

PipeSwitch: Fast Pipelined Context Switching for Deep Learning Applications

Zhihao Bai*

Zhen Zhang*

Yibo Zhu[†]

Xin Jin*

**Johns Hopkins University* [†]*ByteDance Inc.*

Abstract

Deep learning (DL) workloads include throughput-intensive training tasks and latency-sensitive inference tasks. The dominant practice today is to provision dedicated GPU clusters for training and inference separately. Due to the need to meet strict Service-Level Objectives (SLOs), GPU clusters are often over-provisioned based on the peak load with limited sharing between applications and task types.

We present PipeSwitch, a system that enables unused cycles of an inference application to be filled by training or other inference applications. It allows multiple DL applications to *time-share* the same GPU with the entire GPU memory and *millisecond-scale* switching overhead. With PipeSwitch, GPU utilization can be significantly improved without sacrificing SLOs. We achieve so by introducing *pipelined context switching*. The key idea is to leverage the *layered* structure of neural network models and their *layer-by-layer* computation pattern to *pipeline* model transmission over the PCIe and task execution in the GPU with model-aware grouping. We also design unified memory management and active-standby worker switching mechanisms to accompany the pipelining and ensure process-level isolation. We have built a PipeSwitch prototype and integrated it with PyTorch. Experiments on a variety of DL models and GPU cards show that PipeSwitch only incurs a task startup overhead of 3.6–6.6 ms and a total overhead of 5.4–34.6 ms (10–50× better than NVIDIA MPS), and achieves near 100% GPU utilization.

1 Introduction

Deep learning (DL) powers an emerging family of intelligent applications in many domains, from retail and transportation, to finance and healthcare. GPUs are one of the most widely-used classes of accelerators for DL. They provide better trade-off between performance, cost and energy consumption than CPUs for deep neural network (DNN) models.

DL workloads include throughput-intensive training tasks and latency-sensitive inference tasks. The dominant practice today is to provision dedicated GPU clusters for training and inference separately. Inference tasks cannot be served with training clusters under flash crowds, and training tasks cannot utilize inference clusters when the inference load is low. Consequently, inference clusters are often over-provisioned for

the peak load, in order to meet strict Service Level Objectives (SLOs). Even for inference itself, production systems are typically provisioned to each application on per-GPU granularity to limit the interference between applications.

Ideally, multiple DL applications should be able to be packed to the same GPU server to maximize GPU utilization via time-sharing. This is exactly how operating systems achieve high CPU utilization via task scheduling and context switching. The idea of fine-grained CPU time-sharing has been further extended to cluster scheduling. For example, Google Borg [1] packs online services and batch jobs, and saves 20%-30% machines (compared with not packing them). Why can't we use GPUs in the same way?

The gap is that GPU has high overhead when switching between tasks. Consequently, naively using GPUs in the same way as CPUs will not satisfy the requirements of DL inference that have strict SLOs in the range of tens to hundreds of milliseconds [2, 3]. If a GPU switches to a DNN model (e.g., ResNet) that has not been preloaded onto the GPU, it can take multiple seconds before serving the first inference request, even with state-of-the-art tricks like CUDA unified memory [4] (§6). In contrast, CPU applications can be switched in milliseconds or even microseconds [5].

To avoid such switching overhead, the existing solution is to spatially share the GPU memory. For example, although NVIDIA Multiple Process Sharing (MPS) [6] and Salus [7] allow multiple processes to use the same GPU, they require all processes' data (e.g., DNN models) to be preloaded into the GPU memory. Unfortunately, the GPU memory is much more limited than host memory and cannot preload many applications. Sometimes, just one single memory-intensive training task may consume all the GPU memory. Moreover, the memory footprints of inference tasks are also increasing—the models are getting larger, and request batching is prevalently used to increase throughput [3]. In addition, this approach does not provide strong GPU memory isolation between applications.

As such, we argue that a context switching design that minimizes the switching overhead, especially quickly switching the contents on GPU memory, is a better approach for efficiently time-sharing GPUs. The DNN models can be held in host memory, which is much larger and cheaper than GPU memory, and the GPU can quickly context-switch between

the models either for training or inference. This way, *the number of* applications that can be multiplexed is not limited by the GPU memory size, and each application is able to use the entire GPU compute and memory resources during its time slice. To our best knowledge, no existing solution offers such context switching abstraction for GPU.

To this end, we propose PipeSwitch, a system that (i) enables GPU-efficient multiplexing of many DL applications on GPU servers via fine-grained time-sharing, and (ii) achieves millisecond-scale latencies and high throughput as dedicated servers. PipeSwitch enables unused cycles of an inference application to be filled by training or other inference applications. We achieve so by introducing a new technology called *pipelined context switching* that exploits the characteristics of DL applications to achieve millisecond-scale overhead for switching tasks on GPUs. Such small switching overhead is critical for DL applications to satisfy strict SLO requirements.

To understand the problem, we first perform a measurement study to profile the task switching overhead and analyze the overhead of each component. We divide the switching overhead into four components, which are old task cleaning, new task initialization, GPU memory allocation, and model transmission via PCIe from CPU to GPU. Every component takes a considerable amount of time, varying from tens of milliseconds to seconds. Such overhead is significant, because an inference task itself only takes tens of milliseconds on a GPU and the latency SLOs are typically a small multiple of the inference time [3].

We take a holistic approach, and exploit the characteristics of DL applications to minimize the overhead of all the components. Our design is based on a key observation that DNN models have a *layered* structure and a *layer-by-layer* computation pattern. As such, there is no need to wait for the entire model to be transmitted to the GPU before starting computation. Based on this observation, we design a pipelined model transmission mechanism, which *pipelines* model transmission over the PCIe and model computation in the GPU. Naive pipelining on per-layer granularity introduces high overhead on tensor transmission and synchronization. We divide layers into groups, and design an optimal model-aware grouping algorithm to find the best grouping strategy for a given model.

The computation of a DL task is layer by layer, which has a simple, regular pattern for memory allocation. The default general-purpose GPU memory management (e.g., CUDA unified memory [4]) is an overkill and incurs unnecessary overhead. We design unified memory management with a dedicated memory daemon to minimize the overhead. The daemon pre-allocates the GPU memory, and re-allocates it to each task, without involving the expensive GPU memory manager. The DNN models are stored only once in the memory daemon, instead of in every worker, to minimize memory footprint. We exploit that the memory allocation for a DNN model is deterministic to eliminate extra memory copies between the daemon and the workers and reduce the IPC overhead.

We use an active-standby mechanism for fast worker switching and process-level isolation. Each server contains an active worker and multiple standby workers. The active worker executes the current task on the GPU; the standby workers stay on the CPU and wait for the next task. Our mechanism parallelizes old task cleaning in the active worker and new task initialization in the standby worker to minimize worker switching overhead. With separate worker processes, PipeSwitch enforces process-level isolation.

Pipelining is a canonical technique widely used in computer systems to improve system performance and maximize resource utilization. Prior work in DL systems such as PipeDream [8] and ByteScheduler [9] has applied pipelining to distributed training. These solutions focus on *inter-batch* pipelining to overlap computation and *gradient* transmission of different batches for *training* workloads of the *same* DNN model. The key novelty of PipeSwitch is that it introduces *intra-batch* pipelining to overlap *model* transmission and computation to reduce the overhead of switching between *different* DNN models, which can be *either inference or training*. Unlike pipelining for the same task, PipeSwitch requires us to address new technical challenges on memory management and worker switching across *different* processes. We design new techniques to not only support training, but also inference that has strict SLOs.

In summary, we make the following contributions.

- We propose PipeSwitch, a system that enables GPU-efficient fine-grained time-sharing for multiple DL applications, and achieves millisecond-scale context switching latencies and high throughput.
- We introduce *pipelined context switching*, which exploits the characteristics of DL applications, and leverages pipelined model transmission, unified memory management, and active-standby worker switching to minimize switching overhead and enforce process-level isolation.
- We implement a system prototype and integrate it with PyTorch. Experiments on a variety of DL models and GPU cards show that PipeSwitch only incurs a task startup overhead of 3.6–6.6 ms and a total overhead of 5.4–34.6 ms (10–50× better than NVIDIA MPS), and achieves near 100% GPU utilization.

2 Motivation

In this section, we identify the inefficiencies in today’s shared GPU clusters, and motivate running DL workloads on GPUs in the fine-grained time-sharing model.

2.1 GPU Clusters

Shared GPU clusters. To run DNN workloads in a large scale, enterprises build GPU clusters that are either privately [10, 11] or publicly [12–14] shared by multiple users. Such GPU clusters are usually specifically designed with dedicated physical forms and power supplies, along with high speed networks and specialized task schedulers.

Why build a shared cluster instead of a dedicated one for each user? The main reason is to bring down the cost. The demand of training is not well predictable—it would depend on the progress of different developers. The demand of inference is more predictable, e.g., an inference task for a particular application usually has a daily periodical pattern based on the application usage. Nevertheless, the patterns can still vary across different tasks. Like traditional CPU workloads, a shared cluster by different tasks would increase the resource utilization via time-sharing.

No sharing between training and inference. However, such “shared” clusters are not shared between training and inference. Even though training and inference both use GPUs, the current practice is to build dedicated clusters for training and inference separately. This brings several inefficiencies.

