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

Peifeng Yu [1] **Mosharaf Chowdhury** [1]

Abstract

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 deep learning (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*, to achieve fine-grained GPU sharing among multiple DL applications. Salus is an efficient, consolidated execution service that exposes the GPU to different DL applications, and it 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. Our integration of Salus with TensorFlow and evaluation on popular DL jobs shows that Salus can improve the average completion time of DL training jobs by $3.19\times$, GPU utilization for hyper-parameter tuning by $2.38\times$, and GPU utilization of DL inference applications by $42\times$ over not sharing the GPU and $7\times$ 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 (LeCun et al., 2015). Consequently, both industry and academia are building DL solutions – e.g., TensorFlow (Abadi et al., 2016), CNTK (Yu et al., 2014), and others (Dean et al., 2012; Chen et al., 2015; Paszke et al., 2019; Bergstra et al., 2011) – to enable both *training* 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 (Jia et al., 2018; Zhu et al., 2016; Jouppi et al., 2017; Abadi et al., 2016). Unfortunately, the minimum granularity of GPU allocation today is often the entire GPU – *an application can have multiple GPUs, but each GPU can only be allocated to exactly one application* (Jeon et al., 2019). 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.

[1]Department of Electronic Engineering and Computer Science, University of Michigan, Michigan, USA. Correspondence to: Peifeng Yu <peifeng@umich.edu>.

Proceedings of the 3^{rd} *MLSys Conference*, Austin, TX, USA, 2020. Copyright 2020 by the author(s).

First, the coarse-grained, one-at-a-time GPU allocation model[1] hinders the scheduling ability of GPU cluster managers (Gu et al., 2019; Xiao et al., 2018; Hindman et al., 2011; Bernstein, 2014; Vavilapalli et al., 2013; Dutta & Huang, 2019; NVIDIA, 2020b). 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 fully be 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 (Dutta & Huang, 2019; Jeon et al., 2019), which is susceptible to the head-of-line (HOL) blocking problem; or they suffer large overhead when using preemptive scheduling (Gu et al., 2019).

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% (§2.1) due to varied memory usage over time and between iterations. Similar patterns 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

[1]It is possible to forcibly run multiple processes on the same GPU, but that leads to high overhead compared to exclusive mode.

cost by decreasing the number of GPUs needed in serving clusters (Crankshaw et al., 2017; Migacz, 2017).

Additionally, the increasingly popular trend of automatic hyper-parameter tuning of DL models (Bergstra et al., 2013; Li et al., 2016; Rasley et al., 2017) further emphasizes the need to improve GPU utilization. This can be viewed as "pre-training." One exploration task usually generates hundreds of training jobs in parallel, 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 exploration jobs arrive in waves and the result is useful only after all jobs are finished.

We address these issues by presenting Salus, which enables 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 (§2.2). 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, we implement a variety of GPU scheduling solutions (§4).

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

2 Background and Motivation

2.1 DL Workloads Characteristics

We analyzed a collection of 15 DL models (Table 2 in Appendix) to understand the resource usage patterns of DL jobs. This set of models are compiled from the official TensorFlow CNN benchmarks (TensorFlow, 2020) and other 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

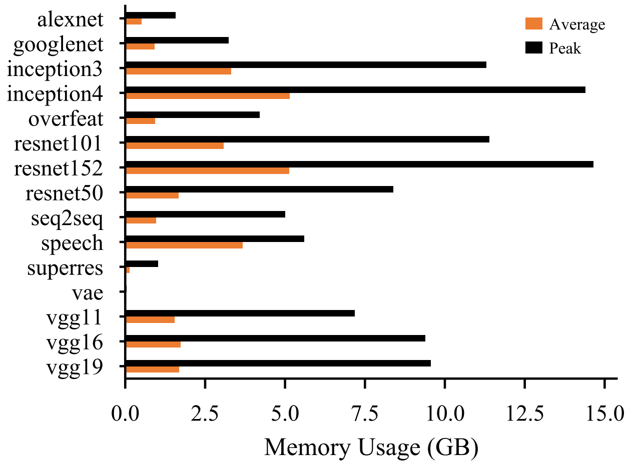


Figure 1. Average and peak GPU memory usage per workload, measured in TensorFlow and running on NVIDIA P100. The average and peak usage for vae is 22 MB, 35 MB, which are too small to show in the figure. The appendix also includes the measurement in PyTorch (Figure 14), which shares a similar pattern.

