LithOS: An Operating System for Efficient Machine Learning on GPUs

Patrick H. Coppock, Brian Zhang, Eliot H. Solomon, Vasilis Kypriotis, Leon Yang[†], Bikash Sharma[†], Dan Schatzberg[†], Todd C. Mowry, and Dimitrios Skarlatos

Carnegie Mellon University   [†]Meta

Abstract

The surging demand for GPUs in datacenters for machine learning (ML) workloads has made efficient GPU utilization crucial. However, meeting the diverse needs of individual ML models while optimizing resource usage is challenging. To enable transparent, fine-grained management of GPU resources that maximizes GPU utilization and energy efficiency while maintaining strong isolation, an operating systems (OS) approach is needed. Hence this paper introduces LithOS, a first step towards a GPU OS.

LithOS includes the following new abstractions and mechanisms for efficient GPU resource management: (i) a novel *TPC Scheduler* that supports spatial scheduling at the granularity of individual TPCs, unlocking efficient TPC stealing between workloads; (ii) transparent *kernel atomization* to reduce head-of-line blocking and allow dynamic resource reallocation mid-execution; (iii) a lightweight *hardware right-sizing* mechanism that dynamically determines the minimal TPC resources needed per atom; and (iv) a transparent *power management* mechanism that reduces power consumption based upon in-flight work characteristics.

We implement LithOS in Rust and evaluate its performance across a broad set of deep learning environments, comparing it to state-of-the-art solutions from NVIDIA and prior research. For inference stacking, LithOS reduces tail latencies by 13× compared to MPS; compared to the best-performing SotA, it reduces tail latencies by 3× while improving aggregate throughput by 1.6×. Furthermore, in hybrid inference-training stacking, LithOS reduces tail latencies by 4.7× compared to MPS; compared to the best-performing SotA, it reduces tail latencies by 1.18× while improving aggregate throughput by 1.35×. Finally, for a modest performance hit under 4%, LithOS's hardware right-sizing provides a quarter of GPU capacity savings on average, while for a 7% hit, LithOS's transparent power management delivers a quarter of a GPU total energy savings on average. Overall, LithOS transparently increases GPU efficiency, establishing a foundation for future OS research on GPUs.

1 Introduction

The widespread adoption of machine learning (ML) workloads has led to massive GPU deployments across datacenters. However, despite growing concerns around power consumption and hardware supply constraints, GPU resources
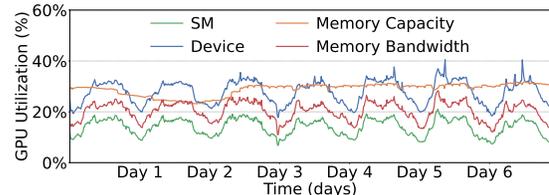

Figure 1. *GPU utilization metrics over a week in a production Ads inference service at Meta.*

remain significantly underutilized. Public reports from Microsoft and Alibaba cite average and median GPU utilization rates of just 52% [23] and 10% [49], respectively. Our analysis of a production *Ads* service at Meta reveals similarly low utilization, averaging just 27%, as shown in Figure 1. Given the high monetary cost and rising power demands—now exceeding 1,000W per GPU [26, 31]—this is unsustainable.

It is challenging to achieve high utilization without GPU sharing. While dedicating a GPU to a single workload leads to high performance, individual workloads often fail to keep the GPU fully utilized: GPU cores idle on communication stalls, low batch sizes result in insufficient parallelism, dynamic request loads lead to overprovisioning, and so on [18, 20, 49]. As GPUs become more powerful with increasing streaming multiprocessor (SM) counts and memory bandwidth [8, 31], achieving high utilization will become more challenging.

One potential approach to GPU sharing is collocating *latency-critical* (LC) tasks for which performance is of utmost importance with *best-effort* (BE) tasks that lack hard deadlines. However, existing systems do not offer a practical solution for prioritizing LC tasks over BE tasks when they contend for resources. Many approaches lack transparency, rendering them incompatible with large parts of the ML software stack [1, 12, 13, 19, 20, 22, 29, 32, 34, 39, 41, 44]. For instance, some are tied to specific versions of frameworks like PyTorch or TVM that are no longer maintained [1, 13, 19, 44, 49]. Other solutions like TGS [48] or Clockwork [19] fall short of achieving high GPU utilization due to limited temporal scheduling that cannot execute multiple models in parallel. Spatial scheduling solutions, including NVIDIA's MPS [7] and MIG [10] or research efforts like REEF [20], Orion [44],

P. H. Coppock, B. Zhang, E. H. Solomon, V. Kypriotis, L. Yang, B. Sharma, D. Schatzberg, T. C. Mowry, and D. Skarlatos

and others [13], enable parallel execution of multiple applications. However, they are too coarse-grained, scheduling entire inference requests, training batches, or DNN operators, resulting in low utilization and head-of-line (HoL) blocking [1, 4, 13, 15, 20, 27, 29, 32, 39, 44, 45, 48, 50]. Efficient multitenant scheduling on GPUs has remained elusive.

Beyond collocation mechanisms, to address GPU inefficiencies without sacrificing performance or transparency, datacenter GPU management must evolve past static provisioning. Current systems fail to account for changing deep learning workload characteristics—such as fluctuating levels of compute intensity and parallelism across different models and execution phases. This is another reason why GPUs are often not efficiently utilized, even as they draw significant power. Bridging this gap requires new approaches that can adapt resource allocation and power consumption to the fine-grained characteristics of ML workloads.

This utilization crisis is in stark contrast with the situation for CPUs, where time-sharing operating systems allocate tasks to cores via inexpensive context switches, providing isolation, resource allocation, power management, and transparency. The extreme data-parallel nature of GPUs imposes different trade-offs than do CPUs, but also exposes the limitations of current abstractions built around compilers, frameworks, and drivers. To transparently improve utilization and efficiency, we believe that GPUs must evolve toward an operating system model—one that brings first-class support for control, isolation, and resource management.

1.1 Our Approach: An Operating System for GPUs

To address datacenter GPU efficiency challenges, we introduce LithOS, which brings an operating system approach to deep learning on GPUs. LithOS is fully transparent to the ML stack, allowing seamless integration without requiring any modifications to models, runtimes, or frameworks. LithOS moves the bulk of GPU scheduling from proprietary drivers and hardware into software, allowing, for the first time, fine-grained temporal and spatial scheduling of ML workloads. LithOS operates at the granularity of individual kernel thread blocks that are dynamically mapped onto the GPU's thread processing clusters (TPCs). To achieve this, LithOS introduces novel abstractions and mechanisms that decouple kernel work submission from thread block execution on GPUs, enabling intelligent scheduling decisions, resource allocation, and power management.

First, LithOS introduces a novel fine-grained *TPC Scheduler* that asynchronously determines the compute unit allocation and submission time for each piece of work. It enables precise control at the granularity of individual TPCs, providing strong isolation between workloads. The scheduler is guided toward efficient scheduling decisions by an online kernel latency predictor and incorporates a technique called *TPC Stealing* to improve GPU utilization.

To overcome the lack of hardware preemption, LithOS introduces a *kernel atomization* mechanism that transparently—without compiler, runtime, source, or PTX code modifications—splits kernels into independently schedulable units called *atoms*. Each atom consists of a subset of a kernel's thread blocks, reducing head-of-line blocking and cross-workload interference. Atomization also enables LithOS to dynamically reconfigure TPC allocations mid-execution, allowing scheduling flexibility that is impossible with monolithic kernels.

Building on this foundation, LithOS introduces a dynamic *hardware right-sizing* mechanism that uses lightweight modeling to determine the minimal TPC resources required for each kernel and its atoms, yielding significant capacity savings. Finally, LithOS presents a fine-grained *power management* mechanism that adjusts the GPU's frequency in response to the characteristics of in-flight work, achieving substantial energy savings.

We implement LithOS in Rust and evaluate its performance across a broad set of deep learning environments, comparing it to state-of-the-art solutions from NVIDIA and prior research. For inference stacking, LithOS reduces tail latencies by 13× compared to MPS; compared to the best-performing SotA, it reduces tail latencies by 3× while improving aggregate throughput by 1.6×. Furthermore, in hybrid inference-training stacking, LithOS reduces tail latencies by 4.7× compared to MPS; compared to the best-performing SotA, it reduces tail latencies by 1.18× while improving aggregate throughput by 1.35×. Finally, for a modest performance hit under 4%, LithOS's hardware right-sizing provides a quarter of GPU capacity savings on average, while for a 7% hit, LithOS's transparent power management delivers a quarter of a GPU total energy savings on average.

Overall, LithOS transparently increases GPU efficiency, establishing a foundation for future OS research on GPUs.

This paper makes the following contributions:

- A comprehensive study of an *Ads* inference service at Meta, highlighting the behavior of production ML models and the challenges of GPU underutilization.
- A fine-grained spatial *TPC Scheduler* that dynamically allocates TPCs using *TPC Stealing* to boost utilization.
- A transparent *Kernel Atomizer* that independently schedules subsets of kernel thread blocks, unlocking efficiency.
- A dynamic *hardware right-sizing* mechanism that optimizes TPC allocations for significant capacity savings.
- A transparent *power management* mechanism that adjusts frequency based on kernel characteristics to save energy.
- The design of LithOS, a step towards an OS for GPUs.
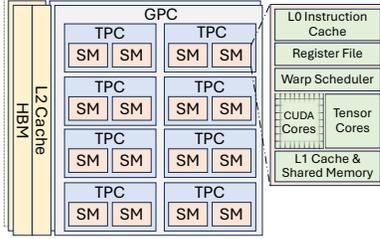- The evaluation of LithOS across varied ML environments.

Figure 2. *GPU Architecture.*

Figure 3. *GPU timeline showcasing the pitfalls of MPS.*

2 Background and Related Work

In this section, we first present a brief background on NVIDIA GPU architectures and then cover related work.

