SwitchFlow: Preemptive Multitasking for Deep Learning

Xiaofeng Wu
The University of Texas at Arlington
xiaofeng.wu@uta.edu

Jia Rao
The University of Texas at Arlington
jia.rao@uta.edu

Wei Chen
Nvidia Corporation
weich@nvidia.com

Hang Huang
Huazhong University of Science and Technology
huanghang@hust.edu.cn

Chris Ding
The University of Texas at Arlington
chqding@uta.edu

Heng Huang
University of Pittsburgh
heng.huang@pitt.edu

ABSTRACT

Accelerators, such as GPU, are a scarce resource in deep learning (DL). Effectively and efficiently sharing GPU leads to improved hardware utilization as well as user experiences, who may need to wait for hours to access GPU before a long training job is done. Spatial and temporal multitasking on GPU have been studied in the literature, but popular deep learning frameworks, such as TensorFlow and PyTorch, lack the support of GPU sharing among multiple DL models, which are typically represented as computation graphs, heavily optimized by underlying DL libraries, and run on a complex pipeline spanning CPU and GPU. Our study shows that GPU kernels, spawned from computation graphs, can barely execute simultaneously on a single GPU and time slicing may lead to low GPU utilization.

This paper presents SwitchFlow, a scheduling framework for DL multitasking. It centers on two designs. First, instead of scheduling a computation graph as a whole, SwitchFlow schedules its subgraphs and prevents subgraphs from different models to run simultaneously on a GPU. This results in less interference and the elimination of out-of-memory errors. Moreover, subgraphs running on different devices can overlap with each other, leading to a more efficient execution pipeline. Second, SwitchFlow maintains multiple versions of each subgraph. This allows subgraphs to be migrated across devices at a low cost, thereby enabling low-latency preemption. Results on representative DL models show that SwitchFlow achieves up to an order of magnitude lower tail latency for inference requests collocated with a training job.

CCS CONCEPTS

• **Computing methodologies** → **Machine learning**; • **Computer systems organization** → **Multicore architectures**.

KEYWORDS

Deep learning framework, preemption scheduling, systems for machine learning

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
Middleware'21, December 6–10, 2021, Virtual Event, Canada
© 2021 Association for Computing Machinery.
ACM ISBN 978-1-4503-8534-3/21/12…$15.00
https://doi.org/10.1145/3464298.3493391

ACM Reference Format:
Xiaofeng Wu, Jia Rao, Wei Chen, Hang Huang, Chris Ding, and Heng Huang. 2021. SwitchFlow: Preemptive Multitasking for Deep Learning. In *Middleware'21, December 6–10, 2021, Québec, Canada.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3464298.3493391

1 INTRODUCTION

Recent advances in deep learning (DL) [24] have led to the wide adoption of machine learning techniques in image classification [19, 24], speech recognition [18], and natural language processing [15]. The success of deep learning can be partially attributed to the enormous amount of data available for model training and the advent of fast graphics processing units (GPUs) that allows more sophisticated models (e.g., deep neural networks (DNNs)) to be trained at a speed an order of magnitude faster than that on CPUs. The surge of deep learning has also given rise to deep learning frameworks, such as TensorFlow [4] and PyTorch [41], which make programming complex models not only easier but also more efficient on various accelerators (e.g., GPUs, TPUs [23], and FPGAs).

As deep learning continues to gain popularity and is increasingly deployed in cloud services [25, 37], there is a growing need for sharing accelerators [7, 8, 31, 34, 38, 48, 50] (e.g., GPUs) among multiple deep learning workloads. Multitasking has been a key feature in modern computing systems to share a single device. *Spatial multitasking* partitions resources among multiple tasks and executes them simultaneously if their combined size can fit in the device. *Temporal multitasking* assigns each task a time quantum during which the device is dedicated to one task at a time. Both mechanisms are proven effective for improving GPU utilization [5, 8, 21, 38, 52]. However, multitasking deep learning workloads, which are inherently more complex than simple GPU kernels, presents unique challenges.

First, DNNs written with DL frameworks contain complex execution flows, typically in the form of a computation graph with thousands of nodes. Each node in the graph is a mathematical operation to be executed on either CPU or GPU. Multitasking deep learning workloads requires that the scheduling of the CPU and GPU nodes (kernels) in the same graph be coordinated. The existing architectural support for GPU multitasking is limited to concurrent execution of independent kernels on GPU hardware, thereby unable to handle the multitasking of complex deep learning models due to the lack of knowledge of computation graphs from multiple users. There are existing works that control the launching of GPU kernels in the runtime to enable GPU sharing [11, 50]. However, switching computation graphs (i.e., DL models) is non-trivial. DL frameworks support two graph execution modes. 1) *Dynamic* *graph* mode, the default execution mode in PyTorch and TensorFlow (also known as eager execution), generates graph nodes on-the-fly as model execution proceeds. It allows for evaluating the output of graph nodes immediately after its execution, thereby offering an intuitive programming interface and facilitating debugging. 2) In contrast, *static graph* mode builds the entire graph before model execution and performs offline graph optimizations. The resulted graph allows for node merging, reordering, and concurrent node execution and is significantly faster than dynamic graphs, especially for large models. Multitasking models with static graphs is much more challenging because node execution does not follow user's code and is asynchronous and interleaved.

Pipeswitch [8] leverages **dynamic graphs** in PyTorch to enable fast model switching via pipelined model transmission. It relies on layer-by-layer model execution in dynamic graphs to overlap model transmission and execution, which is critical to efficient DL multitasking. In this paper, we take the challenge to support DL multitasking on **static graphs**, which are widely adopted in production systems due to high performance, efficient computation graph optimization, ease of co-design of hardware acceleration and compiler optimization.

Second, DL frameworks, such as TensorFlow (TF), rely heavily on machine learning libraries, e.g., NVIDIA cuBLAS and cuDNN [12, 33] to accelerate frequently used routines in DNNs, such as convolution and matrix multiplication. The DL libraries carefully tune GPU kernels based on GPU resource availability, such as the number of streaming multiprocessors (SMs), cores per SM, and the size of device memory. Since there lack mechanisms for dynamically reconfiguring GPU resources, the tuning must be performed before model execution. Therefore, users need to explicitly set resource limits, e.g., memory size, for each DL model. This requires that either DL models be allocated with statically partitioned resources to allow concurrent model execution or the entire GPU should be allocated to one model and models have to be executed one after another. While TF allows for dynamic memory growth, which allocates GPU memory only when models actually use it, TF does not support reclaiming GPU memory until model execution is completed. Thus, it is not suitable for DL multitasking. Recent work AntMan [52] realizes elastic memory management for DL models based on GPU unified memory [30] and allows model data to be freely allocated on both GPU and host memory. However, AntMan does not address job preemption and can only switch DL jobs at the completion of mini-batches. In practice, DNN training jobs are usually allocated dedicated GPUs [1, 16, 22] while multiple inference jobs may be packed on a single GPU [37]. As a result, training jobs cannot share a GPU for lack of memory and inference jobs may experience high latency waiting for training jobs to complete due to the lack of an effective preemption mechanism.

Third, new challenges and opportunities arise surrounding multitasking DL workloads: 1) Like in conventional workload collocation, DL multitasking should meet different service-level objectives (SLOs) for heterogeneous workloads. While model training is throughput oriented and requires high resource utilization, model serving (i.e., inference) has stringent latency requirements. However, model training is significantly more resource-intensive than inference, not only requiring an order of magnitude more GPU memory but also computing power for model parameter updates (i.e., updated weights) through iterations. In addition, inference could experience high latency due to the long preemption latency of training. 2) New multitasking scenarios emerge as DL continues to evolve [28]. Multi-task learning [44] trains multiple models from the same training data set. For example, separate models should be trained to detect pedestrians and vehicles, respectively, from the same set of sensing data in autonomous driving. Since these DL models share the input and possibly some layers of a DNN, running them on the same GPU opens up new opportunities for exploiting data locality. During model training, a large set of training samples is usually divided into mini-batches, each of which can fit in GPU memory. The existing DL frameworks repeatedly load mini-batches into GPU for training separate models even though each mini-batch can be shared among models. This motivates the development of a new GPU multitasking scheme that allows for fine-grained data reuse on GPU across different models.

In this paper, we present SwitchFlow, a scheduling framework for multitasking DL jobs. We identify **several issues** in static computation graph execution in TensorFlow, the arguably best-performing DL framework for production systems. **First**, computation graph execution typically employs multiple worker threads to exploit concurrency in executing computation graph nodes. There is a lack of an effective and efficient preemption mechanism to enforce priority between different graphs from multiple jobs in static execution mode. **Second**, our empirical study revealed that DL operations, which are optimized and automatically configured by DL libraries, barely can simultaneously execute on a single GPU, though there is ample concurrency in the computation graph. **Third**, graph execution is a complex pipeline spanning CPU and GPU. DL multitasking should efficiently utilize the heterogeneous devices. Existing work such as session-based time slicing [20, 51], which allows models to exclusively access both CPU and GPU and runs them iteration by iteration, leads to low GPU utilization because GPU waits for CPU to feed input data [29] during each iteration.

