Enabling GPGPU Low-Level Hardware Explorations with MIAOW: An Open-Source RTL Implementation of a GPGPU

RAGHURAMAN BALASUBRAMANIAN, VINAY GANGADHAR, ZILIANG GUO, CHEN-HAN HO, CHERIN JOSEPH, and JAIKRISHNAN MENON,
University of Wisconsin-Madison
MARIO PAULO DRUMOND, University of Wisconsin-Madison and Universidade Federal de Minas Gerais
ROBIN PAUL, SHARATH PRASAD, PRADIP VALATHOL, and KARTHIKEYAN SANKARALINGAM, University of Wisconsin-Madison

Graphic processing unit (GPU)-based general-purpose computing is developing as a viable alternative to CPU-based computing in many domains. Today's tools for GPU analysis include simulators like GPGPU-Sim, Multi2Sim, and Barra. While useful for modeling first-order effects, these tools do not provide a detailed view of GPU microarchitecture and physical design. Further, as GPGPU research evolves, design ideas and modifications demand detailed estimates of impact on overall area and power. Fueled by this need, we introduce MIAOW (Many-core Integrated Accelerator Of Wisconsin), an open-source RTL implementation of the AMD Southern Islands GPGPU ISA, capable of running unmodified OpenCL-based applications. We present our design motivated by our goals to create a realistic, flexible, OpenCL-compatible GPGPU, capable of emulating a full system. We first explore if MIAOW is realistic and then use four case studies to show that MIAOW enables the following: physical design perspective to "traditional" microarchitecture, new types of research exploration, and validation/calibration of simulator-based characterization of hardware. The findings and ideas are contributions in their own right, in addition to MIAOW's utility as a tool for others' research.

Categories and Subject Descriptors: C.1.4 [**Computer Systems Organization**]: Parallel Architectures

General Terms: Design, Performance

Additional Key Words and Phrases: GPU, manycore, SIMD

ACM Reference Format:
Raghuraman Balasubramanian, Vinay Gangadhar, Ziliang Guo, Chen-Han Ho, Cherin Joseph, Jaikrishnan Menon, Mario Paulo Drumond, Robin Paul, Sharath Prasad, Pradip Valathol, and Karthikeyan Sankaralingam. 2015. Enabling GPGPU low-level hardware explorations with MIAOW: An open-source RTL implementation of a GPGPU. ACM Trans. Architec. Code Optim. 12, 2, Article 21 (June 2015), 25 pages.
DOI: http://dx.doi.org/10.1145/2764908

New article, not an extension of a conference paper.
This work is supported by the National Science Foundation under the following grants: CCF-0845751, CCF-0917238, and CNS-0917213.
Authors' address: R. Balasubramanian, V. Gangadhar, Z. Guo, C.-H. Ho, C. Joseph, J. Menon, M. P. Drumond, R. Paul, S. Prasad, P. Valathol, and K. Sankaralingam, 1210 W. Dayton St., Madison, WI - 53706; emails: raghuraman.b@gmail.com, gangadhar@wisc.edu, ziliang@cs.wisc.edu, ho9@wisc.edu, cherin99@gmail.com, menon@cs.wisc.edu, {drumondmp, robinpaulp}@gmail.com, snprasad@wisc.edu, {pradip16, karu}@cs.wisc.edu.

DOI: http://dx.doi.org/10.1145/2764908

1. INTRODUCTION

The trend for the last several years in computer architecture has been the effort to extract more performance from available silicon. One such approach has been to exploit the massive parallelism of graphics cards and use their execution units for general computation instead of simply graphics operations. In light of this trend, much work has been done to improve GPU microarchitecture with novel ideas or modification of existing designs to boost performance or ease their use. To support this work, proper tooling is essential. To this end, we propose MIAOW as another addition to the stable of tools available to researchers.

Related Work. A variety of tools exist to support microarchitecture research for both CPUs and GPUs. On the *CPU* side, tools span performance simulators, emulators, compilers, profiling tools, modeling tools, and recently a multitude of RTL-level implementations of microprocessors; these include OpenSPARC [opensparc 2006], OpenRISC [openrisc 2010], Illinois Verilog Model [Wang and Patel 2006], LEON [Gaisler 2001], and more recently FabScalar [Choudhary et al. 2011] and PERSim [Balasubramanian and Sankaralingam 2014]. In other efforts, clean-slate CPU designs have been built to demonstrate research ideas. These RTL-level implementations allow detailed microarchitecture exploration, understanding and quantifying the effects of area and power, technology-driven studies, prototype-building studies on CPUs, exploring power-efficient design ideas that span CAD and microarchitecture, understanding the effects of transient faults on hardware structures, analyzing di/dt noise, and hardware reliability analysis. Some specific example research ideas include the following: Argus [Meixner et al. 2007] showed, with a prototype implementation on OpenRISC, how to build lightweight fault detectors; Blueshift [Greskamp et al. 2009] and power-balanced pipelines [Sartori et al. 2012] consider the OpenRISC and OpenSPARC pipelines for novel CAD/microarchitecture work.

On the *GPU* side, a number of performance simulators [bar 2009; Bakhoda et al. 2009; del Barrio et al. 2006; Leng et al. 2013], emulators [Ubal et al. 2012; bar 2009], compilers [Diamos et al. 2010; llvmcuda 2009; van der Laan 2010], profiling tools [Diamos et al. 2010; nvprof 2008], and modeling tools [Hong and Kim 2009, 2010; Kim et al. 2012; Sim et al. 2012] are prevalent. However, *RTL-level implementations and low-level detailed microarchitecture specification are lacking*. As discussed by others [Chen 2009; Jeon and Annavaram 2012; Menon et al. 2012; Rech et al. 2012; Tan et al. 2011], GPUs are beginning to see many of the same technology and device reliability challenges that have driven some of the aforementioned RTL-based CPU research topics. The lack of an RTL-level implementation of a GPU hampers similar efforts in the GPU space. As CPU approaches do not directly translate to GPUs, a detailed exploration of such ideas is required. Hence, we argue that an RTL-level GPU framework will provide significant value in exploration of novel ideas and is necessary for GPU evolution complementing the current tools ecosystem. While a few efforts have been made in the past such as the Open Graphics Project and Project VGA, we are not aware of any that actually achieved a release of the RTL code.

This article reports on the design, development, characterization, and research utility of an RTL implementation of a GPGPU called MIAOW (*acronymized as **M**any-core **I**ntegrated **A**ccelerator **O**f **W**isconsin*). MIAOW's RTL source code, simulation infrastructure, and benchmarks are available for download on github: https://github.com/VerticalResearchGroup/miaow.[1] Figure 1 shows a canonical GPGPU

[1]The source code release also includes a publicly available technical report (white paper) version of this publication. The authors have not released to any external entity the copyright to that white paper and therefore are within the guidelines of submission to TACO.

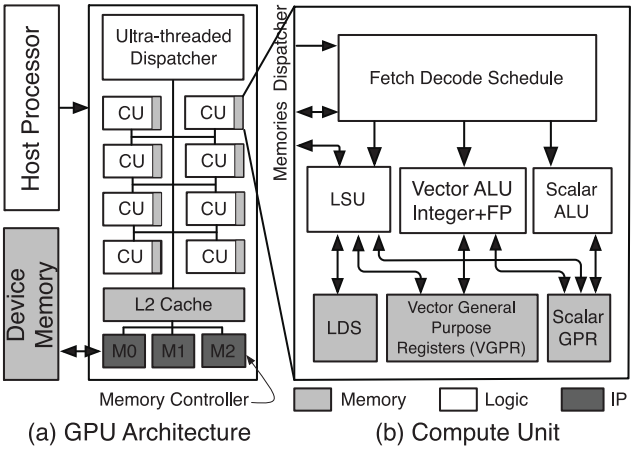

Fig. 1. Canonical GPU organization.

architecture resembling what MIAOW targets (borrowed from the AMD SI specification, we define a few GPU terms in Table III). Specifically, we focus on a design that delivers GPU compute capability and ignores graphics functionality. MIAOW is driven by the following key goals and nongoals.

Goals. The primary driving goals for MIAOW are as follows: (1) *realism:* it should be a *realistic* implementation of a GPU resembling principles and implementation tradeoffs in industry GPUs; (2) *flexible:* it should be flexible to accommodate research studies of various types and the exploration of forward-looking ideas and to form an end-to-end open-source tool; (3) *software compatible:* it should use standard and widely available software stacks like OpenCL or CUDA compilers to enable executing various applications and not be tied to in-house compiler technologies and languages.

It must be emphasized that MIAOW is not a simulator by any conventional definition. As an actual RTL implementation, MIAOW is not attempting to mimic the behavior of a GPU's microarchitecture; MIAOW *is* a GPU, and the results of program execution on it are little different than execution on a physical AMD or NVIDIA GPU.

Nongoals. We also explain nongoals that set up the context of MIAOW's capability. We do not seek to implement graphics functionality. We do not aim to be compatible with every application written for GPUs (i.e., subsetting of features is acceptable). We give ourselves the freedom of leaving some chip functionality as PLI-based behavioral RTL; for example, we do not seek to implement memory controllers, on-chip networks (OCNs), and so forth in RTL. MIAOW is *not* meant to be an ASIC implementable standalone GPGPU. Finally, being competitive with commercial designs was a *nongoal*.

Driven by these goals and nongoals, we have developed MIAOW as an implementation of a subset of AMD's Southern Islands (SI) ISA [amd 2012b]. While we pick one ISA and design style, we feel it is representative of GPGPU design [Fried 2012]—AMD and NVIDIA's approaches have some commonalities [Zhang et al. 2011]. This delivers on all three primary goals. It is a real ISA (machine's internal ISA compared to PTX or AMD-IL, which are external ISAs found in products launched in 2012), is a clean-slate design so likely to remain relevant for a few years, and has a complete ecosystem of OpenCL compilers and applications. In concrete terms, MIAOW focuses on microarchitecture of compute units (CUs) and implements them in synthesizable Verilog RTL and leaves the memory hierarchy and memory controllers as behavioral (emulated) models.

