GPGPU-Sim 3.x
A Performance Simulator for Many-Core Accelerator Research

Tor M. Aamodt
Wilson W. L. Fung
Andrew Boktor

University of British Columbia

Version of simulator corresponding to these slides = GPGPU-Sim 3.1.0
Tutorial Goals

• Make you more effective in your research using GPGPU-Sim
  – Feel free to ask questions when you have them

• After this tutorial, you will be able to:
  – Describe what GPGPU-Sim simulates
  – Setup GPGPU-Sim and run CUDA applications on it
  – Do simple performance analysis on CUDA applications with AerialVision
  – Extend GPGPU-Sim for your own research
Quick Survey

• How many of you are:
  – Graduate students?
  – Faculty members?
  – Working for government?
  – Working for industry?
• Have you written a CUDA or OpenCL program before?
• Have you used GPGPU-Sim?
## Overview

<table>
<thead>
<tr>
<th></th>
<th>Brief Background on GPU Computing</th>
<th>40mins</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>GPGPU-Sim Overview</td>
<td>30mins</td>
</tr>
<tr>
<td>3</td>
<td>Demo 1: Setup and Run</td>
<td>15mins</td>
</tr>
<tr>
<td></td>
<td>Coffee Break (10:00 – 10:30am)</td>
<td></td>
</tr>
<tr>
<td>4a</td>
<td>Microarchitecture Timing Model</td>
<td>60mins</td>
</tr>
<tr>
<td>4b</td>
<td>Software Organization</td>
<td>30mins</td>
</tr>
<tr>
<td></td>
<td>Lunch (12:00 – 1:30pm)</td>
<td></td>
</tr>
<tr>
<td>4b</td>
<td>Software Organization (Timing)</td>
<td>60mins</td>
</tr>
<tr>
<td>4c</td>
<td>The GPU Design Space</td>
<td>30mins</td>
</tr>
<tr>
<td>5a</td>
<td>Demo 2: Debugging Tool</td>
<td>15mins</td>
</tr>
<tr>
<td></td>
<td>Coffee Break (3:15 – 3:45pm)</td>
<td></td>
</tr>
<tr>
<td>5b</td>
<td>Demo 3: Visualizing Performance</td>
<td>30mins</td>
</tr>
<tr>
<td>6</td>
<td>Extending GPGPU-Sim: Walkthrough</td>
<td>40mins</td>
</tr>
<tr>
<td>7</td>
<td>Wrap Up and Discussion</td>
<td>15mins</td>
</tr>
</tbody>
</table>
What is a GPU?

- GPU = Graphics Processing Unit
  - Highly Parallel
  - Highly Programmable
  - Commodity Hardware ("Desktop Supercomputing")
- Nvidia’s GTX580: 16 x 32-wide multiprocessors
  - 512 ALUs
- 24,576 concurrent threads
GPU Computing

- Heterogeneous computing

4 core CPU + 1536 core GPU
Why GPU?

*Slide from GTC 2011, *GPU Computing: Past, Present and Future*, David Luebke, NVIDIA*
Why GPU?

...Using Less Power

*Slide from GTC 2011, GPU Computing: Past, Present and Future, David Luebke, NVIDIA
Why GPU?

- **Single-Core Era**
  - Enabled by: Moore’s Law, Voltage Scaling
  - Constrained by: Power, Complexity
  - Assembly ➔ C/C++ ➔ Java ...

- **Multi-Core Era**
  - Enabled by: Moore’s Law, SMP architecture
  - Constrained by: Power, Parallel SW, Scalability
  - pthreads ➔ OpenMP / TBB ...

- **Heterogeneous Systems Era**
  - Enabled by: Abundant data parallelism, Power efficient GPUs
  - Temporarily Constrained by: Programming models, Comm. overhead
  - Shader ➔ CUDA ➔ OpenCL ➔ !!!

*Slide from AFDS 2011, *The Programmer’s Guide to the APU Galaxy*, Phil Rogers, AMD
Why GPU?

• OpenCL supported GPUs (besides AMD and NVIDIA)
  – Adreno™ 3xx GPU from Qualcomm
  – Mali™-T600 Series GPUs from ARM
  – HD 4000 on Intel’s Ivy Bridge

• GPU Computing is gaining broad industry support.
Programming Model

• Traditional viewpoint
  – CPU offload data parallel code sections onto the GPU

• Correct viewpoint?
  (if you want 100x speedup)
  – GPU = computation workhorse
  – CPU = sequential code “accelerator” and I/O offload engine
GPU Microarchitecture Overview (10,000 feet)

Single-Instruction, Multiple-Threads

GPU

SIMT Core Cluster

SIMT Core

SIMT Core

SIMT Core Cluster

SIMT Core

SIMT Core

SIMT Core Cluster

SIMT Core

SIMT Core

SIMT Core Cluster

SIMT Core

SIMT Core

Interconnection Network

Memory Partition

Memory Partition

Memory Partition

GDDR3/GDDR5

GDDR3/GDDR5

GDDR3/GDDR5

Off-chip DRAM
CUDA and OpenCL

- Extensions of C to support coprocessor model
- GPGPU-Sim support both
  - This tutorial focus on CUDA
    - More applications today
CUDA Thread Hierarchy

- Kernel Launch = Grid of Blocks of Threads
- Threads are scalar threads

Source: CUDA programming manual
CUDA Memory Model

• Memory Spaces
  – Shared
  – Global
  – Local
  – Constant
  – Texture

Source: CUDA programming manual
SIMT Execution Model

- Programmers see MIMD threads (scalar)
- GPU HW bundles threads into warps and runs them in lockstep on SIMD hardware

foo[] = \{4,8,12,16\};

A: v = foo[tid.x];

B: if (v < 10)

C: v = 0;

else

D: v = 10;

E: w = bar[tid.x]+v;
CUDA Syntax Highlights

• Declaration specifiers to indicate where things live
  
  __global__ void foo(...); // runs on GPU, callable from CPU
  __device__ void bar(...); // function callable from a GPU thread

• Parallel kernel launch
  
  foo<<<500, 128>>>(...); // 500 blocks, 128 threads each

• Special variables for thread identification in kernels
  
  dim3 threadIdx; dim3 blockIdx; dim3 blockDim;
CUDA Example Code

Standard C Code
void saxpy_serial(int n, float a, float *x, float *y)
{
    for (int i = 0; i < n; ++i)y[i] = a*x[i] + y[i];
}
// Invoke serial SAXPY kernel
saxpy_serial(n, 2.0, x, y);

CUDA code
__global__ void saxpy_parallel(int n, float a, float *x, float *y)
{
    int i = blockIdx.x*blockDim.x + threadIdx.x;
    if(i<n)
        y[i]=a*x[i]+y[i];
}
main() {
    … // omitted: allocate and initialize memory
    // Invoke parallel SAXPY kernel with 256 threads/block
    int nbblocks = (n + 255) / 256;
    saxpy_parallel<<<nbblocks, 256>>>(n, 2.0, x, y);
    … // omitted: transfer results from GPU to CPU
}
GPGPU-Sim in a Nutshell

• Microarchitecture timing model of contemporary GPUs
• Runs unmodified CUDA/OpenCL
• BSD License
• Focus of this tutorial: GPGPU-Sim version 3.1.0 and later
GPGPU-Sim 3.1.0

• Since GPGPU-Sim 2.1.1b:
  – Refactored for C++ Object-Oriented Implementation
  – Redesigned Timing Models
    • SIMT Core model, Cache models, GDDR5 timing … (later)
  – Asynchronous Kernel Calls
  – Concurrent Kernel Execution
  – Support for CUDA 3.1

• Since GPGPU-Sim 3.0.1:
  – Updated timing model to model Fermi more accurately
  – Much more robust SASS support
  – Support for CUDA 4.0 and later (New runtime flow)
Accuracy

RODINIA Benchmark Suite
Quadro FX5800 SASS
GPGPU-Sim 3.1.0 – Correlation: 98.37%

Accuracy

RODINIA Benchmark Suite
Quadro FX5800 SASS
GPGPU-Sim 3.1.0 – Correlation: 98.37%
Accuracy

RODINIA Benchmark Suite
Fermi SASS
GPGPU-Sim 3.1.0 – Correlation: 97.35%
Dependencies

• Linux
• CUDA Toolkit (3.1, 4.0)
• Standard Development Environment  
  – GCC, Make, etc.
• No GPU Hardware for CUDA
Citation

• If you use GPGPU-Sim (either 2.x or 3.x) in your publication, please cite our ISPASS 2009 paper:


• Please indicate which version of GPGPU-Sim you used / extended
  – E.g. “GPGPU-Sim version 3.1.0”
Session Summary

• GPU Computing
• CUDA Programming Model Concepts
  – Thread Hierarchy
  – Memory Spaces
  – SIMT Execution Model
• GPGPU-Sim:
  Timing simulator of modern GPUs
  – Good accuracy
  – Runs on systems without HW GPUs
## Overview

<table>
<thead>
<tr>
<th></th>
<th>Brief Background on GPU Computing</th>
<th>40mins</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>GPGPU-Sim Overview</td>
<td>30mins</td>
</tr>
<tr>
<td>3</td>
<td>Demo 1: Setup and Run</td>
<td>15mins</td>
</tr>
<tr>
<td></td>
<td><strong>Coffee Break (10:00 – 10:30am)</strong></td>
<td></td>
</tr>
<tr>
<td>4a</td>
<td>Microarchitecture Timing Model</td>
<td>60mins</td>
</tr>
<tr>
<td>4b</td>
<td>Software Organization</td>
<td>30mins</td>
</tr>
<tr>
<td></td>
<td><strong>Lunch (12:00 – 1:30pm)</strong></td>
<td></td>
</tr>
<tr>
<td>4b</td>
<td>Software Organization (Timing)</td>
<td>60mins</td>
</tr>
<tr>
<td>4c</td>
<td>The GPU Design Space</td>
<td>30mins</td>
</tr>
<tr>
<td>5a</td>
<td>Demo 2: Debugging Tool</td>
<td>15mins</td>
</tr>
<tr>
<td></td>
<td><strong>Coffee Break (3:15 – 3:45pm)</strong></td>
<td></td>
</tr>
<tr>
<td>5b</td>
<td>Demo 3: Visualizing Performance</td>
<td>30mins</td>
</tr>
<tr>
<td>6</td>
<td>Extending GPGPU-Sim: Walkthrough</td>
<td>40mins</td>
</tr>
<tr>
<td>7</td>
<td>Wrap Up and Discussion</td>
<td>15mins</td>
</tr>
<tr>
<td>#</td>
<td>Session Title</td>
<td>Duration</td>
</tr>
<tr>
<td>----</td>
<td>---------------------------------------------------</td>
<td>----------</td>
</tr>
<tr>
<td>1</td>
<td>Brief Background on GPU Computing</td>
<td>40mins</td>
</tr>
<tr>
<td>2</td>
<td>GPGPU-Sim Overview</td>
<td>30mins</td>
</tr>
<tr>
<td>3</td>
<td>Demo 1: Setup and Run</td>
<td>15mins</td>
</tr>
<tr>
<td></td>
<td><strong>Coffee Break (10:00 – 10:30am)</strong></td>
<td></td>
</tr>
<tr>
<td>4a</td>
<td>Microarchitecture Timing Model</td>
<td>60mins</td>
</tr>
<tr>
<td>4b</td>
<td>Software Organization</td>
<td>30mins</td>
</tr>
<tr>
<td></td>
<td><strong>Lunch (12:00 – 1:30pm)</strong></td>
<td></td>
</tr>
<tr>
<td>4b</td>
<td>Software Organization (Timing)</td>
<td>60mins</td>
</tr>
<tr>
<td>4c</td>
<td>The GPU Design Space</td>
<td>30mins</td>
</tr>
<tr>
<td>5a</td>
<td>Demo 2: Debugging Tool</td>
<td>15mins</td>
</tr>
<tr>
<td></td>
<td><strong>Coffee Break (3:15 – 3:45pm)</strong></td>
<td></td>
</tr>
<tr>
<td>5b</td>
<td>Demo 3: Visualizing Performance</td>
<td>30mins</td>
</tr>
<tr>
<td>6</td>
<td>Extending GPGPU-Sim: Walkthrough</td>
<td>40mins</td>
</tr>
<tr>
<td>7</td>
<td>Wrap Up and Discussion</td>
<td>15mins</td>
</tr>
</tbody>
</table>
GPGPU-Sim Overview

• What GPGPU-Sim simulates
  – Functional model for PTX/SASS + CUDA/OpenCL
  – Timing model for the compute part of a GPU
• Interface with CUDA applications
• What is new in GPGPU-Sim 3.1.0?
• Roadmap
Session Objective

• After this session, you will be able to:
  1. Summarize what GPGPU-Sim simulates
  2. Describe how GPGPU-Sim interfaces with CUDA applications and supports SASS
  3. Summarize the advances between GPGPU-Sim 2.1.1b and 3.1.0
What GPGPU-Sim Simulates

1. **Functional model for PTX/SASS**
   - PTX = Parallel Thread eXecution
     - A low-level, data-parallel virtual machine defined by Nvidia
     - Scalar ISA
   - SASS = Native ISA for Nvidia GPUs
   - Not DirectX, Not shader model N, Not AMD’s ISA, Not x86, Not Larrabee. **Only PTX or SASS.**

2. **Timing model for the compute part of a GPU**
   - Not for CPU or PCIe
   - Only model microarchitecture timing relevant to GPU compute
Functional Model (PTX)

- Low-level, data-parallel virtual machine by Nvidia
  - Assembly-like: Instructions
  - Parallel threads running in blocks
  - Specify resources with no limit
    - HW dispatches thread blocks according to its limit
- Scalar ISA
  - SIMT execution model
- Converging part of CUDA tool chain:

  .cu → NVCC → PTX → ptxas
  .cl → OpenCL Drv → PTX → ptxas

  G80
  GT200
  Fermi
  Kepler

June 2012
GPGPU-Sim Tutorial (ISCA 2012)
2: GPGPU-Sim Overview
for (int d = blockDim.x; d > 0; d /= 2)
{
    __syncthreads();
    if (tid < d) {
        float f0 = shared[tid];
        float f1 = shared[tid + d];
        if (f1 < f0)
            shared[tid] = f1;
    }
}

• Scalar PTX ISA
• Scalar control flow (if-branch, for-loops)
• Parallel Intrinsic (__syncthreads())
• Register allocation not done in PTX
Functional Model (SASS)

• Native ISA for Nvidia GPUs
  – Better correlation with HW GPU
• Scalar
• We convert SASS to PTXPlus
  – PTX + SASS-specific features
• For Now: G80/GT200 ISA
  – Fermi ISA in progress
When to use SASS?

