Salus: Fine-Grained GPU Sharing Primitives for Deep Learning Applications

Peifeng Yu
peifeng@umich.edu
University of Michigan

Mosharaf Chowdhury
mosharaf@umich.edu
University of Michigan

Abstract

GPU computing is becoming increasingly more popular with the proliferation of deep learning (DL) applications. However, unlike traditional resources such as CPU or the network, modern GPUs do not natively support fine-grained sharing primitives. Consequently, implementing common policies such as time sharing and preemption are expensive. Worse, when a DL application cannot completely use a GPU's resources, the GPU cannot be efficiently shared between multiple applications, leading to GPU underutilization.

We present Salus to enable two GPU sharing primitives: *fast job switching* and *memory sharing*, in order to achieve fine-grained GPU sharing among multiple DL applications. Salus implements an efficient, consolidated execution service that exposes the GPU to different DL applications, and enforces fine-grained sharing by performing iteration scheduling and addressing associated memory management issues. We show that these primitives can then be used to implement flexible sharing policies such as fairness, prioritization, and packing for various use cases. Our integration of Salus with TensorFlow and evaluation on popular DL jobs show that Salus can improve the average completion time of DL training jobs by 3.19×, GPU utilization for hyper-parameter tuning by 2.38×, and GPU utilization of DL inference applications by 42× over not sharing the GPU and 7× over NVIDIA MPS with small overhead.

1 Introduction

Deep learning (DL) has received ubiquitous adoption in recent years across many data-driven application domains, ranging from machine translation and image captioning to chat bots and personal assistants [35]. Consequently, both industry and academia are building DL solutions – e.g., TensorFlow [16], CNTK [52], Caffe2 [1], and others [11, 17, 19, 21, 23, 30, 49] – to enable both *training* of DL models using large datasets as well as serving DL models for *inference*.

GPUs have emerged as a popular choice in this context because they excel at highly parallelizable matrix operations common in DL jobs [9, 16, 31, 54]. Unfortunately, the minimum granularity of GPU allocation today is always the entire GPU – *an application can have multiple GPUs, but each GPU can only be allocated to exactly one application* [5, 10, 13, 14]. While such exclusiveness in accessing a GPU simplifies the hardware design and makes it efficient in the first place, it leads to two major inefficiencies.

First, the coarse-grained, one-at-a-time GPU allocation model hinders the scheduling ability of GPU cluster managers [3, 4, 10, 29, 48]. For flexible scheduling, a cluster manager often has to suspend and resume jobs (i.e., preempt), or even migrate a job to a different host. However, a running DL job must be fully purged from the GPU before another one can start, incurring large performance overhead. As such, GPU clusters often employ non-preemptive scheduling, such as FIFO [4], which is susceptible to the head-of-line (HOL) blocking problem.

Second, not all DL jobs can fully utilize a GPU all the time (§2). On the one hand, DL training jobs are usually considered resource intensive. But for memory-intensive ones (e.g., with large batch sizes), our analysis shows that the average GPU memory utilization is often less than 50% (§2.2 Figure 1) due to varied memory usage over time and between iterations. Similar pattern can also be observed in compute-intensive training jobs. DL model serving also calls for finer-grained GPU sharing and packing. Because the request rate varies temporally within the day as well as across models, the ability to hold many DL models on the same GPU when request rates are low can significantly cut the cost by decreasing the number of GPUs needed in serving clusters [8, 22].

Additionally, the increasingly popular trend of automatic hyper-parameter tuning of DL models (e.g., AutoML [18, 36, 42]) further emphasizes the need to improve GPU utilization. This can be viewed as "pre-training". It is usually done by generating many training jobs in parallel for hyper-parameter exploration, many of which are killed as soon as they are deemed to be of poor quality. Improved GPU utilization by spatiotemporal packing of many of these jobs together results in shorter makespan, which is desirable because of the all-or-nothing property of hyper-parameter exploration jobs – i.e., the result is useful only after all exploration jobs finish.

To address these issues, we present Salus to enable fine-grained sharing of individual GPUs with flexible scheduling policies among co-existing, unmodified DL applications. While simply sharing a GPU may be achievable, doing so in an efficient manner is not trivial (§2.3). Salus achieves this by exposing two GPU sharing primitives: *fast job switching* and *memory sharing* (§3). The former ensures that we can quickly switch the current active DL job on a GPU, enabling efficient time sharing and preemption. The latter ensures high utilization by packing more small DL jobs on the same device. The unique memory usage pattern of DL applications is the key to why such primitives can be efficiently implemented in Salus: we identify three different memory usage types and apply different management policies when handling them (§3.2). Combining these two primitives together, the fine-grained spatiotemporal sharing can be used to implement a variety of solutions (§4).

We have integrated Salus with TensorFlow and evaluated it on a collection DL workload consisting of popular DL models (§5). Our results show that Salus improves the average completion time of DL training jobs by 3.19× by efficiently implementing the shortest-remaining-time-first (SRTF) scheduling policy to avoid HOL blocking. In addition, Salus shows 2.38× improvement on GPU utilization for the hyper-parameter tuning workload, and 42× over not sharing the GPU and 7× over NVIDIA MPS for DL inference applications with small overhead.

2 Background and Motivation

This section overviews DL jobs' structural characteristics (§2.1) and analyzes common DL workloads to understand their resource usage patterns and opportunities for GPU sharing (§2.2). Later we discuss existing techniques for GPU sharing among them (§2.3).

2.1 Deep Learning

Deep learning (DL) is a class of algorithms that use a stack of nonlinear processing layers to solve common machine learning tasks, e.g., classification, clustering, or prediction [35]. A particular layout of such layers forms a *network architecture*, or simply *network*, that is specially designed for domain-specific problems. DL networks must be *trained* before they can be deployed for any practical use. The knowledge gained from the training process is saved to *model parameters*, which are used in addition to input data for the network to compute the final result. Collectively, the network architecture and the model parameters are called a DL *model*. Later, the learned model is used to serve *inference* requests.

Forward and Backward Computation During inference, the input is propagated through each layer in order to gain the final result. This constitutes a *forward pass*. During training, an additional *backward pass* is performed, propagating the gradients back while updating model parameters. DL training proceeds by alternating between forward and backward passes in an iterative fashion. The backward pass is often more expensive than the forward one, because it requires more resources to keep all intermediate results produced between layers to compute gradients (§2.2). Both typically involve a large number of matrix operations, leading to the rising popularity of GPU usage in DL workloads.

2.2 DL Workloads Characteristics

To understand the resource usage patterns of DL jobs, we analyzed a workload consisting of 15 DL models (Table 3 in Appendix). The CNNs are from the official TensorFlow CNN benchmarks [12]; others are selected popular models in respective fields.

In order to cover a wider range of use cases, while keeping the native input characteristics, we varied the batch size to create 45 distinct workloads, as shown in Table 3. Note that the batch size specifies the number of inputs (e.g., images for CNNs) trained in each iteration and also affects the size of model parameters. Thus it has an impact on the time each iteration takes to complete as well as the memory footprint of the model. Throughout the paper, we uniquely identify a workload by the network name plus the input batch size. For example, `alexnet_25` means a job training `alexnet`, with a batch size of 25.

In terms of GPU resource usage, one can consider two high-level resources: GPU computation resources (primarily in terms of computation time, often referred to as GPU utilization in the literture) and GPU memory. We found that both are often correlated with the complexity of the DL model. However, GPU memory is especially important because *the entire DL model and its associated data must reside in memory for the GPU to perform any computation*; in contrast, computations can be staggered over time given sufficient GPU memory.

In the following, we highlight several key characteristics of GPU memory usage in DL workloads that highlight GPU memory underutilization issues and/or opportunities for improvements.

Heterogeneous Peak Memory Usage Across Jobs DL workloads are known for heavy memory usage [16, 20, 37]. Figure 1 visualizes the average and peak memory usages of our workloads. As networks become larger (with more and wider layers) and the batch size increases, memory requirements of DL jobs increase as well. For example, we observed peak memory usages as high as 13.8 GB for `resnet152` and as low as less than 1 GB for `vae`. Such high variations suggest that even during peak allocation periods, it may be possible to run multiple networks on the same GPU instead of running networks in a FIFO manner.

