HAWS: Accelerating GPU Wavefront Execution through Selective Out-of-order Execution

XUN GONG, XIANG GONG, LEIMING YU, and DAVID KAEI, Northeastern University, USA

Graphics Processing Units (GPUs) have become an attractive platform for accelerating challenging applications on a range of platforms, from High Performance Computing (HPC) to full-featured smartphones. They can overcome computational barriers in a wide range of data-parallel kernels. GPUs hide pipeline stalls and memory latency by utilizing efficient thread preemption. But given the demands on the memory hierarchy due to the growth in the number of computing cores on-chip, it has become increasingly difficult to hide all of these stalls.

In this article, we propose a novel Hint-Assisted Wavefront Scheduler (HAWS) to bypass long-latency stalls. HAWS starts by enhancing a compiler infrastructure to identify potential opportunities that can bypass memory stalls. HAWS includes a wavefront scheduler that can continue to execute instructions in the shadow of a memory stall, executing instructions speculatively, guided by compiler-generated hints. HAWS increases utilization of GPU resources by aggressively fetching/executing speculatively. Based on our simulation results on the AMD Southern Islands GPU architecture, at an estimated cost of 0.4% total chip area, HAWS can improve application performance by 14.6% on average for memory intensive applications.

CCS Concepts: • **Computer systems organization** → **Architectures; Parallel architectures; Single instruction, multiple data**; • **Software and its engineering** → **Compilers**;

Additional Key Words and Phrases: Computer architecture, GPU architecture, simulation, wavefront scheduler

ACM Reference format:

Xun Gong, Xiang Gong, Leiming Yu, and David Kaeli. 2019. HAWS: Accelerating GPU Wavefront Execution through Selective Out-of-order Execution. *ACM Trans. Archit. Code Optim.* 16, 2, Article 15 (April 2019), 22 pages.

<https://doi.org/10.1145/3291050>

1 INTRODUCTION

Graphics Processing Units (GPUs) are widely used for accelerating general-purpose applications, given their impressive computational power. More than 60 systems placed in the 2017 TOP500 supercomputers are equipped with AMD and NVIDIA GPUs as accelerators. GPUs have thousands of SIMT cores on chip, making them the platform of choice for accelerating dense matrix operations, neural networks training [6], and machine-learning algorithms [2].

Authors' addresses: X. Gong, X. Gong, L. Yu, and D. Kaeli, Northeastern University, 360 Huntington Avenue, Electrical and Computer Engineering, Boston, MA, 02115, USA; emails: gong.xun@husky.neu.edu, xgong@ece.neu.edu, yu.lei@husky.neu.edu, kaeli@ece.neu.edu.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

© 2019 Copyright held by the owner/author(s). Publication rights licensed to ACM.

1544-3566/2019/04-ART15

<https://doi.org/10.1145/3291050>

To achieve double-digit speedups on a variety of applications, GPUs exploit the Thread-level Parallelism (TLP) exposed by the programmer, using two different approaches. First, GPUs adopt a Single Instruction Multiple Thread (SIMT) [14] execution model. The threads executing the same instruction are grouped into a fixed sized batch, called a *wavefront* (AMD) or a *warp* (NVIDIA). They execute the same instruction on multiple execution lanes. Second, GPUs execute many wavefronts concurrently on a single processing unit. When one wavefront is stalled, the wavefront can be preempted, allowing a non-stalled wavefront to execute. A large portion of GPU die area is dedicated to execution units to support a high degree of concurrency. Execution lanes on a GPU use in-order processing pipelines and do not perform branch prediction. A hardware scheduler hides latencies associated with wavefront execution stalled by long latency operations, including data dependencies and branch divergence, by swapping computation to non-blocked wavefronts.

However, even given the mechanisms described above to maximize execution efficiency, GPU resources are still underutilized. When a memory-intensive workload needs to load a block of data from off-chip DRAM, many threads compete for the limited memory bandwidth to DRAM, saturating the memory interconnect. When we encounter this situation, a new memory request cannot be serviced until older memory requests complete. Since the memory requests are not pipelined, the wavefront scheduler can not totally hide these long latency memory operations by interleaving wavefronts. So wavefronts will stall on instructions that depend on the operand loaded by long latency memory instructions. The number of active wavefronts will decrease as more wavefronts hit the stall point. All the wavefronts will stall and wait until the results from the long latency memory instruction are loaded into registers. This contention can result in even more memory traffic, which may stall the wavefronts even longer.

In this article, we propose a dynamic approach to improve GPU performance. We provide a compiler pass to analyze and mark the instructions that are independent of long latency memory instructions. These instructions are in the *shadow* of the memory instructions. We design a compiler-generated hint that encodes this information and embeds it into the instruction stream. Based on these hints, the wavefront scheduler can choose to issue independent instructions and execute them out-of-order when all wavefronts are stalled due to memory system saturation. The goal is to keep GPU resource utilization high, while allow programs to execute past dependencies. Unlike CPUs, which use techniques such as reservation stations and reorder buffers to support out-of-order execution, it would be costly to replicate hardware components on a GPU, given that there could be hundreds of wavefronts executing concurrently. Instead, HAWS uses hints provided by the compiler to schedule and execute instructions in a selective, out-of-order, fashion.

HAWS executes non-speculative instructions and schedules wavefronts based on the scheduling algorithm we choose (round-robin, GTO, etc.). When all wavefronts stall because of long latency memory operations, HAWS triggers our hint-assisted scheduler, which will only fetch and execute the instructions that are independent from the long latency memory operation. We will not fetch or execute any instruction that has a dependency on a long latency instruction.

In hint-assisted scheduling execution mode, HAWS executes instructions out of order, which introduces the possibility of write-after-read (WAR) and write-after-write (WAW) hazards. There should be no read-after-write (RAW) hazards, since HAWS only fetches and executes instructions without dependencies. Similar to the approach used on a CPU, we employ register renaming to handle WAW and WAR hazards. Hints also offer us an efficient way to rename registers. But unlike a CPU, which fetches instructions in order, HAWS fetches code out of order, only fetching the instructions specified by the hint. The details of HAWS will be described in detail in the following sections.

Fig. 1. Architecture of a compute unit.

In this article, we make the following contributions:

- We analyze dependencies related to long latency memory instructions in a variety of applications, evaluating the potential performance benefits of HAWS.
- We design a novel hint encoding format, embedding the hint in the binary instructions by using the unused bits of each instruction to assist the wavefront scheduler, and efficiently rename registers and use the results from the hint-assisted mode wisely.
- We propose a novel wavefront scheduling algorithm named HAWS, which uses GPU resources more efficiently and can hide long latency memory operations.
- We evaluate the benefits of HAWS across a variety of GPU benchmarks on the AMD Southern Island GPU architecture. We improve performance by up to 34.2% for memory intensive applications.

This article is organized as follows. In Section 2, we present background and describe our baseline GPU architecture. In Section 3, we present our motivation for this work. Section 4 describes how a program will run in the HAWS framework. In Section 5, we discuss our hint format and describe the microarchitecture of HAWS. In Section 6, we evaluate our design and present simulation results. Section 7 discusses related work. We conclude in Section 8 and suggest directions for future work.

