FreeRide: Harvesting Bubbles in Pipeline Parallelism

Jiashu Zhang
University of Waterloo
Waterloo, Canada
jiashu.zhang@uwaterloo.ca

Zihan Pan
University of Waterloo
Waterloo, Canada
z82pan@uwaterloo.ca

Molly (Yiming) Xu
University of Waterloo
Waterloo, Canada
molly.xu@uwaterloo.ca

Khuzaima Daudjee
University of Waterloo
Waterloo, Canada
khuzaima.daudjee@uwaterloo.ca

Sihang Liu
University of Waterloo
Waterloo, Canada
sihangliu@uwaterloo.ca

Abstract

The occurrence of bubbles in pipeline parallelism is an inherent limitation that can account for more than 40% of the large language model (LLM) long training times and is one of the main reasons for the under-utilization of GPU resources in LLM training. Harvesting these bubbles for GPU side tasks can increase resource utilization and reduce training costs but comes with challenges. First, because bubbles are discontinuous with various shapes, programming side tasks becomes difficult while requiring excessive engineering effort. Second, a side task can compete with pipeline training for GPU resources and incur significant overhead. To address these challenges, we propose FreeRide, a middleware system that harvests the hard-to-utilize bubbles in pipeline parallelism systems to run generic GPU side tasks. FreeRide provides programmers with interfaces to implement side tasks easily, manages bubbles and side tasks during pipeline training, and controls access to GPU resources by side tasks to reduce overhead. We demonstrate that FreeRide achieves almost 8% average cost savings with a negligible overhead of about 1% in training LLMs while serving model training, graph analytics, and image processing side tasks.

CCS Concepts

• **Computing methodologies** → **Machine learning**; • **Computer systems organization**;

Keywords

Large language model, pipeline parallelism, utilization, GPU system

ACM Reference Format:
Jiashu Zhang, Zihan Pan, Molly (Yiming) Xu, Khuzaima Daudjee, and Sihang Liu. 2025. FreeRide: Harvesting Bubbles in Pipeline Parallelism. In *26th ACM Middleware Conference (Middleware '25), December 15–19, 2025, Nashville, TN, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3721462.3730950

Middleware '25, December 15–19, 2025, Nashville, TN, USA

ACM ISBN 979-8-4007-1554-9/25/12
https://doi.org/10.1145/3721462.3730950

1 Introduction

Large language models (LLMs) are usually trained on GPUs. As these models continue to increase in size, their GPU memory requirements can easily outstrip the capacity of a single GPU [63, 71]. Consequently, to accommodate this increase in size and to boost the performance of pipeline training, it is a common practice to parallelize the training of LLMs across multiple GPUs distributed over several servers.

Pipeline parallelism is a prevalent training paradigm for LLMs using multiple GPUs. In this paradigm, the model is divided into multiple stages, each consisting of several consecutive layers. These stages are distributed across different GPUs. During each training epoch, a batch of input data is split into multiple micro-batches. Each micro-batch undergoes a forward propagation (FP) and a backward propagation (BP). The FP and BP operations on different micro-batches are carried out in parallel by the pipeline training system at each stage. The pipeline training system schedules these operations in each epoch to train LLMs [10, 12, 21, 24, 29, 34, 38, 39, 52].

An inherent limitation of pipeline parallelism is *bubbles* — periods in pipeline training where the GPU stays idle due to unsatisfied dependencies between FP and BP operations [29, 34]. Experimentally, we observe that bubbles can constitute 42.4% of the pipeline execution time, which results in significant under-utilization of GPU resources used to accelerate pipeline training. Similar levels of under-utilization have also been reported in other studies [7, 71].

GPUs are crucial resources, especially those high-end models required for training LLMs [15, 59, 71]. To enhance utilization, prior work has explored interleaving FP and BP operations [12, 21, 38, 39]. There have also been proposals to shard models into more stages and to deploy these stages on GPUs to better overlap the computation and communication [29, 34]. These approaches are effective for intra-epoch bubbles because they change how operations are interleaved within a pipeline epoch. However, they do not remove the inter-epoch bubbles that occur before and after a pipeline epoch. Prior work has also proposed to decouple the computation of gradients for the input and model weights to mitigate inter-epoch bubbles [51, 61]. However, they increase the size of activations, exacerbating GPU memory consumption, a common bottleneck in training LLMs.

Given the difficulty and overhead incurred in eliminating these bubbles, an alternative approach is to acknowledge their existence and utilize them by running additional workloads on a GPU. For

example, Bamboo [62] uses bubbles to perform redundant computation for the successive layers to improve the reliability of pipeline training on spot instances. PipeFisher computes second-order optimization based on the Fisher information matrix to increase the convergence speed of LLM training [46]. However, Bamboo and PipeFisher only target specialized procedures that are tightly coupled with pipeline training, requiring the training system and the procedures to be highly customized. Consequently, their approaches cannot be used for generic GPU workloads.

In this paper, we present FreeRide, a middleware system that bridges the gap between the available yet hard-to-utilize bubbles in pipeline parallelism and the extra GPU workloads we run to harvest them. We refer to these extra GPU workloads as *side tasks*. There are two main challenges that FreeRide has to overcome. The first challenge is the programming complexity. Bubbles are of various *shapes*, i.e., their duration and available GPU memory. Customizing side tasks for these bubbles by doing ad-hoc implementation requires enormous programming effort. Second, LLM training requires high-end GPUs that are expensive and in high demand. If side tasks interfere with the main pipeline training workload, e.g., overlapping their GPU execution with pipeline training or accessing more GPU resources than bubbles can provide, they will slow down pipeline training and increase training costs.

Our approach to overcoming the programming complexity is based on the observation that many GPU workloads naturally consist of small, repetitive steps, such as the epochs in model training that repeatedly load data and update model weights. FreeRide operates between the pipeline parallel training and the generic GPU side tasks implemented by the user. To reduce the programming effort, FreeRide introduces a framework that abstracts away the implementation details of side tasks, allowing programmers to adapt various side tasks to fit into the bubbles. The key idea is to represent the life cycle of a side task, from its process creation to termination, as states in a state machine. FreeRide provides two sets of unified interfaces — the iterative interface that features lower performance overhead, and the imperative interface that features better versatility. They facilitate the implementation of side tasks as state transitions with little engineering effort. FreeRide manages side tasks through these interfaces and serves them during bubbles.

FreeRide limits the GPU resource consumption of side tasks through the automated side task profiler and the side task manager. The side task profiler first captures the key performance characteristics of the newly implemented side tasks. The side task manager coordinates a group of side task workers, one for each GPU in the platform, and assigns each of the side tasks to one of the workers based on the characteristics. During pipeline training, bubbles are reported to the side task manager from the FreeRide-instrumented pipeline training system. The side task manager starts side tasks when the bubble period begins and pauses them when the bubble ends. A side task worker deploys each side task on top of CUDA MPS, which enables the concurrent execution of CUDA kernels from different processes [43] to limit the side task's GPU memory consumption and uses a containerized environment, e.g., Docker [4] for isolation. These components work collaboratively to serve side tasks during bubbles, achieving a low performance overhead on the primary pipeline training workload.

In summary, FreeRide is a middleware system that bridges the gap between the resourceful yet hard-to-utilize bubbles in pipeline parallelism and the extra GPU workloads we run to harvest the bubbles. It provides a holistic solution to manage and serve side tasks by leveraging bubbles in pipeline training, while maintaining minimal performance overhead and requiring low programming effort. We evaluate FreeRide by deploying it to run side tasks alongside DeepSpeed that runs pipeline training [52]. We measure the time increase in pipeline training as the performance overhead caused by side tasks. As the throughput of different side tasks is not directly comparable with the pipeline training workload, we use the cost of GPUs as a unified metric, i.e., the cost of the extra execution time from co-locating side tasks with pipeline training vs. the cost saved from running side tasks that otherwise would run on dedicated lower-tier GPUs.

The contributions of this paper are as follows:

- We study the bubbles in pipeline parallelism, present their various shapes in terms of duration and available GPU memory, and demonstrate their potential for side tasks.
- We present the programming framework and the interfaces of FreeRide[1] based on a state machine abstraction to implement generic side tasks with little engineering effort.
- We evaluate FreeRide with model training, graph analytics, and image processing side tasks to demonstrate FreeRide's effectiveness in harvesting bubbles in pipeline parallelism while reducing performance overhead.
- By serving side tasks based on the iterative interface, FreeRide achieves average *cost savings* of 7.8% with a low performance overhead (time increase in pipeline parallel training) of 1.1%. This is significantly better than using CUDA MPS [43] directly to co-locate the tasks, which results in a 4.5% *cost increase* and 48.7% overhead. When handling a mix of these three types of side tasks, FreeRide achieves 10.1% cost savings with a 1.1% overhead.

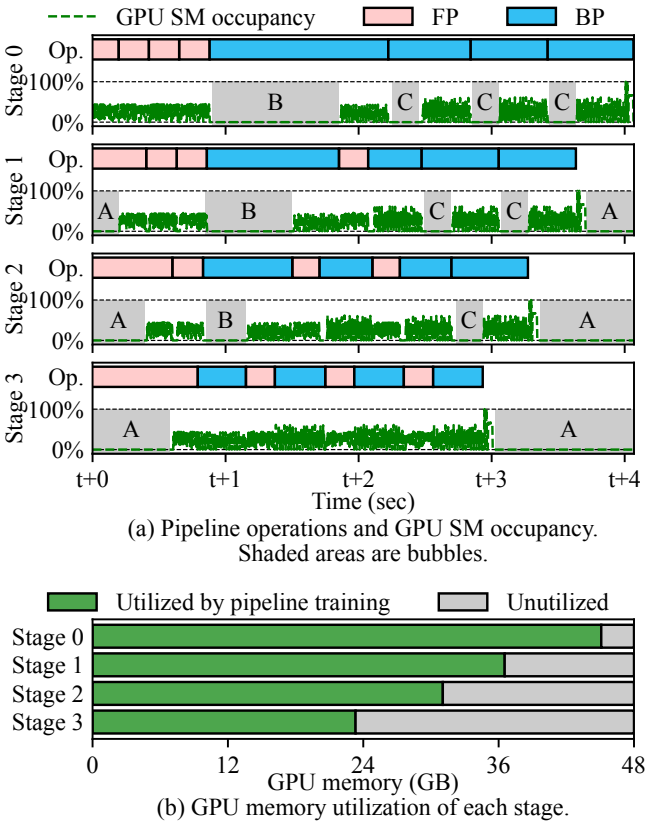
2 Background and Motivation

In this section, we provide an overview of pipeline parallelism for training LLMs, bubbles in the pipeline, and motivation for utilizing the bubbles to execute generic workloads.