• As often as possible
• Important if you use GPGPU-Sim for application performance tuning
  – `ptxas` reschedules instructions after converting PTX to SASS to increase computation-memory overlap.
  – It also convert short branches into predicated instructions.
  – In SASS (for Quadro FX 5800), shared memory and constant memory can be accessed directly as an operand of an instruction.
PTX vs. SASS

**PTX**

```plaintext
$Lt_25_13570:

ld.global.s32 %r9, [%rd5+0];
add.s32 %r10, %r9, %r8;
ld.global.s32 %r11, [%rd5+1024];
add.s32 %r8, %r11, %r10;
add.u32 %r5, %r7, %r5;
add.u64 %rd5, %rd5, %rd6;
ld.param.u32 %r6, [size];
setp.lt.u32 %p2, %r5, %r6;
@%p2 bra $Lt_25_13570;

mov.u32 %r12, 127;
setp.gt.u32 %p3, %r3, %r12;
@%p3 bra $Lt_25_14082;
ld.shared.s32 %r13, [%rd10+512];
add.s32 %r8, %r13, %r8;
st.shared.s32 [%rd10+0], %r8;
$Lt_25_14082: bar.sync 0;
```

**SASS (PTXPlus)**

```plaintext
l0x00000060:

add.half.u32 $r7, $r4, 0x00000400;
ld.global.u32 $r7, [$r4];
ld.global.u32 $r7, [$r7];
add.half.u32 $r0, $r5, $r0;
add.half.u32 $r6, $r8, $r6;
set.gt.u32.u32 $p0/$o127, s[0x0020], $r0;
add.half.u32 $r6, $r7, $r6;
add.half.u32 $r4, $r4, $r3;
@$p0.ne bra l0x00000060;

set.gt.u32.u32 $p0/$o127, $r2, const [0x0000];
@$p0.equ add.u32 $ofs2, $ofs1, 0x00000230;
@$p0.equ add.u32 $r6, $s[$ofs2+0x0000], $r6;
@$p0.equ mov.u32 s[$ofs1+0x0030], $r6;
bar.sync 0x00000000;
```

June 2012

GPGPU-Sim Tutorial (ISCA 2012)
2: GPGPU-Sim Overview

2.9
Timing Model for Compute Parts of a GPU

- GPGPU-Sim models timing for:
  - SIMT Core (SM, SIMD Unit)
  - Caches (Texture, Constant, …)
  - Interconnection Network
  - Memory Partition
  - Graphics DRAM

- It does NOT model timing for:
  - CPU, PCIe
  - Graphics Specific HW (Rasterizer, Clipping, Display… etc.)
Timing Model for GPU Micro-architecture

- GPGPU-Sim simulates the timing model of a GPU running each launched CUDA kernel.
  - Reports # cycles spent running the kernels.
  - Exclude any time spent on data transfer on PCIe bus.
  - CPU may run concurrently with asynchronous kernel launches.
Timing Model for GPU Micro-architecture

- GPGPU-Sim is a *detailed cycle-level* simulator:
  - Cycle-level model for each part of the microarchitecture
  - Research focused
    - Ignoring rare corner cases to reduce complexity
- Different from *cycle-accurate* simulator:
  - Does not match hardware 100%

GPGPU-Sim w/ SASS is ~0.97 correlated to the real HW.

- Why?
  - We can only guess the actual HW implementation
Interfacing GPGPU-Sim to Applications

• GPGPU-Sim compiles into a shared runtime library and implement the API:
  – libcudart.so ← CUDA runtime API
  – libOpenCL.so ← OpenCL API

• Static Linking no longer supported.

• Modify your LD_LIBRARY_PATH to run your CUDA app on GPGPU-Sim (See Manual)
  – Need a config file (gpgpusim.config) and an interconnection config file as well

We provide the config files for modeling a Quadro FX 5800 and a GeForce GTX 480 (Fermi).
GPGPU-Sim Runtime Flow

CUDA 3.1

Cubin
PTX

Application
Source Code (.cpp) Source Code (.cu)

nvcc + ptxas

C/C++ compiler

Executable
PTX SASS

ptxas
register usage

GPGPU-Sim

CUDA 4.0 and Later

Cuobjdump
PTX

Application
Source Code (.cpp) Source Code (.cu)

nvcc + ptxas

C/C++ compiler

Executable
PTX SASS

cuobjdump

PTX SASS ELF

ptxas
register usage

GPGPU-Sim

Cuobjdump
PTXPlus

Application
Source Code (.cpp) Source Code (.cu)

nvcc + ptxas

C/C++ compiler

Executable
PTX SASS ELF

cuobjdump

PTX SASS ELF

cuobjdump_to_ptxplus

register usage

GPGPU-Sim
Debugging and Visualization

• We provides tools for debug and visualize simulated GPU behavior.
  – **GDB macros:**
    Cycle-level debugging
  – **AerialVision:**
    High-level performance dynamics
GPGPU-Sim 3.1.0

• Since GPGPU-Sim 2.1.1b:
  – Refactored for C++ Object-Oriented Implementation
  – Redesigned Timing Models
    • SIMT Core model, Cache models, GDDR5 timing … (later)
  – Asynchronous Kernel Calls
  – Concurrent Kernel Execution
  – Support for CUDA 3.1

• Since GPGPU-Sim 3.0.1:
  – Updated timing model to model Fermi more accurately
  – Much more robust SASS support
  – Support for CUDA 4.0 and later (New runtime flow)
Roadmap

• Fermi SASS support
• Kepler Model
• Alternate Core Models
  – From simple to detailed
  – Model AMD GPUs
• AMD Graphics Core Next (GCN) ISA
• Power Model
Session Summary

• GPGPU-Sim simulates
  – PTX/SASS
  – Timing Model for GPU Compute
• It interface to CUDA/OpenCL application via a shared runtime library
• Enhancements in GPGPU-Sim 3.1.0
<table>
<thead>
<tr>
<th></th>
<th>Overview</th>
<th>Duration</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Brief Background on GPU Computing</td>
<td>40mins</td>
</tr>
<tr>
<td>2</td>
<td>GPGPU-Sim Overview</td>
<td>30mins</td>
</tr>
<tr>
<td>3</td>
<td>Demo 1: Setup and Run</td>
<td>15mins</td>
</tr>
<tr>
<td></td>
<td><strong>Coffee Break (10:00 – 10:30am)</strong></td>
<td></td>
</tr>
<tr>
<td>4a</td>
<td>Microarchitecture Timing Model</td>
<td>60mins</td>
</tr>
<tr>
<td>4b</td>
<td>Software Organization</td>
<td>30mins</td>
</tr>
<tr>
<td></td>
<td><strong>Lunch (12:00 – 1:30pm)</strong></td>
<td></td>
</tr>
<tr>
<td>4b</td>
<td>Software Organization (Timing)</td>
<td>60mins</td>
</tr>
<tr>
<td>4c</td>
<td>The GPU Design Space</td>
<td>30mins</td>
</tr>
<tr>
<td>5a</td>
<td>Demo 2: Debugging Tool</td>
<td>15mins</td>
</tr>
<tr>
<td></td>
<td><strong>Coffee Break (3:15 – 3:45pm)</strong></td>
<td></td>
</tr>
<tr>
<td>5b</td>
<td>Demo 3: Visualizing Performance</td>
<td>30mins</td>
</tr>
<tr>
<td>6</td>
<td>Extending GPGPU-Sim: Walkthrough</td>
<td>40mins</td>
</tr>
<tr>
<td>7</td>
<td>Wrap Up and Discussion</td>
<td>15mins</td>
</tr>
</tbody>
</table>
## Overview

<table>
<thead>
<tr>
<th></th>
<th>Brief Background on GPU Computing</th>
<th>40mins</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>GPGPU-Sim Overview</td>
<td>30mins</td>
</tr>
<tr>
<td>3</td>
<td>Demo 1: Setup and Run</td>
<td>15mins</td>
</tr>
</tbody>
</table>

**Coffee Break (10:00 – 10:30am)**

| 4a | Microarchitecture Timing Model   | 60mins |
| 4b | Software Organization            | 30mins |

**Lunch (12:00 – 1:30pm)**

| 4b | Software Organization (Timing)   | 60mins |
| 4c | The GPU Design Space             | 30mins |
| 5a | Demo 2: Debugging Tool           | 15mins |

**Coffee Break (3:15 – 3:45pm)**

| 5b | Demo 3: Visualizing Performance  | 30mins |
| 6  | Extending GPGPU-Sim: Walkthrough | 40mins |
| 7  | Wrap Up and Discussion           | 15mins |
Software Dependencies

• Linux
• Simulator:
  – CUDA Toolkit (3.1 or 4.0)
  – gcc, g++, make, makedepend
  – xutils, bison, flex, zlib
• AerialVision:
  – python-pmw, python-PLY, python-numpy
  – python-matplotlib, libpng12-dev
• Documentation:
  – doxygen, graphviz
Software Dependencies – 2

- Ubuntu?

```
sudo apt-get install build-essential xutils-dev
    bison zlib1g-dev flex libglul1-mesa-dev doxygen
    graphviz python-pmw python-ply python-numpy
    libpng12-dev python-matplotlib
```
Building GPGPU-Sim

• Get GPGPU-Sim:
  - git clone git://dev.ece.ubc.ca/gpgpu-sim

• Configure your environment:
  - Setup CUDA_INSTALL_PATH
  - source setup_environment release

• Compile:
  - make
Running GPGPU-Sim

• Copy configuration files to your working directory
  – From v3.x/configs

• Configure environment (if new terminal)
  – source setup_environment

• Run the benchmark!!
DEMO
<table>
<thead>
<tr>
<th></th>
<th>Brief Background on GPU Computing</th>
<th>40mins</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>GPGPU-Sim Overview</td>
<td>30mins</td>
</tr>
<tr>
<td>3</td>
<td>Demo 1: Setup and Run</td>
<td>15mins</td>
</tr>
<tr>
<td></td>
<td><strong>Coffee Break (10:00 – 10:30am)</strong></td>
<td></td>
</tr>
<tr>
<td>4a</td>
<td>Microarchitecture Timing Model</td>
<td>60mins</td>
</tr>
<tr>
<td>4b</td>
<td>Software Organization</td>
<td>30mins</td>
</tr>
<tr>
<td></td>
<td><strong>Lunch (12:00 – 1:30pm)</strong></td>
<td></td>
</tr>
<tr>
<td>4b</td>
<td>Software Organization (Timing)</td>
<td>60mins</td>
</tr>
<tr>
<td>4c</td>
<td>The GPU Design Space</td>
<td>30mins</td>
</tr>
<tr>
<td>5a</td>
<td>Demo 2: Debugging Tool</td>
<td>15mins</td>
</tr>
<tr>
<td></td>
<td><strong>Coffee Break (3:15 – 3:45pm)</strong></td>
<td></td>
</tr>
<tr>
<td>5b</td>
<td>Demo 3: Visualizing Performance</td>
<td>30mins</td>
</tr>
<tr>
<td>6</td>
<td>Extending GPGPU-Sim: Walkthrough</td>
<td>40mins</td>
</tr>
<tr>
<td>7</td>
<td>Wrap Up and Discussion</td>
<td>15mins</td>
</tr>
</tbody>
</table>
# Overview

