MIAOW Whitepaper
Hardware Description and Four Research Case Studies

Abstract

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, 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, 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.

1. Introduction

There is active and widespread ongoing research on GPU architecture and more specifically on GPGPU architecture. Tools are necessary for such explorations. First, we compare and contrast GPU tools with CPU tools.

On the *CPU* side, tools span performance simulators, emulators, compilers, profiling tools, modeling tools, and more recently a multitude of RTL-level implementations of microprocessors - these include OpenSPARC [39], OpenRISC [38], Illinois Verilog Model [56], LEON [18], and more recently FabScalar [11] and PERSim [7]. 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 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 [30] showed – with a prototype implementation on OpenRISC how to build lightweight fault detectors; Blueshift [19] and power balanced pipelines [46] consider the OpenRISC and OpenSPARC pipelines for novel CAD/microarchitecture work.

On the *GPU* side, a number of performance simulators [5, 2, 12, 28], emulators [53, 2], compilers [29, 13, 54], profiling tools [13, 37], and modeling tools [22, 21, 47, 26] are prevalent. However *RTL-level implementations and low-level detailed microarchitecture specification is lacking*. As discussed by others [10, 52, 42, 23, 32], 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.

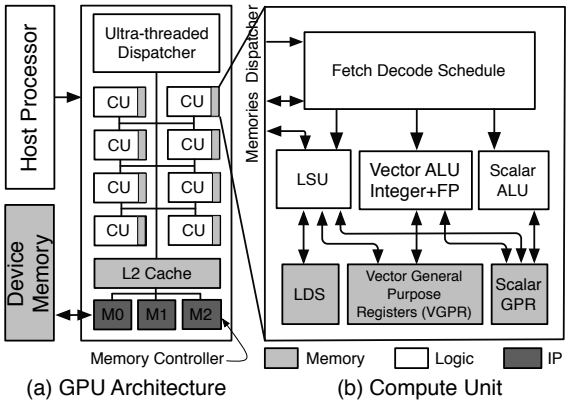


Figure 1: Canonical GPU Organization

This paper reports on the design, development, characterization, and research utility of an RTL implementation of a GPGPU called MIAOW (*acronymized and anonymized for blind review as **M**any-core **I**ntegrated **A**ccelerator **O**f the **W**aterdeep*). Figure 1 shows a canonical GPGPU architecture resembling what MIAOW targets (borrowed from the AMD SI specification, we define a few GPU terms in Table 3). Specifically, we focus on a design that delivers GPU compute capability and ignores graphics functionality. MIAOW is driven by the following key goals and non-goals.

Goals The primary driving goals for MIAOW are: i) *Realism:* it should be a *realistic* implementation of a GPU resembling principles and implementation tradeoffs in industry GPUs; ii) *Flexible:* it should be flexible to accommodate research studies of various types, the exploration of forward-looking ideas, and form an end-to-end open source tool; iii) *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.

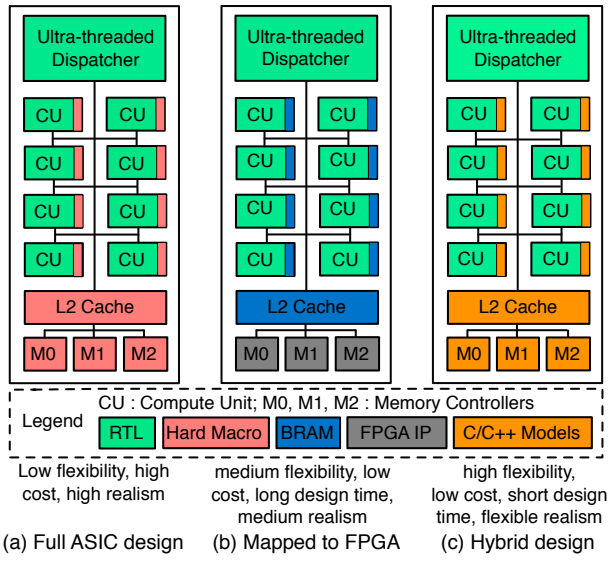


Figure 2: Target platforms for a GPGPU RTL Implementation

Non-Goals We also explain non-goals 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. sub-setting 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 (OCN), etc. in RTL. MIAOW is *not* meant to be an ASIC implementable standalone GPGPU. Finally, being competitive with commercial designs was a *non-goal*.

Driven by these goals and non-goals, we have developed MIAOW as an implementation of a subset of AMD's Southern Islands(SI) ISA [3]. While we pick one ISA and design style, we feel it is representative of GPGPU design [15] — AMD and NVIDIA's approaches have some commonalities [57]. 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 the compute units 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 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 which 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 as it satisfies all three goals. A modified design can also be synthesized for the Virtex7 FPGA though its limitations will be discussed in a later section. Table 1 compares MIAOW to state-of-the-art commercial products. The contributions of this paper include methodological techniques and ideas.