Temporal Memory Usage Variations Within a Job Within each job, however, each iteration of a DL training job

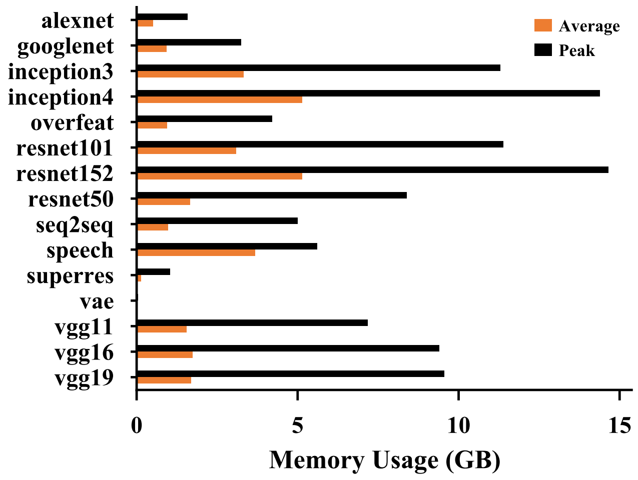


Figure 1: Average and peak GPU memory usage per workload, measured in TensorFlow and running on NVIDIA P100 with 16 GB memory. The average and peak usage for vae is 22 MB, 35 MB, which are too small to show in the figure. The appendix also includes the measurement in PyTorch (Figure 16), which shares a similar pattern.

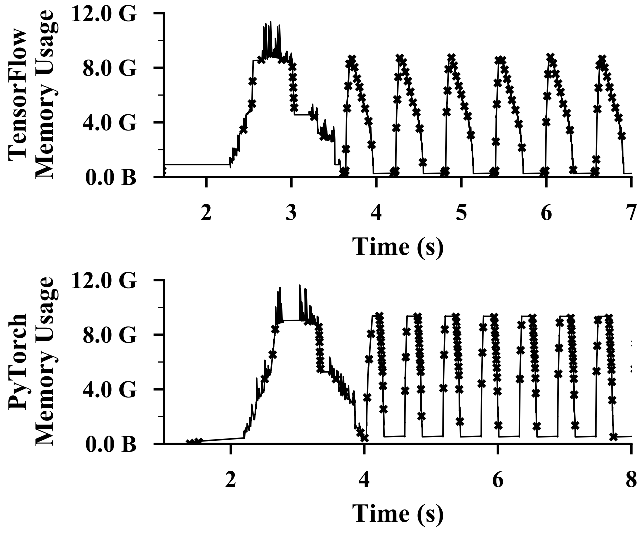


Figure 2: Part of the GPU memory usage trace showing the spatiotemporal pattern when training `resnet101_75` and running on NVIDIA P100 with 16 GB memory, using TensorFlow and PyTorch.

is highly predictable with a well-defined peak memory usage and a trough in between iterations. Figure 2 shows an example. This is because DL jobs go through the same sequence of operations and memory allocations in each iteration. The presence of predictable peaks and troughs can help us identify scheduler invocation points.

Low Persistent Memory Usage Another important characteristic of GPU memory usage of DL jobs is the use of persistent memory to hold the model of a network – this corresponds to the consistent troughs across iterations. Even though the peak usage can be very high, most of it is temporary data created and destroyed within the same iteration. Fortunately, the size of persistent memory is often very low in comparison to the temporary data, ranging from 110.9 MB for `googlenet_25` to 822.2 MB for `resnet152_75`. *As long as the model is already in GPU memory, we can quickly start an iteration of that network.* This gives us an additional opportunity to improve sharing and utilization.

2.3 Existing Techniques for Sharing GPUs

Given that DL workloads leave ample room for GPU sharing, a straw man approach would be disabling the exclusive access mode and statically partitioning (SP) the GPU memory among multiple applications. This cannot completely address the underutilization problem due to high peak-to-average memory usage of DL jobs. Moreover, static partitioning has significant slowdown compared to the exclusive mode.

NVIDIA's Multi-Process Service (MPS) [2] is the official way to achieve GPU sharing. Although users still have to use static partitioning of the GPU memory for each concurrently running job, the performance is better. Nonetheless, MPS has limited support for DL frameworks: not all DL framework versions are supported according to our experiments and bug reports on various DL frameworks [5, 13, 14]. It is possible to achieve GPU memory overcommitting with Unified Memory Access (UMA) [34], but it performs poorly due to paging between GPU and the system memory.

A recent work, Gandiva [50], aims to improve latency and efficiency of DL training by coarse-grained time slicing (e.g., minutes-long slices with about a second switching delay) and static memory partitioning.

NVIDIA's TensorRT Inference server [8] achieves simultaneous DL inference in parallel on one GPU using GPU streams [38]. But it lacks scheduling ability and does not support DL training.

Prior works on fine-grained GPU sharing fall into several categories. Some attempt to intercept GPU calls – CUDA calls in the case of NVIDIA GPUs – and dynamically introduce concurrency by time-slicing kernel execution at runtime [39, 40, 43]. Unfortunately, they are either limited to optimizing the efficiency for a single job without considering overall resource utilization or require extensive changes to the underlying GPU architecture. Others call for new APIs for GPU programming [47, 51, 53] but require rewriting existing applications. To summarize, these solutions are designed for jobs with a few GPU kernels; as such, they are not scalable to DL applications, where the number of unique kernels can easily goes up to several hundreds.

Table 1 summarizes the aforementioned approaches and a set of desirable features for an ideal solution.

Approach	DL Support	Efficiency	Fast Switching	Flexible Scheduling
Non DL approaches	No	-	-	-
SP	Yes	No	No	No
SP + MPS	Partial	Yes	Yes	No
SP + MPS + UMA	Partial	No	Yes	Yes
Gandiva	Yes	Yes	No	No
TensorRT	Yes	Yes	Yes	No
Salus	Yes	Yes	Yes	Yes

Table 1: Comparison of GPU sharing approaches (§2.3).

3 Salus

Salus[1] is our attempt to build an ideal solution to GPU sharing. It is designed to enable efficient, fine-grained GPU sharing while maintaining compatibility with existing frameworks (§3.1). Its overall design is guided by the unique memory usage characteristics of DL jobs. While existing DL frameworks are limited by the job-exclusive GPU usage scenario, packing multiple jobs onto one GPU changes the combined memory allocation patterns and special care must be taken to mitigate increased fragmentation. Salus addresses both temporal and spatial aspects of the memory management problem by enabling two GPU sharing primitives:

1. Fine-grained time sharing via *efficient job switching* among ongoing DL jobs (§3.2);
2. Dynamic memory sharing via the *GPU lane* abstraction (§3.3).

Together, these primitives open up new scheduling and resource sharing opportunities. Instead of submitting one job at a time, which can easily lead to HOL blocking, one can perform preemption or run multiple DL jobs in a time- or space-shared manner – all of which can be utilized by a GPU cluster scheduler [50]. We demonstrate the possibilities by implementing common scheduling policies such as preempting jobs to implement shortest-remaining-time-first (SRTF), performing fair sharing between jobs, and packing many jobs in a single GPU to increase its utilization (§4).

3.1 Architectural Overview

At the highest level, Salus is implemented as a singleton *execution service*, which consolidates all GPU accesses, thus enabling GPU sharing while avoiding costly context switch among processes on the GPU. As a result, any unmodified DL job can leverage Salus using a DL framework-specific *adaptor* (Figure 3).

From a framework's point of view, the adaptor abstracts away low level details, and Salus can be viewed as another (virtual) computation device.

From a user's perspective, the API of the framework does

[1] We have open sourced the system at https://github.com/SymbioticLab/Salus

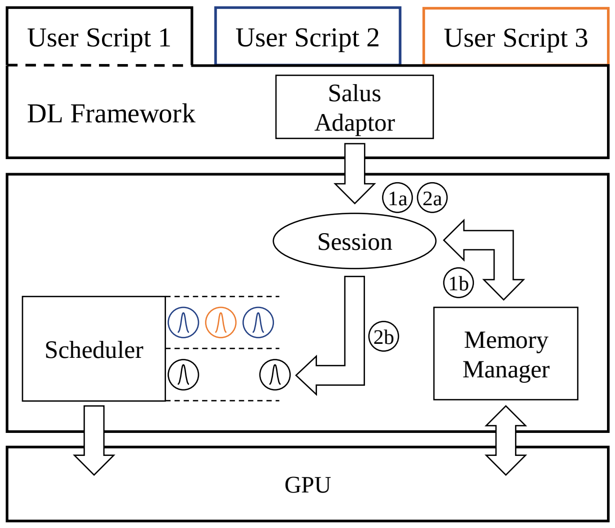


Figure 3: Salus sits in between DL frameworks and the hardware in the DL stack, being transparent to users.

not change at all. All their scripts will work the same as they did before.

