Attention, Watch Your Progress: Balancing Warp-Specialized GPU Pipelines

Fangjia Shen
NVIDIA Corporation
Santa Clara, California
fangjias@nvidia.com

Nicolai Oswald
NVIDIA Corporation
Santa Clara, California
noswald@nvidia.com

Timothy G. Rogers
NVIDIA Corporation; Purdue University
West Lafayette, Indiana
timrogers@purdue.edu

***Abstract*—Contemporary GPU kernels for deep learning workloads, particularly attention mechanisms in transformers, employ sophisticated parallel programming techniques to maximize hardware utilization. These techniques create software pipelines that synchronize the operation of specialized hardware units such as Tensor Cores, Tensor Memory Accelerators, and SIMT cores at the warp level. However, current warp scheduling mechanisms are not designed for these new programming paradigms, leading to suboptimal performance due to imbalance and inefficient instruction issue patterns.**

We propose Progress-Aware Warp Scheduling (PAWS), a lightweight scheduling mechanism that adapts to modern attention kernels by leveraging pre-existing machine code information to transmit priority information to the warp scheduler. Our approach enables the warp scheduler to identify and prioritize warps executing slower program phases, thereby improving overall pipeline throughput. We demonstrate that compilers can automatically implement this mechanism, providing significant performance improvements for attention kernels on modern GPU architectures. We perform a comprehensive parameter sweep of 1800 attention implementations that represent both current and anticipated future attention kernels. PAWS consistently performs better than state-of-the-art hardware warp schedulers, delivering 28% speedup when phases are highly skewed and 15% improvement on attention configurations found in QWen3, Llama4, and Grok 1.0. RTL synthesis results indicate low hardware overhead to implement PAWS in silicon.

***Index Terms*—GPU microarchitecture, warp scheduling, attention kernels, transformers**

I. Introduction

GPUs have emerged as the most prevalent architecture running artificial intelligence. Designed initially around general-purpose *Single Instruction, Multiple Thread* (SIMT) cores for broad parallel workloads, AI's computational demands have reshaped GPU architecture. The rise of deep learning, where matrix multiplications dominate the training and inference of neural networks, has driven GPU vendors to move beyond generic parallelism and integrate specialized accelerators. Modern GPUs now feature bulk, asynchronous co-processors such as *Tensor Cores* [1], and *Tensor Memory Accelerators* (TMA) [1]. Although these new units boost performance, they have also altered the GPU programming model and the characteristics of optimized workloads [2], [3].

Large language models (LLMs) are the dominant workload executing on contemporary GPUs, mainly due to their immense computational requirements and widespread adoption in enterprise and consumer AI applications [4]–[6]. Within these models, the attention layer is the most computationally expensive component [7], [8]. To maximize throughput and efficiency, state-of-the-art implementations fuse *General Matrix Multiply* (GEMM) operations with their dependent element-wise and reduction operations, minimizing memory traffic and exploiting data locality [2], [3]. These implementations create software pipelines within the kernel to overlap the execution of the TMA, Tensor Cores, and SIMT units [1], [9]. Figure 1 shows the conceptual dataflow of the attention layer, which consists of steps that stress different hardware units of the GPU *Streaming Multiprocessor* (SM). The key to achieving high compute performance in the attention kernel is to efficiently overlap the execution of these steps, ensuring that the hardware units operate concurrently with minimal idle periods.

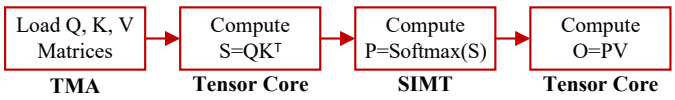


Fig. 1: Functional phases of the attention algorithm. Each phase stresses a specific hardware unit, but all instructions are issued from the same front end.

Optimized implementations of attention orchestrate this overlap by creating warps specialized for each stage of the pipeline [3], [10] and exploiting pipeline parallelism at the matrix tile granularity. Inter-stage dependencies are managed through hardware-accelerated synchronization mechanisms. This software architecture enables the hardware to automatically handle low-level overlapping of instructions across parallel warps through dynamic scheduling, while more effectively distributing in-core resources such as register file capacity[1] among computational tasks [12], [13]. These pipelined, uneven workloads fundamentally differ from traditional GPU programs, where computation is uniformly spread across the GPU's threads.

Although these advanced attention implementations generally achieve high performance, current GPU warp schedulers fail to optimally handle the aforementioned workload charac-

[1]Since the Hopper architecture, register file resources can be redistributed when warps perform distinct tasks that require varying amounts of registers [11].

teristics, leading to suboptimal hardware utilization and performance. Two primary issues arise from the uneven software pipelines. First, the workload in each program phase is often imbalanced, due to the nature of the attention algorithm, varying LLM configurations, and differing throughput of various hardware units. Second, instructions controlling the bottleneck functional unit (typically either the Special Function Unit, or SFU, for exponentiations in the softmax step, or the Tensor Core in the GEMM steps) are issued only by specific warps, which must compete with other warps for issue bandwidth.

The GPU's issue-level warp scheduler is responsible for selecting and issuing instructions from the SM's active warps. In any given cycle, the warp scheduler may have pending instructions from any of the four software pipeline stages shown in Figure 1. An even round-robin approach to issuing instructions from each warp paradoxically leads to unequal progress in each pipeline stage because the workload and hardware throughput across stages vary. This results in the longest stage running alone for extended periods where no other warp can cover stalls from structural or data hazards. Furthermore, the warp issuing instructions to the bottleneck hardware unit must compete with other warps for issue bandwidth, delaying issue of critical instructions and underutilizing bottleneck resources despite optimization efforts in the software. Contemporary hardware and existing academic work on warp scheduling [14]–[29] are oblivious to the emergent characteristics of optimized LLM implementations, resulting in the underutilization of the bottlenecking functional unit due to warp scheduling decisions.

We propose *Progress-Aware Warp Scheduling* (PAWS): a compiler-informed warp scheduler that dynamically balances software pipelines by prioritizing instruction-issue from warps that are further from their synchronization points. To our knowledge, this is the first warp scheduler that targets highly optimized, fused machine learning kernels that leverage warp specialization. PAWS introduces a compilation pass that augments GPU kernels with progress information at minimal machine code overhead, from which the warp scheduler can derive at runtime the predicted uncontested latency from every instruction to the end of the software pipeline stage it belongs. The most critical instructions in these optimized software pipelines usually have a fixed, deterministic latency. GPU compilers already encode the necessary wait times for deterministic instructions directly within each instruction [30]–[32] to reduce costly scoreboard entries in hardware. We extend this latency-insertion pass in the compiler to encode the predicted total cycles of each software pipeline stage at their beginnings. While this cycle count is a prediction since some kernels may have data-dependent control flow and long-latency memory accesses, control flow in optimized machine learning kernels is highly regular while data is accurately and timely prefetched. By embedding this information at the beginning of each stage, the warp scheduler can keep track of the remaining progress of each stage towards their ends, hence detect and prioritize lagging pipeline stages, allocating more of the GPU's valuable issue bandwidth to warps that are falling behind and ultimately improve utilization of the bottleneck hardware unit.

To demonstrate the impact issue bandwidth contention can have on GEMM-heavy, warp-specialized kernels, we perform a limit study using an optimized, warp-specialized CUTLASS kernel executing on a silicon H100 machine. By varying the number of instructions in "dummy" stress warps, co-located with the Tensor Core warps on the same SM, we demonstrate that even when the Tensor Core is the limiting stage in the software pipeline contemporary hardware suffers a 9% slowdown when Tensor Core warps are running alongside SIMT warps attempting to issue simple integer instructions.

Demonstrating the performance impact of warp scheduling in modern deep learning kernels, engineers at DeepSeek [33] reported more than a 10% performance improvement by modifying what they speculate is a yield flag in SASS instructions, allowing control to shift from one warp to another. PAWS builds on the idea by granting software more fine-grained control over warp scheduling, advancing beyond simple yielding to enable explicit prioritization when workloads are knowingly imbalanced. We believe the compiler/hardware scheduler co-design of PAWS opens the door for other interesting low-level optimizations for the compiler with more contextual information of the program in the era of specialized hardware and sophisticated programming paradigms.

This paper makes the following contributions:

1) We perform the first quantitative analysis on the impact of warp scheduling in the context of highly optimized, fused machine learning kernels that leverage warp specialization.
2) We demonstrate the impact of instruction issue-level contention on Tensor Core performance in contemporary H100 hardware.
3) We propose PAWS, a novel, compiler-informed hardware scheduler that dynamically balances asymmetric software pipelines by prioritizing warps further from synchronization points. PAWS effectively limits the skew between when warps reach their synchronization points, limiting the time hardware resources sit underutilized.

We evaluate the effectiveness of PAWS across a broad cross-section of 1800 different attention implementations that represent both contemporary and forward-looking attention kernels with different context lengths, number of attention heads, and head dimensions. PAWS demonstrates consistently better performance than state-of-the-art hardware schedulers across these points: a 28% performance improvement when the work in each phase is highly skewed, and a 15% improvement on contemporary attention configurations found in QWen3 [34], Llama4 Scout [35], and Grok 1.0 [36].

II. Background

GPUs execute programs with massive thread parallelism. Threads are statically bundled into collections of 32 or 64 threads called "warps" or "wavefronts". At any clock cycle, all active threads in a warp execute an instruction at the same program counter. Each warp issues and executes instructions in program order, with potentially many cycles of

delay between consecutive instructions due to data or control hazards. The compiler extensively annotates the machine code to embed dependency information [30]–[32], [37], and cycle-level timing information for fixed-latency instructions, thus simplifying the hazard-detection circuits. GPUs leverage the warp scheduler to perform dynamic scheduling among active warps [38] to compensate for the lack of intra-warp instruction-level parallelism. The compiler may also annotate some long-latency instructions with a "yield" hint to encourage the warp scheduler to schedule another warp [30], [33], [37].

Two basic common warp scheduling policies are Loose Round Robin (LRR) and Greedy-then-Oldest (GTO) [14]. LRR issues one instruction for a warp, then switches to the next ready warp in round robin order; it shines when equal progress among warps is desired. GTO aggressively issues instructions from one warp until it stalls then switches to the oldest ready warp; it is good at staggering load requests from warps thus preventing a stampede to the memory system, it also improves L1 data cache hit rate when warps have high memory locality but the cache can only contain the working set for a subset of active warps at a time. A number of advanced warp scheduling mechanisms are reviewed in Section VII.