create 45 distinct workloads, as shown in Table 2. Note that the batch size specifies the number of samples (e.g., images for CNNs) trained in each iteration and affects the size of model parameters. Thus the larger the batch size, the longer it takes to compute an iteration. Throughout the paper, we uniquely identify a workload by the model 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: (i) GPU computation resources (primarily in terms of computation time, often referred to as GPU utilization in the literature) and (ii) 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 a few key GPU memory usage patterns in DL workloads that lead to memory underutilization issues and/or opportunities for improvements.

Heterogeneous Peak Memory Usage Across Jobs DL workloads are known for heavy memory usage (Abadi et al., 2016; Li et al., 2014; Chilimbi et al., 2014). Figure 1 visualizes the average and peak memory usages of our workloads. As models 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

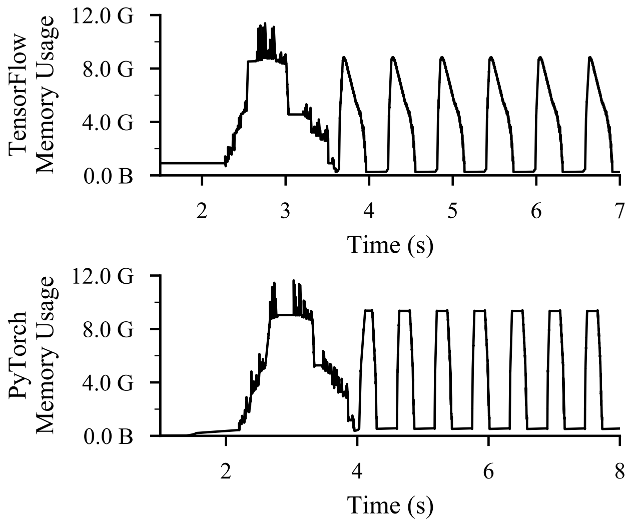


Figure 2. Part of the GPU memory usage trace showing the spatiotemporal pattern when training `resnet101_75` on NVIDIA P100, using TensorFlow and PyTorch.

even during peak allocation periods, it may be possible to run multiple models on the same GPU instead of FIFO.

Temporal Memory Usage Variations Within a Job Within each job, however, each iteration of a DL training job 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 parameters of a model – 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 peak, ranging from 110.9 MB for `googlenet_25` to 822.2 MB for `resnet152_75`. *As long as the model parameter is already in GPU memory, we can quickly start an iteration of that model.* This gives us an additional opportunity to improve sharing and utilization.

2.2 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 the GPU memory among DL jobs. This cannot completely address the underutilization problem due to high peak-to-average memory usage of DL jobs.

NVIDIA's Multi-Process Service (MPS) (NVIDIA, 2020a) can be used to speed up the static partitioning approach for GPU sharing by avoiding costly GPU context switches. Nonetheless, MPS has limited support for DL frameworks or companies' in-house monitoring tool according to our experiments and various bug reports.

Static partitioning and MPS also fail to provide performance isolation. Co-located DL jobs can cause large and hard to predict interferences. A recent work, Gandiva (Xiao et al., 2018) approaches this by trial and error and fallback to non-sharing mode. Xu et al. (2019) propose to use machine learning model to predict and schedule GPU-using VMs in the cluster to minimize interferences.

NVIDIA's TensorRT Inference server (Migacz, 2017) and the prior work by Samanta et al. (2019) achieve simultaneous DL inference in parallel on a single GPU. But they lack support for DL training.

Finally, earlier works on fine-grained GPU sharing fall into two categories. Some attempt to intercept GPU driver API calls and dynamically introduce concurrency by time-slicing kernel execution at runtime (Pai et al., 2013; Ravi et al., 2011; Park et al., 2015). Others call for new APIs for GPU programming (Suzuki et al., 2016; Yeh et al., 2017; Zhang et al., 2018). 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 go up to several hundreds.

