Understanding GPU Resource Interference One Level Deeper

Paul Elvinger
ETH Zurich
Switzerland

Foteini Strati
ETH Zurich
Switzerland

Natalie Enright Jerger
University of Toronto
Canada

Ana Klimovic
ETH Zurich
Switzerland

Abstract

GPUs are vastly underutilized, even when running resource-intensive AI applications, as GPU kernels within each job have diverse resource profiles that may saturate some parts of a device while often leaving other parts idle. Colocating applications is known to improve GPU utilization, but is not common practice as it becomes difficult to provide predictable performance due to workload interference. Providing predictable performance guarantees requires a deep understanding of how applications contend for shared GPU resources such as block schedulers, compute units, L1/L2 caches, and memory bandwidth. We study the key types of GPU resource interference and develop a methodology to quantify a workload's sensitivity to each type. We discuss how this methodology can serve as the foundation for GPU schedulers that enforce strict performance guarantees and how application developers can design GPU kernels with colocation in mind to improve efficiency.

CCS Concepts

• **Computing methodologies** → **Graphics processors**; **Machine learning**.

Keywords

GPU interference, GPU utilization

ACM Reference Format:
Paul Elvinger, Foteini Strati, Natalie Enright Jerger, and Ana Klimovic. 2025. Understanding GPU Resource Interference One Level Deeper. In *ACM Symposium on Cloud Computing (SoCC '25), November 19–21, 2025, Online, USA.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3772052.3772270

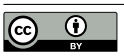
1 Introduction

Graphics Processing Units (GPUs) are widely used in AI training and inference to maximize performance per Watt. The power needs of AI workloads now comprise a significant percentage of datacenter power [40, 42, 44] and contribute significant cost, since GPU hardware is power hungry and expensive. To minimize total cost of ownership and make optimal use of the limited power budget, cloud providers need to operate GPU clusters at high utilization. Yet many recent studies show that GPUs are vastly underutilized [8, 10, 11, 17, 45, 48, 51, 60]. Even when serving multi-billion-parameter LLMs with large batch sizes, some GPU components may remain idle as resource requirements vary across compute vs. memory intensive phases of a job [14, 62]. For example, Microsoft reports less than 10% compute utilization during the memory-bound decoding phase of the Llama3-8B model on A100 GPUs [14]. GPUs can also be underutilized due to small batch sizes, communication, data preprocessing bottlenecks, and checkpointing [8, 9, 57].

GPU schedulers aim to improve utilization by colocating workloads. Temporal-sharing schedulers [10, 49–51, 55] execute one workload at a time to avoid resource interference, but fail to address single-workload GPU underutilization, and can cause severe queuing delays [45]. In contrast, spatial sharing systems [11, 15, 43, 45, 52, 56] allow concurrent workload execution, and propose strategies to minimize interference. However, existing spatial sharing systems do not provide reliable performance guarantees. Most systems rely on limited metrics to evaluate GPU resource utilization and define colocation strategies. As we show in §3, state-of-the-art systems such as Orion [45] and Usher [43] oversimplify GPU utilization and interference modeling, resulting in colocation decisions that can significantly degrade application performance. This prevents users from applying such colocation mechanisms in practice.

Analogous to how CPU schedulers make informed scheduling decisions by understanding how workloads interfere on CPU resources [6, 7], designing efficient GPU schedulers that spatially share resources while providing performance guarantees requires a deep understanding of GPU utilization and interference. In this paper, we characterize how GPU workloads (e.g., LLM decode) utilize and contend for resources within a GPU and discuss the implications for designing GPU kernels and scheduling systems. We study key sources of interference that arise from sharing the multifaceted components of GPUs, including streaming multiprocessors, warp schedulers, computation cores, high-bandwidth memory, caches, and shared memory. While some of these interference sources (memory bandwidth, caches) are well-known from CPUs, they remain largely understudied in modern GPUs. We propose techniques to assess the sensitivity of GPU kernels and workloads to contention for each type of resource, using a combination of GPU profiling tools and a customizable suite of GPU microbenchmarks. Our microbenchmarks are available at https://github.com/eth-easl/gpu-util-interference and our LLM profiling scripts are available at https://github.com/eth-easl/vllm_profile.

We draw several key takeaways from our analysis. First, scheduling decisions should be made at fine (per-kernel) granularity, to avoid head-of-line blocking at the GPU block scheduler. Second, allocating kernels to separate Streaming Multiprocessors (SMs) helps eliminate some sources of interference, but contention for shared resources like L2 cache and memory bandwidth can still cause significant slowdowns. Third, sharing SMs between kernels can be beneficial, though it is subject to multiple sources of interference, such as shared memory bandwidth, warp scheduling, and compute pipelines' contention. Finally, we observe that trading off per-kernel marginal performance improvements for higher colocation opportunities can significantly benefit GPU efficiency and cost. Using these insights, we describe ways for designing GPU schedulers



This work is licensed under a Creative Commons Attribution 4.0 International License.
SoCC '25, Online, USA
© 2025 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2276-9/2025/11
https://doi.org/10.1145/3772052.3772270

that effectively colocate applications to reduce cost while providing strict performance guarantees. We also discuss implications for GPU application programmers, such as how to design GPU kernels to be more suitable for colocation to improve efficiency.

2 Background

We describe the internal architecture of GPUs, introducing terminology and utilization metrics that we will refer to later.

2.1 GPU Hardware Overview

GPU architecture: Figure 1 depicts a modern GPU.[1] GPUs consist of multiple clusters of Streaming Multiprocessors (SMs). Each SM consists of subpartitions (SMSP) that contain a warp scheduler (which can schedule 32 threads/cycle), an L0 instruction cache, a register file, and various compute units for different data types (e.g., int32, fp32) and different operations, referred to as *pipelines* (e.g., tensor cores) [38]. The mapping of pipelines to compute units is hidden from users, though there have been attempts to reverse engineer it [20, 24, 25, 32]. The GPU contains a main memory shared by all SMs, accessed through a two-level on-chip cache hierarchy. The L2 cache is shared across all SMs, while each SM has a private memory space combining L1 cache and shared memory. The allocation between L1 cache and shared memory can be manually configured by the user [23].

