenGL support)

# Contributors

- AMD Research: Brad Beckmann, Alex Dutu, Tony Gutierrez, Michael LeBeane, Brandon Potter, Sooraj Puthoor, & many more
- UW-Madison: Matthew Sinclair, Vishnu Ramadas, Daniel Kouchekinia, Marco Kurzynski, Jarvis Jia, Anushka Chandrashekar, Gaurav Jain, Charles Jamieson, Jing Li, Kyle Roarty, Mingyuan Xiang, Bobbi Yogatama, & others

# Graphics Processing Units (GPUs)

- 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

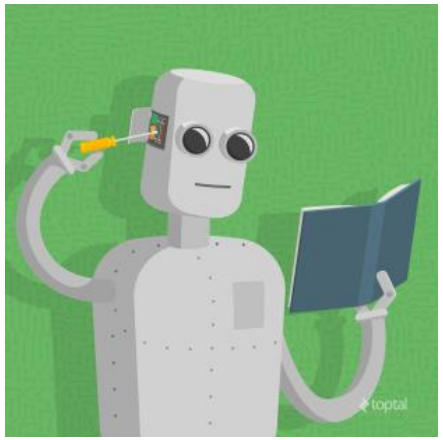
GeForce 8800

Tesla S870

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

# Evolution of GPUs

- From being used for graphics
- To having a new killer application: 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).
  - Identify what additional resources gem5-resources provides.
  - Run basic GPU tests on the (AMD) GPU model in both SE and FS modes
  - Be able to checkpoint applications and restore from checkpoint in FS mode
  - Be able to offload computation of certain kernels onto CPU in FS mode

# Outline

- Background: GPU Architecture & Programming Basics
- Modeling & Using GPUs in gem5
- Running GPU programs in gem5

# 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)

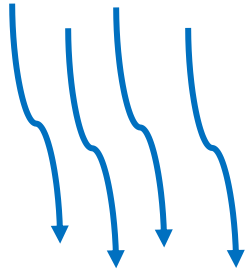
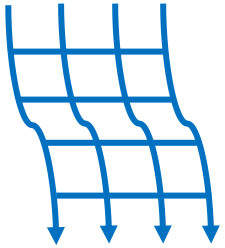
- SIMD (Single Instruction, Multiple Data):
  - Split identical, independent work over multiple execution units
  - More efficient: eliminate redundant fetch/decode vs. SPMD/MIMD
  - Use single PC and single register file

# 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

| | 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

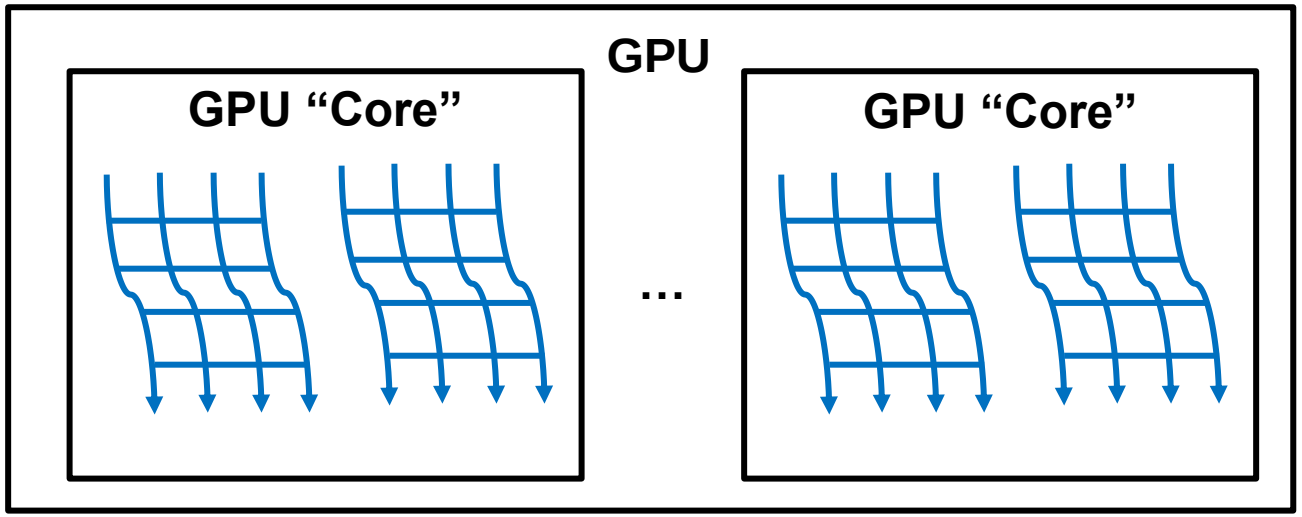
- GPUs optimized for streaming computations
  - Thus, we have a lot of streaming memory accesses

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

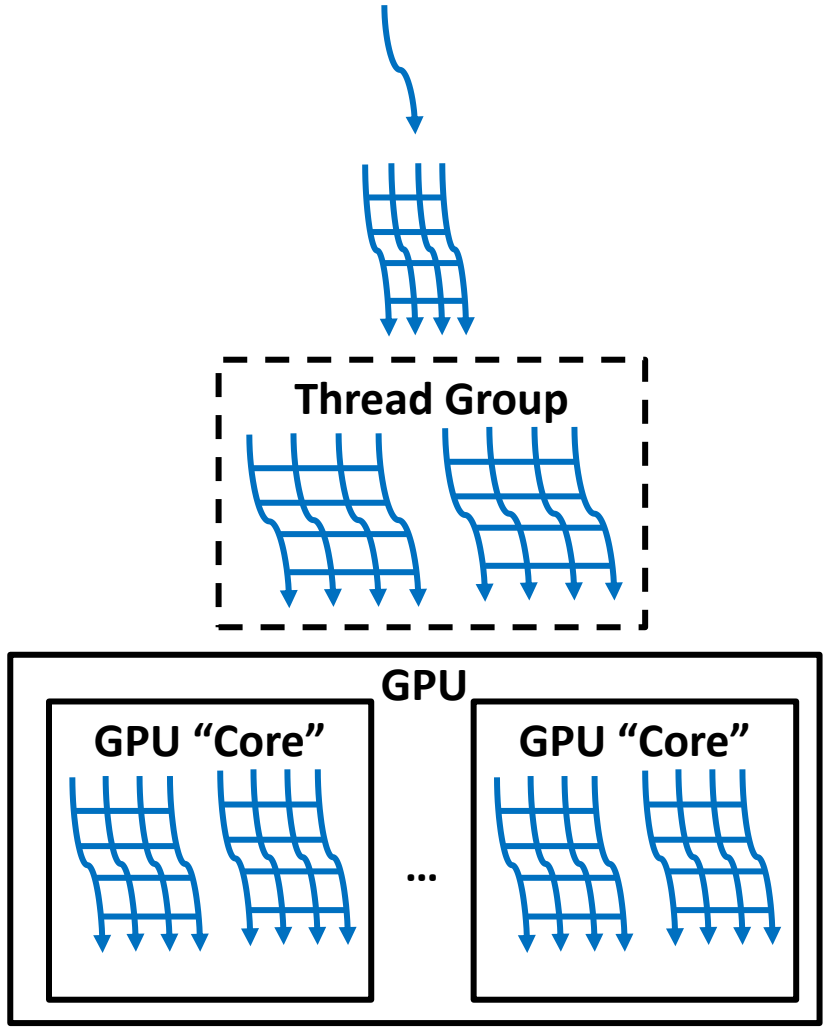
- Traditional CPU approaches:
  - Caches → Need spatial/temporal locality ✗
    - Streaming applications have little reuse
  - OOO/Dynamic Scheduling → Need ILP ✗
    - Too power hungry, diminishing returns for GPU applications
  - Multicore/Multithreading/SMT → need independent threads ✓

