Multi2Sim 5.0

A MULTI-ISA HETEROGENEOUS SIMULATOR FOR NEXT-GENERATION ACCELERATED SYSTEMS: A TUTORIAL

XIANG GONG, XUN GONG, DAVID KAELI, YIFAN SUN, AMIR KAVYAN ZIABARI

Northeastern University Computer Architecture Research Group
Boston, MA
Outline (1/2)

Part 1 – Introduction
- Getting Started
- Multi2sim Simulation Framework
- Seamless OpenCL Execution
- Existing Supporting Architectures
- A Unified and Coherent Memory Hierarchy
- Visualization Tool

Part 2 – Multi2Sim Kepler Architecture
- CUDA support on Multi2Sim
- Kepler functional simulation
- Timing simulation
- Kepler microarchitecture
- Execution pipeline
- Architecture exploration

Break
Outline (2/2)

Part 3 – HSASim for HSAIL
- Introduction to HSA and its features
- Emulator design
- HSAIL emulation
- CPU GPU Communication
- Runtime Implementation
- Virtual driver
- Unified memory
- Emulation result and demo

Part 4 – GPU Compilation
- Introduction: from OpenCL source to kernel binary
- Compiler design
- Multi2Sim + Compiler
- Optimization Exploration

Concluding remarks
Introduction

Getting Started

• Follow our demos
  —  http://www.ece.neu.edu/groups/nucar/iiswc2016

• Installing Docker on Ubuntu
  https://docs.docker.com/engine/installation/linux/ubuntulinux/

• On Ubuntu 14.04 (kernel version must be 3.10+)
  $ uname -r
  $ sudo su
  $ apt-get install linux-image-extra-$\{uname -r\} linux-image-extra-virtual
  $ apt-key adv --keyserver hkp://p80.pool.sks-keyservers.net:80 --recv-keys 58118E89F3A912897C070ADBFF76221572C52609D
  $ echo "deb https://apt.dockerproject.org/repo ubuntu-trusty main" > /etc/apt/sources.list.d/docker.list
  $ apt-get update
  $ apt-get install docker-engine
  $ service docker start
  $ usermod aG docker <ubuntu-user-name>
  $ docker pull nucar/iiswc2016

• Run Docker
  $ docker run -it nucar/iiswc2016 bash

• Location of the demos
  — /root/part2 demo for Kepler
  — /root/part3 demo for HAS
  — /root/part4 demo for Compilation
Introduction

Multi2sim Simulation Framework

- A simulator for CPU, GPU and Heterogeneous systems
  - Support for CPU architectures: x86, ARM, and MIPS
  - Support for GPU architectures: AMD Southern Islands, NVIDIA Kepler
  - Support for HSA Intermediate Language

- Based on C++11
  - Following a coding guideline for ultimate clarity of the source

- Maintained through Github and TopofTrees
  - Github for Issue/Bug/Request tracking
  - TopOfTrees for Discussion and Agile software development model
Introduction

First Execution

• Source code

```c
#include <stdio.h>

int main(int argc, char **argv)
{
    int i;
    printf("Number of arguments: %d\n", argc);
    for (i = 0; i < argc; i++)
        printf("argv[%d] = %s\n", i, argv[i]);
    return 0;
}
```

• Native execution

```
$ test-args hello there

Number of arguments: 4
arg[0] = 'test-args'
arg[1] = 'hello'
arg[2] = 'there'
```

• Execution on Multi2Sim

```
$ m2s test-args hello there

< Simulator message in stderr >
Number of arguments: 4
arg[0] = 'test-args'
arg[1] = 'hello'
arg[2] = 'there'
< Simulator statistics >
```
Introduction
Simulator Input/Output Files

• Example of INI file format

; This is a comment.

[ Section 0 ]
Color = Red
Height = 40

[ OtherSection ]
Variable = Value

• Multi2Sim uses INI file for
  — Configuration files
  — Output statistics
  — Statistic summary in standard error output
Simulation Framework

Application-Level Simulation

- Full-system simulation
  An entire OS runs on top of the simulator. The simulator models the entire ISA, and virtualizes native hardware devices, similar to a virtual machine. Very accurate simulations, but extremely slow.