Modern GPU architectures have various CISC-style asynchronous units like the TMA and Tensor Core to speed up deep learning workloads. These units typically serve the compute/memory needs of a large group of threads at once, therefore a single warp per Cooperative Thread Array (CTA) suffices to program each. Warps that control these units tend to have a different resource demand compared with other warps, and warp specialization allows individual warps to release or acquire hardware resources to suit their specific needs [11]–[13].

The attention mechanism [7] is a computationally demanding part of modern LLMs. Functionally, it takes the output of one GEMM operation ($S = QK^\intercal$), applies row-wise softmax ($P = \text{Softmax}(S)$), then performs another GEMM with the result ($O = PV$). The S and P matrices are often much larger than the Q, K, V matrices, therefore naive implementations that serially compute the three steps incur huge memory traffic [2] by writing S and P to memory. Optimized kernels [3], [10], [39]–[41] for attention fuse the three operations in one kernel and save S and P from being written to memory. The fused kernel has the added advantage of improving core utilization through fine-grained overlapping of GEMM and softmax operations. As we will show in Section V-C, the ratio of GEMM and softmax work is directly proportional to the attention head dimension but independent of context length and head count.

III. Motivation

In this section, we start with a simple intuition about the inefficiencies of fair round robin scheduling in attention kernels, introduce polar schedulers, then use three experiments conducted on real GPU and simulator to back our intuition and demonstrate the effect of warp scheduling on an imbalanced software pipeline. The polar scheduler is a simple experimental concept instrumental to the development and evaluation of our proposed technique.

A. Fair Round Robin Scheduling can be Suboptimal

Modern ML kernels commonly use warp specialization to decompose a complex program into a software pipeline of several stages, each of which has distinct characteristics. For example, Flash Attention 3 and 4 [3], [39] have specialized warps for each of the data movement, GEMM, and Softmax operations; they each stress the TMA, Tensor Core, and SIMT pipeline which have drastically different single-instruction latencies. The total work in each warp can also be unequal. Here, work refers to the amount of absolute time spent processing a matrix tile, or pipeline task.

As shown in Figure 2a, warp-specialized programs commonly use FIFO queues managed by semaphores [42] to move data between warps[2]. Semaphores impose synchronization barriers among warps, and as a result, warps tend to exhibit periodic idleness due to the imbalance of work among software pipeline stages. Given the changing nature of software algorithms, this imbalance is common because it is difficult to code a program that perfectly balances the work among pipeline stages. The difficulty is only exacerbated with the advent of specialized hardware units (e.g., TMA and Tensor Core) programmed by CISC-style instructions.

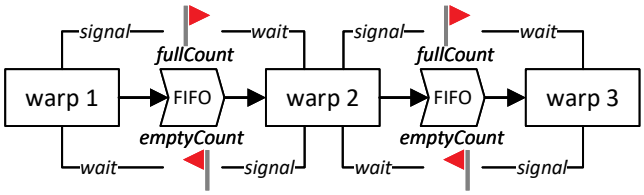
Warp schedulers designed with fairness in mind are not the best for warp-specialized programs. Figure 2b shows the dataflow of a hypothetical warp-specialized program of three stages, where the middle stage (mapped to warp 2) contains the greatest amount of work. With a fair round robin scheduler (Figure 2c), warp 2 equally shares issue slots with the other two warps, which prevents it from issuing ready instructions as early as possible in order to saturate its bottleneck hardware unit; after both warps 1 and 3 have finished, the scheduler will have no other warps to issue when warp 2 is stalled by structural or data hazards.

A simple first-order improvement is to always prioritize the slowest warp, as shown in Figure 2d. The scheduler will always issue instructions from warp 2 as soon as it is ready, and use the occasional stalling moments of warp 2 to issue instructions from other warps. This way, the slowest warp can potentially finish sooner, and the scheduler has a better chance at finding any warp to issue when the slowest warp is stalled.

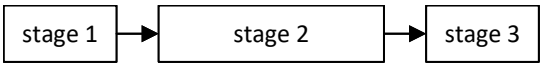
B. Polar Schedulers

For the rest of this paper, we use the term "polar scheduler" to describe schedulers that always prioritize a particular subset of warps, like the one we just described. For attention kernels mainly consisting of gemm and softmax warps, a polar scheduler could either be configured to prioritize the gemm warps or the softmax warps. To study the effects of polar schedulers, we modify Example 48 (Hopper Warp-Specialized

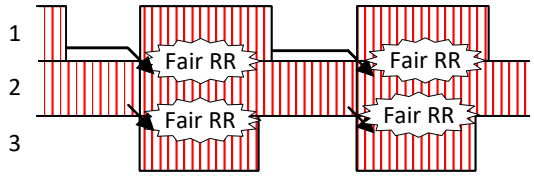
[2]The semaphore is an abstract concept that can be implemented by various hardware primitives: compare-and-swap loops, named barriers (PTX `bar.sync`), or mbarrier objects operated on by hardware-accelerated `mbarrier.arrive` / `mbarrier.test_wait` instructions.



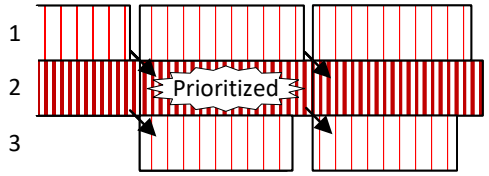
(a) A semaphore-based mechanism for implementing the three-stage pipeline in a warp-specialized program.



(b) Dataflow of an imbalanced three-stage software pipeline.



(c) Timeline of Fair Round Robin scheduling.



(d) Timeline when prioritizing the slowest warp.

Fig. 2: For a warp-specialized program with imbalanced work, prioritizing the longest warp improves the performance over fair round robin scheduling. The density of hatch lines represents issue bandwidth.

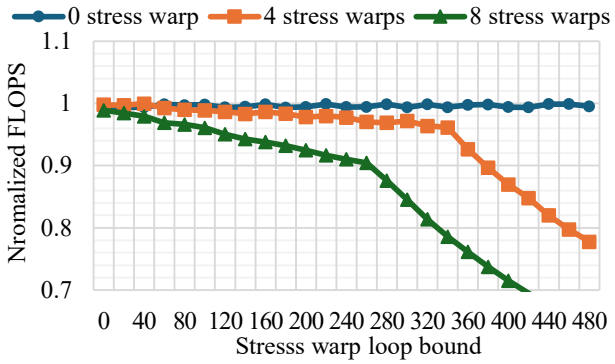


Fig. 3: Stress warps' effect on GEMM throughput. Measured on H100 using the Tensor Core warp specialized kernel from CUTLASS, augmented with stress warps.

GEMM) from CUTLASS [43]. The original program had eight warps, with warp 0 being the TMA warp, warps 1-3 unused, warps 4-7 being the MMA (Matrix Multiply Add) warpgroup[3]. The tensor core from Hopper requires a minimum of one warpgroup (four warps) to program. Because Hopper has four warp schedulers per SM, and the four warps of a warpgroup are guaranteed to bind to each of the four schedulers, CUTLASS Example 48 has one gemm warp per scheduler.

We fix the amount of GEMM workload and add 4 or 8 artificial stress warps per SM to Example 48 and parameter-sweep the amount of stress work done in each warp. Stress warps do not affect the functional correctness of the GEMM computation, but execute a long sequence of arithmetic instructions designed to create contention at the warp scheduler against the useful MMA warp. The stress instructions are purely integer operations and barely compete for functional units with the MMA warp. The pseudocode in Listing 1 sketches the structure of our modified program (we eschew the fast-running TMA warp for simplicity).

While this is a warp-specialized program, there exists no synchronization between the gemm and stress warps. However, we argue that the end-to-end behavior of this kernel is a representation of the period between two consecutive barriers from a real warp-specialized program, like the one shown in Figure 2c.

We study this program both by measuring its performance on real hardware and by simulating its trace on a GPU simulator.

Listing 1: Pseudocode of a stress warp program

```
def gpu_kernel(stress_warp_loop_bound):
  if warp_id < 4 :
    # GEMM warps use tensor core
    for (tile_A,tile_B,tile_C) in \
    make_tiles(mat_A,mat_B,mat_C):
      tile_C+=tile_A*tile_B
  else if warp_id >= 4:
    # Stress warps compute arithmetic nonsense
    for i in range(0, stress_bound):
      x = thread_id<<4
      y = cta_id>>4
      z = (x^0x1234)+(y^0x5678)
      ...
def main():
  # Launch kernel with various warp counts
  # and stress bounds
  gpu_kernel[warp_count= 8](stress=  0)
  gpu_kernel[warp_count=12](stress=  0)
  gpu_kernel[warp_count= 8](stress=100)
  gpu_kernel[warp_count=12](stress=100)
  ...
```

Figure 3 shows the normalized GEMM throughput measured on a real H100 device at different stress warp counts and stress loop bounds. Throughput slowly degrades as the stress level increases, and rapidly declines once stress is above a threshold. This threshold is the stress level where the stress warps start to overtake the MMA warps as the longest-running warps. By the time stress level reaches the threshold, GEMM throughput has degraded by 4% and 9% respectively, indicating the stress warps are stealing issue slots from the MMA warp at the warp scheduler.

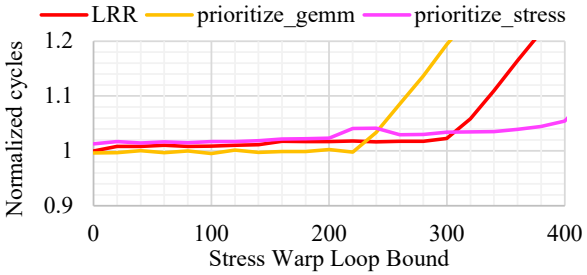
Figure 4 shows the normalized cycle count of kernel execution collected from simulation modeling an H100 in NVArchSim [45] at different stress warp counts and stress

[3]PTX [44] specifies "A warpgroup is a set of four contiguous warps such that the warp-rank of the first warp is a multiple of 4. Warp rank of a warp is defined as (%tid.x +%tid.y * %ntid.x + %tid.z * %ntid.x * %ntid.y) / 32."