3 SALUS

Salus [2] is our attempt to build an ideal solution to GPU sharing. It is designed to enable efficient, fine-grained sharing while maintaining compatibility with existing frameworks (§3.1). Its overall design is guided by the unique memory usage characteristics of DL jobs. Packing multiple jobs onto one GPU changes the combined memory allocation patterns and special care must be taken to mitigate increased fragmentation, because existing DL frameworks are designed for the job-exclusive GPU usage scenario. 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* (§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

[2]Salus is available as an open-source software at `https://github.com/SymbioticLab/Salus`.

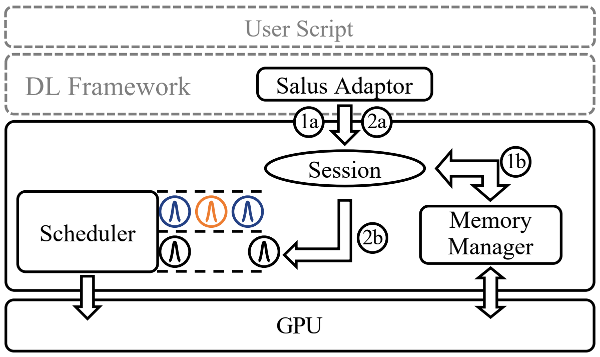


Figure 3. Salus sits between DL frameworks and the hardware in the DL stack, being transparent to users.

space-shared manner – all of which can be utilized by a GPU cluster scheduler (Xiao et al., 2018; Gu et al., 2019). We demonstrate the possibilities by implementing common scheduling policies such as preempting jobs for shortest-remaining-time-first (SRTF) or fair sharing, 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 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 not change at all. All scripts will work the same as before.

It is perhaps better to explain the architecture via an example of the life cycle of a DL job. When a 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 sent 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 elaborate on a performance-efficiency tradeoff in choosing this granularity (§3.2.2).

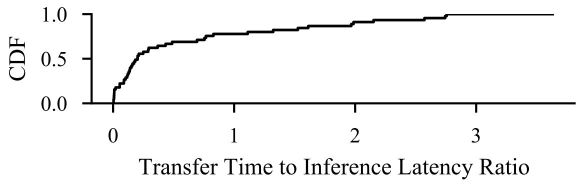


Figure 4. Cumulative distribution function (CDF) of theoretical minimal transfer time to model inference latency ratio for 15 models. Transfer time is calculated using 30 GB/s for transfer speed.

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 (Xiao et al., 2018) 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 transferred, the communication time is still 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

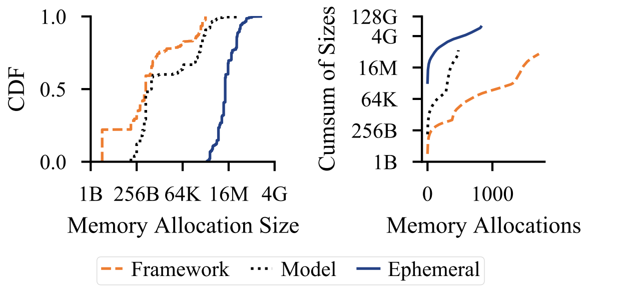


Figure 5. Memory allocation distribution for ephemeral, model and framework-internal memory, measured using `inception3_50`.

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 than ephemeral ones in 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.*

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 (Xiao et al., 2018), 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

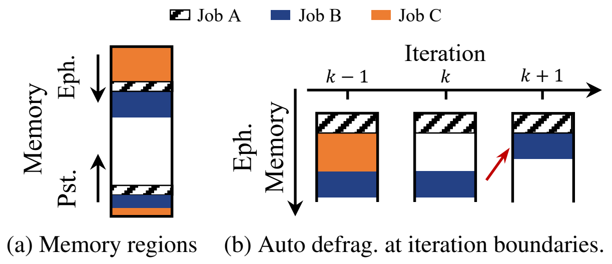


Figure 6. The memory layout of the GPU lane scheme.

as kernel batching and pipelining.

3.3 Spatial 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* (Eph.) and *persistent* (Pst.) regions, growing from both end of the memory space (Figure 6a). A DL job's ephemeral memory goes into the ephemeral region, while other types of memory is allocated in the persistent 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, which are time-shared within the lane.

The lane's restriction on execution is necessary because different from the other two types of memory, ephemeral allocations happens in small chunks and cannott be predicted ahead. As a result, simply putting two iterations together may cause deadlock because there is no swapping [3] for the oversubscribed memory.