- Inference clusters are over-provisioned for the peak load, because they directly serve user requests and need to meet strict SLOs. Although inference clusters are not always running at high utilization, they cannot be utilized by training.
- Training clusters are equipped with powerful GPUs to run training tasks, which are often elastic and do not have strict deadlines. However, when there is a flash crowd (e.g., an application suddenly becomes popular and the demand grows beyond the operator’s expectation), the training cluster cannot preempt the training tasks for inference tasks.
- Even for inference tasks, production systems often allocate GPUs to applications on per-GPU granularity (e.g., binding GPUs to the VMs, containers or processes of an application), in order to limit the interference between different applications and satisfy the SLO requirements.

One of the reasons for separately provisioning is that GPUs designed for inference tasks might be too wimpy for training tasks. This, however, has started to change with the arrival of new GPU hardware, most notably NVIDIA T4. Compared with NVIDIA V100 which has up to 32GB GPU memory and 15.7 TFLOPS (single-precision), NVIDIA T4 has comparable performance with 16GB GPU memory and 8.1 TFLOPS (single-precision). Also, new algorithms and systems for distributed training [8, 9, 15, 16] enable multiple GPUs to accelerate training, if one GPU is not fast enough.

Our industry collaborator, a leading online service provider, confirms this observation. This service provider currently runs more than 10K V100 GPUs for training, and at least $5\times$ as many T4 GPUs for inference. The computation power on both sides is within the same order of magnitude. The inference workload fluctuates in correlation with the number of active users, and shows clear peaks and valleys within each day—the peak demand during daytime is $> 2\times$ of the valley at midnight. It would be a great match to utilize inference GPUs during less busy times for training models that require daily updates with latest data. A good example is to fine-tune BERT using daily news. This means great opportunity in improving GPU utilization by Borg-like [1] systems for GPUs.

2.2 Fine-Grained Time-Sharing GPU

We envision to build GPU clusters that can be shared across different applications including training and inference. We propose to pack multiple DL applications onto the same GPU via fine-grained time-sharing abstraction to maximize GPU utilization. This is inspired by the OS scheduler and context switching in the CPU world. It has the following advantages.

- It would dramatically improve the resource utilization, especially because inference and training workloads have complementary usage patterns. Online inference services are often more idle during midnight, while many training developers would start a time-consuming job at night. Besides, inference loads on different models have different patterns, which also benefits from the time sharing.
- Similar to CPU workloads, fine-grained time-sharing can provide better utilization than provisioning dedicated resources, while providing necessary process-level isolation.
- It would greatly simplify the design of load balancers and schedulers as any server would be able to run any task with low overhead to switch between different applications.

The gap: the precious GPU memory and slow switching.

To achieve this goal, however, we face a major challenge—fast GPU context switching between different processes. A modern server can be equipped with several TB of host memory, enabling it to load many applications. However, task execution on GPUs require GPU memory, which is very limited even on high-end GPUs, e.g., 16 GB for T4 and 32 GB for V100. More importantly, GPU memory is purposed for task execution, not for storing the state of idle applications. DL tasks, especially training, require a large amount, or even all of the memory on a GPU.

Storing the models in the GPU like Salus [7] cannot support training tasks which are memory-intensive or even multiple inference tasks which have large models. This is particularly important as state-of-the-art models are getting deeper and larger, and thus even idle applications can occupy large memory space. In addition, request batching is prevalently used to increase throughput [3], which further increases the GPU memory requirement of inference applications. Ideally, the active application should be able to utilize the entire GPU memory for its purpose, and the number of applications that can be served by a GPU server should only be limited by its host memory size. Consequently, switching a task would require heavy memory swapping.

Unfortunately, many online inference workloads require strict SLOs that naive memory swapping between the host memory and the GPU memory cannot meet. For example, we test the strawman scenario where we stop a training task and then start an inference task. The first inference batch would require several seconds to finish (§4.1). Existing support such as NVIDIA MPS is not optimized for DL workloads, and incurs hundreds of milliseconds overhead (§6).

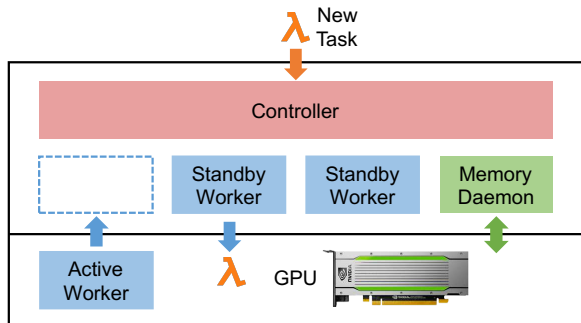


Figure 1: PipeSwitch architecture.

The opportunity: DL workloads have well-defined structures. Fortunately, the structure and computation pattern of DNN models allow us to highly optimize task switching and achieve millisecond-scale overhead. DNN models are usually *deep*, consisting of multiple layers stacking one on another. Furthermore, the computation of DNN models takes place layer by layer as well. Thus, it is possible to build a pipeline that overlaps the computation and GPU memory swapping for fast context switching.

In the following sections, we will show that such pipeline is indeed feasible and effective. In addition, we will also need to resolve other challenges like memory management and worker switching. Combining all the ideas into our system, PipeSwitch, we close the gap of GPU memory sharing and switching, and enable the design of an efficient time-sharing GPU cluster for DL workloads.

3 PipeSwitch Overview

PipeSwitch enables GPU-efficient multiplexing of multiple DL applications on GPU servers. It exploits the characteristics of DL applications to achieve *millisecond-scale* task switching overhead in order to satisfy SLO requirements. Such fast task switching enables more flexible fine-grained scheduling to improve GPU utilization for dynamic workloads. It benefits switching not only between inference and training, but also between inference on different models. Here we provide an overview of the architecture and task execution.

System architecture. Figure 1 shows the architecture of a PipeSwitch server. This server contains four types of components: a controller, a memory daemon, an active worker, and multiple standby workers.

- **Controller.** The controller is the central component. It receives tasks from clients, and controls the memory daemon and the workers to execute the tasks.
- **Memory daemon.** The memory daemon manages the GPU memory and the DNN models. It allocates the GPU memory to the active worker, and transfers the model from the host memory to the GPU memory.
- **Active worker.** The active worker is the worker that currently executes a task in the GPU. Here a worker is a process that executes tasks on one GPU.

Instance Type	g4dn.2xlarge	p3.2xlarge
GPU Type	NVIDIA T4	NVIDIA V100
Task Cleaning	155 ms	165 ms
Task Initialization	5530 ms	7290 ms
Memory Allocation	10 ms	13 ms
Model Transmission	91 ms	81 ms
Total Overhead	5787 ms	7551 ms
Inference Time	105 ms	32 ms

Table 1: Measurement results of task switching overhead and the breakdown of individual components. *All* components should be optimized to meet the SLOs.

- **Standby worker.** A server has one or more standby workers. A standby worker is idle, is initializing a new task, or is cleaning its environment for the previous task.

Task execution. The controller queues a set of tasks received from the clients. It uses a scheduling policy to decide which task to execute next. It supports canonical scheduling policies such as first come first serve (FCFS) and earliest deadline first (EDF), and can be easily extended to support new policies. We focus on fast context switching, and the specific scheduling algorithm is orthogonal to this paper. The scheduling is *preemptive*, i.e., the controller can preempt the current task for the next one based on the scheduling policy. For example, if the current task is a training task, the controller can preempt it for an inference task that has a strict latency SLO.

To start a new task, the controller either waits for the current task to finish (e.g., if it is inference) or preempts it by notifying the active worker to stop (e.g., if it is training). At the same time, the controller notifies an idle standby worker to initialize its environment for the new task. After the active worker completes or stops the current task, the controller notifies the memory daemon and the standby worker to load the task to GPU to execute with pipelined model transmission (§4.2). The memory daemon allocates the memory to the standby worker (§4.3), and transmits the model used by the new task from the host memory to the GPU memory. The standby worker becomes the new active worker to execute the new task, and the active worker becomes a standby worker and cleans the environment for the previous task (§4.4). The primary goal of this paper is to design a set of techniques based on the characteristics of DL applications to minimize the task switching overhead in this process.

4 PipeSwitch Design

We first perform a measurement study to profile the task switching overhead and break it down to individual components. Then we describe our design to systematically minimize the overhead of each component.

4.1 Profiling Task Switching Overhead

In order to understand the problem, we perform a measurement study to profile the task switching overhead. The mea-

surement considers a typical scenario that a server stops a training task running on the GPU, and then starts an inference task. The DNN model used in the measurement is ResNet152 [17]. The measurement covers two types of instances on Amazon AWS, which are g4dn.2xlarge with NVIDIA T4 and p3.2xlarge with NVIDIA V100. We assume the inference task has arrived at the server, and focus on measuring the time to start and execute it on the GPU. We exclude the network time and the task queuing time.

Table 1 shows the results. The total times to start the inference task on the GPUs are 5787 ms and 7551 ms, respectively. We break the overhead down into the four components.

- **Task cleaning.** The training task stops and cleans its GPU environment, such as freeing the GPU memory.
- **Task initialization.** The inference task creates and initializes its environment (i.e., process launching, PyTorch CUDA runtime loading, and CUDA context initialization).
- **Memory allocation.** The inference task allocates GPU memory for its neural network model.
- **Model transmission.** The inference task transmits the model from the host memory to the GPU memory.

The inference time on V100 is lower than that on T4, and both of them are significantly lower than the total overheads. The reason for lower overhead on T4 is that task switching largely depends on CPU, and g4dn.2xlarge is equipped with better CPU than p3.2xlarge (Intel Platinum 8259CL vs. Intel Xeon E5-2686 v4). A strawman solution that simply stops the old task and starts the new task would easily violate SLOs.

