Visualization of OpenCL Application Execution on CPU-GPU Systems

Amir Kavyan Ziabari *, Rafael Ubal*, Dana Schaa†[1] and David Kaeli*
*Department of Electrical and Computer Engineering, Northeastern University, Boston, MA. USA
{aziabari,ubal,kaeli}@ece.neu.edu

†Advanced Micro Devices, San Jose, CA. USA
schaa@amd.com

Abstract—**Evaluating the performance of parallel and heterogeneous programs and architectures can be challenging. An emulator or simulator can be used to aid the programmer. To provide guidance and feedback to the programmer, the simulator needs to present traces, reports, and debugging information in a coherent and unambiguous format. Although these outputs contain a lot of detailed information relative to the logical and physical transactions about the execution, they are usually extremely large and hard to analyze. What is needed is an interface into the simulator that can help programmers and architects shift through this myriad of data.**
In this contribution, we describe the M2S-Visual trace-driven visualization tool, a complementary addition to Multi2sim (M2S) heterogeneous system simulator. M2S-Visual provides a graphical representation of parallel program execution on the simulator. M2S is an established simulator, designed with an emphasis on running parallel applications on graphics processing units, and provides a number of instrumentation capabilities that enable research in architecture exploration and application characterization. This visualization framework, added to Multi2sim, aims to complement (and potentially replace) text-based statistical profiling, enabling the user to better understand each software transaction executed on the simulated hardware. While M2S supports emulation of both OpenCL and CUDA programs, our visualization framework presently only supports OpenCL execution. M2S supports execution on both CPUs (X86, ARM and MIPS) and GPUs (AMD Evergreen and Southern Islands, and NVIDIA Fermi and Kepler), but presently only supports detailed visualization on a multicore X86 CPU and AMD Evergreen and Southern Islands GPUs. Besides supporting OpenCL programming and debugging, an additional goal is to deliver a reliable product for teaching the details parallel programming execution on heterogeneous systems. Given the move to many-core architectures in the industry, this toolset is timely and addressing a growing gap in our educational infrastructure. The tool is also designed to support the research community, providing analysis of performance bottlenecks of OpenCL programs. We also introduce some new visualization which provide deeper insight into application performance and hardware resource utilization.

I. INTRODUCTION

Over the course of three decades, computer systems have grown from single-core processing systems to systems with hundreds to thousands of computing cores, such as GPUs. To guarantee efficient execution within, and communication between, so many cores, requires hardware and software designs that have growing complexity. Evaluating the trade-offs of design complexity and efficiency leads to innovations in system architecture. The traditional approach to quantitatively evaluate design trade-offs is to use detailed cycle-based simulation [6], [7], [10], [17].

Simulators have a variety of uses. They are used for both pre-silicon and post-silicon for design evaluation and validation. They can also be used for workload characterization and software debugging/tuning [11], [19]. Simulators can provide valuable performance data running a wide range of workloads. The output of a simulation can be in the form of spreadsheets, logs, graphs, or other visual aids. A simulator can also be an educational tool. Simulators can make the complex world of computer systems much more digestible by any user. Automatically generated graphs and visualization tools are nice complements to a simulator, presenting detailed information to the user in a clear and coherent format.

A visualization tool is a framework for graphically displaying data, converting simulator output (e.g., traces, logs) into data structures and diagrams easily understandable by a user. Visualization tools have been used widely as a complementary tool for popular textbooks [3], [8], [14], [22]. By providing a visual interface, the user is compelled to more fully explore the simulation data. Visualization tools allow a user to take advantage of the rich pattern recognition capabilities of the human visual system, discovering patterns and trends within the data.

In this paper we present the Multi2sim visualization framework. This framework is an addition to Multi2sim, a recognized simulation environment for evaluating OpenCL application on a CPU, GPU, or heterogeneous system [17]. The Multi2sim visualization framework will also consider how best to be used as a teaching aid when considering CPU-GPU programming and architecture.

The Multi2sim visualization framework consists of an interactive cycle-by-cycle trace-driven visualization tool (M2S-Visual), that is built using the GTK library [15]. The framework provides a number of post-simulation built-in graphical tools for analyzing the characteristics of OpenCL kernels as run on the simulated system. M2S-Visual is an open-source tool released under GPU general public license, and is being used across the Multi2sim community. In this paper we provide

[1]This work was done before the author joined AMD

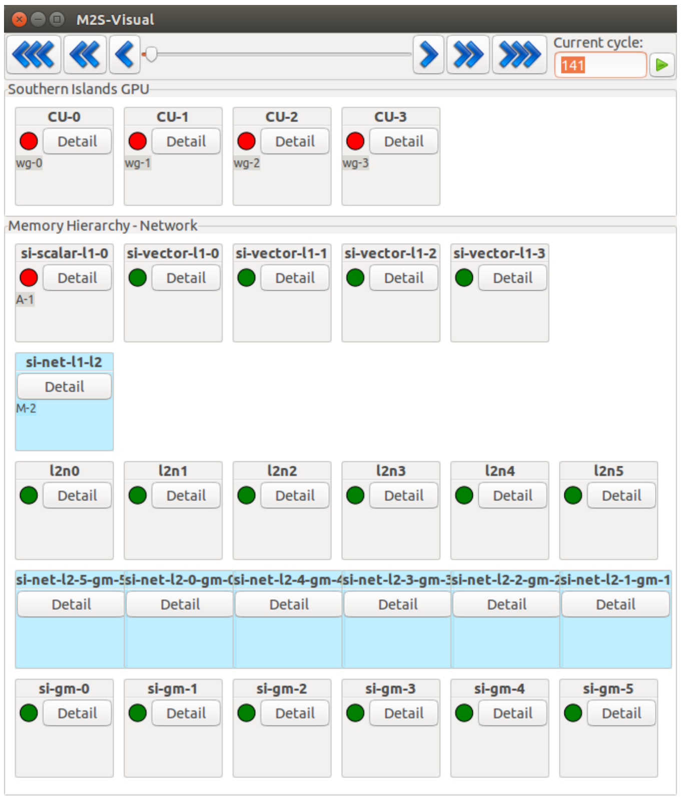
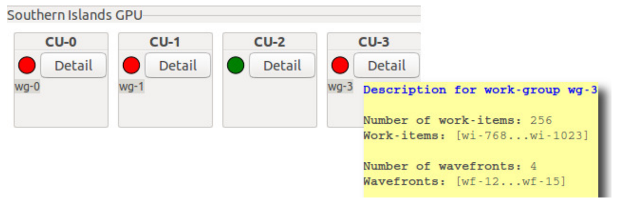