Threads, Blocks, and Warps. GPU programs consist of CUDA kernels, executed by one or more GPU threads [22]. From a software perspective, GPU threads are grouped into *blocks*, which are arranged in a *grid*. Each thread has access to a set of registers and shared memory for its whole lifetime. Threads in a block communicate through shared memory and synchronize using barriers or other atomic operations. At the hardware level, the GPU executes threads in groups called *warps*, typically consisting of 32 threads. Each kernel is associated with a *CUDA stream*, which defines sequential execution of operations. If enough resources are available, kernels launched from multiple streams can execute concurrently.

GPU Scheduling: NVIDIA GPUs schedule kernels at multiple levels. First, the thread block scheduler maps thread blocks to SMs. A block remains on an SM until all its threads complete execution. Scheduling is constrained by SM resource limits (max number of blocks, threads, registers, and shared memory) that depend on the GPU architecture. A block is scheduled on an SM only if that SM has enough resources left to accommodate all threads of that block. Once a block is scheduled on an SM, its warps are assigned to one of the subpartitions and are considered active. Each clock cycle, the warp scheduler in each subpartition chooses one eligible warp and schedules one or more instructions from that warp.

2.2 GPU Utilization Metrics

Most GPU schedulers rely on GPU utilization metrics to make colocation decisions. There are many different GPU utilization metrics due to the multi-faceted nature of GPU resources. We outline popular metrics used in prior works below and metrics that will be used throughout our analysis. Most metrics can be reported by NVIDIA tools such as Nsight Compute (NCU) [26].

[1]We focus on NVIDIA GPUs and terminology. AMD GPUs follow similar architecture [1], and provide tools such as Omniperf to get insights about kernels' execution [2].

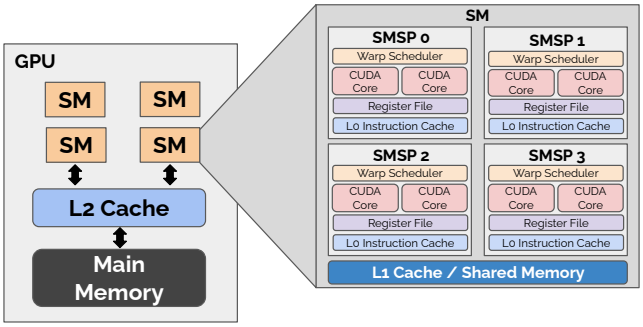


Figure 1: Simplified diagram of an NVIDIA GPU (based on an H100), focusing on a Streaming Multiprocessor (SM).

GPU utilization from `nvidia-smi`/NVML [29, 30] depicts the percentage of time at least one kernel is active on a GPU, without revealing how well the kernels utilize the various GPU resources. This metric is used by various works [3, 12, 49, 53, 54].

SM utilization refers to the number of SMs needed by a kernel, taking into account the kernel's grid size, block size, registers/thread, shared memory/thread, and the respective SM resource limits of a GPU. Orion [45] and REEF [11] use this metric.

Arithmetic intensity refers to the ratio of floating-point operations to total data movement, and can be found using NCU's roofline model [31]. Orion [45] uses this metric to classify kernels as compute-bound or memory-bound.

Achieved occupancy measures how many active warps exist per SM per clock cycle on average [33, 35] and is obtained by NCU, using the `sm__warps_active.avg.pct_of_peak_sustained_active` metric. It is used by Usher [43] to estimate the compute requirement of a kernel.

Pipe utilization measured by the NCU metric `sm__inst_executed_pipe_*.avg.pct_of_peak_sustained_active` (* indicates the pipeline, e.g. `fma`, `tensor`) expresses how effectively a pipeline is used (when executing at least one warp) relative to its peak performance.

Issued instructions per cycle (IPC): The NCU metric `sm__inst_issued.avg.per_cycle_active` (also called `IPC` [34]) represents the average number of warp instructions issued per cycle per SM. Our GPUs have 4 subpartitions per SM, each capable of issuing one warp instruction per cycle, i.e., the maximum IPC per SM is 4.

L2 cache throughput and hit rate The metrics `lts__throughput.avg.pct_of_peak_sustained_elapsed` and `lts__t_sector_hit_rate.pct` measure the average L2 cache throughput and hit rate.

Shared Memory instruction bandwidth: The NCU metric `l1tex__data_pipe_lsu_wavefronts_mem_shared.sum.pct_of_peak_sustained_elapsed` measures the performance of shared memory load/store wavefronts[2] processed through the L1 data pipe. It provides insights into how effectively the available shared memory bandwidth is used.

Memory bandwidth utilization: The NCU metric `gpu__dram_throughput.avg.pct_of_peak_sustained_elapsed` measures the utilization of the memory bandwidth.

[2]A *wavefront* is the maximum unit that can pass through a pipeline stage per cycle [38].

3 Pitfalls of GPU schedulers

Many existing GPU schedulers aim to improve utilization by colocating workloads, with temporal and/or spatial sharing. We focus on spatial sharing schedulers as they allow concurrent workload execution and hence require strategies to minimize interference.

We find that state-of-the-art GPU schedulers consider only a *subset* of GPU utilization metrics, resulting in unreliable performance guarantees. We present common pitfalls below:

Pitfall 1: Relying on a single utilization metric. Most schedulers rely on a single metric to assess GPU utilization and make colocation decisions. For example, Usher [43] relies on achieved occupancy to assess a kernel's compute requirements, and colocates two kernels if the sum of achieved occupancy values is < 100%. [3] Achieved occupancy can be misleading, as a kernel can saturate GPU resources even with low achieved occupancy. To demonstrate this, we launch two instances of a `compute` kernel on an H100. The kernel performs several iterations of independent element-wise fp32 multiplications and we launch both instances with 132 blocks and 128 threads/block per kernel. The launch configuration is chosen to have one thread block running per SM and one warp per SMSP. NCU reports an achieved occupancy of 6.25% per kernel, suggesting that colocation should not result in performance degradation. We show two counter-examples. First, we follow Usher's suggestion of constraining each kernel to a percentage of SMs equal to its achieved occupancy. Thus, we limit each kernel to 6.25% of the GPU SMs, setting the `CUDA_MPS_ACTIVE_THREAD_PERCENTAGE` [19] variable. We observe a 8.57× increase in each kernel's latency indicating that the number of SMs needed is not aligned with the achieved occupancy. Second, we run both kernels concurrently with the same launch configuration in separate CUDA streams and let them use all available SMs. We observe a 1.73× increase in each kernel's latency, despite their low achieved occupancy. These examples indicate that predicting interference and required compute resources based only on the number of active warps is not sufficient.

