2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA) | 979-8-3503-2658-1/24/$31.00 ©2024 IEEE | DOI: 10.1109/ISCA59077.2024.00011

GhOST: a GPU Out-of-Order Scheduling Technique for Stall Reduction

Ishita Chaturvedi[1], Bhargav Reddy Godala[1], Yucan Wu[1], Ziyang Xu[1],
Konstantinos Iliakis[2], Panagiotis-Eleftherios Eleftherakis[2], Sotirios Xydis[2], Dimitrios Soudris[2],
Tyler Sorensen[3], Simone Campanoni[4], Tor M. Aamodt[5], David I. August[1]

[1]Princeton University, USA, [2]National Technical University of Athens, Greece, [3]UC Santa Cruz, USA,
[4]Northwestern University, USA, [5]University of British Columbia, Canada

***Abstract*—Graphics Processing Units (GPUs) use massive multi-threading coupled with static scheduling to hide instruction latencies. Despite this, memory instructions pose a challenge as their latencies vary throughout the application's execution, leading to stalls. Out-of-order (OoO) execution has been shown to effectively mitigate these types of stalls. However, prior OoO proposals involve costly techniques such as reordering loads and stores, register renaming, or two-phase execution, amplifying implementation overhead and consequently creating a substantial barrier to adoption in GPUs. This paper introduces GhOST, a minimal yet effective OoO technique for GPUs. Without expensive components, GhOST can manifest a substantial portion of the instruction reorderings found in an idealized OoO GPU. GhOST leverages the decode stage's existing pool of decoded instructions and the existing issue stage's information about instructions in the pipeline to select instructions for OoO execution with little additional hardware. A comprehensive evaluation of GhOST and the prior state-of-the-art OoO technique across a range of diverse GPU benchmarks yields two surprising insights: (1) Prior works utilized Nvidia's intermediate representation PTX for evaluation; however, the optimized static instruction scheduling of the final binary form negates many purported improvements from OoO execution; and (2) The prior state-of-the-art OoO technique results in an average slowdown across this set of benchmarks. In contrast, GhOST achieves a 36% maximum and 6.9% geometric mean speedup on GPU binaries with only a 0.007% area increase, surpassing previous techniques without slowing down any of the measured benchmarks.**

***Index Terms*—GPU, Parallelism, out-of-order execution, GPU Microarchitecture, low overhead out-of-order execution**

I. Introduction

Graphics Processing Units (GPUs) have significantly enhanced the efficiency of computationally intensive applications, including medical imaging [55], machine learning [30], and computer vision [48], achieving orders of magnitude improvement. This efficiency is attributed to the GPUs' capability to (1) support a massively parallel programming model and (2) reduce latency through concurrency, allowing rapid switching between concurrent warps. Some Nvidia devices, for instance, can handle up to 64 warps per core [41], [61], [62].

Despite these capabilities, many GPU applications still encounter significant stall cycles [2], [31], during which none of the warps can issue instructions due to data, control, or structural hazards. Many techniques to address GPU stall cycles focus on thread-level parallelism. Previous techniques have typically focused on two approaches: The first involves increasing the number of warps to improve the probability of finding a non-stalled warp. However, prior work has shown that doing so may result in worse performance due to increased resource contention [25], [34], [53]. The second approach has been to re-prioritize warps to enhance TLP and/or cache locality [12], [23], [33], [40], [53], [63]. However, all techniques focusing on TLP are limited, in part, because they only consider the next in-order instruction per warp. Consequently, the GPU core will stall when the oldest instructions for all warps encounter stalls.

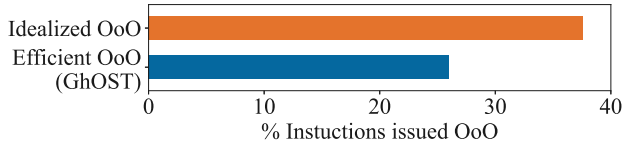

Fig. 1. Percentage of reordered instructions with idealized OoO execution (perfect branch prediction, register renaming and load-store queues) and efficient OoO execution which elides these techniques.

Recent efforts have employed out-of-order (OoO) execution, a technique successfully applied to CPUs, to enhance instruction-level parallelism on GPUs [16], [19], [20], [28], [66]. While OoO GPU approaches show promising initial results, they still suffer from inefficiencies, including re-issuing instructions, saving state resources such as registers or memory for more complicated warp switches, and making ISA modifications. Like GhOST, SOCGPU [19] is an inexpensive OoO mechanism for GPUs. Developed coincidentally, it further validates this paper's goals. While SOCGPU is implemented mostly in the front-end like GhOST, it has the constraint that an instruction cannot be removed from the instruction buffer until it is written back. This has the potential to fill the instruction buffer and create a type of stall not found in GhOST. This could explain why SOCGPU experiences slowdowns while GhOST does not. Thus, another important contribution of this paper is in demonstrating that a write-back constraint is unnecessary and should be avoided. LOOG [20] is the prior state-of-the-art that enables complete OoO execution for GPUs. However, LOOG requires expensive features like load-store queues for reordering memory instructions, result broadcasting, and register renaming.

979-8-3503-2658-1/24/$31.00 ©2024 IEEE
DOI 10.1109/ISCA59077.2024.00011

While previous approaches aim to enable CPU-style out-of-order (OoO) mechanisms on GPUs, GPU applications exhibit markedly different characteristics (see Section II). Historically, the first dynamically scheduled processor, the CDC6600 [60], enabled OoO execution without renaming or speculation back when resources were far more constrained than today. We see merit in revisiting such simpler, yet not fully-featured, designs in the context of today's throughput architectures where efficiency is paramount.

We observed that having instructions from independent warps in the instruction buffer (IB) provides a high level of flexibility, leading to significant out-of-order (OoO) performance without the need for register renaming, bookkeeping, or speculation. Figure 1 demonstrates that GPUs offer significant reordering opportunities to hide stalled instructions without requiring speculation, register renaming, and reordering of load and store instructions. These results were obtained across a comprehensive benchmark suite (see Section V) and measuring how many instructions could be reordered in an fully featured OoO configuration as well as our simpler, and more efficient, implementation (GhOST).

Building on this insight, this paper introduces GhOST: a low-overhead, hardware-only OoO technique for GPUs designed to leverage the unique characteristics of GPU architecture to achieve minimal hardware overhead. GhOST is the first work that utilizes the issue buffer as a large pool of instructions, thereby increasing the probability for GhOST to find a ready instruction without the need for the expensive techniques required by prior work. GhOST splits the instruction scheduler between the GhOST scheduler and the warp scheduler, leaving the complexity of the warp scheduler unmodified. This results in a simple and effective non-speculative OoO engine, which resides almost entirely in the decode stage.

GhOST outperforms LOOG [20], the prior state-of-the-art OoO technique while using significantly less area and power. GhOST achieves a maximum speedup of up to 37%, with a geometric mean speedup of 6.9% across 28 applications, demonstrating notable performance improvement in low-occupancy applications. GhOST does not cause performance degradation in any evaluated workload. Furthermore, GhOST has an extremely efficient implementation, requiring only an area increase of 0.007% and a 1.1707 mW increase in power. While GhOST is a hardware-only technique, we show that compiler techniques like loop unrolling and software renaming work well with GhOST, bringing the OoO performance closer to idealized OoO execution.

Concretely, our contributions are:

1) A comprehensive study that measures the performance impact of various out-of-order (OoO) optimizations on GPU workloads, emphasizing the minimal returns provided by traditional OoO components (Section VI-F).
2) Through GhOST, we demonstrate that an efficient and performant OoO implementation can be predominantly contained within the decode stage while providing ample scheduling freedom with minimal changes to the interface with other stages (Section IV).

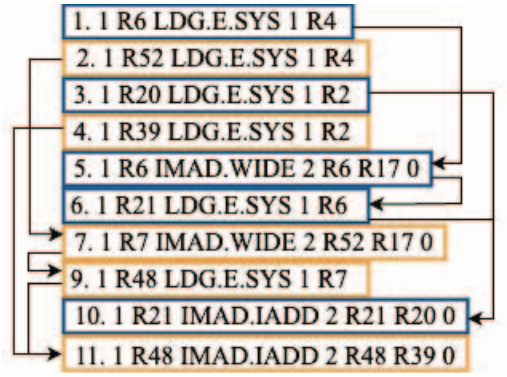

Fig. 2. SASS example of a typical basic block that appears in GPU programs. The two colors indicate the two independent dependence chains.

3) An extensive experimental study of GhOST compared to typical state-of-art GPUs and LOOG, the best performing prior GPU OoO mechanism. The study shows that GhOST outperforms LOOG while using a fraction of its area and power overhead (Section VI).

The paper is organized as follows: Section II motivates OoO execution on GPUs and advocates for a new OoO design. Section III describes the baseline GPU execution model and micro-architecture. Section IV provides a detailed overview of GhOST's design. Section V outlines the experimental setup, and Section VI presents the evaluation results, including a comparison with LOOG. Section VII covers related work, and Section VIII concludes the paper.

II. Motivation

GPU's performance comes from massively parallel programming and using TLP to hide latencies. However, some modern GPU workloads lack sufficient TLP to fully utilize GPU resources. Nvidia's introduction of Multi-Process Service (MPS) [42] aims to enhance TLP through the concurrent execution of multiple processes on the GPU. However, it falls short in improving the performance of individual applications, underscoring the importance of ILP enhancements. Despite the significant hardware support for TLP in GPUs, there remains untapped potential for exploiting instruction-level parallelism (ILP) in these devices. Previous studies have explored techniques such as out-of-order execution [16], [20], [28], [66] and thread coarsening [4], [38], [56] to enhance ILP, with the goal of improving stall hiding.