Fig. 1: The M2S-Visual main window.



```
Description for work-group wg-3

Number of work-items: 256
Work-items: [wi-768...wi-1023]

Number of wavefronts: 4
Wavefronts: [wf-12...wf-15]
```

Fig. 2: A Southern-Islands GPU, as part of M2S-Visual main window.

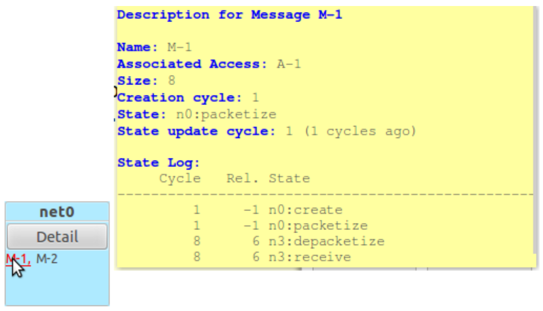
an overview of this rich toolset, and outline a number of new additions and features.

II. Multi2sim Visualization Framework

The information generated in a single trace of a short application can be in the order of million lines. Our visualization tool (M2S-Visual) is designed to translate these traces into an interactive, user-friendly, graphical representation. Multi2sim has been modified to produce post-simulation graphics that can help students and researchers explore OpenCL program execution on the heterogeneous systems.

A. *M2S-Visual: Multi2sim Interactive Visualization Tool*

M2S-Visual allows the user to reason about parallel program execution from both a software and hardware perspective. This allows our toolset to address the needs of both programmers and architects. The state of CPU and GPU software entities (i.e., contexts, work-groups, wavefronts, and work-items), memory entities (i.e., accesses, sharers, owners), and



```
Description for Message M-1

Name: M-1
Associated Access: A-1
Size: 8
Creation cycle: 1
State: n0:packetize
State update cycle: 1 (1 cycles ago)

State Log:
     Cycle   Rel. State
--------------------------------------------------
         1     -1 n0:create
         1     -1 n0:packetize
         8      6 n3:depacketize
         8      6 n3:receive
```

Fig. 3: Pop-up window, showing the properties and log of an in-flight message on the network.

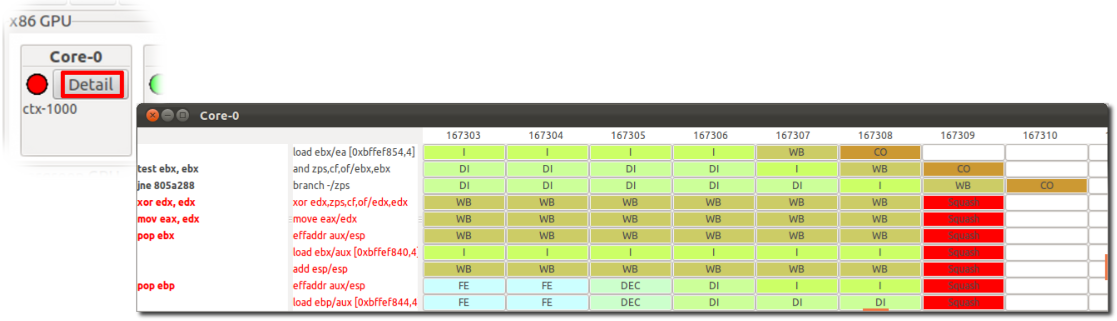
network entities (i.e., messages and packets) are captured in M2S-Visual, along with the state of the CPU and GPU hardware resources (i.e., cores, compute units), memory hierarchy (i.e., L1 cache, L2 cache and the main memory), and network resources (i.e., nodes, buses, links and buffers). The main window shows an overview of the CPU, GPU, memory and networks. Secondary windows can be launched, allowing the user to navigate through the simulation results.

Figure 1 shows the main window of M2S-Visual. The main window has four components: 1) a cycle bar which allows the navigation through different cycles, 2) a CPU panel which shows the current state of the compute cores and the list of contexts running on those cores, 3) a GPU panel which shows the status of compute units and the list of work-groups running on those compute units, and 4) a memory hierarchy and network panel. The availability of CPU and GPU panel depends on the type of detailed simulation that was run. The CPU panel is available for detailed CPU simulations, while the GPU panel becomes available for GPU detailed simulations. During any cycle, there can be an active work-group for each Southern Islands compute unit. Information on the active work-groups can be obtained by clicking their corresponding labels, as shown in Figure 2.

In the Memory Hierarchy and Network panels, each gray board represents a cache or main memory module (directory), along an indication of which modules are currently busy serving accesses. The list of accesses currently being served is shown below. Each blue board represents a network between higher level memory modules and lower level memory modules. The list of messages traversing the network is shown as a list on the board. The Detail buttons on the boards show the detailed state of each module or network. The label representing an access or message can be clicked on to provide information about the state and history of that access or the message (Figure 3).