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 correctly. 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 simulta-

[3]The existing memory overcommit technique Unified Memory Access is too slow to use. See §5.4.

neously. While there are works implementing a memory planner before actually starting the iteration (Chen et al., 2015), they are not available to all frameworks.

Since our goal is to fully support existing workloads with minimal impact on the user, we approach the problem by limiting 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 – 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 6b, when job C stops, its lane space is quickly reclaimed (the red arrow) at the iteration boundary by job B 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:

$$\sum_{\text{job } i} P_i + \sum_{\text{lane } l} \max_{\text{job } j \text{ in } l}(E_j) \leq C$$

where P and E are respectively the persistent (model and framework-internal) and ephemeral memory usage of a job. C is the capacity of the GPU. The second term is the sum of all lanes' size, which is defined as the maximum ephemeral memory usage of all jobs in the lane.

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.

Implementation-wise, the system is event-driven, and reacts when there are jobs arriving or finishing, or at iteration boundaries when auto defragmentation happens. The lane finding logic is shown in Algorithm 1, which outputs a suitable lane given a job's memory requirement.

Algorithm 1 Find GPU Lane for Job

```
1:  Input: $P$: the job's persistent memory requirement
           $E$: the job's ephemeral memory requirement
           $C$: total memory capacity of the GPU
           $P_i$: persistent memory usage of existing job $i$
           $L_j$: lane size of existing lane $j$
           $\mathbb{L}$: set of existing lanes
2:  if $\sum_i P_i + P + \sum_j L_j + E \leq C$ then
3:      $lane \leftarrow$ new GPU lane with capacity $E$
4:      return $lane$
5:  end if
6:  for all $j \in \mathbb{L}$ do
7:      if $L_j \geq E$ and is the best match then
8:          return $j$
9:      end if
10: end for
11: for $r \in \mathbb{L}$ in $L_r$ ascending order do
12:     if $\sum_i P_i + P + \sum_j L_j - L_r + E \leq C$ then
13:         $L_r \leftarrow E$
14:         return $r$
15:     end if
16: end for
17: return not found
```

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, which can lead to HOL blocking. Although recent works (Xiao et al., 2018; Gu et al., 2019) have proposed time sharing, they enforce sharing over many minutes due to high switching overhead.

Thanks to its fine-grained GPU sharing primitives, Salus makes it possible to pack jobs together to increase efficiency, or to enforce any priority criteria with preemption. It opens up a huge design space to 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 memory usages. However, packing too many lanes exceeding the GPU memory capacity will either crash the jobs or incur costly paging overhead (if memroy overcommit 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. No fairness is considered among lanes.

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, Salus can significantly decrease the GPU requirements of model serving systems like Clipper (Crankshaw et al., 2017).

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 model. 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. This way, Salus can support job priorities based on arbitrary criteria, including 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 (Peng et al., 2018), 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. 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 while maintaining the safety condition, and equalize total service over time for jobs in each lane.

5 Evaluation

We have integrated Salus with TensorFlow and evaluated it using a collection of training, hyper-parameter tuning, and inference workloads (TensorFlow, 2020; Sutskever et al., 2014; Kingma & Welling, 2013; Shi et al., 2016; Hannun et al., 2014) to understand its effectiveness and overhead. The highlights of our evaluation are as follows:

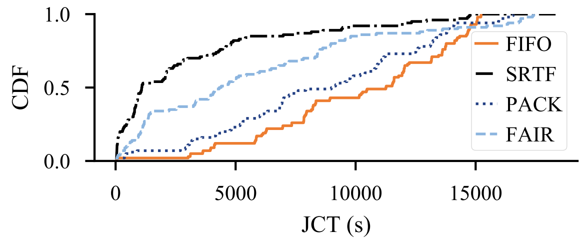


Figure 7. CDFs of JCTs for all four scheduling policies.

Sched.	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 1. Makespan and aggregate statistics for different schedulers.

- 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).
- Salus can run multiple DL jobs during hyper-parameter tuning, 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 available in CloudLab (Duplyakin et al., 2019). Each GPU has 16GB on-chip memory. TensorFlow v1.5.0 and CUDA 8.0 are used in all cases.

