Multi2Sim 4.1

Multi-Architecture ISA-Level Simulation of OpenCL

Dana Schaa, Rafael Ubal

Northeastern University
Boston, MA

Outline

Introduction

Simulation methodology

Part 1 – Simulation of an x86 CPU

Emulation

Timing simulation

Memory hierarchy

Visualization tool

OpenCL on the host

Part 2 – Simulation of a Southern Islands GPU

OpenCL on the device

The Southern Islands ISA

The GPU architecture

Southern Islands simulation

Validation results

Improving heterogeneity

Concluding remarks

Introduction

Getting Started

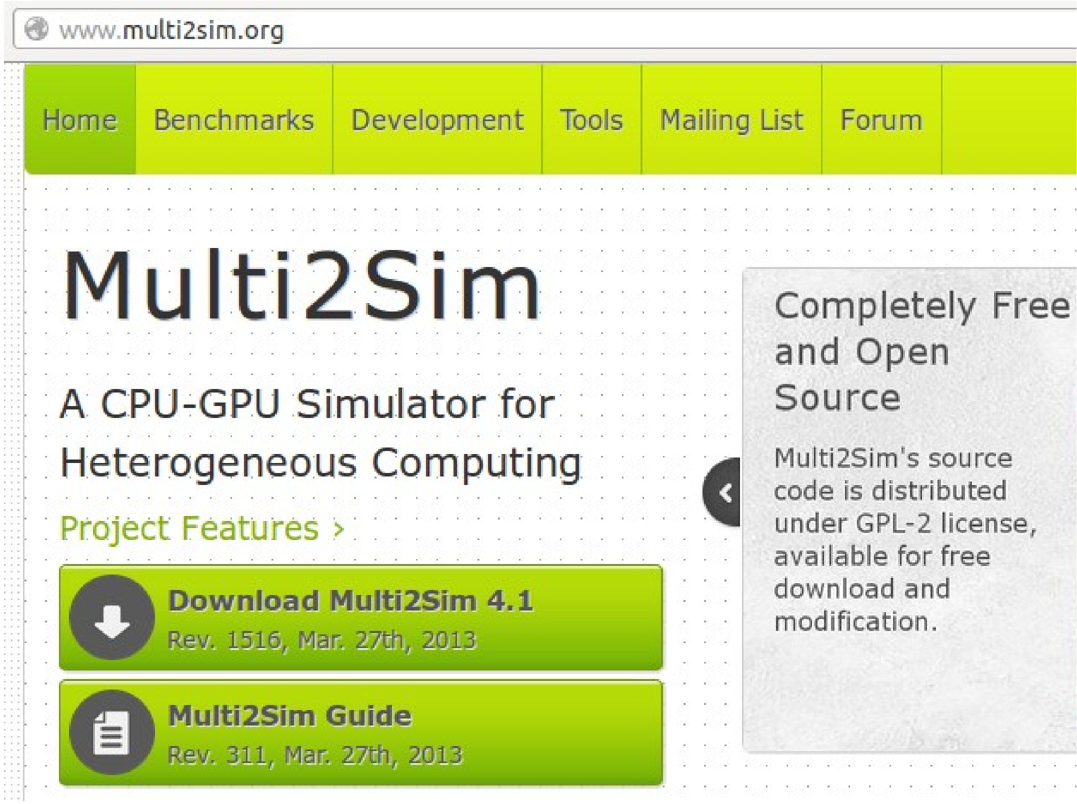
Follow our demos!

- User accounts for demos

```
$ ssh iwocl<N>@fusion1.ece.neu.edu -X
    Password: iwocl2013
```

- Installation of Multi2Sim

```
$ wget http://www.multi2sim.org/files/multi2sim-4.1.tar.gz
$ tar -xzf multi2sim-4.1.tar.gz
$ cd multi2sim-4.1
$ ./configure && make
```



Multi2Sim

A CPU-GPU Simulator for Heterogeneous Computing

Project Features ›

Download Multi2Sim 4.1
Rev. 1516, Mar. 27th, 2013

Multi2Sim Guide
Rev. 311, Mar. 27th, 2013

Completely Free and Open Source

Multi2Sim's source code is distributed under GPL-2 license, available for free download and modification.

Introduction

First Execution

- Source code

```
#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("\targv[%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 >
```