Figure 2 describes a spectrum of implementation strategies and the tradeoffs. We

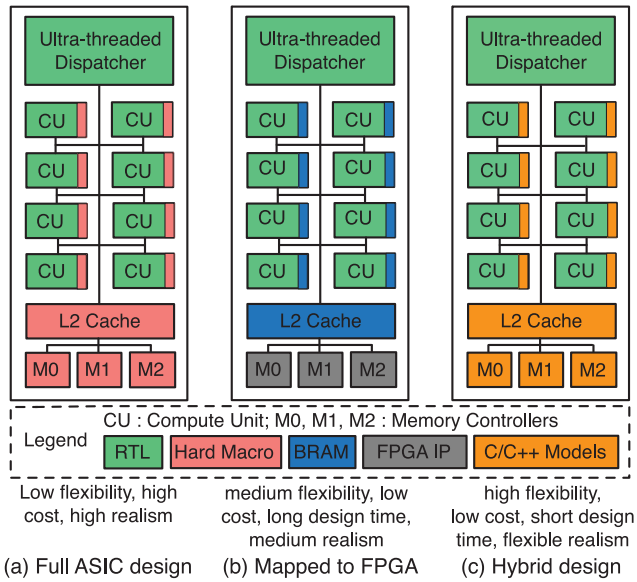

Fig. 2. Target platforms for a GPGPU RTL implementation.

Table I. MIAOW RTL Versus State-of-the-Art Products (Radeon HD)

Name	# Cores	GFLOPS	Core Clock	Tech Node	Freq. in FO4 Delay
7350 (Jan '12)	80	104	650MHz	40nm	-
7990 (Apr '13)	4,096	8,192	1GHz	28nm	-
MIAOW (Now)	64-2,048	57-1,820	222MHZ	32nm	310

show how a canonical GPU organization can be implemented in these three strategies (the shaded portion of the CU denotes the register file and SRAM storage as indicated in Figure 1(b)). First, observe that in all three designs, the register files need some special treatment besides writing Verilog RTL. A full ASIC design results in reduced flexibility, long design cycle, and high cost and makes it a poor research platform, since memory controller IP and hard macros for SRAM and register files may not be redistributable. Synthesizing for FPGA sounds attractive, but there are several resource constraints that must be accommodated and that can impact realism. In the hybrid strategy, some components, namely, L2 cache, OCN, and memory controller, are behavioral C/C++ modules. This strikes a good balance between realism, flexibility, and a framework that can be released. MIAOW takes this third approach. A modified design can also be synthesized for the Virtex7 FPGA, though its limitations will be discussed in a later section. Table I compares MIAOW to state-of-the-art commercial products. Our contribution includes methodological techniques and ideas.

Methodology. Methodologically, we provide a detailed microarchitecture description and design tradeoff of a GPGPU. We also demonstrate that MIAOW is realistic along with characterization and comparison of area, power, and performance to industry designs. MIAOW was not designed to be a replica of existing commercial GPGPUs. Building a model that is an exact match of an industry implementation requires reverse engineering of low-level design choices and hence was not our goal. The aim when comparing MIAOW to commercial designs was to show that our design is reasonable and that the quantitative results are in similar range. We are not quantifying accuracy since we are defining a new microarchitecture and thus there is no reference to compare

Table II. Case Studies Summary

Direction	Research Idea	MIAOW-Enabled Findings
Traditional μarch	Thread-block compaction	○ Implemented TBC in RTL ○ Significant design complexity ○ Increase in critical path length
New directions	Circuit-failure prediction (Aged-SDMR)	○ Implemented entirely in μarch ○ Idea works elegantly in GPUs ○ Small area, power overheads
	Timing speculation (TS)	○ Quantified TS error rate on GPU ○ TS framework for future studies
Validation of simulator studies	Transient fault injection	○ RTL-level fault injection ○ More gray area than CPUs (due to large RegFile) ○ More silent structures

Table III. Definition of Southern Islands ISA Terms and Correspondence to NVIDIA/CUDA Terminology

SI Term	NVIDIA Term	Description
Compute unit (CU)	SM	A compute unit is the basic unit of computation and contains computation resources, architectural storage resources (registers), and local memory.
Workitem	Thread	The basic unit of computation. It typically represents one input data point. Sometimes referred to as a "thread" or a "vector lane."
Wavefront	Warp	A collection of 64 work items grouped for efficient processing on the compute unit. Each wavefront shares a single program counter.
Workgroup	Thread-block	A collection of work items working together, capable of sharing data and synchronizing with each other. Can consist of more than one wavefront but is mapped to a single CU.
Local data share (LDS)	Shared memory	Memory space that enables low-latency communication between work items within a workgroup, including between work items in a wavefront. Size: 32KB limit per workgroup.
Global data share (GDS)	Global memory	Storage used for sharing data across multiple workgroups. Size: 64KB.
Device memory	Device memory	Off-chip memory provided by DRAM possibly cached in other on-chip storage.

to. Instead, we compare to a nearest neighbor to show trends are similar. Further, the RTL, entire tool suite, and case study implementations are released as open source.

Ideas. In terms of ideas, we examine three perspectives of MIAOW's transformative capability in advancing GPU research as summarized in Table II. First, it adds a *physical design perspective to "traditional" microarchitecture* research; here we revisit and implement in RTL a previously proposed warp scheduler technique [Fung and Aamodt 2011] called thread block compaction to understand the design complexity issues. Or, put another way, we see if an idea (previously done in high-level simulation only) still holds up when considering its "actual" implementation complexity. The second perspective is *new types of research exploration*, thus far infeasible for GPU research (in academia); here we look at two examples: (1) we take the Virtually Aged Sampling-DMR [Balasubramanian and Sankaralingam 2013] work proposed for fault prediction in CPUs and implement a design for GPUs and evaluate complexity, area, and power overheads; and (2) we examine the feasibility of timing speculation and its error rate/energy savings tradeoff. The final perspective is *validation/calibration of simulator-based characterization of hardware*. Here, we perform transient fault injection analysis and compare our findings to simulator studies.

The article is organized as follows. Section 2 describes the MIAOW design and architecture, Section 3 describes the implementation strategy, and Section 4 investigates the question of whether MIAOW is realistic. Sections 5, 6, and 7 investigate case studies

Table IV. Supported ISA

Type	Instructions	
Vector	ALU: {U32, I32, F32}	add, addc, sub, mad, madmk, mac, mul, max, max3, min, subrev
	Bitwise: {B32}	and, or, xor, not, mov, lshrrev, lshlrev, ashlrev, ashrrev, bfe, bfi, cndmask
	Compare: {U32, I32, F32}	cmp_{ lt, eq, le, gt, lg, ge, ne, ngt, neq }
Scalar	ALU: {U32, I32}	add, addk, sub, max, min, mul, mulk
	Bitwise: {B32}	and, andn2, or, xor, not, mov, movk, lshl, lshr, ashr, saveexec
	Compare: {U32, I32}	cmp_{ eq, lt, gt, ge, lt, le, eq, lg, gt, ge, lt, le }
	Conditional:	barrier, branch, cbranch, endpgm, waitcnt
Memory	Scalar_Mem: – SMRD DWORD Format: {x, x2, x4}	load, buffer_load
	Vector_Mem: – Buffer Format: {x, xyzw}	tbuffer_load, tbuffer_store
	Date Share (LDS, GDS): {B32}	ds_read, ds_write

OP	Operands for instruction. Each format has its own operands.	SIMM	16 bit immediate value
VDST	Vector destination register, can address only vector registers.	SDST	Scalar destination register, can address only scalar registers.
SRC0	Source 2, can address vector, scalar and special registers, also can indicate constants.	VSRC1	Vector source 1, can address only vector registers.
SSRC1	Scalar source 1, can address only scalar registers.	SBASE	Scalar register that contains the size and base address.
IMM	Flag that marks whether OFFSET is an immediate value or the address of a scalar register	OFFSET	Offset to the base address specified in SBASE

along the three perspectives. Section 8 concludes. The authors have no affiliation with AMD or GPU manufacturers. All information about AMD products used and described is either publicly available (and cited) or reverse-engineered by authors from public documents.

2. MIAOW ARCHITECTURE

This section describes MIAOW's ISA support, processor organization, microarchitecture of compute units, and pipeline organization and provides a discussion of design choices.

2.1. ISA Support

MIAOW implements a subset of the Southern Islands ISA, which we summarize later. The *architecture state and registers* defined by MIAOW's ISA include the program counter, execute mask, status registers, mode register, general-purpose registers (scalar s0-s103 and vector v0-v255), local data share (LDS), 32-bit memory descriptor, scalar condition codes, and vector condition codes. *Program control* is defined using predication and branch instructions. The *instruction encoding* is of variable length having both 32-bit and 64-bit instructions. Scalar instructions are organized in five formats [SOPC, SOPK, SOP1, SOP2, SOPP]. Vector instructions come in four formats, of which three [VOP1, VOP2, VOPC] use 32-bit instructions and one [VOP3] uses 64-bit instructions to address three operands. Scalar memory reads (SMRD) are 32-bit instructions involved only in memory read operations and use two formats [LOAD, BUFFER_LOAD]. Vector memory instructions use two formats [MUBUF, MTBUF], both being 64 bits wide. Data share operations are involved in reading and writing to LDS and global data share (GDS). Four commonly used instruction encodings are shown in Table IV. Two memory *addressing modes* are supported: base+offset and base+register.