To illustrate opportunities for Instruction-Level Parallelism (ILP), consider the SASS code snippet from the single-source shortest path (SSSP) benchmark in the Pannotia suite [6], as shown in Figure 2; some details have been omitted for clarity. This code pattern, found in GPU workloads, involves multiple load instructions from different load-use chains. The compiler interleaves these chains to increase the distance between loads and their use. This block repeats several times in the critical loop of the SSSP benchmark. Figure 3 displays the histogram of latencies for the load instructions shown in Figure 2. The

TABLE I
RELATED WORK COMPARISON

Technique	No ISA Change	No in-order to OoO mode switching	Low Hardware Overhead	Speedup evaluated with SASS
Perfect OoO				22% with perfect branch prediction, unbounded register renaming, and perfect memory alias checking
Warped Pre-Execution [28]	✓	✗	✗	No SASS reported
MIPSGPU [66]	✓	✗	✗	No SASS reported
LOOG [20]	✓	✓	✗	16.5% slowdown (SASS §VI-G), LOOG shows significant slowdowns after optimized compilation with SASS. 16% speedup on PTX.
HAWS [16]	✗	✗	✓	No SASS reported
GhOST [This Work]	✓	✓	✓	6.9%

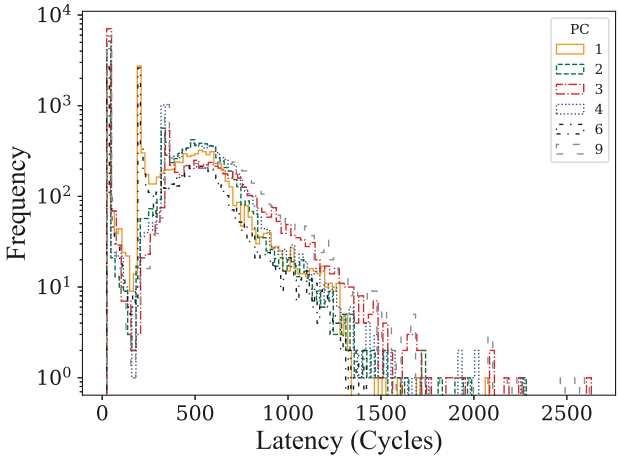

Fig. 3. Histogram showing the variable load latency during program execution from Figure 2.

histogram was plotted by collecting load latencies for every instance of the load instructions in Figure 2 being executed using the AccelSim [26] simulator.

The wide range of latencies poses a challenge for statically placing loads to achieve optimal latency hiding. While compilers attempt static scheduling to hide these latencies, achieving a universally optimal schedule for every basic block execution remains challenging. In contrast, out-of-order (OoO) execution dynamically issues ready instructions as operand values become available, offering a more adaptable approach.

A. Idealized OoO Performance on GPUs

We measured the idealized performance of OoO execution on GPUs, leveraging perfect branch prediction, perfect memory aliasing, unbounded register renaming, and a deep reorder queue. Figure 4 illustrates the speedup potential offered by this idealized OoO across a large benchmark suite. The figure demonstrates a geometric mean of 22% using the mentioned optimizations. OoO identifies independent chains at runtime and achieves performance gains by adapting to the latency experienced per warp. Thus, our results show that there is a significant performance gain to be had with OoO execution; however, it must be optimized for the GPU architecture. This speedup sets an upper limit to the performance that can be obtained by OoO execution.

B. Existing GPU Out-of-Order Approaches

Previous research has demonstrated the effectiveness of out-of-order (OoO) execution on GPUs, but these techniques have come with a high hardware cost to improve GPU performance [16], [20], [28], [66]. Register renaming was a common technique used to eliminate false dependencies in all prior work approaches, while some also implemented memory instruction reordering using alias checking or speculative execution, adding considerable hardware overhead [16], [20]. Additionally, some techniques required two-phase execution, switching between in-order and OoO modes, necessitating extensive hardware to save the state of the GPU during mode switches [16], [28], [66].

The best-performing (as self-reported) proposal, LOOG [20], has a self-reported speedup of 16%. However, the evaluation is done using the schedule provided by Nvidia's intermediate representation (PTX), and does not include the additional compiler optimizations when lowering to machine-code SASS. When evaluating with SASS, LOOG demonstrates a 16.5% slowdown. The compiled SASS code has already optimized for target GPUs and has shown more Instruction Level Parallelism than PTX [26]. Our approach thus navigates a more complex optimization landscape, contrasting with previous studies that relied on PTX. Additionally, LOOG would increase the area by at least 1.29% compared to a baseline GPU architecture that employs operand collectors, and an unreported (but possibly significant) impact on clock frequency due to the introduction of long wires for result broadcast. Moreover, LOOG requires repurposing operand collector units, originally intended to enable a multiported register file [35], [36], to enable OOO execution. However, recent GPUs instead employ a

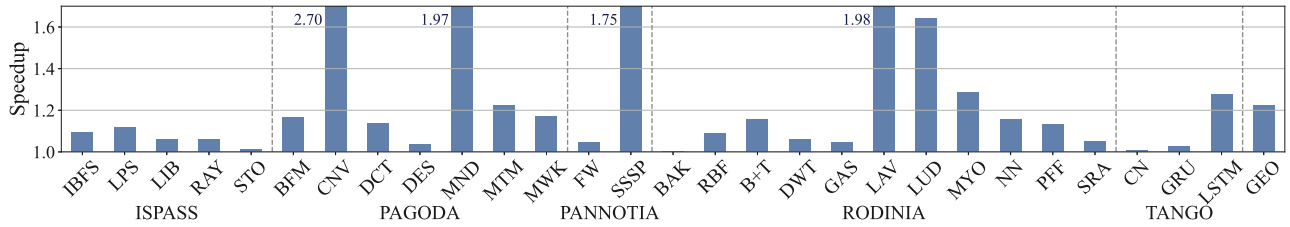

Fig. 4. Idealized performance of OoO execution on GPU.

software-managed operand reuse cache [1], [12], [13], [18], [22] likely worsening overheads and complexity for LOOG.

This paper presents GhOST, a low-overhead OoO approach that requires significantly less area while demonstrating superior performance compared to previous techniques. Table I presents a comparison between GhOST and prior work, highlighting the effectiveness of GhOST's design. GhOST does not require any modifications to the Nvidia ISA, making it immediately implementable and is non-speculative. It does not require register renaming and load-store queues and does not switch between OoO and in-order modes during execution. GhOST achieves 6.9% perfromance gain with only a 0.007% area increase while outperforming state-of-the-art prior work.

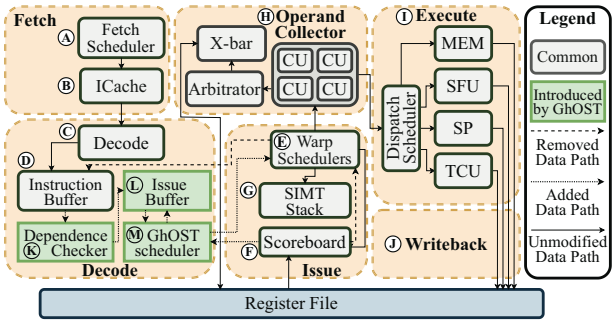

Fig. 5. The GPU pipeline before and after modifications for GhOST.

III. Baseline Execution Model

This section provides an overview of the GPU programming model and describes the micro-architecture of the baseline GPU pipeline modeled in Accel-Sim, which serves as the basis for the GhOST model.

GPU programming model: A GPU program consists of host code that executes on the machine's CPU and device code that executes on the GPU. The device code, called a kernel, is executed in a Single Instruction, Multiple Threads (SIMT) manner [1], [29]. A thread is the basic unit of computation. Threads are grouped in disjoint sets, called warps, that execute in lock-step: they synchronously execute the same instruction and share a program counter. Warps are grouped into disjoint sets called *blocks* or *Cooperative Thread Arrays* (CTAs).

The SIMT stack, which is private for each warp, handles the serialization of the execution of divergent control paths. SIMT stack handling and thread convergence of Nvidia GPUs were changed after the Volta generation [43]; for the ease of presentation, we describe GhOST using the older mechanism.

GPU architecture: The GPU architecture consists of many streaming multiprocessors (SMs). Each SM has up to four scheduling units, with each warp scheduling unit selecting one warp to issue an instruction. It utilizes concurrency to hide latencies; if one warp stalls, the SM can efficiently use fine-grained multi-threading to switch to another warp to execute an instruction. The ratio of the number of warps active on an SM to the maximum warps supported by an SM is called warp *occupancy*. While high occupancy is almost always desirable, very high occupancy can lead to slowdowns in some cases due to increased cache misses and memory contention [25], [34], [53].

The GPU offers various on and off-chip memory structures to support program execution. Each SM has its own L1 cache, shared memory, and various specialized memory regions [45]. All SMs share a common L2 cache, and misses to the L2 are serviced by the off-chip DRAM. Each thread is assigned a set of registers for the entirety of its execution, allowing for faster context switching in the GPU since register file states do not need to be saved. Due to the large number of threads each SM can support, each SM has a large number of registers (64,000 32-bit registers for Nvidia RTX 2060S).

GPU Streaming Multiprocessor (SM) Micro-Architecture: We consider the baseline SM micro-architecture to be the one used in Accel-Sim [26] when configured to model the Nvidia RTX-2060 Super GPU. The overview of the model is shown in Figure 5. It consists of six stages: fetch, decode, issue, read operands, execute, and writeback.

Fetch and Decode: The fetch scheduler Ⓐ selects warps waiting to access the instruction cache (ICache) Ⓑ and executes its fetch request; in the next cycle, the fetched instructions are decoded Ⓒ. A dependence bit-vector marks any RAW or WAW dependencies between the registers of this instruction and any in-flight instructions or older instructions in the IB [8]. The decoded instruction, along with the dependence bit-vector, is placed in the IB, and the valid bit for the instruction is set. The IB has space to hold two instructions per warp.

Issue: Each SM has multiple warp scheduling units Ⓔ.

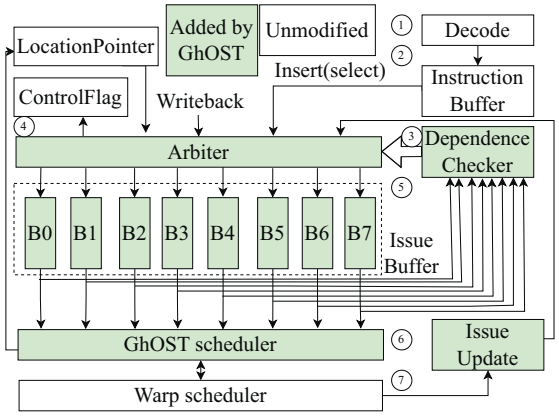

Fig. 6. Architectural diagram for GhOST modifications at the decode stage.