Other schedulers also consider only a single metric, such as SM utilization or the utilization metric reported by nvidia-smi, which leads to suboptimal decisions [3, 11, 12, 49, 53, 54].

Pitfall 2: Ignoring critical metrics. Although GPU schedulers may consider more metrics, they often overlook essential ones. For example, Orion [45] colocates kernels with complementary resource profiles (i.e., compute vs. memory bound kernels), which it determines based on arithmetic intensity. However, it does not consider the IPC metric. While arithmetic intensity correlates with IPC, Orion overlooks cases where a compute kernel's IPC is too high and will interfere with any other colocated kernel.

To demonstrate this, we reuse the `compute` kernel from the previous example and a `copy` kernel, which repeatedly copies a 4GB input to output array. We tune the number of iterations so the two kernels have similar execution times. On an H100 GPU, we launch both kernels with 132 blocks and 1024 threads/block, such that we have one thread block of each kernel running on each SM and use all available threads per SM. Using NCU, we confirm that `compute` is compute-bound and `copy` is memory-bound, thus Orion would colocate them, expecting low interference. However, we observe that the execution time of `copy` doubles under colocation. NCU shows that the `compute` kernel already saturates the available IPC per SM issuing 3.99 inst/cycle/SM on average. The colocated `copy` kernel with an IPC of 0.57 will therefore experience warp scheduling interference. §4.4.2 elaborates on IPC interference.

The pitfalls observed in related work raise the question: *how to accurately measure GPU utilization and estimate interference?*

4 GPU interference analysis

We highlight the main GPU resources where interference can occur for popular AI workloads (e.g., LLM decode) and how to identify whether a kernel is susceptible to a specific kind of interference.

4.1 Evaluation Setup

We develop custom CUDA benchmarks, each stressing a specific GPU resource. We assess the sensitivity of various workloads to resource interference by colocating them with these benchmarks. For intra-sm colocation (§4.4) we use CUDA streams [18], while for inter-SM colocation (§4.3), we additionally use CUDA Green Contexts [36] to partition SMs into mutually exclusive sets, each assigned to a separate stream.[4] Each benchmark maintains a constant level of interference throughout the lifetime of the colocated application. Most experiments focus on large language model (LLM) inference and the Time Between Tokens (TBT) during the decode phase. Increased TBT directly impacts user experience in applications such as chatbots. We use the Gemma3-1B-IT [47] and Llama3.1-8B-Instruct [16] models, both running on a modified fork of vLLM (May 2025) adapted to support kernel colocation. We evaluate workloads on a NVIDIA H100 NVL (132 SMs, CUDA 12.9, driver 575.57.08) and RTX3090 GPUs (82 SMs, CUDA 12.6, driver 560.35.03).

4.2 Block Scheduler Interference

As described in §2, the block scheduler assigns blocks to SMs as long as resource constraints are met. When resources are insufficient, blocks are serialized, increasing latency. We demonstrate this on an H100 GPU by colocating the decode phase of the Llama3.1-8B-Instruct model [16] with a lightweight `sleep` kernel that repeatedly invokes `nanosleep`. At each of the 10 decode steps, we launch the `sleep` kernel in a separate stream with an approximate sleep duration of 10 ms, using 132 thread blocks, to ensure one thread block per SM[5] and 128 threads per block (1 warp per SMSP).

In isolation, the P90 TBT of Llama3-8B (batch size 1, prompt length 1000) is 7.53 ms, increasing to 16.56 ms when colocated with the `sleep` kernel. Figure 2 shows the Nsight Systems CUDA trace [27] for the first decode iteration in both cases. Although both workloads launch simultaneously, their overlap is minimal, limited to a few short kernels at the start. From the fourth kernel onward, decode execution is delayed until the `sleep` kernel completes due to SM resource contention. The `sleep` kernel uses 16 registers per thread (2048 per block), leaving $65536 - 2048 = 63288$ registers

[3] We focus on how Usher makes colocation decisions, not its extra components, such as operator graph merging.

[4] Contrary to common belief, NVIDIA Multi-Process Service (MPS) does not enforce mutual exclusion of SMs between MPS clients when setting limits. It only restricts the maximum number of SMs available to a client [37].

[5] We confirmed that each SM hosts one block of the `sleep` kernel by having each block print its SM ID [39].

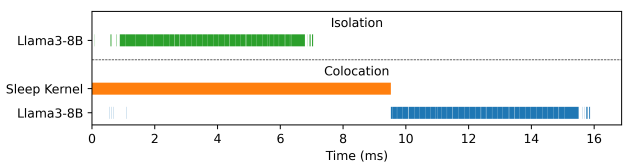


Figure 2: Decode of Llama3-8B in isolation vs colocated with a `sleep` kernel on H100 (batch size 1, prompt size 1000)

per SM available on the H100. The fifth decode kernel, however, requires 64512 registers per block, exceeding the remaining registers and preventing concurrent execution. Consequently, the block scheduler stalls decode execution until the `sleep` kernel finishes, significanlty increasing TBT and underutilizing the GPU. Several subsequent kernels encounter the same issue. We observe similar behavior on the H100 for Gemma3-1B [47], though colocation with this model is feasible on the RTX3090 (§4.4.2).

Takeaway: Simply scheduling two kernels on the same GPU does not ensure colocation, as they compete for SM resources such as shared memory, registers, and threads. To prevent head-of-line blocking, where one kernel hinders others, schedulers should make decisions at fine granularity, e.g., per set of kernels [62], per kernel [11, 45] or per set of thread blocks [5]. The above example also shows that if developers write kernels that use as many GPU resources as possible (common today), they leave the scheduler with little opportunities to colocate and optimize efficiency.