- Application-only simulation
  Only an application runs on top of the simulator. The simulator implements a subset of the ISA, and needs to virtualize the system call interface (ABI). Multi2Sim falls in this category.
Simulation Methodology

Four-Stage Simulation Process

- Modular implementation
  - Four clearly different software modules per architecture (x86, MIPS, ...)
  - Each module has a standard interface for stand-alone execution, or interaction with other modules
## Simulation Methodology

### Current Architecture Support in Multi2sim 5.0

<table>
<thead>
<tr>
<th></th>
<th>Disasm.</th>
<th>Emulation</th>
<th>Timing Simulation</th>
<th>Graphic Pipelines</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARM</td>
<td>×</td>
<td>In progress</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>MIPS</td>
<td>×</td>
<td>In progress</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>x86</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td><strong>AMD Southern Islands</strong></td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td><strong>NVIDIA Kepler</strong></td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>–</td>
</tr>
<tr>
<td><strong>HSA Intermediate Language</strong></td>
<td>×</td>
<td>×</td>
<td>In progress</td>
<td>In progress</td>
</tr>
</tbody>
</table>

- **Available in Multi2Sim 5.0**
  - Southern Islands, and x86 fully supported
  - Three other CPU/GPU architectures in progress
  - This tutorial will touch on x86, Southern Islands, and will introduce Kepler and HSAIL in detail.
Seamless OpenCL Execution
Native vs. Simulated Environment

Legend
- Black text: component generic for any runtime/driver interaction
  - Blue text: example specific for an AMD OpenCL runtime/driver/GPU.

![Diagram](image-url)

IISWC Tutorial, September 2016
Seamless OpenCL Execution

Runtime

• Compilation of Runtime Libraries
  – Seamlessly along side Multi2sim compilation

• Benchmark Compilation

<table>
<thead>
<tr>
<th>Guest program Linked \textit{dynamically} with \textbf{Multi2sim runtime}</th>
<th>Guest program Linked \textit{statically} with \textbf{Multi2sim runtime}</th>
<th>Guest program Linked \textit{dynamically} with vendor’s runtime</th>
</tr>
</thead>
<tbody>
<tr>
<td>For adding new features to runtime</td>
<td>When portability is an issue</td>
<td>Program validation purposes</td>
</tr>
</tbody>
</table>

• Supported Runtime Libraries

<table>
<thead>
<tr>
<th>Runtime</th>
<th>Vendor</th>
<th>Multi2sim</th>
</tr>
</thead>
<tbody>
<tr>
<td>OpenCL 1.1</td>
<td>libOpenCL.so</td>
<td>libm2s-opencl.so</td>
</tr>
<tr>
<td>CUDA 4.0</td>
<td>libcuda.so</td>
<td>libm2s-cuda.so</td>
</tr>
</tbody>
</table>
Seamless OpenCL Execution

OpenCL Object Management

- **OpenCL execution**
  - NDRange
  - Workgroups
  - Work-items

- **OpenCL memory**
  - Global Memory
  - Constant Memory
  - Local Memory
  - Private Memory

Diagram:

- ND-Range
  - Work-group
  - Work-group
  - ...
  - Global Memory
  - Independent set of work-groups, with no synch or communication.

- Work-Group
  - Work-item
  - Work-item
  - ...
  - Local Memory
  - Intra work-group synchronization and communication allowed at this level.

- Work-Item
  - _kernel func()
  - {
    ...
  }
  - Private Memory
  - Common OpenCL C kernel code executed by all work-items.
Existing Architectures for Detailed Simulation

X86 Superscalar Pipelines

- **Superscalar x86 pipelines**
  - 6-stage pipeline with configurable latencies
  - **Supported features** include speculative execution, branch prediction, micro-instruction generation, trace caches, out-of-order execution, …
  - **Modeled structures** include fetch queues, reorder buffer, load-store queues, register files, register mapping tables, …
Existing Architectures for Detailed Simulation

X86 Multithreaded and Multicore Processors

- **Multithreading**
  - Replicated superscalar pipelines with partially shared resources
  - Fine-grain, coarse-grain, and simultaneous multithreading

- **Multicore**
  - Fully replicated superscalar pipelines, communicating through the memory hierarchy
  - Parallel architectures can run multiple programs concurrently, or one program spawning child threads (using OpenMP, pthread, etc.)
Existing Architectures for Detailed Simulation

Simulated AMD Southern Islands (GCN 1.0) Architecture
Existing Architectures for Detailed Simulation

Logical Representation of Compute Unit

- The **instruction memory** of each compute unit contains a copy of the OpenCL kernel.

- A **front-end** fetches instructions, partly decodes them, and sends them to the appropriate execution unit.

- There is **one instance** of the following execution units: scalar unit, vector-memory unit, branch unit, LDS (local data store) unit.

- There are **multiple instances** of SIMD units.
Existing Architectures for Detailed Simulation

The GPU SIMD Compute Pipeline
Existing Architectures for Detailed Simulation

Validation Results – Using Micro-benchmarks

• Comparison in Execution Time

• Absolute time differences between Native and simulated execution
Unified and Coherent Memory Hierarchy

Configuration

• **Flexible hierarchies**
  
  – Any number of caches organized in any number of levels
  
  – Cache levels connected through default cross-bar interconnects, or complex **custom interconnect** configurations
  
  – Each architecture undergoing a timing simulation specifies its own **entry point** (cache memory) in the memory hierarchy, for data or instructions
Multi2sim Memory Management Policy