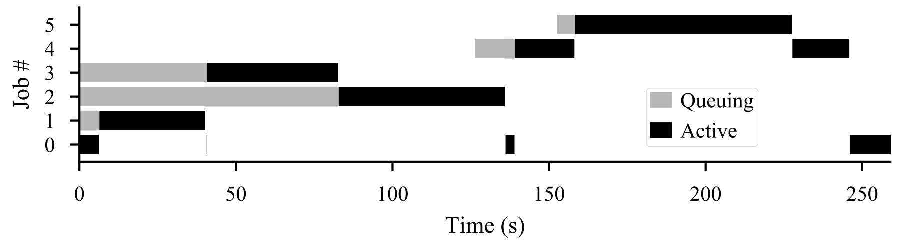
Baseline (s) Our primary baseline is the FIFO scheduling commonly used in today's GPU clusters (Xiao et al., 2018). We also compare against NVIDIA MPS.

5.1 Long-Running Training

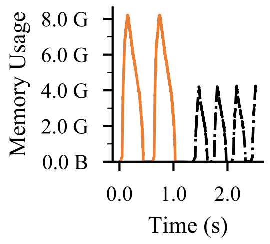
We start by focusing on Salus's impact on training. To this end, we evaluate Salus using a job trace of 100 workloads, generated using the jobs described in Table 2. We considered multiple batch sizes and durations of each training job in the mix. The overall distribution followed one found in a production cluster (Gu et al., 2019).

We compare four different schedulers:

1. **FIFO** refers to processing jobs in order of their arrival.



(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. Black areas represent active execution.



(b) Memory usage during a job switching.

Figure 8. Details of a snapshot during the long trace running with SRTF. In both slices, time is normalized.

This is the de facto mechanism in use today.

2. **SRTF** is a preemptive shortest-remaining-time-first scheduler. We assume that the duration is known or can be estimated (Peng et al., 2018).
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 7 presents the distributions of JCTs for all four policies, while Table 1 presents makespan and aggregate statistics. Given the similarities of makespan values between FIFO, SRTF, and FAIR, we can conclude 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 allows cheap preemption implementation, which, in turn, makes possible the shortest-remaining-time-first (SRTF) scheduling policy. Second, we show fast job switching can achieve fair sharing among DL jobs in seconds-granularity – instead of minutes (Xiao et al., 2018). 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 models. 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.

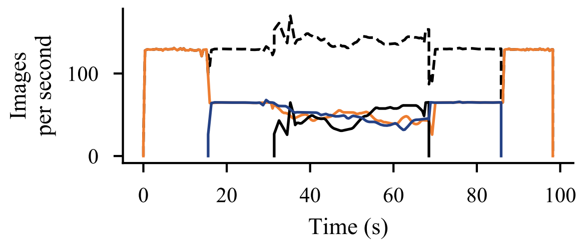


Figure 9. Fair sharing among three `inception3_50` training jobs. Black dashed line shows the overall throughput.

We pick a segment in the long job trace, containing exact the scenario, and record its detailed execution trace, showing in Figure 8a. 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 8b is another example demonstrating Salus's ability to fast switch. It visualizes memory allocations 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 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 9.

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

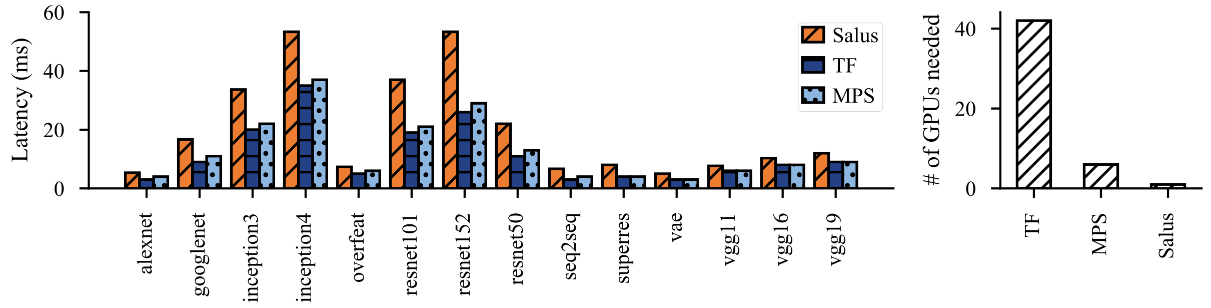


Figure 10. The latencies and number of GPUs needed to host 42 DL models for inference at the same time. 3 instances of each model is created. Each instance has a low request rate.

