Orion: Interference-aware, Fine-grained GPU Sharing for ML Applications

Foteini Strati
ETH Zurich

Xianzhe Ma
ETH Zurich

Ana Klimovic
ETH Zurich

Abstract

GPUs are critical for maximizing the throughput-per-Watt of deep neural network (DNN) applications. However, DNN applications often underutilize GPUs, even when using large batch sizes and eliminating input data processing or communication stalls. DNN workloads consist of data-dependent operators, with different compute and memory requirements. While an operator may saturate GPU compute units or memory bandwidth, it often leaves other GPU resources idle. Despite the prevalence of GPU sharing techniques, current approaches are not sufficiently fine-grained or interference-aware to maximize GPU utilization while minimizing interference at the granularity of 10s of μs. We propose Orion, a system that transparently intercepts GPU kernel launches from multiple clients sharing a GPU. Orion schedules work on the GPU at the granularity of individual operators and minimizes interference by taking into account each operator's compute and memory requirements. We integrate Orion in PyTorch and demonstrate its benefits in various DNN workload collocation use cases. Orion significantly improves tail latency compared to state-of-the-art baselines for a high-priority inference job while collocating best-effort inference jobs to increase per-GPU request throughput by up to 7.3×, or while collocating DNN training, saving up to 1.49× in training costs compared to dedicated GPU allocation.

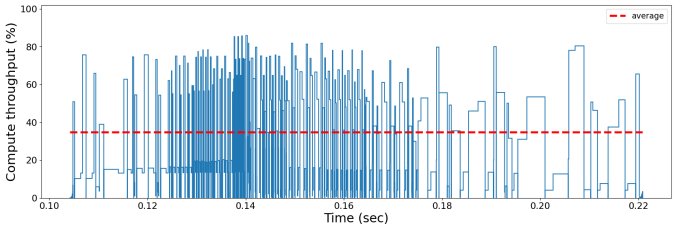
CCS Concepts: • **Computing methodologies** → **Machine learning**.

Keywords: Machine Learning, GPUs

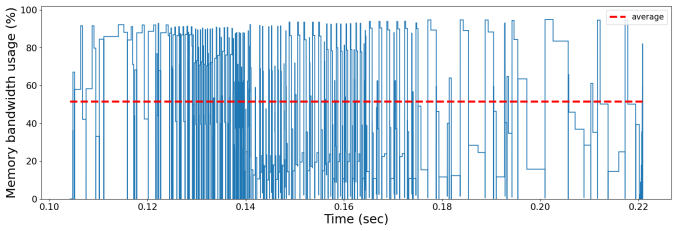
ACM Reference Format:
Foteini Strati, Xianzhe Ma, and Ana Klimovic. 2024. Orion: Interference-aware, Fine-grained GPU Sharing for ML Applications. In *Nineteenth European Conference on Computer Systems (EuroSys '24), April 22–25, 2024, Athens, Greece.* ACM, New York, NY, USA, 18 pages. https://doi.org/10.1145/3627703.3629578

EuroSys '24, April 22–25, 2024, Athens, Greece

ACM ISBN 979-8-4007-0437-6/24/04…$15.00
https://doi.org/10.1145/3627703.3629578



(a) GPU Compute Throughput Utilization



(b) GPU Memory Bandwidth Utilization

Figure 1. GPU resource utilization for a MobileNetV2 training iteration with batch size 96 [84]. Utilization is bursty and often low as individual operators saturate compute units or memory bandwidth, often leaving one resource type idle.

1 Introduction

Deep Neural Network (DNN) applications achieve orders of magnitude higher throughput-per-Watt when executing on GPUs compared to CPUs [89, 96, 99, 101]. However, due to the high price of GPU hardware, the ultimate energy and cost savings of using GPUs for DNN jobs depend on operating the accelerators at high utilization.

Despite the high compute and memory intensity of DNN computations [86], individual DNN workloads often underutilize GPU hardware [98, 99]. Latency-critical inference jobs, such as Autonomous Driving [77], fraud detection and recommendation systems [16] use small batch sizes to meet service level objectives, resulting in insufficient parallelism to keep GPU compute units busy [49, 50, 75]. Training jobs attempt to maximize batch size for high throughput, however the batch size affects model convergence and must be tuned with other hyperparameters [47, 56, 88, 95, 97]. Hence, in practice, training jobs run with "large enough" batch sizes, which may underutilize sizeable GPU memory capacity. Training jobs may also stall waiting for input data [48, 62, 72, 73, 108] or bottleneck on communication [80, 107], leaving GPUs idle.

However, even if we eliminate input data stalls (e.g., by scaling out data preprocessing [34, 48, 108]), alleviate communication stalls (e.g., with asynchronous updates [40, 44], gradient compression [68], and in-network aggregation [85]), and pragmatically maximize batch sizes, *DNN workloads still underutilize GPU hardware*. Figure 1 shows the fundamental reason why: a DNN workload consists of many data-dependent operators that run for short periods of time (10s-1000s of μs), each with different compute and memory requirements. Utilization is bursty and low on average (see red dotted lines) as individual operators saturate compute units or memory bandwidth, but often leave a resource type idle. For the MobileNetV2 training job in Figure 1, GPU compute throughput and memory bandwidth utilization are below 40% and 55%, on average, respectively. The problem of GPU underutilization is only getting worse as hardware vendors continue to scale GPU memory and compute capacity [11].

A common solution is to share GPUs between jobs. The main challenge is maximizing utilization while mitigating interference between jobs for high performance. Temporal sharing techniques time-slice the GPU at the granularity of an inference request or training minibatch [39, 49, 100, 102–104]. This can lead to head-of-line blocking, since incoming inference requests or training minibatches need to wait for the ongoing tasks to finish execution on the GPU before being scheduled (see section 6), and still wastes resources when the operators of individual tasks do not consume all GPU compute or memory bandwidth. Spatial sharing improves utilization, however, current techniques are either too coarse-grained (e.g., Multi-Instance GPUs [12], Zico [67], Tick-Tock [94]) or are not sufficiently interference-aware (e.g., Multi-Process Service (MPS) [25], GPU Streams [15], REEF [50], Paella [75]).

Figure 2 shows that state-of-the-art GPU sharing techniques leave performance on the table. We collocate three pairs of DNN jobs whose aggregate resource requirements fit on a single V100-16GB GPU (see Table 1). The first job in each pair is high-priority while the second is best-effort. Each job issues one request at a time in a closed loop. The stacked bar plot shows each job's throughput on a shared GPU with various techniques, compared to the total throughput when jobs execute on dedicated GPUs (denoted as Ideal). Temporal sharing, MPS, Streams, and Tick-Tock (proposed for training job collocation) achieve far below ideal aggregate throughput. REEF achieves high performance for the high-priority job, but barely executes the best-effort job.

To help close this gap, we propose Orion[1], a fine-grained, interference-aware GPU scheduler. Orion maintains performance for a high-priority workload while collocating best-effort jobs to maximize GPU utilization and save costs. Orion is a software system that intercepts GPU kernel launches from client applications sharing a GPU device. Orion schedules requests based on client job priority, operator size, and whether an operator is compute or memory bound. By scheduling at the granularity of individual operations, Orion spatially shares the GPU to make best use of GPU compute units and memory bandwidth that may be underutilized by the high-priority job for only 10-1000s of μs.

[1]Orion is available at https://github.com/eth-easl/orion

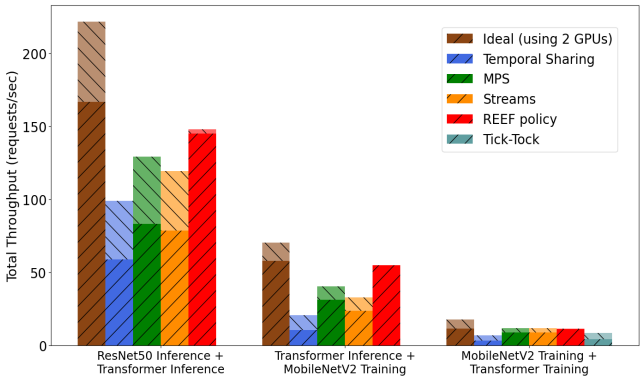


Figure 2. Existing GPU collocation techniques leave performance on the table. Bold bars show high-priority DNN job throughput, faint bars show best-effort job throughput.

Orion improves GPU resource efficiency (and cost) for a variety of DNN collocation use cases, with minimal impact on high-priority job performance. When collocating latency-sensitive inference with best-effort offline inference, Orion improves aggregate throughput by up to 7.3× compared to dedicated GPU execution, while maintaining p99 latency within 15% on average for the high-priority job. When collocating a latency-sensitive inference job with a training job, Orion maintains p99 inference latency within 14% on average while increasing the GPU's aggregate throughput up to 2.3×. Orion reduces cost by 1.29× when collocating training jobs while ensuring the high-priority training job maintains throughput within 16% of its dedicated GPU throughput.

2 GPU Architecture Background

Figure 3 shows a typical GPU architecture. Without loss of generality, we use NVIDIA hardware and CUDA programming terminology [2, 13]. A GPU consists of multiple Streaming Multiprocessors (SMs), each containing various types of compute cores (e.g., fp64 units, tensor cores), register files, and L1 cache. The GPU also has shared caches and memory.

GPU programming abstractions. Developers define their DNN application as a collection of operations using high-level APIs in a framework like PyTorch [79] or TensorFlow [29]. The application framework compiles these operations (e.g., convolution, batch normalization) for the target GPU architecture and submits operations as CUDA computation *kernels* to the GPU, along with CUDA memory management operations that allocate, initialize, and free GPU memory.

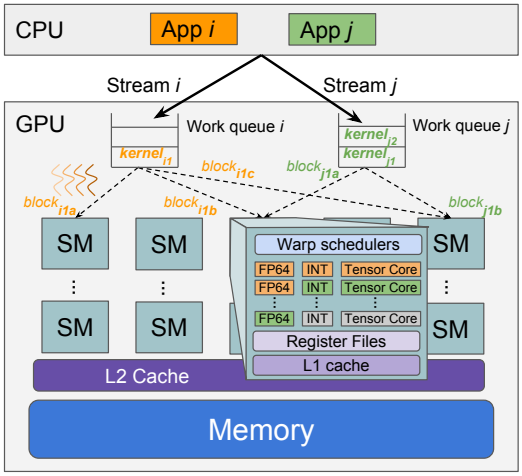


Figure 3. Simplified GPU architecture.

Submitting a kernel involves specifying its resource requirements (e.g., the number of thread blocks[2], registers, threads per block, and shared memory required). The application associates each kernel launch and memory operation to a particular CUDA *stream*. A stream is

a sequence of operations that are guaranteed to execute in order. Each application process has its own default stream. To increase concurrency, applications can create additional streams (optionally with different priorities [15]) and submit kernels across streams.

GPU hardware scheduling. As shown in Figure 3, the GPU buffers each CUDA stream's kernels in a separate work queue on the device. Most GPUs, in particular NVIDIA GPUs, do not allow users to preempt kernels after submission [50]. The GPU hardware scheduler dispatches thread blocks from kernels in each work queue based on stream priority. The scheduler assigns a thread block to an SM when the thread block's data dependencies are met and an SM with sufficient resources is available. Users cannot control which SM will execute a particular thread block, though researchers have reverse-engineered hardware scheduling policies for popular GPU architectures [30, 45, 46]. When a thread block is assigned to an SM, the SM will schedule and execute all thread warps from that block. SMs can execute multiple warps concurrently, from thread blocks that may belong to different kernels and streams [2]. However, if any warp saturates a resource on the SM (e.g. the number of registers), the SM's warp scheduler will wait until no resource is saturated before scheduling any additional warps, even if some resources on the SM are available (e.g. compute units or shared memory).