<table>
<thead>
<tr>
<th></th>
<th>Session Title</th>
<th>Duration</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Brief Background on GPU Computing</td>
<td>40mins</td>
</tr>
<tr>
<td>2</td>
<td>GPGPU-Sim Overview</td>
<td>30mins</td>
</tr>
<tr>
<td>3</td>
<td>Demo 1: Setup and Run</td>
<td>15mins</td>
</tr>
<tr>
<td></td>
<td><strong>Coffee Break (10:00 – 10:30am)</strong></td>
<td></td>
</tr>
<tr>
<td>4a</td>
<td>Microarchitecture Timing Model</td>
<td>60mins</td>
</tr>
<tr>
<td>4b</td>
<td>Software Organization</td>
<td>30mins</td>
</tr>
<tr>
<td></td>
<td><strong>Lunch (12:00 – 1:30pm)</strong></td>
<td></td>
</tr>
<tr>
<td>4b</td>
<td>Software Organization (Timing)</td>
<td>60mins</td>
</tr>
<tr>
<td>4c</td>
<td>The GPU Design Space</td>
<td>30mins</td>
</tr>
<tr>
<td>5a</td>
<td>Demo 2: Debugging Tool</td>
<td>15mins</td>
</tr>
<tr>
<td></td>
<td><strong>Coffee Break (3:15 – 3:45pm)</strong></td>
<td></td>
</tr>
<tr>
<td>5b</td>
<td>Demo 3: Visualizing Performance</td>
<td>30mins</td>
</tr>
<tr>
<td>6</td>
<td>Extending GPGPU-Sim: Walkthrough</td>
<td>40mins</td>
</tr>
<tr>
<td>7</td>
<td>Wrap Up and Discussion</td>
<td>15mins</td>
</tr>
</tbody>
</table>
Timing Model Overview

- What is a warp?
- SIMT Core Internals
  - SIMT Frontend
  - Memory Unit
- Interconnection Network
- Clock Domains
- Memory Partition
  - DRAM Timing Model
Session Objectives

• After this session, you will be able to
  – Explain what is a warp, and how it handle branch divergence
  – Summarize the microarchitecture modeled by GPGPU-Sim
  – Group the microarchitecture components into the different clock domains
  – Explain why a DRAM timing model is needed
Thread Hierarchy Revisited

- Recall, kernel = grid of blocks of warps of threads
- Thread blocks (CTAs) contains up to 1024 threads
- Threads are grouped into warps in hardware

Each block is dispatched to a SIMT core as a unit of work: All of its warps run in the core’s pipeline until they are all done.
Warp = SIMT Execution of Scalar Threads

- **Warp** = Scalar threads grouped to execute in lockstep
- **SIMT vs SIMD**
  - **SIMD**: HW pipeline width must be known by SW
  - **SIMT**: Pipeline width hidden from SW

(★) Can still write software that assumes threads in a warp execute in lockstep (e.g. see reduction in NVIDIA SDK)
GPU Microarchitecture Overview

SIMT Core Cluster
- SIMT Core
- SIMT Core

SIMT Core Cluster
- SIMT Core
- SIMT Core

SIMT Core Cluster
- SIMT Core
- SIMT Core

Interconnection Network

Memory Partition

GDDR3/GDDR5

Memory Partition

GDDR3/GDDR5

Memory Partition

GDDR3/GDDR5

Off-chip DRAM

GDDR3/GDDR5
Inside a SIMT Core

- Fine-grained multithreading
  - Interleave warp execution to hide latency
  - Register values of all threads stays in core
Inside a SIMT Core (2.0)

- Started from a 5-stage In-Order Pipeline
  - Add fine-grained multithreading
  - Add SIMT stacks
Inside a SIMT Core (3.0)

- Redesign Model
  - Three decoupled warp schedulers
  - Scoreboard
  - Operand collector
  - Multiple SIMD functional unit
Fetch + Decode

- Arbitrate the I-cache among warps
  - Cache miss handled by fetching again later
- Fetched instruction is decoded and then stored in the I-Buffer
  - 1 or more entries / warp
  - Only warp with vacant entries are considered in fetch
Instruction Issue

• Select a warp and issue an instruction from its I-Buffer for execution
  – Round-Robin Priority
  – GT200 (e.g. Quadro FX 5800): Allow dual issue
  – Fermi: Odd/Even scheduler

• For each issued instruction:
  – Functional execution
  – Obtain info from functional simulator
  – Generate coalesced memory accesses
  – Reserve output register in scoreboard
  – Update SIMT stack
Scoreboard

- Checks for RAW and WAW dependency hazard
  - Flag instructions with hazards as *not ready* in I-Buffer (masking them out from the scheduler)
- Instructions reserves registers at issue
- Release them at writeback
foo[] = \{4,8,12,16\};

A: v = foo[tid.x];
B: if (v < 10)
C: v = 0;
else
D: v = 10;
E: w = bar[tid.x]+v;

Handles \textit{Branch Divergence}
Operand Collector

- Operand Collector Architecture (US Patent: 7834881)
  - Interleave operand fetch from different threads to achieve full utilization

```
add.s32  R3, R1, R2;  No Conflict
mul.s32  R3, R0, R4;  Conflict at bank 0
```
Operand Collector

(from instruction issue stage)

Arbitrator

Bank 0

Bank 1

Bank 2

Bank 3

Single-Ported Register File Banks

Crossbar

Collector Units

SIMD Execution Unit
ALU Pipelines

- SIMD Execution Unit
- Fully Pipelined
- Each pipe may execute a subset of instructions
- Configurable bandwidth and latency (depending on the instruction)
- Default: SP + SFU pipes
Writeback

• Each pipeline has a result bus for writeback
• Exception:
  – SP and SFU pipe shares a result bus
  – Time slots on the shared bus is pre-allocated
Memory Unit

- Model timing for memory instructions
- Support half-warp (16 threads)
  - Double clock the unit
  - Each cycle service half the warp
- Has a private writeback path
Constant Cache

• A Read-only cache for constant memory
• GPGPU-Sim simulates 1 read ports
  – A warp can access 1 constant cache locations in a single memory unit cycle
  – If more than 1 locations accessed
    • reads are serialized causing pipeline stalls
  – # of ports is not configurable
Texture Cache

• Read-only cache with FIFO retirement

• GPGPU-Sim support 1-D and 2-D textures

• 2-D locality should be preserved when texture cache blocks are fetched from memory
  – GPGPU-Sim uses a 4-D blocking address scheme to promote spatial locality in 2-D
    • Based on Hakura et al. The Design and Analysis of a Cache Architecture for Texture Mapping, ISCA 1997
Shared Memory

• Explicitly managed scratchpad memory
  – As fast as register files in absence of bank conflicts
• Threads in a block can cooperate via shared memory
• Each SIMT core has its own shared memory
  – Dynamically allocated to thread blocks
• 16kB/48kB per SIMT core in current NVIDIA GPUs (Fermi)
Shared Memory (cont.)

- Many threads accessing memory
  - Therefore Shared memory is highly banked
- Each bank serves one address per cycle
- Multiple access to a bank in a single cycle cause bank conflicts
  - Conflicting accesses must be serialized
- Shared memory in NVIDIA GPUs has 16/32 banks
  - GPGPU-Sim models 16 banks
Shared Memory Bank Conflicts

No bank conflict

8-way bank conflict

Figures taken from CUDA manual by NVIDIA
Global Memory

• Global memory is the off-chip DRAM memory
  – The largest and slowest memory available
  – Accesses must go through interconnect, memory partition and off-chip DRAM
  – Optionally cached in HW
    • L1 Data Cache
    • L2 Unified Cache
Coalescing

• Combining memory accesses made by threads in a warp into fewer transactions
  – E.g. if threads in a warp are accessing consecutive 4-byte sized locations in memory
    • Send one 128-byte request to DRAM (coalescing)
    • Instead of 32 4-byte requests

• This reduces the number of transactions between SIMT cores and DRAM
  – Less work for Interconnect, Memory Partition and DRAM
Coalescing (Cont.)

• CUDA Capability 1.3 (e.g. GTX280)
  – Coalescing done per half-warp
  – Can create 128-byte, 64-byte or 32-byte transactions

• CUDA Capability 2.0 (e.g. Fermi)
  – Coalescing done for a full warp
  – Cached: Only creates 128-byte transactions
  – Not Cached: Can create 128/64/32-byte transactions

• GPGPU-Sim supports both
Coalescing (cont.)

- Coalescing example

![Diagram showing coalescing example]

- One 128-Byte Transaction
- Two 128-Byte Transactions

Figures taken from CUDA manual by NVIDIA

= 4-bytes in memory
L1 Data Cache

- For both local and global memory space
  - With different policies

<table>
<thead>
<tr>
<th></th>
<th>Local Memory</th>
<th>Global Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>Write Hit</td>
<td>Write-back</td>
<td>Write-evict</td>
</tr>
<tr>
<td>Write Miss</td>
<td>Write no-allocate</td>
<td>Write no-allocate</td>
</tr>
</tbody>
</table>

- Non-coherent
- Single ported (128-Byte wide)
  - Takes multiple cycles to service non-coalesced accesses
Memory Access Tracking

- Cached access
  - Miss Status Holding Registers (MSHR)

- Non-cached access
  - Encode warp, target register in request packet
  - Memory Unit writes replied data directly to target request
Miss Status Holding Registers

- MSHRs keep track of outstanding memory requests
  - keep track of threads, target registers, request addresses

- GPGPU-Sim: Each cache has its set of MSHRs
- Each MSHR contains one or more memory requests to the same address
  - MSHRs are limited (configurable)
  - Memory unit stalls if cache runs out of MSHRS

- One approach that might make sense
  - No details available from NVIDIA / AMD
Atomic Operations

- Both CUDA and OpenCL support atomic operations
  - Read-modify-write on a single memory location
- Coalescing rules ~ global memory access
  - Put accesses to same memory location in separate transactions
- GPGPU-Sim simulate these as:
  - Load operations inside a SIMT core
    - Skips L1 data cache
  - Store operations at memory partition
SIMT Core Cluster

• Collection of SIMT cores
Clock domains

• Simulate independent clock domains for
  – SIMT cores
    • GT200: Set to \( \frac{1}{4} \) of *shader clock* to compensate for using SIMD width of 32 instead of 8
    • Fermi: Set to \( \frac{1}{2} \) of *shader clock* to compensate for using SIMD width of 32 instead of 16
  – Interconnection network
  – L2 cache (if enabled)
  – DRAM
    • This is real clock (command clock)
    • Effective clock is 2x this clock due to DDR
Clock Domain Crossing

• We simulate send and receive buffers at clock crossing boundaries
• The buffers are filled and drained in different clock domains
• E.g. consider the buffer from interconnect to memory partition
  – Filled at interconnect clock rate
  – Drained at DRAM clock rate
Interconnection Network Model

• Intersim (Booksim) a flit level simulator
  – Topologies (Mesh, Torus, Butterfly, …)
  – Routing (Dimension Order, Adaptive, etc.)
  – Flow Control (Virtual Channels, Credits)

• We simulate two separate networks
  – From SIMT cores to memory partitions
    • Read Requests, Write Requests
  – From memory partitions to SIMT cores
    • Read Replies, Write Acks
Topology Examples

(a) Crossbar

(b) Mesh

(c) Ring
Interconnection Network Config

• Booksim has its own config file
  – Topology (topology, k, n)
  – Virtual channels (num_vcs)
  – Buffers per VC (vc_buf_size)
  – Routing (routing_function)
  – Speedups (input_speedup, internal_speedup)
  – Allocators (vc_allocator, sw_allocator)

• Specific to GPGPU-sim
  – Channel Width (flit_size)
  – Setting memory partition locations (use_map)
Interconnect Injection Interfaces

![Interconnect Injection Interfaces Diagram]

- SIMT Core
- Router
- 1 Packet / Cycle
- 1 Flit / Cycle
- Core Clock Domain
- Interconnect Clock Domain
- Clock Boundary
Interconnect Injection Interfaces

- Memory Partition
- DRAM Clock Domain
- Router
- Interconnect Clock Domain
- Clock Boundary

1 Packet / Cycle → 1 Flit / Cycle
Interconnect Injection Interfaces

- L2 Cache
- Router
- L2 Clock Domain
- Interconnect Clock Domain
- Clock Boundary
- 1 Packet / Cycle
- 1 Flit / Cycle
Interconnect Ejection Interfaces

- 1 Ejection/boundary buffer per VC (1 flit / cycle)
- A credit is sent back to router as a flit goes from ejection to boundary buffer
GPU Microarchitecture Overview

GPU

SIMT Core Cluster
SIMT Core
SIMT Core

SIMT Core Cluster
SIMT Core
SIMT Core

SIMT Core Cluster
SIMT Core
SIMT Core

Interconnection Network

Memory Partition
Memory Partition

GDDR3/GDDR5
GDDR3/GDDR5

Off-chip DRAM

GDDR3/GDDR5
Memory Address Mapping

• Off-chip memory partitioned among several memory partitions
  – GT200 has 8 memory partitions
  – G80 and Fermi had 6 memory partitions
  – Each memory partition has a DRAM controller

• Successive 256-byte regions of memory are assigned to successive memory partitions
  – Address mapping is configurable in GPGPU-Sim
Mem. Address Mapping (Cont.)