**GPU Only**

- Inclusive, write-back, non-coherent caches
- Non-coherence is implemented as a 3-state “coherence” protocol: NSI
- Blocks in N state are merged on write-back using write bit masks

**N**on-coherent – Block can be accessed for read/write access, while shared with other caches.

**S**hared – Block is accessible for read access, possibly shared with other caches.

**I**nvalid – Block does not contain valid data.
Unified Memory Hierarchy

Through in-house NMOESI protocol

- Combining MOESI protocol with the Non-coherent State (NMOESI)
  - Seamlessly executing non-coherent operations on the CPU
  - Executing coherent exclusive operations on GPU
  - Support for atomic operations through memory
  - Faster synchronization between CPU and GPU

<table>
<thead>
<tr>
<th>Block state</th>
<th>load</th>
<th>store</th>
<th>n-store</th>
<th>Eviction</th>
<th>Read request</th>
<th>Write request</th>
</tr>
</thead>
<tbody>
<tr>
<td>N</td>
<td>hit</td>
<td>write request → M</td>
<td>hit</td>
<td>writeback → I</td>
<td>-</td>
<td>send data → I</td>
</tr>
<tr>
<td>M</td>
<td>hit</td>
<td>hit</td>
<td>hit</td>
<td>writeback → I</td>
<td>send data → O</td>
<td>send data → I</td>
</tr>
<tr>
<td>O</td>
<td>hit</td>
<td>write request → M</td>
<td>hit</td>
<td>writeback → I</td>
<td>send data → I</td>
<td>send data → I</td>
</tr>
<tr>
<td>E</td>
<td>hit</td>
<td>hit → M</td>
<td>hit → N</td>
<td>→ I</td>
<td>send data → S</td>
<td>send data → I</td>
</tr>
<tr>
<td>S</td>
<td>hit</td>
<td>write request → M</td>
<td>hit → N</td>
<td>→ I</td>
<td>-</td>
<td>→ I</td>
</tr>
<tr>
<td>I</td>
<td>read request → S or → E</td>
<td>write request → M</td>
<td>read request → N</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>
Visualization Tool

Main Panel
Visualization Tool

Pipeline Diagrams

- Cycle bar on main window for navigation
- Clicking on the Detail buttons opens a secondary window
Pipeline Visualization Tool

Memory Hierarchy

- Panel on main window shows how memory accesses traverse the memory hierarchy
- Clicking on a *Detail* button opens a secondary window with the cache memory representation
- Each row is a set, each column is a way
- Each cell shows the tag and state (color) of a cache block
- Additional columns show the number of sharers and in-flight accesses
Visualization Tool

Interconnection Network

- Information about individual nodes in the network graph
- State of the packets in the buffers
- Occupancy of the buffers and links
Part 2

Simulation of a Kepler GPU
CUDA on the Host

Execution Framework

— The following slides show the **modular organization** of the CUDA execution framework, based on 4 software/hardware entities.

— In each case, we compare **native execution** with **simulated execution**.
CUDA on the Host
The CUDA CPU Host Program

• **Native**
  An x86 CUDA host program performs an CUDA API call

• **Multi2Sim**
  Exact same scenario

```
  x86 executable
  "vector-add" CUDA host program
  API call

  x86 executable
  "vector-add" CUDA host program
  API call
```
CUDA on the Host
The CUDA Runtime Library

• **Native**
  NVIDIA's CUDA runtime library handles the call, and communicates with the driver through system calls *ioctl, read, write*, etc. These are referred to as ABI calls.

• **Multi2Sim**
  Multi2Sim's CUDA runtime library, running guest code, transparently intercepts the call. It communicates with the Multi2Sim driver using a ioctl call in Linux.

![Diagram](image-url)
CUDA on the Host
The CUDA Device Driver

- **Native**
  The NVIDIA driver (kernel module) handles the ABI call and communicates with the GPU through the PCIe bus.

- **Multi2Sim**
  A CUDA driver module (Multi2Sim code) intercepts the ABI call and communicates with the GPU emulator.
CUDA on the Host

The GPU Emulator

- **Native**
  The command processor in the GPU handles the messages received from the driver.

- **Multi2Sim**
  The GPU emulator updates its internal state based on the message received from the driver.

---

![Diagram showing command through PCIe bus and calls to emulator]
CUDA on the Host

Transferring Control

• Beginning execution on the GPU
  — The key CUDA call that effectively triggers GPU execution is `cudaLaunchKernel`.