GPU utilization metrics. The most common GPU utilization metric is *SM utilization*, which is the percentage of SMs that are busy (i.e., executing at least one warp). SM utilization does not fully capture GPU utilization as an SM is considered busy even if only a small part of its resources are in use. *Compute throughput utilization* is the utilization of SM compute units, such as FP32, FP64, FP16, FMA units, tensor cores, etc. [9]. Using the NVIDIA Nsight Compute tool [26], we can get information about the utilization of each individual component [3]. The reported utilization is the maximum of all distinct components utilizations. *Memory capacity utilization* is the percentage of memory allocated on the GPU. *Memory bandwidth utilization* is the percentage of peak GPU internal memory bandwidth consumed.

3 Understanding DNN GPU Utilization

Although DNN applications typically have high compute and memory intensity [86], they often underutilize GPUs [98, 99, 102, 103]. Prior work has identified reasons for low GPU utilization and proposed solutions. Input data preprocessing bottlenecks on host CPUs can leave GPUs idle while waiting to ingest data [58, 62, 72, 73]. We can alleviate input stalls by disaggregating and scaling out data preprocessing [34, 48, 91, 108]. Communication between nodes can limit distributed training throughput and idle GPUs [80, 107]. Aggressive pipelining [53, 74], gradient compression [68], asynchronous updates [40, 44], in-network aggregation [85] help hide communication stalls. Gang scheduling in multi-GPU clusters can leave some GPUs idle while other GPUs for a job become available [103]. Recent DNN systems address this issue with elastic GPU allocation [33, 66, 76, 82].

However, even after eliminating input data, communication, and gang-scheduling bottlenecks, *DNN jobs still struggle to keep GPUs fully utilized*, especially when modest batch sizes are used. Real-time inference jobs, such as computer-vision tasks in self-driving cars [42, 77], speech recognition services [42] and online recommendation systems [16] usually employ small batch sizes, in order to avoid SLO violations [41, 49, 50, 75].

Throughput-oriented training jobs use large batch sizes, but maximizing batch sizes to reach GPU memory limits is not always beneficial [71, 78, 95]. Increasing the batch size beyond a certain point can degrade the *statistical efficiency* of training [56, 57, 61, 70, 82, 88], and have diminishing returns in the training procedure, increasing the time needed to reach a target accuracy [78], and decreasing the model's validation performance [82]. Shallue et al. [88] studied the effects of increasing the batch size in a variety of models and tasks. They observed that, beyond a certain point, further increase in the batch size does not lead to reductions in training time. Researchers have proposed adapting the learning rate as the batch sizes increase [47]. Nevertheless, these

[2] A thread *block* is a group of threads, which will execute on the same SM and can communicate via shared memory [4]. A thread block typically consists of multiple *warps*, which are groups of threads that execute the same instruction. There are typically 32 threads per warp.

[3] According to the NVIDIA Nsight Compute tool, X% utilization of a compute unit (e.g. FP16 unit), means that the unit was active for X% of the time (similar to CPU utilization).

Model	Workload	Batch size	Avg SMs busy (%)	Compute Throughput(%)	Memory Bandwidth(%)	Memory Capacity(%)
ResNet50	Inference	4	24	30	22	9
MobileNetV2	Inference	4	6	18	21	7
ResNet101	Inference	4	29	24	37	9
BERT-large	Inference	2	95	72	28	14
Transformer	Inference	4	61	52	29	10
ResNet50	Training	32	81	48	45	32
MobileNetV2	Training	64	71	34	49	43
ResNet101	Training	32	85	50	43	39
BERT-basic	Training	8	61	44	21	38
Transformer	Training	8	49.5	29	30	53

Table 1. Average GPU utilization for popular DNN workloads on a V100-16GB NVIDIA GPU. SM utilization is from the `Nsight Systems` tool. We measure the compute throughput and memory bandwidth utilization of each kernel using `sm_throughput` and `gpu_compute_memory_throughput` in `Nsight Compute` [8]. We monitor memory capacity with the `nvidia-smi` command.

techniques are model-specific and require significant tuning and expertise [32, 57, 95]. DeepPool [78] demonstrates convergence issues in distributed setups with large global batch sizes, when per-GPU batch size remains constant while increasing the number of GPUs. Hence, they recommend *strong scaling*: when scaling out training to more GPUs for large models, the optimal per-GPU batch size decreases. Similarly, Crossbow [57] exhibits the best time-to-accuracy with smaller batch sizes. These trends leave memory capacity and GPU resources underutilized.

Recently, Large Language Models (LLMs) have become very prevalent, due to their high performance in a diverse spectrum of tasks. Since LLMs have exceptionally large memory capacity requirements (even with small batch sizes [60]), the opportunities to share GPUs among LLM workloads are more limited. Hence, LLMs are not our target workloads for GPU sharing. Nevertheless, in Section 7, we discuss GPU sharing opportunities for LLMs as the sequential token generation phase of LLM inference is memory-bound and underutilizes the GPU's compute throughput and SMs [55, 60].

3.1 Profiling the GPU utilization of DNN jobs

We profile a variety of popular DNN workloads executing on an NVIDIA V100-16GB GPU without stalls. We use batch sizes for each workload based on configurations commonly used in prior work for the same or similar GPU hardware [21, 23, 51, 54, 84, 97]. We use the `Nsight Compute` tool to profile the compute and memory utilization of individual kernels, and get a kernel execution trace using the `Nsight Systems` tool. By aligning each kernel's start and end points with resource profile information, we generate resource utilization traces for the entire workload (e.g. Figure 1). We then compute the average utilization across the whole workload. Across all workloads, we observe that GPU compute throughput and memory bandwidth utilization are bursty, as shown in the example of Figure 1, and low on average,

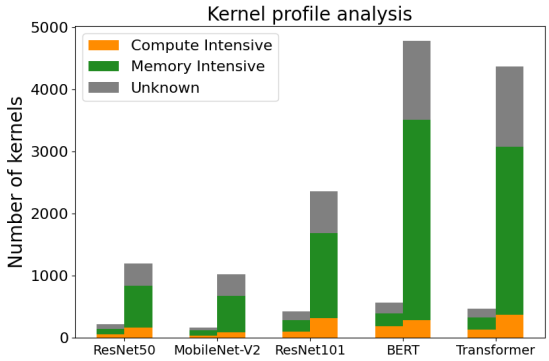


Figure 4. Compute vs. memory intensive kernels for model inference request (left) and model training minibatch (right).

as summarized in Table 1. Compute throughput utilization ranges from 18% to 72%, even though up to 95% of SMs can be "busy" on average (an SM is busy if it executes at least one warp). Furthermore, workloads consume only 21-49% of GPU memory bandwidth and 7-53% of GPU memory capacity.

When analyzing DNN kernel execution traces, we observe that GPU compute utilization spikes often occur at different points in time than memory utilization spikes. DNN workloads consist of many kernels with different resource requirements. Figure 4 classifies each workload's kernels as *compute-intensive* (performance is bounded by GPU compute throughput) or *memory-intensive* (performance is bounded by GPU memory bandwidth).[4] Kernels typically execute for 10s to 100s of μs (for inference) or 100s to 1000s of μs (for training). As kernels from an individual DNN job execute sequentially due to data dependencies, when a kernel saturates GPU compute or memory bandwidth, it often leaves other GPU resources idle for short periods of time.

[4]Some kernels are labeled unknown because the `NSight Compute` tool does not provide a Roofline analysis for all kernels.

Kernel pairs	Sequential	Collocated	Speedup
Conv2d-Conv2d	2.59 ms	2.63 ms	0.98×
BN2d-BN2d	1.78 ms	1.65 ms	1.08×
Conv2d-BN2d	2.15 ms	1.52 ms	**1.41**×

Table 2. Toy experiment collocating Conv2d (compute-intensive) with BN2d (memory-intensive) kernels. Collocation leads to significant speedup over sequential execution when kernels have opposite resource intensity.

3.2 Exploring GPU kernel collocation

A promising way to improve GPU utilization is to collocate kernels with opposite resource intensity. While overlapping kernel execution within a DNN job is limited due to data dependencies, we can collocate kernels from different jobs. We conduct a toy experiment with a compute-intensive kernel *Conv2d* (a 2D convolution) and a memory-intensive kernel *BN2d* (a 2D batch normalization), commonly used in vision models. With a batch size of 32, Conv2d executes in 1.35 ms, consuming 100% of SMs on a dedicated V100 GPU. BN2d executes in 0.93 ms, consuming 40% of SMs. Table 2 shows the execution time when executing the kernels sequentially (on a single CUDA stream) and concurrently (on separate CUDA streams). Collocating two Conv2d kernels is not beneficial, as the two kernels compete for GPU SMs, ultimately running sequentially. Collocating two BN2d kernels leads to a small speedup compared to sequential execution. Despite each kernel only consuming 40% of the GPU's SMs, their memory-intensive nature causes significant interference. In contrast, collocating a Conv2d with a BN2d kernel reduces aggregate latency by 1.41× compared to sequential execution, since they have different resource demands. Conv2d consumes 89% and 20% of GPU compute throughput and memory bandwidth, respectively, while BN2d has 14% compute throughput and 80% memory bandwidth utilization.

Takeaway: Individual DNN jobs consist of kernels with various compute and memory requirements, as shown in Figure 4. Since kernels need to execute sequentially, due to data dependencies, they often underutilize GPU's compute and memory bandwidth. Sharing GPUs between DNN jobs is necessary to maximize utilization. Our toy experiment, described in Table 2, has shown that spatial collocation is most effective for kernels with opposite compute vs. memory intensity. Since DNN jobs consist of both compute- and memory-intensive kernels (Figure 4), colocating opposite-profile kernels from different DNN jobs would help increase utilization, while minimizing interference.

4 Related Work on GPU Sharing

We summarize current approaches for GPU sharing and discuss why they are not sufficiently fine-grained or interference-aware to make use of GPU resources that a high-priority DNN job may underutilize for ∼100s of μs at a time.

Temporal sharing. Temporal sharing techniques time-slice the GPU by context switching between multiple jobs to improve utilization. Prior systems focus on multiplexing multiple DNN models per GPU whose collective state does not fit in GPU memory. Hence, the main challenge these systems address is efficiently swapping state as requests for particular models arrive. Gandiva [102] uses a suspend-and-restart mechanism to transfer state between host and GPU memory during context switches. Salus [104] reduces context switching by optimizing which state should remain on the GPU. Clockwork [49] serves thousands of DNNs per GPU with predictable latency by determining upfront whether the GPU can meet the request deadline based on the expected time to load/unload DNN state and run inference. Antman [103] dynamically adjusts job memory allocations to enable more efficient cluster-level job collocation per GPU for temporal sharing. Transparent GPU Sharing (TGS) [100] enables application-agnostic temporal GPU sharing for containerized workloads. However, these systems still execute one job at a time. As discussed in §3, this underutilizes GPUs as an individual DNN job's kernels often do not consume all GPU compute units and memory. The goal of our fine-grained, interference-aware sharing is to fill spare GPU capacity for such jobs. Our work complements the above approaches, which efficiently swap state to fit more models per GPU.

Spatial sharing. Spatial sharing mechanisms enable jobs to simultaneously use different regions of a GPU [106]. NVIDIA Multi-Instance GPU (MIG) [12] offers coarse-grained GPU partitioning, but lacks the agility to opportunistically harvest resources that are underutilized for short time slots. MIG partitions take 100s of ms to create and models take 10s of seconds to resume execution from checkpoints after a new partition is created [65]. NVIDIA Multi-Process Service (MPS) [25] enables multiple processes to run in parallel on a GPU, but leads to high interference, as processes freely share caches, compute, and memory resources (see Figure 2). REEF [50] schedules kernels at a fine granularity based on their size and priority, and is designed to collocate high and low priority inference jobs. Zico [67] and Tick-Tock [94] collocate training jobs on GPUs by scheduling forward and backward passes to minimize total memory consumption.