Name	# cores	GFLOPS	Core clock	Tech node
7350 (Jan '12)	80	104	650 MHz	40nm
7990 (Apr '13)	4096	8192	1 GHz	28nm
MIAOW (Now)	64-2048	57-1820	222 MHZ	32nm

Table 1: MIAOW RTL vs. state-of-art products (Radeon HD)

Methodology Methodologically, we provide 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[1]. 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 2. First it adds a *physical design perspective to "traditional" microarchitecture* research - here we revisit and implement in RTL a previously proposed warp scheduler technique [17] 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: i) We take the Virtually Aged Sampling-DMR [6] work proposed for fault-prediction in CPUs and implement a design for GPUs and evaluate complexity, area, and power overheads. ii) 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 paper 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 along the three perspectives. Section 8

[1] 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 to. Instead we compare to a nearest neighbor to show trends are similar.

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 2: Case studies summary

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, processor organization, microarchitecture of compute units and pipeline organization, and provides a discussion of design choices.

2.1. ISA

MIAOW implements a subset of the Southern Islands ISA which we summarize below. The *architecture state and registers* defined by MIAOW's ISA includes the program counter, execute mask, status registers, mode register, general purpose registers (scalar s0-s103 and vector v0-v255), 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 (both 32-bit and 64-bit) are organized in 5 formats [SOPC, SOPK, SOP1, SOP2, SOPP]. Vector instructions come in 4 formats of which three [VOP1, VOP2, VOPC] use 32-bit instructions and one [VOP3] uses 64-bit instructions to address 3 operands. Scalar memory reads (SMRD) are 32-bit instructions involved only in memory read operations and use 2 formats [LOAD, BUFFER_LOAD]. Vector memory instructions use 2 formats [MUBUF, MTBUF], both being 64-bits wide. Data share operations are involved in reading and writing to local data share (LDS) and global data share (GDS). Four commonly used instruction encodings are shown in Table 4. 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 4. 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 elimination of graphics-related instructions. In short, the ISA defines a processor which 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 multi-threading 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 [45] and the HEP multi-threaded processor [49]. The recent Maven [27] 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, memory controller, OCN, and a cached memory hierarchy[2]. MIAOW allows scheduling up to 40 wavefronts on each CU.

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. Below is a brief description of the functionalities of each microarchitectural component – further details are deferred to an accompanying technical report.

Fetch (Fig. 3b) 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 which 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 (Fig. 3b) 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.

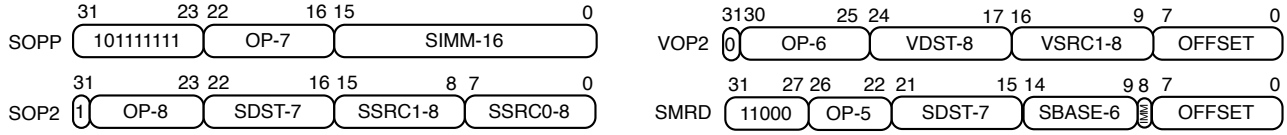
Decode This unit handles instruction decoding. It also collates the two 32-bit halves of 64-bit instructions. The Decode Unit decides which unit will execute the instruction based on the instruction type and also performs the translation of logical register addresses to physical addresses.

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

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 comprise more than one wavefront, but is mapped to a single CU.
Local data store (LDS)	Shared memory	Memory space that enables low-latency communication between work-items within a work-group, 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: 64 KB.
Device memory	Device memory	Off-chip memory provided by DRAM possibly cached in other on-chip storage.

Table 3: Definition of Southern Islands ISA terms and correspondence to NVIDIA/CUDA terminology

Type	Instructions
Vector	ALU: {U32, I32, F32, F64} - add, addc, sub, mad, madmk, mac, mul, max, , max3, min, subrev Bitwise: {B32, B64} - and, or, xor, mov, lshrrev, lshlrev, ashlrev, ashrrev, bfe, bfi, cndmask Compare: {U32, I32, F32} - cmp_{ lt, eq, le, gt, lg, ge, nge, nlg, ngt, nle, neq}
Scalar	ALU: {U32, I32, F32, F64} - add, addc, sub, mad, madmk, mac, mul, max, , max3, min, subrev Bitwise: {B32, B64} - and, or, xor, mov, lshrrev, lshlrev, ashlrev, ashrrev, bfe, bfi, cndmask Compare: {U32, I32, F32} - cmp_{ lt, eq, le, gt, lg, ge, nge, nlg, ngt, nle, neq} Conditional: {U32, I32, F32} - cmp_{ lt, eq, le, gt, lg, ge, nge, nlg, ngt, nle, neq}
Memory	**SMRD[?]**: {U32, I32, F32, F64} - add, addc, sub, mad, madmk, mac, mul, max, , max3, min, subrev **vector[?]**: {U32, I32, F32} - cmp_{ lt, eq, le, gt, lg, ge, nge, nlg, ngt, nle, neq} **data share[?]**: {U32, I32, F32} - cmp_{ lt, eq, le, gt, lg, ge, nge, nlg, ngt, nle, neq}



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

Table 4: Supported ISA

Issue/Schedule (Fig. 3c) 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 (Fig. 3d) Vector ALUs perform arithmetic or logical operation (integer and floating point) on the data for all 64 threads of the wavefront, depending on the execution mask. We have 16-wide vector ALUs, four each of integer and floating point – one wavefront is processed as 4 batches of 16.