2.1 A Brief Background on GPUs

GPU Architecture. Modern GPUs have immense hardware resources catering to the needs of ML workloads. Figure 2 depicts a typical GPU architecture. Each GPU consists of several General Processing Clusters (GPCs). Each GPC is a collection of multiple Thread Processing Clusters (TPCs) and each TPC includes a small number of Streaming Multiprocessors (SMs). Each SM is composed of tens of cores. For example, NVIDIA's H100 [8] includes 8 GPCs, 9 TPCs per GPC, 2 SMs per TPC, and 128 cores per SM.

GPU Programming. GPU applications are composed of kernels that execute specific operators (e.g., convolution). A kernel defines its resources—thread blocks, threads, registers, and shared memory—at launch time. Programmers divide a kernel's work among the thread blocks. Each thread block executes on an SM and consists of multiple SIMD threads.

GPU Streams. CUDA streams enable concurrent execution of independent tasks, similar to CPU threads. Stream work is executed in FIFO order. Some CUDA calls are asynchronous, while others wait for all previous tasks to finish.

2.2 Related Work

Cooperative multitenancy. Cooperative scheduling involves tenants coordinating to share resources, typically at the ML framework level, with all models running in the same process [1, 12, 13, 19, 20, 22, 29, 32, 34, 39, 41, 44]. These approaches require custom ML frameworks and are hence limited by their inability to support arbitrary applications. Some also rely on extensive offline profiling [20, 44] or kernel source modifications [20, 32] which are impractical at scale. Finally, any non-cooperating tenant invalidates guarantees made by the runtime, making adoption difficult in practice.

Transparent multitenancy. Transparent GPU sharing solutions support unmodified applications. They include native mechanisms like time slicing, MPS [7], and MIG [10] offered by NVIDIA. Nearly all prior software solutions are not transparent and rely on application or framework modifications. TGS [48] is one exception that offers transparent
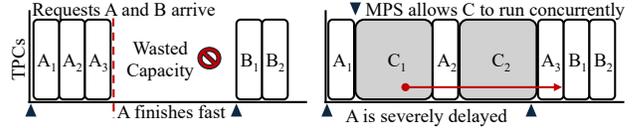
sharing between containerized applications. In practice, uncooperative tasks and limited application-specific information make transparent multitasking a serious challenge.

Temporal multitenancy. Temporal multitenancy dedicates the entire GPU to a single task at a time via native time slicing or software scheduling. Some approaches work at the level of entire inference requests (e.g., Clipper [12], Nexus [41], TensorFlow-Serving [34], Clockwork [19], and INFaaS [39]), while others schedule individual GPU kernels (e.g., PipeSwitch [1], AntMan [49], and TGS [48]). *Time slicing* is NVIDIA's default temporal multitenancy solution. It shares the GPU in a round-robin fashion, giving each task exclusive access for several milliseconds. These methods execute only one job at a time, leading to low utilization.

Spatial multitenancy. Spatial multitenancy typically builds on MIG or MPS to enable multiple applications to run concurrently on a GPU and improve utilization. *MPS* multiplexes multiple GPU contexts onto one, allowing multiple tasks to use the GPU concurrently. This can yield greater throughput but leads to performance interference. *MIG* partitions the GPU's compute and memory resources along GPC boundaries, providing strong hardware isolation. However, the coarse granularity of its partitioning and steep reconfiguration overheads (>5s [47]) can leave resources idle.

Like temporal systems, existing spatial sharing systems are coarse-grained, operating at the level of inference requests or kernels. Their goal is to protect latency-critical (LC) applications by restricting kernels launched by other jobs [13, 44] or limiting GPU resources allocated to besteffort tasks, as seen in systems like REEF [20], MuxFlow [29], and others [4, 15, 22, 27, 32, 45, 50]. However, the coarseness of these approaches limits control over GPU resources, often leading to HoL blocking, low utilization, and interference. Figure 3 highlights the challenges of spatial sharing. In Figure 3(a), a single workload runs on the GPU, issuing two requests with five total kernels. This results in fast kernel completion for A and B but leaves much of the GPU underutilized. When MPS enables concurrent execution of multiple tasks in Figure 3(b), utilization is improved, but the original task's requests face significant delays. Overall, prior works have tackled some multitenant ML scheduling challenges but fail to offer a complete, transparent solution. Importantly, prior temporal and spatial strategies operate at a coarse granularity, limit utilization, and cause HoL blocking, which interferes with collocated workloads.

Right-sizing. Prior efforts have explored GPU job right-sizing to improve resource efficiency. However, these approaches often rely on hardware modifications [5, 54], lack transparency to application software [3, 4, 15, 25, 53], and depend on offline profiling [4–6, 15, 25, 53]. Crucially, most existing solutions operate at the granularity of entire jobs, which limits their ability to fully exploit the benefits of fine-grained right-sizing and can lead to suboptimal performance.

Dynamic Voltage Frequency Scaling. Recent efforts [24, 36, 37, 43, 52] have applied dynamic voltage frequency scaling (DVFS) to minimize the power consumption of GPUs with a particular focus on LLM inference clusters [24, 43]. Such approaches are based on extensive offline profiling across several input lengths and train dedicated output length predictors, failing to provide a transparent mechanism. Prior work on DVFS operates at a coarser granularity, observing the performance of the whole inference request and missing finer optimization opportunities.

3 Motivation

In this section, we showcase a detailed study of production GPU infrastructure challenges and opportunities.

3.1 Understanding GPU Utilization in Datacenters

To understand GPU utilization in datacenters, we analyze a subset of *Ads* inference services at Meta, which serve deep learning models across its fleet. At Meta, *Ads* inference relies in part on NVIDIA H100 GPU nodes. Each node has 8 GPUs, each partitioned via MIG. The production service performs offline analysis of each model, assigning models to GPU/MIG partitions for deployment. The goal is to meet tight SLAs on tail response times requirements for each model.

GPU Utilization. In Figure 1, we show GPU compute and memory utilization over a week. Device compute utilization ranges from 17% to 40%, averaging 27%. SM utilization is even lower, averaging 14%, with peaks at 21% and lows of 6.7%. Memory bandwidth utilization averages 20%. Overall, utilization follows a diurnal pattern tied to inference traffic. However, memory capacity remains steady at 28%, as models are kept loaded in GPU memory to meet tight SLAs. These SLAs also enforce small batch sizes, preventing full GPU resource saturation even at high request loads.

Inference Traffic. To investigate low GPU utilization, we first examine inference traffic. Figure 4 shows the mean-normalized requests per second (RPS) over a week, revealing a diurnal pattern. RPS can scale by 2.2× between minimum and maximum traffic, closely correlating with the GPU utilization trends shown in Figure 1. Next, we analyze model request frequencies. We sample thirteen of the most commonly used models and plot in Figure 5 the normalized frequency of inference requests over the same week. The distribution's variance is significant, with the most popular model *A* receiving several hundred times more requests than the least
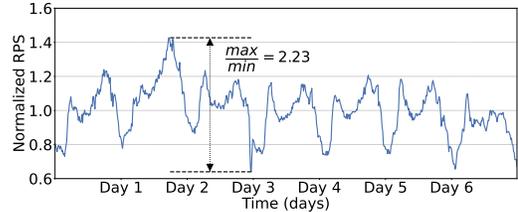

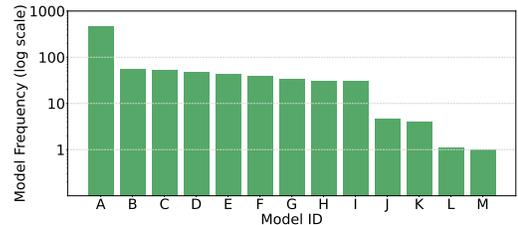
Figure 4. *Mean normalized traffic.*

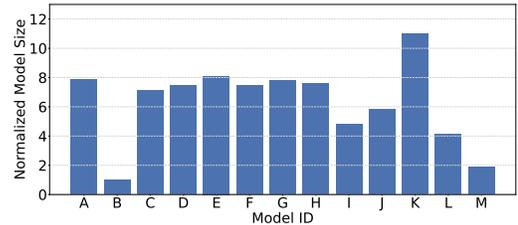**Figure 5.** *Model frequency distribution.*

Figure 6. *Model size distribution.*

popular model *M*. Over-provisioning GPUs for such a wide request distribution can lead to underutilization, particularly for less popular models.

Model Sizes. To better understand GPU utilization, we examine the sizes of the most commonly used models based on weights, parameters, and embeddings. As shown in Figure 6, model sizes vary significantly, with a more than a 10× difference between the largest and smallest models. Half are relatively large, while the rest are smaller. Both large and small models are frequently used: for example, the smallest model *B* has usage comparable to larger models *E* and *G*. This highlights the opportunity to collocate models of different sizes while meeting each of their service-level agreements.

GPU Sharing Limitations and Takeaways. Despite the urgent need to improve GPU utilization, datacenters often rely on limited GPU sharing or hardware approaches like MIG due to requirements for compatibility and transparency within the ML software stack. Non-transparent solutions that require framework or application changes for multitenancy are impractical at scale, given the complexity of maintaining multiple ML frameworks, runtimes, and compilers. Importantly, in the rapidly evolving ML space, transparent solutions help avoid the risk of locking infrastructure into rigid, outdated designs. Based on these insights, we design LithOS as a fully transparent OS for efficient ML multitenancy.
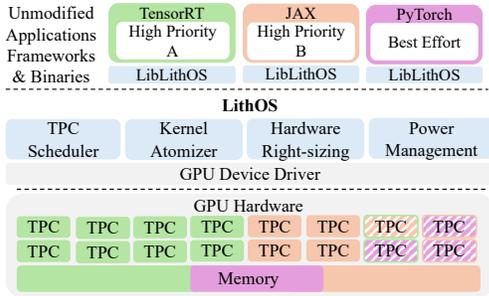
Figure 7. *LithOS architecture overview.*

4 LithOS Design

We propose LithOS, an OS designed to address GPU inefficiencies in datacenters. LithOS operates transparently across the ML stack, enabling efficient machine learning on GPUs.