Of a total of over 400 instructions in SI, MIAOW's instruction set is a carefully chosen subset of 95 instructions and the generic instruction set is summarized in Table IV. This subset was chosen based on benchmark profiling, the type of operations in the data path that could be practically implemented in RTL by a small design team, and

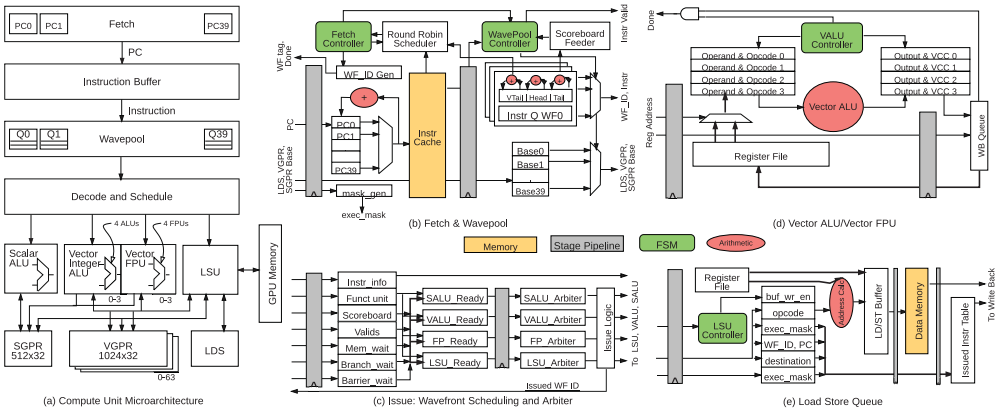

Fig. 3. MIAOW compute unit block diagram and design of submodules.

elimination of graphics-related instructions. MIAOW does not support 64-bit integer and floating operations as of now, and this is one of the limitations of MIAOW. But the AMD APP kernels [amd 2013] do not use 64-bit operations aggressively and all of them could be run on MIAOW, thus making MIAOW 32-bit software compatible with OpenCL applications. In short, the ISA defines a processor that is a tightly integrated hybrid of an in-order core and a vector core all fed by a single instruction supply and memory supply with massive multithreading capability. The complete SI ISA judiciously merges decades of research and advancements within each of those designs. From a historical perspective, it combines the ideas of two classical machines: the Cray-1 vector machine [Russell 1978] and the HEP multithreaded processor [Smith 1981]. The recent Maven [Lee et al. 2011] design is most closely related to MIAOW and is arguably more flexible and includes/explores a more diverse design space. From a practical standpoint of exploring GPU architecture, we feel it falls short on realism and software compatibility.

2.2. MIAOW Processor Design Overview

Figure 1 shows a high-level design of a canonical AMD Southern-Islands-compliant GPGPU. The system has a host CPU that assigns a kernel to the GPGPU, which is handled by the GPU's ultra-threaded dispatcher. It computes kernel assignments and schedules wavefronts to CUs, allocating wavefront slots, registers, and LDS space. The CUs shown in Figure 1(b) execute the kernels and are organized as scalar ALUs, vector ALUs, a load-store unit, and an internal scratch pad memory (LDS). The CUs have access to the device memory through the memory controller. There are L1 caches for both scalar data accesses and instructions and a unified L2 cache. The MIAOW GPGPU adheres to this design and consists of a simple dispatcher, a configurable number of compute units, a memory controller, an OCN, and a cached memory hierarchy.[2] MIAOW allows scheduling up to 40 wavefronts on each CU, which may belong to different workgroups.

2.3. MIAOW Compute Unit Microarchitecture

Figure 3 shows the high-level microarchitecture of MIAOW with details of the most complex modules, and Figure 4 shows the pipeline organization. To follow is a brief

[2]The reference design includes a 64KB GDS, which we omitted in our design since it is rarely used in performance-targeted benchmarks.

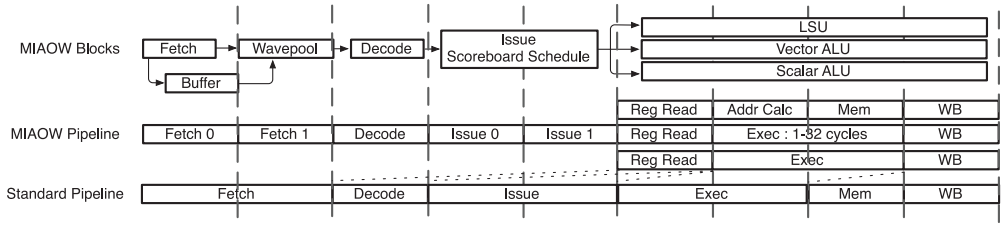

Fig. 4. MIAOW compute unit pipeline stages.

description of the functionalities of each microarchitectural component; further details are deferred to an accompanying technical report.

Fetch (Figure 3(b)). Fetch is the interface unit between the ultra-threaded dispatcher and the compute unit. When a wavefront is scheduled on a compute unit, the fetch unit receives the initial PC value, the range of registers and local memory it can use, and a unique identifier for that wavefront. The same identifier is used to inform the dispatcher when execution of the wavefront is completed. It also keeps track of the current PC for all executing wavefronts.

Wavepool (Figure 3(b)). The Wavepool unit serves as an instruction queue for all fetched instructions. Up to 40 wavefronts—supported by 40 independent queues—can be resident in the compute unit at any given time. The wavepool works closely with the fetch unit and the issue unit to keep instructions flowing through the compute unit.

Decode. This unit handles instruction decoding. It also collates the two 32-bit halves of 64-bit instructions. The decode unit decides which unit will execute the instruction based on the instruction type and also performs the translation of logical register addresses to physical addresses.

Issue/Schedule (Figure 3(c)). The issue unit keeps track of all in-flight instructions and serves as a scoreboard to resolve dependencies on general-purpose and special registers. It ensures that all the operands of an instruction are ready before issue. It also handles the barrier and halt instructions.

Vector ALU (Figure 3(d)). Vector ALUs perform arithmetic or logical operations (integer and floating point) on the data for all 64 threads of the wavefront, depending on the execution mask. We have four integer (SIMD—Single Instruction Multiple Data) and four floating point (SIMF—Single Instruction Multiple Floating point) ALUs, each being 16-wide: one wavefront is processed as four batches of 16.

Scalar ALU. Scalar ALUs (SALUs) execute arithmetic and logic operations on a single value per wavefront. Branch instructions are also resolved here.

Load Store Unit (Figure 3(e)). The load store unit handles both vector and scalar memory instructions. It handles loads and stores from the global GPU memory as well as from the LDS.

Register Files. The CU accommodates 1,024 vector registers and 512 scalar registers separately, accessible by all wavefronts using a base register address local to the wavefront and the virtual address that is used to calculate the physical register address. Both register files are banked for the purpose of parallel data access, allowing a read or write to operate on a word from each bank in a single operation.

Vector register files are organized as 64 pages or banks, each page corresponding to the registers for one of the 64 threads of a wavefront. Each page of the register file is further divided into a number of banks that varies according to the design used. The

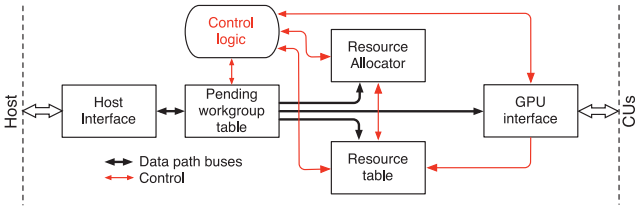

Fig. 5. MIAOW ultra-threaded dispatcher block diagram.

scalar register file is organized as four banks. There are also a set of special registers associated with each wavefront, namely, exec (a 64-bit mask that governs which of the 64 threads will execute an instruction), vcc (a 64-bit value that holds the condition code generated on execution of a vector instruction), scc (a 1 bit value which holds the condition code on execution of a scalar instruction), and the M0 register (a 32-bit temporary memory register).

2.4. MIAOW Ultra-Threaded Dispatcher

MIAOW's ultra-threaded dispatcher does global wavefront scheduling, receiving workgroups from a host CPU and passing them to CUs. It also ensures that the wavefronts' addressing spaces for LDS, GDS, and the register files never overlap on CUs. MIAOW features two versions for the ultra-threaded dispatcher: a synthesizable RTL model and a C/C++ model.

Figure 5 presents a block diagram of the RTL version of the dispatcher. The workgroups arrive through the host interface, which handles all communication between the host and MIAOW. If there are empty slots in the pending workgroup table (PWT), the host interface accepts the workgroup; otherwise, it informs the host that it cannot handle it. An accepted workgroup can be selected by the control unit for allocation and is passed to the resource allocator, which tries to find a CU that has enough resources for it. If a free CU is found, the CU ID and allocation data are passed to the resource table and to the GPU interface so that execution can begin. Workgroups that cannot be allocated go back to the PWT and wait until there is a CU with enough resources. The resource table registers all the allocated resources, clearing them after the end of execution. It also updates the resource allocator CAMs, allowing one to use that information to select a CU. The control unit is responsible for flow control and it blocks workgroup allocation to CUs whose resource tables are busy. Finally, the GPU interface divides a workgroup into wavefronts and passes them to the CU, one wavefront at a time. Once the execution starts, the ultra-threaded dispatcher will act again only when the wavefront ends execution in the CU and is removed.

The RTL dispatcher provides basic mechanisms for workgroup allocation, leaving the allocation policy encapsulated in the resource allocator. Currently the allocator selects the CU with the lowest ID that has enough resources to run a workgroup, but this policy can be easily changed by modifying the allocator.

2.5. Design Choices

Table V summarizes the important microarchitectural design choices organized in terms of how our choices impact our two goals: realism and flexibility. We also discuss physical design impact in terms of area and power. Commenting on realism is hard, but AMD's *Graphics Core Next (GCN)* architecture [amd 2012a], which outlines implementation of SI, provides sufficient high-level details. Comments are based on our interpretation of GCN and are not to be deemed authoritative. We also comment on design decisions based on machine balance, which uses content from Section 4.

Table V. Impact of Design Choices

Design Choice	Realistic	Flexibility	Area/Power Impact
Fetch bandwidth (1)	Balanced†	Easy to change	Low
Wavepool slots (6)	Balanced†	Parameterized	Low
Issue bandwidth (1)	Balanced†	Hard to change	Medium
# int FU (4)	Realistic	Easy to change	High
# FP FU (4)	Realistic	Easy to change	High
Writeback queue (1)	Simplified	Parameterized	Low
RF ports (5,4)	Simplified	Hard to change	High
RF ports (SRAM) (1)	Realistic	Hard to change	Low
Types of FU	Simplified	Easy to change	High

†*Fetch optimized for cache hit, rest sized for balanced machine. Numbers in parenthesis indicate the design parameters.*

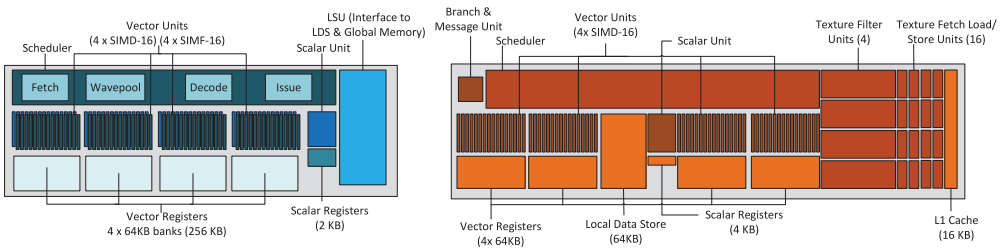

Fig. 6. MIAOW comparison to Kaveri [BOUVIER and SANDER 2014].