In each cycle, the warp scheduler selects one warp from its assigned warps and issues an instruction from it. The warp schedulers can be instantiated with various policies. The RTX 2060S warp schedulers as modeled by AccelSim use the greedy-then-oldest (GTO) scheduling policy, where instructions from one warp are prioritized in every cycle until the warp stalls and the scheduler moves on to the next warp.

An instruction is issued when: (1) its program counter (PC) matches the PC expected by the SIMT stack; (2) it has no data dependencies; and (3) it secures available issue-pipeline registers (addressing structural hazards). In recent GPUs, warp schedulers issue one instruction per cycle from the selected warp. As such, this paper models the RTX 2060S device to issue only one instruction per cycle. After issuing the instruction, the SIMT stack Ⓖ [10], and the scoreboard are updated.

Read Operands: After issue, instructions are allocated space in the Collector Unit (CU) Ⓗ for operand collection and are kept in the CU till all operands are collected.

Execute: The GPU has different types of execution units Ⓘ. Compute instructions are sent to the Scalar Processing Units (SPs), special function units (SFUs), or tensor core units (TCUs). Memory operations are forwarded to the Memory Unit (MEM).

Writeback: After executing, instructions issue a written request to the RF arbitrator and exit the pipeline Ⓙ. Finally, the scoreboard entry for the instruction is cleared and the dependence vector bits for instructions waiting in the IB for the released registers are cleared.

IV. GhOST

This section presents GhOST, a non-speculative, lightweight, hardware-only Out-of-Order (OoO) instruction scheduling technique designed to reduce GPU stalls. GhOST works with existing warp-scheduling techniques by selecting an unblocked instruction from each warp to be considered for issue by the warp scheduler.

For an instruction to be issued OoO, it must not have any dependence conflicts with issued in-flight instructions, as well as older instructions that are still waiting to be issued in the IsB. Figure 6 shows the GPU pipeline modifications at the decode stage made by GhOST to enable OoO execution. GhOST selects instructions in-order from the instruction buffer ② after they are decoded by the decoder ① and places them in a time-shared Dependence Checker (DC) ③. The DC marks any dependence of this instruction with in-flight instructions and instructions currently in the IsB ⑤ to stop any incorrect OoO execution. Additionally, it ensures no reordering of load instructions against stores and the correct execution of control instructions.

TABLE II
DECODE CHECKER CONFIGURATION

Field	Bits	Usage
Valid	1	Instruction has no control hazard
Index	3	Tracks age of instructions
Mask	32	Tracks active threads for the instruction
IB-dep	11	Checks for dependence with an older instruction in the IsB
Instruction	64	Decoded instruction to be executed
Combined	111	

After dependence checking, the instructions are placed in the IsB. The time-shared GhOST scheduler ⑥ selects and stores 2 instructions per warp, which do not have any dependence conflicts with in-flight instructions and older instructions in the IsB in the Instruction Table (ITab) which is contained within the GhOST scheduler. The warp scheduler ⑦ remains unmodified and looks at one instruction per warp in the ITab to be considered for issue. Details of these structures are explained below.

A. Dependence Checker

Figure 7 shows the hardware diagram of the DC, which is a time-shared structure for warps in a scheduler. Each scheduler has four DCs, equal to the decode throughput. The DC comprises of 8 *IB-dep Calculation Logic* (IB-Calc) structures. Each IB-Calc concurrently checks for any WAR, RAW, and WAW dependence on a register used by this instruction against registers used by instructions of the same warp in the IsB. This is in addition to the RAW and WAW dependence checks performed in the baseline with in-flight instructions. Doing so ensures the correct prevention of incorrect Out-of-Order (OoO) issuing of instructions from the IsB. If any dependence is found between the instruction in the DC and any instruction in the IsB, the IB-dep bit-vector entry corresponding to the instruction in the IsB is set to 1. It also Additionally, it prevents the reordering of loads against stores and stops OoO execution till control instructions are resolved. Details of this are discussed below. When the DC has space, the oldest instruction in the IB from a warp, based on the decode scheduler, is placed in the DC. The DC sets the following bits for an instruction before it is placed in the IsB:

- **Valid**: Validity of instruction.
- **Mask**: The mask for the threads in the warp is copied from the SIMT stack.

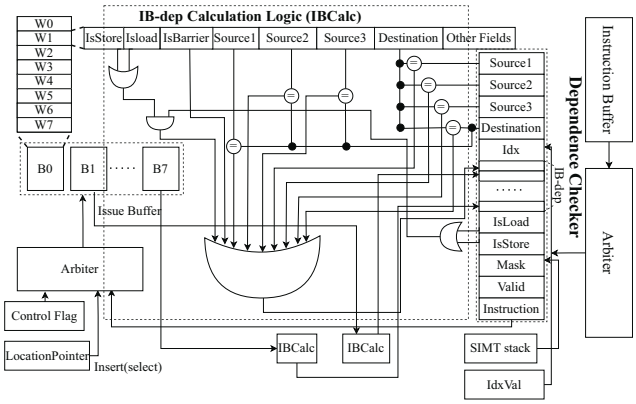

Fig. 7. Dependence Checking of new instruction with one instruction updating its corresponding IB-dep bit in IB-dep vector.

- **IB-dep**: Bit-vector where each bit corresponds to an instruction position in the IsB and is set by an IBCalc and a bit for each register to check if it depends on an in-flight instruction (discussed later).
- **Instruction**: The decoded instruction.
- **Idx**: Bits to track the order in which instructions were placed in the IsB for a warp.

Table II shows the breakdown of the total bits per instruction in the DC.

Each IB-Calc unit performs the following checks against the corresponding IsB instruction before setting its bit in the IB-dep vector:

1) **Data Dependencies:** In addition to keeping track of any RAW and WAW dependencies with in-flight and pending IsB instructions as in the baseline, it also checks for any WAR dependencies of this instruction with any older instruction in the IsB. To keep track of dependencies in the IsB, the read registers of the instruction in the DC are checked against the write registers of the instruction in the IsB to identify RAW hazards. Similarly, the write register of this instruction is checked against the read and write registers of the instructions in the IsB to identify WAW and WAR hazards. If any dependence is found, the bit in the IB-dep vector corresponding to the hazard-causing instruction in the IsB is set to 1.
2) **Memory Instructions:** GhOST enforces memory instructions to execute in-order except for load-load pairs. This implies that load instructions to the same addresses can also be reordered, which is often not allowed in memory consistency models. However, the only way a thread can observe if two load instructions from the same address have been reordered is if another thread writes to the location. However, this would be a data race [37], and the load re-ordering is allowed. GhOST does not reorder instructions marked as atomics by the compiler as this would violate memory model semantics. The *IsLoad* and *IsStore* bits are used to determine if the instruction is a memory instruction. If the instruction in the DC is a load or a store instruction, and the instruction in the IsB it is being compared against is a store instruction or an atomic memory instruction, then the DC IB-dep bit vector entry corresponding to the IsB memory instruction is set to 1. Since instructions are issued only if their IB-dep bit vector is 0, this ensures that only load-load re-orderings are allowed.

Control Instructions: GhOST does not implement branch prediction. Perfect branch prediction leaves the performance of GhOST virtually unmodified (see Table V). Additionally, speculative execution comes with the additional complexity of enabling rollback for thousands of threads on mis-speculation.

GhOST allows the reordering of branch instructions with older instructions in the IsB while respecting all dependencies. Instructions after a branch instruction are not issued until the branch is resolved, avoiding any speculation. The Mask for the branch instruction is popped from the SIMT stack and placed in the DC, which is copied to the IsB when the instruction is moved. This avoids the serialization of instructions against branch convergence points.

Synchronization Instructions: When a synchronization instruction enters the DC, it is not moved to the IsB until all the instructions from the IsB have been issued. When the synchronization instruction (e.g., `syncthreads`) is moved to the IsB, the control flag is raised, and new instructions are not moved to the IsB. After the synchronization instruction has been issued and the branch has been resolved, the control flag is unset, and instructions can be moved from the IB to the DC.

Mask and Idx: The thread mask for the instruction is picked up from the SIMT stack, and the entry is popped from the SIMT stack. The IdxVal is copied as the Idx in the DC, and the IdxVal counter for the warp is incremented.

B. Issue Buffer

Instructions are moved from the DC to the IsB from where they are selected for issue using the GhOST scheduler. When an instruction is moved into the IsB, it is placed in an empty slot pointed to by the Location Pointer for the warp, which is a FIFO queue of locations of empty IsB slots for the warp. The Location Pointer then points to the next empty location or marks the IsB as full, so new instructions are not moved from the DC to the IsB. Index bits are used to track the order of instructions. The IsB is modeled as 8-way banked structure, and each entry of a given bank corresponds to a different warp. Instructions are read from the IsB by the GhOST scheduler to find valid instructions with no data dependence, which can be considered for issue by the warp scheduler. When an instruction is issued from a warp, the IB-dep bits corresponding to the issued instruction are set to 0 for the instructions in the IsB for that warp.

C. GhOST Scheduler

GhOST introduces the GhOST scheduler which is a time-shared structure across warps that serves two functionalities:

1) It selects up to two instructions per warp from the IsB that do not have a data dependence on any older instructions in the IsB and in-flight instructions. 2) It buffers these instructions in the instruction table (ITab) for the warp scheduler to consider during the issue stage scheduling. The selection logic involves choosing instructions for a given warp at any given time. The GhOST scheduler selects a warp to place instructions in the ITab if 1) There was a writeback to the warp or 2) the warp has empty space in the ITab. The time-shared nature of the GhOST scheduler reduces the area and power overhead of implementing GhOST. Additionally, it leaves the interface to the warp scheduler unmodified.

a) Selecting Warp for Dependence Checking: The GhOST scheduler maintains a *Warp-WriteBack* (WWb) bit-vector and an *ITabEmpty* (ITE) bit vector, where each bit corresponds to a warp. When a warp has a write-back, the corresponding bit in the WWb is set. When a warp has empty space in the ITE, the corresponding ITE bit is set to one. The ITE bit is unset for a warp after instructions are placed in the ITab.