4.1 Architecture Overview

Figure 7 presents LithOS's architecture. LithOS runs on CPU cores and interposes at the driver level, providing a dynamically linked library, LibLithOS, that mimics the native CUDA library. As a GPU operating system, LithOS maintains a system-wide view of GPU state across applications with varying priorities, enabling efficient scheduling and management. Applications follow the CUDA programming model and submit kernels to LithOS, which decouples submission from GPU execution. This transparently shifts scheduling control from the driver and hardware to the LithOS layer. The *TPC Scheduler* manages resources at the granularity of individual TPCs, unlocking *TPC Stealing* opportunities. Idle TPCs are lent to other tasks, improving utilization.

LithOS also introduces the *Kernel Atomizer*, which—without access to application source or PTX code—transparently breaks kernels into smaller thread block chunks called *atoms*. This enables finer-grained GPU scheduling and reduces head-of-line (HoL) blocking. Building on fine-grained control, LithOS supports dynamic hardware right-sizing, using lightweight models to reduce TPC allocations for individual kernels and atoms, yielding substantial capacity savings. Finally, LithOS applies transparent fine-grained DVFS, adjusting GPU frequency based on in-flight work to save energy. Together, these mechanisms enable intelligent scheduling policies that maximize GPU utilization and efficiency across diverse ML workloads. The rest of this section details how these mechanisms operate and interact, referencing Figure 8.

4.2 Interface with Userspace

Kernel Submission. Applications interact with LithOS via *launch queues* that buffer work (Figure 8, Step ①), giving LithOS full control over when work is dispatched to the GPU. This is important because once submitted, a kernel's priority or resources cannot be changed, nor can it be rescheduled. Eagerly dispatching work can lead to sub-optimal scheduling. LithOS therefore defers dispatch to minimize outstanding

work on the GPU. A launch queue is created when an application creates a stream via `cuStreamCreate`. On asynchronous CUDA calls like `cuLaunchKernel`, LithOS enqueues the kernel and returns control to the application.
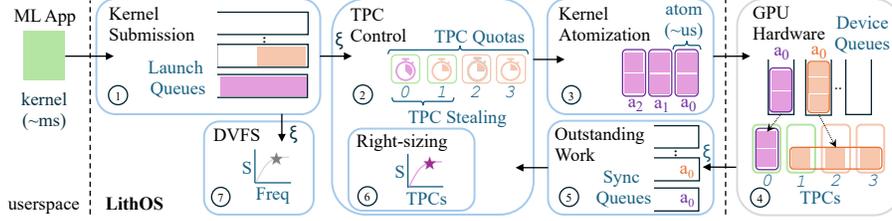
Compute Quotas. LithOS allows users and system administrators to define GPU resource limits, exposing TPC quotas (Figure 8, Step ②) that guarantee each application access to a specified number of TPCs when work is available. Internally, LithOS manages TPCs analogously to how a traditional OS manages CPU cores, enabling fine-grained control over GPU resources. As we will see next, LithOS relies on a highly efficient *TPC Scheduler* that interacts with launch queues and TPC quotas to optimize GPU utilization and efficiency.

4.3 TPC Scheduler

LithOS introduces a novel scheduler that operates at the granularity of individual TPCs, offering several advantages. TPC-level control enables fine-grained GPU resource management. Unlike static partitioning schemes like MIG, LithOS supports dynamic, on-the-fly TPC allocation, allowing each kernel to run on a different set of TPCs without reconfiguration overhead. This flexibility maximizes utilization without coarse partitioning or slow reallocation. Kernels are scheduled on their assigned TPCs, ensuring guaranteed resources for high-priority applications. However, as shown in Section 3, fixed allocations often leave TPCs idle due to traffic patterns or model variability. To address this, LithOS employs dynamic scheduling and TPC Stealing to reassign idle resources. We believe TPC scheduling lays the foundation for evolving GPU policies, much like CPU scheduling has matured over time [17, 35].

Operation. At a high level, the TPC Scheduler uses dispatcher threads to monitor launch queues (Figure 8, Step ①) and submit work to the GPU. A key goal is to keep the GPU busy while maintaining scheduling flexibility. The scheduler faces two main challenges: varying kernel durations and balancing flexibility with GPU starvation. To address the former, it applies Kernel Atomization (Figure 8, Step ③, Section 4.4) to split long-running kernels into smaller thread block chunks called *atoms*. To address the latter, it tracks outstanding work via sync queues (Figure 8, Step ⑤), throttling submissions until the backlog drops below a tunable threshold. A dedicated Tracker thread monitors task completion and updates scheduler state.

TPC Stealing. To improve work conservation, the scheduler dynamically reassigns underutilized TPCs across applications. In Figure 9(a), static allocation leads to idle TPCs. In Figure 9(b), stealing allows A_1 to borrow TPCs from an idle workload, reducing waste. However, this may cause head-of-line (HoL) blocking from priority inversion if a new request B is delayed by C_2 occupying the stolen TPCs. To mitigate this, the scheduler adopts a layered strategy. It

Figure 8. *LithOS operations overview.*

maintains per-TPC timers informed by a latency prediction module, estimating kernel (and atom) durations at submission time. These timers help avoid stealing from long-running TPCs. As tasks complete, sync queues are cleared and timers updated—potentially refining predictions (Section 4.7). LithOS also limits outstanding atoms and uses lower hardware stream priorities for work on stolen TPCs. Combined with kernel atomization, these mechanisms boost utilization while minimizing interference.

4.4 Kernel Atomizer

At the core of LithOS lies the *Kernel Atomizer*. The Kernel Atomizer transforms kernels into small chunks called *atoms*, each containing a subset of the grid's thread blocks (Figure 8, Step ③). Importantly, the Kernel Atomizer operates without any access to source or PTX code, making it fully transparent to the entire ML software stack. This allows LithOS to dispatch work at thread-block rather than kernel granularity. This is a critical requirement for an OS targeting GPUs, as the execution time of kernels can vary wildly from a few microseconds to tens of milliseconds.

Impact of Kernel Scheduling on Latency. To illustrate the need for kernel atomization, Figure 10 presents P_{99} kernel latencies across various training and inference workloads. Figure 10(a) shows how P_{99} latency increases with larger training batch sizes. Since the typical batch size for each model varies, we normalize by plotting memory usage at each size. Most models quickly produce long-running kernels lasting several milliseconds, with DLRM [30] standing out with kernel latencies exceeding 30ms. While training workloads are the major culprit, in Figure 10(b) we see that large language model (LLM) inference based on a trace from Microsoft Azure [43] containing small (*S*), medium (*M*), and large (*L*) prompt lengths can also produce several-millisecond-long kernels for large prompts. Based on this analysis and given that models can have very tight SLO constraints (in the low tens of milliseconds), we guide the design of LithOS toward a finer-grained scheduling unit that mitigates head-of-line blocking effects.

Operation. When a long-running kernel is about to be scheduled, LithOS predicts the duration of the kernel given its TPC assignment using the predictor module (detailed in Section 4.7). LithOS then computes the number of atoms into which to split the kernel by dividing the predicted kernel

Algorithm 1 Prelude Kernel Pseudocode.

```
1  kernel fn prelude(*args):
2      let atom : *const AtomMetadata = AtomMetadataAddr as _
3      let block_idx = blockIdx.z * gridDim.y * gridDim.x
4                    + blockIdx.y * gridDim.x
5                    + blockIdx.x
6      if atom->block_idx_lo <= block_idx < atom->block_idx_hi:
7          atom->kernel_entrypoint(*args)
```

duration by a tunable parameter called the `atom_duration`. If this parameter is set too low, an atomized kernel may actually take longer to complete. Crucially, LithOS is able to transparently chunk kernels into atoms at runtime. Atoms are then submitted to the GPU and can be scheduled on the TPCs dictated by the TPC Scheduler (Figure 8, ④). As a result, LithOS resolves a major challenge faced by prior works that operate higher in the stack: the Kernel Atomizer works on applications written in any framework, that use any libraries (including closed-source ones like cuDNN), and are built with any compiler.

To understand the benefits of scheduling at atom granularity, we return to Figure 9(b). Stealing improves the schedule but does not eliminate HoL blocking and wasted capacity. By dividing the kernels into atoms, work can be packed more tightly, as in Figure 9(c), and TPC allocations can be dynamically adjusted throughout a kernel's execution. Now, B_1 is no longer blocked by C_2, as stealing is disabled for the latter's subsequent atoms $\hat{C}_2$ once request B is submitted.

To demonstrate how LithOS's Kernel Atomizer operates, we consider a `Conv` kernel with a grid dimension of {8,8,1}, resulting in 64 blocks with `block_idx` ranging from 0 to 63. Instead of launching the `Conv` kernel directly, LithOS invokes a `Prelude` kernel, which calls into the original kernel using the same launch configuration. The prelude kernel is shown in Algorithm 1. At a high level, it checks whether `block_idx` falls within a specified range—calling `Conv` if so, or exiting early otherwise. For example, to partition the grid into 2 atoms, the kernel atomizer launches the prelude twice with block index ranges [0,32) and [32,64). Using this technique, LithOS can divide the kernel into up to 64 atoms. By specifying non-overlapping block ranges, the atomizer ensures each block is executed once, maintaining correctness.

