Measuring GPU utilization 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

GPU hardware is vastly underutilized. Even resource-intensive AI applications have diverse resource profiles that often leave parts of GPUs idle. While colocating applications can improve utilization, current spatial sharing systems lack performance guarantees. 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 propose a methodology to profile resource interference of GPU kernels across these dimensions and discuss how to build GPU schedulers that provide strict performance guarantees while colocating applications to minimize cost.

1 INTRODUCTION

Graphics Processing Units (GPUs) are widely used for applications like AI training and inference to maximize performance per Watt. However, GPUs are expensive and power-hungry, with AI workloads' power needs composing a significant percentage of datacenter grids [38, 40, 42]. To minimize total cost of ownership and make optimal use of the limited power budget, users should operate GPU clusters at high utilization. Yet many recent studies show that GPUs are vastly underutilized [6, 8, 9, 15, 43, 45, 48, 57]. Even when serving GB-scale LLMs with large batch sizes, some GPU components may be idle as resource requirements vary across compute vs. memory intensive phases of a job [12, 59]. For example, Microsoft reports less than 10% compute utilization during the memory-bound decoding phase of serving the Llama3-8B model on A100 GPUs [12]. GPUs can also be underutilized due to small batch sizes, communication, data preprocessing bottlenecks, and checkpointing [6, 7, 54].

GPU utilization can be improved with spatial colocation of multiple applications, that is, allowing more than one workload to execute on a GPU concurrently [8, 9, 13, 15, 41, 43, 57]. However, colocating applications leads to interference; this resource contention increases the execution time of individual GPU kernels. While prior GPU schedulers that support spatial sharing aim to minimize interference, no system currently provides reliable performance guarantees. Most systems rely on limited metrics to evaluate GPU resource utilization and define colocation strategies. For example, Orion [43] uses the SM utilization and roofline models of individual kernels to colocate kernels with complementary resource profiles. Usher [41] considers each kernel's achieved occupancy as an indicator of the kernel's computational requirements, and requires the sum of achieved occupancy values to be lower than 100% for colocation. However, as we show in §3, these state-of-the-art systems oversimplify GPU utilization and interference modeling, resulting in colocation decisions that can significantly degrade application performance.

Inspired by Ousterhout's advice to always measure one level deeper [37] and the iBench [5] microbenchmark suite for quantifying interference on datacenter CPU servers, we conduct a series of experiments to more deeply understand GPU utilization and its role in the design of GPU scheduling systems. GPUs are inherently heterogeneous, comprising various components such as streaming multiprocessors, warp schedulers, tensor cores, high-bandwidth memory, caches, and register files. We use microbenchmarks to demonstrate the different resources where interference can occur, and explain how users can identify whether a kernel is susceptible to each kind of interference. Finally, we discuss how our insights pave the way to designing software and hardware GPU schedulers that effectively colocate applications to reduce cost while providing strict performance guarantees.

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

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

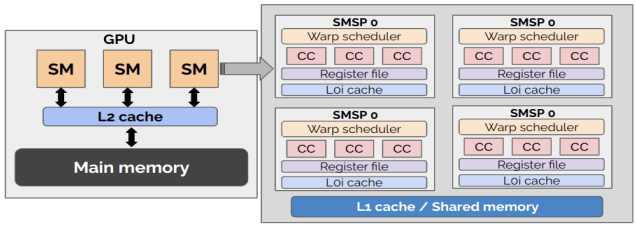

Figure 1: Simplified diagram of an NVIDIA GPU (based on an H100), focusing on a Streaming Multiprocessor.

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) [36]. The mapping of pipelines to compute units is hidden from users, though there have been attempts to reverse engineer it [18, 23, 24, 32]. The GPU also contains a main memory, shared by all SMs, and accessed through a two-level on-chip cache hierarchy. The L2 cache is shared across all SMs, while the L1 cache is private to each SM and shared among its subpartitions. The L1 cache can partially be configured as software-defined shared memory [22].

Threads, blocks, and warps. GPU programs consist of CUDA kernels, which are executed by one or more GPU threads [20]. 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 can communicate through shared memory and can 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 will map 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). A block will be 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. Second, at the subpartition level, active warps are divided into *eligible* and *stalled* warps based on their ability to issue an instruction. A warp is eligible if its instruction has been fetched, all dependencies are met, and the required functional units are available. Each clock cycle, the warp scheduler in each subpartition chooses one eligible warp and schedules one or more instructions from that warp.

2.2 Utilization metrics

We describe several GPU utilization metrics reported by NVIDIA tools such as Nsight Compute (NCU) [25].