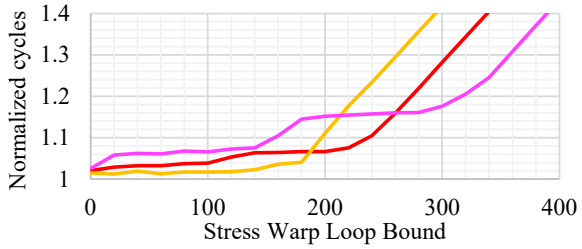
levels. We evaluate three different schedulers: a Loose Round Robin scheduler, a polar scheduler for stress warps ("prioritize_stress"), and a polar scheduler for the MMA warp ("prioritize_gemm").

The trend of performance change from the LRR curve is in line with real hardware data (a performance degradation of 2.5% and 8%, respectively, at threshold stress level), indicating an acceptable level of correlation. At low stress levels, the prioritize_gemm scheduler keeps the performance very close to the no-stress level, but shoots up earlier than the LRR scheduler as the stress level increases. At high stress levels above the threshold, the prioritize_stress scheduler outperforms the LRR scheduler by a margin of up to 20% and 15%, but it performs the worst among all three schedulers at low stress levels.

These findings support the intuition from Section III-A that the polar scheduler can outperform a fair round robin scheduler when the pipeline is imbalanced. However, we need a more adaptable mechanism than the polar scheduler to optimally cover the wide range of software pipeline imbalances that occur in various LLM configurations.



(a) 4 stress warps.



(b) 8 stress warps.

Fig. 4: Effect of warp scheduling policy and stress level. Measured in simulation using NVArchSim [45] modeling H100.

C. Evidence of Warp Progress Imbalance in Real Attention Kernels

To measure warp imbalance directly, we instrument the CUTLASS optimized FMHA kernel at 32k context length on Blackwell: we bracket every call site of a blocking barrier wait with a `clock64` pair, and accumulate the measured latency in a CTA-resident buffer indexed by warp role and barrier name.

We evaluate two precision and head-dimension configurations: (fp16, d=128) and (fp8, d=32). The d=128 setting matches most production LLMs (Qwen, LLaMA, Grok, Mistral). The two configurations stress the tensor core and SFU respectively: MMA latency scales with FP precision while SFU latency is constant, and the ratio of MMA to SFU instructions scales linearly with head dimension (Section V-C). Table I reports the mean cycles per blocking barrier handshake at each warp's output slot.

Warp	fp16 d=128	fp8 d=32	ratio
Softmax	1318	507	0.38×
MMA	312	605	1.94×

TABLE I: Mean cycles per blocking barrier handshake at the output slot of each warp. Each row averages over the relevant call sites in the warp-specialized mainloop. Going from GEMM-dominant (fp16 d=128) to softmax-dominant (fp8 d=32), the dominant wait moves from the Softmax warp (1318 → 507 cycles) to the MMA warp (312 → 605 cycles).

GEMM-dominant regime (fp16 d=128). The Softmax warp spends 1318 cycles per call waiting for its downstream output-queue slot to free, 4.2× the wait of the MMA warp (312). This is the silicon manifestation of "MMA dominates and the softmax warp waits more", the pattern predicted by the round-robin-vs-polar simulation in Section III-B.

Softmax-dominant regime (fp8 d=32). The direction reverses. The MMA warp's wait nearly doubles to 605 cycles per call: each tensor-core MMA is fast at fp8/d=32, so MMA arrives at the next handshake before softmax has released the previous FIFO slot. Symmetrically, the Softmax warp's wait drops to 507 cycles (2.6× less), because softmax is now slower and no longer blocked on data.

To conclude, the warp on the slower side of each producer-consumer queue spends extra time blocked at the barrier, and which warp that is shifts predictably with the GEMM-to-softmax work ratio. A scheduler that knows which side of each handshake is slower can shorten the waits Table I measures by prioritizing the bottleneck warp.

IV. Progress-Aware Warp Scheduling

We propose Progress-Aware Warp Scheduling to improve software pipeline imbalance in contemporary machine learning workloads. In Sections IV-A and IV-B, we take concurrent, synchronous program regions of specialized warps, and annotate their instructions with a monotonically decreasing sequence of natural numbers ("progress bits") indicating the expected execution time to finish the task (i.e., matrix tile) in each software pipeline stage; additionally, we use a dynamically incrementing counter ("task id") to signify the current loop count. In Section IV-C we describe the hardware and software changes, and the warp sorting logic of PAWS. In Section IV-D, we describe compiler support for automatically identifing task boundaries and assigning task id and progress bits.

A. Instruction Delay Bits (Background)

NVIDIA's fixed-length SASS instruction is long (128 bits in Blackwell and Hopper) and contains rich control information. Though not officially documented, explanations for the layout and purpose of SASS control bits can be found from public sources [37], [46].

One of the control bit fields is the "stall count", which specifies the number of cycles a warp has to wait before it can issue the next instruction. The stall count can be as low as 1, meaning the warp can issue every cycle; it can be higher when data- or structural hazards exist among instructions. For fixed-latency instructions, the stall count can be statically determined at compile time. Therefore, embedding this information in the instruction simplifies the scoreboarding hardware for managing register dependencies. The "yield" flag that we mention in Section I used by DeepSeek is another member of the control bits.

For the rest of this paper, we use the term "delay bits", or D, to refer to the stall count of a SASS instruction.

Though not a contribution of this paper, we informally describe how the stall count can be calculated given the dataflow of instructions and the functional unit latencies. Figure 5a shows the program order of five instructions, the minimum stall count is 1 to satisfy the constraint of in-order single issue. Figure 5b shows the dataflow, where numbers on the edges represent the time lapse from the issue of the source instruction to the cycle when the result is available, so this number incorporates the functional unit latency plus the data forwarding latency. To determine the final assignment of stall counts, think of a single-chain weighted graph (as shown in Figure 5c) where the weight of each edge is assigned such that the path distance between any two nodes in this chain is equal to the longest path distance between these two nodes in both the control flow diagram and dataflow diagram.

B. Progress Bits and Task Id

We define "task" as a sequence of instructions $(I_1, I_2, \ldots, I_N)$. We define "progress bits", or P, as the suffix sum of the delay bits D of instructions in the task, i.e., $P_i = \sum_{j=i}^{N} D_j, 1 \leq i \leq N$. For example, if D=[1,2,3,4], then P=[10,9,7,4]. The progress bits of a task is a monotonically decreasing sequence of natural numbers approaching zero, as shown in Figure 6a. By annotating the static instructions of each pipeline stage with progress bits, as shown in Figure 6b, the warp scheduler can make scheduling decisions to attempt to let all warps reach the end of their tasks at the same time, or at least narrow the gap between their completion times. Boxed numbers in Figure 6c represent progress bits; PAWS prioritizes the warp with the highest progress bits value at any time.

Circled numbers in Figure 6c represent task ids. They help the scheduler correctly identify the slowest warp when the software pipeline has not reached steady state. Before reaching steady state, software stages have different task throughputs because FIFOs absorb the throughput difference. Figure 7 shows one example of a fast producer and slow consumer. Without considering the task id, the warp scheduler will sometimes prioritize the producer because the producer resets its progress bit to the highest value after each task, as in the cross-hatched area.

C. Extending the Warp Scheduler and Instruction Set

The hardware changes required for the warp scheduler are shown in Figure 8. We propose a table indexed by warp id, with each entry tracking the task id and progress bits of a warp. Whenever a warp issues an instruction, the scheduler updates the corresponding entry with the progress and task information from the issued instruction. Recent GPU architectures support up to 16 warps per scheduler [47], [48], so with 20 bits each for the task id and progress bits, our storage overhead per scheduler is $16 \times (20 + 20) = 640$ bits. When sorting warps, warps with a lower task id are prioritized; between warps of the same task id, the warp with the higher progress bits value (meaning it has more remaining work to finish) is prioritized.

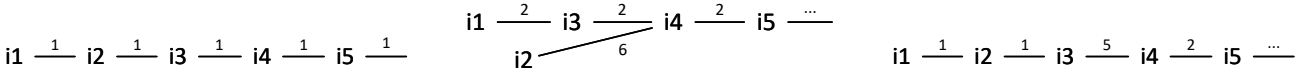
On the software side, the instruction set extension is straightforward and minimal. Conceptually and naively, each instruction should contain a new field encoding the progress bits, but note that one can use chained base-delta compression to store the change of progress bits instead of the progress bits themselves. Since the pre-existing delay bits D are the negation of the forward difference of the progress bits sequence: $D_i = -(P_{i+1} - P_i)$, it is possible only to make a one-time declaration of the first value of the progress bits sequence at the beginning of a task, then use an accumulator to compute the progress bits on the fly by subtracting the delay bits of each instruction. As a result, existing SASS instruction need not be modified, and one new instruction `declare_task_delay` is requisite at the beginning of each task to declare the end-to-end task delay.

We propose two more instructions for managing the task id: warps may use `set_task_id` to explicitly set it at the beginning of a program, and use `inc_task_id` to increment it after each loop iteration. How these instructions update the warp scheduler table is shown in Figure 8. Although not part of our proposal, these instructions alone could be used to set absolute warp priorities in which case PAWS degenerates to a polar scheduler.

D. Compiler Support

To demonstrate the role of compilers in our proposed mechanism, we use the example program in Figure 9. This program is equivalent to Figure 2a.

The method of determining the progress bits has already been described in Sections IV-A and IV-B; we will not repeat it here. How to determine the task id and demarcate task boundaries for each warp remains to be explained. We first identify the fullCount semaphores used in the program, specifically, `f_12` and `f_23` (①). They can be distinguished from the emptyCount semaphores by the fact that they are initialized to zero while the emptyCount semaphores are initialized to the buffer capacity. We then trace the direction of data flow among the warps by linking the signal-wait pairs

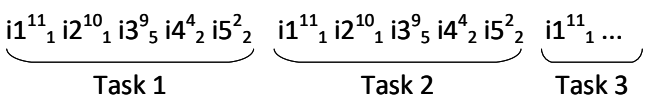


(a) Program order, with hardware issue latency on edges.

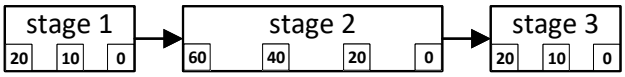
(b) Dataflow, with data latency on edges.

(c) Minimum delay to enforce at issue stage to prevent data hazards.