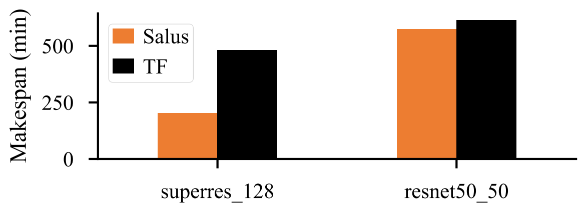


Figure 11. Makespan of two hyper-parameter tuning multi-jobs each of which consists of 300 individual jobs.

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. 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 11. 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 while the GPU has enough memory, computation becomes the bottleneck under such heavy sharing. Consequently, the makespan 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 – 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 (Crankshaw et al., 2017; Migacz, 2017).

In order to keep responsive to requests, DL models have to be online 24×7 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 10. 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× 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 12 shows per iteration training time in Salus, normalized by

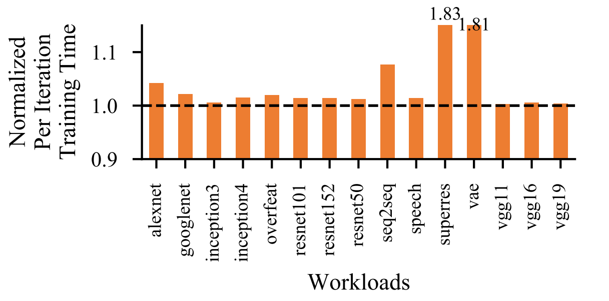


Figure 12. Per iteration time per workload in Salus, normalized by that of TensorFlow. Only the largest batch size for each model is reported, as other batch sizes have similar performance.

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. Two `alexnet_25` training jobs are started at the same time and each runs for a minute. The jobs share a single GPU using *Salus*, *static partitioning* (SP), *static partitioning with MPS* (SP+MPS), and *static partitioning with MPS and memory overcommit* (SP+MPS+OC). We collect and compare the average JCT and report the result in Figure 13.

The result confirms that MPS is indeed better than SP due to the avoidance of GPU context switching. Unfortunately, the 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.

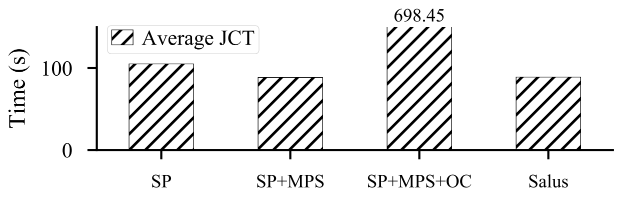


Figure 13. Two concurrent `alexnet_25` training jobs for 1 min.

6 Concluding Remarks

GPUs have emerged as the primary computation devices for deep learning (DL) applications. However, modern GPUs and their runtimes do not allow efficient multiple coexisting processes 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 enables fine-grained GPU sharing among complex, unmodified DL jobs 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 new policies as well.

However, Salus is only a first attempt, and it opens many interesting research challenges. First, 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.

Acknowledgments

We thank the ConFlux team from the University of Michigan for providing GPU servers, which made many experiments on Salus possible. We would also like to thank Yuchen Wang, who helped profile PyTorch memory usage, and Yiwen Zhang, who helped with evaluating an earlier version of Salus that used GPU kernel-granularity scheduling. Special thanks to all the SymbioticLab members for their helpful comments and feedback. This research was supported in part by NSF grants CCF-1629397 and CNS-1909067.

References

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

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

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

Bernstein, D. Containers and Cloud: From LXC to Docker to Kubernetes. *IEEE Cloud Computing*, pp. 81–84, 2014.

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

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

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

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

Duplyakin, D., Ricci, R., Maricq, A., Wong, G., Duerig, J., Eide, E., Stoller, L., Hibler, M., Johnson, D., Webb, K., Akella, A., Wang, K., Ricart, G., Landweber, L., Elliott, C., Zink, M., Cecchet, E., Kar, S., and Mishra, P. The design and operation of CloudLab. In *ATC*, 2019.