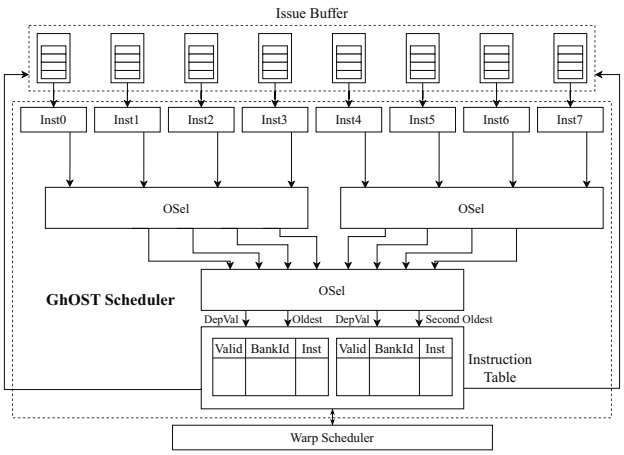

Fig. 8. The time-shared GhOST scheduler.

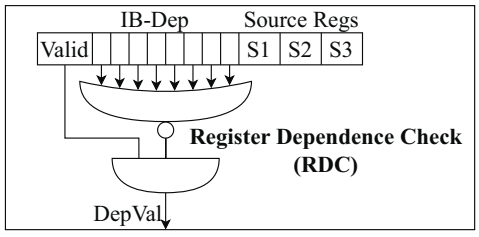

Fig. 9. Register dependence checking hardware for the GhOST scheduler.

b) Selecting Ready Instructions: The hardware implementation of the GhOST scheduler is shown in Figure 8. When the GhOST scheduler selects a warp (as described above), the Register Dependence Checker (RDC) is used to check for

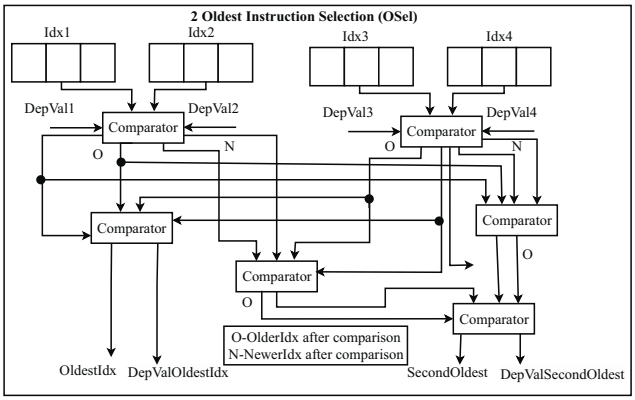

Fig. 10. Oldest Instruction selection hardware for the GhOST scheduler.

data dependencies (See Figure 9). The IB-Dep vector of each instruction in the warp is checked for data hazards with another instruction in the IsB or any in-flight instructions.

Additionally, it is checked if the instruction is valid (See Figure 9). If the instruction has no data dependencies and is valid, the RDC gives an output of 1. This output is fed into the OSel (see Figure 10). The OSel selects the two oldest instructions with no data dependencies. It places the instructions in the ITab, sets the bit of the placed instructions as 1, and stores the bank id of the location of this instruction in the IsB.

c) Issuing Instructions: The warp scheduler selects instructions for issue from the ITab. When an instruction is issued from the ITab, for all the instructions in the IsB belonging to the same warp as the issued instruction, the IB-dep bit corresponding to the issued instruction is set to 0. This marks all WAR dependencies of the issued instruction on newer instructions in the IsB as resolved. RAW and WAW dependencies of in-flight and older IsB instructions are checked against the dependence vector before issue (same as baseline), ensuring correct execution. Additionally, for all instructions from the warp with an *idx* value greater than the *idx* of the issued instruction, the *idx* value is reduced by one. The entry of this instruction in the IsB is marked as invalid, and the bank id is added to the Location Pointer queue for the warp.

D. Warp scheduler

The warp scheduler is unmodified by GhOST. The warp scheduler considers the oldest ready instruction pointed to by the GhOST scheduler. If the GhOST scheduler has not marked any instruction as ready for issue for a warp, then the warp scheduler considers the oldest instruction for the warp. The warp scheduler checks for any structural hazards before issuing an instruction for operand collection.

E. Software Optimizations for OoO Execution

GPUs boast a vast register space, offering up to 255 registers per thread [51]. However, this potential is often not fully

harnessed during program execution [21], [27]. By leveraging these underutilized register files, the compiler can enhance OoO performance through strategic loop unrolling and register renaming, thereby mitigating false dependencies. This optimization broadens the scope for reordering in GhOST, leading to a marked improvement in Out-of-Order performance, as detailed in Section VI-F.

F. Exception Handling:

Nvidia GPUs have very limited support for exception handling [50], and the process running on the GPU is terminated on encountering errors [47]. When a warp encounters a page fault, the page fault is treated as a long stall on Nvidia GPUs [44]. Since GhOST is non-speculative, it can handle page faults in GPUs without any modifications. Like current GPUs, GhOST can return the error code for an exception during execution. However, below, we describe how to add support for precise exceptions if needed.

GhOST-Precise: GhOST can be extended to support OoO execution with precise exception handling for virtual memory in the event kernel-level context switching [52], [58] on page faults is desirable. Prior work, iGPU [39], explored how to support demand paging exceptions in GPUs by exploiting idempotent code regions in the context of *in-order* instruction execution. An idempotent instruction sequence is one that can be re-executed without changing the result. Building on the insights of iGPU, GhOST-Precise utilizes compiler support to mark non-idempotent instructions and adds two additional constraints to GhOST:

- Load instructions are not reordered against each other
- Non-idempotent instructions are not issued till all prior load instructions have hit the TLB.

The additional hardware required to support GhOST-Precise is described below:

- **LoadCounter**: Each warp has a load counter to maintain the number of loads in the pipeline which have not done a TLB check. When a load instruction enters the DC unit it increments the *LoadCounter*, and decrements it after the TLB check.
- **InstLoadCounter**: When an instruction enters the DC the value of the *LoadCounter* is copied into the *InstLoadCounter* and placed with the instruction in the IsB. When a load instruction for a warp has a TLB hit the *InstLoadCounter* for all entries in the IsB for the warp are right-shifted by 1 bit.

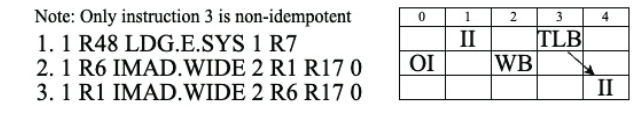

Fig. 11. GhOST-Precise does not issue non-idempotent instructions until all prior load instructions hit in the TLB. Legend: OI: OoO issue, II: In-order issue, TLP: TLB hit, WB: writeback.

If an instruction is marked as non-idempotent by the compiler, it can be issued only after the *InstLoadCounter* value for that instruction is 0. Figure 11 shows how instruction 3 which is non-idempotent is not issued OoO against instruction 1 even though there is no dependence between them till instruction 1 hits in the TLB. If instruction 1 missed in the TLB, GhOST-Precise can start re-execution from instruction 1, re-issue instruction 2 as it is idempotent and then issue instruction 3.

G. GhOST in Action

Figure 12 shows the movement of instructions between the IB, DC, and IsB as a small program executes OoO using GhOST. Figure 12(a) shows an execution cycle where instruction 1 has been issued. Instruction 2 is stalled as it has a RAW dependence on instruction 1. The GhOST scheduler has marked instruction 2 as the oldest instruction ready for issue, and instruction 4 as the next oldest ready for issue in the previous cycle. The warp scheduler issues instruction 3 out-of-order after checking for structural hazards. Warp 0 goes to the GhOST scheduler to get the new oldest ready instructions.

Figure 12(b) shows the next cycle, where instruction 6 is moved from the DC to the IsB, and instruction 7 is moved into the DC. The GhOST scheduler is pointing to instruction 4 as the oldest ready instruction, which is issued OoO from the warp scheduler.

V. Evaluation Methodology

TABLE III
GPU Microarchitectural Parameters

Parameter	**Nvidia RTX 2060S**
Simulator	Accel-Sim
Generation	Turing
SM clock Frequency	1905 MHz
SMs/GPU	34
Warp Schedulers/SM	4
Warp Scheduling Policy	Greedy-Then-Oldest (GTO)
SIMT lane width	32
Max #warps/SM	32
Max #threads/SM	1024
Max Registers/SM	65536
Register File Size	256 KB, 34k 32bit Registers
L1 Cache Size	64KB
IB Size	4 inst/warp
Inst. Issue/Warp	1
Parameter	**Nvidia RTX 3070**
Simulator	Accel-Sim
Generation	Ampere
SM clock Frequency	1132 MHz
SMs/GPU	46
Warp Schedulers/SM	4
Warp Scheduling Policy	Greedy-Then-Oldest (GTO)
SIMT lane width	32
Max #warps/SM	48
Max #threads/SM	1536
Max Registers/SM	65536
Register File Size	256 KB, 34k 32bit Registers
L1 Cache Size	64KB
IB Size	4 inst/warp
Inst. Issue/Warp	1

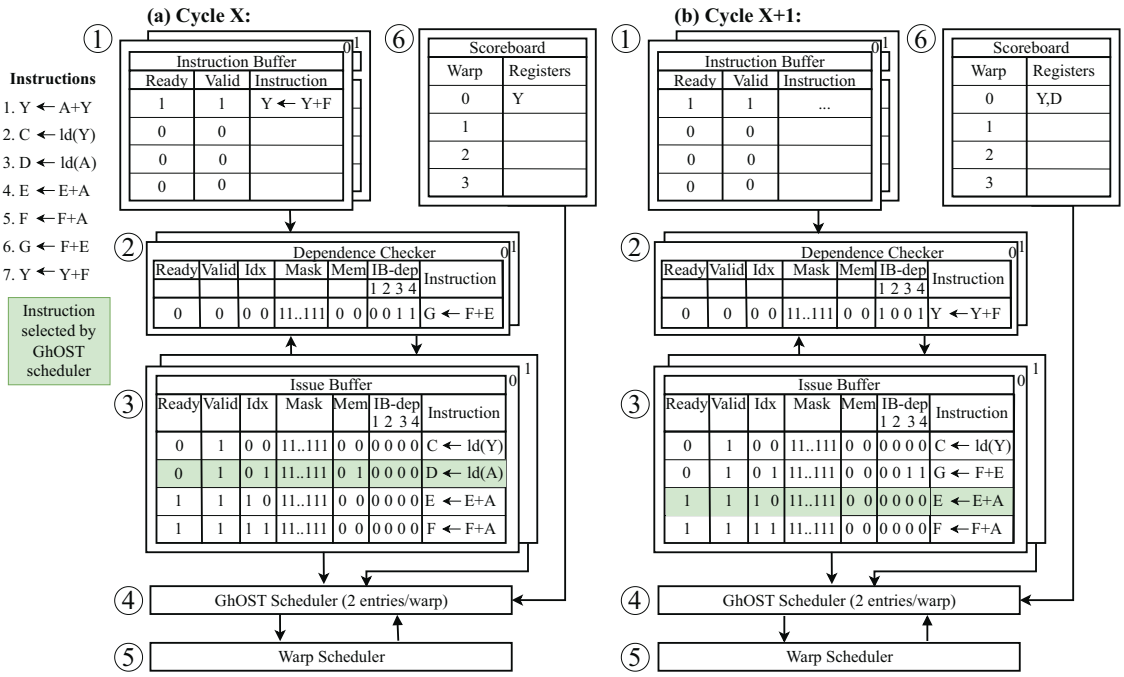

Fig. 12. Execution of a small program on GhOST.

