Improving GPU Sharing Performance through Adaptive Bubbleless Spatial-Temporal Sharing

Shulai Zhang
Shanghai Jiao Tong University
Shanghai, China
zslzsl1998@sjtu.edu.cn

Quan Chen
Shanghai Jiao Tong University
Shanghai, China
chen-quan@cs.sjtu.edu.cn

Weihao Cui
Shanghai Jiao Tong University
Shanghai, China
weihao@sjtu.edu.cn

Han Zhao
Shanghai Jiao Tong University
Shanghai, China
zhaohan_miven@sjtu.edu.cn

Chunyu Xue
Shanghai Jiao Tong University
Shanghai, China
dicardo@sjtu.edu.cn

Zhen Zheng
Microsoft
Beijing, China
zhengzhen@microsoft.com

Wei Lin
Alibaba Group
Hangzhou, China
weilin.lw@alibaba-inc.com

Minyi Guo
Shanghai Jiao Tong University
Shanghai, China
guo-my@cs.sjtu.edu.cn

Abstract

Data centers now allow multiple applications that have lightweight workloads to share a GPU. Existing temporal or spatial sharing systems struggle to provide efficient and accurate quota assignments. We observe that the performance of the multi-user system is often underestimated because of the existence of unused GPU "bubbles" and can be enhanced by squeezing the bubbles. Based on this observation, we design Bless, a bubble-less spatial-temporal sharing GPU system that fine-tunes the GPU resource allocation to improve multi-user performance. Bless leverages precise computing resource management and fine-grained kernel scheduling to ensure stringent quota guarantees and reduce latency fairly for applications with varying GPU quotas. We implement and evaluate Bless with multiple applications and workloads. Our result shows that Bless achieves 21.1% – 37.3% average latency reduction over the state-of-the-art while guaranteeing the promised quota for all applications.

CCS Concepts: • **Computer systems organization** → **Cloud computing**.

Keywords: GPU sharing, cloud computing

EuroSys '25, March 30–April 3, 2025, Rotterdam, Netherlands

ACM ISBN 979-8-4007-1196-1/25/03…$15.00
https://doi.org/10.1145/3689031.3696070

ACM Reference Format:
Shulai Zhang, Quan Chen, Weihao Cui, Han Zhao, Chunyu Xue, Zhen Zheng, Wei Lin, and Minyi Guo. 2025. Improving GPU Sharing Performance through Adaptive Bubbleless Spatial-Temporal Sharing. In *Twentieth European Conference on Computer Systems (EuroSys '25), March 30–April 3, 2025, Rotterdam, Netherlands.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3689031.3696070

1 Introduction

GPUs are widely used to run AI-enhanced applications [13, 15], scientific computing [27, 50], etc. While some applications (e.g., deep learning models [14] and video transcoding [16]) often have lightweight workloads, they cannot fully utilize a whole GPU [26, 32, 54]. The GPU utilization can be greatly improved by allowing multiple applications to share a GPU. To this end, current data centers [12, 17] allow an application to employ part of a GPU with a provisioned quota. *Temporal sharing* and *spatial sharing* are often used to multiplex a GPU. Figure 1 shows an example of two applications sharing a GPU with different multiplexing methods.

Temporal sharing allocates GPU time slices to applications by controlling the GPU kernel launching frequency based on each application's quota [10, 42, 55, 67]. However, due to the heterogeneity and un-preemptable nature of GPU kernels, applications cannot precisely occupy their provisioned quotas, as observed from Figure 1(a). As for spatial sharing [24, 28, 37], a GPU is spatially divided and statically allocated to applications according to their quotas. For instance, Nvidia's Multi-Process Service (MPS) [2] only divides and allocates GPU's SMs (streaming multiprocessors), letting the applications compete for global memory bandwidth. Therefore, applications may experience performance degradation compared to their required quotas due to the interference. Nvidia Multi-Instance GPU (MIG) [9] guarantees quota through physical isolation of GPU resources. As

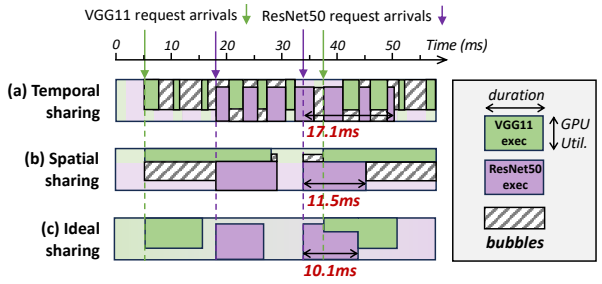


Figure 1. An example of handling two applications' requests on an Nvidia A100 GPU with a real-world trace [5]. The two co-located applications are VGG11 [57] (Quota: 1/3 GPU, latency: 10.2ms, GPU utilization: 81%) and ResNet50 [35] (Quota: 2/3 GPU, latency: 8.7ms, GPU utilization: 86%).

observed from Figure 1(b), through spatial sharing, even if an application does not use its assigned SMs, other applications are not permitted to use them.

Therefore, neither temporal sharing nor spatial sharing can provide efficient and accurate resource guarantees. As observed in Figure 1, many "bubbles" exist during a GPU's execution, resulting in low performance of applications. In this work, we aim to squeeze these bubbles to improve the multi-user GPU performance with applications' required quota guarantee. As shown in Figure 1(c), the ideal scenario demonstrates the elimination of the bubbles. The latency of the marked request decreases from 17.1ms with temporal-sharing and 11.5ms with spatial-sharing, to 10.1ms after bubble squeezing, without slowing down the other application.

Two challenges have to be addressed to achieve the ideal sharing depicted in Figure 1(c), even though the built-in GPU hardware scheduler is capable of filling the bubbles through kernel concurrency. The hardware scheduler is unaware of the properties of GPU kernels, resulting in interfered execution of concurrent kernels, increasing the latency of independent requests. Therefore, the first challenge is how to precisely schedule kernels to provide expected GPU quotas for applications. Since applications provisioned various quotas have distinct performance targets, a mechanism is required to manage the execution behavior of kernels from different applications. The second challenge is how to minimize the bubbles when heterogeneous kernels are running concurrently. It requires low-cost fine-grained resource management for kernels at runtime. The current resource management through narrow vendor primitives (e.g., CUDA streams [1], MPS contexts [2]) is coarse-grained and static, lacking the reconfigurability in a multi-user scenario.

This paper proposes **Bless**, a bubble-less spatial-temporal GPU sharing scheme. Bless allows each request to utilize the entire GPU whenever the resources are idle, and shrinks its resources instantly when other requests arrive. Bless leverages unused GPU resources to reduce the latency of co-located applications with GPU quotas provisioned. Bless also optimizes the concurrent execution of kernels when requests are overlapped to further reduce all requests' latencies.

Same to prior work on co-locating GPU applications in private data centers [22, 58, 73], Bless uses *offline profiling* (§4.2) to collect some performance data, with low overhead. The overhead is negligible for long-running applications in production data centers (e.g., AI inference services). Bless is feasible in public clouds if users allow the provider to profile their long-term applications beforehand for better future performance [52]. To schedule the concurrent requests from different applications, Bless designs a *multi-task scheduler* (§4.3) on the host side and launches kernels with the granularity of *kernel squads*. A kernel squad is a group of kernels from different applications. Since the duration of kernel squads is much shorter than the requests, we can realize resource re-configuration within the execution of requests, thereby achieving the precise GPU quota assignment. Bless optimizes the execution of kernel squads based on the observation that the execution duration under various concurrent configurations is predictable. Bless proposes an *execution configuration determiner* (§4.4) to identify the optimal kernel group execution configuration at runtime through the low-cost estimators. Bless precisely controls the compute resource that each kernel uses in a squad using a *concurrent kernel manager* (§4.5). Such precise control is implemented by launching kernels with different resource configurations to heterogeneous pre-established GPU contexts.

We have implemented Bless, and the user applications are built with TVM [6] and Pytorch. We evaluate Bless on an Nvidia A100 GPU, with various workloads and applications, as well as real-world query invocation trace loads [5, 74]. Experimental results show that Bless reduces the average latency by up to 37.3%, compared with SOTA GPU-sharing systems. In a more specific scenario, co-locating two BERT [62] inference applications on a GPU using Bless reduces average latency by 39.1%. When four BERT instances are co-located, the latency reduction reaches 41.2%. Furthermore, all applications can experience reduced latency compared to scenarios where applications are deployed with computing resources provisioned as quotas. The main contributions of Bless are three-fold.

1) Sophisticate analysis of "bubbles" when a GPU is shared by multiple applications. The analysis proves that it is feasible to improve the performance of some applications by eliminating the bubbles without hurting the performance of all applications.

2) The precise modeling of the kernel execution with temporal-spatial sharing. It allows Bless to build near-optimal kernel squads based on the accurate latency prediction at runtime with minimal cost.

3) A systematic solution that improves the applications' performance without mutual impact. The solution combines fine-grained scheduling and precise resource management for kernels to ensure stringent quota assignment.

2 Related Works

Various techniques could be used to improve GPU utilization and enhance the performance of co-located applications. Some previous works [19, 22, 33, 58, 68, 73] aim for a biased GPU-sharing system to handle both real-time and best-effort tasks. Among them, Reef [33] applies kernel fusion [45] to exploit intra-operator parallelism and kernel preemption to control the kernel launch sequence. Orion [58] mitigates the interference of co-located tasks through comprehensive offline profiling. However, biased sharing schemes reduce the latency of real-time tasks at the expense of best-effort task performance. They lack overall fine-grained scheduling for applications with fixed GPU quotas.

Many other works exploit spatial partitioning across SMs [2, 18, 38, 49, 61, 78] through runtimes and microarchitectural techniques. Multi-user GPU systems based on these methods provide static resources for applications [23, 24, 28, 30, 67] and lack the adaptability to manage resources to enhance performance. There are also software approaches, e.g., Paella [46], Reef [33] and so on [44, 51, 66], controlling the resource usage of applications through code transformation. There are also intra-SM partitioning techniques to partition resources within SMs [64, 65, 69]. However, these techniques are not equipped on commodity GPUs.

