GPU Simulation

GPU Cycle Level Simulation (Part 1)

Learning Objectives

1. Describe cycle-level performance modeling for CPUs and GPUs
2. Explain queue-based performance modeling
3. Describe the basic simulation code structures for CPUs and GPUs
4. Get ready for the architecture modeling programming project

Performance Modeling Techniques

1. Cycle level simulation
2. Event driven simulation
3. Analytical Model
4. Sampling based techniques
5. Data based statistical/ML modeling
6. FPGA based emulation

Cycle Level Simulation

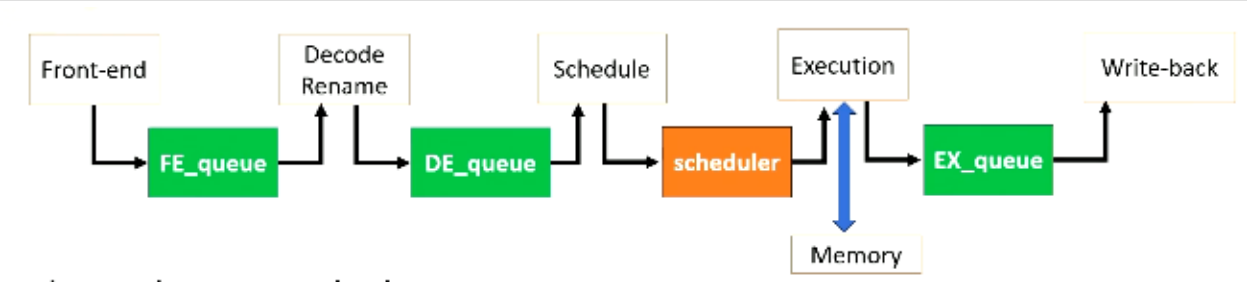
1. Commonly used in many architecture simulators
2. Typically, a global clock exists
3. Each cycle, events, such as instruction fetch and decode are modeled
4. Multiple clock domains can exist (e.g., memory clock, processor clock, network-on-chip (NoC) clock)

Execution Driven vs Trace Driven Simulation

1. Execution driven: Instructions are executed during the simulation
    - Execute-at-fetch vs execute-at-execute
    - Execute-at-fetch: Instruction is executed when fetched
    - Execute-at-execute: Instruction is executed at the execution stage
2. Trace driven: Traces are collected; simulation and execution are decoupled
3. Benefits of execution driven: No need to store traces
4. Trace driven cannot model run-time dependent behaviors (lock-acquire, barriors)
5. Trace-driven simulators are simpler and often lighter and easier to develop
    - e.g., Memory traces only for memory simulation or cache

Queue Based Modeling

1. Instructions are moving between queues
2. Scheduler selects instructions that will be sent to the execution stage among ready instrucitons; not implemented as a queue structure
3. Other queues are FIFO
4. When instruction is complete, the dependent instructions are ready
    - The dependency chain needs to be modeled and broadcasting also needs to be modeled
5. Cache and memory are modeled to provide memory instruction latency



Queue Based Modeling

Modeling Parameters with Queue Based Modeling

1. Number of cycles in each pipeline stage -> depth of queue
2. How many instructions can move between queues represent pipeline width (e.g., issue/execution bandwidth)
3. Question: How do you know the latency of each instruction?
4. Instruciton latency assumptions:
   - Instruction latency is given as a parameter (ADD takes 1 cycle, MUL takes 3 cycles)
   - Latency can be obtained from literature or simulators like CACTI or RTL simulation

5-Stage CPU Processor Modeling

```
int main(int argc, char** argv)
{
    macsim_c* sim;

    // Instantiate
    sim = new macsim_c();

    // Initialize simulation state
    sim->initialize(argc, argv);

    // Run simulation
    // report("run core (single threads)");
    while (sim->run_a_cycle())
        ;


    // Finalize simulation state
    sim->finalize();

    return 0;
}

macsim::run_a_cycle(...)
{
    cycle++;
    cpu->run_a_cycle();
    mem->run_a_cycle();
    noc->run_a_cycle();
}
```

```
cpu->run_a_cycle(...)
{
    wb->run_a_cycle();
    exec->run_a_cycle();
    sch->run_a_cycle();
    de->run_a_cycle();
    fe->run_a_cycle();
}
```