Atomization Considerations. Kernels launch with an explicit set of resources; thus, the kernel atomizer ensures that the `Prelude` kernel uses the same set of resources as the original `Conv` kernel. Furthermore, the `Prelude` kernel
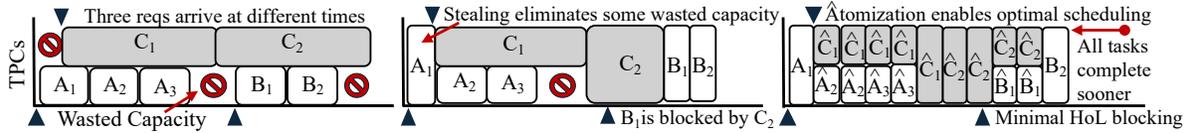
Figure 9. *GPU timeline for two workloads showcasing (a) TPC Scheduling, (B) Stealing, and (C) Atomization.*
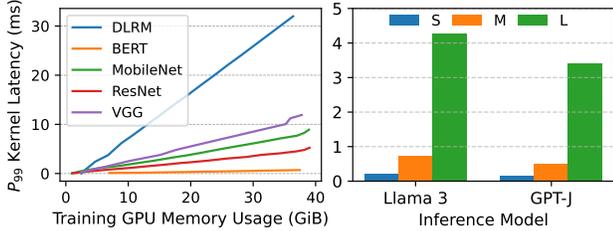

Figure 10. *(a) P_{99} kernel latency at different training batch sizes normalized to memory usage. (b) P_{99} kernel latency for different inference prompt sequence lengths for LLMs.*

needs to know the entry point to the `Conv` kernel. The Kernel Atomizer passes this information to the `Prelude` kernel in an `AtomMetadata` struct as seen in Algorithm 1.

Performance Optimizations. LithOS continuously monitors the effectiveness of the Kernel Atomizer to enhance performance. First, to avoid the overhead introduced by additional code in the `Prelude` kernel for kernels with many short threads, LithOS may disable atomization for such kernels. Additionally, for kernels with a large number of thread blocks, the Kernel Atomizer dynamically adjusts the `atom_-duration` parameter to control its aggressiveness. This minimizes the performance penalty due to the increased thread block traffic from early-exiting threads.

4.5 Right-Sizing Hardware Resources

LithOS's ability to schedule at the TPC level unlocks new opportunities for fine-grained GPU right-sizing. Figure 11 highlights this potential by plotting kernel speedups as a function of allocated TPCs for representative workloads (Section 6). The selected kernels collectively account for 99% of total execution time, with color gradients indicating each kernel's relative contribution. As an example, for Llama inference, general matrix multiplication (GEMM) and multihead attention kernels exhibit diminishing returns, while the kernel responsible for applying the token frequency penalty does not scale. The results show that whole-model right-sizing is suboptimal—there is no single TPC configuration that fits all kernels. Instead, substantial opportunity lies in right-sizing at the kernel level. First, individual kernels exhibit diverse scaling behaviors: some scale linearly, while others show diminishing returns. Second, the extent to which execution time is distributed across many kernels varies from workload to workload—highlighting the need for adaptive, per-kernel scheduling to fully optimize GPU resource consumption.

Modeling Kernel Scaling. LithOS introduces on-the-fly TPC right-sizing at the granularity of kernels (Figure 8, Step ⑥). The atoms of a given kernel inherit its allocated TPCs, as they exhibit the same scaling behavior as the kernel itself. To this end, LithOS introduces a model-based approach that interpolates the scaling of individual kernels based on two points: the latencies of a kernel running with all TPCs and just one TPC. It then fits a curve of the form

$$l = \frac{m}{t} + b$$

to these points, where l is the predicted latency, t is the corresponding number of TPCs, and m and b are constants. Note that the form of this curve is consistent with Amdahl's law for parallel speedup. Intuitively, b can be thought of as how long it takes for a single one of the kernel's thread blocks to complete on a single SM, and m quantifies the extent to which a kernel can take advantage of parallel processors.

Filtering Outliers. We find that, in practice, this simple model accurately captures the scaling behavior of most deep learning kernels. However, a small number of outlier kernels—typically those with very short runtimes—deviate from the model, as they fail to benefit from large TPC allocations and are inherently harder to model. To handle these cases, we introduce a *filtering* heuristic based on a kernel's thread block occupancy. Specifically, we estimate the number of TPCs a kernel can effectively utilize by dividing its total number of thread blocks by the occupancy per TPC—that is, the number of thread blocks a single TPC can execute concurrently. LithOS already tracks thread blocks per kernel as part of atomization, while occupancy can be queried from the driver API [33]. This heuristic provides an intuitive upper bound on useful TPC allocations per kernel, helping avoid overprovisioning even for difficult-to-model kernels.

Operation. When a kernel is submitted to LithOS, the dispatch thread first applies the filtering heuristic to estimate an upper bound on the number of TPCs the kernel can effectively utilize. If this estimate is lower than the job's allocated TPCs, the kernel is launched using the estimated bound. Otherwise, the dispatch thread leverages the learned scaling model to determine the minimum number of TPCs that would increase the kernel's latency by, at most, a multiplicative factor k that we call the *latency slip parameter*. This tunable parameter allows users and administrators to intuitively configure LithOS, for example, by specifying that up to 10% performance degradation is acceptable. Overall, LithOS enables highly efficient fine-grained right-sizing, while its
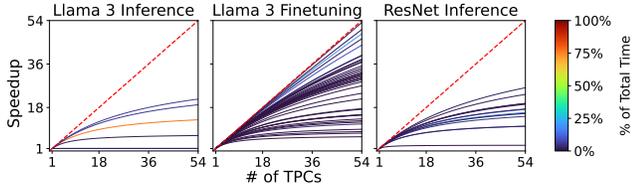
Figure 11. *LithOS's interpolated TPC scaling curves.*

Figure 12. *LithOS's interpolated frequency scaling curves.*

modeling and scaling techniques offer a robust and accurate solution—as we will see in Section 7.2.

4.6 Transparent Power Management

LithOS is well-positioned to enable transparent and efficient power management via DVFS. Just as right-sizing lets LithOS adapt resource allocation based on kernel scalability across TPCs, DVFS enables vertical scaling through frequency adjustment. Figure 12 shows how kernels from various workloads respond to frequency scaling. Many exhibit predictable behavior, creating opportunities for energy savings with bounded performance impact. To achieve efficient DVFS, LithOS must address two key challenges. First, current GPUs support relatively slow frequency switching ($\approx$50ms). While future architectures may reduce this latency [11], DVFS remains impractical for models with very short kernels. Thus, LithOS must consider the cumulative impact of scaling across kernel sequences. Second, although many kernels scale linearly with frequency—enabling significant energy savings—LithOS must carefully balance these gains against increased latency.

Modeling Frequency Scaling. LithOS introduces a transparent sequence-based kernel frequency scaling model that guides DVFS (Figure 8, Step ⑦). Similarly to right-sizing, the atoms of a given kernel inherit its frequency target, as they exhibit the same scaling behavior as the kernel itself. Specifically, each kernel is assigned a weight w, the ratio of its total runtime to the cumulative runtime of all the kernels in a particular stream. Then, LithOS approximates each kernel's relative slowdown as proportional to the fractional drop in frequency based on a first-order Taylor approximation:

$$k = \frac{lat(f_{th})}{lat(f_{max})} - 1 = s \cdot \left(\frac{f_{max}}{f_{th}} - 1\right)$$

where $lat(f)$ is the kernel's latency at frequency f. Specifically, f_{max} is the maximum frequency, and f_{th} is one of the device's supported frequencies. Each kernel's sensitivity is

$$s = \frac{k}{\frac{f_{max}}{f_{th}} - 1}$$

and the aggregate sensitivity S across all kernels is equal to $\sum w * s$. Similarly, the total slowdown is equal to

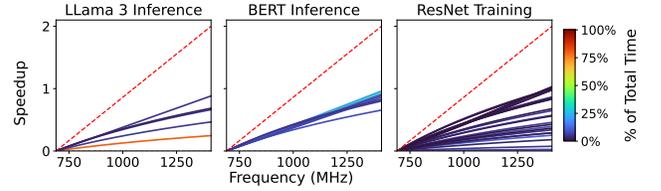$$S \cdot \left(\frac{f_{max}}{f_{final}} - 1\right) \leq k$$

and thus the final frequency that LithOS assigns to the workload is $f_{final} = \frac{f_{max}}{1+\frac{k}{S}}$. Intuitively, compute-bound kernels whose slowdown scales linearly with frequency reduction skew the final frequency closer to the maximum according to their sensitivity, while memory-bound kernels whose slowdown is frequency-insensitive skew the final frequency to lower levels depending on their weight.

Operation. Similar to right-sizing, LithOS uses a multiplicative factor k, the *latency slip parameter*, to guide DVFS decisions. At runtime, this parameter is used to evaluate the scaling model and select a target frequency. Due to the high latency of switching, LithOS adopts a conservative strategy and extends its learning period to avoid unnecessary transitions. Initially, LithOS collects per-kernel metadata at maximum frequency, forcing unseen kernels to run at max frequency. On first appearance, a kernel is assumed to scale linearly, and its frequency is reduced based on the configured k. Depending on the observed performance, LithOS either further lowers the frequency or stops after confirming linear behavior. Over time, it fits the collected data to the scaling model described earlier, enabling more informed and efficient DVFS decisions.

4.7 Online Latency Prediction

The latency prediction module learns the execution time of kernels, enabling the optimizations carried out by all of LithOS's components. In particular, it enhances TPC Stealing by estimating the duration of outstanding tasks and guides the number of atoms the Kernel Atomizer splits each kernel into. It further assists right-sizing and DVFS by providing the latencies that are used to calculate speedups based on TPCs and frequency scaling. This obviates the need for extensive offline profiling, which is impractical for a transparent OS.