Interconnection Network

SIMT Core

Memory Partition 0
Memory Partition 1
Memory Partition 2
Memory Partition 3
Memory Partition 4
Memory Partition 5
Memory Partition 6
Memory Partition 7
Memory Partition

- Service memory request (Load/Store/AtomicOp)
  - Contains L2 cache bank, DRAM timing model
  - Model Raster Operations Pipeline (ROP) latency
L2 Cache Bank

- GT200: Caches only texture
- Fermi: Caches All memory spaces
- Similar to L1 Data Cache

<table>
<thead>
<tr>
<th></th>
<th>Local Memory</th>
<th>Global Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>Write Hit</td>
<td>Write-back</td>
<td>Write-evict</td>
</tr>
<tr>
<td>Write Miss</td>
<td>Write no-allocate</td>
<td>Write no-allocate</td>
</tr>
</tbody>
</table>

- Missed requests are sent to DRAM
DRAM

- DRAM Memory
  - Off-chip, high-density and high capacity
- DRAM access time is **Not** constant
  - It has non-uniform access latencies
- That’s why we model it!
DRAM Access

• Row access
  – Activate a row or page of a DRAM bank
  – Load it to row buffer

• Column access
  – Select and return a block of data in row buffer

• Precharge
  – Write back the opened row into DRAM
  – Otherwise it will be lost!
DRAM Row Access Locality

$t_{RC} = \text{row cycle time}$

$t_{RP} = \text{row precharge time}$

$t_{RCD} = \text{row activate time}$
DRAM Bank-level Parallelism

- To increase DRAM performance and utilization
  - Multiple banks per DRAM chip
- To increase bus width
  - Multiple chips per Memory Controller
Scheduling DRAM Requests

- Scheduling policies supported
  - First in first out (FIFO)
    - In-order scheduling
  - First Ready First Come First Serve (FR-FCFS)
    - Out of order scheduling
    - Requires associative search
Session Summary

• Microarchitecture Timing Model in GPGPU-Sim
  – SIMT Core
  – Cache Model
  – Interconnection Network
  – Memory Partition + Address Mapping
  – DRAM Scheduling and Timing
# Overview

<table>
<thead>
<tr>
<th></th>
<th>Brief Background on GPU Computing</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>GPGPU-Sim Overview</td>
<td>30mins</td>
</tr>
<tr>
<td>3</td>
<td>Demo 1: Setup and Run</td>
<td>15mins</td>
</tr>
<tr>
<td></td>
<td>Coffee Break (10:00 – 10:30am)</td>
<td></td>
</tr>
<tr>
<td>4a</td>
<td>Microarchitecture Timing Model</td>
<td>60mins</td>
</tr>
<tr>
<td>4b</td>
<td>Software Organization</td>
<td>30mins</td>
</tr>
<tr>
<td></td>
<td>Lunch (12:00 – 1:30pm)</td>
<td></td>
</tr>
<tr>
<td>4b</td>
<td>Software Organization (Timing)</td>
<td>60mins</td>
</tr>
<tr>
<td>4c</td>
<td>The GPU Design Space</td>
<td>30mins</td>
</tr>
<tr>
<td>5a</td>
<td>Demo 2: Debugging Tool</td>
<td>15mins</td>
</tr>
<tr>
<td></td>
<td>Coffee Break (3:15 – 3:45pm)</td>
<td></td>
</tr>
<tr>
<td>5b</td>
<td>Demo 3: Visualizing Performance</td>
<td>30mins</td>
</tr>
<tr>
<td>6</td>
<td>Extending GPGPU-Sim: Walkthrough</td>
<td>40mins</td>
</tr>
<tr>
<td>7</td>
<td>Wrap Up and Discussion</td>
<td>15mins</td>
</tr>
</tbody>
</table>
## Overview

<table>
<thead>
<tr>
<th></th>
<th>Brief Background on GPU Computing</th>
<th>40mins</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>GPGPU-Sim Overview</td>
<td>30mins</td>
</tr>
<tr>
<td>3</td>
<td>Demo 1: Setup and Run</td>
<td>15mins</td>
</tr>
<tr>
<td></td>
<td><strong>Coffee Break (10:00 – 10:30am)</strong></td>
<td></td>
</tr>
<tr>
<td>4a</td>
<td>Microarchitecture Timing Model</td>
<td>60mins</td>
</tr>
<tr>
<td>4b</td>
<td>Software Organization</td>
<td>30mins</td>
</tr>
<tr>
<td></td>
<td><strong>Lunch (12:00 – 1:30pm)</strong></td>
<td></td>
</tr>
<tr>
<td>4b</td>
<td>Software Organization (Timing)</td>
<td>60mins</td>
</tr>
<tr>
<td>4c</td>
<td>The GPU Design Space</td>
<td>30mins</td>
</tr>
<tr>
<td>5a</td>
<td>Demo 2: Debugging Tool</td>
<td>15mins</td>
</tr>
<tr>
<td></td>
<td><strong>Coffee Break (3:15 – 3:45pm)</strong></td>
<td></td>
</tr>
<tr>
<td>5b</td>
<td>Demo 3: Visualizing Performance</td>
<td>30mins</td>
</tr>
<tr>
<td>6</td>
<td>Extending GPGPU-Sim: Walkthrough</td>
<td>40mins</td>
</tr>
<tr>
<td>7</td>
<td>Wrap Up and Discussion</td>
<td>15mins</td>
</tr>
</tbody>
</table>
Software Organization Overview

• Introduce GPGPU-Sim modules
• Functional Simulation
  – Interfacing with CUDA and OpenCL
  – Details of PTX simulation

Lunch

• Timing Model
Why should you learn about functional simulation?

• GPGPU-Sim already runs many of the existing GPU workloads:
  – ISPASS 2009
  – RODINIA
  – CUDA SDK

• We implement features on a demand basis:
  – When they are needed by an application that we are interested to run.
Why should you learn about functional simulation?

• As researchers, we are interested in enhancing GPUs for future workloads.
  – Some of them will not work on GPGPU-Sim out-of-the-box.

• At some point, you will need to extend the function simulator in GPGPU-Sim to run your research workloads.
Our Own Experience

• Thread Block Compaction (HPCA 2011)
  – Implemented low-level CUDA API for ray-tracing workload
  – Allow GPU to access CPU memory space directly for VisBench

• Hardware Transactional Memory for GPU Architectures (MICRO 2011)
  – Extended functional simulator to support transactions
Session Objectives

• After this session, you will be able to:
  – Extend GPGPU-Sim to get your workload running on it for your research
    • Implement new instructions
    • Update behavior for existing instructions
    • Fill in unimplemented parts of CUDA/OpenCL API
  – Evaluate your research microarchitectures on GPGPU-Sim
    • Implement your research idea in the timing model
Three Modules

- CUDA/OpenCL API library interface
- PTX instruction set emulator
- Timing model
Interface to CUDA/OpenCL

- API Interface
- How GPGPU-Sim starts
- PTX Loading / Parsing
- Stream Manager
Interface to CUDA/OpenCL

• CUDA and OpenCL applications:
  – Include code that runs on the host (CPU)
  – Use an API to communicate with GPU
  – API is defined in various header files
  – Implementation in DLL

• GPGPU-Sim:
  – Run host code on CPU
  – Functionally emulate API
  – I.e., if you run GPGPU-Sim on a Core 2 Duo machine, the host code runs natively on your Core 2 Duo processor.
Interface to CUDA/OpenCL API

- We implement versions of OpenCL / CUDA interface calls in a new DLL.
- Adjust LD_LIBRARY_PATH and CUDA/OpenCL application runs on simulator rather than GPU hardware.
- Given our research focus, we have implemented only what was required to get applications we were interested in running.
Interface to CUDA API

Example of interface code between host and simulator. This is the code that actually starts running the functional and timing models. Call to cudaLaunch is generated by nvcc from "<<<>>>" notation.

```c
__host__ cudaError_t CUDARTAPI cudaLaunch( const char *hostFun )
{
    CUctx_st* context = GPGPUSim_Context();
    char *mode = getenv("PTX_SIM_MODE_FUNC");
    if( mode )
        sscanf(mode,"%u", &g_ptx_sim_mode);
    gpgpusim_ptx_assert( !g_cuda_launch_stack.empty(), "empty launch stack" );
    kernel_config config = g_cuda_launch_stack.back();
    struct CUstream_st *stream = config.get_stream();
    printf("\nGPGPU-Sim PTX: cudaLaunch for 0x%p (mode=%s) on stream %u\n", ... );
    kernel_info_t *grid = gpgpu_cuda_ptx_sim_init_grid(hostFun,config.get_args(),
                                                     config.grid_dim(),
                                                     config.block_dim(),context);

    std::string kname = grid->name();
    dim3 gridDim = config.grid_dim();
    dim3 blockDim = config.block_dim();
    printf("GPGPU-Sim PTX: pushing kernel '%s' to stream %u, ... ");
    stream_operation op(grid,g_ptx_sim_mode,stream);
    g_stream_manager->push(op);
    g_cuda_launch_stack.pop_back();
    return g_last_cudaError = cudaSuccess;
}
```
GPGPU-Sim Startup Details

1. Some CUDA code called before main() during initialization of global variables.
   __cudaRegisterFunction, __cudaRegisterVar
   These provide information about device code (and how to call it).

2. First call to any CUDA API function triggers simulator initialization (gpgpu_ptx_sim_init_perf()).
   – Read environment variables (debug info, sim mode)
   – Parse option files
   – Initialize GPU uArch Model, Stream Manager

3. First call to __cudaRegisterFatBinary()
   – Load-parse PTX kernels, determine post-dominators
Loading PTX

• CUDA 3.1: Fatbin
  – Overload `__cudaRegisterFatBinary()` to extract PTX from executable

• CUDA 4.0 and later: cuobjdump
  – Call cuobjdump to obtain PTX/SASS
  – Executable must be compiled with CUDA 4.0 or later

• Multiple versions of kernel PTX
  – Default: Use PTX with highest SM version
PTX Parsing, Post-Dominator Detection

- GPGPU-Sim has a flex+bison parser to read in PTX
  - Default: Read PTX text embedded within binary

- Same parser is used for SASS (PTXPlus)

- Why flex+bison? Flexibility.
  - When NVIDIA update their PTX syntax, we can update our parser accordingly.

- Post-dominators
  - Handle warp divergence in simulation
  - Determined with standard control flow analysis
Parsing PTX

• Next two slides illustrate snippets of code from lexer and parser
cuda-sim/ptx.l – find tokens

```
abs TC; ptx_lval.int_value = ABS_OP; return OPCODE;
add TC; ptx_lval.int_value = ADD_OP; return OPCODE;
and TC; ptx_lval.int_value = AND_OP; return OPCODE;
...
.align TC; return ALIGN_DIRECTIVE;
.byte TC; return BYTE_DIRECTIVE;
.const[[0-9]+] TC; return CONST_DIRECTIVE;
...
"%tid" TC; ptx_lval.int_value = TID_ID; return SPECIAL_REGISTER;
...
.u32 TC; return U32_TYPE;
.u64 TC; return U64_TYPE;
.f16 TC; return F16_TYPE;
.f32 TC; return F32_TYPE;
...
equ TC; return EQU_OPTION;
.neu TC; return NEU_OPTION;
.ltu TC; return LTU_OPTION;
...
"}" TC; return RIGHT_SQUARE_BRACKET;
"<" TC; return LEFT_ANGLE_BRACKET;
">" TC; return RIGHT_ANGLE_BRACKET;
"(" TC; return LEFT_PAREN;
...```
cuda-sim/ptx.y – read instructions

...%token <string_value> STRING
%token <int_value> OPCODE
%token ALIGN_DIRECTIVE
%token BYTE_DIRECTIVE
...

%%

input: /* empty */
    | input directive_statement
    | input function_defn
    | input function_decl
    ;
...

instruction: opcode_spec LEFT_PAREN operand RIGHT_PAREN { set_return(); } COMMA operand COMMA LEFT_PAREN operand_list RIGHT_PAREN
    | opcode_spec operand COMMA LEFT_PAREN operand_list RIGHT_PAREN
    | opcode_spec operand_list
    | opcode_spec
    ;
...
Stream Manager + Abstract GPU

- **GPGPU-Sim Thread**
  - gpgpu_sim_thread_concurrent()

