# Investigating Machine Learning Warp Schedulers in General Purpose GPUs

Ryan Wang Eliot Yoon Steve Zang
*Department of Computer Science*
*University of California, Los Angeles*
Los Angeles, USA
{ryanwang25, eyoon1131, zangbruin007}@ucla.edu

***Abstract*—GPUs rely heavily on warp scheduling for latency hiding of operations such as loads from memory. Therefore, warp scheduling becomes important for finding ready warps to ensure the GPU is always achieving the highest throughput. Different GPU workloads exhibit vastly different behavior depending on whether they are memory bound, compute bound, or utilize techniques like warp specialization. Therefore, effective scheduling strategies for different kernels look vastly different depending on their characteristics. However, existing warp scheduling strategies like linear scanning, round robin and greedy then oldest (GTO) are static. This motivates a need for a warp scheduler that can adapt to a kernel's characteristics. One area that is promising in this regard is machine learning, where machine learning models can use runtime features to approximate a scheduling function that ranks candidate warps according to their expected usefulness. In this paper, we propose a machine learning warp scheduler and implement it in the Vortex RISCV GPGPU simulator [1]. Subsequently, we compare its performance to existing static strategies across a variety of workloads. Importantly, we find that our ML scheduler is competitive with both GTO and RR policies, motivating further research.**

## I. Introduction

GPUs are parallel processors that employ a single instruction multiple threads (SIMT) model, in which many threads execute the same instruction stream in parallel over different data elements. In hardware, the most basic unit of execution is a thread and groups of threads executing the same instruction are grouped into an abstraction called a warp [2]. As early as 2005, papers like Sun Microsystems' Niagara advocated for latency hiding, arguing that "The parallel execution of many threads effectively hides memory latency" [3]. This concept was extended to GPUs in which warps take advantage of a coarse-grained multi-threading but nonetheless benefit from how parallel execution of many warps can hide memory latency. However, the ability to hide latency heavily depends on the warp scheduling policy finding the ready warps that maximize the execution units in use while other warps perform a long latency memory operation. The effectiveness of latency hiding therefore depends not only on the number of resident warps, but also on the scheduler's ability to identify and issue ready warps while others are stalled on long-latency operations. Poor scheduling decisions can leave execution units idle even when many warps are active, reducing overall throughput.

The hardware structure in charge of selecting which warps to execute is the warp scheduler. The warp scheduler maintains the state of resident warps and selects from among the ready warps each cycle. When the currently executing warp stalls, the scheduler must select a different ready warp to continue execution. The warp scheduling policy is the algorithm used to choose the warp to be executed. While NVIDIA's GPU warp scheduling policy remains proprietary, previous works have modeled NVIDIA's warp scheduling policy using linear scanning, round robin, or greedy then oldest [4]. Importantly these scheduling policies are static and make specific implicit assumptions about the workload behavior. However, modern kernels have vastly varying characteristics, such as being memory bound, compute bound, or having many diverging branches. Thus the best scheduler varies across applications, and an adaptive scheduling policy must take all these characteristics into account. To account for the need for an adaptive scheduling policy that takes in a wide variety of features, we investigate the effectiveness of machine learning based warp scheduling policies in this paper. Specifically, we propose a lightweight machine learning predictor that scores candidate warps using runtime features. This method avoids relying on fixed heuristics and instead attempts to identify scheduling decisions that are better aligned with the characteristics of a particular kernel.

In this work, we make the following contributions:

- We implement and evaluate multiple warp scheduling policies in Vortex [1].
- We design a lightweight ML-based scheduler using runtime warp features.
- We compare ML-based scheduling against baseline, round-robin, and GTO policies.
- We evaluate the impact on IPC and execution time across representative GPU kernels.
- We analyze the strengths and limitations of a perceptron-inspired scheduling model relative to traditional heuristic schedulers.

## II. Related Work

### A. Traditional Warp Scheduling Policies

A significant body of work has been devoted to traditional static heuristic-based warp scheduling because they provide a

practical balance between performance and hardware complexity. However, because these heuristics are fixed, it can make them inflexible when faced with diverse workloads.

One of the simplest scheduling policies is round robin which maintains all warps in a sequence and always selects the next ready warp in the sequence, looping back to the beginning after it reaches the end. This approach maximizes fairness and tries to expose thread level parallelism, but may sacrifice locality by frequently switching between warps [5].