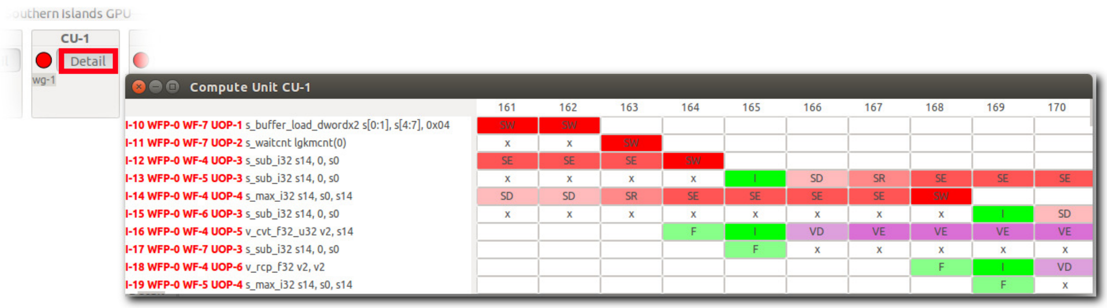
For each CPU core and GPU compute unit, a timing diagram can be generated and navigated by pressing the Detail button on the main window, as shown in Figure 4. Multiple timing diagrams can be opened at the same time period running on multiple computing elements (cores or compute units). All execution is synchronized whenever the cycle navigation control on the main window is updated.

M2S-Visual support both X86 multi-core execution and GPU many-core parallel execution. Figure 4(a) shows a CPU



		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	

(a) Timing diagram for the x86 CPU core pipeline.



	161	162	163	164	165	166	167	168	169	170
I-10 WFP-0 WF-7 UOP-1 s_buffer_load_dwordx2 s[0:1], s[4:7], 0x04	SW	SW								
I-11 WFP-0 WF-7 UOP-2 s_waitcnt lgkmcnt(0)	x	x	SW							
I-12 WFP-0 WF-4 UOP-3 s_sub_i32 s14, 0, s0	SE	SE	SE	SW						
I-13 WFP-0 WF-5 UOP-3 s_sub_i32 s14, 0, s0	x	x	x	x	I	SD	SR	SE	SE	SE
I-14 WFP-0 WF-4 UOP-4 s_max_i32 s14, s0, s14	SD	SD	SR	SE	SE	SE	SE	SW		
I-15 WFP-0 WF-6 UOP-3 s_sub_i32 s14, 0, s0	x	x	x	x	x	x	x	x	I	SD
I-16 WFP-0 WF-4 UOP-5 v_cvt_f32_u32 v2, s14				F	I	VD	VE	VE	VE	VE
I-17 WFP-0 WF-7 UOP-3 s_sub_i32 s14, 0, s0					F	x	x	x	x	x
I-18 WFP-0 WF-4 UOP-6 v_rcp_f32 v2, v2								F	I	VD
I-19 WFP-0 WF-5 UOP-4 s_max_i32 s14, s0, s14									F	x

(b) Timing diagram for Southern-Islands GPU compute-unit pipeline.

Fig. 4: Timing diagrams of pipeline of different architectures, currently supported in M2S-Visual.

timing diagram which contains three columns. The left column is the x86 instructions in flight. The middle column contains the micro-instructions generated for the x86 instructions. Though the micro-instructions are effectively generated in the decode stage on a real machine, the timing diagram shows them starting in the fetch stage, as soon as the cache block containing the associated x86 instruction is fetched from the instruction cache. The right column contains a table, representing the pipeline stage where a micro-instruction is located during each cycle. Speculative x86 instructions and micro-code in the two left-most columns are colored red. These instructions are squashed at the time a mispeculated branch reaches the write-back or commit stage, depending on the configuration for branch resolutions. More information on the x86 CPU timing diagram is presented in [18].

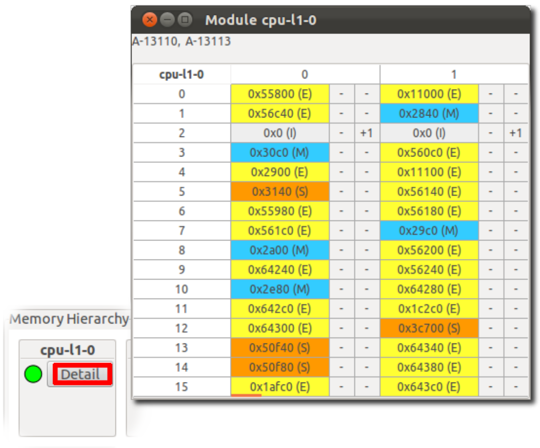
Figures 4(b) shows a timing diagram generated for Southern Islands GPU compute units (M2S-Visual also supports a pipeline diagram for AMD's Evergreen architecture – see [18]). Each OpenCL kernel is compiled to Southern Islands ISA instructions. The left column in the diagram describes which work-group, wave-front and work-item each instruction belongs. The middle column is the machine code of instructions running on the compute unit, in the same order that they were fetched. The columns in the right table represent cycles, while the rows correspond to instructions in flight. The contents of each table cell represents the pipeline stage where instructions are located.

In the Southern Islands compute unit, fetch and issue of the instructions are performed in the front-end unit. The duration of the fetch stage depends on the type of the instruction and the parameters of the front-end, specified in the configuration file. The fetch and issue stages are shown in the pipeline diagram as (F) and (I), respectively. The (x) marks show the duration of the fetch operation. After issue, the instruction can continue its path in one of five different pipelines.