Because all the components take considerable time compared to the inference time, we emphasize that *all* the components should be optimized to achieve minimal switching overhead and meet the SLOs.

4.2 Pipelined Model Transmission

Transmitting a task from CPU to GPU is bounded by the PCIe bandwidth. The PCIe bandwidth is the physical limit on how fast an arbitrary task can be loaded to the GPU. We exploit the characteristics of DL applications to *circumvent* this physical limit. Our key observation is that DNN models have a *layered* structure. The computation is performed layer by layer. An inference task only performs a forward pass from the first layer to the final layer to make a prediction; each iteration in a training task performs a forward pass and then a backward pass. In both cases, a task does not need to wait for the entire model to be transmitted to the GPU before beginning the computation. Instead, the task can start the computation of a layer as soon as the layer is loaded in the GPU and the input of the layer is ready (i.e., the previous layers have finished their computation), regardless of its following layers. Figure 2 illustrates the advantage of pipelining over the strawman solution.

PipeSwitch requires the knowledge of models. PipeSwitch does not modify the model structure, and only adds hooks for PyTorch to wait for transmission or synchronize the execution.

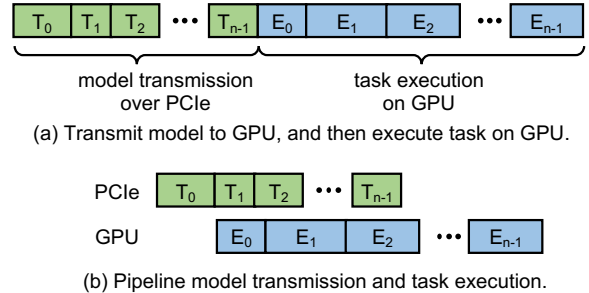


Figure 2: PipeSwitch pipelines model transmission and task execution. The example shows an inference task that only has a forward pass in task execution.

Adding hooks can be automated, and PipeSwitch can be implemented as a part of the DNN framework, e.g., PyTorch, so it can gather the model structure information while remaining transparent to users and cluster managers.

Optimal model-aware grouping. The basic way for pipelining is to pipeline on *per-layer* granularity, i.e., the system transmits the layers to the GPU memory one by one, and the computation for a layer is blocked before the layer is transmitted. Pipelining brings two sources of system overheads. One is the overhead to invoke multiple calls to PCIe to transmit the data. For a large amount of data (e.g., combining the entire model to a large tensor to transmit together), the transmission overhead is dominated by the data size. But when we divide the model into many layers, invoking a PCIe call for each layer, especially given that some layers can be very small, would cause significant extra overhead. The other is the synchronization overhead between transmission and computation, which is necessary for the computation to know when a layer is ready to compute. Pipelining on per-layer granularity requires synchronization for every layer.

We use grouping to minimize these two sources of overhead. We combine multiple layers into a group, and the pipelining is performed on *per-group* granularity. In this way, the pipelining overhead is paid once for each group, instead of each layer. Grouping introduces a trade-off between pipelining efficiency and pipelining overhead. On one hand, using small groups (e.g., per-layer in the extreme case) enables more overlap between transmission and computation, which improves pipelining efficiency, but it also pays more pipelining overhead. On the other hand, using big groups (e.g., the entire model in one group in the extreme case) has minimal pipelining overhead, but reduces the chance for overlapping.

Grouping must be model-aware, because models have different structures in terms of the number of layers and the size of each layer. Naively, we can enumerate all possible combinations to find the optimal grouping strategy. This is not amenable because large models can have hundreds of layers and the time complexity for enumeration is exponential.

In order to find the optimal grouping strategy efficiently, we introduce two pruning techniques based on two insights.

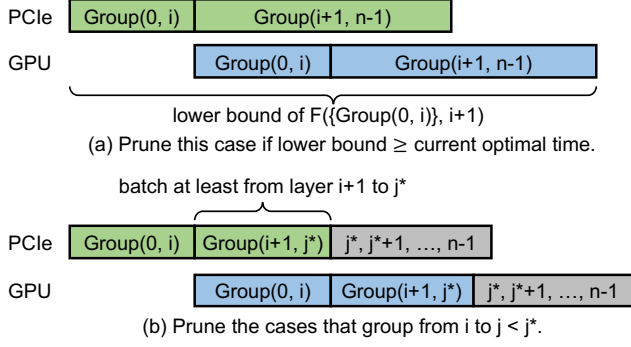


Figure 3: Examples for two pruning techniques.

Before we dive into the details, we first formulate the problem. Let the number of layers be n . Let $F(B, i)$ be a function that returns the total time of the optimal grouping strategy from layer i to $n-1$ given that layer 0 to $i-1$ have formed groups represented by B . Then we have the following recursive formula.

$$F(\{\}, 0) = \min_i F(\{group(0, i)\}, i+1) \quad (1)$$

Specifically, to find the optimal grouping strategy for the entire model (i.e., $F(\{\}, 0)$), we divide all possible combinations into n cases based on how the first group is formed, i.e., case i means the first group contains layer 0 to i . This formula can be applied recursively to compute $F(\{group(0, i)\}, i+1)$.

Our first insight is that it is not necessary to examine all the n cases, because if the first group contains too many layers, the computation of the first group would be delayed too much to compensate the pipeline efficiency. Let $T(i, j)$ and $E(i, j)$ be the transmission and execution times for a group from layer i to j respectively, where $T(i, j)$ is calculated based on the size of layer i to j and PCIe bandwidth, and $E(i, j)$ is profiled on the GPU. Note that the overhead of invoking multiple calls is included in $T(i, j)$. As illustrated by Figure 3(a), we compute a lower bound for the total time for each case in Equation 1.

$$F(\{group(0, i)\}, i+1) \geq \min(T(0, i) + T(i+1, n-1), T(0, i) + E(0, i) + E(i+1, n-1)) \quad (2)$$

The lower bound considers the best case that all the remaining layers are combined in one group for transmission and computation, and that the computation and communication can be perfectly overlapped, i.e., its computation can happen right after the computation of the first group finishes. If the lower bound of case i is already larger than the total time of the best grouping strategy found so far, then case i (i.e., the recursive computation for $F(\{group(0, i)\}, i+1)$) can be pruned.

Our second insight is that other than the first group, we can safely pack multiple layers in a group based on the progress of computation without affecting pipeline efficiency. Figure 3(b) shows an example for this insight. Suppose that we have already fixed the first group to be from layer 0 to i , and we apply Equation 1 recursively to enumerate the cases for the

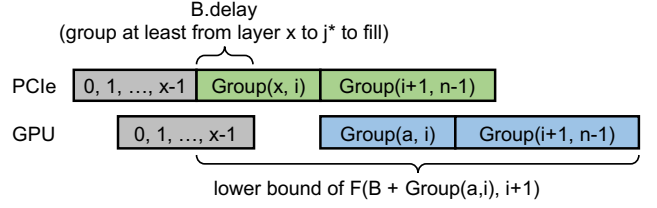


Figure 4: General case for the two pruning techniques.

second group. We can hide the transmission of the second group into the computation of the first group, as long as the transmission finishes no later than the computation of the first group. The least number of layers to group can be computed using the following equation.

$$j^* = \arg \max_j T(i+1, j) \leq E(0, i) \quad (3)$$

Group from layer $(i+1)$ to $j < j^*$ is no better than grouping from $(i+1)$ to j^* because it does not increase the pipeline efficiency and has higher pipeline overhead. Therefore, we can prune the cases that group from layer $(i+1)$ to $j < j^*$ and only search for $j \geq j^*$.

Algorithm. Based on these two insights, we design an algorithm to find the optimal grouping strategy for a given model. We emphasize that this algorithm runs *offline* to find the strategy, and the resulting strategy is used online by PipeSwitch for context switching. Algorithm 1 shows the pseudo code. The function *FindOptGrouping* recursively finds the optimal grouping strategy based on Equation 1 (line 1-27). It takes two inputs: B represents the groups that have already formed, x is the first layer that have not formed a group. It uses *opt_groups* to store the best grouping strategy from layer x given B , which is initialized to none (line 2). The algorithm applies the second pruning insight to form the first group from layer x (line 3-9). Equation 3 and Figure 3(b) illustrate this insight with a special example that B only contains one group from layer 0 to i . In general, B can contain multiple groups formed by previous layers, and we use $B.delay$ to denote the time to which the group can be formed, as shown in Figure 4. The algorithm finds j^* based on $B.delay$ (line 4-9), and the enumeration for i can skip the layers from x to j^*-1 (line 11). For case i , the algorithm applies the first insight to compute the lower bound (line 12-17). Again, the example in Equation 2 and Figure 3(a) is a special case when x is 0. For the general case, the computation from x has to wait for both its transmission (i.e., $T(x, i)$) and the computation of the previous groups (i.e., $B.delay$), as shown in Figure 4. If the lower bound is already bigger than the current optimal time, then case i is pruned (line 18-19). Given the group from layer x to i is formed, the function recursively applies itself to find the optimal groups from layer $i+1$ to $n-1$ (line 21-23), and updates *opt_groups* if the current strategy is better (line 24-26). Finally, it returns *opt_groups* (line 27). In practice, we use a heuristic that bootstraps *opt_groups* with a relative