SwitchFlow addresses these issues through two designs. **First**, SwitchFlow maintains multiple versions of a computation graph, which includes subgraphs that run on different devices. Replicated subgraphs, each individually optimized for different devices (CPU or GPU) for the same computation, enable SwitchFlow to freely migrate the execution of subgraphs between CPU and GPU and vice versa. **Second**, unlike in TensorFlow, wherein nodes in a computation graph are indistinguishably scheduled by workers, SwitchFlow treats nodes to be executed on different devices, e.g., CPU or GPU, differently in scheduling, following two principles: 1) GPU nodes from different models are not scheduled simultaneously on a single GPU, allowing exclusive access to GPU; 2) all other nodes, including CPU nodes and GPU nodes on a different GPU, are allowed to run without restrictions to improve pipeline efficiency.

The result is a design that allows users to provision GPU resources for their models without concerns about interference from other models or memory over-commitment. It also enables a low-latency, low-cost preemption mechanism to deschedule an entire computation graph without throughput loss. Experimental results on representative DL models and three different GPUs show that 1) SwitchFlow achieves a 19.05x tail latency improvement for inference requests when collocated with a heavy-weight training job compared to an variant of TF. 2) SwitchFlow is more efficient than

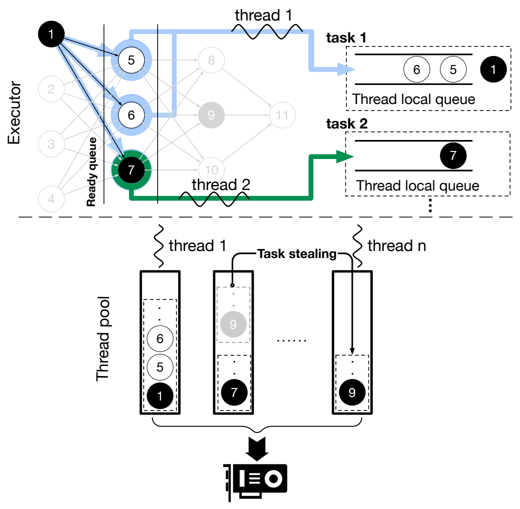


Figure 1: computation graph scheduling in TensorFlow.

the existing time slicing-based approaches in utilizing heterogeneous devices. 3) With user-provided hints, SwitchFlow is able to merge multiple computation graphs of similar models to share the data preprocessing stage and achieves up to 65% performance improvement compared to time slicing in multi-task learning.

2 BACKGROUND AND MOTIVATION

In machine learning frameworks, such as TensorFlow, learning algorithms are represented as *computation graphs* wherein nodes describe operations while edges specify dataflows between those operations. Expressing machine learning models as computation graphs offers several benefits. First, the execution of a learning algorithm can be accelerated by optimizing the directed graph, e.g., pruning, merging, and partitioning. Second, the abstract representation of computation allows operations to be individually implemented using different machine learning libraries, making them portable across heterogeneous devices. For example, a graph can be executed entirely on CPU or on a hybrid CPU/GPU system. Third and most importantly, computation graphs specify the order of execution and allow concurrent operations to be scheduled in parallel.

In what follows, we discuss the challenges of sharing a single GPU among multiple DL models, each with its own computation graph. Without loss of generality, we focus our discussion on the graph execution mode in TensorFlow (TF), which is based on static graphs.

2.1 Executing Computation Graphs in TF

To execute a computation graph, resources need to be provisioned for graph nodes, i.e., operators (*ops*), and a schedule plan needs to be determined to run them on different devices, e.g., CPUs and GPUs, while enforcing node dependency. TF's graph execution centers on three techniques: *session*, *executor*, and *thread pool*.

Session is a runtime instance created by users to execute graph nodes associated with an output node. The target output can be an intermediate node or the final node of a graph. In the former case a subgraph is executed while in the latter the full graph is executed. In deep learning, a `session.run` performs one iteration of training or inference. For training, the parameters (weights) of the model are updated after each session run. During session construction, different devices (e.g., CPU and GPU) are added to a session and a cost model is used to determine the backend device to execute each node. TF makes use of external, highly-optimized numerical libraries, such as MKL [3], cuBLAS [33], and cuDNN [12], to implement operations (kernels) on CPU or GPU. Session also optimizes graph execution by partitioning the full graph into subgraphs, which can be independently executed by *executors*.

Executor dispatches operations from a subgraph to several task queues from where they are executed by worker threads in a **thread pool**. Figure 1 shows how operations are scheduled to run in an executor. Note that there could be multiple executors in a session, each including nodes to be executed on a single device. For example, in a 2-GPU system, there are typically three executors, one for operations running on CPU and one for each of the two GPUs. Executor uses the input size and the type of operation to determine the cost of each node and classifies them into expensive and inexpensive ops. There is a single *ready queue* for each executor wherein nodes in the subgraph are inserted in a breadth-first manner. Initially, all nodes (expensive or inexpensive) in the ready queue are concurrent and dispatched to separate *local queues*, each of which will be processed by a worker from the thread pool. Workers launch ops (kernels) from their local queues in FIFO order. After a node is done, its subsequent nodes which have a direct edge from the current node are inserted in the ready queue. Unlike in the initial dispatch, only expensive nodes require to be placed on a new local queue and inexpensive nodes are sent to the local queues of their parent node. Each local queue is assigned to a worker for node processing and a thread is put to sleep if its queue is empty. Before sleep, a thread performs random work stealing from other threads to balance the load.

2.2 Challenges in Multitasking DL Workloads

Executor-based computation graph scheduling exposes ample concurrency to build an efficient execution pipeline for DL models: 1) Since executors are associated with different devices, their computation is independent from each other and can be done in parallel except for cross-device data transfer. Therefore, stages (subgraphs) for reading input data and preprocessing, which are typically done on CPU, can overlap with training or inference on GPU. 2) Concurrent nodes in a subgraph are processed by multiple workers in parallel. However, expressing and executing DL algorithms using computation graphs present great challenges in sharing computing systems among multiple DL workloads.

Task preemption is the mechanism to suspend the currently executing task, save its states, and switch to another task. It is crucial to enable differentiation and time sharing among tasks. However, there is no effective and efficient preemption mechanism for DL workloads. Due to the parallelism in computation graphs, there could be multiple GPU kernels of the same DL model simultaneously running on GPU or waiting at the launch queue. Additionally, multiple CPU threads processing the CPU executor can run asynchronously with their GPU counterparts on multiple CPUs. To preempt a DL workload, all the three types of tasks associated with the DL model should be suspended and their contexts be saved. The existing hardware

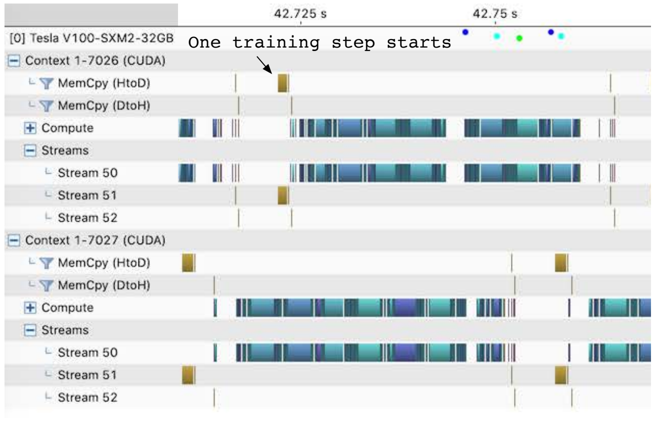


Figure 2: The timeline of training two ResNet50 sharing a single NVIDIA V100 GPU.

mechanisms for GPU context switching [39] lack the knowledge of computation graphs and are limited to preempting a single GPU kernel (node).

Furthermore, DL operations, such as those in DNN training, are memory intensive. The intermediate data generated during model training, e.g., gradients, could be an order of magnitude larger than the input [27, 43]. For a mini-batch of 64-128 images, its input size ranges from tens to hundreds of megabytes (MBs). Suspending tasks of a DL model during training requires to save a context of a few to tens of gigabytes of data, leading to not only high storage cost but also long context saving time. Alternatively, preemption can happen until an iteration (a session in TF) is finished so that only the data that should persist across iterations, such as model weights, is saved. However, as reported in [51] and verified by our experiments, an iteration of training can take up to 1s and add a sizable delay to preemption. Long preemption latency is not acceptable for latency-sensitive DL inference workloads [14, 20, 25, 37].

As DL workloads are executed on complex pipelines across multiple devices, e.g., CPU and GPU, it is non-trivial to efficiently share heterogeneous devices among DL workloads. While there have been extensive studies on spatial [5] and temporal multitasking [38] on GPUs, we demonstrate that DL operations can hardly execute simultaneously on GPU and time slicing GPU can lead to low efficiency.

Ineffective spatial multitasking. Modern GPUs support concurrent execution of several small kernels to improve device utilization. For example, with the help of the Hyper-Q technology [9], NVIDIA multi-process service (MPS) [34] and CUDA streams allow multiple kernels to be launched to multiple hardware work queues. If the kernels are truly independent and their aggregate resource demand fits in the GPU, they can be simultaneously executed. However, spatial multitasking is not effective for DL workloads.