Summary. In short, our design choices lead to a realistic and balanced design. Many of the decisions for parameters such as fetch width or number of functional units were influenced by the desire to have one instruction complete per cycle, a throughput rate that we felt was adequate performance wise. As such, the parameters selected complement each other, with the fetch and issue units readying one instruction per cycle and the writeback queue also accommodating one instruction per cycle. The vector functional units are also pipelined in such a way as to allow one operation to complete per cycle. This analysis serves the purpose of researchers seeking to understand the MIAOW's strengths and weaknesses relative to "real" GPU implementations. Our language and choice of words is *intentionally* nonauthoritative as there is no concrete or publicly available information about GPUs. Figure 6 shows that at the block level MIAOW has similarities, from which the strongest possible statement we can make is: "MIAOW is a reasonable design." To place this in context, Fabscalar, a recently released and pioneering OOO design, is very different from any commercial OOO design but has served researchers well. More details follow.

Impossible Goal. Some researchers may feel there is a need for a comparison to pipeline-level details of MIAOW to real GPUs. We argue that this is probably impossible, and obtaining detailed pipeline-level understanding of a real GPU is a publication in its own right. Our description is sufficient for researchers to understand MIAOW's similarity to other "high level" pipeline organizations, as described in GPGPUSim for example. Researchers must ultimately decide for themselves whether or not the nature of their research exploration is intimately tied to a commercial "real" microarchitecture, and in such case, using MIAOW is NOT recommended. Our case studies comment about these in further sections.

Fetch Bandwidth (1). We optimized the design assuming instruction cache hits and single instruction fetch. In contrast, the GCN specification has a fetch bandwidth in the order of 16 or 32 instructions per fetch, presumably matching a cache line.

The fetch bandwidth chosen to be 1 (optimized for cache hit) is not limited in case of cache misses too. The instruction cache of each CU is a nonblocking cache, and it continues accepting requests while it services the misses from lower-level hierarchy. The fetch unit can store up to 40 PCs (instructions). To handle the miss latency and to not affect performance, there are enough instructions of other wavefronts to feed the following issue, SIMD, SIMF, and SALU units to continue execution. MIAOW includes an additional buffer between fetch and wavepool to buffer the multiple instructions fetched for each wavefront. The design can be changed easily by changing the interface between the fetch module and the instruction memory.

Wavepool Slots (6). Based on the back-of-the-envelope analysis of load balance, we decided on six wavepool slots. Our design evaluations show that all six slots of the wavepool are filled 50% of the time, suggesting that this is a reasonable and balanced estimate considering our fetch bandwidth. We expect the GCN design to have many more slots to accommodate the wider fetch. The number of slots is parameterized and can be easily changed. Since this pipeline stage has a smaller area, it has less impact on area and power.

Issue Bandwidth (1). We designed this to match the fetch bandwidth and provide a balanced machine as confirmed in our evaluations. Increasing the number of instructions issued per cycle would require changes to both the issue stage and the register read stage, increasing the register read ports. Compared to our single-issue width, GCN's documentation suggests an issue bandwidth of 5. For GCN, this seems an unbalanced design because it implies issuing four vector and one scalar instruction every cycle, while each wavefront is generally composed of 64 threads and the vector ALU is 16 wide (needs four cycles to process 64 threads and five such wavefronts need to be handled).

Number of Integer and Floating-Point Functional Units (4, 4). We incorporate four integer and four floating-point vector functional units to match industrial designs like the GCN, and the high utilization of FUs by Rodinia benchmarks indicates the number is justified. These values are parameterizable in the top-level module, and these are the major contributors to area and power.

Number of Register Ports (5,4,1). The register file as provided in MIAOW is a generic flip-flop-based design with five ports for the SGPR and four for the VGPR. The SGPR possesses three read ports and two write ports, whereas the VGPR possesses three read ports and one write port. Storage elements like register files are often dependent on the technology process used for fabrication. We thus provide the generic version for general-purpose use of MIAOW. Only for the purpose of area analysis, we utilized a single-ported SRAM-based version generated using the Synopsys Design Compiler. We have refrained from developing dedicated logic for handling contention as the purpose of the SRAM-based register was only area analysis, and not *functional correctness*. This decision will result in a model with a small underestimation of area. We believe future versions would have a functionally correct SRAM-based RF design along with necessary stalling logic and operand collectors. Note that all our simulations and detailed power analysis use our functionally correct multiported RF. For those interested in performing power analysis using a specific technology process for the register elements (including SRAM based), many register compilers, hard macros, and modeling tools like CACTI are available, providing a spectrum of accuracy and fidelity for MIAOW's users. Researchers can easily study various configurations [Abdel-Majeed and Annavaram 2013] by swapping out our module. For another such technology-process-specific implementation of the register files produced by us, please refer to the FPGA section.

Number of Slots in Writeback Queue per Functional Unit (1). To simplify implementation, we used one writeback queue slot, which proved to be sufficient in design

evaluation. The GCN design indicates a queuing mechanism to arbitrate access to a banked register file. Each FU has the capability to process only one instruction per cycle, and hence the writeback slot for each FU was enough to hold one instruction. Also, the register file access (RFA) unit makes sure there is no contention in writing to the register file (SGPR or VGPR) with a back-pressure signal going to the issue unit and ensuring that the next instruction is not issued to SIMD and SIMF until the existing one is written. Our design choice here probably impacts realism significantly. The number of writeback queue slots is parameterized and thus provides flexibility. The area and power overhead of each slot are negligible.

3. IMPLEMENTATION

In this section, we first describe MIAOW's hybrid implementation strategy of using synthesizable RTL and behavioral models and the tradeoffs introduced. We then briefly describe our verification strategy, physical characteristics of the MIAOW prototype, and a quantitative characterization of the prototype.

3.1. Implementation Summary

Figure 2(c) shows our implementation denoting components implemented in synthesizable RTL versus PLI or C/C++ models.

Compute Unit, Ultra-Threaded Dispatcher. As described in AMD's specification for SI implementations, "the heart of GCN is the new Compute Unit (CU)," and so we focus our attention on the CU, which is implemented in synthesizable Verilog RTL. There are two versions of the ultra-threaded dispatcher, a synthesizable RTL module and a C/C++ model. The C/C++ model can be used in simulations where dispatcher area and power consumption are not relevant, saving simulation time and easing the development process. The RTL design can be used to evaluate the complexity, area, and power of different scheduling policies.

On-Chip Network (OCN), L2 Cache, Memory, Memory Controller. Simpler PLI models are used for the implementation of the OCN and memory controller. The OCN is modeled as a cross-bar between CUs and memory controllers. To provide flexibility, we stick to a behavioral memory system model, which includes device memory (fixed delay), instruction buffer, and LDS. This memory model handles coalescing by servicing diverging memory requests. We model a simple and configurable cache that is nonblocking (FIFO-based simple MSHR design), set associative, and write-back with a LRU replacement policy. The size, associativity, block size, and hit and miss latencies are programmable. A user has the option to integrate more sophisticated memory subsystem techniques [Singh et al. 2013; Hechtman and Sorin 2013]. These behavioral modules constitute only a small part of the overall chips' area/power consumption compared to all of the functional units.

3.2. Verification

Figure 7 shows the verification flow of MIAOW. A standard flow of unit tests and randomly generated regression tests (generated using a random program generator that produces assembly) are passed through a compiler (AMD OpenCL compiler) to produce AMD-IL, then the device driver to produce binaries (AMD GPU device drivers). These binaries are fed to the RTL through a PLI module that populates the memories. Verification hooks are inserted in the RTL to produce an architectural trace of committed instructions. This architecture trace is compared with a reference trace produced by an instruction emulator (Multi2sim [Ubal et al. 2012]). Figure 7 also shows which pieces of the design are implemented using C/C++ behavioral models that interact with the Verilog RTL through PLI. Our workhorse for verification was a random program

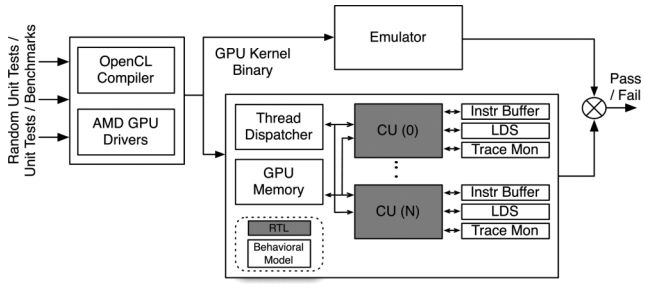

Fig. 7. MIAOW verification (LDS: local data share, Trace Mon: trace monitor unit).

generator that is parameterizable to produce different instruction mixes. The multi-threaded nature and out-of-order retirement of wavefront introduces some challenges in trace comparison. The environment uses PLIs that should work on any Verilog simulator. MIAOW used Synopsys VCS as the Verilog simulator for verification.

3.3. Physical Design

Physical design was relatively straightforward using the Synopsys Design Compiler for synthesis and the IC Compiler for place and route with the Synopsys 32nm library. Based on Design Compiler synthesis, our CU design's area is 15mm^2 using multiported register file design and it consumes on average 1.1W of power across all benchmarks (detailed analysis in Section 4). We are able to synthesize the design at a clock period of 4.5ns (FO4 delay being 14.5ps at 32nm node and thus frequency of 310 FO4s). The main critical path of the design is in the instruction arbiter of the issue unit, which chooses one wavefront to issue every cycle. The issue unit has to wait for ready bits from all the wavefronts and execution units and then choose a wavefront's instruction for execution. The single longest path of the design is difficult to determine because of back-pressure signals to all units. Layout introduces challenges because of the dominant usage of SRAM and register files. The automatic flat layout without floorplanning fails. While blackboxing these produced a layout, detailed physical design is pushed to future work.

