A Framework for Visualization of OpenCL Applications Execution

A. Ziabari*, R. Ubal*, D. Schaa**, D. Kaeli*

*NUCAR Group, Northeastern Universiy
**AMD

Outline

Introduction
Simulation methodology

Part 1 – Visualization of CPU and GPU Pipelines

Emulation of an x86 CPU
Timing simulation of an x86 CPU
OpenCL on the host
OpenCL on the device
The AMD Southern Islands ISA
The GPU Compute Pipelines
Visual Tool

Part 2 – Visualization of Memory System and Interconnects

Memory system
Interconnection network

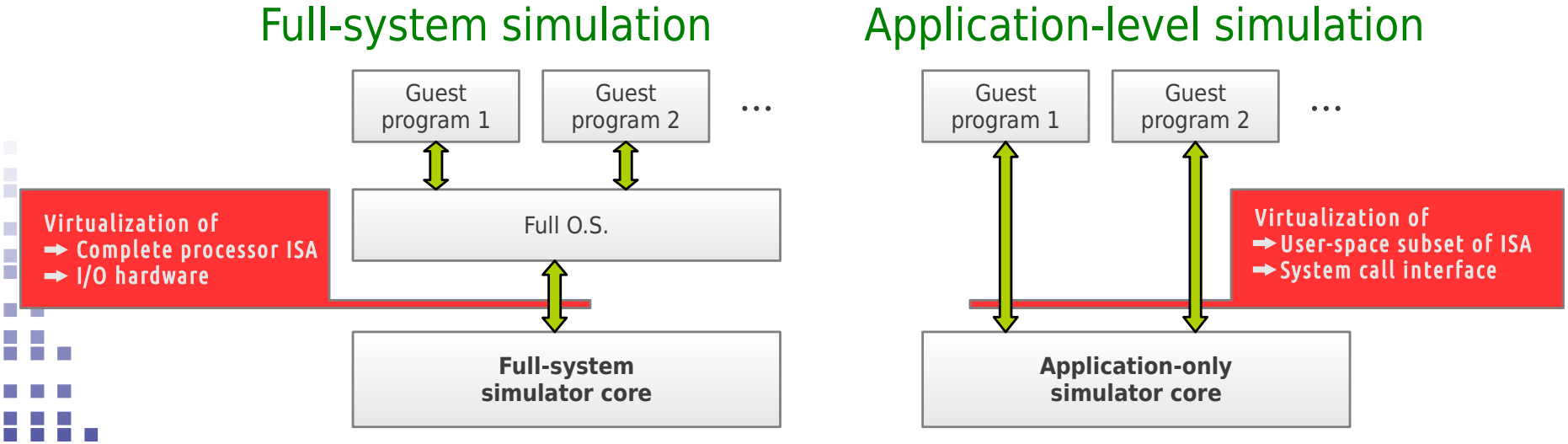
Part 3 – Ongoing work

New architectures and benchmarks
HSA architecture
Compilation support

Introduction

The Multi2Sim Simulation Framework

- Simulation framework for heterogeneous CPU-GPU systems.
- Support for CPU architectures: x86, ARM, MIPS
- Support for GPU architectures: AMD Evergreen, AMD Southern Islands, NVIDIA Kepler, HSA intermediate language
- Application-level simulation



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 >
```

Introduction

Simulator Input/Output Files

- Example of INI file format

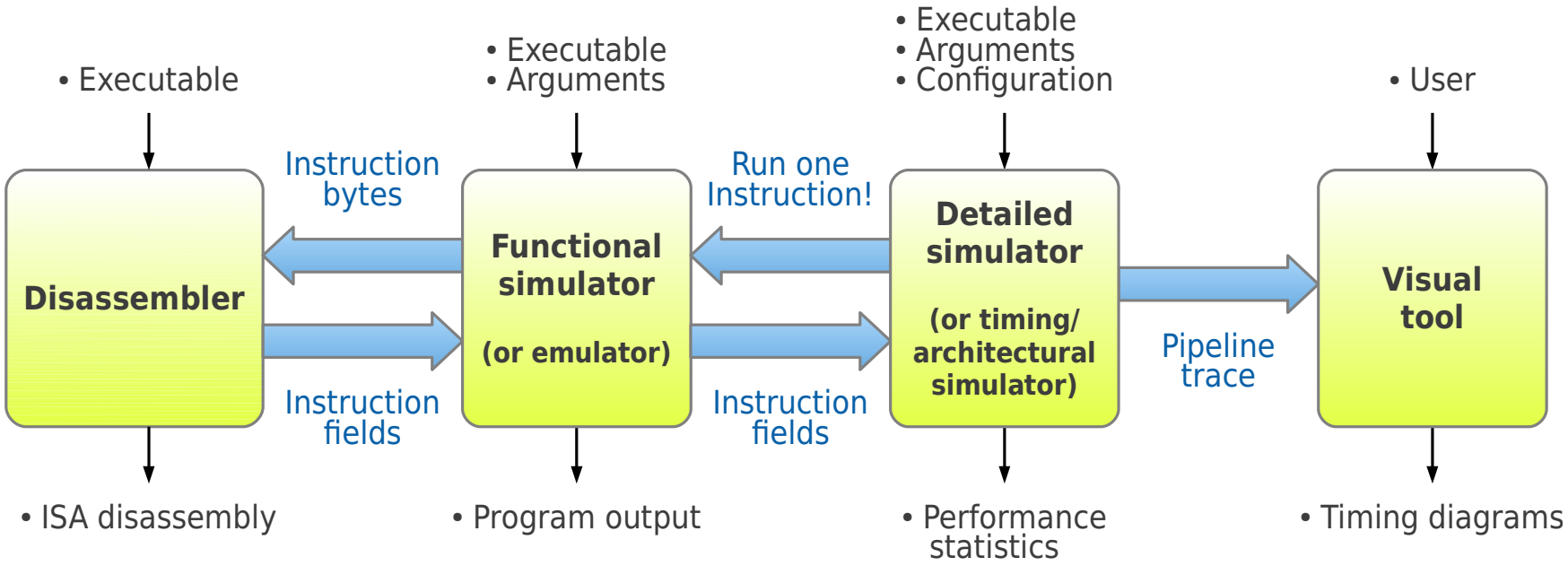
```
[ Context 0 ]
Exe = test-threads
Args = 10