We executed two 2D convolution (`tf.nn.conv2d`) operations, a commonly used routine in DNNs, from two CUDA streams and compared their execution with one stream on a single NVIDIA GPU. We used `nvprof` [35] to collect statistics of primitive routines, such as the block size, number of registers, and used shared memory. Concurrent kernel launch from two streams does not offer much performance benefit. The completion time of two streams is close to executing the two operations sequentially. An analysis of the limiting factors in the kernels using NVIDIA's GPU occupancy calculator [36] revealed that 10 of the 13 kernels were bottlenecked by GPU register files and cannot run concurrently.

We further train two ResNet50 models concurrently on a single NVIDIA V100 GPU with a batch size of 16 on the ImageNet dataset. Figure 2 shows the execution timeline of the two models. The color areas are kernel execution on GPU and white areas are the time spent on CPU. We made two observations: 1) while some kernels of the two model can be simultaneously executed on GPU, their execution times were significantly prolonged due to contentions on shared GPU resources. 2) There still exists significant serialization on GPU between the two models in which kernels from one model exclusively occupied GPU while those from the other model were waiting to be issued by CPU. As a result, the training throughput of individual models dropped from 226 to 116 images per second due to GPU sharing. It suggests that spatial multiplexing is barely beneficial.

The reason no two heavy kernels can execute simultaneously without performance loss even with multi streams in the GPU is that primitive routines in cuDNN or cuBLAS are optimized to fully utilize resources on GPU. Although modern GPUs support resource partitioning and library can adapt kernels to meet the constrains, allocation needs to be done when the computation graph is constructed. As workloads in shared systems are dynamic, static resource partitioning likely leads to underutilization. Alternatively, it is possible to control the resource demand of DNNs during runtime without static resource partitioning. By changing the input size, e.g., batch size, the size of kernels can be dynamically adjusted to fit in GPU. However, changing the batch size may lead to longer training time, negating the benefit of resource sharing. Furthermore, the memory demand of individual kernels need to be carefully controlled not to exceed the capacity of GPU memory. Otherwise, DL jobs may crash due to out-of-memory (OOM) errors.

Inefficient temporal multitasking. Time-slicing GPUs among multiple DL workloads has been explored to provide early feedback in training [51] and better quality-of-service (QoS) for inference [20]. For time sharing, computation graphs are switched at the end of a session. Therefore, during a time slice (consisting of one or more sessions), only the nodes from one graph are executed and GPU is dedicated to one DL workload. However, the DL execution pipeline comprises stages on CPU and GPU. As GPUs continue to improve, the early stages on CPU for data loading and preprocessing will increasingly become the bottleneck [13]. Session-based time slicing dedicates the entire pipeline (both GPU and CPU) to one DL job. If the job cannot efficiently utilize the heterogeneous resources, devices may be left idle.

To demonstrate the severity of pipeline inefficiency, we measured GPU idling periods during training and inference on three NVIDIA GPUs: a cost-effective GPU (GeForce RTX 2080 Ti), a high-end GPU (V100), and a power-efficient embedded GPU (Jetson TX2). We used the TF *timeline* profiler [17] to measure the GPU busy time in a session and the length of the session. The difference between the two is the GPU idling period. Figure 3 shows the execution time breakdown of 9 CNN models. The measurements were the average of 200 sessions in each model. The white area above GPU time refers to GPU idling. The input was randomly selected images in

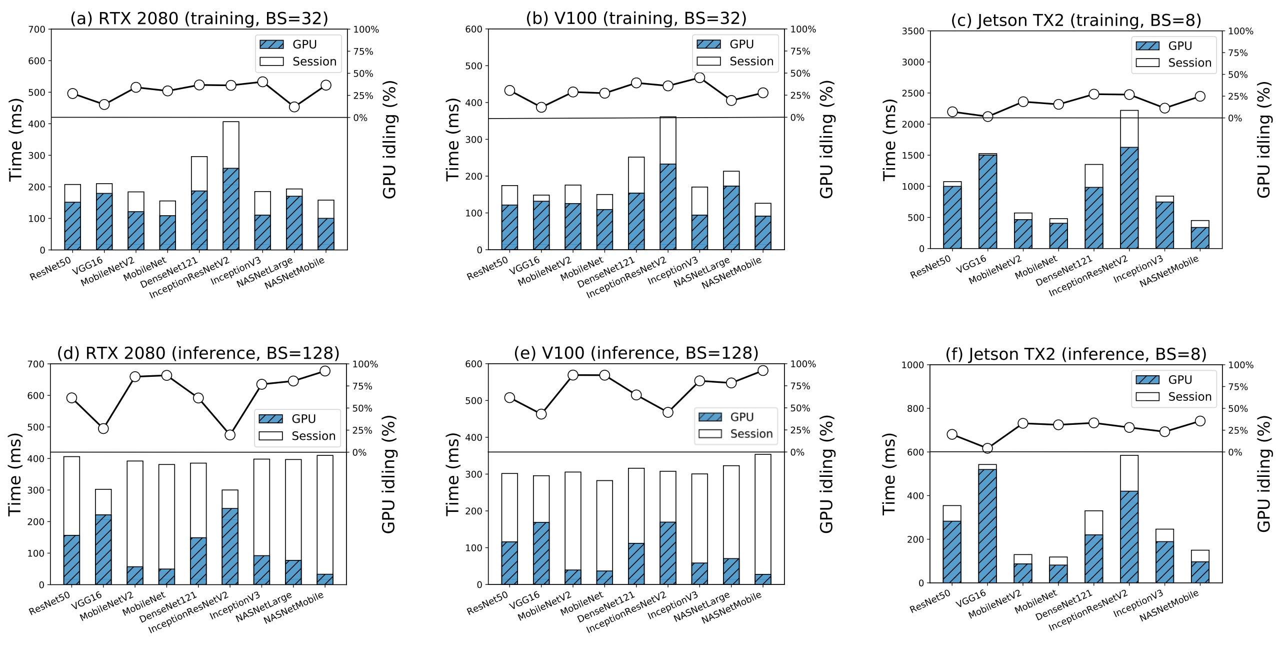


Figure 3: The percentage of GPU idle periods due to inefficient DL execution pipeline on three different types of GPU.

JPEG format from the ImageNet dataset. Images were grouped into batches to improve GPU utilization. We chose the commonly used batch sizes for training (32) and inference (128). `tf.data.Dataset` was used for data preprocessing. Input prefetching and parallel data workers (32 preprocessing threads) were enabled.

As shown in Figure 3 (d) and (e), most models caused long GPU idle periods when performing inference. For instance, in model *NASNetMobile*, for more than 90% of the time, the V100 GPU was idle waiting for CPU to feed data. In contrast, the computation on CPU and GPU can be better overlapped in training, as shown in Figure 3 (a) and (b), because training includes one forward and one backward pass in each session and requires more GPU computation. For the embedded GPU (TX2), GPU was the bottleneck in most models for both training and inference. We also observed two trends. First, faster GPU results in more GPU idling. Second, increasing the batch size leads to more GPU compute time but will further exacerbate GPU underutilization in a session as data preprocessing becomes even longer.

Summary. We have shown the difficulties in running multiple DL jobs on a GPU simultaneously and the low efficiency of GPU time slicing. This motivated us to develop a more flexible and efficient approach for DL multitasking.

3 SWITCHFLOW: SYSTEM DESIGN

3.1 Overview

Deep learning workloads can be broadly categorized into training and serving. Training workloads are throughput-oriented, computationally expensive, and long-term. By contrast, serving workloads have a tight latency requirement but execute for the short term, leading to low utilization in a production environment since online inference queries often arrive unpredictably and stochastically.

The major problem of session-based computation graph execution is the coupling of stages (executors) running on heterogeneous devices. Simultaneously running multiple sessions causes contention on bottleneck devices, making it hard to guarantee QoS and even resulting in OOM crashes due to memory overcommitment. On the other hand, restricting only one session to access computational resources (time slicing) leads to low utilization. In contrary, SwitchFlow views computation graphs as a set of executors that can be flexibly managed and scheduled across sessions.

This design offers several benefits: 1) By replicating executors for each available device during graph construction, the execution of graphs can be timely suspended and migrated, enabling low-latency, low-cost DL job preemption. 2) Executors from similar jobs can be assembled to exploit data reuse in multi-task learning. 3) Executors of different types and from different jobs can be interleaved to efficiently utilize heterogeneous resources.