  ```
  do {
      Wait for streamOp;
      do {
          Obtain streamOp;
          Perform streamOp;
          gpu->cycle();
          active = gpu->active() or streamMgr->empty();
          if (gpu->finished_kernel())
              gpu->print_stats();
      } while (active);
  } while (!gpu_done);
  ```
Functional Simulator

• Key Objects in Module
• Three main aspects
  – How are threads simulated?
  – How are instructions simulated?
  – How are values communicated between threads?
• Memory Space Buffer
• Pure Functional Simulator
Inside Functional Simulation: Key Objects

- **gpgpu_t**
  - **kernel_info**
    - Grid/Block Dim
    - Launch Status
    - Param Memory
  - **ptx_cta_info**
    - Barrier
    - Shared Memory
  - **ptx_thread_info**
    - Thread IDs
    - Registers
    - Program Counters
    - Call stack
    - Local Memory

- **function_info**
  - PC(0x0)
    - ptx_instruction
  - PC(0x8)
    - ptx_instruction
  - PC(0x10)
    - ptx_instruction
    - Symbol Table
    - Control Flow Analysis

- **core_t**
  - SIMT Stacks
  - Global Memory
  - GPU Memory Management
How are threads simulated (functionally)?

Thread = program counter +
set of registers +
set of local memory locations

CTA (block) = set of threads with access to
a shared memory + barrier

New in GPGPU-Sim 3.x: Expose notion of
“warp” in functional simulator

• Support undocumented barrier behavior
How are instructions simulated (functionally)?

- Code for simulating instructions in cuda-sim/instruction.cc
  - 1:1 mapping opcode → implementation function
  - Mapping in opcode.def

- Threads initialized during launching of blocks.
- Functional execution at issue stage of timing pipeline:
  - Calls ptx_thread_info::ptx_exec_inst()
  - Inside: “giant switch statement” simulation approach (emulation)
- Lookup instruction “object” corresponding to program counter
  - PTX: Assume 8-byte per instruction
How are values communicated between threads?

- Threads can communicate through
  - Global memory
  - Shared memory

- We simulate all instructions as they reach the issue stage of pipeline
  - This includes loads and stores that access memory.
  - Except for atomics: We simulate them functionally once atomic operation exits the memory partition in timing model.
    - `atom_impl()` calculates effective memory address and setups callback function

- Most CUDA code avoids intra-kernel communication through global memory
  - Relaxed memory ordering
Memory Space Buffer

- Functionally implements various memory spaces
  - Map of memory address → pages
  - Each memory space buffer may contain infinite # pages
- Global, texture, constant memory spaces share the same buffer
- Each block has a shared memory buffer
- Each thread has a local memory buffer

- Implementation in cuda-sim/memory.[h,cc]
Pure Functional Simulator

• Great for:
  – Profiling application behavior
  – Prototyping new instructions
• Recreated in GPGPU-Sim v3.1.0
• Mostly contained in a single class: `functionalCoreSim`
• Execute one thread block at a time
  – Execute warps in round-robin
<table>
<thead>
<tr>
<th></th>
<th>Overview</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Brief Background on GPU Computing</td>
<td>40mins</td>
</tr>
<tr>
<td>2</td>
<td>GPGPU-Sim Overview</td>
<td>30mins</td>
</tr>
<tr>
<td>3</td>
<td>Demo 1: Setup and Run</td>
<td>15mins</td>
</tr>
<tr>
<td></td>
<td><strong>Coffee Break (10:00 – 10:30am)</strong></td>
<td></td>
</tr>
<tr>
<td>4a</td>
<td>Microarchitecture Timing Model</td>
<td>60mins</td>
</tr>
<tr>
<td>4b</td>
<td>Software Organization</td>
<td>30mins</td>
</tr>
<tr>
<td></td>
<td><strong>Lunch (12:00 – 1:30pm)</strong></td>
<td></td>
</tr>
<tr>
<td>4b</td>
<td>Software Organization (Timing)</td>
<td>60mins</td>
</tr>
<tr>
<td>4c</td>
<td>The GPU Design Space</td>
<td>30mins</td>
</tr>
<tr>
<td>5a</td>
<td>Demo 2: Debugging Tool</td>
<td>15mins</td>
</tr>
<tr>
<td></td>
<td><strong>Coffee Break (3:15 – 3:45pm)</strong></td>
<td></td>
</tr>
<tr>
<td>5b</td>
<td>Demo 3: Visualizing Performance</td>
<td>30mins</td>
</tr>
<tr>
<td>6</td>
<td>Extending GPGPU-Sim: Walkthrough</td>
<td>40mins</td>
</tr>
<tr>
<td>7</td>
<td>Wrap Up and Discussion</td>
<td>15mins</td>
</tr>
</tbody>
</table>
Software Organization Overview

- Introduce GPGPU-Sim modules
- Interfacing with CUDA and OpenCL
- Details of PTX simulation
- **Timing Model**
Abstract HW Model

- Interface between functional simulation and timing simulation

Abstract Model

- `gpgpu_t`
- `core_t`
- `ptx_instruction_t`

Timing Model

- `gpgpu_sim`
- `shader_core_ctx`
- `warp_inst_t`
Microarchitecture Model (Revisited)

GPU

SIMT Core Cluster
- SIMT Core
- SIMT Core

SIMT Core Cluster
- SIMT Core
- SIMT Core

SIMT Core Cluster
- SIMT Core
- SIMT Core

Interconnection Network

Memory Partition

GDDR3/GDDR5

Off-chip DRAM

GDDR3/GDDR5
Timing Model (Software Overview)
Timing Model: gpgpu_sim

• Top Level Object
• A virtual GPU with timing model
  – Contains all components for timing simulation
• Specific responsibilities:
  – Initialization
  – Kernel Launch
  – Clock Domain
  – Main Simulation Loop
• Implemented in gpgpu-sim/gpu-sim.[h,cc]
Timing Model: Initialization

- \texttt{gpgpu\_sim::gpgpu\_sim()}
  - Allocate and initialize microarchitecture model and statistic collection structures
  - Called in \texttt{gpgpu\_ptx\_sim\_init\_perf()}
    - At the first CUDA API call
  - Only one global instance: \texttt{g\_the\_gpu}
do {
    Wait for streamOp;
    do {
        Obtain streamOp;
        if (streamOp == Launch)
            gpu->launch();
            gpu->cycle();
        active = gpu->active() or
            streamMgr->empty();
        if (gpu->finished_kernel())
            gpu->print_stats();
    } while (active);
} while (!gpu_done);

General Stats Log

Push a kernel into GPU launch queue
Simulate a cycle in GPU
Probe GPU Activity:
  - Running a kernel
  - Accessing memory
Timing Model: Main Simulation Loop

- Inside `gpgpu_sim::cycle()`:
  - Check all clock domains
    - if `(clock_mask & CORE) { ... }
  - Execute the ones that are ready
  - Currently, 4 domains:
    - CORE – SIMT Core + Thread Block Issue
    - ICNT – Interconnection Network
    - DRAM – DRAM + Request Scheduler
    - L2 – L2 Cache in Memory Partition
Timing Model: Thread Block Issue

• `gpgpu_sim::issue_block2core()`: 
  – Issue new thread blocks to core
    • Limit calculated with `shader_core_config::max_cta()`
  – Initializes the threads inside the new blocks
    • Calls interface to functional model: `ptx_sim_init_thread()`
  – Hierarchical Block Distribution
    • Top Level → SIMT Core Clusters → SIMT Cores
Timing Model: simt_core_cluster

- Collection of SIMT cores
  - core_cycle(): simulation a cycle in each core
- Interface them with interconnect

```
incnt_cycle()
incnt_inject_request_packet()
```

```
shader_memory_interface
```

June 2012
GPGPU-Sim Tutorial (ISCA 2012)
4b: Software Organization
Timing Model: SIMT Core

• Class `shader_core_ctx` in gpgpu-sim/shader.[h,cc]
  – Derived from `core_t`
  – Contains the state of a SIMT Core

• Major components in separate classes
  – `scheduler_unit`
  – `scoreboard`
  – `opndcoll_rfu_t` (Operand Collector)
  – `simd_function_unit`
  – `ldst_unit` (Memory Unit)
Timing Model: SIMT Core

- `shader_core_ctx::cycle()`
Timing Model: Pipeline Connection

shd_warp_t
- Timing state of a warp
- I-Buffer Entries for warp

warp_inst_t

core_t
  SIMT Stack

ptx_thread_info

Fetch
  Branch Target PC

I-Cache

i_fetch_buffer_t

Decode

I-Buffer

Valid[1:N]

Score Board

SIMT-Stack

Issue

Pred.

Operands Collector

ALU

MEM

Done (WID)
Timing Model: Warp Instruction

warp_inst_t
- A dynamic “SIMD” instruction executed by a warp
- Pipeline Register

• Implements access coalescing logic
  - Groups individual accesses from threads into wider/bank-conflict-free accesses
  - Per-thread info → mem_access_t
  - See warp_inst_t::generate_mem_accesses()
Timing Model: Memory Access

**mem_access_t**
- Coalesced memory access from a warp
- Consumed in ldst_unit

**mem_fetch**
- Memory request structure that is passed to the cache model and modules in the memory sub-system.
- Has a copy of the warp_inst_t + mem_access_t that created the request
  - No need for tracker inside SIMT Core
  - Access to atom_callback via warp_inst_t
- Generated in ldst_unit::cycle()
- Destroyed in ldst_unit::writeback() for reads and inside memory_partition_unit for writes
Timing Model: Fetch Stage

1. Checks for warps that completed execution
   - Release resource

2. Selected a warp via m_last_warp_fetch and access I-cache
   - Hit: Send to decode stage
   - Miss: Send instruction fetch request off core, try again after I-cache is filled
Timing Model: Decode Stage

• Obtain instruction (warp_inst_t) from functional simulator
  – Calls ptx_fetch_inst(pc)

• Push into I-Buffer of the corresponding warp
  – Calls shd_warp_t::ibuffer_fill()
  – Can push up to 2 instructions per-cycle
Timing Model: Issue Stage

- Calls `scheduler_unit::cycle()` for every instance of `scheduler_unit`
- Calls `shader_core_ctx::issue_warp()` for each warp selected by the scheduler unit
  - Perform functional execution:
    - Calls `func_exec_inst() \rightarrow execute_warp_inst_t()`
  - Coalesce memory accesses and push them into `m_access_q` in `warp_inst_t`:
    - Calls `warp_inst_t::generate_mem_accesses()`
  - Update SIMT Stack for warp
  - Lock output register(s) in scoreboard
  - Send instruction to operand collector
Timing Model: Scheduler Unit

`scheduler_unit` in `gpgpu-sim/shader.[h,cc]`
- `add_supervised_warp_id()`
- `cycle()`

- Scheduler for a subset of warps
  - Model dual scheduler in Fermi

- Warp can be issued if:
  - Valid instruction in I-Buffer
  - Instruction does not read/write locked register (Scoreboard)
  - Execution unit is available
Timing Model: Scoreboard

**Scoreboard** in gpgpu-sim/scoreboard.[h,cc]

- reserveRegister(): Lock register for instruction
- releaseRegister(): Unlock register
- checkCollision(): Check if instruction accesses any locked register
Timing Model: SIMT Stack

**simt_stack** in abstraction_model.[h,cc]
- launch()
- get_active_mask()
- update()

Interaction with shader_core_ctx only in issue():
- In scheduler_unit::cycle()
  - Calls get_active_mask()
- In shader_core_ctx::issue_warp() \rightarrow
  core_t::updateSIMTStack()
  - Calls update() after functional execution
Timing Model: Register Read Stage

Scheduler Units

m_dispatch_port [ID_OC_SP]
m_dispatch_port [ID_OC_SFU]
m_dispatch_port [ID_OC_MEM]

Operand Collector
(opndcoll_rfut m_operand_collector)

m_issue_port [OC_EX_SP]
m_issue_port [OC_EX_SFU]
m_issue_port [OC_EX_MEM]

m_sp_unit
m_sfu_unit
m_ldst_unit
Timing Model: Operand Collector

`opndcoll_rfu_t` in gpgpu-sim/shader.[h,cc]

- Set of subclasses

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>arbiter_t</td>
<td>Arbitrate accesses to the register file banks</td>
</tr>
<tr>
<td>op_t</td>
<td>A read/write to a single register</td>
</tr>
<tr>
<td>allocation_t</td>
<td>Link <code>op_t</code> to an allocated access in <code>arbiter_t</code></td>
</tr>
<tr>
<td>input_port_t</td>
<td>Input to operand collector</td>
</tr>
<tr>
<td>collector_unit_t</td>
<td>Collector Unit</td>
</tr>
<tr>
<td></td>
<td>- One for each warp instruction</td>
</tr>
<tr>
<td>dispatch_unit_t</td>
<td>Dispatch a warp instruction with all operands fetched to the corresponding unit</td>
</tr>
</tbody>
</table>
Timing Model: Operand Collector

\texttt{opndcoll\_rfu\_t::step()} \leftarrow \text{Cycle function}

1. For each input port,\texttt{allocate\_cu()}
   \begin{itemize}
   \item Allocate a free collector unit for the warp instruction
   \item Push its register reads into queues in the arbiter unit
   \end{itemize}

2. \texttt{allocate\_reads()}
   \begin{itemize}
   \item Process reads that have no bank conflict
   \end{itemize}

3. \texttt{dispatch\_ready\_cu()}
   \begin{itemize}
   \item Each dispatch unit selects a collector unit with fetched operands
   \item Sends its instruction to output port
   \end{itemize}

\texttt{opndcoll\_rfu\_t::writeback()}
\begin{itemize}
\item Allocate bank for writes to registers
   (has priority over reads)
\end{itemize}
Timing Model: Execution Stage

• Calls cycle() for every functional unit
  – ALU Units (SP, SFU)
  – Memory Unit

• Implements a result bus reservation system for groups of ALU units that shares a common writeback bus
  – Prevent stalling inside the units
Timing Model: ALU Pipeline

pipelined_simd_unit

- SP unit and SFU unit
- Instruction-dependent BW and Latency
Timing Model: Memory Unit

ldst_unit

- Instantiates and operates on all in-core memories
  - Texture cache: m_L1T
  - Constant cache: m_L1C
  - Data cache: m_L1D
  - Shared memory: m_pipeline_reg
- Off-core interface: m_icnt
- Mem_fetch allocator: m_mf_allocator
- Operates at a clock_multiplier() rate
  - Model half-warps
Timing Model: Memory Unit

**ldst_unit::cycle()**
- Process memory responses from m_response_fifo
- Service instruction at m_dispatch_reg
  - One access (mem_access_t) / cycle
  - Stall until every access in instruction’s m_access_q is processed
  - shared_cycle()
  - constant_cycle()
  - texture_cycle()
  - memory_cycle()

**ldst_unit::writeback()**
- Writeback for one of the component (client)
  - Optional: Fill caches
  - Writeback data to register via operand collector
  - Signal scoreboard to unlock register(s)
Timing Model: Cache Models

See gpgpu-sim/gpu-cache.[h,cc]

Common Components:

- **tag_array**: Cache Line States, Replacement
- **mshr_table**: Queues of Pending Requests

No data array – Data is stored in functional simulator

Three Models:

- **read_only_cache**: Read accesses only
- **data_cache**: Allow read and write accesses
- **texture_cache**: Read accesses in FIFO
- Each instance has a reference to a memory interface (m_memport) for cache requests/writebacks
Timing Model: Cache Models

Common interface:

• **access(mem_fetch *mf)**
  – Probe the tag_array (no state change)
  – **Hit**: Access the tag_array (Modifies LRU stack, line state)
  – **Miss**: Try to allocate resource to handle miss
    • A cache line for replacement
    • Entry in mshr_table
    • Entry in m_miss_queue
  – Fail to allocate → Return **Resource Failure**

• **cycle()**
  – Move mem_fetch from m_miss_queue to m_memport

• **fill()**
  – Update cache block status
  – Mark entry in mshr_table as ready
Timing Model: Interfaces

• SIMT Core
  – **mem_fetch_interface**: Generic interface used by cache models
    • `full(unsigned int size, bool write)`
    • `push(mem_fetch *mf)`
  – **shader_memory_interface**: To SIMT Core Cluster/Interconnection Network
  – **perfect_memory_interface**: Direct loop back to model zero latency + infinite bandwidth
Timing Model: Interfaces

- **Interconnection Network:**
  - `icnt_has_buffer()`: Check for input buffer space
  - `icnt_push()`: Push packet into network
  - `icnt_pop()`: Pop packet from network
  - `icnt_transfer()`: Run network for a cycle
Timing Model: Interfaces

• Memory Partition (L2 + DRAM):
  – full(): Queues in memory partition full?
  – push(): Push request into memory partition
  – pop(): Obtain info for completed request
  – top(): Pop completed request

Flow Control