2 BACKGROUND

2.1 SIMT Programming Model

The Single Instruction Multiple Thread (SIMT) programming model is the execution model used in modern GPUs and is supported by a number of programming frameworks including OpenCL [29], CUDA [22], and HSA [13]. In this article, we will be targeting OpenCL as run on an AMD GPU, so will use OpenCL terminology. Functions executed on an OpenCL device are called kernels, which

Fig. 2. Instruction sequences produced for different benchmarks.

specify an NDRange. When a kernel is called, N copies of the kernel are executed in parallel by N different OpenCL work-items. An instance of an OpenCL kernel is called a work-item. A wavefront of 64 work-items are grouped to execute together, similar to a CUDA warp. All work-items in the same wavefront share the same program counter and execute the same instruction. A work-group consists of several wavefronts. All work-items in the same work-group have two basic properties: (i) within a single work-group, work-items can perform efficient synchronization operations, and (ii) work-items within the same work-group can share data through a low-latency shared memory. All work-groups from the NDRange and share a common global memory.

2.2 Baseline GPU Architecture

A GPU is comprised of multiple compute units (CU). During execution time, one compute unit can have one or more work-groups allocated. These work-groups are split into wavefronts, which are assigned to compute units that execute a thread. Compute units are identical, with the major units and organization provided in Figure 1. On every cycle, the CU front-end fetches instructions from instruction memory for the different wavefronts, and sends them to the appropriate execution unit. There are several execution units present in a compute unit, such as the vector memory unit, the scalar unit, the branch unit, the vector memory unit, the LDS unit (Local Data Store), and a set of SIMD units. The LDS unit is interconnected to local memory to service instructions, while the scalar and vector memory units can access global memory, shared by all compute units.

In our baseline model, each compute unit features a wavefront scheduler and a set of 4 SIMD execution units. The wavefront scheduler keeps assigning different wavefronts, from the wavefront pool, to the SIMD units when they become available. The number of SIMD units matches exactly the number of wavefront pools, so wavefronts from different wavefront pools share a common front-end. When an instruction is selected by the scheduler, before issuing it to the execution units, the wavefront scheduler uses the scoreboard to perform a dependency check, and generates a ready bit if the instruction passes the check. If the instruction is ready to issue, and there are still available resources, then it is sent to the corresponding unit to execute. Each SIMD unit contains 16 lanes or stream cores, and it takes the SIMD unit four cycles to commit a wavefront, since there are 64 work-items in a single wavefront.

3 MOTIVATION

The performance of modern processors still suffers from the memory wall [17] and branch penalties. One way to alleviate this problem is to introduce instruction hints to increase the cache hit

```

1  ISETP.LT.AND P0, PT, R7, R4, PT;
2  @P0 BRA I22;
3  IMUL R9, R5, R4;
4  IADD R9, R9, R7;
5  SHL R9, R9, 0x2;
6  IMUL R9, R5, R4;
7  IADD R9, R9, R7;
8  SHL R9, R9, 0x2;
9  IADD R9, R0, R9;
10 LD R10, [R9];
11 IMUL R9, R7, R4;
12 IADD R9, R9, R6;
13 SHL R9, R9, 0x2;
14 IADD R9, R2, R9;
15 MOV R9, R9;
16 LD R9, [R9];
17 FMUL R9, R10, R9;
18 FADD R8, R8, R9;
19 IADD R7, R7, 0x1;
20 BRA I1;
21 IMUL R0, R5, R4;

```

```

{ int row = blockIdx.y * TILE_WIDTH + threadIdx.y;
  int col = blockIdx.x * TILE_WIDTH + threadIdx.x;
  float pvalue = 0;
  for (int k = 0; k < width; ++k)
    pvalue += md[row * width + k] * nd[k * width + col];
  pd[row * width + col] = pvalue;
}

```

RAW Stall

Fig. 3. Binary instructions from a GPU kernel.

rate and decrease the branch misprediction rate. In some power-efficient processor models, architects elected to remove the hardware branch predictor and use a software branch hint to recover the lost performance. This was realized in the IBM's Cell Synergistic Processing Units [10]. When using software branch hints, hint instructions are inserted in the application, specifying that the branch instructions located at a specific PC address will jump to a specific target address. The processor will start to speculatively execute target instructions after executing a hint instruction, when the specified branch instruction is still in flight. Our approach for GPUs is inspired by this approach, specifically when all wavefronts are stalled waiting for long latency memory instructions. The wavefront scheduler will keep issuing instructions in a selective out-of-order fashion, according to the instruction hint.

In this section, we begin by walking through a sample GPU execution flow using an simple code example. Figure 3 shows an instruction snippet for a matrix multiplication GPU kernel, which is commonly used a wide class of GPU applications, including machine learning [2] and image processing [23].

In the kernel, there is a loop that calculates each element in the output matrix. Every time the kernel calculates the *pvalue* in the loop, it needs to read data from both input matrices *md* and *nd*. The kernel also calculates the value of *k* during each iteration, whose value is independent from the values in *md* and *rd*.

Inspecting the assembly code, the kernel reads a value from global memory from matrices *md* and *nd*, which are instructions 10 and 16 in assembly code listing. When the PC reaches instruction 17, which is a multiplication of *md* and *nd*, since R9 is a source operand whose values is generated by instruction 16, the wavefront will stall due to the long latency memory operation. What is worse, as future wavefronts stall at this point, memory contention will further increase the stall time. However, not all of the instructions after instruction 17 are dependent on instruction 16, so a number of instructions can still be fetched and executed if we selectively perform out-of-order execution. As we discussed earlier, for each iteration, the kernel calculates the value of *k* and compares it with the value of *width*. Both the values of *k* and *width* are independent of *md* and *rd*. Returning to the assembly code, it is clear that instruction 19, 20, and instruction 1-9, whose source operands are independent of register R9, which is the destination operand of instruction 16. So there are a significant number of instructions we can fetch and execute if we keep issuing instructions in a selective out-of-order fashion in this example.

Unlike GPUs, CPUs rely on extra hardware, such as reservation stations and reorder buffers, to support aggressive out of order execution and increase instruction level parallelism (ILP). So while HAWS modifies a GPU to incorporate some CPU-like features, our approach uses static analysis

Fig. 4. Percentage of long latency RAW stalls.

by the compiler to identify independent instructions, producing instruction hints to assist the wavefront scheduler. HAWS can support selective out of order execution by following guidance provided by a hint. Unlike reservation stations, HAWS can increase GPU ILP by only adding a small amount of extra hardware. All data dependency analysis is handled by the compiler. Similar to IBM’s Cell Synergistic Processing Units [10], HAWS does not significantly impact the GPU’s power efficiency, while increasing performance. More details of these benefits will be described in the following sections.