Latency prediction operates separately for independent launch queues, allowing LithOS to dynamically adapt to the behavior of different applications. During execution, the module records kernel latencies and uses this data to refine its predictions. Each kernel's latency varies based on the allocated TPCs, the GPU frequency, and the granularity at which it is atomized; therefore, the prediction module must monitor these conditions to produce accurate estimates. In the case where such metadata are not available for a specific atom, the prediction module is conservative, assuming optimal linear scaling. For instance, if an atom was previously

executed with a TPC allocation of 100%, it fits a linear trend to estimate the duration when given half of the GPU.

One pitfall in achieving accurate kernel latency prediction is assuming that a given kernel function always has the same latency. In actuality, the duration can depend on the kernel's launch parameters and input arguments. For instance, a single `Conv` kernel function can be used multiple times across model layers that have varying tensor sizes. This necessitates that the latency prediction module track operators rather than kernel functions themselves.

By recording explicit synchronization events from the application, we can determine the start and end of a batch. We associate kernel launches with an ordinal index k, referring to the kth kernel after the start of a batch. This uniquely identifies operator nodes in the model's data flow graph (DFG), despite LithOS lacking explicit access to this high-level information. This additional ordinal index is sufficient to identify model operators and make accurate latency predictions.

5 Implementation

We implement a prototype of LithOS targeting NVIDIA GPUs in ~5000 lines of Rust, excluding macro-generated code for interposing the entire CUDA Driver API. Our prototype supports applications running natively or in containers. To enable concurrent execution across GPU contexts, we build on top of MPS. We defer some low-level details of the implementation to a separate technical report.

Interposition Architecture. LithOS is fully transparent to applications, supporting the diverse ML ecosystem and full GPU stack. A key implementation decision is abstracting the CUDA Driver API, the lowest common denominator across the stack. Instead of accessing the driver directly, applications interact with LithOS, which preserves CUDA call semantics. This abstraction ensures generality and transparency at the OS level. LithOS seamlessly supports unmodified ML applications using frameworks and libraries like PyTorch, TensorFlow, JAX, TensorRT, and closed-source libraries like cuDNN. Beyond transparency, LithOS avoids the complexity of interposing across the CUDA Runtime and other libraries that eventually call into the driver. Instead, it implements a small subset of CUDA APIs (e.g., `cuLaunchKernel`), while the rest are auto-generated via our toolchain. The LithOS library also eliminates complex data marshaling across address spaces, unlike prior CUDA API interposition systems [51]. This approach enables rapid support for new CUDA versions with minimal effort, enhancing long-term OS maintainability.

Isolation and Faults. In LithOS, applications run in separate address spaces and cannot access each other's memory. Illegal accesses lead to termination of the offending application. To handle other faults, LithOS enables graceful termination for common errors [29] by intercepting signals and terminating the application without affecting other contexts.

Model	Mem. (GiB)	Batch Size	Latency (ms)
VGG-19 [42]	17.4	120	291
ResNet-50 [21]	18.4	184	281
MobileNetV2 [40]	18.4	216	254
DLRM [30]	6.7	32768	74
BERT-Large [14]	17.3	20	159
Llama 3 Finetuning	32.0	4	690

Table 1. *Training model parameters.*

6 Experimental Setup and Methodology

Testbed. Experiments were conducted on a `1x A100 (SXM4)` Lambda Labs GPU instance with 30 CPU cores and 216 GB of host memory. The A100 GPU has 108 SMs with 40 GB of memory. The server was configured with Ubuntu 22.04, CUDA 12.8, Rust 1.83.0-nightly, Python 3.10, PyTorch 2.3, TensorRT 10.7, TensorRT-LLM 0.16.0, and Triton 24.12.

Baselines. We compare LithOS with all four NVIDIA GPU sharing methods: *Time slicing*, *MPS*, stream *Priority*, and *MIG*. We further compare against SOTA prior work across the spectrum of transparent solutions *TGS* [48], application modifications *REEF* [20], and both application modifications and offline profiling *Orion* [44]. We used the open-source TGS directly but had to re-implement Orion and REEF using our own interposition infrastructure, since the available code was tied to specific CUDA drivers and software stacks. We extend REEF and Orion to handle multiple HP apps in a straightforward manner. For REEF, BE kernels are not launched if *any* HP app is running. For Orion, BE kernels are not launched if they contend with *any* HP kernel.

Models and Configurations. All high priority inference tasks run on NVIDIA's Triton Inference Server with dynamic batching [9]. RetinaNet runs on ONNX Runtime while the other served models run on NVIDIA's TensorRT and TensorRT-LLM backends. We choose three representative vision models (RetinaNet [28], YOLOv4 [2], and ResNet-50 v1.5 [21]) and three language models (Llama 3 8B [16], GPT-J 6B [46], and BERT-Large [14]) as inference workloads. For large language models, we use a Microsoft Azure trace [43]. For the best effort training tasks, we use three vision models, ResNet-50, MobileNetV2, VGG-19, a deep learning recommendation model (DLRM), a language model BERT-Large, and LLM fine-tuning with Llama 3 as listed. The training batch size is adjusted to use at most half of the GPU DRAM to keep all models in memory when stacking. The best effort training task runs continuously. More details are shown in Table 1.

7 Evaluation

Our evaluation answers the following questions:

1. Does LithOS improve performance for different multi-tenancy environments and SOTA prior works?
2. What are the capacity savings due to LithOS's hardware right-sizing?
3. What are the energy savings of LithOS's DVFS?
4. How do different LithOS features affect performance?

Model	Framework	Load (rps)	Constraint (ms)
ResNet [21]	TensorRT	1000	15
RetinaNet [28]	ONNX Runtime	9	100
Llama 3 [16]	TensorRT-LLM	0.5	2000
GPT-J [46]	TensorRT-LLM	0.5	2000
BERT [14]	TensorRT	30	130

Table 2. *Inference services for inference-only multitenancy.*

Figure 14. *Inference-only multitenancy: goodput by app.*
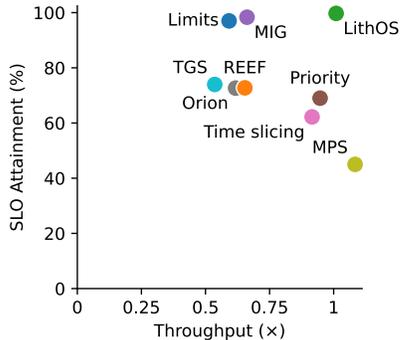

Figure 13. *SLO attainment and throughput by system.*

7.1 Performance in Multitenant Environments

In the following experiments, we disable right-sizing and power management features of LithOS to provide an apples-to-apples comparison to other systems in terms of scheduling efficiency alone. We evaluate these features afterwards.

Inference-only Multitenancy. We evaluate LithOS in a multitenant environment with three inference applications: one best-effort (BE) and two high-priority (HP). The first HP app, *HP A*, has a latency-oriented SLO: percentage of requests executed within a latency constraint. The second, *HP B*, has a throughput-oriented SLO: attained throughput as a percentage of the case where it executes alone. These vary according to the model (Table 2).

The BE and HP B models are chosen from Llama 3, GPT-J, and BERT. For HP A, we add ResNet and RetinaNet. We use latency constraints from the MLPerf Datacenter inference benchmark [38] (Table 2). We run all possible combinations. HP apps follow Poisson load and run on the Triton inference server, while BE apps execute in a closed loop. Latencies for all models, including LLMs, are measured end-to-end.

We compare LithOS against all configurations. For systems that support partitioning, HP A and HP B are isolated on partitions of 75% and 25%, respectively. MIG's limited partitioning configurations cannot support a 25%-75% split, so we use a 3/7-4/7 split instead. MIG and Limits cannot support a BE app, but only apps with provisioned resources; therefore, the BE does not run on these systems. There is no way to isolate multiple latency-sensitive applications on systems like Priority, REEF, TGS, and Orion. For these, we set both of the HP apps to high priority and the BE to low priority.
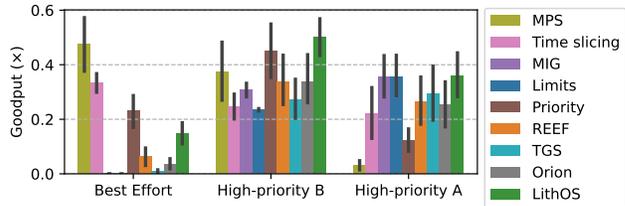
Figure 13 compares all systems across two dimensions: SLO attainment and throughput. "SLO" of 100% means both HPs reach 100% attainment. "Throughput" of 1 means that the throughput achieved is as much as if any of the apps had the entire device. Unsurprisingly, MPS sets the bar for throughput. MPS's fine-grained, intra-SM stacking ensures device resources are maximally utilized, and it allows more throughput when stacking than any application could have alone; hence, it achieves a throughput of 1.08. MPS's throughput comes at the cost of SLO attainment, at 42%. MIG and thread limits both successfully meet SLOs. This is expected, as each system minimizes interference by devoting resources to individual apps. However, the partitions are not fully utilized without a BE app. As a result, aggregate throughput drops to 0.66 and 0.59 for thread limits and MIG, respectively. Without isolating HP apps, priority-only systems cannot attain SLOs, with TGS leading at 72%. LithOS provides the best of both worlds, as it provides spatial isolation like MIG with an SLO attainment of 100% and a throughput of 1.

Where do the benefits of LithOS come from? Figure 14 shows LithOS consistently leading in goodput (throughput excluding HP A requests that violate the SLO constraint) for the HP apps while still allowing significant (0.15) BE throughput. While the partitioning systems match LithOS in HP A goodput, they lack in HP B goodput: MIG at 0.31 vs. LithOS at 0.50. Partitioning schemes cannot support any BE throughput, while LithOS allows HP apps to steal unused resources from one another and further support BE throughput. No SotA system can perform effectively across all requirements. Specifically, REEF and Orion underperform in latency-sensitive goodput and TGS in throughput. Only LithOS provides the best HP goodput while sustaining high BE throughput.