Example

1. Each modeled instruction has an op data structure
2. Op data structure tracks instruction progress and cycle
3. op->done_cycle = op->schedule_cycle + latency
4. Question: What if latency is not fixed?
    - e.g., Cache miss
    - After cache simulation (cache hit/miss hierarchy), we can know the memory instruction latency

Scheduler

1. Scheduler checks ready instructions
2. Scheduler handles resource constraints
3. e.g., # of FP instructions, # of load and store queues, execution units, ports in cache, etc.
4. Ensuring resources are available for multiple cycles

GPU Cycle-level Modeling

1. Similar to CPU modeling
2. The simulation modeling unit is a warp
    - Warp instruciton fetched/decoded/scheduled/executed
3. Scheduler chooses instructions for the head of each warp
4. Differences from CPUs:
    - In-order scheduling within a warp
    - Out-of-order across warps
5. Major differences between CPU vs GPU
    - Handling divergent warps
    - Warp, thread block, and kernel concepts
    - Scheduler

End of Simulation

1. Entire thread block scheduled to one SM
2. Tracking complete threads
3. All threads within a CUDA block, the corresponding CUDA block completes
4. When all thread blocks are complete, the kernel ends
5. When all kernel ends, the application ends

Summary

1. CPU and GPU cycle-level simulation techniques review
2. Review how to model latency and bandwidth using queues

GPU Cycle Level Simluation (Part 2)

Learning Objectives

1. Describe advanced GPU cycle-level modeling techniques such as divergent warps, coalesced memory
2. Introduce several open source GPGPU simulators

Modeling Unit

1. Instructions are modeled at a warp level
2. Accessing I-cache, PC registers
3. Reflecting microarchitecture access behavior
4. Mask bits are needed to keep track of resource constraints
5. Question: How to model divergent warps and memory coalescing?

Recap: Divergent Branches

1. Within a warp, instructions take different paths
2. Simulator also needs to model the SIMT stack
3. Execution driven simulation:
   - Faithfully model SIMT stack
4. Trace driven simulation
   - Follow how the traces are collected
   - It's challenging to simulate different divergent branch handling mechanisms than the trace collection's machine
   - The trace should contain all the paths already
   - The trace needs to have the contents of mask bits

Memory Coalescing Model

1. Modeling memory coalescing is critical
2. Memory requests need to be merged
3. Typically, this follows cache line sizes
4. Assume a 64B cache line size

Modeling Memory Coalescing with Trace

1. The trace should contain all the memory address from each warp
2. The trace generator can insert all memory instructions individually
   - e.g., va1, va2, va3, etc.
3. Or trace generator already coalesces memory requests -> can reduce the trace size
   - e.g., 0x0, 0x4, 0x8, 0xC, 0x10 etc. vs 0x0 and size 28

Cache Hierarchy Modeling

1. After addresses are coalesced, memory requests access TLB, L1, L2 caches depending on GPU microarchitecture

Sectored Cache Modeling

1. Modern GPUs adopt sectored cache
2. Sectored cache allows bringing a sector of the cache block instead of the entire cache block
3. Benefit: Reduces bandwidth
4. Drawback: Reduces spatial locality
5. Share the tag

GPU Simulators

1. Several open-source GPU simulators are available
2. GPU simulators for different ISAs

Name	ISA	Type	Architecture	Open Source
GPGPU-Sim	NVIDIA PTX/SASS	Execution driven	GPGPU only	http://www.gpgpu-sim.org/
Accel-Sim	NVIDIA PTX/SASS	Trace driven	GPGPU and accelerator	https://accel-sim.github.io/
MGPU-Sim	AMD GPU	Execution driven	Multi GPUS are supported	[ISCA2019]
Macsim	NVIDIA/Intel GPU	Trace driven	Heterogeneous computing	https://github.com/gthparch/macsim
Gem5-GPGPU-Sim	AMD GPU or NVIDIA PTX	Execution driven	Heterogeneous computing	https://cpu-gpu-sim.ece.wisc.edu/