It is perhaps better to explain the architecture via an example of the life cycle of a DL job. When a DL job is created in an user script, *Salus adaptor* in the DL framework creates a corresponding session in Salus (1a). The computation graph of the DL job is also transferred to Salus during the creation.

The session then proceeds to request a lane from the *memory manager* (1b). Depending on current jobs in the system, this process can block and the session will be queued (§3.3).

During the job's runtime, either training or inferencing, iterations are generated by the user script and forwarded to the corresponding session in Salus (2a). They are then scheduled according to their associated GPU lanes by the iteration scheduler (2b), and send to GPU for execution.

The Salus execution service thus achieves GPU sharing via iteration-granularity scheduling of DL jobs. We will elaborate on a performance-efficiency tradeoff in choosing this granularity (§3.2.2)

3.2 Efficient Job Switching

The ability to switch between jobs is paramount to implement time sharing and preemption – two techniques extensively used by modern schedulers in many contexts. Suspending a running job and resuming the same or another one have always been possible on GPU as well. Modern DL frameworks extensively use checkpointing to mitigate data and computation loss due to the long running nature of DL training jobs. The same technique is applied by Gandiva [50] to achieve second-scale suspend/resume. Nevertheless, checkpointing can result in large data transfers from and to the GPU memory, even in the best case when only model parameters are transfered, the communication time is still

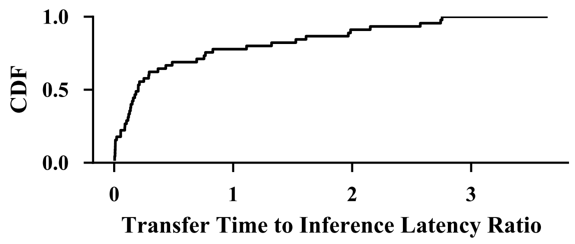


Figure 4: Theoretical minimal transfer time compared with model inference latency. Data collected from our workloads and transfer time is calculated using 30 GB/s for transfer speed.

non-negligible. It even becomes unacceptable if the system ever wants to support inference workloads: the theoretical minimal transfer time can be even several times longer than the inference latency itself, according to the measurement on our collection of workloads (Figure 4).

Observation 1 *Transferring GPU memory back and forth is not practical to achieve low latency given current GPU communication bandwidth.*

3.2.1 Characterizing DL Memory Allocations

We observe that one can push things further by taking a close look at different types of memory allocations in a DL job. Specifically, we define three types of memory allocations with unique characteristics.

1. *Model:* These mostly hold model parameters and typically consist of a few large chunks of memory. They are persistent because they have to be available throughout the whole job's lifetime. Because the model size is typically fixed during the entire training process, model data has little or no temporal variations and is predictable.
2. *Ephemeral:* These are the scratch memory needed during each iteration. These memory usually hold intermediate layers' outputs as well as temporary data generated by the algorithm itself. They are only needed during computations and are released between iterations, giving rise to the temporal memory usage patterns of DL jobs. They are often large memory allocations as well.
3. *Framework-internal:* These are usually used by the DL framework for book-keeping or for data preparation pipeline. They often persist across iterations.

Collectively, model and framework-internal memory are *persistent* across iterations. As an example, Figure 5 gives the memory allocation size distribution for a popular CNN workload: `inception3_50`.

Observation 2 *There is significantly less persistent memory usage than ephemeral memory for a DL job. It is possible to keep more than one job's persistent memory in GPU while still having enough space for either one's ephemeral memory.*

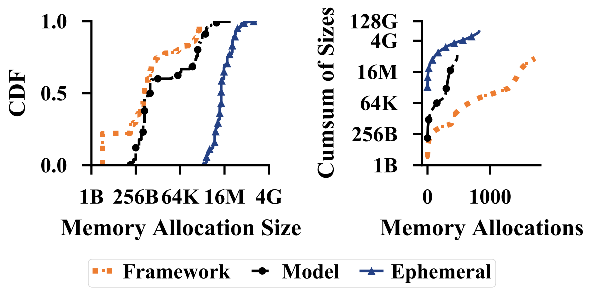


Figure 5: Memory allocation size and number distribution for ephemeral, model and framework-internal memory, using `inception3_50` as an example.

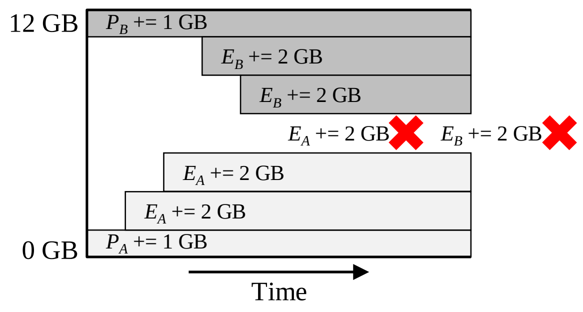


Figure 6: Deadlock from progressive memory allocations.

The above two observations naturally lead to the conclusion that fast job switching can be enabled by not removing persistent memory from GPU at all. Thus unlike existing works [50], Salus is designed to enable significantly faster suspend/resume operations by keeping persistent memory around, and then an iteration-granularity job scheduler (e.g., time-sharing or preemption-based) decides which job's iteration should be run next.

3.2.2 Scheduling Granularity

Given that iterations are typically short in DL jobs (ranging from tens of milliseconds to a few seconds), with an even finer granularity, e.g., at the GPU kernel level, it may be possible to further utilize GPU resources. However, finer-grained scheduling also adds more overhead to the execution service. Indeed, there is a tradeoff between maximum utilization and efficiency for a given scheduling granularity.

To understand this tradeoff, we prototyped a GPU kernel-level switching mechanism as well only to find that scheduling at that level incurs too much overhead for little gain. It requires all GPU kernels to go through a central scheduler, which, in addition to becoming a single bottleneck, breaks common efficiency optimizations in DL frameworks such as kernel batching and pipelining.

To make things worse, a unique deadlock issue arises due to the progressive memory allocations performed by many DL frameworks: a job can start its iteration as long as its model memory is available, and then the ephemeral memory is allocated gradually by a series of GPU kernels. Now consider the following scenario with 12 GB GPU memory capacity, and two iterations from jobs A and B. Their model memory usages are $P_A = P_B = 1$ GB and ephemeral memory usages are $E_A = E_B = 7$ GB (we are ignoring framework-internal usage because of its relatively smaller size). Instead of allocating all 8 GB at once, each iteration of a job allocates in different increments. For example, consider a possible allocation order shown in Figure 6, where (Y_X += N GB) refers to job X allocating N GB of type Y memory. After a few rounds of successful allocations, if both jobs attempt to allocate their remaining requirements as follows: (E_A += 3 GB) and (E_B += 3 GB), neither will be able to proceed, causing a deadlock! Mitigating the deadlock would have been simple if GPUs provided program controlable fast paging mechanisms, which unfortunately is not the case today.

In contrast, our choice of switching in between iterations allows us to sidestep the problems of progressive memory allocations. This is because all ephemeral allocations are released by the framework after each iteration, and model and framework-internal allocations remain constant across iterations.

3.3 Memory Sharing via GPU Lane

Although DL jobs' memory usages have spatiotemporal variations, many cannot reach the total capacity of a GPU's memory. Naturally, we must consider ways to better utilize the unused memory.

Built on top of the efficient job switching, we design a special memory layout scheme, the *GPU Lane*, that achieves memory sharing and improves memory utilization.

First of all, learning from classic memory management techniques of stack and heap to seperate dynamic allocations from static ones, we divide GPU memory space into *ephemeral* and *persistent* regions, growing from both end of the memory space (Figure 7). A DL job's model and framework-internal memory is allocated in the persistent region, while its ephemeral memory goes into, obviously, the ephemeral region.

The ephemeral region is further divided into *lanes*, which are continuous memory spaces that can contain ephemeral memory allocation for iterations. Lanes are not only about memory, though. Iteration execution is serialized within a lane and parallelism is achieved across lanes, which is implemented using GPU streams. Each lane can be assigned to multiple DL jobs, in which case efficient job switching primitive discussed in previous section is used to time share the lane.

The lane's restriction on execution is necessary because ephemeral memory differs from the other two types of memory in terms of allocation patterns and timing. As a result, simply putting allocations from two different iterations together can cause deadlock as already discussed if the memory is oversubscribed.