GPU utilization from `nvidia-smi`/NVML [28, 30] depicts the percentage of time a kernel is active on a GPU, without revealing how well this kernel utilizes the various GPU resources. This metric is used by various works [3, 10, 46, 50, 51].

SM utilization refers to the amount of SMs needed by a kernel, and can be found by taking into account the kernel's grid size, block size, registers/thread, shared memory/thread and the respective SM thresholds of a GPU (as explained in 2.1). Orion [43] and REEF [9] 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 [43] 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 [41].

Pipe utilization captures the utilization of GPU pipelines (FMA, Tensor Cores, etc). We take the `sm__inst_executed_pipe_*.avg.pct_of_peak_sustained_active` metric from NCU (where * indicates the pipeline, e.g. `fma`, `tensor`). It expresses how effectively a pipeline is used (when executing at least one warp) relative to its peak capability, averaged across all SMs that executed at least one warp.

Issued instructions per cycle (IPC): The NCU metric `sm__inst_issued.avg.per_cycle_active` (also called `IPC` [34]) represents the number of warp instructions issued per cycle per SM. This metric averages IPC across all active SMs (i.e., SMs that have at least one warp scheduled on them). Our GPUs have 4 subpartitions per SM, and each subpartition has a warp scheduler capable of issuing one warp instruction per cycle, i.e the maximum IPC per SM is 4.

GPU workload distribution: The `sm__cycles_active.[avg/min/max]` metric from NCU measures the cycles with at least one active warp per SM, assessing kernel execution balance across SMs. Large discrepancies between average, minimum, and maximum values indicate unbalanced GPU utilization, with SMs idling. Metrics as IPC, pipe utilization and achieved occupancy should be combined with workload distribution to ensure balanced workload partitioning.

3 PITFALLS OF GPU SCHEDULERS

Multiple GPU schedulers aim to improve utilization by colocating workloads. Temporal-sharing schedulers, such as Clockwork [8], Gandiva [47], Salus [52], Antman [48], and TGS [46] execute one workload at a time to avoid resource interference, but fail to address single-workload GPU underutilization,

and can cause severe queuing delays [43]. In contrast, spatial sharing systems such as Usher [41], Orion [43], REEF [9], Igniter [49], Missile [53], and Zico [14] allow concurrent workload execution, and propose strategies to minimize interference. We identify common pitfalls in state-of-the-art GPU schedulers, caused by their reliance on only a *subset* of GPU metrics, leading to a lack of performance guarantees.

Pitfall 1: Not taking IPC into account: The issued instructions per cycle (IPC) metric (see §2.2) measures warp scheduler utilization. Ignoring IPC can lead to severe interference. We illustrate the significance of IPC using the Orion scheduler [43] as an example. Orion colocates kernels with complementary resource profiles (i.e., compute vs. memory bound kernels), which it determines based on arithmetic intensity. 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 use a `compute` and a `copy` kernel, which perform independent element-wise fp32 multiplication and array copying, respectively, for several iterations. We use a 4096-byte input array for `compute` and a 4GB input and output array for `copy`. We tune the number of iterations so that the two kernels have similar execution times. We launch the kernels with 132 blocks and 1024 threads/block. 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 `compute` has an IPC of 4, which leads to warp scheduling interference, as we will detail in §4.2.2.

Pitfall 2: Relying on achieved occupancy: Another pitfall is relying on a single metric, such as achieved occupancy, to assess a kernel's compute requirements. Achieved occupancy can be misleading, as a kernel can saturate GPU resources even with low achieved occupancy. For instance, Usher [41] colocates two kernels if the sum of achieved occupancy values is $< 100\%$. As a counterexample, we launch two instances of the `compute` kernel (same as the previous example), with 132 blocks and 128 threads/block per kernel. 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` [17] variable. We observe a 15.8× 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 1.85× 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.

NVIDIA GPU	Num SMs	CUDA version	Driver version
H100 NVL [27]	132	12.5	555.42.06
GeForce RTX3090 [19]	82	12.6	560.35.03

Table 1: Hardware Setup used in all experiments.