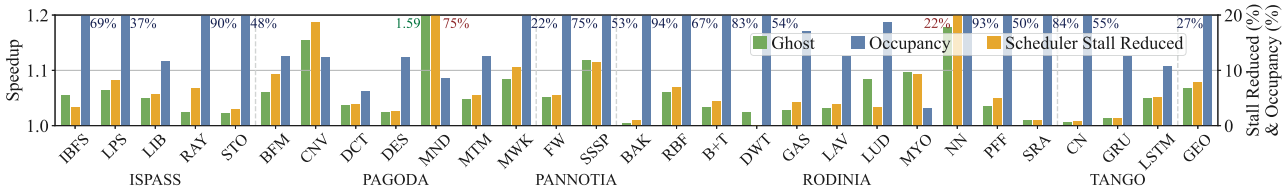

Fig. 13. The performance of GhOST and the reduction in scheduler stalls with OoO execution plotted against the occupancy of applications on the simulated RTX 2060S GPU.

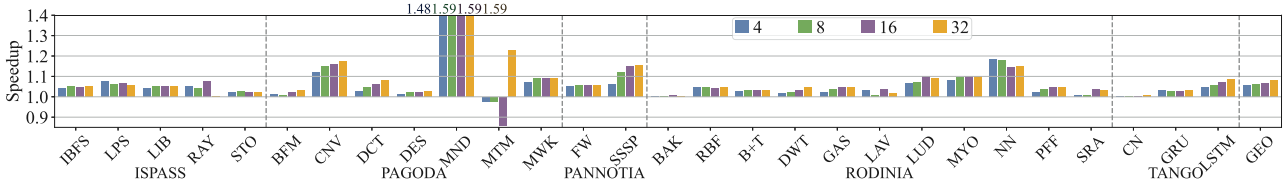

Fig. 14. The performance variance of GhOST-Precise with changing IsB size on the simulated RTX 2060S GPU.

This section describes the experimental context used to evaluate GhOST.

Modelling GhOST: GhOST is implemented on top of the simulated Nvidia RTX 2060S and RTX 3070 GPUs, which are included with the Accel-Sim [26] simulator; Table III shows the microarchitectural parameters of the GPU.

To implement GhOST in the simulator, a dependence checker stage was added between the IB and the IsB. The GhOST scheduler was connected to IsB, the scoreboard and the warp scheduler. OoO control logic was added for the GhOST scheduler. The depth of the decoder, IB and IsB was parameterized to study the effect of increasing IsB size on GhOST.

Benchmarks: 28 GPU applications from standard benchmark suites, namely Rodinia [7], GPGPU-Sim [26], Pannotia [6], and Tango [24] were used to evaluate GhOST.

TABLE IV
EVALUATED BENCHMARKS

Benchmark Suite	Benchmarks
Ispass [26]	bfs (IBFS), LPS, LIB, RAY, STO
Pagoda [64]	beamformer (BFM), DCT, DES, convolution (CNV), mandelbrot (MND), matrixMul (MTM), multiwork (MWK)
Pannotia [6]	SSSP, Floyd-Warshall (FW)
Rodinia 3.1 [7]	b+tree (B+T), dwt2d (DWT), Gaussian (GAS), lavaMD (LAV), LUD, NN, bfs (RBF), myocyte (MYO), particlefilter-float (PFF), srad-v1 (SRA), backprop (BAK)
Tango [24]	CN, GRU, LSTM

The benchmarks cover various GPU domains, including irregular graph processing and large-scale neural networks, and are classified in Table IV. SASS traces were collected on the Nvidia RTX 2060S GPU for these applications to feed into the simulator. Inputs were selected to maximize the input size while ensuring that the GPU did not run out of memory during SASS collection. We could not collect SASS traces for all benchmarks as the GPU ran out of memory when running these applications, the NVbit tool [46] which is used for collecting SASS traces crashed or the application ran for over six days.

Best Performing Prior Out-of-Order Approach: We compare GhOST with LOOG [20], which implements a CPU-style OoO mechanism for the GPU, containing expensive OoO components, such as register renaming and load-store queues. The LOOG Accel-Sim implementation provided with the publication has been used.

VI. Evaluation of GHOST

A. GhOST and Application Occupancy

Figure 13 shows the speedup of GhOST over the SASS baseline, the reduction in scheduler stalls with OoO execution and the average occupancy for each benchmark on the simulated RTX 2060S GPU. GPU occupancy measures the ratio of active warps to the maximum possible warps on an SM, reflecting the utilization of computational resources. Both GhOST and the baseline have 4 instructions in the IB. GhOST gives a speedup of 6.9% over the SASS baseline on the simulated RTX 2060S GPU. The reduction in the scheduler stalls when a scheduler cannot find any instruction ready for issue due to data and/or structural hazards reduces in proportion to the speedup lent by GhOST using OoO execution. Although SASS offers the most optimized binary for the hardware it runs on, GhOST can further improve the runtime of the programs by dynamically reordering instructions, even for applications with high occupancy, while significantly improving the performance of low occupancy workloads.

B. GhOST with Various Warp Scheduling Policies

Figure 16 shows the performance of GhOST with three warp scheduling policies on the simulated RTX 2060S GPU:

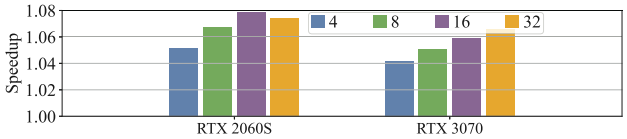

Fig. 15. The performance variance of GhOST with changing IsB size on simulated Nvidia RTX 2060S and RTX 3070 GPUs.

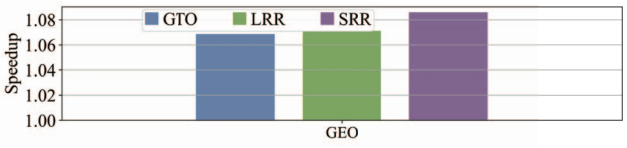

Fig. 16. The performance variance of GhOST with different scheduling policies on the simulated RTX 2060S GPU.

Greedy-the-oldest (GTO) which selects the same warp for scheduling until it stalls, then moves to the oldest ready warp; Lose round robin (LRR) which round robins over warps to find a ready warp for issue; and strong round robin (SRR) which only looks at the next warp to be considered for scheduling. As the scheduling policy becomes less aggressive (SRR), the performance of GhOST improves as OoO execution has a greater impact on finding instructions that are ready for issue.

C. Sensitivity of GhOST to Issue Buffer Size:

Figure 15 shows that the size of the IsB influences the performance of GhOST's OoO execution, while the size of the IB remains constant on the Nvidia RTX 2060S and RTX 3070 GPU models belonging to the Turing and Ampere architecture respectively. GhOST consistently outperforms in-order execution, and the geomean speedups for an IsB size of 8 is 6.9% on RTX 2060S and 5% on RTX 3070. As the depth increases further to 16 and 32 IsB entries per warp, the performance of GhOST further improves to 7.9% and 7.5% for RTX 2060S and to 5.9% and 6.5% for RTX 3070 (for 16 and 32 respectively). Even though configurations 16 and 32 perform better than IsB size 8, these configurations increase the control logic to enable GhOST exponentially. We chose an IsB size of 8 for GhOST, considering the trade-off between OoO performance and hardware overhead. GhOST gives 31% of limit study speedup on RTX 2060S.

Furthermore, performing a sensitivity analysis around IB size reveals that increasing this size to 8 results in a negligible performance improvement of 0.09%.

D. GhOST-Precise

Figure 14 shows that GhOST-precise has a small slowdown due to the constraints added to support precise exceptions for virtual memory handling. GhOST-precise reduces the geomean performance to 6.3% on the simulated RTX 2060S GPU. Increasing the IsB size improves the performance of GhOST-Precise.

E. Area and Power Efficiency

We modeled the design of the Dependence Checker, Issue Buffer, and the GhOST scheduler in RTL and used Synopsys [14] to calculate the area and power of the components added by GhOST on the 45 nm technology node. The GhOST units add 1276 um^2 of area per scheduler, or 0.173536 mm^2 area per GPU. The Nvidia RTX 2060S GPU was developed on the 12nm technology node and has an area of 445mm^2 [59]. To get the area estimate for GhOST, we use a multiplicative factor of 0.17 to convert the area analysis to a 14nm technology node to get an upper limit estimate for the area increase with GhOST [57]. GhOST increases the area of the GPU by 0.007% with a 1.1707 mW increase in power.

F. Effect of Various OoO Optimizations on GhOST

Table V delves into a limit study analyzing various optimizations applied to OoO execution on GPUs with a very deep reorder queue, considering perfect branch prediction, unbounded register renaming, and perfect memory address alias checking. The results highlight that perfect branch prediction and reordering of load operations against store operations offer limited performance benefits. Notably, register renaming enhances performance significantly by expanding the number of instructions for OoO execution. The performance benefits from these optimizations are discussed in detail below:

Perfect Branch Prediction: Perfect branch prediction improves OoO execution by enabling extensive instruction reordering without stalling on branches. It ensures correct branch direction decisions, even in cases involving long-latency operations for branch resolution. Branch prediction has a limited impact on the OoO performance of GPUs. Additionally, implementing perfect branch prediction is prohibitively expensive, requiring the rollback of all threads in a warp in case of misprediction. Due to this high overhead, there has yet to be prior implementation of branch prediction on GPUs to the best of our knowledge.