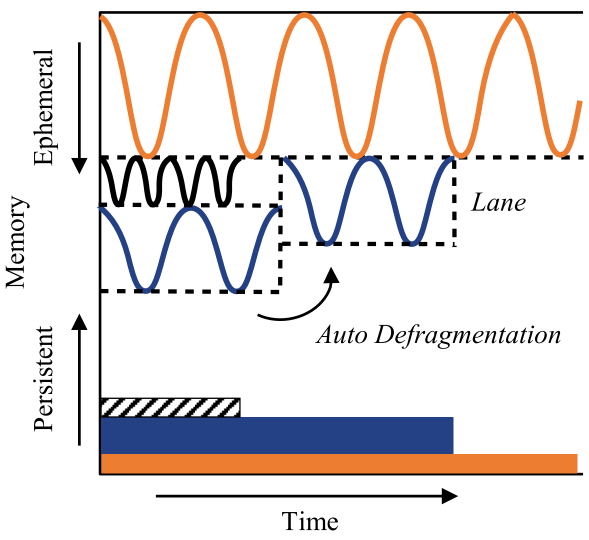


Figure 7: GPU memory is divided into stack and heap, with auto defragmentation at iteration boundaries.

Even if enough memory is ensured for both peak memory usage for two iterations, memory fragmentation can still cause superfluous out-of-memory errors if not handled. More specifically, while the framework-internal memory allocations are small in size, they can have a large impact on the overall memory layout and may create more memory fragments when multiple iterations are allocating simultaneously. While there are works implementing a memory planner before actually starting the iteration [19], they are not available to all frameworks.

We approach the problem by first implementing an application-aware bin-based allocator to mitigate the fragmentation. However, it breaks memory optimizations commonly used in DL frameworks because they assume single job running at a time. Since our goal is to fully support existing workloads with minimal impact on the user, we choose to limit the dynamic allocation in the ephemeral region and isolate memory allocations across lanes to ensure maximum compatibility while achieving adequate flexibility.

3.3.1 Lane Auto Defragmentation

Having lanes does not eliminate memory fragmentation, it moves fragmentation within lane to fragmentation at the lane level. However, defragmentation is much easier at this level. Traditionally, defragmentation is achieved by first moving data out of memory and later moving it back again. In case of lanes, the allocations are released completely at the end of each iteration and goes back at the start of next iteration – they are ephemeral memory after all. Therefore, defragmentation happens almost automatically at no cost: no extra memory movement is needed.

Consider the situation illustrated in Figure 7, when the small job stops, its lane space is quickly reclaimed at the iteration boundary by the job that was allocated below it.

3.3.2 Lane Assignment

It is vital to determine the size and number of lanes in the GPU, as well as how lanes are assigned to jobs. Salus uses a simple yet efficient algorithm to decide between opening a new lane and putting jobs into existing lanes.

Throughout the process, the following "safety" condition is always kept to make sure the persistent region and ephemeral region do not collide into each other:

$$\sum_{jobs} P_i + \sum_{lanes} L_j \leq C$$

$$L_j = \max_{i \text{ in } j} E_i$$

where P_i and E_i are respectively the persistent (model and framework-internal) and ephemeral memory usage of job i. L_j is the lane size of lane j, which is again defined as the maximum ephemeral memory usage of all jobs in the lane. C is the capacity of the GPU.

By ensuring enough capacity for persistent memory of all the admitted jobs and enough remaining for the iteration with the largest temporary memory requirement, Salus increases the utilization while making sure that at least one job in the lane can proceed.

At the highest level, the algorithm tries to obtain a lane in the following order, returning once a suitable lane is found and safety condition is met:

- Open a new lane
- Use an existing lane
- Reorganize lane assignments to existing jobs to reduce the size of ephemeral region

As shown in Algorithm 1, the system is event-driven and reacts when there are jobs arriving or finishing, or at iteration boundaries when auto defragmentation happens.

How to reorganize lane assignments is an open question. We find the one implemented in our algorithm works fairly well in practice, but there are more possibilities about finding the optimal number of lanes given a set of jobs.

4 Scheduling Policies in Salus

The state-of-the-art for running multiple DL jobs on a single GPU is simply FIFO – regardless of the DL framework [5, 13, 14] – that can lead to HOL blocking. Although Gandiva [50] recently proposed a time sharing approach, it enforces sharing over many minutes because of high switching overhead. It uses MPS for memory sharing with admittedly unpredictable performance.

By enabling fine-grained GPU sharing primitives, Salus makes it possible to pack multiple jobs together to increase efficiency, to preempt long-running jobs in favor of shorter ones (or based on other priority criteria), and many others,

Algorithm 1 GPU Lane Assignment

```
1:  $Q \leftarrow \emptyset$                    ▷ The pending queue for new jobs
2:  procedure JOBARRIVE($P, T$)
3:    $P$: new job's persistent memory requirement
4:    $E$: new job's ephemeral memory requirement
5:    $Q \leftarrow Q \cup \{(P, E)\}$
6:    PROCESSREQUESTS($Q$)

7:  procedure JOBFINISH($lane$)
8:    $lane$: the lane that the finished job assigned to
9:    $\text{ref}(lane) \leftarrow \text{ref}(lane) - 1$
10:   if $\text{ref}(lane) == 0$ then
11:     Delete $lane$
12:     PROCESSREQUESTS($Q$)

    ▷ After a lane is moved due to auto defragmentation
13: procedure LANEMOVED
14:   PROCESSREQUESTS($Q$)

15: procedure PROCESSREQUESTS($Q$)
16:   for all $(P, E) \in Q$ do
17:     $lane \leftarrow$ FINDLANE($P, E$)
18:     if Found $lane$ then
19:       $\text{ref}(lane) \leftarrow \text{ref}(lane) + 1$
20:       Assign $lane$ to the corresponding job

21: procedure FINDLANE($P, E$)
22:   $C$: size of total capacity
23:   $P_i$: persistent memory usage of existing job $i$
24:   $L_j$: lane size of existing lane $j$
25:   $\mathbb{L}$: set of existing lanes
      ▷ Try to create a new lane
26:   if $\sum_i P_i + P + \sum_j L_j + E \leq C$ then
27:     $lane \leftarrow$ new GPU lane with capacity $E$
28:     return $lane$
      ▷ Try to put into an existing lane
29:   for all $j \in \mathbb{L}$ do
30:     if $L_j \geq E$ and is the best match then
31:       return $j$
      ▷ Try to replace an existing lane
32:   for $r \in \mathbb{L}$ in $L_r$ ascending order do
33:     if $\sum_i P_i + P + \sum_j L_j - L_r + E \leq C$ then
34:       $L_r \leftarrow E$
35:       return $r$
36:   return not found
```

opening up a huge design space that can be explored in future works.

To demonstrate the possibilities, in our current work, we have implemented some simple scheduling policies, with Salus specific constrains (i.e., safety condition). The PACK policy aims to improve resource utilization and thus makespan, the SRTF policy is an implementation of shortest-remaining-time-first (SRTF), and the FAIR policy tries to equalize resource shares of concurrent jobs.

4.1 PACK to Maximize Efficiency

To achieve higher utilization of GPU resources, many jobs with different GPU memory requirements can be packed together in separate GPU lanes based on their peak memory usages. However, packing too many lanes exceeding the GPU memory capacity will either crash the jobs or incur costly paging overhead (if UMA is enabled), both of which would do more harm than good.

Consequently, this policy works with "safety" condition to ensure that the total peak memory usage across all lanes is smaller than the GPU memory capacity. Because each lane has guaranteed resources, there is no fairness consideration in this case.

Apart from training many different jobs or many hyper-parameter searching jobs in parallel, this can also enable highly efficient inference serving. By simultaneously holding many models in the same GPU's memory, we can significantly decrease the GPU requirements of model serving systems like Clipper [22].

4.2 SRTF to Enable Prioritization

Developing DL models are often an interactive, trial-and-error process where practitioners go through multiple iterations before finding a good network. Instead of waiting for an on-going large training to finish, Salus can enable preemption – the large job is paused – to let the smaller one finish faster on the same GPU lane. In this way, Salus can support job priorities based on arbitrary criteria, including based on size and/or duration to implement the shortest-remaining-time-first (SRTF) policy. The higher priority job is admitted as long as its own safety condition is met – i.e., at least, it can run alone on the GPU – regardless of other already-running jobs.

Note that we assume the job execution time is known and thus it is possible to implement SRTF. While there are works on how to estimate such job execution time [41], the subject is beyond the scope of this paper and we only focus on providing primitives to enable the implementation of such schedulers.

