Multi2Sim: A Simulation Framework for CPU-GPU Computing

Rafael Ubal[†]
ubal@ece.neu.edu

Byunghyun Jang[‡] [*]
bjang@cs.olemiss.edu

Perhaad Mistry[†]
pmistry@ece.neu.edu

Dana Schaa[†]
dschaa@ece.neu.edu

David Kaeli[†]
kaeli@ece.neu.edu

[†]Electrical and Computer Engineering Dept.
Northeastern University
360 Huntington Ave., Boston, MA 02115

[‡]Computer and Information Science Dept.
University of Mississippi
P. O. Box 1848, University, MS 38677

ABSTRACT

Accurate simulation is essential for the proper design and evaluation of any computing platform. Upon the current move toward the CPU-GPU heterogeneous computing era, researchers need a simulation framework that can model both kinds of computing devices and their interaction. In this paper, we present Multi2Sim, an open-source, modular, and fully configurable toolset that enables ISA-level simulation of an x86 CPU and an AMD Evergreen GPU. Focusing on a model of the AMD Radeon 5870 GPU, we address program emulation correctness, as well as architectural simulation accuracy, using AMD's OpenCL benchmark suite. Simulation capabilities are demonstrated with a preliminary architectural exploration study, and workload characterization examples. The project source code, benchmark packages, and a detailed user's guide are publicly available at *www.multi2sim.org*.

1. INTRODUCTION

GPUs have become an important component of High Performance Computing (HPC) platforms by accelerating the ever demanding data-parallel portions of a wide range of applications. The success of GPU computing has made microprocessor researchers in both academia and industry believe that CPU-GPU heterogeneous computing is not just an alternative, but the future of HPC. Now, GPUs are showing up as integrated accelerators for general purpose platforms [8, 5, 9]. This move attempts to leverage the combined capabilities of multi-core CPU and many-core GPU architectures.

As CPU-GPU heterogeneous computing research gains momentum, the need to provide a robust simulation environment becomes

more critical. Simulation frameworks provide a number of benefits to researchers. They allow pre-silicon designs to be evaluated and performance results to be obtained for a range of design points. A number of CPU simulators supporting simulation at the ISA level have been developed [11, 14] and successfully used in a range of architectural studies. Although there are tools that are currently available for simulating GPUs at the intermediate language level (e.g., PTX) [12, 13], the research community still lacks a publicly available framework integrating both fast functional simulation and cycle-accurate detailed architectural simulation at the ISA level that considers a true heterogeneous CPU-GPU model.

In this paper we present Multi2Sim, a simulation framework for CPU-GPU computing. The proposed framework integrates a publicly available model of the data-parallel AMD Evergreen GPU family [3][1] with the simulation of superscalar, multi-threaded, and multicore x86 processors. This work also offers important insight into the architecture of an AMD Evergreen GPU, by describing our models of instruction pipelines and memory hierarchy, to a deeper extent than previous public work, to the best of our knowledge, has done before.

Multi2Sim is provided as a Linux-based command-line toolset, designed with an emphasis on presenting a user-friendly interface. It runs OpenCL applications without any source code modifications, and provides a number of instrumentation capabilities that enable research in application characterization, code optimization, compiler optimization, and hardware architecture design. To illustrate the utility and power of our toolset, we report on a wide range of experimental results based on benchmarks taken from AMD's Accelerated Parallel Processing (APP) SDK 2.5 [1].

The rest of this paper is organized as follows. Section 2 introduces the functional simulation model in Multi2Sim. Section 3 presents the Evergreen GPU architecture and its simulation. Section 4 reports our experimental evaluation. We summarize related work in Section 5, and conclude the paper in Section 6.

---

[*]This work was done while the author was with AMD.

---

[1]AMD has used the Evergreen ISA specification for the implementation of its mainstream Radeon 5000 and 6000 series of GPUs.

2. THE MULTI2SIM PROJECT

The Multi2Sim project started as a free, open-source, cycle-accurate simulation framework targeting superscalar, multi-threaded, and multicore x86 CPUs. The CPU simulation framework consists of two major interacting software components: the *functional simulator* and the *architectural simulator*. The functional simulator (i.e., *emulator*) mimics the execution of a guest program on a native x86 processor, by interpreting the program binary and dynamically reproducing its behavior at the ISA level. The architectural simulator (i.e., *detailed* or *timing simulator*) obtains a trace of x86 instructions from the functional simulator, and tracks execution of the processor hardware structures on a cycle-by-cycle basis.

The current version of the CPU functional simulator supports the execution of a number of different benchmark suites without any porting effort, including single-threaded benchmark suites (e.g., SPEC2006 and Mediabench), multi-threaded parallel benchmarks (SPLASH-2 and PARSEC 2.1), as well as custom self-compiled user code. The architectural simulator models many-core superscalar pipelines with out-of-order execution, a complete memory hierarchy with cache coherence, interconnection networks, and additional components.

Multi2Sim integrates a configurable model for the commercial AMD Evergreen GPU family (e.g., Radeon 5870). The latest releases fully support both functional and architectural simulation of a GPU, following the same interaction model between them as for CPU simulation. While the GPU emulator provides traces of Evergreen instructions, the detailed simulator tracks execution times and architectural state.

All simulated programs begin with the execution of CPU code. The interface to the GPU simulator is the Open Compute Language (OpenCL). When OpenCL programs are executed, the host (i.e., CPU) portions of the program are run using the CPU simulation modules. When OpenCL API calls are encountered, they are intercepted and used to setup or begin GPU simulation.

2.1 The OpenCL Programming Model

OpenCL is an industry-standard programming framework designed specifically for developing programs targeting heterogeneous computing platforms, consisting of CPUs, GPUs, and other classes of processing devices [7]. OpenCL's programming model emphasizes parallel processing by using the *single-program multiple-data* (SPMD) paradigm, in which a single piece of code, called a *kernel*, maps to multiple subsets of input data, creating a massive amount of parallel execution.

Figure 1 provides a view of the basic execution elements hierarchy defined in OpenCL. An instance of the OpenCL kernel is called a *work-item*, which can access its own pool of *private memory*. Work-items are arranged into *work-groups* with two basic properties: i) those work-items contained in the same work-group can
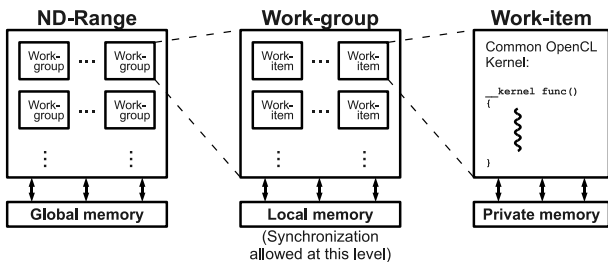
perform efficient synchronization operations, and ii) work-items within the same work-group can share data through a low-latency *local memory*. The totality of work-groups form the *ND-Range* (grid of work-item groups) and share a common *global memory*.