# Multicore/Multhidreading/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

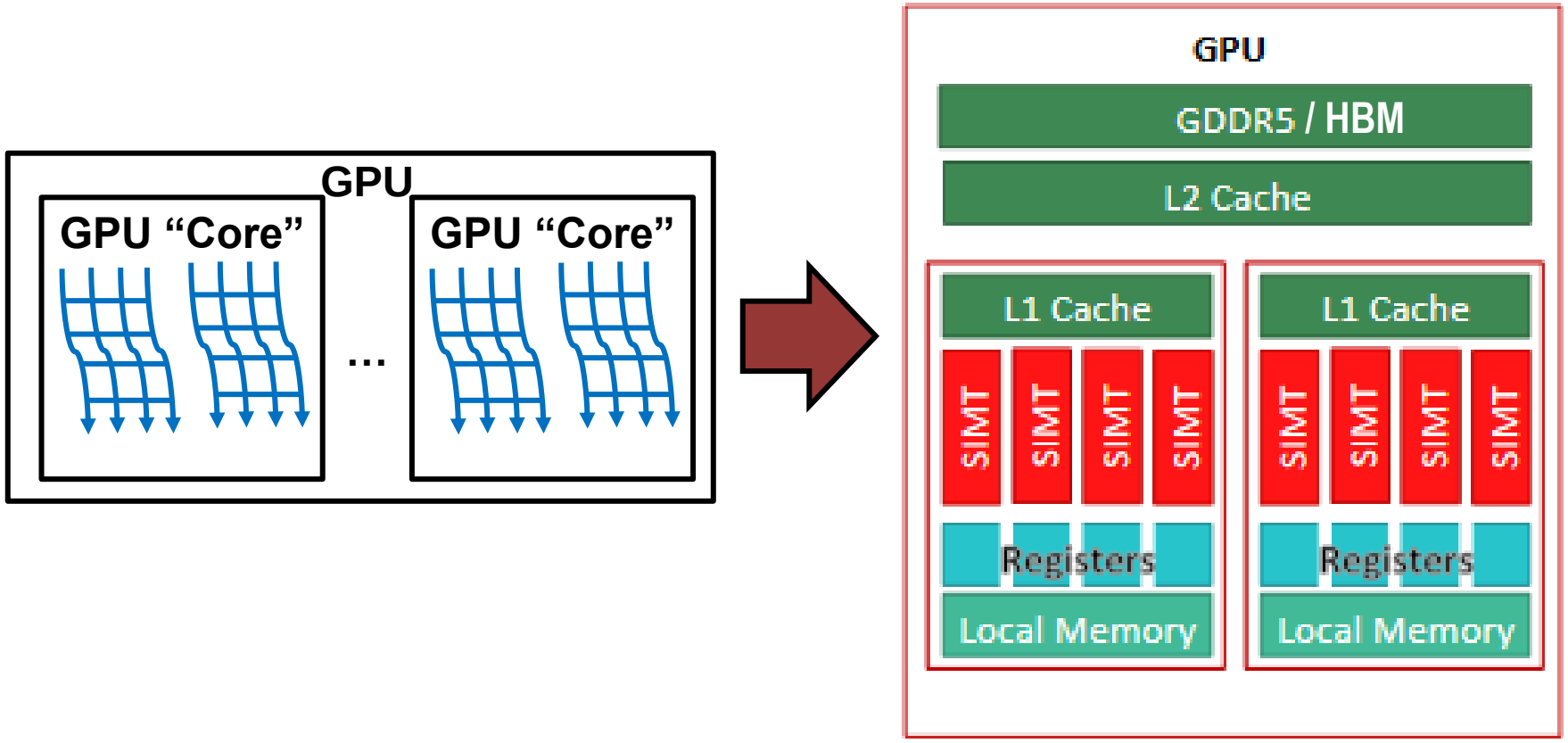



| CUDA/HIP | OpenCL |
|---|---|
| Thread | Work-item |
| Warp | Wavefront |
| Thread Block/CTA | Workgroup |
| Grid (Kernel) | NDRange (Kernel) |

# Programming GPUs

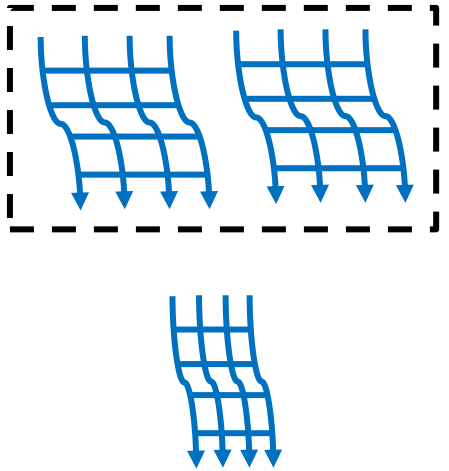
- Program it with CUDA, HIP, or OpenCL
  - CUDA = Compute Unified Device Architecture
    - NVIDIA's proprietary solution
  - OpenCL = Open Computing Language
    - Open, industrywide standard
  - HIP = Heterogeneous interface for portability
    - AMD's open solution, its successor to OpenCL
    - OpenCL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

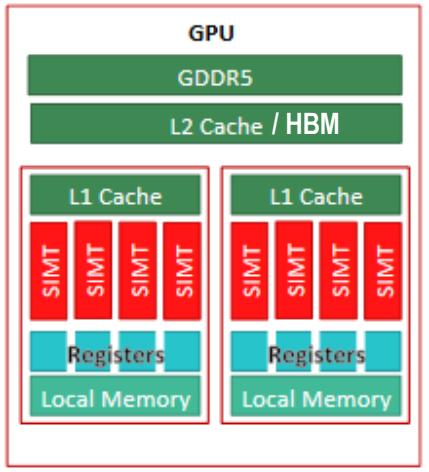


# Compute Unit (CU) – The GPU Core

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



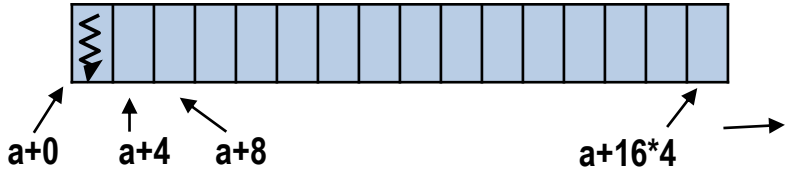
- SIMT unit: runs wavefronts/warps
  - Run the threads
  - AMD: size N (e.g., 10) wavefront instruction buffer
    - 4 cycles to execute one wavefront
    - Average: fetch and commit 1 wavefront/cycle



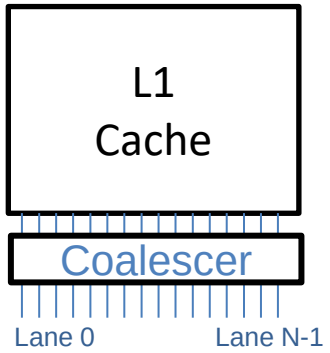
# Memory Accesses

- Pseudo CUDA for contiguous access:

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

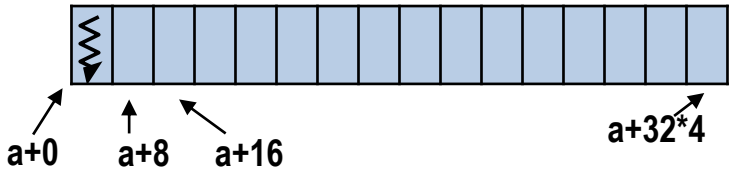



Instead issue one access
A[n:n+16*4]




- Pseudo CUDA for non-contiguous access:

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

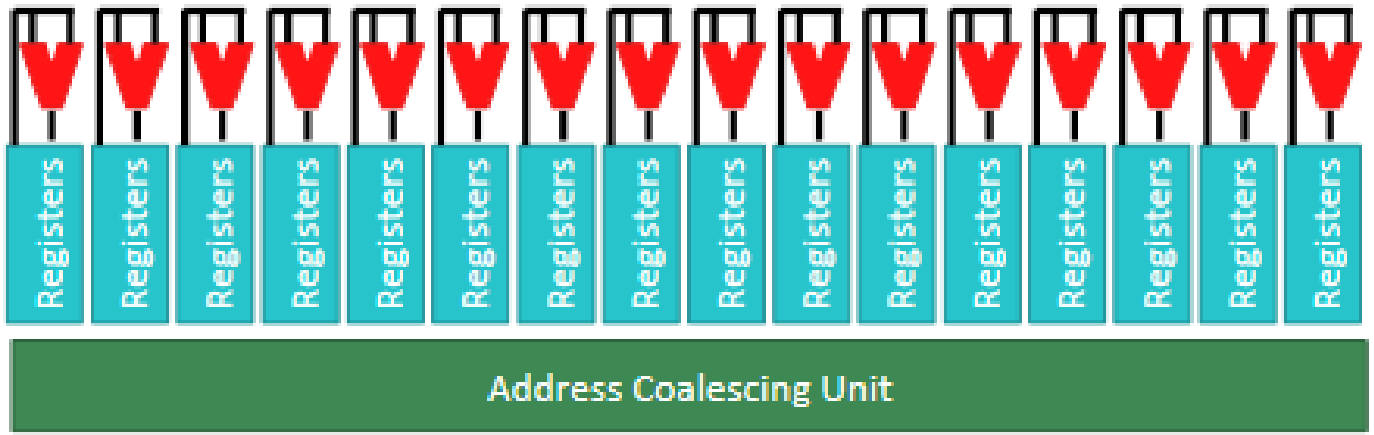



Instead issue two access
A[n :n+16*4],
A[n+16*4:n+32*4]