Dutta, D. and Huang, X. Consistent multi-cloud AI lifecycle management with kubeflow. In *OpML*, 2019.

Gu, J., Chowdhury, M., Shin, K. G., Zhu, Y., Jeon, M., Qian, J., Liu, H., and Guo, C. Tiresias: A GPU Cluster Manager for Distributed Deep Learning. In *NSDI*, pp. 485–500, 2019.

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

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

Jeon, M., Venkataraman, S., Phanishayee, A., Qian, J., Xiao, W., and Yang, F. Analysis of Large-Scale Multi-Tenant GPU Clusters for DNN Training Workloads. In *ATC*, pp. 947–960, 2019.

Jia, Z., Maggioni, M., Staiger, B., and Scarpazza, D. P. Dissecting the NVIDIA Volta GPU Architecture via Microbenchmarking. *arXiv preprint arXiv:1804.06826*, 2018.

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

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

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

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

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

Migacz, S. 8-bit inference with tensorrt. In *GTC*, 2017.

NVIDIA. CUDA Multi-Process Service. `https://web.archive.org/web/20200228183056/https://docs.nvidia.com/deploy/mps/index.html`, 2020a. Accessed: 2020-02-28.

NVIDIA. Programming Guide :: CUDA Toolkit Documentation. `https://web.archive.org/web/20200218210646/https://docs.nvidia.com/cuda/cuda-c-programming-guide/`, 2020b. Accessed: 2020-02-28.

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

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

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Kopf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J., and Chintala, S. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems 32*, 2019.

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

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

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

Samanta, A., Shrinivasan, S., Kaufmann, A., and Mace, J. No DNN Left Behind: Improving Inference in the Cloud with Multi-Tenancy. *arXiv preprint arXiv:1901.06887*, 2019.

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

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

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

TensorFlow. TensorFlow Benchmarks. https://web.archive.org/web/20200228184228/https://github.com/tensorflow/benchmarks/tree/cnn_tf_v1.5_compatible, 2020. Accessed: 2020-02-28.

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

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

Xu, X., Zhang, N., Cui, M., He, M., and Surana, R. Characterization and Prediction of Performance Interference on Mediated Passthrough GPUs for Interference-aware Scheduler. In *HotCloud*, 2019.

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

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

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

Zhu, M., Liu, L., Wang, C., and Xie, Y. 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.

A WORKLOADS

Table 2 is the full list of workloads and their batch sizes we used in our evaluation.

Figure 14 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 2. DL models, their types, and the batch sizes we used. Note that the entire model must reside in GPU memory when it is running. This restricts the maximum batch size we can use.

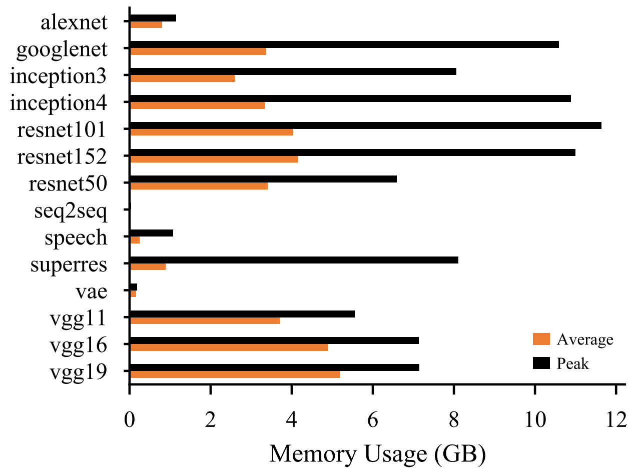


Figure 14. Average and peak GPU memory usage per workload, measured in PyTorch and running on NVIDIA P100 with 16 GB memory. The average and peak usage for vae is 156 MB, 185 MB, which are too small to show in the figure.

B ARTIFACT APPENDIX

B.1 Abstract

The artifact includes the server implementation of Salus, modified tensorflow, as well as modified tensorflow benchmarks used in the paper evaluation. This artifact requires NVIDIA GPU and CUDA.