• Order of events
  — The **host program** performs an API call `cudaLaunchKernel`.
  — The **runtime** intercepts the call, and enqueues a new task in a CUDA command queue object. A user-level thread associated with the command queue eventually processes the command, performing a `LaunchKernel` ABI call.
  — The **driver** intercepts the ABI call, reads Grid parameters, and launches the GPU emulator.
  — The **GPU emulator** enters a simulation loop until the Grid completes.
CUDA on the Device

Execution Model

• **Execution components**
  - **Threads** execute multiple instances of the same kernel code.
  - **Blocks** are sets of threads that can synchronize and communicate efficiently.
  - The **Grid** is composed by all blocks, not communicating with each other and executing in any order.
CUDA on the Device

Execution Model

• **Software-hardware mapping**
  — When the kernel is launched by the Kepler driver, a CUDA **Grid** is mapped to a **compute device** (Fig. a).
  — The **thread blocks** are mapped to a **streaming multiprocessor (SM)** (Fig. b).
  — The **threads** are executed by the **SIMD lanes** (Fig. c).

— This is a simplification of the GPU architecture.
Simulation Methodology

Four-Stage Simulation Process

- **Disassembler**
  - Executable ELF file
  - Instruction bytes
  - Instruction fields
  - Instructions dump

- **Emulator (or functional simulator)**
  - Executable file, program arguments
  - Instruction information
  - Program output

- **Timing simulator (or detailed/architectural)**
  - Executable file, program arguments, processor configuration
  - Run one instruction
  - Performance statistics
The Kepler Disassembler
The Kepler Disassembler

Vector Addition Kernel

- Our simulator supports NVIDIA shader assembly (SASS)

- Source code

```c
__global__ void vectorAdd(
    const float *A,
    const float *B,
    float *c,
    int numElements )
{
    int i = blockDim.x * blockIdx.x + threadIdx.x;
    if (i < numElements)
    {
        C[i] = A[i] + B[i];
    }
}
```

- Assembly code

```assembly
// MOV R1, c[0x0][0x44];
// S2R R0, SR_CTAID.X;
// S2R R3, SR_TID.X;
// IMAD R0, R0, c[0x0][0x28], R3;
// ISETP.GE.AND P0, PT, R0, c[0x0][0x14c], PT;
// @P0 BRA.U 0x78;
// @P0 ISCADD R3, R0, c[0x0][0x140], 0x2;
// @P0 ISCADD R2, R0, c[0x0][0x144], 0x2;
// @P0 LD R3, [R3];
// @P0 ISCADD R0, R0, c[0x0][0x148], 0x2;
// @P0 LD R2, [R2];
// @P0 FADD R3, R3, R2;
// @P0 ST [R0], R3;
// MOV RZ, RZ;
// EXIT;
// BRA 0x90;
// NOP;
// NOP;
// NOP;
// NOP;
// NOP;
// NOP;
```
The Kepler Emulator

- Disassembler
- Emulator (or functional simulator)
- Timing simulator (or detailed/architectural)
The Kepler Emulator

Emulation Loop

• **Thread block execution**
  — Thread blocks can execute in **any order**. This order is irrelevant for emulation purposes.
  — The chosen policy executes **one thread block at a time**, starting thread blocks in increasing order based on their ID in each dimension.

• **Warp execution**
  — Warps within a thread block can also execute in any order, as long as **synchronization** is respected.
  — The chosen policy executes one warp at a time **until it hits a barrier**, if any.
The Kepler Emulator

• **Benchmark Support**
  - CUDA SDK 6.5 benchmark suite
  - Working on more benchmark suites (Rodinia, Parboil, etc.)
The Kepler Timing Simulation
Kepler Timing Simulation
The GPU Architecture

• When the grid is created, the Gigathread engine (scheduler) assigns thread blocks in SMs as new slots become available.
Kepler Timing Simulation

Streaming Multiprocessor

- **Sequence**
  - The **front-end** fetches instructions, decodes them, and sends them to the appropriate execution units.
  - Each unit includes 32 lanes which can execute 32 threads (a warp) simultaneously.
  - There is **one instance** of each of the following execution units: a special functional unit, a load store unit, and an integer math unit.
  - There are **multiple instances** of single precision units, double precision units, and branch units.
Kepler Timing Simulation

Streaming Multiprocessor
• **Sequence**

  — Thread blocks are allocated to 4 different *warp pools*. Each warp from a thread block is assigned a slot in the warp pool.

  — On each cycle, the **fetch stage** allows all 4 warp pools to submit requests to instruction memory.

  — The dispatch **stage** consumes an instruction from each fetch buffer and sends it to the corresponding execution unit's issue buffer, depending on the instruction type.

  — Decode and scoreboard operations also resolved in dispatch stage.
Kepler Timing Simulation

The Front-End
Kepler Timing Simulation

The Front-End
Kepler Timing Simulation
The Execution Unit

• Sequence

— Runs arithmetic-logic instructions such as integer and floating point.

— Private units can only execute instructions from the warp pool they belong to.

— Shared units can execution instructions from any warp pool.

— The execution unit pipeline is modeled with 5 stages: decode, read, execute, write, and complete.