Diving deeper, we next look into the latencies of the HP A app in Figure 15. The figure shows the P_{99} latencies for each model averaged across all combinations. Latencies diverge in many cases, with only LithOS and the partitioning systems limiting latencies to the constraints. MPS is the worst-performing regarding latencies; LithOS's latencies are 13× better. LithOS reduces latencies by 12× compared to Orion. This is expected as Orion cannot handle multiple HP apps. TGS limits latencies much more effectively than Orion and REEF, but LithOS still improves over it by 3×. Overall, LithOS provides a robust solution for inference stacking.
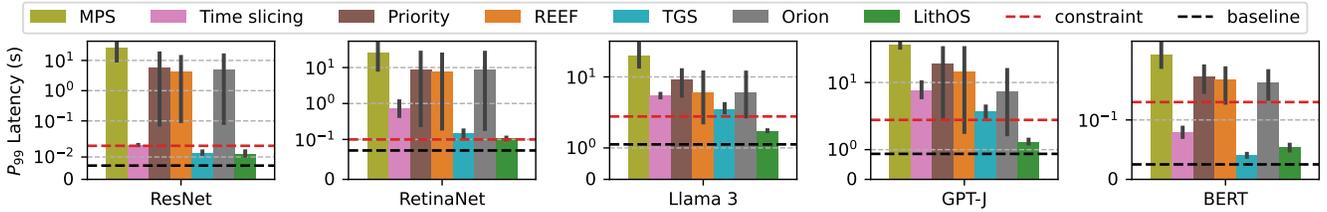
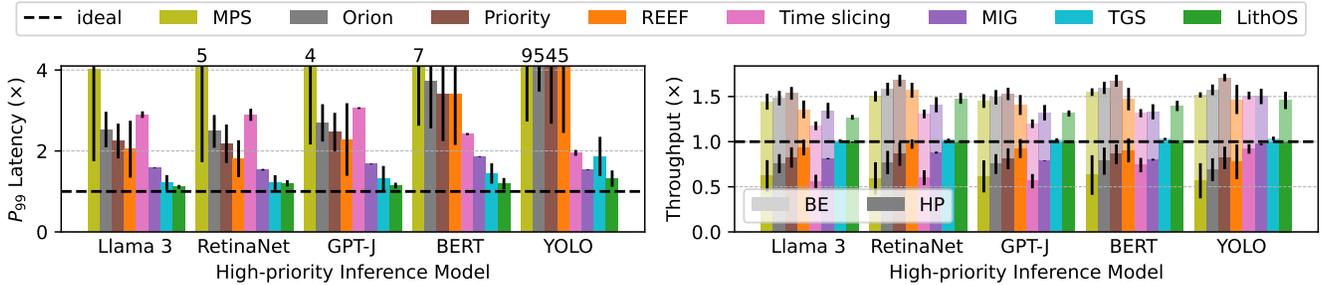**Figure 15.** *Inference stacking multitenancy: HP A tail latencies by model.*

Figure 16. *Hybrid multitenancy: (a) P_{99} service latency and (b) aggregate throughput.*

Hybrid Inference/Training Multitenancy. In this experiment, we stack an HP that has a latency-oriented SLO with a training BE app. Similar to the inference-stacking experiment, resources unused by the sensitive inference app should be donated to the best-effort training job. At the same time, service latency must not increase. We choose the inference model from the set, Llama 3 8B, GPT-J 6B, BERT-Large, RetinaNet, and YOLOv4. We choose the training model from those listed in Table 1. We run all model combinations, and our client creates Poisson loads. Load parameters are chosen to keep GPU utilization around 80% for the HP app.

Figure 16 shows the P_{99} HP latency and aggregate throughput, averaged across all training models. HP throughput is normalized to the load before being added to the BE throughput, normalized to the case where the BE model runs alone on the device. Latencies are also normalized to the case where the HP runs alone on the device. MPS yields latencies 5.83× the ideal case, and its service throughput is the lowest at 60%. Time slicing fares better as it enables the long-running kernels of the best-effort models to be preempted, guaranteeing the service approximately 50% of the GPU time. MIG performs similarly to time slicing by allocating 50% of the GPU to the service spatially rather than temporally. However, both methods fail to sustain peak HP throughput. Stream priority also falls short, leading to a 2.89× increase in service latency and service throughput as low as 68%.

Both TGS and REEF also struggle to maintain low service latencies. TGS has an average inference latency of 1.41× the ideal, and REEF 2.89×. TGS's poor performance stems from its adaptive rate control mechanism, which assumes a constant work arrival rate. This assumption is invalid for inference services, which have unpredictable load patterns. REEF fails to sufficiently throttle the training model, allowing
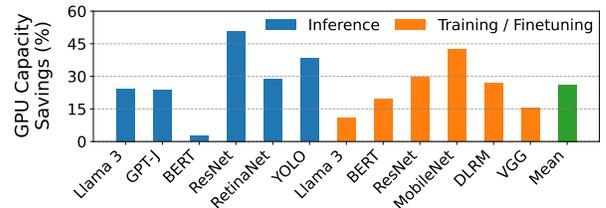

Figure 17. *Hardware right-sizing GPU capacity savings.*

tail latencies to reach 8.93×. In contrast, LithOS maintains a tail latency within 20% of the ideal. On average, this is a 2.34× and 1.18× over REEF and TGS, respectively. Compared to the native MPS solution, LithOS reduces latency by up to 13.54× and 4.7× on average. LithOS maintains service throughput within 1% of load in the worst case. LithOS improves training throughput by an average of 34× and aggregate throughput by 1.35× vs. TGS. In total, LithOS improves aggregate throughput 1.23×–1.57× with an average of 1.38×.

7.2 Kernel-SM Right-Sizing

Capacity Savings. Figure 17 shows the capacity savings due to right-sizing with LithOS. We compute savings by comparing the time-weighted average of TPC utilization before and after right-sizing. LithOS provides excellent savings of up to 51%, and a mean of 26% across all workloads. We expect that in future GPU architectures with an increased number of TPCs, the fine-grained right-sizing approach of LithOS will provide even more aggressive saving potential.

Latency and Throughput Cost. With a latency slip parameter of 1.1, the performance cost of right-sizing in terms of P_{99} and throughput is modest. The mean increase in P_{99} and decrease in throughput are both 4%. Our latency slip parameter is conservative because not all of the end-to-end
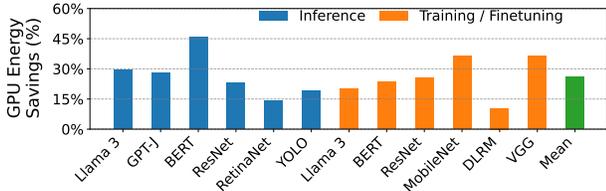
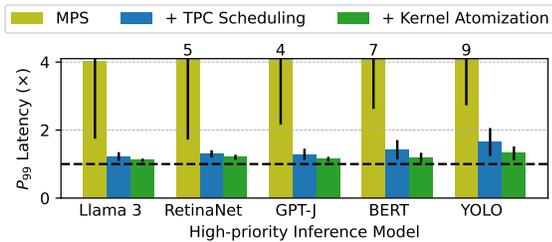**Figure 18.** *Power management GPU energy savings.*

Figure 19. *Breakdown of LithOS features for inf-train.*

execution time of each inference or training iteration is spent inside a GPU kernel; this does not impede tuning in practice.

Accuracy. To quantify the accuracy of our prediction technique, we compute the kernel-execution-time weighted average of the R^2 values for the curves we fit (i.e., for kernels where the possible TPCs value exceeds the threshold). Across all of the evaluated workloads, the average R^2 values range from 0.92 (Llama finetuning) to 0.99 (RetinaNet inference), indicating that our technique is highly accurate.

7.3 Kernel-Dependent DVFS

Energy Savings. Figure 18 shows the energy savings of LithOS's DVFS mechanism across different inference and training workloads. We define energy savings by recording the difference between executing the workload at maximum frequency, and under LithOS's DVFS policy. LithOS provides significant energy savings of up to 46%, and a mean of 26% across all workloads without offline profiling requirements.

Performance Cost. The slip parameter for this experiment was set at 1.1, and the mean increase in P_{99} latency is 7%. The minimal increase in P_{99} latency demonstrates that LithOS's DVFS policy is inherently conservative. It respects latency constraints across workloads while transparently providing substantial energy savings. Finer-grained frequency control could unlock additional energy savings.

7.4 Ablation and Case Studies

Multi-tenancy Breakdown. Figure 19 presents a performance analysis for inference-training as explored in Figure 16. Enabling the TPC scheduler improves HP tail latencies to 1.38× ideal by throttling BE work, while maintaining ideal HP throughput. Kernel Atomization offers additional gains, reducing tail latencies to an average of 1.19× and up to 1.55×, by splitting long BE kernels and improving TPC Stealing. Because of space limitations, we plot only latencies.
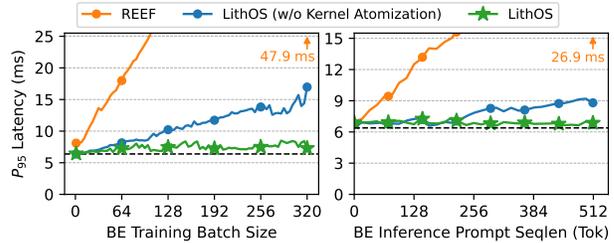

Figure 20. P_{95} *latency of HP inference collocated with varied (a) batch sizes training and (b) sequence lengths inference.*

Kernel Atomization introduces a 10% throughput overhead, as LithOS prioritizes HP workloads by reducing BE throughput. Overall, each of LithOS's features plays a crucial role in optimizing end-to-end performance.