2.2 OpenCL Simulation

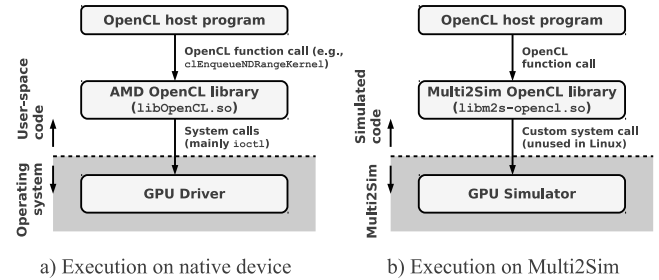a) Execution on native device   b) Execution on Multi2Sim

Figure 2: Comparison of software modules of an OpenCL program: native AMD GPU based heterogeneous system versus Multi2Sim simulation framework.

The call stack of an OpenCL program running on Multi2Sim differs from the native call stack starting at the OpenCL library call, as shown in Figure 2. When an OpenCL API function call is issued, our implementation of the OpenCL runtime (`libm2s-opencl.so`) handles the call. This call is intercepted by the CPU simulation module, which transfers control to the GPU module as soon as the guest application launches the device kernel execution. This infrastructure allows unmodified x86 binaries (pre-compiled OpenCL host programs) to run on Multi2Sim with total binary compatibility with the native environment.

3. ARCHITECTURAL SIMULATION OF AN AMD EVERGREEN GPU

This section presents the architecture of a generic AMD Evergreen GPU device, focusing on hardware components devoted to general purpose computing of OpenCL kernels. As one of the novelties of this paper, the following block diagrams and descriptions provide some insight into the instruction pipelines, memory components, and interconnects, which tend to be kept private by the major GPU vendors, and remain undocumented in currently available tools. All presented architectural details are accurately modeled on Multi2Sim, as described next.

3.1 The Evergreen GPU Architecture

A GPU consists of an *ultra-threaded dispatcher*, an array of independent *compute units*, and a *memory hierarchy*. The ultra-threaded dispatcher processes the ND-Range and maps waiting work-groups onto available compute units. Once a work-group is assigned to a compute unit, it remains in the compute unit until its execution completes. As a work-group executes, work-items fetch and store data through the global memory hierarchy, formed of two levels of cache, interconnects, and memory controllers. Figure 3a shows a block diagram of the Evergreen family compute device.

A compute unit consists of three *execution engines*, a *local memory*, and a *register file*. The three execution engines, called *control flow* (CF), *arithmetic-logic* (ALU), and *texture* (TEX) engines, are devoted to execute different portions of an OpenCL kernel binary, referred to as CF, ALU, and TEX clauses, respectively (see Section 3.2). A block diagram of the compute unit is illustrated in Figure 3b.
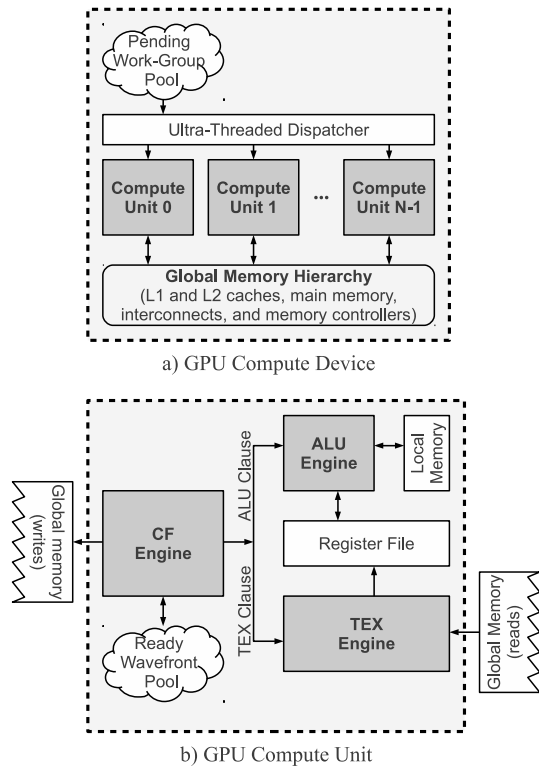

Figure 1: OpenCL programming and memory model.

a) GPU Compute Device

b) GPU Compute Unit

Figure 3: Block diagram of the GPU architecture.

The ALU engine contains a set of *stream cores*, each devoted to the execution of one work-item's arithmetic operations. ALU instructions are organized as 5-way VLIW bundles, created at compile time. Each instruction in a VLIW bundle is executed on one of the 5 VLIW lanes forming the stream core.

An Evergreen GPU defines the concept of a *wavefront* as a group of work-items executing in a Single-Instruction Multiple-Data (SIMD) fashion. Each instruction is executed concurrently by every work-item comprising a wavefront, although each work-item uses its private data for the computations. This model simplifies instruction fetch hardware by implementing a common front-end for a whole wavefront.

3.2 The Evergreen Instruction Set Architecture (ISA)

When the GPU functional simulator receives the OpenCL kernel to execute, an emulation loop starts by fetching, decoding, and executing Evergreen instructions. The basic format of the AMD Evergreen ISA can be observed in the sample code from Figure 4.

Evergreen assembly uses a clause-based format. The kernel execution starts with a CF instruction. CF instructions affect the main program control flow (such is the case for CF instruction 03), write data to global memory (04), or transfer control to a secondary clause, such as an ALU clause (00, 02), or a TEX clause (01). ALU clauses contain instructions performing arithmetic-logic operations and local memory accesses, while TEX clauses are exclusively devoted to global memory read operations.