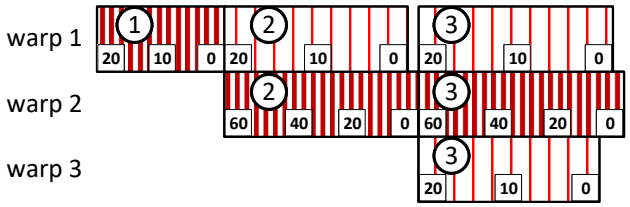
Fig. 5: Calculation of the compiler-inserted delay bits for instructions.



(a) Subscripts represent delay bits. Superscripts represent progress bits, that is, the remaining progress till end of task.



(b) Each software pipeline stage has a monotonically decreasing sequence of progress bits.



(c) Progress-aware scheduling. Boxed numbers represent progress bits. Circled numbers represent task ids. Density of hatch lines reflects issue bandwidth as a result of scheduling decisions.

Fig. 6: Visualizations of the compiler-inserted progress bits in the instruction stream and at runtime, spread across active warps.

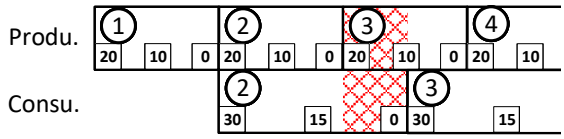


Fig. 7: Visualization of a producer/consumer software pipeline, before reaching steady state. The circled task ids tell PAWS to prioritize the consumer in the hatched period.

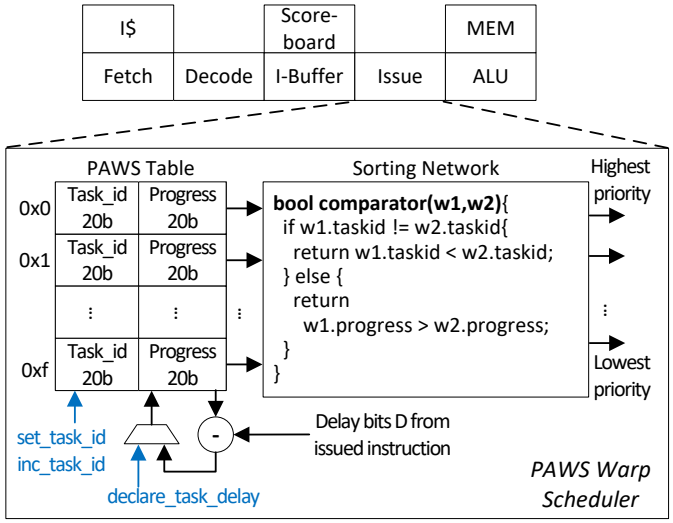


Fig. 8: PAWS extends the warp scheduler with a table for progress bits and task id, and a sorting network for determining warp priority.

of the semaphores. The pair ②③ links warp 1 and warp 2 as a producer-consumer pair. The pair ④⑤ links warp 2 and warp 3 as another producer-consumer pair. Together, this tells us the data flows from warp 1 to warp 2, then to warp 3, and from this we determine the initial task id of each warp: assuming the initial task id of warp 1 is `x`, the initial task id of warp 2 should be `x+1`, and the initial task id of warp 3 `x+2`. We insert the `set_task_id` instruction before the loop entry of each warp to initialize task id.

We assume the warp-specialized program is written such that each warp acquires and releases all necessary semaphores back-to-back, meaning all inter-warp synchronization activity happens within a small, contiguous region of code of each warp. This characteristic helps us determine the boundaries of each task. We insert the `inc_task_id` and `declare_task_delay` instructions immediately after the synchronization section of each warp (⑥⑦⑧).

Sketch of compiler pass. CUTLASS and Flash Attention's warp-specialized FMHA kernels implement semaphores using the mbarrier primitive and expose them through a four-function API [49]: producer-acquire, producer-commit, consumer-wait, and consumer-release, which respectively (i) waits for an empty-slot mbarrier, (ii) signals a full-slot mbarrier, (iii) waits for a full-slot mbarrier, and (iv) signals an empty-slot mbarrier[4]. Our pass therefore operates at the NVPTX intrinsic level, where the choice of implementation is exposed but the recognition table is small: any call lowering an mbarrier-arrive is a *signal*; any call lowering an mbarrier-test-wait or a `bar.sync` family wait is a *wait*. Listing 2 sketches the algorithm in Python-like syntax.

To evaluate the effectiveness of our complier pass, we implemented a python prototype that consumes PTX compiled from real workloads. We evaluated the pass on four attention implementations spanning two GPU generations: Dao's Flash Attention 3 implementation, a warp-specialized cutlass example, the attention kernel from FlashInfer (all from Hopper),

[4]Some instructions like those that control TMA data movement incorporate mbarirer intrinsics. Strictly speaking, the API set is larger than four functions.

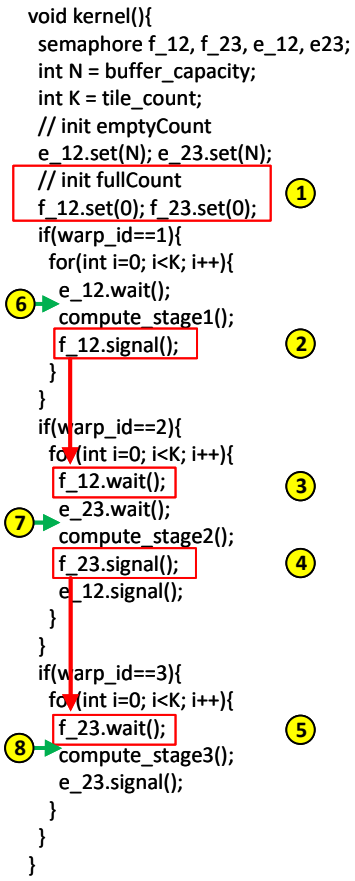


Fig. 9: Three-stage software pipeline with semaphores. This is a simplified illustration of the pipeline flow present in CUTLASS's warp specialized attention kernels.

and the Cutlass Optimized FHMA kernel descibed above (Blackwell). We cross-referenced the synchronization points identified by the pass with a visual inspection of the source code and validated that the producer/consumer structure of the warps are correctly identified.

V. Methodology

In this section, we describe the setup of our simulator and attention workload, then we characterize our workloads by highlighting that the attention head dimension is the determining factor of imbalance in the software pipeline of attention. We conclude with a mapping from the choice of attention parameters used in real LLMs to our evaluation space.

A. Simulator

We evaluate PAWS using a modified version of Accel-Sim [50] v1.3, a trace-driven GPU simulator modelling the Nvidia B200 GPU. We determine B200 functional unit (FU) latencies from microbenchmarks and validate their accuracy by correlating the simulator's performance with the performance of real hardware. Our modified version of the open-source Accel-Sim matches both the silicon performance trend in

Listing 2: PAWS compiler pass over lowered NVPTX IR.

```
# F: the device kernel after template instantiation,
#    inlining, and intrinsic lowering. Mutated in place
#    and returned, in standard compiler-pass style.
# SIG and WAIT are barrier intrinsic recognition tables.
# barrier_of(I) returns the mbarrier or named-barrier
#   object operated on by instruction I.
def run_paws_pass(F):
    # (1) Recover warp roles from the static
    #     warp_idx -> role dispatch.
    role_bb = find_warp_role_dispatch(F)

    # (2) For each role, collect signal / wait ops
    #     grouped by the barrier object.
    PROD, CONS = defaultdict(set), defaultdict(set)
    for role, bb in role_bb.items():
        for I in instructions(bb):
            if I in SIG:  PROD[barrier_of(I)].add(role)
            if I in WAIT: CONS[barrier_of(I)].add(role)

    # (3) Build the producer-to-consumer DAG of warp roles.
    G = DiGraph()
    for b in PROD.keys() | CONS.keys():
        for p in PROD[b]:
            for c in CONS[b]:
                G.add_edge(p, c, label=b)

    # (4) Sort the DAG to assign task ids.
    task_id = {r: i for i, r
               in enumerate(topo_sort(G))}

    # (5) For each role: emit set_task_id before
    #     entering the task loop; at the start of
    #     each iteration emit inc_task_id and
    #     declare_task_delay.
    for role, bb in role_bb.items():
        loop = task_loop(bb)
        emit_set_task_id(loop_preheader(loop),
                         task_id[role])
        T = sum(delay_bits(I) for I in loop_body(loop))
        emit_inc_task_id(loop_header(loop))
        emit_declare_task_delay(loop_header(loop), T)
    return F
```

Parameter	Value
Warp scheduling policy	PAWS/LRR/GRR/Polar
Issue bandwidth	1 instruction per scheduler per cycle
Issue to register-forwarding	FU latency + 2 cycles minimum delay
FU pool per scheduler	1 FP, 1 INT, 1 SFU, 1 BRA, 1 UDP
FU latency/initiation interval	FP=INT=4/2, BRA=UDP=4/4, SFU=23/8
TMEM read latency	4 cycles
Tensor core throughput	2048 fp16 MACs per cycle

TABLE II: SM-level configuration parameters.

Figure 3 and the NVArchSim results shown in Figure 4. Our setting for FU latencies aligns with what the delay bits from machine code indicate. Table II shows the SM-level configuration we use.

We evaluate PAWS against the following schedulers:

- Loose round robin (LRR): issues one instruction for a warp, then switches to the next ready warp in round robin order;
- Greedy round robin (GRR): issues instructions from one warp until it stalls then switches to the next ready warp in round robin order.
- Two polar schedulers (prioritize-softmax / prioritize-gemm): studied in Section III-B. Each cycle, the warp scheduler first sorts warps using the LRR policy, then performs a stable sort to move prioritized warps in front of other warps.
- Synchronization-Aware Warp Scheduling (SAWS) [22] and

Barrier-Aware Warp Scheduling (BAWS) [5] [29].

We also consider the greedy-then-oldest scheduler (GTO) and Crago's scheduler (WASP) [51], for which we use the schedulers mentioned above as proxies. GTO is similar to GRR but picks the oldest ready warp whenever it switches; it behaves almost identically to GRR in our experiments because only one CTA is active in each SM, meaning all warps are the same age. Crago et al. propose a pipeline-aware warp scheduler to complement their fine-grained register file queues for inter-warp data movement. It prioritizes warps whose input queue is full, followed by warps whose input queue is not empty, followed by warps executing earlier stages of the software pipeline. We reason that with oracle knowledge of which warp is the slowest, Crago's scheduler can be approximated by the better of the two polar schedulers on a workload-by-workload basis. We focus on scheduling and do not evaluate other orthogonal contributions from WASP.

