PIPEFILL: USING GPUS DURING BUBBLES IN PIPELINE-PARALLEL LLM TRAINING

Daiyaan Arfeen [1 *] **Zhen Zhang** [2] **Xinwei Fu** [2] **Gregory R. Ganger** [1] **Yida Wang** [2]

ABSTRACT

Training Deep Neural Networks (DNNs) with billions of parameters generally involves pipeline-parallel (PP) execution. Unfortunately, PP model training can use GPUs inefficiently, especially at large scale, due to idle GPU time caused by *pipeline bubbles*, which are often 15–30% and can exceed 60% of the training job's GPU allocation. To improve the GPU utilization of PP model training, this paper describes PIPEFILL, which fills pipeline bubbles with execution of *other* pending jobs. By leveraging bubble GPU time, PIPEFILL reduces the GPU utilization sacrifice associated with scaling-up of large-model training. To context-switch between fill jobs and the main training job with minimal overhead to the main job, and maximize fill job efficiency, PIPEFILL carefully fits fill job work to measured bubble durations and GPU memory availability, introduces explicit pipeline-bubble instructions, and orchestrates placement and execution of fill jobs in pipeline bubbles. Experiments show that PIPEFILL can increase overall utilization by up to 63% for GPUs used in large-scale LLM training, with <2% slowdown of the training job, and 5–15% even for low-scale LLM training. For large-scale LLM training on 8K GPUs, the 63% increase translates to up to 2.6K additional GPUs worth of work completed.

1 INTRODUCTION

DNN models with billions of parameters have exploded in popularity with the emergence of generative AI applications. For example, popular large-language models (LLMs), such as GPT (3) and LLaMA (26; 27), are creating disruptive change in many domains. But training such models can take several weeks or months even using thousands of GPUs[1].

A common approach (20; 33) of training on thousands of GPUs is to employ a combination of parallelization techniques. Pipeline-parallelism (PP) (19; 9) is used to partition the model across multiple nodes, creating a pipeline of stages. The full pipeline is then replicated using data-parallelism, allowing for parallel processing of multiple data samples. Within each pipeline stage, tensor-parallelism is applied to partition the model weights, enabling parallel computation. Each minibatch of data is further divided into smaller subsets called *microbatches*. The forward and backward passes for each microbatch are then executed in a pipelined manner across the stages.

Unfortunately, such highly-parallelized training can use GPUs inefficiently especially at large scale, because too much GPU time may be wasted on *pipeline bubbles*. Pipeline bubbles occur because the pipeline must be fully drained and then restarted for each minibatch, leading to idle time on each of the GPUs. The greater the parallelization is, whether from longer pipelines (taking longer to fill and drain) or more pipeline replicas (reducing the number of microbatches per replica as the global minibatch

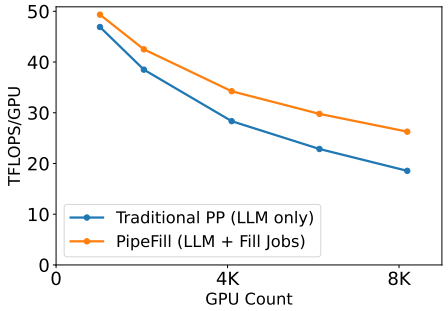


Figure 1. Scaling out training of a 40B-parameter LLM from 1K GPUs to 8K GPUs to reduce training time from 82 days to 26 days. Traditionally, the increasing pipeline bubbles when scaling out leads to over 60% lower GPU utilization at 8K. PIPEFILL is able to fill much of that bubble GPU time with useful work, without slowing the LLM training. Section 4 details the experimental setup.

[*]Work done during internship at Amazon Web Services. [1]Carnegie Mellon University [2]Amazon Web Services. Correspondence to: Daiyaan Arfeen <marfeen@andrew.cmu.edu>.

Proceedings of the 8^{th} MLSys Conference, Santa Clara, CA, USA, 2025.