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

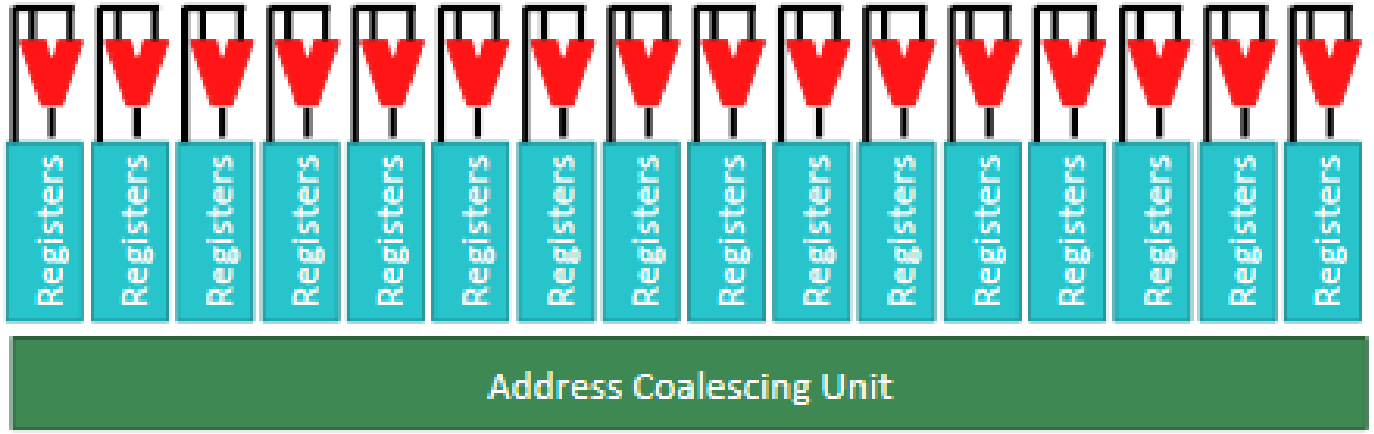
# SIMT Unit – A GPU Pipeline

- Similar to a wide CPU pipeline, except only fetch 1 instr.
- 16-wide physical ALU – specific to the AMD GPU uArch supported by gem5
- 64 KB register state/SIMD unit – 4 SIMD units per CU
  - Much bigger (~64X) than CPUs
- Addressing coalescing key to good performance
  - Each thread potentially fetches a different piece of data
  - 64 separate addresses for AMD, 32 for NVIDIA (tradeoffs)



# L1 Caches

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



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

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

# GPU Caches

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

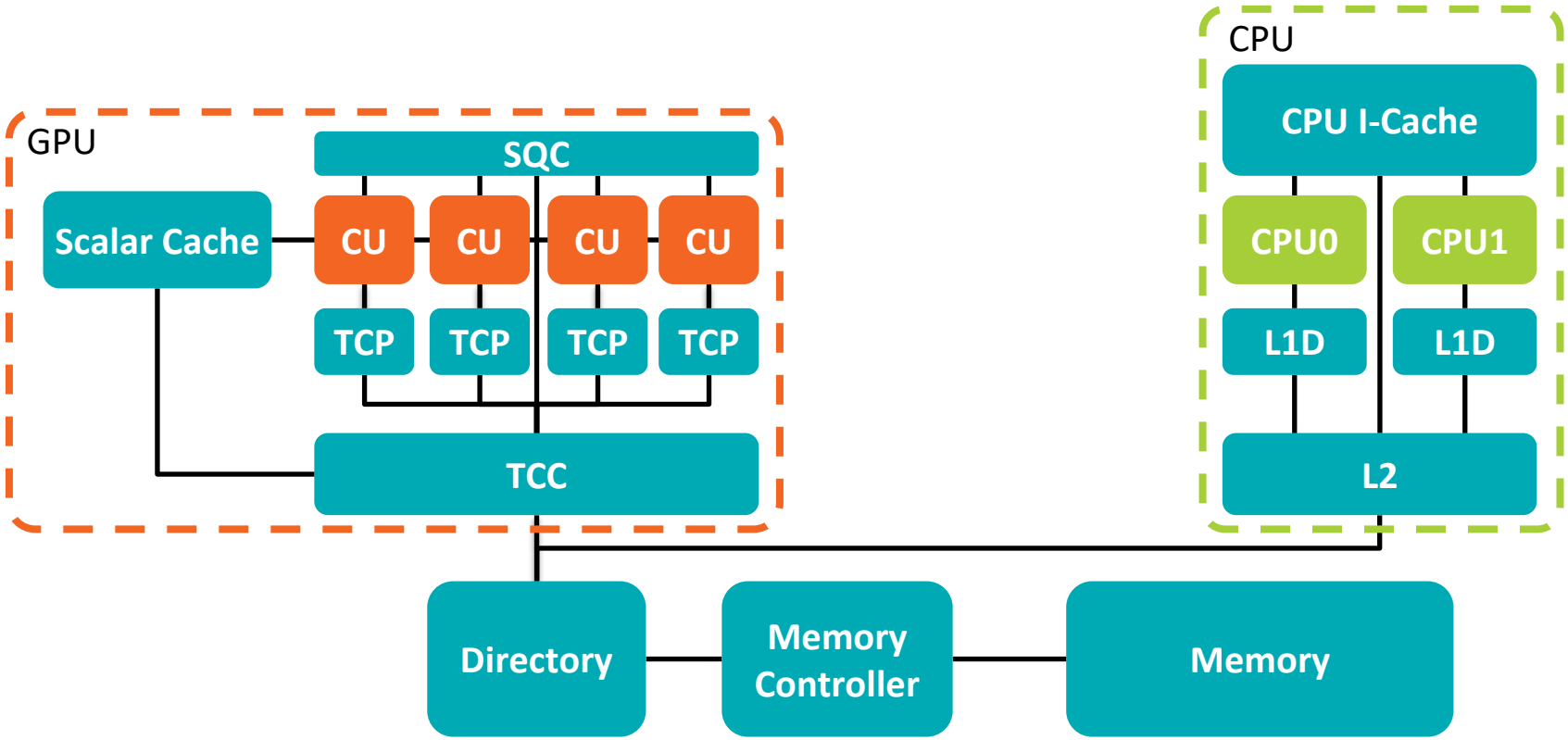
- L1 cache:
  - Coalesce requests to same cache block by different threads
  - Keep around long enough for all threads in warp/wavefront to hit
    - Once
  - **Ultimate goal**: **reduce number of requests sent to DRAM**