HAWS selects the instructions that are in the shadow of a long latency memory operation, and are independent of the memory operation. To quantify the number of independent instructions present in the program, and evaluate the potential benefits of HAWS, we analyze a series of benchmarks selected from the AMD OpenCL SDK 2.5. In Figure 2, we show the instruction sequence for a kernel binary. The dark grey bars represent long latency memory operations, and the black bars represent the instructions that are independent of the previous memory instruction. We can clearly see there are some black bars appear after the dark gray bars. This trend is present in all of our benchmarks, suggesting that there are many opportunities in GPU applications for HAWS to improve GPU memory performance.

We then identify the impact of long latency RAW stalls for the set of application kernels described in Section 2. In Figure 4, we can see that the fraction of long latency RAW stalls in 10 out of 12 of our benchmarks is around, or more than, 20%. We classify these applications as *memory intensive benchmarks*. In four of the workloads, the percentage is over 30%. For a majority of our workloads, HAWS should be able to reduce memory latencies by using compiler-generated hints. We classify the other two workloads that have around 10% as *non-memory intensive benchmarks*, and include them in our study to show that we do not degrade performance for these workloads.

4 HINT ASSISTED EXECUTION

In this section, we show how we insert hints in GPU instructions. We then demonstrate the potential benefits of the presence of a hint on kernel execution performance using a detailed example.

First, we define an instruction that has dependency with a long latency memory operation as a *stall point*. In Figure 5, instructions 3 and 9 are stall points. We also define the instructions between any 2 stall points as a hint group. When a wavefront stalls due to encountering a long latency memory operation, the hint will assist the wavefront scheduler to fetch and issue independent instructions within a hint group, fetching as many as it can, until the long latency flag is released by the scoreboard.

In Figure 5, we show the execution timeline for a GPU, guided by a round-robin scheduler and HAWS. For simplicity, we assume arithmetic instructions take 1 cycle to execute; memory

Fig. 5. Execution timeline for round-robin scheduler and HAWS. The Top 2 are running with a 19 cycles memory latency. The bottom 2 are running with a 14 cycles memory latency.

instructions take 14 or 19 cycles in our example. There are four wavefronts executing in parallel, with only one of them being issued each cycle. Each wavefront will fetch a new instruction every cycle. We vary the latency of memory to better characterize dependencies in our experiments, which are presented next.

Experiment 1: Memory latency is 19 cycles: When the wavefront scheduler tries to issue instruction 3, from cycles 9 to 12, due to the dependency with instruction 1, the wavefront stalls in the GPU with a round-robin scheduler and the long latency operation flag in the scoreboard is set to 1. With HAWS, the wavefront will keep searching for independent instructions within the hint group. If one is found, then the hint will offer to the wavefront scheduler the offset of the next independent instruction. HAWS decides to issue instruction 4 in cycle 13 and keeps executing, informed by the instruction hint. Wavefronts that use round-robin scheduling still stall, due to long latency memory operations. After fetching instruction 4, HAWS reads the offset of next instruction that should be fetched from the hint, which is instruction 6. After instruction 6 is finished, since there are no independent instructions we can issue, HAWS will go back to the original stall point, fetches instruction 3, and waits for the memory instructions to finish. The memory instructions are finished, starting in cycle 20. Both HAWS and the Round-Robin schedulers receive results from r1, the scoreboard releases the long latency flag, so that instruction 3 is issued successfully for both schedulers. Considering the execution latencies, since HAWS better utilizes the computational resources of the GPU by carrying out selective out-of-order execution in the shadow of a memory result, this work reduces execution time by 25% speedup.

Experiment 2: Memory latency is 14 cycles: In this case, there are limited benefits for HAWS, since the memory latency is reduced by 5 cycles. Just as in experiment 1, HAWS begins selective out-of-order execution in cycle 13, issuing instruction 4. Starting in cycle 15, the memory results are available, so the scoreboard is updated to reset the long latency operation flag. HAWS receives the flag, and transitions the wavefront back to in-order execution by flushing the fetch buffer and refetching instruction 3. HAWS provides a dynamic decision. Whenever a long latency operation is resolved, HAWS returns to in-order execution. In other words, instructions in the wavefronts are executing speculatively in terms of hint-assisted mode. That is why we see instruction 3 issued in cycle 17 for HAWS, and instead of instruction 6. More architectural details will be provided in the next section. In this case, since the memory latency is reduced, HAWS leads to a performance increase of 12.5%. Instruction hints provide detailed information to the HAWS scheduler to support the next hint.

Fig. 6. Hint encoding format for HAWS.

Fig. 7. Percentage of register usage.

5 MICROARCHITECTURE OF HAWS

5.1 HINT Format

As discussed earlier in this article, HAWS is guided by compiler-generated hints to select the next instruction to fetch/execute. The two different formats for instruction hints are shown in Figure 6. The formats differ for different classes of instructions: format 2 is for branch instructions, while format 1 is for all the other types of instructions.

For format 1, the target offset field (9 bits) is used to help the wavefront scheduler find the next independent instruction to be fetched and issued. The target offset is the relative distance to the next out-of-order instruction, starting from the last stalled instruction (stall point), which triggered hint-assisted scheduling. The rename bit is used to communicate to the wavefront scheduler if this instruction needs to perform register renaming based on the destination register. If the rename bit is set, then the Dst field provides the renamed register number, otherwise, it is the original destination register number.

In HAWS execution, when instructions retire, we use register renaming to avoid WARs and WAWs when storing results, as necessary. Registers files in the GPU are underutilized in many applications, which has been discussed in many recent studies [7, 8, 16]. Based on these studies and our evaluation, we propose to use the underutilized registers for register renaming. We also analyzed a variety of benchmarks and none of them has a 100% register usage. Figure 7 shows that in the 12 benchmarks we evaluated, there is an average of 36.3% scalar register usage and 43.8% vector register usage. In other words, there is a lot of space for supporting register renaming. There are a couple of ways to implement register renaming efficiently. We can use the compiler to implement

register renaming and insert the renamed register number directly in the hint. Using this scheme, we can utilize the registers more efficiently by renaming selected registers with static analysis.

Please note, we only rename registers if necessary. For instructions that do not have a WAR dependency with any previous skipped instruction during HAWS execution, associated registers are not renamed. In other words, we only rename a register if their new value will affect the source register of any of the previous skipped instructions when the wavefront returns to normal execution. More details are provided in the following example.

In format 2, we need two fields to hold the target offset in the hint metadata, since there are two destinations for a conditional branch. The type field specifies the instruction type with 1 bit, 0 indicating a conditional branch, and 1 indicating a direct branch. In format 2, each offset field is 11 bits, two more bits than the offset assumed in format 1, since we are dealing with branch instructions. HAWS only adds hints to instructions that are independent of the long latency memory instruction within a hint group. For the instructions that have dependencies on long latency memory instructions, HAWS will not add any hints, reserving instruction hints for only a limited number of instructions.