2.1 Pipeline Parallelism and Bubbles

Pipeline parallelism is a widely used paradigm for distributed training of LLMs that far exceed the memory capacity of a single GPU [52, 57, 71]. In pipeline parallelism, the model is divided into multiple stages, where each stage executes several consecutive layers of the model. These stages are deployed across different GPUs to form a pipeline. To parallelize the computation at each stage and reduce GPU memory consumption, one batch of input data is split into micro-batches during each training epoch. Each micro-batch undergoes forward propagation (FP) and backward propagation (BP). In both FP and BP operations, after a stage finishes processing one micro-batch of data, it passes its output to the next stage and immediately moves on to the next micro-batch. The FP and BP operations constitute the epochs in pipeline training systems [10, 12, 21, 24, 29, 34, 38, 39, 52]. A myriad of frameworks have

[1] https://github.com/jiashu-z/freeride



(a) Pipeline operations and GPU SM occupancy. Shaded areas are bubbles.

(b) GPU memory utilization of each stage.

Figure 1: A pipeline training epoch in DeepSpeed.

been developed to support pipeline training. For example, DeepSpeed [52] and Megatron [56] are extensively used to train various LLMs such as OPT [71], Turing-NLG [53], and MT-NLG [57].

There are periods in pipeline training when the GPU streaming multiprocessor (SM) occupancy is low, as depicted by the green curves in Figure 1(a). We refer to these periods as *bubbles* in the pipeline, which are marked as shaded areas. Bubbles inherently exist in pipeline parallelism and occur repetitively throughout training, as they are fundamentally caused by unsatisfied dependencies between FP and BP operations [29, 34]. In the example of Figure 1, Stage 1 must wait for input from Stage 0 before it can execute its first FP operation, creating a bubble in Stage 1 that starts from $t + 0$.

2.2 Bubble Characterization

To study bubbles in pipeline parallelism, we train a 3.6B-parameter LLM adapted from previous work [6, 23] using DeepSpeed [52] on a 4-GPU server (detailed setup in Section 6.1). The training is deployed as a 4-stage pipeline, and each stage takes one GPU as shown in Figure 1. Overall, we observe that bubbles exhibit different characteristics across all 4 stages. Next, we take a closer look at each type of bubble.

2.2.1 Bubble Categorization. We categorize bubbles into 3 types based on their positions in the training pipeline and their causes.

• **Type-A bubbles** appear at the start and end of each epoch in all stages except for the first stage. They are due to cascading dependencies between operations at the start and end of an epoch. When an epoch starts, the FP operations start at Stage 0, while all other stages wait for input data from their preceding stages to start their first FP operation. Likewise, at the end of an epoch, the last BP operation starts at Stage 3 and all other stages wait for their succeeding stages to start their last BP operation.

• **Type-B bubbles** occur in the middle of each epoch on all stages except the last one. They are caused by dependencies between interleaved FP and BP operations. Once the first FP operation reaches the last stage, all previous stages must wait for the corresponding BP operation before they can proceed with other operations, which causes Type-B bubbles.

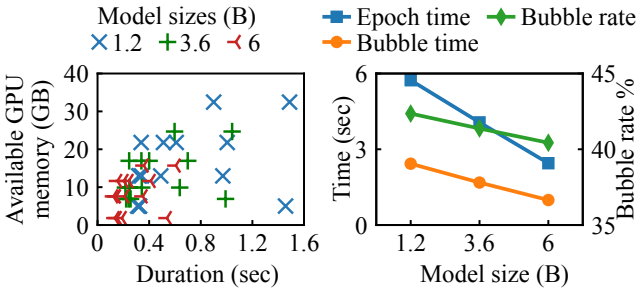
• **Type-C bubbles** also occur in the middle of each epoch. Since BP operations typically take longer than FP operations [74], interleaved yet unaligned FP and BP operations create bubbles in each stage except the last. For instance, in Figure 1(a), when Stage 2 finishes its third BP operation, it must wait for input to its fourth BP operation, which is still being processed in Stage 3, causing a type-C bubble.

Duration. In our training setup, the duration of a bubble ranges from 0.22 to 1.04 seconds, depending on its type and stage. The duration increases for Type-A bubbles but decreases for Type-B bubbles from Stage 0 to Stage 3. This is because of the cascading dependency from Stage 3 to Stage 0 for Type-A bubbles and from Stage 0 to Stage 3 for Type-B bubbles. For example, a Type-B bubble at Stage 2 is due to an unfinished BP operation at Stage 3, with the same bubble at Stage 1 caused by Stage 2. The accumulated time to satisfy dependencies elongates Type-A or Type-B bubbles at later stages. However, Type-C bubbles are caused by unaligned FP and BP operations. Therefore, they have a short duration and do not exhibit the same stage-dependent variations.

Available GPU Memory. Determined by the stage, the available GPU memory of a bubble ranges from less than 3 GB to more than 20 GB in our setup. As shown by Figure 1(b), within a stage, the GPU memory consumption of pipeline training remains the same. Thus, the bubbles within the same stage have the same amount of available GPU memory. Because the later stages consume less GPU memory to store activations used by BP operations [34], the available GPU memory increases from Stage 0 to Stage 3.

We further study pipeline training of models of different sizes. As shown in Figure 2(a), bubble shapes differ. Overall, bubbles in larger LLMs have less available memory and shorter duration, but the distributions are similar as training follows the same pipeline schedule. Even larger models do not eliminate the inherently exist bubbles. Under the same configuration, bubbles have the same characteristics during training, as epochs are repetitive and stable.

2.2.2 Bubble Rate. Besides the bubble shape, we evaluate the overall *bubble rate*, i.e., the total bubble time over pipeline training time. When the model size increases from 1.2B to 6B parameters, as shown in Figure 2(b), both the per-epoch time in pipeline training and the total per-stage bubble time decrease. Therefore, the bubble rate drops only slightly from 42.4% to 40.4%. We also evaluate a larger micro-batch number, i.e., an increase from 4 (used in Figures 1 and 2) to 8. The bubble rate drops to 26.2% as each epoch takes longer.



(a) Distribution of bubbles under different model sizes. (b) Durations and bubble rates under different model sizes.

Figure 2: Statistics of bubbles under different model sizes.

Prior work has focused on reducing bubbles in pipeline parallelism. One approach is designing different ways of interleaving FP and BP operations [12, 21, 38, 39]; another type of optimization divides the model into more stages and orchestrates their deployment to overlap computation and communication [29, 34]. These approaches optimize the scheduling strategies and interleave FP and BP operations within an epoch. Therefore, they are effective for Type-B and Type-C bubbles that appear inside an epoch but not for Type-A bubbles. There has also been work that reduces Type-A bubbles by decoupling the computation of gradients for the input and the model weights [51, 61]. This comes at a cost of higher GPU memory usage due to the extra activation storage, exacerbating the GPU memory bottleneck in LLM training. Despite these efforts, none of the approaches fully eliminate bubbles in pipeline training.

2.3 Utilizing Bubbles

The difficulties in mitigating bubbles in pipeline parallelism motivate an alternative approach — acknowledging their existence and leveraging their resources by allocating additional GPU workloads to them. Prior work has utilized bubbles to run procedures that enhance pipeline training. For example, Bamboo uses bubbles to perform redundant computation for successive layers to improve the reliability of pipeline training on spot instances [62]; PipeFisher computes second-order optimization based on the Fisher information matrix to speed up convergence [46]. However, they tightly couple the pipeline training system with specialized procedures. Implementing specialized procedures is complicated, especially since such customization should consider various bubble shapes — with durations ranging from 0.22 to 1.04 seconds, and available GPU memory from less than 3 GB to more than 20 GB on each GPU (Section 2.2).

GPUs used for training are generally compute-rich, with sufficient GPU memory available during the bubbles to accommodate other GPU workloads. Therefore, bubbles can be used to run workloads that otherwise require dedicated GPUs. For instance, training a ResNet18 model with batch size 64 takes only 2.63 GB of GPU memory with each iteration taking only 30.4 ms on our platform — small enough to fit into most of the bubbles in Figure 1(a). By doing so, the resources available in bubbles present prime opportunities for serving GPU workloads, which can amortize the expensive cost of LLM training with effective GPU workload execution. We refer to these GPU workloads served during bubbles as *side tasks*. Prior solutions that target specialized co-running procedures [46, 62] do not apply to generic workloads.

In this work, we aim to *make bubble resources available to generic workloads, allowing for a programmable and efficient use of bubbles.*

2.4 Challenges

To execute generic GPU side tasks during bubbles, we identify two major challenges.

Challenge 1: programming effort required to implement side tasks. Typically, GPU workloads are implemented assuming that they have access to the full GPU and can run continuously until they finish execution. However, bubbles are *intermittent* and largely vary in duration, as discussed in Section 2.2. A side task should be tailored to bubble patterns — the side task automatically pauses or resumes when a bubble ends or starts. Customizing the training framework to embed side tasks is conceptually feasible but limits the flexibility of implementing and executing generic GPU workloads, much like the limitations from prior work on co-running specialized procedures [46, 62].

Challenge 2: limiting the impact of side tasks. LLM training can span months on expensive high-end GPUs and cost millions of dollars [28, 71]. Even with side tasks placed in the under-utilized bubbles, they may still interfere with pipeline training, significantly increasing the cost of LLM training and offsetting the benefit of running side tasks. However, limiting the impact of side tasks is not trivial. As the shape of bubbles varies, naively implementing side tasks may consume more resources than bubbles have — exceeding the duration of bubbles or even crashing the main task due to excessive GPU memory allocation. Ideally, bubbles should be utilized without impacting the more expensive and prioritized LLM training task.

3 FreeRide Design Overview

FreeRide is our middleware system that addresses the aforementioned challenges in utilizing bubbles in pipeline training to serve generic GPU side tasks. It minimizes the performance impact of side tasks on pipeline training. In this section, we present the high-level ideas of FreeRide.

3.1 Side Task Programming Interface

Given the high cost and priority of the main pipeline training workload, the side task should not overlap with this main task to avoid competing for GPU resources. This requirement is challenging from a programmer's perspective, as it is difficult to tailor every workload to different bubble shapes. We observe that GPU workloads are not monolithic, rather, they can be often divided into small, repeated *steps* with largely predictable per-step duration and resource consumption, i.e., GPU memory. For example, epochs in model training, iterations in graph analytical workloads [26, 47, 67], and steps to process each image in image-processing workloads [41] all follow this pattern. On the other hand, bubbles also demonstrate repeating and predictable patterns, as discussed in Section 2.2.