4.3 FAIR to Equalize Job Progress

Instead of increasing efficiency or decreasing the average completion time, one may want to time share between many DL jobs during high contention periods [50]. Note that there may be many different so-called *fair* algorithms based on time sharing; we demonstrate the feasibility of implementing one or more of them instead of proposing the optimal fairness policy. Specifically, we admit new jobs into the GPU lane while maintaining the safety condition, and equalize total service over time for jobs in each lane.

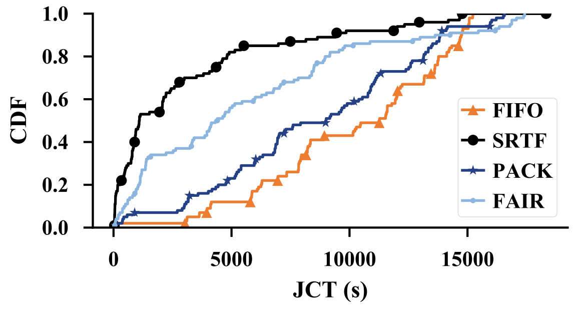


Figure 8: CDFs of JCTs for all four scheduling policies.

Scheduler	Makespan	Avg. Queuing	Avg. JCT	95% JCT
FIFO	303.4 min	167.6 min	170.6 min	251.1 min
SRTF	306.0 min	28.6 min	53.4 min	217.0 min
PACK	287.4 min	129.9 min	145.5 min	266.1 min
FAIR	301.6 min	58.5 min	96.6 min	281.2 min

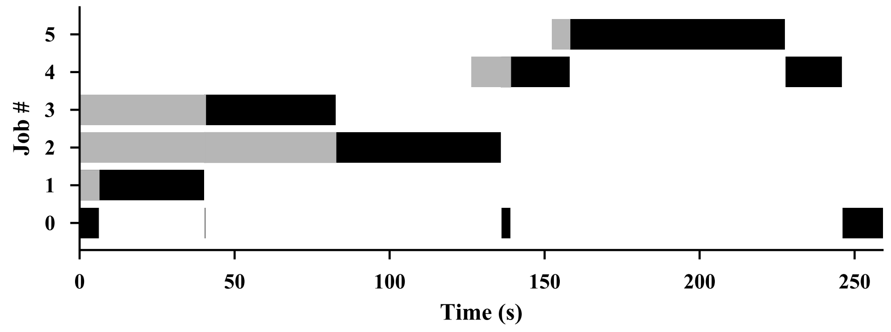
Table 2: Makespan and aggregate statistics for different schedulers.

5 Evaluation

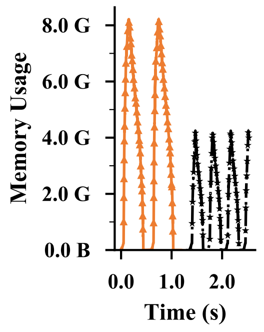
We have integrated Salus with TensorFlow and evaluated it using a collection of training, hyper-parameter tuning, and inference workloads [12, 28, 32, 45, 46] to understand its effectiveness and overhead. The highlights of our evaluation are as follows:

- Salus can be used to implement many popular scheduling algorithms. For example, the preemptive SRTF scheduler implemented in Salus can outperform FIFO by 3.19× in terms of the average completion time of DL training jobs (§5.1).
- Using Salus, one can run multiple DL jobs during hyper-parameter tuning stage, increasing GPU utilization by 2.38× (§5.2).
- Similarly, for inference, Salus can improve the overall GPU utilization by 42× over not sharing the GPU and 7× over NVIDIA MPS (§5.3).
- Salus has relatively small performance overhead given its flexibility and gains (§5.4).

Environment All experiments were done on a x86_64 based Intel Xeon E5-2670 machine with 2 NVIDIA Tesla P100 GPUs. Each GPU has 16GB on-chip memory. TensorFlow v1.5.0 and CUDA 8.0 are used in all cases.



(a) A slice of 6 jobs switching between each other. Gray areas represents the waiting between a job arrives and it actually gets to run.



(b) Memory usage during a job switching.

Figure 9: Details of a snapshot during the long trace running with SRTF. In both slices, time is normalized.

Baseline(s) Our primary baseline is the FIFO scheduling commonly used in today's GPU clusters [50]. We also compare against NVIDIA MPS.

5.1 Long-Running Training

First and foremost, we focus on Salus's impact on training. To this end, we evaluate Salus using a job trace of 100 workloads, generated using the jobs described in Table 3. We considered multiple batch sizes and durations of each training job in the mix. The overall distribution followed one found in a production cluster [50].

We compare four different schedulers:

1. **FIFO** refers to processing jobs in order of their arrival. This is the de facto mechanism in use today in the absense of Salus.
2. **SRTF** is a preemptive shortest-remaining-time-first scheduler. We assume that the duration is known or can be estimated using existing techniques [41].
3. **PACK** attempts to pack as many jobs as possible in to the GPU. The goal is to minimize the makespan.
4. **FAIR** uses time sharing to equally share the GPU time among many jobs.

5.1.1 Overall Comparison

Figure 8 presents the distributions of JCTs for all four policies, while Table 2 presents makespan and aggregate statistics. Given the similarities of makespan values between FIFO, SRTF, and FAIR, we can say that Salus introduces little overhead. Furthermore, packing jobs can indeed improve makespan. Note that because of online job arrivals, we do not observe large improvement from PACK in this case. However, when many jobs arrive together, PACK can indeed have a larger impact (§5.2).

These experiments also reestablishes the fact that in the presence of known completion times, SRTF can indeed improve the average JCT – 3.19× w.r.t. FIFO in this case.

5.1.2 Impact of Fast Job Switching

We evaluate Salus's ability to perform fast job switching in two contexts. First, we show that it can allow fast preemption, which, in turn, allows us to implement the shortest-remaining-time-first (SRTF) scheduling policy. Second, we show that how Salus can allow seconds-granularity fair sharing between multiple DL jobs – as opposed to minutes-granularity [50]. In both cases, we consider a single GPU lane.

SRTF Consider the following scenario: a large training job has been running for a while, then the user wants to quickly do some test runs for hyper-parameter tuning for smaller networks. Without Salus, they would have to wait until the large job finishing – this is an instance of HOL blocking. Salus enables preemption via efficient switching to run short jobs and resumes the larger job later.

We pick a segment in the long job trace, containing exact the scenario, and record its detailed execution trace, showing in Figure 9a. When job #1 arrives, the background job #0 is immediately stopped and Salus switches to run the newly arrived shorter job. Job #2 comes early than job #3, but since #3 is shorter, it is scheduled first. And finally since job #5 is shorter, #4 is preempted and let #5 run to completion. During the process, the background job #0 is only scheduled when there is no other shorter job existing.

Figure 9b is another example demonstrating Salus's ability to fast switch. It is the visualization of memory allocation activities in the scale of seconds: at the moment of a job switching, the second job's iteration starts immediately after the first job stops.

Time Sharing/Fairness Figure 10 is an snapshot of the job trace running under the FAIR policy. 4 training jobs: `googlenet_100`, `resnet50_25`, `alexnet_100` and `resnet50_50` are active during the snapshot, and Salus tries to equalize their GPU time. Again the switches all happen at sub-second granularity.

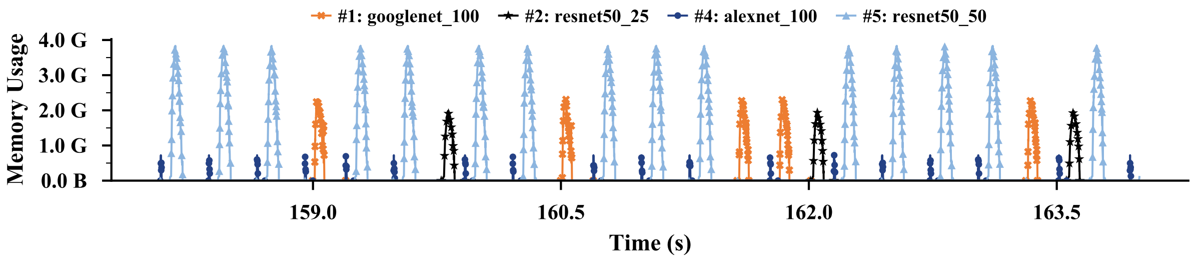


Figure 10: A slice of memory usage for the long trace using FAIR. Note that memory usage does not include actual computation; hence, the temporal gaps.