[1]This paper uses "GPU" or "device" to refer to any computation accelerator for deep learning jobs, such as GPUs, TPUs (12), or AWS Trainium (1).

size needs to be fixed), the greater the inefficiency becomes due to bubbles. For example, Figure 1 shows that a 40B-parameter auto-regressive-transformer LLM, parallelized over 8K GPUs achieves 60% lower TFLOPS-per-GPU than using just 1K GPUs because of pipeline bubbles—but using only 1K GPUs would make LLM training take over 3× longer (26 days vs. 82 days; shown in Figure 5a). The overall consequence is a major tension between LLM training time and GPU cluster efficiency.

PIPEFILL is a new GPU management system that mitigates this tension by filling large training jobs' pipeline bubbles with *other* jobs, which we call *fill* jobs. The GPUs for any given pipeline stage switch to a fill job at the start of a bubble and switch back at the end of that bubble. By doing so, PIPEFILL recaptures otherwise wasted GPU time to accomplish pending inference and training jobs, which can enable scaling-up large-model training with much less sacrifice in GPU utilization. Figure 1 shows how, with bubble filling, PIPEFILL mitigates the GPU utilization penalty as LLM training scales out. At 8K GPUs, for example, PIPEFILL increases GPU utilization by over 45% with a mix of training and inference fill jobs. If using just less GPU memory intensive batch inference jobs, the GPU utilization increase grows to 63% (see Figure 5c).

Filling pipeline bubbles effectively requires overcoming a number of challenges. First, fill job execution needs to be configured to fit within bubble constraints, including bubble length (to minimize inter-bubble context) and available GPU memory. PIPEFILL introduces a *Pipeline Bubble Instruction* to collect bubble constraints, and a *Fill Job Execution Plan Algorithm* to partition a fill job into chunks prior to bubble filling as necessary. Second, the right fill jobs need to be matched to the right GPUs, given that pipeline bubbles exhibits heterogeneous characteristics and users may have different optimization objectives. PIPEFILL uses a *Fill Job Scheduler*, which accepts user-defined scheduling policies. Our Fill Job Scheduler orchestrates the assignment of fill jobs to GPUs by synergizing the user-defined policy with the characterization of the main job's pipeline bubbles.

Experiments (real system and simulation) confirm that PIPEFILL can recapture significant GPU utilization lost to pipeline bubbles, allowing huge DNNs (like LLMs) to be scaled out without much lower GPU efficiency consequences. At each scale, aggregated TFLOPS/GPU (fill jobs plus LLM) is higher with PIPEFILL, from 5–15% at (slow) low-scale LLM training to over 63% for scaled-out training, with <2% slowdown of the LLM training. Detailed analysis of different fill jobs options shows that, as expected, the limited memory and intermittent time available for fill job execution in bubbles reduces their efficiency differently–the data in Figure 1 is for a fill job mix derived from an ML job trace, but using just bubble-efficient batch inference jobs increases the gains by ≈50%. Additional results confirm that PIPEFILL's benefits are realized for both GPipe (9) and 1F1B (19) pipeline schedules, with moderate reduction (17%) in benefits for 1F1B at low-scale and minimal difference ($< 5\%$) at large-scale, and show fill job efficiency sensitivity to changes in bubble durations, available memory during bubbles, and fill-job scheduling policy.

Contributions. This paper makes four main contributions: 1) It introduces the concept of filling pipeline bubbles in PP model training with execution of *other* ML jobs; 2) It describes a system (PIPEFILL) that realizes this concept and can recover idle GPU-time lost to pipeline bubbles; 3) It introduces approaches for assigning fill jobs to pipeline bubbles, and for configuring fill job execution within its assigned bubble, to maximize efficiency of recovered GPU time; 4) It experimentally shows that PIPEFILL can significantly increase GPU utilization for scale-out LLM training without significantly harming LLM training efficiency.

2 BACKGROUND AND MOTIVATION

DNN training involves running variants of stochastic gradient descent (SGD) to optimize parameters that minimize a loss function. SGD takes a number of samples (a minibatch) from a dataset on each iterations and calculates a gradient on those samples to update the parameters. Training large DNNs consumes significant GPU memory, often surpassing the capacity of a single device, necessitating the use of partitioning strategies like tensor and pipeline parallelism, in addition to data parallelism, to distribute the workload across multiple devices. For training large DNNs, such as LLMs, it is common to combine all three parallelization techniques.