With these observations in mind, our idea is to provide an iterative programming interface that can incorporate the step-by-step execution of side tasks to bubbles with various shapes. The user

only has to implement the side task step without being concerned with the bubble shapes, and the bubbles can serve these side tasks with largely predictable durations to avoid lack of GPU resource or overlapping of side task execution and pipeline training. The iterative programming interface provided by FreeRide employs a state machine abstraction for the life cycle of a side task composed of different states during its execution. The execution of side tasks within bubbles can be implemented as state transition functions in Figure 4 (details in Section 4.1). FreeRide works as the middleware layer in between the side tasks and the bubbles of pipeline training, managing the side tasks' start and pause through controlling their state transitions. In this way, FreeRide fits the side tasks into bubbles and minimizes the performance impact on pipeline training.

We recognize that not all GPU workloads can be easily adapted to our iterative model. To accommodate these workloads, FreeRide provides the imperative interface as an alternative. The idea is to enable pausing and resuming of execution using transparent signals from a lower level. For this reason, it does not require complex adaption but comes with a slightly higher performance overhead. We discuss both interfaces in Section 4.2.

3.2 Profiling-guided Side Task Management

As bubbles have different shapes, when a side task is newly added to FreeRide, it should be assigned to a stage whose bubbles have enough GPU memory available. When a side task is served during bubbles, there should be mechanisms that make sure the side task does not consume more resources than available by the bubbles to minimize the overhead of FreeRide, e.g., excessively allocating GPU memory or not pausing when a bubble ends.

To judiciously manage side tasks on bubbles, FreeRide leverages profiling to understand the shapes of bubbles, which can be done offline or during the first few epochs of pipeline training. Then, when a side task is newly submitted to FreeRide, as shown in Figure 3, FreeRide's automated side task profiler tracks its GPU memory consumption and per-step duration. During execution time, FreeRide employs one *side task manager* and multiple *side task workers*, one for each GPU. The side task manager assigns the newly submitted side task to one of the side task workers, which will create the side task process, based on the resulting profile. We instrument DeepSpeed to report the start timestamps and duration of bubbles to the side task manager that will initiate state transitions of each side task through remote procedure calls (RPCs) at the start and end of each bubble.

FreeRide minimizes performance overhead on the main pipeline training workload by limiting the GPU resource consumed by side tasks (Section 4.5). For GPU memory, the side task worker of FreeRide leverages CUDA MPS [43] to impose a limit on GPU memory consumed by a side task process. For GPU execution time, FreeRide uses a twofold mechanism — a *program-directed* mechanism through the programming interface, and a *framework-enforced* mechanism based on the side task manager and workers. In addition, the side task worker can deploy side task processes in Docker containers [4] for isolation.

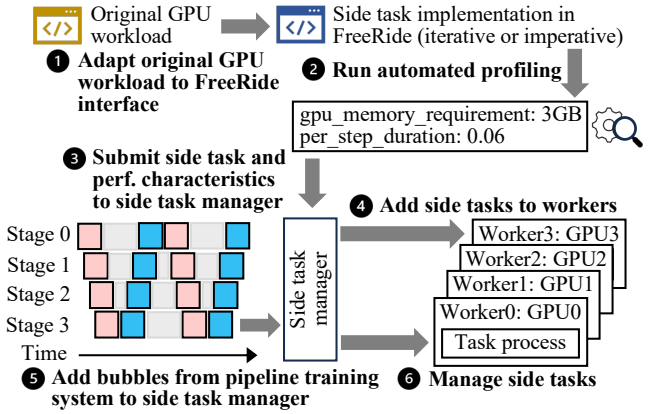
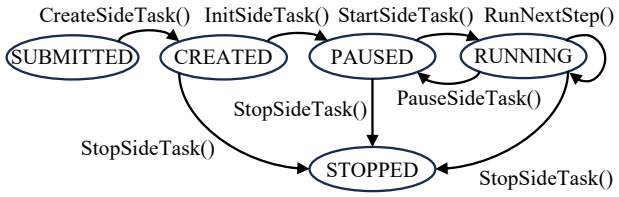


Figure 3: Workflow of FreeRide.



(a) State machine abstraction of side tasks

(b) Demonstration of side task states and pipeline operations

Figure 4: State transitions in a side task program.

3.3 FreeRide Workflow

Putting the aforementioned ideas together, we present the workflow of FreeRide in Figure 3. First, programmers adapt their side task implementation using the interface provided by FreeRide (step ❶). FreeRide then automatically generates a profile of the side task's characteristics (step ❷), which is submitted with the side task to the side task manager of FreeRide (step ❸). After the side task is submitted, based on the memory allocation of pipeline training and the characteristics of the side task, the side task manager will assign this side task to one of the workers (step ❹). When the main pipeline training workload is running, the side task manager continuously adds bubbles from the instrumented LLM training framework (step ❺); at the same time, it starts/pauses side tasks based on the available bubbles (step ❻).

4 Implementation of FreeRide

In this section, we first introduce how FreeRide supports side tasks through its framework and interfaces. Then, we present details of FreeRide's profiling-guided side task management. Finally, we

discuss FreeRide's GPU resource limit mechanisms including the implementation details.

4.1 Programming Framework of FreeRide

Figure 4(a) describes the programming framework of a side task. The framework's core is a state machine with five states and six state transitions. These five states capture the life cycle of a side task, from process creation to process termination, and correspond to different usage of hardware resources, e.g., GPU memory and GPU execution time. The six state transitions are used by the programmer to implement the user-defined logic of a side task. The programmer can override the state transition functions to customize their behavior, e.g., allocating or releasing hardware resources or performing computation on GPU. Once the side task is implemented, FreeRide automatically handles the state transitions at runtime. Next, we discuss the states and their transitions.

• `SUBMITTED`. This state means that FreeRide has profiled a task and submitted it to the side task manager, but the side task worker has not created the side task process yet. State transition `CreateSideTask()` happens automatically after the side task manager assigns a side task to a worker and the worker creates the side task process.

• `CREATED`. In this state, the worker has created the side task process, and this process has loaded its context to the main memory but not to the GPU memory. Take a model training side task as an example. When it is in the `CREATED` state, it has already created and initialized the dataset, the data loader, the loss function, and the optimizer states in CPU memory. However, the side task process will not load them into GPU memory until the side task manager initiates the state transition `InitSideTask()`. The state transition `InitSideTask()`, initiated by the side task manager, means that the side task will finish initialization.

• `PAUSED`. This state is where the side task starts to use GPU memory. The side task process has loaded its context, e.g., model weights and optimizer states, in the GPU memory. However, this process waits in the `PAUSED` state until the side task manager transitions its state to `RUNNING` through `StartSideTask()`.

• `RUNNING`. In this state, the side task executes the step-wise GPU workload. Referring to the example above of the model training side task, this step involves reading the next batch, computing the output and loss, updating the model weights, and resetting the optimizer states. The side task iteratively enters the `RunNextStep()` state transition to execute these steps until the side task manager transitions its state through `PauseSideTask()`. Therefore, in this state, the side task process uses both GPU memory and SMs.

• `STOPPED`. This state marks the end of the life cycle of a side task, where the side task process releases all of its hardware resources and terminates. It can be transitioned from states `CREATED`, `PAUSED`, and `RUNNING` through `StopSideTask()` initiated by the side task manager.

Figure 4(b) shows state transitions of a side task in Stage 0 of Figure 1. Initially, the side task is in the `PAUSED` (`P`) state. After four FP operations in the main training workload have finished, a bubble starts and the side task manager initiates `StartSideTask()` to transition the side task to the `RUNNING` (`R`) state. After the first bubble ends, the side task manager initiates `PauseSideTask()` to pause the side task. Then, the main training workload has BP operations and bubbles interleaved, leading to back-and-forth transitions between `PAUSED` and `RUNNING` states of the side task.

4.2 Interface for Side Task Implementation

Given the FreeRide programming framework, the next step is to implement side tasks, which have two requirements. First, the programmer should be able to implement the side task in a way that can pause at the end of a bubble and resume at the start of the next bubble. Second, the side task should be able to communicate with the side task manager to receive state transition RPCs (Section 4.6) for pausing and resuming. To lift programming burdens, FreeRide provides two programming interfaces supported in C++ and Python. Once implemented using either interface, FreeRide will handle the side tasks and their state transitions transparently at runtime. We discuss both interfaces next.

Iterative programming interface. This is the preferred interface for side tasks in FreeRide. It periodically checks whether the side task manager has initiated any state transitions. If so, it executes the state transition functions in Figure 4(a) and updates the state of the side task. Then, if the side task is currently in the `RUNNING` state, it executes `RunNextStep()`. The programmer only has to override these transition functions to implement the side task. Pausing and resuming the side task, the transition of states, and communication with the FreeRide side task manager are all handled by the interface itself. GPU workloads that are naturally step-wise, e.g., model training, can be easily adapted to the iterative interface. We will discuss the adaption to this interface in Section 5 using an example.

Imperative programming interface. Not all side tasks can be explicitly implemented step-wise. Therefore, FreeRide provides the imperative interface as a fallback solution. The core is the function `RunGpuWorkload()` that allows the programmer to implement generic GPU side tasks without breaking them into steps. When the side task manager changes the state of the side task to `RUNNING` for the first time, the interface calls the `RunGpuWorkload()` function to execute the side task. The interface implements the pausing and resuming through signals (`SIGTSTP` and `SIGCONT` [14]) and calls `StartSideTask()` and `PauseSideTask()` inside the handlers of the two signals. The imperative interface offers better versatility at the cost of higher performance overhead (discussed in Section 5 and evaluated in Section 6.2).

4.3 Profiling Bubbles and Side Tasks

Bubbles. To know the shapes of bubbles before serving side tasks with them, FreeRide runs DeepSpeed, monitors its estimated SM occupancy and GPU memory consumption through the PyTorch profiler [50], and automatically measures each bubble's duration and available GPU memory. Since the pipeline schedule determines bubbles, this offline profiling is done only once for each model and pipeline scheduling on the same hardware platform.

Side tasks. After the programmer has implemented the side task, FreeRide profiles it with the automated profiling tool for its performance characteristics of GPU memory consumption and per-step duration, which FreeRide uses for side task management and limiting GPU resources. For side tasks implemented using the iterative interface, this procedure is fully automated. The profiling tool runs

the side task, monitors its GPU memory consumption, and records the timestamps at the start and end of `RunNextStep()` state transition for the per-step duration. For side tasks implemented using the imperative interface, the tool profiles GPU memory consumption in the same way. However, since the side task is not step-wise, the automated profiling tool does not measure the per-step duration.