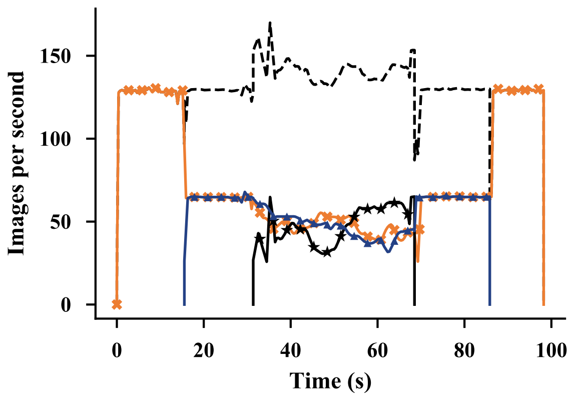


Figure 11: Fair sharing among three jobs, all training `inception3_50`. Black dashed line shows the overall throughput.

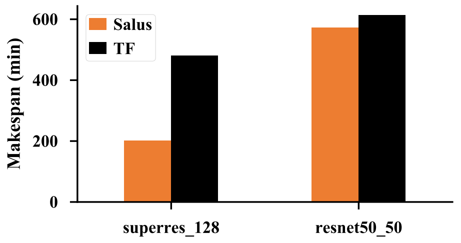


Figure 12: Makespan of two hyper-parameter tuning multi-jobs each of which consists of 300 individual jobs.

Note that the depicted memory usage is not a complete representation of the activity of that iteration. Computation actually continues to run in the gaps.

To better illustrate the impact of fairness, we show another microbenchmark, demonstrating Salus's ability to switch jobs efficiently using 3 training jobs and focusing on the fair sharing of GPU throughput in Figure 11.

For ease of exposition, we picked three jobs of the same DL model `inception3_50` – this allows us to compare and aggregate training throughput of the three models in terms of images processed per second. In this figure, in addition to the throughput of individual jobs, the black dashed line shows the aggregate throughput.

The training jobs start at time 0s, 15s and 30s. At 15s, when the second job starts, while the total throughput remains unchanged, each job's share is halved. It further reduces to about a third when the third job arrives. Similarly, the reverse happens when jobs finishes in the reverse order. The system throughput roughly remains the same throughout the experiment. Note that Salus reacts almost immediately for job arriving and leaving events.

In contrast, FIFO scheduling or other sharing policies (e.g., MPS) cannot enforce fair sharing.

5.2 Hyper-Parameter Exploration

Using Salus to PACK many jobs is especially useful when many/all jobs are ready to run. One possible use case for this is automatic hyper-parameter tuning for DL models. Typically, hundreds of training jobs are generated in parallel for parameter exploration. Most of the generated models will be killed shortly after they are deemed to be of poor quality. In this case, increasing the concurrency on GPU can help improve the parameter exploration performance by running multiple small jobs together, whereas today only FIFO is possible.

We evaluate two sets of hyper-parameter exploration jobs: `resnet50_50` and `superres_128`, for image classification and resolution enhancement, respectively. Each set has 300 jobs, and each one completes after all 300 complete. A comparison of achieved makespan using FIFO (in TensorFlow) and Salus is shown in Figure 12. In the `resnet50_50` case, there is 1.07× makespan improvement while it is 2.38× for `superres_128`.

Little improvement is seen for `resnet50_50` because even if the GPU has enough memory to hold many of the jobs together, computation likely becomes the bottleneck under such heavy sharing. Consequently, the makespan of the whole set of jobs does not see much improvement.

5.3 Inference

So far we have only discussed DL training, but we note that serving a trained model, i.e., inference, can also be a good

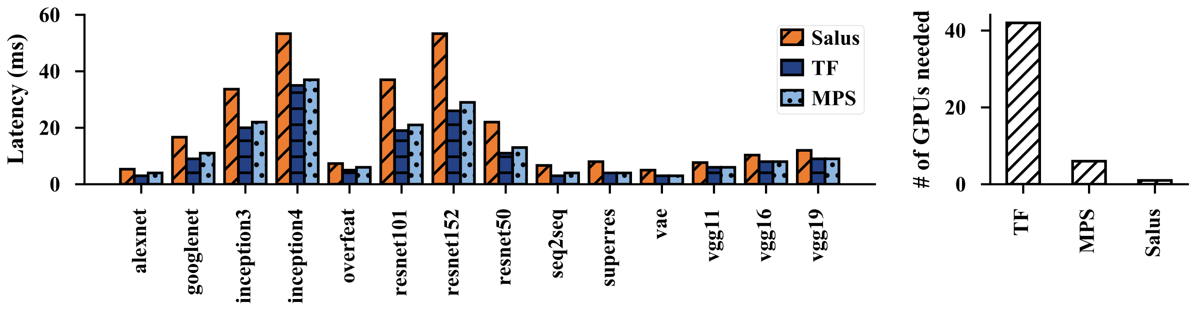


Figure 13: The latencies and number of GPUs needed to host 42 DL models for inference at the same time. 3 instances of each network is created. Each model have a low request rate.

– if not better – candidate for GPU memory sharing. Rather than focusing on throughout when training, latency of individual inference request becomes a more important requirement when serving DL models [8, 22].

In order to keep responsive to requests, DL models have to be online 24x7 hours. In the traditional setting, each model must reside on a dedicated GPU. However, the traffic of serving requests is not always constant throughout the day, and there are times when the request rate is significantly lower compared to peak. Consolidating DL models into fewer GPUs while remain responsive can save the maintains cost for service providers.

We demonstrate Salus's ability to reduce the number of GPUs needed while maintaining reasonable response latency in Figure 13. 42 DL inference jobs are selected consisting of 14 different models, 3 instances for each model. Without MPS or Salus, 42 GPUs are needed to hold these DL models. In contrast, Salus needs only 1 GPU, achieving 42× utilization improvement, while the average latency overhead is less than 5ms. For comparison, MPS needs 6 GPUs.

A future work is to detect current request rate for inference jobs and automatically scale up or down horizontally. Nevertheless, Salus provides the essential primitives that makes the implementation possible.

5.4 Overhead

Salus has to be efficient, otherwise the benefits gained from sharing can be easily offset by the overhead. Figure 14 shows per iteration training time in Salus, normalized by per iteration training time in baseline TensorFlow.

For most CNN models, Salus has minimal overhead – less than 10%, except for a few. The common point of these high-overhead DL models is that they also performs large portion of CPU computation in addition to heavy GPU usage. Since Salus implements its own execution engine, the CPU computation is also redirected and sent to Salus for execution, which is not yet heavily optimized.

We finally proceed to compare the performance to run two jobs on a single GPU using existing solutions. We consider the following approaches to enable sharing in addition to using Salus:

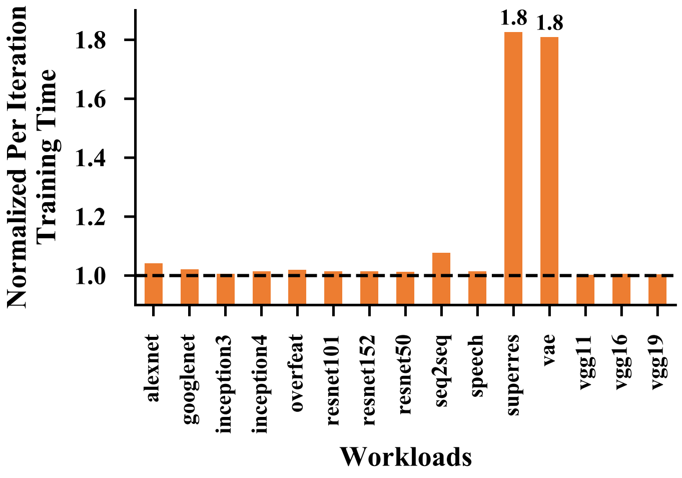


Figure 14: Per iteration time per workload in Salus, normalized by that of TensorFlow. Only the largest batch size for each network is shown, as other batch sizes have similar performance.

1. **Static Partitioning (SP)**: Jobs can be executed on the same GPU as long as the sum of two or more consecutive jobs' peak memory do not exceed the device capacity, using non-exclusive mode.
2. **SP + NVIDIA MPS**: NVIDIA MPS is intended to reduce GPU context switch and speed up the execution. It does not manage the GPU memory, so static partitioning is still needed.
3. **SP + MPS + Overcommit (OC)**: Using the unified memory access and GPU page fault feature introduced in CUDA 8, we can overcommit the GPU memory and let more jobs come in.