Some prior works [28, 29, 34, 43, 47, 70] adopt GPU-sharing for DNN inference systems to satisfy service-level objectives (SLO). Since the latency SLOs are much more relaxed than the solo-run latencies of applications, existing mechanisms are not feasible to guarantee each application's performance with a specific provisioned quota. As for training, Zico [41] and Wavelet [63] coordinate the forward pass and the backward pass of training iterations to reduce memory footprint.

To estimate the performance of concurrent kernel execution, Abacus [26] proposes a black-box predictor to predict the latency of overlapped operators without spatial isolation. There are also approaches to model the performance of concurrent kernel execution with spatial isolation [76, 77]. However, these modeling methods rely on underlying hardware information and are not applicable at runtime.

3 Background and Motivation

In this section, we introduce the background of GPU sharing, investigate the inefficiencies of current solutions, and discuss opportunities for bubble-less spatial-temporal sharing.

3.1 Workflow of GPU sharing

Figure 2 shows a general multi-user GPU-sharing workflow. The client applications comprise DAGs of operators, which are either launched as computational kernels or other kernels (e.g., memory management kernels and synchronization kernels). At the deployment stage, the host first initializes the resources (contexts, memories) for each registered application through vendor APIs and prepares GPU functions

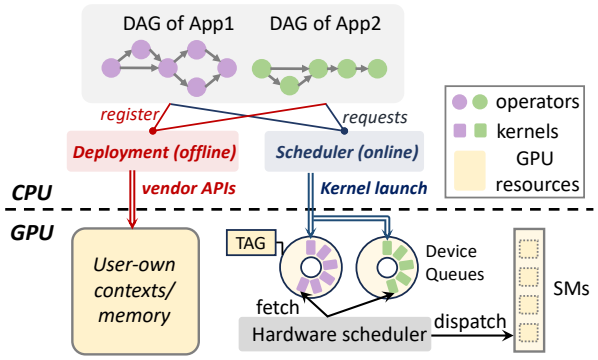


Figure 2. A general GPU-sharing workflow.

in GPU contexts. At runtime, multiple applications submit their requests to the host scheduler, which is responsible for further launching kernels into different device queues. Device queues are implemented as ring buffers to store kernels so that they can be accessed by the CPU and the GPU simultaneously through DMA [33, 59].

For computational kernels, the GPU hardware scheduler fetches blocks of kernels from device queues and dispatches them onto different streaming multiprocessors (SMs) , thereby enabling the concurrency execution of requests from different applications. For memory management kernels, they contend for the PCI-e bandwidth to transfer data between CPU and GPU. To isolate the resources that computational kernels use, developers can create GPU contexts with resource restriction (e.g., `cuCtxCreate_v3` for Nvidia MPS [2])) for various applications. Subsequently, kernels from different contexts would be spatially restricted as delivered to context-bonded device queues **tagged** with resource restriction. For all GPU kernels including memory management kernels, the host scheduler can also control the launch timing to control the kernels' execution sequence on the GPU.

3.2 Inefficiencies of existing GPU sharing solutions

To handle requests from multiple applications, the host is responsible for organizing and scheduling kernels with various mechanisms. We review the state-of-the-art scheduling schemes in GPU multiplexing and discuss the performance issues in scheduling kernels from heterogeneous applications. To highlight their performance gaps, we conduct an experiment to measure the latencies of executing a VGG11 request and a ResNet50 request simultaneously using various schemes. The experimental results are shown in Figure 4(b).

Static sharing. To isolate the performance of requests from co-located applications, state-of-the-art solutions [23, 28, 30] assign fixed resources to kernels through vendor mechanisms [2, 3, 9]. Within the lifecycle of a request, its assigned SM/memory resources are unchanged to achieve stable latencies. However, such a static isolation scheme will produce many un-exploited GPU bubbles, as shown

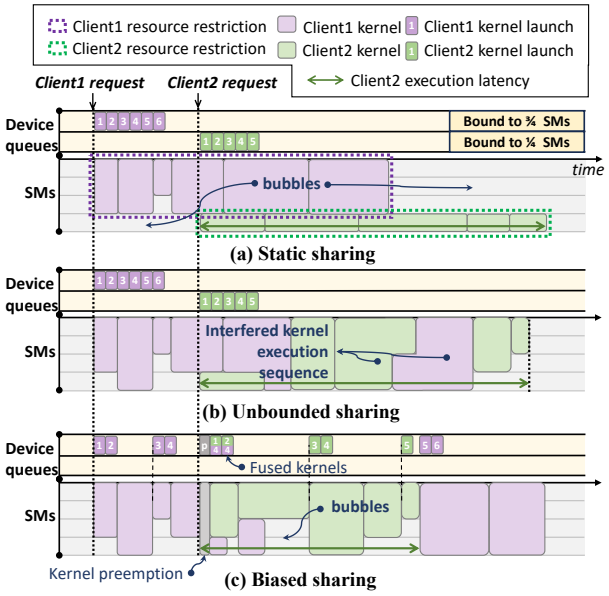


Figure 3. An example of GPU scheduling schemes when handling requests from two clients simultaneously. The GPU has 4 SMs. Request of client1 has 6 kernels while that of client2 has 5 kernels.

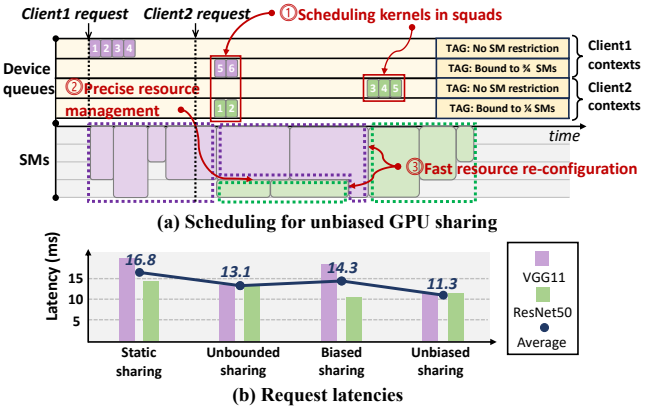


Figure 4. (a) The scheduling scheme for unbiased GPU sharing. (b) The latencies of executing a VGG11 request and a ResNet50 request simultaneously using various schemes.

in Figure 3(a). In our experiment, the static sharing scheme provides an average 16.8*ms* latency for VGG11 and ResNet50.

Unbounded sharing. To fully utilize the GPU computational resource, each client can be assigned an MPS context or a stream and then rely on the hardware scheduler to utilize the entire GPU without any SM restrictions. However, when kernels from different MPS contexts/streams are co-located, the execution order of kernels is interfered as shown in Figure 3(b). Thus, despite the high GPU utilization, the latency of each request is neither predictable nor optimal (13.1*ms* on average in Figure 4(b)). Wavelet [63] and Zico [41] utilize unbounded sharing to overlap the training iterations of multiple models, improving the GPU utilization.

Biased sharing. The above schemes launch kernels in the request granularity. It means that once a request arrives, all kernels of the request would be launched into device queues asynchronously. The host loses control of the launched kernels in this case. To this end, Reef [33] launches kernels periodically and realizes deterministic kernel execution through kernel fusion. The latency of the real-time request (client2 in illustration) can be well maintained, while the performance of the co-located best-effort tasks is sacrificed. Meanwhile, the kernel fusion method for co-location would also introduce bubbles as shown in Figure 3(c). With this scheduling scheme, the latency of the real-time task is 10.4*ms* and the latency of the best-effort task is 18.2*ms* (14.3*ms* on average). The sharing strategy of Orion [58] is also designed for prioritized jobs and falls under biased sharing.

3.3 Opportunities

To tackle the inefficiencies discussed above, we present a better scheduling scheme for unbiased multi-user GPU-sharing in Figure 4. The scheme promises better utilization and lower latency of requests from three aspects.

Scheduling kernels in squads. When requests do not overlap, Client1's request can use all available resources on GPU with no restriction. When Client2's request arrives, the scheduler has to shrink Client1's resources to execute Client2's kernels instantly. Kernel-level preemption [21, 33] is promising for the precise schedule of kernels but requires modification of user code. To avoid interfering with user code, we schedule kernels in fine granularity and lazily wait for their completion rather than preempting them.

Nonetheless, finer scheduling granularity is not always better. With the advent of powerful compute units (e.g., tensor cores), typical matrix-multiplication kernels' duration can be reduced to several microseconds, which is comparable to the kernel launch duration (around 3*us*). To conceal the kernel launch time, kernels should be scheduled in groups with a proper granularity. In Bless, we refer to these groups as **kernel squads**. In the illustration, the number of kernels in each kernel squad is restricted to 4.

Precise resource management. To reduce the latencies of all co-located requests, the duration of kernel squads should be reduced as much as possible. It is achievable by sufficiently utilizing unused GPU resources through tailored resource management for kernels. Comprehensive modeling of concurrent kernels is required to determine optimal resource management within kernel squad execution.

Fast resource re-configuration. Resources for kernels have to be re-configured across and within kernel squads to avoid idle GPU time. Since the switch between device queues with different SM restrictions can be achieved efficiently

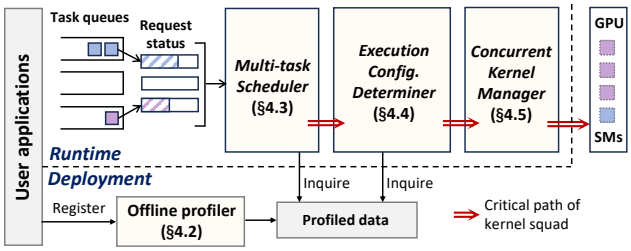


Figure 5. System overview of Bless.

through MPS context APIs, we realize quick resource reconfiguration by maintaining multiple MPS contexts.

With fine-grained kernel squad scheduling and delicate computing resource management, the execution of all co-located requests can be optimized, and the average latency can be reduced to 11.3*ms* in the experiment in Figure 4(b).

4 Bless Methodology

We propose Bless to enhance performance for concurrent GPU applications with quota provisions in an unbiased manner. In this section, we first take an overview look at Bless and then introduce the components in Bless separately.

4.1 Overview of Bless

Bless comprises an offline deployment stage and an online runtime as shown in Figure 5. At the deployment stage, each application registers with the system and Bless leverages an *offline profiler (§4.2)* to profile the performance of kernels (including computational kernels as well as other kernels) with varying computational resources through MPS. According to the profiled data, Bless accepts stationary applications with deterministic computation patterns at runtime, and the applications are required to invoke GPU functions at the granularity of GPU kernels. As for the runtime system, there are three components: *multi-task scheduler*, *execution configuration determiner*, and *concurrent kernel manager*.