— The register file is accessed in the read and write stages to consume input and produce output operands, respectively.
Kepler Timing Simulation

The Execution Unit
Kepler Timing Simulation

The LS Unit

- Modeled with 5 stages – decode, read, execute/memory, write, complete
**Kepler Simulation**

Functional Simulation

- **Runs one thread block at a time to completion**
  - Emulates instructions and updates registers and memory

- **Produces some limited statistics**
  - Number of executed grids and blocks
  - Dynamic instruction mix of the kernel

- ** Produces an ISA-level trace**
  - Includes an initial memory image
  - Instruction emulation trace
Kepler Simulation

Architectural Simulation

• Models SMs and the memory hierarchy

• Maps thread blocks onto SMs and warp pools

• Emulates instructions and propagates state through the execution pipelines
  — Models resource usage and contention
Need a break?
Part 3

Multi2sim-HSA
Multi2sim-HSA

Motivation

• Heterogeneous System Architecture
  — GPU as a first-class citizen
  — Better CPU-GPU integration

• Behavior & Performance
  — Not adequately studied
Multi2sim-HSA

Motivation

• Educational
  — Trying out HSA
  — Understanding HSA instructions, APIs, and features

• For Developers
  — Debugging at HSAIL level
  — Profiling

• For Computer Architects
  — Understanding the CPU-GPU interaction
  — Verifying compiler/finalizer/assembler design
  — Exploring performance optimizations
Heterogeneous System Architecture (HSA)

Background

• HSA foundation
  — [www.hsafoundation.com](http://www.hsafoundation.com)
  — Founded in 2002

• Goals
  — Reducing heterogeneous system complexity
  — Exploiting compute capabilities
  — Enabling programming in standard languages
  — Pursuing high performance
  — Increasing developer productivity
Heterogeneous System Architecture (HSA)

Features

• Shared Virtual Memory + Flat Addressing
  — Global memory
  — Memory protection
  — Address translation
  — Segmented spaces

• Coherency and Consistency
  — Global memory coherency
  — Scoped consistency
Heterogeneous System Architecture (HSA)

Features

• **User mode queueing + AQL**
  — From any device to any device
  — Memory based flexiblity

• **Signaling**
  — Send a message to and from any device
  — No busy waiting
Heterogeneous System Architecture (HSA)

HSAIL

• Why HSAIL?
  — Abstract away the implementation specific code
  — Faster to finalize – with BRIG

```c
prog kernel &__vector_copy_kernel kernarg_u64 %in, kernarg_u64 %out){
    @__vector_copy_kernel_entry:
    workitemabsid_u32 $s0, 0;
    cvt_s64_s32 $d0, $s0;
    shl_u64 $d0, $d0, 2;
    ld_kernarg_align(8)_width(all)_u64 $d1, [%out];
    add_u64 $d1, $d1, $d0;
    ...
    ld_global_u32 $s0, [$d0];
    st_global_u32 $s0, [$d1];
    ret;
}
```
Heterogeneous System Architecture (HSA)

HSAIL in Big Picture

- OpenCL App
- C++ HC App
- C++ 17 App
- CUDA App
- JAVA App

- OpenCL Runtime
- C++ HC Runtime
- C++ 17 Runtime
- CUDA Runtime
- JAVA Runtime

- HSA Runtime
- HSA Finalizer

- HSA Vendor 1

- HSA Vendor n
Heterogeneous System Architecture (HSA)

Software Stack

- **C++ HC**
  - HCC Compiler
  - ROCR Runtime
  - ROCT Trunk Interface
  - ROCK Kernel Driver
  - Devices

- **OpenCL**
  - CLOC Compiler + SNACK
  - ROC-smi
  - ROCnRDMA
  - Hardware

- **CUDA**
  - HIP
  - Programming Model

- **CUDA**
  - Compiler
  - Runtime
  - Kernel mode Driver
Multi2sim-HSA

Simulation Methodology

- Host program runs on Multi2Sim-X86
- M2S specific HSA runtime and driver
- Emulator consumes BRIG File
Multi2sim-HSA
HSA Emulator

• **Virtual Device File**
  — “/dev/hsa”

• **IOCTL System Call**
  — IOCTL(hsa_runtime->fd, ABI_CALL_NAME, DATA);

• **Runtime functions**
  — Prepares data for driver functions
  — Invokes IOCTL call
Multi2sim-HSA
Simulation Methodology

BRIG

Disassembler

Instruction Fields

Instruction Bytes

HSAIL

Driver Control

Emulator

Program output

Detailed simulator

Visual tool
Multi2sim-HSA

HSAIL Emulator

Emulator
- CPU
- GPU

Agent
- Kernel

Kernel
- WG
- WF
- WI
- SF
- Reg File
- PC
- Var Table

WG
- Control by Driver

WF
- Control by Kernel

WorkItem
- SF

Frame

Global Segment

Private Segment

Kernarg Segment

Group Segment

Memory

X86 Emulator
Multi2sim-HSA
HSAIL Emulator

- Emulator
  - CPU
  - GPU
- Agent
  - Kernel
- Kernel
  - WG
    - WF
    - WI
- WG
  - WF
  - WI
- WF
  - WI
- WorkItem
  - SF
  - PC
  - Var
  - Table
- Frame
  - Reg
  - File
- Controlled by Configuration
- Controlled by Driver
- Controlled by Kernel
- Kernarg Segment
- Group Segment
- Private Segment
- Global Segment
- Memory
- X86 Emulator
Multi2sim-HSA

