Juggler: A Dynamic Warp Scheduler for GPGPUs

Mohammad Nasser, Nitesh Narayana GS, Abhijit Das, *Member, IEEE,* and Debiprasanna Sahoo *Member, IEEE*

***Abstract*—General-Purpose Graphics Processing Units (GPGPUs) accelerate various applications, ranging from graphics rendering to scientific simulations and machine learning (ML). The warp scheduler is a critical element in GPGPUs. It needs to manage thread execution in a way that optimizes performance as well as efficiency. However, different applications, and even different phases of the same application, have varying performance requirements. Hence, fixed static scheduling algorithms are suboptimal. This brief introduces a dynamic warp scheduler named *Juggler*, which dynamically switches between different scheduling strategies at runtime based on phase-specific performance characteristics. Juggler predicts and switches to the most suitable scheduling mode at runtime while improving the overall performance. Evaluations demonstrate that Juggler improves performance by up to 24.84%, with an average gain of 4.66% over state-of-the-art schedulers. These results emphasize the capability of dynamic scheduling to unlock even greater computational power of the GPGPUs.**

***Index Terms*—GPGPU, Warp Scheduler, Dynamic Scheduling.**

I. Introduction

GPGPUs are programmable computing devices originally designed for graphics rendering, but their utility extends to a wide range of applications through CUDA [1] and OpenCL [2]. GPGPUs are particularly suitable for parallel processing tasks, scientific simulations, high-performance computing, and ML. GPGPUs use warps, a group of threads that execute together through a single instruction multiple data (SIMD) lane [3]. GPGPUs rely on warp schedulers to efficiently execute multiple warps. A well-designed scheduling algorithm is critical for maximizing throughput and performance by reducing resource contention, minimizing idle cycles, and balancing the workload. Reducing the number of stalls stands out as a key factor to optimize performance, supported by a plethora of micro-architectural strategies. For instance, in the general-purpose CPU's out-of-order engine, techniques like speculative register reclamation (SRR) that minimizes pipeline stalls by early release of registers [4].

Extensive research highlights the success of the greedy-then-oldest (GTO) scheduling algorithm [5], which utilizes the execution pipeline efficiently and benefits a broad class of applications. However, some applications perform better with alternative scheduling schemes, such as the fair scheduling paradigm offered by loosely round robin (LRR) [5]. Some applications are biased toward the two-level (TL) warp scheduler [6], preferring the hiding of long-latency instructions over fairness. This affinity is not limited to the whole application,

M. Nasser and N. N. GS are with the Universitat Politècnica de Catalunya, Barcelona 08034, Spain (e-mail: mohammad.nasser@upc.edu). A. Das is with the Indian Institute of Technology Hyderabad, Telangana 502284, India. D. Sahoo is with the Indian Institute of Technology Bhubaneswar, Odisha 752050, India. Manuscript received October 25; revised January 26; accepted April XX. Date of publication May XX. *(Corresponding author: M. Nasser.)*

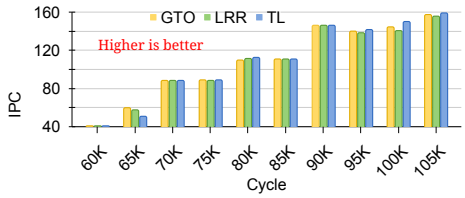

Figure 1: IPC of 'Histogram' benchmark with GTO, LRR, TL.

as different sets of instructions can have different execution behaviors, each of which could potentially perform better with the ideal scheduler that tunes to multiple paradigms depending on the application-specific characteristics of its execution. This raises a critical question: could a single scheduling algorithm consistently achieve optimal performance throughout the entire application? We answer by comparing LRR and TL with the state-of-the-art GTO scheduler. Figure 1 plots the instructions per cycle (IPC) sampled every 5,000 cycles during the run of the 'Histogram' benchmark. GTO initially performs the best until the second half, when it under-performs compared to TL, indicating a phase where TL is favored. This observation highlights the necessity for a dynamic scheduling overcoming fixed schedulers by adapting to various execution phases.

In this work, we propose ***Juggler*, a mode-selection-based warp scheduler** that enables seamless transitions between scheduling strategies during execution, adapting to the application phase's affinity towards a specific scheduling scheme. More specifically, we make the following major contributions:

1) We analyze a diverse set of applications and show that distinct execution phases favor different warp scheduling policies (Section III).
2) We demonstrate the limitations of static scheduling and propose *Juggler*, a hardware mechanism that dynamically adapts scheduling policies to phase behavior (Section IV).
3) We implement Juggler on a state-of-the-art architecture and show performance improvements of up to 24.84% (4.66% on average) with negligible hardware overhead.