- 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

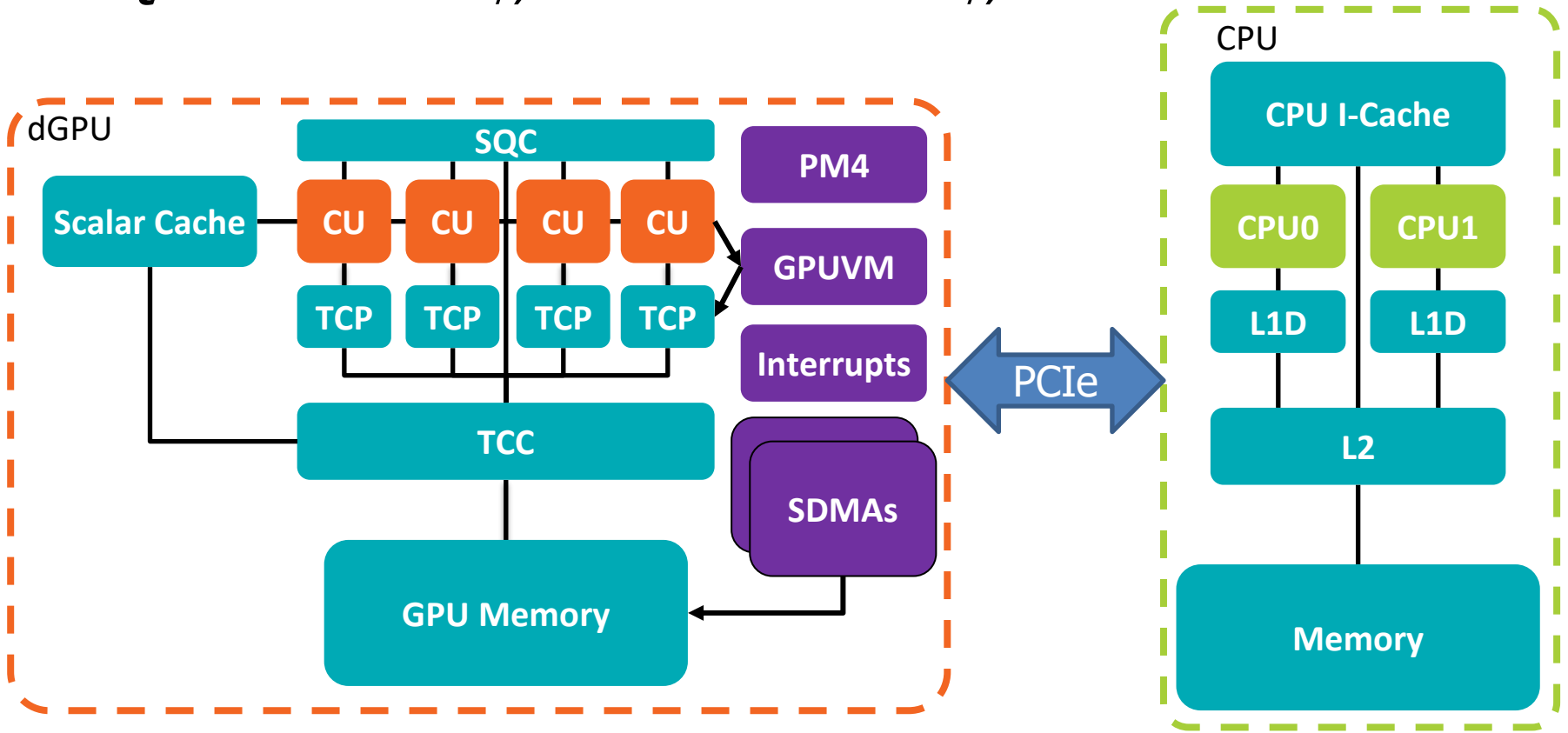
# APU

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



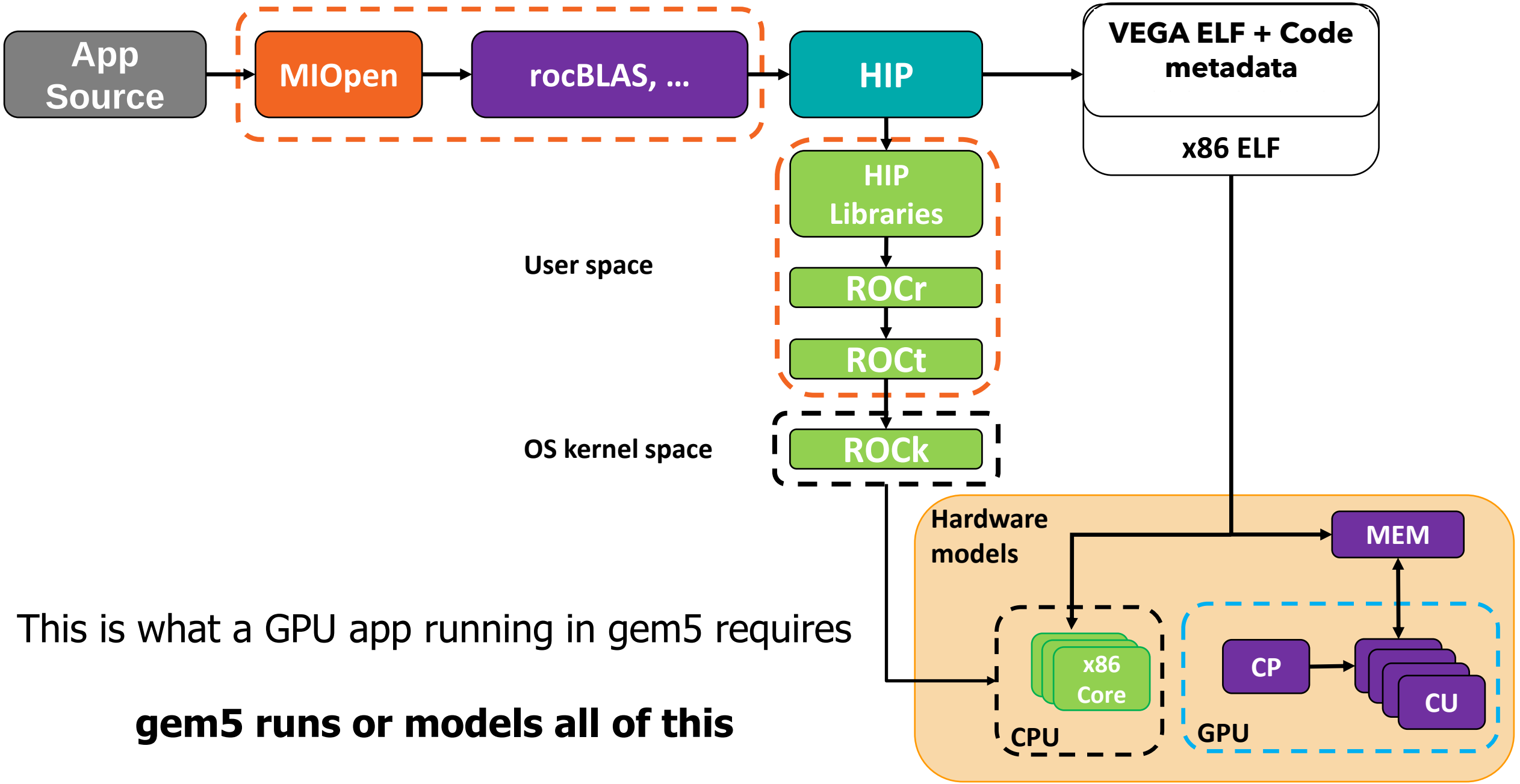
# dGPU

- dGPU = CPU and GPU have separate, discrete address spaces
- GPU Virtual Memory (GPUVM), DMA engines (SDMA), PM4 packet processor, host data bypass path, and interrupt handler are added (purple boxes):
- *Sidenote: SQC = GPU L1 I$, TCP = GPU L1 D$, TCC = unified GPU L2$*



# Outline

- Background: GPU Architecture & Programming Basics
- **Modeling & Using GPUs in gem5**
  - **What libraries are required?**
  - Where is GPU code located?
  - What support is provided?
- Running GPU programs in gem5




This is what a GPU app running in gem5 requires

**gem5 runs or models all of this**

# AMD's ROCm Stack

- ROCm == Radeon Open Compute
- ROCm stack
  - Runtime layer – ROCr
  - Thunk (user-space driver) – ROCt
  - Kernel fusion driver (KFD) – ROCk (in linux)
  - MIOpen – machine intelligence (ML) library
  - rocBLAS – BLAS (e.g., GEMMs) library
  - HIP – GPU programming language (roughly: LLVM backend, clang front-end)
  - …

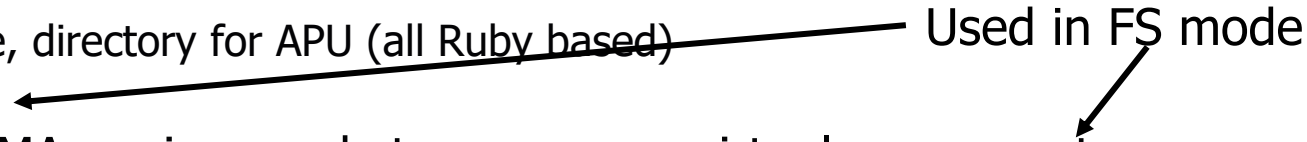
- In SE mode, gem5 simulates all of these except ROCk, which it emulates through docker
- In FS mode, the disk image contains the entire ROCm stack and gem5 simulates it

# Outline

- Background: GPU Architecture & Programming Basics
- **Modeling & Using GPUs in gem5**
  - What libraries are required?
  - **Where is GPU code located?**
  - What support is provided?
- Running GPU programs in gem5