[ Context 1 ]
Exe = test-args
Args = a b c
```

- Uses of INI files for
  - Configuration files.
  - Output statistics.
  - Statistic summary in standard error output.

Simulation Methodology

Four-Stage Simulation Process



- Four isolated software modules for each architecture (x86, MIPS, ...)
- Each module has a command-line interface for stand-alone execution, or an API for interaction with other modules.

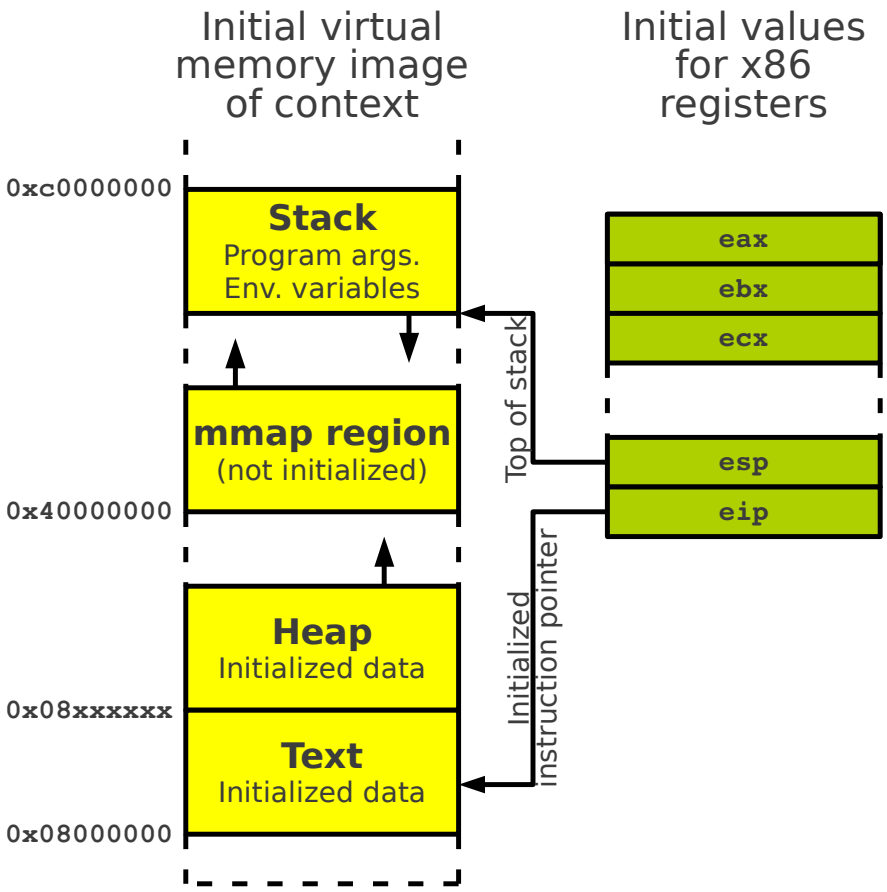
Part 1

Visualization of CPU and GPU Pipelines

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

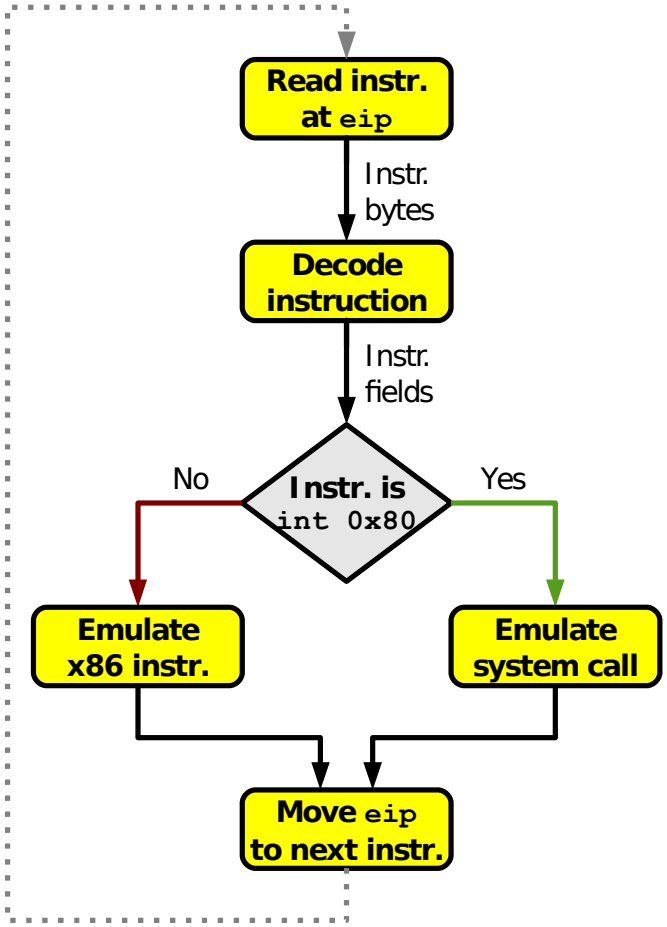
- Program headers.
- Arguments.
- Environment variables.

3) Initialize registers

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

Emulation of an x86 CPU

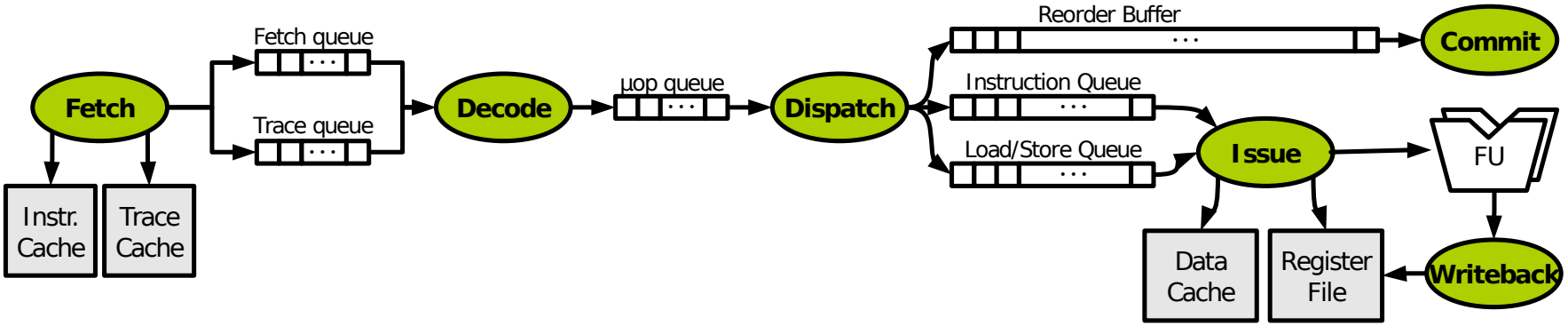
Emulation Loop



- 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)*

Timing Simulation of an x86 CPU

Superscalar Processor



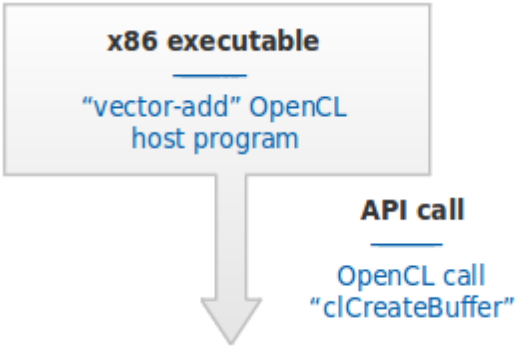
- Superscalar x86 pipelines
  - 6-stage pipeline with configurable latencies.
  - Support for multithreading