Kernel Start

• **AQL Queue**
  – Doorbell signal of the AQL queue
  – Notify agent to check the AQL queue

• **On packet detected**
  – Grid, Work-group, Wavefront, Work-item are created
Multi2sim-HSA

HSAIL Emulator

Emulator
- CPU
- GPU

Agent
- Kernel

Kernel
- WG
  - WF
  - WF
  - WF
  - WF
  - WF
  - ...
  - WF
- WG
- WG
- WG
- ...
- WG

WG
Controlled by Driver

WF
Controlled by Kernel

WorkItem
- SF
- SF
- SF
- SF
- ...
- SF

Frame
- Reg File
- PC
- Var Table

Global Segment

Private Segment

Group Segment

Kernarg Segment

Memory

X86 Emulator
Multi2sim-HSA

Execution and Synchronization

• No lock step
  — Device independent

• Work-item status
  — Running
  — Waiting for barrier

• Barrier Queue

• Stack per work-item
  — Work-item may execute a different function
  — State associated with a stack frame
Multi2sim-HSA
Execution and Synchronization

• No lock step
  — Device independent

• Work-item status
  — Running
  — Waiting for barrier

• Barrier Queue

• Stack per work-item
  — Work-item may execute a different function
  — State associated with a stack frame
Multi2sim-HSA

HSAIL Emulator

- Emulator
  - CPU
  - GPU

- Agent
  - Kernel

- Kernel
  - WG
    - WG
    - WG
    - WG
    - ...
    - WG

- WG
  - WF
    - WI
    - ...
    - WF

- WF
  - WI
    - SF
    - ...
    - SF

- WorkItem
  - SF
    - PC
    - Var
    - Table

- Frame
  - Reg
  - File

- Controlled by Configuration

- Controlled by Driver

- Controlled by Kernel

- Kernarg Segment

- Group Segment

- Private Segment

- Global Segment

- Memory

- X86 Emulator
Multi2sim-HSA

Stack Frame

- **Register Files**
  - Pre-scan BRIG file for register usage
  - Register file + register offset table

- **Variable Table**
  - Variable name
  - The value of a variable is always

![Register Table](image)

<table>
<thead>
<tr>
<th>Segment</th>
<th>Var Name</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>Global</td>
<td>%p</td>
<td>0x0004</td>
</tr>
<tr>
<td>Kernarg</td>
<td>%in</td>
<td>0x0008</td>
</tr>
<tr>
<td>Private</td>
<td>%k</td>
<td>0x00016</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Empty space is not even allocated in the simulator
Multi2sim-HSA
HSAIL Emulator

Emulator
- CPU
- GPU

Agent
- Kernel

Kernel
- WG
- WF
- WI
- SF
- Reg File
- PC
- Var Table

Controlled by Configuration

Controlled by Driver

Controlled by Kernel

Global Segment

Memory

X86 Emulator
Multi2sim-HSA
Segmented Memory Manager

• **Shared with CPU**
  — Supports pointer passing
  — Virtual addressing
  — Acquires main memory at hsa_init()

• **Segment manager**
  — Delegates a memory region
  — 2 mode, pre-allocation or by request
Multi2sim-HSA

HSAIL Emulator

Emulator
- CPU
- GPU

Agent
- Kernel

Kernel
- WG
  - WF
  - Wi

WG
- WF
- Wi

WF
- Wi
- SF

WorkItem
- SF
- Pc
- Var
- Table

Frame

Controlled by Configuration

Controlled by Driver

Controlled by Kernel

Kernarg Segment

Group Segment

Private Segment

Global Segment

Memory

X86 Emulator
Multi2sim-HSA

How to use the Simulator

• **Compilation**
  - OpenCL kernel: `cloc.sh`
  - Handwritten HSAIL kernel: `HSAILasm`
  - Native HSA host: `gcc -m32`, dynamic link HSA runtime
  - SNACK: `snack.sh -m32`