B. Attention Workload

Contemporary deep learning kernels exist in a fast-paced world of evolving algorithms and iterating hardware architectures. The vastness of algorithm parameters, operator fusion opportunities, and device-specific performance tuning guides makes it difficult for GPU architects to build or maintain a holistic forward-looking library of top-performance kernel traces. Highly-regarded, well-optimized kernels only represent a few particular combinations of fusion and optimization choices, but leave a vast space of implementation possibilities untouched.

To support credible research into contemporary attention implementations and to explore configurations not yet developed in software, we construct an attention kernel generator based on NVIDIA's highly-optimized CUTLASS implementation of fused multi-head attention for Blackwell [10]. The implementation details of this kernel are very similar to the recently released Flash Attention 4 [39]. The kernel has tunable parameters to select the head count, context length, sequence length, and head dimension. We trace the fp16 version of the kernel at head count 64, context length 1k, sequence length 1k, head dimension 128, and validate that the simulator's performance estimates track B200 silicon on this particular kernel[6].

We use this trace to generate a wide sweep of 1800 SASS programs of different compute-bound attention configurations. At the sizes supported by the Blackwell CUTLASS kernel, we validate that our trace generation methodology produces the same instruction mix and performance as the unaltered kernel. Our sweep also covers configurations presently impossible or poorly optimized to run on the original CUTLASS kernel due to limitations in tile sizes, TMEM, SMEM, and register file allocations. Though a more nuanced software implementation (with different tile sizes or different variable allocations in TMEM, SMEM, or register files) could work well, that would require extensive software changes for every design point.

In Section V-C we will show the determinant of these configurations is the attention head dimension and in Section V-D we will describe the coverage of the generated traces.

Note that our Blackwell CUTLASS kernel makes use of all the deep CUDA optimizations that engineers on the NVIDIA CUTLASS team know how to exploit. This is not always the case with kernels implementing attention in open frameworks like PyTorch [52] and vLLM [53], where software takes time to catch up with hardware advances [2], [54]. For example, the attention kernels in vLLM do not make use of the TMA because they implement paged attention [53], [55]. Although it would be possible to use the TMA and paged attention together, software developers have not yet integrated such an implementation into vLLM, likely due to the complexity. We chose the CUTLASS kernel as our starting point since it represents a highly-informed implementation attempting to extract peak FLOPs from a Blackwell GPU, and our scheduler optimizations only make sense when the kernel has already been extensively optimized.

C. Head Dimension Determines the Softmax-to-GEMM Ratio

The key attention parameters that determine the amount of work in the gemm and softmax warps are attention head count (H), sequence length (N), context length (N_{ctx}), head dimension (d). For each attention head h ($0 \leq h < H$), the matrices involved in the attention computation are:

$$\begin{aligned}
\text{Query matrix } Q_h & \in \mathbb{R}^{N \times d} \\
\text{Key matrix } K_h & \in \mathbb{R}^{N_{ctx} \times d} \\
\text{Softmax score matrix } S_h = Q_h K_h^\intercal & \in \mathbb{R}^{N \times N_{ctx}} \\
\text{Softmax probability matrix } P_h = \text{softmax}(S_h) & \in \mathbb{R}^{N \times N_{ctx}} \\
\text{Value matrix } V_h & \in \mathbb{R}^{N_{ctx} \times d} \\
\text{Output matrix } O_h = P_h V_h & \in \mathbb{R}^{N \times d}
\end{aligned}$$

The number of key operations in each step is:

$S = QK^\intercal$	$(N \times N_{ctx} \times d)$ MACs
$P = \text{softmax}(S)$	$(N \times N_{ctx})$ exp-sums
$O = PV$	$(N \times N_{ctx} \times d)$ MACs

Therefore, the ratio of MACs to exp-sums is mainly dependent on the head dimension d but independent of the sequence and context lengths or the number of heads.

Attention variants like Multi-Query Attention [56], Grouped Query Attention [57], and Multi-head Latent Attention [58] are techniques to reduce the KV cache size; given fixed H, N, N_{ctx}, d, they reduce data movement and improve reuse of data, but the amount of MACs and exp-sums for each attention head remains the same.

Applying masks to the softmax score matrix S reduces the amount of computation, but state-of-the-art compilers aggressively track data dependencies, eliminate dead code, and propagate constants. The result is that the number of MACs and exp-sums are reduced by the same ratio; therefore,

[5]BAWS also proposes a synergistic instruction fetch unit which we optimistically approximate with ideal instruction fetch bandwidth.

[6]The simulator, traces, and trace generator are made public on `https://github.com/purdue-aalp/progress_aware_warp_scheduling`.

masking does not change the work ratio between gemm and softmax.

D. Effective Head Dimension of Generated Traces

We denote *sw* as the number of softmax warps, *gw* as the number of gemm warps, *st* as the number of softmax tiles per warp, and *gt* as the number of gemm tiles per warp. Each generated trace's warp imbalance characteristic reflects that of an attention kernel with a unique head dimension. Because the CUTLASS kernel has a head dimension of 128, the effective head dimension of the generated trace is: *effective head dimension = 128 * (gt * gw) / (st * sw)*. Our parameter sweep covers 1800 unique points from a space spanned by the Cartesian product of $st \in [5, 100]$, $sw \in \{1, 2\}$, $gt \in [10, 100]$, $gw = 1$. The effective head dimension ranges from 4 to 5120.

E. Position of Real Models in the Space of Generated Traces

In Table III, we examine the model parameters of some LLMs to put them in the space of our attention trace generator. The head dimension is what determines the gemm-to-softmax ratio of the attention kernel. Therefore, QWen3, Llama4 Scout, and Grok 1.0 have similar characteristics regarding gemm-softmax work imbalance. DeepSeek V2 has a gemm-to-softmax ratio 4 times higher than the rest.

	QWen3 [34]	Llama4 Scout [35]	DeepSeek V2 [58]	Grok 1.0 [36]
# of heads	16∼64	64	128	48
Context length	32k∼128k	10M	128k	8k
Head dimension	128	128	512	128

TABLE III: Model parameters of large language models

VI. Evaluation

We evaluate the space of attention kernels generated from 1800 different combinations to understand the nature of warp imbalance at various head dimensions and demonstrate the effectiveness of PAWS. Furthermore, we examine the hardware overhead and performance sensitivity of PAWS to the delay of the sorting network, and we use simulation to study the end-to-end speedup effect of PAWS on an LLM inference server.

A. Speedup

We plot the speedup of various schedulers against LRR into S-curves in Figure 10. PAWS outperforms all other schedulers on most workload parameters, reaching a maximum of 28% speedup. SAWS, BAWS, and GRR have almost the same curve for the following reason. SAWS and BAWS are designed to handle cases where multiple CTAs share the same scheduler: in SAWS, once a warp from any CTA hits a barrier, the warp scheduler deprioritizes warps from all other CTAs on the scheduler; BAWS' warp scheduler and instruction fetch unit prioritizes the CTA with the most warps waiting at a barrier. Unfortunately, SAWS and BAWS both degenerate to their baseline GTO scheduler because modern attention kernels launch only one CTA per SM. GTO behaves similarly to GRR because all warps of a CTA launch together and have the same age.

In Figure 11a, we subsample the evaluation space and focus on head dimensions $d = 64, 128, 256, 512$. First, notice that none of the schedulers except PAWS can deliver the highest performance on all head dimensions. Second, notice that the higher performance improvements come from configurations where three warps are scheduled. The explanation for the second observation is that warp scheduling becomes trivial at two warps per scheduler, so the performance of LRR is close to ideal. At three warps per scheduler, the choice of which warp to issue next becomes non-trivial when the scheduler decides to switch.

B. Spread of Warp Completion Time

To evaluate how effectively PAWS closes the gap between the fastest and slowest warps, we measure *spread fraction* in Figure 11b. *Spread fraction* is defined as the difference between the fastest and slowest warps' completion time, divided by the kernel completion time. We use the negative sign to indicate softmax warp(s) completed earlier than the gemm warp, and the positive sign otherwise. Of all the schedulers, PAWS has the lowest spread fraction on all subsampled head dimensions. This explains PAWS's consistent performance improvements across all configurations.

If we consider the spread fraction as a function of head dimension, parameterized by the warp count and choice of scheduler, the zero of this function in the LRR-scheduled 1-softmax-1-gemm case is important because it indicates the point where the gemm warp has the same amount of work as the softmax warp. Our parameter sweep indicates this zero is at head dimension ≈ 410. In the LRR-scheduled 2-softmax-1-gemm case, the zero is at head dimension ≈ 180, which is just 44% of 410. The reason why the zero has been reduced by more than 2x after adding a second softmax warp reveals the nature of fair scheduling: When the work of one softmax warp is distributed to two softmax warps, the two warps as a whole are given twice the number of issue opportunities; therefore, the gemm warp needs to reduce its work to complete at the same time with the two softmax warps. This effect is essential to consider when writing optimized kernels: adding more softmax warps not only potentially improves utilization of softmax-related functional units (e.g., the SFU for exponentiations), but also shifts the balance point of the head dimension for equal progress.