  - Support for multicore processors

OpenCL on the Host

The OpenCL CPU Host Program

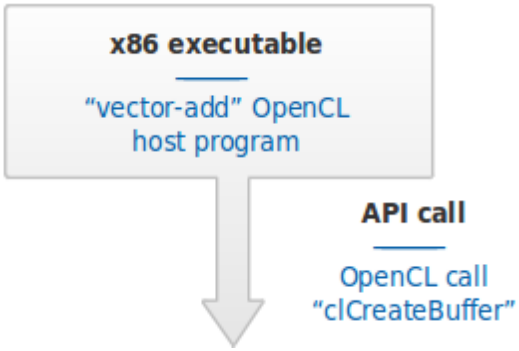
Native

An x86 OpenCL host program performs an OpenCL API call.



Multi2Sim

Same

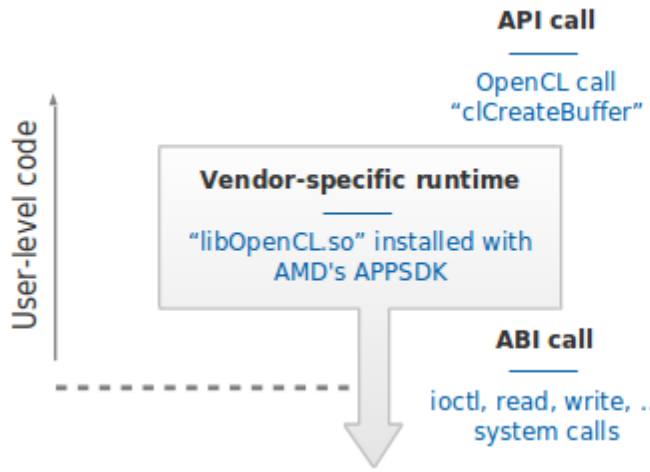


OpenCL on the Host

The OpenCL Runtime Library

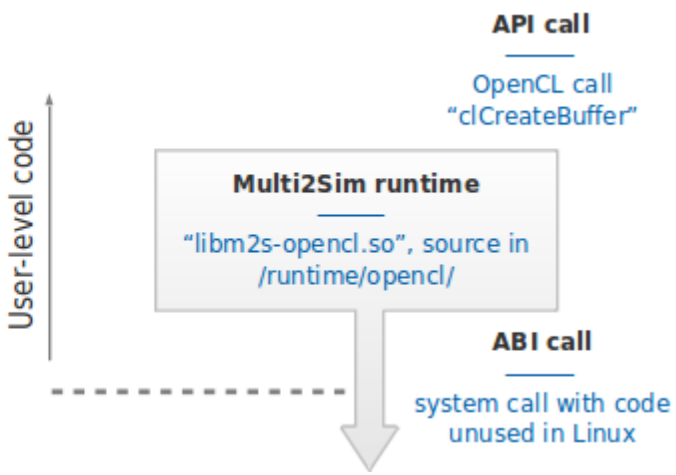
Native

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.



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.

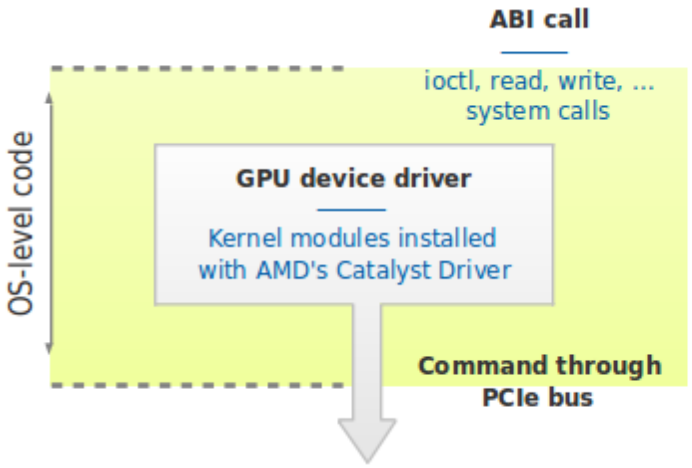


OpenCL on the Host

The OpenCL Device Driver

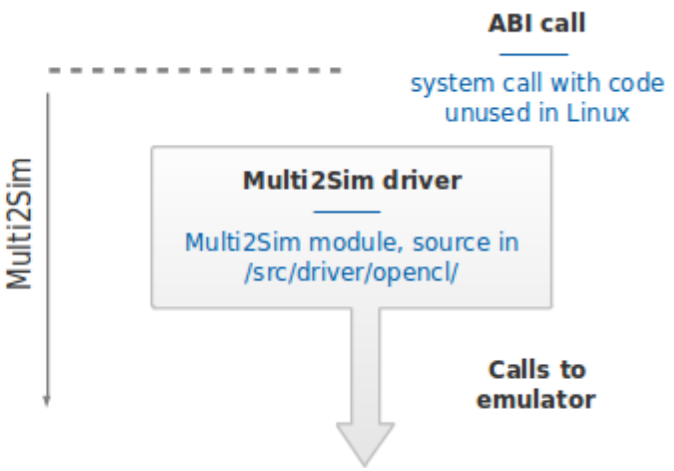
Native

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.

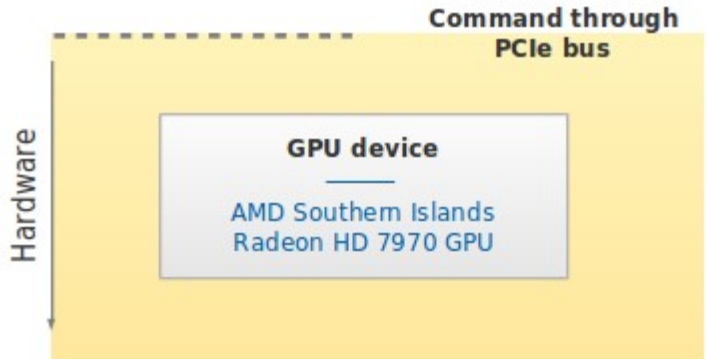


OpenCL on the Host

The GPU Emulator

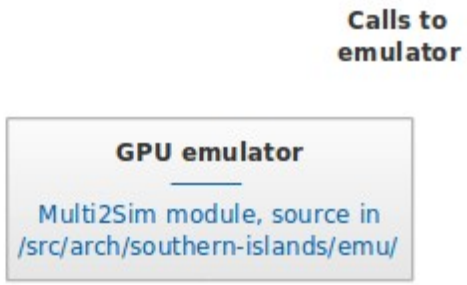
Native

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

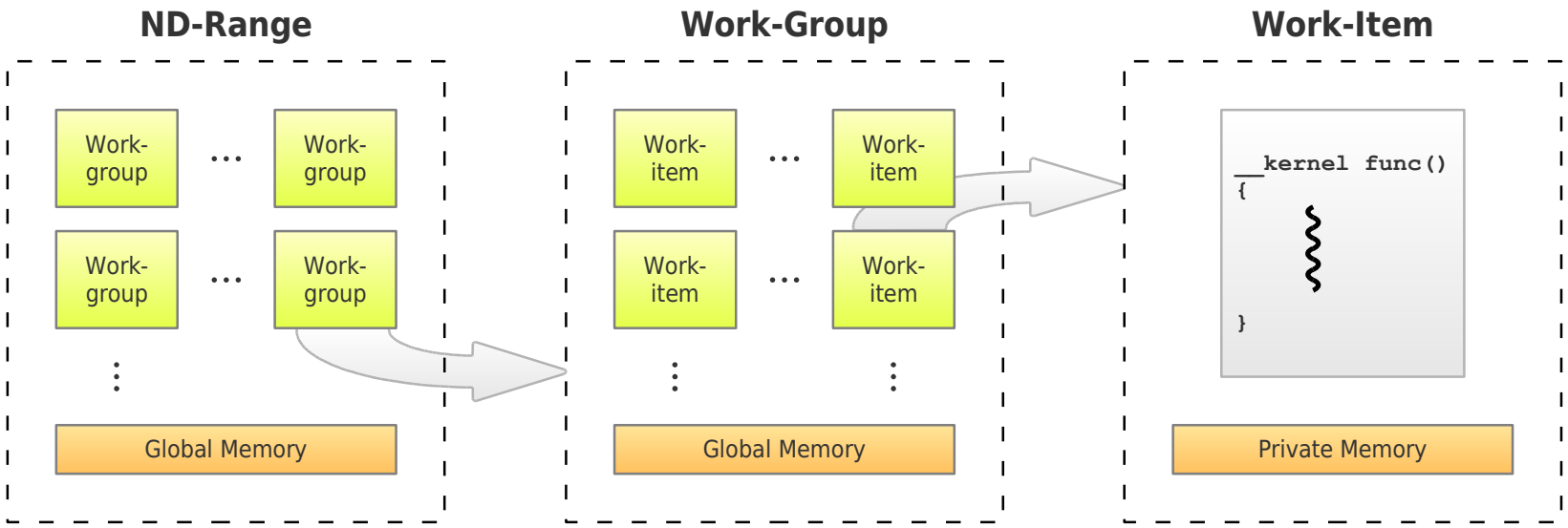
Transferring Control

- 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.

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.



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];
}
```