5.2 Compiler Work

As mentioned previously, we use an in-house OpenCL compiler (Multi2C) for generating instruction hints to support HAWS execution. The compiler plays two roles in the implementation of HAWS. First, it analyzes data dependencies and decides which instructions can be scheduled and executed by HAWS, providing associated instruction hints. Second, it implements register renaming and insert operations directly into the hint, guiding HAWS to use registers efficiently.

Referring back to Figure 5, the compiler finds independent instructions I4 and I6 after analyzing data dependencies, adding corresponding hints to I3 and I4 to assist the wavefront scheduler to find the correct instructions for fetching and issuing. The compiler also inserts the destination register number in the instruction hint. More details will be provided next.

5.3 The Detailed HAWS Pipeline

When wavefronts stall due to a long latency memory operation, the long latency operation flag of the scoreboard is set, which triggers HAWS. The wavefront state flag bit is set 1 to indicate that the wavefront is in the hint-assisted execution state.

In the fetch stage, HAWS fetches the instructions specified by the hint, one by one, using the hint's guidance. HAWS will not attempt to access any instruction that are not pointed to by hints. After fetching and decoding the selected instructions, HAWS will send the source and destination register numbers to the scoreboard to see if there are any dependencies. Since HAWS will issue and execute new instructions, which will affect the state of original scoreboard, there are a few solutions to solve this issue. One is to use two scoreboards for tracking the pending register state to support using hints. Every time HAWS is triggered, the in-order scoreboard is copied to the HAWS scoreboard, since there may be some other instructions in flight. But this method increases hardware complexity by too much, so we adopt an alternative approach that dramatically decreases hardware costs and saves GPU energy. More details will be provided in the following paragraphs.

Branch instructions: A branch instruction is executed normally in HAWS if the branch was identified as independent of any earlier long latency instructions. To support branch execution in HAWS, we duplicate the SIMT stack, since we need to track program control flow and provide the active mask for each wavefront. We copy the data from the original SIMT stack every time hint mode is triggered.

Barrier instructions: In regular (i.e., non-HAWS) GPU execution, since we normally execute instructions in order, whenever a wavefront reaches a barrier, it will stop and wait for the other

Fig. 8. Organization of result collection unit.

wavefronts to reach the barrier. In HAWS execution, the hint in HAWS will not select any barrier instructions, but the instructions past a barrier can still be selected and executed if they are independent of the previous long latency instruction. This feature can provide us with some significant performance benefits.

Memory instructions: Memory instructions will not be identified by a hint if a data hazard could occur. An exception to this rule are memory instructions beyond barriers will be ignored by HAWS, even if they will not generate any data hazards. The reason for this decision is based on the GPU memory model. To ensure correct results when parallel work-items cooperate, all stores from work-items in other wavefronts in the same work-group are only visible to the current wavefront after a barrier.

After execution completes, the original destination register number, as well as the offset field and the renamed register number, are stored in the Result Collection Unit (RCU). This unit includes a result buffer with several entries. The unit has multiple roles in the design. First, the RCU stores the results of the out-of-order retired instructions, maintaining the mapping between the original register and the newly assigned register. The RCU uses the information provided by the hint to assist the wavefront schedulers, issuing instructions more efficiently and bypassing unnecessary instructions whenever the wavefront switches back to in-order execution. The detailed organization of the RCU is present in Figure 8. The RCU consists of a buffer with several entries, each entry has 6 different fields. Every instruction scheduled and executed by HAWS will have a entry in the result buffer.

Each entry in the result buffer has the following fields: (i) offset (9 bits), (ii) dst (8 bits), (iii) renamed (8 bits), (iv) ready (1 bit), (v) control (1 bit), and (vi) valid (1 bit). The six fields maintain information about the offset of the current instruction, the destination register number, the renamed destination register number, status indicating whether the result is ready to use, if this instruction is a branch, and whether the entry is valid or not. The total size of the result buffer entry is 28 bits. We set the depth of the RCU to 8, which was selected based on our experiments.

Second, as mentioned above, we do not duplicate the scoreboard for HAWS execution. To reduce the hardware cost, and also improve the energy efficiency during HAWS execution, we reuse the dst and renaming fields in the RCU to assist HAWS execution. Since the RCU stores the original destination register number of the instructions executed by HAWS, when a new instruction is issued in HAWS mode, the wavefront scheduler will check not only the scoreboard but also the previous entries in the RCU to be sure it does not have a RAW dependency (WARs and WAWs are handled using a hint and register renaming). Although the RCU holds the renamed destination register number for each instruction executed by HAWS, we also introduce a register renaming table, which maintains the mapping between the original register number and the most recently associated renamed register number. During HAWS execution, multiple instructions may have the same destination register, which could be renamed to different registers. In addition, the renaming table always records the latest mapping of a register and its renamed register, whenever a potential RAW hazard could occur. Thus, the following instruction will read the correct register value from the renaming table.

Fig. 9. Example of hint information and RCU operation for instructions in the first experiment of Figure 5. For clarity, all offset fields are shown as the instruction it points to.

When an instruction that was scheduled by HAWS is sent to a functional unit to execute, all fields for this entry in the RCU are updated, except for the ready field. We use the instructions in Figure 5 as an example. The hint information, as well as RCU operation for I4 and I6 are shown in Figure 9. Please note for clarity, we show the relationship between instructions for individual offset fields.

Before we analyze the execution, we first focus on the instruction hint. We can see the rename bit for both instructions is 0, which means we do not have to rename the destination registers. For I4, since it is the first instruction in the hint group, which can be potentially scheduled by the HAWS scheduler, there is no instruction before I4 that will use r4 as a source register, so we do not need to rename r4. For I6, the instruction before I6, which is skipped by HAWS scheduling is I5. Since I6 does not have a WAR dependency with I5, we do not have to rename r5 either.

In the first set experiments in Figure 5, when I4 is fetched by HAWS, the RCU receives the information and updates the result buffer with information for I4. In our example, I4 passes the check on to the scoreboard, and since there is no valid previous entries in the RCU, the HAWS scheduler issues I4. HAWS continues at I6. Before issuing I6, the HAWS scheduler will check both the scoreboard and the previous entries in the RCU. Since I4 is already complete at this time, the RAW dependency based on the source register r4 in I6 is released, and then I6 is issued.

When I4's execution is finished, the result is written to the destination register, and the ready bit is set to 1, indicating that this instruction is finished and its results are ready to use. If there is no register renaming, then the renamed field is the same as the dst field. If renaming is used, then the register renaming table in the hint mode will be updated.

Although HAWS does selective out-of-order execution, the instruction with the shortest offset will be chosen first by HAWS. So all of the entries in the RCU are ordered accordingly, in increasing distance values from the dependent instruction.

5.4 Returning to In-order Execution

Once the long latency operation completes, the long latency operation flag maintained in the scoreboard is cleared. When the wavefront scheduler receives this information, instruction execution switches back to in-order execution. The first instruction that is dependent on a long latency operation is allowed to issue, and the wavefront starts to execute instructions one-by-one in program

Fig. 10. Fetch logic.

order. At the same time, the wavefront scheduler also has to check if the instruction to be fetched has already been selected by HAWS. Using information provided by the RCU, these issues can be resolved intelligently, only fetching the necessary instructions.