3.2 Session Management

The central idea in SwitchFlow session management is allowing sessions from any DL jobs to access all available devices on a machine. Unlike TF, in which sessions are statically configured with a fixed number of devices and each session has its own thread pool for graph execution, SwitchFlow shares all devices and a single global thread pool among sessions, as shown in Figure 4. The temporary thread pool is used for fast preemption and will be discussed in Section 3.3. Furthermore, SwitchFlow creates multiple executors, each corresponding to an available device on the machine, for each subgraph during graph construction. Initially, for a subgraph, the executor and

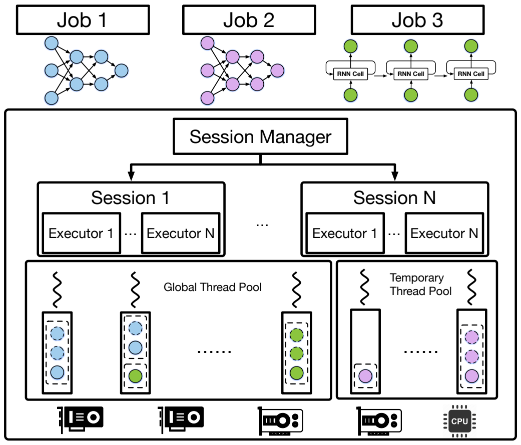


Figure 4: Design of SwitchFlow.

its associated backend device are determined by the ML framework based on a cost model. The additional copies of executors are used for migrating a subgraph from one device to another.

The session manager determines when to schedule executors from sessions while preserving the dependency within a session. It allows executors from different sessions to interleave but ensures that executors from consecutive runs of the same session follow sequential order. SwitchFlow supports independent DL jobs as well as multi-job scenarios, where a user runs a set of jobs on the same training set to tune hyper-parameters, such as the number of layers/weights, mini-batch size, and learning rate, of a model [51] or to train multiple models. In multi-job scenarios, SwitchFlow merges subgraphs from different but correlated sessions based on user-provided configuration.

3.3 Preemption

SwitchFlow addresses several challenges in task preemption. First, to preempt a DL job, all its tasks queued in the ready queue, local thread queues, dispatched onto GPUs, and currently running on CPUs must be stopped in a coordinated and timely manner. Second, the context of the suspended job must be saved and the storage cost should be contained. Third, task resumption should also be fast to avoid throughput loss.

Recall from previous discussions that worker threads independently dispatch tasks and can steal from each other. It is necessary to isolate high priority jobs (preempters) from those to be preempted (preemptees). As shown in Figure 4, SwitchFlow maintains a temporary thread pool to handle the preemptees until preemption is completed. With the help of the executor scheduler (discussed in Section 3.4), SwitchFlow guarantees low preemption latency.

Task suspension. Upon the arrival of a high priority DL job, the session manager first aborts the nodes that are currently queued in the ready queue and thread local queues from the preempted job. The kernels that have been dispatched onto GPU are allowed to finish as they may be interleaved with other jobs' launched kernels and there is a lack of mechanisms to selectively stop kernels of a particular job. Second, the session manager reconstructs the computation graph of the preempted job to replace the executor (subgraph) on the bottleneck device with an alternative executor on a different device. For example, if *job1* is to preempt *job2* on GPU1, *job2*'s executor on GPU1 will be replaced with an executor on GPU2 or CPU. As such, subsequent sessions of *job2* will be run on a different device, isolated from the high priority job. Furthermore, subsequent sessions of the preempted job will be handled by the temporary thread pool until preemption is completed. This ensures that the launching of the new job is not interfered.

After preemption is done, the preempted job can be moved back to the global thread pool. In the case that there is no GPU available and the preempted job has to run on CPU, e.g., using an executor implemented with the Intel MKL library, we keep it in the temporary pool to prevent a large number of MKL operations from exhausting the global thread pool. At initialization, SwtichFlow spawns as many threads as the number of cores in each thread pool and uses wakeup signals to control the number of active threads. Thread count in the temporary pool can be configured by configuration and is a tradeoff between isolation and the performance of preempted jobs. SwitchFlow ensures that the total number of workers in the two thread pools matches the number of cores.

Context saving and task resumption. For training jobs, model weights that persist across iterations (i.e., session runs) need to be saved to preserve training progress. For inference, no cross-session state needs to be saved since prediction requests do not update model weights and are independent. ML frameworks keep model weights in GPU memory across iterations and copy updated weights back to host memory after training is completed. At the end of each iteration, intermediate data, such as calculated gradients, is discarded but weights remain in GPU memory. To save job context, SwitchFlow tracks persistent variables in a session through TF's resource manager on each device. To reduce the delay caused by state transfer, SwitchFlow does not initiate the transfer when preemption is in progress, i.e., the session of a preempted job is being aborted. Instead, SwitchFlow waits until a new session run of the preempted job is started.

A preempted job is migrated to a different device and allowed to resume immediately. The session manager uses the newly constructed graph to start a new session run of the preempted job. The new session is populated with the tasks of the aborted session run so that no work is lost. Most importantly, before the job is resumed, SwitchFlow copies the model weights from the GPU where the job is preempted to the new device using asynchronous memory copy. Note that the state transfer is off the critical path of preemption and can be performed concurrently with the high priority job. However, this requires model weights to be preserved on the source GPU until state transfer finishes, occupying GPU memory that can be used by the new job. We think this is a necessary tradeoff for minimizing preemption latency. As will be shown in Section 5.2.3, intermediate data dominates model memory usage [27, 43] and weights only account for less than 10% of the total memory usage.

3.4 Scheduling

Recall the two issues of computation graph execution: 1) primitive routines (kernels) are highly optimized by DL libraries to improve hardware efficiency, thereby unable to co-run on a single GPU without performance loss; 2) the execution pipeline of a single model

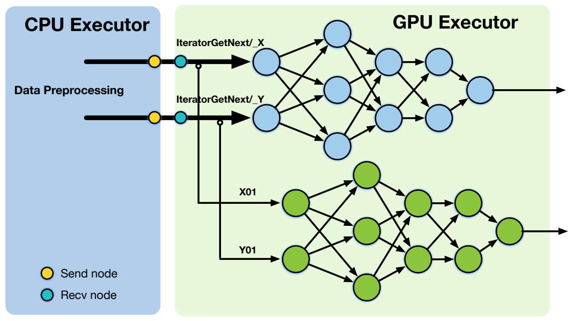


Figure 5: Input data reuse in multi-task learning.

cannot efficiently utilize both CPU and GPU. To this end, SwitchFlow maintains two **scheduling invariants**:

First, no two GPU executors are scheduled on a single GPU simultaneously. It is worth noting that this constraint not only helps more efficiently utilize GPU through time slicing but also effectively avoids OOM errors as well as offering flexibility for resource provisioning. Users can assume they have exclusive access to GPUs they request and configure their models accordingly, e.g., selecting an appropriate batch size. Since model weights need to be preserved on GPU, the aggregate size of persistent variables of all models sharing the same GPU should not exceed GPU memory size. SwitchFlow allows one GPU executor to finish before switching to another. As such, intermediate data is discarded and a majority of GPU memory is freed.

We use preemption to demonstrate how this scheduling constraint achieves low preemption latency without OOM errors. A high priority job is allowed to start immediately after submission. The new job goes through computation graph construction before its executors are ready to run. If the preempted job is still being aborted, its GPU executor is running and will prevent the new job's GPU executor from starting. Therefore, the abort time and preparation time can be overlapped.

Second, executors on different devices can be scheduled freely. SwitchFlow does not impose restrictions on the scheduling of CPU executors or executors on different GPUs. This is in stark contrast to session-based time slicing wherein no executors from other sessions can be scheduled. SwitchFlow allows any CPU executor to run while a GPU is occupied. It helps to overlap data preprocessing on CPU in one job with GPU processing in another job. Note that we did not observe much contention on CPU because when one job's GPU executor runs, its CPU executor only prefetches input for the next session without processing them. Therefore, CPU executors may not reach their peak demands at the same time.

SwitchFlow supports **customized scheduling** as directed by user configuration. In multi-task learning, users use the same input to train or perform inference on multiple models. During hyperparameter tuning, the same input data is used to navigate the hyperparameter space, e.g., learning rate, momentum, and dropout rate, on the same model. For these use cases, a straightforward approach is to replicate the input and run multiple jobs separately. Not only will it lead to redundant data preprosessing but it also results in low pipeline efficiency. Research in DL showed that multi-task learning can be achieved by sharing the hidden layers of a neural network among multiple models while keeping model-specific output layers [10].

Listing 1: The `launch.py` program for sharing the data preprocessing stage between two models.

```python
# Setup
os.environ['TF_SET_REUSE_INPUTS'] = 'True'
os.environ['TF_REUSE_INPUT_OP_NAME_MASTER_X'] = 'X00'
os.environ['TF_REUSE_INPUT_OP_NAME_MASTER_y'] = 'y00'

# For a master and a secondary model(X01,y01)
os.environ['TF_REUSE_INPUT_OPS_NAME_SUB_X'] = 'X01'
os.environ['TF_REUSE_INPUT_OPS_NAME_SUB_y'] = 'y01'

def launch():
  # master graph
  t0 = threading.Thread(
    name='t0', target=user_00.BuildAndRunGraph,
    args=('graph_00', 'X00', 'y00'))

  # secondary graph
  t1 = threading.Thread(
    name='t1', target=user_01.BuildAndRunGraph,
    args=('graph_01', 'X01', 'y01')
```

However, the internal structure of these models must be similar and they have to be deployed together.