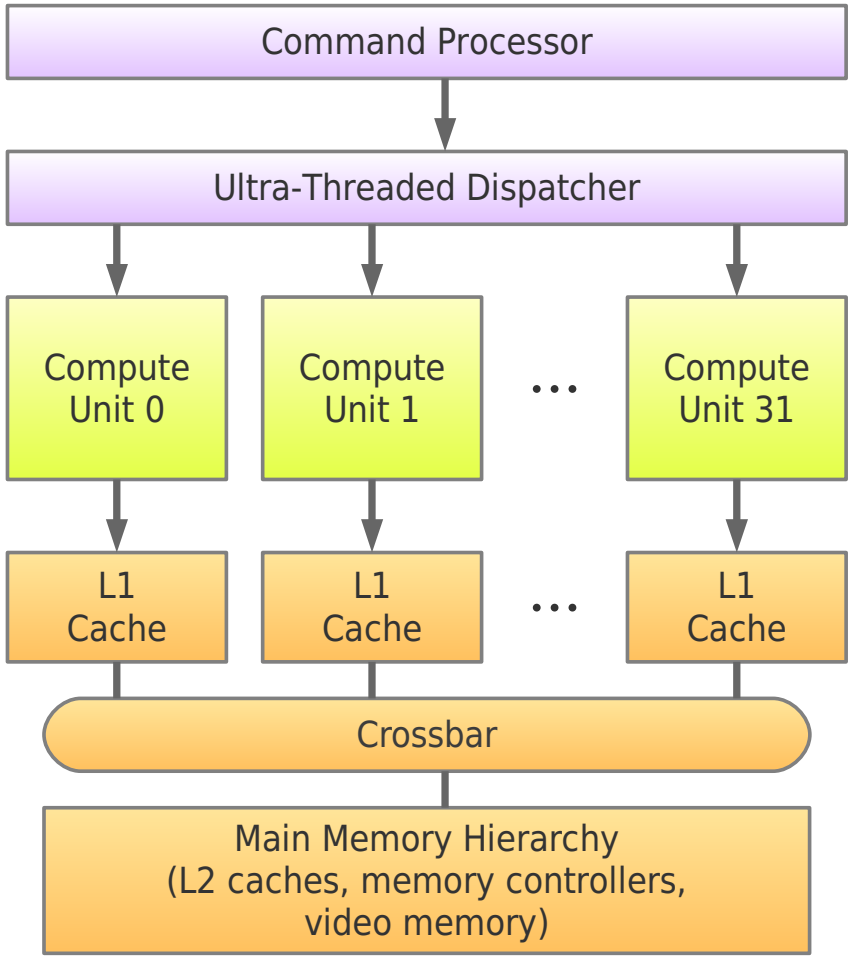
The Southern Islands ISA

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 GPU Compute Pipelines

Compute Device

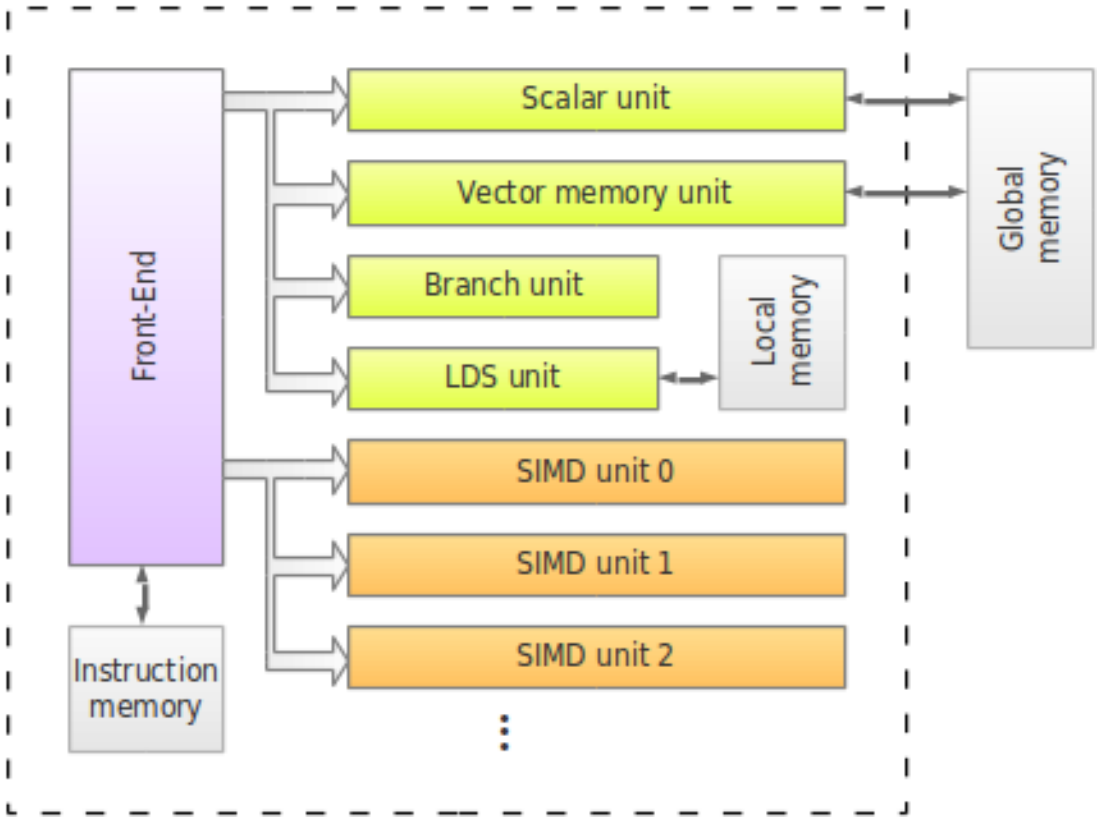


- A **command processor** receives commands and data from the CPU.
- A **dispatcher** splits the ND-Range in work-groups and sends them into the compute units.
- A set of **compute units** runs work-groups.
- A **memory hierarchy** serves global memory accesses