ALU instructions are packed into VLIW bundles. A VLIW bundle is run one at a time on a stream core, where each ALU instruction label reflects the VLIW lane assigned to that instruction. An ALU instruction operand can be any output from the previously executed VLIW bundle using the *Previous Vector* (PV) or the *Pre-*
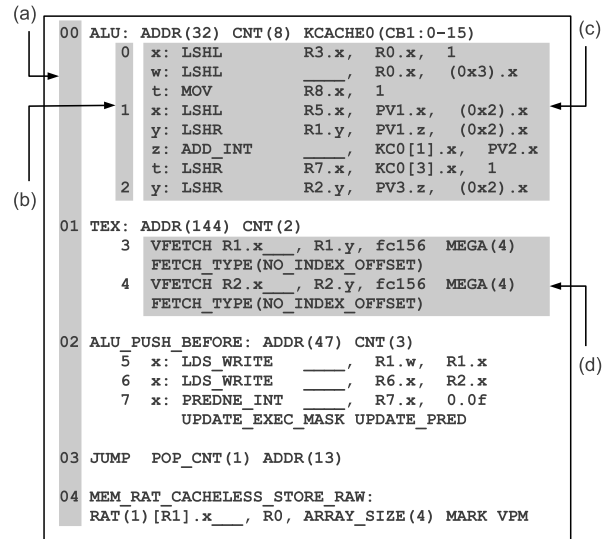

Figure 4: Example of AMD Evergreen assembly code: (a) main CF clause instruction counter, (b) internal clause instruction counter, (c) ALU clause, (d) TEX clause.

vious Scalar (PS) special registers. Finally, *constant memory* is an additional globally accessible storage initialized by the CPU, which can also be used as ALU instruction operands (KC).

From our discussion above of Evergreen ISA characteristics, we can observe a couple of important differences from working with higher level intermediate languages, such as AMD's IL [4] or NVIDIA's PTX [6]. For example, in AMD's Evergreen ISA there is a limited number of general purpose registers, so there are restrictions on how to form VLIW bundles, and there are specific rules to group machine instructions forming clauses. In general, there are many properties of the ISA run directly by the machine that need not be considered working with an intermediate language. Thus, significant performance accuracy can be gained with ISA-level simulation.

3.2.1 Kernel Execution Model

When an OpenCL kernel is launched by a host program, the ND-Range configuration is provided to the GPU. Work-groups are then created and successively assigned to compute units when they have available execution resources. The number of work-groups that can be assigned to a single compute unit is determined by four hardware limitations: i) the maximum number of work-groups supported per compute unit, ii) the maximum number of wavefronts per compute unit, iii) the number of registers on a compute unit, and iv) the amount of local memory on a compute unit. Maximizing the number of assigned work-groups per compute unit is a performance-sensitive decision that can be evaluated on Multi2Sim.

Each work-group assigned to a compute unit is partitioned into wavefronts, which are then placed into a *ready wavefront pool*. The CF engine selects wavefronts from the wavefront pool for execution, based on a *wavefront scheduling* algorithm. A new wavefront starts running the main CF clause of the OpenCL kernel binary, and subsequently spawns secondary ALU and TEX clauses. The wavefront scheduling algorithm is another performance sensitive parameter, which can be evaluated with Multi2Sim.

When a wavefront is extracted from the pool, it is only inserted back in when the executed CF instruction completes. This ensures that there is only a single CF instruction in flight at any time for a

given wavefront, avoiding the need for branch prediction or speculative execution in case flow control is affected. The performance penalty for this serialization is hidden by overlapping the execution of different wavefronts. Determining the extent to which overlapping execution is occurring and the cause of bottlenecks are additional benefits of simulating execution with Multi2Sim.

3.2.2 Work-Item Divergence

In a SIMD execution model, *work-item divergence* is side-effect generated when a conditional branch instruction is resolved differently for any work-items within a wavefront. To address work-item divergence present during SIMD execution, the Evergreen ISA provides each wavefront with an *active mask*. The active mask is a bit map, where each bit represents the active status of an individual work-item in the wavefront. If a work-item is labeled as inactive, the result of any arithmetic computation performed in its associated stream core is ignored, preventing the work-item from changing the kernel state.

This work-item divergence strategy attempts to converge all work-items together across all possible execution paths, allowing only those active work-items whose conditional execution matches the currently fetched instruction flow to continue execution. To support nested conditionals and procedure calls, an *active mask stack* is used to push and pop active masks, so that the active mask at the top of the stack always represents the active mask of the currently executing work-items. Using Multi2Sim, statistics related to work-item divergence are available to researchers (see Section 4.3).

3.3 The Instruction Pipelines

In a compute unit, the CF, ALU, and TEX engines are organized as instruction pipelines. Figure 5 presents a block diagram of each engine's instruction pipeline. Within each pipeline, decisions about scheduling policies, latencies, and buffer sizes must be made. These subtle factors have performance implications, and provide another opportunity for researchers to benefit from experimenting with design decisions within Multi2Sim.

The CF engine (Figure 5a) runs the CF clause of an OpenCL kernel. The *fetch* stage selects a new wavefront from the wavefront pool on every cycle, switching among them at the granularity of one single CF instruction. Instructions from different wavefronts are interpreted by the *decode* stage in a round-robin fashion. When a CF instruction triggers a secondary clause, the corresponding execution engine (ALU or TEX engine) is allocated, and the CF instruction remains in the *execute* stage until the secondary clause completes. Other CF instructions from other wavefronts can be executed in the interim, as long as they do not request a busy execution engine. CF instruction execution (including all instructions run in a secondary clause, if any) finishes in order in the *complete* stage stage. The wavefront is returned to the wavefront pool, making it again a candidate for instruction fetching. Global memory writes are run asynchronously in the CF engine itself, without requiring a secondary engine.

The ALU engine is devoted to the execution of ALU clauses from the allocated wavefront (Figure 5b). After the *fetch* and *decode* stages, decoded VLIW instructions are placed into a *VLIW bundle buffer*. The *read* stage consumes the VLIW bundle and reads the source operands from the register file and/or local memory for each work-item in the wavefront. The *execute* stage issues an instance of a VLIW bundle to each of the stream cores every cycle. The number of stream cores in a compute unit might be smaller than the number of work-items in a wavefront. Thus, a wavefront is split into *subwavefronts*, where each subwavefront contains as many work-items as there are stream cores in a compute unit. The

result of the computation is written back to the destination operands (register file or local memory) at the *write* stage.

The TEX engine (Figure 5c) is devoted to the execution of global memory fetch instructions in TEX clauses. The TEX instruction bytes are stored into a TEX instruction buffer after being fetched and decoded. Memory addresses for each work-item in the wavefront are read from the register file and a read request to the global memory hierarchy is performed at the *read* stage. Completed global memory reads are handled in order by the *write* stage. The fetched data is stored into the corresponding locations of the register file for each work-item. The lifetime of a memory read is modeled in detail throughout the global memory hierarchy, as specified in the following sections.

3.4 Memory Subsystem

The GPU memory subsystem contains different components for data storage and transfer. With Multi2Sim, the memory subsystem is highly configurable, including customizable settings for the number of cache levels, memory capacities, block sizes, number of banks, and ports. A description of the memory components for the Evergreen model follows:

Register file (GPRs). Multi2Sim provides a model with no contention for register file accesses. In a given cycle, the register can be accessed by the TEX and ALU engines simultaneously by different wavefronts. Work-items within and among wavefronts always access different register sets.