Kernel Atomization. To highlight the challenges of scheduling long-running kernels, we collocate an HP BERT inference workload with either a BE VGG training or a BE Llama 3 inference. In Figure 20, we vary (a) the batch size of the BE training job and (b) the sequence length of the BE inference job and measure the P_{95} latency of the HP inference job. LithOS outperforms REEF by 6.5× and 3.9× in (a) and (b), respectively. Unlike REEF, which simply throttles BE work, LithOS accounts for kernel durations, which can vary significantly. To understand the impact of Kernel Atomization, we further evaluate LithOS with Kernel Atomization disabled. Kernel Atomization provides an improvement of 2× and 1.3× in (a) and (b), respectively. As described in Figure 10, kernel durations grow with training batch size and inference input sequence length. As Kernel Atomization allows LithOS to schedule at thread block granularity, HoL blocking is minimized. Consequently, the HP tail latency for the full LithOS system is within 14% (or 1ms) or 7% (or 0.45ms) of ideal for even the largest batch size or sequence length, respectively.

Latency Prediction Module. Next, we evaluate the accuracy of the latency prediction module of LithOS that enhances the TPC Scheduler and the Kernel Atomizer. Specifically, we record the predicted atom latencies and compare them with the corresponding recorded CUDA events. We consider absolute errors greater than 50μs to be mispredictions. Overall, we find very low misprediction rates of just 0.9% and 0.38% for the HP workloads in inference-inference and inference-training environments, respectively. Additionally, the prediction error tails are small with P_{99} values of 49μs and 31μs. Misprediction rates for the BE workloads are higher at 14% and 11% for inference-inference and inference-training, respectively. This is acceptable as BE work is frequently preempted by HP work and has lower priority for GPU resources.

8 Conclusion

This paper introduces LithOS, a first step towards an operating system for efficient machine learning on GPUs. LithOS

operates transparently to the entire ML stack. Through mechanisms like TPC Scheduling, Kernel Atomization, hardware right-sizing, and power management, LithOS significantly improves GPU efficiency while laying the foundation for future OS research on GPUs.

9 Acknowledgments

References

[1] Zhihao Bai, Zhen Zhang, Yibo Zhu, and Xin Jin. 2020. {PipeSwitch}: Fast pipelined context switching for deep learning applications. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 499–514.

[2] Alexey Bochkovskiy, Chien-Yao Wang, and Hong-Yuan Mark Liao. 2020. YOLOv4: Optimal Speed and Accuracy of Object Detection. arXiv:2004.10934 [cs.CV] https://arxiv.org/abs/2004.10934

[3] Qichen Chen, Hyerin Chung, Yongseok Son, Yoonhee Kim, and Heon Young Yeom. 2021. smCompactor: a workload-aware fine-grained resource management framework for GPGPUs. In *Proceedings of the 36th Annual ACM Symposium on Applied Computing* (Virtual Event, Republic of Korea) *(SAC '21)*. Association for Computing Machinery, New York, NY, USA, 1147–1155. doi:10.1145/3412841.3441989

[4] Seungbeom Choi, Sunho Lee, Yeonjae Kim, Jongse Park, Youngjin Kwon, and Jaehyuk Huh. 2022. Serving Heterogeneous Machine Learning Models on Multi-GPU Servers with Spatio-Temporal Sharing. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. USENIX Association, Carlsbad, CA, 199–216. https://www.usenix.org/conference/atc22/presentation/choi-seungbeom

[5] Marcus Chow, Ali Jahanshahi, and Daniel Wong. 2023. KRISP: Enabling Kernel-wise RIght-sizing for Spatial Partitioned GPU Inference Servers. In *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. 624–637. doi:10.1109/HPCA56546.2023.10071121

[6] Marcus Chow and Daniel Wong. 2024. CoFRIS: Coordinated Frequency and Resource Scaling for GPU Inference Servers. In *Proceedings of the 14th International Green and Sustainable Computing Conference* (Toronto, ON, Canada) *(IGSC '23)*. Association for Computing Machinery, New York, NY, USA, 45–51. doi:10.1145/3634769.3634808

[7] NVIDIA Corporation. [n. d.]. Multi-Process Service. https://docs.nvidia.com/deploy/mps/index.html. Accessed: April 14, 2025.

[8] NVIDIA Corporation. 2023. *NVIDIA H100 Tensor Core GPU Architecture*. Technical Report. NVIDIA Corporation, Santa Clara, CA.

[9] NVIDIA Corporation. 2024. Triton Inference Server. https://developer.nvidia.com/triton-inference-server. Accessed: May 8, 2024.

[10] NVIDIA Corporation. 2025. NVIDIA Multi-Instance GPU User Guide. https://docs.nvidia.com/datacenter/tesla/mig-user-guide/index.html. Accessed: April 14, 2025.

[11] NVIDIA Corporation. 2025. NVIDIA RTX BLACKWELL GPU ARCHITECTURE. https://images.nvidia.com/aem-dam/Solutions/geforce/blackwell/nvidia-rtx-blackwell-gpu-architecture.pdf.

[12] Daniel Crankshaw, Xin Wang, Guilio Zhou, Michael J. Franklin, Joseph E. Gonzalez, and Ion Stoica. 2017. Clipper: A Low-Latency Online Prediction Serving System. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. USENIX Association, Boston, MA, 613–627. https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/crankshaw

[13] Weihao Cui, Han Zhao, Quan Chen, Ningxin Zheng, Jingwen Leng, Jieru Zhao, Zhuo Song, Tao Ma, Yong Yang, Chao Li, and Minyi Guo. 2021. Enable simultaneous DNN services based on deterministic operator overlap and precise latency prediction. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (St. Louis, Missouri) *(SC '21)*. Association for Computing Machinery, New York, NY, USA, Article 15, 15 pages. doi:10.1145/3458817.3476143

[14] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. arXiv:1810.04805 [cs.CL] https://arxiv.org/abs/1810.04805

[15] Aditya Dhakal, Sameer G Kulkarni, and K. K. Ramakrishnan. 2020. GSLICE: Controlled Spatial Sharing of GPUs for a Scalable Inference Platform. In *Proceedings of the 11th ACM Symposium on Cloud Computing* (Virtual Event, USA) *(SoCC '20)*. Association for Computing Machinery, New York, NY, USA, 492–506. doi:10.1145/3419111.3421284

[16] Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, and Angela et al. Fan. 2024. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783* (2024).

[17] Joshua Fried, Zhenyuan Ruan, Amy Ousterhout, and Adam Belay. 2020. Caladan: Mitigating Interference at Microsecond Timescales. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 281–297. https://www.usenix.org/conference/osdi20/presentation/fried

[18] Yanjie Gao, Yichen He, Xinze Li, Bo Zhao, Haoxiang Lin, Yoyo Liang, Jing Zhong, Hongyu Zhang, Jingzhou Wang, Yonghua Zeng, et al. 2024. An Empirical Study on Low GPU Utilization of Deep Learning Jobs. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 1–13.

[19] Arpan Gujarati, Reza Karimi, Safya Alzayat, Wei Hao, Antoine Kaufmann, Ymir Vigfusson, and Jonathan Mace. 2020. Serving DNNs like Clockwork: Performance Predictability from the Bottom Up. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 443–462. https://www.usenix.org/conference/osdi20/presentation/gujarati

[20] Mingcong Han, Hanze Zhang, Rong Chen, and Haibo Chen. 2022. Microsecond-scale Preemption for Concurrent GPU-accelerated DNN Inferences. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. USENIX Association, Carlsbad, CA, 539–558. https://www.usenix.org/conference/osdi22/presentation/han

[21] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2015. Deep Residual Learning for Image Recognition. arXiv:1512.03385 [cs.CV] https://arxiv.org/abs/1512.03385

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

[23] Myeongjae Jeon, Shivaram Venkataraman, Amar Phanishayee, unjie Qian, Wencong Xiao, and Fan Yang. 2019. Analysis of Large-Scale Multi-Tenant GPU Clusters for DNN Training Workloads. In *Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference* (Renton, WA, USA) *(USENIX ATC '19)*. USENIX Association, USA, 947–960.

[24] Andreas Kosmas Kakolyris, Dimosthenis Masouros, Petros Vavaroutsos, Sotirios Xydis, and Dimitrios Soudris. 2024. SLO-aware GPU Frequency Scaling for Energy Efficient LLM Inference Serving. arXiv:2408.05235 [cs.DC] https://arxiv.org/abs/2408.05235

[25] Yunseong Kim, Yujeong Choi, and Minsoo Rhu. 2022. PARIS and ELSA: an elastic scheduling algorithm for reconfigurable multi-GPU

inference servers. In *Proceedings of the 59th ACM/IEEE Design Automation Conference* (San Francisco, California) *(DAC '22)*. Association for Computing Machinery, New York, NY, USA, 607–612. doi:10.1145/3489517.3530510

[26] Beth Kindig. 2024. AI power consumption: Rapidly becoming mission-critical. https://www.forbes.com/sites/bethkindig/2024/06/20/ai-power-consumption-rapidly-becoming-mission-critical/

[27] Baolin Li, Tirthak Patel, Siddharth Samsi, Vijay Gadepally, and Devesh Tiwari. 2022. MISO: Exploiting Multi-Instance GPU Capability on Multi-Tenant GPU Clusters. In *Proceedings of the 13th Symposium on Cloud Computing* (San Francisco, California) *(SoCC '22)*. Association for Computing Machinery, New York, NY, USA, 173–189. doi:10.1145/3542929.3563510

[28] Tsung-Yi Lin, Priya Goyal, Ross Girshick, Kaiming He, and Piotr Dollár. 2018. Focal Loss for Dense Object Detection. arXiv:1708.02002 [cs.CV] https://arxiv.org/abs/1708.02002

[29] Xuanzhe Liu, Yihao Zhao, Shufan Liu, Xiang Li, Yibo Zhu, Xin Liu, and Xin Jin. 2024. MuxFlow: efficient GPU sharing in production-level clusters with more than 10000 GPUs. *Science China Information Sciences* 67, 12 (2024), 222101. doi:10.1007/s11432-024-4227-2

[30] Maxim Naumov, Dheevatsa Mudigere, Hao-Jun Michael Shi, Jianyu Huang, Narayanan Sundaraman, Jongsoo Park, Xiaodong Wang, Udit Gupta, Carole-Jean Wu, Alisson G. Azzolini, Dmytro Dzhulgakov, Andrey Mallevich, Ilia Cherniavskii, Yinghai Lu, Raghuraman Krishnamoorthi, Ansha Yu, Volodymyr Kondratenko, Stephanie Pereira, Xianjie Chen, Wenlin Chen, Vijay Rao, Bill Jia, Liang Xiong, and Misha Smelyanskiy. 2019. Deep Learning Recommendation Model for Personalization and Recommendation Systems. arXiv:1906.00091 [cs.IR] https://arxiv.org/abs/1906.00091

[31] Microsoft Network. 2024. Dell exec reveals Nvidia has a 1,000 watt GPU in the works. https://www.msn.com/en-us/lifestyle/other/dell-exec-reveals-nvidia-has-a-1-000-watt-gpu-in-the-works/ar-BB1jlE8f. Accessed: June 24, 2024.