The GPU Compute Pipelines

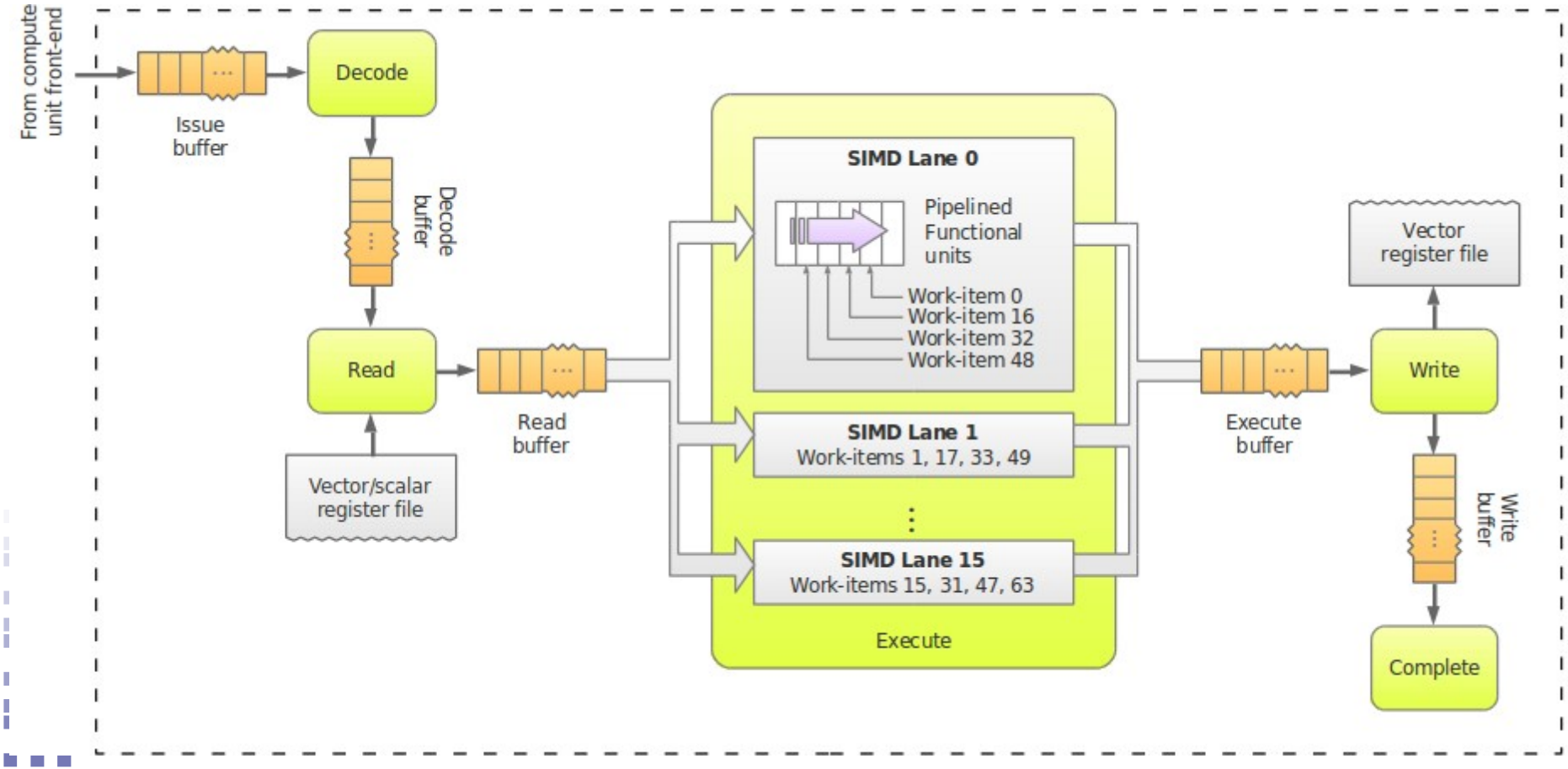
Compute Unit

- 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.



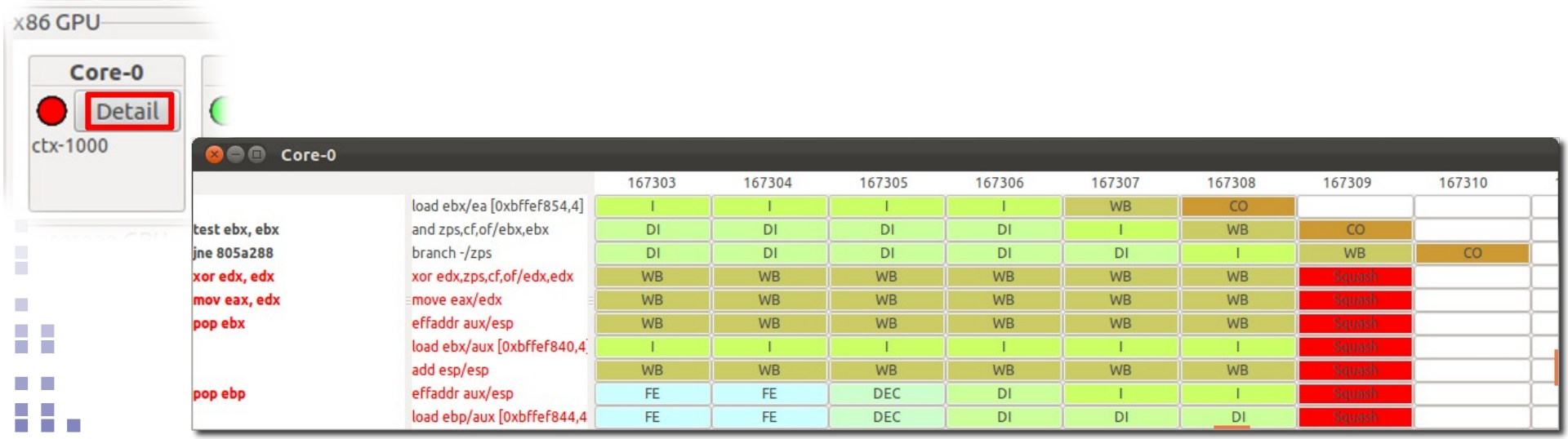
The GPU Compute Pipelines

The SIMD Unit



Visual Tool

- 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.