Local Memory. A separate local memory module is present in each compute unit, and is modeled in Multi2Sim with a configurable latency, number of banks, ports, and allocation chunk size. In an OpenCL kernel, accesses to local memory are defined by the programmer by specifying a variable's scope, whose accesses are then compiled into distinct assembly instructions. Contention to local memory is modeled by serializing accesses to the same memory bank whenever no read or write port is available. Also, memory access coalescing is considered by grouping those accesses from different work-items to the same memory block.

Global memory. The GPU global memory is accessible by all compute units. It is presented to the programmer as a separate memory scope, and implemented as a memory hierarchy managed by hardware in order to reduce access latency. In Multi2Sim, the global memory hierarchy has a configurable number of cache levels and interconnects. A possible configuration is shown in Figure 6a, using private L1 caches per compute unit, and multiple L2 caches that are shared between subsets of compute units. L1 caches provide usually a similar access time as local memory, but they are managed transparently by hardware, similarly to how a memory hierarchy is managed on a CPU.

Interconnection networks. Each cache in the global memory hierarchy is connected to the lower-level cache (or global memory) using an interconnection network. Interconnects are organized as point-to-point connections using a switch, whose architecture block diagram is presented in Figure 6b. A switch contains two disjoint inner subnetworks, each devoted to package transfers in opposite directions.

Cache access queues. Each cache memory has a buffer where access requests are enqueued, as shown in Figure 6c. On one hand, access buffers allow for asynchronous writes that prevent stalls in instruction pipelines. On the other hand, memory access coalescing is handled in access buffers at every level of the global memory hierarchy (both caches and global memory). Each sequence of subsequent entries in the access queue reading or writing to the same cache block are grouped into one single actual memory access. The coalescing degree depends on the memory block size,
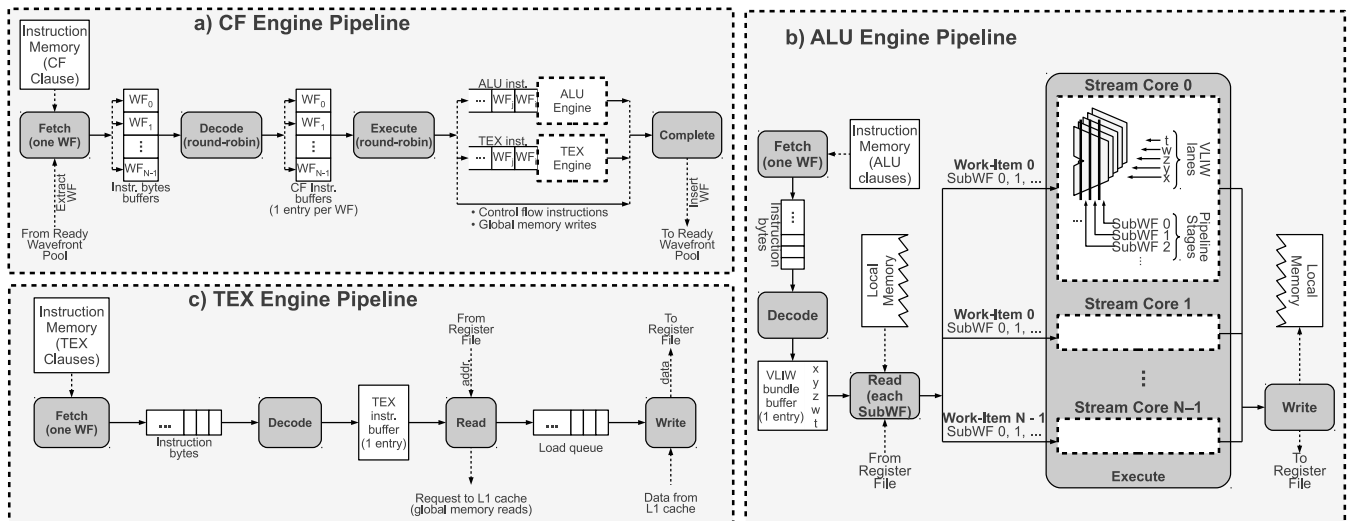
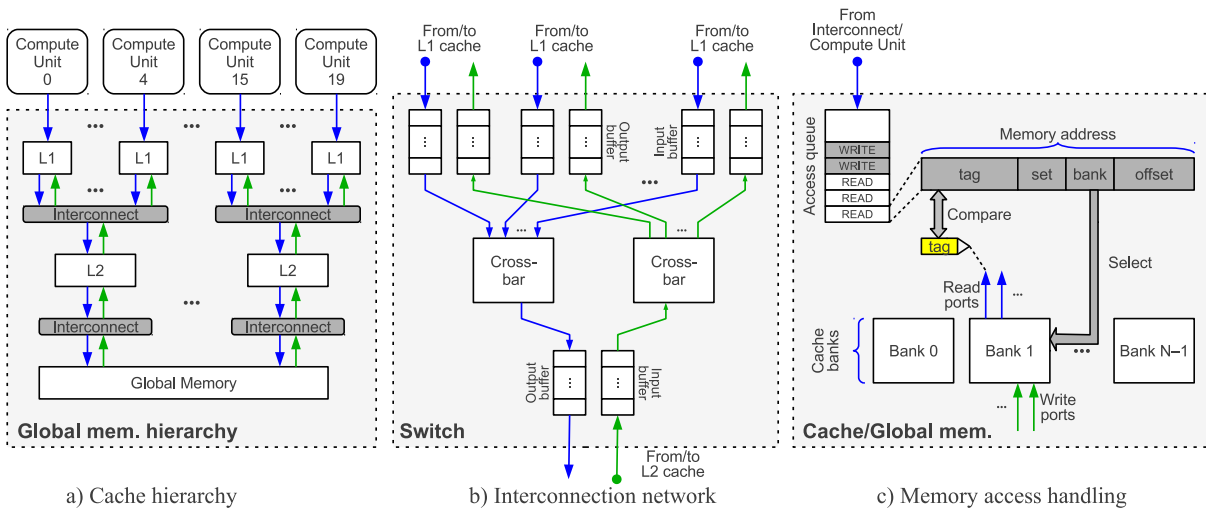**Figure 5: Block diagram of the execution engine pipelines.**

a) Cache hierarchy   b) Interconnection network   c) Memory access handling

Figure 6: Components of the GPU global memory hierarchy, as modeled in Multi2Sim.

the access queue size, and the memory access pattern, and is a very performance sensitive metric measurable with Multi2Sim.