Multi-task scheduler (§4.3). In Bless, each application has its own dedicated task queue. The multi-task scheduler handles the requests of each application in a FIFO order and only processes requests of each application one at a time. The scheduler tracks the request status using a kernel queue. It selects kernels from concurrent active requests to form a **kernel squad**.

Execution Configuration Determiner (§4.4). The execution configuration determiner is tasked with optimizing the execution of kernel squads. In each scheduling round, it estimates the execution duration of kernel squads under various configurations using two performance estimators, ultimately selecting the best configuration for execution.

Concurrent kernel manager (§4.5). The concurrent kernel manager is responsible for launching kernels to different MPS GPU contexts. During runtime, it launches kernels as scheduled by the multi-task scheduler and manages resources following the optimal configuration searched by the execution configuration determiner. Bless runtime on the host side runs parallel with the kernels on the GPU.

4.2 Offline Profiling

Bless utilizes an offline profiler to obtain the overall performance of applications as well as the statistics of detailed kernels. Bless leverages the profiled data to determine whether a registered application can be deployed and the appropriate deployment strategy. The profiled data is also required for the scheduler at runtime. Without profiling, the scheduler of Bless would be degraded to the plain MPS scheduler.

4.2.1 Application profiling. For each application provisioned with $n\%$ percentage of the GPU, we identify the isolated latency $T[n\%]$ of it running with MPS, as well as its required GPU memory size. For each kernel k, we record its duration with $n\%$ SMs as $t[n\%][k]$ and the duration from the beginning of the application to the end of k as $\tau[n\%][k]$. We also record the proportion of the maximum active SM usage $d\%$ for each kernel.

In practice, the profiler first runs the application one time to obtain its overall performance. Then, the profiler runs the application for another N times to obtain the duration of kernels under different SM configurations using MPS. N represents the number of GPU partitions. When N is large, the execution configuration search space at runtime would be unnecessarily large. We empirically set $N = 18$ for Nvidia A100 GPU (108 SMs) to prevent the explosion of configuration search space as well as large profiling overhead. Then the kernels are measured on 6%, 12%, ..., 100% of the GPU leveraging CUDA events. Suppose there are M applications, then the overall profiling complexity is $O(MN)$.

Bless avoids using heavy Nsight Compute [8] or Nsight System [4] to profile other detailed properties of kernels. The offline profiling finishes in seconds (1.9 seconds on average) as evaluated in Table 1. For stationary applications such as inference or training jobs that may run hours or days [36, 40, 56], the profiling cost at the deployment stage is amortized and thus negligible. Note that for applications with fixed patterns such as training, only one iteration is profiled. The profiling requires the same GPU as runtime. If production systems use various types of GPUs, then profiling needs to be done on a same-model GPU as the target for runtime.

4.2.2 Deployment according to profiled data. Once all the applications are profiled, the profiled data helps to make the deployment decision. According to the kernel duration information, Bless avoids placing applications with short kernels and applications with extremely long kernels together beforehand, thereby preventing the former applications from being starved in every kernel squad. Bless works well to co-locate most deep learning applications, with the average kernel duration varying from 10*us* to 300*us*. Before

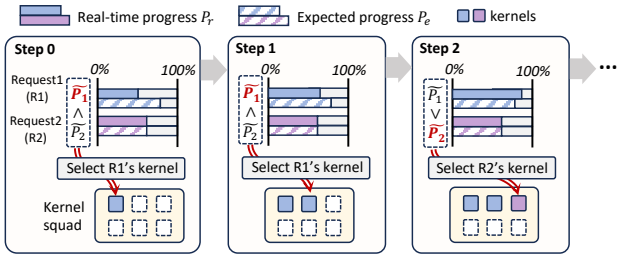


Figure 6. An example of kernel squad generation.

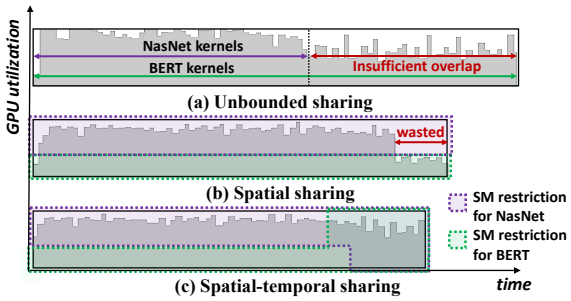


Figure 7. Optimizing a kernel squad (NasNet: 58 kernels, BERT: 142 kernels). Case(a): No spatial restriction. Case(b): 66 SMs for NasNet/42 SMs for BERT. Case(c): The first 80% kernels are spatially restricted while the restriction is removed for the rear 20%.

deploying each application, Bless also checks the remained memory capacity on the GPU and guarantees that the placement would not cause out-of-memory.

As for the scenario in which applications have to be coordinated and deployed on multiple GPUs as GPUlet [23], Bless can also be extended by replicating its runtime components for each active GPU. In such a case, a central controller can leverage the memory requirement and profiled kernel information to decide which specific GPU to place applications to avoid conflict.

4.3 Multi-task Scheduling

Once the offline deployment of applications completes, Bless starts the runtime multi-task scheduler. The objective of the scheduler is to approach the isolated latency target and reduce the latency unbiasedly of all co-located applications.

4.3.1 Request progress perception. The scheduling of kernels depends on the status of active requests as well as the isolated latency target $T[n\%]$. The core concept of unbiased sharing is to evenly distribute the execution progress among concurrent active requests.

During each scheduling cycle, the multi-task scheduler keeps track of the last scheduled kernel k and estimates the elapsed time $t_e[n\%]$ from the arrival of each active request to the end of the tracked kernel. As introduced in §4.2, the expected value of $t_e[n\%]$ is $\tau[n\%][k]$. Thus, the real-time progress of the request at the scheduling time is approximately $P_r = t_e[n\%]/T[n\%]$, while the expected duration progress $P_e = \tau[n\%][k]/T[n\%]$. To achieve latencies lower than the isolated latencies, the real-time progress P_r should be greater than P_e for each co-located application.

4.3.2 Schedule with kernel squads. To fairly reduce the latencies of all co-running requests, the kernel squad is generated as illustrated in Figure 6. In each generation step, a kernel is selected from the request that has the shortest relative progress $\tilde{P} = P_r/P_e$, indicating it is falling behind by other active requests and is the most urgent for execution. The process is terminated in two situations: (1) The kernel count surpasses the predefined maximum number of kernels per squad, which is 6 in Figure 6. (2) The selected kernel is the last kernel of a request. Note that the maximum number of kernels per squad is an empirical parameter that trades off the request latency and GPU sharing ability (§6.7). Smaller squads indicate finer scheduling granularity and thus can provide a more delicate sharing ability for the GPU, while larger squads minimize the impact of kernel squad switching due to prolonged kernel squad durations.

Since we check the progress of each request every time we generate a kernel squad, we in fact schedule and compensate for lagged requests at a fine granularity. Thus, even when a biased slowdown occurs because of various interference (e.g., inter-SM interference, internal memory interference, and DMA/PCI-e interference), multi-task scheduling is still able to ensure fair execution.

4.4 Execution Configuration Determiner

The execution configuration determiner aims to find the optimal execution configurations for kernel squads to reduce kernel squad duration, assuming they have already been generated by the multi-task scheduler.

4.4.1 Execution configuration space. In this part, we begin by showcasing the opportunities for execution optimization. We then elaborate on the configuration space to search for the optimal configuration.

Spatial sharing within the kernel squad. Enabling concurrent kernel execution within the kernel squad is crucial. To further reduce the duration of kernel squads, it is important to constrain the resources that kernels use within the squad. Reasonable spatial sharing can effectively increase GPU utilization within a kernel squad that consists of kernels from different device queues. As shown in Figure 7(a), relying on the scheduling of the hardware scheduler, the execution sequence of kernels is uncontrollable and the insufficient overlapping among kernels induces low GPU utilization. Through strict spatial partitioning (Figure 7(b)), the time slice with low GPU utilization is shortened, resulting in a shorter squad duration (from 8.5ms to 7.3ms) and increased average GPU utilization (from 69.7% to 73.5%).

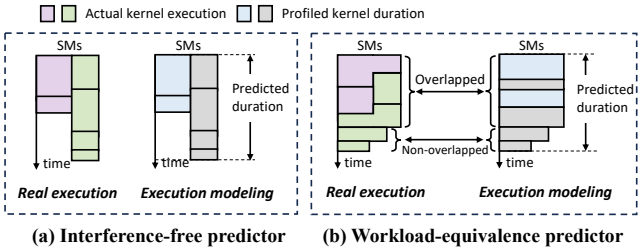


Figure 8. The kernel squad performance estimators.

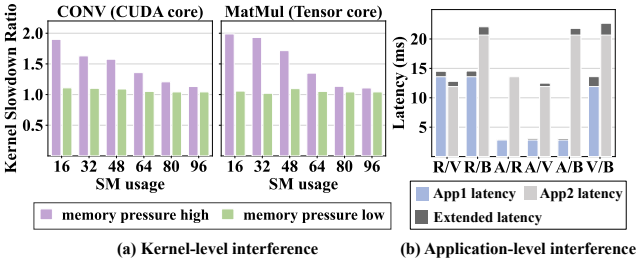


Figure 9. (a) Kernel-level interference. (b) Application-level interference (R:ResNet50, V:VGG11, A:AlexNet, B:BERT).

Spatial-temporal sharing within the kernel squad. As shown in Figure 7(b), strict spatial partitioning within the kernel squad is suboptimal because it cannot perfectly ensure that the kernels from different requests finish simultaneously. Thus, we can remove the SM restriction for kernels at the rear of the kernel squad, allowing them to utilize the full GPU (Figure 7(c)) in a spatial-temporal manner. In Bless, we realize it through the lightweight context switch promised by the concurrent kernel manager (§4.5). The kernel squad duration is then reduced from 7.3*ms* to 6.9*ms*, and the GPU utilization is increased from 73.5% to 81.4%.