However, none of these approaches co-schedule kernels based on their compute and memory profiles, which we showed in §3 is critical to minimize interference while maximizing GPU utilization.

5 Orion

We propose Orion, a fine-grained, interference-aware GPU scheduler. Orion's goal is to maintain high performance for a high-priority job while using spare GPU resources for best-effort jobs. Orion is transparent to end users and requires no API changes. We implement Orion as a dynamically linked

library that controls GPU operations submitted by an application framework (e.g., PyTorch). As shown in Figure 5, Orion intercepts GPU operations submitted by each client and buffers the operations in per-client software queues. Operations include GPU kernels (e.g., convolution, batch normalization) and memory management operations (e.g., memory allocations, memory copies). Orion submits operations from per-client software queues to the GPU hardware using the scheduling policy described in §5.1, leveraging kernel characteristics collected during an offline workload profiling phase, described in §5.2. Orion operates on the level of a single GPU device. In distributed DNN job deployments, a separate instance of Orion runs per GPU device.

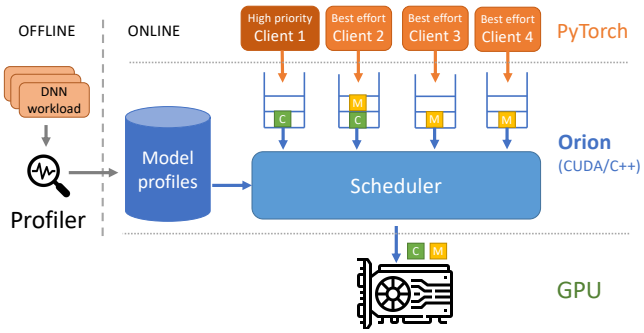


Figure 5. Orion system architecture.

5.1 Orion Scheduler

We describe Orion's GPU kernel scheduling policy (§5.1.1) along with the mechanisms we use to implement the scheduling policy (§5.1.2) and manage GPU memory (§5.1.3). Orion's policy is developed and evaluated with closed-source GPUs in mind, which do not allow users to control the physical placement of the kernels in the SMs, or preempt kernels after submission.

5.1.1 GPU Kernel Scheduling Policy. The pseudo code in Listing 1 shows Orion's scheduling policy, assuming for simplicity that two clients share the GPU: a high priority client (`client_hp`) and a best-effort client (`client_be`). Orion generalizes to an arbitrary number of best-effort clients by serving clients round-robin.

Orion executes the `run_scheduler` method to continuously poll each client's software queue (lines 4-6). If a kernel from the high-priority job is present (line 7), Orion submits it directly to the GPU hardware on a dedicated GPU stream (line 8). The GPU hardware executes kernels submitted to the same stream sequentially, thus respecting data dependencies.

If a kernel from a best-effort job is present (line 10), Orion executes `schedule_be()` to decide if it is currently suitable to launch the kernel on the GPU. In order to reduce interference between the high-priority and best-effort tasks, Orion takes into account the compute and memory profiles of the high-priority and best-effort kernels, as well as the Streaming Multiprocessor demands and duration of the best-effort kernels. A best-effort kernel is suitable to schedule if there is no high-priority task ongoing. The best-effort kernel is also suitable to schedule if it is sufficiently small (in terms of the number of SMs it requires) *and* if the kernel has an opposite resource profile (compute vs. memory bound) compared to the high-priority kernel (lines 27-29). Orion considers the number of SMs (in addition to the compute/memory resource profiles) since large best-effort kernels might occupy all the SMs of the GPU and starve high-priority kernels. By default, we set `SM_THRESHOLD` to the total number of SMs on the GPU device, however, this parameter can also be tuned dynamically to increase utilization while monitoring high-priority job performance. For example, when the high-priority job is a throughput-oriented job, such as training, we find that `SM_THRESHOLD` can be increased for more aggressive collocation. Tuning is done by monitoring the throughput of the high-priority job and adjusting the `SM_THRESHOLD` with binary search, using the maximum number of SMs needed by any kernel of the best-effort job as the max value and zero as the min value in the search. Kernels with unknown resource profiles tend to be very short-running, hence Orion optimistically allows these kernels to execute with compute or memory-bound kernels.

To minimize interference despite the asynchronous nature of GPU kernel execution and the lack of kernel preemption software available to the users, Orion checks one additional condition before submitting a best-effort kernel to the GPU hardware (line 12). Orion keeps track of best-effort kernels that are still outstanding on the GPU (i.e., submitted but not yet completed) and their expected duration, obtained from an initial profiling phase (described in §5.2). Orion will only launch the best-effort kernel if the expected total duration of the outstanding best-effort kernels is not already close to the high-priority job request latency, if the high-priority job is an inference workload, or iteration duration, if the high-priority job is a training workload. Throttling long-running sequences of best-effort kernels is necessary since as soon as they start executing on the GPU, they cannot be preempted even if high-priority kernels get submitted. `DUR_THRESHOLD` is a tunable percentage of the high-priority job request latency. We explore Orion's performance sensitivity to `DUR_THRESHOLD` (see §6.4) and empirically set the default percentage to 2.5%.

Finally, in line 19, Orion submits the best-effort kernel on a separate GPU stream and keeps track of the outstanding best-effort kernel durations. In the case of multiple best-effort clients, Orion uses a separate GPU stream for each client.

5.1.2 Scheduling Implementation Mechanisms. Orion uses two key mechanisms to implement the policy.

GPU stream priorities. Closed-source GPUs, such as NVIDIA GPUs, do not expose the hardware scheduler to

```
1  def run_scheduler(client_q_hp, client_q_be):
2      be_duration = 0, be_submitted = Event()
3      hp_task_running = False
4      while True:
5          op_hp = client_q_hp.pop()
6          op_be = client_q_be.peek()
7          if (op_hp != None):
8              launch_kernel(op_hp, stream_hp)
9              hp_task_running = True
10         if (op_be != None):
11             schedule = schedule_be(op_hp, op_be)
12             if (be_duration > DUR_THRESHOLD):
13                 if (be_submitted.finished()):
14                     be_duration = 0
15                 else:
16                     schedule = False
17             if (schedule):
18                 client_q_be.pop()
19                 launch_kernel(op_be, stream_be)
20                 be_duration += op_be.duration
21                 be_submitted.record(stream_be)
22
23 def schedule_be(op_hp, op_be):
24     schedule = False
25     if (!hp_task_running):
26         schedule = True
27     else if (op_be.sm_needed < SM_THRESHOLD
28         and have_different_profiles(op_hp,op_lp)):
29         schedule = True
30     return schedule
```

Listing 1. Orion scheduling algorithm assuming one high priority and one best-effort client.

software. However, we can influence the behavior of the hardware scheduler by using *streams priorities* [15]. The GPU hardware prioritizes scheduling an incoming kernel from a high-priority stream, even when low-priority kernels are pending (i.e. not executing yet). However, there is no guarantee that thread blocks from a high-priority stream will preempt the execution of a kernel running on a stream with a lower priority [7]. The lack of preemption is why we throttle best-effort kernels with the `DUR_THRESHOLD` check.

CUDA events. CUDA Events [18] provide a way to monitor the progress of each stream in the GPU without expensive stream synchronization operations, which would block the CPU scheduler thread. Upon submitting a best-effort kernel, Orion *records* its submission in a CUDA event (line 21 in Listing 1). The GPU sets the event status as "finished" after the submitted kernel completes. Orion uses `cudaEventQuery` to query the status of the best-effort stream without blocking.

5.1.3 Memory Management.

Orion uses the policy in §5.1.1 to schedule GPU kernels on SM units. In contrast, memory allocation and memory copy operations consume only CPU-GPU PCIe bandwidth. In our current design, Orion directly submits memory operations to the GPU. We plan to extend our current implementation with techniques that manage PCIe bandwidth interference [39]. Orion could schedule each `cudaMemcpy` operation by considering its PCIe bandwidth requirements and current bus bandwidth utilization.

Orion maintains application semantics for all memory operations. For blocking operations, like `cudaMemcpy` and `cudaMemset`, Orion blocks until the operation completes on the GPU. For asynchronous memory operations, such as `cudaMemcpyAsync`, clients continue executing after Orion intercepts the operation. For memory operations that cause device synchronization (e.g., `cudaMalloc`, `cudaFree`), Orion synchronizes all clients to avoid invalid memory accesses.

The current implementation of Orion assumes that the cluster manager chooses to collocate jobs that fit in GPU memory, as assumed in other works like REEF [50]. Orion is orthogonal to and hence can be combined out-of-the-box with existing mechanisms for GPU memory swapping, such as making use of the NVIDIA Unified Memory mechanism [5], or more sophisticated swapping mechanisms as the ones proposed in Salus [104], PipeSwitch [35], ClockWork [49], and vLLM [60]. We intend to integrate layer-by-layer offloading [83] to Orion. This involves maintaining the high-priority task on the GPU while gradually swapping layers of best-effort job(s) in and out of the GPU if the whole model(s) do not fit in the remaining GPU memory. As GPU-CPU interconnects are becoming faster and faster [24], we expect that swapping will have lower overhead.

5.2 Workload Profiling

Orion's scheduling policy requires information about the compute vs. memory intensity of each kernel, the expected execution time and SM requirements of each best-effort job kernel, and the request latency of high-priority jobs. Before execution, Orion profiles each DNN workload offline and generates a file containing profile information for each kernel in the model. The Orion scheduler loads the profiling information in an in-memory lookup table, indexed by unique kernel ID.

Kernel latency and resource profiles. Orion uses the `Night Compute` [26] and `Nsight Systems` [27] tools from NVIDIA to collect the compute throughput, memory throughput, and execution time of each kernel. We use the roofline analysis in `Nsight Compute`, which classifies a kernel as compute-bound or memory-bound. Since the tool does not include roofline analysis for all kernels, we further classify a kernel as compute or memory bound if its compute throughput or memory bandwidth utilization is over 60%, respectively, as recommended by the `Nsight Compute` tool. If both compute throughput and memory bandwidth utilization are below 60% and roofline analysis is not available for the kernel, we classify its resource profile as unknown. In practice,

we find that the unknown kernels mostly occur in the update phase of a training iteration, are very small, and introduce negligible interference when collocated with other jobs. Thus, Orion allows best-effort kernels of unknown resource profiles to be collocated with any high-priority kernel.

Kernel SM requirements. For each kernel in the best-effort jobs, Orion uses the `Nsight Compute` tool to get the number of blocks, the number of threads per block, the number of registers per thread, and the amount of shared memory that the kernel requires. For each kernel k, we first determine $\texttt{blocks\_per\_sm}_k$, which is the number of blocks that can be supported per SM on the target GPU architecture for that kernel. $\texttt{blocks\_per\_sm}_k$ can be limited by the number of threads, the number of registers, or the amount of shared memory available per SM that the kernel k requires. We then calculate the number of SMs required per kernel as: $\texttt{sm\_needed}_k = \text{ceil}(\texttt{num\_blocks}_k / \texttt{blocks\_per\_sm}_k)$.

Request latency. To determine the `DUR_THRESHOLD` parameter, which Orion uses to throttle best-effort kernel launches based on their duration relative to high-priority request execution, Orion must also profile high-priority request latency, when the job is running alone in a dedicated GPU. For inference jobs, a request refers to a single batch of inference requests. For training jobs, a request refers to a single training iteration.

5.3 Integration in DNN framework

Orion's scheduling policy is agnostic to the application framework. Orion is dynamically linked to the DNN framework and is transparent to end users.