II. Background

Modern GPUs adapt the single instruction multiple threads (SIMT) model, where threads run simultaneously on SIMD cores. Figure 2 abstracts the Turing GPU architecture [3], where each SIMD lane has dedicated resources such as L0 cache, dispatcher, register file, and warp scheduler. A group of lanes forms a streaming multiprocessor (SM) that includes a private L1 cache. SMs are organized hierarchically into texture processing units (TPUs) and graphics processing clusters (GPCs), enabling GPUs to support massive thread parallelism. The GPU's thread block scheduler assigns concurrent thread arrays (CTAs) to SMs for execution. A CTA represents a grid

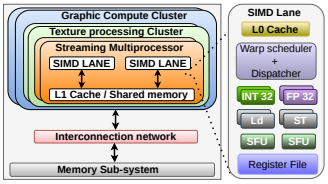

Figure 2: A standard GPU architecture.

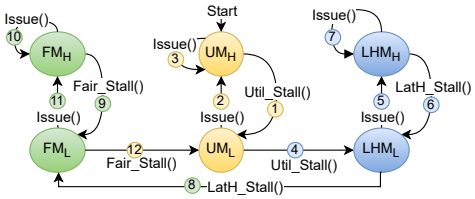

Figure 3: Predictor state machine diagram.

of threads, and each SM further divides these CTAs into warps, which typically consist of 32 threads in NVIDIA GPUs [3]. Warps are issued for execution following the implemented scheduling policy. Although the specific warp schedulers in commercial GPUs are not officially disclosed, it is widely acknowledged that the GTO policy [5] is commonly used.

III. Challenges and Motivation

Prior research [7] shows that while GTO outperforms LRR and TL warp schedulers in several benchmarks (e.g., 'backdrop-rodinia-2.0' and 'hotspot-rodinia-2.0'), it underperforms in others such as 'AsyncAPI' and 'Polybench-2DConv', where LRR excels. Similarly, TL performs better in benchmarks like 'Histogram' and 'Parboil SGEMM'. This shows no single scheduler is optimal across all workloads, motivating the need for adaptive schemes. Our experimental study shows that benchmarks exhibit phase-like behavior and may favor particular schedulers in different phases, as discussed in Section I, suggesting an adaptation of dynamic warp scheduling. As performance is critically sensitive to a plethora of factors, including cache hits and stalls, our work draws its inspiration from RLWS [8], which proposes an adaptive scheduler based on reinforcement learning aimed at minimizing stalls. We calculate the Pearson correlation between IPC and stall cycles, obtaining a high geometric mean of 0.87 across 13 of 15 benchmarks, highlighting the critical role of stall reduction. Motivated by this, we introduce Juggler, a dynamic warp scheduler that can switch modes in runtime depending on the execution patterns and performance metrics across phases.

IV. Dynamic Warp Scheduling With Juggler

A. Predictor Design

Juggler employs three scheduling modes: Utilization (UM), Fairness (FM), and Latency-Hiding (LHM), which correspond to the GTO, LRR, and TL schedulers, respectively. A naïve state machine might switch modes solely based on current stall behavior, ignoring historical stall patterns. To overcome this limitation, we generalize the design using an $n.m$-bit predictor. Here, n bits (set to 2) distinguish between the three scheduling modes, while m bits track 2^m mode histories to improve transition accuracy. Larger values of m provide higher prediction confidence. Such bit-based predictors are widely used in micro-architecture designs, including branch predictors [9] and prefetchers [10], for effectively capturing history. Juggler dynamically updates the confidence level of the current scheduling mode based on stall behavior: fewer stalls strengthen confidence, while more stalls reduce it. Stall events are continuously monitored in the background and evaluated against mode-specific thresholds. This process is governed by a carefully designed state machine that avoids bias and oscillation, allowing circular switching across UM, LHM, and FM, as shown in Figure 3. This design ensures fair opportunities across modes with minimal hardware complexity. Juggler begins execution in UM to maximize pipeline utilization. Experiments further show that switching to LHM before FM yields better performance, since LHM more effectively mitigates chains of compulsory misses caused by long-latency instructions [6].