Demo 1

Introduction

Simulator Input/Output Files

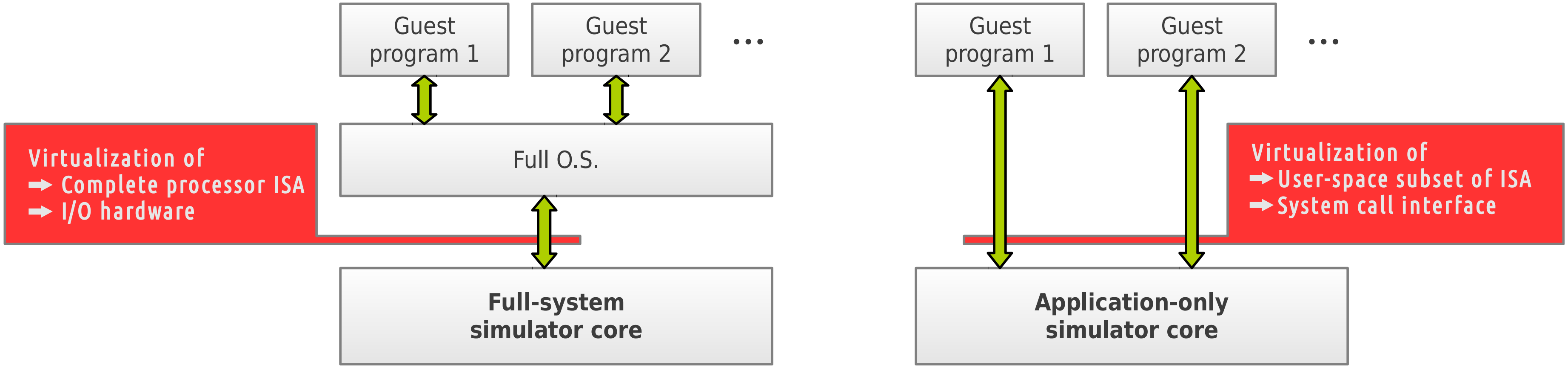
- Example of INI file format

```
; 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 Methodology

Application-Only vs. Full-System



- Full-system simulation

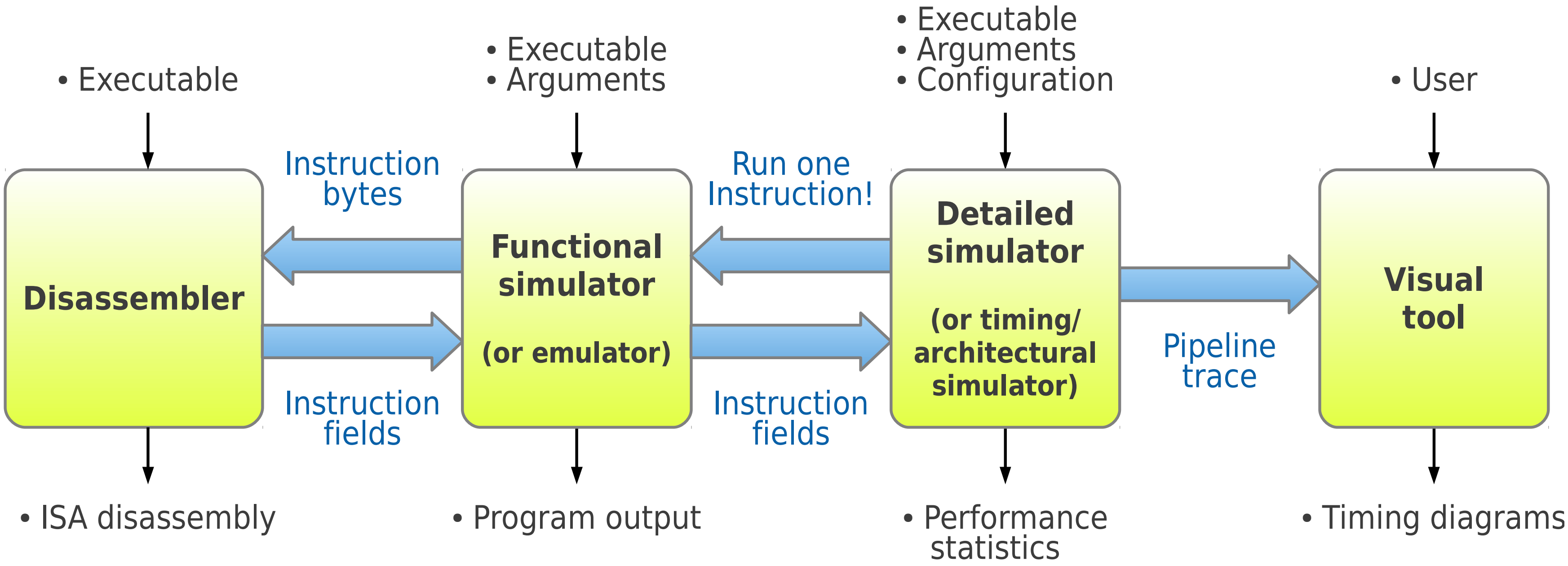
  An entire OS runs on top of the simulator. The simulator models the entire ISA, and virtualizes native hardware devices, similar to a virtual machine. Very accurate simulations, but extremely slow.

- 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

	Disasm.	Emulation	Timing Simulation	Graphic Pipelines
ARM	X	In progress	-	-
MIPS	X	-	-	-
x86	X	X	X	X
AMD Evergreen	X	X	X	X
AMD Southern Islands	X	X	X	X
NVIDIA Fermi	X	In progress	-	-

- Available in Multi2Sim 4.1
  - Evergreen, Southern Islands, and x86 fully supported.
  - Three other CPU/GPU architectures in progress.
  - This tutorial will focus on x86 and Southern Islands.

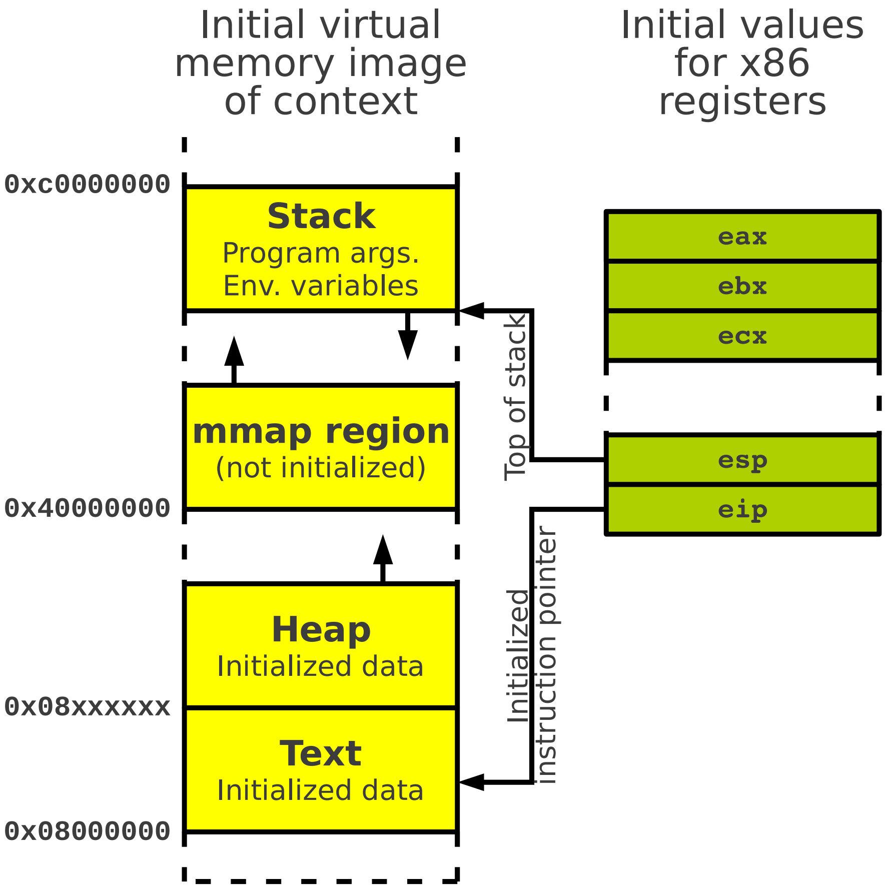
Part 1

Simulation of an x86 CPU

Emulation of an x86 CPU

Program Loading

- Initialization of a process state



1) Parse ELF executable

- Read ELF sections and symbols.
- Initialize code and data.

2) Initialize stack

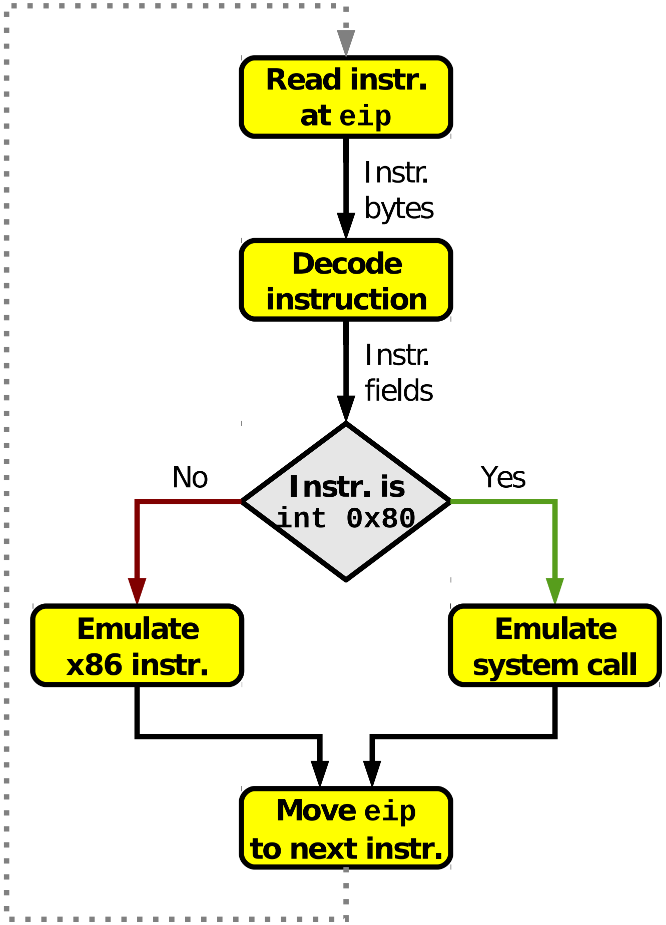
- Program headers.
- Arguments.
- Environment variables.

3) Initialize registers

- Program entry → *eip*
- Stack pointer → *esp*

Emulation of an x86 CPU

Emulation Loop

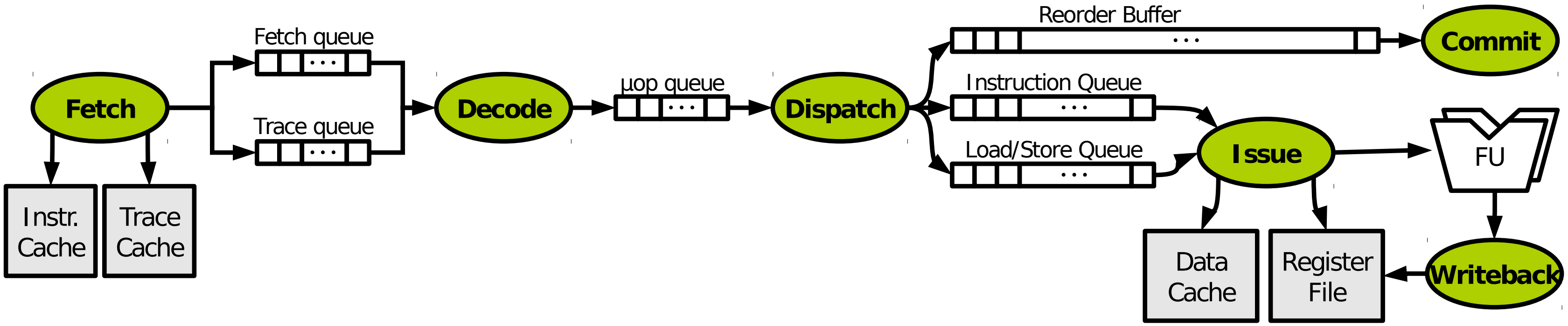


- Emulation of x86 instructions
  - Update x86 registers.
  - Update memory map if needed.
  - Example: *add [bp+16], 0x5*
- Emulation of Linux system calls
  - Analyze system call code and arguments.
  - Update memory map.
  - Update register *eax* with return value.
  - Example: *read(fd, buf, count)*

Demo 2

Timing Simulation of an x86 CPU

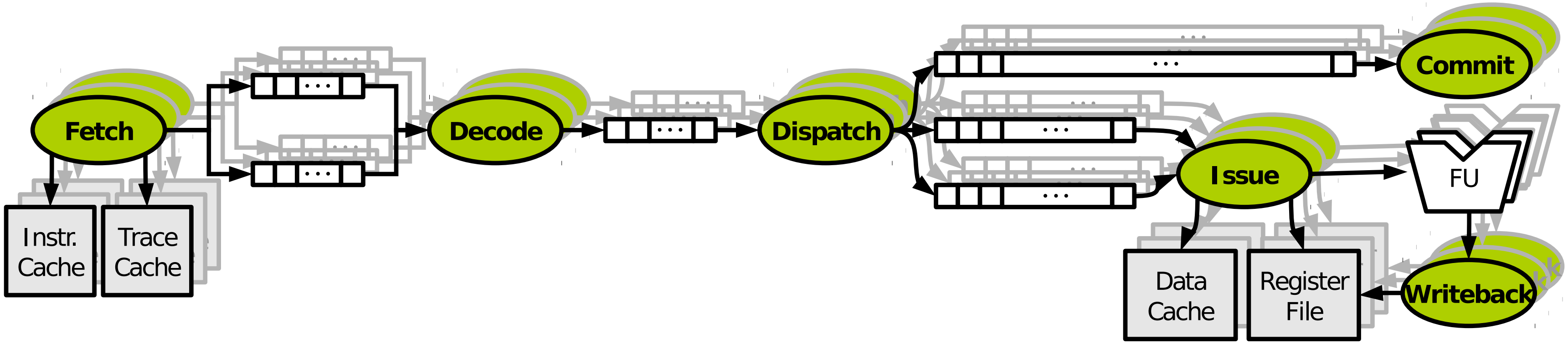
Superscalar Processor



- 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, ...

Timing Simulation of an x86 CPU

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.)

Demo 3

The Memory Hierarchy

Configuration

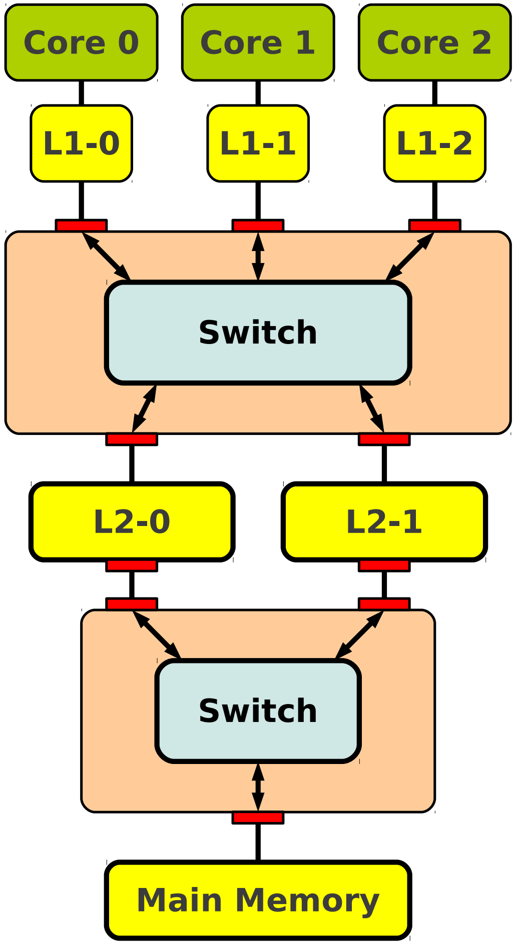
- Flexible hierarchies
  - **Any number of caches** organized in any number of levels.
  - Cache levels connected through default cross-bar interconnects, or complex **custom interconnect** configurations.
  - Each architecture undergoing a timing simulation specifies its own **entry point** (cache memory) in the memory hierarchy, for data or instructions.
  - Cache coherence is guaranteed with an implementation of the 5-state **MOESI protocol**.

The Memory Hierarchy

Configuration Examples

Example 1

Three CPU cores with private L1 caches, two L2 caches, and default cross-bar based interconnects. Cache L2-0 serves physical address range [0, 7ff...ff], and cache L2-1 serves [80...00, ff...ff].



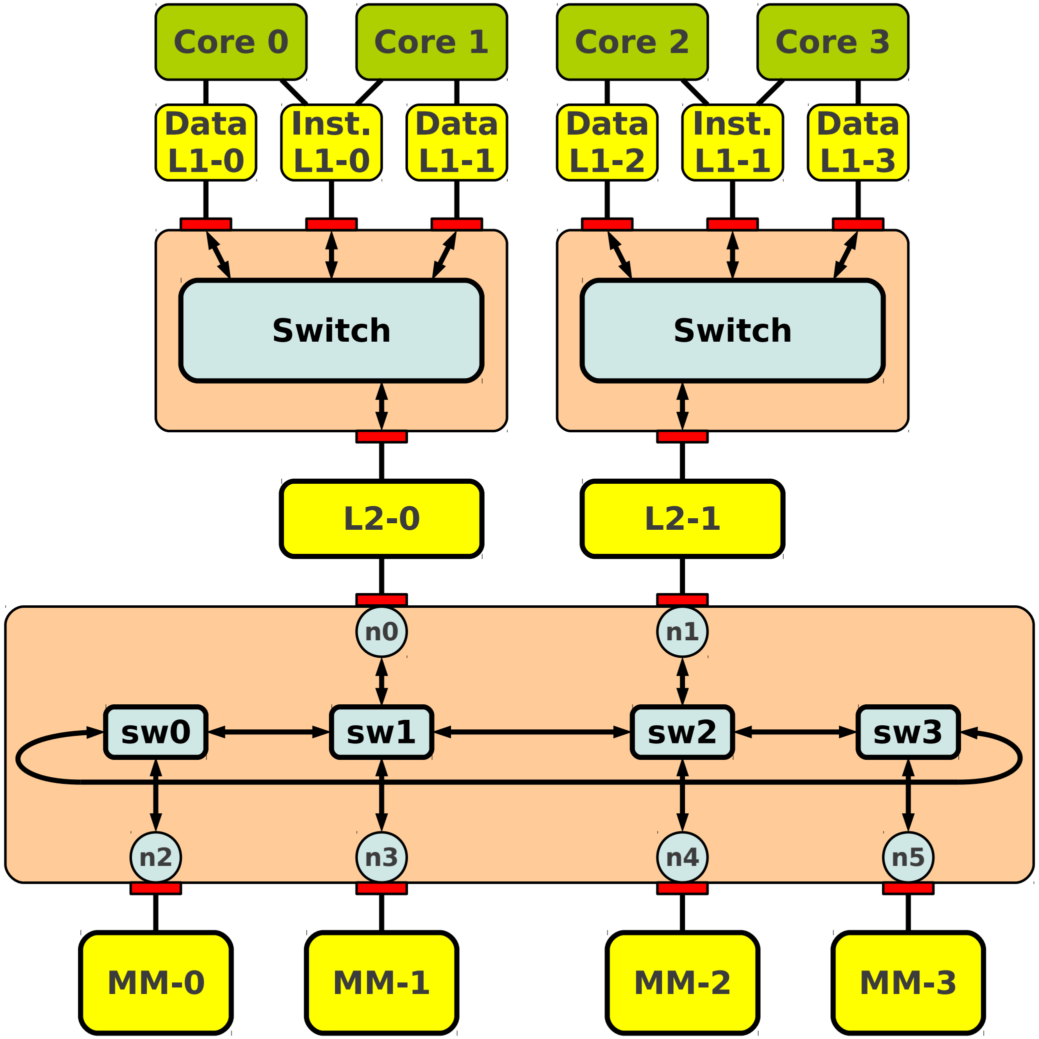
Demo 4

The Memory Hierarchy

Configuration Examples

Example 2

Four CPU cores with private L1 data caches, L1 instruction caches and L2 caches shared every 2 cores (serving the whole address space), and four main memory modules, connected with a custom network on a ring topology.

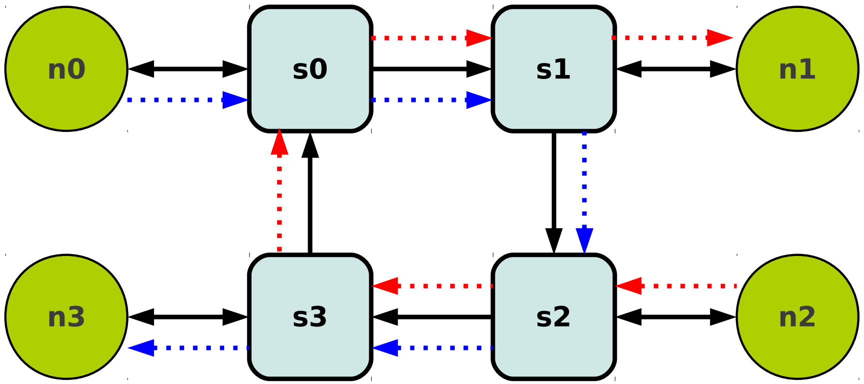


The Memory Hierarchy

Configuration Examples

Example 3