The setup is simple where two `alexnet_25` training jobs are started at the same time and each runs for a minute. The jobs run on a single GPU made possible using one of the above sharing solution. We then collect and compare the average JCT and report the result in Figure 15.

The result confirms that MPS is indeed better than SP due

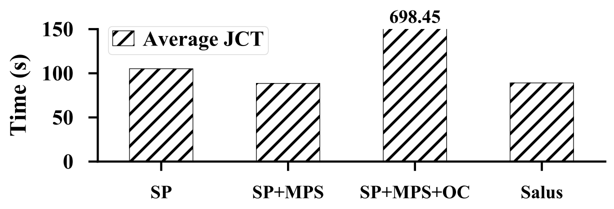


Figure 15: Two concurrent jobs of training `alexnet_25` for one minute.

to the avoidance of GPU context switching. Unfortunately, the promising SP+MPS+OC solution has significantly bad performance that is beyond useful at the moment. Salus manages to achieve almost the same performance as MPS while providing much more flexibility in scheduling policy. As shown before, in lightly-loaded inference scenarios, it can significantly outperform MPS in terms of utilization.

6 Related Work

Container-Based Sharing At the largest granularity, container-based solutions have been adopted by modern cluster managers such as Apache YARN [48], Mesos [29], and Kubernetes [3]. They typically treat GPUs at device-granularity; i.e., an allocation consists of one or more whole GPUs. Clearly, the problem of underutilization *per-gpu* remains.

Recent work on nvidia-docker [6] makes using one NVIDIA GPU in multiple containers possible, but it is essientially the same as removing exclusive-mode on GPU. As already discussed, this is not ideal and has performance issues. It is possible to use NVIDIA MPS, but the support is not complete yet [7].

Therefore, Salus is complementary to these work and can be used to enable fractional GPU allocations.

GPU Virtualization via Library Interception To share one single GPU among applications, virtualization via library interception or API remoting is a popular trick to bypass default hardware/driver implementations. Examples are gVirtuS [25], GViM [27], vCUDA [44], rCUDA [24], and work from Ravi et al. [43] to share GPU in cloud environments. However, most of the work focus on sharing in terms of GPU applications of no more than a few kernels. Modern DL applications usually make use of hundreds of unique GPU kernels during their training, and they also rely on advanced CUDA APIs (e.g., CUDA stream callbacks) that are often not supported in these works.

In addition, as an official implementation with similar techniques, CUDA MPS lacks wide support for DL frameworks. For example, TensorFlow crashes when running two instances on the MPS Server [14]. Gandiva [50] uses MPS as well, and the authors report unpredictable performance.

New API Instead of hacking library API, others choose to create new sets of API from scratch; e.g., Pagoda [51], GeMTC [33], etc.. While this achieves the most flexibility and efficiency, it is in practice hard if not impossible for existing DL frameworks to be adapted to the new API.

Efforts for Increasing GPU Utilization Rather than packing more applications into one GPU, another completely different approach to increase GPU utilization focuses on single application use cases. Some attempts to statically fuse multiple tasks together to increase efficiency; e.g., TensorFlow XLA [15], NNVM [19], and [26]. While other works focus on GPU kernel level concurrency and scheduling [39]. Salus is complementary to these approaches.

7 Concluding Remarks

GPUs have emerged as the primary computation devices for deep learning (DL) applications. However, modern GPUs and their runtimes do not allow multiple processes to coexist in a GPU. As a result, unused memory of a DL job remains unaccessible to other jobs, leading to large efficiency, performance loss, and head-of-line (HOL) blocking.

Salus is a consolidated execution service that enables fine-grained GPU sharing between complex, unmodified DL jobs. It achieves this by exposing two important primitives: (1) *fast job switching* that can be used to implement time sharing and preemption; and (2) the *GPU lane* abstraction to enable dynamic memory partitioning, which can be used for packing multiple jobs on the same GPU. Together, they can be used to implement unforeseen scheduling policies as well. Our integration of Salus with TensorFlow shows that Salus can allow multiple DL jobs to coexist, enable fair sharing and preemption between them, and improve overall efficiency and DL training performance in a shared-GPU environment.

However, Salus is only a first attempt, and it opens many interesting research challenges. First and forement, Salus provides a mechanism but the question of policy – what is the best scheduling algorithm for DL jobs running on a shared GPU? – remains open. Second, while not highlighted in the paper, Salus can be extended to multiple GPUs or even other accelerators on the same machine. Finally, we plan to extend it to GPUs across multiple machines leveraging RDMA.

References

[1] Caffe 2. https://caffe2.ai. Accessed: 2017-04-21.

[2] CUDA Multi-Process Service. https://goo.gl/R57gNW. Accessed: 2017-04-27.

[3] Google Container Engine. http://kubernetes.io. Accessed: 2018-04-21.

[4] KubeFlow. https://github.com/kubeflow/kubeflow. Accessed: 2018-08-21.

[5] MXNet issue #4018. https://github.com/apache/incubator-mxnet/issues/4018. Accessed: 2017-10-01.

[6] NVIDIA Container Runtime for Docker. https://github.com/NVIDIA/nvidia-docker. Accessed: 2018-08-02.

[7] NVIDIA Docker Issue #419. https://github.com/NVIDIA/nvidia-docker/issues/419. Accessed: 2018-09-07.

[8] NVIDIA TensorRT. https://devblogs.nvidia.com/nvidia-serves-deep-learning-inference. Accessed: 2018-09-18.

[9] Nvidia Volta unveiled: GV100 GPU and Tesla V100 accelerator announced. https://goo.gl/bHk1c2.

[10] Programming Guide :: CUDA Toolkit Documentation. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html. Accessed: 2018-09-19.

[11] PyTorch. http://pytorch.org. Accessed: 2017-04-21.

[12] TensorFlow Benchmarks. https://github.com/tensorflow/benchmarks. Accessed: 2018-09-19.

[13] TensorFlow issue #4196. https://github.com/tensorflow/tensorflow/issues/4196. Accessed: 2017-10-01.

[14] TensorFlow issue #9080. https://github.com/tensorflow/tensorflow/issues/9080. Accessed: 2017-10-01.

[15] TensorFlow XLA. https://www.tensorflow.org/performance/xla. Accessed: 2017-10-01.

[16] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng. TensorFlow: A system for large-scale machine learning. In *OSDI*, 2016.

[17] J. Bergstra, F. Bastien, O. Breuleux, P. Lamblin, R. Pascanu, O. Delalleau, G. Desjardins, D. Warde-Farley, I. Goodfellow, A. Bergeron, and Y. Bengio. Theano: Deep learning on GPUs with Python. In *BigLearn, NIPS Workshop*, 2011.

[18] J. Bergstra, D. Yamins, and D. D. Cox. Hyperopt: A Python Library for Optimizing the Hyperparameters of Machine Learning Algorithms. *SCIPY*, 2013.

[19] T. Chen, M. Li, Y. Li, M. Lin, N. Wang, M. Wang, T. Xiao, B. Xu, C. Zhang, and Z. Zhang. MXNet: A flexible and efficient machine learning library for heterogeneous distributed systems. *arXiv preprint arXiv:1512.01274*, 2015.

[20] T. Chilimbi, Y. Suzue, J. Apacible, and K. Kalyanaraman. Project Adam: Building an efficient and scalable deep learning training system. In *OSDI*, 2014.

[21] R. Collobert, K. Kavukcuoglu, and C. Farabet. Torch7: A Matlab-like environment for machine learning. In *BigLearn, NIPS Workshop*, 2011.

[22] D. Crankshaw, X. Wang, G. Zhou, M. J. Franklin, J. E. Gonzalez, and I. Stoica. Clipper: A low-latency online prediction serving system. In *NSDI*, 2017.

[23] J. Dean, G. S. Corrado, R. Monga, K. Chen, M. Devin, Q. V. Le, M. Z. Mao, M. Ranzato, A. Senior, P. Tucker, K. Yang, and A. Y. Ng. Large scale distributed deep networks. In *NIPS*, 2012.

[24] J. Duato, A. J. Pena, F. Silla, R. Mayo, and E. S. Quintana-Orti. rCUDA: Reducing the number of GPU-based accelerators in high performance clusters. In *HPCS*, 2010.

[25] G. Giunta, R. Montella, G. Agrillo, and G. Coviello. A GPGPU Transparent Virtualization Component for High Performance Computing Clouds. In *EuroPar*, 2010.