Table 1: Scheduling mode conditions.

	Condition
Util_Stall()	$Stall_c > Stall_p +$ UTILIZATION_THRESHOLD (UTH)
Fair_Stall()	$Stall_c > Stall_p +$ FAIRNESS_THRESHOLD (FTH)
LatH_Stall()	$Stall_c > Stall_p +$ LATENCY_THRESHOLD (LTH)

A challenge with Juggler is that rapid switching can limit the effective exploitation of a mode, particularly under high stall rates. To address this, we introduce a threshold-based scheme in which transitions occur only when predefined stall conditions are met (Table 1). Each mode tolerates up to X stalls before confidence is lowered or a transition is triggered. Once the threshold is reached, a control register is updated to switch the scheduling mode; this update occurs outside the critical execution path and therefore incurs no additional latency. For example, starting in high-confidence UM_H, if stalls reach the utilization threshold (UTH) and satisfy the $Util\_Stall()$ condition, Juggler shifts to low-confidence UM_L (Figure 3, transition T1), resets the stall counter, and sets $histBit$ to '0'. If, while in UM_L, stalls again reach UTH with $histBit = 0$, the mode switches to LHM_L (Figure 3, T4). Issuing instructions in low-confidence states increases confidence without changing the mode (T2, T5, T11), while high-confidence states persist upon instruction issue (T3, T7, T10). The stall counter resets after every confidence or mode update. This unbiased, round-robin switching among UM_L, LHM_L, and FM_L guarantees fairness with low complexity. While alternative metrics, such as memory instruction rate or instruction mix variation, could also indicate phase changes, Juggler intentionally relies on stalls to minimize hardware complexity and ensure rapid, low-overhead responsiveness.

B. Juggler Workflow: An Example Walkthrough

Figure 4 illustrates a simplified example of Juggler's runtime behavior, omitting thresholds for clarity. The example assumes that the benchmark is already in execution and illustrates a snapshot taken during steady-state operation, rather than the initial scheduling state. Each window corresponds to a cycle and shows the warp instruction queues. Fetch

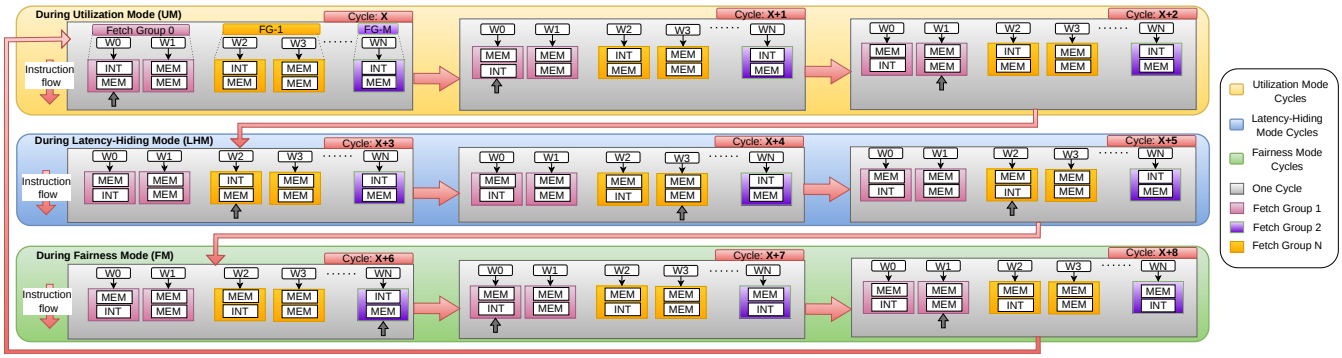

Figure 4: Juggler workflow.

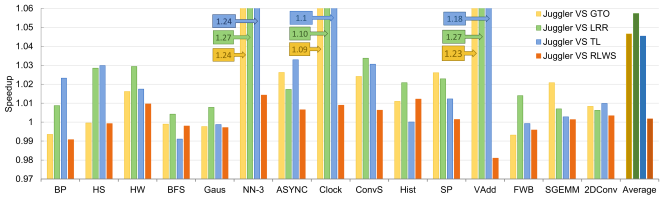

Figure 5: Juggler speedup compared to GTO, LRR, TL, and RLWS; greater than 1 is better.

groups (FG), collections of active warps, are marked for each cycle. Within each window, an arrow indicates the active warp selected by the scheduler. If the top instruction of that warp does not cause a stall, Juggler issues it for execution.