The scalar pipeline is responsible for executing scalar arithmetic logic and scalar memory instructions. The branch pipeline is responsible for executing control flow instructions. The SIMD pipeline is responsible for executing vector arithmetic logic instructions. There can be multiple SIMD pipelines for each compute unit. Each SIMD pipeline can have multiple lanes. The maximum number of instructions executed in each cycle depends on the number of lanes (see the Multi2sim guide [18] and the AMD GCN specification [2] for more details). The number of SIMD pipelines and the number of lanes are configurable in Multi2sim configuration file. The Load-Data-Share (LDS) pipeline is responsible for handling all local memory instructions. The last pipeline is the vector memory unit, which is responsible for handling all vector global memory operations.

Each pipeline is color-coded in M2S-Visual to provide ease of differentiation between different instructions. This allows the user to visually identify different phases within the execution of an OpenCL kernel.

The simulated memory hierarchy panel is structured in levels of caches or memory modules, where a higher module is a module closer to the processor. Figure 5 is an example of the visual representation of a cache module when the Detail button of the cache board on the main window is clicked. In this example, the cache unit is a 2-way set-associative L1 cache with 16 sets. Each cell is a cache block, containing a tag and



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)	-	-

Fig. 5: A visual representation of a cache in M2S-Visual.

a state. The state can be one of the five states in the MOESI cache coherence protocol, and it determines the color of the cell. The two smaller cells on the right of the tag represent the number of sharers in the upper-level cache for this block, and the number of in-flight accesses for the block, respectively. Detailed information on an access and its history, and owners and sharers of a cache-block can be obtained by clicking the corresponding labels and cells [18].

The network boards is a new feature added to the main panel of M2S-Visual, as shown as blue boards in Figure 1. The networks in the main panel have message granularity. This means, the board contains a list of messages that traverse the network. Each message in the network is associated with an access, associated with a cache control message or a cache block access. Figure 6 shows an example of the visual window of a simulated network when the Detail button of the network board is pressed. The blue boards represent end-nodes and pink boards represent switches and buses in the network. For system simulation, each end-node is connected to a cache unit. In this case, the name of each cache unit is presented in brackets on the node board, following the node name itself.

The network graph is generated based on a general layered drawing algorithm, as presented by Tollis et al. [16]. Nodes are interconnected with uni-directional or bi-directional links. A color spectrum from, green to red, is used to show utilization of each link during every cycle. The lower a link is utilized, the closer its color is to bright green. In the case a link is idle during a cycle, the link is colored gray.

Each node in the network is equipped with a Detail button that, once clicked, opens a separate window that describes the contents of each buffer of the switch or end-node. Figure 7 shows the window for an arbitrary node. The node window in M2S-Visual has a packet-based granularity. Every message is divided to smaller packets in the end-nodes. Packets traverse the network components (i.e., buffers and links). The size of the packets can be defined in the network configuration file. The node window contains a table which includes a list of the node's buffers, their type (input and output), their size, their current occupancy, the corresponding link denoting which buffer it is connected to, and the overall utilization of that link (from the beginning of the simulation). The corresponding boxes in the same row of each buffer, represent packets in each buffer in the current cycle.

When a packet is at the head of the buffer, there are multiple reasons that can stall the packet from traversing the link to the next node in the next cycle. A packet can be stalled because the destination buffer is busy reading another packet (DBB), the destination buffer is full (DBF), the link that the packet has to traverse is busy (LB), of possibly /it it is not this packet's turn as determined by bus arbitration (BA), switch arbitration (SA) or virtual-channel arbitration (VCA). These reasons are highlighted with different colors in the node window.

The Multi2sim simulator also provides a tool to stress any individual interconnection network, as defined in the configuration file, using synthetic traffic. This tool is referred to as the stand-alone network simulator, and is integrated with the rest of the simulator functionality. Recent additions to network visualization in the M2S-Visual tool allow the user to visualize stand-alone networks.

B. *Multi2sim Post-Simulation Visualizations of Data*

M2S-Visual can provide a highly detailed record of the impact of an OpenCL kernel execution on different components within the targeted micro-architecture. In many instances, a summary of the kernel execution is sufficient. Therefore, Multi2sim is equipped to provide static graphs that show the performance of different components of the targeted system. Multi2sim uses *gnuplot* to produce these graphical reports [].

1) The GPU Occupancy Graphs: There is a limit on the number of OpenCL software elements that can run on a Southern Islands GPU, referred to here as *compute unit occupancy*. Multi2sim can provide occupancy graphics based on static and run-time characteristics of the executed OpenCL kernels.

Figure 8 shows these post-simulation graphs for a Southern Islands compute unit while it is running a matrix multiplication kernel. In these graphs, the red dots indicate the values used for the simulation. The blue dots are produced by the simulator to help users choose work-group sizes for their kernel in order to achieve the highest occupancy of their GPU designs. All three graphs should be used collectively for optimizing the kernel (see Section III-B for an example). The GPU occupancy for the Southern Islands compute unit is a new addition to Multi2sim. The previous release of Multi2sim provided the same occupancy graphs for GPUs based on the Evergreen architecture. A similar tool is provided by NVIDIA to compute the multiprocessor occupancy of a GPU for a given CUDA kernel [1].

Figure 8(a) shows the number of registers used for each work-item and its relation to the number of wave-fronts for each SIMD unit. The total number of registers in a compute unit is limited. If a work-item uses too many registers, it will eventually prevent other wave-fronts from being executed concurrently. The number of registers per work-item is determined statically at compile time, and does not depend on any run-time decisions by the OpenCL host program.