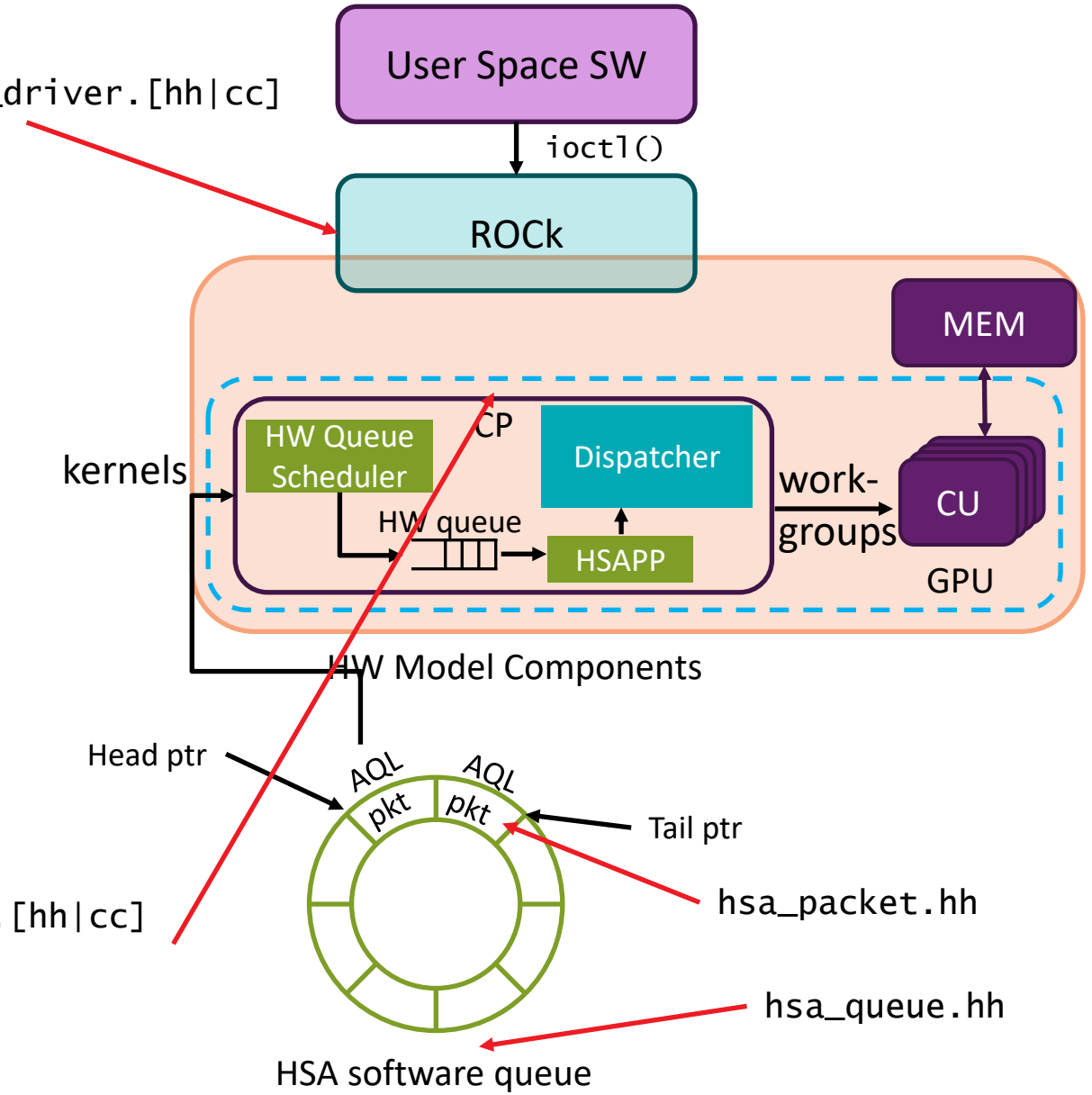
# GPU Model Codebase

- gem5 ← Top-level directory
  - `src/`
    - `arch/amdgpu/`
      - `vega/` ← Vega ISA
    - `gpu-compute/` ← GPU core (CU) model
      - Instruction buffering, Registers, Vector ALUs
    - `mem/protocol/` ← Memory model
    - `mem/ruby/` ← Memory model
      - L1I cache, L1D cache, L2 cache, directory for APU (all Ruby based)
    - `dev/hsa/` ← HSA device models
    - `dev/amdgpu/` ← ROCr runtime, DMA engine, packet processors, virtual memory etc.
  - `configs/`
    - `example/` ← `apu_se.py`
      - Connects multiple CUs, caches, etc. together to create overall APU model
    - `example/gpufs` ← mi200.py, mi300.py, Disjoint_VIPER.py
      - mi200.py, mi300.py connect multiple Cus, caches, together to create overall dGPU model
      - Disjoint_VIPER.py configures the cache hierarchy and interconnects
    - `ruby/` ← GPU_VIPER.py
      - APU protocol configs



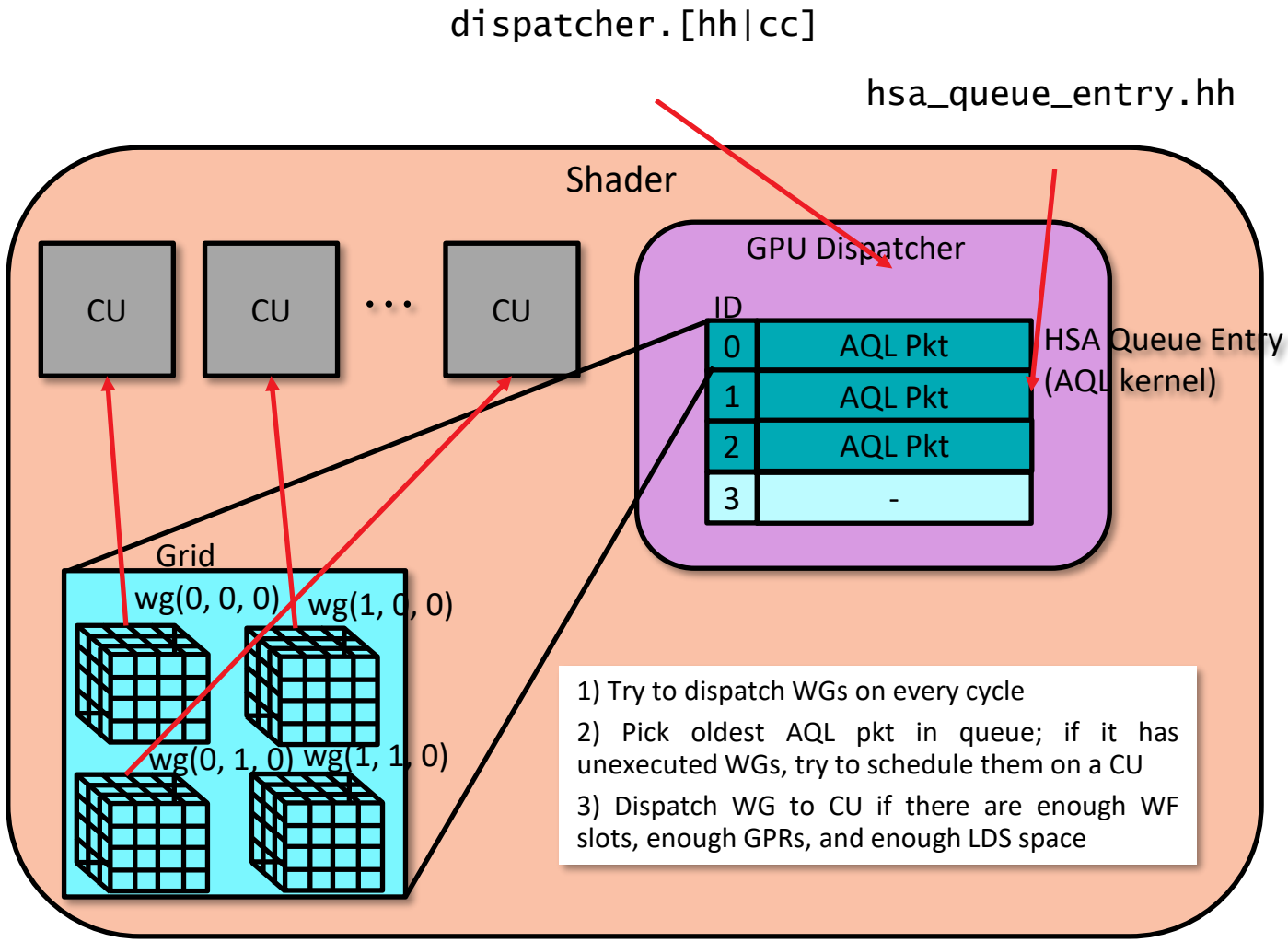
# GPU Kernel Execution

- User space SW talks to GPU via ioctl()
  - ROCk (handles ioctl) is emulated in SE mode
  - ROCk is simulated in FS mode
- 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…



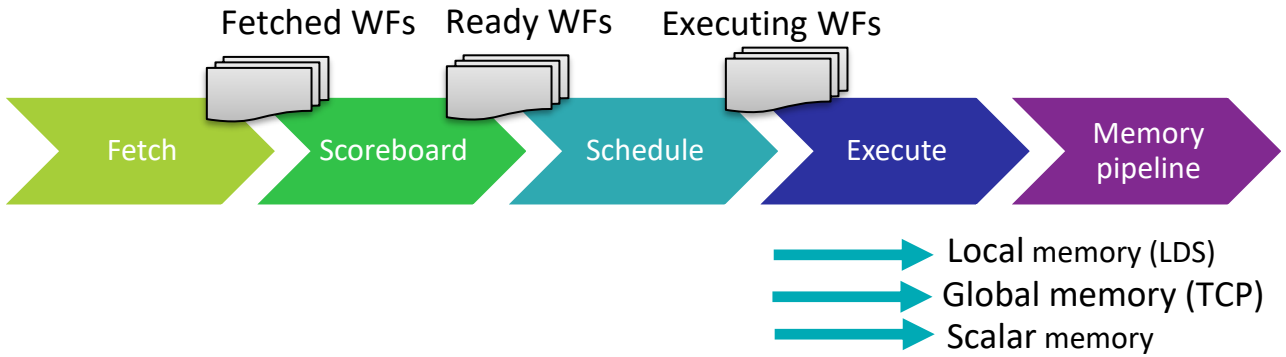
# GPU Kernel Execution (cont.)