Greedy Then Oldest (GTO) scheduling was proposed as an alternative to round robin which tries to makes up for its weaknesses. Compared to round robin which switches to a different warp at every scheduling decision, GTO continues issuing instructions from the currently executing warp until it stalls. At this point, the scheduler then selects the oldest ready warp and repeats this greedy then oldest selection process. Empirically, GTO often outperforms round robin by improving locality and thus reducing cache contention [5].

However, works investigating this observe that the effectiveness of a warp scheduling policy is highly workload dependent, thus no single static scheduler consistently achieves the best performances across all applications [5]. Memory bound kernels may benefit from different scheduling policies than compute bound kernels and divergence heavy kernels may benefit from a completely different policy. This directly motivates more sophisticated warp scheduling policies like two-level scheduling and cache concious wavefront scheduling which incorporate additional workload information like cache information into scheduling decisions [5], [6].

Despite differences in all of the above scheduling policies, they still share the commonality that they all use fixed heuristics that are hand crafted. Therefore, the effectiveness regarding a set of applications is still dependent on how well this heuristic reflect the behavior of the particular application. These limitations motivate the exploration of adaptive scheduling approaches that can leverage runtime information rather than relying solely on hand-crafted heuristics. This motivates our investigation into machine learning based warp scheduling.

### B. Warp Specialization and Workload Aware Execution

Warp specialization is a GPU optimization technique in which different warps are assigned distinct functional roles within a kernel rather than having every warp execute the same sequence of operations. An example of a kernel that utilizes this is FlashAttention-3 which has warps that specialize as a producer and consumer to hide the latency of memory loads and keep the Tensorcore at maximum utilization [7]. Specifically, having producer and consumer warpgroups allows overlapping of memory loads and tensorcore execution in a staggered two stage pipeline fashion. This has also been used to great effect in NVIDIA's CUTLASS library which is a collection of hand written kernels for performing operations like GEMM [8]. While warp specialization has been used to great effect to improve throughput, it also introduces heterogeneity among warps. Some warps become critical producers while other warps become consumers, introducing dependencies between warps. Thus, warp scheduling decisions become critical to maintaining high throughput because stalling of one producer can lead to stalls on multiple downstream consumers. One approach that takes advantage of this kernel characteristic is WASP which proposes a modified warp scheduler that give the hardware explicit metadata on what pipeline stage each warp executes and schedules warps accordingly [9]. Our work shares WASP's observation that warp characteristics should influence scheduling decisions, but instead investigates whether a lightweight machine-learning predictor can learn these priorities directly from runtime behavior rather than relying on explicitly provided metadata.

### C. Machine Learning in Hardware Prediction

One area in hardware where machine learning has already seen success is in hardware prediction problems. One of the most influential applications is branch prediction, where a machine learning model called a perceptron was utilized to great success [10]. Unlike previous approaches which utilize heuristics or exact pattern matching with lookup tables, perceptron branch predictors can learn correlations between program history and branch outcomes specific to each workload. Furthermore, perceptron branch prediction demonstrates that machine learning approaches can provide competitive accuracy in a diverse set of workloads while maintaining reasonable hardware complexity, meeting the per-cycle latency requirements to be implemented on the critical path. Specifically, it is worth noting that these perceptron branch predictors are very adept at capturing long range correlations, that traditional methods struggle with or require large memories to capture. Although perceptrons are now most often employed as part of a hybrid predictor with TAGE, they remain important in demonstrating the effectiveness of machine learning techniques in hardware. Their continued inclusion in modern hybrid predictors suggests that learned models are capable of capturing execution patterns that fixed heuristic approaches may fail to recognize. More broadly, this success illustrates how machine learning can complement traditional hardware heuristics by adapting to workload-specific behavior. Inspired by this success, our work investigates whether a lightweight perceptron-inspired scoring model can similarly improve GPU warp scheduling decisions. Unlike branch prediction, however, our implementation does not perform online weight updates and instead uses statically chosen weights to evaluate the feasibility of learned scoring functions in the warp scheduling context.

## III. Methods

### A. Warp Scheduling Policies

*1) Linear Scanning [baseline]:*

It replicates Vortex's original scheduling behavior. On every call to `selectWarp` it iterates over warp indices $0, 1, \ldots, N-1$ and returns the first index $w$ satisfying `active_warps.test(w) && !stalled_warps.test(w)`. The policy is fully

deterministic and stateless, but it biases toward low-indexed warps, which can delay progress for high-indexed warps when the warp pool is large.