4.4 Side Task Management

FreeRide's side task management has two main roles. First, upon receiving a new side task, the side task manager assigns it to a suitable side task worker. Second, when the pipeline training system adds bubbles to the side task manager, the side task manager initiates the state transitions of side tasks (Figure 4(a)) through RPCs. In this way, the side tasks are only served during bubbles and do not compete for GPU resources with the main pipeline training workload.

To keep track of side tasks and workers, the side task manager maintains the following fields for each worker, used by Algorithms 1 and 2 for side task management:

- *GPUMem*: the available GPU memory size.
- *TaskQueue*: the queue of side tasks ordered by submission timestamps.
- *CurrentTask*: the side task that is currently served.
- *CurrentBubble*: the bubble that is currently valid.

Algorithm 1 Procedure upon a new side task.

```
1:  Input: new task Task, workers' metadata Workers
2:  MinNumTasks ← ∞
3:  SelectedWorker ← None
4:  for all Worker in Workers do
5:      if Worker.GPUMem > Task.GPUMem then
6:          NumTasks ← Worker.GetTaskNum()
7:          if NumTasks < MinNumTasks then
8:              MinNumTasks ← NumTasks
9:              SelectedWorker ← Worker
10: if SelectedWorker ≠ None then
11:     SelectedWorker.Add(Task)
12: else
13:     RejectSideTask()
```

Algorithm 1 describes how the side task manager assigns side tasks to workers. When the side task manager receives a new side task together with its GPU memory requirement (through profiling, Section 4.3), it first filters out all workers with enough available GPU memory (lines 4—5). Then, from these workers, it selects the one with the smallest number of tasks (lines 6—9). If the side task manager has selected a worker, it will assign the side task to that worker (lines 10—11). Otherwise, it will reject the side task because of insufficient GPU memory (line 13).

Algorithm 2 describes how the side task manager manages bubbles and side tasks during pipeline training. The side task manager iterates through all workers (line 3). If *CurrentBubble* has just ended for a worker, the side task manager will pause *CurrentTask* of the worker and clear *CurrentBubble* (lines 4—8). Upon a new bubble, the side task manager updates the *CurrentBubble* of this worker

Algorithm 2 Managing bubbles and side tasks.

```
1:  Input: workers' metadata Workers
2:  while SideTaskManagerIsRunning do
3:      for all Worker in Workers do
4:          if Worker.CurrentBubble ≠ None then
5:              if Worker.CurrentBubble.HasEnded() then
6:                  if Worker.CurrentTask ≠ None then
7:                      Worker.CurrentTask.PauseSideTask()
8:                  Worker.CurrentBubble ← None
9:          if Worker.HasNewBubble() then
10:             Worker.UpdateCurrentBubble()
11:             if Worker.CurrentTask = None then
12:                 if Worker.TaskQueue.IsEmpty() then
13:                     continue
14:                 nextTask ← Worker.TaskQueue.Next()
15:                 Worker.CurrentTask ← nextTask
16:             if Worker.CurrentTask.IsCreated() then
17:                 Worker.CurrentTask.InitSideTask()
18:             else if Worker.CurrentTask.IsPaused() then
19:                 Worker.CurrentTask.StartSideTask()
```

(lines 9—10). It then checks if the worker has a *CurrentTask*. If not, it will select the one with the smallest submission timestamp from *TaskQueue* as *CurrentTask* (lines 11–15). After that, the side task manager initiates `InitSideTask()` if the newly added *CurrentTask* is in `CREATED` state (lines 16—17); otherwise, its state is `PAUSED` and the side task manager initiates `StartSideTask()` (lines 18—19).

4.5 GPU Resource Limit

In this section, we introduce the mechanisms in FreeRide that reduce the impact of side tasks on the main pipeline training workload through side task resource control for both GPU memory and GPU execution time.

GPU Memory. FreeRide leverages MPS to impose GPU memory limit on side tasks. I.e., when a worker creates a side task, it sets GPU memory limits using MPS. The side task process triggers an out-of-memory (OOM) error when its memory consumption exceeds the limit, but other processes remain unaffected. However, FreeRide is also compatible with other mechanisms for limiting GPU memory, e.g., multi-instance GPU (MIG) [45] or manually implemented accounting through intercepting CUDA kernel calls [60].

GPU Execution Time. FreeRide limits GPU execution time using two mechanisms. (1) *The program-directed mechanism* is tailored for the iterative interface. When the side task manager makes an RPC to initiate `StartSideTask()` state transition of a side task, it also sends the end time of this bubble to the side task. After the state transition finishes, the side task enters the `RUNNING` state. Before the side task automatically starts `RunNextStep()`, the program-directed mechanism checks if the remaining time of the bubble is enough for the side task to execute the next step. The side task will only execute the next step if the remaining time exceeds the per-step duration. (2) *The framework-enforced mechanism* supports side tasks implemented using the imperative interface and is also a fallback mechanism for the iterative interface. After the side task manager initiates the `PauseSideTask()` state transition for a side

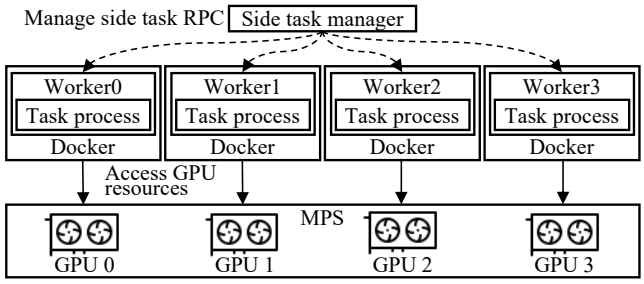


Figure 5: Architecture of FreeRide.

task, it waits for a short grace period before checking the *last paused timestamp* — a timestamp maintained by the interface that records the last time the side task was paused. If this timestamp is not updated after the state transition begins, the side task manager assumes that the interface failed to pause the side task correctly and subsequently instructs the corresponding worker to terminate the side task process using `SIGKILL`. The side task initialization, `InitSideTask`, which runs only once throughout the life cycle of a side task, is also protected by this mechanism.

4.6 Implementation

We use DeepSpeed 0.12.2 [9] as the framework for pipeline training. We modify DeepSpeed in three places with 55 lines of code: (1) before the start and at the end of an epoch for Type-A bubbles, (2) after all FP operations preceding the first BP operation for Type-B bubbles, and (3) after the first BP operation following the last FP operation for Type-C bubbles. The instrumented code reports bubbles to the side task manager in FreeRide, as shown in step ❺ of Figure 3. The modifications are done once, as the framework can be used for training different models.

To isolate the side task processes from the pipeline training process, FreeRide deploys workers (and side tasks of these workers) inside Docker containers, as illustrated in Figure 5. FreeRide implements the side task manager and each side task worker in separate processes. Communication among the pipeline training system, side tasks, and FreeRide components is facilitated through RPCs utilizing gRPC [16].

5 Use of Side Tasks Interface

This section describes FreeRide's iterative interface and imperative interface in detail.

Iterative programming interface. Figure 6 is an example of implementing a side task to train ResNet18 using the iterative interface of FreeRide in Python. Less important lines such as importing dependencies and parsing arguments are simplified. Porting this example mainly involves 6 steps. Step ①: import FreeRide dependencies and inherit the iterative interface class, which includes an implementation for the state machine abstraction, communication with the side task manager, and the program-directed mechanism to limit the GPU execution time. The programmer only needs to adapt the original GPU workload to the interface. Steps ② and ③: implement the side task initialization in 2 state transition functions, `CreateSideTask()` and `InitSideTask()`, to load the context into main memory and GPU memory respectively. Step ④: wrap the original loop implementation with `RunNextStep()`. Step ⑤: the main function parses arguments and runs the side task interface.

Most modifications are trivial, e.g., wrapping implementations with side task state transition functions in Step ②, ③, and ④, which are required by Python. Aside from this, the programmer can directly copy the important logic, e.g., loading the dataset and training the model, from the original implementation. In addition, if the programmer customizes the model architecture, the model implementation does not require modification.

Imperative programming interface. This interface does not require the programmer to implement the side task in a step-wise way. Therefore, instead of implementing the side task in multiple functions (steps ② — ⑤), the programmer can merge them in `RunGpuWorkload()`, as discussed in Section 4.2. However, this approach trades performance for less programming effort, as pausing side tasks through the framework-enforced mechanism incurs more overheads. When the side task manager initiates `PauseSideTask()` state transition via an RPC at the end of a bubble, even though the CPU process of the side task is paused by the framework-enforced mechanism (Section 4.5) after the state transition, CUDA kernels that have already started cannot be paused because they are asynchronous [42]. As a result, these CUDA kernels will overlap with pipeline training, causing a higher performance overhead than the iterative interface.

6 Evaluation

In this section, we evaluate the benefits and overhead of using FreeRide to serve side tasks.

6.1 Methodology

We describe the experimental setup of our evaluation.

6.1.1 Server setup. We use a main server (Server-I) with four RTX 6000 Ada GPUs each with 48 GB of GPU memory to evaluate all pipeline training workloads and side tasks. We use a second server (Server-II) with an RTX 3080 GPU with 10 GB of memory to run side tasks separately. Due to the global shortage of cloud GPUs, we quote prices from a community cloud vendor [54] that has GPUs available. The prices of the two servers are $P_{\text{Server-I}}$ = \$3.96/hour and $P_{\text{Server-II}}$ = \$0.18/hour, respectively (as of June, 2024). The price differences between higher- and lower-tier GPUs in major cloud GPU platforms are similar [1, 2, 27]. In addition to experiments on GPUs, we use a third server (Server-CPU) with 8 cores of an Intel Xeon Platinum 8269Y CPU and 16 GB of memory to evaluate side task performance on CPU. We deploy both pipeline training and side tasks in Docker 26.1.2 [4].

6.1.2 Comparison points. We evaluate FreeRide for side tasks developed with both the iterative and imperative interfaces. For comparison, we evaluate MPS [43], where we set pipeline training with the highest priority and side tasks with a lower priority. We also evaluate a naive co-location approach by directly co-running side tasks and the main pipeline training workload on the same GPU.