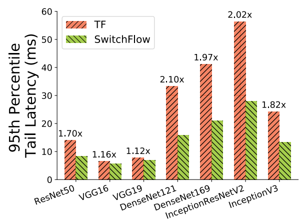
As shown in Figure 5, SwitchFlow offers an alternative way to jobs with same input pipeline. It merges multiple computation graphs to share the data preprocessing stage. Specifically, the `recv` nodes on GPU executors are linked together to share the tensors received from the CPU executor. Note that the input tensor may be modified during GPU processing and is deallocated after the GPU executor finishes, SwitchFlow maintains an immutable copy of the tensor in GPU global memory and makes its address public to all GPU executors sharing the tensor. Models are executed in lockstep. All models should finish processing an input tensor before moving onto the next input batch. To this end, SwitchFlow executes a strict schedule: a shared CPU executor for data loading and preprocessing followed by each model's GPU executor in a round-robin manner.

4 IMPLEMENTATION

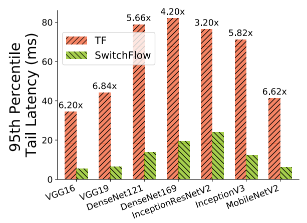
In this work, we have implemented a prototype of SwitchFlow in TensorFlow. We made changes to TF with 3K+ lines of Python and C++ code. Most changes were made to TF's session and executor management as well as the provisioning of the temporary thread pool and implementing preemption. Executor scheduling was implemented in each session by imposing synchronization among GPU executors that share the same GPU. As the number of models sharing a GPU is typically small (2-3), we used atomic instructions to synchronize on a flag and did not observe noticeable scalability issues. Sessions that do not share GPU schedule independently.

Since SwitchFlow runs models using one global thread pool, the models need to run within one TF instance (process) as opposed to one model per instance in the vanilla TF. In the prototype, we employed multiple Python threads to launch models from the same TF instance. As such, users' models written in Python need to be converted into modules and imported to a main `launch.py` program. This implementation can be improved to employ the gRPC interface for model submission, in a way similar to TF serving [37].

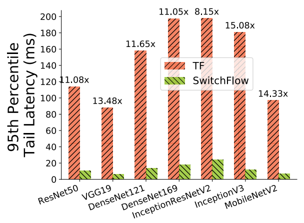
It is straightforward to adapt Python TF models to run with SwitchFlow. It takes 1 line of code (LOC) to configure priority for model preemption and 4 LOCs to restrict one GPU executor at a time to run on a shared GPU. Listing 1 shows a more sophisticated



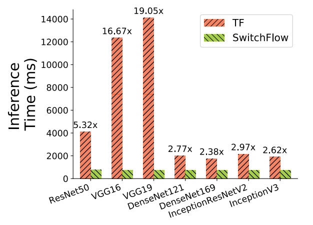
(a) MobileNetV2 (training).



(b) ResNet50 (training).



(c) VGG16 (training).



(d) NMT inference with different CNN training jobs (data series).

Figure 6: The 95th percentile tail latency of inference when co-run with background training jobs. The inference request has a higher priority and a batch size of 1. The title of each subfigure shows the background training job.

case to share the input preprocessing stage between two models. Only 5 LOCs need to be added to a launcher program. The required changes are to add environment variables to configure input sharing. Input reuse can be conveniently enabled/disabled (line 1) and models which share the input with a master model link their `recv` nodes on the GPU executor to the `recv` nodes in the master model (line 2-8). The two models are then launched from two Python threads with the shared stage as an argument (line 10-19).

5 EVALUATION

This section evaluates the effectiveness of SwitchFlow for representative DNNs on four different GPUs. Since TF does not support sharing a GPU, we compare SwitchFlow against two variants of TF: i) multi-threaded TF that uses multiple streams for spatial sharing and ii) TF with session-based time slicing, similar to Gandiva [51]. iii) NVIDIA MPS [34]. Experimental results show that 1) SwitchFlow's preemption mechanism is effective, achieving up to an order of magnitude improvement on prediction tail latency (Section 5.2.1) and maintaining high throughput (Section 5.2.2), 2) Input reuse among correlated models (Section 5.3) and interleaved execution of independent models (Section 5.4) leads to significant performance improvements in prediction jobs.

5.1 Experimental Setup

Machine configuration. Experiments were conducted on two servers and a Jetson TX2 development kit, all running Ubuntu 16.04. One server was equipped with two different NVIDIA GPUs: GeForce GTX 1080 Ti (11 GB device memory) and RTX 2080 Ti (11 GB) and the other server was with 4 NVIDIA Tesla V100 GPUs (32 GB). Both servers had dual 18-core Intel Xeon processors and over 250GB memory. The CPU and memory performance of the servers is comparable. Jetson TX2 is an embedded computing board with a quad-core ARM Cortex-A57, a 256-core Pascal GPU, and 8GB memory shared between the CPU and GPU.

Software. We implemented SwitchFlow on TensorFlow and used variants of TF with the same version for comparison. The CUDA version was v10.0 and the machine learning library used was cuDNN v7.6.4.

Benchmarks. Multiple CNN models were selected from Keras applications: ResNet50, VGG16, VGG19, DenseNet121, DenseNet169, InceptionResNetV2, InceptionV3, MobileNet, MobileNetV2, NasNetLarge, NasNetMobile; one recurrent neural network (RNN) model: NMT.

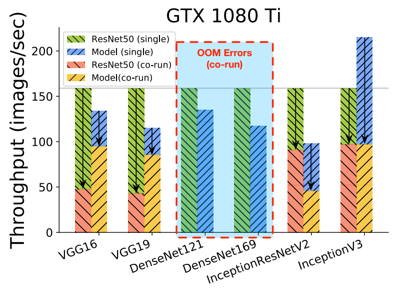
The dataset for CNN models was ImageNet raw JPEG images for evaluation. The dataset for NMT was German-English WMT'16 dataset [2]. For training, the mini-batch size was 32; for inference, if not otherwise stated, batch size (BS) was set to the largest that does not lead to an OOM error.

5.2 Effectiveness of Preemption

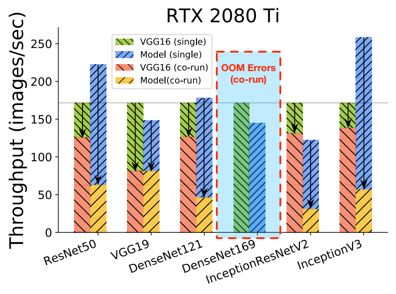
5.2.1 Tail Latency. To evaluate the effectiveness of SwitchFlow's preemption mechanism, we co-ran an inference job with a background compute-intensive training job. Inference requests were configured with higher priority and each contained only one image (BS=1). This ensures that the GPU has sufficient resources to serve the requests and only scheduling affects latency. We first launched the background training job, waited for its warmup, and then submitted inference requests as a continuous stream. The baseline was the multi-threaded TF running training and inference in separate threads, which allowed the two jobs to freely run on GPU.

Figure 6 shows the 95th tail latency of inference requests due to TF and SwitchFlow. The results show that SwitchFlow achieved significant better tail latency compared to TF. The performance gap varied depending on the resource intensity of the training job. As shown in Figure 6 (a)-(c), the performance gap enlarges as models become more computationally expensive. The largest improvement on tail latency (19.05x) was from the test with NMT inference and VGG16 training (Figure 6 (d)). RNN inference itself is fairly expensive on GPU and was significantly slowed down when co-running with another expensive model VGG16.

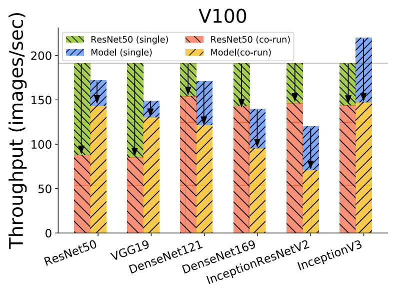
We also evaluated two variants of multi-threaded TF and they incurred even longer delay to inference requests, thereby their results not shown. The first TF variant enforced task priority in the global thread pool. However, since worker threads perform work stealing oblivious of job type, priority inversion occurred and tasks execution of the training and inference jobs were interleaved. The second TF variant employed session-based time slicing and assigned inference a higher priority. Because this approach lacks preemption, in the worst case, inference had to wait for a full session of training to finish.



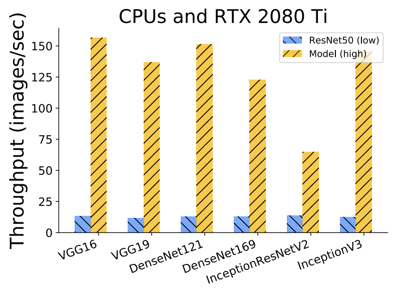
(a) Threaded TF: Co-train with ResNet50.



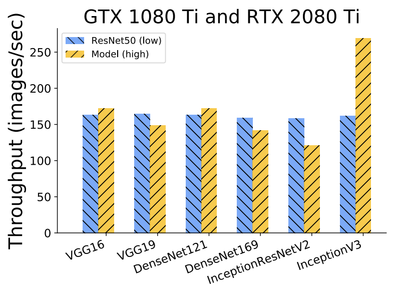
(b) Threaded TF: Co-train with VGG16.