*2) Round Robin Scheduler [RR]:*
It maintains a single integer `last_selected_warp` initialized to $-1$. Each invocation begins scanning at index $(\texttt{last\_selected\_warp} + 1) \bmod N$ and wraps around until it either finds a ready warp or exhausts all candidates. The selected warp index is stored in `last_selected_warp` for the next cycle. This provides fairness across the warp pool but may reduce temporal locality because it switches warps even when the current warp is still ready.

*3) Greedy Then Oldest Scheduler [GTO]:*
It implements a two-phase policy. In the greedy phase, if the warp issued last cycle (`current_warp`) is still active, unstalled, and has a non-empty ibuffer, it is re-selected immediately without scanning. In the oldest phase, if the current warp is no longer ready, the scheduler scans all warps and returns the one with the smallest value in its `last_executed_cycle` array—i.e., the warp that has waited the longest since its last execution. The `notifyExecution` callback updates `last_executed_cycle_[wid]` each time a warp issues an instruction. GTO promotes cache and register-file reuse through the greedy phase while the oldest-first fallback prevents indefinite starvation.

*4) ML Scheduler [ML]:*

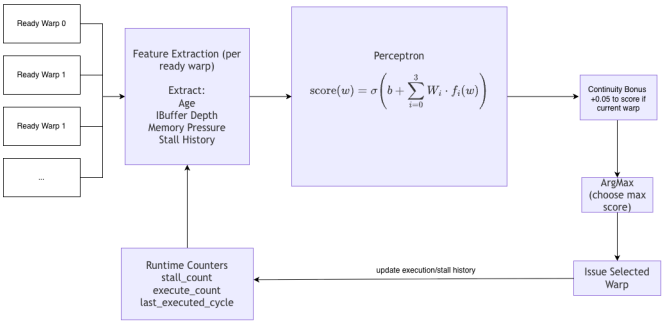


Fig. 1. Overview of the ML-based scheduler. At every cycle, the perceptron scores all ready warps and chooses the one with the max score. The issued warp then informs execution history and is utilized for the next scheduling decision.

Unlike online-learning perceptron predictors, the weights in our current implementation are fixed and chosen based on scheduling intuition. Consequently, the scheduler should be viewed as a perceptron-inspired scoring model rather than a fully learned online predictor. It assigns a scalar score to every ready warp at each scheduling step and issues the highest scoring warp. The scoring model is a single-layer perceptron with a sigmoid output activation. Four features are extracted per warp per cycle:

1) **Age** (`FEAT_AGE`): Cycles elapsed since the warp last executed, computed as `cycle` − `last_executed_cycle_[wid]`, capped at 100 and normalized to $[0, 1]$.
2) **Ibuffer Depth** (`FEAT_IBUFFER_DEPTH`): Current number of decoded instructions in the warp's instruction buffer (`warp.ibuffer.size()`), capped at 32 and normalized to $[0, 1]$.
3) **Memory Pressure** (`FEAT_MEMORY_PRESSURE`): Cumulative stall rate, computed as `stall_count_[wid]`/(`execute_count_[wid]`+ `stall_count_[wid]`).
4) **Stall History** (`FEAT_STALL_HISTORY`): Cumulative stall count, capped at 20 and normalized to $[0, 1]$.

The perceptron scoring function is:

$$\text{score}(w) = \sigma\left(b + \sum_{i=0}^{3} W_i \cdot f_i(w)\right), \quad \sigma(x) = \frac{1}{1 + e^{-x}}$$

The fixed weight vector and bias are tuned to encode known scheduling intuitions: $W_{\text{age}} = 1.2$ (favor warps not recently issued to prevent starvation), $W_{\text{ibuf}} = 0.7$ (prefer warps with decoded work ready), $W_{\text{mem}} = -0.9$ (penalize warps under sustained memory pressure), $W_{\text{stall}} = -0.4$ (mildly penalize chronically stalled warps), and $b = 0.2$. A continuity bonus of $+0.05$ is added to the currently executing warp's score before the argmax to discourage spurious context switches, analogous to GTO's greedy phase. Stall counts are incremented via the `notifyStall` callback, which is forwarded from `Emulator::notifyWarpStall`. This callback is triggered within `Core::decode()` at the ibuffer backpressure point. Execution counts are updated through `notifyExecution`. All per-warp counters are zeroed in `reset()`, which is called when a new kernel is launched.