GPU Simulators

Summary

1. Review of GPU and GPU cycle-level simulation techniques
2. To support divergent warps, it is necessary to either model the SIMT stack

Summary

1. Recap of techniques for accelerating simulation speed
2. Emphasis on sampling and workload reduction or include mask bits inside the trace
3. Modeling memory coalescing is also the most crucial in the simulation

Analytical Models of GPUs

Learning Objectives

1. Describe analytical models of GPUs
2. Apply analytical models to determine the first order of GPU design space explorations
3. Explain CPI and interval-based analysis
4. Describe the roofline model

Analytical Models

1. Analytical models do not require execution of the entire program
2. Analytical models are typically simple and capture the first order of performance modeling
3. Analytical models often provide insights to understand performance behavior

First Order of GPU Architecture Design (1)

1. Let's consider accelerating a vector dot product with a goal of 1T vector dot products per second (sum += x[i] * y[i])
2. For compute units, we need to achieve 2T FLOPS operations (multiply and add) or 1T FMA/sec
3. If GPU operates at 1GHz, 1000 FMA units are needed; at 2GHz, 500 FMA units are needed
4. Memory units need to supply 2 memory bytes with a 2TB/sec memory bandwidth

First Order of GPU Architecture Design (2)

1. 500 FMA units are approximately equal to 16 warps (with 32 threads)
2. If each SM can execute 1 warp per cycle at 2GHz and there are 16 SMs, it can compute 1T vector dot products
3. Alternatively, 8 SMs with 2 warps per cycle can also achieve this

First Order of GPU Architecture Design (3)

1. Multithreading: How about the total number of active warps in each SM?
2. W_width: The number of threads (warps) that can run in one cycle
   - (W_width * W_depth) number of warps are resident in one SM
3. W_depth: The number of threads (warps) that can be scheduled during one stall cycle

W_depth and W_width Hardware Constraints

1. W_width is determined by the number of ALU units (along with the width of the scheduler)
2. W_depth is determined by the number of registers (along with the number of PC registers)
3. W_depth 20 means 20x32 (W_width) x (# register per thread) number of registers are needed
4. W_depth 20 also means at least 20 x W_depth number of PC registers are needed

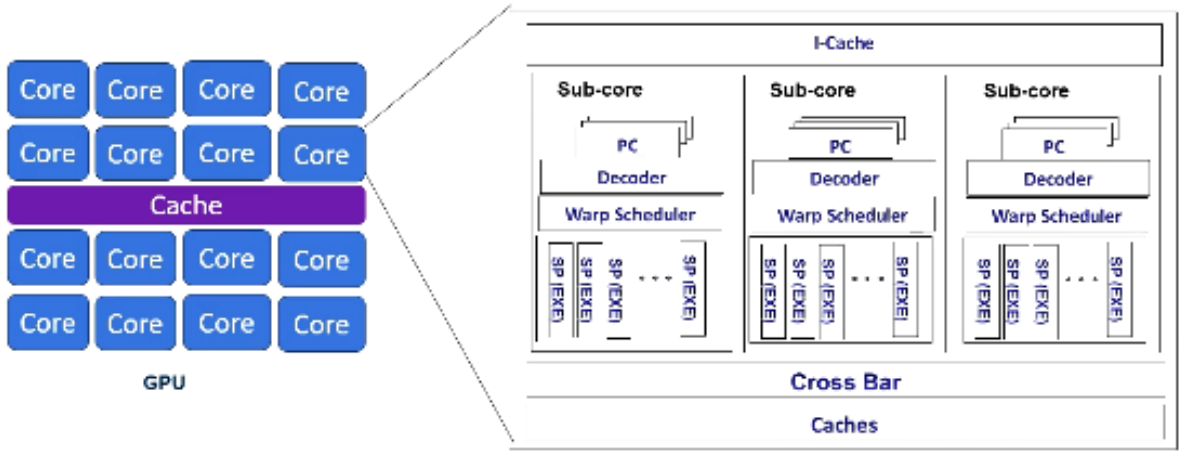
Finding W_depth

1. Strong correlation factor of W_depth is memory latency
2. In the dot product example, assume memory latency is 200 cycles
3. Case 1) 1 comp, 1 memory (dot product):
   - To hide 200 cycles, 200/(1 comp + 1 memory) = 100 warps are needed
4. Case 2) If we have 1 memory instruciton per 4 compute instructions
   - To hide 200 cycles, 200/(1 + 4) = 40 warps are needed