4.3 Inter-SM Interference: Global Memory and L2 cache

While workloads can be configured to run on separate SMs, global memory and L2 cache remain shared resources. As we illustrate below, interference can arise in two forms: (1) L2 cache pollution, which degrades locality, (2) L2 and memory bandwidth contention.

L2 Cache Pollution A kernel is susceptible to L2 pollution if it repeatedly accesses data from L2 and it exhibits a high L2 hit rate. To study this, we colocate two instances of the `copy` kernel, in separate CUDA green contexts on the H100 (with a 50MB L2). We split the SMs across both contexts: one kernel uses 64 thread blocks (1024 threads/block) on 64 SMs; the other uses 68 thread blocks (1024 threads/block) on 68 SMs. To minimize L1 effects, we configure the kernels to maximize for shared memory. We gradually increase the data size per instance and measure the slowdown of one kernel when colocated versus running in isolation.

Figure 3 shows the results along with the isolated L2 hit rate. For input sizes ≤ 8MB, there is no slowdown, as both instances' input and output arrays fit into L2. At 16MB, slowdown peaks at 2.15× due to the combined 64MB working set exceeding L2 capacity. Beyond 26MB, slowdown plateaus around 1.12×, as L2 locality in isolation is lost (more data needs to be loaded from main memory). Further L2 pollution by a colocated kernel creates no additional slowdown. The L2 hit rate stabilizes near 50%, as every store is counted as a hit by NCU, making half of accesses appear as hits [21].

L2 Cache/Memory Bandwidth Interference Kernels with high memory demand often lack L2 locality, making them less prone to pollution but more sensitive to bandwidth contention. To

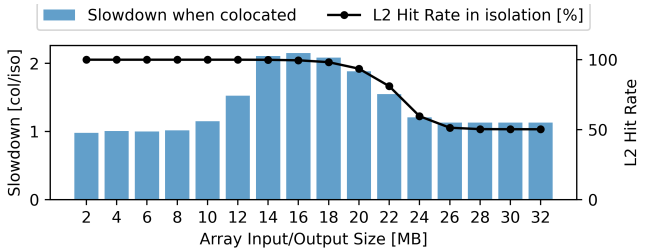


Figure 3: L2 cache interference due to cache pollution between two `copy` kernels on an H100.

Thread Blocks of interference kernel	0	34	68	102	136
L2 Band Util (copy kernel) [%]		37	68	87	95
Mem Band Util (copy kernel) [%]		27	51	69	81
P90 TBT (ms)	16.9	17.6	18.38	19.92	22

Table 1: Effect of memory bandwidth interference on the Time Between Tokens for the LLAMA-3.1-8B on H100 GPU, batch size 8, 16384 tokens per prompt. When the LLM runs on the whole GPU (no green contexts), P90 TBT is 14.2 ms

demonstrate this, we colocate the decode phase of Llama3.1-8B with a `copy` kernel that transfers a 4GB input using vectorized loads. Each workload runs in its own CUDA green context, with 64 SMs allocated to Llama and 68 SMs to the `copy` kernel. Table 1 reports the P90 TBT when generating 10 tokens for the LLM running alone vs. when colocated with the `copy` kernel, along with the `copy` kernel's L2 and Memory bandwidth throughput utilization. As the number of `copy` thread blocks increases, bandwidth pressure rises, causing up to 1.3× slowdown. Since L2 access is mandatory, higher memory bandwidth also translates to higher L2 throughput. We observe similar trends across other prompt and batch sizes. We observe that all LLM decode kernels do not benefit much from L2 cache locality and have to load most of their data from main memory. This makes them susceptible to L2 and memory bandwidth interference; in the examples of Table 1, most LLM decode kernels experienced significant slowdown.

Takeaway: Even when workloads run on disjoint SMs, using strict SM isolation mechanisms such as green contexts, they can still contend for L2 capacity and L2 and global memory bandwidth.

4.4 Intra-SM Interference

When workloads run on the same SM, they can interfere in the warp scheduler, the computation pipelines or shared memory.

4.4.1 Shared Memory. As mentioned in §2.1, each SM has a private memory space partitioned between the L1 cache and shared memory. To achieve high performance, kernels with high arithmetic intensity load chunks of data from global memory into shared memory for computation before writing results back to global memory. Modern NVIDIA GPUs (≥ CC 5.x) have 32 memory banks, each with a bandwidth of 32 bits/cycle. Successive 32-bit words map to successive banks. An *n-way bank conflict* occurs when $n > 1$ threads within the same warp access different data in the same bank. An n-way bank conflict is resolved by serializing the requests over n conflict-free requests, thereby decreasing throughput.

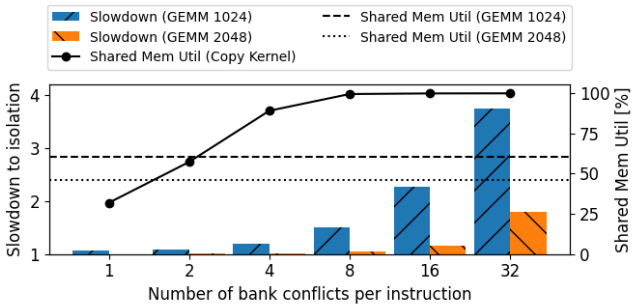


Figure 4: Shared Memory interference. Effect of increasing shared memory bank conflicts on two cuBLAS GEMM implementations for dimensions 1024 and 2048 on an H100.

Kernels running concurrently on the same SM can contend for shared memory bandwidth. We demonstrate this by colocating the PyTorch `torch.mm` operator [41] with a custom n-strided copy kernel that repeatedly loads and stores elements of a 4KB array within shared memory. Varying the access stride affects the shared memory bank conflicts within a warp. Figure 4 shows the slowdown under colocation vs. running in isolation of the `torch.mm` operator on the H100 as we increase the number of bank conflicts within our copy kernel. Depending on the input matrix dimensions, `torch.mm`, uses different implementations of cuBLAS GEMM. For a 32-way bank conflict, the slowdown is 1.79× for dims 2048 and 3.75× for dims 1024. The cuBLAS implementation for dimension 1024 has a higher utilization of the shared memory pipeline, which explains its higher sensitivity to interference. The slowdown for both implementations increases significantly beyond 4 bank conflicts. Since the shared memory pipeline is already saturated by the copy kernel, further increases in bank conflicts lead to further serialization of shared memory requests, thereby increasing the `torch.mm` latency.