## IV. Methodology

### A. Simulation Environment

All scheduling policies were implemented and evaluated on the SimX software simulator provided by the open-source Vortex RISC-V GPGPU platform [11]. We chose SimX over RTL simulation due to its significantly faster development and evaluation cycle, enabling efficient experimentation and performance analysis while preserving the warp scheduling dynamics.

### B. Scheduler Interface and Integration

To support dynamic scheduling policies, we introduce a `WarpScheduler` abstract base class in `sim/simx/scheduler.h`. The interface exposes three virtual methods: `selectWarp`, which is called once per simulated cycle and returns the index of the next warp to issue (or $-1$ if none is ready); `notifyExecution`, which is called after a warp successfully issues an instruction so that schedulers can update per-warp execution timestamps; and `notifyStall`, which is called when the ibuffer stage detects pipeline backpressure on a warp. The concrete implementations are compiled into `scheduler.cpp` and linked into the SimX build via an addition to `sim/simx/Makefile`.

### C. Benchmark

We evaluate each scheduling policy on a diverse set of GPU kernels that span a range of computation and memory access patterns. We utilize a Vortex configuration set to `DNUM_WARPS` = 16 as opposed to the default of `DNUM_WARPS` = 4, in order to emphasize the impact of each warp scheduling policy. The tests are briefly described below, with deviations from the default test configuration (such as matrix dimensions, tile size, etc) noted:

- **vecadd**: Element-wise vector addition.
- **conv3**: $3 \times 3$ convolution with per-output-element accumulation.
- **sgemm**: Naive single-precision general matrix multiply (GEMM).
- **sgemm2 (tiled)**: Tiled SGEMM using shared-memory blocking to improve data reuse and reduce off-chip bandwidth pressure. Run with A and B matrices of size $32 \times 32$.
- **sgemm_tcu**: Tensor-core-accelerated SGEMM using Vortex's `EXT_TCU` extension. Run with matrix dimensions of $64 \times 64 \times 64$.
- **diverge**: Branch divergence test designed to exercise different control-flow patterns.
- **dogfood**: Arithmetic/instruction unit test suite.
- **dotproduct**: Parallel dot product using shared memory reduction. Run with number of words set to 128.
- **madmax**: FPU stress test with heavy register pressure and sustained FPU throughput.
- **mstress**: Indirect memory access stress test with pointer chasing and non-coalesced loads.
- **sort**: Parallel rank-based sort with heavy memory pressure.
- **stencil3d**: 3D box filter/averaging stencil kernel.

### D. Metrics

Performance assessment relies on metrics collected by the `Emulator` and printed by `main.cpp` at the end of the simulation:

- **Instructions Per Cycle (IPC)**: Calculated as `instruction_count / cycle_count`, where `instruction_count` increments in `Emulator::step()` after every successful instruction execution and `cycle_count` directly correlates with wall-clock execution time on the simulated hardware.

## V. Evaluation

### A. IPC Results

Table I presents the IPC achieved by each scheduling policy across all benchmarks.

Broadly speaking, the kernels can be grouped into three categories based on IPC:

- **High**: sgemm, madmax, sort, and stencil3d all fall into this cateogry, with IPCs ranging from 2.3 to 3.6. madmax is the ceiling, which makes sense given that it is essentially a kernel of pure FMA chains. The most notable pattern with this group is that for all of them, Linear performs significantly worse than the others. For sgemm and madmax, RR and GTO perform the best, while for sort and stencil3d, ML performs the best. This is an interesting result, and makes sense given the kernel profiles. sgemm and madmax are purely compute-bound (for sgemm, the matrix sizes are small enough that they entirely fit in cache, meaning memory pressure is not a large factor), so the primary scheduling goal is to simply avoid re-issuing stalled warps, which RR and GTO achieve well with no need for more sophisticated heuristics. sort and stencil3d, by contrast, exhibit more dynamic runtime behavior. In these cases, the ML scheduler's ability to reason about features such as instruction buffer depth and stall history allows it to make more informed per-warp decisions. This becomes more meaningful when warp behavior is heterogeneous across warps.
- **Medium**: conv3, sgemm2, and sgemm_tcu fall into this category, with IPCs ranging from 1.7 to 2.1. The most notable result is that for all of these kernels, ML outperforms or matches the others. In particular, sgemm2 shows the most significant improvement with the ML scheduler. This is likely due to the shared memory tiling, and the ML scheduler's ability to recognize when warps have decoded work ready vs. when they are stalled due to memory latency and shared memory synchronization barriers. conv3 is interesting in that while its profile is very similar to sgemm (compute-heavy with cached data from global memory), its per-thread memory access pattern is more complex, placing it at the boundary between compute-bound and memory-bound behavior. This is where the ML scheduler's reasoning ability becomes useful, leading to its advantage over the other schedulers. sgemm_tcu is notable in that Linear outperforms both RR and GTO and performs similarly to ML. This is consistent with the TCU pipeline issuing work in long fixed-latency bursts: while RR spreads issues across warps that all stall on the same pipeline boundary, Linear's warp-0 bias aligns better with the TCU's issue policy. The ML scheduler is also able to approximate this behavior implicitly.
- **Low**: vecadd, diverge, dogfood, dotproduct, and mstress fall into this category, with IPCs ranging from 0.4 to 0.9. These kernels all cluster below 1.0 and do not show significant differences across schedulers. vecadd and dotproduct are fundamentally memory-bound with low arithmetic intensity, meaning no scheduling policy can issue data faster than the memory hierarchy can deliver it. For mstress, the serial load-use dependency chain means the second load cannot be issued until the first completes. diverge has heavy branch divergence that kills throughput and causes warp serialization. dogfood is purely a minimal functional unit test with no ILP or reuse. The commonality among these kernels is that all of them have a fundamental bottleneck that no scheduling policy can address. In other words, the scheduler is not able to make any meaningful decisions.

TABLE I
IPC BY KERNEL AND WARP SCHEDULING POLICY

| Scheduler | vecadd | conv3 | sgemm | sgemm2 | sgemm_tcu | diverge | dogfood | dotproduct | madmax | mstress | sort | stencil3d |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Linear | 0.480 | 1.931 | 2.464 | 1.943 | 1.871 | 0.896 | 0.709 | 0.708 | 3.031 | 0.919 | 2.303 | 2.603 |
| RR | 0.466 | 1.947 | 2.962 | 1.916 | 1.766 | 0.924 | 0.691 | 0.699 | 3.596 | 0.870 | 3.028 | 2.916 |
| GTO | 0.466 | 1.968 | 2.978 | 1.931 | 1.737 | 0.912 | 0.689 | 0.702 | 3.590 | 0.856 | 3.051 | 2.882 |
| ML | 0.479 | 2.008 | 2.713 | 2.132 | 1.886 | 0.909 | 0.697 | 0.707 | 3.348 | 0.874 | 3.051 | 3.095 |

### *B. Discussion*

While it is difficult to draw a concrete conclusion from this limited test suite, a few observations stand out. Most significantly, the ML warp scheduler never performs the worst out of the schedulers. For all of the medium IPC kernels, ML performs the best, while for the high IPC kernels, even when ML is outperformed by RR and GTO, it still significantly outperforms Linear. For the low IPC kernels, all schedulers converge to similar performance, which is expected. This suggests that the ML scheduler's feature set, especially memory pressure and stall history, provides a useful signal in kernels with dynamic behavior, while still performing near-parity with the simpler policies on kernels where scheduling is less important.

Another notable observation is that the baseline Linear warp scheduling policy is highly inefficient. As Vortex is ultimately intended for research and not performance, this is reasonable, although it is an important observation for developers who may wish to use Vortex as a performance baseline or extend it for more practical workloads.

It is important to note that all of our results were achieved on a specific Vortex configuration with 16 available warps, designed to maximize the impact of the warp scheduling decision. Additionally, we specifically finetuned our ML scheduler's feature weights based on manual analysis and experimentation rather than training. A production implementation would likely need to be pretrained on a much wider variety of kernel types with additional features to learn the optimal weights. Nevertheless, we argue that the primary benefit is not that the perceptron provides optimal scheduling of every warp, but rather that it provides robust performance across diverse kernel types. Finally, the hardware overhead from using such a policy is minimal: a warp's score can be computed in a single cycle using a dedicated FMA unit, making it a low-cost addition even relative to simple policies like RR or GTO.