Algorithm 1 Optimal Model-Aware Grouping

```
1: function FINDOPTGROUPING( $B, x$ )
2:    $opt\_groups \leftarrow \emptyset, opt\_groups.time \leftarrow \infty$ 
3:   // find first group from layer  $i$  to  $j^*$ 
4:    $j^* \leftarrow x$ 
5:   for layer  $i$  from  $x$  to  $n - 1$  do
6:     if  $T(x, i) \leq B.delay$  then
7:        $j^* \leftarrow i$ 
8:     else
9:       break
10:  // recursively find the optimal grouping
11:  for layer  $i$  from  $j^*$  to  $n - 1$  do
12:    if  $opt\_groups \neq \emptyset$  then
13:      // compute lower bound
14:       $trans\_time \leftarrow T(x, i) + T(i + 1, n - 1)$ 
15:       $exec\_time \leftarrow \max(T(x, i), B.delay)$ 
16:         $+ E(x, i) + E(i + 1, n - 1)$ 
17:       $lower\_bound \leftarrow \min(trans\_time, exec\_time)$ 
18:      if  $lower\_bound > opt\_groups.time$  then
19:        continue
20:    // recursively find rest groups
21:     $first\_group \leftarrow Group(x, i)$ 
22:     $rest\_groups \leftarrow FindOptGrouping($ 
23:       $B + first\_group, i + 1)$ 
24:     $cur\_groups \leftarrow first\_group + rest\_groups$ 
25:    if  $cur\_groups.time < opt\_groups.time$  then
26:       $opt\_groups \leftarrow cur\_groups$ 
27:  return  $opt\_groups$ 
```

good strategy (e.g., group every ten layers). Given n layers, there are 2^{n-1} different grouping strategies, so the time complexity of Algorithm 1 is $O(2^n)$, as in the worst case it needs to enumerate all strategies. The two pruning techniques are able to prune most of the strategies, and can quickly find the optimal one as we will show in §6. We have the following theorem for the algorithm.

Theorem 1. *Algorithm 1 finds the optimal grouping strategy that minimizes the total time for the pipeline.*

Proof. Algorithm 1 computes the recursive function $FindOptGrouping(B, x)$. Let $m = n - x$, which is the number of layers the function considers. We use induction on m to show that $FindOptGrouping(B, x)$ outputs the optimal grouping strategy from layer x to $n - 1$ given that previous layers have formed groups represented by B .

Base case. When $m = 1$, the function only examines one layer. Because there is only one strategy which is layer x itself is one group, this strategy is the optimal strategy.

Inductive step. Assume that for some $k \geq 1$ and any $m \leq k$, $FindOptGrouping(B, x)$ outputs the optimal strategy. Consider $m = k + 1$, i.e., the algorithm now considers $k + 1$ layers. The algorithm divides the problem into $k + 1$ cases, where case i ($0 \leq i \leq k$) forms the first group from layer x to $x + i$.

For case i where $0 \leq i \leq k - 1$, because $FindOptGrouping(B + Group(x, x + i), x + i + 1)$ only considers $k - i \leq k$ layers, it outputs the optimal grouping strategy for case i based on the assumption.

For case $i = k$, the first group contains all layers from x to $n - 1$. The optimal strategy for this case is one group.

Because these cases are exclusive and cover the entire search space, by choosing the optimal grouping strategy from these cases, the algorithm outputs the optimal grouping strategy for $m = k + 1$.

The algorithm uses two pruning techniques. The first technique prunes the cases if their lower bounds are no better than the current found optimal. It is obvious that this technique does not affect the optimality. The second technique prunes the case if their first groups are from layer x to $j < j^*$. Because these cases cannot advance the computation to an earlier point than grouping from x to at least j^* , pruning these cases also do not affect the optimality. \square

Generality. Algorithm 1 achieves optimality for a given list of layers. This, however, does not require the models to be linear. In general, the layers or operators in a DNN model can be connected as an arbitrary computation graph, instead of a simple chain. Models like ResNet and Inception are technically non-linear directed acyclic graph (DAGs). Yet, there is an execution order that the layers/operators in the DAG are issued to the GPU one by one. Algorithm 1 does not have any special assumptions on the execution order. It is only interested in finding out how to group the layers given the execution order (and corresponding data dependencies) to achieve high pipelining efficiency and low pipelining overhead. It even applies for graphs with loops, in which the order is based on the first time an operator is executed. The order does not affect correctness, because an operator can be executed only when it is transmitted to the GPU and the input is ready. Thus, our pipelined model transmission is applicable to the general case.

4.3 Unified Memory Management

Task execution in a GPU requires GPU memory. A GPU has its own memory management system, and provides a `malloc` function (e.g., `cudaMalloc` for NVIDIA GPUs) similar to CPUs for memory allocation. NVIDIA also provides CUDA unified memory [4] to automatically handle memory movement between the host memory and the GPU memory for applications. A naive solution for GPU memory management is that each task uses the native `cudaMallocManaged` function for GPU memory allocation, and delegates model transmission to CUDA unified memory. This solution incurs high overhead for DL applications because of two reasons. First, DL applications have large models and generate large amounts of intermediate results, which require a lot of GPU memory. Second, the native `cudaMalloc` function and CUDA unified memory are designed for general-purpose applications, and may incur unnecessary overhead for DL applications.

We exploit two characteristics of DL applications to minimize GPU memory management overhead. A DL task stores two important types of data in the GPU memory: the DNN model (including the model parameters), and the intermediate results. First, the amount of memory allocated to the DNN model is *fixed*, and does not change during task execution. An

inference task only uses the model for inference, and does not change the model itself. While a training task updates the model, it only updates the model parameters (i.e., the weights of the neural network), not the DNN structure, and the amount of memory needed to store them stays the same.

Second, the intermediate results change in a simple, regular pattern, which do not cause *memory fragmentation*. For an inference task, the intermediate results are the outputs of each layer, which are used by the next layer. After the next layer is computed, they are no longer needed and can be safely freed. A training task differs in that the intermediate results generated in the forward pass cannot be immediately freed, because they are also used by the backward pass to update the weights. However, the backward pass consumes the intermediate results in the reverse order as that the forward pass generates them, i.e., the intermediate results are first-in-last-out. The memory allocation and release can be handled by a simple stack-like mechanism, without causing memory fragmentation. The general-purpose GPU memory management does not consider these characteristics, and is too heavy-weight for DL applications that require fast task switching.

Minimize memory allocation overhead. Based on these two characteristics, we design a memory management mechanism tailored for DL applications. PipeSwitch uses a dedicated memory daemon to manage the GPU memory. To be compatible with the existing system and incur minimal changes, instead of replacing the GPU memory manager, the memory daemon uses `cudaMalloc` to obtain the GPU memory when the system starts, and then dynamically allocates the memory to the workers at runtime. This eliminates the overhead for each worker to use `cudaMalloc` to get a large amount of memory to store their models and intermediate results. The memory daemon only needs to pass memory pointers to the workers, which is light-weight. The daemon ensures that each time only one worker owns the GPU memory to guarantee memory isolation between workers. Each worker uses a memory pool to allocate the memory to store its model and intermediate results, and recycles the memory to the pool after the intermediate results are no longer needed.

The memory management of PipeSwitch extends that of PyTorch. It is designed and optimized for efficient GPU memory allocation between different tasks, while the memory management in PyTorch handles memory allocation for a task itself. PipeSwitch inserts GPU memory blocks to PyTorch GPU memory pool, and PyTorch creates tensors on them.

Minimize memory footprint and avoid extra memory copies. The server stores the DNN models in the host memory. Replicating the models in each worker incurs high memory footprint, and reduces the number of models a server can store, and consequently the types of tasks the server can execute. On the other hand, storing the models in a dedicate process has minimal memory footprint as each model is only stored once, but it incurs an extra memory copy from this process to a

worker to start a task, which hurts the task switching time. We use unified memory management with the memory daemon to both achieve minimal memory footprint and eliminate extra memory copies. PipeSwitch stores the models in the memory daemon so that the server only needs to keep one copy of each model in the host memory. Because the memory daemon also manages the GPU memory, it directly transmits the model from the host memory to the GPU memory for task startup, which eliminates the extra memory copy from the memory daemon to the worker.

Minimize IPC overhead. After the model is transmitted to the GPU, the memory daemon needs to notify the worker and export the relevant GPU memory handlers to the worker, so that the worker can access the model to execute its task. This can be implemented by IPC APIs provided by GPUs, e.g., `cudaIpcOpenMemHandle` for NVIDIA GPUs. We have measured the performance of these IPC APIs and found that they incur high overhead (§6). The overhead is exacerbated by the pipeline because the pipeline needs to invoke the IPCs frequently to synchronize model transmission and task execution for *every pipeline group*, instead of invoking the IPC only once for the entire model transmission.

We leverage a property of DL applications to minimize the IPC overhead. The property is that the memory allocation process for a neural network model is *deterministic*. Specifically, given the same GPU memory region and the same model, as long as the memory daemon and the worker uses the same *order* to allocate memory for the model parameters, the memory pointers for the parameters would be the *same*. It is easy to keep the same order for the memory daemon and the worker because the neural network model is known and given, and the memory daemon only needs to use the same order to transmit the model as the worker would. As a result, the memory daemon can minimize the usage of expensive GPU IPCs. It only uses the GPU IPC once to initialize the worker, and then uses cheap CPU IPCs to notify the worker which pipeline group has been transmitted.