Takeaway: A kernel with non-optimal shared memory access may result in many bank conflicts, saturating the SM's shared memory pipeline and starving memory accesses of colocated kernels.

4.4.2 IPC interference. To demonstrate how the architectural limit of 4 instr/cycle/SM can become a bottleneck, we colocate the memory-bound decode phase of the Gemma3-1B model [47] with four versions of our `compute` kernel (S_1 to S_4), each increasing the Instruction Level Parallelism (ILP) to put growing pressure on the warp scheduler. To enable concurrent intra-SM execution and avoid sequential kernel launches due to shared memory constraints, we configure the L1 cache/shared memory split to maximize shared memory [23]. We conducted the experiments on the RTX3090 GPU, as certain Gemma kernels on the H100 saturate the thread capacity per SM, preventing intra-SM colocation (§4.2). In each scenario (S_0 to S_4), we generate 10 new tokens and report the 90th percentile of TBT. For scenarios S_1 to S_4, we launch the modified `compute` kernel with 82 thread blocks (one per SM) and 128 threads per block (one warp per SMSP). Scenario S_0 represents the baseline, with the Gemma model running in isolation and without setting any custom L1 cache/shared memory configuration.

Table 2 shows that the TBT remains largely unaffected in scenarios S_0 to S_3, but experiences significant degradation as the `compute` kernel's IPC approaches the hardware limit in S_4. Comparing S_0 and

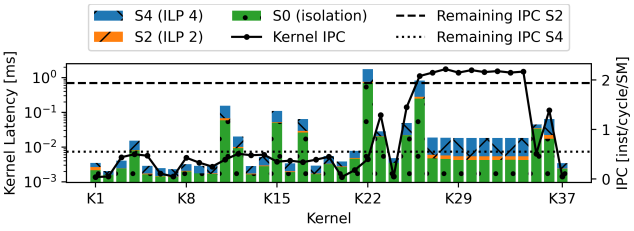


Figure 5: Kernel latencies for scenarios S_0, S_2, S_4 running a Gemma3-1B decode iteration with a single hidden layer on the RTX3090 (batch size 8, prompt size 1000). Dashed lines represent left-over IPC when `compute` kernel is running.

	S_0	S_1	S_2	S_3	S_4
IPC `compute` [instr/cycle/SM]	0	1.18	2.06	2.9	3.45
Prompt Size 1000, Batch Size 1	5.59	5.75	6.00	6.24	10.74
Prompt Size 1000, Batch Size 8	6.08	6.23	6.56	6.94	12.52

Table 2: P90 Time Between Tokens (TBT) latency in `ms` when generating 10 tokens with the Gemma3-1B [47] model on the RTX3090. The model is run in isolation (S_0) and colocated with a compute bound kernel that emits an increasing number of instructions per cycle ($S_1 - S_4$).

S_1 shows that manually setting the shared memory configuration to maximize for shared memory has no noticeable impact on the TBT of the Gemma model. Nevertheless, this potential impact should be kept in mind whenever manually setting the configuration. Figure 5 highlights IPC interference for a prompt of 1000 tokens and batch size 8, illustrating the latency slowdown experienced by each kernel for the Gemma3-1B model configured with a single hidden layer (in contrast to 26 in the full model). In S_2, only kernels $K26$ to $K34$ experience high slowdown due to high IPCs. However, in S_4, all kernels slow down as the combined IPC approaches or exceeds the architectural limit of 4 *instr*/cycle/SM. This highlights that intra-SM colocation under limited interference is feasible, but must be managed carefully to prevent performance degradation.

Takeaway: While blocks from different kernels can benefit from colocation on the same SM, a kernel that has a very high IPC can cause significant slowdown to other kernels, due to warp scheduling interference. In that case, scheduling to separate SMs (e.g., using green contexts) might be a better approach.

4.4.3 Pipeline Interference. We modify our `compute` kernel to utilize fp64 multiplication `__dmul_rn` [28]. We examine scenarios S_1 to S_4 increasing ILP from one to four. We colocate two kernels, each with 132 thread blocks and 128 threads per block, to have one thread block per SM and one warp per kernel on each SMSP. Table 3 shows that when the *aggregated* FP64 pipeline utilization (computed by summing up the per-kernel utilizations from Table 3) remains below 100% (S_1 and S_2), colocation offers a significant speedup (> 1.87×). However, the speedup decreases significantly as utilization exceeds 100% (S_3 and S_4), despite IPC not being a bottleneck. Even though saturating a computation pipeline is frequently associated with a high IPC, compute interference can still occur before the warp scheduler saturates. In the H100 architecture, the peak performance

	S_1	S_2	S_3	S_4
IPC `compute` [instr/cycle/SM]	0.64	1.10	1.53	1.96
FP64 Pipe Utilization [%]	24.22	47.71	69.42	90.68
Speedup [seq/col]	1.93	1.87	1.33	1.03

Table 3: Speedup of colocating two FP64 `compute` kernels over running them sequentially on H100. IPC and FP64 Pipe Util correspond to profiling the `compute` kernel in isolation.

for arithmetic FP64 operations is only half that of FP32 operations, thus we demonstrate interference for FP64 operations.

This example also illustrates why the achieved occupancy alone is insufficient for colocation decisions, as already seen in §3. Despite having an achieved occupancy of 6.25% in all four scenarios, the kernel saturates an entire computation pipeline due to a high number of independent arithmetic instructions. While this scenario is less likely to occur in practice (since most GPU workloads do not use FP64), this demonstrates another potential source of interference.

Takeaway: SM's pipelines can saturate even before the warp scheduler saturates, which might require scheduling to different SMs to avoid pipeline interference.

5 Research Outlook and Conclusions

We discuss how insights from §4 can guide the design of an interference-aware GPU scheduler (§5.1), how GPU vendors can enable more flexible and efficient colocation (§5.2), and discuss how GPU programmers can develop GPU kernels with colocation in mind (§5.3).