```c
mem_req = m_mem_partition[x]->top();
if (icnt_has_buffer(mem_req.info)) {
    icnt_push(mem_req);
    m_mem_partition[x]->pop();
}
```
Timing Model: Memory Partition

memory_partition_unit in gpgpu_sim/l2cache.[h,cc]
Timing Model: Memory Partition

```
cache_cycle()

full()
push()

top()
pop()

m_L2_icnt_queue  m_dram_L2_queue

Atomic
Operation
Execution

m_L2cache

m_icnt_L2_queue  m_L2_dram_queue

m_dram

m_dram_latency_queue

Off-Chip
DRAM
Channel

m_dram
```
Timing Model: Memory Partition

- L2 Cache = data_cache
  - The same model for L1
- Atomic operation executed in pop()
  - Calls mem_fetch::do_atomic()
  - Calls atom_callback() for each thread
- DRAM Access Schedule + Timing = dram_t
Timing Model: 
Memory Partition

Flow Control:

- Queues = `fifo_pipeline`
  - Limited Capacity: Model flow control (bandwidth)
- ROP Queue and DRAM Latency Queue
  - Model empty pipe latency
  - At Push:
    Obtain `ready_cycle = current_cycle + min_latency`
  - At Pop: Wait until `current_cycle == ready_cycle`
Timing Model: DRAM Model

DRAM Scheduler:

\texttt{frfcfs\_scheduler} in \texttt{dram\_sche.[h/cc]}

- Models a First-Ready-First-Come-First-Serve Access Scheduler (Rixner et al.)

DRAM Timing Model:

\texttt{dram\_t} in \texttt{dram.[h/cc]}

- Models the DRAM access timing
  - Detail GDDR3/GDDR5 spec
- Contains a FIFO scheduler
Timing Model: DRAM Model

Modeling Detail GDDR3/GDDR5 Timing:

• Set of Constraints → Counters
• Each action has a set of constraints
  – Delay action until all constraints met
    (i.e. all corresponding counters == 0)
  – Action creates new constraints for other actions
    (i.e. reset counters to timing parameter)
• All counters are decremented every cycle
Timing Model: DRAM Model

Inside dram_t::cycle()

... if (RRDc == 0 and
    bank[j]->state == BANK_IDLE and
    bank[j]->RPCc == 0 and
    bank[j]->RCc == 0)
{
    Activate Row for bank[j];
    RRDc = m_config->tRRD;
    bank[j]->RCDc = m_config->tRCD;
    bank[j]->RASc = m_config->tRAS;
    bank[j]->RCc = m_config->tRC;
}

... if (RRDc > 0) RRDc--;
if (bank[j]->RCDc > 0) bank[j]->RCDc--;
...
Configuration Organization

- Each major module has its own configuration structure
  - Hooked up to the global option parser
Adding Configuration Options

• Use `option_parser` module
  – Automatically parses options to linked variables
  – `option_parser_register(opp, ...)`
    link options to variables

• See `gpgpu_sim_config::reg_options()` in `gpu-sim.cc` for examples
Session Summary

• Software Organization of GPGPU-Sim
  – Interface to CUDA/OpenCL Application
  – PTX Functional Simulation
  – Timing Model
# Overview

<table>
<thead>
<tr>
<th></th>
<th>Outline</th>
<th>Duration</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Brief Background on GPU Computing</td>
<td>40mins</td>
</tr>
<tr>
<td>2</td>
<td>GPGPU-Sim Overview</td>
<td>30mins</td>
</tr>
<tr>
<td>3</td>
<td>Demo 1: Setup and Run</td>
<td>15mins</td>
</tr>
<tr>
<td></td>
<td><strong>Coffee Break (10:00 – 10:30am)</strong></td>
<td></td>
</tr>
<tr>
<td>4a</td>
<td>Microarchitecture Timing Model</td>
<td>60mins</td>
</tr>
<tr>
<td>4b</td>
<td>Software Organization</td>
<td>30mins</td>
</tr>
<tr>
<td></td>
<td><strong>Lunch (12:00 – 1:30pm)</strong></td>
<td></td>
</tr>
<tr>
<td>4b</td>
<td>Software Organization (Timing)</td>
<td>60mins</td>
</tr>
<tr>
<td>4c</td>
<td>The GPU Design Space</td>
<td>30mins</td>
</tr>
<tr>
<td>5a</td>
<td>Demo 2: Debugging Tool</td>
<td>15mins</td>
</tr>
<tr>
<td></td>
<td><strong>Coffee Break (3:15 – 3:45pm)</strong></td>
<td></td>
</tr>
<tr>
<td>5b</td>
<td>Demo 3: Visualizing Performance</td>
<td>30mins</td>
</tr>
<tr>
<td>6</td>
<td>Extending GPGPU-Sim: Walkthrough</td>
<td>40mins</td>
</tr>
<tr>
<td>7</td>
<td>Wrap Up and Discussion</td>
<td>15mins</td>
</tr>
</tbody>
</table>
## Overview

<table>
<thead>
<tr>
<th></th>
<th>Title</th>
<th>Duration</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Brief Background on GPU Computing</td>
<td>40mins</td>
</tr>
<tr>
<td>2</td>
<td>GPGPU-Sim Overview</td>
<td>30mins</td>
</tr>
<tr>
<td>3</td>
<td>Demo 1: Setup and Run</td>
<td>15mins</td>
</tr>
<tr>
<td></td>
<td>Coffee Break (10:00 – 10:30am)</td>
<td></td>
</tr>
<tr>
<td>4a</td>
<td>Microarchitecture Timing Model</td>
<td>60mins</td>
</tr>
<tr>
<td>4b</td>
<td>Software Organization</td>
<td>30mins</td>
</tr>
<tr>
<td></td>
<td>Lunch (12:00 – 1:30pm)</td>
<td></td>
</tr>
<tr>
<td>4b</td>
<td>Software Organization (Timing)</td>
<td>60mins</td>
</tr>
<tr>
<td>4c</td>
<td>The GPU Design Space</td>
<td>30mins</td>
</tr>
<tr>
<td>5a</td>
<td>Demo 2: Debugging Tool</td>
<td>15mins</td>
</tr>
<tr>
<td></td>
<td>Coffee Break (3:15 – 3:45pm)</td>
<td></td>
</tr>
<tr>
<td>5b</td>
<td>Demo 3: Visualizing Performance</td>
<td>30mins</td>
</tr>
<tr>
<td>6</td>
<td>Extending GPGPU-Sim: Walkthrough</td>
<td>40mins</td>
</tr>
<tr>
<td>7</td>
<td>Wrap Up and Discussion</td>
<td>15mins</td>
</tr>
</tbody>
</table>
Session Objectives

• After this session, you will be able to
  – Setup simulation configuration for GPGPU-Sim
  – Locate the provided configuration files
  – Organize the options inside the configuration files into categories
GPU Design Space

• GPU architectures evolve:
  – Different Vendors (Nvidia, AMD, Intel… etc.)
  – Generations (G80, GT200, Fermi…)

• GPGPU-Sim comes with 70+ options to configure the timing model (and growing)

• We provide the following configurations (in `<GPGPUSIM_ROOT>/configs/`):
  – QuadroFX5800
  – Fermi (Geforce GTX480)
Simulation Configurations

• GPGPU-Sim always loads two configuration files from the working directory:
  – gpgpusim.config
  – An interconnection configuration file
Example Config: Fermi
Simulation Setup

# functional simulator specification
-gpgpu_ptx_sim_mode 0
-gpgpu_ptx_force_max_capability 20

# SASS execution (only supported with CUDA >= 4.0)
-gpgpu_ptx_convert_to_ptxplus 0
-gpgpu_ptx_save_converted_ptxplus 0

# stat collection
-gpgpu_ptx_instruction_classification 0
-gpgpu_memlatency_stat 14
-gpgpu_runtime_stat 500
-enable_ptx_file_line_stats 1
-visualizer_enabled 0
Example Config: Fermi
High Level Architecture

# high level architecture configuration
-gpgpu_n_clusters 15
-gpgpu_n_cores_per_cluster 1
-gpgpu_n_mem 6

# Fermi clock domains
#-gpgpu_clock_domains <Core Clock>:<Interconnect Clock>:<L2 Clock>:<DRAM Clock>
# In Fermi, each pipeline has 16 execution units, so the Core clock needs to be divided
# by 2. (GPGPU-Sim simulates a warp (32 threads) in a single cycle). 1400/2 = 700
-gpgpu_clock_domains 700.0:1400.0:700.0:1848.0
Example Config: Fermi
SIMT Core Pipeline + Scheduler

# shader core pipeline config (32K registers per core)
-gpgpu_shader_registers 32768

# This implies a maximum of 48 warps/SM
-gpgpu_shader_core_pipeline 1536:32
-gpgpu_shader_cta 8
-gpgpu_simd_model 1

# Fermi has two schedulers per core
-gpgpu_num_sched_per_core 2
-gpgpu_max_insn_issue_per_warp 1

# Pipeline widths and number of FUs
# ID_OC_SP,ID_OC_SFU,ID_OC_MEM,OC_EX_SP,OC_EX_SFU,OC_EX_MEM,EX_WB
-gpgpu_pipeline_widths 2,1,1,2,1,1,2
-gpgpu_num_sp_units 2
-gpgpu_num_sfu_units 1
Example Config: Fermi
Operand Collector + Execution

# operand collector
-gpgpu_num_reg_banks 16
-gpgpu_operand_collector_num_units_sp 6
-gpgpu_operand_collector_num_units_sfu 8
-gpgpu_operand_collector_num_in_ports_sp 2
-gpgpu_operand_collector_num_out_ports_sp 2

# Instruction latencies and initiation intervals
# "ADD,MAX,MUL,MAD,DIV"
- ptx_opcode_latency_int 4,13,4,5,145
- ptx_opcode_initiation_int 1,2,2,1,8
- ptx_opcode_latency_fp 4,13,4,5,39
- ptx_opcode_initiation_fp 1,2,1,1,4
- ptx_opcode_latency_dp 8,19,8,8,330
- ptx_opcode_initiation_dp 8,16,8,8,130
Example Config: Fermi
Memory Unit

# In Fermi, the cache and shared memory can be configured to 16kb:48kb(default) or 48kb:16kb
-gpgpu_cache:dl1 32:128:4:L:R:m,A:32:8,8
-gpgpu_shmem_size 49152

# The alternative configuration for fermi in case cudaFuncCachePreferL1 is selected
{-#gpgpu_cache:dl1 64:128:6:L:R:m,A:32:8,8
{-#gpgpu_shmem_size 16384

-gpgpu_const_cache:l1 64:64:2:L:R:f,A:2:32,4
-gpgpu_shmem_warp_parts 1
Example Config: Fermi Interconnect + Memory Partition

# interconnection
-network_mode 1
-inter_config_file icnt_config_fermi_islip.txt

# memory partition latency config
-rop_latency 120
-dram_latency 100

# 64 sets, each 256 bytes 8-way for each memory partition. This gives 786KB L2 cache
-gpgpu_cache:dl2 64:256:8:L:R:m,A:32:4,4
-gpgpu_cache:dl2_texture_only 0
Example Config: Fermi DRAM

# dram model config
-gpgpu_dram_scheduler 1
-gpgpu_dram_sched_queue_size 16

# for Fermi, bus width is 384bits, this is 8 bytes (4 bytes at each DRAM chip) per memory partition
-gpgpu_n_mem_per_ctrlr 2
-gpgpu_dram_buswidth 4
-gpgpu_dram_burst_length 4
-gpgpu_mem_address_mask 1
-gpgpu_mem_addr_mapping
  dramid@8;00000000.00000000.00000000.00000000.0000RRRR.RRRRRRRR.RRBBBCCC.
  CCCSSSSS

# GDDR5 timing from hynix H5GQ1H24AFR
# to disable bank groups, set nbkgrp to 1 and tCCDL and tRTPL to 0
Configuration Categories (Recap)

• Simulation Setup
• High Level Architecture
• SIMT Core Pipeline + Scheduler
• Operand Collector + Execution
• Memory Unit
• Interconnect
• Memory Partition
• DRAM
Session Summary

• GPGPU-Sim simulation configurations
  – 70+ Options
  – Organized in categories

• More details in manual:
  http://gpgpu-sim.org/manual/
## Overview

<table>
<thead>
<tr>
<th></th>
<th>Content</th>
<th>Duration</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Brief Background on GPU Computing</td>
<td>40mins</td>
</tr>
<tr>
<td>2</td>
<td>GPGPU-Sim Overview</td>
<td>30mins</td>
</tr>
<tr>
<td>3</td>
<td>Demo 1: Setup and Run</td>
<td>15mins</td>
</tr>
<tr>
<td></td>
<td><strong>Coffee Break (10:00 – 10:30am)</strong></td>
<td></td>
</tr>
<tr>
<td>4a</td>
<td>Microarchitecture Timing Model</td>
<td>60mins</td>
</tr>
<tr>
<td>4b</td>
<td>Software Organization</td>
<td>30mins</td>
</tr>
<tr>
<td></td>
<td><strong>Lunch (12:00 – 1:30pm)</strong></td>
<td></td>
</tr>
<tr>
<td>4b</td>
<td>Software Organization (Timing)</td>
<td>60mins</td>
</tr>
<tr>
<td>4c</td>
<td>The GPU Design Space</td>
<td>30mins</td>
</tr>
<tr>
<td>5a</td>
<td>Demo 2: Debugging Tool</td>
<td>15mins</td>
</tr>
<tr>
<td></td>
<td><strong>Coffee Break (3:15 – 3:45pm)</strong></td>
<td></td>
</tr>
<tr>
<td>5b</td>
<td>Demo 3: Visualizing Performance</td>
<td>30mins</td>
</tr>
<tr>
<td>6</td>
<td>Extending GPGPU-Sim: Walkthrough</td>
<td>40mins</td>
</tr>
<tr>
<td>7</td>
<td>Wrap Up and Discussion</td>
<td>15mins</td>
</tr>
</tbody>
</table>
## Overview

<table>
<thead>
<tr>
<th></th>
<th>Content</th>
<th>Duration</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>1</strong></td>
<td>Brief Background on GPU Computing</td>
<td>40mins</td>
</tr>
<tr>
<td><strong>2</strong></td>
<td>GPGPU-Sim Overview</td>
<td>30mins</td>
</tr>
<tr>
<td><strong>3</strong></td>
<td>Demo 1: Setup and Run</td>
<td>15mins</td>
</tr>
<tr>
<td></td>
<td><strong>Coffee Break (10:00 – 10:30am)</strong></td>
<td></td>
</tr>
<tr>
<td><strong>4a</strong></td>
<td>Microarchitecture Timing Model</td>
<td>60mins</td>
</tr>
<tr>
<td><strong>4b</strong></td>
<td>Software Organization</td>
<td>30mins</td>
</tr>
<tr>
<td></td>
<td><strong>Lunch (12:00 – 1:30pm)</strong></td>
<td></td>
</tr>
<tr>
<td><strong>4b</strong></td>
<td>Software Organization (Timing)</td>
<td>60mins</td>
</tr>
<tr>
<td><strong>4c</strong></td>
<td>The GPU Design Space</td>
<td>30mins</td>
</tr>
<tr>
<td><strong>5a</strong></td>
<td>Demo 2: Debugging Tool</td>
<td>15mins</td>
</tr>
<tr>
<td></td>
<td><strong>Coffee Break (3:15 – 3:45pm)</strong></td>
<td></td>
</tr>
<tr>
<td><strong>5b</strong></td>
<td>Demo 3: Visualizing Performance</td>
<td>30mins</td>
</tr>
<tr>
<td><strong>6</strong></td>
<td>Extending GPGPU-Sim: Walkthrough</td>
<td>40mins</td>
</tr>
<tr>
<td><strong>7</strong></td>
<td>Wrap Up and Discussion</td>
<td>15mins</td>
</tr>
</tbody>
</table>
Session Overview

• GDB Macro
• Debug Levels
• Interactive Debugger
• AerialVision
Session Objectives

• After this session, you will be able to:
  – Use GDB to observe the simulated microarchitecture timing states in GPGPU-Sim
  – Tell GPGPU-Sim to dump out extra debugging info
  – Visualize high-level performance dynamics in CUDA applications
GDB Macros

• We provide a set of GDB macros that visualize detail microarchitecture states:

<table>
<thead>
<tr>
<th>Macro</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dp &lt;core_id&gt;</td>
<td>Display Pipeline State in the SIMT Core with ID = &lt;core_id&gt;</td>
</tr>
<tr>
<td>dm &lt;partition_id&gt;</td>
<td>Display Internal States in Memory Partition with ID = &lt;partition_id&gt;</td>