• **Execution**
  
m2s [m2s args] hsa_program [program args]
Multi2sim-HSA

Logs

2371 WorkItem: 0
2372 Executing: cbr_b1 $c0, @BB1_2;
2373 ***** Stack frame *****
2374 Function: &__OpenCL_kmeans_kernel_swap_kernel,
2375 Program counter (offset in code section): 0xa54,
2376 @BB1_2:
2377
2378 ***** Registers *****
2379  $c0: true
2380  $d0: 134765088, 134765088, 0.000000, 0x0000000008085a20
2381  $d1: 134773968, 134773968, 0.000000, 0x0000000008087cd0
2382  $d2: 134765856, 134765856, 0.000000, 0x0000000008085d20
2383  $d3: 134773980, 134773980, 0.000000, 0x0000000008087cdc
2384  $s0: 30, 30, 0.000000, 0x00000001e
2385  $s1: 64, 64, 0.000000, 0x000000040
2386  $s2: 256, 256, 0.000000, 0x0000000100
2387  $s3: 4, 4, 0.000000, 0x00000004
2388  $s4: 1128202240, 1128202240, 191.000000, 0x433f0000
2389  ***** ********* *****
2390
2402
2403 ***** Backtrace *****
2404 #1 &__OpenCL_kmeans_kernel_swap_kernel (To be supported)
2405 ***** ********* *****
Case Study

Reliability

• Motivation
  — GPU memory is usually not protected by ECC
  — In Shared Virtual Memory (SVM), GPU errors can propagate back to the CPU side

• Implementation

• Command

  m2s --hsa-register-fault-injection \\ [num_instruction] hsa-program [program args]
Multi2sim-HSA

RoadMap

• **Timing simulator**
  — HSAIL timing simulation?
  — Finalized code and simulate final ISA

• **Volcanic-Islands simulator**
  — Disassembler – Emulator – Simulator

• **Flexible Memory Configuration**
  — Physical – virtual mapping

• **Visual Debugger**
  — Instruction by instruction
Part 4

Compilation of GPU Kernels
Compiler Toolchain
Three-Stage Compilation Process

- Modular implementation
  - Each module allows for stand-alone execution
Compiler Front-End

Clang Front-End → LLVM Back-End → Southern Islands Assembler
Compiler Front-End

- **Clang Front-End**
  - Based on Clang 3.8
  - Translate an OpenCL Kernel (.cl) to LLVM bitcode (.bc)

- **Work with libccl**
  - Libccl implements library requirements of OpenCL 1.1
  - Translate OpenCL built-in functions to target specific functions
Compiler Front-End

OpenCL and LLVM Code

- **OpenCL Source Code**

```c
__kernel void vector_add(
  __read_only __global int *src1,
  __read_only __global int *src2,
  __write_only __global int *dst)
{
  int id = get_global_id(0);
  dst[id] = src1[id] + src2[id];
}
```

- **LLVM Disassembly**

```llvm
define void @vector_add(
 i32 addrspace(1)* %src1,
 i32 addrspace(1)* %src2,
 i32   addrspace(1)* %dst)
{
  block_0:
    %tmp_0 = call i32 @__get_global_id_u32(i32 0)
    %tmp_3 = getelementptr i32 addrspace(1)* %dst, i32 %tmp_0
    %tmp_6 = getelementptr i32 addrspace(1)* %src1, i32 %tmp_0
    %tmp_7 = load i32 addrspace(1)* %tmp_6
    %tmp_10 = getelementptr i32 addrspace(1)* %src2, i32 %tmp_0
    %tmp_11 = load i32 addrspace(1)* %tmp_10
    %tmp_12 = add i32 %tmp_7, %tmp_11
    store i32 %tmp_12, i32 addrspace(1)* %tmp_3
  ret void
}
```

- **OpenCL to LLVM Interface**
  - Choose translation from OpenCL specifics to LLVM
  - Ex.) Address spaces, built in functions, etc.
Compiler Front-End

• Benchmark Support
  ─ Currently support most of the AMDAPP-2.9 benchmark suite
  ─ Support for more benchmark suites (Rodinia, Parboil, etc.) on the horizon

Demo
Compiler Back-End

Clang Front-End → LLVM Back-End → Southern Islands Assembler
Compiler Back-End

- Translate LLVM bitcode (.ll) to SI Assembly (.s)

- Based on LLVM AMDGCN Backend
  - Initially designed for OpenGL shaders
  - Customized passes and modifications
Compiler Back-End

- **SIMD Execution**
  - OpenCL and LLVM assume a perspective of a single thread
  - Assembly instructions execute in groups of 64 (wavefronts)

- **Thread Divergence**
  - Assembly instructions within a wavefront can have different paths of execution when they evaluate conditions differently
  - The entire wavefront must follow all possible execution paths, selectively activating work-items.
  - The active mask (AM) and active mask stack (AMS) data structures are used
  - The compiler must generate instructions to manage the AM and AMS
  - Already handled by the LLVM backend for OpenGL
The Southern Islands Back-End

Instruction Scheduling

1) Instruction Scheduling affects performance

2) Pre-RA and Post-RA instruction scheduling passes

3) Pre-RA applies at DAG level

4) Post-RA applies at ISA level
The Southern Islands Back-End

Register Allocation

1) LLVM has 4 register allocation algorithms

2) AMDGCN has scalar and vector registers

3) Register usage affects GPU occupancy

4) Default: Greedy register allocator
The Southern Islands Back-End

Ongoing Work / In Progress

Demo Time
The Southern Islands Assembler

- OpenCL Front-End
- Southern Islands Back-End
- Southern Islands Assembler
The Southern Islands Assembler