- 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



# GPU Execution Pipeline

- 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
- **Modeling & Using GPUs in gem5**
  - What libraries are required?
  - Where is GPU code located?
  - **What support is provided?**
- Running GPU programs in gem5

# Current Support

- ROCm supported in gem5: ROCm v4.0 (SE), ROCmv6.1 (FS)
- AMD GPU support
  - Vega (gfx900 – dGPU, gfx902 – APU)
  - MI200 (gfx90a – dGPU), MI300 (gfx942 – dGPU)
  - MI200/MI300 models tensor cores but use a VEGA-like ISA
  - MI200 is currently better tested than MI300
  - If you want to run application on the VEGA, MI200, or MI300 model in gem5, you need to compile applications for the appropriate gfx9* model
  - Only officially supported gfx9* GPUs can be run in gem5 Full System with real driver
- Standard library: currently not supported – use apu_se.py, mi200.py, mi300.py instead
- Currently only supports Ruby
- **GPUFS is only supported on Vega/MI300 with dGPU devices**

# GPU Full System Simulation

- While SE mode simulates an APU, GPU full system mode (GPUFS) simulates application in dGPU
  - **Caveat**: As of gem5 24.0, X86 KVM CPU and Atomic CPU are supported
  - When using KVM CPU, 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
  - GPU virtual memory support is available in GPUFS
  - Faster simulation speeds because of simpler CPU models

# Creating Portable gem5 Resources

- Docker container
  - Properly installs ROCm software stack for use in SE mode
  - Used for compiling applications for both SE and FS mode


- **Publicly Available!**
  - Integrated into gem5 repo: https://github.com/gem5/gem5
  - 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

- **Some of our experiments today will assume this docker support**
  - `docker pull ghcr.io/gem5/gcn-gpu:v24-0` ← For running gem5 v24.0 in SE mode
  - `docker pull ghcr.io/gem5/gpu-fs:latest` ← For compiling applications for gem5 v24.0 in FS mode

Not needed today since the codespace has all the required dependencies
Might be required later if host system cannot compile GPGPU apps

# Outline

- Background: GPU Architecture & Programming Basics
- Modeling & Using GPUs in gem5
  - What libraries are required?
  - Where is GPU code located?
  - What support is provided?
- **Running GPU programs in gem5**
  - **Running in SE mode**
  - Running in FS mode
  - Checkpoint Creating and Restoration
  - Running ML workloads in PyTorch in gem5

# 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)
  - Already downloaded into `workspaces/gem5-bootcamp-env`
- Running:
  - `cd /workspaces/gem5-bootcamp-env/`
  - `docker run –v $(pwd):$(pwd) -v /usr/local/bin:/usr/local/bin -w $(pwd) ghcr.io/gem5/gcn-gpu:v24-0 gem5-vega-se gem5/configs/example/apu_se.py -n 3 -c 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**

# Output Statistics

- 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)
:
```

# Output Statistics (cont.)

- Some other stats unique to GPUs and not available from profiling tools:
  - CU-0's L1 cache misses and hits: `system.tcp_cntrl0.L1cache.m_demand_misses` and `system.tcp_cntrl0.L1cache.m_demand_hits`
  - L2 cache misses and hits: `system.tcc_cntrl0.L2cache.m_demand_misses` and `system.tcc_cntrl0.L2cache.m_demand_hits`
  - CU-0's LDS bank conflict: `system.cpu3.CUs0.ldsBankConflictDist::total`
  - Number of coalesced accesses at CU-0's L1: `system.l1_coalescer0.coalescedAccesses`
  - CU-0's vector ALU utilization: `system.cpu3.CUs0.vALUUtilization`
  - And many more…

# GPU Configuration Parameters

- Some parameters used to configure GPUs

```
parser.add_argument(
    "-u",
    "--num-compute-units",
    type=int,
    default=4,
    help="number of GPU compute units",
),
```

```
parser.add_argument(
    "--mem-req-latency",
    type=int,
    default=50,
    help="Latency for requests from the cu to ruby.",
)
parser.add_argument(
    "--mem-resp-latency",
    type=int,
    default=50,
    help="Latency for responses from ruby to the cu.",
)
```

```
parser.add_argument(
    "--vreg-file-size",
    type=int,
    default=2048,
    help="number of physical vector registers per SIMD",
)
parser.add_argument(
    "--vreg-min-alloc",
    type=int,
    default=4,
    help="Minimum number of registers that can be allocated "
    "from the VRF. The total number of registers will be "
    "aligned to this value.",
)

parser.add_argument(
    "--sreg-file-size",
    type=int,
    default=2048,
    help="number of physical vector registers per SIMD",
)
parser.add_argument(
    "--sreg-min-alloc",
    type=int,
    default=4,
    help="Minimum number of registers that can be allocated "
    "from the SRF. The total number of registers will be "
    "aligned to this value.",
)
```

**And many more in**
**gem5/configs/example/apu_se.py**

**Example command:**

```
gem5-vega-se configs/example/apu_se.py -n 3 -c
square -num-compute-units=20 [<other options>…]
```

# GPU Configuration Parameters (cont.)

```
    parser.add_argument(
        "--num-tccs",
        type=int,
        default=1,
        help="number of TCC banks in the GPU",
    )
    parser.add_argument(
        "--sqc-size", type=str, default="32kB", help="SQC cache size"
    )
    parser.add_argument(
        "--sqc-assoc", type=int, default=8, help="SQC cache assoc"
    )

    parser.add_argument(
        "--WB_L2", action="store_true", default=False, help="writeback L2"
    )
    parser.add_argument(
        "--TCP_latency",
        type=int,
        default=4,
        help="In combination with the number of banks for the "
        "TCP, this determines how many requests can happen "
        "per cycle (i.e., the bandwidth)",
    )
    parser.add_argument(
        "--mandatory_queue_latency",
        type=int,
        default=1,
        help="Hit latency for TCP",
    )
    parser.add_argument(
        "--TCC_latency", type=int, default=16, help="TCC latency"
    )
    parser.add_argument(
        "--tcc-size", type=str, default="256kB", help="agregate tcc size"
    )
    parser.add_argument("--tcc-assoc", type=int, default=16, help="tcc assoc")
    parser.add_argument(
        "--tcp-size", type=str, default="16kB", help="tcp size"
    )
```