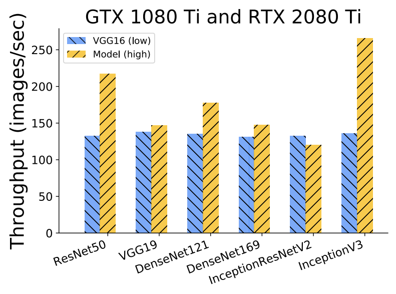
(c) MPS: Co-train with ResNet50.



(d) Co-train with ResNet50 (low).



(e) Co-train with ResNet50 (low).



(f) Co-train with VGG16 (low).

Figure 7: The throughput of two training jobs sharing a single GPU. The high priority job is shown in data series and the low priority background job is indicated in each figure. In (a)-(c), the bars in each group correspond to the performance of the two models, with the arrows showing the performance degrations compared to running in solo. (a) and (b) are under multi-threaded TF, (c) is under MPS, and (d)-(f) are under SwitchFlow.

In contrast, SwitchFlow achieved consistently low latency across all workloads. The absolute tail latencies against different models were similar, suggesting that SwitchFlow was able to timely preempt current jobs regardless of their resource intensity. The key advantage of SwitchFlow is the isolation between training and inference jobs.

Table 1: The overhead of model state transfer.

Model Name	Stateful Variables (MiB)	Transfer time (ms) GPU to GPU (PCIe 3.0)
ResNet50	198.53	28.838
VGG16	1055.58	103.747
VGG19	1096.09	109.416
DenseNet121	64.83	39.823
DenseNet169	108.61	45.236
InceptionResNetV2	426.18	82.137
InceptionV3	182.00	31.613
MobileNetV2	27.25	17.505

5.2.2 Throughput. We are also interested in the performance of a preempted job and evaluated the throughput of two co-running training jobs. We considered a scenario in which a high priority training job needs to preempt a low priority job to run on a GPU. The GPU could be the only one on a machine or the faster one among multiple GPUs. In the vanilla TF that does not support GPU sharing, the low priority job has to be killed. Therefore, we used the multi-threaded TF as the baseline, under which the two models can freely share GPU. We also compared SwitchFlow with NVIDIA MPS on the V100, the most powerful GPU in our testbed. Figure 7 (a) and (b) show that resource contention on GPU led to significant slowdowns to both models. More seriously, allowing models to freely access GPU resources resulted in some model crashes due to OOM errors on both GPUs. Users need to carefully determine which models cannot be collocated. This is a tedious process since memory demands also depend on model input. Multi-threaded TF causes OOM errors when the aggregated, real-time memory demand of the two models at any point exceeds device memory. Worse, all models crash under NVIDIA MPS on the 1080 Ti and 2080 Ti GPUs because the two processes in MPS, each running a separate model, do not share GPU memory allocation. Thus, when the aggregated peak memory demand exceeds GPU capacity training crashes. Model co-training under MPS only can complete on V100 since it has triple device memory. Similar to multi-threaded TF, MPS also inflicted significant slowdowns to both models.

In contrast, SwitchFlow does not require user-side tuning and allows models to access full GPU capacity. Upon the arrival of a high priority model, the low priority one is preempted and migrated to a different device, whether be another slower GPU or CPU. We make the following observations in Figure 7 (d)-(f): 1) there was no crash. 2) in all cases, the high priority job achieved much higher throughput than that in multi-threaded TF. 3) The low priority job achieved acceptable throughput when migrated to a slower GPU

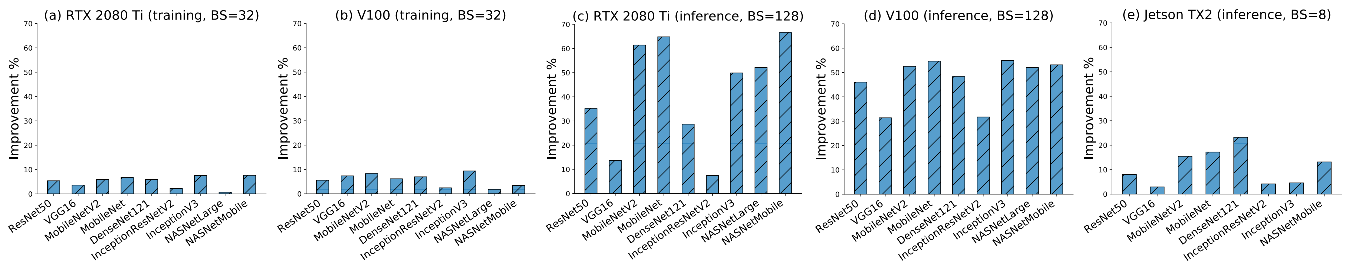


Figure 8: Performance improvement due to input reuse among multiple identical models against session-based time slicing.

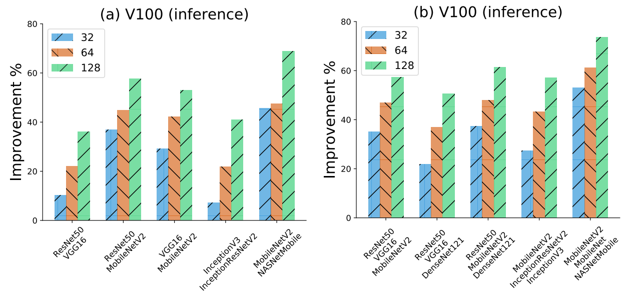


Figure 9: Performance improvement due to input reuse among different models.

but suffered drastic throughput drop when migrated to CPU since the DL operations were not designed to run on CPU. 4) The high priority job still experienced throughput loss compared to running in solo. While it ran a dedicate GPU, the low priority job occupied a few worker threads, which may delay task dispatching in the high priority job.

5.2.3 Preemption Overhead. Preemption in SwitchFlow involves aborting the execution of outstanding nodes (kernels), allocating space on a destination device, transferring model states (weights) to the destination, and freeing the memory of model states on the source device. Only waiting for the outstanding nodes of a preempted job to complete is on the critical path of a new job. Figure 3 shows kernel time ranging from a few tens of milliseconds. Therefore, the worst case preemption latency is approximately a few tens of milliseconds. The aborted operations are stochastic when preemption occurs, so the preemption latency is implicitly subject to the worst case operation.

Another source of overhead is the memory space needed to retain the model states of the preempted job until they are transferred to the destination device. Since the state transfer is asynchronous, the retained states occupy the GPU memory that could otherwise be used by a new job. Table 1 shows the amount of data need to be transferred as model states and the time required for GPU-to-GPU transfer via x16 PCIe 3.0. The largest model (VGG19) occupied about 10% of the GPU memory, e.g., 11GB device memory on GTX 1080 Ti and RTX 2080 Ti, and it takes at most 110*ms* before the states are transferred and memory is released.

5.3 Sharing Inputs among Similar Models

Allowing models to simultaneously share GPU may cause OOM errors while dedicating heterogeneous devices to one model leads to low efficiency. In this section, we evaluate the efficacy of sharing the data preprocessing stage among models that take the same input batches and can be trained in a lockstep manner [44]. SwitchFlow is configured to alternate model executions before moving to the next input batch. The baseline is **session-based time slicing**, an approach adopted by Gandiva [51], wherein models have exclusive access to both CPU and GPU during one session run and each model is allowed one session run at a time. Note that there is no data reuse in the baseline. Performance is measured by the completion time of 200 iterations (training or inference) from each model after warmup.

In cases where multiple models have same preprocessing pipelines, to mitigate the upstream data preprocessing [13, 32] time, batched input data are reused between different models. As downstream GPU keep consuming input data without involving repeated data preprocessing, training or inference workloads which are bottlenecks at CPU side can achieve speedup. Initially, the master model carries out data preprocessing and data augmentation. Next, the processed input are cached for the subsidiary models to exploit again in the following session runs.

Figure 8 shows normalized performance improvement due to SwitchFlow against the baseline on three GPUs. In this evaluation, we co-ran two identical models as their sessions are guaranteed to have the same length so that the maximum gain of input reuse can be determined for each specific model. Figure 8 (a) and (b) suggest that there was marginal performance gain due to input reuse for training jobs. Since each iteration of training lasts hundreds of milliseconds on GPU, the existing mechanisms in TF, such as input prefetching and parallel data preprocessing, can effectively overlap GPU and CPU time, leaving little room for further improvements. In contrast, Figure 8 (c) and (d) show significant improvements when two inference jobs were collocated. Input reuse saved as much as 65% compared to session-based time slicing. An interesting observation is that faster GPU (V100) led to higher gain in complex models (e.g., ResNet50, VGG16, and InceptionResNetV2) but lower gain in lightweight models (e.g., MobileNet and NASNetMobile). Jetson TX2 has limited shared memory between CPU and GPU and thus is not intended for training. Figure 8 (e) shows lower gain for inference on TX2 as the embedded GPU is much slower. Again, the higher gains were from lightweight models, which require less GPU computation.