4. EXPERIMENTAL EVALUATION

This section presents a set of experiments aimed at validating and demonstrating the range of functional and architectural simulation features available with Multi2Sim. All simulations are based on a baseline GPU model resembling the commercial AMD Radeon 5870 GPU, whose hardware parameters are summarized in Table 1.

For the simulator performance studies, simulations were run on a machine with four quad-core Intel Xeon processors (2.27GHz, 8MB cache, 24GB DDR3). Experimental evaluations were performed using a subset of the AMD OpenCL SDK [1] applications, representing a wide range of application behaviors and memory access patterns [16]. The applications discussed in this paper are listed in Table 2, where we include a short description of the programs and the corresponding input dataset characteristics.

4.1 Validation

Our validation methodology for establishing the fidelity of the GPU simulator considered the correctness of both the functional and architectural simulation models, though we follow two different validation methodologies. For the functional simulator, the correctness of the instruction decoder is validated by comparing the disassembled code to the Evergreen output that is generated by the AMD compiler. We also validate the correctness of each benchmark's execution by comparing the simulated application output with the output of the application run directly on the CPU. All simulations generate functionally correct results for all programs studied and input problem sets.

Regarding the fidelity of the architectural model, Multi2Sim's performance results have been compared against native execution performance (native here refers to the actual Radeon 5870 hardware), using ten different input sizes within the ranges shown in Table 2 (column *Input Range*). Since our architectural model is cycle-based, and the native execution is measured as kernel execution time, it is challenging to compare our metrics directly. To con-
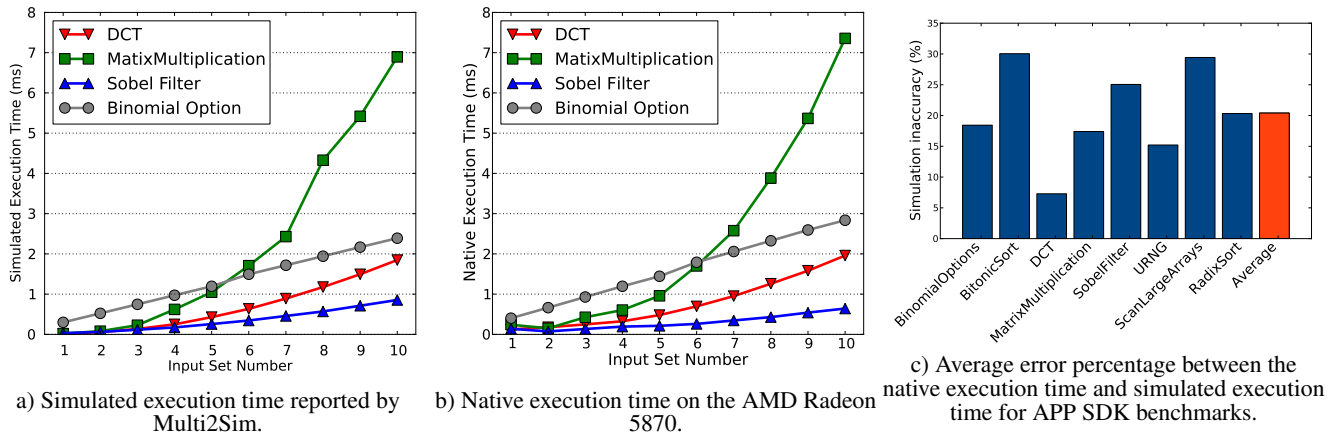
a) Simulated execution time reported by Multi2Sim.

b) Native execution time on the AMD Radeon 5870.

c) Average error percentage between the native execution time and simulated execution time for APP SDK benchmarks.

Figure 7: Validation for the architectural simulation, comparing simulated and native absolute execution times.

a) Correlation between simulated execution times and native execution times.

b) Comparison of cache hit rate in simulated and native executions, along with measured difference in overall execution time. Workloads are sorted from compute bound to memory bound.
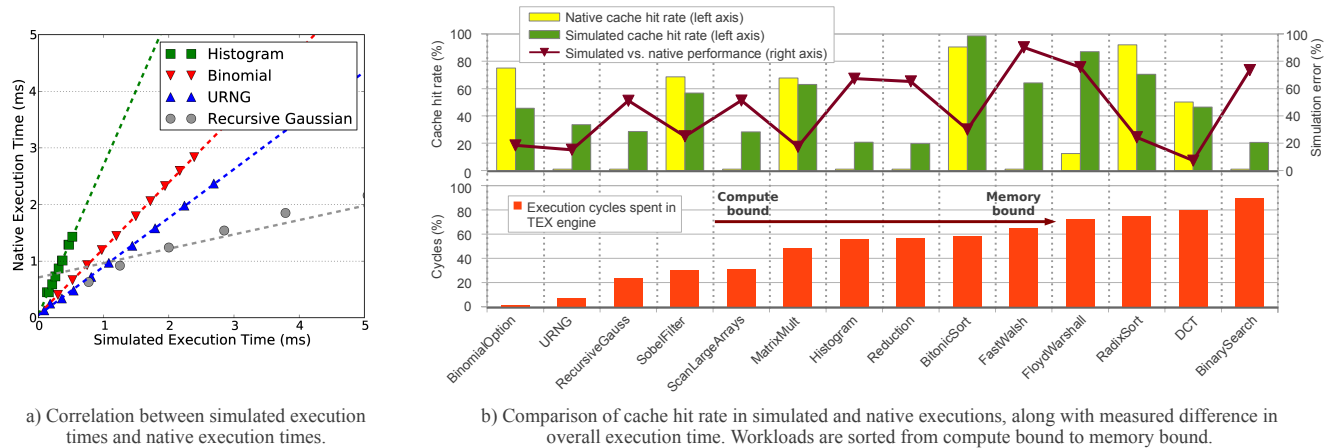
Figure 8: Validation for architectural simulation, comparing trends between simulated and native execution times.

vert simulated cycles into time, we use the documented ALU clock frequency of 850MHz of the 5870 hardware. The native execution time is computed as the average time of 1000 kernel executions for each benchmark. Native kernel execution time was measured using the AMD APP profiler [2]. The execution time provided by the APP profiler does not include overheads such as kernel setup and host-device I/O [2].

Figure 7a and Figure 7b plot simulated execution time and native execution time performance trends, respectively (only four benchmarks are shown for clarity). Figure 7c shows the percentage difference in performance for a larger selection of benchmarks. The value shown for each benchmark in Figure 7c is the average of the absolute percent error for each input of the benchmark. For those cases where simulation accuracy decreases, Figure 8 shows detailed trends, leading to the following analysis.