Tensor parallelism (TP) spreads computation across devices and resolves data dependencies through communication operations, which introduces overhead. This approach is most effective within a single node with high-bandwidth connections (e.g., NVLink) to reduce communication delays but limits scalability to the GPU memory available within that node (20; 33).

Pipeline parallelism (PP) divides the model by layers, allowing multi-node scaling by processing microbatches (partitions of a minibatch) in a pipeline. However, data dependencies and gradient synchronizations creates pipeline *bubbles* which are periods of idle time when the GPUs do not yet have work to do. For unidirectional, synchronous pipeline schedules (such as GPipe and 1F1B), the fraction of time spent in bubbles is $(p-1)/(m+p-1)$, where p is the number of pipeline stages, and m is the number of microbatches that splits from minibatch (20).

Data parallelism (DP) replicates model instances across devices, each processing a distinct partition of each minibatch.

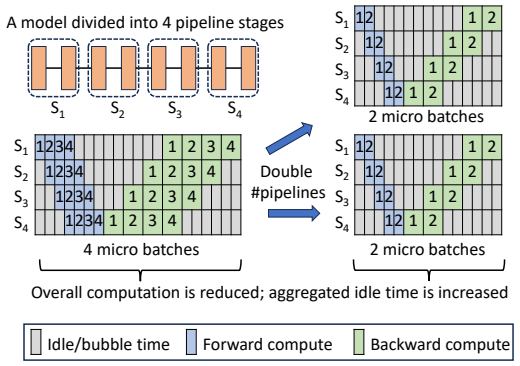


Figure 2. PP combined with DP. Replicating the pipeline (double the number of GPUs) with the overall minibatch size fixed (at 4 microbatches) leads to shorter per-minibatch execution time but a larger fraction of GPU time is lost to pipeline bubbles.

Synchronization is maintained through collective all-reduce operations at the end of each minibatch.

2.1 Pipeline Bubbles Lower GPU Utilization

In combined parallelism, training larger models requires more pipeline stages. Since TP is limited by the number of GPUs in a node, once this limit is reached, the only option is to increase the number of pipeline stages to ensure each partition fits within the available GPU memory (increasing p in the pipeline bubble ratio). Additionally, the minibatch size, or the number of samples on which each gradient update is calculated, is usually set by machine learning experts and remains constant when scaling up training. For example, both LLaMA-1 (26) and LLaMA-2 (27) training use 4 million tokens for each model update. Experts are reluctant to increase the minibatch size, as it can hurt model quality at the end of training (17). With a fixed total computation workload for each round of model updating, increasing DP results in a smaller number of microbatches, or m in the pipeline bubble ratio expression. Thus, inefficient GPU utilization caused by pipeline bubbles is inevitable when employing PP, and it becomes particularly noticeable when scaling up the training of large models using DP. There are also more advanced methods like automatically explore the best possible sharding strategies (33) which may achieve better training throughput than manual decisions. However, all of them are inherently bounded by the total computation workload per model updating, hence getting diminishing return while increasing the compute resource.

2.2 Solution: Fill Bubbles w/ Independent Jobs

How can idle GPU time resulting from pipeline bubbles be utilized to improve GPU utilization? Existing works fill dependent jobs of the training job running with PP into the pipeline bubbles. PipeFisher (21) accelerates convergence by utilizing the pipeline bubbles to execute K-FAC, a second-order optimization method. Similarly, Bamboo (25) enhances training resilience at a minimal cost by filling redundant computations into the pipeline bubbles. However, the jobs filled into the pipeline bubbles by existing works are dependent on the training job running with PP providing extra work, making them only applicable to specific types of training jobs (training jobs optimized using K-FAC in the case of PipeFischer, jobs running on faulty/spot machines in the case of Bamboo).