Cycle X: As per Figure 3, with Juggler being in UM_L, issuing an `INT` instruction from the top of warp `W0`. The successful issuance increases confidence in the UM mode (transition: T3).

Cycle X+1: Warp `W0` encounters a `MEM` instruction that stalls, reducing Juggler's confidence in the UM mode, transitioning back to UM_L (T1). Juggler then schedules warp `W1`.

Cycle X+2: Warp `W1` also encounters a `MEM` stall. With UM confidence at its minimum, Juggler transitions to LHM_L (T4) and switches to warp `W2` from FG-1.

Cycle X+3: In LHM_L, Juggler schedules warp `W2`, which issues an `INT` instruction, increasing confidence in the LHM mode and transitioning to LHM_H (T5).

Cycle X+4: Warp `W3` from FG-1 issues a `MEM` instruction, causing a stall and lowering confidence in the LHM mode, thereby transitioning back to LHM_L (T6).

Cycle X+5: Following LHM protocol, Juggler next schedules warp `W2`. Another `MEM` stall occurs, as the confidence in LHM reaches its minimum. Juggler then transitions to FM_L (T8).

Cycle X+6: In FM_L, Juggler schedules warp `WN`, which issues an `INT` instruction and increases FM confidence, transitioning to FM_H (T11). FM advances in its circular queue to warp `W0`.

Cycle X+7: Warp `W0` stalls on a `MEM` instruction, reducing confidence in the FM mode and transitioning to FM_L (T9).

Cycle X+8: A subsequent `MEM` stall in warp `W1` causes Juggler to revert to the UM mode, transitioning to UM_L (T12).

This walkthrough shows how Juggler dynamically transitions between scheduling modes to adapt to application behavior. In practice, these transitions are governed by the stall-based thresholds in Table 1, ensuring balanced and effective scheduling throughout the application execution.

Table 2: Simulation GPU configuration.

Simulator	GPGPU-SIM [11]	Version	v4.0.1
Architecture	Turing	Model	RTX 2060
SMs	30	Shared mem.	64KB
Warp scheduler/SM	4	L1D	64KB, Full assoc.
Active Warps/SM	32		128B, 4 Banks
Threads	1024	L2	128KB/MC
Warp size	32 threads		16-way, 128B Line
Benchmarks	Rodinia 2.0/3.1 [12], SDK 4.2 [13], Pannotia [14], Parboil [15], Polybench [16]		

V. Evaluation

A. Performance Analysis

Figure 5 illustrates Juggler's performance gains over GTO, LRR, TL, and RLWS by dynamically switching between scheduling modes based on workload phases. It shows particularly strong results in LRR-friendly benchmarks like 'ASYNC' and '2DConv' as well as TL-friendly benchmarks such as 'VAdd', 'Hist', and 'SGEMM', aligning with our discussion about phase affinity in Section III. For instance, Juggler achieves an 18.17% performance improvement over TL in the TL-friendly benchmark 'VAdd' and a 1.73% over LRR in the LRR-friendly 'ASYNC'. Even in GTO-friendly benchmarks like 'Clock' and 'HW' where intermediate scheduling modes (FM and LHM) are leveraged during different execution phases, Juggler outperforms GTO by 9.85% and 1.61%, respectively. However, Juggler shows slight performance declines of 0.68% in 'FWB' and 0.64% in 'BP' compared to GTO, the best performer. These drops are attributed to the predictor's limitations under long stall sequences, as emphasized in Section V-B. Overall, Juggler delivers average speedups of 4.66% over GTO, 5.73% over LRR, and 4.53% over TL, demonstrating its broad adaptability and performance optimization across various benchmarks. Compared to RLWS [8], Juggler boosts performance across benchmarks like 'HW', 'Hist', 'Clock', and 'ASYNC' with minor slowdowns on 'VAdd ', 'BP ', and 'FWB.' On average, Juggler achieves a 0.17% speedup while incurring 12.23× less hardware overhead, as detailed in Section V-E.