Pin memory. The OS would swap a memory page to disk if the page is inactive for a certain amount of time. GPUs require a page in the host memory to be pinned (or page-locked) in order to transmit the data in the page to the GPU memory. Otherwise, a temporary pinned page is created for the transmission. We pin the pages of the memory daemon to the host memory, to eliminate this overhead.

4.4 Active-Standby Worker Switching

PipeSwitch aims to provide fast task switching and ensure process-level isolation. Process-level isolation is desirable because it ensures that one task cannot read the memory of another task, and that the crashing of one task, e.g., because of a bug, does not affect other tasks or the entire system.

A naive solution is to use separate processes and start the new task after the current task is stopped. As we have profiled

	No Task Cleaning Overhead	No Task Initialization Overhead	Process- Level Isolation
Two Processes	×	×	✓
One Process	×	✓	×
Active-Standby	✓	✓	✓

Table 2: Comparison of worker switching mechanisms.

in Table 1, such sequential execution incurs long delay due to old task cleaning and new task initialization.

Another possible solution is to let the current and new tasks share the same process with a warm CUDA context, so that the new task can reuse the GPU environment of the current task. This avoids the new task initialization, but it still has the overhead for the current task to clean its status. In addition, it does not provide process-level isolation between tasks.

We design an active and standby worker switching mechanism that hides the overhead of both task cleaning and task initialization, and also ensures process-level isolation. Similar to the naive solution, we use separate processes to achieve process-level isolation. PipeSwitch has an active worker and multiple standby workers. Each worker is a separate process, and initializes its own GPU environment (i.e., CUDA context) when it is first created. This eliminates the GPU environment initialization overhead when a new task is assigned to a worker. When a current task is stopped, a major job is to clear asynchronous CUDA functions queued on the GPU. We insert synchronization points into training tasks, so the number of queued functions are limited and can be quickly cleared. Synchronization points are not needed for inference tasks as they are short and not preempted. Another job is to free its GPU memory. An important property of the cleaning procedure is that it does not modify the content of the memory, but only cleans the metadata, i.e., GPU memory pointers. As the GPU memory is managed by PipeSwitch, the cleaning procedure deletes the pointers pointing to the tensor data rather than freeing the actual data. Therefore, it is safe for the new task to transmit its model to the GPU memory at the same time. In other words, we can parallelize the task cleaning of the current task and the pipelined model transmission of the new task, to hide the task cleaning overhead. This choice is optimized for performance, and is not a problem for a trusted environment. It is possible that a latter process can read the memory data of a previous process. If this is a concern, an additional zero-out operation can be added. GPU has high memory bandwidth (e.g., 900GB/s for V100). It would incur sub-millisecond overhead for zeroing-out most models like ResNet-152 (around 240MB). On the other hand, for a trusted environment, it is unnecessary to release all allocated memory for the preempted process if the new process does not require entire GPU memory, and this could be achieved by some simple coordination. Table 2 summarizes the differences between these three solutions.

In summary, to switch workers, the controller signals the current active worker to stop, deletes the GPU memory allocated to it, and allocates the GPU memory to the new active worker. The controller ensures only one active worker to guarantee exclusive occupation of the GPU.

There is a trade-off between the number of standby workers and their GPU memory consumption. On one hand, task cleaning takes time. If a new task arrives before a standby worker finishes cleaning a previous task, the new task needs to wait, which increases its startup time. On the other hand, it is possible to have many standby workers so that there is always at least one idle standby worker. However, every standby worker needs to maintain its own CUDA context, which consumes a few hundred MB GPU memory. Our experience is that two standby workers are sufficient to ensure at least one idle worker, which eliminates the waiting time and has moderate GPU memory consumption.

4.5 Discussion

PipeSwitch is focused on single-GPU tasks for training and inference. For inference tasks, strict SLOs require requests to be handled in small batches for low latency, so it is common to execute an inference task with a single GPU [18]. Multi-GPU inference tasks can be supported by performing PipeSwitch on each GPU with *transactions*. A transaction here means a model is switched in or out on all of its GPUs to enable or disable inference on this model.

For training tasks, PipeSwitch supports single-GPU training and asynchronous multi-GPU training for data parallel strategies, as preempting one GPU does not affect other GPUs. However, it does not work out of the box with synchronous multi-GPU training. We have analyzed a production GPU training trace from Microsoft [19, 20]. Among 111,883 tasks in this trace, 96,662 tasks (or 86% of all the tasks) are single-GPU training tasks. Thus, a significant fraction of tasks in real-world workloads currently use a single GPU, and PipeSwitch is applicable to them out of the box. However, these jobs only account for 18% of total GPU hours and we expect the share of multi-GPU jobs to increase in the future. One way to seamlessly use PipeSwitch for synchronous multi-GPU training is to use elastic synchronous training, which allows the dynamic changing of the number of GPUs used for training. Unfortunately, current training frameworks do not have mature support of elastic training. This remains an active research topic and is orthogonal to PipeSwitch.

5 Implementation

We have implemented a system prototype for PipeSwitch with ~ 3600 lines of code in C++ and Python, and we have integrated it with PyTorch [21].

PyTorch Plugins. We add C++ and Python functions to the GPU memory management module of PyTorch. To share GPU memory between the controller and the workers, we add functions for allocating GPU memory, sharing the GPU

memory to workers through CUDA IPC API, and getting the shared GPU memory. We also add functions which insert the received GPU memory into PyTorch GPU memory pool for a specific CUDA stream or clear the GPU memory from the pool. Note that the shared GPU memory can be inserted into the PyTorch GPU memory pool for multiple times for different CUDA streams, and the controller guarantees that only one of these CUDA streams is active.

Controller and memory daemon. The controller process consists of a TCP thread and a scheduler thread. For better performance, the scheduler and the memory daemon are implemented together. The TCP thread accepts task through TCP from clients, and sends the task to the scheduler thread. The scheduler thread allocates and shares the GPU memory with workers, activates or deactivates workers, sends the task to a worker, and transfers parameters for the corresponding model to the GPU memory. Before starting a task, the user should register the model in the scheduler to notify the controller to load the model from the disk to the CPU memory. When the controller schedules a task, it determines whether to switch to another worker. There is no need for context switching if the application is already loaded in the GPU. If a new model should be loaded to the GPU, the controller will notify the current active worker to stop, and transfers the parameters of the new model to the GPU after receiving the current active worker’s reply. Parameters are transmitted to the GPU memory in groups in a pipeline. After each group is transferred, the controller notifies the worker to start computing the corresponding layers.

Worker. The worker process consists of two threads. The termination thread waits for the termination signal from the controller, and notifies the main thread. The main thread manages the DNN models and performs the computation for inference or training. Similar to the controller, the worker also requires the user to register the model before starting a task, so the worker can load the models and add the hooks to wait for parameter transmission or terminate on notification. Note that the worker only loads the model structures, which is small, not the model parameters. The parameters are only stored once in the memory daemon for minimal memory footprint. When the models are loaded, they are attached to different CUDA streams, and their parameters are assigned to locations in the shared GPU memory. Different models might use the same GPU memory location, but the value is not valid until the controller transfers the corresponding parameters to these locations. After loading the models, the worker waits for the scheduler to transfer required parameters for DNN models, and performs inference or training.

6 Evaluation

In this section, we first use end-to-end experiments to demonstrate the benefits of PipeSwitch, and then show the effectiveness of the design choices on each component.

Setup. All experiments are conducted on AWS. We use two EC2 instance types. One is p3.2xlarge, which is configured with 8 vCPUs (Intel Xeon E5-2686 v4), 1 GPU (NVIDIA V100 with 16 GB GPU memory), PCIe 3.0 \times 16, and 61 GB memory. The other is g4dn.2xlarge, which is configured with 8 vCPUs (Intel Platinum 8259CL), 1 GPU (NVIDIA T4 with 16 GB GPU memory), PCIe 3.0 \times 8, and 32 GB memory. The software environment includes PyTorch-1.3.0, torchvision-0.4.2, scipy-1.3.2, and CUDA-10.1. We use PyTorch with our plugins for all mechanisms in comparison for consistency, which provides better results for stop-and-start than native PyTorch from Python-PyPI used in Table 1.

Workloads. The models include ResNet152 [17], Inception_v3 [22] and Bert_base [23], which are standard benchmarks for evaluating DL systems. We use representative configurations for each model. The experiments cover both training and inference. We use single-GPU inference and training tasks as discussed in §4.5. Training tasks periodically checkpoint their models to the host memory, and restart from the latest checkpoint after preemption. The checkpointing frequency of training tasks is set according to the scheduling cycle to minimize checkpointing overhead. The default batch size for training is 32, and that for inference is 8.

Metrics. We use throughput and latency as evaluation metrics. Each number is reported with the *average* of 100 runs. For Figure 6(b), we additionally report the minimum and maximum latencies using the error bar, because the latency of the first batch and those of later batches in one scheduling cycle can differ significantly due to switching overhead.

6.1 End-to-End Experiments

Minimizing end-to-end overhead. In this experiment, a client sends an inference task to a GPU server, and the GPU server preempts the training task to execute the inference task and sends a reply back to the client. We measure the end-to-end latency experienced by the client. We compare the following mechanisms.

- **Ready model.** There is no training task. The process with the required model is already loaded in the GPU. This solution provides the lower bound, which is the lowest latency we can achieve for an inference task.
- **Stop-and-start.** It stops the training task in the GPU, and then starts the inference task. This solution is used by existing systems like Gandiva [24] for task switching, which reported similar *second-scale* overhead.
- **NVIDIA MPS.** This is the multi-process support from NVIDIA which allows the inference process to share the GPU with the training process. We initialize separate processes in advance. The training task occupies the entire GPU memory and does not stop when inference tasks come. CUDA unified memory is used for memory swapping.
- **PipeSwitch.** This is the proposed system. The properties are described in §4.