Perfect Memory Alias Checking: Memory alias checking is crucial for reordering load and store instructions. Implementing perfect alias checking, where address alias information is known at the time of instruction issue, has limited impact on OoO performance, as shown in Table V. This implementation necessitates load-store queues or speculative execution of load instructions, resulting in high hardware overhead with limited performance benefits.

Register Renaming: Aggressive register renaming can help the performance of OoO execution by removing false dependencies. In the perfect register renaming implementation, a new register is assigned to an instruction at the issue stage in case of an anti or output dependence from an unbounded list of free registers. In many GPUs, all warps share a single physical register file. In Table V, we do not consider the potential negative impact of reduced warp occupancy that may be required to free up registers for renaming.

GPUs have a large register space per thread with a maximum limit of 255 registers per thread [51]. The GPU register file is not fully utilized through the execution of the program [21], [27]. The compiler can also assist in removing false dependencies by unrolling loops and using more registers during the program's execution. Figure 18 compares the performance gains of unbounded register renaming in hardware against aggressive unrolling and renaming in software with a limited set of registers (255 registers per thread) for the lavaMD and convolution benchmarks which show significant performance gains by unbounded register renaming. The results show that while unbounded renaming does outperform renaming and unrolling done by the compiler, this performance gap is not significant. This suggests that more aggressive unrolling and higher register utilization to reduce name dependencies can enable more aggressive OoO execution with GhOST without implementing register renaming in hardware.

Despite minimal impacts from individual GPU out-of-order (OoO) optimizations, their collective implementation significantly boosts performance. Issues like false dependencies stemming from name conflicts limit gains from alias checking and branch prediction by hindering OoO instruction issuance. Integrating register renaming with these optimizations overcomes these limitations, enabling more aggressive OoO execution and resulting in significant performance improvements.

G. GhOST against LOOG

The evaluation depicted in Figure 17 and Figure 19 demonstrates GhOST's superior performance compared to the state-of-the-art prior work, LOOG [20] on the RTX 2060S GPU model. Figure 17 shows that GhOST does not negatively affect the performance of any applications, unlike LOOG. Results for the LIB and multiwork benchmarks are missing as LOOG crashed for the applications. In Figure 19 GhOST shows a higher maximum performance than LOOG with no slowdowns on any applications, while LOOG shows a maximum slowdown 0.35×. The cumulative frequency distribution of LOOG shows a very high variability in the performance of LOOG for various applications, while GhOST has a more uniform, non-negative performance impact on applications.

Although LOOG outperforms GhOST in specific applications by efficiently renaming registers and reordering memory instructions, it experiences substantial slowdowns in all other scenarios. This is due to LOOG's reliance on the operand collection stage for out-of-order (OoO) execution, where instructions wait in the reservation station until the write-back phase. As a result, the operand collector may become congested with stalled instructions due to true dependencies. This congestion prevents ready-to-issue instructions from proceeding due to structural hazards, leading to significant performance degradation. Furthermore, LOOG lacks a specific mechanism for congestion-aware Compute Unit (CU) allocation, rendering it highly susceptible to the warp scheduler's scheduling policy. In contrast, GhOST does not negatively affect the performance of any application and is independent of the scheduling policy. Despite efforts to improve performance by increasing the operand collector's size fourfold to 32, it remains smaller than GhOST's IsB capacity for holding instructions.

TABLE V
EFFECT OF VARIOUS OOO OPTIMIZATIONS ON GHOST ON THE SIMULATED RTX 2060S GPU.

Unbounded Register Renaming	✓	✓	✓	✗	✓	✗	✗	✗
Perfect Memory Alias Check	✓	✓	✗	✓	✗	✓	✗	✗
Perfect Branch Prediction	✓	✗	✓	✓	✗	✗	✓	✗
Speedup %	15.1%	14.4%	13.7%	8%	10.1%	7.4%	7.3%	6.9%

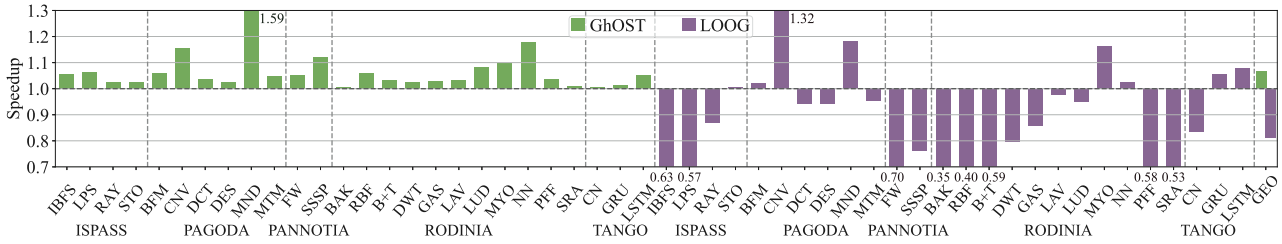

Fig. 17. The performance of GhOST and LOOG on the simulated RTX 2060S GPU.

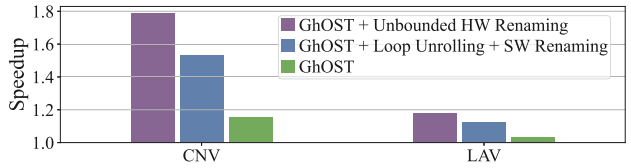

Fig. 18. Comparison of unbounded register renaming in hardware with bounded register renaming (255 registers) in software .

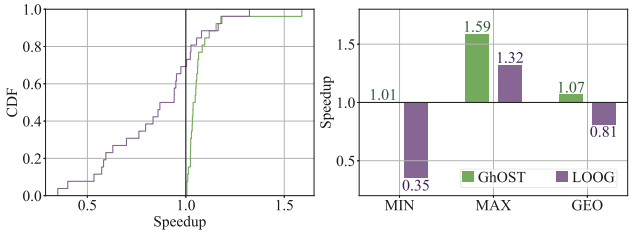

Fig. 19. Performance of GhOST and LOOG against in-order execution on the simulated RTX 2060S GPU configuration.

LOOG introduces significant complexity to the pipeline through a register alias table linked to all collector units for register renaming, load-store queues for memory address disambiguation across threads, and a result broadcast bus. These structures need to manage information for every thread in a warp per instruction, necessitating substantial buffers for storage and broadcast. LOOG modifies the issue, operand collector, and writeback stages. In contrast, GhOST avoids this complexity by making limited modifications to the decode stage with the DC, IsB, and the GhOST scheduler, eliminating the need to store per-thread information for each instruction in a warp.

These findings deviate from their previously published work, which employed the PTXPlus flag on GPGPU-Sim 3 to simulate SASS, which offers limited support for an older version of SASS (10+-year-old Nvidia GT200 GPU) [17] and is incompatible with the newer Pascal architecture they claim to use for their evaluation. As a result, performance numbers in the LOOG paper were overestimated due to the unoptimized version of SASS modeled by the simulator. Moreover, we identified and rectified issues related to synchronization instructions, branches, and dependence checking in their SASS model implementation in the simulator. We have confirmed our results for LOOG when using trace-driven SASS simulation with the LOOG authors.

VII. RELATED WORK

Prior work has explored both software and hardware approaches to enhance GPU utilization.

A. Scheduling Policies

Prior work explores warp interleavings using various warp scheduling techniques to improve GPU resource utilization and reduce stalling. Narasiman *et al.* [40] propose a Two Level Scheduler (TLS) combined with a Large Warp Microarchitecture (LWM). TLS helps distribute long-latency stalls over time, while LWM focuses on creating SIMD-width dynamic subwarps to improve execution units' utilization in warp divergence. RLWS [3] proposes a reinforcement learning-based warp scheduler that can learn and adapt to various workloads. MASCAR [54] implements a memory-aware warp prioritizing scheduler with a cache-access re-execution mechanism to overlap more computation with memory accesses. These schedulers are typically tailored for specific workloads, such as cache-sensitive or compute-heavy.

Nvidia's Shader Execution Reordering (SER) [49] is a scheduling technique that regroups threads on the fly to reduce

execution and data divergence. GhOST can be used with SER to improve the performance of GPUs further.

Twin-Kernel (TK) [15] optimizes GPU memory access by compiling altered kernels with varied memory patterns, pairing them for efficient execution. However, TK's static nature requires creating two versions per kernel, not always possible. GhOST offers a dynamic alternative without workload assumptions.

HyperQ [5] by Nvidia allows empty SMs to start running subsequent independent kernels. This enables multiple kernels to run simultaneously on the GPU, thus reducing the total run time of the program. However, this does not help address the stalls each kernel will face. GhOST is compatible with HyperQ.

Breathing Operand Windows [11] bypasses register file accesses, passing values directly between instructions within the same window, improving the rate of instructions being issued from the OC, and can be used with GhOST to further improve performance.

B. Warp Throttling Techniques

Many works have studied how multiple warps executing on an SM affect the performance of the L1D cache, network bandwidth utilization, and DRAM; and observe that multiple warps sharing the L1D cache can lead to cache thrashing and performance degradation. Studies have been done on reducing the maximum number of active warps at a time on one SM (TLP throttling) or prioritizing specific warps' access to shared resources. Rogers *et al.* proposed CCWS [53], which uses a scoring system to identify warps that re-reference their data in the L1D cache and prioritize their memory accesses.

Dyncta [25] uses a CTA scheduling mechanism based on the application's characteristics to limit the number of CTAs assigned to an SM. This reduces the contention for shared resources between warps. OWL [23] develops a two-level warp scheduler with CTA prioritization. LCS [32] develops warp scheduling techniques to throttle TLP and better utilize memory resources. PCAL [34] showed that while TLP throttling improves L1D footprint, it can lead to under-utilization of other resources and proposed a system to prioritize a subset of warps to use caches while allowing remaining warps to use other GPU resources.

Poise [9] uses machine learning combined with a run-time inference engine to make warp scheduling decisions, optimizing both the TLP and the memory system performance. Virtual threads [65] suggest that increasing the maximum number of CTAs on an SM can favor some applications. GhOST GPU improves the performance of applications by exploiting the hidden ILP in applications at the micro-architectural level using OoO. It thus improves the performance of workloads with regular and irregular memory access patterns. This complements the existing TLP models.