B. Juggling Effect on Total Stalls

Juggler aims to predict the scheduling mode that minimizes total stalls. Figure 6 shows an average stall reduction of 11.36% across benchmarks. The largest gains occur for

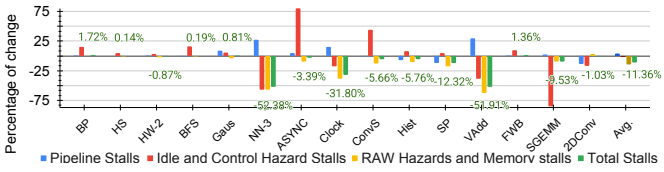

Figure 6: Juggler vs. GTO stalls along with their breakdown.

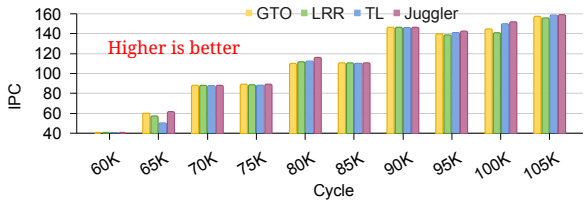

Figure 7: IPC of 'Hist' with GTO, LRR, TL, and Juggler.

NN-3 (24.84%), VAdd (23.67%), and Clock (9.85%), corresponding to stall reductions of 52.38%, 51.91%, and 31.80%, respectively, primarily due to fewer RAW- and control-hazard stalls. Benchmarks with moderate gains, such as SP (2.60%), SGEMM (2.62%), and ConvS (2.08%), exhibit stall reductions of 12.32%, 9.53%, and 5.66%; these improvements mainly arise from reduced RAW-hazard stalls, except for SGEMM, which also benefits from an 85.38% reduction in control-hazard stalls. In contrast, GTO-favored benchmarks such as BP and FWB experience slight stall increases (1.72% and 1.36%), leading to minor performance losses (0.64% and 0.68%). Overall, Juggler primarily reduces RAW-, memory-, and control-hazard stalls, improving performance across benchmarks.

C. Scheduling Mode Switching at Runtime

Figure 7 shows Juggler's sensitivity to affinity changes by reporting the cumulative IPC of the 'Hist' benchmark, sampled every 5K cycles from 60K to 105K. Unlike static schedulers, Juggler dynamically alternates between scheduling modes based on the application behavior. For example, having 100 instructions between cycles 60K to 65K, instead of committing to a single scheduler, Juggler may allocate 50 to GTO, 30 to LRR, and 20 to TL, reducing stalls and improving IPC. This behavior is evident at cycle 65K, where GTO begins to outperform TL and LRR, indicating an affinity for UM, which Juggler exploits, improving IPC. By cycle 80K, TL surpasses GTO and LRR, showing affinity towards LHM; Juggler adapts accordingly, achieving cumulative performance gains over TL. This highlights Juggler's ability to track phase affinity, initially UM and later LHM, optimizing performance across phases.

We link this observation to a study on Juggler's cycle distribution, demonstrating its adaptability through execution. On average, across 15 benchmarks, Juggler spent 45.16% of cycles in UM, 28.99% in FM, and 25.85% in LHM. For example, in 'Hist', Juggler allocates 67.3% of the cycles in UM, 16.3% in FM, and 16.4% in LHM, favoring UM, while we notice more even cycle-distribution in 'VAdd,' with 39.5% in UM, 29.1% in FM, and 31.4% in LHM. This adaptive phase-aware scheduling underlies Juggler's effectiveness.

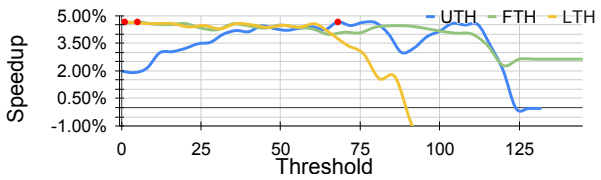

Figure 8: Exploring the impact of the thresholds (UTH, FTH, LTH), aiming for the lowest values that yield optimal speedup.