• Module to translate SI Assembly (.s) to a GPU compliant binary (.bin)

• Related Work – CLRadeonExtender
The Southern Islands Assembler

Structure

• Structure is similar to known assemblers
  — gas (GNU Assembler)
  — PCSpim MIPS assembler

• Five Sections in Assembly File

  1) .global
     — Kernel name

  2) .metadata
     — Assembly directives for the creation of the final binary

  3) .args
     — Kernel arguments with information about storage, type, etc.

  4) .data
     — Declarations of static initialized memory for the kernel

  5) .text
     — Southern Islands Assembly Instructions
The Southern Islands Assembler

File Formatting

• **Assembly File Format**
  - Format is very similar to the original OpenCL source whenever possible

• **OpenCL Function Prototype**

  ```
  __kernel void
  binarySearch(
    __global uint4 * outputArray,
    __const __global uint * sortedArray,
    unsigned int findMe,
    unsigned int globalLowerBound,
    unsigned int globalUpperBound,
    unsigned int subdivSize
  )
  ```

• **“.args” Section in Assembly File**

  ```
  .global binarySearch
  .args
    u32[4]* outputArray 0 uav10 RW
    u32* sortedArray 16 uav11 const RO
    u32 findMe 32
    u32 globalLowerBound 48
    u32 globalUpperBound 64
    u32 subdivSize 80
  ```
The Southern Islands Assembler

SI Binary File Format

- Nested ELF Structure of SI Binary Format

![Diagram showing Nested ELF Structure of SI Binary Format]

- ELF Header
  - Sections: .shstrtab, .symtab, .strtab, .rodata, .text
  - Symbol Table: kernel, header, metadata

- Outer ELF
  - ELF Header
  - Sections: .shstrtab, .symtab, .strtab, .rodata, .text
  - Symbol Table: kernel, header, metadata

- Inner ELF
  - ELF Header
  - Sections: .shstrtab, .symtab, .strtab, .data, .text
  - Symbol Table: UAVs, Constant Buffers

SI Assembly
The Southern Islands Assembler

Impact

• **Research**
  - The assembler guarantees a specified execution for a GPU (i.e., no optimization). This allows testing of individual units in the GPU such as the ALUs.

• **Validation**
  - SI Assembler is being used to validate the SI Emulator by creating custom assembly files to stress particular instructions.
The Southern Islands Assembler

Future Work

• **Support the entire AMDAPP-2.9 Benchmark Suite**
  — Currently, 23/39 benchmarks are support by the SI Assembler

• **Support more benchmark suites**
  — Rodinia
  — Parboil
Concluding Remarks
Simulation Support

Supported Benchmarks

• **Ongoing Research and Development**

  ✓ Expanding the soft error modeling capabilities of Multi2sim
  ✓ Developing new visualization capabilities
  ✓ Continued development of novel NoC configurations and technologies
  ✓ Developing a detailed simulation environment for DDR4, HBM, and HMC memories
  ✓ Developing a new Verilog+Multi2sim simulation platform through the Verilog Procedural Interface
  ✓ Developing a new power model interface between Multi2sim and DSENT
  ✓ Developing a detailed power model for Southern Islands GPU Architecture
  ✓ Exploring side-channel modeling capabilities in Multi2sim
  ✓ … and more
Simulation Support

Supported Benchmarks

- **CPU benchmarks**
  - SPEC 2000 and 2006
  - Mediabench
  - SPLASH2
  - PARSEC 2.1

- **GPU benchmarks**
  - AMD SDK 2.5 Evergreen
  - AMD SDK 2.5 Southern Islands
  - AMD SDK 2.5 x86 kernels
  - Rodinia
  - Parboil
The Multi2Sim Community

Publications

• **Main Multi2Sim Publications**

• **Example conferences in citations**
  — MICRO 2012
  — HPCA 2013
  — ICS 2015
  — NOCS 2015
  — IISWC 2016
  — TACO 2016 … and more
The Multi2Sim Community

Collaboration Opportunities

• **Current collaborators**
  — Univ. of Mississippi, Univ. of Toronto, Univ. of Texas, Univ. Politecnica de Valencia (Spain), Edinburgh Napier College, Boston University, AMD, NVIDIA

• **Multi2Sim on Github**
  — Requesting and tracking the development of new features

Top of Trees (**www.TopOfTrees.com**)  
— Online framework for collaborative software development  
— Code peer reviews  
— Forum  
— Agile Software development

• **Multi2Sim Project**  
  — 655 users registered (9/24/2016)
The Multi2Sim Community

Thank you to our many sponsors.........
Thank you!