In Figure 8a, we show the correlation between the native execution time and the simulated execution time for the studied benchmarks. For some of the benchmarks (e.g., *Histogram* or *Recursive-Gauss*), execution times vary significantly. However, we still see a strong correlation between each of the native execution points and their associated simulator results for all benchmarks. In other words, a change in the problem size for a benchmark has the same

relative performance impact for both native and simulated executions. The linear trend-line is represented using a curve-fitting algorithm that minimizes the squared distances between every data point and itself. For the benchmarks that are modeled accurately using the simulator, the data points lie on the $45°$ line. The reason for the occurrence of divergent slopes can be attributed to the lack of precise representation of the memory hierarchy in the 5870 GPU, including the following factors:

Specialized Memory Path Design. The AMD Radeon 5870 consists of two paths from compute units to memory [2], each with different performance characteristics. The *fast path* performs only basic operations, such as loads and stores for 32-bit data types. The *complete path* supports additional advanced operations, including atomics and stores for sub-32-bit data types. This design has been deprecated in later GPU architectures for a more conventional layout [17], which is similar to the one currently implemented in Multi2Sim.
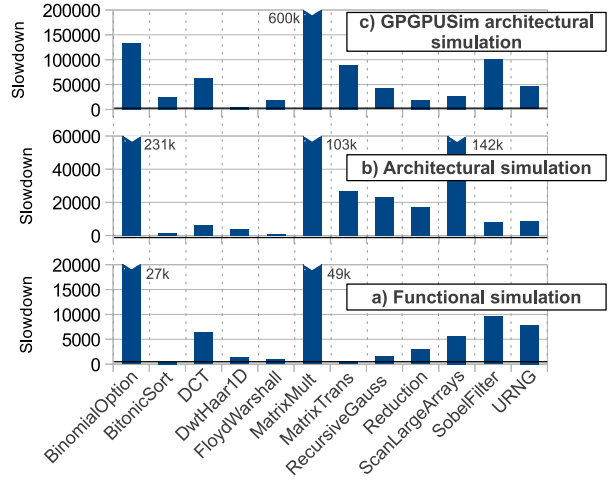
Cache Interconnects. The specification of the interconnection network between the L1 and L2 caches has not been published. We use an approximation where four L2 caches are shared between compute units (Table 1).

Table 1: Baseline GPU simulation parameters.

Compute Device Configuration		
Compute resources	# of compute units (CU)	20
	# of stream cores (SC) / CU	16
	# of processing elements / stream core	5
	Total # of processing elements	**20*16*5 = 1600**
Register file	# of vector registers	16,384
	# of 32 bit registers / vector	4
	Total amount of register memory	**16k*4*4=256KB**
Local memory	# of banks	32
	Block Size	16B
	# of Read / Write ports (1 per SC)	16R / 16W
	Total amount of local memory	**32kB**

Global Memory Configuration		
L1 Cache (one per CU) Accessed by 16 SCs	# of banks	32
	Associativity	8 way
	Latency (cycles)	1
	Block Size	64B
	# of Read / Write ports	16R / 16W
	Total L1 cache size	**8KB**
L2 Cache (each shared between 5 CUs)	# of L2 caches	4
	# of banks	32
	Associativity	8 way - set
	Latency (cycles)	10
	Block Size	256B
	# of Read / Write ports	2R / 2W
	Total L2 cache size	**4x128KB =512KB**
Main memory	# of banks	128
	Bus width	256B / cycle
	Latency (cycles)	100
	Global memory size	**1GB**

Table 2: List of OpenCL benchmarks used for experiments. Column *Input base* contains the baseline problem size used, and column *Input range* contains the range of problem sizes used during simulator validation.

Benchmark	Input feature	Inp. base	Input range
BinarySearch	Array Size	1M	2k - 2M
BinomialOption	Num. of options	640	64 – 640
BitonicSort	Vector size	4k	512 – 10k
DCT	Matrix size	1024x1024	128x128–1280x1280
DwtHaar1D	Array size	256k	1k – 512k
FastWalsh	Array size	640	1k – 512k
FloydWarshall	Num. of nodes	128	64 – 512
Histogram	Array size	1M	16k - 2M
MatrixMult	Matrix sizes	1024x1024	128x128–1280x1280
RadixSort	Array size	32k	8k - 128k
RecursiveGauss	Image size	512x512	256x256–1536x1536
Reduction	Array size	4M	400k – 4M
ScanLargeArrays	Array size	1M	512 – 2M
SobelFilter	Image size	1024x768	256x256–1536x1536
URNG	Image size	1024x768	256x256–1536x1536

Figure 9: Simulation slowdowns over native execution for functional and architectural simulation.

Cache Parameters. The latency and associativity of the different levels of the cache hierarchy are not known. Some sources of simulation inaccuracy can be attributed to cache parameters, as shown in Figure 8, where the percent error is minimum for the cases where the native cache hit ratios and simulated cache hit ratios vary the least.

4.2 Simulation Speed

For the benchmarks used in this paper, Multi2Sim's simulation overhead is plotted in Figure 9 as a function of the slowdown over native execution time. The average functional simulation slowdown is $8700\times$ (113s), and the average architectural simulation time is $44000\times$ (595s). It should be noted that simulation time is not necessarily related to native execution time (e.g., simulating one 100-cycle latency instruction is faster than simulating ten 1-cycle instructions), so these results only aim to provide some representative samples of simulation overhead.

Simulation performance has been also evaluated for an architectural simulation on GPGPUSim, an NVIDIA-based GPU simulator [10]. This simulator has been used as experimental support for recent studies on GPU computing, exploring alternative memory controller implementations [18] and dynamic grouping of threads (work-items) to minimize thread divergence penalty [15], for example. To enable this comparison, the APP SDK benchmarks were adapted to run on GPGPUSim. Figure 9c shows the performance slowdown over native execution, which averages about $90000\times$ (1350s).

4.3 Benchmark Characterization

As a case study of GPU simulation, this section presents a brief characterization of OpenCL benchmarks carried out on Multi2Sim, based on instruction classification, VLIW bundle occupancy, and control flow divergence. These statistics are dynamic in nature, and are reported by Multi2Sim as part of its simulation reports.

Figure 10a shows Evergreen instruction mixes executed by each OpenCL kernel. The instruction categories are control flow instructions (jumps, stack operations, and synchronizations), global memory reads, global memory writes, local memory accesses, and arithmetic-logic operations. Arithmetic-logic operations form the bulk of executed instructions (these are GPU-friendly workloads).