Demo 1

Part 2

Visualization of Memory System and Interconnects

The Memory Hierarchy

Configuration

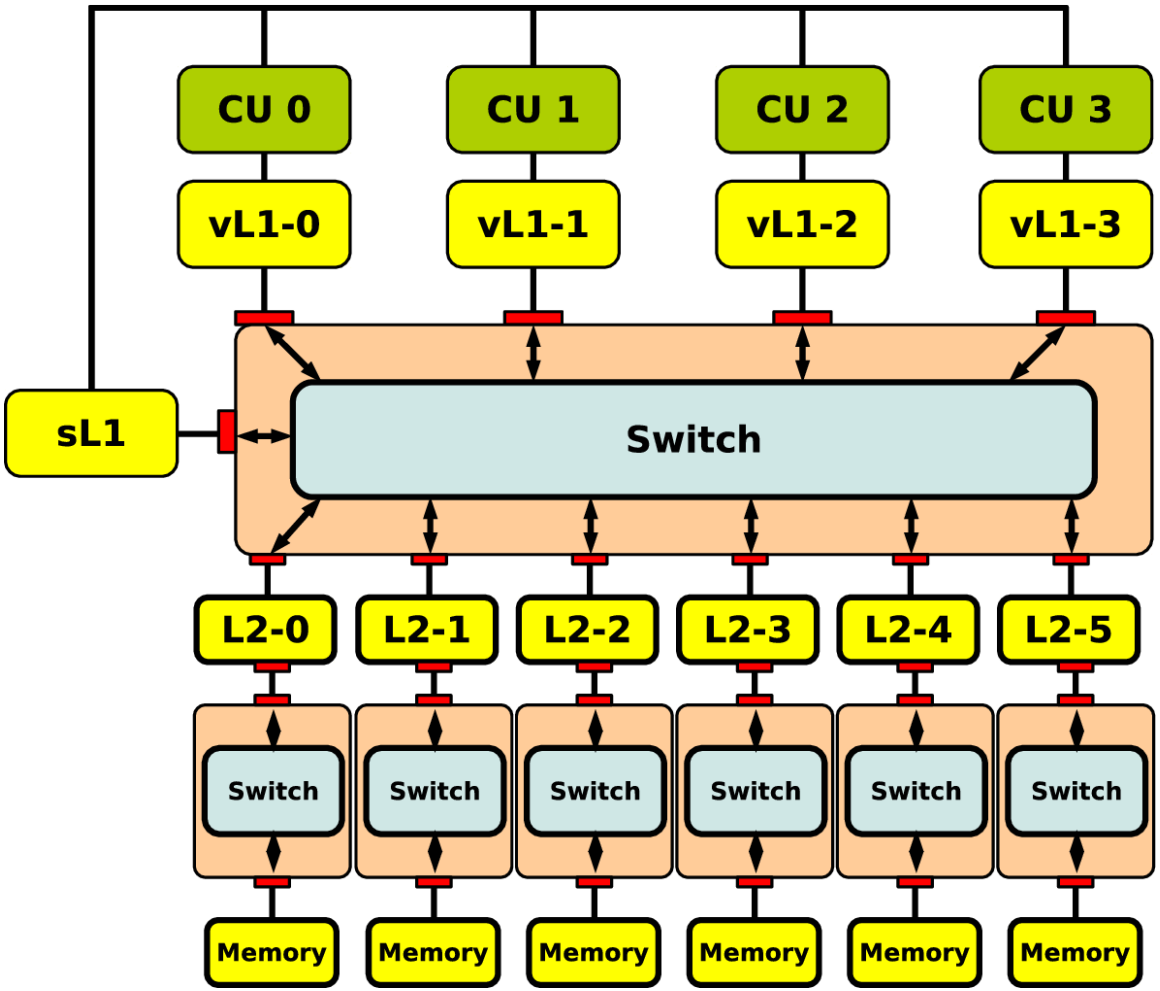
- Flexible hierarchies
  - **Any number of caches** organized in any number of levels.
  - Cache levels connected through default switch cross-bar interconnects, or complex **custom interconnect** configurations.
  - Each architecture undergoing a timing simulation specifies its own **entry point** (cache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

The Memory Hierarchy

Memory Example

Example 1

- Four compute units with private vector L1 caches and a shared scalar L1 cache.
- Memory address space is split into 6 partition, each connected to its dedicated L2 Cache.
- Default crossbar switch interconnects are used.



The Memory Hierarchy

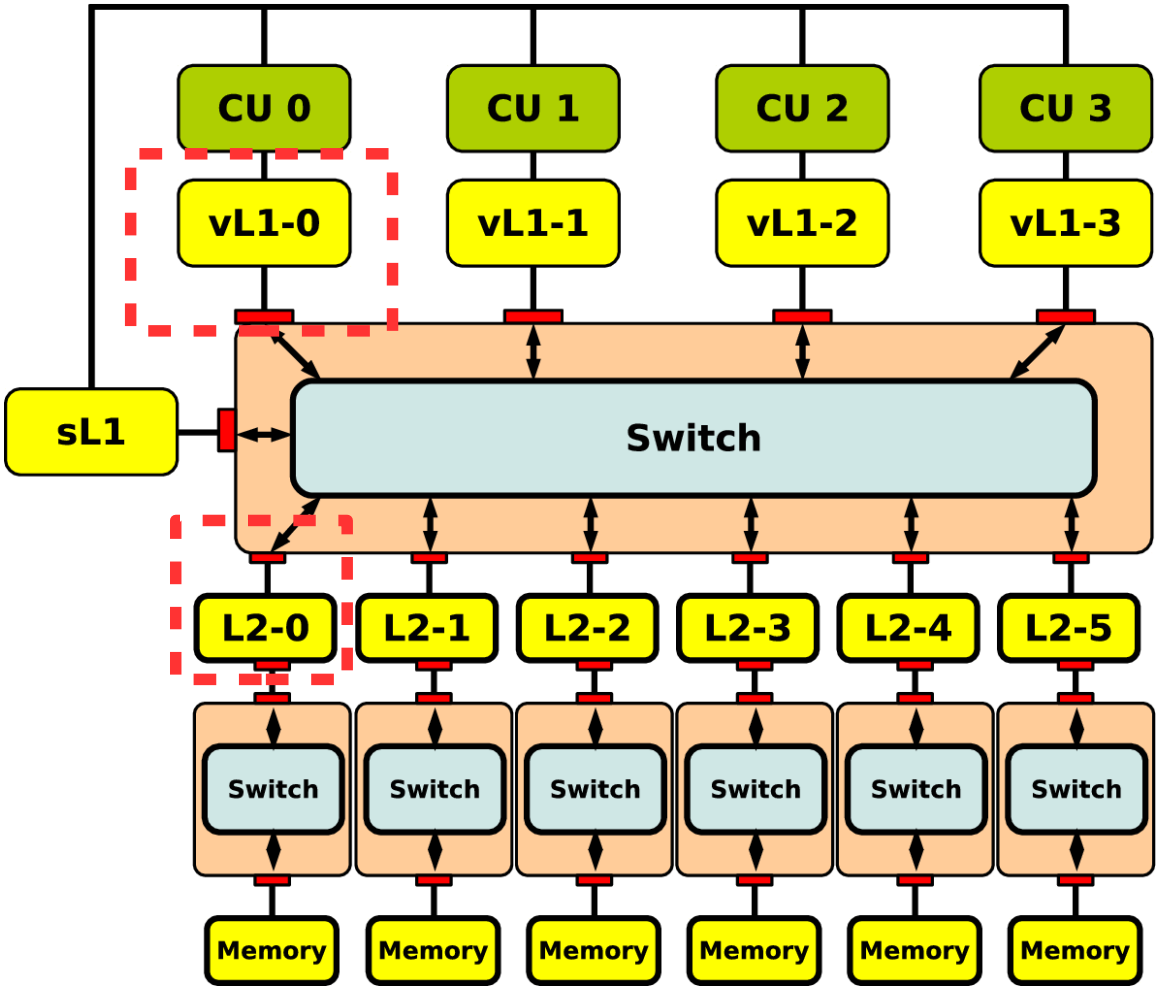
Memory Example

- Example of INI file to configure cache modules

```
[CacheGeometry A]
Sets = ...
Assoc = ...
BlockSize = ...
Latency = ...

[Module vL1-0]
Type = Cache
Geometry = A
LowNetwork = <networkName>
LowNetworkNode = ...
LowModules = l2-0 l2-1 …

[Entry CU-0]
Arch = SouthernIslands
ComputeUnit = 0
DataModule = vL1-0
ConstantDataModule = sL1
```