</tr>
<tr>
<td>ptxdis &lt;start_pc&gt; &lt;end_pc&gt;</td>
<td>Display PTX Instruction Between &lt;start_pc&gt; and &lt;end_pc&gt;</td>
</tr>
</tbody>
</table>

• Useful for cycle-by-cycle debugging
• See Manual for other macros
GDB Macros: Usage

• Copy .gdbinit in GPGPU-Sim distribution to:
  – Your home directory or
  – Where GDB is launched

• GDB loads the macros automatically and display this message:

  ** loading GPGPU-Sim debugging macros... **
GDB Macros: DP

**Front End:**
- I-Buffer
- SIMT Stack
- Scoreboard

**Data Path:**
- Operand Collector
- ALU Units

**Memory Unit:**
- Caches
- Pending Requests
Debug Levels

• Tell GPGPU-Sim to dump out extra information for debugging
• Set via env. var. PTX_SIM_DEBUG
• Most useful levels (see Manual for others):

<table>
<thead>
<tr>
<th>Level</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>Display modified register(s) by each executed instruction</td>
</tr>
<tr>
<td>10</td>
<td>Display all registers of a thread executing the instruction</td>
</tr>
</tbody>
</table>
Debug Levels

- **PTX_SIM_DEBUG_THREAD_UID**
  - Limit debug output to a single thread
  - Useful for tracing a single thread
Interactive Debugger

• GDB-like interface
  – Complements debug level

• Features
  – Breakpoints for a specific thread
  – Watchpoints for address
  – Single stepping execution to next core cycle

• To enable: Set GPGPUSIM_DEBUG to 1
  – Currently limited to performance simulation
GDB Macros Demo
<table>
<thead>
<tr>
<th></th>
<th>Activity</th>
<th>Duration</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Brief Background on GPU Computing</td>
<td>40 mins</td>
</tr>
<tr>
<td>2</td>
<td>GPGPU-Sim Overview</td>
<td>30 mins</td>
</tr>
<tr>
<td>3</td>
<td>Demo 1: Setup and Run</td>
<td>15 mins</td>
</tr>
<tr>
<td></td>
<td><strong>Coffee Break (10:00 – 10:30am)</strong></td>
<td></td>
</tr>
<tr>
<td>4a</td>
<td>Microarchitecture Timing Model</td>
<td>60 mins</td>
</tr>
<tr>
<td>4b</td>
<td>Software Organization</td>
<td>30 mins</td>
</tr>
<tr>
<td></td>
<td><strong>Lunch (12:00 – 1:30pm)</strong></td>
<td></td>
</tr>
<tr>
<td>4b</td>
<td>Software Organization (Timing)</td>
<td>60 mins</td>
</tr>
<tr>
<td>4c</td>
<td>The GPU Design Space</td>
<td>30 mins</td>
</tr>
<tr>
<td>5a</td>
<td>Demo 2: Debugging Tool</td>
<td>15 mins</td>
</tr>
<tr>
<td></td>
<td><strong>Coffee Break (3:15 – 3:45pm)</strong></td>
<td></td>
</tr>
<tr>
<td>5b</td>
<td>Demo 3: Visualizing Performance</td>
<td>30 mins</td>
</tr>
<tr>
<td>6</td>
<td>Extending GPGPU-Sim: Walkthrough</td>
<td>40 mins</td>
</tr>
<tr>
<td>7</td>
<td>Wrap Up and Discussion</td>
<td>15 mins</td>
</tr>
</tbody>
</table>
AerialVision: Visualizing Complex Dynamics in GPU

Visualizer Tool for GPGPU-Sim
Introducing AerialVision

• Visualizer for GPGPU-Sim
  – Time Lapse View: Performance metrics vs. time
  – Source Code View: Performance metrics vs. the CUDA source code
  – Implemented in Python: very extensible

• GPGPU-Sim modified to generate inputs for AerialVision:

![Diagram of AerialVision](attachment:image.png)
AerialVision: Motivation

• Common case in architectural research:
  – Researchers implemented their new proposals in an architecture simulator (in this case GPGPU-Sim)
  – Simulation results across a suite of benchmarks usually look like this:

Need to understand reason behind slowdowns!
Need to validate these results!
AerialVision: Motivation

• Current ways to gain insights (GPGPU-Sim)
  – Performance statistics log @ end of kernel launch
    • Ignores runtime dynamics in the microarchitecture
  – GDB cycle by cycle stepping
    • Information overload
    • Slow

• GPU is parallel and programmable
  – Lots of insight gained from having a global view
    • Time-lapse performance variations
    • View multiple units in parallel
  – Relate performance metrics to source code
AerialVision: Motivation

• Why is runtime dynamics important for GPUs?
  – Many core accelerator architecture
  – Intermittent contentions effect performance, but not captured by end-to-end performance statistics

DRAM utilization is ~uniform for both case!
Bottleneck goes undetected!
AerialVision: Motivation

• CUDA kernels are growing large
  – MUMMerGPU = ~400 lines
  – GPUDG = ~200 lines
  – …

• Many GPGPU-Sim users are CUDA app dev
  – Use the simulator to help understand the performance of their CUDA applications
  – Need to pin point performance bottleneck in the kernel code
  – Benefits hardware architects as well
Outline

• AerialVision Motivation + Introduction
• Time Lapse View
• Source Code View
• Demo
• Overhead
• Summary
Time Lapse View

• Visualize performance metrics versus time
  – Plot up to 5 metrics in a figure for direct visual comp.
  – Infinite # figures (until memory runs out)

• Different type of plots for different metrics
  – Line plots → global singulars (e.g. IPC)
  – Parallel intensity plot → metrics across multi. HW units
  – Stack bar charts → component breakdowns
  – PC-Histogram → relate thread dynamics and source code

• Plots are generated with Matplotlib
  – Navigation tool bar for zoom and pan
  – More formatting options exposed with extra widgets
Time Lapse View – Metric(s) Selection

Add a new tab (a new figure)
Select a visualizer trace file
Select metric to plot
Select type of plot
Add more plots to the figure
Configure each subplot

Click this to start plotting!
Time Lapse View – Figure

Switch between different figures

Figure plotting selected metrics

Formatting Tool

Navigation Tool
Time Lapse View – Parallel Intensity Plot

- View performance metrics for multiple parallel hardware units vs. time
  - Change color mapping with “Change Colormap Max/Min”

Mapping from color to value of the metric

Data for each shader core
Time Lapse View – Stacked Bar Chart

- Shows the breakdown of a metric vs. time
  - Warp divergence
  - Load/Store latency breakdown

Each component is represented by a unique color.
Time Lapse View – PC-Histogram (CFLog)

- A time series of histograms, representing the portion of the program that the threads has touched during a given sample period.
- A thread is considered to have touched an instruction:
  - After it has fetched the instruction
  - Until it fetches a new instruction

Each line here represents:
- A PTX instruction
OR
- A line in CUDA source code.

Color at each dot indicates # threads touching the instruction during that sampling period.

CFLOG stands for “Control Flow” Log
Time Lapse View – Navigation and Formatting Tools

- **Change Colormap Max/Min**
  - Configure how metric values are mapped to the color spectrum
  - Allow user to normalize among all plots
  - Choose a different color scheme

- **Change Binning**
  - Modify the frequency of tick labels on the axes

- **Edit Labels**
  - Edit the labels on x and y axes, and the title/colormap label
  - Choose the fonts size for labels
Source Code View

Performance Profiler

• "VTune™" for CUDA (on GPGPU-Sim)
  – Narrow bottleneck down to a single line of source code in your application

• Features:
  – Shows performance metrics side-by-side with the source code
  – Navigation graph
    • Shows the big picture
    • Single-click to jump to the “code of interest”
Source Code View – Metric(s) Selection

Select the CUDA C file or PTX file to show with the metric

Choose how metrics should be combined (More on this next)

Choose the metric to be displayed

Click this to proceed
Source Code View – Combine Metrics

- GPGPU-Sim samples performance metrics for each PTX instruction.
- Each line of CUDA C $\rightarrow$ multiple lines of PTX instructions.
  - This mapping can be acquired via debug info generated by NVCC
- Question: How should the metrics from multiple PTX instructions be reduced?
  - Max ← Latency / Execution Count
  - Sum ← Quantity counters

\[
\begin{align*}
\text{Sum?} & : 221 & \text{Max?} & : 200 \\
\text{a[i] = b[i] + c;} & \\
\text{ld.s32 %r2, [%r1];} & : 200 \\
\text{add.f32 %r4, %r2, %r3;} & : 1 \\
\text{st.s32 %r4, [%r1];} & : 20
\end{align*}
\]

- We provide suggestions for each metric in the manual.
- Ratio between metrics? We use Max for both.
Source Code View – Navigation

CUDA C Source Code

Navigation graph

Metric Data

Format Tool etc…
Source Code View – Navigation

Viewer jumps to here, with the clicked line highlighted

Right-Click Here

Line number in CUDA C File
Overhead + Extension

- GPGPU-Sim need to do extra work to generate inputs for AerialVision
  - Simulation speed ~30% slower
  - Data dumped to HDD = 3.4kB per sample

- Turned on by default
  - sampling frequency = Every 1000 cycles

- Adding new metrics for AerialVision
  - Next Session
Setup / Installation

• To be able to use AerialVision, you only need to install its dependencies:
  – python-pmw
  – python-PLY
  – python-numpy
  – libpng12-dev
  – python-matplotlib
• Run bin/aerialvision.py in GPGPU-Sim distribution.
AerialVision Demo
Session Summary

- GDB Macro
- Debug Level
- Interactive Debugger
- AerialVision – Performance Visualizer for GPGPU-Sim
  - Time Lapse View
    - Metric vs. Time
  - Source Code View
    - Metric vs. Code
## Overview

<table>
<thead>
<tr>
<th></th>
<th>Title</th>
<th>Duration</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Brief Background on GPU Computing</td>
<td>40mins</td>
</tr>
<tr>
<td>2</td>
<td>GPGPU-Sim Overview</td>
<td>30mins</td>
</tr>
<tr>
<td>3</td>
<td>Demo 1: Setup and Run</td>
<td>15mins</td>
</tr>
<tr>
<td></td>
<td>Coffee Break (10:00 – 10:30am)</td>
<td></td>
</tr>
<tr>
<td>4a</td>
<td>Microarchitecture Timing Model</td>
<td>60mins</td>
</tr>
<tr>
<td>4b</td>
<td>Software Organization</td>
<td>30mins</td>
</tr>
<tr>
<td></td>
<td>Lunch (12:00 – 1:30pm)</td>
<td></td>
</tr>
<tr>
<td>4b</td>
<td>Software Organization (Timing)</td>
<td>60mins</td>
</tr>
<tr>
<td>4c</td>
<td>The GPU Design Space</td>
<td>30mins</td>
</tr>
<tr>
<td>5a</td>
<td>Demo 2: Debugging Tool</td>
<td>15mins</td>
</tr>
<tr>
<td></td>
<td>Coffee Break (3:15 – 3:45pm)</td>
<td></td>
</tr>
<tr>
<td>5b</td>
<td>Demo 3: Visualizing Performance</td>
<td>30mins</td>
</tr>
<tr>
<td>6</td>
<td>Extending GPGPU-Sim: Walkthrough</td>
<td>40mins</td>
</tr>
<tr>
<td>7</td>
<td>Wrap Up and Discussion</td>
<td>15mins</td>
</tr>
</tbody>
</table>
## Overview

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Brief Background on GPU Computing</td>
<td>40mins</td>
</tr>
<tr>
<td>2</td>
<td>GPGPU-Sim Overview</td>
<td>30mins</td>
</tr>
<tr>
<td>3</td>
<td>Demo 1: Setup and Run</td>
<td>15mins</td>
</tr>
<tr>
<td></td>
<td>Coffee Break (10:00 – 10:30am)</td>
<td></td>
</tr>
<tr>
<td>4a</td>
<td>Microarchitecture Timing Model</td>
<td>60mins</td>
</tr>
<tr>
<td>4b</td>
<td>Software Organization</td>
<td>30mins</td>
</tr>
<tr>
<td></td>
<td>Lunch (12:00 – 1:30pm)</td>
<td></td>
</tr>
<tr>
<td>4b</td>
<td>Software Organization (Timing)</td>
<td>60mins</td>
</tr>
<tr>
<td>4c</td>
<td>The GPU Design Space</td>
<td>30mins</td>
</tr>
<tr>
<td>5a</td>
<td>Demo 2: Debugging Tool</td>
<td>15mins</td>
</tr>
<tr>
<td></td>
<td>Coffee Break (3:15 – 3:45pm)</td>
<td></td>
</tr>
<tr>
<td>5b</td>
<td>Demo 3: Visualizing Performance</td>
<td>30mins</td>
</tr>
<tr>
<td>6</td>
<td>Extending GPGPU-Sim: Walkthrough</td>
<td>40mins</td>
</tr>
<tr>
<td>7</td>
<td>Wrap Up and Discussion</td>
<td>15mins</td>
</tr>
</tbody>
</table>
Adding a Two-Level Warp Scheduler

- Energy-efficient mechanisms for managing thread context in throughput processors. [Gebhart and Johnson et al.]
Single-Level (current) Warp Scheduler

- Loose Round-Robin Policy
- Pick a ready warp and issue it
- Next cycle, start by the next warp

Mark Gebhart 2011
Two-Level Warp Scheduler

- Two pools of warps
  - **Ready:** Ready or soon to be
  - **Pending:** Waiting on long operations

![Diagram of Two-Level Warp Scheduler]

Mark Gebhart 2011
Two-Level Warp Scheduler

• Hiding short latencies (e.g. Arith pipeline)
  – Requires a few warps: use ready pool

• Hiding long latencies (e.g. DRAM)
  – Requires a lot of warps: use pending pool
Two-Level Warp Scheduler

Policy:

- Schedule warps from the Ready pool
- Move warps to Pending pool if they read a register that will be written by a long latency operation
- Move warps to Ready pool if there is space
Why?

• Improves cache hit rates
• Lower power
  – Scheduler walks a shorter list
• (Improves register file cache performance)
Current Implementation

Class scheduler_unit{
    scheduler_unit(...);

    void add_supervised_warp_id(int);
    void cycle();
}


Virtual Base Class

Class scheduler_unit{
    scheduler_unit(...);

    virtual void add_supervised_warp_id(int);
    virtual void cycle()=0;
}

Class LooseRoundRobbinScheduler:

```cpp
public scheduler_unit {
    virtual void void cycle();
}
```

Class TwoLevelScheduler:

```cpp
public scheduler_unit {
    virtual void void cycle();
}
```
Tracking Long Operations

Augment Scoreboard to track registers depending on long operations

```cpp
bool Scoreboard::islongop(int regnum);
```
Adding a Config Option

Add the option using the option parser in 
gpu-sim.cc:

```cpp
optionParser_register( ... );
```

Read the option in shader.cc

```cpp
std::string sched_config =
    m_config->gpgpu_scheduler_string
```
Coding Time :)


## Overview

<table>
<thead>
<tr>
<th></th>
<th>Brief Background on GPU Computing</th>
<th>40mins</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>GPGPU-Sim Overview</td>
<td>30mins</td>
</tr>
<tr>
<td>3</td>
<td>Demo 1: Setup and Run</td>
<td>15mins</td>
</tr>
<tr>
<td></td>
<td>Coffee Break (10:00 – 10:30am)</td>
<td></td>
</tr>
<tr>
<td>4a</td>
<td>Microarchitecture Timing Model</td>
<td>60mins</td>
</tr>
<tr>
<td>4b</td>
<td>Software Organization</td>
<td>30mins</td>
</tr>
<tr>
<td></td>
<td>Lunch (12:00 – 1:30pm)</td>
<td></td>
</tr>
<tr>
<td>4b</td>
<td>Software Organization (Timing)</td>
<td>60mins</td>
</tr>
<tr>
<td>4c</td>
<td>The GPU Design Space</td>
<td>30mins</td>
</tr>
<tr>
<td>5a</td>
<td>Demo 2: Debugging Tool</td>
<td>15mins</td>
</tr>
<tr>
<td></td>
<td>Coffee Break (3:15 – 3:45pm)</td>
<td></td>
</tr>
<tr>
<td>5b</td>
<td>Demo 3: Visualizing Performance</td>
<td>30mins</td>
</tr>
<tr>
<td>6</td>
<td>Extending GPGPU-Sim: Walkthrough</td>
<td>40mins</td>
</tr>
<tr>
<td>7</td>
<td>Wrap Up and Discussion</td>
<td>15mins</td>
</tr>
<tr>
<td></td>
<td>Session Title</td>
<td>Duration</td>
</tr>
<tr>
<td>---</td>
<td>------------------------------------------------------</td>
<td>----------</td>
</tr>
<tr>
<td>1</td>
<td>Brief Background on GPU Computing</td>
<td>40mins</td>
</tr>
<tr>
<td>2</td>
<td>GPGPU-Sim Overview</td>
<td>30mins</td>
</tr>
<tr>
<td>3</td>
<td>Demo 1: Setup and Run</td>
<td>15mins</td>
</tr>
<tr>
<td></td>
<td><strong>Coffee Break (10:00 – 10:30am)</strong></td>
<td></td>
</tr>
<tr>
<td>4a</td>
<td>Microarchitecture Timing Model</td>
<td>60mins</td>
</tr>
<tr>
<td>4b</td>
<td>Software Organization</td>
<td>30mins</td>
</tr>
<tr>
<td></td>
<td><strong>Lunch (12:00 – 1:30pm)</strong></td>
<td></td>
</tr>
<tr>
<td>4b</td>
<td>Software Organization (Timing)</td>
<td>60mins</td>
</tr>
<tr>
<td>4c</td>
<td>The GPU Design Space</td>
<td>30mins</td>
</tr>
<tr>
<td>5a</td>
<td>Demo 2: Debugging Tool</td>
<td>15mins</td>
</tr>
<tr>
<td></td>
<td><strong>Coffee Break (3:15 – 3:45pm)</strong></td>
<td></td>
</tr>
<tr>
<td>5b</td>
<td>Demo 3: Visualizing Performance</td>
<td>30mins</td>
</tr>
<tr>
<td>6</td>
<td>Extending GPGPU-Sim: Walkthrough</td>
<td>40mins</td>
</tr>
<tr>
<td>7</td>
<td>Wrap Up and Discussion</td>
<td>15mins</td>
</tr>
</tbody>
</table>
Summary

- Basics of GPGPU-Sim
  - Timing Simulator of a modern GPU for GPU Compute
  - Setup and Run
- Internal Design
  - Timing Model
  - Software Organization
- Debug and Performance Visualization
- Extension Walkthrough
Citation

• If you use GPGPU-Sim (either 2.x or 3.x) in your publication, please cite our ISPASS 2009 paper:


• Please indicate which version of GPGPU-Sim you used / extended
  – E.g. “GPGPU-Sim version 3.1.0”
Keep in Touch

Website: www.gpgpu-sim.org
GIT Server:
  - git://dev.ece.ubc.ca/gpgpu-sim

Please give us some feedback (fill in the survey):
  http://www.surveymonkey.com/s/8LDQSL5

Need help?
  • Talk to us at ISCA
  • Check our online manual
    - www.gpgpu-sim.org/manual/
  • Subscribe to Google Group
  • Bugzilla Server (beta)
    - www.gpgpu-sim.org/bugs/