PyTorch Prototype. For our prototype, we implement Orion in PyTorch [79] in 3000 lines of C++/CUDA code. In native PyTorch, client applications launch kernels using the CUDA runtime API [14] and libraries like CUBLAS [19] and CUDNN [20], which provide high-performance implementations for common DNN operations. Orion intercepts CUDA kernel launches by overriding them with wrapper functions, which submit the necessary information (kernel identifier and arguments) to the per-client software queues. Orion currently implements wrappers for memory management operations (`cudaMalloc`, `cudaMemcpy`, `cudaMemset`, `cudaFree`, etc) and kernel launch operations (`cudaLaunchKernel`) from the CUDA runtime API, as well as CUDNN and CUBLAS functions for convolution, batch normalization, and matrix-matrix multiplication. These wrappers are sufficient to support all the DNN workloads in our evaluation, though more can be added. Intercepting operations and managing the per-client software queues is lightweight. The overhead of using Orion's wrappers is less than 1%, as we show in §6.5.

In our current prototype and evaluation, client applications and the Orion scheduler run as different threads of the same process, enabling in-process memory sharing and fast communication. Orion can also be used for applications executing as different processes. In this case, Orion executes as a separate process and clients submit kernels to queues in shared memory regions. This requires the GPU to support concurrent access from multiple processes, such as NVIDIA's MPS feature [25].

6 Evaluation

We evaluate Orion to answer the following key questions:

- How does Orion's performance compare to other GPU sharing approaches?
- What are the cost and GPU utilization benefits of using Orion compared to dedicating GPUs for each job?
- Which aspects of Orion's scheduling policy contribute most to performance benefits?
- How does Orion generalize to a new GPU architecture?
- How does Orion scale to multiple best-effort clients?
- What are the overheads of Orion's kernel profiling and kernel launch interception mechanisms?

6.1 Methodology

Experiment testbed. We evaluate Orion on an NVIDIA V100-16GB GPU using a Google Cloud `n1-standard-8` VM, which has 8 vCPU cores and 30 GB of DRAM. We use PyTorch 1.12 with Python 3.9 and CUDA 10.2. We also show that Orion generalizes to other GPU architectures by evaluating Orion on an A100-40GB GPU using an `a2-highgpu-1g` VM, with CUDA 11.3. For all experiments, we ensure jobs execute with no data preprocessing or communication bottlenecks. Hence, we evaluate Orion's ability to improve GPU utilization while minimizing interference in the most challenging setting, where each individual job maximizes its own GPU utilization. We repeat each experiment three times.

Workloads. We consider three common GPU sharing use cases. First, we collocate a high-priority, latency-sensitive inference job with a best-effort training job (*inf-train*). Next, we collocate high-priority and best-effort training (*train-train*). Finally, we collocate a high-priority, latency-sensitive inference job with best-effort offline inference jobs (*inf-inf*).

For each use case, we consider popular DNN models from computer vision and natural language processing (NLP) domains. ResNet50, ResNet101 [51] and MobileNet-v2 [84] are representative vision models. We use their TorchVision implementations [28]. BERT [43] and Transformer [92] are representative NLP models. We use the implementations from NVIDIA [22]. We use full-precision for both training and inference. Table 1 summarizes the batch size for each workload. We match batch sizes to those used in prior works on the same or comparable GPU platforms [21, 23, 51, 54, 84, 97].

We consider uniform and Poisson request arrival distributions for inference jobs. A uniform distribution is representative of application domains such as autonomous driving (e.g., where cameras detect obstacles [36, 37]), whereas Poisson arrivals are representative of event-driven, real-time DNN applications (e.g., speech recognition [50, 52]). We select

	Inf-Inf		Inf-Train
Model	Uniform	Poisson	Poisson
ResNet50	80	50	15
MobileNet-v2	100	65	40
ResNet101	40	25	9
BERT	8	5	4
Transformer	20	12	8

Table 3. Requests per second (RPS) for DNN inference jobs.

the mean request arrival rates (shown in Table 3) to match the mean invocation request rates of the top 20 most frequently executed functions in the Microsoft Azure Functions trace [87], as used in other works [49, 105] to stress-test GPU collocation scenarios. For the vision models, we also use an inference trace collected from a real object detection model deployment in the Apollo autonomous driving system [36]. This inference trace is from the DISB inference serving benchmark [17], first used to evaluate REEF [50]. For experiments with the Apollo trace, we use the trace's invocation timestamps for the high-priority inference job and assume uniform request inter-arrival for collocated best-effort inference jobs. Meanwhile, training jobs submit requests in a closed loop.

Baselines. We compare Orion to temporal sharing, which time-slices the GPU by executing one job's request at a time, while prioritizing the high-priority job. We also compare Orion to NVIDIA MPS [25] and GPU Streams [3] spatial sharing mechanisms. GPU Streams allow multiple client applications to share a GPU, as long as they are part of the same process. Hence, for this baseline, we run each DNN application client as a separate *thread*, which submits requests to a separate CUDA stream. We assign a high-priority stream to the high-priority job, and a default-priority stream to each of the best-effort jobs. MPS is a feature in NVIDIA GPUs with compute capability 3.5 or higher, which enables multiple processes to spatially share the GPU.

We also compare to the state-of-the-art REEF [50] GPU sharing policy. REEF is originally developed for AMD GPUs, which allows users to preempt kernels during execution. For NVIDIA GPUs, the authors proposed a restricted version of kernel preemption, REEF-N, which allows high-priority kernels to bypass best-effort kernels in software queues before submission to GPU. Since only the AMD GPU version of REEF is currently open source, we implement REEF-N and the kernel selection rules from the original paper, which schedule kernels based on their size (number of SMs) and expected latency. We use a software queue size of 12 kernels, based on discussions with the REEF authors. While REEF was primarily designed to collocate inference jobs (since training jobs include non-idempotent kernels that update model state), REEF-N is safe to use for training jobs as kernel execution is never preempted. Hence, we compare Orion to the REEF-N policy for all collocation use cases.

For training job collocation (*train-train*), we compare Orion to Tick-Tock [94], which offsets the forward and backward passes of training minibatch iterations to minimize aggregate memory usage and reduce interference. Zico [67] also implements this approach. Since neither system has an available open-source implementation for PyTorch, we implement the approach based on the papers.

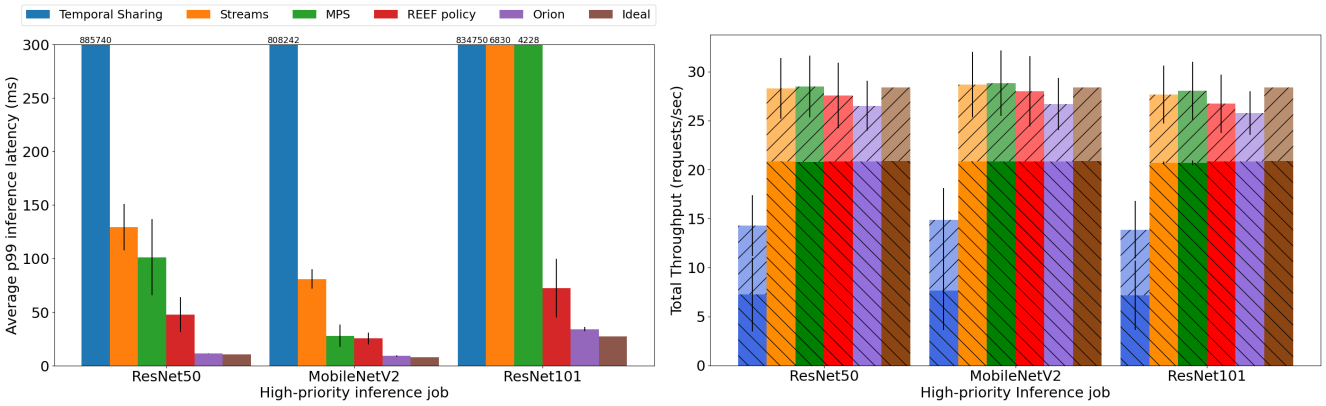
As a performance upper bound, we measure the latency and throughput of each workload on a dedicated GPU. The Ideal baseline has latency equal to the high-priority job's latency with no collocation and throughput equal to the sum of high-priority's and best-effort jobs' dedicated GPU throughput. This baseline is a lower bound for latency and an upper bound for throughput.

6.2 Performance and cost benefits

We evaluate *inf-train*, *train-train*, and *inf-inf* collocation use cases with two DNN clients at a time: one high-priority and one best-effort. For the high-priority inference job, we report the *p99* latency. We observe similar trends for all baselines for the p50 and p95 latency, so we omit the respective plots for brevity.

6.2.1 Inference-Training. Figure 6 shows the p99 latency and aggregate throughput of a high-priority inference job with Apollo trace request arrivals, collocated with a best-effort training job. Each bar represents the performance of a high-priority inference workload (labeled on the x-axis), averaged across experiments that collocate each one of the DNN training jobs from Table 3. The standard deviation bars show how inference latency varies across the five different collocated training jobs. Figure 7 shows results for a similar *inf-train* setup, but assuming Poisson arrivals for the high-priority inference job.

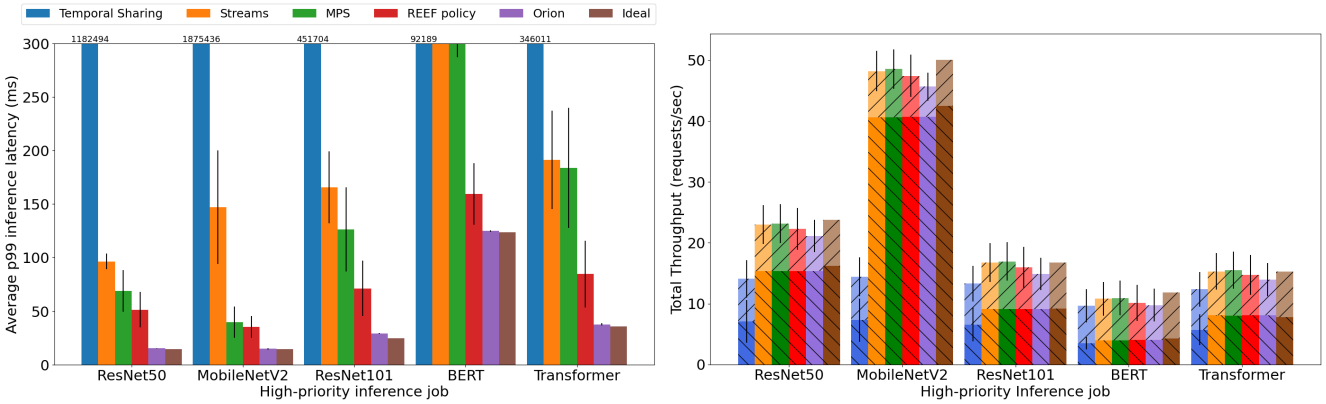
Temporal sharing leads to high tail inference latency, despite prioritizing inference requests. An incoming inference request must wait for any ongoing training iteration to complete before it starts executing, resulting in high queuing delays. Spatial sharing mechanisms improve performance by parallelizing request execution across multiple streams. However, GPU Streams and MPS mechanisms only maximize aggregate throughput and do not prioritize the latency-sensitive inference job, leading to high tail latency. MPS generally achieves lower latency than Streams, since multi-processing is more efficient than multi-threading in Python applications. Clients in the Streams baseline run as different threads, hence they contend for the Python global interpreter lock [10]. The REEF policy is also unable to maintain low p99 inference latency, as it lacks interference-aware scheduling and does not sufficiently throttle the best-effort training job. On average, REEF's inference latency is 3.44× higher than



(a) p99 latency of inference job averaged across collocation experiments with each model training job.

(b) The bold bar is inference throughput (for high-priority model on x-axis). The faint bar is the average best-effort training throughput.

Figure 6. Inference-Training (Apollo trace): inference latency and total throughput



(a) p99 latency of inference job averaged across collocation experiments with each model training job.