C. Analysis of Attention Kernels from LLMs

QWen3, Llama4 Scout, and Grok 1.0 have the same head dimension 128, below the balance point of 410 (180) for the two (one) softmax warps-per-scheduler configuration, so their tensor core is underutilized. With LRR scheduling, distributing the softmax work to two warps per scheduler reduces the spread fraction from 0.62 to 0.26 compared with the 1-softmax per scheduler configuration. However, with two softmax warps to schedule, LRR leaves 15% performance headroom for PAWS. DeepSeek V2 has a head dimension of 512 which

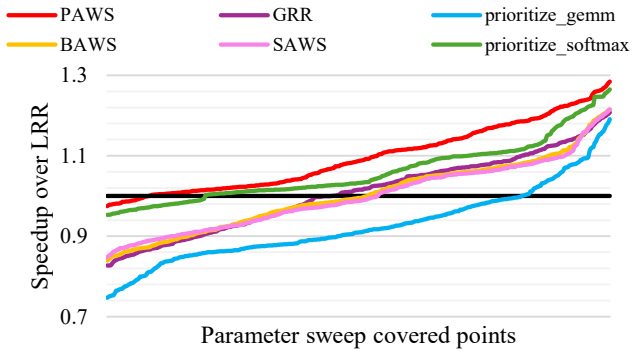
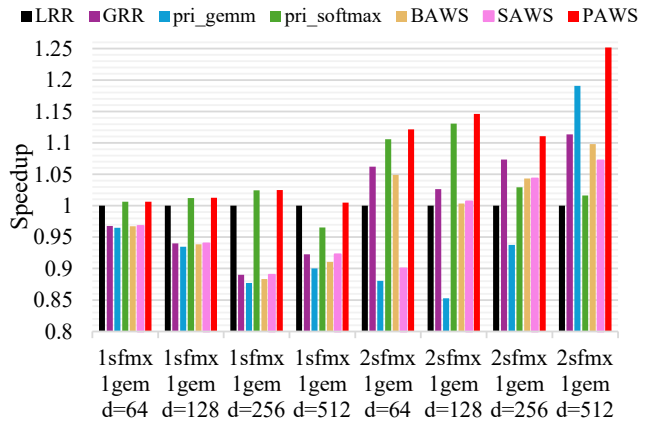
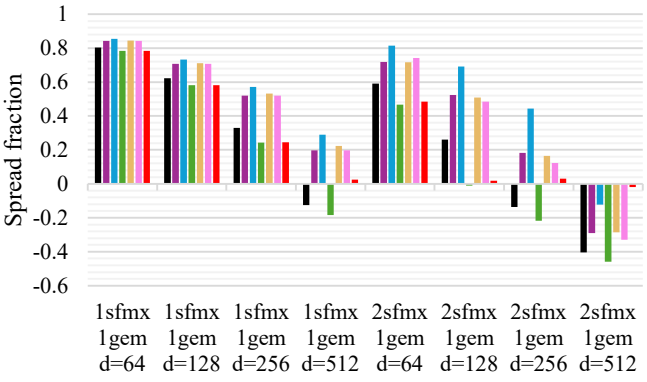


Fig. 10: S-curves of various schedulers' speedup compared with LRR scheduling on the parameter sweep across 1800 different attention configurations.



(a) Normalized speedup.



(b) Spread fraction.

Fig. 11: Comparison of schedulers using specific design points in the attention configuration space.

is close to the balance point of 410 in the 1-softmax-1-gemm case with a spread fraction of just -0.12. DeepSeek's open source FlashMLA kernel [40], written for Hopper, indeed uses an equal amount of warps for gemm and softmax, indicating a well-balanced software pipeline. PAWS closes the spread fraction to 0.02 but delivers minimal speedup over LRR.

D. PAWS Effectively Minimizes Spread

Figure 12 shows a scatter plot of speedup versus spread fraction at all 1800 data points. PAWS effectively reduces the gap between the fastest and slowest warp and unlocks performance. While other schedulers also deliver some speedup, they are ineffective at reducing the spread fraction. Worse, the two polar schedulers can "oversteer" by giving too much priority to the slowest warp and cause the spread fraction to flip sign. This is evident from the abundance of prioritize_softmax and prioritize_gemm points in the 3rd and 4th quadrants.

Figure 13 shows a conclusive picture of PAWS's effectiveness. We bin each sample point from our parameter sweep of 1800 points to the nearest head dimension that is a multiple of 128, and calculate the average spread fraction at each head dimension. All schedulers except PAWS cross the x-axis without stopping, but PAWS is able to maintain a very low spread fraction between the head dimension range of 384 and 896 and converge with the better of the two polar schedulers in the two extreme cases. Our evaluation demonstrates that PAWS is effective in mitigating the performance issues caused by an even scheduler on imbalanced workloads: PAWS maximizes thread level parallelism and minimizes warp spread by closely matching the progress of warps, and gives priority to warps issuing critical instructions that leverage bottleneck functional units.

E. Hardware Overhead

We extend Vortex [59], a verilog implementation of a GPU, with PAWS' table for progress bits and task id and a sorting network for comparing warp priority (shown in Figure 8), and we functionally validate the design. We synthesize Vortex with a 15nm process design kit [60] using Cadence Genus at 1 GHz. Synthesis results show less than 0.1% gate overhead and 2.5% flip-flop overhead over the default baseline configuration. Considering Vortex is a basic architecture, we expect the hardware overhead to be smaller on more advanced commercial designs.

PAWS uses a bitonic sorting network of depth 10 to order sixteen warps. To evaluate the sensitivity of performance to the delay of a multi-cycle implementation of the sorting network, we simulate an attention kernel in Accel-Sim where we only refresh the sorted warp order every N cycles and hold the warp order constant during the gap. In Table IV, we sweep $N \in \{1, 2, 4, 8\}$. PAWS' speedup over GRR stays between 12.99% (N=8) and 13.08% (N=2). Therefore, PAWS' gain is insensitive to the latency of the sorting network.

N	Speedup over GRR
1	13.07%
2	13.08%
4	13.05%
8	12.99%

TABLE IV: PAWS' speedup over GRR with respect to different sorting-network refresh periods N.

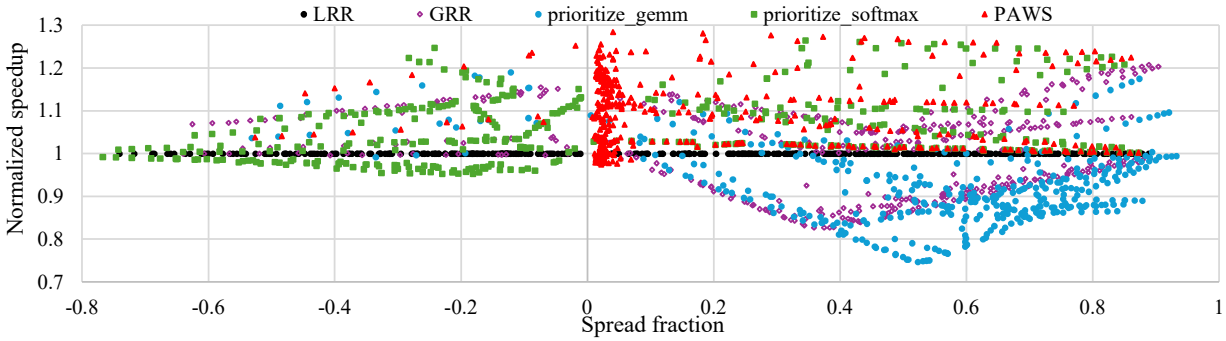


Fig. 12: A scatter plot of speedup versus spread fraction, collected from 1800 attention configurations.

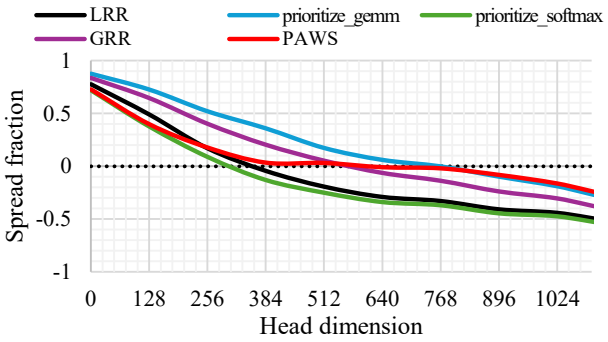


Fig. 13: Average spread fraction of various schedulers on different head dimensions.

	32k input	64k input	128k input
Share of compute time spent in:			
Prefill + mixed iters	60.4%	78.9%	91.9%
Decode-only iters	39.6%	21.1%	8.1%
Attention's share within:			
Prefill + mixed iters	61.4%	67.9%	72.1%
Decode-only iters	40.5%	47.2%	50.4%
Overall attention share	**37%+16% = 53%**	**54%+10% = 64%**	**66%+4% = 70%**

TABLE V: Attention's contribution to GPU compute time as input context length grows.

F. End-to-End LLM Serving Impact

To establish that PAWS' kernel-level speedups translate into system-level gains, we evaluate PAWS in an end-to-end serving setting using LLMServingSim 2.0 [61], [62]. We simulate Qwen3 30B-A3B [34] on a single MoE node (TP=2, EP=2), serving ten requests with 128k input tokens and 512 output tokens each, arriving one request per second. The simulator runs vLLM-style continuous batching with chunked prefill at a 2048-token per-iteration budget.

Table V decomposes attention's share of GPU compute time at three input context lengths (32k, 64k, 128k) into (i) how the system splits time between prefill/mixed and decode-only iterations and (ii) attention's share within each phase.

Two trends compound. First, longer prompts enlarge the percentage of time spent in prefill/mixed iterations. Note the fraction of compute time spent in prefill or chunked-prefill iterations rises from 60% at 32k input to 92% at 128k. Second, attention's share within each phase also climbs, from 61% to 72% in prefill/mixed iterations and from 41% to 50% in decode-only iterations. The product pushes attention's overall share of GPU compute from 53% to 70% as input context length quadruples.

We isolate two very different attention populations at 128k context length. Prefill and mixed iterations are 91.9% of compute time with attention 72.1% within, so *prefill/mixed attention consumes* $\approx$66% *of total GPU compute time* ($0.919\times 0.721 = 0.663$); decode-only iterations are 8.1% with attention 50.4% within, so *decode attention is only* $\approx$4% ($0.081 \times 0.504 = 0.041$). The two populations also differ in their amenability to PAWS: long-input prefill attention is compute-bound on the FlashAttention kernel, therefore it stresses the issue stage warp scheduler and benefits from PAWS, while decode attention at small batch is memory-bound on KV-cache traffic and insensitive to warp scheduling decisions. We therefore evaluate PAWS' end-to-end impact by speeding up only prefill attention (including the prefill share of mixed iterations), leaving decode attention at $1.0\times$.

Table VI reports the resulting time-to-first-token (TTFT), time-per-output-token (TPOT), and inter-token latency (ITL) at 128k context length as we conduct a parameter sweep of the speedup factor of prefill-attention on LLMServingSim. A $1.25\times$ prefill-attention speedup reduces mean TTFT by 10.1%, p99 TTFT by 10.4%, mean TPOT by 9.1%, and mean ITL by 9.1%; even a $1.05\times$ speedup yields a $\sim$2.5% reduction across all four metrics. End-to-end gains track the prefill-attention multiplier closely, confirming that PAWS' kernel-level improvements pass directly into user-visible latency.