To model and estimate the kernel squad performance, we assume that the computational restriction for each request is static within the kernel squad. Thus, for each kernel squad, the configuration space includes the no computation restriction case (Figure 7(a)), as well as C_{N-1}^{K-1} configurations that promise stringent spatial isolation (Figure 7(b)), where K refers to the number of active requests and N refers to the number of SM partitions. With an Nvidia A100 GPU split to $N = 18$ partitions when there are 2 active requests, the size of the configuration space is $C_{18-1}^{2-1} + 1 = 18$.

4.4.2 Kernel squad performance estimator.

We then leverage two performance estimators to predict the duration of kernel squads with different execution configurations. The purpose of the estimators is to find the configuration with which the kernel squad runs fastest at runtime. Thus, the design of the estimators takes both the runtime efficiency and prediction accuracy into consideration.

Interference-free predictor. When kernels from different requests are strictly isolated, we propose the interference-free predictor (Figure 8(a)). The estimated squad duration is calculated as Equation 1, which is the time of the longest stacked up durations of each active request's kernels. In Equation 1, k_i^j refers to the i-th kernel of the application j in the squad, $n^j\%$ is the SM restriction proportion of application j and R is the set of active requests within the kernel squad.

$$\hat{t} = \max_{j \in \mathrm{R}} \Big(\sum_i t[n^j\%][k_i^j] \Big) \tag{1}$$

We test the predictors with 1500 pair-wise combinations from Bless's testbed and the average prediction error of the interference-free predictor is 6.7%. The interference among kernels is not considered but it does not influence the prediction accuracy much. We analyze the reasons as follows: As most inter-SM interference comes from the L2 cache conflict and competition for bandwidth [76, 77], the large L2 cache size and high bandwidth on high-performance GPUs have restricted the slowdown of kernels. As shown in Figure 9(a), we examine the inter-SM interference by testing the slowdown of kernels under different memory pressures. The kernel-level slowdown ratio is no larger than 2 even when co-locating with a highly memory-intensive kernel. At the application level as shown in Figure 9(b), the average slowdown caused by interference is 7% when co-locating applications in mutual pairs.

Workload-equivalence predictor. When kernels from all requests are not strictly spatially isolated, we propose the workload-equivalence predictor. In this case, the SM usage of a kernel is not only restricted but also influenced by other kernels, and the contention between kernels is unknown. As shown in Figure 8(b), we first estimate the kernels that would overlap with each other. The execution of overlapped kernels is then modeled as a sequential execution where every kernel occupies all active SMs. The durations of non-overlapped kernels are added to the total duration. Thus, the predicted duration is calculated as Equation 2.

$$\hat{t} = \sum_{i=0}^{q} \sum_{j \in \mathrm{R}} t[\sum_{j \in \mathrm{R}} d_i^j\%][k_i^j] \tag{2}$$

In Equation 2, $d_i^j\%$ is the original SM usage proportion of k_i^j without restriction, q is the maximum number of kernels among each request in the squad, and the durations are summed up in a breadth-first manner over requests in the kernel squad[1]. Notice that the duration of a kernel using the desired number of SM is interpolated if it cannot utilize so many SMs. The average prediction error of the workload-equivalence predictor is 7.1%.

In both estimators, the durations of memory management kernels (e.g., H2D, D2H memcpy) are added together with

[1] In our test, Volta and later architecture's hardware scheduler provides a simple mechanism to fairly schedule kernels from equal-priority device queues, according to the buffered kernel number in queues.

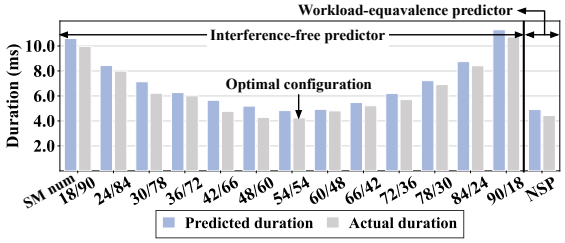


Figure 10. The performance of two performance estimators to predict a {NasNet+ResNet50} squad. X-axis: different configurations (NSP for no spatial restriction).

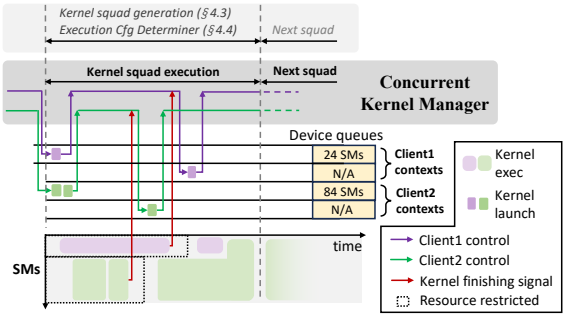


Figure 11. Control flow of the concurrent kernel manager.

the durations of computational kernels, no matter if they actually overlap at runtime. This may cause the predicted duration to be slightly longer than the actual duration. However, since the predicted duration is extended similarly for all configurations, such an approximation rarely influences the correctness of selecting the optimal execution configuration. As shown in Figure 10, we can identify the optimal execution configuration with the help of the predictors. In this case, the predicted optimal configuration is 54SMs/54SMs, which is indeed the actual optimal spatial split ratio. We select part of kernels with different start and end positions from the inference applications in Table 1 and combine them to form 2260 kernel groups. In such a testbed, the percentage that the predicted optimal configuration matches the real optimal scheme is 96.2%. Even when unexpected contentions lead to a sub-optimal kernel squad and applications are slowdowned unevenly, the multi-task scheduler (§4.3) can still promise fairness by selecting more kernels from lagged applications when generating the next kernel squad.

4.5 Concurrent Kernel Management

In this subsection, we introduce the concurrent kernel manager. We first explain how it realizes the spatial resource management for kernels. Then, we introduce its operating mechanism to interact with multiple GPU contexts.

4.5.1 Kernel resource management. There exist multiple methods that support computational resource restriction, including software methods based on code transformation [66, 75]. However, code transformation methods are intrusive and require additional registers and shared memory for kernels, the extended latency is non-negligible (about 12% on average in our testbed). Additionally, to ensure applicability to modern commodity GPUs, we refrain from making modifications to the hardware architecture.

We implement Bless on Nvidia GPUs. While utilizing MPS, we employ the `cuCtxContext_v3` API to create a context with SM affinity which specifies the number of SMs that the context is limited to use. Then the device queues bonded to this context and all the kernels sent to them would be spatially restricted. Bless can also be adapted for other GPUs (e.g., AMD GPUs with the HIP runtime) with similar resource management properties and kernel scheduling logic [48]. The methodology of Bless is general for accelerators with spatial and temporal sharing support.

4.5.2 GPU context switch. The concurrent kernel manager is responsible for launching kernels to different GPU contexts with runtime control. Launching kernels into different contexts of a client by instant context switch can realize the spatial-temporal sharing within the kernel squad (§ 4.4.1).

As shown in Figure 11, when a kernel squad is configured with spatial sharing, kernels from different requests are first launched into contexts that are spatially restricted. To facilitate spatial-temporal sharing within the squad, for each co-running request, the kernel manager synchronously waits for all the kernels from the previous context to finish before sending kernels to the next context without SM restrictions. Thus, for all the kernels of a request in this kernel squad, there are $c\%$ spatially restricted. The left $1-c\%$ kernels are not spatially restricted and would content for SMs with kernels from other requests freely. The tunable parameter **split ratio** $c\%$ provides a tradeoff between the no spatial-restriction execution (Figure 7(a)) and the spatial partitioning execution (Figure 7(b)), which is set to 50% empirically (§6.7).

5 Implementation

We build Bless in 5,000+ lines of C++ code. Bless provides a gRPC interface for multiple users to deliver requests into their own task queues. The inference applications are DNN models compiled with nnfusion [45] with kernels generated by TVM [6, 79]. The training applications are based on the Pytorch framework.

At the deployment stage, each client is allocated a default CUDA context with no resource restriction and several MPS contexts with stringent resource restrictions. All kernel functions of each application are then injected into the allocated GPU contexts. The memory for each client is also allocated in advance and mapped into corresponding contexts.

At runtime, the multi-task scheduler runs a background process to collect the status of active requests from multiple applications. The concurrent kernel manager runs in another

Table 1. Applications used in benchmarks.

		VGG	R50	R101	NAS	BERT
Inference	Duration (ms)	10.2	8.7	17.2	32.7	12.8
	# of kernels	31	80	148	458	382
	Profile cost (s)	0.56	0.38	0.77	1.61	0.50
Training	Duration (ms)	11.2	25.2	40.1	157.8	186.1
	# of kernels	80	306	598	2824	5035
	Profile cost (s)	0.49	0.59	0.82	6.31	6.88

process to take control of all established GPU contexts. Bless launches kernels with a unified API `LaunchKernel(kernel, kernel_args, tag)` when Bless is privileged to manage the submission of operations. Otherwise, the unified API is a wrapper of the CUDA runtime APIs (e.g., `cudaLaunchKernel`, `cudaMalloc`, `cudaMemset`). It intercepts and remotes the CUDA API calls from frameworks such as Pytorch, through a dynamically linked library. The concurrent kernel manager leverages the API to automatically send kernels to client-owned contexts according to the kernel's resource restriction. The kernel manager listens on the return status of kernels and synchronizes kernels in different contexts with the `cudaDeviceSynchronize()` API to ensure the correct execution sequence of kernels.

6 Evaluation

6.1 Experimental setup

We conduct our experiments on a GPU server, which consists of one AMD EPYC 7302 16-Core CPU and one Nvidia A100 GPU (108 SMs and 40GB memory). The software environment is configured with CUDA 11.8 and Ubuntu 20.04.

Benchmarks. We use the inference and training of five typical DNN models as the multi-user applications: VGG-11(VGG) [57], ResNet50(R50) [35], ResNet101(R101) [35], NasNet(NAS) [80], and BERT [62]. The applications are highly heterogeneous using various compute cores, with kernel durations varying from 3us to 3ms. The properties of applications used are shown in Table 1. Notice that inference and training are based on different frameworks and compute cores, providing us with highly heterogeneous applications, with kernel durations varying from 3us to 3ms. For inference, BERT uses tensor cores while other applications use CUDA cores. For training, the applications are directly implemented with Pytorch 2.1. Thus, the training duration is not proportional to the corresponding inference duration. The profiling cost is also not proportional to the application's duration because the SM affinity of applications are also different.