Figure 10 shows the fetch logic when the wavefront is back to normal execution, which assists the wavefront to find the necessary instructions to fetch. In the first step, the fetch logic checks if the current pc is equal to the pc in the head entry of the RCU. If they are the same, then the fetch logic will keep checking whether the result of the instruction is written in the original register or renamed register by comparing the renaming field with dst field in the entry. If the result is written in the original register, then the scheduler can just skip this instruction and fetch the next instruction. If the result is not written to the original register, then the instruction will be fetched and issued. Since the result of the instruction is stored in the renamed register, we need to copy this value to the original register. After the instruction is issued, it will bypass the execution stage and update the register file. Please note that it is necessary to fetch and issue the instructions whenever register renaming occurs, since the scoreboard needs to be updated and perform dependency checking to prevent WAW and WAR hazards.

In our example in Figure 8, when the PC reaches I4, it checks with the head entry in the RCU to find which register where the result is written. Since there is no register renaming happening during I4's execution, the scheduler just skips I4 and fetches the next instruction. The same thing occurs when the PC reaches I6.

5.5 The Advantages of HAWS Over Static Scheduling

We can also use the compiler to reorder independent instructions to try to hide some of the long memory latency stall time. We provide the following example to compare HAWS scheduling with static compiler-based scheduling, demonstrating the advantages of HAWS.

In this example, we have a block of code that includes 3 main parts: blocks 1, 2, and 3. Each block represents 15t execution cycles as shown in Figure 11. There are also two components associated with each long memory stall, stall 1 and stall 2, which are 5t and 3t cycles, respectively. During blocks 2 and 3, we execute code c and f, respectively, which are instructions that are not dependent on any long latency memory operations present in the whole block.

For the normal execution in A, the program instruction blocks are issued and executed in the original order. Block 1 is executed first, after 5t cycles a long memory stall occurs, and then block 2

Fig. 11. Comparison of normal execution, statically scheduled execution, and HAWS execution.

is executed. Block 3 is executed 3t cycles after block 2 completes, due to another long latency stall. The total execution time is 53t cycles. Since we can execute some independent instructions during both memory stalls, the compiler performed a reordering of instructions in those portions of c and f.

In example B, the compiler schedules all independent instructions in sections c and f after the program encounters the first memory stall. It takes 8t cycles in total to finish execution of c and f. Although the code block 2 can be scheduled after 5t cycles, it has to wait until section f is finished. But when encountering the second stall, since there are no more independent instructions, the program will just wait there for 3t cycles. In this case, the total execution time is reduced to 48t cycles, since the static scheduling performed by the compiler fully covers the first memory stall by identifying independent instructions. But the compiler can not predict how long the memory stall will take, so it over-scheduled independent instructions after the first memory stall, which leads to sub-optimal performance.

In example C, the compiler breaks code section c into subsections c1 and c2, first executing c1 (which take 3t cycles) when the program encounters the first memory stall. It then executes c2 and f (which take 5t cycles) when the program encounters the second memory stall. Since we execute some instructions to overlap with the memory stall time, the total number of execution cycles is reduced to 47t cycles. Again, similar to example B, the compiler can not know how long it will take for each long latency memory stall to be resolved. It may not schedule enough independent instructions to execute to hide the first stall.

In example D, we use HAWS scheduling. When the program encounters the first stall at the end of block 1, HAWS will try to identify as many independent instructions as possible to be included in the hint group. This will allow the hardware to execute all of the instructions in part c, completely covering memory stall 1. When the program hits memory stall 2, HAWS executes all instructions in part f, which overlap nicely with the duration of stall 2. The total execution time of HAWS execution is 45t cycles, which is 2t cycles faster than using static scheduling in example C and 3t cycles faster than example B.

Comparing examples B, C, and D, when the program encounters the first stall, the number of independent instructions scheduled by the compiler are either too few to completely hide the delays associated with the long latency stall, or too many, impeding independent instructions from covering other stalls. In contrast, HAWS will try to execute as many instructions as possible, which will cover all of the stall time in the long latency stall example. This is why HAWS scheduling does better than static scheduling in this example.

Since there are many factors that may influence the duration of long latency memory stalls during execution, even for the same kernel, the stall time for the long latency memory operation could be dependent on the input data size used for the kernel. It is challenging to use static scheduling to cover all memory stalls. Since HAWS makes dynamic decisions in concert with the wavefront scheduler, guided by a compiler-generated instruction hints, it tries to execute as many instructions as possible. HAWS will provide us with better performance in most cases.

5.6 Area Estimation

As described earlier, we will have to add extra hardware to support HAWS execution. The Result Collection Unit (RCU) must be added, which includes eight entries, with 28 bits per entry. Our baseline GPU can support up to 40 wavefronts per compute unit (CU), so the total size of the RCU is 8,960 bits. We need to copy the SIMT stack when there is a branch instruction scheduled by HAWS. Each entry in the SIMT stack contains the active mask (64 bits), the current pc, and the reconvergence pc, with 32 bits for each field. The SIMT stack is 128 bits per entry, with eight entries. So the total size for the SIMT stack is 40,960 bits per CU. The renaming table consists of two parts, the scalar register renaming table and the vector register renaming table. Since there are 8 entries in the RCU, supporting up to eight instructions executed every time when the wavefront switches to HAWS mode, we also set the renaming tables to eight entries. For scalar registers, each entry has 16 bits, 8 bits for the original register number and 8 bits for the renaming register number. We assume the same design for the vector register renaming table, with 16 bits per entry. The total size for the register renaming table is 10,240 bits. The overall overhead is 7.34KBytes/CU.

Based on CACTI modeling [20], the additional elements associated with HAWS takes 0.045mm^2 per CU. Compared with the baseline GPU model, which is based on the AMD Radeon 7970 [15], which contains 32 CUs and a die size of 352mm^2 , the total area overhead is only 0.4% of the total chip size.

6 EVALUATION

6.1 Methodology

To evaluate our design, we select representative benchmarks from the AMD APP SDK 2.5 benchmark suite. The benchmarks provide us with a rich set of both memory-intensive and compute-intensive benchmarks, as well as a mix of both, representing a wide range of application behaviors. The 12 benchmarks evaluated in this work are listed in Table 1, along with some of their application properties. Again HAWS scheduling is targeting memory intensive benchmarks, so most a majority of the benchmarks on our list are memory intensive, though we include compute-intensive to make sure we do not impact their performance.

We model our baseline GPU architecture using the Mult2Sim simulator [5, 30], which is a cycle-level heterogeneous simulation framework supporting both GPU and CPU simulation. All experiments are based on a baseline GPU model similar to the AMD Radeon 7970, whose hardware specifications are summarized in Table 2. We extended the timing part of the AMD GPU model in Multi2Sim to support HAWS. The Radeon 7970 is a high-end AMD GPU that supports instruction-level parallelism, allowing multiple instructions to be issued in the same cycle.