Fundamentally, pipeline bubbles exist due to data dependencies within the computation of pipeline parallelism. Our key insight is that, rather than directly addressing the data dependency issue within a training job pattern or introducing other dependencies by filling dependent jobs, we leverage independent jobs, unrelated to the training job running with pipeline parallelism, to fill the pipeline bubbles. Specifically, we remove the constraint that the training job must execute exclusively on the GPUs during the entirety of the job. One can context-switch to a different job during the bubbles to reduce the amount of idle time of GPUs, and context-switch back to the main training job in time for the training job to experience no overhead from sharing the GPU during the pipeline bubbles. To fill independent jobs into pipeline bubbles, we need to address the following challenges:

- **Memory Management.** How can one fill independent jobs into the pipeline bubbles when the GPU memory is primarily occupied by the main training job? Even during pipeline bubbles, the main training job dominates the GPU memory. Naively filling independent jobs into pipeline bubbles without careful memory management may result in GPU OOM errors or sub-optimal performance of fill jobs. Effective memory management is crucial not only to mitigate OOM risks but also to optimize available memory for fill jobs.

- **Context Switching.** How can one ensure that filling independent jobs into the pipeline bubbles does not incur performance penalties for the main training job? To maintain the performance of the main training job, only pipeline bubbles can be utilized for running fill jobs. However, it's not guaranteed that a fill job can be completed within one bubble. Therefore, filling independent jobs into pipeline bubbles without carefully crafted context switching may introduce performance penalties to the main training job.

- **Fill Job Scheduling.** When faced with numerous pipeline bubbles exhibiting heterogeneous characteristics, how can one effectively schedule the filling process to align with user-specific objectives? Pipeline bubbles across various pipeline stages exhibit distinct characteristics, such as duration and HBM availability. Additionally, users may harbor unique optimization goals; for instance, some prioritize GPU utilization, while others emphasize meeting job deadlines promptly. Naively scheduling the filling process without accounting for bubble characteristics and

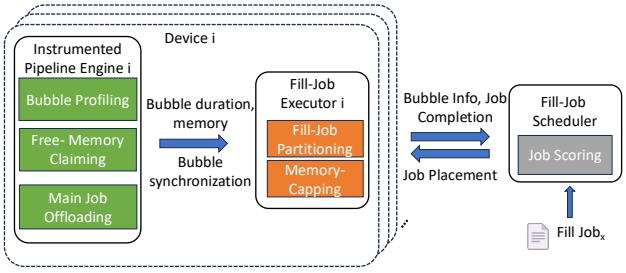


Figure 3. System overview

users' optimization objectives risks compromising the performance of fill jobs and falling short of users' expectations.

Other works, such as Muri (32) and Antman (29), explore interleaving multiple jobs on shared GPUs. However, these works do not specifically address scheduling alongside a main job running with PP, thus failing to leverage the unique characteristics for optimization. Muri only considers job duration of each job as a constraint and assumes all jobs fit together in GPU memory. Thus, Muri lacks support for guaranteeing the main job performance and also falls short in memory management and fill job scheduling. Antman utilizes device statistics to assign memory caps to jobs based on priority and fills idle GPU cycles with opportunistic kernels. However, when a main job is running with PP, pipeline bubbles often appear as long-running communication kernels, causing Antman to struggle in determining context switches between the main job and fill jobs. Moreover, the main job typically consumes the majority of the memory, making simply setting memory caps insufficient for memory management.

3 Design and Implementation

Shown in Figure 3, PipeFill consists of three major components: *Instrumented Pipeline Engine*, *Fill Job Executor*, and *Fill Job Scheduler*. The Instrumented Pipeline Engine uses the *pipeline bubble instruction* to measure when a pipeline bubble begins and ends, and the available memory during a pipeline bubble. The Fill Job Executor then leverages those information to decide the strategy of filling a job into bubbles, including whether and how to partition a filled job into execution chunks and whether to offload the memory of the main job to free up space for the filled job. The pipeline engine signals the Executor using synchronization primitives when a bubble begins and it can begin running the fill job. The Fill Job Scheduler accepts user-defined scheduling policies and schedules filled jobs onto device pipeline-bubbles to optimize the chosen policy and synergize the user-defined policy with the characterization of the main job's pipeline bubbles.