Ring connection between four switches associated with end-nodes with routing tables calculated automatically based on shortest paths. The resulting routing algorithm can contain cycles, potentially leading to routing deadlocks at runtime.

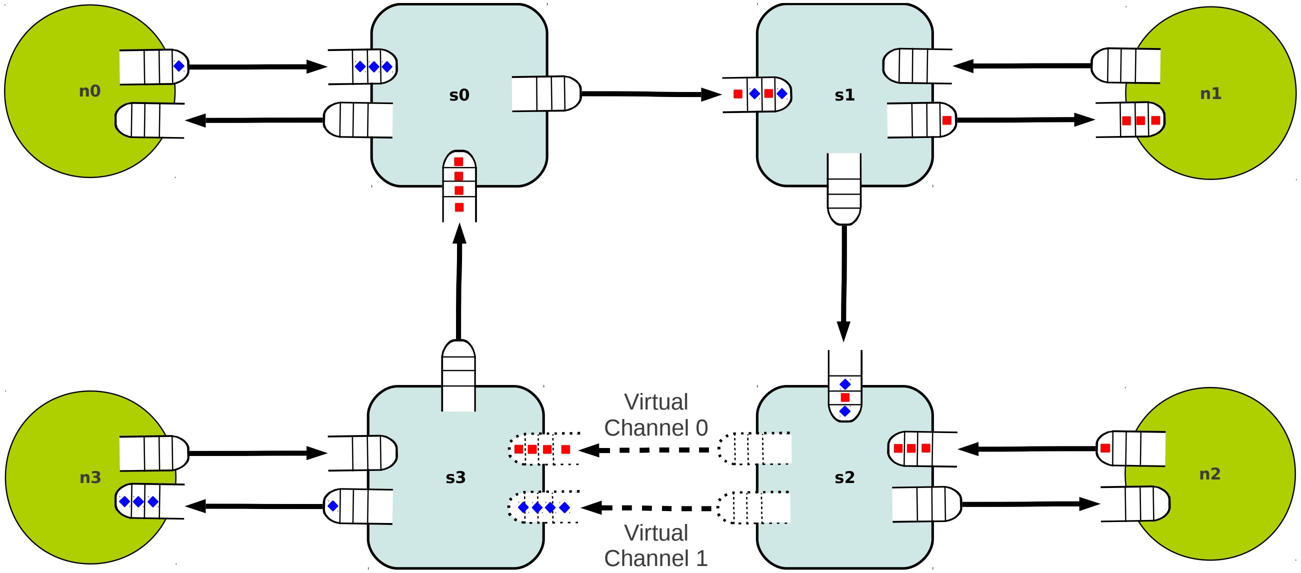


The Memory Hierarchy

Configuration Examples

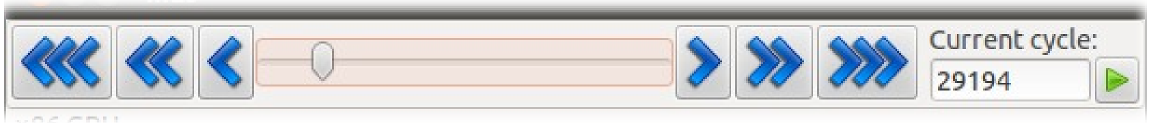
Example 4

Ring connection between for switches associated with end nodes, where a routing cycle has been removed by adding an additional virtual channel.

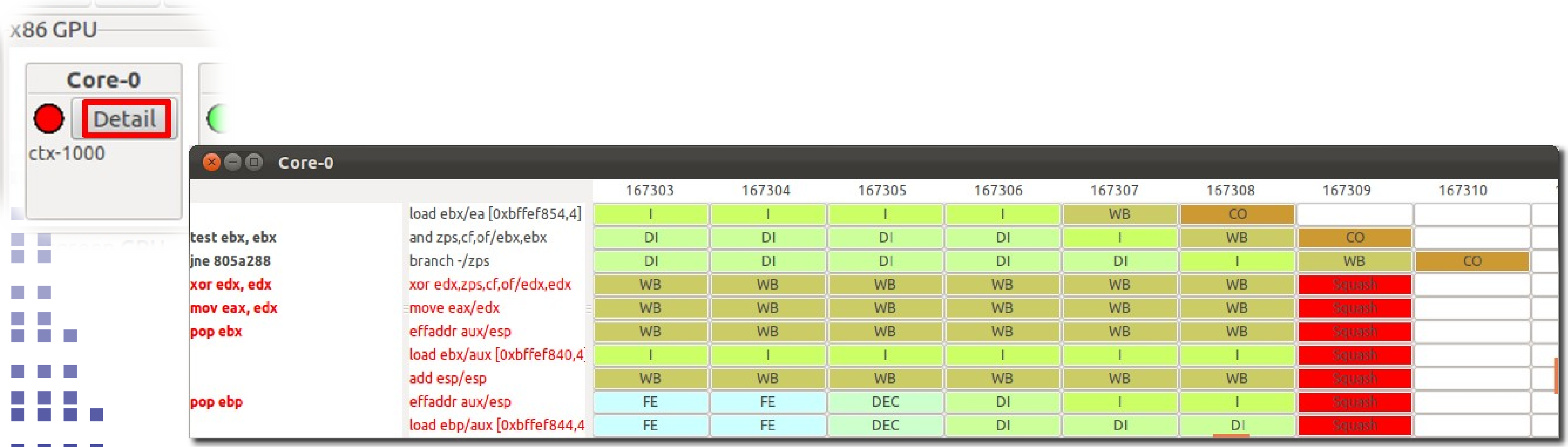


Pipeline Visualization Tool

Pipeline Diagrams



- Cycle bar on main window for navigation.
- Panel on main window shows software contexts mapped to hardware cores.
- Clicking on the *Detail* button opens a secondary window with a pipeline diagram.

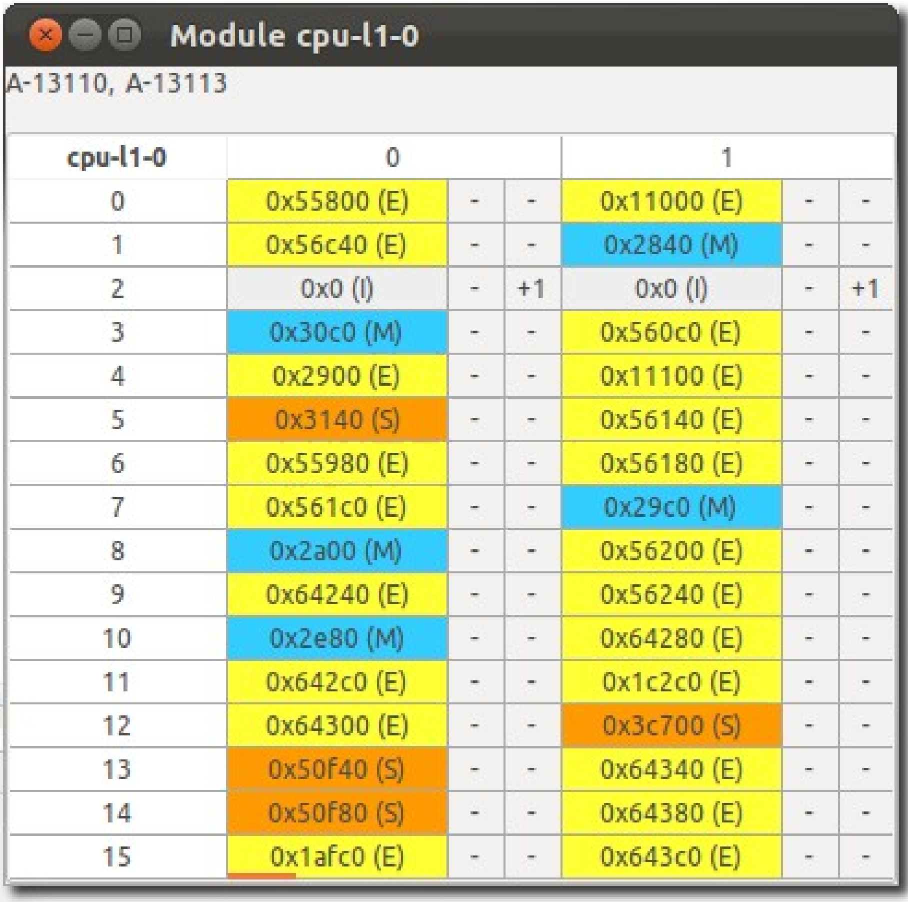


		167303	167304	167305	167306	167307	167308	167309	167310
	load ebx/ea [0xbffef854,4]	I	I	I	I	WB	CO		
test ebx, ebx	and zps,cf,of/ebx,ebx	DI	DI	DI	DI	I	WB	CO	
jne 805a288	branch -/zps	DI	DI	DI	DI	DI	I	WB	CO
xor edx, edx	xor edx,zps,cf,of/edx,edx	WB	WB	WB	WB	WB	WB	Squash	
mov eax, edx	move eax/edx	WB	WB	WB	WB	WB	WB	Squash	
pop ebx	effaddr aux/esp	WB	WB	WB	WB	WB	WB	Squash	
	load ebx/aux [0xbffef840,4	I	I	I	I	I	I	Squash	
	add esp/esp	WB	WB	WB	WB	WB	WB	Squash	
pop ebp	effaddr aux/esp	FE	FE	DEC	DI	I	I	Squash	
	load ebp/aux [0xbffef844,4	FE	FE	DEC	DI	DI	DI	Squash	

Pipeline Visualization Tool

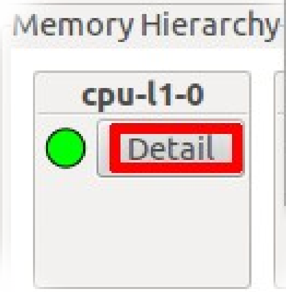
Memory Hierarchy

- 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.



A-13110, A-13113

cpu-l1-0	0			1		
0	0x55800 (E)	-	-	0x11000 (E)	-	-
1	0x56c40 (E)	-	-	0x2840 (M)	-	-
2	0x0 (I)	-	+1	0x0 (I)	-	+1
3	0x30c0 (M)	-	-	0x560c0 (E)	-	-
4	0x2900 (E)	-	-	0x11100 (E)	-	-
5	0x3140 (S)	-	-	0x56140 (E)	-	-
6	0x55980 (E)	-	-	0x56180 (E)	-	-
7	0x561c0 (E)	-	-	0x29c0 (M)	-	-
8	0x2a00 (M)	-	-	0x56200 (E)	-	-
9	0x64240 (E)	-	-	0x56240 (E)	-	-
10	0x2e80 (M)	-	-	0x64280 (E)	-	-
11	0x642c0 (E)	-	-	0x1c2c0 (E)	-	-
12	0x64300 (E)	-	-	0x3c700 (S)	-	-
13	0x50f40 (S)	-	-	0x64340 (E)	-	-
14	0x50f80 (S)	-	-	0x64380 (E)	-	-
15	0x1afc0 (E)	-	-	0x643c0 (E)	-	-



Demo 5

OpenCL on the Host

Execution Framework

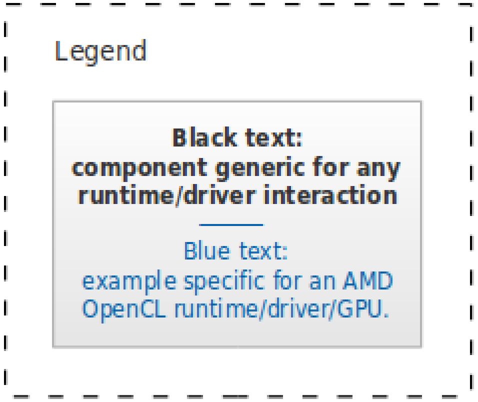
- Multi2Sim 4.1 includes a **new execution framework for OpenCL**, developed in collaboration with University of Toronto.
- The new framework is a more accurate analogy to a native execution, and is fully **AMD-compliant**.
- When working with x86 kernel binaries, the OpenCL runtime can perform both **native** and **simulated** execution correctly.
- When run natively, an OpenCL call to *clGetDeviceIDs* returns only the **x86 device**.
- When run on Multi2Sim, *clGetDeviceIDs* returns **one device per supported architecture**: x86, Evergreen, and Southern Islands devices (more to be added).

OpenCL on the Host

Execution Framework

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

- In each case, we compare **native execution (left)** with **simulated execution (right)**.



Legend

Black text:
component generic for any runtime/driver interaction

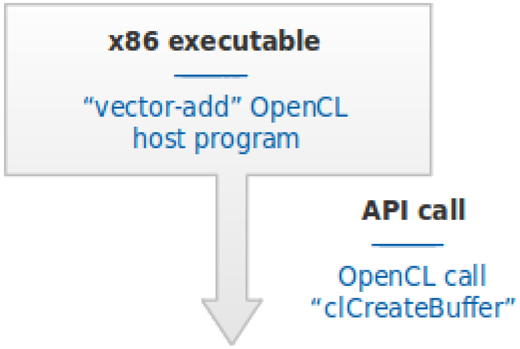
Blue text:
example specific for an AMD OpenCL runtime/driver/GPU.

OpenCL on the Host

The OpenCL CPU Host Program