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

Load Store Unit (Fig. 3e) 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 1024 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. 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. This will be further discussed in the design choices section. The scalar register file is organized as 4 banks. There are also a set of special registers associated with each wavefront namely exec (a 64 bit mask which governs which of the 64 threads will execute an instruction), vcc (a 64 bit value which 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).

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 (1,5)	Realistic	Easy to change	Low
Types of FU	Simplified	Easy to change	High

†Fetch optimized for cache-hit, rest sized for balanced machine.

Table 5: Impact of Design Choices

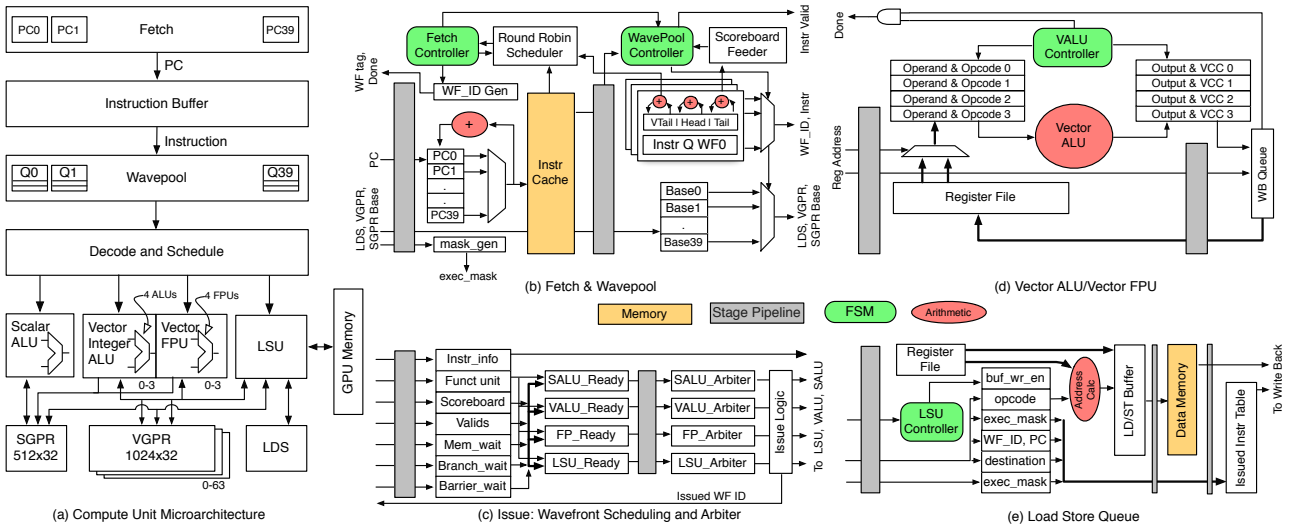


Figure 3: MIAOW Compute Unit Block Diagram and Design of Submodules

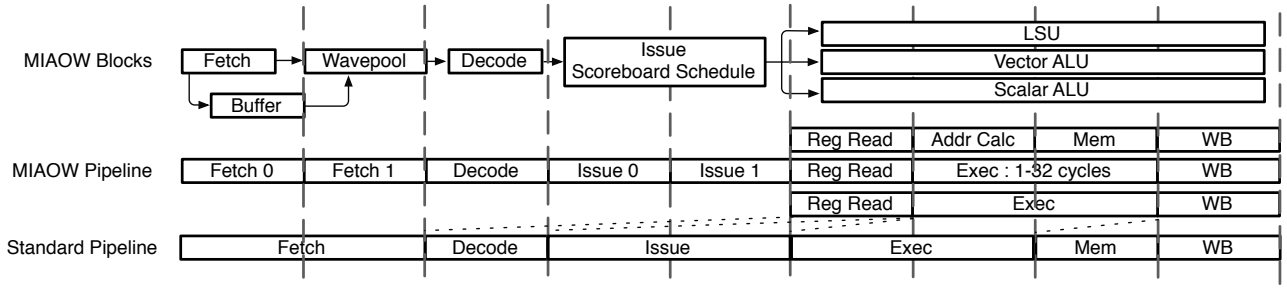


Figure 4: MIAOW Compute Unit Pipeline stages

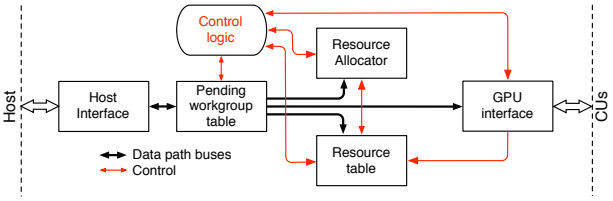


Figure 5: MIAOW Ultra Threaded Dispatcher Block Diagram

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 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 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, making the dispatcher very flexible.

2.5. Design choices

Table 5 summarizes the most 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 [1], 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 bal-

ance which uses evaluation content in Section 4. In short, our design choices lead to a realistic and balanced design.