(b) The bold bar is inference throughput (for high-priority model on x-axis). The faint bar is the average best-effort training throughput.

Figure 7. Inference-Training (Poisson): inference latency and total throughput.

ideal for the Apollo trace experiments and 2.5× higher than ideal for the Poisson arrival experiments.

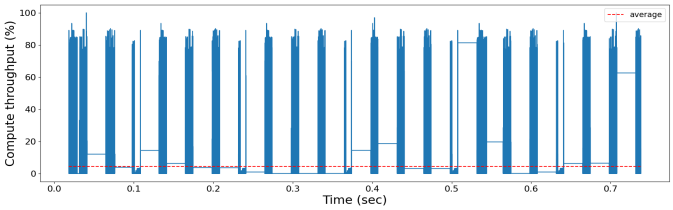
In contrast, Orion keeps p99 latency of the high-priority inference job within 14% of the ideal latency, which is 2.3-3× lower than REEF, on average. Orion also achieves low variance in inference latency across collocations with different models. At the same time, Orion increases aggregate throughput by up to 1.3× and 2.3× compared to inference throughput on a dedicated GPU, for the Apollo and Poisson experiments, respectively.

Table 4 shows the cost benefits of using Orion to collocate a Poisson arrival inference job with different training jobs on a single GPU, compared to dedicating separate GPUs for each job. We calculate the cost savings as:

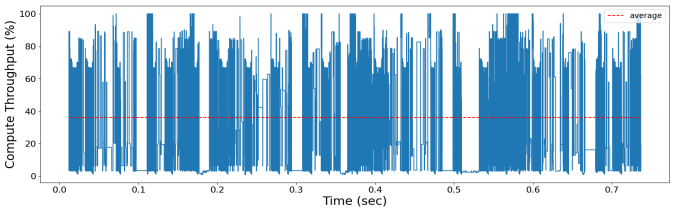
$$cost\ savings = \frac{2\ GPU \cdot JCT_{dedicated}}{1\ GPU \cdot JCT_{collocated}} = \frac{2 \cdot Throughput_{collocated}}{Throughput_{dedicated}}$$

where JCT is the job completion time. Dedicated and collocated scenarios use 2 GPUs and 1 GPU, respectively. Throughput is the inverse of job completion time. Overall, Orion achieves 1.26× to 1.49× cost savings.

We also measure Orion's impact on GPU utilization. Figure 8a plots V100 GPU Compute Throughput utilization over time for a ResNet50 inference job running alone, while Figure 8b shows utilization when using Orion to collocate the inference job with a best-effort ResNet50 training job. The inference job submits requests with a uniform distribution at 100 requests per second. Orion fills in the fine-grained low-utilization periods of the inference job, increasing the average Compute Throughput utilization from 7% to 36%. The remaining periods of low utilization in Figure 8b correspond to times when memory copy operations execute from the host CPU to GPU. The GPU hardware cannot schedule kernels during memory copies [1]. Similarily, Figure 9a and 9b

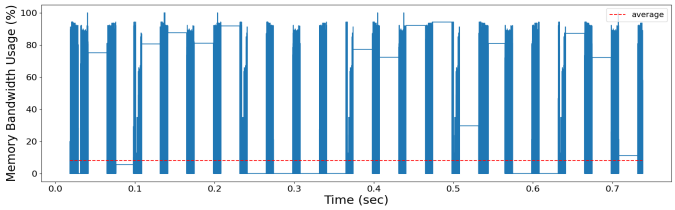


(a) ResNet50 Inference alone

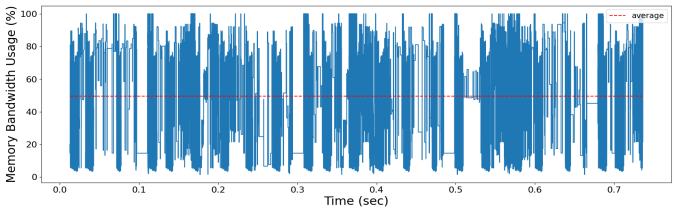


(b) ResNet50 Inference + ResNet50 Training

Figure 8. Inference job Compute throughput utilization on a dedicated GPU vs. collocated with training.



(a) ResNet50 Inference alone



(b) ResNet50 Inference + ResNet50 Training

Figure 9. Inference Memory bandwidth utilization on a dedicated GPU vs. collocated with training.

Model	Dedicated Training iterations/sec	Collocated Training iterations/sec	Cost Savings
ResNet50	10.3	7.45	1.45×
MobileNetV2	12.5	8.78	1.4×
ResNet101	6.3	4.7	1.49×
BERT	4.91	3.1	1.26×
Transformer	6	3.9	1.3×

Table 4. Training throughput on a dedicated GPU for each model vs. its average training throughput when collocated with inference jobs using Orion. Cost savings come from collocating on a single GPU vs. using 2 separate GPUs.

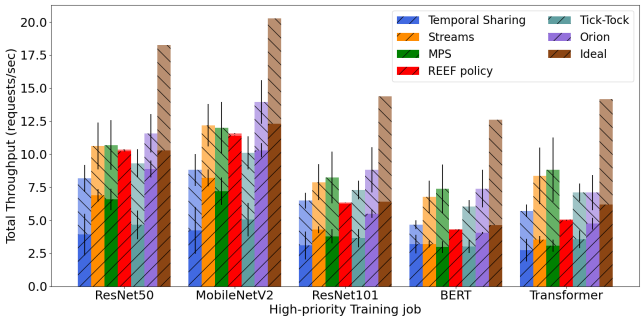


Figure 10. Average throughput of high priority (bold bar) and best-effort training jobs (faint bar).

plot the GPU memory bandwidth utilization of the inference job running alone and colocated with training, respectively. In that case, Orion improves Memory Bandwidth Utilization from 10% to 47%. Finally, Orion improves SM utilization from 11% to 49%.

6.2.2 Training-Training. Figure 10 shows average aggregate throughput when collocating high-priority and best-effort training jobs. With MPS and Streams, high-priority job throughput is, on average, 1.7× less than the job's throughput on a dedicated GPU, due to interference. MPS achieves up to 6% higher throughput than Streams due to process-based parallelism. Tick-Tock exhibits the lowest throughput among all baselines, mainly due to synchronization at the beginning and end of the forward and backward passes. This causes the fastest job to wait for the slowest one, reducing the throughput of the high-priority training job by 1.93×.

In contrast, REEF maintains high-priority job throughput within 8% of ideal. However, Figure 10 shows that REEF heavily throttles best-effort kernels, as few best-effort training iterations complete. Orion achieves the best of both worlds. It maintains high-priority job throughput within 16% of ideal, while achieving up to 1.6× higher throughput compared to dedicating the GPU to the high-priority job. By collocating the best-effort job during low-utilization periods of the high-priority job, Orion effectively increases overall GPU utilization. For example, when collocating a high-priority job training BERT with a best-effort job training MobileNet, Orion increases average SM utilization, compute utilization, and memory bandwidth utilization by 1.4×, 1.34×, and 1.7×, respectively.

By making progress on best-effort training jobs while serving a high-priority training job, Orion reduces the total time to complete a set of training jobs (i.e., job makespan). We compare the cost of training all examined models on a single GPU with Orion versus executing training jobs sequentially

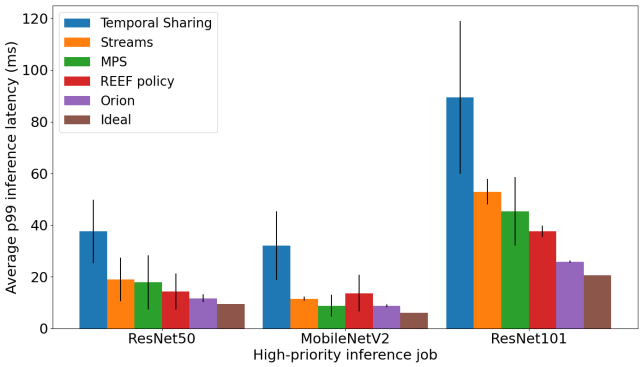


Figure 11. Inference-inference (Apollo): p99 latency of high-priority model (on x-axis) averaged across collocation experiments with all other models.

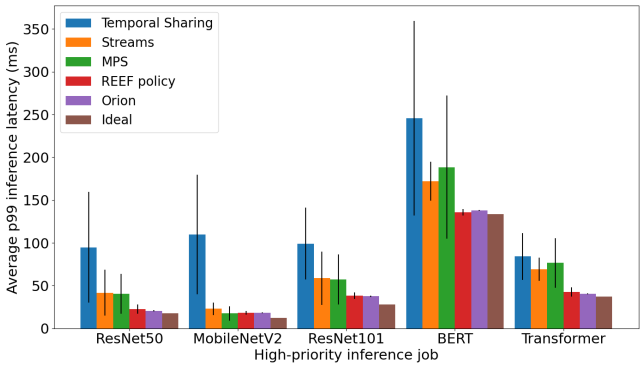


Figure 12. Inference-Inference (Poisson): p99 latency of high-priority model (on x-axis) averaged across collocation experiments with all other models.

on a dedicated GPU. We run ResNet50, ResNet101, and BERT as high-priority training jobs and MobileNet-V2 and Transformer as best-effort jobs. Orion reduces the job makespan by 1.29×, leading to 1.29× cost savings as the GPU is required for less time to complete the jobs. In comparison, the MPS baseline achieves only 1.14× cost savings compared to sequential execution, and at the expense of 1.25× higher job completion time for high-priority jobs compared to Orion. Compared to REEF, Orion reduces JCT and cost by 1.29×.

6.2.3 Inference-Inference. Figure 11 shows the p99 inference latency when collocating a high-priority vision inference job with a best-effort inference job. The model on the x-axis is the high-priority model, which receives inference requests based on the Apollo trace. The best-effort inference job receives requests with uniform inter-arrival distribution. Figure 12 shows a similar experiment, but assuming Poisson arrivals for both jobs. In our *inf-inf* experiments, all baselines achieve similar aggregate throughput (not shown in the plots), but the tail latency of the high-priority inference job differs greatly across baselines.

GPU Streams and MPS incur high p99 latency overhead: on average 1.89× higher than the ideal p99 latency and with high variance across model collocations. Since REEF does not take into account the compute versus memory intensity of the kernels it schedules, its p99 latency for the high-priority job is 1.86× and 1.25× higher than the ideal case, for the Apollo and Poisson experiments, respectively.

In contrast, Orion keeps the high-priority inference p99 latency within 22% and 15% of the ideal latency for the Apollo and Poisson experiments, respectively, while increasing aggregate inference throughput by up to 2× (for Apollo) and 7.3× (for Poisson) compared to dedicating the GPU to the high-priority job only. Overall, Orion provides cost savings of 2× for 2-client *inf-inf* use cases compared to the dedicated GPU case, since Orion serves the models on a single GPU instead of one for each job.

6.3 Generalizing to other GPUs and more clients

As GPU hardware resource capacity continues to scale with each new generation [11], more tenants can be collocated. To show that Orion can scale to multiple clients and generalize to other GPU architectures, we evaluate Orion with 5 inference clients sharing a A100-40GB GPU. Figure 13 plots p99 latency of the high-priority inference job (labeled on x-axis) collocated with 4 best-effort inference jobs serving the other models in Table 3, all with Poisson request arrivals. We show standard deviation across three runs of the same experiment. We compare MPS and REEF with Orion. We omit temporal sharing and Streams baselines due to their poor performance (tail latency is 3 orders of magnitude higher than ideal).

MPS leads to 2.2× higher p99 latency than ideal. Although employing REEF's fine-grained scheduling policy helps reduce tail latency, it is still 21% higher than ideal. In contrast, Orion's interference-aware policy and control mechanisms keep the p99 latency of all workloads within 9% of ideal, showcasing Orion's ability to generalize across GPU generations and scale to multiple clients.