Figure 8(b) shows the amount of local memory used for each work-group and its relation to the number of wave-

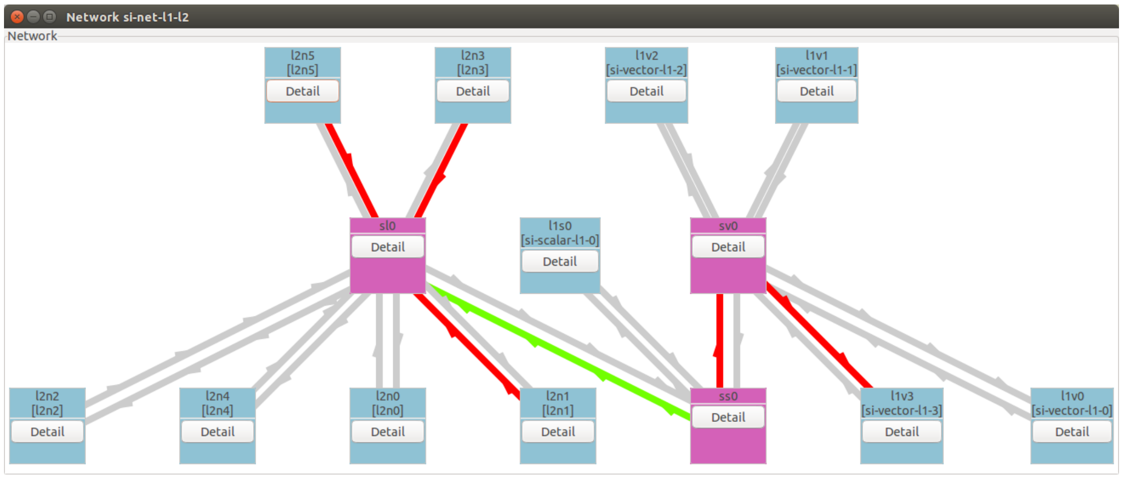
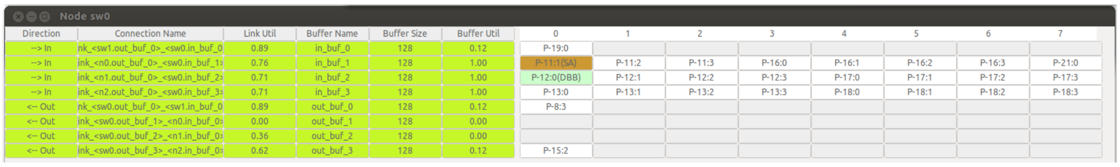


Fig. 6: A visual representation of a network in M2S-Visual.


Node sw0

Direction	Connection Name	Link Util	Buffer Name	Buffer Size	Buffer Util	0	1	2	3	4	5	6	7
--> In	nk_<sw1.out_buf_0>_<sw0.in_buf_0	0.89	in_buf_0	128	0.12	P-19:0							
--> In	ink_<n0.out_buf_0>_<sw0.in_buf_1>	0.76	in_buf_1	128	1.00	P-11:1(SA)	P-11:2	P-11:3	P-16:0	P-16:1	P-16:2	P-16:3	P-21:0
--> In	ink_<n1.out_buf_0>_<sw0.in_buf_2>	0.71	in_buf_2	128	1.00	P-12:0(DBB)	P-12:1	P-12:2	P-12:3	P-17:0	P-17:1	P-17:2	P-17:3
--> In	ink_<n2.out_buf_0>_<sw0.in_buf_3>	0.71	in_buf_3	128	1.00	P-13:0	P-13:1	P-13:2	P-13:3	P-18:0	P-18:1	P-18:2	P-18:3
<-- Out	nk_<sw0.out_buf_0>_<sw1.in_buf_0	0.89	out_buf_0	128	0.12	P-8:3							
<-- Out	ink_<sw0.out_buf_1>_<n0.in_buf_0>	0.00	out_buf_1	128	0.00								
<-- Out	ink_<sw0.out_buf_2>_<n1.in_buf_0>	0.36	out_buf_2	128	0.00								
<-- Out	ink_<sw0.out_buf_3>_<n2.in_buf_0>	0.62	out_buf_3	128	0.12	P-15:2							

Fig. 7: A visual representation of a node in M2S-Visual.

fronts for each SIMD unit. A compute unit has also a limited amount of local memory. When the compute unit allocates a work-group that needs more local memory, it will reduce the total number of wave-fronts that can be allocated to the same compute unit. The shape of this curve depends on the local memory available on a Southern Islands compute unit, the local memory used by each work-group, the wavefront/work-group sizes, and the memory allocation chunk size.

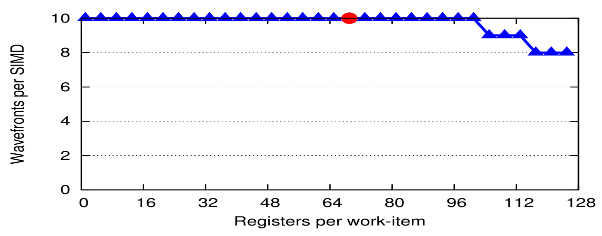
Figure 8(c) shows the total number of wave-fronts that can be allocated to a work-group and its relation to the number of wave-fronts for each SIMD unit. Since the allocation unit for a compute unit is an entire work-group (set of wave-fronts), this plot shows distinct peaks instead of a continuously decreasing slpe. The work-group size is determined at run-time by the OpenCL host program.

2) Graphical Representation of Memory Accesses: The memory system is typically one of the most dominant sources of performance loss on a computer architecture. If we can provide a detailed record of the memory access behavior present in OpenCL applications and carefully characterize that behavior, this can be extremely valuable when tuning memory performance. Multi2sim has been modified to produce graphical output to characterize memory access patterns present in OpenCL applications. For every execution, Multi2sim can generate two sets of graphs to present application patterns for load and store accesses. Figure 9 shows example graphs for load accesses present in an OpenCL matrix multiplication kernel (multiplying a 64×64 matrix by a 64×16 matrix).