We use 5 inference workloads to test Bless under different request loads: (A) High load, (B) Medium load, (C) Low load, (D) Real-world trace load, and (E) Biased load. In the workloads (A,B,C), the deployed applications issue requests in closed-loop, while the interval between requests is set to 1/3, 2/3, 1 of each model's solo-run latency (Thus, the QPS of the low load is identical to the low load in Reef [33]). Workload (D) involves two real-world traces: a widely-used Twitter request trace [5] in multi-user inference systems [20, 53, 71] and a cloud serverless function trace [74]. The workload(E) provides an extremely biased workload (consisting of an R50 that provisions a quota of 8/9 GPU but the load is low, as well as another application that provisions a quota of 1/9 GPU but the load is high). Additionally, for training workloads, we deploy two models to evenly share the GPU for fair comparison among all sharing schemes for training. The detailed configurations are shown in Table 2.

Table 2. Workloads used in benchmarks.

	A	B	C	D	E
Workloads	High load	Medium load	Low load	Real-world traces [5, 74]	Biased load
Deployed Models	5 models (symmetric), R50 + 4 others (asymmetric)			5 models (mutual pair-wise)	R50 + 4 others
2-model quotas	$(\frac{1}{3}, \frac{2}{3}), (\frac{7}{18}, \frac{11}{18}), (\frac{4}{9}, \frac{5}{9}), (\frac{1}{2}, \frac{1}{2}), (\frac{5}{9}, \frac{4}{9}), (\frac{11}{18}, \frac{7}{18}), (\frac{2}{3}, \frac{1}{3})$				
4-model quotas	(10%, 20%, 30%, 40%)				
8-model quotas	(5%, 5%, 10%, 10%, 15%, 15%, 20%, 20%)				

Comparing Targets. We first identify the right latency targets of applications with specific GPU quota provisioned and refer to the target as **ISO**. ISO is the baseline in which each application is provisioned with the SM quota as assigned and running isolatedly using MPS. This is the ideal scenario where multiple users are spatially partitioned with no slowdown caused by interference.

We then compare Bless with systems that facilitate GPU sharing among users with varying quotas, including **Temporal**: Multiple applications temporally share the GPU through round-robin time slice scheduling and context switch, **MIG** [9], and **GSlice** [28]: Multiple inference applications spatially share the GPU through MPS and an adaptive algorithm is applied when workload changes.

We further compare Bless with mechanisms that evenly share the GPU with typical scheduling schemes. **Unbound**: Multiple applications share the same GPU unboundedly with MPS or CUDA streams. **Reef+**: Reef [33] enables instant preemption and biased GPU concurrency control. We improve the controlled concurrency method of Reef and replace its kernel padding with MPS to support even spatial partitioning. Reef+ is adopted in the inference scenario. **Zico** [41]: Zico unboundedly shares the GPU within the iterations of multiple training requests. Zico is adopted only in training.

6.2 Measuring the performance

We use a latency chart to illustrate the users' performance in the scenario where two applications are deployed on a GPU. As shown in Figure 12, each point in the chart refers to the average latencies of two applications' requests under a specific GPU quota assignment configuration.

The latency of pair-wise applications is constrained within the mint green region. The origin of this region represents

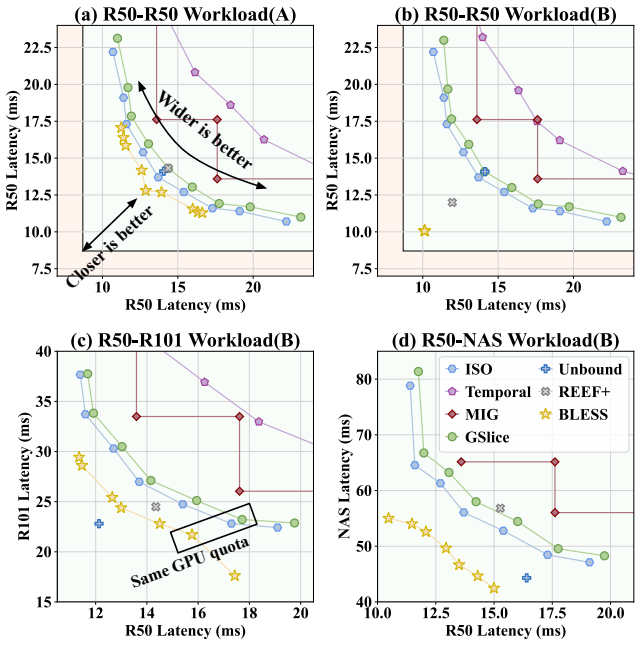


Figure 12. The latency chart of two applications deployed on a GPU with Bless. Case(a) and (b) are with symmetric workloads. Case(c) deploys two applications with homogeneous kernels while Case(d) with heterogeneous kernels.

a scenario where two applications use the entire GPU independently. The closer the points are to the origin, the lower the average latencies of applications. Additionally, a wider range of coordinates indicates the system's ability to deploy applications with more diverse quota configurations.

Thus, we propose two metrics to evaluate the efficiency of a multi-user system. First, the **average latency** of requests from different applications under a specific quota assignment configuration. Second, the **average latency deviation** across various quota assignments. Under a specific quota assignment $n^j\%$ for application j, the baseline latency target for the application is the ISO latency $T^j[n^j\%]$. The latency deviation of a system is then calculated as $\sum_j \max(T^j_{sys}[n^j\%] - T^j[n^j\%], 0)$. The larger the latency deviation, the worse the performance of the mechanism under this certain quota assignment. The average latency deviation indicates the flexibility of multi-user systems for various quota assignments.

6.3 Overall performance

We evaluate Bless and other approaches with heterogeneous applications, workloads, and GPU quota assignments.

Performance with symmetric workloads. We first examine the reduction of end-to-end latencies when two applications share a GPU evenly. As shown in Figure 13, Bless can provide at most 26.9%, 37.1% and 47.6% latency reduction in workload(A), (B), and (C) separately, compared to schemes with other scheduling policies. For inference workloads, Bless reduces the latency by 37.3%, 34.2%, 21.1%, 16.5%, and 13.5% on average compared to Temporal, MIG, GSlice, Unbound, and Reef+ respectively. We also evaluate Bless when all inference requests arrive continuously, in which case the GPU is fully saturated and there are no bubbles that can be utilized. Bless then provides a similar latency with GSlice (with lower than 3% latency extension rooted from scheduling overhead).

Bless can provide shorter latencies than other systems especially when the request load is low. The phenomenon is also observed in Figure 12. When the load is lower (Figure 12(b)), the end-to-end latency point is closer to the origin. It is because when the load is low, Bless can (1) fully utilize the bubbles that Temporal, MIG and GSlice waste, (2) organize the kernels reasonably to avoid the interfered execution that prolongs the request latency with Unbound, and (3) further shorten the execution of kernel squads than Reef+ leveraging precise resource management.

In the training scenario, Bless reduces the average latency of a training epoch by 26.5%, 7.5%, 12.5% and 9.9% compared to Temporal, MIG, Unbound, and Zico respectively. GSlice and Reef+ are not compared in training because they are designed only for inference systems.

Performance with uneven quotas. We then test the average latency deviation of 5 symmetric deployed applications as well as 4 asymmetric deployed applications under 7 different quota assignment configurations with various loads. As shown in Figure 14, Bless maintains a rather low latency deviation. MIG fails to provide such diverse quota configurations because of its fixed hardware isolation. The average latency deviation of Temporal, GSlice, and Bless is 14.3ms, 2.1ms, and 0.6ms, separately.

To be specific, in all 4 cases shown in Figure 12, the latency of the application pairs is shorter than ISO under all quota assignment configurations. Temporal performs the worst because its GPU utilization is the lowest. GSlice endures higher latencies than the isolated baseline because of the interference between requests. Unbound and Reef+ can possibly provide latencies shorter than the isolated baseline with even GPU quota assignment. However, since the execution and resources of co-located kernels are not controllable in Unbound and Reef+, the latency deviation is large under various quota assignment configurations.

Performance with real-world traces. We use 10 mutual pairs of 5 DNN inference applications to examine the performance of Bless with real-world traces. With the twitter trace, when the assigned quota for each client is 50%, Bless reduces the latency by 18.4%, 20.5% and 7.3% compared with Temporal, MIG and GSlice separately. When the assigned quota for each application is uneven $(1/3, 2/3)$, the latency is reduced by 14% compared with GSlice and no latency deviation compared with ISO. The latency reduction is lower because the tenancy workload is dense and the extra bubbles

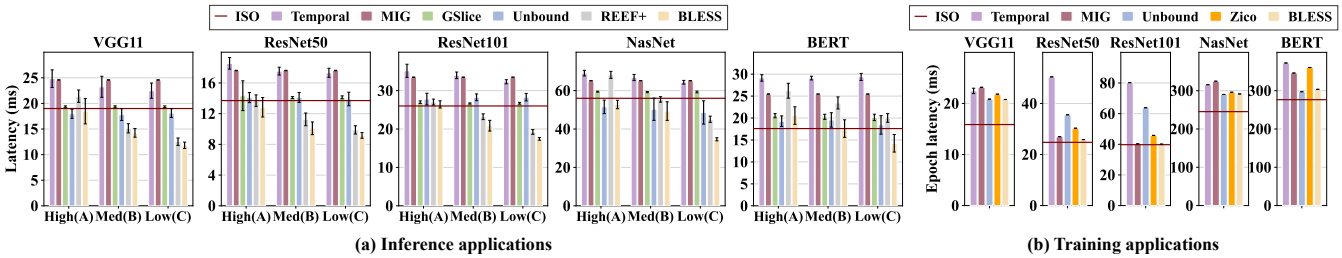


Figure 13. Average latency of two symmetric applications with even GPU quotas provisioned.

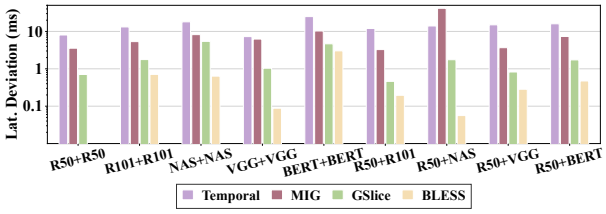


Figure 14. Average latency deviation of 9 pair-wise applications with uneven GPU quotas. The Y-axis is in log scale.