Putting together. Fill jobs are initially received by the Scheduler, which makes scheduling decisions about which device's pipeline bubbles to execute a fill-job on. Scheduling policies can be defined to modify the behavior of the Scheduler. Each device has a fill-job Executor process. Once a fill-job arrives at a device, the Executor uses profiling data to construct an execution plan for the job. The plan maximizes the throughput of the job by choosing a batch size and creating partitions of the job's computational graph that maximize the amount of work completed during the pipeline bubbles without violating bubble duration or free-memory constraints. The Executor uses synchronization primitives to know when to begin execution of the next fill-job graph partition. The pipeline engine runs on every device worker. The pipeline engine uses the pipeline bubble instruction to know when a pipeline bubble is beginning so it can signal the Executor (using the aforementioned synchronization primitives) to begin execution. Before the pipeline engine signals the Executor, it tells the device memory allocator to release all transient/unused memory buffers to increase the free-memory available to the Executor and waits for any main-job offloading operations to complete.

Fill Jobs. In this work, we use deep learning training and batch-inference jobs as fill jobs. Deep learning jobs can be classified as training or inference. Training is typically not latency sensitive and is often long-running. Inference can be broken down into real-time/online inference and batch/offline inference; the former is latency sensitive, with SLOs being on the order of milliseconds, while the later is often not latency sensitive. Batch inference, which is not latency-sensitive, is widely used in applications such as content recommendation systems, data analytics, and other back-end services. Due to the intermittent property of pipeline bubbles, latency-sensitive jobs are not suitable for use as fill-jobs. Therefore, PipeFill supports training and batch-inference jobs as fill-jobs. PipeFill takes as input the model used for the fill-job, as well as valid batch-sizes; given the job configuration, it will attempt to execute the fill-job with maximum throughput.

3.1 Pipeline Engine Instrumentation

To not impact the main job, we must keep the context switching and execution of fill jobs completely within the duration of the pipeline bubbles. We must also know exactly how much GPU memory the fill job can use during its execution so that the main job does not experience an OOM error. In order to achieve this, PipeFill augments existing pipeline engines with a new pipeline bubble instruction. Existing pipeline engines execute a sequence of pipeline instructions, which include sending/receiving activations and gradients, executing forward/backward computations on specific microbatches, and synchronizing parameters. Taken together as a periodically repeating sequence, these instructions constitute a pipeline schedule, which can have

multiple pipeline bubbles appearing as instructions that wait on some event (e.g. activation data to arrive from the previous stage). PIPEFILL's bubble instruction is inserted into the schedule to indicate where large bubbles are expected to occur.

Bubble characterization. Before any bubble filling, the pipeline engine must determine the duration of each pipeline bubble in the pipeline schedule and how much memory is available for the fill jobs to use. To this end, at the beginning of the main training job the pipeline engine does profiling. For each bubble instruction, the pipeline engine will wait certain amount of time (e.g. 100 ms) before proceeding to execute the next instruction. It will then observe the main job's throughput, if it is unaffected then on the next minibatch iteration it will wait $2\times$ amount of time. This will continue until the pipeline engine observes a drop in the main job's throughput, at which point it will know the duration of the pipeline bubble. To profile the amount of memory available for the engine to use for the fill job during a pipeline bubble, the engine relies on PyTorch's `torch.cuda.memory_allocated()` function to know how much memory is held by the main training job during the bubble; the remaining device memory is considered free, but to ensure there are no out-of-memory errors PIPEFILL may opt only to allocate some fraction of the free memory. Additionally, to ensure transient/temporary memory buffers are not counted as allocated by the main job (and instead can be used by the fill-jobs), the engine will tell the memory allocator to free all such buffers (by calling torch.cuda.empty_cache()). The bubble duration and free-memory capacity is passed to the Executor so it can avoid violating those constraints.

Bubble signaling. Once the engine has characterized the pipeline bubbles they can be filled. The engine starts a new Executor process (with a shared synchronization primitive) and passes the bubble information to it. When a new fill-job is sent by the Scheduler, the engine passes the job description (as well as the necessary profiles) to the Executor. Every time the pipeline engine reaches a bubble instruction, it 1) tells the memory allocator to free all unused memory 2) waits for any main job offloading operations to complete 3) signals the Executor to begin running its fill-job.