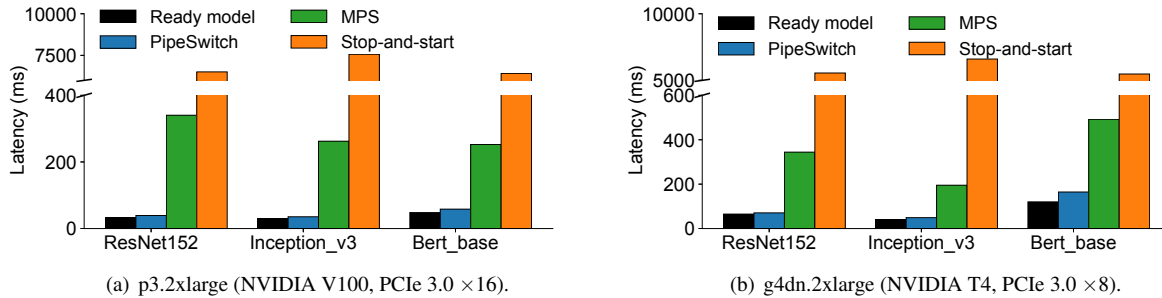


Figure 5: Total latency experienced by the client for different mechanisms.

	p3.2xlarge (NVIDIA V100, PCIe 3.0 × 16)			g4dn.2xlarge (NVIDIA T4, PCIe 3.0 × 8)		
	ResNet152	Inception_v3	Bert_base	ResNet152	Inception_v3	Bert_base
Stop-and-start	6475.40 ms	7536.07 ms	6371.32 ms	5486.74 ms	6558.76 ms	5355.95 ms
NVIDIA MPS	307.02 ms	232.25 ms	204.52 ms	259.20 ms	193.05 ms	338.25 ms
PipeSwitch	6.01 ms	5.40 ms	10.27 ms	5.57 ms	7.66 ms	34.56 ms

Table 3: Total overhead, i.e., the difference on total latency between different mechanisms and ready model.

	ResNet152	Inception_v3	Bert_base
p3.2xlarge	3.62 ms	4.82 ms	3.62 ms
g4dn.2xlarge	2.53 ms	5.49 ms	6.57 ms

Table 4: The startup overhead for PipeSwitch to start computing the first layer.

Salus [7] is not directly comparable because it requires the models to be preloaded to the GPU, and has several limitations described in §2.2. Its performance is similar to the ready model when the model is preloaded, and is similar to NVIDIA MPS when the model is in the host memory. Figure 5 shows the latency experienced by the client, and Table 3 shows the total overhead. The total overhead is the difference between the latency of a mechanism and that of the *ready model*. It is obvious that stop-and-start performs the worst, which takes several seconds. The main source of the overhead is CUDA context initialization and first-time library loading operations in PyTorch. NVIDIA MPS has lower overhead compared to stop-and-start, but still incurs several hundred milliseconds overhead, which prevents MPS from meeting strict SLOs. One source of the overhead is the contentions both on the computation and memory of the GPU, as the training task do not stop when an inference task comes. Another source is GPU memory swapping. PipeSwitch performs the best and is close to the lower bound. The overhead of PipeSwitch for most configurations is up to 10ms, except for BERT on T4, which is due to the large model size and the smaller PCIe bandwidth on T4 than that on V100. Since it also takes longer (120ms) to compute BERT on T4 even with the ready model, the relative overhead is acceptable.

We also show the task startup overhead for PipeSwitch in Table 4, which is the difference between the time for

	ResNet152	Inception_v3	Bert_base
# of Layers	464	189	139
Algorithm 1	1.33 s	0.18 s	0.34 s
Only Pruning 1	2.09 s	0.30 s	0.88 s
Only Pruning 2	3.44 h	5.07 s	> 24 h
No Pruning	> 24 h	> 24 h	> 24 h

Table 5: Effectiveness of two pruning techniques.

PipeSwitch to start computing the first layer and that for the ready model to start computing. The startup overhead of PipeSwitch is only a few milliseconds.

Enabling fine-grained scheduling cycles. In this experiment, we compare throughput and end-to-end latency of different mechanisms under different scheduling cycles. We use ResNet152 for both training and inference on eight p3.2xlarge instances, and switch between these two tasks after each scheduling cycle. Figure 6(a) shows the inference throughput. The dashed line is the upper bound, which is the throughput of the ready model assuming no task switching. The throughput of stop-and-start is nearly zero for scheduling cycles smaller than 10 s, because it takes several seconds for task switching. MPS keeps poor throughput around 100 batches per second. We define GPU utilization as the ratio to the upper bound. PipeSwitch has high throughput close to the upper bound, achieving near 100% GPU utilization.

Figure 6(b) shows the average latency of the inference tasks. The dashed line is the lower bound, which is the average latency of the ready model assuming no task switching. The error bar indicates the minimum and maximum latency. Stop-and-start has poor latency because the first batch has several seconds overhead. MPS has about 80 ms average latency, and has several hundred milliseconds latency for the first batch.

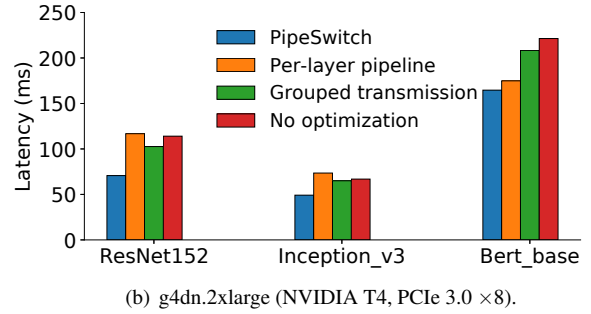
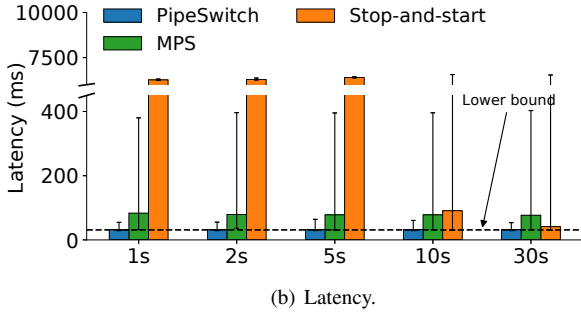
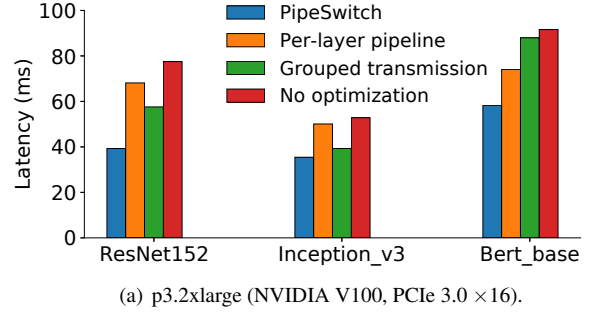
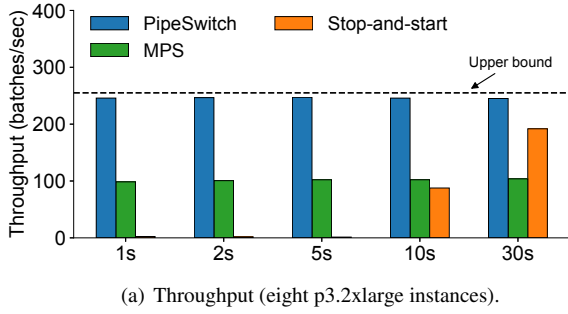


Figure 6: Throughput and latency under different scheduling cycles for ResNet on p3.2xlarge.

PipeSwitch incurs only a few milliseconds overhead for task switching, and achieves low latency close to the lower bound.

6.2 Pipelined Model Transmission

To evaluate the effectiveness of pipelined model transmission, we keep all other components of PipeSwitch the same, and compare the following mechanisms discussed in §4.2.

- **No optimization.** It transmits the model layer by layer (with many PCIe calls), and then executes the task.
- **Grouped transmission.** It groups the entire model in one transmission, and then executes the task.
- **Per-layer pipeline.** It transmits model parameters layer by layer. Computation starts, once parameters are transmitted.
- **PipeSwitch.** It is the pipelining mechanism with optimal model-aware grouping in PipeSwitch.

Figure 7 shows the total time measured by the client for an inference task to preempt a training task and finish its inference. No optimization performs the worst in most cases. Grouped transmission improves no optimization by combining the layers of the model into one big tensor and transmitting it in one group. Per-layer pipeline overlaps transmission and computation at the granularity of layer. But because it has PCIe overhead and synchronization overhead for every layer, for the models with many layers but relatively light computation such as ResNet152 and Inception, it can perform worse than grouped transmission and sometimes even no pipeline. PipeSwitch uses model-aware grouping and achieves the best trade-off between pipeline overhead and efficiency. It reduces the total time by up to 38.2 ms compared to other solutions.

Figure 7: Effectiveness of pipelined model transmission.

Note that this reduction is significant, especially considering that it is evaluated when the optimizations on memory management and worker switching have already been applied. We would like to emphasize that to meet strict SLOs, it is important to reduce all overheads for task switching, not only the most significant one.