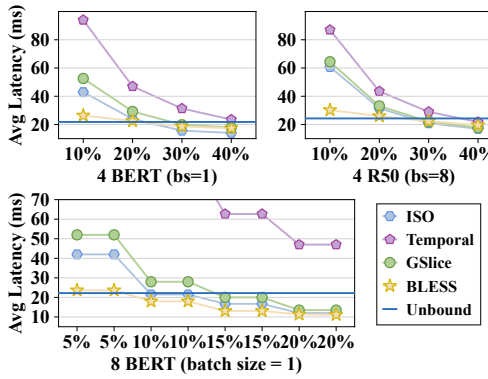


Figure 15. Performance of multiple co-located requests.

that can be utilized are not abundant. With the Microsoft Azure Trace [74], a real-world function trace, Bless reduces the latency by 49.3%, 41.2% and 32.1% compared with Temporal, MIG and GSlice separately on average. The reduction mainly comes from the abundant bubbles originating from the low load feature of this trace.

6.4 Beyond Pair-wise Sharing

In this subsection, we examine the ability of Bless to deploy more than two applications. In the experiment shown in Figure 15, the requests from 4 applications and 8 applications arrive at the same time. In this experiment, since the optimal spatial partitioning configuration of kernels cannot be determined at runtime in Reef+, we only compare Bless with Temporal, GSlice, and Unbound.

When four applications co-run, Bless reduces the average latency by 41.2% and 18.3% compared with Temporal and GSlice. When eight applications co-run, the average latency is reduced by 80.8% and 35.5% separately. Moreover, Bless's latency deviation is 0 in this scenario, while Temporal and GSlice have an average 74*ms* and 5*ms* latency deviation separately compared with ISO. Unbound does not support uneven GPU quota assignment, and it provides approximate latencies for all identical applications. Compared with ISO, Unbound endures a 3.8*ms* latency deviation. With the increase in the number of co-located applications, the bubbles with Temporal and GSlice drastically increase, and the interference among requests deteriorates in Unbound. Bless becomes more prominent with more co-located applications.

Performance with biased workloads. With the extremely biased workload(E), App2 consistently submits requests to the GPU with extremely dense workloads. Then, as shown in Figure 16, App1's latency with GSlice is 6% longer than ISO because of interference. Bless also suffers an average 9% latency increment because of its property to lazily wait for the end of execution of kernel squads. The slight sacrifice of App1's latency brings the high throughput of the co-located App2's requests. Bless provides an average 2.2× throughput improvement for App2 compared with GSlice.

6.5 Guaranteeing SLOs

Bless can natively guarantee the SLO of applications without modifying any internal mechanisms. It is achievable by simply replacing the isolated latency $T[n\%]$ in §4.3.1 with the required QoS target. We use two sets of experiments to test the performance of Bless in promising SLOs: (a) The QoS target is tight (1.2× and 2× isolated latency for two applications separately) and a medium workload (Workload(B)) is applied. (b) The QoS target is loose (1.5× and 3× isolated latency) and a heavy workload (Workload(A)) is applied. While Unbound and GSlice cannot promise QoS with a 38.8% and 50.1% of QoS violation on average, Bless has a merely 0.6% QoS violation, indicating its superiority in guaranteeing SLOs.

6.6 Performance of kernel squads

Bless reduces the kernel squad duration by managing computing resources delicately. In this subsection, we conduct experiments in the scale of kernel squads to analyze the effect of various proposed mechanisms in Bless.

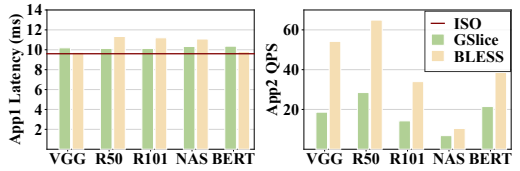


Figure 16. The performance of Bless when App1 (R50, 8/9 GPU) is deployed together with others (App2, 1/9 GPU).

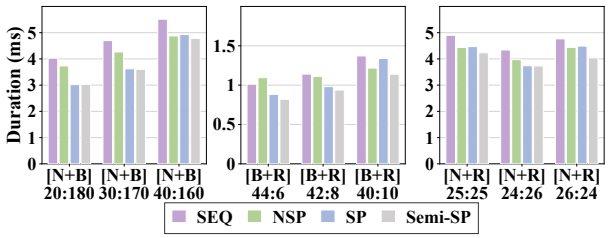


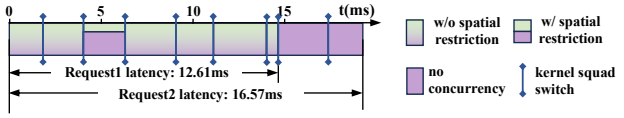
Figure 17. The kernel squad duration. The three application pairs are {NAS+BERT}, {BERT+R50}, and {NAS+R50}.

Optimize the kernel squad. To analyze the effectiveness of the execution configuration determiner in optimizing the kernel squad, we select three pairs of applications and observe the kernel squad duration using different optimization methods, as shown in Figure 17. SEQ refers to the scenario where kernels are executed sequentially from one device queue; NSP refers to the configuration where kernels are not spatially restricted and contend freely for SMs; SP refers to the scenario where kernels are strictly spatial partitioned with the optimal configuration using MPS; Semi-SP refers to how the spatial restrictions are removed for the last 50% kernels of each request for better spatial-temporal sharing.

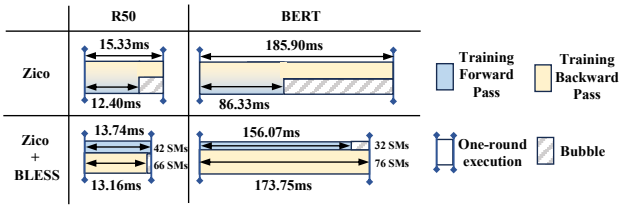
Compared with SEQ, the duration of squads with NSP, SP, and Semi-SP is shortened by 6.5%, 12.9%, and 17.6% separately. Semi-SP achieves the shortest kernel squad duration because it not only reduces inefficient kernel overlapping, as in NSP, but also capitalizes on the underutilized resources resulting from imprecise spatial partitioning in SP.

Fine-grained analysis. In Figure 18, the requests from two R50 applications arrive simultaneously and are scheduled by the multi-task scheduler at the same time. Request 1 is from an application that is assigned 70% GPU quotas while request 2 is from an application assigned 30% GPU. Thus, the multi-task scheduler selects more kernels from request 1 into the first several kernel squads, resulting in the earlier finishing of request 1. Notice that in the third kernel squad in Figure 18(a), the kernel squad is spatially isolated (78SMs for request 1 and 30SMs for request 2) for a shorter duration, according to the configuration determiner's identification of the optimal kernel squad configuration.

Bless can effectively compensate for prior sharing systems by eliminating bubbles within. As shown in Figure 18(b), Zico coordinates the iterations of training epochs to reduce



(a) Two overlapped R50 requests arrive simultaneously. Green: request1 kernels; Purple: request2 kernels.



(b) Bless reduces the training iteration latency on top of Zico.

Figure 18. The fine-grained analysis of Bless.

memory footprint. However, the unbounded sharing characteristic induces bubbles that are not utilized. By organizing the kernels within a round as a squad, Bless optimizes the squad through the SP policy. The iteration latency is reduced by 8.5% compared with Zico.

6.7 Impacts of hyper-parameters in Bless

In this subsection, we discuss the key parameters that influence the performance of Bless.

Kernel squad granularity. The maximum kernel count per squad impacts the flexibility of Bless. The duration of kernel squads in our evaluation ranges from 0.7ms to 10ms, depending on the operator duration of diverse applications. There is a tradeoff between the application latency and the system's ability to support various GPU quota assignments.

When the kernel count per squad is large, the overhead of the kernel squad switch is trivial and the overall latency is reduced. As shown in Figure 19(a), with the increasing of the kernel count per squad, the average latency decreases from 24.2ms to 20.6ms with an even GPU quota assignment.

However, a large kernel squad sacrifices scheduling flexibility, thus restricting the ability to support various GPU quota assignments. When the max kernel count per squad is 20, Bless can provide a maximum 8/9 quota of the GPU, according to the achievable latency. However, when the max kernel count per squad is 100, the maximum GPU quota that can be promised to each application is no larger than 3/4. The granularity of the kernel squad is set to 50 in our testbed.

Split ratio in kernel squad execution. The split ratio $c\%$ in the Semi-SP sharing scheme seeks a balance between the no-spatial partitioning scheme and the strict spatial partitioning scheme within kernel squads. As shown in Figure 19(b), the normalized duration of kernel squads reaches an optimal point when $c\% = 50\%$. It should be noted that semi-spatial sharing can only be optimal when spatial partitioning is estimated to be better than no-spatial partitioning by the performance estimators.

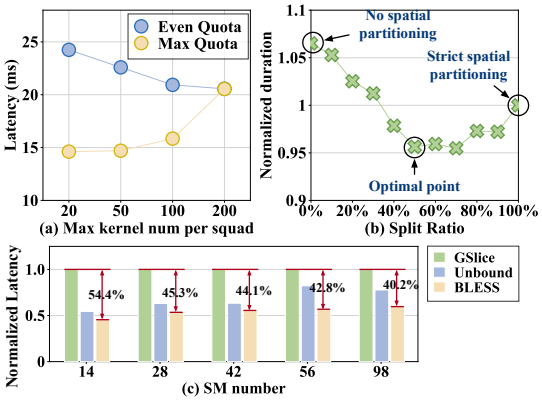


Figure 19. The impact of key parameters in Bless.

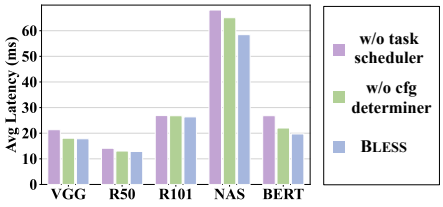


Figure 20. Ablation study of Bless.