Fetch bandwidth (1) We optimized the design assuming instruction cache hits and single instruction fetch. In contrast, the GCN specification has fetch bandwidth on the order of 16 or 32 instructions per fetch, presumably matching a cache-line. It includes an additional buffer between fetch and wavepool to buffer the multiple fetched instructions for each wavefront. MIAOW's design can be changed easily by changing the interface between the Fetch module and Instruction memory.

Wavepool slots (6) Based on the back-of-the-envelope analysis of load balance, we decided on 6 wavepool slots. Our design evaluations show that all 6 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 has many more slots to accommodate the wider fetch. The number of queue slots is parameterized and can be easily changed. Since this pipeline stage has 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 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 4 vector and 1 scalar instruction every cycle, while each wavefront is generally composed of 64 threads and the vector ALU being 16 wide. We suspect the actual issue width for GCN is lower.

# of integer & 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 by Rodinia benchmarks indicate the number is justified. These values are parameterizable in the top level module and these are major contributors to area and power.

# of register ports (1,5) We use two register file designs. The first design is a single ported SRAM based register file generated using synopsys design compiler which is heavily banked to reduce contention. In simulations, we observed that there was contention on less then 1% of the accesses and hence we are using a behavioral module. This decision will result in a model with a small under-estimation of area and power and over-estimation of performance. This design, however, is likely to be similar to GCN and we report the power/area/performance results based on this register file. Since it includes proprietary information and the configuration cannot be distributed, we have a second verison - a flip-flop based register file design which has five ports. While we have explored these two register file designs, 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 [4] by swapping out our module.

# 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. 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 is negligible.

Types of functional units GCN and other industry GPUs have more specialized FUs to support graphic computations. This choice restricts MIAOW's usefulness to model graphics workloads. It has some impact on realism and flexibility depending on the workloads studied. However this aspect is extendable by creating new datapath modules.

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 vs. 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 to 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 complexity, area and power of different scheduling policies.

OCN, L2-cache, Memory, Memory Controller Simpler PLI models are used for the implementation of 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 which is non-blocking (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 sub-system techniques [48, 20].

3.2. Verification and Physical Design

We followed a standard verification flow of unit tests and in-house developed random program generator based regression tests with architectural trace comparison to an instruction emulator. Specifically, we used Multi2sim as our reference

instruction emulator and enhanced it in various ways with bug-fixes and to handle challenges in the multithreaded nature and out-of-order retirement of wavefronts. We used the AMD OpenCL compiler and device drivers to generate binaries.

Physical design was relatively straight-forward using Synopsys Design Compiler for synthesis and IC Compiler for place-and-route with Synopsys 32nm library. Based on Design Compiler synthesis, our CU design's area is $15mm^2$ and it consumes on average 1.1W of power across all benchmarks. We are able to synthesize the design at an acceptable clock period range of 4.5ns to 8ns, and for our study we have chosen 4.5ns. Layout introduces challenges because of the dominant usage of SRAM and register files and automatic flat layout without floorplanning fails. While blackboxing these produced a layout, detailed physical design is future work.

3.3. FPGA Implementation

In addition to software emulation, MIAOW was successfully synthesized on a state-of-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

Design Neko is composed of a MIAOW compute unit attached to an embedded Microblaze softcore processor via the AXI interconnect bus. The Microblaze implements the ultra-threaded dispatcher in software, handles pre-staging 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 SIMD and SIMF) than a standard MIAOW compute unit which has four SIMD and four SIMF units for vector integer and floating point operations respectively. The consequence of this is that while Neko can perform any operation a full compute unit can, its throughput is lower due to the fewer computational resources. Mapping the ALUs to Xilinx provided IP cores (or DSP slices) may help in fitting more onto the FPGA as the SIMD and especially SIMF units consume a large proportion of the LUTs. This however changes the latencies of these significantly (multiplication using DSP slices is a 6 stage pipeline, while using 10 DSPs can create a 1 stage pipeline) and will end up requiring modifications to the rest of the pipeline and takes away from ASIC realism. We defer this for future work. One other difference is Neko's register file architecture. Mapping MIAOW's register files naively to flip-flops causes excessive usage and routing difficulties considering, especially with the vector ALU register file which has 65536 entries. Using block RAMs is not straight-forward 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 6 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 using Neko. Our paper's reports results using the full MIAOW design using long VCS simulations.

Module	LUT Count	# BRAMs	Module	LUT Count	# BRAMs
Decode	3474	-	SGPR	647	8
Exec	8689	-	SIMD	36890	-
Fetch	22290	1	SIMF	55918	-
Issue	36142	-	VGPR	2162	128
SALU	1240	-	Wavepool	27833	-
Total	195285	137			

Table 6: Resource utilization

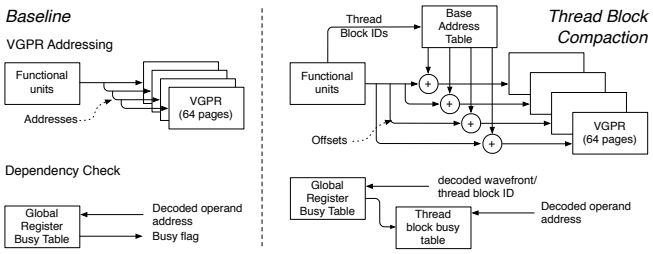


Figure 6: Critical modifications in Thread block compaction

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 a 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 7 summarizes the methodology and key results and show MIAOW is realistic.