Next, we evaluate input sharing among different models. Although models have different internal structures, they are CNN models for image classification. Thus, they can share the preprocessing stage. Figure 9 shows the results with different batch sizes and a varying number of collocated models. The findings are 1) larger

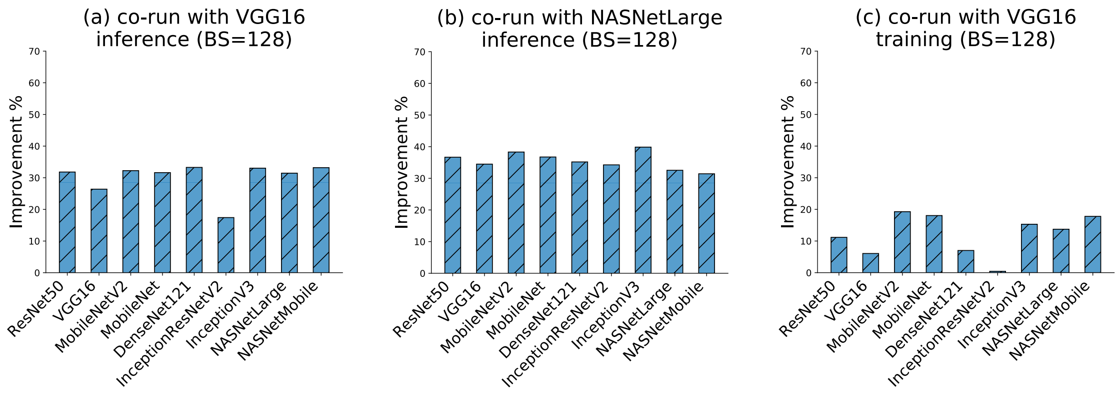


Figure 10: Performance improvement due to interleaving executors of independent models against session-based time slicing.

batch sizes led to higher improvements, indicating CPU increasingly became the bottleneck as more images included in a batch. 2) Co-running more models had diminishing gains, especially among complex models. According to the results, it is recommended that no more than three models should co-run on a single GPU.

5.4 Interleaving Independent Models

In this section, we relax the requirement of sharing the input and evaluate how much SwitchFlow improves executor scheduling among independent models. SwitchFlow alternates GPU executors from different models but allows CPU executors to freely run. Figure 10 shows performance improvements in three scenarios: (a) inference jobs sharing with inference of a heavy-weight model (VGG16), (b) sharing with inference of a lightweight model (NASNetLarge), and (c) sharing with training of a heavy-weight model (VGG16). The GPU used was V100. The figure shows that SwitchFlow's performance gain compared to the baseline was much lower than that in the input reuse tests. This is expected since scheduling may not perfectly overlap GPU and CPU processing while sharing inputs entirely bypasses the CPU stage. Still, SwitchFlow was able to consistently achieve 30% among inference jobs, regardless of the model type. When co-ran with training, the gain diminished but for lightweight models (e.g., MobileNetV2) the gain was up to 20%.

6 RELATED WORK

Various approaches to share GPUs in a multitasking environment are proposed to meet a number of objectives, such as responsiveness, throughput, resource utilization, isolation.

Temporal and spatial GPU multitasking. Existing studies in GPU multitasking include: (1) time-sliced scheduling [38]; (2) spatial partitioning scheduling [5, 40]; (3) space-time scheduling [21, 26, 34]. The proposed strategies can be categorized into different scheduling granularity: context level [47], kernel level [21, 46], thread block level [11], SM level [49, 53], and graph nodes level for DL workloads [20]. Time-sliced scheduling controls the state transitions, priority to ensure responsiveness and fairness. Interrupt request triggers context switch between a serial of applications. Spatial partitioning scheduling relies on data slicing, kernel slicing and fusion to split data and kernel into a number of smaller chunks so that they can co-schedule sub-kernels to different CUDA streams or SMs.

These work focus on low-level management of GPU kernels and memory copy. SwitchFlow takes both low-level kernel launching constrains and DL DAG computation graph characteristics into consideration to enforce time-slicing a GPU exclusively. Thus, high GPU utilization can be achieved without interference.

DL workloads scheduling, preemption and migration. ByteScheduler [42] is a generic priority-based scheduler for DNN distributed training. Gandiva [51] is a cluster scheduling framework to improve latency and efficiency of training DNN models by time-slicing GPUs. Olympian [20] proposes a scheduling policy to enable fair sharing multiple DNNs in TF-Serving [37]. Pretzel [25] applies multi-model optimizations for ML.Net [6] prediction serving systems. Previous work consider either training or inference phases, SwitchFlow instead focuses on both workloads to maximize GPU utilization and throughput, and to minimize latency. While PipeSwitch [8] enables fast model switching to allow multiple DL models to share a single GPU, it relies on dynamic graph execution for layer-to-layer model transmission and execution. In contrast, SwitchFlow focuses on DL multitasking on static graphs, which are more efficient but challenging to switch. AntMan [52] proposes elastic memory management, which can potentially help in DL multitasking and is orthogonal to SwitchFlow. However, it requires unified GPU memory and may incur high overhead.

Olympian [20] interleaves with graph nodes but do not preempt ongoing graph. In Gandiva [51], preemption and migration is extended by the Tensorflow already supported checkpoint APIs, which may incur considerable overhead compared with SwitchFlow by saving and restoring several hundreds of MiB or few Gib checkpoint [51] that cannot be tolerated for inference jobs. Our design does not preempt an issued GPU kernel since it can be expensive [39, 45, 50]. We transfer stateful variables to another device without involving checkpoint. Also, DL systems consists of data preprocessing pipelines for both training and inference [13]. SwitchFlow leverages overlapping data preprocessing and kernel execution to maximize throughput.

7 CONCLUSION

This paper presents SwitchFlow, a scheduling framework for DL multitasking. Spatial and temporal multitasking are either ineffective or inefficient in DL frameworks that employ computation graphs. We demonstrated that by carefully controlling the scheduling of GPU executors, one can simultaneously achieve high pipeline efficiency and OOM-free execution. We evaluated SwitchFlow with representative DNN models. The results show that SwitchFlow achieved significant performance improvements on both inference latency and training throughput compared to TensorFlow.

ACKNOWLEDGEMENTS

We are grateful to the anonymous reviewers for their comments on this paper. This work was supported in part by U.S. NSF grants CCF-1845706 and IIS-1852606.

REFERENCES

[1] 2015. *How to prevent tensorflow from allocating the totality of a GPU memory*. Retrieved Jan 16, 2021 from https://stackoverflow.com/questions/34199233/

[2] 2016. *English-German WMT'16 Translation Task*. Retrieved Jan 16, 2021 from http://www.statmt.org/wmt16/translation-task.html

[3] 2018. *Tips to Improve Performance for Popular Deep Learning Frameworks on CPUs*. Retrieved Jan 16, 2021 from https://software.intel.com/content/www/us/en/develop/articles/tips-to-improve-performance-for-popular-deep-learning-frameworks-on-multi-core-cpus.htm

[4] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. 2016. Tensorflow: A System for Large-scale Machine Learning. In *Proc. of USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.

[5] Jacob T Adriaens, Katherine Compton, Nam Sung Kim, and Michael J Schulte. 2012. The Case for GPGPU Spatial Multitasking. In *In Prof. of IEEE IEEE International Symposium on High-Performance Computer Architecture (HPCA)*.

[6] Zeeshan Ahmed, Saeed Amizadeh, Mikhail Bilenko, Rogan Carr, Wei-Sheng Chin, Yael Dekel, Xavier Dupre, Vadim Eksarevskiy, Senja Filipi, Tom Finley, et al. 2019. Machine Learning at Microsoft with ML. NET. In *Proc. of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*.

[7] Rachata Ausavarungnirun, Vance Miller, Joshua Landgraf, Saugata Ghose, Jayneel Gandhi, Adwait Jog, Christopher J Rossbach, and Onur Mutlu. 2018. Mask: Redesigning the GPU Memory Hierarchy to Support Multi-Application Concurrency. In *Proc. of International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[8] Zhihao Bai, Zhen Zhang, Yibo Zhu, and Xin Jin. 2020. PipeSwitch: Fast Pipelined Context Switching for Deep Learning Applications. In *Proc. of USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.

[9] Thomas Bradley. 2013. Hyper-Q. http://developer.download.nvidia.com/compute/DevZone/C/html_x64/6_Advanced/simpleHyperQ/doc/HyperQ.pdf.

[10] Richard Caruana. 1993. Multitask Learning: A Knowledge-Based Source of Inductive Bias. In *Proc. of the Tenth International Conference on Machine Learning*.

[11] Guoyang Chen, Yue Zhao, Xipeng Shen, and Huiyang Zhou. 2017. EffiSha: A Software Framework for Enabling Effficient Preemptive Scheduling of GPU. In *Proc. of Symposium on Principles and Practice of Parallel Programming (SIGPLAN)*.

[12] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. 2014. cuDNN: Efficient Primitives for Deep Learning. In *arXiv preprint arXiv:1410.0759*.