6.4 Performance analysis breakdown

In Figure 14, we analyze which aspects of Orion's policy contribute most to performance benefits. We show results for the *inf-train* use case with Poisson inference arrivals. Our conclusions apply to the other use cases as well. We start by simply assigning each client to a different CUDA stream, each with the default priority. The `GPU Streams` bar in Figure 14 shows this approach has high latency. The `Stream Priorities` baseline shows that using the highest CUDA priority for the stream of the high-priority inference job helps reduce the p95 latency by up to 25%. Adding the first component of Orion's policy, which schedules best-effort job kernels based on their compute-memory resource profiles, reduces the p95 latency by an additional 48%. Finally, taking into account the sizes (number of SMs) of the kernels reduces the latency by up to 54% on top of the `Compute/Mem`

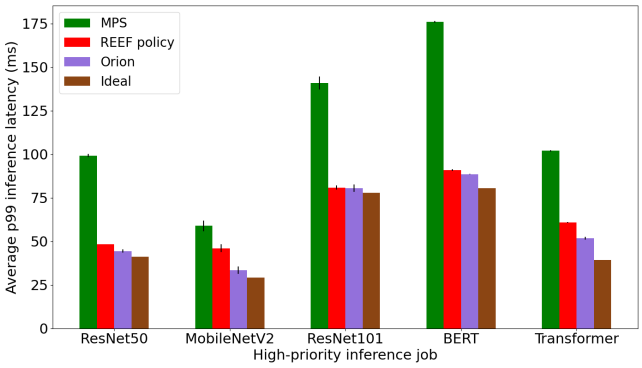


Figure 13. p99 latency of high-priority inference job when collocated with 4 best-effort inference jobs on A100 GPU.

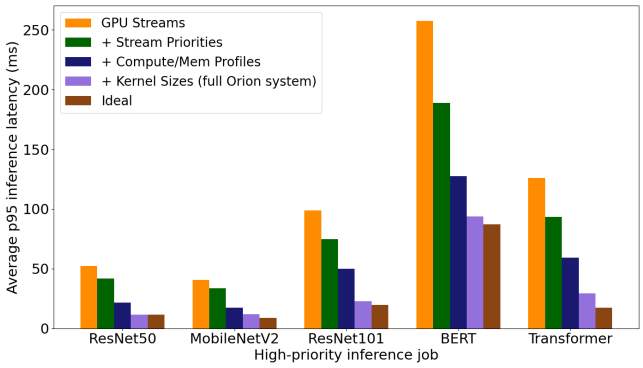


Figure 14. Orion performance analysis breakdown for *inf-train*, showing which aspects benefit tail latency the most.

`profiles` baseline. Hence, compute/memory-aware and size-aware scheduling are roughly equally important. We then check whether stream priorities are essential to the Orion system after applying compute/memory profiles and kernel size-based scheduling. The stream priority mechanism has only marginal improvements at this point, hence Orion can also be used in settings where the GPU hardware does not support stream priorities (e.g., in MPS mode [46]).

We also tune `DUR_THRESHOLD`. We find that Orion has stable performance for `DUR_THRESHOLD` values below 3%. Linear increases in `DUR_THRESHOLD` beyond 3% lead to approximately linear decrease in high-priority job performance, due to less throttling of best-effort kernels. For example, when collocating ResNet101 inference with best-effort training, inference latency is 23ms, 26ms, and 30ms for `DUR_THRESHOLD` values of 10%, 15%, and 20%, respectively, while best-effort training throughput is 8.7, 9.26, and 9.75 iterations/sec. Users can tune `DUR_THRESHOLD` based on high-priority job service level objectives. We use 2.5% in our experiments.

6.5 Overheads

Kernel launch interception. We measure the execution time of each inference and training job on a dedicated GPU using Orion's kernel interception mechanism to directly schedule kernels. Compared to native PyTorch, Orion's overhead remains less than 1% across all jobs.

Kernel resource profiling. We use the `Nsight Systems (NSYS)` and `Nsight Compute (NCU)` tools from NVIDIA to profile the first 10 mini-batches of a training job or 10 requests of an inference job. The `Nsys` tool adds up to 5% overhead in the iteration time. The `NCU` tool performs a much more detailed resource analysis for each kernel (e.g. cache misses, warp scheduler statistics), and the profiling time is proportional to the number of kernels. In our experiments, it takes ~2-5 seconds per kernel. Since profiling is offline, the tools do not affect the actual job execution.

7 Discussion

Cluster manager co-design. Orion is currently implemented as a per-GPU scheduler. In the future, we plan to explore co-design with cluster management. By using each job's compute and memory intensity kernel profiles, the cluster manager can place jobs with complementary resource profiles on the same GPU(s) to maximize resource utilization and mitigate interference.

Software/hardware co-design. Optimizing GPU utilization and performance is challenging with the limited interface that GPU hardware currently exposes to host software. We draw inspiration from the OpenSSD [59] platform, which enables research in Flash storage device controller hardware and host software co-design by exposing a lower-level interface to software. Similar to how software/hardware co-design can minimize interference on shared SSDs [31, 63, 64, 69], enabling software/hardware co-design for GPU scheduling (e.g., controlling the placement of GPU kernels across SMs) can allow applications to tune end-to-end performance on shared GPUs. This is particularly important as recent trends in GPU programming increasingly offload more and more scheduling to GPU hardware. For instance, CUDA graphs [6] schedule entire graphs of kernels in the GPU with a single CUDA API call to reduce CPU launching overhead. In this context, Orion's scheduling policy could be implemented either at the GPU driver or GPU scheduler level, to interleave kernels from multiple graphs while minimizing interference.

GPU cache interference. We currently do not consider GPU cache interference. NVIDIA tools provide cache miss statistics [26], which can be used to infer more specific profiles of kernels and model interference more accurately.

Security. We assume the clients sharing a GPU are in the same trust domain, which is a reasonable assumption in DNN clusters operated by the same organization [98, 99]. Hence, Orion minimizes performance interference, but does not guarantee secure isolation between untrusted clients sharing a GPU. Trusted execution environments for heterogeneous hardware are an active area of research [93].

Applicability to Large Language Models (LLMs). We plan to further investigate Orion's applicability to Large Language Models [38, 90]. Previous works [55, 60] have shown that the token generation phase of LLM inference, which happens sequentially, token after token, is memory-bound, while underutilizing GPU's compute throughput and SMs. Thus, we can employ Orion's resource-aware scheduling policy to colocate LLM inference with computationally intensive workloads. However, the large size of LLMs [38], as well as the *Key-Value cache*, used to speedup token generation [81], significantly inflate the memory requirements of LLM inference. Therefore, when colocating with other workloads, additional memory swapping mechanisms must be employed. As outlined in section 5.1.3, we plan to enhance Orion with existing DNN swapping mechanisms. One such mechanism is PagedAttention [60] which offers dynamic allocation and swapping for LLM inference and can be seamlessly integrated with Orion.

8 Conclusion

Orion is a GPU scheduler that transparently schedules tasks from multiple clients sharing a GPU at the granularity of individual operators. By considering the size, compute, and memory profiles of each operator, Orion reduces interference to maintain high performance for a high-priority workload while saving up to 1.49× in cost by making progress on collocated best-effort jobs (compared to dedicating GPUs to individual jobs). Unlike other GPU sharing techniques, Orion schedules at the granularity of individual GPU kernels, enabling it to leverage spare GPU resources available for short time periods (e.g., 10s to 1000s of μs) during DNN job execution. This approach significantly reduces tail latency for high-priority jobs compared to prior GPU sharing systems.

Acknowledgement

We thank our anonymous reviewers and our shepherd, Youngjin Kwon, for their valuable feedback. We thank Benoit Steiner, Amar Phanishayee, Bowen Wu, and Maximilian Böther for their helpful insights at various stages of the project. Foteini Strati is supported by the Swiss National Science Foundation (Project Number 200021_204620).

References

[1] 2012. How to Overlap Data Transfers in CUDA C/C++. https://developer.nvidia.com/blog/how-overlap-data-transfers-cuda-cc/.
[2] 2013. GPU Performance Optimization: Programming Guidelines and GPU Architecture Reasons Behind Them. https://on-demand.gputechconf.com/gtc/2013/presentations/S3466-Programming-Guidelines-GPU-Architecture.pdf.
[3] 2014. CUDA Streams. https://on-demand.gputechconf.com/gtc/2014/presentations/S4158-cuda-streams-best-practices-common-pitfalls.pdf.
[4] 2014. How the Fermi Thread Block Scheduler Works (Illustrated). https://www.cs.rochester.edu/~sree/fermi-tbs/fermi-tbs.html.
[5] 2017. Unified Memory for CUDA Beginners. https://developer.nvidia.com/blog/unified-memory-cuda-beginners/.
[6] 2019. Getting Started with CUDA Graphs. https://developer.nvidia.com/blog/cuda-graphs/.
[7] 2019. High priority stream preemption. https://forums.developer.nvidia.com/t/how-high-priority-stream-preemption/78183/1.
[8] 2020. Metric references and description. https://forums.developer.nvidia.com/t/metric-references-and-description/111750.
[9] 2020. NVIDIA, Metrics references and description. https://forums.developer.nvidia.com/t/metric-references-and-description/111750/2.
[10] 2021. Python GlobalInterpreterLock. https://wiki.python.org/moin/GlobalInterpreterLock.
[11] 2022. NVIDIA Hopper, Ampere GPUs Sweep Benchmarks in AI Training. https://blogs.nvidia.com/blog/2022/11/09/mlperf-ai-training-hpc-hopper/.
[12] 2022. NVIDIA Multi-Instance GPU User Guide. https://docs.nvidia.com/datacenter/tesla/mig-user-guide/.
[13] 2023. Cuda Programming. https://docs.nvidia.com/cuda/cuda-c-programming-guide/.
[14] 2023. CUDA RUNTIME API. https://docs.nvidia.com/cuda/cuda-runtime-api/index.html.
[15] 2023. cudaStreamCreateWithPriority. https://docs.nvidia.com/cuda/cuda-runtime-api/group__CUDART__STREAM.html#group__CUDART__STREAM_1ge2be9e9858849bf62ba4a8b66d1c3540.
[16] 2023. Deploy machine learning models in production environments. https://learn.microsoft.com/en-us/azure/cloud-adoption-framework/innovate/best-practices/ml-deployment-inference.
[17] 2023. DISB: DNN Inference Serving Benchmark. https://github.com/SJTU-IPADS/disb.
[18] 2023. Event Management. https://docs.nvidia.com/cuda/cuda-runtime-api/group__CUDART__EVENT.html.
[19] 2023. NVIDIA cuBLAS. https://docs.nvidia.com/cuda/cublas/index.html.
[20] 2023. NVIDIA cuDNN Documentation. https://docs.nvidia.com/deeplearning/cudnn/developer-guide/index.html.
[21] 2023. NVIDIA Deep Learning Examples, BERT for PyTorch. https://github.com/NVIDIA/DeepLearningExamples/tree/master/PyTorch/LanguageModeling/BERT.
[22] 2023. NVIDIA Deep Learning Examples for Tensor Cores. https://github.com/NVIDIA/DeepLearningExamples.
[23] 2023. NVIDIA Deep Learning Examples, Trasnformer-XL for PyTorch. https://github.com/NVIDIA/DeepLearningExamples/tree/master/PyTorch/LanguageModeling/Transformer-XL.
[24] 2023. NVIDIA DGX GH200. https://www.nvidia.com/en-us/data-center/dgx-gh200/.
[25] 2023. NVIDIA MPS. https://docs.nvidia.com/deploy/mps/.
[26] 2023. NVIDIA Nsight Compute. https://developer.nvidia.com/nsight-compute.
[27] 2023. NVIDIA Nsight Systems. https://developer.nvidia.com/nsight-systems.
[28] 2023. TorchVision Models. https://pytorch.org/vision/stable/models.html.
[29] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. 2016. Tensorflow: A system for large-scale machine learning. In 12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16). 265–283.
[30] Tanya Amert, Nathan Otterness, Ming Yang, James H. Anderson, and F. Donelson Smith. 2017. GPU Scheduling on the NVIDIA TX2: Hidden Details Revealed. In 2017 IEEE Real-Time Systems Symposium (RTSS). 104–115. https://doi.org/10.1109/RTSS.2017.00017
[31] Mijin An, In-Yeong Song, Yong-Ho Song, and Sang-Won Lee. 2022. Avoiding Read Stalls on Flash Storage. In Proceedings of the 2022 International Conference on Management of Data (SIGMOD '22).