Figure 10b represents the average occupancy of VLIW bundles

a) Classification of instruction types.   b) VLIW bundles occupancy.   c) Control flow divergence.
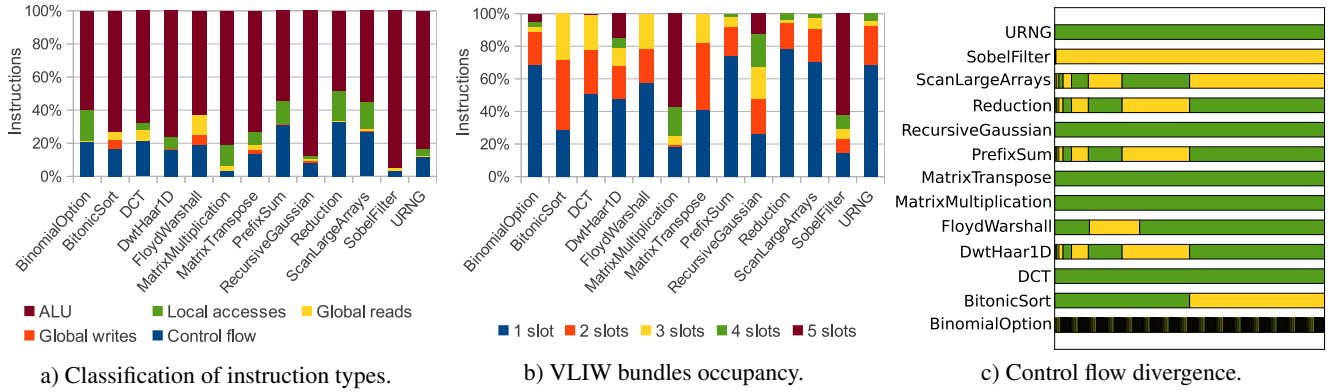
Figure 10: Examples of benchmarks characterization, based on program features relevant to GPU performance. IPC is calculated as total number of instructions the for entire kernel, divided by total cycles to execute the entire kernel.

executed in the stream cores of the GPU's ALU engine. If a VLIW instruction uses less than 5 slots, there will be idle VLIW lanes in the stream core, resulting in an underutilization of available execution resources. The Evergreen compiler tries to maximize the VLIW slot occupancy, but there is an upper limit imposed by the available instruction-level parallelism in the kernel code. Results show that we rarely utilize all 5 slots (except for *SobelFilter* thanks to its high fraction of ALU instructions), and the worst case of only one single filled slot is encountered frequently.

Finally, Figure 10c illustrates the control flow divergence effect among work-items. When work-items within a wavefront executing in a SIMD fashion diverge on branch conditions, the entire wavefront must go through all possible execution paths. Thus, frequent work-item divergence has a negative impact on performance. For each benchmark in Figure 10c, each color stride within a bar represents a different control flow path through the program. If a bar has one single stride, then only one path was taken by all work-items for that kernel. If there are n strides, then n different control flow paths were taken by different work-items. Notice that different colors are used here with the only purpose of delimiting bar strides, but no specific meaning is assigned to each color. The size of each stride represents the percentage of work-items that took that control flow path for the kernel. Results show benchmarks with the following divergence characteristics:

- No control flow divergence at all (*URNG*, *DCT*).

- Groups of divergence with a logarithmic decreasing size due to different number of loop iterations (*Reduction*, *DwtHaar1D*).

- Multiple divergence groups depending on input data (*BinomialOption*[2]).

4.4 Architectural Exploration

The architectural GPU model provided in Multi2Sim allows researchers to perform large design space evaluations. As a sample of the simulation flexibility, this section presents three case studies, where performance significantly varies for different input parameter values. In each case, we compare two benchmarks with respect

to their architectural sensitivity. Performance is measured using the number of instructions per cycle (IPC), where the instruction count is incremented by one for a whole wavefront, regardless of the number of comprising work-items.

Figure 11a shows performance scaling with respect to the number of compute units. The total memory bandwidth provided by global memory is shared by all compute units, so increasing the number of compute units decreases the available bandwidth per executed work-group. The available memory bandwidth for the device in this experiment only increases between compute unit counts which are a multiple of 5 when a new L2 is added (Table 1). When the total bandwidth is exhausted, the trend (as seen between 10-15 and 15-20 compute units) flattens. This point is clearly observed when we increase the number of compute units in compute-intensive kernels with high ALU-to-Fetch instruction ratios (e.g., *URNG*) and less so in memory-intensive benchmarks (e.g., *Histogram*).

Figure 11b presents the performance achieved by varying the number of stream cores per compute unit. In the *BinomialOption* kernel we observe a step function, where each step corresponds to a multiple of the wavefront size (64). This behavior is due to the fact that the number of stream cores determines the number of sub-wavefronts (or time-multiplexed slots) that stream cores deal with for each VLIW bundle. When an increase in the number of stream cores causes a decrease in the number of subwavefronts (e.g., 15 to 16, 21 to 22, and 31 to 32), performance improves. When the number of stream cores matches the number of work-items per wavefront, the bottleneck due to a serialized stream core utilization disappears. This effect is not observed for *ScanLargeArrays* due to a lower wavefront occupancy.

Figure 11c plots the impact of increasing the L1 cache size. For benchmarks that lack temporal locality and exhibit large strided accesses in the data stream, performance is insensitive to increasing cache size, as seen in *Reduction*. In contrast, benchmarks with locality are more sensitive to changes in the L1 cache size, as observed for *FloydWarshall*.

5. RELATED WORK

While numerous mature CPU simulators at various levels are available, GPU simulators are still in their infancy. There continues to be a growing need for architectural GPU simulators that model
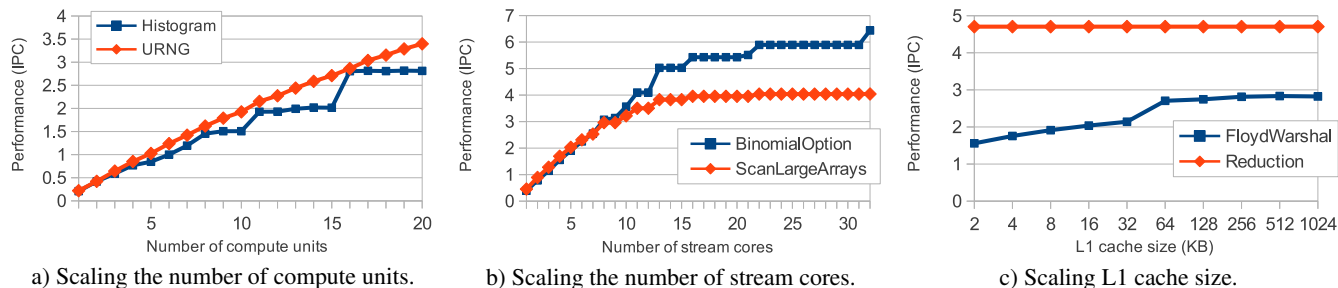
---

[2]The darker color for *BinomialOption* is caused by many small divergence regions represented in the same bar.

a) Scaling the number of compute units.   b) Scaling the number of stream cores.   c) Scaling L1 cache size.