We demonstrated two key pitfalls in state-of-the-art schedulers, but to our knowledge, the problem of focusing on only a subset of GPU resources is common among all schedulers. For example, REEF [9] considers only SM utilization but overlooks L2 cache or memory bandwidth interference (see §4.1). Approaches relying on `nvidia-smi` to measure GPU utilization [3, 10, 46, 50, 51] might falsely consider a GPU fully utilized even if only a single SM is occupied. These observations 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, both across SMs (§4.1) and within an SM (§4.2). An interference-aware GPU scheduler should take all these resources into account to give reliable performance guarantees. We demonstrate interference at each level with microbenchmarks, and explain how users can identify whether a kernel is susceptible to this kind of interference. Finally, we show an example of how our methodology can identify which types of interference a real kernel from a representative AI workload is susceptible to (§4.3). Table 1 shows our hardware setup. Unless otherwise stated, all experiments use CUDA streams [16] for colocating two kernels on the GPU.

4.1 Inter-SM interference

4.1.1 Block scheduler interference. As described in §2, the block scheduler assigns blocks to SMs, as long as they meet all resource constraints. When any of these SM resources are insufficient, it schedules blocks sequentially as resources free up, increasing latency. We demonstrate this on an H100 GPU using a kernel that iteratively calls the `nanosleep` function [21]. We launch two instances of this kernel, each with 1024 threads per block (allowing each SM to host up to 2 blocks). NCU reveals low IPC (< 0.2) and pipe utilization for each kernel. With 132 thread blocks per kernel, the two kernels execute in different streams with perfect overlap, showing no interference. However, when launching with 264 thread blocks per kernel, the colocated kernel latency matches the sequential kernel latency, despite the kernel's

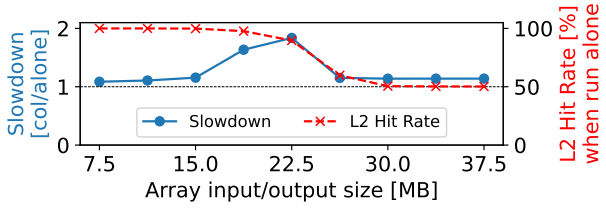

Figure 2: L2 cache interference on an H100

minimal resource usage. This is due to a single kernel saturating the 2048 threads/SM limit on the H100, preventing concurrent block execution between kernels. The metrics `launch__occupancy_limit_[blocks/warps/registers/shared_mem]` in NCU provide insights into a kernel's susceptibility to block scheduler interference.

4.1.2 L2 cache interference. To examine interference in the L2 cache (shared between SMs), we colocate two instances of the copy kernel (from §3) on the H100, each using 66 thread blocks and 1024 threads. This allows both kernels to run on separate SMs, eliminating intra-SM interference. Figure 2 shows the slowdown of the copy kernel when colocated compared to running alone as array size increases. For small arrays, colocation causes minimal performance impact. However, the slowdown increases until an array size of 22.5MB, reaching 1.79×, indicating L2 cache interference. After this point, the kernel's L2 cache hit rate in isolation starts to drop significantly, meaning that the kernel will have to go through main memory due to L2 cache misses. This reduces the impact of additional L2 contention, caused by colocation. This is why the slowdown is reduced after that point.

We observe a correlation between L2 cache hit rate and susceptibility to L2 cache interference. Thus, for identifying a kernel's sensitivity to L2 cache interference, the L2 hit rate from NCU (`lts__t_sector_hit_rate.pct`) can be used.

4.1.3 Memory bandwidth interference. To examine memory bandwidth interference, we colocate two instances of the copy kernel in separate processes, each copying a 4GB input array (exceeding L2 cache size). We use MPS to allocate a non-overlapping set of 50% of SMs to each process to eliminate intra-SM interference. Table 2 shows the slowdown of the copy kernel compared to running alone as we increase the number of thread blocks. [2] On the RTX3090, a single process achieves up to 90% of the max memory bandwidth when run alone. It suffers from an approximate 1.9× slowdown when colocated. On the H100, a single process achieves around 69% of the max memory bandwidth, and suffers from a 1.36× slowdown when colocated. Thus, despite kernels running on distinct sets of SMs, they may suffer from memory bandwidth interference, as found in prior works [13, 43, 53].

[2]Setting an MPS share of 50% on the RTX3090 will result in 40 SMs available to the process.

Thread Blocks / Kernel H100	**33**	**66**	**99**	**132**
Memory Bandwidth Utilization [%]	21.33	40.12	55.06	69.18
Slowdown [col/alone]	1.06	1.14	1.23	1.36
Thread Blocks / Kernel RTX3090	**10**	**20**	**40**	**80**
Memory Bandwidth Utilization [%]	24.48	45.34	71.96	90.69
Slowdown [col/alone]	1.06	1.27	1.61	1.91

Table 2: Memory Bandwidth Interference. Slowdown of copy kernel under colocation over running alone on H100/RTX3090 on 50% of the SMs using MPS. Reported Memory Bandwidth Utilization is relative to the highest actual achieved memory bandwidth.