[32] Joel André, Foteini Strati, and Ana Klimovic. 2022. Exploring Learning Rate Scaling Rules for Distributed ML Training on Transient Resources. In Proceedings of the 3rd International Workshop on Distributed Machine Learning (Rome, Italy) (DistributedML '22). Association for Computing Machinery, New York, NY, USA, 1–8. https://doi.org/10.1145/3565010.3569067

[33] Sanjith Athlur, Nitika Saran, Muthian Sivathanu, Ramachandran Ramjee, and Nipun Kwatra. 2022. Varuna: Scalable, Low-Cost Training of Massive Deep Learning Models. In Proceedings of the Seventeenth European Conference on Computer Systems (EuroSys '22).

[34] Andrew Audibert, Yang Chen, Dan Graur, Ana Klimovic, Jiri Simsa, and Chandramohan A. Thekkath. 2022. A case for disaggregation of ML data processing. arXiv:2210.14826

[35] 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). USENIX Association, 499–514. https://www.usenix.org/conference/osdi20/presentation/bai

[36] Baidu. 2023. Apollo. https://apollo.auto/.

[37] Abhishek Balasubramaniam and Sudeep Pasricha. 2022. Object Detection in Autonomous Vehicles: Status and Open Challenges. CoRR abs/2201.07706 (2022). https://arxiv.org/abs/2201.07706

[38] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel Ziegler, Jeffrey Wu, Clemens Winter, Chris Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language Models are Few-Shot Learners. In Advances in Neural Information Processing Systems, H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin (Eds.), Vol. 33. Curran Associates, Inc., 1877–1901. https://proceedings.neurips.cc/paper_files/paper/2020/file/1457c0d6bfcb4967418bfb8ac142f64a-Paper.pdf

[39] Quan Chen, Hailong Yang, Jason Mars, and Lingjia Tang. 2016. Baymax: QoS Awareness and Increased Utilization for Non-Preemptive Accelerators in Warehouse Scale Computers. In Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '16).

[40] Trishul Chilimbi, Yutaka Suzue, Johnson Apacible, and Karthik Kalyanaraman. 2014. Project Adam: Building an Efficient and Scalable Deep Learning Training System. In Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation (OSDI'14).

[41] 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

[42] 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

[43] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT.

[44] Shaoduo Gan, Jiawei Jiang, Binhang Yuan, Ce Zhang, Xiangru Lian, Rui Wang, Jianbin Chang, Chengjun Liu, Hongmei Shi, Shengzhuo Zhang, Xianghong Li, Tengxu Sun, Sen Yang, and Ji Liu. 2021. Bagua: Scaling up Distributed Learning with System Relaxations. Proc. VLDB Endow. 15, 4 (2021).

[45] Guin Gilman, Samuel S. Ogden, Tian Guo, and Robert J. Walls. 2021. Demystifying the Placement Policies of the NVIDIA GPU Thread Block Scheduler for Concurrent Kernels. SIGMETRICS Perform. Eval. Rev. 48, 3 (2021).

[46] Guin Gilman and Robert J. Walls. 2022. Characterizing Concurrency Mechanisms for NVIDIA GPUs under Deep Learning Workloads. SIGMETRICS Perform. Eval. Rev. 49, 3 (2022).

[47] Priya Goyal, Piotr Dollár, Ross B. Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. 2017. Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour. CoRR abs/1706.02677 (2017).

[48] Dan Graur, Damien Aymon, Dan Kluser, Tanguy Albrici, Chandramohan A. Thekkath, and Ana Klimovic. 2022. Cachew: Machine Learning Input Data Processing as a Service. In 2022 USENIX Annual Technical Conference (USENIX ATC 22). 689–706.

[49] 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).

[50] 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).

[51] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep Residual Learning for Image Recognition. In Proceedings of 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR '16).

[52] Geoffrey Hinton, Li Deng, Dong Yu, George E. Dahl, Abdel-rahman Mohamed, Navdeep Jaitly, Andrew Senior, Vincent Vanhoucke, Patrick Nguyen, Tara N. Sainath, and Brian Kingsbury. 2012. Deep Neural Networks for Acoustic Modeling in Speech Recognition: The Shared Views of Four Research Groups. IEEE Signal Processing Magazine 29, 6 (2012).

[53] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Mia Xu Chen, Dehao Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V. Le, Yonghui Wu, and Zhifeng Chen. 2019. GPipe: Efficient Training of Giant Neural Networks Using Pipeline Parallelism.

[54] Andrei Ivanov, Nikoli Dryden, Tal Ben-Nun, Shigang Li, and Torsten Hoefler. 2021. Data Movement Is All You Need: A Case Study on Optimizing Transformers. In Proceedings of Machine Learning and Systems.

[55] Yunho Jin, Chun-Feng Wu, David Brooks, and Gu-Yeon Wei. 2023. S^3: Increasing GPU Utilization during Generative Inference for Higher Throughput. arXiv:2306.06000 [cs.AR]

[56] Nitish Shirish Keskar, Dheevatsa Mudigere, Jorge Nocedal, Mikhail Smelyanskiy, and Ping Tak Peter Tang. 2017. On Large-Batch Training for Deep Learning: Generalization Gap and Sharp Minima. In 5th International Conference on Learning Representations, ICLR.

[57] Alexandros Koliousis, Pijika Watcharapichat, Matthias Weidlich, Luo Mai, Paolo Costa, and Peter Pietzuch. 2019. Crossbow: Scaling Deep Learning with Small Batch Sizes on Multi-GPU Servers. Proc. VLDB Endow. 12, 11 (jul 2019), 1399–1412. https://doi.org/10.14778/3342263.3342276

[58] Michael Kuchnik, Ana Klimovic, Jiri Simsa, Virginia Smith, and George Amvrosiadis. 2022. Plumber: Diagnosing and Removing Performance Bottlenecks in Machine Learning Data Pipelines. In Proc. of Machine Learning and Systems, Vol. 4. 33–51.

[59] Jaewook Kwak, Sangjin Lee, Kibin Park, Jinwoo Jeong, and Yong Ho Song. 2020. Cosmos+ OpenSSD: Rapid Prototype for Flash Storage Systems. ACM Trans. Storage 16, 3, Article 15 (jul 2020).

[60] 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 (Koblenz, Germany) (SOSP '23). Association for Computing Machinery, New York, NY, USA, 611–626. https://doi.org/10.1145/3600006.3613165

[61] Joel Lamy-Poirier. 2023. Breadth-First Pipeline Parallelism. arXiv:2211.05953 [cs.DC]

[62] Gyewon Lee, Irene Lee, Hyeonmin Ha, Kyunggeun Lee, Hwarim Hyun, Ahnjae Shin, and Byung-Gon Chun. 2021. Refurbish Your Training Data: Reusing Partially Augmented Samples for Faster Deep Neural Network Training. In USENIX Annual Technical Conference (ATC'21). 537–550.

[63] Sangjin Lee, Alberto Lerner, André Ryser, Kibin Park, Chanyoung Jeon, Jinsub Park, Yong Ho Song, and Philippe Cudré-Mauroux. 2022. X-SSD: A Storage System with Native Support for Database Logging and Replication. In SIGMOD '22: International Conference on Management of Data.

[64] Sungjin Lee, Ming Liu, Sangwoo Jun, Shuotao Xu, Jihong Kim, and Arvind Arvind. 2016. Application-Managed Flash. In Proceedings of the 14th Usenix Conference on File and Storage Technologies (FAST'16).

[65] 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. https://doi.org/10.1145/3542929.3563510

[66] Jiamin Li, Hong Xu, Yibo Zhu, Zherui Liu, Chuanxiong Guo, and Cong Wang. 2023. Lyra: Elastic Scheduling for Deep Learning Clusters. In Proc. of European Conference on Computer Systems (EuroSys '23).

[67] 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 (USENIX ATC 21).

[68] Yujun Lin, Song Han, Huizi Mao, Yu Wang, and William Dally. 2018. Deep Gradient Compression: Reducing the Communication Bandwidth for Distributed Training. https://openreview.net/pdf?id=SkhQHMW0W

[69] Heiner Litz, Javier Gonzalez, Ana Klimovic, and Christos Kozyrakis. 2022. RAIL: Predictable, Low Tail Latency for NVMe Flash. ACM Trans. Storage 18, Article 5 (2022).

[70] Saeed Maleki, Madan Musuvathi, Todd Mytkowicz, Olli Saarikivi, Tianju Xu, Vadim Eksarevskiy, Jaliya Ekanayake, and Emad Barsoum. 2021. Scaling Distributed Training with Adaptive Summation. In Proceedings of Machine Learning and Systems, A. Smola, A. Dimakis, and I. Stoica (Eds.), Vol. 3. 335–349. https://proceedings.mlsys.org/paper_files/paper/2021/file/427e0e886ebf87538afdf0badb805b7f-Paper.pdf

[71] Dominic Masters and Carlo Luschi. 2018. Revisiting Small Batch Training for Deep Neural Networks. arXiv:1804.07612 [cs.LG]

[72] Jayashree Mohan, Amar Phanishayee, Ashish Raniwala, and Vijay Chidambaram. 2021. Analyzing and Mitigating Data Stalls in DNN Training. In VLDB 2021.

[73] Derek G. Murray, Jiri Simsa, Ana Klimovic, and Ihor Indyk. 2021. tf.data: A Machine Learning Data Processing Framework. In VLDB 2021, Vol. 14.

[74] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R. Devanur, Gregory R. Ganger, Phillip B. Gibbons, and Matei Zaharia. 2019. PipeDream: Generalized Pipeline Parallelism for DNN Training. In Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP '19).

[75] 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 (Koblenz, Germany) (SOSP '23). Association for Computing Machinery, New York, NY, USA, 595–610. https://doi.org/10.1145/3600006.3613163

[76] Andrew Or, Haoyu Zhang, and Michael None Freedman. 2022. VirtualFlow: Decoupling Deep Learning Models from the Underlying Hardware. In Proceedings of Machine Learning and Systems, Vol. 4.

[77] Nathan Otterness, Ming Yang, Sarah Rust, Eunbyung Park, James H. Anderson, F. Donelson Smith, Alex Berg, and Shige Wang. 2017. An Evaluation of the NVIDIA TX1 for Supporting Real-Time Computer-Vision Workloads. In 2017 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS). 353–364. https://doi.org/10.1109/RTAS.2017.3

[78] Seo Jin Park, Joshua Fried, Sunghyun Kim, Mohammad Alizadeh, and Adam Belay. 2022. Efficient Strong Scaling Through Burst Parallel Training. In Proceedings of Machine Learning and Systems, D. Marculescu, Y. Chi, and C. Wu (Eds.), Vol. 4. 748–761. https://proceedings.mlsys.org/paper_files/paper/2022/file/b99e69074b2fa1d8c8fe0d5b60e19397-Paper.pdf

[79] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In Advances in Neural Information Processing Systems.

[80] Yanghua Peng, Yibo Zhu, Yangrui Chen, Yixin Bao, Bairen Yi, Chang Lan, Chuan Wu, and Chuanxiong Guo. 2019. A Generic Communication Scheduler for Distributed DNN Training Acceleration. In Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP '19).