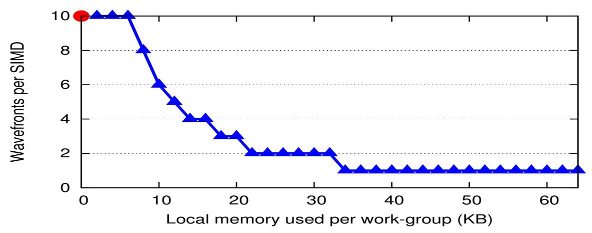
Multi2sim accepts two arguments to create these graphs. The first argument is the memory size granularity, in bytes. Multi2sim, then divides the memory address space into blocks, with the size set by the input argument. Every access to that address range is recorded by Multi2sim. This argument can be set to any size, (e.g., a single cache line (typically 64B), the page size (typically 4KB), etc.). The second argument is the sampling interval, specified in cycles. Multi2sim breaks down the overall execution time into time slots that are equal in length to this interval, and records the number of accesses to different memory blocks (as determined by the first argument). Multi2sim outputs 2-D and 3-D graphs that show the recorded values recorded for each interval (Figure 9).

3) Graphical Representation of Network Bandwidth: Network performance can be critical, especially as we increase the number of functional units or compute units on a single device. Visual analysis of network transactions will be very helpful if we want to understand how an application utilizes the network resources. Researchers often perform detailed analysis on network bandwidth for network systems (e.g., LANs and WANs). The interconnection network within a computing system in no different. Detailed analysis of the bandwidth required by different applications can guide the design to insure that we have a performant and cost-effective network. Multi2sim produces graphical output that reports the required bandwidth of the application, sampled across it's execution.

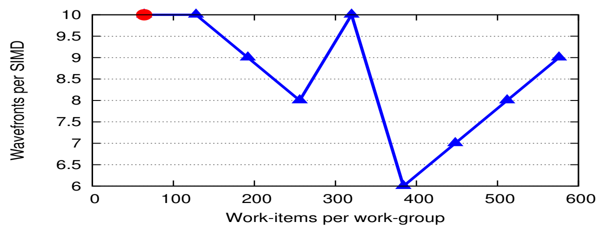
Figure 10 shows two examples of network reporting. Figure 10(a) shows the network bandwidth consumed between the L1 and L2 cache units by an application from the Splash2 benchmark suite [21], that performs a radix sort on an array of 262,144 elements, executed on a many-core CPU system with 32 CPUs. Figure 10(b) shows the network bandwidth between the L1 and L2 cache units consumed by an OpenCL application from the AMDAPP SDK benchmark [5]. This



(a)Number of registers used per work-item.

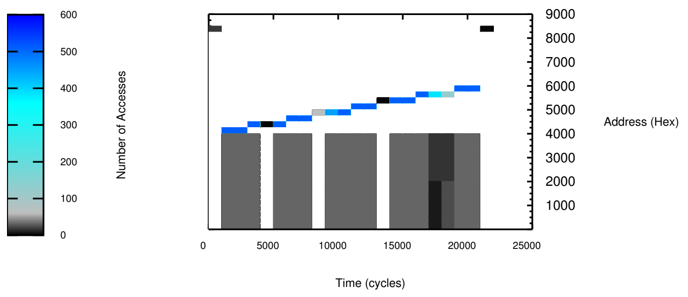


(b)Local memory size per work-group.

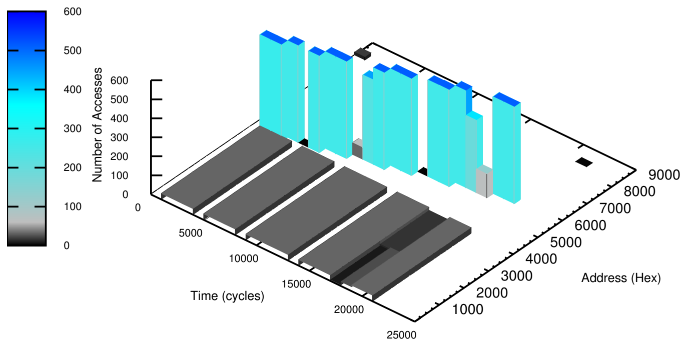


(c)Work-group size.

Fig. 8: Output GPU occupancy plots generated by Multi2sim



(a) 2D plot for load access pattern of an application during execution



(b) 3D plot for load access pattern of an application during execution

Fig. 9: Example of a memory access plot generated by Multi2sim. In this example, the memory size granularity is 1KB and time interval is 1000 cycles.

workload performs a radix sort on an array with the same size as AMD's Southern Islands GPU system with 32 compute units. In both cases, the network bandwidth is logged in 1000 cycles intervals.

By comparing the two figures, one can clearly see the impact different applications can have on the CPU and GPU network and memory resources (the network resides between the L1 and L2 memory units). This different highlights the different classes of parallelism (SPMD vs. SIMD) implemented in these applications.

III. Applications

In this section we cover example uses of the visualization options that are available in the Multi2sim simulator. We also discuss how Multi2sim's visualization platform compares to other existing visualization tools.