Native

An x86 OpenCL host program performs an OpenCL API call.



x86 executable

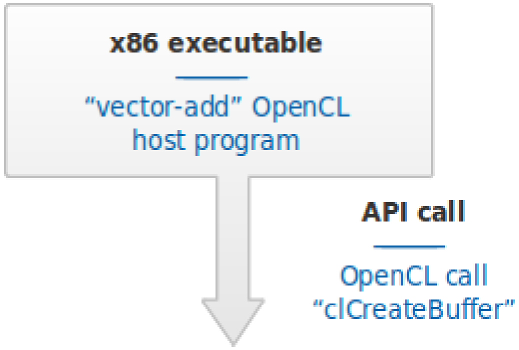
"vector-add" OpenCL host program

API call

OpenCL call "clCreateBuffer"

Multi2Sim

Exact same scenario.



x86 executable

"vector-add" OpenCL host program

API call

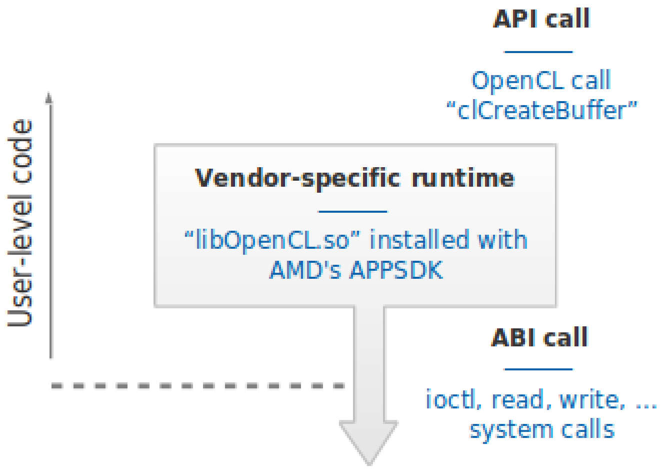
OpenCL call "clCreateBuffer"

OpenCL on the Host

The OpenCL Runtime Library

Native

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



User-level code

API call

OpenCL call "clCreateBuffer"

Vendor-specific runtime

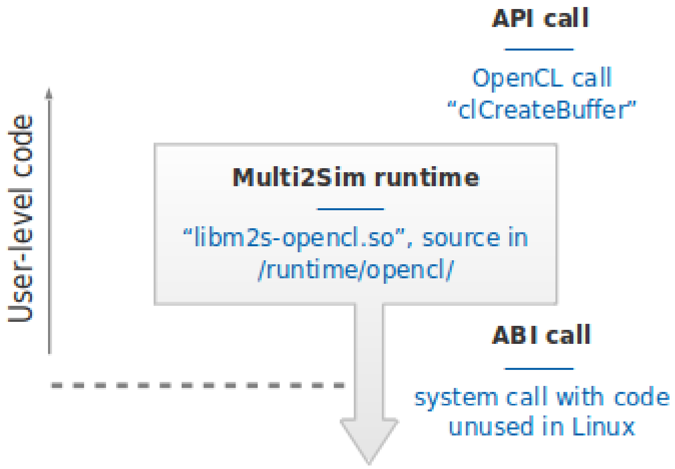
"libOpenCL.so" installed with AMD's APPSDK

ABI call

ioctl, read, write, ... system calls

Multi2Sim

Multi2Sim's OpenCL runtime library, running with guest code, transparently intercepts the call. It communicates with the Multi2Sim driver using system calls with codes not reserved in Linux.



User-level code

API call

OpenCL call "clCreateBuffer"

Multi2Sim runtime

"libm2s-opencl.so", source in /runtime/opencl/

ABI call

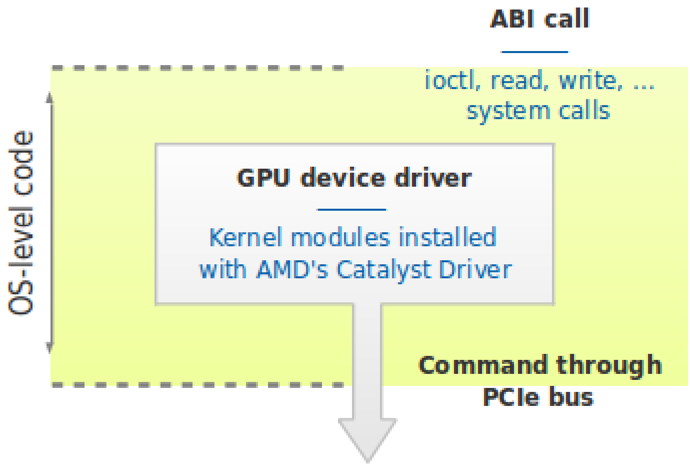
system call with code unused in Linux

OpenCL on the Host

The OpenCL Device Driver

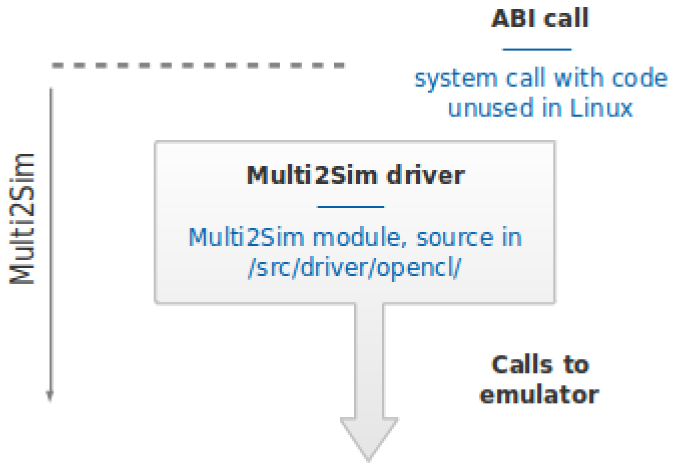
Native

The AMD Catalyst driver (kernel module) handles the ABI call and communicates with the GPU through the PCIe bus.



Multi2Sim

An OpenCL driver module (Multi2Sim code) intercepts the ABI call and communicates with the GPU emulator.



OpenCL on the Host

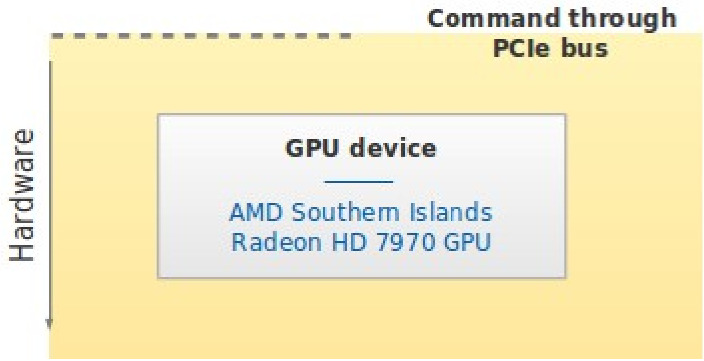
The GPU Emulator

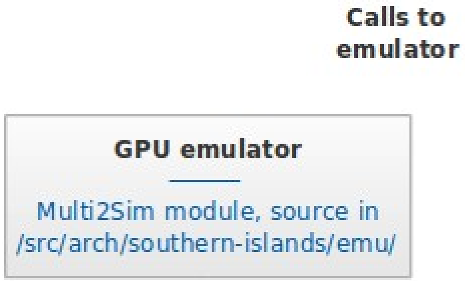
Native

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.





OpenCL on the Host

Transferring Control

- Beginning execution on the GPU
  - The key OpenCL call that effectively triggers GPU execution is *clEnqueueNDRangeKernel*.
- Order of events
  - The **host program** performs API call *clEnqueueNDRangeKernel*.
  - The **runtime** intercepts the call, and enqueues a new task in an OpenCL 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 ND-Range parameters, and launches the GPU emulator.
  - The **GPU emulator** enters a simulation loop until the ND-Range completes.

Need a break?

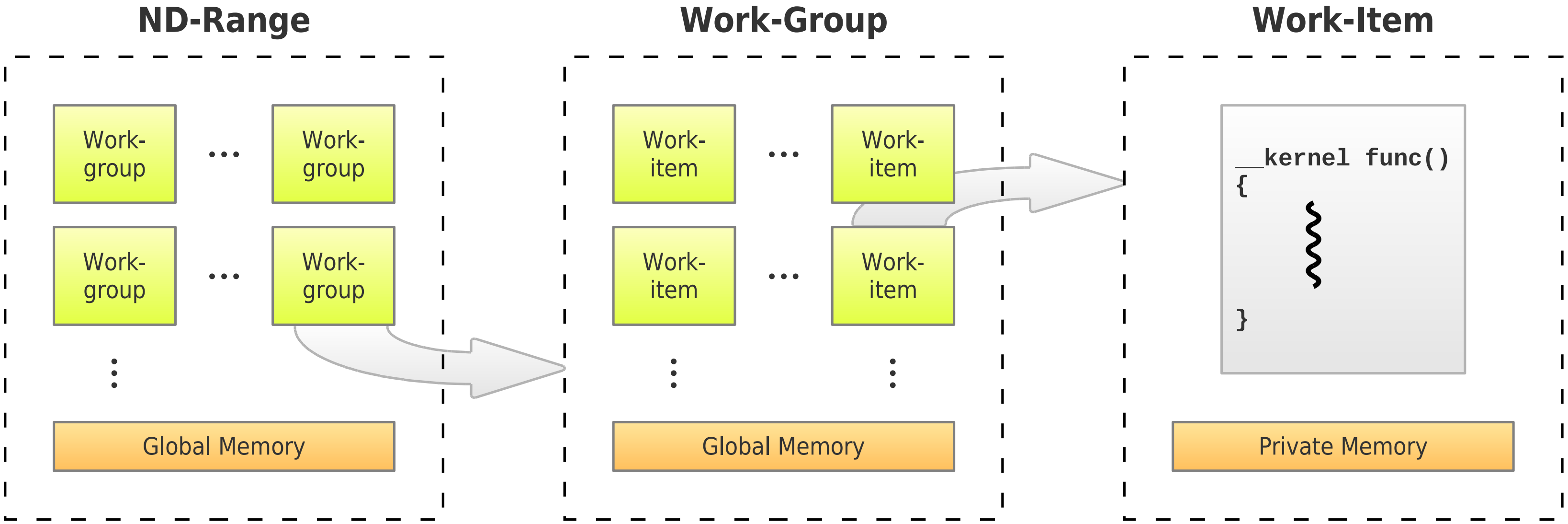
Part 2

Simulation of a Southern Islands GPU

OpenCL on the Device

Execution Model

- Execution components
  - **Work-items** execute multiple instances of the same kernel code.
  - **Work-groups** are sets of work-items that can synchronize and communicate efficiently.
  - The **ND-Range** is composed by all work-groups, not communicating with each other and executing in any order.

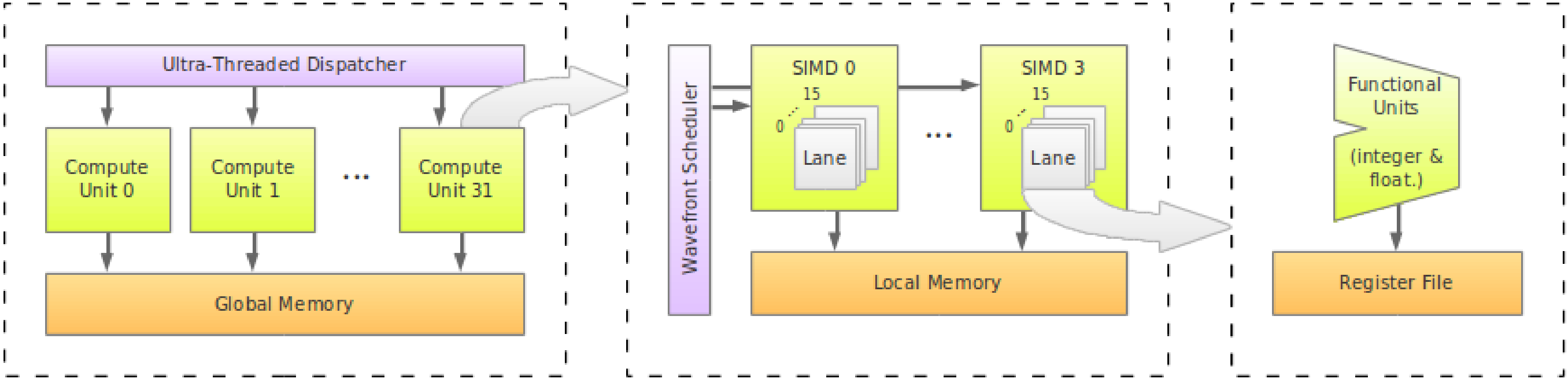


OpenCL on the Device

Execution Model

- Software-hardware mapping
  - When the kernel is launched by the Southern Islands driver, the OpenCL **ND-Range** is mapped to the **compute device** (Fig. a).
  - The **work-groups** are mapped to the **compute units** (Fig. b).
  - The **work-items** are executed by the **SIMD lanes** (Fig. c).

  - This is a simplification of the GPU architecture. The following slides show a more detailed structure of a Southern Islands compute unit.



a) Compute device.

b) Compute unit.

c) SIMD lane (stream core).

The Southern Islands ISA

Vector Addition Source