Decision Factors for the Number of SMs

1. Previous example: 500 FMA units
2. 1 warp x 16 SMs vs 2 warps x 8 SMs
3. Large and fewer SMs vs small and many SMs
4. Cache and registers also need to be split
5. Large cache with fewer SMs vs small cache with many SMs
6. Large cache increases cache access time, but large cache can increase cache hits among multiple CUDA blocks
7. Sub-core can also be a design decision factor
   - Many of these decisions require the analysis of trade-off between size vs time

Sub-core

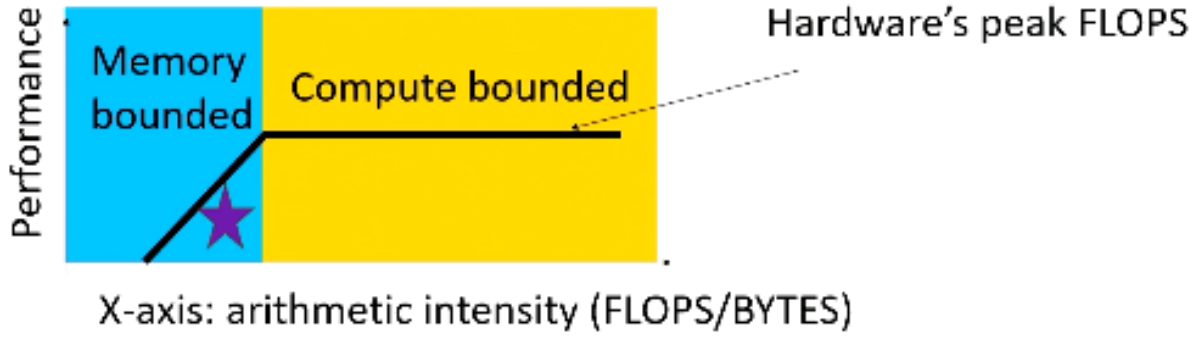


Resource management

Sub-core

Roofline Model

1. A visual performance model to determine whether an application (or a processor) is limited by the compute bandwidth or memory bandwidth)
2. Vector sum example: 2 bytes per 1 FLOPs
   - Arithmetic intensity = 0.5
3. Another example: sum += x[i] * x[i] * y[i] * y[i]
   - Arithmetic intensity = 2



Roofline

CPU (Cycle Per Instruction) Computation

1. CPI = CPI(steady state) + CPI(event1) + CPI(event2) + ...
2. CPI(steady state): Sustainable performance without any miss events
3. Example: 5 stage in-order processor
   - Assumptions:
     - CPI(steady state) = 1
     - CPI(br misprediction) = 3
     - CPI(cache miss) = 5

- 2% instructions have branch misprediction
- 5% instructions have cache misses
- Average CPI = 1 + 0.02 * 3 + 0.05 * 5 = 1.31
  - Easy to compute the average performance
  - All penalties assumed to be serialized

CPI Computation for Multi-threading

1. CPI(ideal multithreading) = CPI(single thread) / W_depth
2. W_depth: The number of warps that can be scheduled during the stall cycles
3. CPU(multithreading) = CPI(ideal multithreading) + CPI(resource contention)
4. Resource contention:
   - MSHR: Number of memory misses
   - Busy states of execution units
   - DRAM bandwidth
5. GPU is modeled for multi-threading

Interval-based Modeling: Extension of CPI Modeling

1. Simplified model
   - Interval: One steady state and one long latency event
2. First-order Modeling: Model the effect of pipeline drain and refill time
3. Depending on the events (branch misprediction vs cache miss): pipeline drain shape is different