Table 5 shows the running time of Algorithm 1, as well as the effects of the two pruning techniques mentioned in § 4.2. Note that the number of layers includes both weighted and unweighted layers, as both contribute to the computation time. We measure the parameter size and running time for each layer in advance. Algorithm 1 takes only several seconds to compute an optimal grouping strategy, even for ResNet152 which has hundreds of layers. On the contrary, no pruning does not finish for all three models after running for 24 hours.

6.3 Unified Memory Management

To evaluate the effectiveness of unified memory management, we keep all other components of PipeSwitch the same, and compare the following five mechanisms discussed in §4.3.

- **No unified memory management.** Each worker uses `cudaMalloc` to allocate GPU memory, and transmits the model to GPU by its own.
- **No IPC optimization.** The memory daemon handles GPU memory allocation and model transmission, but creates and sends GPU memory handlers to workers. To compare, PipeSwitch simply sends a 64-bit integer offset for the shared GPU memory to workers.
- **No pin memory.** It has all optimizations on unified memory management except that the pages of the memory daemon are not pinned to the main memory.

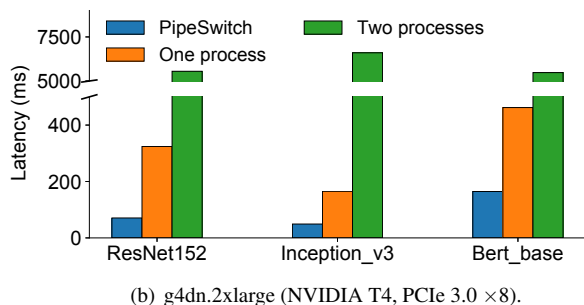
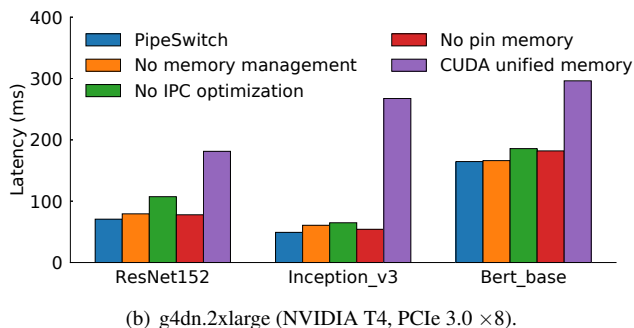
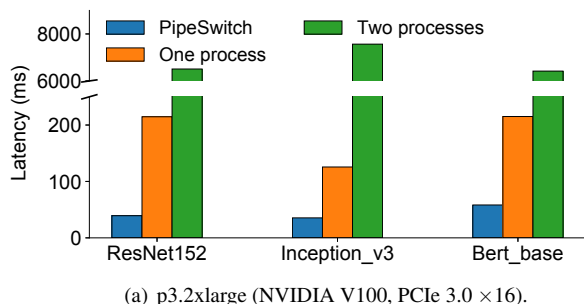
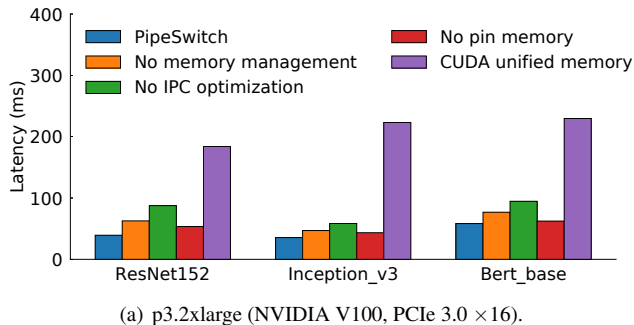


Figure 8: Effectiveness of unified memory management.

- **CUDA unified memory.** Each worker allocates GPU memory with `cudaMallocManaged`, and CUDA automatically transmits the model to GPU when needed.
- **PipeSwitch.** It is the unified memory management mechanism used by PipeSwitch.

Figure 8 shows the total time measured by the client. First, compared to no unified memory management, PipeSwitch saves 2–23 ms by eliminating the memory allocation overhead with the memory daemon. It is also important to note that no unified memory management requires each worker to keep a copy for each DNN model, which increases the memory footprint. Second, IPC optimization is important, which reduces the latency by 16–48 ms. Without IPC optimization, the latency is even higher than no unified memory management. Third, pinning the pages to the host memory can reduce the latency with a few milliseconds. Finally, CUDA unified memory is not optimized for DL applications, and introduces more than one hundred milliseconds overhead than PipeSwitch. Overall, this experiment demonstrates that all the optimizations on memory management are effective.

6.4 Active-Standby Worker Switching

To evaluate the effectiveness of active-standby worker switching, we keep all other components of PipeSwitch the same, and compare the following mechanisms discussed in §4.4.

- **Two processes.** The process of the old task cleans the GPU environment, and then another process is created and initialized for the new task.
- **One process.** The process cleans the GPU environment for the old task, and reuses the environment for the new task.

Figure 9: Effectiveness of active-standby switching.

- **PipeSwitch** It is the active-standby worker switching mechanism used by PipeSwitch.

Figure 9 shows the results. Two processes perform the worst as it stops the training task and initializes a new process for the new task. The new process needs to create a new CUDA environment, which dominates the total time. One process reuses the CUDA environment, but still pays the overhead to clean the environment. PipeSwitch uses an active-standby worker switching mechanism to parallelize old task cleaning and new task initialization, and incurs minimal overhead. It reduces the latency by 116–307 ms compared to one process, and 5–7 s compared to two processes.

7 Related Work

Many frameworks have been developed for deep learning, such as TensorFlow [25], PyTorch [21] and MXNet [26]. Several algorithms and systems have been designed for executing and scheduling deep learning tasks on clusters, including both training and inference tasks [3, 10, 24, 27–32]. These scheduling solutions are orthogonal and complementary to PipeSwitch. They focus on what scheduling decisions to make, while PipeSwitch focuses on how to realize a scheduling decision. Importantly, PipeSwitch enables the scheduler to change the resource allocation more often with millisecond-scale task switching. Many techniques and systems have been proposed to optimize communication and improve distributed training [8, 9, 15, 33–42]. The most relevant ones are PipeDream [8], ByteScheduler [9] and Poseidon [40]. They use inter-batch pipelining for training of the same task, while PipeSwitch introduces intra-batch pipelining to fast start both

training and inference tasks and enables fast switching across tasks. Other works like vDNN [43] and SwapAdvisor [44] also have GPU memory management module, but they focus on memory management for a single training task of large models, which are not directly comparable to PipeSwitch.

Cluster managers [45–48] typically allocate GPUs to VMs or containers at device granularity. Several solutions have been proposed to share a GPU at application granularity using techniques like library interception [6, 49–53]. They are general-purpose and focus on sharing only a few kernels. As such, they are not suitable for deep learning applications that typically require hundreds of kernels. NVIDIA MPS [6] provides official support for sharing a GPU between multiple processes. It is also not specially designed for deep learning and thus cannot meet strict SLOs of inference tasks as shown in §6. There are many efforts on GPU optimization to improve the performance of running a single task, such as tensor fusion and kernel-level concurrency and scheduling [54–58]. These solutions are complementary to PipeSwitch.

8 Conclusion

We present PipeSwitch, a system that enables GPU-efficient fine-grained time-sharing for multiple DL applications. We introduce pipelined context switching to minimize task switching overhead on GPUs for DL applications. Pipelined context switching includes three key techniques, which are pipelined model transmission, unified memory management and active-standby worker switching. With these techniques, PipeSwitch is able to achieve millisecond-scale task switching time, and enables DL applications on time-sharing GPUs to meet strict SLOs. We demonstrate the performance of PipeSwitch with experiments on a variety of DNN models and GPU cards. PipeSwitch can significantly increase GPU utilization and improve the agility of DL applications.

Acknowledgments. We thank our shepherd Madan Musuvathi and the anonymous reviewers for their valuable feedback. Zhihao Bai, Zhen Zhang and Xin Jin were supported in part by an AWS Machine Learning Research Award.