```
__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];
}
```

The Southern Islands ISA

Wavefront

- Up to 64 OpenCL work-item (software threads) are combined into a single hardware thread called a **wavefront**

- A wavefront executes on a SIMD unit (single PC, different data per work-item)
  - An **execution mask** is used to mask results of inactive work-items

The Southern Islands ISA

Wavefront – Scalar Opportunities

- Sometimes all work-items in a wavefront will execute an instruction using the same data
  - Loading the base address of a buffer
  - Incrementing/evaluating loop counters
  - Loading constant values)

- To optimize for these scenarios, AMD separates **scalar instructions** from **vector instructions** in their ISA
  - Scalar instructions execute on a new hardware unit called the **scalar unit**

The Southern Islands ISA

Disassembly for Vector Addition Kernel

```
s_buffer_load_dword  s0, s[4:7], 0x04                     // 00000000: C2000504
s_buffer_load_dword  s1, s[4:7], 0x18                     // 00000004: C2008518
s_buffer_load_dword  s4, s[8:11], 0x00                    // 00000008: C2020900
s_buffer_load_dword  s5, s[8:11], 0x04                    // 0000000C: C2028904
s_buffer_load_dword  s6, s[8:11], 0x08                    // 00000010: C2030908
s_load_dwordx4  s[8:11], s[2:3], 0x58                     // 00000014: C0840358
s_load_dwordx4  s[16:19], s[2:3], 0x60                    // 00000018: C0880360
s_load_dwordx4  s[20:23], s[2:3], 0x50                    // 0000001C: C08A0350
s_waitcnt     lgkmcnt(0)                                  // 00000020: BF8C007F
s_min_u32     s0, s0, 0x0000ffff                          // 00000024: 8380FF00 0000FFFF
v_mov_b32     v1, s0                                      // 0000002C: 7E020200
v_mul_i32_i24  v1, s12, v1                                // 00000030: 1202020C
v_add_i32     v0, vcc, v0, v1                             // 00000034: 4A000300
v_add_i32     v0, vcc, s1, v0                             // 00000038: 4A000001
v_lshlrev_b32  v0, 2, v0                                  // 0000003C: 34000082
v_add_i32     v1, vcc, s4, v0                             // 00000040: 4A020004
v_add_i32     v2, vcc, s5, v0                             // 00000044: 4A040005
v_add_i32     v0, vcc, s6, v0                             // 00000048: 4A000006
tbuffer_load_format_x  v1, v1, s[8:11], 0 offen format:
    [BUF_DATA_FORMAT_32,BUF_NUM_FORMAT_FLOAT]             // 0000004C: EBA01000 80020101
tbuffer_load_format_x  v2, v2, s[16:19], 0 offen format:
    [BUF_DATA_FORMAT_32,BUF_NUM_FORMAT_FLOAT]             // 00000054: EBA01000 80040202
s_waitcnt     vmcnt(0)                                    // 0000005C: BF8C1F70
v_add_i32     v1, vcc, v1, v2                             // 00000060: 4A020501
tbuffer_store_format_x  v1, v0, s[20:23], 0 offen format:
    [BUF_DATA_FORMAT_32,BUF_NUM_FORMAT_FLOAT]             // 00000064: EBA41000 80050100
s_endpgm                                                  // 0000006C: BF810000
```

The Southern Islands ISA

Instruction Set Features

- AMD 6000-series GPUs (**Northern Islands**) had 1 SIMD with 16 4-way VLIW lanes per compute unit
  - 64 lanes total per compute unit

- AMD 7000-series GPUs (**Southern Islands**) have 4 SIMDs, each with 16-lanes, per compute unit
  - Still 64 lanes total per compute unit

The GPU Architecture

Instruction Set Features

- Each compute unit has 4 **wavefront pools** (our term) where allocated wavefronts reside