3.4. FPGA Implementation

In addition to software emulation, MIAOW was successfully synthesized on a state-of-the-art very large FPGA. This variant, dubbed Neko, underwent significant modifications in order to fit the FPGA technology process. We used a Xilinx Virtex7 XC7VX485T, which has 303,600 LUTs and 1,030 block RAMs, mounted on a VC707 evaluation board. Prototyping MIAOW on the FPGA provided a useful measurement of how feasible the basic design is when it comes to physical hardware implementation.

Design. Neko is composed of a MIAOW compute unit attached to an embedded Microblaze soft-core processor via the AXI interconnect bus. The Microblaze implements the ultra-threaded dispatcher in the software, handles prestaging of data into the register files, and serves as an intermediary for accessing memory (Neko does not interface directly to a memory controller). Due to FPGA size limits, Neko's compute unit has a smaller number of ALUs, one-vector integer (SIMD), and floating-point (SIMF) modules, respectively, than a standard MIAOW compute unit, which has four-vector integer and floating-point modules, respectively. The consequence of this is that, while Neko can perform any operation that a full compute unit can, its throughput is lower due to the fewer computational resources. It must be noted that a significant percentage of the resource consumption for MIAOW came from the use of flip-flops to hold state information due to pipelining. Therefore, attempting to use onboard resources such as DSP

Table VI. Resource Utilization

Module	LUT Count	# BRAMs	Module	LUT Count	# BRAMs
Decode	3,474	-	SGPR	647	8
Exec	8,689	-	SIMD	36,890	-
Fetch	22,290	1	SIMF	55,918	-
Issue	36,142	-	VGPR	2,162	128
SALU	1,240	-	Wavepool	27,833	-
Total	195,285	137			

slices and the like would not have reduced the resource consumption of the modules enough to fit more ALUs on the design. One other difference is Neko's register file architecture. Mapping MIAOW's register files naively to flip-flops causes excessive usage and routing difficulties, especially considering the vector ALU register file, which has 65,536 entries. Using block RAMs is not straightforward either; they only support two ports each, fewer than what the register files need. This issue was ultimately resolved by banking and double-clocking the BRAMs to meet port and latency requirements.

Resource Utilization and Use Case. Table VI presents breakdowns of resource utilization by the various modules of the compute unit for LUTs and block RAMs, respectively. Neko consumes approximately 64% of the available LUTs and 16% of the available block RAMs. Since Neko's performance is lower than MIAOW due to the trimmed-down architecture, one needs to consider this when interpreting research findings from Neko. Our article reports results using the full MIAOW design using VCS simulations.

4. IS MIAOW REALISTIC?

We now seek to understand and demonstrate whether MIAOW is a realistic implementation of a GPU. To accomplish this, we *compare* it to industry GPUs on three metrics: area, power, and performance. To reiterate the claim made in Section 1, MIAOW does not aim to be an exact match of any industry implementation. To check if quantitative results of the aforementioned metrics follow trends similar to industry GPGPU designs, we *compare* MIAOW with the AMD Tahiti GPU, which is also an SI GPU. In cases where the relevant data is not available for Tahiti, we use model data, simulator data, or data from NVIDIA GPUs. Table VII summarizes the methodology and key results and shows that MIAOW is realistic.

Area. When comparing the area of MIAOW CUs to Tahiti CUs, MIAOW's is larger. This is not surprising as MIAOW is still a relatively immature design and its functional units are nowhere as optimized. MIAOW's single-ported SRAM-based RF design (functionally incorrect) has an area estimate of 9.31mm^2 and the multiported RF design has an area of 15mm^2. Both RF designs were considered to get an idea of MIAOW's area if evaluated with a realistic SRAM design. Figure 8 presents a detailed total area breakdown of MIAOW's CU with multiported RF design. Fetch/queue/decode/schedule (FQDS), functional units (FUs), and register file (RF) area contribution are also shown.

Power. The MIAOW RTL model allows accurate runtime power measurements across each component in the CU. Again, multiported RF design is used for the power analysis for all benchmarks and the total area is 1.1W. Figure 9 presents the total power breakdown of MIAOW's CU and individual power numbers of the submodules. Since the variation of power breakdown at the CU level across different benchmarks was found to be small, the figure shows the average across all the AMD APP benchmarks.

Performance. Performance analysis is somewhat complicated due to a variety of factors. We initially planned to compare the performance with execution of the same kernels on an AMD SI-based GPU but were not successful due to the lack of compute

Table VII. Summary of Investigations of MIAOW's Realism

Area Analysis

Goal
- ○ Is MIAOW's total area and breakdown across modules representative of industry designs?

Method
- ○ Perform MIAOW synthesis using Synopsys Design Compiler.
- ○ Synthesized with multiported RF and functionally incorrect one-ported SRAM register file.
- ○ For release, multiported flip-flop-based register file.
- ○ Compare to AMD Tahiti (SI GPU) implemented at 28nm; scaled to 32nm for absolute comparisons.

Key results
- ○ Total area using one-port Synopsys RegFile - 9.31mm^2 compared to 6.92mm^2 for Tahiti CU.
- ○ Total area using multiported flip-flop-based RegFile - 15mm^2.
- ○ Area breakdown (multiported RF) matches intuition; 34% in functional units & 54% in RFs.
- ○ More area details in Figure 8.

Power Analysis

Goal
- ○ Is MIAOW's total power and breakdown across modules representative of industry designs?

Method
- ○ Synopsys Power Compiler runs with SAIF activity file generated by running benchmarks through VCS.
- ○ Compared to GPU power models of NVIDIA GPU [Hong and Kim 2010]. Breakdown and total power for industry GPUs not publicly available.

Key results
- ○ MIAOW breakdown (multiported RF): FQDS: 11.86%, RF: 19.71%, FU: 55.94%.
- ○ NVIDIA breakdown: FQDS: 36.7%, RF: 26.7%, FU: 36.7%.
- ○ MIAOW CU's total power: 867.87mW; CU + Local Cache modules: 1102mW $\approx$ 1.1W.
- ○ Compared to NVIDIA model, more power in functional units.
- ○ FQDS and RFs roughly have similar contributions compared to NVIDIA model.
- ○ Total power is 1.1 watts. No comparison reference available.
- ○ We feel 1.1W is low. Likely because Synopsys 32nm technology library is targeted to low-power design (1.05V, *300MHz typical frequency*).
- ○ More power details in Figure 9.

Performance Analysis

Goal
- ○ Is MIAOW's performance realistic?

Method
- ○ Failed in comparing to AMD Tahiti performance using AMD performance counters (bugs in vendor drivers).
- ○ Compared to similar style NVIDIA GPU Fermi 1-SM GPU.
- ○ Performance analysis done by obtaining CPI for each class of instructions across benchmarks.
- ○ Performed analysis to evaluate balanceand sizing.

Key results
- ○ CPI breakdown across execution units is below.

CPI	DMin	DMax	BinS	BSort	MatT	PSum	Red	SLA
Scalar	1	3	3	3	3	3	3	3
Vector	1	6	5.4	2.1	3.1	5.5	5.4	5.5
Memory	1	100	14.1	3.8	4.6	6.0	6.8	5.5
Overall	1	100	5.1	1.2	1.7	3.6	4.4	3.0
NVIDIA	*1*	_	*20.5*	*1.9*	*2.1*	*8*	*4.7*	*7.5*

- ○ MIAOW is close on three benchmarks.
- ○ CPIs being in similar range shows MIAOW's realism.
- ○ MIAOW seems to be a balanced design.

performance profiling tools and bugs in vendor-provided drivers. Instead, we compare instruction latencies to a single SM of an NVIDIA Fermi-based GPU using NVVP (NVIDIA Visual Profiler). The last row in the performance analysis section of Table VII shows measured CPI for this processor for the selected benchmarks. Six OpenCL benchmarks as part of the Multi2sim environment were chosen, which we list along with three characteristics: - # *of work groups*, # *of wavefronts per workgroup*, and # *of compute cycles per workgroup*: BinarySearch (4, 1, 289), BitonicSort (1, 512, 97,496), MatrixTranspose (4, 16, 4,672), PrefixSum (1, 4, 3,625), Reduction (4, 1, 2,150), ScanLargeArrays (2, 1, 4). Overall, MIAOW is close on three benchmarks. On another three, MIAOW's CPI is two times lower, the primary reasons for which are (1) the instruction count after finalization of PTX on the NVIDIA GPU is different from the

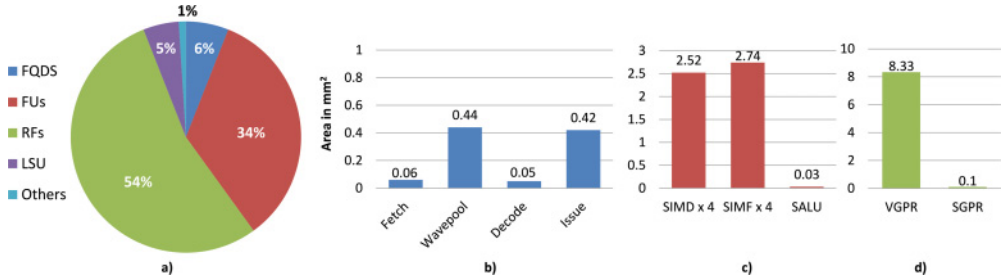

Fig. 8. (a) CU area breakdown with multiported RF (15mm^2); (b) FQDS area; (c) FUs area; (d) RF area (FQDS: fetch/queue/decode/schedule; LSU: load store unit; Others: execute mask, register file access). (All the area numbers above are in mm^2.)

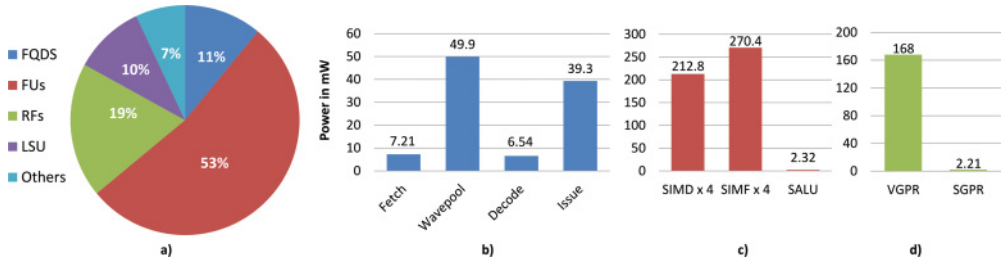

Fig. 9. (a) CU power breakdown with multiported RF (1.1W); (b) FQDS power; (c) FUs power; (d) RF power (FQDS: fetch/queue/decode/schedule; LSU: load store unit; Others: execute mask, register file access). (All the power numbers above are in mW.)