[26] C. Gregg, J. Dorn, K. M. Hazelwood, and K. Skadron. Fine-grained resource sharing for concurrent GPGPU kernels. In *HotPar*, 2012.

[27] V. Gupta, A. Gavrilovska, K. Schwan, H. Kharche, N. Tolia, V. Talwar, and P. Ranganathan. GViM: GPU-accelerated Virtual Machines. In *Workshop on System-level Virtualization for HPC*. ACM, 2009.

[28] A. Hannun, C. Case, J. Casper, B. Catanzaro, G. Diamos, E. Elsen, R. Prenger, S. Satheesh, S. Sengupta, A. Coates, et al. Deep speech: Scaling up end-to-end speech recognition. *arXiv preprint arXiv:1412.5567*, 2014.

[29] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, and I. Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *NSDI*, 2011.

[30] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell. Caffe: Convolutional architecture for fast feature embedding. In *ACMMM*, 2014.

[31] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, et al. In-datacenter performance analysis of a Tensor Processing Unit. In *ISCA*, 2017.

[32] D. P. Kingma and M. Welling. Auto-encoding variational bayes. *arXiv preprint arXiv:1312.6114*, 2013.

[33] S. J. Krieder, J. M. Wozniak, T. Armstrong, M. Wilde, D. S. Katz, B. Grimmer, I. T. Foster, and I. Raicu. Design and Evaluation of the Gemtc Framework for GPU-enabled Many-task Computing. In *HPDC*, 2014.

[34] R. Landaverde, T. Zhang, A. K. Coskun, and M. Herbordt. An investigation of Unified Memory Access performance in CUDA. In *HPEC*, 2014.

[35] Y. LeCun, Y. Bengio, and G. Hinton. Deep learning. *Nature*, 521(7553):436–444, 2015.

[36] L. Li, K. Jamieson, G. DeSalvo, A. Rostamizadeh, and A. Talwalkar. Hyperband: A Novel Bandit-Based Approach to Hyperparameter Optimization. *arXiv preprint arXiv:1603.06560*, 2016.

[37] M. Li, D. G. Andersen, J. W. Park, A. J. Smola, A. Ahmed, V. Josifovski, J. Long, E. J. Shekita, and B.-Y. Su. Scaling distributed machine learning with the parameter server. In *OSDI*, 2014.

[38] NVIDIA. Programming guide :: Cuda toolkit documentation.

[39] S. Pai, M. J. Thazhuthaveetil, and R. Govindarajan. Improving GPGPU concurrency with elastic kernels. In *ASPLOS*, 2013.

[40] J. J. K. Park, Y. Park, and S. Mahlke. Chimera: Collaborative preemption for multitasking on a shared GPU. In *ASPLOS*, 2015.

[41] Y. Peng, Y. Bao, Y. Chen, C. Wu, and C. Guo. Optimus: an efficient dynamic resource scheduler for deep learning clusters. In *EuroSys*, 2018.

[42] J. Rasley, Y. He, F. Yan, O. Ruwase, and R. Fonseca. Hyperdrive: Exploring hyperparameters with pop scheduling. In *Middleware*, 2017.

[43] V. T. Ravi, M. Becchi, G. Agrawal, and S. Chakradhar. Supporting GPU sharing in cloud environments with a transparent runtime consolidation framework. In *HPDC*, 2011.

[44] L. Shi, H. Chen, J. Sun, and K. Li. vCUDA: GPU-Accelerated High-Performance Computing in Virtual Machines. *IEEE Transactions on Computers*, 61(6):804–816, June 2012.

[45] W. Shi, J. Caballero, F. Huszár, J. Totz, A. P. Aitken, R. Bishop, D. Rueckert, and Z. Wang. Real-time single image and video super-resolution using an efficient sub-pixel convolutional neural network. In *CVPR*, 2016.

[46] I. Sutskever, O. Vinyals, and Q. V. Le. Sequence to sequence learning with neural networks. In *NIPS*, 2014.

[47] Y. Suzuki, H. Yamada, S. Kato, and K. Kono. Towards multi-tenant GPGPU: Event-driven programming model for system-wide scheduling on shared GPUs. In *MaRS*, 2016.

[48] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, B. Saha, C. Curino, O. O'Malley, S. Radia, B. Reed, and E. Baldeschwieler. Apache Hadoop YARN: Yet another resource negotiator. In *SOCC*, 2013.

[49] W. Wang, G. Chen, T. Dinh, J. Gao, B. Ooi, and K. Tan. SINGA: A distributed system for deep learning. Technical report, NUS Tech Report, 2015.

[50] W. Xiao, R. Bhardwaj, R. Ramjee, M. Sivathanu, N. Kwatra, Z. Han, P. Patel, X. Peng, H. Zhao, Q. Zhang, F. Yang, and L. Zhou. Gandiva: Introspective cluster scheduling for deep learning. In *OSDI*, 2018.

[51] T. T. Yeh, A. Sabne, P. Sakdhnagool, R. Eigenmann, and T. G. Rogers. Pagoda: Fine-grained GPU resource virtualization for narrow tasks. In *PPoPP*, 2017.

[52] D. Yu, A. Eversole, M. Seltzer, K. Yao, O. Kuchaiev, Y. Zhang, F. Seide, Z. Huang, B. Guenter, H. Wang, J. Droppo, G. Zweig, C. Rossbach, J. Gao, A. Stolcke, J. Currey, M. Slaney, G. Chen, A. Agarwal, C. Basoglu, M. Padmilac, A. Kamenev, V. Ivanov, S. Cypher, H. Parthasarathi, B. Mitra, B. Peng, and X. Huang. An introduction to computational networks and the computational network toolkit. Technical report, Microsoft Research, October 2014.

[53] K. Zhang, B. He, J. Hu, Z. Wang, B. Hua, J. Meng, and L. Yang. G-NET: Effective GPU Sharing in NFV Systems. In *NSDI*, 2018.

[54] M. Zhu, L. Liu, C. Wang, and Y. Xie. CNNLab: a novel parallel framework for neural networks using GPU and FPGA-a practical study with trade-off analysis. *arXiv preprint arXiv:1606.06234*, 2016.

Appendix

7.1 Workloads

Table 3 is the full list of workloads and their batch sizes we used in our evaluation.

Figure 16 is the same peak and average GPU memory usage measurement done in PyTorch, except `overfeat`, which we could not find a working implementation.

Model	Type	Batch Sizes
`alexnet`	Classification	25, 50, 100
`googlenet`	Classification	25, 50, 100
`inception3`	Classification	25, 50, 100
`inception4`	Classification	25, 50, 75
`overfeat`	Classification	25, 50, 100
`resnet50`	Classification	25, 50, 75
`resnet101`	Classification	25, 50, 75
`resnet152`	Classification	25, 50, 75
`vgg11`	Classification	25, 50, 100
`vgg16`	Classification	25, 50, 100
`vgg19`	Classification	25, 50, 100
`vae`	Auto Encoder	64, 128, 256
`superres`	Super Resolution	32, 64, 128
`speech`	NLP	25, 50, 75
`seq2seq`	NLP	Small, Medium, Large

Table 3: DL models, their types, and the batch sizes we used. Note that the entire network must reside in GPU memory when it is running. This restricts the maximum batch size we can use for each network.

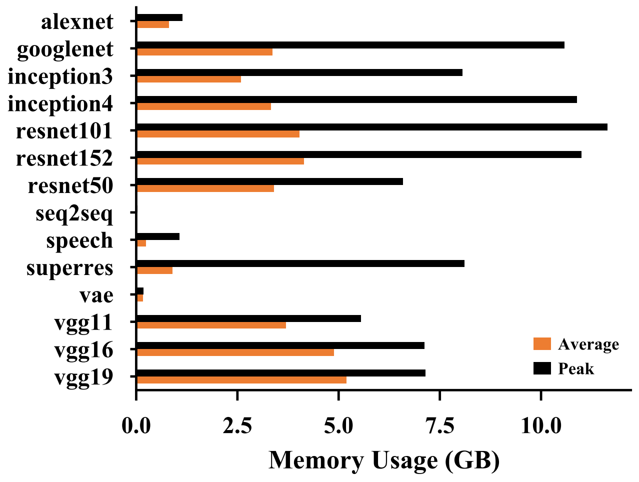


Figure 16: Average and peak GPU memory usage per workload, measured in PyTorch and running on NVIDIA P100 with 16 GB memory. The average and peak usage for vae is 156 MB, 185 MB, which are too small to show in the figure.