B.2 Artifact check-list (meta-information)

- **Algorithm: yes**
- **Compilation: G++7 with CMake**
- **Binary: Docker image available**
- **Run-time environment: Ubuntu 16.04 with CUDA 9.1 and CUDNN. Root access not required.**
- **Hardware: GPU with reasonable large memory**
- **Metrics: Job completion time**
- **Output: log files with provided parsing scripts**
- **How much disk space required (approximately)?: 100GB**
- **How much time is needed to prepare workflow (approximately)?: 1 day**
- **How much time is needed to complete experiments (approximately)?: 1 week**
- **Publicly available?: yes**
- **Code licenses (if publicly available)?: Apache-2.0**
- **Workflow framework used?: none**
- **Archived (provide DOI)?: Salus (`https://doi.org/10.5281/zenodo.3606893`), tensorflow-salus (`https://doi.org/10.5281/zenodo.3606903`), tf_benchmarks (`https://doi.org/10.5281/zenodo.3606901`)**

B.3 Description

B.3.1 How to access

- Server implementation: `https://github.com/SymbioticLab/Salus`
- Tensorflow-salus: `https://github.com/SymbioticLab/tensorflow-salus`
- Benchmark: `https://github.com/Aetf/tf_benchmarks`

B.3.2 Hardware dependencies

The server code requires NVIDIA P100 GPUs.

B.3.3 Software dependencies

Ubuntu 16.04 OS with the following dependencies:

- `g++@7`
- `cuda@9.1` with `cudnn@7`
- `boost@1.66.0`
- `cppzmq@4.3.0`
- `zeromq@4.2.5`
- `nlohmann-json@3.1.2`
- `protobuf@3.4.1`
- `gperftools@2.7`
- `bazel@0.5.4`
- `Oracal JDK 8`

To run inside docker, NVIDIA docker runtime is also needed: `https://github.com/NVIDIA/nvidia-docker`

B.3.4 Data sets

Benchmarking uses random generated dataset.

B.4 Installation

To simply run the server with prebuilt docker image:

```
docker run --rm -it
registry.gitlab.com/salus/salus
```

B.4.1 Compilation

There are two seperate parts that need to be built in order: `tensorflow-salus` and the `salus` server.

The easiest way is to follow the same instructions used to build Docker images. Starting from `https://gitlab.com/Salus/builder`, which is the base docker image for `tensorflow-salus`, one can follow the commands on a Ubuntu 16.04 system and have an environment similar to what is used in the container.

After creating the builder environment, build `tensorflow-salus`.

Bazel is needed for building tensorlfow, so first install it from the Github release page: `https://github.com/bazelbuild/bazel/releases/tag/0.5.4`. We need version `0.5.4`.

Our modified version of tensorflow uses a helper script to configure the build system. First run

```
    inv deps && inv init
```

to create the configuration file `invoke.yml`. Next, adjust the configuration file as needed. Most likely the GPU compute capabilities need to be changed. It is under the key `TF_CUDA_COMPUTE_CAPABILITIES`. Now the configuration can be used to update the build system and actually build the package:

```
    inv config
    inv build
```

After successfully building tensorflow, use the following command to install it into the current virtual environment

```
    inv install
```

Do not delete the source tree for tensorflow-salus, which is needed for building the Salus server. The detailed command can be found in the `Dockerfile` located at the root of the Salus repo.

B.5 Experiment workflow

Python scripts under the `benchmarks` folder can be used to drive experiments.

Some additional python packages are required, install them with `pip install -r requirements.txt`.

The driver can be invoked from the root folder in the Salus repo

```
  python -m benchmarks.driver expXXX
```

where `expXXX` the file `expXXX.py` under the directory `benchmarks/exps/`.

Use `python -m benchmarks.driver --help` to see all options. Most likely various paths need to be changed.

B.6 Evaluation and expected result

The figures included in the paper are generated using the following experiment scripts: `card250`, `card260`, `card270`, `card271`, `card272`, `card274`, `card275`, `exp1`, `exp17`, `exp62`.

After running the experiment, a folder named after the experiment will be created in the output log folder, containing txt log files that can be further parsed. A corresponding parse script located in `scripts` can then be used to parse the result.

B.7 Experiment customization

New experiment can be done by adding a new experiment script. Please refer to other scripts for the API.