VIII. Conclusion

This paper introduces GhOST, a novel and lightweight out-of-order (OoO) technique for GPUs primarily residing in the decode stage. GhOST leverages the decode stage's pool of instructions, thereby expanding available instruction choices and eliminating the need for expensive OoO techniques to identify reordering opportunities. GhOST introduces instruction reordering by presenting an alternative instruction to the warp scheduler through the existing interface rather than offering the oldest instruction in each warp. The result is a simple and efficient OoO engine that operates predominantly in the decode stage. Experimental results on the simulated Nvidia RTX 2060S and RTX 3070 GPUs using SASS demonstrate that GhOST outperforms previous GPU OoO techniques, enhancing GPU performance by 6.9% and 5% across 28 benchmarks, with only a 0.007% increase in area.

Acknowledgment

We thank members of the Liberty Research Group for their support and feedback on this work. We also thank the anonymous reviewers for the comments and suggestions that strengthened this work. This material is based upon work supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research, under contract numbers DE-SC0022138 and DE-SC0022268. This material is based upon work supported by the National Science Foundation under grants CCF-2107257 and CCF-2107042.

Appendix

A. Abstract

This paper's artifact comprises the source code for the GhOST OoO execution technique, models tuned for Nvidia RTX 2060S and RTX 3070 GPUs, SASS traces to run benchmarks, a pre-compiled LOOG binary, installation instructions, and the source code of support scripts for reproducing results. To facilitate easy execution, a Singularity image is provided to set up the environment. The artifact is available publicly through an archived repository.

Additionally, the artifact describes the requirements and contains instructions for evaluating GhOST, which are detailed in this appendix. A linux operating system with support to run singulatiy container, Synopsys PrimeTime PX, slurm workload manager and python is required to test the artifact. Users of the artifact can reproduce the key GhOST results shown in Figures 3, 13-17, and 19, and evaluate the area and power overhead of GhOST.

B. Artifact check-list (meta-information)

- **Program:** The artifact includes SASS traces of the benchmarks shown in 4 collected on the Nvidia RTX 2060S GPU using the Nvbit tool and are publicly available.
- **Compilation:** GCC 7.5.0 and NVCC V11.6.55.
- **Binary:** The artifact repository contains pre-compiled LOOG binary.
- **Run-time environment:** A Singularity image is provided to run on Linux. Root access is not required. Python3 is necessary for analysis. Synopsys is used for area and power overhead calculation, and Slurm is utilized to run experiments. For diagram plotting, the required Python3 libraries can be installed using the following command:

`pip3 install numpy pandas matplotlib>=3.8.0`

Python3, Synopsys, and Slurm are software dependencies that are not included in the Singularity container.

- **Hardware:** Machine with $>$ 20 GB RAM.
- **Execution:** The experiments can take up to 48 hours to complete.
- **Metrics:** IPC speedup over baseline configuration calculated from Accel-Sim stats is the main metric of comparison reported.
- **Output:** Validation graphs for Figures 3, 13-17, and 19, and raw numbers for the area and power overhead of GhOST are provided.
- **Experiments:** The artifact includes a set of scripts that clone and build the GhOST repository; download the SASS traces, LOOG binary, and Verilog files for area and power validation for GhOST; run the experiments following the experimental methodology in the paper and finally generate Figures 3, 13-17, and 19 and give the area and power overhead values.
- **How much disk space required (approximately)?:** 100 GB.
- **How much time is needed to prepare workflow (approximately)?:** 20 mins.
- **How much time is needed to complete experiments (approximately)?:** The artifact evaluation starts 555 slurm jobs. On a 319-node machine, it took 40 hours to complete all the runs.
- **Publicly available?:** Yes.
- **Archived (provide DOI)?:** 10.5281/zenodo.10874714

C. Description

1) How to access:

- The artifact is available at Zenodo at:
  - 10.5281/zenodo.10874714

  Use the download and installation instructions below to set up the artifact.
- Download Size : 15 GB
- Will require approximately 55 GB of disk space when extracted

2) Hardware dependencies:

- A machine with $>$ 20GB memory required.
- A machine with $>$ 100GB disk space required.
- A machine with $>$ 20 cores for faster execution.

3) Software dependencies: Linux operating system with Singularity container support, Slurm workload manager, and Synopsys PrimeTime PX is required.

D. Download and Installation

- Execute the following commands to add Synopsys to your `$PATH`:

```
export SYNOPSYS_PATH=/path/to/
licensed/synopsys
export SYNOPSYS_ICC2_PATH=/path/to/
licensed/synopsys-icc2
```

- Download and setup the system in `$BASE_DIR`.:

```
curl -sSL https://raw.githubuser
content.com/ishitachaturvedi/
GhOST-accel-sim-framework/dev/
collect_results_artifact/setup.sh |
bash
```

This script downloads the the GhOST repository, LOOG binary, GhOST RTL and evaluation scripts from Zenodo and builds the GhOST repository. It starts a *basic test* to check if the setup is correct.

E. Basic Test

`setup.sh` starts slurm jobs to check if the setup is correct and plots the output result after the jobs are complete.

The jobs should take $<$ 10 mins to finish.

To check if the setup is correct please compare the figure generated in `$BASE_DIR/GhOST-accel-sim-framework/results/min_example.png` against the figure in file `$BASE_DIR/GhOST-accel-sim-framework/gold_files/min_example.png`

F. Experiment workflow

Note: Ensure that the basic test runs correctly before moving to this step and that you have added `$SYNOPSY_PATH` and `$SYNOPSYS_ICC2_PATH` to your `$PATH`.

The primary experiments consist of running GhOST in various configurations and LOOG for performance results, and Synoposys for calculating the area and power overhead for GhOST.

The `GhOST_artifect_evaluation.sh` starts the collection of the performance and GhOST overhead results. To run the script, please do the following:

```
cd $BASE_DIR/GhOST-accel-sim-framework
/ghost_scripts
bash GhOST_artifect_evaluation.sh
```

After it completes the slurm jobs it issues, it generates the plots, figures, and text files for artifact evaluation.

It can take up to 60 hours to generate the plots.

G. Evaluation and expected results

- Evaluating Figures 3, 13-17, and 19:
  The generated Figures will be placed in `$BASE_DIR/GhOST-accel-sim-framework/results`. The corresponding gold files are located in `$BASE_DIR/GhOST-accel-sim-framework/gold_files`.
- Area and power overhead for GhOST as shown in section 6E: The area and power overhead calculated by Synopsys will be placed in `$BASE_DIR/GhOST-accel-sim-framework/GhOST-area-calculation/results`.
  The corresponding gold files are located in `$BASE_DIR/GhOST-accel-sim-framework/GhOST-area-calculation/gold_files`.

H. Experiment customization

Different Makefile flags can be used to build the GhOST binary and collect results on additional configurations:

- `OoO_ON`: OoO execution with GhOST
- `RENAME_REGS_ON`: Turn on perfect register renaming (needs to be used with `OoO_ON`)
- `branch_prediction_ON`: Turn on perfect branch prediction (needs to be used with `OoO_ON`)

- `GhOSTPrecise_ON`: Turn on GhOST Precise (needs to be used with `OoO_ON`)
- `printLDSTLatency_ON`: Print latency of load instructions

REFERENCES

[1] T. M. Aamodt, W. W. L. Fung, and T. G. Rogers, "General-purpose graphics processor architectures," *Synthesis Lectures on Computer Architecture*, vol. 13, no. 2, pp. 1–140, 2018.

[2] J. Alsop, M. D. Sinclair, R. Komuravelli, and S. V. Adve, "GSI: A GPU stall inspector to characterize the sources of memory stalls for tightly coupled GPUs," in *2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 2016, pp. 172–182.

[3] J. Anantpur, N. G. Dwarakanath, S. Kalyanakrishnan, S. Bhatnagar, and R. Govindarajan, "RLWS: A Reinforcement Learning based GPU Warp Scheduler," *arXiv preprint arXiv:1712.04303*, 2017.

[4] P. Barua, J. Shirako, and V. Sarkar, "Cost-driven thread coarsening for gpu kernels," in *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques*, 2018, pp. 1–14.

[5] T. Bradley, "Hyper-Q example," in *NVidia Corporation. Whitepaperv1. 0,*. IEEE, 2012.

[6] S. Che, B. M. Beckmann, S. K. Reinhardt, and K. Skadron, "Pannotia: Understanding irregular GPGPU graph applications," in *2013 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 2013, pp. 185–195.

[7] 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 *2009 IEEE international symposium on workload characterization (IISWC)*. Ieee, 2009, pp. 44–54.

[8] B. W. Coon, P. C. Mills, S. F. Oberman, and M. Y. Siu, "Tracking register usage during multithreaded processing using a scoreboard having separate memory regions and storing sequential register size indicators," Oct. 7 2008, uS Patent 7,434,032.

[9] S. Dublish, V. Nagarajan, and N. Topham, "Poise: Balancing thread-level parallelism and memory system performance in gpus using machine learning," in *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2019, pp. 492–505.

[10] A. ElTantawy, J. W. Ma, M. O'Connor, and T. M. Aamodt, "A scalable multi-path microarchitecture for efficient gpu control flow," in *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2014, pp. 248–259.

[11] H. A. Esfeden, A. Abdolrashidi, S. Rahman, D. Wong, and N. Abu-Ghazaleh, "Bow: Breathing operand windows to exploit bypassing in gpus," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2020, pp. 996–1008.

[12] M. Gebhart, D. R. Johnson, D. Tarjan, S. W. Keckler, W. J. Dally, E. Lindholm, and K. Skadron, "Energy-efficient mechanisms for managing thread context in throughput processors," in *2011 38th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2011, pp. 235–246.

[13] M. Gebhart, S. W. Keckler, and W. J. Dally, "A compile-time managed multi-level register file hierarchy," in *Proceedings of the 44th annual IEEE/ACM international symposium on microarchitecture*, 2011, pp. 465–476.

[14] R. Goldman, K. Bartleson, T. Wood, K. Kranen, C. Cao, V. Melikyan, and G. Markosyan, "Synopsys' open educational design kit: capabilities, deployment and future," in *2009 IEEE International Conference on Microelectronic Systems Education*. IEEE, 2009, pp. 20–24.

[15] X. Gong, Z. Chen, A. K. Ziabari, R. Ubal, and D. Kaeli, "TwinKernels: an execution model to improve GPU hardware scheduling at compile time," in *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 2017, pp. 39–49.