```
parser.add_argument(
    "--glc-atomic-latency", type=int, default=1, help="GLC Atomic Latency"
)
parser.add_argument(
    "--atomic-alu-latency", type=int, default=0, help="Atomic ALU Latency"
)
parser.add_argument(
    "--tcc-num-atomic-alus",
    type=int,
    default=64,
    help="Number of atomic ALUs in the TCC",
)
parser.add_argument(
    "--tcp-num-banks",
    type=int,
    default="16",
    help="Num of banks in L1 cache",
)
parser.add_argument(
    "--tcc-num-banks",
    type=int,
    default="16",
    help="Num of banks in L2 cache",
)
parser.add_argument(
    "--tcc-tag-access-latency",
    type=int,
    default="2",
    help="Tag access latency in L2 cache",
)
parser.add_argument(
    "--tcc-data-access-latency",
    type=int,
    default="8",
    help="Data access latency in L2 cache",
)
```

**And many more in `gem5/configs/ruby/GPU_VIPER.py`**

# Outline

- Background: GPU Architecture & Programming Basics
- Modeling & Using GPUs in gem5
  - What libraries are required?
  - Where is GPU code located?
  - What support is provided?
- **Running GPU programs in gem5**
  - Running in SE mode
  - **Running in FS mode**
  - Checkpoint Creating and Restoration
  - Running ML workloads in PyTorch in gem5

# Creating GPUFS Resources

- Docker Container
  - Contains an installation of the ROCm software stack
  - Used to **build** applications to run in full system simulations
- Disk Image & Linux Kernel
  - Linux Kernel: https://storage.googleapis.com/dist.gem5.org/dist/v24-0/gpu-fs/kernel/vmlinux-gpu-ml.gz
  - Disk Image: https://storage.googleapis.com/dist.gem5.org/dist/v24-0/gpu-fs/diskimage/x86-ubuntu-gpu-ml.gz
  - Contains a version of Linux and ROCm to be used for Full System **simulation**
  - Pre-downloaded and present in `/workspaces/gem5-bootcamp-env`
- Disks can also be created manually for more recent versions

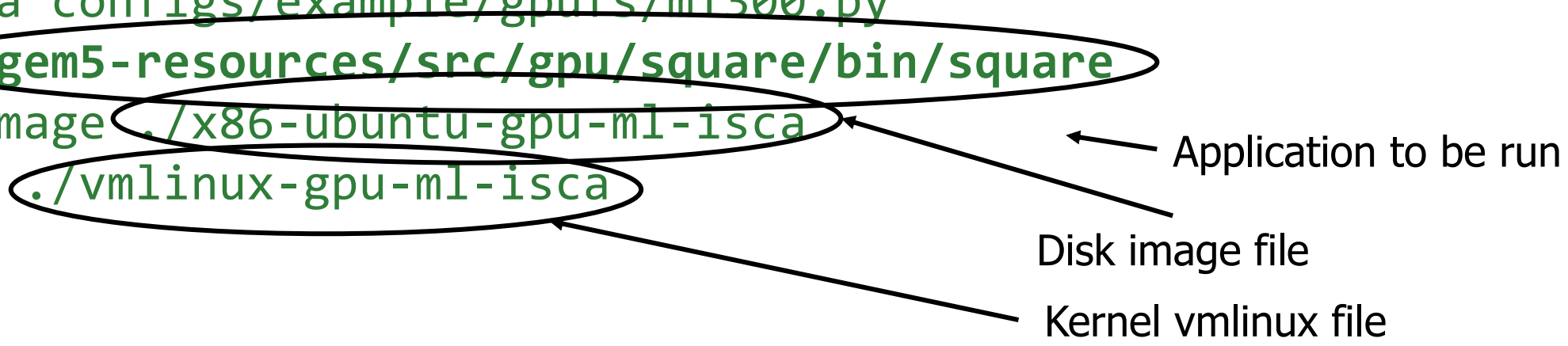
# Building Square for GPUFS

- Need a different binary because we will be simulating the latest MI300 GPU in FS mode
- If m5ops hasn't been built already:
  - `cd /workspaces/gem5-bootcamp-env/gem5/util/m5`
  - `scons build/x86/out/m5`
- To build:
  - `cd /workspaces/gem5-bootcamp-env/gem5-resources/src/gpu/square`
  - `cp /workspaces/gem5-bootcamp-env/materials/isca24/10-gpufs/Makefile ./`
  - `cd gem5-resources/src/gpu/square`
  - `make`

These commands are also present in `/workspaces/gem5-bootcamp-env/materials/isca24/10-gpufs/README`

# Running Square in GPUFS

- Running GPUFS does not require docker since all the required libraries are part of the disk image. Any library or system call gets simulated in FS mode through calls to respective files in disk image
- Command:
  - `cd /workspaces/gem5-bootcamp-env`
  - `gem5-vega configs/example/gpufs/mi300.py --app ./gem5-resources/src/gpu/square/bin/square --disk-image ./x86-ubuntu-gpu-ml-isca --kernel ./vmlinux-gpu-ml-isca`




Note: All files passed to command lines are **inputs** and must be valid
This requires that you have built the disk image and kernel

CPU's console output redirected to `m5out/system.pc.com_1.device`

# Outline

- Background: GPU Architecture & Programming Basics
- Modeling & Using GPUs in gem5
  - What libraries are required?
  - Where is GPU code located?
  - What support is provided?
- **Running GPU programs in gem5**
  - Running in SE mode
  - Running in FS mode
  - **Checkpoint Creating and Restoration**
  - Running ML workloads in PyTorch in gem5

# Checkpoint Creation and Restoration

- Simulations are very time consuming – large scale machine learning applications can take several days to finish
- However, only certain parts of the application are of interest to computer architect
  - A few iterations of the core algorithm might be enough to study the effect of the underlying system on its performance
- In such situations, simulation speed can be significantly improved by checkpointing an application right before a region of interest
  - Later simulations can then resume from this point onwards and save time that would otherwise be spent running uninteresting code

# Checkpoint Creation (cont.)

- Checkpointing in square

```
23 #include <stdio.h>
24 #include "hip/hip_runtime.h"
25 #include "gem5/m5ops.h"
26 #include <m5_mmap>
27
28 #define CHECK(cmd) \
29 {\
30     hipError_t error  = cmd;\
31     if (error != hipSuccess) { \
32       fprintf(stderr, "error: '%s'(%d) at %s:%d\n", hipGetErrorString(error), error,__FILE__, __LINE__); \
33     exit(EXIT_FAILURE);\
34     }\
35 }
36
37 /*
38  * Square each element in the array A and write to array C.
39  */
40 template <typename T>
41 __global__ void
42 vector_square(T *C_d, const T *A_d, size_t N)
43 {
44     size_t offset = (hipBlockIdx_x * hipBlockDim_x + hipThreadIdx_x);
45     size_t stride = hipBlockDim_x * hipGridDim_x ;
46
47     for (size_t i=offset; i<N; i+=stride) {
48         C_d[i] = A_d[i] * A_d[i];
49     }
50 }
51
52
53 int main(int argc, char *argv[])
54 {
55     float *A_h, *C_h;
56     size_t N = 1000000;
57     size_t Nbytes = N * sizeof(float);
58     m5op_addr = 0xFFFF0000;
59     map_m5_mem();
60     hipDeviceProp_t props;
61     CHECK(hipGetDeviceProperties(&props, 0/*deviceID*/));
62     printf ("info: running on device %s\n", props.name);
63     #ifdef __HIP_PLATFORM_HCC__
64       printf ("info: architecture on AMD GPU device is: %d\n",props.gcnArch);
65     #endif
66     printf ("info: allocate host and device mem (%6.2f MB)\n", 2*Nbytes/1024.0/1024.0);
67     CHECK(hipHostMalloc(&A_h, Nbytes));
68     CHECK(hipHostMalloc(&C_h, Nbytes));
69     // Fill with Phi + i
70     for (size_t i=0; i<N; i++)
71     {
72         A_h[i] = 1.618f + i;
73     }
74     m5_checkpoint_addr();
75     const unsigned blocks = 512;
76     const unsigned threadsPerBlock = 256;
```

Include files for checkpoint creation

Initialize m5ops variables required for FS mode

Create checkpoint at this place

# Checkpoint Creation (cont.)

- Update the `/workspaces/gem5-bootcamp-env/gem5-resouces/src/gpu/square/Makefile` with:
  - `GEM5_PATH = /workspaces/gem5-bootcamp-env/gem5`
  - `CFLAGS += -I$(GEM5_PATH)/include`
  - `CFLAGS += –I$(GEM5_PATH)/util/m5/src`
  - `LDFLAGS += -I$(GEM5_PATH)/util/m5/build/x86/out –lm5`

- Not required for today's tutorial. We already copied a Makefile with these updates when building square

# Checkpoint Creation (cont.)

- To build:
  - `cd /workspaces/gem5-bootcamp-env`
  - `cp materials/isca24/10-gpufs/square-cpt/square.cpp gem5-resources/src/gpu/square/`
  - `cp materials/isca24/10-gpufs/mi300.py gem5/configs/example/gpufs/`
  - `cd gem5-resources/src/gpu/square`
  - `make clean && make`

# Checkpoint Creation (cont.)

- Running:
  - `cd /workspaces/gem5-bootcamp-env`
  - ```
    gem5-vega gem5/configs/example/gpufs/mi300.py
    --disk-image ./x86-ubuntu-gpu-ml-isca
    --kernel ./vmlinux-gpu-ml-isca
    --app ./gem5-resources/src/gpu/square/bin/square
    --checkpoint-dir=m5out/
    ```
- Creates a checkpoint file `m5.cpt` in `m5out/`
  - Contains values of all registers, TLB entries, HSA queue states, packet processor states, etc.
  - Also contains a dump of the GPU memory contents and a list of all the addresses that were in the cache when checkpoint was taken
  - Checkpoints can only be taken at kernel boundaries
    ```

# Checkpoint Restoration

- Running:
  - `gem5-vega gem5/configs/example/gpufs/mi300.py --disk-image ./x86-ubuntu-gpu-ml-isca --kernel ./vmlinux-gpu-ml-isca --app ./gem5-resources/src/gpu/square/bin/square --restore-dir m5out/`
- Restores all the state captured in the checkpoint state and resumes execution from the instruction right after the call to `m5_checkpoint_addr()`.
- Restoration includes mapping the memory dump to GPU's memory and warming up the cache.