6.1.3 Pipeline training setup. We train nanoGPT [6, 23] with model sizes 1.2B, 3.6B, and 6B with DeepSpeed 0.12.2 [9] in a 4-stage pipeline on Server-II (stages 0—3 in Figure 1). We always maximize

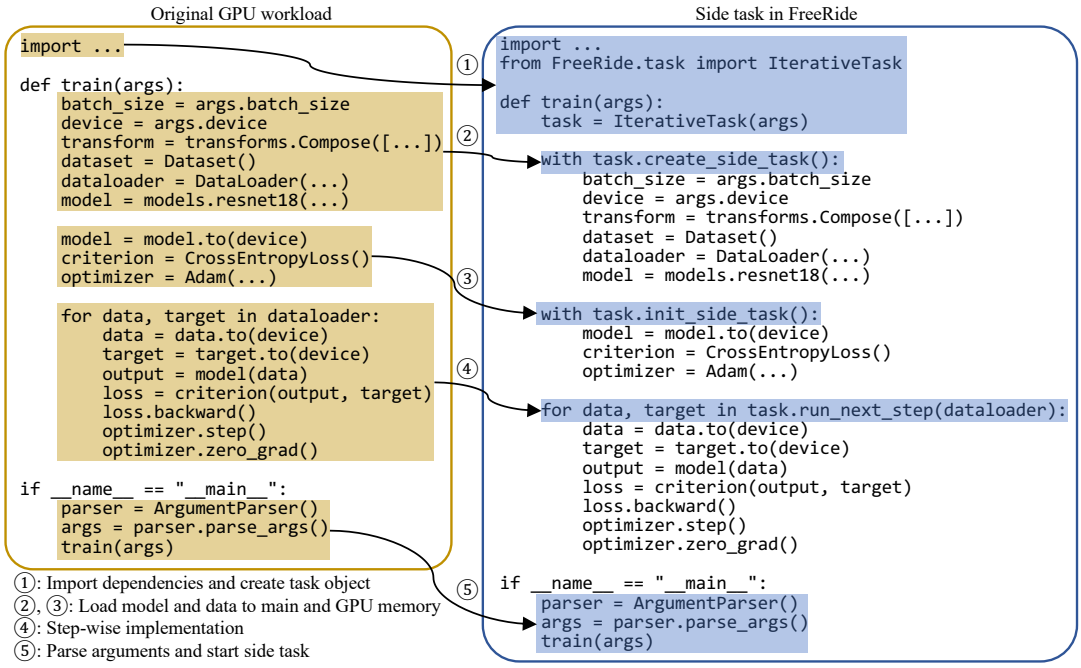


Figure 6: Example of implementing ResNet18 training using the iterative interface of FreeRide.

the micro-batch size (until just before OOM) to make full use of GPU memory during training.

6.1.4 Side task workloads. We implement 3 types of side tasks: model training, graph analytics, and image processing using both the iterative and the imperative interfaces of FreeRide. *Model training* side tasks include ResNet18, ResNet50, and VGG19. We implement the training procedure using out-of-the-box models from PyTorch [49]. *Graph analytics* side tasks are adapted from Gardenia [67]. It includes PageRank (PR) which is based on the PageRank algorithm [47] and Graph SGD (SGD) which uses stochastic gradient descent to solve matrix factorization [26], both using the Orkut dataset [68]. The *image processing* (Image) side task resizes an input image and adds a watermark, which we adapt from Nvidia's code [41].

6.1.5 Metrics. We use *time increase I* and *cost savings S* in Dollars due to side tasks as metrics. Time increase describes the performance overhead of co-locating side tasks with the main pipeline training workload. It is the ratio of extra time of pipeline training with side tasks, compared with the original DeepSpeed without any side tasks, and lower time increase means lower overhead. It is defined as

$$I = \frac{T_{\text{withSideTasks}} - T_{\text{noSideTask}}}{T_{\text{noSideTask}}} .$$

Cost savings describe the benefits of running side tasks. It is hard to directly measure the benefits of running side tasks for two reasons. First, the throughput of different size tasks and the main pipeline training workload cannot be directly compared. Second, the workloads of side tasks are typically deployed on GPUs of smaller scales, while pipeline training mostly uses flagship GPUs. To compare the value of side tasks and pipeline training that runs on different GPUs with different throughputs and to calculate the benefits, we use their costs (dollars spent on GPUs) as a proxy. First, we define the cost of pipeline training without side tasks as

$$C_{\text{noSideTask}} = P_{\text{Server-I}} \times T_{\text{noSideTask}}$$

and the cost of pipeline training with side tasks as

$$C_{\text{withSideTasks}} = P_{\text{Server-I}} \times T_{\text{withSideTasks}} .$$

Then, we compute the cost of running the same side tasks on dedicated lower-tier GPUs as

$$C_{\text{sideTasks}} = \sum_{\text{Each sideTask}} P_{\text{Server-II}} \times \frac{W_{\text{sideTask,Server-I}}}{Th_{\text{sideTask,Server-II}}}$$

where $W_{\text{sideTask,Server-I}}$ is the work done by a side task on Server-I, e.g., the number of epochs for model training side tasks, the number of iterations for graph analytics side tasks, and the number of images for the image processing side task. $Th_{\text{sideTask,Server-II}}$ is the throughput of running the same side task on Server-II, which we measure by running side tasks individually on Server-II. Finally, we define the cost savings S below, where the higher the S value, the greater the benefit.

$$S = \frac{C_{\text{sideTasks}} - (C_{\text{withSideTasks}} - C_{\text{noSideTask}})}{C_{\text{noSideTask}}} .$$

$C_{\text{sideTasks}}$ considers the cost of the side tasks served during bubbles, measured by the cost of running the same side task workloads on dedicated GPUs. $C_{\text{withSideTasks}} - C_{\text{noSideTask}}$ is the extra cost from co-location, measured by the increased costs due to co-locating side tasks and pipeline training.

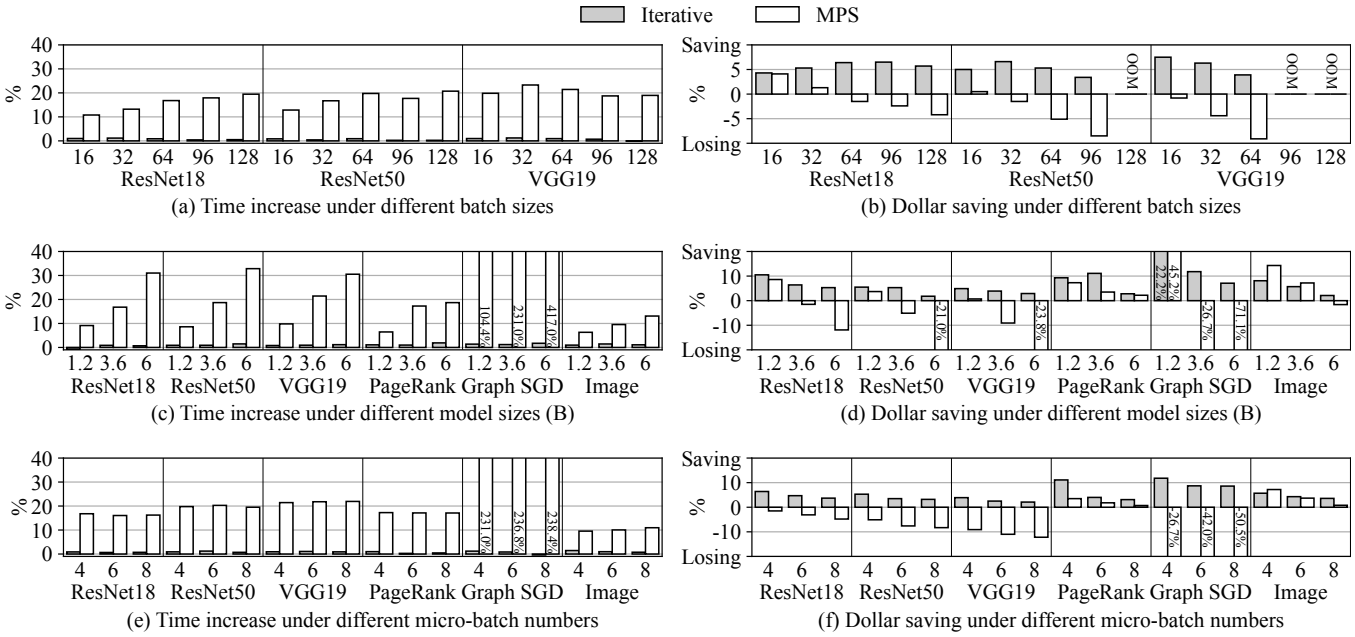


Figure 7: Sensitivity studies of FreeRide.

Table 1: Throughput of GPU side tasks on different platforms measured as iterations per second.

Side task	Iterative	Server-II	Server-CPU
ResNet18	1586.6	998.7	26.5
ResNet50	533.1	393.4	9.1
VGG19	170.7	161.8	3.0
PageRank	333.9	126.3	11.1
Graph SGD	4.2	1.5	0.6
Image	12.2	7.8	1.6

Table 2: Time increase I (lower the better) and cost savings S (positive=benefit, negative=loss, higher the better) of running DeepSpeed with side tasks.

	FreeRide				Comparative Methods			
	Iterative		Imperative		Nvidia MPS		Naive co-location	
Side task	I %	S %	I %	S %	I %	S %	I %	S %
ResNet18	0.9	6.4	2.2	6.0	16.8	-1.5	49.8	-30.7
ResNet50	0.9	5.3	3.8	3.9	19.8	-5.1	61.9	-44.0
VGG19	0.9	3.9	5.0	1.4	21.4	-9.1	53.4	-39.7
PageRank	1.0	11.1	2.5	16.4	17.3	3.5	45.1	-16.0
Graph SGD	1.2	11.8	4.1	22.8	231.0	-26.7	62.4	-9.1
Image	1.4	5.7	2.7	6.1	9.5	7.2	46.0	-29.3
Mixed	1.1	10.1	4.3	11.0	24.8	0.2	64.3	-35.5

6.2 Performance Evaluation

We run DeepSpeed to train a 3.6B model for 128 epochs with side tasks from Section 6.1.4 and compare the performance overhead, i.e., time increase (I) and cost savings (S) of using FreeRide with the two interfaces and the two comparative methods (as mentioned in Section 6.1.2). For model training side tasks, we set the batch size as 64. We run the *same side task* in all workers if they have enough GPU memory. We also run a *mixed workload* with 4 side tasks: PageRank, ResNet18, Image, and VGG19, each in one worker corresponding to the GPU of stages 0—3 in Section 6.1.3, respectively.

Performance compared to lower-tier GPU and CPU. Table 1 compares the throughput of side task workloads in Section 6.1.4 running on bubbles using the iterative interface of FreeRide (the Iterative column), on Server-II and Server-CPU as introduced in Section 6.1. The throughput is measured as iterations per second. For ResNet18, ResNet50, and VGG19 GPU workloads, one iteration corresponds to one batch of training data. For PageRank and Graph SGD, in each iteration, the graph algorithm runs over the input graph for one step. For image processing workload, one iteration processes one image.