[16] X. Gong, X. Gong, L. Yu, and D. Kaeli, "HAWS: Accelerating GPU wavefront execution through selective out-of-order execution," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 16, no. 2, pp. 1–22, 2019.

[17] GPGPU-Sim. (2015) Manual. [Online]. Available: http://gpgpu-sim.org/manual/index.php/Main_Page#PTXPlus_support

[18] S. Gray. Assembler for NVIDIA Maxwell architecture. https://github.com/NervanaSystems/maxas.

[19] R. Huerta, J.-M. Arnau, and A. Gonzalez, "Simple out of order core for gpgpus," in *Proceedings of the 15th Workshop on General Purpose Processing Using GPU*, 2023, pp. 21–26.

[20] K. Iliakis, S. Xydis, and D. Soudris, "Repurposing GPU Microarchitectures with Light-Weight Out-Of-Order Execution," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 2, pp. 388–402, 2021.

[21] H. Jeon, "Resource underutilization exploitation for power efficient and reliable throughput processor," Ph.D. dissertation, University of Southern California, 2015.

[22] Z. Jia, M. Maggioni, J. Smith, and D. P. Scarpazza, "Dissecting the nvidia turing t4 gpu via microbenchmarking," *arXiv preprint arXiv:1903.07486*, 2019.

[23] A. Jog, O. Kayiran, N. Chidambaram 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 *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '13. New York, NY, USA: Association for Computing Machinery, 2013, p. 395–406. [Online]. Available: https://doi.org/10.1145/2451116.2451158

[24] A. Karki, C. P. Keshava, S. M. Shivakumar, J. Skow, G. M. Hegde, and H. Jeon, "Tango: A deep neural network benchmark suite for various accelerators," in *2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 2019, pp. 137–138.

[25] O. Kayıran, A. Jog, M. T. Kandemir, and C. R. Das, "Neither more nor less: Optimizing thread-level parallelism for GPGPUs," in *Proceedings of the 22nd international conference on Parallel architectures and compilation techniques*. IEEE, 2013, pp. 157–166.

[26] M. Khairy, Z. Shen, T. M. Aamodt, and T. G. Rogers, "Accel-Sim: An Extensible Simulation Framework for Validated GPU Modeling," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, 2020, pp. 473–486.

[27] F. Khorasani, H. A. Esfeden, A. Farmahini-Farahani, N. Jayasena, and V. Sarkar, "Regmutex: Inter-warp gpu register time-sharing," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2018, pp. 816–828.

[28] K. Kim, S. Lee, M. K. Yoon, G. Koo, W. W. Ro, and M. Annavaram, "Warped-preexecution: A GPU pre-execution approach for improving latency hiding," in *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2016, pp. 163–175.

[29] D. B. Kirk and W. H. Wen-Mei, *Programming massively parallel processors: a hands-on approach*. Morgan kaufmann, 2016.

[30] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," *Advances in neural information processing systems*, vol. 25, 2012.

[31] J. Lee, Y. Ha, S. Lee, J. Woo, J. Lee, H. Jang, and Y. Kim, "GCoM: a detailed GPU core model for accurate analytical modeling of modern GPUs," in *Proceedings of the 49th Annual International Symposium on Computer Architecture*, 2022, pp. 424–436.

[32] M. Lee, S. Song, J. Moon, J. Kim, W. Seo, Y. Cho, and S. Ryu, "Improving gpgpu resource utilization through alternative thread block scheduling," in *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2014, pp. 260–271.

[33] S.-Y. Lee and C.-J. Wu, "CAWS: Criticality-aware warp scheduling for GPGPU workloads," in *Proceedings of the 23rd international conference on Parallel architectures and compilation*, 2014, pp. 175–186.

[34] D. Li, M. Rhu, D. R. Johnson, M. O'Connor, M. Erez, D. Burger, D. S. Fussell, and S. W. Redder, "Priority-based cache allocation in throughput processors," in *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2015, pp. 89–100.

[35] J. E. Lindholm, M. Y. Siu, S. S. Moy, S. Liu, and J. R. Nickolls, "United States Patent #7,339,592: Simulating Multiported Memories Using Lower Port Count Memories (Assignee NVIDIA Corp.)," Patent, March, 2008.

[36] S. Lui, J. E. Lindholm, M. Y. Siu, B. W. Coon, and S. F. Oberman, "United States Patent Application 11/555,649: Operand Collector Architecture (Assignee NVIDIA Corp.)," Patent, May, 2008.

[37] D. Lustig, S. Sahasrabuddhe, and O. Giroux, "A formal analysis of the nvidia ptx memory consistency model," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2019, pp. 257–270.

[38] A. Magni, C. Dubach, and M. O'Boyle, "Automatic optimization of thread-coarsening for graphics processors," in *Proceedings of the 23rd international conference on Parallel architectures and compilation*, 2014, pp. 455–466.

[39] J. Menon, M. De Kruijf, and K. Sankaralingam, "igpu: exception support and speculative execution on gpus," *ACM SIGARCH Computer Architecture News*, vol. 40, no. 3, pp. 72–83, 2012.

[40] 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 *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, 2011, pp. 308–317.

[41] J. Nickolls and W. J. Dally, "The GPU computing era," *IEEE micro*, vol. 30, no. 2, pp. 56–69, 2010.

[42] Nvidia. (2015) NVIDIA Multi-Process Service. [Online]. Available: https://docs.nvidia.com/deploy/mps/index.html

[43] ——. (2017) Nvidia tesla v100 gpu architecture. [Online]. Available: https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf

[44] ——. (2018) Nvidia on demand paging. [Online]. Available: https://on-demand.gputechconf.com/gtc/2018/presentation/s8430-everything-you-need-to-know-about-unified-memory.pdf

[45] ——. (2018) NVIDIA Turing GPU Architecture. [Online]. Available: https://images.nvidia.com/aem-dam/en-zz/Solutions/design-visualization/technologies/turing-architecture/NVIDIA-Turing-Architecture-Whitepaper.pdf

[46] ——. (2019) Nvidia nvbit tool. [Online]. Available: https://research.nvidia.com/publication/2019-10_nvbit-dynamic-binary-instrumentation-framework-nvidia-gpus

[47] ——. (2021) CUDA DEBUGGING. [Online]. Available: https://leimao.github.io/downloads/blog/2022-05-25-Proper-CUDA-Error-Checking/cuda_training_series_cuda_debugging.pdf

[48] ——. (2022) Chipmaker Nvidia launches new system for autonomous driving. [Online]. Available: https://www.reuters.com/technology/chipmaker-nvidia-launches-new-system-autonomous-driving-2022-09-20/

[49] ——. (2022) Shader execution reordering - developer.nvidia.com. [Online]. Available: https://developer.nvidia.com/sites/default/files/akamai/gameworks/ser-whitepaper.pdf

[50] ——. (2023) CUDA C++ Programming Guide. [Online]. Available: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

[51] ——. (2023) NVIDIA ada-tuning-guide. [Online]. Available: https://docs.nvidia.com/cuda/ada-tuning-guide/index.html

[52] J. J. K. Park, Y. Park, and S. Mahlke, "Chimera: Collaborative preemption for multitasking on a shared gpu," *ACM SIGARCH Computer Architecture News*, vol. 43, no. 1, pp. 593–606, 2015.

[53] T. G. Rogers, M. O'Connor, and T. M. Aamodt, "Cache-conscious wavefront scheduling," in *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE, 2012, pp. 72–83.

[54] A. Sethia, D. A. Jamshidi, and S. Mahlke, "Mascar: Speeding up GPU warps by reducing memory pitstops," in *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2015, pp. 174–185.

[55] L. Shi, W. Liu, H. Zhang, Y. Xie, and D. Wang, "A survey of GPU-based medical image computing techniques," *Quantitative imaging in medicine and surgery*, vol. 2, no. 3, p. 188, 2012.

[56] N. Stawinoga and T. Field, "Predictable thread coarsening," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 15, no. 2, pp. 1–26, 2018.

[57] A. Stillmaker and B. Baas, "Scaling equations for the accurate prediction of cmos device performance from 180 nm to 7 nm," *Integration*, vol. 58, pp. 74–81, 2017.

[58] I. Tanasic, I. Gelado, J. Cabezas, A. Ramirez, N. Navarro, and M. Valero, "Enabling preemptive multiprogramming on gpus," *ACM SIGARCH Computer Architecture News*, vol. 42, no. 3, pp. 193–204, 2014.

[59] TechPowerUp. NVIDIA GeForce RTX 2060 SUPER. [Online]. Available: https://www.techpowerup.com/gpu-specs/geforce-rtx-2060-super.c3441

[60] J. E. Thornton, "The cdc 6600 project," *Annals of the History of Computing*, vol. 2, no. 4, pp. 338–348, 1980.

[61] L. Wang, M. Huang, and T. El-Ghazawi, "Towards efficient gpu sharing on multicore processors," *ACM SIGMETRICS Performance Evaluation Review*, vol. 40, no. 2, pp. 119–124, 2012.

[62] J. Wei, Y. Zhang, Z. Zhou, Z. Li, and M. A. Al Faruque, "Leaky DNN: Stealing deep-learning model secret with gpu context-switching side-channel," in *2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2020, pp. 125–137.

[63] Q. Xu and M. Annavaram, "PATS: Pattern aware scheduling and power gating for GPGPUs," in *Proceedings of the 23rd international conference on Parallel architectures and compilation*, 2014, pp. 225–236.

[64] T. T. Yeh, A. Sabne, P. Sakdhnagool, R. Eigenmann, and T. G. Rogers, "Pagoda: A gpu runtime system for narrow tasks," *ACM Trans. Parallel Comput.*, vol. 6, no. 4, nov 2019. [Online]. Available: https://doi.org/10.1145/3365657

[65] M. K. Yoon, K. Kim, S. Lee, W. W. Ro, and M. Annavaram, "Virtual thread: Maximizing thread-level parallelism beyond GPU scheduling limit," *ACM SIGARCH Computer Architecture News*, vol. 44, no. 3, pp. 609–621, 2016.

[66] C. Yu, Y. Bai, and R. Wang, "Mipsgpu: Minimizing pipeline stalls for gpus with non-blocking execution," *IEEE Transactions on Computers*, vol. 70, no. 11, pp. 1804–1816, 2020.