  - Each wavefront pool is associated with one SIMD

- Each cycle, wavefronts from one wavefront pool are considered
  - One instruction from up to 5 wavefronts can be issued per datapath
  - One instruction can be issued per datapath

The GPU Architecture

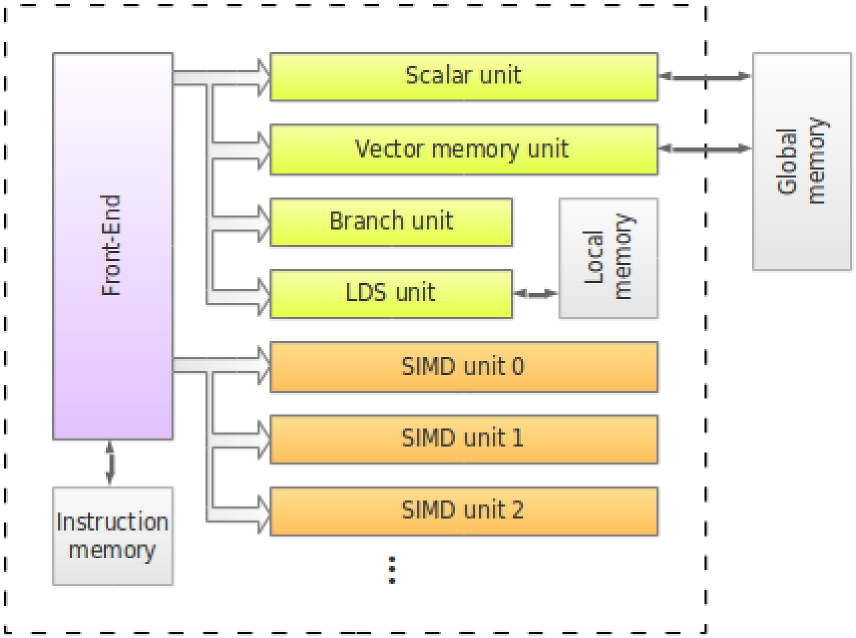
Instruction Set Features

- Simulated datapaths are:
  - Vector ALU (SIMD)
  - Vector memory (global memory)
  - Scalar unit (ALU and scalar memory)
  - Branch unit
  - LDS unit (local memory)

The GPU Architecture

Compute Unit

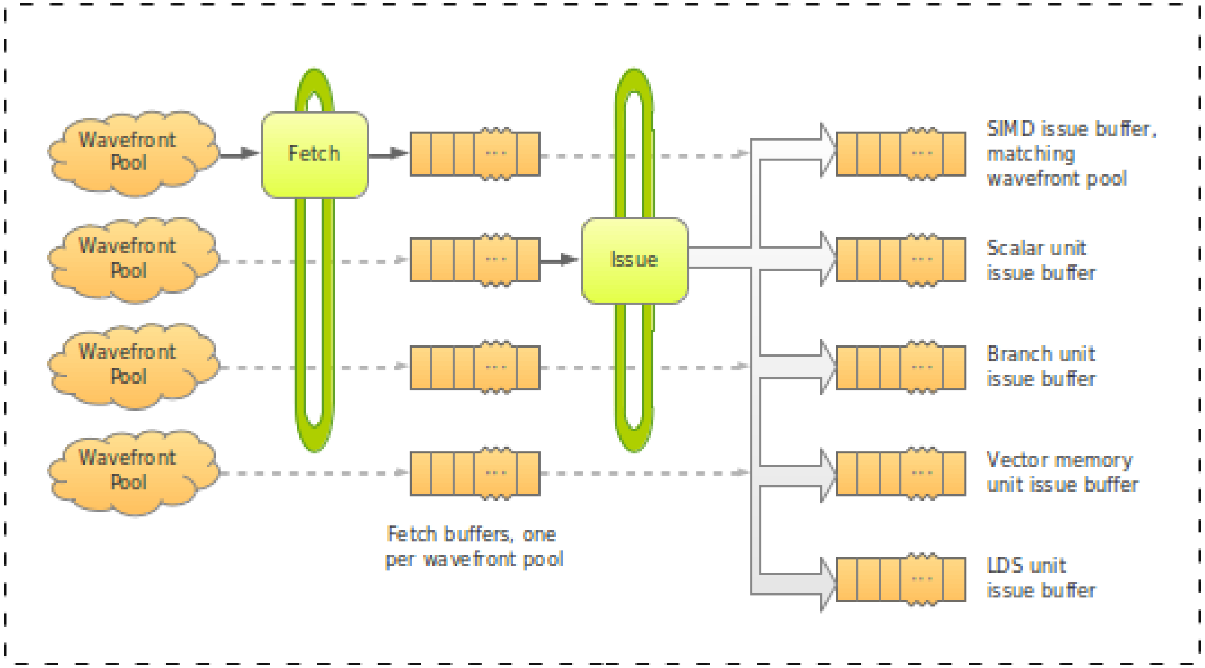
- The **instruction memory** of each compute unit contains a copy of the OpenCL kernel.
- A **front-end** fetches instructions, partly decodes them, and sends them to the appropriate execution unit.
- There is **one instance** of the following execution units: scalar unit, vector-memory unit, branch unit, LDS (local data store) unit.
- There are **multiple instances** of SIMD units.



The GPU Architecture

The Front-End

- Work-groups are allocated to 4 different **wavefront pools**. Each wavefront from a work-group is assigned a slot in the wavefront pool.
- Each cycle, the **fetch stage** allows one wavefront pool to submit requests to instruction memory
- The **issue stage** consumes an instructions from one fetch buffer and sends it to the corresponding execution unit's issue buffer, depending on the instruction type.



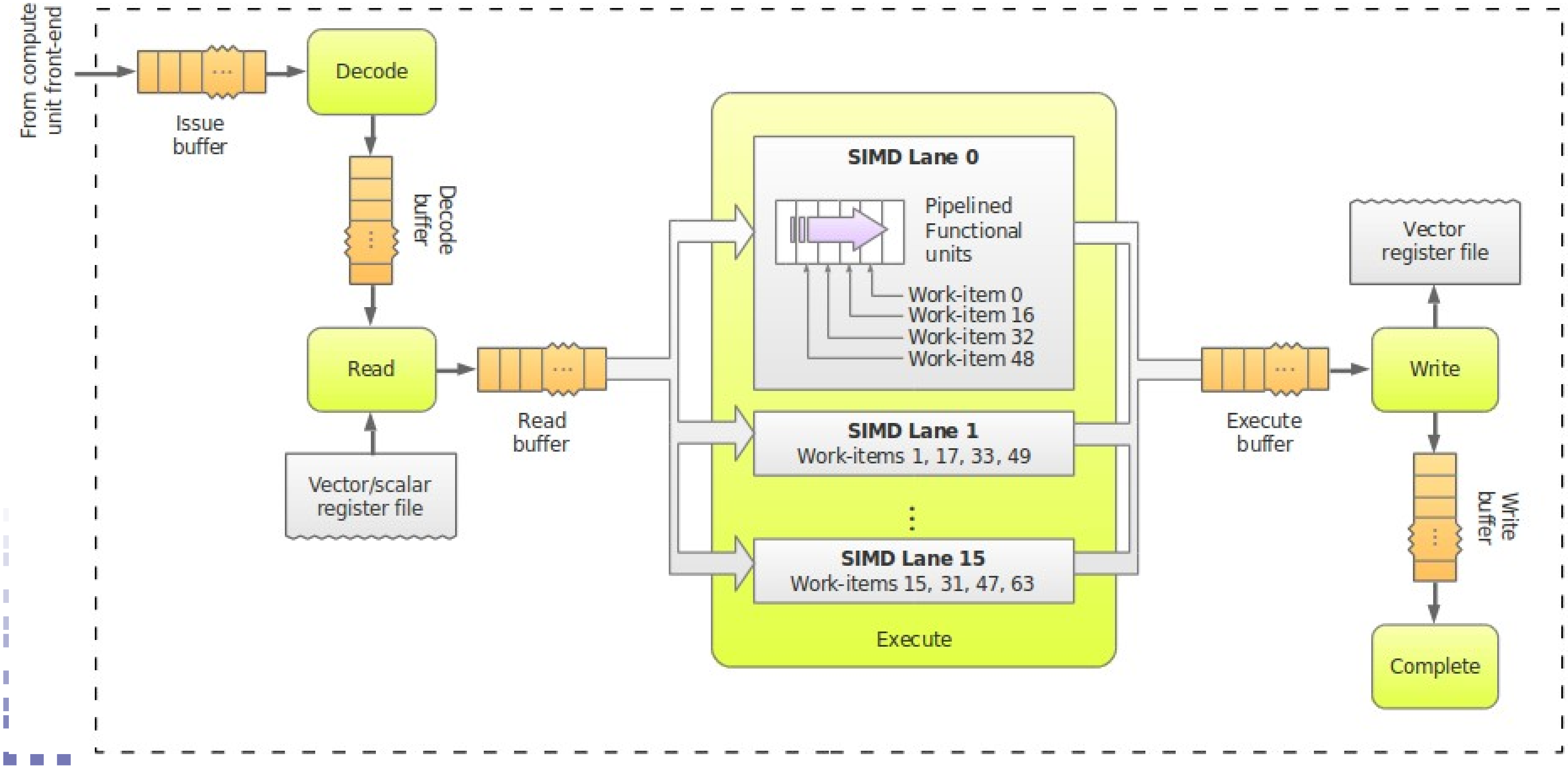
The GPU Architecture

The SIMD Unit

- Runs **arithmetic-logic vector** instructions.
- There are **4 SIMD units**, each one associated with one of the 4 wavefront pools.
- The SIMD unit pipeline is modeled with **5 stages**: decode, read, execute, write, and complete.
- In the **execute stage**, a wavefront (64 work-items max.) is split into 4 **subwavefronts** (16 work-items each). Subwavefronts are pipelined over the **16 stream cores** in 4 consecutive cycles.
- The vector register file is accessed in the **read** and **write** stages to consume input and produce output operands, respectively.

The GPU Architecture

The SIMD Unit



Southern Islands Simulation

Functional Simulation

- Sets up the memory image
- Runs one work-group at a time to completion
  - Emulates instructions and updates registers and memory
- Produces some limited statistics
  - Number of executed ND-Ranges and work-groups
  - Dynamic instruction mix of the ND-Range
- Produces an ISA trace
  - Listing memory image initialization
  - Instruction emulation trace

Demo 6

Southern Islands Simulation

Architectural Simulation

- Models compute units and the memory hierarchy
- Maps work-groups onto compute units and wavefront pools
- Emulates instructions and propagates state through the execution pipelines
  - Models resource usage and contention

Southern Islands Simulation

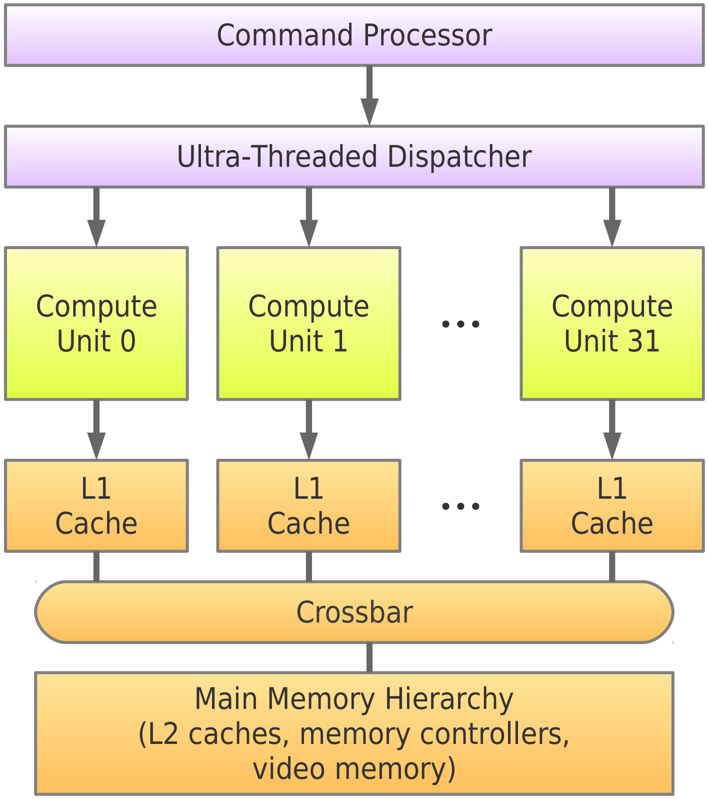
Architectural Simulation

- Fully configurable via **configuration files**
  - Number of compute units
  - Number of each execution unit (e.g. SIMDs) per compute unit
  - Latencies of pipeline stages
  - Memory modules and cache hiearchy
    - Levels, banks, sets, associativity, line size, read/write ports, interconnect network, link bandwidths, etc.
  - Issue policy (oldest instruction first, greedy)

- Configuration files are provided with Multi2Sim that model existing GPU models
  - Provided in: multi2sim/samples/southern-islands
  - 7970, 7870, 7850, 7770 are available