This comparison shows that although side tasks run only during bubbles, they still achieve higher throughput compared to running on a standalone lower-tier GPU (RTX 3080) or the 8-core CPU instance. By introducing little overhead to pipeline parallel training, FreeRide harvests GPU resources that support a throughput of 1.06–2.82× of a standalone lower-tier GPU, and 7–59.9× of the CPU. These results demonstrate FreeRide's effectiveness in harvesting the bubbles in pipeline parallel training.

Co-location performance. Table 2 shows the time increase and cost savings of running DeepSpeed with side tasks of different co-location methods. FreeRide consistently exhibits lower overhead than the comparative methods, showing only a 1.1% average time increase while achieving 7.8% average cost savings through side

tasks using the iterative interface. The imperative interface achieves comparable cost savings but with a higher overhead as it relies on the less efficient framework-enforced mechanism to limit the side task's execution time (Section 4.5). In comparison, the average time increase and cost savings for MPS are 48.7% and -4.5%, and for Naive are 54.7% and -29.2%. Their *negative* cost savings indicate that these approaches can increase the total cost. Notably, the time increase of Graph SGD with MPS is as high as 231.0%. This anomaly is due to Graph SGD's high compute intensity. We conclude that FreeRide effectively utilizes bubbles in pipeline training for serving side tasks. While the comparative methods can utilize bubbles, unlike FreeRide, they are not designed for this purpose. Thus, they are inefficient in using bubbles, leading to higher costs.

6.3 Sensitivity Study

We change the side task batch size, DeepSpeed model size, and DeepSpeed micro-batch numbers of side tasks, and study the time increase and cost savings of FreeRide with the iterative interface.

(1) Varying batch sizes. Figure 7(a) includes model training side tasks under variable batch sizes. Other side tasks are not included as they do not run with batch sizes. *OOM* means that the GPU in Server-II does not have enough GPU memory for the configuration, so the cost savings cannot be calculated. FreeRide has low performance overheads, with around 1% increase in execution time, and cost savings of 3.4% – 7.5%.

(2) Varying model sizes. In Figure 7(b), the performance overheads of FreeRide range from -0.7% to 1.9%, and cost savings range from 1.8% to 22.2%. The main reason is the shorter bubble durations when training larger models as the main workload, which was also shown in Figure 2.

(3) Varying micro-batch numbers. In Figure 7(c), the performance overhead of FreeRide increases from -0.4% to 1.5%, and cost savings reduces from 2.1% to 11.8%. When the micro-batch number increases, because of the lower bubble rate (Section 2.2), the cost savings decrease.

6.4 Effectiveness of GPU Resource Limit

We use training ResNet18 as an example to demonstrate the GPU resource limit mechanism in FreeRide.

Side task execution time limit. Figure 8(a) demonstrates a case where the side task does not pause after the bubble that ends at $t + 2$. With GPU resource limit, as shown by the green and purple curves, the worker terminates the side task after a grace period via the framework-enforced mechanism.

Side task GPU memory limit. Figure 8(b) illustrates another case where the side task keeps allocating GPU memory despite its 8 GB limit. Without FreeRide's GPU resource limit mechanism, the side task's GPU memory allocation is only capped by the physical memory limit of the GPU, potentially interfering with the main training workload. With GPU resource limit, after the side task process exceeds its 8 GB GPU memory limit, it is terminated to release GPU memory.

6.5 Bubble Time Breakdown

In Figure 9, we present a breakdown of bubble utilization in FreeRide under the iterative interface. *No side task: OOM* means that some

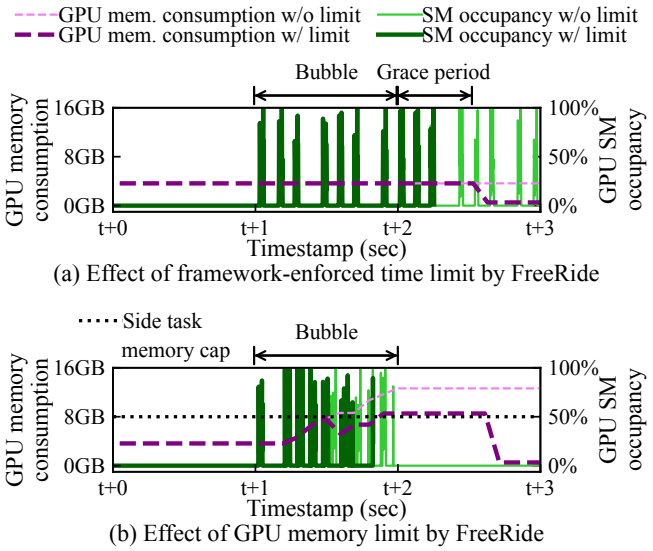


Figure 8: Demonstration of GPU resource limit in FreeRide.

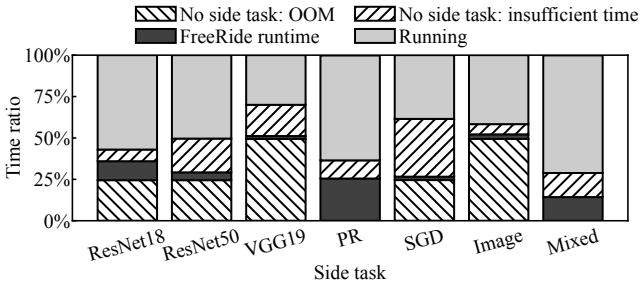


Figure 9: Bubble time breakdown.

bubbles are unused due to their limited available GPU memory. For instance, the GPU memory consumption of VGG19 or the Image side task is larger than the GPU memory of bubbles in stages 0 and 1, so they cannot use half of the bubble time. *No side task: insufficient time* refers to idle time because the remaining time of a bubble is not enough for the next step of the side task. *FreeRide runtime* is the time consumed by running FreeRide, including the interface code and the side task manager. Most of the bubble time with enough available GPU memory size is used by side tasks. For side tasks with shorter per-step durations, e.g., PageRank, the ratio of FreeRide runtime is higher because more iterations of the iterative interface are executed. In contrast, side tasks with longer per-step durations have lower bubble utilization because of insufficient time.

7 Related Work

Pipeline parallelism and bubbles. Prior work has aimed to improve the schedule of pipeline training to reduce bubbles [10, 12, 21, 24, 29, 34, 38, 39, 51, 52, 61]. Other work leverages bubbles in pipeline parallelism by assigning specialized procedures coupled with pipeline training to enable fault tolerance through

replicated computation [62], or for accelerating the training algorithms [19, 46]. These approaches require changes to the training framework and are limited to certain types of workloads. In contrast, FreeRide does not require any changes to, or coupling with, pipeline training to serve generic GPU side tasks.

GPU sharing. Gandiva time-slices GPUs for multiple jobs with fallback to non-sharing GPUs [65]. Salus designs job switching and memory sharing primitives for GPU sharing [70]. PipeSwitch further improves GPU sharing by designing fast context switch mechanisms between the host memory and the GPU memory [3]. AntMan designs dynamic scaling mechanisms for distributed deep learning workloads [66]. Zico tracks the GPU memory allocation and reclamation of deep learning jobs and shares the GPU memory reclaimed by one job with other jobs [30]. Veltair proposes a compiler that co-optimizes the compiling results of co-located GPU workloads [33]. TGS achieves GPU sharing of deep learning workloads in container clouds through rate control and transparent shared memory at the OS level [64]. Recently, Orion proposes GPU sharing by intercepting and scheduling CUDA kernel calls made by PyTorch [60]. While these approaches propose methods to share GPUs and continuously improve the utilization of GPUs, they share the GPUs aggressively, without minimizing the impact on high-priority and high-cost workloads. Therefore, they would cause high overhead (time increase) if used to co-locate LLM training and side tasks, and subsequently yield little to no cost savings due to their high overhead. In comparison, FreeRide achieves GPU sharing while maintaining a very low overhead. PilotFish on the other hand leverages the free cycles in cloud gaming for deep learning workloads [72], while FreeRide harvests the bubbles in pipeline parallelism for other generic GPU workloads. MPS and MIG [43, 45] are mechanisms provided by Nvidia for GPU sharing and virtualization. FreeRide leverages MPS to impose GPU memory limits on side tasks.

8 Discussion

Security. Prior GPU sharing solutions tend to prioritize efficiency and assume a safe environment [18, 30, 65, 70, 72]. E.g., Orion assumes that the co-located GPU workloads are in the same trust domain [60]. FreeRide provides the same security and isolation guarantees as the lower-level system it is built on. It incorporates MPS to limit GPU memory which provides separate GPU address spaces [44] for pipeline training and side tasks, and Docker for environment isolation [11, 69]. Orthogonally, security for co-located GPU workloads is an active research area [25, 32, 36, 37, 48, 73]. We expect future work to co-design security with efficient GPU sharing.

Fault tolerance. Since FreeRide supports generic side tasks, it is not possible for FreeRide to exhaustively implement fault tolerance mechanisms for all side tasks. Instead, FreeRide assumes that side tasks implement their own fault tolerance mechanisms to tolerate the failure of side tasks themselves and of pipeline training. In addition, since FreeRide deploys side tasks in Docker containers as processes that are independent of the pipeline training, failures of side tasks, such as illegal memory access, will not impact the main pipeline training workload.

Side task management. By implementing different strategies in its side task manager, FreeRide can incorporate more sophisticated management, e.g., co-locating multiple side tasks with various performance characteristics in the same worker to improve the utilization of bubbles [33] or serving side tasks with fairness or performance guarantees [5, 13].

Scalability. FreeRide can be extended for better scalability. As FreeRide implements communication among its components using RPCs, it can be easily extended to distributed settings with side tasks on multiple servers. During training, the side task manager of FreeRide receives bubbles from all GPUs from both remote servers and manages the side tasks that co-locate with each GPU. FreeRide can also be extended to support multi-GPU side tasks, e.g., distributed training and big data processing [8, 31], by launching workers with access to multiple GPUs.

Stability of pipeline training. FreeRide follows the same assumption as the previous pipeline parallel training works that pipeline training has a stable throughput and pattern, and that the training sequences have the same length after padding [12, 21, 34, 51].

Other ML accelerators. This work targets GPUs due to their widespread accessibility. FreeRide's mitigation for bubbles fundamentally applies to other ML accelerators, such as Google's TPU [22] and Meta's MTIA [35], provided that the platform has isolation and resource limit options for each process. We anticipate future work to incorporate the approach of FreeRide with other ML platforms.