For performance studies we choose six OpenCL benchmarks that are part of the Multi2sim environment, which we list along with three characteristics – # work groups, # wavefronts per workgroup, and # compute-cycles per work group: BinarySearch (4, 1, 289), BitonicSort (1, 512, 97496), MatrixTranspose (4, 16, 4672), PrefixSum (1, 4, 3625), Reduction (4, 1, 2150), ScanLargeArrays (2, 1, 4). MIAOW can also run four Rodinia [9] benchmarks at this time – kmeans, nw, backprop and gaussian. We use these longer benchmarks for the case studies in Section 5 onward[3].

5. Physical Design Perspective

Description: Fung et al. proposed Thread Block Compaction (TBC) [16]. which belongs in a large body of work

[3]Others don't run because of they use instructions outside MIAOW's subset.

Area analysis

Goal	○ Is MIAOW's total area and breakdown across modules representative of industry designs?
Method	○ Synthesized with Synopsys 1-ported register-file ○ For release, 5-ported flip-flop based regfile. ○ Compare to AMD Tahiti (SI GPU) implemented at 28nm; scaled to 32nm for absolute comparisons
Key results	○ Area breakdown matches intuition; 30% in functional units & 54% in register files. ○ Total area using 1-port Synopsys RegFile 9.31 mm^2 compared to 6.92mm^2 for Tahiti CU ○ Higher area is understandable: our design is not mature, designers are not as experienced, our functional units are quite inefficient (from Opencores.org), and not optimized as industry functional units would be.

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 [22]. Breakdown *and* total power for industry GPUs not publicly available.
Key results	○ MIAOW breakdown: FQDS: 13.1%, RF: 16.9% FU: 69.9% ○ NVIDIA breakdown: FQDS: 36.7%, RF: 26.7% FU: 36.7% ○ Compared to model more power in functional units (likely because of MIAOW's inefficient FUs); FQDS and RF roughly similar contributions in MIAOW and model. ○ Total power is 1.1 Watts. No comparison reference available. But we feel this is low. Likely because Synopsys 32nm technology library is targeted to low power design (1.05V, *300MHz typical frequency*

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 balance and 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 3 benchmarks.
○ On another three, MIAOW's CPI is 2× lower, the reasons for which are many: i) the instructions on the NVIDIA GPU are PTX-level and not native assembly; ii) cycle measurement itself introduces noise; and iii) microarchitectures are different, so CPIs will be different.
○ CPIs being in similar range shows MIAOW's realism
○ The # of wavepool queue slots was rarely the bottleneck: in 50% of the cycles there was at least one free slot available (with 2 available in 20% of cycles).
○ The integer vector ALUs were all relatively fully occupied across benchmarks, while utilization of the 3rd and 4th FP vector ALU was less than 10%.
○ MIAOW seems to be a balanced design.

Table 7: Summary of investigations of MIAOW's realism

on warp scheduling [31, 44, 16, 43, 35, 25, 24], 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 are formed 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. 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 [16]. In MIAOW, the modules that needed significant modifications were: fetch, wavepool, decode, 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 after reconvergence) 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 it does not interfere with any other logic in the SALU apart from the branch unit.

The issue and VGPR modules suffered more drastic modifications, shown in figure 6. 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 6 , 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 1024 for the entire CU in the original implementation. In figure 6, the baseline issue stage observed in the lower-left

corner and in the lower-right 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 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 6.

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 hand-written assembly resembling benchmarks in [16]. 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 post-denominator stack-based reconvergence mechanism (PDOM) [33], without any kind of wavefront formation. We compiled our tests and ran them on two versions of MIAOW: one with PDOM and other with TBC.

Quantitative results The performance results obtained matched the results from [16]: 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 from 8.00ns to 10.59ns. We also observed that the issue stage area grew from $0.43mm^2$ to $1.03mm^2$.

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 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 re-pipelining, 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 [6] (Aged-SDMR). They show that Aged-SDMR provides low design complexity, low overheads, generality (supporting various types of wearout including soft and hard breakdown) 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 and by sampling and virtually aging 1% of the epochs provides an effective system. Their design (shown in Figure 7) is developed in the context of multi-core CPUs and requires the following: i) operating system involvement to schedule the sampled threads, ii) some kind of system-level checkpoints (like Revive [41], ReviveIO [34], Safetynet [51]) at the end of every epoch, iii) some system and microarchitecture support for avoiding incoherence between the sampled threads [50], iv) some microarchitecture support to compare the results of the two cores, and v) 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 [10], having this capability is desirable.