SM number of GPU. The benefits of Bless for applications are influenced by the number of SMs on a GPU. We conducted an experiment using an Nvidia A100 GPU, leveraging MIG to control the number of usable SMs. With GPU instances featuring different numbers of SMs, we run a microbenchmark where two symmetric R50 applications share the GPU under low load (Workload(C)). As shown in Figure 19(c), as the number of SMs increases, the normalized latency reduction of Bless decreases from 54.4% to 40.2% compared to GSlice. This phenomenon results from the combined effects of the GPU's compute capability and the application's utilization: with a larger number of SMs, the application is less likely to saturate all SMs, resulting in a less significant latency reduction compared to MPS-based sharing methods. Compared to GSlice, unbounded sharing does not exhibit a stable trend in latency reduction due to unpredictable interference. From the perspective of applications, those with higher original utilization benefit more from Bless because they can better exploit GPU bubbles.

6.8 Ablation study

In this experiment, we maintain the ability of Bless to allow applications to utilize the entire GPU when requests are not overlapped, and analyze the impacts of the multi-task scheduler and the execution configuration determiner on the average latency of applications in Bless.

We apply the 5 symmetric pair-wise services with workload(B) and even 50% quotas for the comparison in the ablation study as shown in Figure 20. Compared with the complete Bless, the Bless without the multi-task scheduler can not dynamically control the kernel number from each active request, thereby extending the average latency of services by 16.5%. The Bless without execution configuration determiner prolongs the average latency by another 7.6% because it lacks thorough searching to find the optimal execution configuration for kernel squads.

6.9 Scheduling overhead

The overhead of Bless primarily comes from three runtime operations that may hinder optimal GPU utilization.

1. The kernel squad switch: When a kernel squad finishes its job, the concurrent kernel manager synchronizes with the multi-task scheduler to start the execution of the next kernel squad. The synchronization overhead is 20us. Within the kernel launching time of the first kernel in each kernel squad, the GPU is also idle. The average kernel launching duration is merely 3us.

2. The GPU context switch: When the concurrent kernel manager decides to switch the control of kernel launching from one GPU context to another through MPS, there will exist a vacuum period of about 50us. This idle period would not hinder the execution of kernels in other device queues.

3. The overspending of the multi-task scheduler: When the multi-task scheduler on the host side runs longer than the kernels on the device side, the GPU waits for the scheduler. We test the average scheduling time per kernel: the multi-task scheduling (3.7us), the execution configuration space searching (2us), and the kernel squad generation (1us). Thus, as long as the average kernel execution duration is larger than the scheduling duration (6.7us), the overspending problem would be eliminated.

Additionally, more GPU memory would be allocated for each application to accommodate extra MPS contexts. An MPS context typically consumes about 230MB GPU memory.

6.10 Discussion

Scheduling granularity. Bless schedules applications at the kernel granularity on the host side. To reduce kernel launching overhead and minimize CPU-GPU context switching, techniques such as CUDA graphs [7] and HIP graphs [11] allow for launching a sequence of kernels to the GPU with a single API call. To support applications implemented with these techniques, Bless can be adapted by either switching the scheduling granularity from kernels to graphs for applications comprising multiple graphs, or by integrating the kernel scheduler into the GPU driver [33].

Dynamic applications. For Bless to effectively allocate GPU resources among applications, it requires knowledge of each application's deterministic runtime progress. Therefore, Bless is best suited for stationary applications. For dynamic applications, where the computation graph changes at runtime, Bless must treat each separate compute DAG as an individual application and profile them during the deployment

stage. For example, in the inference of Large Language Models [39, 60], which exhibit an autoregressive computation pattern, Bless could be enhanced by treating each forward pass as a distinct application DAG for scheduling. An interesting future work would be designing more efficient profiling and modeling for dynamic GPU applications [25, 31, 72].

Resource management within kernel squads. Bless only controls the computing resources (SMs) of kernels within the squads to narrow the execution configuration space for scheduling. In addition to SMs, there are also other resources that can be managed at runtime, such as shared memory and register usage for kernels. Careful management of these resources may further reduce interference and improve the performance of multi-tenant applications. Achieving this may necessitate intrusive analysis of the applications' kernels at the deployment stage [56, 58].

7 Conclusion

In this paper, we propose Bless, a multi-user GPU system to deploy multiple applications on a single GPU. Bless utilizes the unused "bubbles" of a GPU to provide promised latency for applications with specific quota provisioned. Bless schedules kernel in units of fine-grained kernel squads and enables precise resource management. We evaluate Bless with numerous applications and workloads. As a GPU sharing system, Bless reduces the average latency by 21.1% – 37.3% for co-located applications compared to state-of-the-art systems under various GPU quota assignments.

Acknowledgement

We thank our anonymous reviewers and our shepherd, Alexander M. Merritt, for their valuable feedback. This work is partially sponsored by the National Key Research and Development Program of China (2023YFB3001504) and National Natural Science Foundation of China (62232011, 62302302, 61832006). Quan Chen is the corresponding author.

References

[1] 2012. Nvidia CUDA Stream Management. https://docs.nvidia.com/cuda/cuda-runtime-api/group__CUDART__STREAM.html.
[2] 2012. Nvidia Multi-Process Service. https://docs.nvidia.com/deploy/mps/index.html.
[3] 2016. AMD ROCm Stream Management. https://rocmdocs.amd.com/projects/HIP/en/develop/.doxygen/docBin/html/group___stream.html.
[4] 2018. Nvidia Nsight Systems. https://developer.nvidia.com/nsight-systems.
[5] 2018. Twitter request trace. https://archive.org/details/archiveteam-twitter-stream-2018-04.
[6] 2019. Apache TVM. https://tvm.apache.org/.
[7] 2019. Getting Started with CUDA Graphs. https://developer.nvidia.com/blog/cuda-graphs/.
[8] 2019. Nvidia Nsight Compute. https://developer.nvidia.com/nsight-compute.
[9] 2020. Nvidia Multi-Instance GPU. https://www.nvidia.com/en-us/technologies/multi-instance-gpu/.
[10] 2021. Alibaba Cloud cGPU. https://www.alibabacloud.com/help/en/elastic-gpu-service/latest/what-is-the-cgpu-service.
[11] 2021. Graph management - HIP runtime. https://rocm.docs.amd.com/projects/HIP/en/latest/doxygen/html/group___graph.html.
[12] 2022. Amazon Sagemaker. https://aws.amazon.com/sagemaker/.
[13] 2022. ChatGPT. https://openai.com/blog/chatgpt.
[14] 2022. Run multiple deep learning models on GPU with Amazon SageMaker multi-model endpoints. https://aws.amazon.com/blogs/machine-learning/run-multiple-deep-learning-models-on-gpu-with-amazon-sagemaker-multi-model-endpoints/.
[15] 2022. Stable Diffusion. https://stability.ai/stable-diffusion.
[16] 2024. Delivering video content with fractional GPUs in containers on Amazon EKS. https://aws.amazon.com/blogs/containers/delivering-video-content-with-fractional-gpus-in-containers-on-amazon-eks/.
[17] 2024. Meta Data Centers. https://datacenters.atmeta.com/.
[18] Paula Aguilera, Katherine Morrow, and Nam Sung Kim. 2014. Fair share: Allocation of GPU resources for both performance and fairness. In *2014 IEEE 32nd International Conference on Computer Design (ICCD 14)*. IEEE, 440–447.
[19] Paula Aguilera, Katherine Morrow, and Nam Sung Kim. 2014. QoS-aware dynamic resource allocation for spatial-multitasking GPUs. In *2014 19th Asia and South Pacific Design Automation Conference (ASP-DAC 14)*. IEEE, 726–731.
[20] Ahsan Ali, Riccardo Pinciroli, Feng Yan, and Evgenia Smirni. 2020. Batch: machine learning inference serving on serverless platforms with adaptive batching. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC 20)*. IEEE, 1–15.
[21] Guoyang Chen, Yue Zhao, Xipeng Shen, and Huiyang Zhou. 2017. Effisha: A software framework for enabling efficient preemptive scheduling of gpu. In *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP 17)*. 3–16.
[22] Quan Chen, Hailong Yang, Jason Mars, and Lingjia Tang. 2016. Baymax: Qos awareness and increased utilization for non-preemptive accelerators in warehouse scale computers. *ACM SIGPLAN Notices* 51, 4 (2016), 681–696.
[23] 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 (ATC 22)*. 199–216.
[24] 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 23)*. IEEE, 624–637.
[25] Weihao Cui, Zhenhua Han, Lingji Ouyang, Yichuan Wang, Ningxin Zheng, Lingxiao Ma, Yuqing Yang, Fan Yang, Jilong Xue, Lili Qiu, et al. 2023. Optimizing dynamic neural networks with brainstorm. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*. 797–815.
[26] Weihao Cui, Han Zhao, Quan Chen, Ningxin Zheng, Jingwen Leng, Jieru Zhao, Zhuo Song, Tao Ma, Yong Yang, Chao Li, et al. 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 (SC 21)*. 1–15.
[27] Lorenzo Dematté and Davide Prandi. 2010. GPU computing for systems biology. *Briefings in bioinformatics* 11, 3 (2010), 323–333.
[28] Aditya Dhakal, Sameer G Kulkarni, and KK Ramakrishnan. 2020. Gslice: controlled spatial sharing of gpus for a scalable inference platform. In *Proceedings of the 11th ACM Symposium on Cloud Computing (SoCC 20)*. 492–506.
[29] Jiarui Fang, Yang Yu, Chengduo Zhao, and Jie Zhou. 2021. Turbotransformers: an efficient gpu serving system for transformer models. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP 21)*. 389–402.

[30] Jianfeng Gu, Yichao Zhu, Puxuan Wang, Mohak Chadha, and Michael Gerndt. 2023. FaST-GShare: Enabling efficient spatio-temporal GPU sharing in serverless computing for deep learning inference. In *Proceedings of the 52nd International Conference on Parallel Processing (ICPP 23)*. 635–644.

[31] Yue Guan, Yuxian Qiu, Jingwen Leng, Fan Yang, Shuo Yu, Yunxin Liu, Yu Feng, Yuhao Zhu, Lidong Zhou, Yun Liang, et al. 2024. Amanda: Unified Instrumentation Framework for Deep Neural Networks. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 24)*. 1–18.

[32] 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)*. 443–462.

[33] 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)*. 539–558.