5.1 Interference-aware GPU Scheduling

Our analysis in §4 helps us identify key requirements that a GPU scheduler should satisfy to enable efficient GPU workload colocation with strict performance guarantees. First, scheduling decisions should be made at a *fine granularity*, per set of kernels [62], per kernel [45], or per thread blocks [5], to avoid head-of-line blocking caused by long-running, resource-demanding kernels preventing colocation, as shown in §4.2.

Second, a scheduler should be able to predict and quantify interference between colocated kernels and workloads. As a first step, we propose developing a kernel-level interference estimator to predict the performance of kernels under colocation. For each workload, the estimator can collect each kernel's metrics and resource sensitivity as outlined in §4. The estimator can then predict each kernel's slowdown due to interference at each resource. Existing interference estimators only consider a subset of interference sources [52, 59]. Themis [58] and GPUPool [46] consider many of the resources outlined in §4, but treat them as a black-box input to an ML model, and present their analysis and evaluation only in simulation. Instead, we demonstrated interference caused by contention for these resources on high-end NVIDIA GPUs. The kernel-level estimator provides a foundation for implementing a workload-level interference estimator that can predict a job's interference sensitivity. Using that estimator, a GPU scheduler can find workload pairs suitable for colocation.

Third, a GPU scheduler should balance per-workload performance and GPU efficiency. For example, restricting a workload or a set of kernels to a subset of SMs can increase GPU efficiency with acceptable performance degradation (in §4.3, we saw that restricting the LLM decode to less than half of the GPU's SMs has only a 1.19× TBT slowdown). Adjusting SM L1/shared memory configurations can also improve colocation, as we saw in §4.4.2.

Finally, GPU schedulers should account for differences in GPU generations. Libraries such as cuDNN and cuBLAS have different implementations for different GPU architectures, meaning that the same workload might have different behavior on two different GPUs, as shown for the LLM decode phase in §4. Thus, a scheduler should reprofile a GPU workload when running on a different GPU. Although profiling with tools such as NCU can span minutes to hours, this is acceptable since models are trained or deployed for large periods of time.

5.2 Hardware Support for Spatial GPU Sharing

The closed-source nature of NVIDIA GPUs limits user control over kernel execution as various hardware mechanisms are a black box. Exposing some hardware features can help programmers take better control of the GPU. Better intra-SM visibility is needed, providing insights into the warp scheduling algorithm and the mapping of instructions to physical cores. Enhancing profiling tools such as NCU to allow for collecting GPU metrics under kernel colocation would be very beneficial for better understanding interference, although very challenging since these profilers heavily rely on deterministic kernel replay. Furthermore, programmer-friendly ways to partition SMs and DRAM channels at the kernel level can mitigate intra-SM and DRAM bandwidth interference. CUDA green contexts already provide much stricter isolation compared to MPS, but they only partition SMs and not DRAM channels. Related work proposes limiting a kernel's blocks to specific DRAM channels, but they are code-intrusive and unsuitable for ML workloads with closed-source kernels [4, 13, 56, 61]. Finally, enabling kernel preemptibility could improve kernel colocation, especially in real-time tasks, as shown by REEF [11] for AMD GPUs.

5.3 GPU Kernel Design Tradeoffs

Most existing high-performance GPU workloads are not designed with colocation in mind. Kernels often will try to maximize their intra-SM resource usage (registers, amount of shared memory), and launch large grid sizes, preventing the thread block scheduler from running anything else in the GPU at the same time (see §4.2). However, as related work has shown, even high-performance kernels leave parts of the GPU severely underutilized [14, 45]. This presents a tradeoff between maximizing individual kernel performance and enabling efficient colocation [62]. Marginal performance gains may come at the cost of GPU utilization and overall efficiency. Thus, if the goal is to increase GPU efficiency with workload colocation, GPU programmers can aid the GPU schedulers by providing colocation-friendly kernel variants.

We hope our characterization of GPU interference inspires and informs future work on GPU scheduling and custom kernel design, and we encourage similar studies across other vendors (e.g. AMD).

6 Acknowledgments

We thank the HotOS'25 and SoCC'25 reviewers for their insightful feedback. Foteini Strati is supported by the Swiss National Science Foundation (project number 200021_204620). Natalie Enright Jerger is supported in part by the Canada Research Chairs program.

References

[1] AMD. Amd gpu hardware basics. https://www.olcf.ornl.gov/wp-content/uploads/2019/10/ORNL_Application_Readiness_Workshop-AMD_GPU_Basics.pdf, 2019.

[2] AMD. Omniperf documentation. https://rocm.docs.amd.com/projects/omniperf/en/docs-6.2.0/, 2024.

[3] Vivek M. Bhasi, Aakash Sharma, Rishabh Jain, Jashwant Raj Gunasekaran, Ashutosh Pattnaik, Mahmut Taylan Kandemir, and Chita Das. Towards slo-compliant and cost-effective serverless computing on emerging gpu architectures. In *Proceedings of the 25th International Middleware Conference*, Middleware '24, page 211–224, New York, NY, USA, 2024. Association for Computing Machinery.

[4] Binghao Chen, Han Zhao, Weihao Cui, Yifu He, Shulai Zhang, Quan Chen, Zijun Li, and Minyi Guo. Maximizing the utilization of gpus used by cloud gaming through adaptive co-location with combo. In *Proceedings of the 2023 ACM Symposium on Cloud Computing*, SoCC '23, page 265–280, New York, NY, USA, 2023. Association for Computing Machinery.

[5] Patrick H. Coppock, Brian Zhang, Eliot H. Solomon, Vasilis Kypriotis, Leon Yang, Bikash Sharma, Dan Schatzberg, Todd C. Mowry, and Dimitrios Skarlatos. Lithos: An operating system for efficient machine learning on gpus, 2025.

[6] Christina Delimitrou and Christos Kozyrakis. Paragon: Qos-aware scheduling for heterogeneous datacenters. *SIGPLAN Not.*, 48(4):77–88, March 2013.

[7] Christina Delimitrou and Christos Kozyrakis. Quasar: resource-efficient and qos-aware cluster management. *SIGPLAN Not.*, 49(4):127–144, February 2014.