References

- [1] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes, “Large-scale cluster management at Google with Borg,” in *EuroSys*, 2015.
- [2] J. Dean and L. A. Barroso, “The tail at scale,” *Communications of the ACM*, vol. 56, 2013.
- [3] H. Shen, L. Chen, Y. Jin, L. Zhao, B. Kong, M. Philipose, A. Krishnamurthy, and R. Sundaram, “Nexus: A GPU cluster engine for accelerating DNN-based video analysis,” in *ACM SOSP*, 2019.
- [4] “CUDA Unified Memory.” <https://devblogs.nvidia.com/unified-memory-cuda-beginners/>.
- [5] A. Ousterhout, J. Fried, J. Behrens, A. Belay, and H. Balakrishnan, “Shenango: Achieving high CPU efficiency for latency-sensitive datacenter workloads,” in *USENIX NSDI*, 2019.
- [6] “CUDA Multi-Process Service.” https://docs.nvidia.com/deploy/pdf/CUDA_Multi_Process_Service_Overview.pdf.
- [7] P. Yu and M. Chowdhury, “Salus: Fine-grained GPU sharing primitives for deep learning applications,” in *Conference on Machine Learning and Systems*, 2020.
- [8] D. Narayanan, A. Harlap, A. Phanishayee, V. Seshadri, N. R. Devanur, G. R. Ganger, P. B. Gibbons, and M. Zaharia, “PipeDream: generalized pipeline parallelism for DNN training,” in *ACM SOSP*, 2019.
- [9] Y. Peng, Y. Zhu, Y. Chen, Y. Bao, B. Yi, C. Lan, C. Wu, and C. Guo, “A generic communication scheduler for distributed DNN training acceleration,” in *ACM SOSP*, 2019.
- [10] J. Gu, M. Chowdhury, K. G. Shin, Y. Zhu, M. Jeon, J. Qian, H. Liu, and C. Guo, “Tiresias: A GPU cluster manager for distributed deep learning,” in *USENIX NSDI*, 2019.
- [11] M. Jeon, S. Venkataraman, A. Phanishayee, u. Qian, W. Xiao, and F. Yang, “Analysis of large-scale multi-tenant GPU clusters for DNN training workloads,” in *USENIX ATC*, 2019.
- [12] “Amazon Web Services.” <https://aws.amazon.com/>.
- [13] “Microsoft Azure.” <https://azure.microsoft.com/>.
- [14] “Google Cloud Platform.” <https://cloud.google.com/>.
- [15] A. Sergeev and M. Del Balso, “Horovod: fast and easy distributed deep learning in tensorflow,” *arXiv preprint arXiv:1802.05799*, 2018.
- [16] 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 *USENIX OSDI*, 2014.
- [17] K. He, X. Zhang, S. Ren, and J. Sun, “Deep residual learning for image recognition,” in *IEEE Conference on Computer Vision and Pattern Recognition*, 2016.
- [18] “Nvidia data center deep learning product performance.” <https://developer.nvidia.com/deep-learning-performance-training-inference>.

- [19] M. Jeon, S. Venkataraman, A. Phanishayee, J. Qian, W. Xiao, and F. Yang, "Analysis of large-scale multi-tenant GPU clusters for DNN training workloads," in *USENIX ATC*, 2019.
- [20] "Philly traces." <https://github.com/msr-fiddle/philly-traces>.
- [21] "PyTorch." <https://pytorch.org/>.
- [22] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, "Rethinking the inception architecture for computer vision," in *IEEE Conference on Computer Vision and Pattern Recognition*, 2016.
- [23] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of deep bidirectional transformers for language understanding," in *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, 2019.
- [24] W. Xiao, R. Bhardwaj, R. Ramjee, M. Sivathanu, N. Kwatra, Z. Han, P. Patel, X. Peng, H. Zhao, Q. Zhang, *et al.*, "Gandiva: Introspective cluster scheduling for deep learning," in *USENIX OSDI*, 2018.
- [25] "TensorFlow." <https://www.tensorflow.org/>.
- [26] "MXNet." <https://mxnet.apache.org/>.
- [27] H. Zhang, L. Stafman, A. Or, and M. J. Freedman, "Slaq: quality-driven scheduling for distributed machine learning," in *ACM Symposium on Cloud Computing*, 2017.
- [28] Y. Peng, Y. Bao, Y. Chen, C. Wu, and C. Guo, "Optimus: an efficient dynamic resource scheduler for deep learning clusters," in *EuroSys*, 2018.
- [29] K. Mahajan, A. Balasubramanian, A. Singhvi, S. Venkataraman, A. Akella, A. Phanishayee, and S. Chawla, "Themis: Fair and efficient GPU cluster scheduling," in *USENIX NSDI*, 2020.
- [30] R. Liaw, R. Bhardwaj, L. Dunlap, Y. Zou, J. E. Gonzalez, I. Stoica, and A. Tumanov, "HyperSched: Dynamic resource reallocation for model development on a deadline," in *ACM Symposium on Cloud Computing*, 2019.
- [31] R. Dathathri, O. Saarikivi, H. Chen, K. Laine, K. Lauter, S. Maleki, M. Musuvathi, and T. Mytkowicz, "CHET: An optimizing compiler for fully-homomorphic neural-network inferencing," in *ACM Conference on Programming Language Design and Implementation*, 2019.
- [32] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan, H. Shen, M. Cowan, L. Wang, Y. Hu, L. Ceze, *et al.*, "TVM: An automated end-to-end optimizing compiler for deep learning," in *USENIX OSDI*, 2018.
- [33] Y. Huang, Y. Cheng, A. Bapna, O. Firat, D. Chen, M. Chen, H. Lee, J. Ngiam, Q. V. Le, Y. Wu, *et al.*, "Gpipe: Efficient training of giant neural networks using pipeline parallelism," in *Advances in Neural Information Processing Systems*, 2019.
- [34] G. Wang, S. Venkataraman, A. Phanishayee, J. Thelin, N. Devanur, and I. Stoica, "Blink: Fast and generic collectives for distributed ML," in *Conference on Machine Learning and Systems*, 2020.
- [35] "NVIDIA Collective Communications Library (NCCL)." <https://developer.nvidia.com/nccl>.
- [36] J. Liu, J. Wu, and D. K. Panda, "High performance RDMA-based MPI implementation over infiniband," *Int. J. Parallel Program.*, vol. 32, 2004.
- [37] Q. Ho, J. Cipar, H. Cui, S. Lee, J. K. Kim, P. B. Gibbons, G. A. Gibson, G. Ganger, and E. P. Xing, "More effective distributed ML via a stale synchronous parallel parameter server," in *Advances in Neural Information Processing Systems*, 2013.
- [38] A. A. Awan, C.-H. Chu, H. Subramoni, and D. K. Panda, "Optimized broadcast for deep learning workloads on dense-GPU infiniband clusters: MPI or NCCL?," in *Proceedings of the 25th European MPI Users' Group Meeting*, 2018.
- [39] J. Daily, A. Vishnu, C. Siegel, T. Warfel, and V. Amaty, "GossipGraD: Scalable deep learning using gossip communication based asynchronous gradient descent," *CoRR*, vol. abs/1803.05880, 2018.
- [40] H. Zhang, Z. Zheng, S. Xu, W. Dai, Q. Ho, X. Liang, Z. Hu, J. Wei, P. Xie, and E. P. Xing, "Poseidon: An efficient communication architecture for distributed deep learning on GPU clusters," in *USENIX ATC*, 2017.
- [41] Z. Zhang, C. Chang, H. Lin, Y. Wang, R. Arora, and X. Jin, "Is network the bottleneck of distributed training?," in *ACM SIGCOMM Workshop on Network Meets AI & ML (NetAI)*, August 2020.
- [42] Y. Chen, Z. Liu, B. Ren, and X. Jin, "On efficient constructions of checkpoints," in *International Conference on Machine Learning (ICML)*, July 2020.
- [43] M. Rhu, N. Gimelshein, J. Clemons, A. Zulfiqar, and S. W. Keckler, "vDNN: Virtualized deep neural networks for scalable, memory-efficient neural network design," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2016.
- [44] C.-C. Huang, G. Jin, and J. Li, "SwapAdvisor: Pushing deep learning beyond the GPU memory limit via smart swapping," in *ACM ASPLOS*, 2020.

- [45] “Kubernetes.” <https://kubernetes.io/>.
- [46] “NVIDIA Container Runtime for Docker.” <https://github.com/NVIDIA/nvidia-docker>.
- [47] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. H. Katz, S. Shenker, and I. Stoica, “Mesos: A platform for fine-grained resource sharing in the data center,” in *USENIX NSDI*, 2011.
- [48] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, *et al.*, “Apache Hadoop YARN: Yet another resource negotiator,” in *ACM Symposium on Cloud Computing*, 2013.
- [49] G. Giunta, R. Montella, G. Agrillo, and G. Coviello, “A GPGPU transparent virtualization component for high performance computing clouds,” in *European Conference on Parallel Processing*, 2010.
- [50] V. Gupta, A. Gavrilovska, K. Schwan, H. Kharche, N. Tolia, V. Talwar, and P. Ranganathan, “GVim: GPU-accelerated virtual machines,” in *Proceedings of the 3rd ACM Workshop on System-level Virtualization for High Performance Computing*, 2009.
- [51] J. Duato, A. J. Pena, F. Silla, R. Mayo, and E. S. Quintana-Ortí, “rCUDA: Reducing the number of GPU-based accelerators in high performance clusters,” in *2010 International Conference on High Performance Computing & Simulation*, 2010.
- [52] V. T. Ravi, M. Becchi, G. Agrawal, and S. Chakradhar, “Supporting GPU sharing in cloud environments with a transparent runtime consolidation framework,” in *Proceedings of the 20th international symposium on High performance distributed computing*, 2011.
- [53] L. Shi, H. Chen, J. Sun, and K. Li, “vCUDA: GPU-accelerated high-performance computing in virtual machines,” *IEEE Transactions on Computers*, vol. 61, 2011.
- [54] “TensorFlow XLA.” <https://www.tensorflow.org/xla/>.
- [55] 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.
- [56] C. Gregg, J. Dorn, K. Hazelwood, and K. Skadron, “Fine-grained resource sharing for concurrent GPGPU kernels,” in *Presented as part of the 4th USENIX Workshop on Hot Topics in Parallelism*, 2012.
- [57] S. Pai, M. J. Thazhuthaveetil, and R. Govindarajan, “Improving GPGPU concurrency with elastic kernels,” *ACM SIGARCH Computer Architecture News*, vol. 41, 2013.
- [58] Z. Jia, O. Padon, J. Thomas, T. Warszawski, M. Zaharia, and A. Aiken, “TASO: optimizing deep learning computation with automatic generation of graph substitutions,” in *ACM SOSP*, 2019.