Applying Interval Analysis on GPUs

1. Naive approach: Consider GPU as just a multithreading processor
2. Major performance differences between GPU and multi-threading processor
   - Branch divergence: Not all warps are active; some part of branch code is serialized
   - Memory divergence: Memory latency can be significantly different depending on whether memory is coalesced or uncoalesced
3. Newer models improve the performance models by modeling sub-core, sectored cache, and other resource contentions more accurately

Summary

1. CPU and GPU cycle level simulation techniques are reviewed
2. Analytical model provides the first order of processor design parameters or application performance analysis
3. CPI computation was reviewed
4. Roofline was introduced
5. Compute bounded or memory bandwidth bounded is the most critical factor

Accelerating GPU Simulation

Learning Objectives

1. Describe techniques to accelerate simulation speed
2. Identify challenges in cycle-level simulation
3. Explore techniques for accelerating simulation
4. Describe sampling based techniques

Challenges of Cycle Level Simulation

1. Cycle level simulation takes too long

2. If a simulation speed is 10KIPS (10 instructions per sec), simulating 1B instructions (1 sec in a real machine) takes ~= 28 hours
3. ML workload takes hours
    - 10 hours of ML workloads -> 100 years of simulation

Accelerating Simulations

1. Accelerating simulation itself
    - Parallelizing the simulator
    - Event drive simulation
    - Simplifying the model
    - Sampling
    - Statistical modeling
    - ML based modeling
2. Reducing the workloads
    - Micro benchmarks
    - Reducing the workload size
    - Create small representative workloads

Parallelizing Simulator

1. Each p-thread simulates a core. Cores are typically run in parallel anyway
2. Memory/NoC (network on chip) needs to be communicated
3. Bottlenecks on the memory

Event Drive Simulation

1. Instead of operating by cycle, simulator is operated with events
2. Speeding up long latency operation simulations is possible
3. New event is inserted into the event queue

Simplifying Models

1. Depending on which one to model, we could simplify the modeling
2. Non-critical components are modeled based on the average throughputs and average latency
3. Detailed modeling is needed for modeling any resource contention
4. Option 1: Simplify the pipeline
    - Instead of modeling instruction fetch/decode/execution, we could assume IPC = issue width
    - Model only caches and memory
        - z-sim CPU simulator simplifies the core pipeline
5. Option 2: Simplify the memory model
    - Assume memory system has a fixed latency

Sampling Techniques

1. Random sampling: Simple; randomly choose where to simulate
    - Execution drive: Fast forward the part that won't be simulated or use a checkpoint based method
    - Trace driven: Generate traces only for simulating sessions
2. Handling state information (cache, branch predictors)
    - Warm up time is needed
    - Or, running time is sufficiently long enough to overcome
3. Program phase based sampling (e.g., Simpoint)

GPU Sampling Techniques

1. Program phase based sampling is non-trivial

- Execution length for one single thread is too short

2. Solutions
   - CUDA block level sampling: Simulate 100 CUDA blocks instead of 1000s of CUDA blocks
   - Kernel level sampling: Simulate 1-2 kernels instead of 10s of kernels
   - Warp level sampling: Reduce the number of warps to simulate

Reducing Workloads

1. Reducing iteration counts
   - Machine learning workloads run 1000 iterations. Each iteration shows very similar characteristics; instead of 1000 iterations, iterate only once
2. Reducing input sizes
   - Graph algorithm traverses for 1B nodes -> graph algorithm traverses for 1M nodes
3. Identify dominant kernels
   - A program has 100s of functions
   - One or two functions dominate 90% of application time; change the application to run only those two functions

Data-driven Modeling

1. Collect important data on a program execution
   - Number of instructions
   - Number of loads
   - Number of divergent branches
   - Hardware performance counters
2. Use statistical or ML analysis to build a model
   - Cycle count = C1 x instruction ^ exp1 + C2 x FP instructions ^ exp2 + C3 x memory instructions ^ exp3
   - Find the coefficients with machine learning model

Summary

1. Recap of techniques for accelerating simulation speed
2. Emphasis on sampling and workload reduction