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 (i), (ii) and (iii) 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. Comparision 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 7 shows the implemenation 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, i.e for a sampling rate of 1%, 1 out of 100 work groups are dispatched to another compute unit called checker. This is run under the stressed conditions and we disable the global writes so that it does not affect the normal execution of the workgroups in the checked CU. We could 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

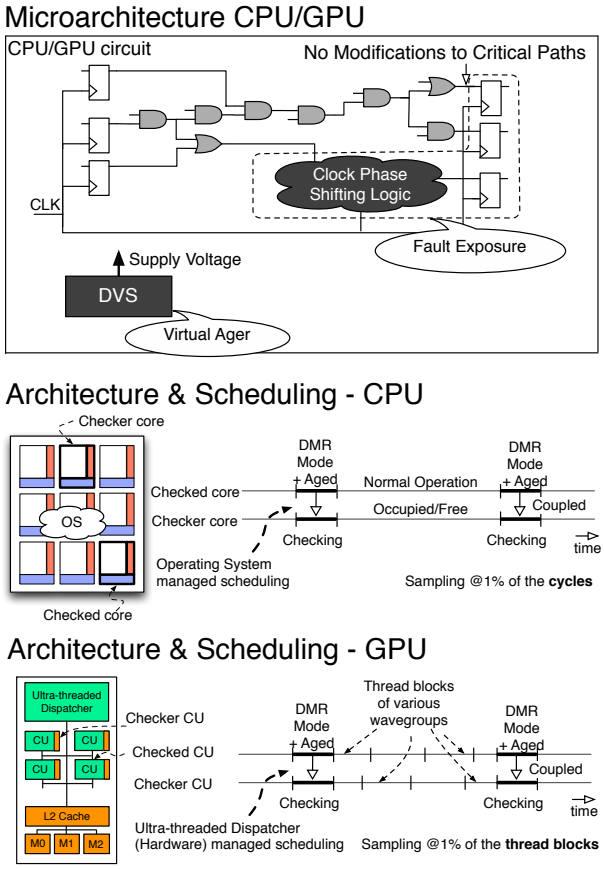


Figure 7: Virtually Aged SDMR implemenation on a GPU

	MIAOW		OpenSPARC [6]	
	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%

† OpenSPARC: with 16K L1 Instruction & 8K L1 Data cache.

Table 8: Overheads for fault exposure

the checked CU for the sampled workgroup. The checksums are compared for correcteness and a mismatch detects a fault. 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 the 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 is relatively small supporting and thus is of low design complexity.

Quantitative results and Analysis: The first important result obtained from this study 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 8, logic area overheads are similar to CPUs and are small.

We also report on performance overheads of one pathological case. There could be non-trivial 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, 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 35% and 50% and with 10% it becomes 80% and 103% 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 [14] and Blueshift [19] 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 [32] from the iGPU design which supports timing speculation. Since these are 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 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} vs. logic delay for a set of paths to determine an empirical mapping of delay change to V_{dd} reduction, thus obtain error-rate vs. V_{dd} reduction. Other approaches like approximating delay-aware simulation and timing-speculation emulation could also be used [36, 8, 40].

Quantitative results and analysis: Figure 8 shows the variation of error rate as a function of operating voltage for 9 benchmarks. Observed trends are similar to those from Razor [14] and suggest timing speculation could be of value to

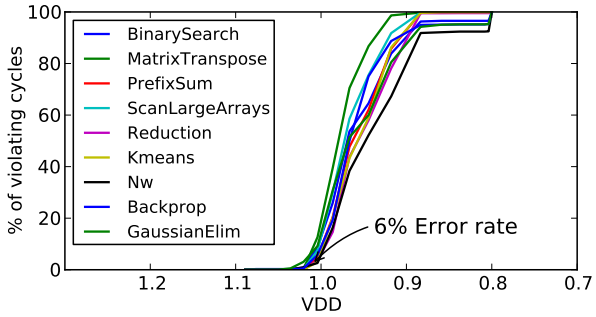


Figure 8: Voltage level vs % of violating cycles.

GPUs as well. Error rate grows slowly at first with V_{dd} reduction before a rapid increase at some point - at 6% error rate, there is a 115 mV voltage reduction, 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 of simulator characterization

Description: Previous hardware-based research studies on transient faults have focused on CPUs [55] and a recent GPU study focused on a simulator based evaluation [52]. Wang et al., [55] show that simulator based studies miss many circuit level phenomenon 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 [42] 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 2000 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 5000 cycles after fault injection. In every experiment, we capture changes in all architecture state 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 which has no fault injection. Every experiment is classified into one of four types:

- *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 1 or more flip-flops ⇒ no corruption yet, but can't rule out SDC.