[8] Yanjie Gao, Yichen He, Xinze Li, Bo Zhao, Haoxiang Lin, Yoyo Liang, Jing Zhong, Hongyu Zhang, Jingzhou Wang, Yonghua Zeng, Keli Gui, Jie Tong, and Mao Yang. An empirical study on low gpu utilization of deep learning jobs. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ICSE '24, New York, NY, USA, 2024. Association for Computing Machinery.

[9] Dan Graur, Damien Aymon, Dan Kluser, Tanguy Albrici, Chandramohan A. Thekkath, and Ana Klimovic. Cachew: Machine learning input data processing as a service. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 689–706, Carlsbad, CA, July 2022. USENIX Association.

[10] Arpan Gujarati, Reza Karimi, Safya Alzayat, Wei Hao, Antoine Kaufmann, Ymir Vigfusson, and Jonathan Mace. Serving DNNs like clockwork: Performance predictability from the bottom up. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 443–462. USENIX Association, November 2020.

[11] Mingcong Han, Hanze Zhang, Rong Chen, and Haibo Chen. Microsecond-scale preemption for concurrent GPU-accelerated DNN inferences. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 539–558, Carlsbad, CA, July 2022. USENIX Association.

[12] Yiyuan He, Minxian Xu, Jingfeng Wu, Wanyi Zheng, Kejiang Ye, and Chengzhong Xu. Uellm: A unified and efficient approach for llm inference serving, 2024.

[13] Saksham Jain, Iljoo Baek, Shige Wang, and Ragunathan Rajkumar. Fractional gpus: Software-based compute and memory bandwidth reservation for gpus. In *2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 29–41, 2019.

[14] Aditya K Kamath, Ramya Prabhu, Jayashree Mohan, Simon Peter, Ramachandran Ramjee, and Ashish Panwar. Pod-attention: Unlocking full prefill-decode overlap for faster llm inference, 2024.

[15] Gangmuk Lim, Jeongseob Ahn, Wencong Xiao, Youngjin Kwon, and Myeongjae Jeon. Zico: Efficient GPU memory sharing for concurrent DNN training. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 161–175. USENIX Association, July 2021.

[16] Meta. Llama-3.1 8b instruct. https://huggingface.co/meta-llama/Llama-3.1-8B-Instruct, 2024.

[17] Kelvin K. W. Ng, Henri Maxime Demoulin, and Vincent Liu. Paella: Low-latency model serving with software-defined gpu scheduling. In *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP '23, page 595–610, New York, NY, USA, 2023. Association for Computing Machinery.

[18] NVIDIA. Gpu pro tip: Cuda 7 streams simplify concurrency. https://developer.nvidia.com/blog/gpu-pro-tip-cuda-7-streams-simplify-concurrency/, 2015.

[19] NVIDIA. Nvidia multi-process service. https://docs.nvidia.com/deploy/mps/index.html, 2015.

[20] NVIDIA. Question about sp and sm. https://forums.developer.nvidia.com/t/questions-about-sp-and-sm/76700/6, 2019.

[21] NVIDIA. Nsight compute, l2 hit rate. https://forums.developer.nvidia.com/t/l2-hit-rate-always-at-100/257458/2?u=elpaul, 2023.

[22] NVIDIA. Cuda c++ programming guide. https://docs.nvidia.com/cuda/cuda-c-programming-guide/, 2024.

[23] NVIDIA. Cuda shared memory configuration. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#shared-memory-7-x, 2024.

[24] NVIDIA. Is there a document about in which hardware unit(ie. alu fmu…) an instruction is executed? https://forums.developer.nvidia.com/t/is-there-a-document-about-in-which-hardware-unit-ie-alu-fmu-an-instruction-is-executed/227475, 2024.

[25] NVIDIA. Mapping of pipelines to functional units. https://forums.developer.nvidia.com/t/mapping-of-pipelines-to-functional-units/315200, 2024.

[26] NVIDIA. Nsight compute. https://developer.nvidia.com/nsight-compute, 2024.

[27] NVIDIA. Nsight systems. https://developer.nvidia.com/nsight-systems, 2024.

[28] NVIDIA. Nvidia double precision intrinsics. https://docs.nvidia.com/cuda/cuda-math-api/cuda_math_api/group__CUDA__MATH__INTRINSIC__DOUBLE.html, 2024.

[29] NVIDIA. Nvidia management library (nvml). https://developer.nvidia.com/management-library-nvml, 2024.

[30] NVIDIA. nvidia-smi. https://docs.nvidia.com/deploy/nvidia-smi/index.html, 2024.

[31] NVIDIA. Roofline charts. https://docs.nvidia.com/nsight-compute/ProfilingGuide/index.html#roofline-charts, 2024.

[32] NVIDIA. Separate cuda core pipeline for fp16 and fp32? https://forums.developer.nvidia.com/t/separate-cuda-core-pipeline-for-fp16-and-fp32/302018, 2024.

[33] NVIDIA. What does achieved active warps per sm in nsight means and how to calculate it? https://forums.developer.nvidia.com/t/what-does-achieved-active-warps-per-sm-in-nsight-means-and-how-to-calculate-it/128256l, 2024.

[34] NVIDIA. what is ipc(instructions per cycle)? https://forums.developer.nvidia.com/t/what-is-ipc-instructions-per-cycle/66138, 2024.

[35] NVIDIA. Achieved occupancy. https://docs.nvidia.com/gameworks/content/developertools/desktop/analysis/report/cudaexperiments/kernellevel/achievedoccupancy.htm, 2025.

[36] NVIDIA. Cuda green context. https://docs.nvidia.com/cuda/cuda-driver-api/group__CUDA__GREEN__CONTEXTS.html, 2025.

[37] NVIDIA. Cuda green context. https://docs.nvidia.com/deploy/mps/index.html#volta-mps-execution-resource-provisioning, 2025.

[38] NVIDIA. Nsight compute metrics decoder. https://docs.nvidia.com/nsight-compute/ProfilingGuide/index.html#metrics-decoder, 2025.

[39] NVIDIA. Special register smid. https://docs.nvidia.com/cuda/parallel-thread-execution/#special-registers-smid, 2025.