[13] Dami Choi, Alexandre Passos, Christopher J Shallue, and George E Dahl. 2019. Faster Neural Network Training with Data Echoing. In *arXiv preprint arXiv:1907.05550*.

[14] Daniel Crankshaw, Xin Wang, Guilio Zhou, Michael J Franklin, Joseph E Gonzalez, and Ion Stoica. 2017. Clipper: A Low-latency Online Prediction Serving System. In *Proc. of USENIX Symposium on Networked Systems Design and Implementation (NSDI)*.

[15] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. Bert: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *arXiv preprint arXiv:1810.04805*.

[16] Google. 2020. *Use a GPU in TensorFlow*. Retrieved Jan 16, 2021 from https://www.tensorflow.org/guide/gpu

[17] Google. 2021. *TensorFlow Profiler*. Retrieved Jan 16, 2021 from https://www.tensorflow.org/guide/profiler

[18] Alex Graves, Abdel-rahman Mohamed, and Geoffrey Hinton. 2013. Speech Recognition with Deep Recurrent Neural Networks. In *In Proc. of International Conference on Acoustics, Speech and Signal Processing*.

[19] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep Residual Learning for Image Recognition. In *Proc. of the IEEE Conference on Computer Vision and Pattern Recognition*.

[20] Yitao Hu, Swati Rallapalli, Bongjun Ko, and Ramesh Govindan. 2018. Olympian: Scheduling GPU Usage in a Deep Neural Network Model Serving System. In *Proc. of the 19th International Middleware Conference*.

[21] Paras Jain, Xiangxi Mo, Ajay Jain, Harikaran Subbaraj, Rehan Sohail Durrani, Alexey Tumanov, Joseph Gonzalez, and Ion Stoica. 2011. Dynamic Space-Time Scheduling for GPU Inference. In *Proc. of Conference on Neural Information Processing Systems (NeurIPS)*.

[22] Myeongjae Jeon, Shivaram Venkataraman, Junjie Qian, Amar Phanishayee, Wencong Xiao, and Fan Yang. 2018. Multi-tenant GPU Clusters for Deep Learning Workloads: Analysis and implications. In *Tech. Rep.*

[23] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. 2017. In-datacenter performance analysis of a tensor processing unit. In *Proc. of the 44th Annual International Symposium on Computer Architecture(ISCA)*.

[24] LeCun, Yann and Bengio, Yoshua and Hinton, Geoffrey. 2015. Deep learning. In *Nature*.

[25] Yunseong Lee, Alberto Scolari, Byung-Gon Chun, Marco Domenico Santambrogio, Markus Weimer, and Matteo Interlandi. 2018. PRETZEL: Opening the Black Box of Machine Learning Prediction Serving Systems. In *Proc. of USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.

[26] Yun Liang, Huynh Phung Huynh, Kyle Rupnow, Rick Siow Mong Goh, and Deming Chen. 2014. Efficient GPU Spatial-temporal Multitasking. In *IEEE Transactions on Parallel and Distributed Systems*.

[27] Chen Meng, Minmin Sun, Jun Yang, Minghui Qiu, and Yang Gu. 2017. Training Deeper Models by GPU Memory Optimization on TensorFlow. In *Proc. of ML Systems Workshop in NIPS*.

[28] Risto Miikkulainen, Jason Liang, Elliot Meyerson, Aditya Rawal, Daniel Fink, Olivier Francon, Bala Raju, Hormoz Shahrzad, Arshak Navruzyan, Nigel Duffy, et al. 2019. Evolving deep neural networks. In *Proc. of Artificial Intelligence in the Age of Neural Networks and Brain Computing*.

[29] Jayashree Mohan, Amar Phanishayee, Ashish Raniwala, and Vijay Chidambaram. 2020. Analyzing and Mitigating Data Stalls in DNN Training. (2020).

[30] NVIDIA. 2017. *Maximizing Unified Memory Performance in CUDA*. Retrieved Jan 16, 2021 from https://devblogs.nvidia.com/maximizing-unified-memory-performance-cuda/

[31] NVIDIA. 2018. *NVIDIA Virtual GPU Software Documentation*. Retrieved Nov 4, 2020 from https://docs.nvidia.com/grid/latest/grid-vgpu-user-guide/index.html

[32] NVIDIA. 2019. *GPUDirect Storage: A Direct Path Between Storage and GPU Memory*. Retrieved Jan 16, 2021 from https://devblogs.nvidia.com/gpudirect-storage/

[33] NVIDIA. 2020. *CUDA Basic Linear Algebra Subroutine library*. Retrieved Jan 16, 2021 from https://docs.nvidia.com/cuda/cublas/index.html

[34] NVIDIA. 2020. *Multi-Process Service*. Retrieved Jan 16, 2021 from https://docs.nvidia.com/deploy/mps/index.html

[35] NVIDIA. 2020. *The user manual for NVIDIA profiling tools for optimizing performance of CUDA applications*. Retrieved Jan 16, 2021 from https://docs.nvidia.com/cuda/profiler-users-guide/index.html

[36] NVIDIA. November 28, 2019. CUDA Occupancy Calculator. https://docs.nvidia.com/cuda/cuda-occupancy-calculator/index.html.

[37] Christopher Olston, Noah Fiedel, Kiril Gorovoy, Jeremiah Harmsen, Li Lao, Fangwei Li, Vinu Rajashekhar, Sukriti Ramesh, and Jordan Soyke. 2017. Tensorflow-Serving: Flexible, High-performance ML Serving. In *In Proc. of Neural Information Processing Systems (NeurIPS), Long Beach, CA, USA*.

[38] Sreepathi Pai, Matthew J Thazhuthaveetil, and Ramaswamy Govindarajan. 2013. Improving GPGPU Concurrency with Elastic Kernels. In *Proc. of International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[39] Jason Jong Kyu Park, Yongjun Park, and Scott Mahlke. 2015. Chimera: Collaborative preemption for multitasking on a shared GPU. In *Proc. of International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[40] Jason Jong Kyu Park, Yongjun Park, and Scott Mahlke. 2017. Dynamic Resource Management for Efficient Utilization of Multitasking GPUs. In *Proc. of International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[41] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. 2017. Automatic Differentiation in PyTorch.

[42] Yanghua Peng, Yibo Zhu, Yangrui Chen, Yixin Bao, Bairen Yi, Chang Lan, Chuan Wu, and Chuanxiong Guo. 2019. A Generic Communication Scheduler for Distributed DNN Training Acceleration. In *Proc. of Symposium on Operating Systems Principles (SOSP)*.

[43] Minsoo Rhu, Natalia Gimelshein, Jason Clemons, Arslan Zulfiqar, and Stephen W Keckler. 2016. vDNN: Virtualized deep neural networks for scalable, memory-efficient neural network design. In *Proc. of International Symposium on Microarchitecture (MICRO)*.

[44] Sebastian Ruder. 2017. An Overview of Multi-task Learning in Deep Neural Networks. In *arXiv preprint arXiv:1706.05098*.

[45] Ivan Tanasic, Isaac Gelado, Javier Cabezas, Alex Ramirez, Nacho Navarro, and Mateo Valero. 2014. Enabling Preemptive Multiprogramming on GPUs. In *Proc.*

of International Symposium on Computer Architecture (ISCA).

[46] Mohamed Wahib and Naoya Maruyama. 2014. Scalable Kernel Fusion for Memory-bound GPU Applications. In *Proc. of the International Conference for High Performance Computing, Networking, Storage and Analysis*.

[47] Lingyuan Wang, Miaoqing Huang, and Tarek El-Ghazawi. 2011. Exploiting Concurrent Kernel Execution on Graphic Processing Units. In *Proc. of High performance computing and simulation (HPCS)*.

[48] Zhenning Wang, Jun Yang, Rami Melhem, Bruce Childers, Youtao Zhang, and Minyi Guo. 2016. Simultaneous Multikernel GPU: Multi-tasking throughput Processors via Fine-grained Sharing. In *Proc. of IEEE International Symposium on High Performance Computer Architecture (HPCA)*.

[49] Bo Wu, Guoyang Chen, Dong Li, Xipeng Shen, and Jeffrey Vetter. 2015. Enabling and Exploiting Flexible Task Assignment on GPU through SM-centric Program Transformations. In *Proc. of International Conference on Supercomputing (SC)*.

[50] Bo Wu, Xu Liu, Xiaobo Zhou, and Changjun Jiang. 2017. Flep: Enabling Flexible and Efficient Preemption on GPUs. In *Proc. of International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

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

[52] Wencong Xiao, Shiru Ren, Yong Li, Yang Zhang, Pengyang Hou, Zhi Li, Yihui Feng, Wei Lin, and Yangqing Jia. 2020. AntMan: Dynamic Scaling on GPU Clusters for Deep Learning. In *Proc. of USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.

[53] Chao Yu, Yuebin Bai, Hailong Yang, Kun Cheng, Yuhao Gu, Zhongzhi Luan, and Depei Qian. 2018. SMGuard: A Flexible and Fine-grained Resource Management Framework for GPUs. In *IEEE Transactions on Parallel and Distributed Systems*.