Energy consumption. There has been recent interest in building energy- and carbon-efficient systems for machine learning workloads [6, 17, 20, 40, 55, 58]. We anticipate future work on energy efficiency of co-locating side tasks with LLM training.

9 Conclusion

We propose FreeRide, a middleware system that bridges the gap between the available yet hard-to-utilize bubbles in pipeline parallelism and running generic GPU side tasks to harvest them. It provides programming interfaces that abstract the life cycle of a side task as different states of a state machine and allows programmers to implement side tasks with little engineering effort. The side task manager and side task workers manage bubbles and side tasks, and reduce the performance overhead of side tasks on pipeline training. Our evaluation shows that, on average, FreeRide achieves 7.8% cost savings for long-running and expensive pipeline training with a negligible performance overhead of about 1%.

Acknowledgments

We thank the anonymous reviewers and our shepherd Ruben Mayer for their constructive feedback to improve this paper. This work was supported by the Natural Sciences and Engineering Research Council of Canada (NSERC) and the Undergraduate Research Assistantship program of the Cheriton School of Computer Science at the University of Waterloo.

References

[1] Amazon. Price of AWS G4 instances. https://aws.amazon.com/ec2/instance-types/g4/, 2024.

[2] Amazon. Price of AWS P4 instances. https://aws.amazon.com/ec2/instance-types/p4/, 2024.

[3] Zhihao Bai, Zhen Zhang, Yibo Zhu, and Xin Jin. PipeSwitch: Fast pipelined context switching for deep learning applications. In *14th USENIX Symposium on*

Operating Systems Design and Implementation (OSDI), 2020. https://dl.acm.org/doi/abs/10.5555/3488766.3488794.

[4] David Bernstein. Containers and cloud: From LXC to Docker to Kubernetes. *IEEE Cloud Computing*, 2014. https://doi.org/10.1109/MCC.2014.51.

[5] Quan Chen, Hailong Yang, Minyi Guo, Ram Srivatsa Kannan, Jason Mars, and Lingjia Tang. Prophet: Precise QoS prediction on non-preemptive accelerators to improve utilization in warehouse-scale computers. In *Proceedings of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2017. https://doi.org/10.1145/3037697.3037700.

[6] Sangjin Choi, Inhoe Koo, Jeongseob Ahn, Myeongjae Jeon, and Youngjin Kwon. EnvPipe: Performance-preserving DNN training framework for saving energy. In *USENIX Annual Technical Conference (ATC)*, 2023. https://www.usenix.org/conference/atc23/presentation/choi.

[7] Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, Parker Schuh, Kensen Shi, Sashank Tsvyashchenko, Joshua Maynez, Abhishek Rao, Parker Barnes, Yi Tay, Noam Shazeer, Vinodkumar Prabhakaran, Emily Reif, Nan Du, Ben Hutchinson, Reiner Pope, James Bradbury, Jacob Austin, Michael Isard, Guy Gur-Ari, Pengcheng Yin, Toju Duke, Anselm Levskaya, Sanjay Ghemawat, Sunipa Dev, Henryk Michalewski, Xavier Garcia, Vedant Misra, Kevin Robinson, Liam Fedus, Denny Zhou, Daphne Ippolito, David Luan, Hyeontaek Lim, Barret Zoph, Alexander Spiridonov, Ryan Sepassi, David Dohan, Shivani Agrawal, Mark Omernick, Andrew M. Dai, Thanumalayan Sankaranarayana Pillai, Marie Pellat, Aitor Lewkowycz, Erica Moreira, Rewon Child, Oleksandr Polozov, Katherine Lee, Zongwei Zhou, Xuezhi Wang, Brennan Saeta, Mark Diaz, Orhan Firat, Michele Catasta, Jason Wei, Kathy Meier-Hellstern, Douglas Eck, Jeff Dean, Slav Petrov, and Noah Fiedel. PaLM: Scaling language modeling with pathways. *J. Mach. Learn. Res.*, 2024. https://dl.acm.org/doi/10.5555/3648699.3648939.

[8] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified data processing on large clusters. *Commun. ACM*, 2008. https://doi.org/10.1145/1327452.1327492.

[9] DeepSpeed. Deepspeed 0.12.2. https://github.com/microsoft/DeepSpeed/tree/v0.12.2, 2023.

[10] DeepSpeed. Pipeline Parallelism in DeepSpeed. https://www.deepspeed.ai/tutorials/pipeline/, 2023.

[11] Docker. Docker security. https://docs.docker.com/engine/security/, 2024.

[12] Shiqing Fan, Yi Rong, Chen Meng, Zongyan Cao, Siyu Wang, Zhen Zheng, Chuan Wu, Guoping Long, Jun Yang, Lixue Xia, Lansong Diao, Xiaoyong Liu, and Wei Lin. DAPPLE: a pipelined data parallel approach for training large models. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2021. https://doi.org/10.1145/3437801.3441593.

[13] Ali Ghodsi, Matei Zaharia, Benjamin Hindman, Andy Konwinski, Scott Shenker, and Ion Stoica. Dominant resource fairness: Fair allocation of multiple resource types. In *8th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2011. https://dl.acm.org/doi/10.5555/1972457.1972490.

[14] GNU. Job control signals (the GNU C library). https://www.gnu.org/software/libc/manual/html_node/Job-Control-Signals.html.

[15] Erin Griffith. The desperate hunt for the A.I. boom's most indispensable prize. https://www.nytimes.com/2023/08/16/technology/ai-gpu-chips-shortage.html, 2023.

[16] gRPC. gRPC - a high performance, open source universal RPC framework. https://grpc.io/, 2024.

[17] Diandian Gu, Yihao Zhao, Peng Sun, Xin Jin, and Xuanzhe Liu. Greenflow: A carbon-efficient scheduler for deep learning workloads. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 36(2):168–184, 2025. https://doi.org/10.1109/TPDS.2024.3470074.

[18] Mingcong Han, Hanze Zhang, Rong Chen, and Haibo Chen. Microsecond-scale preemption for concurrent GPU-accelerated DNN inferences. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2022. https://www.usenix.org/conference/osdi22/presentation/han.

[19] Qinghao Hu, Zhisheng Ye, Meng Zhang, Qiaoling Chen, Peng Sun, Yonggang Wen, and Tianwei Zhang. Hydro: Surrogate-Based hyperparameter tuning service in datacenters. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pages 757–777, Boston, MA, July 2023. USENIX Association.

[20] Zhanqiu Hu, Esha Choukse, Rodrigo Fonseca, G Edward Suh, Udit Gupta, et al. Ecoserve: Designing carbon-aware ai inference systems. *arXiv preprint arXiv:2502.05043*, 2025. https://doi.org/10.48550/arXiv.2502.05043.

[21] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V Le, Yonghui Wu, and zhifeng Chen. GPipe: Efficient training of giant neural networks using pipeline parallelism. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2019. https://dl.acm.org/doi/abs/10.5555/3454287.3454297.

[22] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA)*, 2017. https://doi.org/10.1145/3079856.3080246.

[23] Andrej Karpathy. nanoGPT: The simplest, fastest repository for training/finetuning medium-sized GPTs. https://github.com/karpathy/nanoGPT, 2024.

[24] Chiheon Kim, Heungsub Lee, Myungryong Jeong, Woonhyuk Baek, Boogeon Yoon, Ildoo Kim, Sungbin Lim, and Sungwoong Kim. torchgpipe: On-the-fly pipeline parallelism for training giant models. *arXiv preprint arXiv:2004.09910*, 2020. https://doi.org/10.48550/arXiv.2004.09910.

[25] Taehun Kim and Youngjoo Shin. GPU side-channel attacks are everywhere: A survey. In *IEEE International Conference on Consumer Electronics - Asia (ICCE-Asia)*, 2020. https://doi.org/10.1109/ICCE-Asia49877.2020.9276805.

[26] Yehuda Koren, Robert Bell, and Chris Volinsky. Matrix factorization techniques for recommender systems. *Computer*, 2009. https://doi.org/10.1109/MC.2009.263.

[27] Lambda. Pricing of Lambda. https://lambdalabs.com/service/gpu-cloud#pricing, 2024.

[28] Chuan Li. OpenAI's GPT-3 language model: A technical overview. https://lambdalabs.com/blog/demystifying-gpt-3, 2020.

[29] Shigang Li and Torsten Hoefler. Chimera: Efficiently training large-scale neural networks with bidirectional pipelines. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2021. https://doi.org/10.1145/3458817.3476145.

[30] Gangmuk Lim, Jeongseob Ahn, Wencong Xiao, Youngjin Kwon, and Myeongjae Jeon. Zico: Efficient GPU memory sharing for concurrent DNN training. In *2021 USENIX Annual Technical Conference (ATC)*, 2021. https://www.usenix.org/conference/atc21/presentation/lim.

[31] Haotian Liu, Bo Tang, Jiashu Zhang, Yangshen Deng, Xiao Yan, Xinying Zheng, Qiaomu Shen, Dan Zeng, Zunyao Mao, Chaozu Zhang, Zhengxin You, Zhihao Wang, Runzhe Jiang, Fang Wang, Man Lung Yiu, Huan Li, Mingji Han, Qian Li, and Zhenghai Luo. GHive: Accelerating analytical query processing in Apache Hive via CPU-GPU heterogeneous computing. In *Proceedings of the 13th Symposium on Cloud Computing (SoCC)*, 2022. https://doi.org/10.1145/3542929.3563503.

[32] Sihang Liu, Yizhou Wei, Jianfeng Chi, Faysal Hossain Shezan, and Yuan Tian. Side channel attacks in computation offloading systems with GPU virtualization. In *IEEE Security and Privacy Workshops (SPW)*, 2019. https://doi.org/10.1109/SPW.2019.00037.

[33] Zihan Liu, Jingwen Leng, Zhihui Zhang, Quan Chen, Chao Li, and Minyi Guo. VELTAIR: Towards high-performance multi-tenant deep learning services via adaptive compilation and scheduling. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2022. https://doi.org/10.1145/3503222.3507752.

[34] Ziming Liu, Shenggan Cheng, Haotian Zhou, and Yang You. Hanayo: Harnessing wave-like pipeline parallelism for enhanced large model training efficiency. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2023. https://doi.org/10.1145/3581784.3607073.

[35] Meta. MTIA v1: Meta's first-generation AI inference accelerator. https://ai.meta.com/blog/meta-training-inference-accelerator-AI-MTIA/, 2023.