Southern Islands Simulation

Architectural Simulation



Southern Islands Simulation

Visualization Tool

- Step through program execution
- View in-flight state of pipelines an memory hierarchy

Demo 7

Southern Islands Simulation

Memory Hierarchy

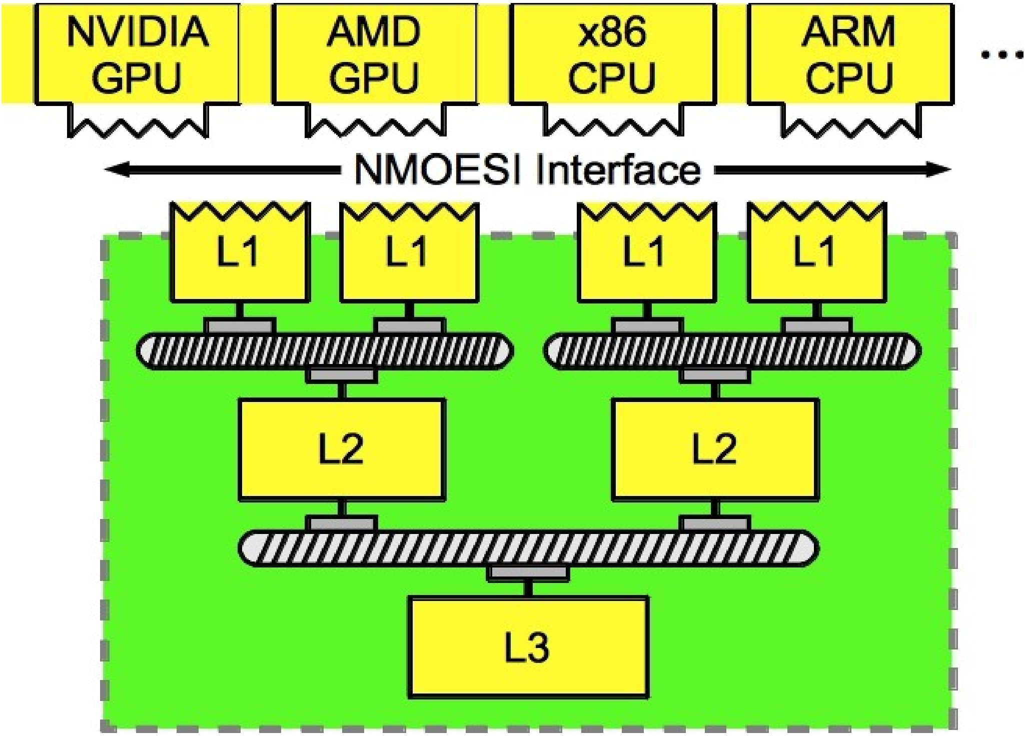
- Fully configurable DRAM and cache modules, based on a 7970 GPU by default
  - 16KB data L1s (per compute unit)
  - Separate scalar L1s (shared by 4 compute units)
  - 6 banks of 128KB L2 (per GPU)
  - L1-to-L2 all-to-all crossbar
  - L2s to DRAM modules

- Cache hiearchy based on 3-state protocol (NSI)
  - N is non-exclusive, modified (similar to Delayed Consistency)

Southern Islands Simulation

Memory Hierarchy

- APU design is possible with caches sharing a single protocol (NMOESI)



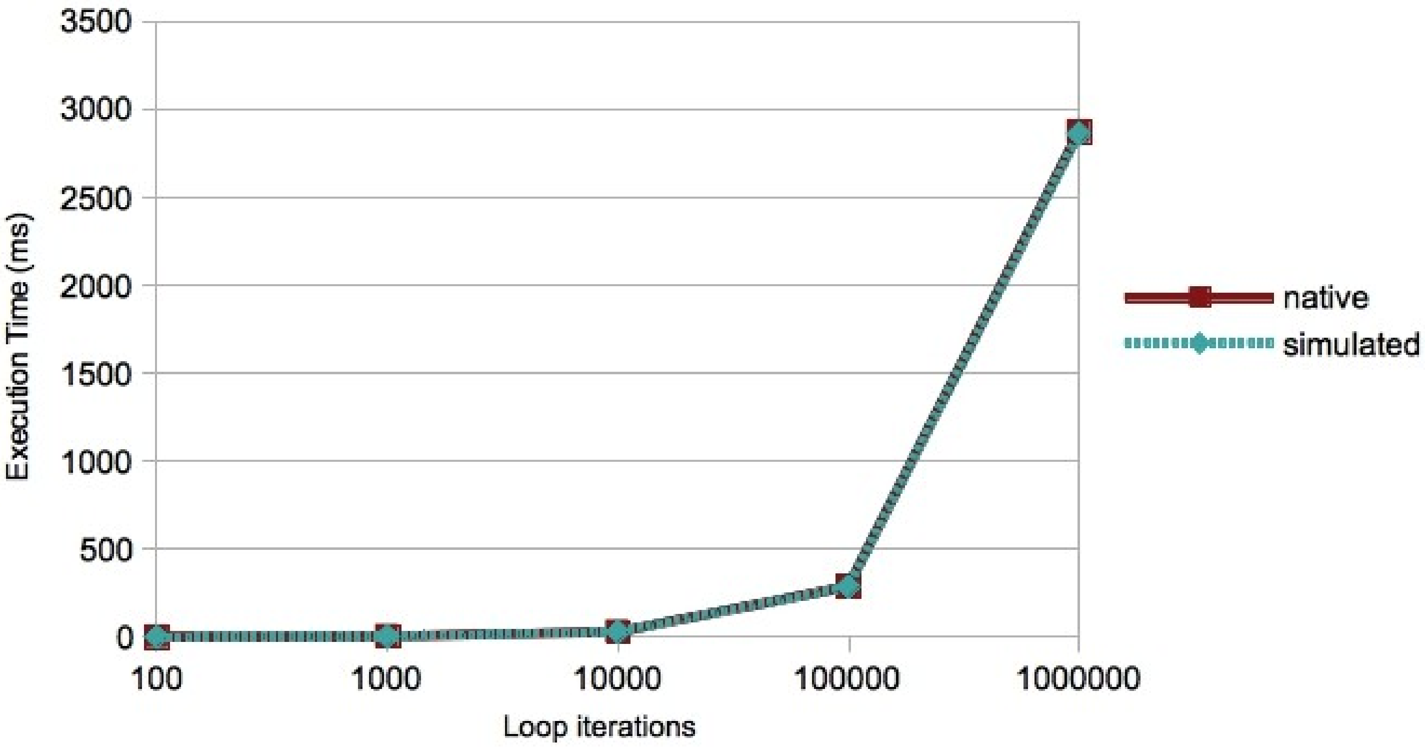
Validation Results

Methodology

- Single wavefront
  - Instruction scheduling
- Multiple wavefronts
  - Scheduling
  - Instruction issue
  - Resource sharing (e.g., SIMD unit)

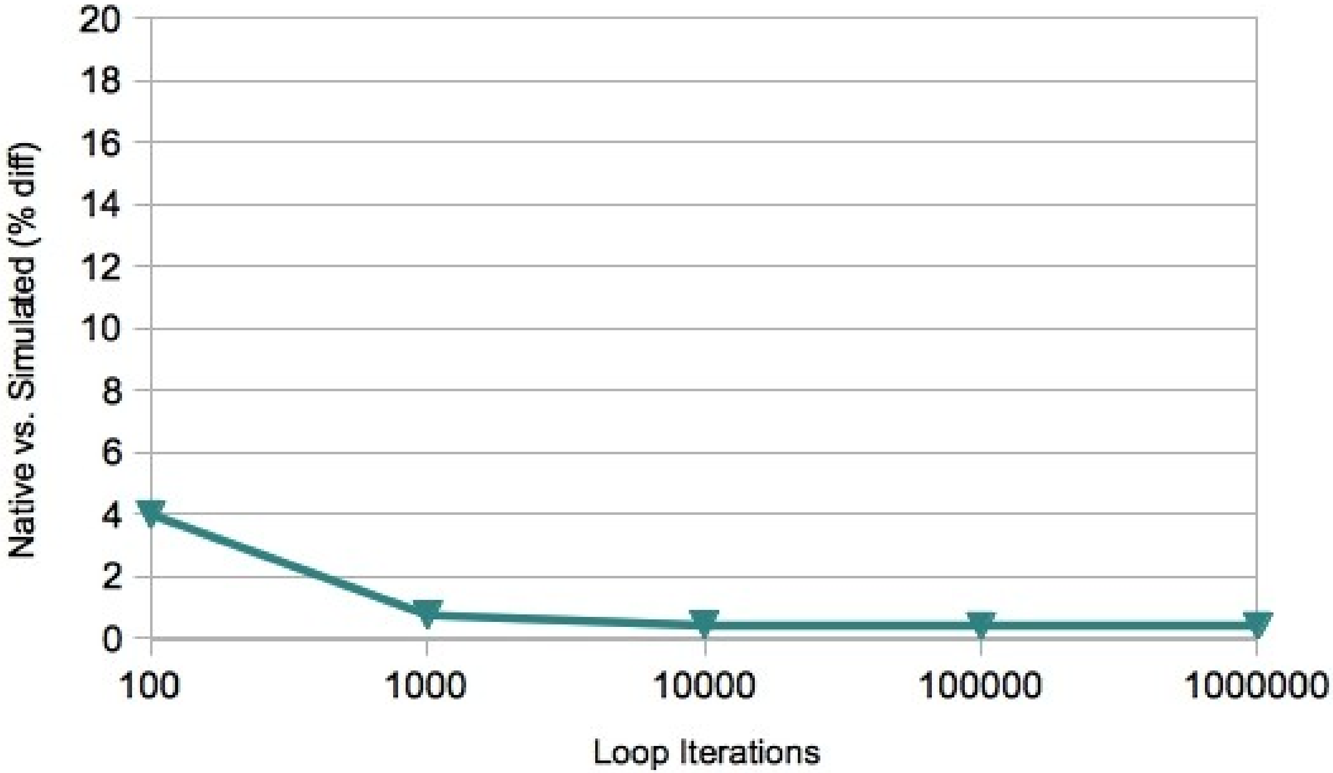
Validation Results

Single Wavefront



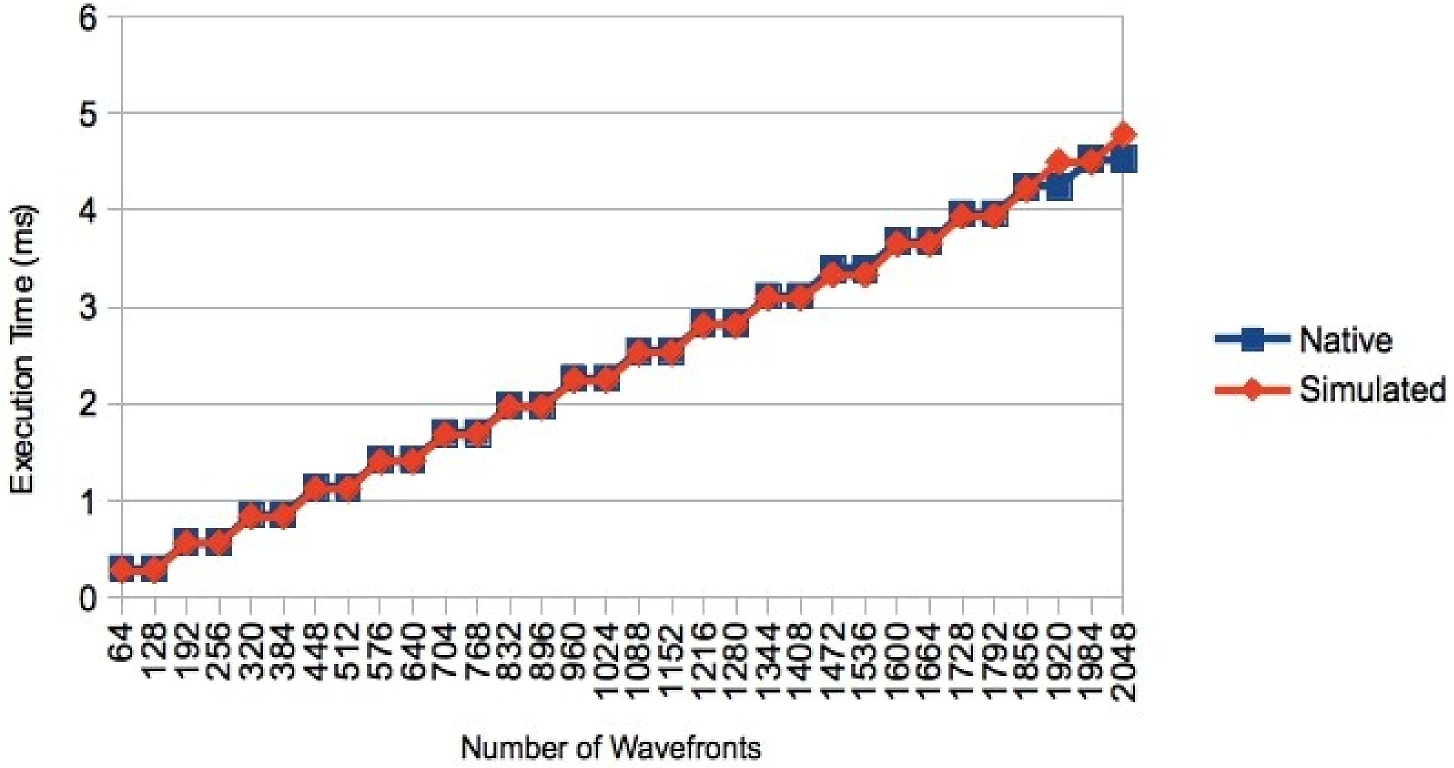
Validation Results

Single Wavefront



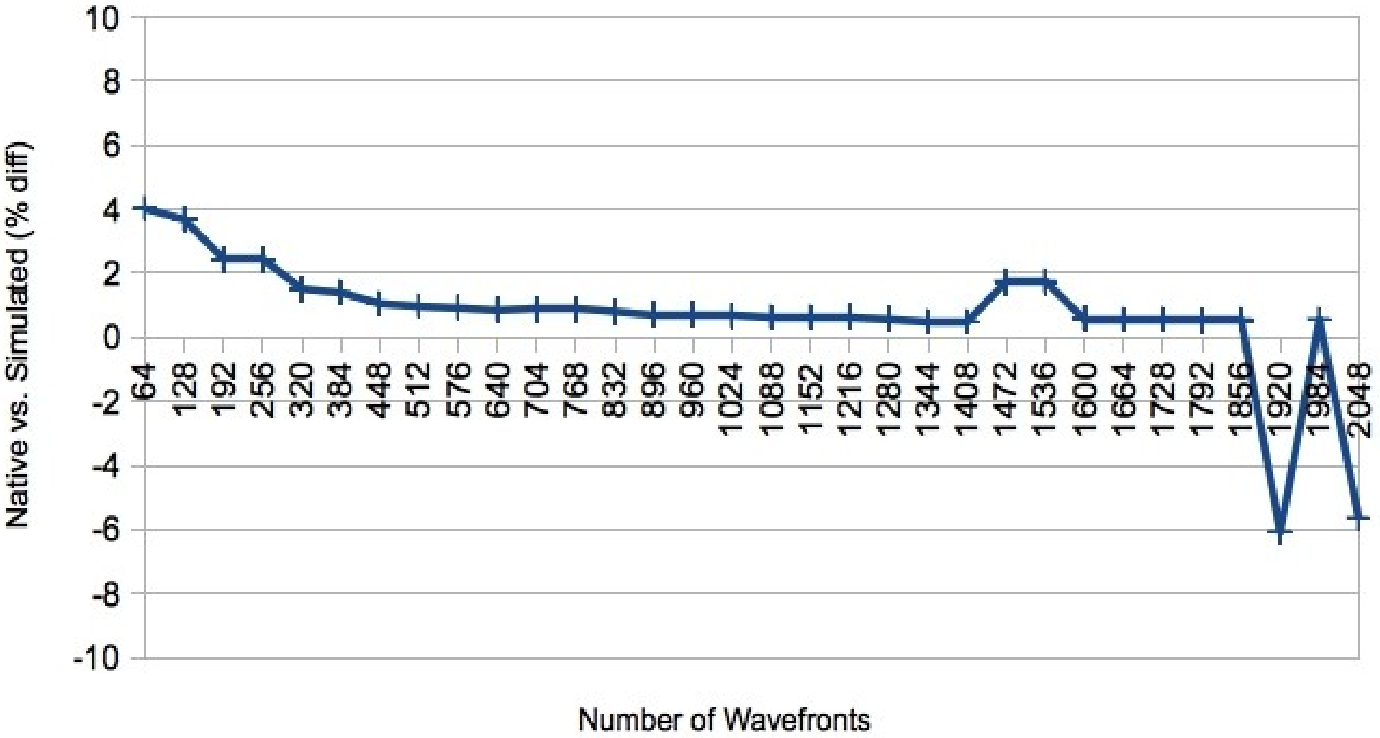
Validation Results

Multiple Wavefronts



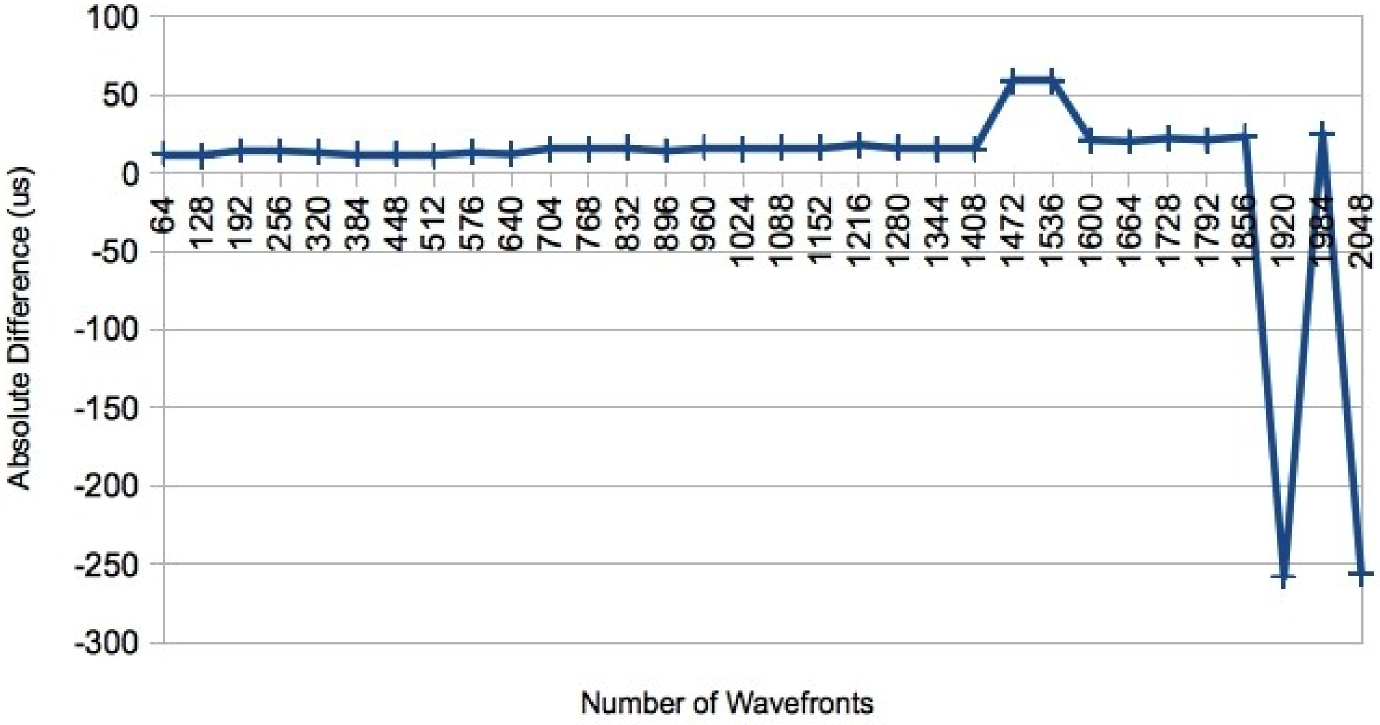
Validation Results

Multiple Wavefronts



Validation Results

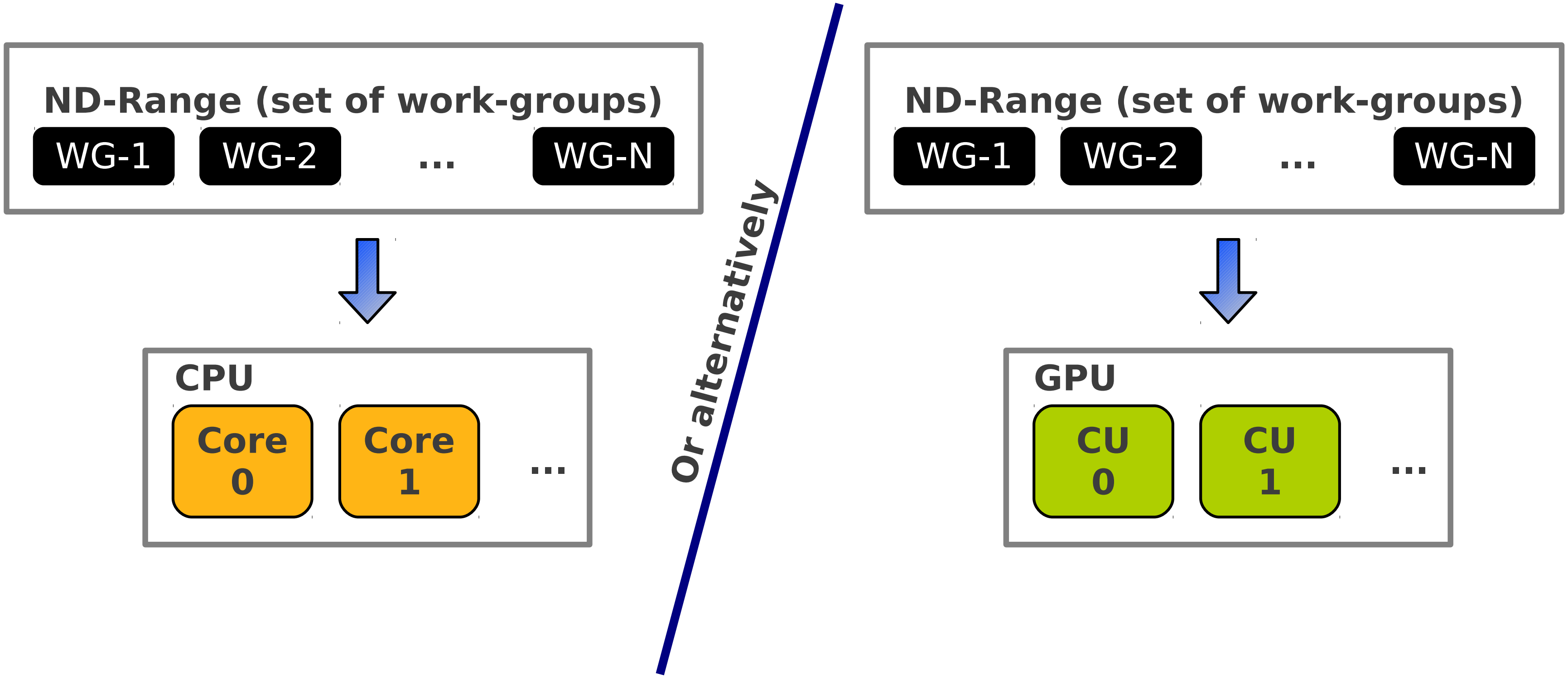
Multiple Wavefronts



Improving Heterogeneity

Current OpenCL model

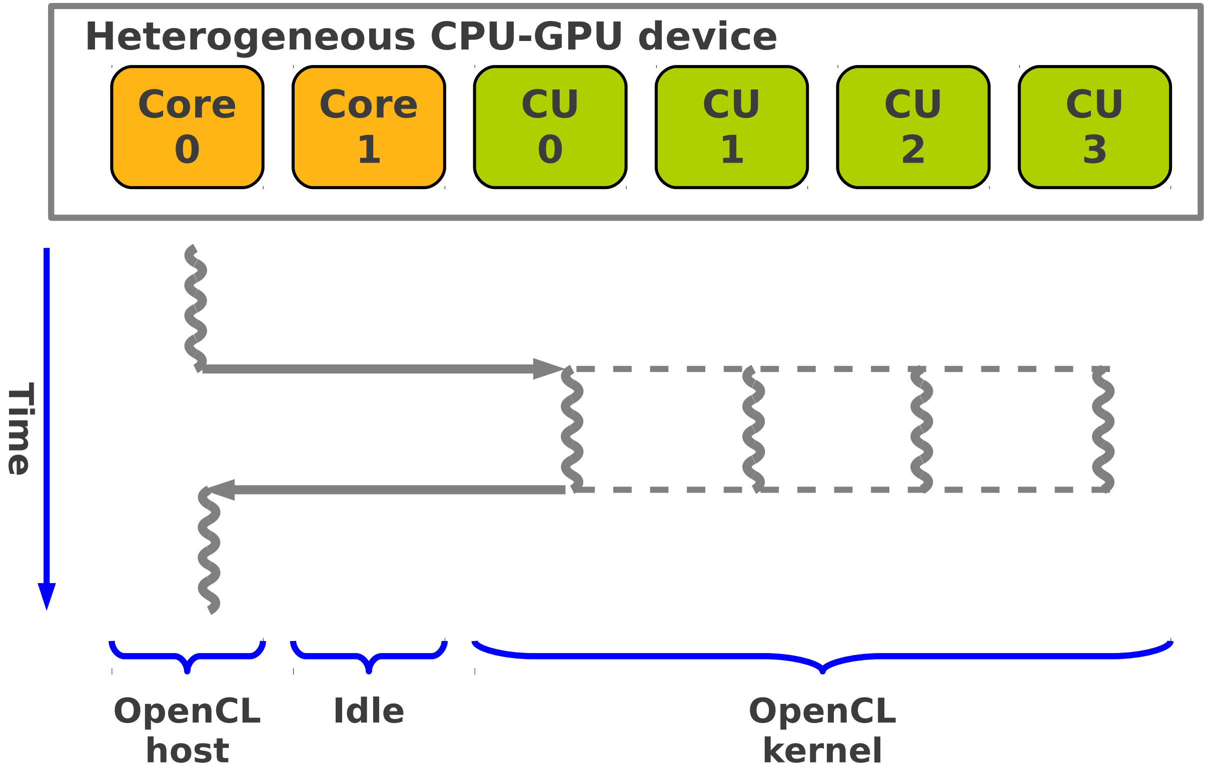
- An OpenCL ND-Range runs **entirely on one device**
- Scheduling done **manually** by programmer



Improving Heterogeneity

Current OpenCL model

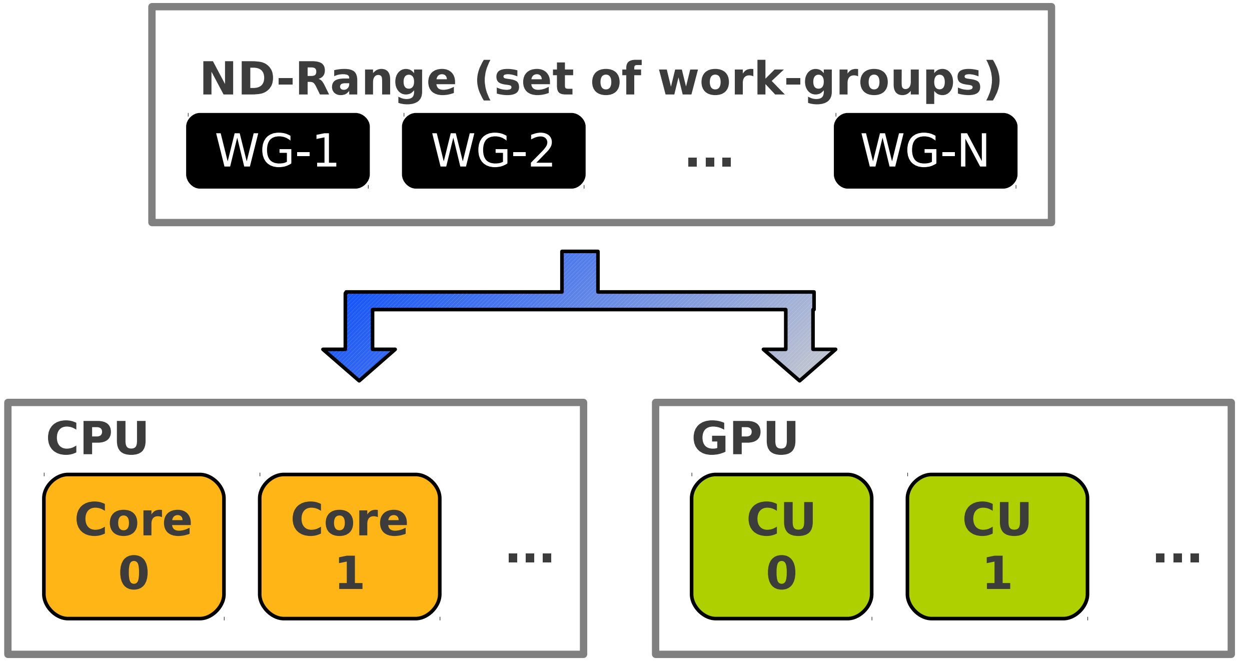
- Only GPU compute units run the OpenCL ND-Range
- **CPU cores stay idle**, unless programmer provides them with work



Improving Heterogeneity

Proposed Enhancement

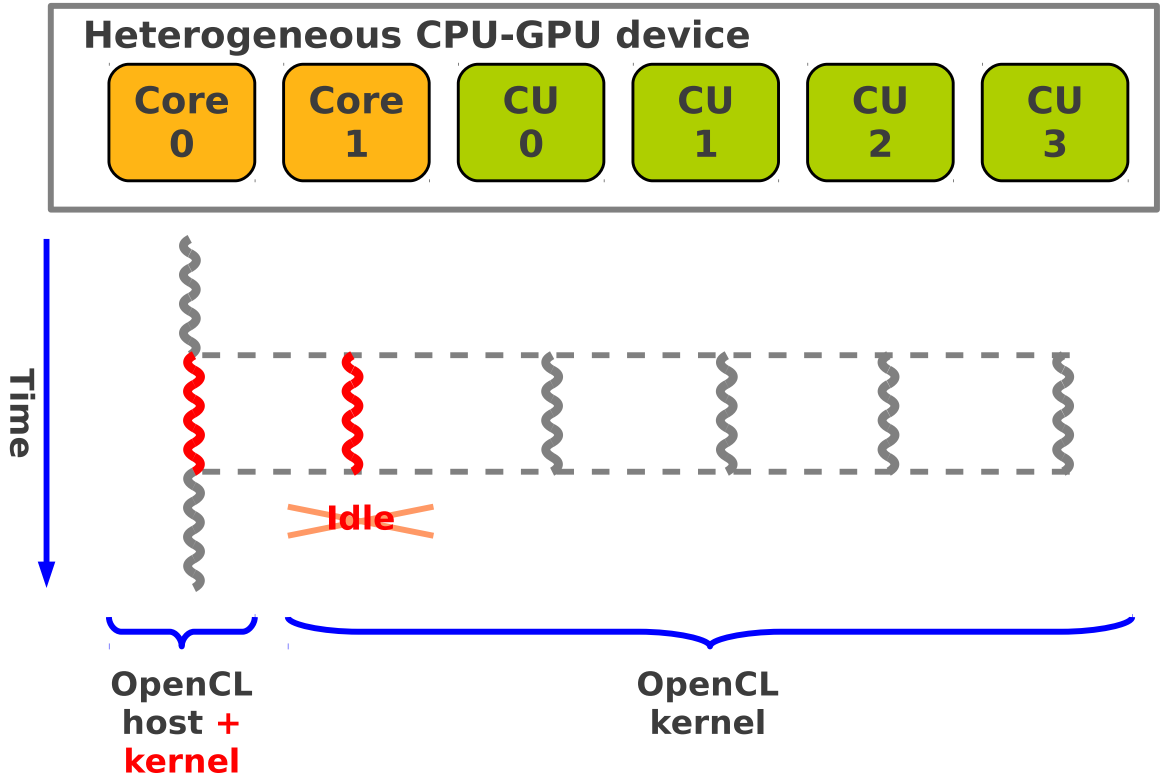
- Allow **multiple devices** to execute the same ND-Range
- **Automatic distribution** of work-groups by the runtime



Improving Heterogeneity

Proposed Enhancement

- All devices run the ND-Range (**higher resource utilization**)
- Complex cores contribute to **reduce the execution time**



Improving Heterogeneity

Proposed Enhancement

- Why hasn't this been done already?
  - Impractical with discrete GPU + CPU
    - Combining sparsely modified buffers from multiple distributed memories
    - Imbalance of processing power
- What's changed?
  - Low-power, shared memory CPU + GPU (i.e., APUs)
    - Removes challenge of combining results
    - Processors have more similar processing capability
- What are the benefits?
  - Programmer does not need to predict load ahead of time
  - The device better suited for execution will automatically run more work groups

Improving Heterogeneity

Proposed Enhancement

- Why Multi2Sim?
  - The complete tool-chain is implemented!

- OpenCL runtime implementation allows extensions to be added

- Device driver model allows scheduler to be implemented

- Memory and cache models allow
  - Coherent memory hierarchies
  - Common physical and virtual address spaces

- Emulator/simulator allows work groups to be processed individually instead of only ND-Ranges

Concluding Remarks

The Multi2Sim Community

The Multi2Sim Community

Additional Material

- The Multi2Sim Guide
  - Complete documentation of the simulator's user interface, simulation models, and additional tools.
- Multi2Sim forums and mailing list