SI device code; (2) cycle measurement itself introduces noise; and (3) the microarchitectures implemented by AMD and NVIDIA are quite different. MIAOW can also run four Rodinia [Che et al. 2009] benchmarks at this time: kmeans, nw, backprop, and gaussian. We use these longer benchmarks for the case studies in Section 5 onward.[3]

Despite the caveats mentioned previously, the quantitative results from the benchmarks indicate that MIAOW is a balanced design when compared to a commercial GPU design. Again, we emphasize that our goal is not to produce a GPU design capable of matching a commercial GPU in raw performance; we simply conducted this comparison to ensure that the fundamental architecture of our design is sound.

5. PHYSICAL DESIGN PERSPECTIVE OF TRADITIONAL MICROARCHITECTURE RESEARCH

Description. Fung et al. proposed Thread Block Compaction (TBC) [Fung and Aamodt 2012], which belongs to a group of work done on warp scheduling [Fung and Aamodt 2012; Jog et al. 2013a, 2013b; Meng et al. 2010; Narasiman et al. 2011; Rhu and Erez 2012; Rogers et al. 2012], any of which we could have picked as a case study. TBC, in particular, aims to increase functional unit utilization on kernels with irregular control flow. The fundamental idea of TBC is that, whenever a group of wavefronts face a branch that forces its work items to follow the divergent program paths, the hardware should dynamically reorganize them in new re-formed wavefronts that contain only those work items following the same path. Thus, we replace the idle work items with active ones from other wavefronts, reducing the number of idle SIMD lanes. Groups of wavefronts that hit divergent branches are also forced to run in similar paces, reducing even more work-item-level diversion on such kernels. Re-formed wavefronts

[3]Others don't run because they use instructions outside MIAOW's subset and because of a lack of 64-bit execution support.

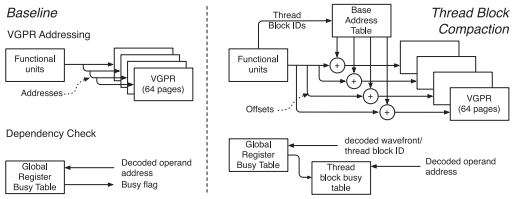

Fig. 10. Critical modifications in thread block compaction.

are constructed observing the originating lane of all the work items: if it occupies the lane 0 in wavefront A, it must reoccupy the same lane 0 in re-formed wavefront B. The wavefront-forming mechanism is completely local to the CU, and it happens without intervention from the ultra-threaded dispatcher. In this study, we investigate the level of complexity involved in the implementation of such microarchitecture innovations in RTL.

Infrastructure and Methodology. We follow the implementation methodology described in Fung and Aamodt [2012]. In MIAOW, the modules that needed significant modifications were fetch, wavepool, decode, Scalar ALU (SALU), issue, and the vector register file. The fetch and wavepool modules had to be adapted to support the fetching and storage of instructions from the re-formed wavefronts. We added two instructions to the decode module: fork and join, which are used in SI to explicitly indicate divergent branches. We added the PC stack (for recovery) and modified the wavefront formation logic in the SALU module, as it was responsible for handling branches. Although this modification is significant, it does not have a huge impact on complexity, as no other logic is involved, apart from SALU's branch unit.

The issue and VGPR modules required more drastic modifications, shown in Figure 10. In SI, instructions provide register addresses as an offset with the base address being zero. When a wavefront is being dispatched to the CU, the dispatcher allocates register file address space and calculates the base vector and scalar registers. Thus, wavefronts access different register spaces on the same register file. Normally, all work items in the wavefront access the same register, but different pages of the register file as shown in the upper left corner of Figure 10, and the register absolute address is calculated during decode. But with TBC, this assumption does not hold anymore. In a re-formed wavefront, all the work items may access registers with the same offset but different base values (from different originating wavefronts). This leads to modifications in the issue stage, now having to maintain information about register occupancy by offset for each re-formed wavefront, instead of absolute global registers. In the worst-case scenario, issue has to keep track of 256 registers for each re-formed wavefront, in contrast to 1,024 for the entire CU in the original implementation. In Figure 10, the baseline issue stage observed in the lower left and lower right corner are the modifications for TBC, adding a level of dereference to the busy table search. In VGPR, we now must maintain a table with the base registers from each work item within a re-formed wavefront, and the register address is calculated for each work-item in access time. Thus, there are two major sources of complexity overheads in VGPR, the calculation and the routing of different addresses to each register page, as shown in the upper right corner of Figure 10.

We had to impose some restrictions to our design due to architectural limitations: first, we disallowed the scalar register file and LDS accesses during divergence, and therefore, wavefront-level synchronization had to happen at GDS. We also were not able to generate code snippets that induced the SI compiler to use fork/join instructions; therefore, we used handwritten assembly resembling benchmarks in Fung and

Aamodt [2012]. It featured a loop with a divergent region inside, padded with vector instructions. We controlled both the number of vector instructions in the divergent region and the level of diversion. Our baseline used a postdenominator stack-based re-convergence mechanism (PDOM) [Muchnick 1997] without any kind of wavefront formation. We compiled our tests and ran them on two versions of MIAOW: one with PDOM and the other with TBC.

Quantitative Results. The performance results obtained matched the results from Fung and Aamodt [2012]: similar performance was observed when there was no divergence, and a performance increase was seen for divergent workloads. However, our most important results came from synthesis. We observed that the modifications made to implement TBC were mostly in the regions in the critical paths of the design. The implementation of TBC caused an increase of 32% in our critical path delay. We also observed that the issue stage area grew from $0.43mm^2$ to $1.03mm^2$. Overall performance, however, remained the same because of the reduced number of cycles needed to complete workloads.

Analysis. Our performance results confirm the ones obtained by Fung et al.; however, the RTL model enabled us to implement TBC in further detail and determine that the critical path delay increases. In particular, we observed that TBC affects the issue stage significantly where most of the CU control state is present dealing with major microarchitectural events. TBC reinforces the pressure over the issue stage, making it harder to track such events. We believe that the added complexity suggests that a microarchitectural innovation may be needed involving further *design* refinements and repipelining, not just implementation modifications.

The goal of this case study is not to criticize the TBC work or give a final word on its feasibility. Our goal here is to show that, by having a detailed RTL model of a GPGPU, one can better evaluate the complexity of any proposed novelties.

6. NEW TYPES OF RESEARCH EXPLORATION

6.1. Sampling DMR on GPUs

Description. Balasubramanian et al. proposed a novel technique of unifying the circuit failure prediction and detection in CPUs using Virtually Aged Sampling DMR [Balasubramanian and Sankaralingam 2013] (Aged-SDMR). They show that Aged-SDMR provides low design complexity, low overheads, generality, and high accuracy. The key idea was to "virtually" age a processor by reducing its voltage. This effectively slows down the gates, mimicking the effect of wearout, and exposes the fault, and Sampling-DMR is used to detect the exposed fault. They show that running in epochs[4] and by sampling and virtually aging 1% of the epochs provide an effective system. Their design (shown in Figure 11) is developed in the context of multicore CPUs and requires the following: (1) operating system involvement to schedule the sampled threads, (2) some kind of system-level checkpoints (like Revive [Prvulovic et al. 2002], ReviveIO [Nakano et al. 2006], and Safetynet [Sorin et al. 2002]) at the end of every epoch, (3) system and microarchitecture support for avoiding incoherence between the sampled threads [Smolens et al. 2006], (4) some microarchitecture support to compare the results of the two cores, and (5) a subtle but important piece, gate-level support to insert a clock-phase shifting logic for fast paths. Because of these issues, Aged-SDMR's ideas cannot directly be implemented for GPUs to achieve circuit failure prediction. With reliability becoming important for GPUs [Chen 2009], this capability is desirable.

[4]Epochs are groups of workgroups belonging to a grid of a kernel, scheduled to different CUs. MIAOW uses an epoch of 100 workgroups for the SMDR case study.

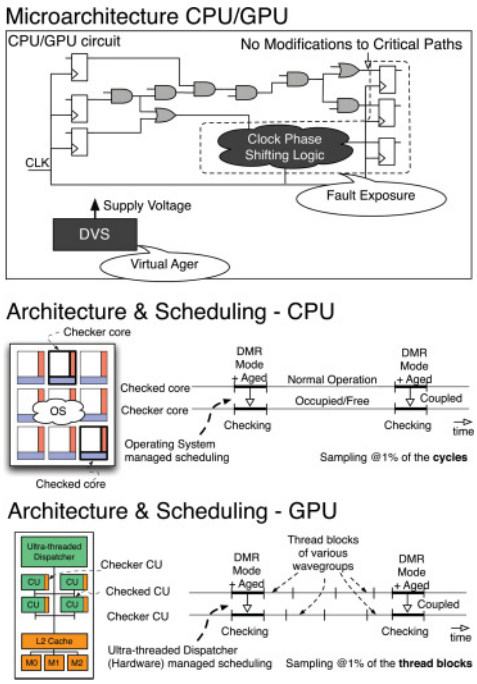

Fig. 11. Virtually aged SDMR implementation on a GPU.

Our Design. GPUs present an opportunity and problem in adapting these ideas. They do not provide system-level checkpoints, nor do they lend themselves to the notion of epochs, making (1), (2), and (3) hard. However, the thread blocks (or workgroups) of compute kernels are natural candidates for a piece of work that is implicitly checkpointed and whose granularity allows it to serve as a body of work that is sampled and run redundantly. Furthermore, the ultra-threaded dispatcher can implement all of this completely in the microarchitecture without any OS support. Incoherence between the threads can be avoided by simply disabling global writes from the sampled thread since other writes are local to a workgroup/compute unit anyway. This assumption will break and cause correctness issues when a single thread in a wavefront does read-modify-writes to a global address. We have never observed this in our workloads and believe programs rarely do this. Comparison of results can be accomplished by looking at the global stores instead of all retired instructions. Finally, we reuse the clock-phase-shifting circuit design as it is. This overall design of GPU-Aged-SDMR is a complete microarchitecture-only solution for GPU circuit failure prediction.