## VI. CONCLUSION

In this paper, we proposed and evaluated a lightweight machine learning warp scheduler for general-purpose GPUs, implemented within the Vortex RISC-V GPGPU SimX simulator targeting version v2.3. We compared the ML scheduler against three static baselines—linear scanning, round-robin, and greedy then oldest—across twelve GPU kernels spanning memory-bound, compute-bound, and tensor-core workloads.

Our results show no single scheduler dominates universally, although patterns emerge across kernel groups. Linear consistently underperforms the other policies, particularly on compute-intensive and memory-latency sensitive kernels. GTO's strength lies in regular compute-intensive workloads where its oldest-warp-first policy effectively hides memory latency, and RR performs similar to GTO across most kernels. In contrast, the ML scheduler delivers its largest gains on workloads with structured yet irregular execution behavior (ex: sgemm2, sgemm_tcu, stencil3d, and conv3). By including features that capture instruction-buffer occupancy and memory pressure, the ML scheduler can make more informed scheduling decisions, achieving IPC improvements of up to 10% over GTO on *sgemm2* and approximately 8% on *sgemm_tcu*.

## VII. STATEMENT OF WORK

Steve implemented the baseline warp scheduling policies in vortex and ran the initial evaluation experiments. Eliot extended on experimentation by exploring benchmark results when simulated on different GPU configurations and a more extensive test suite. Ryan was the primary organizer for the final project paper, compiling the preliminary research to introduce and motivate the problem and outlining the best structure to present the evaluation and conclusion. For the paper writing, Ryan wrote the abstract, introduction, and related works sections, Eliot wrote the methodology and evaluation sections, and Steve wrote the methodology, evaluation, and conclusion sections.

## REFERENCES


[1] F. Elsabbagh, B. Tine, P. Roshan, E. Lyons, E. Kim, D. E. Shim, L. Zhu, S. K. Lim, and H. kim, "Vortex: Opencl compatible risc-v gpgpu," 2020. [Online]. Available: https://arxiv.org/abs/2002.12151

[2] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym, "Nvidia tesla: A unified graphics and computing architecture," *IEEE Micro*, vol. 28, no. 2, pp. 39–55, 2008.

[3] P. Kongetira, K. Aingaran, and K. Olukotun, "Niagara: a 32-way multithreaded sparc processor," *IEEE Micro*, vol. 25, no. 2, pp. 21–29, 2005.

[4] J. Singh, I. S. Olmedo, N. Capodieci, A. Marongiu, and M. Caccamo, "Reconciling qos and concurrency in nvidia gpus via warp-level scheduling," in *2022 Design, Automation Test in Europe Conference Exhibition (DATE)*, 2022, pp. 1275–1280.

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

[6] 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 *2011 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2011, pp. 308–317.

[7] J. Shah, G. Bikshandi, Y. Zhang, V. Thakkar, P. Ramani, and T. Dao, "Flashattention-3: Fast and accurate attention with asynchrony and low-precision," 2024. [Online]. Available: https://arxiv.org/abs/2407.08608

[8] V. Thakkar, P. Ramani, C. Cecka, A. Shivam, H. Lu, E. Yan, J. Kosaian, M. Hoemmen, H. Wu, A. Kerr, M. Nicely, D. Merrill, D. Blasig, A. Atluri, F. Qiao, P. Majcher, P. Springer, M. Hohnerbach, J. Wang, and M. Gupta, "CUTLASS," Jan. 2023. [Online]. Available: https://github.com/NVIDIA/cutlass

[9] N. C. Crago, S. Damani, K. Sankaralingam, and S. W. Keckler, "Wasp: Exploiting gpu pipeline parallelism with hardware-accelerated automatic warp specialization," in *2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2024, pp. 1–16.

[10] D. Jimenez, "Fast path-based neural branch prediction," in *Proceedings. 36th Annual IEEE/ACM International Symposium on Microarchitecture, 2003. MICRO-36.*, 2003, pp. 243–252.

[11] B. Tine, K. P. Yalamarthy, F. Elsabbagh, and K. Hyesoon, "Vortex: Extending the risc-v isa for gpgpu and 3d-graphics," in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 754–766. [Online]. Available: https://doi.org/10.1145/3466752.3480128