VII. Related Work

Crago et al. propose WASP [51] to target warp-specialized programs. WASP introduces ISA-visible pipeline stage IDs and register-backed FIFO queues for fine-grained data transfer. WASP prioritizes warps by input queue fullness. In contrast to WASP, PAWS does not require changes in program structure

Prefill phase attention speedup	mean TTFT (s)	p99 TTFT (s)	mean TPOT (ms)	mean ITL (ms)
1.00×	62	118	130	130
1.05×	61 (-3%)	115 (-3%)	127 (-2%)	127 (-2%)
1.11×	59 (-5%)	112 (-5%)	124 (-5%)	124 (-5%)
1.18×	58 (-8%)	109 (-8%)	121 (-7%)	121 (-7%)
1.25×	56 (-10%)	106 (-10%)	118 (-9%)	118 (-9%)

TABLE VI: End-to-end serving metrics on the 128k-context workload.

and can operate independently of other hardware features; compiler-inserted hints also give PAWS a head start at taking action to balance warp progress. CAWA [63]'s scheduling is driven by dynamic profiling that focuses on branch divergence and memory-centric statistics. Divergence-free, compute-heavy TMA + Tensor Cores software does not suffer from the metrics CAWA monitors, which we have validated via silicon profiling. As a result, CAWA matches the performance of the baseline scheduler. Warp-Level Divergence [26] exploits the fact that some warps in the CTA complete before others. However, optimized ML kernels (Flash Attention 3/4, etc.) use persistent threads, launching just enough CTAs to saturate the GPU's resources in one "wave". These kernels have no yet-to-be-launched CTAs or warps. On optimized attention kernels Reference [26]'s performance will match the baseline scheduler. SAWS [22] and BAWS [29] are both synchronization-aware, but specifically address the case where multiple independent CTAs share one warp scheduler. Both degenerate to baseline scheduler on workloads where each SM (and warp scheduler) only runs warps from one CTA, which is the case for optimized attention kernels. BOWS [28] deprioritizes warps executing busy-wait loops for synchronization, which is today largely mitigated by hardware-accelerated barrier instructions commonly used in deep learning kernels. VISTA [25] proposes a CTA scheduler targeting inter-CTA memory locality, the utility of which is largely realized by the software-based tile schedulers in modern GEMM and attention kernels [64]. PRO [23] defines "progress" to be the sum of thread-instructions executed by a warp or CTA, and uses this metric to sort warps or CTAs for priority. OWL [27] divides co-scheduled CTAs into small groups and prioritizes a single group to overlap long latency memory transfers and improve L1 cache hit rate. CCWS [14] improves L1 cache hit rate by limiting the number of warps actively scheduled. All schedulers above do not apply to contemporary deep learning kernels where most memory system challenges are mitigated, specialized warps execute unique programs, and the number of CTA per SM is often limited to one.

Several papers studied CPU thread scheduling to mitigate barrier synchronization imbalance on Chip Multiprocessors (CMPs). Thrifty Barrier [65] identifies threads arriving early at a barrier, predicts their stall time, and puts the processor in a low-power state to save energy. Meeting Points [66] targets multithread programs, uses the execution count of the parallel for-loop's back edge to gauge the relative progress, and applies DVFS to speed up or slow down individual cores, and gives higher priority to the critical thread at the issue queue in the SMT core pipeline. Thread Criticality Predictors [67] uses L1 and L2 cache miss counts as criticality metrics to direct the task-stealing mechanism of a parallel runtime system and apply DVFS to rebalance the clock rate of threads. These CPU-world solutions do not directly apply to the GPU. The imbalance among GPU warps happens within the SM in GPU ML kernels. Both the small circuit size of sub-SM units and the high frequency of synchronization barriers make DVFS infeasible, not to mention circuit modules in the GPU already use clock gating to save power. Work stealing is possible but limited in utility due to the overhead of transferring contexts among warps, and the software complexity of potentially having to use a different program to complete stolen work (e.g., using the SIMT pipeline to compute outstanding GEMM work intended for the Tensor Core).

VIII. Conclusion

This paper presents Progress-Aware Warp Scheduling (PAWS), a novel warp scheduling mechanism that adapts to modern attention kernels by encoding priority information in the instruction stream. Our approach enables the warp scheduler to identify and prioritize warps executing slower program phases, thereby improving overall pipeline throughput.

The key insight of our work is that modern GPU workloads exhibit significant phase imbalance due to specialized hardware units and sophisticated programming paradigms. By making the scheduler aware of these phases and prioritizing instruction streams targeting the bottleneck functional unit, we can achieve better resource utilization and higher throughput than state-of-the-art scheduling approaches.

The implementation of PAWS requires minimal changes to existing GPU hardware and program, making it a practical solution for improving performance in real-world deep learning applications. We believe this work represents an important step toward more intelligent GPU scheduling that can adapt to the specific characteristics of modern workloads.

Our experimental results demonstrate that PAWS significantly improves throughput for attention kernels, with up to 28% better performance than traditional round-robin scheduling.

IX. Acknowledgments

This work was supported, in part, by NSF CCF #1943379 and NSF CCF #2450447. We thank our colleagues and reviewers for their feedback and suggestions on this work.

References

[1] NVIDIA, "Nvidia h100 tensor core gpu architecture," 2022, white Paper, V1.04, available at https://resources.nvidia.com/en-us-hopper-architecture/nvidia-h100-tensor-c.

[2] T. Dao, D. Y. Fu, S. Ermon, A. Rudra, and C. Ré, "Flashattention: Fast and memory-efficient exact attention with io-awareness," *arXiv preprint arXiv:2205.14135*, 2022. [Online]. Available: https://arxiv.org/abs/2205.14135

[3] J. Shah, G. Bikshandi, Y. Zhang, V. Thakkar, P. Ramani, and T. Dao, "Flashattention-3: Fast and accurate attention with asynchrony and low-precision," *Advances in Neural Information Processing Systems*, vol. 37, pp. 68 658–68 685, 2024.

[4] M. Shoeybi, M. Patwary, R. Puri, P. LeGresley, J. Casper, and B. Catanzaro, "Megatron-lm: Training multi-billion parameter language models using model parallelism," *arXiv preprint arXiv:1909.08053*, 2019.

[5] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei, "Language models are few-shot learners," *Advances in neural information processing systems*, vol. 33, pp. 1877–1901, 2020.

[6] S. Rajbhandari, J. Rasley, O. Ruwase, and Y. He, "Zero: Memory optimizations toward training trillion parameter models," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2020, pp. 1–16.

[7] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," in *Advances in neural information processing systems*, 2017, pp. 5998–6008.

[8] D. Narayanan, M. Shoeybi, J. Casper, P. LeGresley, M. Patwary, V. A. Korthikanti, D. Vainbrand, P. Kashinkunti, J. Bernauer, B. Catanzaro, A. Phanishayee, and M. Zaharia, "Efficient large-scale language model training on gpu clusters using megatron-lm," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '21)*. ACM, 2021. [Online]. Available: https://arxiv.org/abs/2104.04473

[9] NVIDIA, "Nvidia blackwell architecture technical brief," 2024, technical Brief, V2.1, available at https://resources.nvidia.com/en-us-blackwell-architecture/blackwell-architecture-technical-brief.

[10] ——, "CUTLASS example 77: Blackwell fused multi-head attention (FMHA) kernels," GitHub repository, 2025, available at https://github.com/NVIDIA/cutlass/tree/main/examples/77_blackwell_fmha/.

[11] ——, "Ptx instruction: setmaxnreg – dynamic reassignment of register capacity among warp-groups," NVIDIA PTX ISA Documentation, 2024, available at https://docs.nvidia.com/cuda/parallel-thread-execution/index.html#miscellaneous-instructions-setmaxnreg.

[12] M. Bauer, H. Cook, and B. Khailany, "Cudadma: optimizing gpu memory bandwidth via warp specialization," in *Proceedings of 2011 international conference for high performance computing, networking, storage and analysis*, 2011, pp. 1–11.

[13] M. Bauer, S. Treichler, and A. Aiken, "Singe: Leveraging warp specialization for high performance on gpus," in *Proceedings of the 19th ACM SIGPLAN symposium on Principles and practice of parallel programming*, 2014, pp. 119–130.

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

[15] ——, "Divergence-aware warp scheduling," in *46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-46)*, 2013, pp. 99–110.

[16] A. Barnes, F. Shen, and T. G. Rogers, "Mitigating gpu core partitioning performance effects," in *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2023, pp. 530–542.

[17] M. Huzaifa, J. Alsop, A. Mahmoud, G. Salvador, M. D. Sinclair, and S. V. Adve, "Inter-kernel reuse-aware thread block scheduling," *ACM Transactions on Architecture and Code Optimization*, vol. 17, no. 3, pp. 24:1–24:27, 2020.

[18] D. Tripathy, A. Abdolrashidi, L. N. Bhuyan, L. Zhou, and D. Wong, "Paver: Locality graph-based thread block scheduling for gpus," *ACM Transactions on Architecture and Code Optimization*, vol. 18, no. 3, pp. 32:1–32:26, 2021.

[19] B. Pichai, L. Hsu, and A. Bhattacharjee, "Architectural support for address translation on gpus: Designing memory management units for cpu/gpus with unified address spaces," in *19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2014, pp. 743–758.

[20] J. Zhang, S. Gao, N. S. Kim, and M. Jung, "Ciao: Cache interference-aware throughput-oriented architecture and scheduling for gpus," in *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2018, pp. 149–159.

[21] 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 Techniques (PACT)*, 2014, pp. 175–186.

[22] J. Liu, J. Yang, and R. Melhem, "Saws: Synchronization aware gpgpu warp scheduling for multiple independent warp schedulers," in *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2015, pp. 383–394.

[23] J. Anantpur and R. Govindarajan, "Pro: Progress aware gpu warp scheduling algorithm," in *2015 IEEE International Parallel and Distributed Processing Symposium*, 2015, pp. 979–988.

[24] J. Meng, D. Tarjan, and K. Skadron, "Dynamic warp subdivision for integrated branch and memory divergence tolerance," in *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ser. ISCA '10. New York, NY, USA: Association for Computing Machinery, 2010, pp. 235–246. [Online]. Available: https://doi.org/10.1145/1815961.1815992

[25] H. Falahati, N. Mahani, A. Cristal, and O. Unsal, "Vista: Optimizing gpu scheduling through versatile locality-aware data sharing," in *2025 62nd ACM/IEEE Design Automation Conference (DAC)*, 2025, pp. 1–7.