Figure 11 shows the implementation mechanism of GPU-Aged-SDMR. Sampling is done at a workgroup granularity with the ultra-threaded dispatcher issuing a redundant workgroup to two compute units (checker and checked compute units) at a specified sampling rate; that is, for a sampling rate of 1%, one out of 100 workgroups is dispatched to another compute unit called the checker. This is run under the stressed conditions and we disable the global writes so that they do not affect the normal execution of the workgroups in the checked CU. We can use a reliability manager module that compares all retired instructions or we can compute a checksum of the retiring stores written to global memory from the checker and the checked CU for the sampled workgroup. The checksums are compared for correctness and a mismatch detects a fault.

Table VIII. Overheads for Fault Exposure

	MIAOW		OpenSPARC*	
	Logic	CU	Logic	Core†
Gates on fast paths	23%		30%	
Area overhead	18.09%	8.69%	22.18%	6.8%
Peak power increase	5.96%	4.68%	2.21%	0.99%

*Values reported in Balasubramanian and Sankaralingam [2013].
† OpenSPARC: with 16K L1 instruction and 8K L1 data cache.

We have implemented this design in MIAOW's RTL except for the checksum; instead, we behaviorally log all the stores and compare them in the testbench. We also implemented the clock-phase-shifting logic and measured the area and power overheads that this logic introduces. The hardware changes involved minimal modifications to the following modules: memory, compute unit, fetch, workgroup info, and testbench. An extra module called memory logger was added to track the global writes to GDS by checked and checker CUs. Five interface changes had to be made to support the metadata information about the compute unit id, workgroup id, and wavefront id. A total of 207 state bits were added to support Aged-SDMR implementation on GPUs. Considering the baseline design, the modifications done for implementing SDMR in MIAOW are relatively small and thus are of low design complexity.

Quantitative Results and Analysis. Our first important result is that a complete GPU-Aged-SDMR technique is feasible with a pure microarchitecture implementation. Second, identical to the Aged-SDMR study, we can report area and power. As shown in Table VIII, logic area overheads are similar to CPUs and are small.

We also report on performance overheads of one pathological case. There could be nontrivial performance overheads in cases where there is a very small number of wavefronts per workgroup (just one in case of GaussianElim and nw). Hence, even with 1% sampling, a second CU is active for a large number of cycles. We noticed in the Rodinia suite that guassianElim and nw are written this way. In our study, we consider a two-CU system; therefore, resource reduction when sampling is on is quite significant. With 1% sampling, average performance overhead for GaussianElim and nw is 47% and 13%, and with 10% sampling it becomes 50% and 20%, respectively. Further tuning of the scheduling policies can address this issue, and also, as the number of CUs increase, this issue becomes less important.

Overall, GPU-Aged-SDMR is an effective circuit failure prediction technique for GPUs with low design complexity. MIAOW enables such new research perspectives that involve gate-level analysis, which was thus far hard to evaluate.

6.2. Timing Speculation in GPGPUs

Description. Timing speculation is a paradigm in which a circuit is run at a clock period or at a voltage level that is below what it was designed for. Prior CPU works like Razor [Ernst et al. 2003] and Blueshift [Greskamp et al. 2009] have explored timing speculation in CPUs to reduce power and handle process variations. In this case study, we quantitatively explore timing speculation for GPUs and quantify the error rate. While CPUs use pipeline flush for recovery, GPUs lack this. We have also implemented in MIAOW the modifications for idempotent re-execution [Menon et al. 2012] from the iGPU design, which supports timing speculation. Since iGPU implementation is simple, details are omitted.

Infrastructure and Methodology. Our experimental goal is to determine a voltage-to-error-rate-reduction (or clock-period-reduction) relationship for different applications

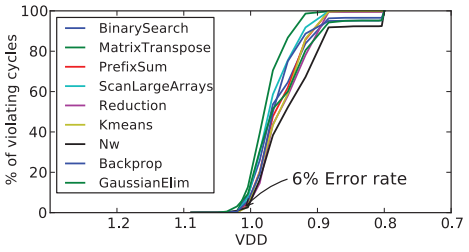

Fig. 12. Voltage level versus % of violating cycles.

and compare it to CPU trends adopting the approach of the Razor work. First, we perform detailed delay-aware gate-level simulations at a period of 4.5ns and record the transition times at the D input of each flip-flop in the design for each benchmark. Then, for different speculative clock periods, we can analyze the arrival times of every flip-flop in every cycle and determine if there is a timing error—producing an error rate for each speculative clock period. We then use SPICE simulations to find V_{dd} versus logic delay for a set of paths to determine an empirical mapping of delay change to V_{dd} reduction, thus obtaining error rate versus V_{dd} reduction. Other approaches like approximating delay-aware simulation and timing-speculation emulation could also be used [Nomura et al. 2011; Bernardi et al. 2007; Pellegrini et al. 2008].

Quantitative Results and Analysis. Figure 12 shows the variation of error rate as a function of operating voltage for nine benchmarks. Observed trends are similar to those from Razor [Ernst et al. 2003] and suggest that timing speculation could be of value to GPUs as well. The error rate grows slowly at first with V_{dd} reduction before a rapid increase at some point—at a 6% error rate, there is a 115mV voltage reduction, with nominal voltage being 1.15V. Thus, MIAOW RTL provides ways to explore this paradigm further in ways a performance simulator cannot, for example, exploring error tolerance of GPGPU workloads and investigations of power overheads of the Razor flip-flops.

7. VALIDATION/CALIBRATION OF SIMULATOR-BASED CHARACTERIZATION OF HARDWARE

Description. Previous hardware-based research studies on transient faults have focused on CPUs [Wang et al. 2004], and a recent GPU study focused on a simulator-based evaluation [Tan et al. 2011]. Wang et al. [2004] show that simulator-based studies miss many circuit-level phenomena, which in the CPU case mask most transient faults, resulting in "fewer than 15% of single bit corruptions in processor state resulting in software visible errors." Our goal is to study this for GPUs, complementing simulator-based studies and hardware measurement studies [Rech et al. 2012], which cannot provide fine-grained susceptibility information.

Infrastructure and Methodology. Our experimental strategy is similar to that of Wang et al. We ran our experiments using our testbench and VCS (Verilog Compiler Simulator) and report results for a single CU configuration. We simulate the effect of transient faults by injecting single bit-flips into flip-flops of the MIAOW RTL. We run a total of 2,000 independent experiments, where in each experiment we insert one bit-flip into one flip-flop. Across the six AMDAPP and four rodinia benchmarks, this allows us to study 200 randomly selected flip-flops. The execution is terminated 5,000 cycles after fault injection. In every experiment, we capture changes in all architecture states after fault injection (output trace of RTL simulation) and the state of every flip-flop at the end of the experiment. We gather this information for a reference run that has no fault injection. Every experiment is classified into one of four types:

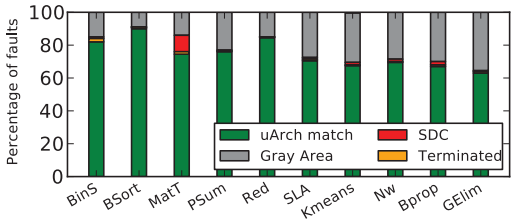

Fig. 13. Fault injection results.

—*Microarchitectural Match:* All flip-flop values match at the end of the execution window *and* no trace mismatches.
—*Silent Data Corruption (SDC):* Mismatch in the output trace ⇒ transient fault-corrupted program output.
—*Terminated:* Deadlock in pipeline and execution hangs.
—*Gray Area:* Output trace match but mismatch in one or more flip-flops ⇒ no corruption yet, but can't rule out SDC.

Quantitative Results and Analysis. We run three variants of this experiment and report the detailed results for the third variant in Figure 13. In the first variant, *MIAOW-All, faults are injected into all flip-flops, including our register file built using flip-flops.* Over 90% of faults fall in the gray area, and of these faults, over 93% are faults in the register files. In MIAOW's design, which uses a flip-flop based register file design, about 95% of flops are in the register file. The second variant, *MIAOW-noRF: exclude register file,* is run without the register file. Though there is a considerable reduction in the number of runs classified as gray area, benchmarks like BinarySearch, MatrixTranspose, Reduction, Kmeans, and Nw were still found to have more than 50% runs in the gray area.[5] The last variant is *MIAOW-noRF/VF: exclude the Vector FP units also since our benchmarks only lightly exercise them.* The gray area is now approximately 20% to 35% of the fault sites, which is closer to CPUs but larger. Also, the gray area in the Rodinia suite workloads corresponds to an uneven number of wavefronts present in each workgroup. Our analysis shows that several structures, such as the scoreboard entries in the issue stage and the workgroup information storage in the fetch stage, are lightly exercised due to the underutilized compute units. When transient faults occur in the unused areas, they do not translate to architectural errors.

8. CONCLUSION

This article has described the design, implementation, and characterization of a Southern Islands ISA-based GPGPU implementation called MIAOW. We designed MIAOW as a tool for the research community with three goals in mind: realism, flexibility, and software compatibility. We have shown that it delivers on these goals. We acknowledge that it can be improved in many ways, and to facilitate this, the RTL and case-study implementations are released open source with this work. We use four case studies to show that MIAOW enables the following: physical design perspective to "traditional" microarchitecture, new types of research exploration, and validation/calibration of simulator-based characterization of hardware. The findings and ideas are

[5]Most of the flip-flops have enable signals turned on by a valid opcode. So once a bit flip occurs, the error stays on, leading to the gray area.

contributions in their own right in addition to MIAOW's utility as a tool for others' research.

REFERENCES

2009. Barrasim: NVIDIA G80 Functional Simulator. Retrieved from https://code.google.com/p/barra-sim/.

2012a. AMD Graphics Cores Next Architecture. Retrieved from http://www.amd.com/la/Documents/GCN_Architecture_whitepaper.pdf.

2012b. Reference Guide: Southern Islands Series Instruction Set Architecture. http://developer.amd.com/wordpress/media/2012/10/AMD_Southern_Islands_Instruction_Set_Architecture.pdf.