[40] Pratyush Patel, Esha Choukse, Chaojie Zhang, Íñigo Goiri, Brijesh Warrier, Nithish Mahalingam, and Ricardo Bianchini. Characterizing power management opportunities for llms in the cloud. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ASPLOS '24, page 207–222, New York, NY, USA, 2024. Association for Computing Machinery.

[41] PyTorch. torch.mm. https://pytorch.org/docs/stable/generated/torch.mm.html, 2024.

[42] Siddharth Samsi, Dan Zhao, Joseph McDonald, Baolin Li, Adam Michaleas, Michael Jones, William Bergeron, Jeremy Kepner, Devesh Tiwari, and Vijay Gadepally. From words to watts: Benchmarking the energy costs of large language model inference. In *2023 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–9, 2023.

[43] Sudipta Saha Shubha, Haiying Shen, and Anand Iyer. USHER: Holistic interference avoidance for resource optimized ML inference. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 947–964, Santa Clara, CA, July 2024. USENIX Association.

[44] Jovan Stojkovic, Chaojie Zhang, Íñigo Goiri, Josep Torrellas, and Esha Choukse. Dynamollm: Designing llm inference clusters for performance and energy efficiency, 2024.

[45] Foteini Strati, Xianzhe Ma, and Ana Klimovic. Orion: Interference-aware, fine-grained gpu sharing for ml applications. In *Proceedings of the Nineteenth European Conference on Computer Systems*, EuroSys '24, page 1075–1092, New York, NY, USA, 2024. Association for Computing Machinery.

[46] Xiaodan Serina Tan, Pavel Golikov, Nandita Vijaykumar, and Gennady Pekhimenko. Gpupool: A holistic approach to fine-grained gpu sharing in the cloud. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, PACT '22, page 317–332, New York, NY, USA, 2023. Association for Computing Machinery.

[47] Gemma Team. Gemma 3. 2025.

[48] Qizhen Weng, Wencong Xiao, Yinghao Yu, Wei Wang, Cheng Wang, Jian He, Yong Li, Liping Zhang, Wei Lin, and Yu Ding. MLaaS in the wild: Workload analysis and scheduling in Large-Scale heterogeneous GPU clusters. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 945–960, Renton, WA, April 2022. USENIX Association.

[49] Bingyang Wu, Zili Zhang, Zhihao Bai, Xuanzhe Liu, and Xin Jin. Transparent GPU sharing in container clouds for deep learning workloads. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 69–85, Boston, MA, April 2023. USENIX Association.

[50] Wencong Xiao, Romil Bhardwaj, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, Zhenhua Han, Pratyush Patel, Xuan Peng, Hanyu Zhao, Quanlu Zhang, Fan Yang, and Lidong Zhou. Gandiva: Introspective cluster scheduling for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 595–610, Carlsbad, CA, October 2018. USENIX Association.

[51] Wencong Xiao, Shiru Ren, Yong Li, Yang Zhang, Pengyang Hou, Zhi Li, Yihui Feng, Wei Lin, and Yangqing Jia. AntMan: Dynamic scaling on GPU clusters for deep learning. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 533–548. USENIX Association, November 2020.

[52] Fei Xu, Jianian Xu, Jiabin Chen, Li Chen, Ruitao Shang, Zhi Zhou, and F. Liu. igniter: Interference-aware gpu resource provisioning for predictable dnn inference in the cloud. *IEEE Transactions on Parallel and Distributed Systems*, 34:812–827, 2022.

[53] Gingfung Yeung, Damian Borowiec, Adrian Friday, Richard Harper, and Peter Garraghan. Towards GPU utilization prediction for cloud deep learning. In *12th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 20)*. USENIX Association, July 2020.

[54] Gingfung Yeung, Damian Borowiec, Renyu Yang, Adrian Friday, Richard Harper, and Peter Garraghan. Horus: Interference-aware and prediction-based scheduling in deep learning systems. *IEEE Transactions on Parallel and Distributed Systems*, 33(1):88–100, 2022.

[55] Peifeng Yu and Mosharaf Chowdhury. Salus: Fine-grained GPU sharing primitives for deep learning applications. *CoRR*, abs/1902.04610, 2019.

[56] Yongkang Zhang, Haoxuan Yu, Chenxia Han, Cheng Wang, Baotong Lu, Yang Li, Xiaowen Chu, and Huaicheng Li. Missile: Fine-grained, hardware-level gpu resource isolation for multi-tenant dnn inference, 2024.

[57] Zhen Zhang, Chaokun Chang, Haibin Lin, Yida Wang, Raman Arora, and Xin Jin. Is network the bottleneck of distributed training? In *Proceedings of the Workshop on Network Meets AI & ML*, NetAI '20, page 8–13, New York, NY, USA, 2020. Association for Computing Machinery.

[58] Wenyi Zhao, Quan Chen, Hao Lin, Jianfeng Zhang, Jingwen Leng, Chao Li, Wenli Zheng, Li Li, and Minyi Guo. Themis: Predicting and reining in application-level slowdown on spatial multitasking gpus. In *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 653–663, 2019.

[59] Xia Zhao, Magnus Jahre, and Lieven Eeckhout. Hsm: A hybrid slowdown model for multitasking gpus. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, page 1371–1385, New York, NY, USA, 2020. Association for Computing Machinery.

[60] Yihao Zhao, Xin Liu, Shufan Liu, Xiang Li, Yibo Zhu, Gang Huang, Xuanzhe Liu, and Xin Jin. Muxflow: Efficient and safe gpu sharing in large-scale production deep learning clusters, 2023.

[61] Zhihe Zhao, Neiwen Ling, Nan Guan, and Guoliang Xing. Miriam: Exploiting elastic kernels for real-time multi-dnn inference on edge gpu. In *Proceedings of the 21st ACM Conference on Embedded Networked Sensor Systems*, SenSys '23, page 97–110, New York, NY, USA, 2024. Association for Computing Machinery.

[62] Kan Zhu, Yilong Zhao, Liangyu Zhao, Gefei Zuo, Yile Gu, Dedong Xie, Yufei Gao, Qinyu Xu, Tian Tang, Zihao Ye, Keisuke Kamahori, Chien-Yu Lin, Stephanie Wang, Arvind Krishnamurthy, and Baris Kasikci. Nanoflow: Towards optimal large language model serving throughput, 2024.