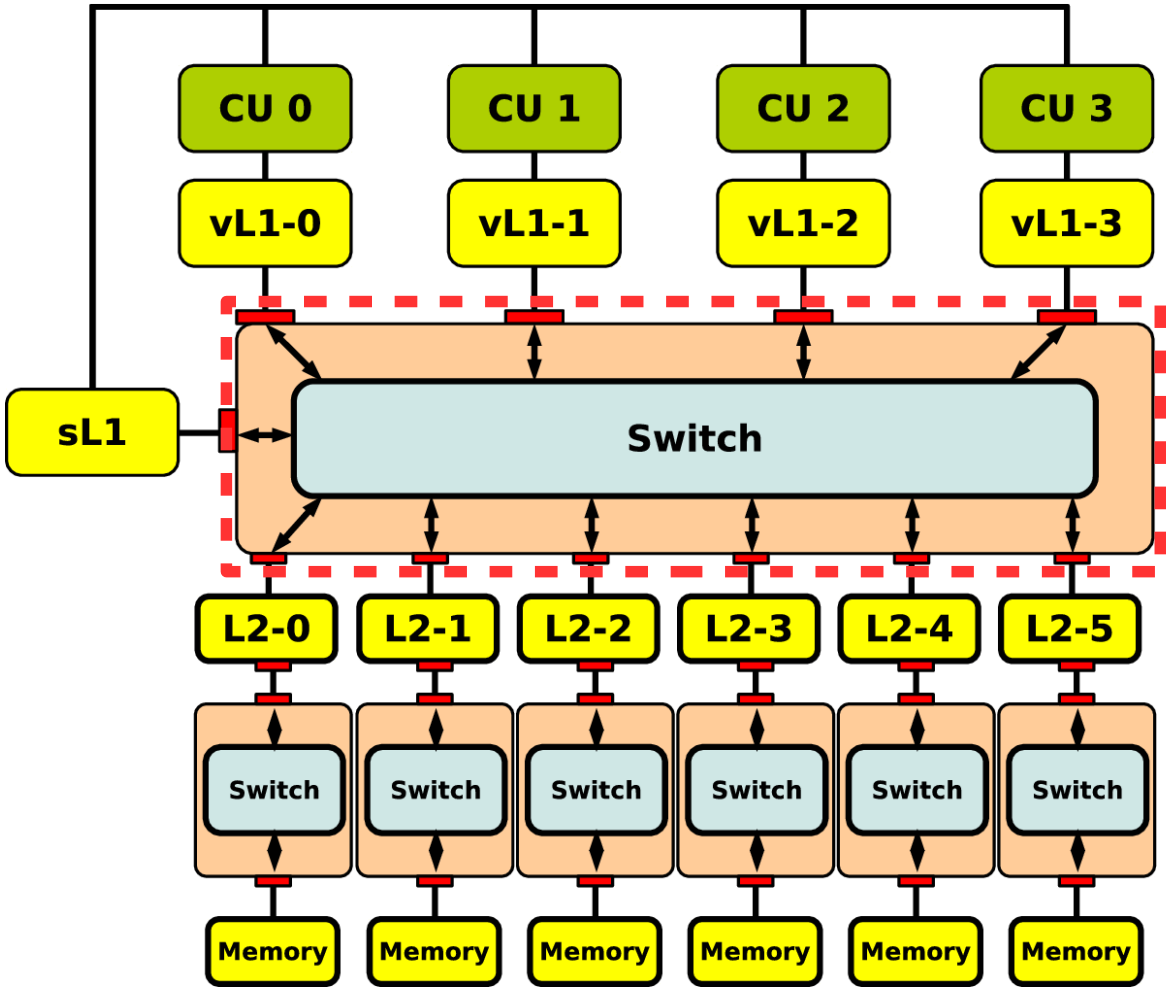
Demo 1

Interconnection Network

Network Examples

Example 2

- Same architecture as Example 1.
- Replacing default crossbar switch interconnect between L1 and L2 caches with its customizable version.
- Separate network configuration file activates the visualization of the network.



Interconnection Network

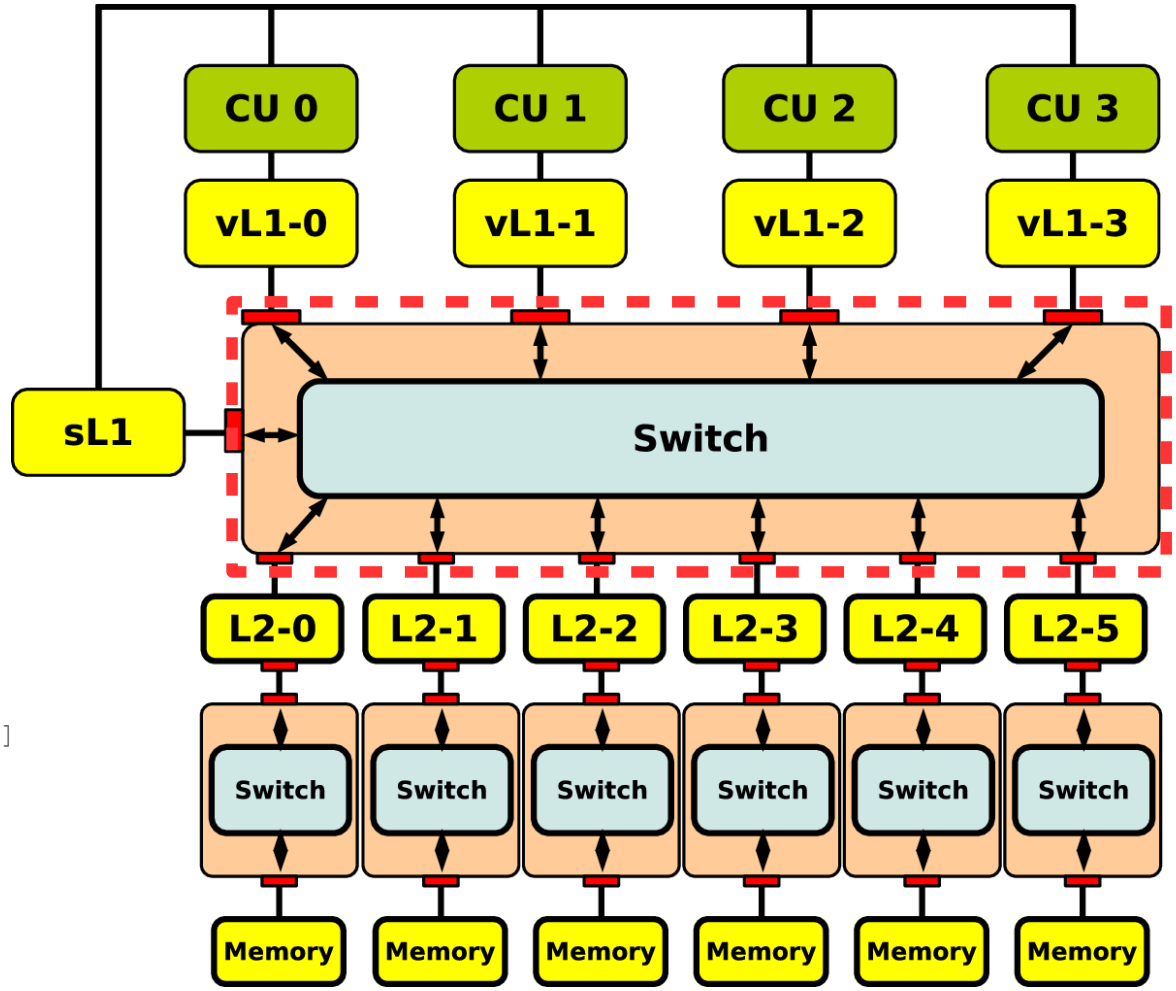
Network Examples

- Example of INI file to configure network modules

```
[Network.<name>]
DefaultInputBufferSize = ...
DefaultOutputBufferSize = ...
DefaultBandwidth = ...
DefaultPacketSize = …

[Network.<name>.Node.l1v0]
Type = EndNode
[Network.<name>.Node.Switch]
Type = Switch
Bandwidth = ...

[Network.<name>.Link.<linkName>]
Type = Bi/Uni-Directional
Bandwidth = ...
Source = ...
Dest = ...
InputBufferSize = …
…
```