D. Sensitivity Analysis

Threshold: The performance implications of adjusting threshold values emphasize the need to balance scheduling modes to avoid mispredictions. Experiments on the Turing architecture reveal that optimal performance is achieved using *Utilization Threshold* $(UTH) = 68$, *Fairness Threshold* $(FTH) = 5$, and *Latency-Hiding Threshold* $(LTH) = 1$. Figure 8 shows that reducing *UTH* impairs performance in UM sensitive benchmarks, while showing improvements in FM and LHM sensitive benchmarks during intermediate phases. Performance peaks at *UTH* = 68, with a 4.66% average speedup over GTO, then declines as the predictor struggles to recover from UM, when the threshold exceeds the number of consecutive stalls possible. Similarly, *FTH* improves performance till a value of 5, after which spending more cycles in FM harms non-fairness-sensitive benchmarks. Beyond *FTH* = 125, performance gains stabilize at 2.63% as Juggler remains in FM mode, as it never satisfies the $Fair\_Stall()$ condition. For *LTH*, performance peaks at 1, with diminishing returns as it increases. Beyond *LTH* = 75, performance drops sharply, particularly in non-latency-sensitive benchmarks, with -12.00% slowdown at *LTH* = 115, as the scheduler locks in LHM, unable to meet the $LatH\_Stall()$ condition. UM requires a higher threshold due to its scheduling policy: repeatedly issuing instructions from the same warp leads to earlier stalls compared to FM and TL, which rotate among warps to defer dependencies. As a result, UM exhibits more concentrated stall bursts, whereas FM and TL distribute stalls over time. However, different execution phases of a benchmark may exhibit varying sensitivity to latency and fairness, which Juggler detects dynamically and exploits by switching scheduling policies accordingly. The thresholds are empirically tuned for the simulated RTX-2060; extending the framework to other GPU configurations may require re-tuning. This motivates future work on dynamic threshold adaptation mechanisms, such as fuzzy-logic or ML-based approaches, which could adjust thresholds online in response to workload phases and architectural characteristics.

History Length: Using a 2-bit predictor in Juggler extends the residency in a given scheduling mode, delaying adaptation to phase shifts and degrading performance. For example, transitioning from UM to LHM may require twice as many mispredictions compared to a 1-bit predictor, introducing a lag in adjusting to the workload's phase affinity. During this lag, the workload may shift back toward UM, further compounding the delay and resulting in a cycle of suboptimal scheduling decisions. Empirically, a 2-bit predictor increases cycles spent in FM by 1.62% compared to a 1-bit predictor, which negatively impacts benchmarks that are not FM-sensitive. In particular,

'NN-3' and 'Clock' show performance drops of 5.21% and 2.17%, respectively, when using the 2-bit predictor.

In highly parallel GPUs, where rapid adaptation is essential, the 1-bit predictor enables Juggler to respond quickly to both phase shifts and workload variations. This results in a higher average speedup of 4.66% over GTO, compared to 4.11% with the 2-bit predictor. We identified the 1-bit history length as optimal for Juggler, as longer histories only exacerbate the adaptation delays observed with the 2-bit predictor.

Ordering Mode: Juggler transitions from UM to LHM, then FM, leveraging LHM's strength in masking long-latency instructions, which are prevalent in many benchmark phases (see Section IV). We developed Reverse-Juggler (R-Juggler), which switches from UM to FM before LHM, to validate this design. R-Juggler achieves a lower average of 4.25%, versus 4.66% for Juggler, confirming that prioritizing LHM over FM aligns better with benchmark affinities and yields more effective optimization.

E. Hardware Overhead Analysis

Juggler overhead relies on bit predictors, threshold values, and previous cycle counts. The bit predictors require $\log_2(n)+m$ bits, where n, m represent the number of modes and history bits, respectively. Juggler's total overhead is: $\log_2(n)+m+32+\log_2(UTH)+\log_2(FTH)+\log_2(LTH)$ bits, where the 32 bits count for cycle tracking overhead. On the Turing architecture, Juggler incurs a total overhead of 43 bits, underscoring its minimal footprint. Other architectures may require more bits based on the threshold values. However, even with conservative estimates of using 32 bits per threshold, the total remains under 17 bytes, an insignificant impact relative to the GPU architecture. Comparing Juggler's 43 bits overhead to RLWS's 208-byte requirement [8], Juggler substantially reduces the hardware cost by 38.69×. Even under worst-case assumptions with exaggerated thresholds, Juggler maintains a significant overhead reduction of 12.23×.

VI. Related Work

Dynamic warp scheduling is critical to optimizing GPU performance. iPAWS [7] alternates the scheduling between greedy and round-robin based on instruction issue patterns, incurring high learning and recovery overheads. Juggler uses mode toggling to achieve comparable performance. RLWS [8] employs a reinforcement-learning–based scheduling that learns workload behavior by rewarding decisions that lead to instruction issues and penalizing those that result in stalls, achieving comparable speedups; the hardware cost is significantly high. Juggler, unlike RLWS, explicitly detects phases as per the state machine in Figure 3 and under threshold transition conditions specified in Table 1. ACWS [17] and the cache-locality-based scheduler [18] focus on cache-sensitive workloads, while Juggler targets execution phases of various workloads. MASCAR [19] favors memory-bound applications, BAWS [20] minimizes barrier stalls, while CAWS [21] and CAWA [22] perform scheduling to improve memory misses and cache contention. These designs are orthogonal to Juggler, making them, with their unique addressing of the scheduling problem that can be extended within Juggler.