Table 1. List of Benchmark Applications

Applications	Abbr.	Type
Binary Search	BS	M
Bitonic Sort	BI	N
Discrete Cosine Transform	DCT	M
DwtHaar1D	DH	M
FFT	FFT	N
FastWarshTransform	FW	M
MatrixMultiplication	MM	M
MatrixTransform	MT	M
PrefixSum	PS	M
Reduction	RD	M
ScanLargeArrays	SA	M
SimpleConvolution	SC	M

Table 2. Baseline GPU Model Configurations

GPU Model	AMD Radeon 7970
Clock Frequency	1,000MHZ
Number of CUs	32
Max Num. Wavefront/CU	40
Scheduler Policy	GTO
SIMT Units/CU	4
SIMT lane width	16
L1 Cache/CU	16KB, 64 Sets, 4 way, 64B/line
L2 Cache	128KB, 128 Sets, 16 way, 64B/line
Scalar Registers/CU	2,048
Vector Register/CU	65,536
LDS/CU	64KB

6.2 Performance

We compare HAWS scheduling against a Greedy-than-oldest (GTO, our baseline) [26], CTA-aware scheduling [9], and warped-preexecution scheduling techniques [12]. Figure 12 shows the overall performance achieved by HAWS and the competing techniques for both memory intensive and non-memory intensive applications, as compared to the baseline GPU model. We separate memory intensive benchmarks and those are not in the chart.

CTA-aware tries to reduce cache contention and utilize data locality by prioritizing sub-groups of wavefronts to execute and access memory, giving them a better chance to better utilize the data and maximize reuse opportunities. We observe a speedup in most of benchmarks for CTA-aware over the baseline model. CTA-aware not only exploits intra-wavefront locality but also exploits inter-wavefront locality, while GTO only gives priority to the oldest wavefronts. However, due to the improvement of our baseline GPU model, CTA-aware is not as effective as reported in the older architecture. It achieves 4% speedup in overall.

Note that the scheduler for both the baseline GPU model and the regular execution in HAWS use a GTO scheme. In all of the benchmarks, HAWS achieves a maximum performance improvement of 34.2%, as found in the BS benchmark. From the results, we can also see HAWS does not negatively

Fig. 12. Speedup of HAWS, relative to the baseline model.

impact non-memory intensive applications, providing an average of 2.3% improvement for FFT and BI, since we benefit from the dynamics of the wavefront scheduler, based on the current instruction hint to increase instruction-level parallelism of the GPU.

Pre-execution scheduling tries to issue future independent instructions whenever a wavefront stalls by relying on the hardware to dynamically detect such opportunities. For most of the workloads we have studied, we observe that HAWS has a better speedup than pre-execution scheduling. The reason for this is HAWS can locate the next independent instruction directly and execute it according to the instruction hints, while pre-execution scheduling needs to use hardware to dynamically detect if the next instruction is independent with the long latency operation or not. Sometimes it takes longer to find an appropriate candidate to fetch and execute. While pre-execution scheduling provides a 12.4% average speedup for memory-intensive workloads, HAWS offers a higher speedup of 14.6%.

The speedup of HAWS scheduling when running memory-intensive workloads is higher than when non-memory intensive benchmarks, given that memory-intensive applications have more long latency memory stalls, which provides the wavefront schedulers with more opportunities to trigger HAWS scheduling.

6.3 Impact on Memory Stalls

Figure 13 shows the breakdown of scheduling cycles for all of the benchmarks. The portion of the stacked bars identified as *Issued* denotes that at least one wavefront in the GPU is eligible to be issued. The *Long Latency Stall* portion is defined as all wavefronts in GPU that are stalled due to a long latency memory operation. Since multiple wavefronts are allowed to be issued by the same wavefront scheduler in the same cycle in AMD GPUs, we label the rest of the stalls as *Mixed Stall*. This denotes that no wavefront can be issued in that cycle, which may be due to a variety of reasons (e.g., there is no available instruction is in fetch buffer, the pipeline is busy, a data dependencies occurred, etc.). For the same benchmark, both the baseline and HAWS execution have a similar total number of *Issued* cycles, which makes perfect sense since they are executed while considering the same resource. In type M benchmarks, the *Long Latency Stall* cycles are greatly reduced by the HAWS scheduler, as compared to the baseline model. We take a closer look at this in Figure 14.

Fig. 13. Breakdown of scheduling cycles (B: Baseline, H: HAWS).

Fig. 14. Percentage of long latency stall, comparing HAWS with baseline model.

Figure 14 shows the percentage of long latency RAW stalls in the benchmarks when using HAWS. For comparison, we also showed earlier the long latency stalls percentages for the baseline model in Figure 4. For those applications that enjoy significant speedup using HAWS, including BS, MT and FW, HAWS can greatly reduce stalls due to long latency RAWs in those benchmarks up to 14.9%, which we can also see there is a better speedup for these benchmarks in Figure 12. For non-memory intensive benchmarks, the long latency stalls can only be reduced marginally by HAWS, as discussed previously, since the total percentage of long latency stalls is not as high.

6.4 Opportunities for Instruction Level Parallelism

Figure 15 shows the average number of SIMT instructions executed by a wavefront every time when using HAWS scheduling. As mentioned previously, HAWS will only select instructions

Fig. 15. Average number of SIMT instructions executed per wavefront for per HAWS scheduling.

Fig. 16. Percentage of total ALU instructions executed by HAWS.

within a hint group to execute. From our study, across all of the benchmarks, 0.55 instructions (less than 1) are executed on average by a wavefront every time when we switch to hint-assisted mode. The memory intensive workloads play a large role in dominating this average. For example, in the DCT workload we see 1.23 instructions executed on average per switch to HAWS.

Figures 16 and 17 show a breakdown of the instruction types executed by HAWS. As shown in Figure 16, MM executes almost 20% of its total ALU instructions in hint mode, while FFT only executes 0.39% of its total ALU instructions. BS executes almost 20% of its total long latency memory instructions in hint mode, as shown in Figure 17. Combining these two figures together, we can conclude that for a workload to benefit from HAWS, it will need to have a high percentage of ALU instructions. In applications such as MM that contain a limited number of memory instructions executed in HAWS mode, if they have a high rate of ALU instructions executed, then they can still achieve good performance with HAWS. Inspecting the code closer, both MM and DCT have long

Fig. 17. Percentage of total Memory instructions executed by HAWS.

latency memory instructions inside a loop. These instructions are used for loading operands that are the inputs for vector multiplication—these are operations that can trigger HAWS execution. Every time the program stalls due to a long latency memory instruction, HAWS schedules ALU instructions for execution. Since HAWS will be triggered repeatedly, this increases the percentage of ALU instructions executed by HAWS.