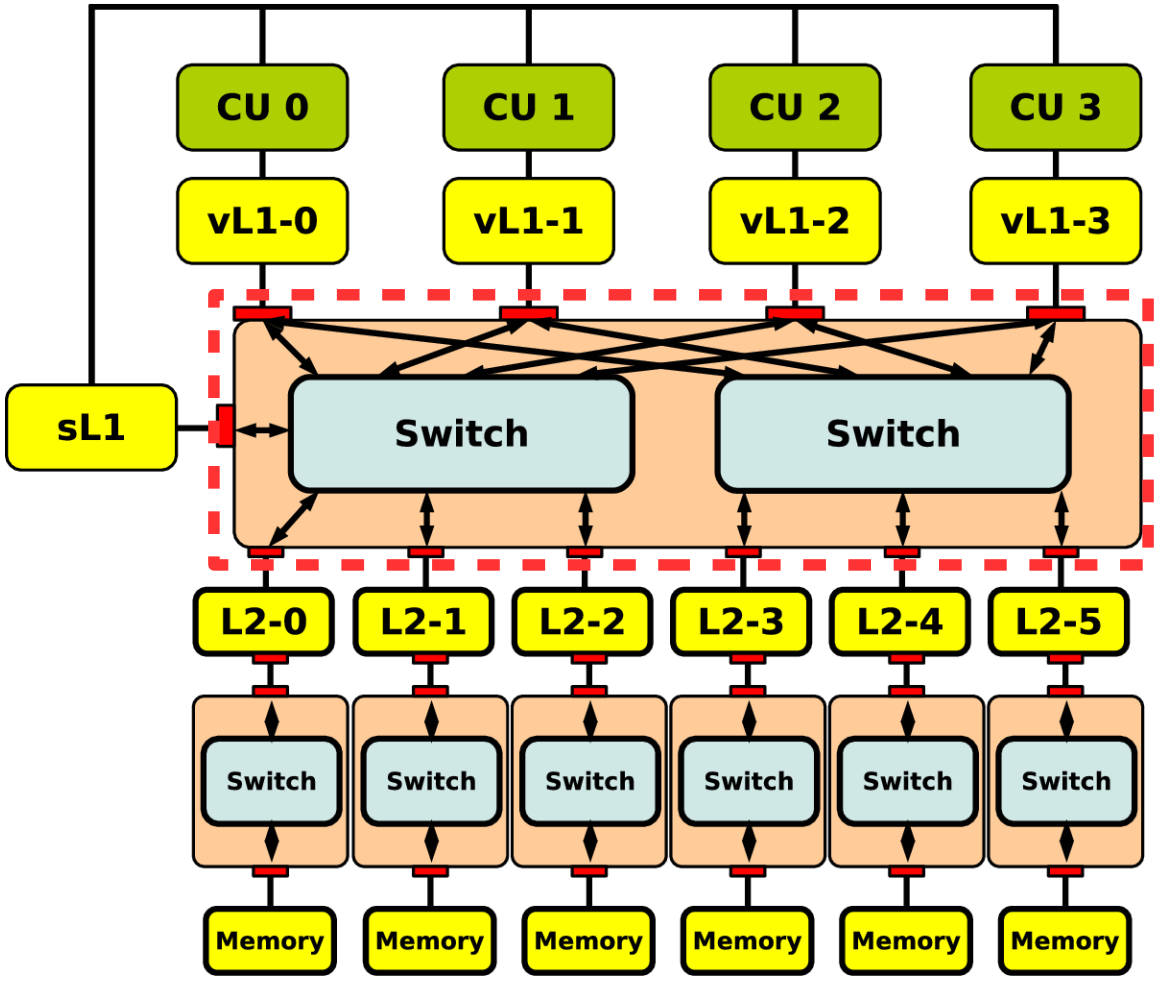
Demo 2

Interconnection Network

Customization of Network

Example 3

- Same architecture as Previous examples.
- Using a customized interconnect with two switches.



Demo 3

Interconnection Network

Customization of Network

- Cycle-by-cycle analysis
  - Identifying contention points in the network when executing OpenCL application
- Overall Link Utilization static graph
  - Efficiency of the network in the GPU architecture

Note: Network analysis with synthetic network is also possible.

Demo 3

Memory Snapshot

- Visualizing the memory access pattern of the OpenCL workload
  - **Identifying temporal and spatial locality:** Exploring Prefetching in GPU designs
  - **Identifying scattered or non-recurring accesses:** Exploring cache bypassing in GPU designs
  - **Identifying patterns in loads and stores:** Exploring memory hierarchy and interconnection network design

Demo 4

Network Snapshot

- Sampling network traffic
  - **Identifying network bottlenecks** in the OpenCL application execution
  - **Finding traffic patterns** in the execution of the application.
    - Helps in implementing run-time management techniques.

Demo 5

Part 3

Ongoing Work

Simulation Support

Supported Architectures

	Disasm.	Emulation	Timing Simulation	Graphic Pipelines
ARM	x	In progress	-	-
MIPS	x	In progress	-	-
x86	x	x	x	x
AMD Evergreen	x	x	x	x
AMD Southern Islands	x	x	x	x
NVIDIA Fermi	x	x	-	-
NVIDIA Kepler	x	In progress	-	-
HSA Intermediate Language	x	**x**	-	-

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

Simulation Support

Heterogeneous Software Architecture (HSA)

- HSA IL simulation
  - Emulation of binaries produced by the official HSA compiler (*BRIG* files)
- Debugger
  - Step-by-step execution of HSAIL binaries
  - Inspection of program state (memory and registers)
- Profiler
  - Profiler API specification currently being discussed by HSA
  - Multi2Sim will adapt to the agreed performance counters
- Benchmark suite
  - HPC benchmarks
  - Mobile/embedded benchmarks
  - Big data benchmarks

Simulation Support

Multi2C: the Multi2Sim Compiler

- LLVM to Southern Islands back-end
  - Instruction-level translation from LLVM to Southern Islands ISA
  - Register allocation
  - Vector-to-scalar optimization pass
- Southern Islands assembler
  - Definition of assembly code format with assembler directives
  - Generation of binary-compatible kernel executables with AMD GPUs

The Multi2Sim Community

Publications

- Main Multi2Sim Publications
  - R. Ubal, J. Sahuquillo, S. Petit, and P. Lopez, *Multi2Sim: A Simulation Framework to Evaluate Multicore-Multithreaded Processors,* SBAC-PAD, 2007, **cited by 112** (Google Scholar, 5/11/2015)
  - R. Ubal, B. Jang, P. Mistry, D. Schaa, D. Kaeli, *Multi2Sim: A Simulation Framework for CPU-GPU Computing*, PACT, 2012, **cited by 73** (Google Scholar, 5/11/2015).
- Example conferences in citations
  - MICRO 2012
  - HPCA 2013
  - ICS 2015
  - NOCS 2015
  - … and others

The Multi2Sim Community

Collaboration Opportunities

- Current collaborators
  - Univ. of Mississippi, Univ. of Toronto, Univ. of Texas, Univ. Politecnica de Valencia (Spain), Boston University, AMD, NVIDIA
- Top of Trees (www.TopOfTrees.com)

  - Online framework for collaborative software development
  - Code peer reviews
  - Forum
  - Bug tracker
- Multi2Sim Project
  - 622 users registered (5/11/2015)

The Multi2Sim Community

Sponsors

Thank you!