NCU's memory bandwidth utilization metric follows our measured utilization trends (i.e. increases with grid size), but is lower (e.g. 42% with 132 blocks on the H100), due to the actual achieved maximum memory bandwidth being lower than the theoretical. We recommend using our microbenchmark to identify if a kernel is susceptible to memory bandwidth interference, as NCU might underestimate a kernel's memory bandwidth usage.

4.2 Intra-SM interference

4.2.1 L1 Cache Interference. The L1 cache, shared among SM subpartitions, can create contention between colocated kernels. To demonstrate this, we colocate two instances of the copy kernel. We make sure each thread block accesses memory aligned to 256KB.[3] Figure 3 shows the average latency for colocated or sequential execution of two kernels. We launch them with 132 thread blocks and 64 threads per block each, to ensure that no two warps run on the same SMSP thereby eliminating any source of interference from within the SMSP. Colocation is beneficial over sequential execution for arrays smaller than 64KB, but not when input and output arrays exceed the L1 cache (256KB). Sequential execution sees an inflection at 108KB, while colocation shows it at the half point (54KB) due to processing double the data. Thus, kernel colocation can substantially increase L1 cache misses and lead to performance degradation. To detect if a kernel is susceptible to L1 cache interference, a user can look at the L1 cache hit rate (`l1tex__t_sector_hit_rate.pct`).

4.2.2 IPC interference. We demonstrate how the architectural limit of 4 instr/cycle/SM can become a bottleneck, using the copy kernel and modified versions of the compute kernel across five scenarios, shown in Table 3. copy processes a 4GB array using 132 thread blocks and 1024 threads per block,

[3]The unified L1 cache size on the H100.

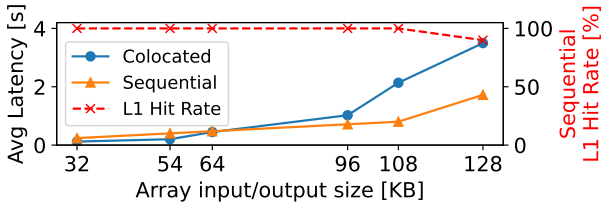

Figure 3: L1 cache interference on an H100

	S_1	S_2	S_3	S_4	S_5
IPC `compute` [instr/cycle/SM]	1.08	2.06	2.9	3.45	4
Speedup [seq/col]	1.915	1.896	1.893	1.65	1.001

Table 3: IPC Interference on H100. Speedup of colocating `copy` and `compute` over running them sequentially.

maintaining an IPC of 0.61 across all scenarios. In scenarios S_1-S_4 we increase Instruction Level Parallelism (ILP) for the `compute` kernel from 1 to 4 independent `__fmul_rn`[29] operations and launch it with 132 blocks and 128 threads/block, ensuring one warp per SMSP. S_5 maintains four operations but doubles the thread count to 256. Higher ILP or increased threads per block increase IPC. We adjusted iteration counts for comparable colocated runtimes across all scenarios. Table 3 shows a colocation speedup of 1.9× while the aggregated IPC remains below 4. However, the speedup diminishes in S_4, when $IPC > 4$, disappearing entirely in S_5 (the Orion example from §3). This demonstrates warp scheduler interference between compute-bound and memory-bound kernels despite their distinct resource dependencies.

4.2.3 Pipeline interference. To demonstrate pipeline interference, we modify our `compute` kernel to utilize fp64 multiplication `__dmul_rn` [26]. We examine scenarios S_1 to S_4 increasing ILP from one to four. We colocate two kernel instances, each with 132 thread blocks and 128 threads per block, ensuring one warp per kernel on each SMSP. Table 4 shows that when the aggregated FP64 pipeline utilization remains below 100% (S_1 and S_2), colocation achieves a nearly 2× speedup. However, the speedup decreases significantly as utilization exceeds 100% (S_3 and S_4), despite IPC not being a bottleneck. Each kernel's achieved occupancy is 6.25% in all scenarios. This shows why the metric alone is insufficient for colocation decisions, as it fails to capture how effectively even small warp counts can saturate specific GPU components, as shown in §3 with the Usher example.