7 RELATED WORK

Using Hints in Microprocessors: Instruction hints have been present in modern microprocessors in previous designs. They have been used primarily for resolving branch mispredictions and cache misses in CPUs. The Intel Itanium 2 [18] ISA defined hint instructions that are used to provide the hardware early information about a future branch, and also direct the instruction prefetch engine to prefetch one or many L2 cache lines to increase cache hit rates. Another popular example is the Synergistic Processing Unit (SPU) in the IBM Cell processor [10]. Jointly developed by Sony, Toshiba, and IBM, architects decided to remove the hardware branch predictor and use software branch hinting to recover lost performance. Compared to the Intel Itanium, the Cell SPUs do not have any hardware branch predictor and rely solely on software branch hints to make the design more power efficient. In the EPIC (Explicitly Parallel Instruction Computing) [27] ISA family, introduced by Hewlett Packard and Intel, the compiler performs direct cache placement and actively manages replacement policies through cache hints. Wang et al. [32] developed an analytical model that predicts which data will be reused by using compiler hints. These hints are provided to improve replacement decisions in set-associative caches. Their work explored options for reducing cache misses by adopting compiler hints to improve replacement decisions. Beyls et al. [1] proposed to generate cache hints from the reuse distance metric, based on two approaches. The first approach uses profiling to statically assign a cache hint to a memory instruction. The second approach is based on an analytical model used to select the most appropriate hint dynamically.

Warp Scheduler: Previous studies have shown that the wavefront scheduler plays a vital role in improving GPU performance. Narasiman et al. [21] introduce a two-level wavefront scheduler, which separates wavefronts into different groups, preventing them from reaching the same long latency instruction at the same time. This design allows the compute unit to hide long latency operations by switching between different groups, while ensuring memory locality within the

same group. Meng et al. [19] propose Dynamic Warp Subdivision (DWS), which splits wavefronts into two wavefront in the presence of a branch divergence or memory divergence. This method allows a compute unit to interleave the computation down different branch paths to hide memory latency. In addition, DWS allows the threads that hit in cache to continue to execute aggressively, even if some of their peer wavefronts encountered a miss (i.e., a memory divergence). Rogers et al. [25] propose Variable Warp Sizing (VWS) scheduling, using a smaller warp size to execute when control flow and memory divergence occurs, improving the performance of divergent applications. Gebhart et al. [3] introduce the use of two-level scheduling to improve GPU energy efficiency, while maintaining performance, on massively threaded GPU designs.

Exploiting Thread Level Parallelism (TLP) on GPUs: Many wavefront scheduler studies have focused on choosing the right amount of TLP for the memory sub-system to avoid over-saturation, and to reduce latency. The Cache-Conscious Wavefront Scheduler (CCWS) [26] preserves intra-wavefront locality by using new hardware to adjust the amount of Thread Level Parallelism (TLP), limiting L1 data cache thrashing and preserving intra-wavefront locality. Jog et al. [9] presented OWL, a CTA-aware scheduler that uses data locality information to limit the number of CTAs in each SM, which targets to reduce cache contention. OWL achieves these benefits by improving both L1 cache hit rates and latency tolerance, improving DRAM bank parallelism, and improving both DRAM row locality and cache hit rates. In addition, Kayiran et al. [11] proposed a dynamic CTA scheduling technique that attempts to allocate optimal number of CTAs per core based on application demands, demonstrating that executing the maximum number of CTAs per core is not always the best solution to boost performance due to high cache and memory contention. Yu et al. [33] presented a Stall-Aware Warp Scheduling (SAWS) policy, which dynamically optimizes the TLP according to pipeline stalls. SAWS can effectively improve pipeline efficiency by reducing structural hazards without introducing new data hazards. Wang et al. [31] introduce a Occlusion Aware Warp Scheduler (OAWS) that focus on TLP on memory resources. Their scheduler monitors and predicts memory resource usage, scheduling wavefronts that can be satisfied by the available memory resource. Rhu et al. [24] introduced the Dual-Path execution model (DPE), which exploits intra-warp parallelism by interleaving execution of different paths when divergence occurs in a warp. Unlike prior approaches to this issue, DPE does not require an extensive redesign of the microarchitectural components, and instead extends the stack to support two concurrent execution paths.

Improving Instruction Level Parallelism (ILP) on GPUs: Instead of reducing latency, several techniques focus on improving the overlap of compute and memory operations to exploit ILP on GPUs. Mascar [28] introduces a bimodal warp scheduling scheme, along with a cache access re-execution system to increase overlap. Very Long Instruction Word (VLIW) instruction set architectures are designed to exploit ILP by encoding multiple independent operations in a single VLIW instruction. VLIW was adopted for AMD's Evergreen GPU architecture. However, VLIW suffers from limited parallelism opportunities, as it solely relies on static analysis. Gong et al. [4] proposed TwinKernels, which takes advantages of the different instruction scheduling algorithms in the compiler to improve overlap of compute and memory operations. Kim et al. [12] proposed a warped-preexecution approach on GPUs. In this technique, wavefronts try to issue future instructions that are independent of the stalling instructions. Their approach relies on hardware to dynamically detect such opportunities, and may take several attempts before finding a proper candidate. By leveraging static analysis at compile time, our approach is free from hardware dependency detection. In addition, guided by the information provided in hints, our approach can commit the correct program state more efficiently when wavefronts return to normal execution.

8 CONCLUSIONS AND FUTURE WORK

In this article, we have presented a Hint Assisted Wavefront Scheduler, a novel GPU wavefront scheduler that uses compiler generated hints to better utilize GPU hardware resources. Our hints are used to identify opportunities for out-of-order execution in the shadow of wavefront stalls caused by long latency memory operations. Compared to static approaches, HAWS provides us with a dynamic solution that can take full advantage of unused cycles due to memory stalls. We presented the design and evaluated potential performance benefits across a variety of benchmarks. Our results show HAWS can improve performance, on average, by 15.3% for memory intensive applications. For non-memory intensive workloads, HAWS is 2.3% faster than the baseline model.

We plan to explore hint usage in our future work. In the current HAWS model, the instructions identified to HAWS use a hint, and are handled one at a time. We believe further opportunities can be explored by using hints to identify multiple instructions to the wavefront scheduler, allowing the wavefront scheduler to decide which ones to issue according to the current hardware resource availability.