[36] Hoda Naghibijouybari, Ajaya Neupane, Zhiyun Qian, and Nael Abu-Ghazaleh. Rendered insecure: GPU side channel attacks are practical. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2018. https://doi.org/10.1145/3243734.3243831.

[37] Hoda Naghibijouybari, Ajaya Neupane, Zhiyun Qian, and Nael Abu-Ghazaleh. Side channel attacks on GPUs. *IEEE Transactions on Dependable and Secure Computing*, 2021. https://doi.org/10.1109/TDSC.2019.2944624.

[38] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R. Devanur, Gregory R. Ganger, Phillip B. Gibbons, and Matei Zaharia. PipeDream: Generalized pipeline parallelism for DNN training. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP)*, 2019. https://doi.org/10.1145/3341301.3359646.

[39] Deepak Narayanan, Amar Phanishayee, Kaiyu Shi, Xie Chen, and Matei Zaharia. Memory-efficient pipeline-parallel DNN training. In *Proceedings of the 38th International Conference on Machine Learning (ICML)*, 2021. https://doi.org/10.48550/arXiv.2006.09503.

[40] Sophia Nguyen, Beihao Zhou, Yi Ding, and Sihang Liu. Towards sustainable large language model serving. *ACM SIGENERGY Energy Informatics Review*, 4(5):134–140, 2024.

[41] Nvidia. Image resize and watermarking example using nvJPEG. https://github.com/NVIDIA/CUDALibrarySamples/tree/master/nvJPEG/Image-

Resize-WaterMark, 2019.

[42] Nvidia. CUDA C programming guide. https://docs.nvidia.com/cuda/cuda-c-programming-guide/, 2024.

[43] Nvidia. Multi-process service. https://docs.nvidia.com/deploy/mps/index.html, 2024.

[44] Nvidia. Nvidia multi-instance GPU memory protection. https://docs.nvidia.com/deploy/mps/index.html#memory-protection, 2024.

[45] Nvidia. Nvidia multi-instance GPU user guide. http://docs.nvidia.com/datacenter/tesla/mig-user-guide/index.html, 2024.

[46] Kazuki Osawa, Shigang Li, and Torsten Hoefler. PipeFisher: Efficient training of large language models using pipelining and fisher information matrices. In *Proceedings of Machine Learning and Systems (MLSys)*, 2023. https://doi.org/10.48550/arXiv.2211.14133.

[47] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The PageRank citation ranking: Bring order to the web. https://www.cis.upenn.edu/~mkearns/teaching/NetworkedLife/pagerank.pdf, 1998.

[48] Manos Pavlidakis, Giorgos Vasiliadis, Stelios Mavridis, Anargyros Argyros, Antony Chazapis, and Angelos Bilas. G-Safe: Safe GPU sharing in multi-tenant environments. *arXiv preprint arXiv:2401.09290*, 2024. https://arxiv.org/abs/2401.09290.

[49] PyTorch. Models and pre-trained weights — Torchvision main documentation. https://pytorch.org/vision/main/models.html.

[50] PyTorch. Pytorch profiler. https://pytorch.org/tutorials/recipes/recipes/profiler_recipe.html.

[51] Penghui Qi, Xinyi Wan, Guangxing Huang, and Min Lin. Zero bubble pipeline parallelism. *arXiv preprint arXiv:2401.10241*, 2023. https://doi.org/10.48550/arXiv.2401.10241.

[52] Jeff Rasley, Samyam Rajbhandari, Olatunji Ruwase, and Yuxiong He. DeepSpeed: System optimizations enable training deep learning models with over 100 billion parameters. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining (KDD)*, 2020. https://doi.org/10.1145/3394486.3406703.

[53] Corby Rosset. Turing-NLG: A 17-billion-parameter language model by Microsoft. https://www.microsoft.com/en-us/research/blog/turing-nlg-a-17-billion-parameter-language-model-by-microsoft/, 2020.

[54] RunPod. RunPod - The cloud built for AI. https://www.runpod.io/, 2024.

[55] Siddharth Samsi, Dan Zhao, Joseph McDonald, Baolin Li, Adam Michaleas, Michael Jones, William Bergeron, Jeremy Kepner, Devesh Tiwari, and Vijay Gadepally. From words to watts: Benchmarking the energy costs of large language model inference. In *2023 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–9. IEEE, 2023.

[56] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-LM: Training multi-billion parameter language models using model parallelism. *arXiv preprint arXiv:1909.08053*, 2020. https://doi.org/10.48550/arXiv.1909.08053.

[57] Shaden Smith, Mostofa Patwary, Brandon Norick, Patrick LeGresley, Samyam Rajbhandari, Jared Casper, Zhun Liu, Shrimai Prabhumoye, George Zerveas, Vijay Korthikanti, Elton Zhang, Rewon Child, Reza Yazdani Aminabadi, Julie Bernauer, Xia Song, Mohammad Shoeybi, Yuxiong He, Michael Houston, Saurabh Tiwary, and Bryan Catanzaro. Using DeepSpeed and Megatron to train Megatron-Turing NLG 530B, A large-scale generative language model. *arXiv preprint arXiv:2201.11990*, 2022. https://doi.org/10.48550/arXiv.2201.11990.

[58] Jovan Stojkovic, Chaojie Zhang, Íñigo Goiri, Josep Torrellas, and Esha Choukse. Dynamollm: Designing llm inference clusters for performance and energy efficiency. *arXiv preprint arXiv:2408.00741*, 2024.

[59] Foteini Strati, Paul Elvinger, Tolga Kerimoglu, and Ana Klimovic. ML training with cloud GPU shortages: Is cross-region the answer? In *Proceedings of the 4th Workshop on Machine Learning and Systems*, 2024. https://doi.org/10.1145/3642970.3655843.

[60] Foteini Strati, Xianzhe Ma, and Ana Klimovic. Orion: Interference-aware, fine-grained GPU sharing for ml applications. In *Proceedings of the Nineteenth European Conference on Computer Systems (EuroSys)*, 2024. https://doi.org/10.1145/3627703.3629578.

[61] Ding Tang, Lijuan Jiang, Jiecheng Zhou, Minxi Jin, Hengjie Li, Xingcheng Zhang, Zhilin Pei, and Jidong Zhai. ZeroPP: Unleashing exceptional parallelism efficiency through tensor-parallelism-free methodology. *arXiv preprint arXiv:2402.03791*, 2024. https://doi.org/10.48550/arXiv.2402.03791.

[62] John Thorpe, Pengzhan Zhao, Jonathan Eyolfson, Yifan Qiao, Zhihao Jia, Minjia Zhang, Ravi Netravali, and Guoqing Harry Xu. Bamboo: Making preemptible instances resilient for affordable training of large DNNs. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2023. https://www.usenix.org/conference/nsdi23/presentation/thorpe.

[63] Pablo Villalobos, Jaime Sevilla, Tamay Besiroglu, Lennart Heim, Anson Ho, and Marius Hobbhahn. Machine learning model sizes and the parameter gap. *arXiv preprint arXiv:2207.02852*, 2022. https://doi.org/10.48550/arXiv.2207.02852.

[64] Bingyang Wu, Zili Zhang, Zhihao Bai, Xuanzhe Liu, and Xin Jin. Transparent GPU sharing in container clouds for deep learning workloads. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2023. https://www.usenix.org/conference/nsdi23/presentation/wu.

[65] 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. Gandiva: Introspective cluster scheduling for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2018. https://dl.acm.org/doi/10.5555/3291168.3291212.

[66] Wencong Xiao, Shiru Ren, Yong Li, Yang Zhang, Pengyang Hou, Zhi Li, Yihui Feng, Wei Lin, and Yangqing Jia. AntMan: Dynamic scaling on GPU clusters for deep learning. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2020. https://dl.acm.org/doi/10.5555/3488766.3488796.

[67] Zhen Xu, Xuhao Chen, Jie Shen, Yang Zhang, Cheng Chen, and Canqun Yang. Gardenia: A graph processing benchmark suite for next-generation accelerators. *J. Emerg. Technol. Comput. Syst.*, 2019. https://doi.org/10.1145/3283450.

[68] Jaewon Yang and Jure Leskovec. Defining and evaluating network communities based on ground-truth. In *Proceedings of the ACM SIGKDD Workshop on Mining Data Semantics*, 2012. https://doi.org/10.1145/2350190.2350193.

[69] Robail Yasrab. Mitigating Docker security issues. *arXiv preprint arXiv:1804.05039*, 2023. https://doi.org/10.48550/arXiv.1804.05039.

[70] Peifeng Yu and Mosharaf Chowdhury. Fine-grained GPU sharing primitives for deep learning applications. In *Proceedings of Machine Learning and Systems (MLSys)*, 2020. https://proceedings.mlsys.org/paper_files/paper/2020/hash/d9cd83bc91b8c36a0c7c0fcca59228f2-Abstract.html.

[71] Susan Zhang, Stephen Roller, Naman Goyal, Mikel Artetxe, Moya Chen, Shuohui Chen, Christopher Dewan, Mona Diab, Xian Li, Xi Victoria Lin, Todor Mihaylov, Myle Ott, Sam Shleifer, Kurt Shuster, Daniel Simig, Punit Singh Koura, Anjali Sridhar, Tianlu Wang, and Luke Zettlemoyer. OPT: Open pre-trained transformer language models. *arXiv preprint arXiv:2205.01068*, 2022. https://doi.org/10.48550/arXiv.2205.01068.

[72] Wei Zhang, Binghao Chen, Zhenhua Han, Quan Chen, Peng Cheng, Fan Yang, Ran Shu, Yuqing Yang, and Minyi Guo. PilotFish: Harvesting free cycles of cloud gaming with deep learning training. In *USENIX Annual Technical Conference (ATC)*, 2022. https://www.usenix.org/conference/atc22/presentation/zhang-wei.

[73] Yicheng Zhang, Ravan Nazaraliyev, Sankha Baran Dutta, Nael Abu-Ghazaleh, Andres Marquez, and Kevin Barker. Beyond the bridge: Contention-based covert and side channel attacks on multi-GPU interconnect. *arXiv preprint arXiv:2404.03877*, 2024. https://arxiv.org/abs/2404.03877v2.

[74] Lianmin Zheng, Zhuohan Li, Hao Zhang, Yonghao Zhuang, Zhifeng Chen, Yanping Huang, Yida Wang, Yuanzhong Xu, Danyang Zhuo, Eric P. Xing, Joseph E. Gonzalez, and Ion Stoica. Alpa: Automating inter- and intra-operator parallelism for distributed deep learning. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2022. https://www.usenix.org/conference/osdi22/presentation/zheng-lianmin.