# Outline

- Background: GPU Architecture & Programming Basics
- Modeling & Using GPUs in gem5
  - What libraries are required?
  - Where is GPU code located?
  - What support is provided?
- **Running GPU programs in gem5**
  - Running in SE mode
  - Running in FS mode
  - Checkpoint Creating and Restoration
  - **Running ML workloads in PyTorch in gem5**

# Running PyTorch in gem5

- Running with PyTorch uses the *full-system* mode in gem5
- We assume no knowledge of PyTorch here
- We will go through the quickstart guide of PyTorch but running in gem5
  - https://pytorch.org/tutorials/beginner/basics/quickstart_tutorial.html
- The quickstart runs the MNIST workload
- The disk image has been preloaded with datasets for MNIST
- We have provided some modified PyTorch applications for gem5
  - These reduce the number of batches, use fast-forwarding techniques, etc.
  - We will demonstrate two fast-forward techniques: Skip-to-kernel and KVM-based
  - **git clone https://github.com/abmerop/gem5-pytorch**

# General strategies for fast-forwarding PyTorch

- KVM-based
  - We can run portions on the CPU (using KVM) and other portions on simulated GPU
  - Requires modifications to the PyTorch code
  - Example: Run training on CPU, move model to GPU, run inference on GPU

- Skip-to-kernel
  - Useful to simulate specific kernels
  - Requires that kernel has no data dependencies
  - Most ML workloads don’t not necessarily depend on data from previous kernels
  - Do not expect to get correct results, but will simulate a kernel

# Running full-system mode

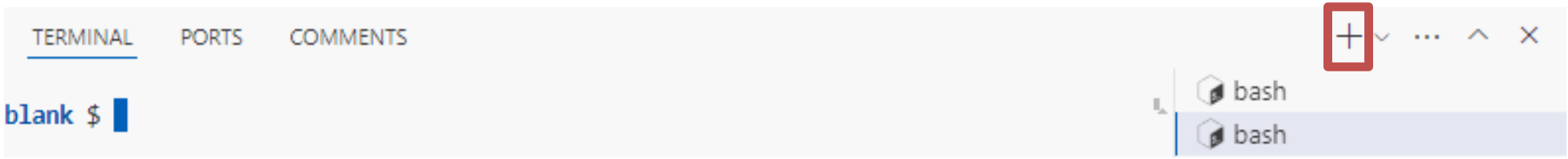
- GPUFS does not yet support stdlib which was taught earlier
- Use the legacy configs instead
- Full-system runs an unmodified software stack including kernel driver
- This limits to officially supported devices
- In general, a GPUFS command looks as follows for this tutorial:

```
/usr/local/bin/gem5-vega gem5/configs/example/gpufs/mi300.py
    --disk-image /tmp/x86-ubuntu-gpu-ml-isca --kernel ./vmlinux-gpu-ml-isca
    --no-kvm-perf --app gem5-pytorch/pytorch_test.py
```

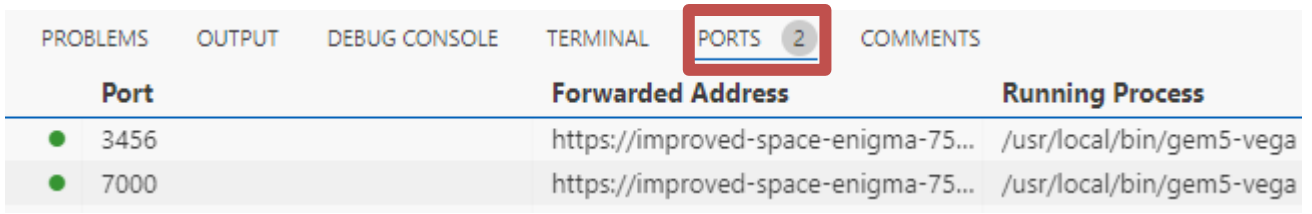
- The option provided to --app will be copied from the host into gem5

# Interacting with full-system mode

- Interact using a the *m5term* utility
- In util/term/ in the gem5 repository:
  - make
- Connect to gem5 while running
  - Open a new terminal in codespace:




  - ./util/term/m5term <port>
  - Port is printed in gem5 output and in codespace



# PyTorch MNIST example

- Full simulations will take several days
- We provide three examples:
  - Training 1 epoch, 1 batch: gem5-pytorch/MNIST/train_1batch/
  - Inference 1 epoch, 1 batch: gem5-pytorch/MNIST/test_1batch/
  - Training on CPU + Inference on GPU: (Shown below)

```
# Train all batches, one epoch on CPU
epochs = 1
for t in range(epochs):
    print(f"Epoch {t+1}\n-------------------------------")
    train(train_dataloader, model, loss_fn, optimizer)

# Copy the CPU trained model to GPU and run 1 batch (gem5_batches == 1)
gpu_model = model.to('cuda')
gpu_loss_fn = loss_fn.to('cuda')
test(test_dataloader, gpu_model, gpu_loss_fn, 'cuda', gem5_batches)
```

# PyTorch MNIST example

**/usr/local/bin/gem5-vega gem5/configs/example/gpufs/mi300.py --disk-image /tmp/x86-ubuntu-gpu-ml-isca --kernel ./vmlinux-gpu-ml-isca --no-kvm-perf --app gem5-pytorch/MNIST/test_1batch/pytorch_qs_mnist.py**

```
/usr/local/bin/gem5-vega gem5/configs/example/gpufs/mi300.py --disk-image
/tmp/x86-ubuntu-gpu-ml-isca --kernel ./vmlinux-gpu-ml-isca --no-kvm-perf --app
gem5-pytorch/MNIST/train_1batch/pytorch_qs_mnist.py
```

```
/usr/local/bin/gem5-vega gem5/configs/example/gpufs/mi300.py --disk-image
/tmp/x86-ubuntu-gpu-ml-isca --kernel ./vmlinux-gpu-ml-isca --no-kvm-perf --app
gem5-pytorch/MNIST/kvm-ff/pytorch_qs_mnist.py
```

# Adding files to disk image

- Many PyTorch workload require input data or have multiple files
- GPUFS scripts can copy in a single file only
- Mount disk image to copy files in:

```
mkdir mnt
mount -o loop,offset=$((2048*512)) /tmp/x86-ubuntu-gpu-ml-isca mnt
```

- Copy nanoGPT into disk image

```
cp -r gem5-pytorch/nanoGPT/nanoGT-ff/ mnt/root/
```

- Unmount image: `umount mnt`

# PyTorch nanoGPT example

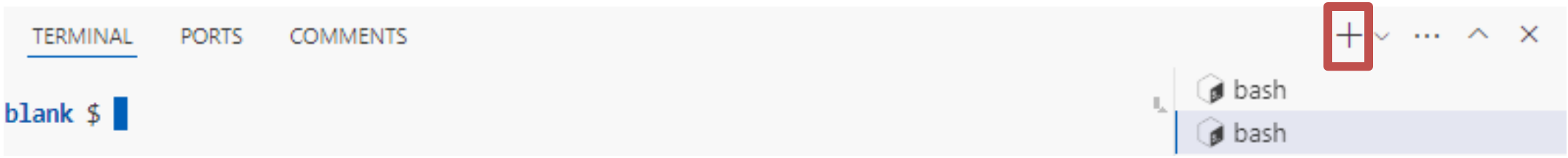
- Forked from https://github.com/karpathy/nanoGPT/
- nanoGPT small enough to generate tokens at reasonable rate in gem5
- Still several days to simulate a full run
- We can fast-forward to kernels of interest using *skip-to-kernel* feature
- Example:

```
build/VEGA_X86/gem5.opt -d tutorial_nanogpt --debug-flags=GPUCommandProc
configs/example/gpufs/mi300.py --disk-image gem5-resources/src/x86-ubuntu-gpu-ml/disk-image/x86-ubuntu-
gpu-ml  --kernel gem5-resources/src/x86-ubuntu-gpu-ml/vmlinux-gpu-ml  --app gem5-pytorch/nanoGPT/train-ff.sh
--skip-until-gpu-kernel=8 --exit-after-gpu-kernel=9
```

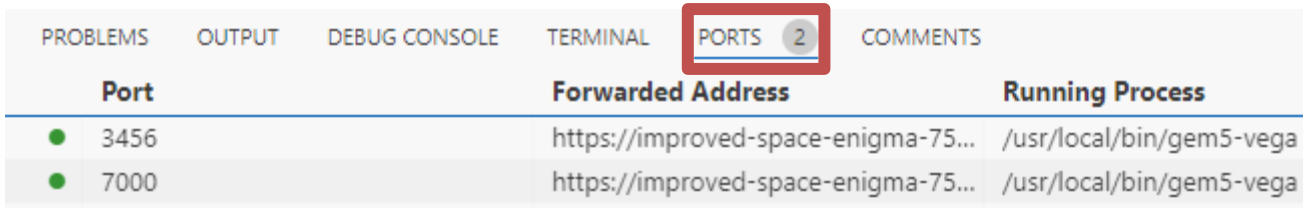
- Can select arbitrary kernel to skip to
  - Typically, would want to use hardware profiling to find kernel or interest

# PyTorch interactive

- We have shown some examples, now you can try your own PyTorch script
- You can run PyTorch interactively using m5term
- In util/term/ in the gem5 repository:
  - make
- Connect to gem5 while running
  - Open a new terminal in codespace:




  - ./util/term/m5term <port>
  - Port is printed in gem5 output and in codespace



# Conclusion

- In this session, we went through:
  - A GPU architecture primer
  - gem5's GPU codebase organization and model behavior
  - Current GPU software support in gem5
  - Running GPU applications in SE and FS modes
  - Checkpointing an application and restoring from it later
  - Offloading kernels onto CPU for better simulator performance
  - Leveraging PyTorch features to fast-forward

# Citing our Work

- If you use GPUFS for your research, please cite the following paper.

- *Simulation Support for Fast and Accurate Large-Scale GPGPU & Accelerator Workloads*. Vishnu Ramadas, Matthew Poremba, Bradford Beckmann and Matthew D. Sinclair. In 3rd Open-Source Computer Architecture Research (OSCAR), June 2024.