VII. Conclusions

Juggler is a dynamic warp scheduling scheme designed to help GPGPUs overcome the limitations of static schedulers by adapting to phase-specific workload behaviors at runtime. By dynamically switching between UM, FM, and LHM based on real-time execution patterns, Juggler optimizes performance and reduces stalls. Our implementation of Juggler achieves an average performance improvement of 4.66%, surpassing state-of-the-art schedulers with a negligible hardware cost. Moreover, it paves the way for future research into scheduling algorithms, encouraging the exploration of additional application characteristics that could inspire new, customized scheduling modes for a diverse range of applications.

VIII. Acknowledgments

This work was supported in part by SPARC-3603, NSF-MEITY 3149156, SERB SRG/2023/001108, and IITH SG/IITH/f382/2025-26/SG-212. The first author acknowledges support from the Indian Council for Cultural Relations (ICCR) during their Master's studies, during which most of this work was carried out. The authors used ChatGPT, a generative AI tool, to assist with language editing, typesetting, scripting, and proofreading; all technical content, experiments, conclusions, and figures are the authors' own.

References

[1] "CUDA, C++ Programming Guide," https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html, accessed: 2025-09.
[2] "OpenCL," https://www.khronos.org/opencl/, accessed: 2025-09.
[3] N. Corporation, "NVIDIA Turing GPU Architecture," 2018.
[4] S. Mehta, "Speculative register reclamation," in *HPCA*, 2023.
[5] T. G. Rogers *et al.*, "Cache-conscious wavefront scheduling," in *MICRO*, 2012.
[6] V. Narasiman *et al.*, "Improving GPU performance via large warps and two-level warp scheduling," in *MICRO*, 2011.
[7] M. Lee *et al.*, "iPAWS: Instruction-issue pattern-based adaptive warp scheduling for GPGPUs," in *HPCA*, 2016.
[8] J. Anantpur *et al.*, "Rlws: A reinforcement learning based gpu warp scheduler," *ArXiv*, vol. abs/1712.04303, 2017.
[9] J. E. Smith, "A study of branch prediction strategies," in *ISCA*, 1981.
[10] B. R. Godala *et al.*, "Pdip: Priority directed instruction prefetching," in *ASPLOS*, 2024.
[11] A. Bakhoda *et al.*, "Analyzing cuda workloads using a detailed gpu simulator," in *ISPASS*, 2009.
[12] S. Che *et al.*, "Rodinia: A benchmark suite for heterogeneous computing," in *IISWC*, 2009.
[13] "NVIDIA C/C++ SDK CUDA Code Samples," https://developer.nvidia.com/cuda-code-samples, accessed: 2025-09.
[14] S. Che *et al.*, "Pannotia: Understanding irregular gpgpu graph applications," in *IISWC*, 2013.
[15] J. A. Stratton *et al.*, "Parboil: A revised benchmark suite for scientific and commercial throughput computing," *CRHPC*, 2012.
[16] S. Grauer-Gray *et al.*, "Auto-tuning a high-level language targeted to GPU codes," in *InPar*, 2012.
[17] W. Chen and W. Tong, "Acws: Adaptive cache-state aware warp scheduling based on cache feature analysis," in *ICFTIC*, 2022.
[18] W. Hu *et al.*, "Cache-locality based adaptive warp scheduling for neural network acceleration on gpgpus," in *SOCC*, 2022.
[19] A. Sethia *et al.*, "Mascar: Speeding up gpu warps by reducing memory pitstops," in *HPCA*, 2015.
[20] Y. Liu *et al.*, "Barrier-aware warp scheduling for throughput processors," in *ICS*, 2016.
[21] S.-Y. Lee *et al.*, "Caws: Criticality-aware warp scheduling for gpgpu workloads," in *PACT*, 2014.
[22] ——, "Cawa: Coordinated warp scheduling and cache prioritization for critical warp acceleration of gpgpu workloads," in *ISCA*, 2015.