A. Teaching

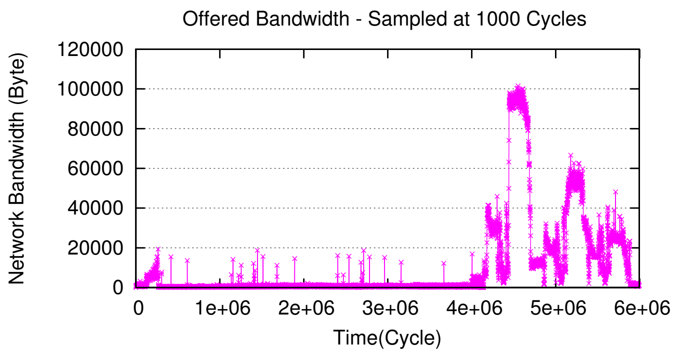
Having the ability to show how parts of several instructions are executed in parallel is a key aspect of teaching instruction pipelining [8], [14], [22]. One goal of the M2S-Visual Tool in Multi2sim is to provide a simple way to familiarize students with the design of the state-of-the-art CPU and GPU instruction pipelines. While different simulators exist to visualize many of the complex aspects of superscalar CPU pipelines, to the best of our knowledge, M2S-Visual is the first to provide details of the pipelines of different GPU architectures. Hence the M2S-Visual can be used as suitable complimentary tool to teach accelerator architecture concepts.

M2S-Visual can also be used to learn about the memory hierarchy and coherency protocols of CPU and GPU systems. The unified memory system is a unique benefit of the M2S-Visual tool. Multi2sim implements a modified MOESI protocol, with an extra state added for non-coherent reads and writes (please see [18]). This memory protocol is supported by M2S-Visual and allows the user to characterize CPU/GPU memory interaction, as well as the individual memory transactions of each devices.

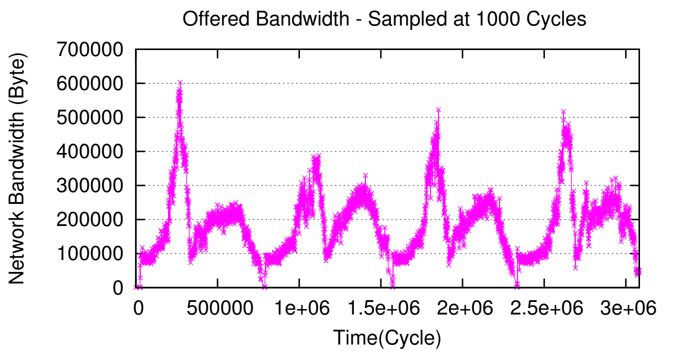
The memory system in M2S-Visual also allows the user to identify the source of an access, which is helpful when it comes to educational use of this tool. This also differentiates M2S-Visual from other trace-based memory visual tools that are specifically designed for studying cache memory [12].

By adding visulization of the interconnection network into M2S-Visual, we enable the user to full life cycle of a memory operation. M2S-Visual shows students how different interconnects between memory components can impact the final execution time of a memory operation, or of subsequent dependent operations. It also provides the possibility to explore tradeoffs in interconnection topologies in order to increase overall system performance.

Without a visualization tool, interconnection network designer may need to analyze complex message exchanges, and characterize dynamic transactions with competing traffic in the network. Many network simulators (such as [4], [9]) generate detailed traces, but they provide limited help for analyzing this



(a) Traffic of a network between L1 and L2 cache units of a manycore CPU system.



(b) Traffic of a network between the L1 and L2 cache units of a GPU system.

Fig. 10: Example of generated network traffic plot by Multi2sim.

data. To the best of our knowledge, M2S-Visual is the first visualization tool with packet-level granularity to model the interconnection network of the simulated computer systems.

B. Research

M2S-Visual and Multi2sim post-simulation visualization extensions can be a great source of inspiration for researchers.

1) Case 1: OpenCL Programing and Compiler Research: Common practice in application programing is to make use of more number of registers, as registers are the fastest memory units. While this is completely true for many-core CPU systems, it does not apply to GPUs. In a compute unit, registers are limited and shared between multiple work-items. If a work-item uses a larger number of registers, fewer work-items can be placed on that compute unit. At the same time, if instead of registers more local memory is used, then the program might not fully benefit from the available registers. Depending on the parameters of the compute unit, allowing controlled register spilling can lead to optimization of the written kernel. The Multi2sim GPU occupancy calculator (Section II-B1) provides the user the possibility to tweak the any kernel (e.g., using loop unrolling) and achieve balance between local memory and the number of registers used.

2) Case 2: Memory Design Research: Characterization of memory access patterns for different programs has been always a driving force for research in memory systems. Temporal and spatial locality can be characterized, and can provide benefits in when designing the memory hierarchy. CPU monitoring frameworks such as Owl [13] have incorporated the production of histograms of memory access patterns present in applications.

By using memory access histograms (Section II-B2), we can quickly exploit these memory access patterns and apply learning algorithms to more carefully label a stream as *regular* or *irregular*.

3) Case 3: Interconnection Network Research: Research on interconnection networks of many-core CPUs and GPUs can be facilitated by using the combination of memory access patterns graphs (Section II-B2), network traffic patterns graphs (Section II-B3) and analysis of network resource utilization (via M2S-Visual and link-utilization graph introduced in [18]). Researcher can identify memory access patterns, correlate these patterns to the traffic of a particular network between levels of the memory hierarchy, and based on these correlations, modify the network design to accelerate program execution [23].

IV. Conclusion and Future Work

In this paper we introduced the Multi2sim visualization framework. The M2S-Visual Tool is designed as an educational resource to familiarize students with parallel code execution on both CPUs and GPUs. A novel aspect of M2S-Visual is that the toolset provides detailed inspection of OpenCL execution on CPU and GPU architectures, including the memory hierarchy and the interconnection network.