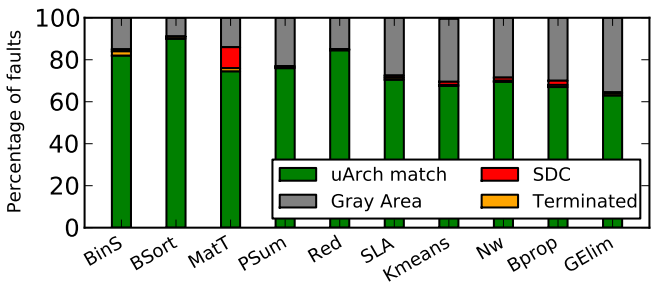


Figure 9: Fault injection results

Quantitative results and analysis: We run three variants of this experiment and report in Figure 9, the detailed results for the last variant. 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* is run without the register file. Though there is a considerable reduction in number of runs classified as Gray Area, benchmarks like BinarySearch, MatrixTranspose, Reduction, Kmeans and Nw still have more than 50% runs in Gray Area. [4]. The last variant, *MIAOW-noRF/VF: exclude the Vector FP units also since our benchmarks only light exercise them.* Gray area is now approximately 20-35% of fault sites which is closer to CPUs but larger. This is unexpected – the reason for the Gray area in CPUs is the large amount of speculative state which a GPU lacks. Also, the Gray area in 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. We conclude with a caveat that further analysis with many workloads is necessary to generalize the result.

8. Conclusion

This paper 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 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 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.

[4] 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.

References

[1] "Amd graphics cores next architecture." [Online]. Available: http://www.amd.com/la/Documents/GCN_Architecture_whitepaper.pdf

[2] "Barrasim: Nvidia g80 functional simulator." [Online]. Available: https://code.google.com/p/barra-sim/

[3] "Reference guide: Southern islands series instruction set architecture, http://developer.amd.com/wordpress/media/2012/10/AMD_Southern_Islands_Instruction_Set_Architecture.pdf."

[4] M. Abdel-Majeed and M. Annavaram, "Warped register file: A power efficient register file for gpgpus," in *HPCA*, 2013.

[5] A. Bakhoda, G. Yuan, W. Fung, H. Wong, and T. Aamodt, "Analyzing CUDA workloads using a detailed GPU simulator," in *ISPASS '09*.

[6] R. Balasubramanian and K. Sankaralingam, "Virtually-aged sampling dmr: Unifying circuit failure prediction and circuit failure detection," in *Proceedings of the 46th International Symposium on Microarchitectures "'(MICRO)"'*, 2013.

[7] ——, "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)"'*, 2014.

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

[9] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC)*, ser. IISWC '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 44–54. [Online]. Available: http://dx.doi.org/10.1109/IISWC.2009.5306797

[10] J. Chen, "GPU technology trends and future requirements," in *IEDM '09*.

[11] N. K. Choudhary, S. V. Wadhavkar, T. A. Shah, H. Mayukh, J. Gandhi, B. H. Dwiel, S. Navada, H. H. Najaf-abadi, and E. Rotenberg, "Fabscalar: composing synthesizable rtl designs of arbitrary cores within a canonical superscalar template," in *ISCA '11*.

[12] V. del Barrio, C. Gonzalez, J. Roca, A. Fernandez, and E. Espasa, "Attila: A cycle-level execution-driven simulator for modern gpu architectures," in *ISPASS '06*.

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

[14] D. Ernst, N. S. Kim, S. Das, S. Pant, R. Rao, T. Pham, C. Ziesler, D. Blaauw, T. Austin, K. Flautner, and T. Mudge, "Razor: A low-power pipeline based on circuit-level timing speculation," in *MICRO '03*.

[15] M. Fried, "Gpgpu architecture comparison of ati and nvidia gpus." [Online]. Available: www.microway.com/pdfs/GPGPU_Architecture_and_Performance_Comparison.pdf

[16] W. W. L. Fung and T. M. Aamodt, "Thread block compaction for efficient simt control flow," in *HPCA '12*.

[17] ——, "Thread block compaction for efficient simt control flow," in *Proceedings of the 2011 IEEE 17th International Symposium on High Performance Computer Architecture*, ser. HPCA '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 25–36. [Online]. Available: http://dl.acm.org/citation.cfm?id=2014698.2014893

[18] J. Gaisler, "Leon sparc processor," 2001.

[19] B. Greskamp, L. Wan, U. Karpuzcu, J. Cook, J. Torrellas, D. Chen, and C. Zilles, "Blueshift: Designing processors for timing speculation from the ground up." in *HPCA '09*.

[20] B. A. Hechtman and D. J. Sorin, "Exploring memory consistency for massively-threaded throughput-oriented processors," in *ISCA*, 2013.

[21] S. Hong and H. Kim, "An analytical model for a gpu architecture with memory-level and thread-level parallelism awareness," in *ISCA '09*.

[22] ——, "An integrated gpu power and performance model," in *ISCA '10*.

[23] H. Jeon and M. Annavaram, "Warped-dmr: Light-weight error detection for gpgpu," in *MICRO '12*.

[24] A. Jog, O. Kayiran, A. K. Mishra, M. T. Kandemir, O. Mutlu, R. Iyer, and C. R. Das, "Orchestrated scheduling and prefetching for gpgpus," in *ISCA*, 2013.

[25] A. Jog, O. Kayiran, N. C. Nachiappan, A. K. Mishra, M. T. Kandemir, O. Mutlu, R. Iyer, and C. R. Das, "Owl: cooperative thread array aware scheduling techniques for improving gpgpu performance," in *ASPLOS*, 2013.