4.3 Example with real ML kernel

In §4.1 and §4.2, we used microbenchmarks to stress specific GPU resources and demonstrate interference at different

	S_1	S_2	S_3	S_4
IPC `compute` [instr/cycle/SM]	0.51	1.02	1.53	2.01
FP64 Pipe Utilization [%]	24.96	49.32	72.99	96.59
Speedup [seq/col]	1.988	1.977	1.356	1.018

Table 4: Pipeline Interference. Speedup of colocating two FP64 `compute` kernels over running them sequentially on H100. IPC and FP64 Pipe Utilization correspond to profiling `compute` in isolation.

levels. We now show that we can apply our methodology to any CUDA kernel, including kernels from real ML workloads.

We analyze the `torch.mm` function, a widely-used PyTorch operator for matrix-matrix multiplication in ML workloads [39]. This operator invokes a cuBLAS GEMM kernel, whose grid and block size depend on the input tensors' dimensions, and its code is closed-source.

Using input tensors of 4 MB each, results in the kernel being launched with 128 blocks and 128 threads/block. We verified in NCU that the launch configuration as well as shared memory/register usage allow blocks from two kernel instances to run concurrently. NCU profiling shows an IPC of 3.0 and 60% FMA pipeline utilization per kernel, thus we anticipate intra-SM interference due to high IPC and pipeline utilization. Indeed, when the kernels are colocated, each kernel's latency increases by 1.7×.

5 RESEARCH OUTLOOK

Towards an interference-aware GPU scheduler: Our methodology for identifying multi-dimensional GPU resource contention can be applied to implement an interference-aware GPU scheduler that overcomes the pitfalls of related work and provides strict performance guarantees. The first step is to develop 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 take a subset of interference sources into account [49, 56]. Themis [55] and GPUPool [44] 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. The ultimate goal is a GPU scheduler that colocates jobs based on the workload interference estimator. The scheduler should take into account

factors such as RPS, SLOs, and queueing delays to come up with optimal colocation policies that satisfy user objectives.

Hardware wishlist for GPU spatial 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. Additionally, a programmer-friendly way to partition SMs and DRAM channels at the kernel level can mitigate intra-SM and DRAM bandwidth interference. While MPS allows limiting applications to a set of SMs, it is quite inelastic due to its coarse granularity (whole workload) [17]. Related work proposes limiting a kernel's blocks to specific SMs and DRAM channels (or alter the grid size), but they are code-intrusive and unsuitable for ML workloads with closed-source kernels [4, 11, 53, 58]. Finally, enabling kernel preemptibility could improve kernel colocation, especially in real-time tasks, as shown by REEF [9] for AMD GPUs.

6 CONCLUSION

We highlight common pitfalls of state-of-the-art GPU schedulers and propose a methodology for characterizing utilization across heterogeneous GPU components and analyzing interference. Based on these insights, we outline a research agenda and hardware requirements for achieving spatial GPU sharing with strict performance guarantees.

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] Christina Delimitrou and Christos Kozyrakis. iBench: Quantifying Interference for Datacenter Workloads. In *Proceedings of the 2013 IEEE International Symposium on Workload Characterization (IISWC)*, September 2013.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

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

[11] 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.

[12] 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.

[13] Sejin Kim and Yoonhee Kim. K-scheduler: dynamic intra-sm multitasking management with execution profiles on gpus. *Cluster Computing*, 25(1):597–617, feb 2022.

[14] 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.

[15] 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.

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

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

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

[19] NVIDIA. Nvidia geforce rtx3090. https://www.nvidia.com/content/PDF/nvidia-ampere-ga-102-gpu-architecture-whitepaper-v2.pdf, 2021.

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

[21] NVIDIA. Cuda, nanosleep function. https://docs.nvidia.com/cuda/cuda-c-programming-guide/#nanosleep-function, 2024.

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

[23] 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.

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

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

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

[27] NVIDIA. Nvidia h100 nvl gpu. https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/h100/PB-11773-001_v01.pdf, 2024.

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

[29] NVIDIA. Nvidia single precision intrinsics. https://docs.nvidia.com/cuda/cuda-math-api/cuda_math_api/group__CUDA__MATH__INTRINSIC__SINGLE.html, 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. Nsight compute metrics decoder. https://docs.nvidia.com/nsight-compute/ProfilingGuide/index.html#metrics-decoder, 2025.

[37] John Ousterhout. Always measure one level deeper. *Commun. ACM*, 61(7):74–83, June 2018.

[38] 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.

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

[40] 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.

[41] 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.

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

[43] 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.

[44] 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.

[45] 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.

[46] 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.

[47] 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.

[48] 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.

[49] 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.

[50] 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.

[51] 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.

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

[53] 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.

[54] 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.

[55] 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.

[56] 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.

[57] 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.

[58] 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.

[59] 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.