We also presented recent features added to Multi2sim which support the creation of various graphical representations in order to help students and researchers better understand the impact of parallel OpenCL execution on hardware resources. We provided a number of examples showing how we can take advantage of these additions. Our hope is to be able to continue this effort and to provide more informative graphical tools to aid students and researchers.

In the future, we are planning to pursue multiple paths for the M2S-Visual visualization framework. Since trace data can grow very large when executing large applications, we need to provide the option to stop the execution, fast-forward the execution and to restart the execution, starting at any user-specified cycle. This allows the user to take a peak at any desired window of application execution. Multi2sim has recently added support for a detailed DRAM model, which will also be incorporated into the M2S-Visual toolset. To further enhance the network-on-chip simulation in M2S-Visual, we plan to add detailed pipeline simulation of the network routers, adding topology detection mechanisms for network graphs. One other path we are considering is how to incorporate power modeling data into this same framework, as well as reliability data (Mult2sim provides for fault injection presently [20]). Finally, we plan to add visual panels for local memory and register files on the GPU.

References

[1] "Nvidia gpu occupancy calculator," ttp://developer.download.nvidia.com/.

[2] "AMD Graphics Cores Next (GCN) Architecture," June 2012, white paper.

[3] G. Adams, "Dlxview–(preliminary) user manual," *http://yara.ecn.purdue.edu/teamaaa/dlxview*.

[4] N. Agarwal, T. Krishna, L.-S. Peh, and N. K. Jha, "Garnet: A detailed on-chip network model inside a full-system simulator," in *Performance Analysis of Systems and Software, 2009. ISPASS 2009. IEEE International Symposium on*. IEEE, 2009, pp. 33–42.

[5] AMD, *AMD Accelerated Parallel Processing OpenCL Programming Guide*. http://developer.amd.com/GPU/AMDAPPSDK/, Jan. 2011.

[6] A. Bakhoda, G. L. Yuan, W. W. Fung, H. Wong, and T. M. Aamodt, "Analyzing cuda workloads using a detailed gpu simulator," in *IEEE International Symposium on Performance Analysis of Systems and Software, 2009*. IEEE, 2009, pp. 163–174.

[7] D. Burger and T. M. Austin, "The simplescalar tool set, version 2.0," *ACM SIGARCH Computer Architecture News*, vol. 25, no. 3, pp. 13–25, 1997.

[8] M. I. Garcia, S. Rodríguez, A. Pérez, and A. García, "p88110: A graphical simulator for computer architecture and organization courses," *Education, IEEE Transactions on*, vol. 52, no. 2, pp. 248–256, 2009.

[9] A. B. Kahng, B. Li, L.-S. Peh, and K. Samadi, "Orion 2.0: A power-area simulator for interconnection networks." Institute of Electrical and Electronics Engineers, 2011.

[10] M. Marty, B. Beckmann, L. Yen, A. Alameldeen, M. Xu, and K. Moore, "Gems: Multifacet's general execution-driven multiprocessor simulator," in *International Symposium on Computer Architecture*, 2006.

[11] M. Mohiyuddin, "Tuning hardware and software for multiprocessors," Ph.D. dissertation, University of California, Berkeley, 2012.

[12] M. Á. V. Rodriguez, J. M. S. Pérez, and J. A. G. Pulido, "An educational tool for testing caches on symmetric multiprocessors," *Microprocessors and Microsystems*, vol. 25, no. 4, pp. 187–194, 2001.

[13] M. Schulz, B. S. White, S. A. McKee, H.-H. S. Lee, and J. Jeitner, "Owl: next generation system monitoring," in *Proceedings of the 2nd conference on Computing frontiers*. ACM, 2005, pp. 116–124.

[14] D. Skrien, "Cpu sim 3.1: A tool for simulating computer architectures for computer organization classes," *Journal on Educational Resources in Computing (JERIC)*, vol. 1, no. 4, pp. 46–59, 2001.

[15] G. Team, "The gtk+ project."," http://www.gtk.org.

[16] I. Tollis, P. Eades, G. Di Battista, and L. Tollis, *Graph drawing: algorithms for the visualization of graphs*. Prentice Hall New York, 1998, vol. 1.

[17] R. Ubal, B. Jang, P. Mistry, D. Schaa, and D. Kaeli, "Multi2sim: A simulation framework for cpu-gpu computing," in *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*. ACM, 2012, pp. 335–344.

[18] R. Ubal, J. Sahuquillo, S. Petit, P. Lopez, Z. Chen, and D. R. Kaeli, "The multi2sim simulation framework: A cpu-gpu model for heterogeneous computing," https://www.multi2sim.org.

[19] D. Uluski, M. Moffie, and D. Kaeli, "Characterizing antivirus workload execution," *ACM SIGARCH Computer Architecture News*, vol. 33, no. 1, pp. 90–98, 2005.

[20] M. Wilkening, V. Sridharan, S. Li, F. Previlon, S. Gurumurthi, and D. R. Kaeli, "Calculating architectural vulnerability factors for spatial multi-bit transient faults," in *Microarchitecture (MICRO), 2014 47th Annual IEEE/ACM International Symposium on*. IEEE, 2014, pp. 293–305.

[21] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, "The splash-2 programs: Characterization and methodological considerations," in *ACM SIGARCH Computer Architecture News*, vol. 23, no. 2. ACM, 1995, pp. 24–36.

[22] Y. Zhang and G. B. Adams III, "An interactive, visual simulator for the dlx pipeline," in *Proceedings of the 1997 workshop on Computer architecture education*. ACM, 1997, p. 2.

[23] A. K. Ziabari, J. L. Abéllan, R. Ubal, C. Chen, A. Joshi, and D. Kaeli, "Leveraging silicon-photonic noc for designing scalable gpus," in *29th International Conference on Supercomputing*. ACM, 2015.