[34] Johann Hauswald, Yiping Kang, Michael A Laurenzano, Quan Chen, Cheng Li, Trevor Mudge, Ronald G Dreslinski, Jason Mars, and Lingjia Tang. 2015. Djinn and tonic: Dnn as a service and its implications for future warehouse scale computers. *ACM SIGARCH Computer Architecture News* 43, 3S (2015), 27–40.

[35] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition (CVPR 16)*. 770–778.

[36] Zicong Hong, Jian Lin, Song Guo, Sifu Luo, Wuhui Chen, Roger Wattenhofer, and Yue Yu. 2024. Optimus: Warming Serverless ML Inference via Inter-Function Model Transformation. In *Proceedings of the Nineteenth European Conference on Computer Systems (EuroSys 24)*. 1039–1053.

[37] 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 19)*. IEEE, 29–41.

[38] Onur Kayıran, Adwait Jog, Mahmut T Kandemir, and Chita R Das. 2013. Neither more nor less: Optimizing thread-level parallelism for GPGPUs. In *Proceedings of the 22nd international conference on Parallel architectures and compilation techniques (PACT 13)*. IEEE, 157–166.

[39] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th Symposium on Operating Systems Principles (SOSP 23)*. 611–626.

[40] Zhuohan Li, Lianmin Zheng, Yinmin Zhong, Vincent Liu, Ying Sheng, Xin Jin, Yanping Huang, Zhifeng Chen, Hao Zhang, Joseph E Gonzalez, et al. 2023. {AlpaServe}: Statistical multiplexing with model parallelism for deep learning serving. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*. 663–679.

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

[42] Yu-Shiang Lin, Chun-Yuan Lin, Che-Rung Lee, and Yeh-Ching Chung. 2019. qcuda: Gpgpu virtualization for high bandwidth efficiency. In *2019 IEEE International Conference on Cloud Computing Technology and Science (CloudCom 19)*. IEEE, 95–102.

[43] Zihan Liu, Jingwen Leng, Zhihui Zhang, Quan Chen, Chao Li, and Minyi Guo. 2022. VELTAIR: towards high-performance multi-tenant deep learning services via adaptive compilation and scheduling. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 22)*. 388–401.

[44] Lixian Ma, Haoruo Chen, En Shao, Leping Wang, Quan Chen, and Guangming Tan. 2024. POSTER: FineCo: Fine-grained Heterogeneous Resource Management for Concurrent DNN Inferences. In *Proceedings of the 29th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming (PPoPP 24)*. 451–453.

[45] Lingxiao Ma, Zhiqiang Xie, Zhi Yang, Jilong Xue, Youshan Miao, Wei Cui, Wenxiang Hu, Fan Yang, Lintao Zhang, and Lidong Zhou. 2020. Rammer: Enabling holistic deep learning compiler optimizations with {rTasks}. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 881–897.

[46] Kelvin KW 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 (SOSP 23)*. 595–610.

[47] 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 preprint arXiv:1712.06139* (2017).

[48] Nathan Otterness and James H Anderson. 2021. Exploring AMD GPU scheduling details by experimenting with "worst practices". In *Proceedings of the 29th International Conference on Real-Time Networks and Systems (RTNS 21)*. 24–34.

[49] Sreepathi Pai, Matthew J Thazhuthaveetil, and Ramaswamy Govindarajan. 2013. Improving GPGPU concurrency with elastic kernels. *ACM SIGARCH Computer Architecture News* 41, 1 (2013), 407–418.

[50] Mohit Pandey, Michael Fernandez, Francesco Gentile, Olexandr Isayev, Alexander Tropsha, Abraham C Stern, and Artem Cherkasov. 2022. The transformational role of GPU computing and deep learning in drug discovery. *Nature Machine Intelligence* 4, 3 (2022), 211–221.

[51] Manos Pavlidakis, Giorgos Vasiliadis, Stelios Mavridis, Anargyros Argyros, Antony Chazapis, and Angelos Bilas. 2024. G-Safe: Safe GPU Sharing in Multi-Tenant Environments. *arXiv preprint arXiv:2401.09290* (2024).

[52] Yajuan Peng, Shuang Chen, Yi Zhao, and Zhibin Yu. 2024. {UFO}: The Ultimate {QoS-Aware} Core Management for Virtualized and Oversubscribed Public Clouds. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*. 1511–1530.

[53] Francisco Romero, Qian Li, Neeraja J Yadwadkar, and Christos Kozyrakis. 2021. {INFaaS}: Automated model-less inference serving. In *2021 USENIX Annual Technical Conference (ATC 21)*. 397–411.

[54] 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 (SOSP 19)*. 322–337.

[55] Lin Shi, Hao Chen, Jianhua Sun, and Kenli Li. 2011. vCUDA: GPU-accelerated high-performance computing in virtual machines. *IEEE Trans. Comput.* 61, 6 (2011), 804–816.

[56] Sudipta Saha Shubha, Haiying Shen, and Anand Iyer. 2024. {USHER}: Holistic Interference Avoidance for Resource Optimized {ML} Inference. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. 947–964.

[57] Karen Simonyan and Andrew Zisserman. 2014. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556* (2014).

[58] 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 (EuroSys 24)*. 1075–1092.

[59] Yusuke Suzuki, Shinpei Kato, Hiroshi Yamada, and Kenji Kono. 2014. {GPUvm}: Why Not Virtualizing {GPUs} at the Hypervisor?. In *2014 USENIX Annual Technical Conference (ATC 14)*. 109–120.

[60] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. 2023. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288* (2023).

[61] Yash Ukidave, Charu Kalra, David Kaeli, Perhaad Mistry, and Dana Schaa. 2014. Runtime support for adaptive spatial partitioning and inter-kernel communication on gpus. In *2014 IEEE 26th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD 14)*. IEEE, 168–175.

[62] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Advances in neural information processing systems (NeurIPS 17)* 30 (2017).

[63] Guanhua Wang, Kehan Wang, Kenan Jiang, Xiangjun Li, and Ion Stoica. 2021. Wavelet: Efficient DNN training with tick-tock scheduling. *Proceedings of Machine Learning and Systems (MLSys 21)* 3 (2021), 696–710.

[64] Zhenning Wang, Jun Yang, Rami Melhem, Bruce Childers, Youtao Zhang, and Minyi Guo. 2016. Simultaneous multikernel GPU: Multi-tasking throughput processors via fine-grained sharing. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA 16)*. IEEE, 358–369.

[65] Zhenning Wang, Jun Yang, Rami Melhem, Bruce Childers, Youtao Zhang, and Minyi Guo. 2017. Quality of service support for fine-grained sharing on GPUs. In *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA 17)*. 269–281.

[66] Bo Wu, Guoyang Chen, Dong Li, Xipeng Shen, and Jeffrey Vetter. 2015. Enabling and exploiting flexible task assignment on GPU through SM-centric program transformations. In *Proceedings of the 29th ACM on International Conference on Supercomputing (ICS 15)*. 119–130.

[67] 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)*. 69–85.

[68] 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 *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 533–548.

[69] Qiumin Xu, Hyeran Jeon, Keunsoo Kim, Won Woo Ro, and Murali Annavaram. 2016. Warped-slicer: efficient intra-sm slicing through dynamic resource partitioning for gpu multiprogramming. *ACM SIGARCH Computer Architecture News* 44, 3 (2016), 230–242.

[70] Feng Yan, Yuxiong He, Olatunji Ruwase, and Evgenia Smirni. 2018. Efficient deep neural network serving: Fast and furious. *IEEE Transactions on Network and Service Management* 15, 1 (2018), 112–126.

[71] Chengliang Zhang, Minchen Yu, Wei Wang, and Feng Yan. 2019. {MArk}: Exploiting Cloud Services for {Cost-Effective},{SLO-Aware} Machine Learning Inference Serving. In *2019 USENIX Annual Technical Conference (ATC 19)*. 1049–1062.

[72] Shulai Zhang, Weihao Cui, Quan Chen, Zhengnian Zhang, Yue Guan, Jingwen Leng, Chao Li, and Minyi Guo. 2022. PAME: precision-aware multi-exit DNN serving for reducing latencies of batched inferences. In *Proceedings of the 36th ACM International Conference on Supercomputing (ICS 22)*. 1–12.

[73] Wei Zhang, Weihao Cui, Kaihua Fu, Quan Chen, Daniel Edward Mawhirter, Bo Wu, Chao Li, and Minyi Guo. 2019. Laius: Towards latency awareness and improved utilization of spatial multitasking accelerators in datacenters. In *Proceedings of the ACM international conference on supercomputing (ICS 19)*. 58–68.

[74] Yanqi Zhang, Íñigo Goiri, Gohar Irfan Chaudhry, Rodrigo Fonseca, Sameh Elnikety, Christina Delimitrou, and Ricardo Bianchini. 2021. Faster and cheaper serverless computing on harvested resources. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP 21)*. 724–739.

[75] Han Zhao, Weihao Cui, Quan Chen, Youtao Zhang, Yanchao Lu, Chao Li, Jingwen Leng, and Minyi Guo. 2022. Tacker: Tensor-cuda core kernel fusion for improving the gpu utilization while ensuring qos. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA 22)*. IEEE, 800–813.

[76] Wenyi Zhao, Quan Chen, Hao Lin, Jianfeng Zhang, Jingwen Leng, Chao Li, Wenli Zheng, Li Li, and Minyi Guo. 2019. Themis: Predicting and reining in application-level slowdown on spatial multitasking GPUs. In *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS 19)*. IEEE, 653–663.

[77] 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 (ASPLOS 20)*. 1371–1385.

[78] Xia Zhao, Zhiying Wang, and Lieven Eeckhout. 2018. Classification-driven search for effective sm partitioning in multitasking gpus. In *Proceedings of the international conference on supercomputing*. 65–75.

[79] Lianmin Zheng, Chengfan Jia, Minmin Sun, Zhao Wu, Cody Hao Yu, Ameer Haj-Ali, Yida Wang, Jun Yang, Danyang Zhuo, Koushik Sen, et al. 2020. Ansor: Generating {High-Performance} tensor programs for deep learning. In *14th USENIX symposium on operating systems design and implementation (OSDI 20)*. 863–879.

[80] Barret Zoph, Vijay Vasudevan, Jonathon Shlens, and Quoc V Le. 2018. Learning transferable architectures for scalable image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition (CVPR 18)*. 8697–8710.