[26] H. Kim, R. Vuduc, S. Baghsorkhi, J. Choi, and W. mei Hwu, *Performance analysis and tuning for GPGPUs. Synthesis Lectures on Computer Architecture.* Morgan & Claypool.

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

[28] J. Leng, T. Hetherington, A. ElTantawy, S. Gilani, N. S. Kim, T. M. Aamodt, and V. J. Reddi, "Gpuwattch: Enabling energy optimizations in gpgpus," in *ISCA '13*.

[29] "User guide for nvptx back-end." [Online]. Available: http://llvm.org/docs/NVPTXUsage.html

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

[31] J. Meng, D. Tarjan, and K. Skadron, "Dynamic warp subdivision for integrated branch and memory divergence tolerance," in *ISCA '10*.

[32] J. Menon, M. De Kruijf, and K. Sankaralingam, "igpu: exception support and speculative execution on gpus," in *ISCA '12*.

[33] S. S. Muchnick, *Advanced compiler design implementation*. Morgan Kaufmann, 1997.

[34] J. Nakano, P. Montesinos, K. Gharachorloo, and J.Torrellas, "Revivei/o: efficient handling of i/o in highly-available rollback-recovery servers," in *HPCA '06*.

[35] V. Narasiman, M. Shebanow, C. J. Lee, R. Miftakhutdinov, O. Mutlu, and Y. N. Patt, "Improving gpu performance via large warps and two-level warp scheduling," in *MICRO '11*.

[36] S. Nomura, K. Sankaralingam, and R. Sankaralingam, "A fast and highly accurate path delay emulation framework for logic-emulation of timing speculation," in *ITC '11*.

[37] "Nvidia cuda profiler user guide." [Online]. Available: http://docs.nvidia.com/cuda/profiler-users-guide/index.html

[38] "Openrisc project, http://opencores.org/project,or1k."

[39] "OpenSPARC T1, http://www.opensparc.net."

[40] A. Pellegrini, K. Constantinides, D. Zhang, S. Sudhakar, V. Bertacco, and T. Austin, "Crashtest: A fast high-fidelity fpga-based resiliency analysis framework," in *CICC '08*.

[41] M. Prvulovic, Z. Zhang, and J. Torrellas, "Revive: cost-effective architectural support for rollback recovery in shared-memory multiprocessors," in *ISCA '02*.

[42] P. Rech, C. Aguiar, R. Ferreira, C. Frost, and L. Carro, "Neutron radiation test of graphic processing units," in *IOLTS '12*.

[43] M. Rhu and M. Erez, "Capri: Prediction of compaction-adequacy for handling control-divergence in gpgpu architectures," in *ISCA '12*.

[44] T. G. Rogers, M. O'Connor, and T. M. Aamodt, "Cache-conscious wavefront scheduling," in *MICRO '12*.

[45] R. M. Russell, "The CRAY-1 Computer System," *Communications of the ACM*, vol. 22, no. 1, pp. 64–72, January 1978.

[46] J. Sartori, B. Ahrens, and R. Kumar, "Power balanced pipelines," in *HPCA '12*.

[47] J. W. Sim, A. Dasgupta, H. Kim, and R. Vuduc, "A performance analysis framework for identifying performance benefits in gpgpu applications," in *PPOPP '12*.

[48] I. Singh, A. Shriraman, W. W. L. Fung, M. O'Connor, and T. M. Aamodt, "Cache coherence for gpu architectures," in *HPCA*, 2013.

[49] B. Smith, "Architecture and applications of the HEP multiprocessor computer system. In SPIE Real Time Signal Processing IV, pages 241–248, 1981."

[50] J. C. Smolens, B. T. Gold, B. Falsafi, and J. C. Hoe, "Reunion: Complexity-effective multicore redundancy," in *MICRO 39*, 2006.

[51] D. J. Sorin, M. M. K. Martin, M. D. Hill, and D. A. Wood, "Safetynet: improving the availability of shared memory multiprocessors with global checkpoint/recovery," in *ISCA '02*.

[52] J. Tan, N. Goswami, T. Li, and X. Fu, "Analyzing soft-error vulnerability on gpgpu microarchitecture," in *IISWC '11*.

[53] R. Ubal, B. Jang, P. Mistry, D. Schaa, and D. Kaeli, " Multi2Sim: A Simulation Framework for CPU-GPU Computing ," in *PACT '12*.

[54] W. J. van der Laan, "Decuda SM 1.1 (G80) disassembler," https://github.com/laanwj/decuda.

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

[56] N. Wang and S. Patel, "Restore: Symptom-based soft error detection in microprocessors," *Dependable and Secure Computing, IEEE Transactions on*, vol. 3, no. 3, pp. 188 –201, 2006.

[57] Y. Zhang, L. Peng, B. Li, J.-K. Peir, and J. Chen, "Architecture comparisons between nvidia and ati gpus: Computation parallelism and data communications," in *IISWC '11*.