  - New version releases and other important information is posted on the Multi2Sim mailing list (no spam, 1 email per month).
  - Users share question and knowledge on the website forum.
- M2S-Cluster
  - Automatic verification framework for Multi2Sim.
  - Based on a cluster of computers running *condor*.
- The Multi2Sim OpenCL compiler **– new!**
  - LLVM-based compiler for GPU kernels written in OpenCL C.
  - Front-ends for CUDA and OpenCL in progress.
  - Back-ends for Fermi, Kepler, and Southern Islands in progress.
  - Back-ends accessible through stand-alone assemblers.

The Multi2Sim Community

Academic Efforts at Northeastern

- The “GPU Programming and Architecture” course
  - We started an unofficial seminar that students can voluntarily attend. The **syllabus** covers OpenCL programming, GPU architecture, and state-of-the-art research topics on GPUs.
  - Average **attendance** of ~25 students per semester.

- Undergraduate directed studies
  - Official alternative equivalent to a **4-credit course** that an undergraduate student can optionally enroll in, collaborating in Multi2Sim development.

- Graduate-level development
  - Lots of research projects at the graduate level depend are based on Multi2Sim, and **selectively included** in the development trunk for public access.
  - Simulation of **OpenGL** pipelines, support for **new CPU/GPU** architectures, among others.

The Multi2Sim Community

Collaborating Research Groups

- Universidad Politécnica de Valencia
  - Pedro López, Salvador Petit, Julio Sahuquillo, José Duato.
- Northeastern University
  - Chris Barton, Shu Chen, Zhongliang Chen, Tahir Diop, Xiang Gong, David Kaeli, Nicholas Materise, Perhaad Mistry, Dana Schaa, Rafael Ubal, Mark Wilkening, Ang Shen, Tushar Swamy, Amir Ziabari.
- University of Mississippi
  - Byunghyun Jang
- NVIDIA
  - Norm Rubin
- University of Toronto
  - Jason Anderson, Natalie Enright, Steven Gurfinkel, Tahir Diop.
- University of Texas
  - Rustam Miftakhutdinov

The Multi2Sim Community

Multi2Sim Academic Publications

- Conference papers
  - *Multi2Sim: A Simulation Framework to Evaluate Multicore-Multithreaded Processors*, SBAC-PAD, 2007.
  - *The Multi2Sim Simulation Framework: A CPU-GPU Model for Heterogeneous Computing*, PACT, 2012.

- Tutorials
  - *The Multi2Sim Simulation Framework: A CPU-GPU Model for Heterogeneous Computing*, PACT, 2011.
  - *Programming and Simulating Fused Devices — OpenCL and Multi2Sim*, ICPE, 2012.
  - *Multi-Architecture ISA-Level Simulation of OpenCL*, IWOCL, 2013.
  - *Simulation of OpenCL and APUs on Multi2Sim*, ISCA, 2013. ← **Upcoming!**

The Multi2Sim Community

Published Academic Works Using Multi2Sim

- Recent
  - R. Miftakhutdinov, E. Ebrahimi, Y. Patt, *Predicting Performance Impact of DVFS for Realistic Memory Systems*, MICRO, 2012.
  - D. Lustig, M. Martonosi, *Reducing GPU Offload Latency via Fine-Grained CPU-GPU Synchronization*, HPCA, 2013.

- Other
  - H. Calborean, R. Jahr, T. Ungerer, L. Vintan, *A Comparison of Multi-objective Algorithms for the Automatic Design Space Exploration of a Superscalar System*, Advances in Intelligent Systems and Computing, vol. 187.
  - X. Li, C. Wang, X. Zhou, Z. Zhu, *Cache Promotion Policy Using Re-reference Interval Prediction*, CLUSTER, 2012.

  - ... and 62 more citations, as per Google Scholar.

Multi2Sim 4.1

Multi-Architecture ISA-Level Simulation of OpenCL

Dana Schaa, Rafael Ubal

Northeastern University
Boston, MA