[32] Kelvin K. W. Ng, Henri Maxime Demoulin, and Vincent Liu. 2023. Paella: Low-latency Model Serving with Software-defined GPU Scheduling. In *Proceedings of the 29th Symposium on Operating Systems Principles* (<conf-loc>, <city>Koblenz</city>, <country>Germany</country>, </conf-loc>) *(SOSP '23)*. Association for Computing Machinery, New York, NY, USA, 595–610. doi:10.1145/3600006.3613163

[33] NVIDIA Corporation. [n. d.]. *NVIDIA CUDA Driver API Documentation: Occupancy*. NVIDIA Corporation. https://docs.nvidia.com/cuda/cuda-driver-api/group__CUDA__OCCUPANCY.html

[34] Christopher Olston, Noah Fiedel, Kiril Gorovoy, Jeremiah Harmsen, Li Lao, Fangwei Li, Vinu Rajashekhar, Sukriti Ramesh, and Jordan Soyke. 2017. TensorFlow-Serving: Flexible, High-Performance ML Serving. arXiv:1712.06139 [cs.DC]

[35] Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, and Hari Balakrishnan. 2019. Shenango: Achieving High CPU Efficiency for Latency-sensitive Datacenter Workloads. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. USENIX Association, Boston, MA, 361–378. https://www.usenix.org/conference/nsdi19/presentation/ousterhout

[36] Pratyush Patel, Esha Choukse, Chaojie Zhang, Íñigo Goiri, Brijesh Warrier, Nithish Mahalingam, and Ricardo Bianchini. 2024. 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* (La Jolla, CA, USA) *(ASPLOS '24)*. Association for Computing Machinery, New York, NY, USA, 207–222. doi:10.1145/3620666.3651329

[37] Haoran Qiu, Weichao Mao, Archit Patke, Shengkun Cui, Saurabh Jha, Chen Wang, Hubertus Franke, Zbigniew Kalbarczyk, Tamer Başar, and Ravishankar K. Iyer. 2024. Power-aware Deep Learning Model Serving with μ-Serve. In *2024 USENIX Annual Technical Conference (USENIX ATC 24)*. USENIX Association, Santa Clara, CA, 75–93. https://www.usenix.org/conference/atc24/presentation/qiu

[38] Vijay Janapa Reddi, Christine Cheng, David Kanter, Peter Mattson, Guenther Schmuelling, Carole-Jean Wu, Brian Anderson, Maximilien Breughe, Mark Charlebois, William Chou, Ramesh Chukka, Cody Coleman, Sam Davis, Pan Deng, Greg Diamos, Jared Duke, Dave Fick, J. Scott Gardner, Itay Hubara, Sachin Idgunji, Thomas B. Jablin, Jeff Jiao, Tom St. John, Pankaj Kanwar, David Lee, Jeffery Liao, Anton Lokhmotov, Francisco Massa, Peng Meng, Paulius Micikevicius, Colin Osborne, Gennady Pekhimenko, Arun Tejusve Raghunath Rajan, Dilip Sequeira, Ashish Sirasao, Fei Sun, Hanlin Tang, Michael Thomson, Frank Wei, Ephrem Wu, Lingjie Xu, Koichi Yamada, Bing Yu, George Yuan, Aaron Zhong, Peizhao Zhang, and Yuchen Zhou. 2019. MLPerf Inference Benchmark. arXiv:1911.02549 [cs.LG]

[39] Francisco Romero, Qian Li, Neeraja J. Yadwadkar, and Christos Kozyrakis. 2021. INFaaS: Automated Model-less Inference Serving. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. USENIX Association, 397–411. https://www.usenix.org/conference/atc21/presentation/romero

[40] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. 2018. Mobilenetv2: Inverted residuals and linear bottlenecks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 4510–4520.

[41] Haichen Shen, Lequn Chen, Yuchen Jin, Liangyu Zhao, Bingyu Kong, Matthai Philipose, Arvind Krishnamurthy, and Ravi Sundaram. 2019. Nexus: A GPU Cluster Engine for Accelerating DNN-Based Video Analysis. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (Huntsville, Ontario, Canada) *(SOSP '19)*. Association for Computing Machinery, New York, NY, USA, 322–337. doi:10.1145/3341301.3359658

[42] Karen Simonyan and Andrew Zisserman. 2015. Very Deep Convolutional Networks for Large-Scale Image Recognition. arXiv:1409.1556 [cs.CV] https://arxiv.org/abs/1409.1556

[43] Jovan Stojkovic, Chaojie Zhang, Íñigo Goiri, Josep Torrellas, and Esha Choukse. 2024. DynamoLLM: Designing LLM Inference Clusters for Performance and Energy Efficiency. arXiv:2408.00741 [cs.AI] https://arxiv.org/abs/2408.00741

[44] Foteini Strati, Xianzhe Ma, and Ana Klimovic. 2024. Orion: Interference-aware, Fine-grained GPU Sharing for ML Applications. In *Proceedings of the Nineteenth European Conference on Computer Systems* (<conf-loc>, <city>Athens</city>, <country>Greece</country>, </conf-loc>) *(EuroSys '24)*. Association for Computing Machinery, New York, NY, USA, 1075–1092. doi:10.1145/3627703.3629578

[45] Cheng Tan, Zhichao Li, Jian Zhang, Yu Cao, Sikai Qi, Zherui Liu, Yibo Zhu, and Chuanxiong Guo. 2021. Serving DNN Models with Multi-Instance GPUs: A Case of the Reconfigurable Machine Scheduling Problem. arXiv:2109.11067 [cs.DC]

[46] Ben Wang and Aran Komatsuzaki. 2021. GPT-J-6B: A 6 Billion Parameter Autoregressive Language Model. https://github.com/kingoflolz/mesh-transformer-jax.

[47] Tianyu Wang, Sheng Li, Bingyao Li, Yue Dai, Ao Li, Geng Yuan, Yufei Ding, Youtao Zhang, and Xulong Tang. 2024. Improving GPU Multi-Tenancy Through Dynamic Multi-Instance GPU Reconfiguration. *arXiv preprint arXiv:2407.13126* (2024).

[48] Bingyang Wu, Zili Zhang, Zhihao Bai, Xuanzhe Liu, and Xin Jin. 2023. Transparent GPU Sharing in Container Clouds for Deep Learning Workloads. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. USENIX Association, Boston, MA, 69–85. https://www.usenix.org/conference/nsdi23/presentation/wu

[49] Wencong Xiao, Shiru Ren, Yong Li, Yang Zhang, Pengyang Hou, Zhi Li, Yihui Feng, Wei Lin, and Yangqing Jia. 2020. AntMan: Dynamic Scaling on GPU Clusters for Deep Learning. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation (OSDI'20)*. USENIX Association, USA, Article 30, 16 pages.

[50] Fei Xu, Jianian Xu, Jiabin Chen, Li Chen, Ruitao Shang, Zhi Zhou, and Fangming Liu. 2023. iGniter: Interference-Aware GPU Resource Provisioning for Predictable DNN Inference in the Cloud. *IEEE Transactions on Parallel and Distributed Systems* 34, 3 (2023), 812–827. doi:10.1109/TPDS.2022.3232715

[51] Hangchen Yu, Arthur Michener Peters, Amogh Akshintala, and Christopher J Rossbach. 2020. AvA: Accelerated virtualization of accelerators. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. 807–825.

[52] Yijia Zhang, Qiang Wang, Zhe Lin, Pengxiang Xu, and Bingqiang Wang. 2024. Improving GPU Energy Efficiency through an Application-transparent Frequency Scaling Policy with Performance Assurance. In *Proceedings of the Nineteenth European Conference on Computer Systems* (Athens, Greece) *(EuroSys '24)*. Association for Computing Machinery, New York, NY, USA, 769–785. doi:10.1145/3627703.

3629584

[53] Yongkang Zhang, Haoxuan Yu, Chenxia Han, Cheng Wang, Baotong Lu, Yunzhe Li, Zhifeng Jiang, Yang Li, Xiaowen Chu, and Huaicheng Li. 2025. SGDRC: Software-Defined Dynamic Resource Control for Concurrent DNN Inference on NVIDIA GPUs. In *Proceedings of the 30th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming* (Las Vegas, NV, USA) *(PPoPP '25)*. Association for Computing Machinery, New York, NY, USA, 267–281. doi:10.1145/3710848.3710863

[54] Xia Zhao, Magnus Jahre, and Lieven Eeckhout. 2020. 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* (Lausanne, Switzerland) *(ASPLOS '20)*. Association for Computing Machinery, New York, NY, USA, 1371–1385. doi:10.1145/3373376.3378457