[26] P. Xiang, Y. Yang, and H. Zhou, "Warp-level divergence in gpus: Characterization, impact, and mitigation," in *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, 2014, pp. 284–295.

[27] 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," *ACM SIGPLAN Notices*, vol. 48, no. 4, pp. 395–406, 2013.

[28] A. ElTantawy and T. M. Aamodt, "Warp scheduling for fine-grained synchronization," in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2018, pp. 375–388.

[29] Y. Liu, Z. Yu, L. Eeckhout, V. J. Reddi, Y. Luo, X. Wang, Z. Wang, and C. Xu, "Barrier-aware warp scheduling for throughput processors," in *Proceedings of the 2016 International Conference on Supercomputing*, ser. ICS '16. New York, NY, USA: Association for Computing Machinery, 2016. [Online]. Available: https://doi.org/10.1145/2925426.2926267

[30] S. Gray, "Maxas: Assembler for nvidia maxwell architecture," https://github.com/NervanaSystems/maxas, 2015, gitHub repository.

[31] A. Lavin, "maxdnn: An efficient convolution kernel for deep learning with maxwell gpus," *CoRR*, vol. abs/1501.06633, 2015. [Online]. Available: http://arxiv.org/abs/1501.06633

[32] G. He and E. Yoneki, "Cuasmrl: Optimizing gpu sass schedules via deep reinforcement learning," in *Proceedings of the 23rd ACM/IEEE International Symposium on Code Generation and Optimization*, ser. CGO '25. New York, NY, USA: Association for Computing Machinery, 2025, pp. 493–506. [Online]. Available: https://doi.org/10.1145/3696443.3708943

[33] DeepSeek AI, "DeepGEMM: FFMA SASS interleaving," https://github.com/deepseek-ai/DeepGEMM/tree/e2d6a107ef538ef2ff1b66d40fdba8d60281dec5?tab=readme-ov-file#ffma-sass-interleaving, 2025, commit e2d6a10; accessed 2025-08-20.

[34] A. Yang, A. Li, B. Yang, B. Zhang, B. Hui, B. Zheng, B. Yu, C. Gao, C. Huang, C. Lv, C. Zheng, D. Liu, F. Zhou, F. Huang, F. Hu, H. Ge, H. Wei, H. Lin, J. Tang, J. Yang, J. Tu, J. Zhang, J. Yang, J. Yang, J. Zhou, J. Zhou, J. Lin, K. Dang, K. Bao, K. Yang, L. Yu, L. Deng, M. Li, M. Xue, M. Li, P. Zhang, P. Wang, Q. Zhu, R. Men, R. Gao, S. Liu, S. Luo, T. Li, T. Tang, W. Yin, X. Ren, X. Wang, X. Zhang, X. Ren, Y. Fan, Y. Su, Y. Zhang, Y. Zhang, Y. Wan, Y. Liu, Z. Wang, Z. Cui, Z. Zhang, Z. Zhou, and Z. Qiu, "Qwen3 technical report," *arXiv preprint arXiv:2505.09388*, 2025.

[35] APXML, "Llama 4 scout," Website, 2025, model information page. [Online]. Available: https://apxml.com/models/llama-4-scout

[36] xAI, "Grok-1," GitHub Repository, 2024, gitHub repository. [Online]. Available: https://github.com/xai-org/grok-1

[37] R. Huerta, M. A. Shoushtary, J.-L. Cruz, and A. González, "Dissecting and modeling the architecture of modern gpu cores," in *Proceedings of the 58th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. ACM, 2025, pp. 369–384.

[38] T. M. Aamodt, W. W. L. Fung, and T. G. Rogers, *General-Purpose Graphics Processor Architectures*, ser. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2018, no. 44. [Online]. Available: https://doi.org/10.2200/S00848ED1V01Y201804CAC044

[39] Dao-AILab, "Flashattention: Fast and memory-efficient exact attention," GitHub repository, 2026, commit "main"; accessed 2026-04-06. [Online]. Available: https://github.com/Dao-AILab/flash-attention

[40] J. Li and S. Liu, "Flashmla: Efficient multi-head latent attention kernels," https://github.com/deepseek-ai/FlashMLA, 2025, accessed 20 Aug. 2025.

[41] J. Dong, B. Feng, D. Guessous, Y. Liang, and H. He, "Flex attention: A programming model for generating optimized attention kernels," 2024. [Online]. Available: https://arxiv.org/abs/2412.05496

[42] A. Silberschatz, P. B. Galvin, and G. Gagne, *Operating System Concepts*, 10th ed. Hoboken, NJ: John Wiley & Sons, Inc., 2018. [Online]. Available: https://os-book.com/OS10/index.html

[43] NVIDIA, "Cutlass example 48: Hopper warp-specialized gemm," GitHub repository, 2025, available at https://github.com/NVIDIA/cutlass/tree/main/examples/48_hopper_warp_specialized_gemm.

[44] NVIDIA Corporation, *Parallel Thread Execution ISA Version 9.0*, NVIDIA Corporation, 2025, accessed: 2025-08-20. [Online]. Available: https://docs.nvidia.com/cuda/archive/13.0.0/parallel-thread-execution/index.html

[45] O. Villa, D. Lustig, Z. Yan, E. Bolotin, Y. Fu, N. Chatterjee, N. Jiang, and D. Nellans, "Need for speed: Experiences building a trustworthy system-level gpu simulator," in *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2021, pp. 868–880.

[46] S. Gray, "Control codes," GitHub Wiki, jan 2016, gitHub Wiki, archived repository. [Online]. Available: https://github.com/NervanaSystems/maxas/wiki/Control-Codes

[47] NVIDIA Corporation, *Blackwell Tuning Guide*, Aug. 2025, accessed on August 21, 2025. [Online]. Available: https://docs.nvidia.com/cuda/blackwell-tuning-guide/index.html

[48] ——, *Hopper Tuning Guide*, Aug. 2025, accessed on August 21, 2025. [Online]. Available: https://docs.nvidia.com/cuda/hopper-tuning-guide/index.html

[49] NVIDIA, "CUTLASS SM100 pipeline primitives (sm100_pipeline.hpp)," GitHub repository, 2025, available at https://github.com/NVIDIA/cutlass/blob/main/include/cutlass/pipeline/sm100_pipeline.hpp.

[50] 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)*. IEEE, 2020, pp. 473–486.

[51] 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)*. IEEE, 2024, pp. 1–16.

[52] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Köpf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "PyTorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems*, vol. 32, 2019.

[53] W. Kwon, Z. Li, S. Zhuang, Y. Sheng, L. Zheng, C. H. Yu, J. E. Gonzalez, H. Zhang, and I. Stoica, "Efficient memory management for large language model serving with PagedAttention," in *Proceedings of the ACM SIGOPS 29th Symposium on Operating Systems Principles*, 2023, pp. 611–626.

[54] T. Dao, "Flashattention-2: Faster attention with better parallelism and work partitioning," 2023. [Online]. Available: https://arxiv.org/abs/2307.08691

[55] R. Prabhu, A. Nayak, J. Mohan, R. Ramjee, and A. Panwar, "vattention: Dynamic memory management for serving LLMs without PagedAttention," in *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2025.

[56] N. Shazeer, "Fast transformer decoding: One write-head is all you need," 2019. [Online]. Available: https://arxiv.org/abs/1911.02150

[57] J. Ainslie, J. Lee-Thorp, M. de Jong, Y. Zemlyanskiy, F. Lebrón, and S. Sanghai, "Gqa: Training generalized multi-query transformer models from multi-head checkpoints," 2023. [Online]. Available: https://arxiv.org/abs/2305.13245

[58] A. Liu, B. Feng, B. Wang, B. Wang, B. Liu, C. Zhao, C. Dengr, C. Ruan, D. Dai, D. Guo *et al.*, "Deepseek-v2: A strong, economical, and efficient mixture-of-experts language model," *arXiv preprint arXiv:2405.04434*, 2024.

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

[60] M. Martins, J. M. Matos, R. P. Ribas, A. Reis, G. Schlinker, L. Rech, and J. Michelsen, "Open cell library in 15nm freepdk technology," in *Proceedings of the 2015 Symposium on International Symposium on Physical Design*, ser. ISPD '15. New York, NY, USA: Association for Computing Machinery, 2015, pp. 171–178. [Online]. Available: https://doi.org/10.1145/2717764.2717783

[61] J. Cho, H. Choi, G. Heo, and J. Park, "LLMServingSim 2.0: A Unified Simulator for Heterogeneous and Disaggregated LLM Serving Infrastructure," in *2026 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2026, pp. 1–14.

[62] J. Cho, M. Kim, H. Choi, G. Heo, and J. Park, "LLMServingSim: A HW/SW co-simulation infrastructure for LLM inference serving at scale," in *2024 IEEE International Symposium on Workload Characterization (IISWC)*, 2024, pp. 15–29.

[63] S.-Y. Lee, A. Arunkumar, and C.-J. Wu, "Cawa: Coordinated warp scheduling and cache prioritization for critical warp acceleration of gpgpu workloads," *ACM SIGARCH Computer Architecture News*, vol. 43, no. 3S, pp. 515–527, 2015.

[64] NVIDIA, "CUTLASS," Jan. 2023. [Online]. Available: https://github.com/NVIDIA/cutlass

[65] J. Li, J. Martinez, and M. Huang, "The thrifty barrier: energy-aware synchronization in shared-memory multiprocessors," in *10th International Symposium on High Performance Computer Architecture (HPCA'04)*, 2004, pp. 14–23.

[66] Q. Cai, J. González, R. Rakvic, G. Magklis, P. Chaparro, and A. González, "Meeting points: using thread criticality to adapt multicore hardware to parallel regions," in *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '08. New York, NY, USA: Association for Computing Machinery, 2008, pp. 240–249. [Online]. Available: https://doi.org/10.1145/1454115.1454149

[67] A. Bhattacharjee and M. Martonosi, "Thread criticality predictors for dynamic performance, power, and resource management in chip multiprocessors," *ACM SIGARCH Computer Architecture News*, vol. 37, no. 3, pp. 290–301, 2009.