Figure 11: Architectural exploration, showing results for those benchmarks with interesting performance trends.

a GPU at the ISA level. And in the near future, we will see a more pressing need for a true CPU-GPU heterogeneous simulation framework. This section briefly summarizes existing simulators targeting GPUs.

Barra [12] is an ISA-level functional simulator targeting the NVIDIA G80 GPUs. It runs CUDA executables without any modification. Since the NVIDIA's G80 ISA specification is not publicly available, the simulator relies on a reverse-engineered ISA provided by another academic project. Similar to our approach, Barra intercepts API calls to the CUDA library and reroutes them to the simulator. Unfortunately, it is limited to GPU functional simulation, lacking an architectural simulation model.

GPGPUSim [10] is a detailed simulator that models a GPU architecture similar to NVIDIA's architecture. It includes a shader core, interconnects, thread block (work-group) scheduling, and memory hierarchy. Multi2Sim models a different GPU ISA and architecture (Evergreen). GPGPUSim can provide us with important insight into design problems for GPUs. However, Multi2Sim also supports CPU simulation within the same tool enabling additional architectural research into heterogeneous architectures.

Ocelot [13] is a widely used functional simulator and dynamic compilation framework that works at a virtual ISA level. Taking NVIDIA's CUDA PTX code as input, it can either emulate or dynamically translate it to multiple platforms such as x86 CPUs, NVIDIA GPUs, and AMD GPUs. Ocelot has objectives different than GPU architectural simulation, so there is an extensive functionality not provided or targeted by Multi2Sim, which makes them complementary tools.

When compared to previous work, Multi2Sim is unique in the following aspects. First, it models the native ISA of a commercially available GPU. Second, it provides an architectural simulation of a real GPU with tractable accuracy. Third, Multi2Sim is a CPU-GPU heterogeneous simulation framework, which can be used to evaluate upcoming architectures where the CPU and GPU are merged on silicon and share a common memory address space [8].

6. CONCLUSIONS

In this paper we have presented Multi2Sim, a full-fledged simulation framework that supports both fast functional and detailed architectural simulation for x86 CPUs and Evergreen GPUs at the ISA level. It is modular, fully configurable, and easy to use. The toolset is actively maintained and is available as a free, open-source project at *www.multi2sim.org*, together with packages of benchmarks, a complete user guide, and active mailing lists and forums.

Ongoing work for Multi2Sim includes expanding benchmark

support by increasing Evergreen ISA coverage. Future releases will include a model for the AMD Fusion architecture, where the CPU and GPU share a common global memory hierarchy and address space. Supporting shared memory for heterogeneous architectures highlights the potential of Multi2Sim, as no other simulator can provide useful architectural statistics in this type of environment. Current development also includes support for OpenGL applications and exploration into OpenCL language extensions. Since Multi2Sim is currently being used by a number of leading research groups, we believe this is a great opportunity to accelerate research on heterogeneous, parallel architectures.

Acknowledgments

7. REFERENCES

[1] AMD Accelerated Parallel Processing (APP) Software Development Kit (SDK). http://developer.amd.com/sdks/amdappsdk/.

[2] AMD Accelerated Parallel Processing OpenCL Programming Guide (v1.3c).

[3] AMD Evergreen Family Instruction Set Arch. (v1.0d). http://developer.amd.com/sdks/amdappsdk/documentation/.

[4] AMD Intermediate Language (IL) Spec. (v2.0e). http://developer.amd.com/sdks/amdappsdk/documentation/.

[5] Intel Ivy Bridge. http://ark.intel.com/products/codename/29902/Ivy-Bridge.

[6] NVIDIA PTX: Parallel Thread Execution ISA. http://developer.nvidia.com/cuda-downloads/.

[7] OpenCL: The Open Standard for Parallel Programming of Heterogeneous Systems. www.khronos.org/opencl.

[8] The AMD Fusion Family of APUs. http://fusion.amd.com/.

[9] The NVIDIA Denver Project. http://blogs.nvidia.com/.

[10] A. Bakhoda, G. Yuan, W. Fung, H. Wong, and T. Aamodt. Analyzing CUDA Workloads Using a Detailed GPU Simulator. In *Proc. of the Int'l Symposium on Performance Analysis of Systems and Software (ISPASS)*, Apr. 2009.

[11] N. L. Binkert, E. G. Hallnor, and S. K. Reinhardt. Network-Oriented Full-System Simulation Using M5. *6th Workshop on Computer Architecture Evaluation using Commercial Workloads (CAECW)*, Feb. 2003.

[12] S. Collange, M. Daumas, D. Defour, and D. Parello. Barra: A Parallel Functional Simulator for GPGPU. In *Proc. of the 18th Int'l Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS)*, Aug. 2010.

[13] G. Diamos, A. Kerr, S. Yalamanchili, and N. Clark. Ocelot: a Dynamic Optimization Framework for Bulk-Synchronous Applications in Heterogeneous Systems. In *Proc. of the 19th Int'l Conference on Parallel Architectures and Compilation Techniques*, Sept. 2010.

[14] P. S. M. et. al. Simics: A Full System Simulation Platform. *IEEE Computer*, 35(2), 2002.

[15] W. W. L. Fung, I. Sham, G. Yuan, and T. M. Aamodt. Dynamic Warp Formation and Scheduling for Efficient GPU Control Flow. In *Proc. of the 40th Int'l Symposium on Microarchitecture*, Dec. 2007.

[16] B. Jang, D. Schaa, P. Mistry, and D. Kaeli. Exploiting Memory Access Patterns to Improve Memory Performance in Data-Parallel Architectures. *IEEE Transactions on Parallel and Distributed Systems*, 22(1), Jan. 2011.

[17] M. Houston and M. Mantor. AMD Graphics Core Next. http://developer.amd.com/afds/assets/presentations/2620_final.pdf.

[18] G. L. Yuan, A. A. Bakhoda, and T. M. Aamodt. Complexity Effective Memory Access Scheduling for Many-Core Accelerator Architectures. In *42nd Int'l Symposium on Microarchitecture*, Dec. 2009.