2013. AMD APP 3.0 SDK, Kernels and Documentation. Retrieved from http://developer.amd.com/tools-and-sdks/opencl-zone/amd-accelerated-parallel-processing-app-sdk.

M. Abdel-Majeed and M. Annavaram. 2013. Warped register file: A power efficient register file for GPGPUs. In *HPCA'13*.

A. Bakhoda, G. L. Yuan, W. W. L. Fung, H. Wong, and T. M. Aamodt. 2009. Analyzing CUDA workloads using a detailed GPU simulator. In *ISPASS'09*.

R. Balasubramanian and K. Sankaralingam. 2013. Virtually-aged sampling DMR: Unifying circuit failure prediction and circuit failure detection. In *Proceedings of the 46th International Symposium on Microarchitectures (MICRO'13)*.

R. Balasubramanian and K. Sankaralingam. 2014. Understanding the impact of gate-level physical reliability effects on whole program execution. In *Proceedings of the 20th International Symposium on High Performance Computer Architecture (HPCA'14)*.

P. Bernardi, M. Grosso, and M. S. Reorda. 2007. Hardware-accelerated path-delay fault grading of functional test programs for processor-based systems. In *GLSVLSI'07*.

D. Bouvier and B. Sander. 2014. Applying AMD's Kaveri APU for heterogeneous computing. In *Hotchips 2014*.

S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron. 2009. Rodinia: A benchmark suite for heterogeneous computing. In *Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC) (IISWC'09)*. IEEE Computer Society, Washington, DC, 44–54.

J. Y. Chen. 2009. GPU technology trends and future requirements. In *IEDM'09*.

N. K. Choudhary, S. V. Wadhavkar, T. A. Shah, H. Mayukh, J. Gandhi, B. H. Dwiel, S. Navada, H. H. Najafabadi, and E. Rotenberg. 2011. FabScalar: Composing synthesizable RTL designs of arbitrary cores within a canonical superscalar template. In *ISCA'11*.

V. M. del Barrio, C. Gonzalez, J. Roca, A. Fernandez, and E. Espasa. 2006. ATTILA: A cycle-level execution-driven simulator for modern GPU architectures. In *ISPASS'06*.

G. Diamos, A. Kerr, S. Yalamanchili, and N. Clark. 2010. Ocelot: A dynamic compiler for bulk-synchronous applications in heterogeneous systems. In *PACT'10*.

D. Ernst, Nam Sung Kim, S. Das, S. Pant, R. Rao, Toan Pham, C. Ziesler, D. Blaauw, T. Austin, K. Flautner, and T. Mudge. 2003. Razor: A low-power pipeline based on circuit-level timing speculation. In *MICRO'03*.

Michael Fried. 2012. GPGPU Architecture Comparison of ATI and NVIDIA GPUs. http://www.microway.com/pdfs/GPGPU_Architecture_and_Performance_Comparison.pdf.

W. W. L. Fung and T. M. Aamodt. 2011. Thread block compaction for efficient SIMT control flow. In *Proceedings of the 2011 IEEE 17th International Symposium on High Performance Computer Architecture (HPCA'11)*. IEEE Computer Society, Washington, DC, 25–36.

W. W. L. Fung and T. M. Aamodt. 2012. Thread block compaction for efficient SIMT control flow. In *HPCA'12*.

J. Gaisler. 2001. LEON Sparc Processor.

B. Greskamp, L. Wan, U. R. Karpuzcu, J. J. Cook, J. Torrellas, D. Chen, and C. Zilles. 2009. Blueshift: Designing processors for timing speculation from the ground up. In *HPCA'09*.

B. A. Hechtman and D. J. Sorin. 2013. Exploring memory consistency for massively-threaded throughput-oriented processors. In *ISCA'13*.

S. Hong and H. Kim. 2009. An analytical model for a GPU architecture with memory-level and thread-level parallelism awareness. In *ISCA'09*.

S. Hong and H. Kim. 2010. An integrated GPU power and performance model. In *ISCA'10*.

H. Jeon and M. Annavaram. 2012. Warped-DMR: Light-weight error detection for GPGPU. In *MICRO'12*.

A. Jog, O. Kayiran, A. K. Mishra, M. T. Kandemir, O. Mutlu, R. Iyer, and C. R. Das. 2013a. Orchestrated scheduling and prefetching for GPGPUs. In *ISCA'13*.

A. Jog, O. Kayiran, N. C. Nachiappan, A. K. Mishra, M. T. Kandemir, O. Mutlu, R. Iyer, and C. R. Das. 2013b. OWL: Cooperative thread array aware scheduling techniques for improving GPGPU performance. In *ASPLOS'13*.

H. Kim, R. Vuduc, S. Baghsorkhi, J. Choi, and W. Hwu. 2012. *Performance Analysis and Tuning for General Purpose Graphics Processing Units (GPGPUs)*. Morgan & Claypool.

Y. Lee, R. Avizienis, A. Bishara, R. Xia, D. Lockhart, C. Batten, and K. Asanović. 2011. Exploring the tradeoffs between programmability and efficiency in data-parallel accelerators. In *ISCA'11*.

J. Leng, T. Hetherington, A. ElTantawy, S. Gilani, N. S. Kim, T. M. Aamodt, and V. J. Reddi. 2013. GPUWattch: Enabling energy optimizations in GPGPUs. In *ISCA'13*.

llvmcuda 2009. User Guide for NVPTX Back-end. http://llvm.org/docs/NVPTXUsage.html.

A. Meixner, M. E. Bauer, and D. Sorin. 2007. Argus: Low-cost, comprehensive error detection in simple cores. In *MICRO'07*.

J. Meng, D. Tarjan, and K. Skadron. 2010. Dynamic warp subdivision for integrated branch and memory divergence tolerance. In *ISCA'10*.

J. Menon, M. De Kruijf, and K. Sankaralingam. 2012. iGPU: Exception support and speculative execution on GPUs. In *ISCA'12*.

S. S. Muchnick. 1997. *Advanced Compiler Design Implementation*. Morgan Kaufmann.

J. Nakano, P. Montesinos, K. Gharachorloo, and J. Torrellas. 2006. ReViveI/O: Efficient handling of I/O in highly-available rollback-recovery servers. In *HPCA'06*.

V. Narasiman, M. Shebanow, C. J. Lee, R. Miftakhutdinov, O. Mutlu, and Y. N. Patt. 2011. Improving GPU performance via large warps and two-level warp scheduling. In *MICRO'11*.

S. Nomura, K. Sankaralingam, and R. Sankaralingam. 2011. A fast and highly accurate path delay emulation framework for logic-emulation of timing speculation. In *ITC'11*.

nvprof. 2008. NVIDIA CUDA Profiler User Guide. Retrieved from http://docs.nvidia.com/cuda/profiler-users-guide/index.html.

openrisc. 2010. OpenRISC Project. Retrieved from http://opencores.org/project,or1k.

opensparc. 2006. OpenSPARC T1. Retrieved from http://www.opensparc.net.

A. Pellegrini, K. Constantinides, D. Zhang, S. Sudhakar, V. Bertacco, and T. Austin. 2008. CrashTest: A fast high-fidelity FPGA-based resiliency analysis framework. In *CICC'08*.

M. Prvulovic, Z. Zhang, and J. Torrellas. 2002. ReVive: Cost-effective architectural support for rollback recovery in shared-memory multiprocessors. In *ISCA'02*.

P. Rech, C. Aguiar, R. Ferreira, C. Frost, and L. Carro. 2012. Neutron radiation test of graphic processing units. In *IOLTS'12*.

M. Rhu and M. Erez. 2012. CAPRI: Prediction of compaction-adequacy for handling control-divergence in GPGPU architectures. In *ISCA'12*.

T. G. Rogers, M. O'Connor, and T. M. Aamodt. 2012. Cache-conscious wavefront scheduling. In *MICRO'12*.

R. M. Russell. 1978. The CRAY-1 computer system. *Communications of the ACM* 22, 1 (January 1978), 64–72.

J. Sartori, B. Ahrens, and R. Kumar. 2012. Power balanced pipelines. In *HPCA'12*.

J. W. Sim, A. Dasgupta, H. Kim, and R. Vuduc. 2012. A performance analysis framework for identifying performance benefits in GPGPU applications. In *PPOPP'12*.

I. Singh, A. Shriraman, W. W. L. Fung, M. O'Connor, and T. M. Aamodt. 2013. Cache coherence for GPU architectures. In *HPCA'13*.

B. J. Smith. 1981. Architecture and applications of the HEP multiprocessor computer system. In *SPIE Real Time Signal Processing IV*, 241–248.

J. C. Smolens, B. T. Gold, B. Falsafi, and J. C. Hoe. 2006. Reunion: Complexity-effective multicore redundancy. In *MICRO'06* (no 39).

D. J. Sorin, M. M. K. Martin, M. D. Hill, and D. A. Wood. 2002. SafetyNet: Improving the availability of shared memory multiprocessors with global checkpoint/recovery. In *ISCA'02*.

J. Tan, N. Goswami, T. Li, and X. Fu. 2011. Analyzing soft-error vulnerability on GPGPU microarchitecture. In *IISWC'11*.

R. Ubal, B. Jang, P. Mistry, D. Schaa, and D. Kaeli. 2012. Multi2Sim: A simulation framework for CPU-GPU computing. In *PACT'12*.

W. J. van der Laan. 2010. Decuda SM 1.1 (G80) disassembler. https://github.com/laanwj/decuda.

N. J. Wang and S. J. Patel. 2006. ReStore: Symptom-based soft error detection in microprocessors. *IEEE Transactions on Dependable and Secure Computing* 3, 3, 188–201. DOI:http://dx.doi.org/10.1109/TDSC.2006.40

N. J. Wang, J. Quek, T. M. Rafacz, and S. J. Patel. 2004. Characterizing the effects of transient faults on a high-performance processor pipeline. In *DSN'04*.

Y. Zhang, L. Peng, B. Li, J.-K. Peir, and J. Chen. 2011. Architecture comparisons between Nvidia and ATI GPUs: Computation parallelism and data communications. In *IISWC'11*.

Received September 2014; revised March 2015; accepted April 2015