[81] Reiner Pope, Sholto Douglas, Aakanksha Chowdhery, Jacob Devlin, James Bradbury, Anselm Levskaya, Jonathan Heek, Kefan Xiao, Shivani Agrawal, and Jeff Dean. 2022. Efficiently Scaling Transformer Inference. arXiv:2211.05102 [cs.LG]

[82] Aurick Qiao, Sang Keun Choe, Suhas Jayaram Subramanya, Willie Neiswanger, Qirong Ho, Hao Zhang, Gregory R. Ganger, and Eric P. Xing. 2021. Pollux: Co-adaptive Cluster Scheduling for Goodput-Optimized Deep Learning. In 15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21).

[83] Minsoo Rhu, Natalia Gimelshein, Jason Clemons, Arslan Zulfiqar, and Stephen W. Keckler. 2016. VDNN: Virtualized Deep Neural Networks for Scalable, Memory-Efficient Neural Network Design. In The 49th Annual IEEE/ACM International Symposium on Microarchitecture (Taipei, Taiwan) (MICRO-49). IEEE Press, Article 18, 13 pages.

[84] Mark Sandler, Andrew G. Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. 2018. MobileNetV2: Inverted Residuals and Linear Bottlenecks. In 2018 IEEE Conference on Computer Vision and Pattern Recognition, CVPR.

[85] Amedeo Sapio, Marco Canini, Chen-Yu Ho, Jacob Nelson, Panos Kalnis, Changhoon Kim, Arvind Krishnamurthy, Masoud Moshref, Dan Ports, and Peter Richtarik. 2021. Scaling Distributed Machine Learning with In-Network Aggregation. In 18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21).

[86] Jaime Sevilla, Lennart Heim, Anson Ho, Tamay Besiroglu, Marius Hobbhahn, and Pablo Villalobos. 2022. Compute Trends Across Three Eras of Machine Learning. In 2022 International Joint Conference on Neural Networks (IJCNN).

[87] Mohammad Shahrad, Rodrigo Fonseca, Inigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. 2020. Serverless in the Wild: Characterizing and Optimizing the Serverless Workload at a Large Cloud Provider. In 2020 USENIX Annual Technical Conference (USENIX ATC 20).

[88] Christopher J. Shallue, Jaehoon Lee, Joseph M. Antognini, Jascha Sohl-Dickstein, Roy Frostig, and George E. Dahl. 2019. Measuring the Effects of Data Parallelism on Neural Network Training. J. Mach. Learn. Res. 20 (2019).

[89] 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.

[90] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurelien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. 2023. LLaMA: Open and Efficient Foundation Language Models. arXiv:2302.13971 [cs.CL]

[91] Taegeon Um, Byungsoo Oh, Byeongchan Seo, Minhyeok Kweun, Goeun Kim, and Woo-Yeon Lee. 2023. FastFlow: Accelerating Deep Learning Model Training with Smart Offloading of Input Data Pipeline. Proc. VLDB Endow. 16, 5 (2023).

[92] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is All You Need. In Proceedings of the 31st International Conference on Neural Information Processing Systems (NIPS'17).

[93] Stavros Volos, Kapil Vaswani, and Rodrigo Bruno. 2018. Graviton: Trusted Execution Environments on GPUs. In 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18).

[94] Guanhua Wang, Kehan Wang, Kenan Jiang, XIANGJUN LI, and Ion Stoica. 2021. Wavelet: Efficient DNN Training with Tick-Tock Scheduling. In Proceedings of Machine Learning and Systems, Vol. 3. 696–710. https://proceedings.mlsys.org/paper/2021/file/c81e728d9d4c2f636f067f89cc14862c-Paper.pdf

[95] Shang Wang, Peiming Yang, Yuxuan Zheng, Xin Li, and Gennady Pekhimenko. 2021. Horizontally Fused Training Array: An Effective Hardware Utilization Squeezer for Training Novel Deep Learning Models. In Proceedings of Machine Learning and Systems (MLSys).

[96] Yuxin Wang, Qiang Wang, Shaohuai Shi, Xin He, Zhenheng Tang, Kaiyong Zhao, and Xiaowen Chu. 2020. Benchmarking the Performance and Energy Efficiency of AI Accelerators for AI Training. In 2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID).

[97] Zhuang Wang, Haibin Lin, Yibo Zhu, and T. S. Eugene Ng. 2023. Hi-Speed DNN Training with Espresso: Unleashing the Full Potential of Gradient Compression with Near-Optimal Usage Strategies. In Proceedings of the Eighteenth European Conference on Computer Systems, (EuroSys '23).

[98] Qizhen Weng, Wencong Xiao, Yinghao Yu, Wei Wang, Cheng Wang, Jian He, Yong Li, Liping Zhang, Wei Lin, and Yu Ding. 2022. MLaaS in the Wild: Workload Analysis and Scheduling in Large-Scale Heterogeneous GPU Clusters. In 19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22).

[99] Lukasz Wesolowski, Bilge Acun, Valentin Andrei, Adnan Aziz, Gisle Dankel, Christopher Gregg, Xiaoqiao Meng, Cyril Meurillon, Denis Sheahan, Lei Tian, Janet Yang, Peifeng Yu, and Kim Hazelwood. 2021. Datacenter-Scale Analysis and Optimization of GPU Machine Learning Workloads. IEEE Micro 41 (2021).

[100] 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).

[101] Carole-Jean Wu, Ramya Raghavendra, Udit Gupta, Bilge Acun, Newsha Ardalani, Kiwan Maeng, Gloria Chang, Fiona Aga, Jinshi Huang, Charles Bai, Michael Gschwind, Anurag Gupta, Myle Ott, Anastasia Melnikov, Salvatore Candido, David Brooks, Geeta Chauhan, Benjamin Lee, Hsien-Hsin Lee, Bugra Akyildiz, Maximilian Balandat, Joe Spisak, Ravi Jain, Mike Rabbat, and Kim Hazelwood. 2022. Sustainable AI: Environmental Implications, Challenges and Opportunities. In Proceedings of Machine Learning and Systems, Vol. 4. 795–813.

[102] Wencong Xiao, Romil Bhardwaj, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, Zhenhua Han, Pratyush Patel, Xuan Peng, Hanyu Zhao, Quanlu Zhang, Fan Yang, and Lidong Zhou. 2018. Gandiva: Introspective Cluster Scheduling for Deep Learning. In 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18).

[103] 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).

[104] Peifeng Yu and Mosharaf Chowdhury. 2020. Fine-Grained GPU Sharing Primitives for Deep Learning Applications. In Proceedings of Machine Learning and Systems, Vol. 2. 98–111.

[105] Hong Zhang, Yupeng Tang, Anurag Khandelwal, and Ion Stoica. 2023. SHEPHERD: Serving DNNs in the Wild. In 20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23).

[106] 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).

[107] Zhen Zhang, Chaokun Chang, Haibin Lin, Yida Wang, Raman Arora, and Xin Jin. 2020. Is Network the Bottleneck of Distributed Training?. In Proceedings of the Workshop on Network Meets AI ML (NetAI '20).

[108] Mark Zhao, Niket Agarwal, Aarti Basant, Buğra Gedik, Satadru Pan, Mustafa Ozdal, Rakesh Komuravelli, Jerry Pan, Tianshu Bao, Haowei Lu, Sundaram Narayanan, Jack Langman, Kevin Wilfong, Harsha Rastogi, Carole-Jean Wu, Christos Kozyrakis, and Parik Pol. 2022. Understanding Data Storage and Ingestion for Large-Scale Deep Recommendation Model Training: Industrial Product. In Proceedings of the 49th Annual International Symposium on Computer Architecture (ISCA '22).

A Artifact Appendix

A.1 Abstract

The artifact consists of the source code of Orion[5], benchmarks for deployment and evaluation, scripts and instructions for characterizing the resource requirements of the GPU kernels for various DNN workloads[6], as well as a Docker image containing all the prerequisites for installing and testing Orion[7].

The evaluation focuses on reproducing key results from the paper, which demonstrate the key benefits of Orion's scheduling mechanism over the baselines. We pick 2 experiments (inference-training and inference-inference). In order to reduce the time and cost of the experiments, we evaluate only 2 high-priority workloads (ResNet50 and MobileNetV2), and focus only on the most competitive baselines (REEF and MPS).

[5] https://github.com/eth-easl/orion

[6] https://github.com/eth-easl/orion/tree/main/profiling

[7] https://hub.docker.com/repository/docker/fotstrt/orion-ae/general

- **Inference-Training** (Figures 7a and 7b): A high-priority inference workload is colocated with a best-effort training workload.
- **Inference-Inference** (Figure 10): Both the high-priority and best-effort clients run inference workloads.

A.2 Description & Requirements

A.2.1 How to access. The artifact is available in https://github.com/eth-easl/orion/tree/main/artifact_evaluation.

The DOI for the artifact evaluation is https://zenodo.org/records/10084464.

A.2.2 Hardware dependencies. The artifact has been tested on a GCP VM with the following specifications:

- **n1-standard-8 type (8 vCPUs, 30 GB DRAM)**
- **1 V100-16GB GPU**

We have set up a Google Cloud Platform (GCP) Project for VM creation and artifact evaluation. We have created a GCP VM image with the NVIDIA drivers installed, to allow for faster deployment. We encourage the reviewers to create a Google Cloud account (with an anonymous email) and reach out to us to be added to the GCP project for conducting experiments.

A.2.3 Software dependencies.

- Ubuntu 18.04
- CMake 3.19
- CUDA 10.2
- CUDNN 7.6.5
- NVIDIA DRIVER version 510.47
- Python >= 3.8
- PyTorch 1.12 (installed from source, fully installed in our custom docker image)
- TorchVision 0.13

We have set up a Docker Image with all the software dependencies pre-installed. We encourage reviewers to deploy and evaluate Orion using this image, as described in our README.

A.2.4 Benchmarks. We run the following models (inference + training scripts):

- ResNet50, MobileNet-V2, ResNet101 from Torchvision [28].
- BERT and Transformer from the NVIDIA repo [22]

All the scripts and their dependencies are included in our Docker Image.

A.3 Set-up

See our README for detailed instructions. The main steps include:

1. Start a Google Cloud VM (using our custom VM image)
2. Connect to the VM
3. Start Orion container
4. Clone Orion repo and install

A.4 Evaluation workflow

A.4.1 Major Claims. In the paper, we show Orion's ability to maintain the performance of the high-priority job, while colocating with best-effort jobs.

- (C1): In the case of high-priority inference jobs colocated with best-effort training jobs, Orion keeps the p95 and p99 latency of the high-priority inference job within 12% and 14% of the ideal, respectively. This is proven by the experiment (E1) shown in section 6 and Figure 7.
- In the case of high-priority inference jobs colocated with best-effort inference jobs, Orion keeps the p95 latency and p99 latency of the high-priority job within 18% and 22% of the ideal, respectively. This is proven by the experiment (E2) shown in section 6 and Figure 11.

Furthermore, in both cases, we see that Orion maintains the tail latency while varying the colocated workloads, while REEF and MPS cause great variability across the experiments.

A.4.2 Experiments. *Experiment (E1): [Inference-Training] [$\tilde{1}$2 GPU-hours] In this experiment, the high-priority inference job submits requests following the Poisson distribution with requests per second as shown in table 3. The best-effort jobs are training workloads.:*

We provide the steps for E1 in our README.

Experiment (E2): [Inference-Inference] [$\tilde{8}$ GPU-hours]: In this experiment, both jobs run inference workloads. The high-priority jobs submit requests from the Apollo trace [36], while the best-effort jobs follow the uniform distribution with rps as shown in table 3.

We provide the steps for E2 in our README.