REFERENCES

- [1] Kristof Beyls and Erik D'Hollander. 2002. Compile-time cache hint generation for EPIC architectures. In *Proceedings of the 2nd Workshop on Explicitly Parallel Instruction Computing Architecture and Compilers (EPIC'02)*.
- [2] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. 2014. cuDNN: Efficient primitives for deep learning. *Arxiv Preprint Arxiv:1410.0759* (2014).
- [3] Mark Gebhart, Daniel R. Johnson, David Tarjan, Stephen W. Keckler, William J. Dally, Erik Lindholm, and Kevin Skadron. 2011. Energy-efficient mechanisms for managing thread context in throughput processors. In *ACM SIGARCH Computer Architecture News*, Vol. 39. ACM, 235–246.
- [4] Xiang Gong, Zhongliang Chen, Amir Kavvan Ziabari, Rafael Ubal, and David Kaeli. 2017. TwinKernels: An execution model to improve GPU hardware scheduling at compile time. In *Proceedings of the International Symposium on Code Generation and Optimization*. IEEE Press, 39–49.
- [5] Xun Gong, Rafael Ubal, and David Kaeli. 2017. Multi2Sim kepler: A detailed architectural GPU simulator. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS'17)*. IEEE, 269–278.
- [6] Honghoon Jang, Anjin Park, and Keechul Jung. 2008. Neural network implementation using CUDA and OpenMP. In *Proceedings of the Conference on Computing: Techniques and Applications (DICTA'08)*. IEEE, 155–161.
- [7] Hyeran Jeon, Gokul Subramanian Ravi, Nam Sung Kim, and Murali Annavaram. 2015. GPU register file virtualization. In *Proceedings of the 48th International Symposium on Microarchitecture*. ACM, 420–432.
- [8] Naifeng Jing, Yao Shen, Yao Lu, Shrikanth Ganapathy, Zhigang Mao, Minyi Guo, Ramon Canal, and Xiaoyao Liang. 2013. An energy-efficient and scalable eDRAM-based register file architecture for GPGPU. In *ACM SIGARCH Computer Architecture News*, Vol. 41. ACM, 344–355.
- [9] Adwait Jog, Onur Kayiran, Nachiappan Chidambaram Nachiappan, Asit K. Mishra, Mahmut T. Kandemir, Onur Mutlu, Ravishankar Iyer, and Chita R. Das. 2013. OWL: Cooperative thread array aware scheduling techniques for improving GPGPU performance. In *ACM SIGPLAN Notices*, Vol. 48. ACM, 395–406.
- [10] James A. Kahle, Michael N. Day, H. Peter Hofstee, Charles R. Johns, Theodore R. Maeurer, and David Shippy. 2005. Introduction to the cell multiprocessor. *IBM J. Res. Dev.* 49, 4.5 (2005), 589–604.
- [11] Onur Kayiran, Adwait Jog, Mahmut Taylan Kandemir, and Chita Ranjan Das. 2013. Neither more nor less: Optimizing thread-level parallelism for GPGPUs. In *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques*. IEEE Press, 157–166.
- [12] Keunsoo Kim, Sangpil Lee, Myung Kuk Yoon, Gunjae Koo, Won Woo Ro, and Murali Annavaram. 2016. Warped-preexecution: A GPU pre-execution approach for improving latency hiding. In *Proceedings of the IEEE International Symposium on High Performance Computer Architecture (HPCA'16)*. IEEE Press, 163–175.
- [13] George Kyriazis. 2012. Heterogeneous system architecture: A technical review. *AMD Fusion Dev. Summit* (2012), 21.
- [14] Erik Lindholm, John Nickolls, Stuart Oberman, and John Montrym. 2008. NVIDIA tesla: A unified graphics and computing architecture. *IEEE Micro* 28, 2 (2008).
- [15] Mike Mantor. 2012. AMD Radeon HD 7970 with graphics core next (GCN) architecture. In *Proceedings of the 24th Hot Chips Symposium (HCS'12)*. IEEE, 1–35.
- [16] Mengjie Mao, Wujie Wen, Yaojun Zhang, Yiran Chen, and Hai Li. 2017. An energy-efficient GPGPU register file architecture using racetrack memory. *IEEE Trans. Comput.* (2017).

- [17] Sally A. McKee. 2004. Reflections on the memory wall. In *Proceedings of the 1st Conference on Computing Frontiers*. ACM, 162.
- [18] Cameron McNairy and Don Soltis. 2003. Itanium 2 processor microarchitecture. *IEEE Micro* 23, 2 (2003), 44–55.
- [19] Jiayuan Meng, David Tarjan, and Kevin Skadron. 2010. Dynamic warp subdivision for integrated branch and memory divergence tolerance. *ACM SIGARCH Comput. Architect. News* 38, 3 (2010), 235–246.
- [20] Naveen Muralimanohar, Rajeev Balasubramonian, and Norman P. Jouppi. 2009. CACTI 6.0: A tool to model large caches. *HP Lab.* (2009), 22–31.
- [21] Veynu Narasiman, Michael Shebanow, Chang Joo Lee, Rustam Miftakhutdinov, Onur Mutlu, and Yale N. Patt. 2011. Improving GPU performance via large warps and two-level warp scheduling. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, 308–317.
- [22] CUDA Nvidia. 2007. Compute unified device architecture programming guide.
- [23] In Kyu Park, Nitin Singhal, Man Hee Lee, Sungdae Cho, and Chris Kim. 2011. Design and performance evaluation of image processing algorithms on GPUs. *IEEE Trans. Parallel Distrib. Syst.* 22, 1 (2011), 91–104.
- [24] Minsoo Rhu and Mattan Erez. 2013. The dual-path execution model for efficient GPU control flow. In *Proceedings of the IEEE 19th International Symposium on High Performance Computer Architecture (HPCA'13)*. IEEE, 591–602.
- [25] Timothy G. Rogers, Daniel R. Johnson, Mike O'Connor, and Stephen W. Keckler. 2015. A variable warp size architecture. In *ACM SIGARCH Comput. Architect. News*, Vol. 43. ACM, 489–501.
- [26] Timothy G. Rogers, Mike O'Connor, and Tor M. Aamodt. 2012. Cache-conscious wavefront scheduling. In *Proceedings of the 45th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 72–83.
- [27] Michael S. Schlansker and B. Ramakrishna Rau. 2000. EPIC: Explicitly parallel instruction computing. *Computer* 33, 2 (2000), 37–45.
- [28] Ankit Sethia, D. Anoushe Jamshidi, and Scott Mahlke. 2015. Mascar: Speeding up GPU warps by reducing memory pit-stops. In *Proceedings of the IEEE 21st International Symposium on High Performance Computer Architecture (HPCA'15)*. IEEE, 174–185.
- [29] John E. Stone, David Gohara, and Guochun Shi. 2010. OpenCL: A parallel programming standard for heterogeneous computing systems. *Comput. Sci. Eng.* 12, 3 (2010), 66–73.
- [30] Rafael Ubal, Byunghyun Jang, Perhaad Mistry, Dana Schaa, and David Kaeli. 2012. Multi2Sim: A simulation framework for CPU-GPU computing. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques (PACT'12)*. IEEE, 335–344.
- [31] Bin Wang, Yue Zhu, and Weikuan Yu. 2016. OAWS: Memory occlusion aware warp scheduling. In *Proceedings of the International Conference on Parallel Architecture and Compilation Techniques (PACT'16)*. IEEE, 45–55.
- [32] Zhenlin Wang, Kathryn S. McKinley, Arnold L. Rosenberg, and Charles C. Weems. 2002. Using the compiler to improve cache replacement decisions. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*. IEEE, 199–208.
- [33] Yulong Yu, Weijun Xiao, Xubin He, He Guo, Yuxin Wang, and Xin Chen. 2015. A stall-aware warp scheduling for dynamically optimizing thread-level parallelism in GPGPUs. In *Proceedings of the 29th ACM on International Conference on Supercomputing*. ACM, 15–24.

Received July 2018; revised February 2019; accepted February 2019