Main job offloading. In some cases, it may be beneficial to increase the amount of free-memory available to the fill-jobs. To achieve this, PIPEFILL enables offloading of main job data from device to CPU memory. In order to do this in a way that is transparent to the main job and does not sacrifice its performance, which data is offloaded must be carefully chosen and the data transfer operations must be coordinated so the main job is never blocked on them. PIPEFILL enables offloading of the main job optimizer states (e.g. moment estimates for Adam(13)) because this data is only required by the main job during the optimizer updates. The offloading is overlapped with forward-pass execution, and the onloading is overlapped with gradient-synchronization; a significant amount of data can be offloaded in this fashion with no impact to the main job. The pipeline engine forward-pass and gradient-synchronization instructions are augmented to launch these operations on a separate CUDA stream.

3.2 Executor

The Executor is a process that executes a fill-job on a device's pipeline bubbles with maximum throughput without violating the bubble duration or free-memory constraints, ensuring that the fill-job execution has no impact on the main job performance. It does this by creating an execution plan for the fill job that chooses a batch size and partitions the job's computational graph, and it relies on signals from the pipeline engine to know when to execute the graph partitions.

Execution plan. When created, the Executor is passed a sequence of bubble durations and free-memory capacities from the pipeline engine. This sequence describes the pipeline bubbles, the resources and durations that they each make available as well as their order. This sequence of bubbles is a cycle of bubbles that repeats every minibatch iteration of the main job. When a fill-job is passed to the Executor, it is accompanied with a set of profiles containing the execution time and memory requirement of each node in the computational graph under a specific configuration . Configurations can be different batch sizes and different execution techniques (e.g. CPU-offloading or NVMe-offloading of parameters/gradients/optimizer states, activation checkpointing/offloading). The Executor linearizes the computational graph and its profiles, turning it into a sequence of nodes with sequential dependency. For each configuration, the Executor packs the computational graph into as few bubble cycles as possible (without violating duration and free-memory constraints). The Executor runs a greedy algorithm (shown in Algorithm 1) that does the following: 1) replicate the graph enough times (each replica represents an iteration) that the total execution time is as high as possible without exceeding the total bubble time (lines 3-6), 2) iteratively packs as many source nodes of the remainder of the computational graph as possible into the next bubble (lines 7-14) without exceeding its duration or memory limits. This sequence of computational graph partitions represents the Executor's plan for the fill-job.

Bubble synchronization and memory capping. When executing the fill-job plan, the Executor waits for signals from the pipeline engine to know when the main job has entered a pipeline bubble. When it receives a signal, it first sets a cap on the amount of device memory that it can use (by using PyTorch's `cuda.set_per_process_memory_fraction` function) to the amount of free-memory available in the

bubble; if the Executor somehow exceeds this memory capacity, it will experience an OOM error, but this error will be isolated to the Executor process and will not affect the main job. The Executor will execute the current graph partition on the current bubble, and then wait for the next signal from the pipeline engine.

Algorithm 1 Partition fill job onto bubbles

Input: A list B of the bubble durations, a list M of bubble free-memory capacities, a list F of the graph-node durations and memory requirements
Output: List P of graph partitions where duration of $P[i] \leq B[i \mod \text{len}(B)]$ and memory of $P[i] \leq M[i \mod \text{len}(M)]$
$F' \leftarrow F$
while $dur(F') + dur(F) < \sum B$ **do**
  $F' \leftarrow F' + F$
end while
$F \leftarrow F'$; $P \leftarrow []$; $i \leftarrow 0$
while len$(F) > 0$ **do**
  $P' \leftarrow []$
  while len$(F) > 0$ **and** $dur(P') + dur(F[0]) < B[i]$ **and** mem$(F[0]) \leq M[i]$ **do**
    $P' \leftarrow P' + F[0]$; $F \leftarrow F[1:]$
  end while
  $P \leftarrow P + P'$; $i \leftarrow (i+1) \mod \text{len}(B)$
end while
return P

3.3 Scheduler

The Scheduler is the interface between the pipeline bubbles of the main job and outside higher-level cluster schedulers, making the bubbles available as additional resources. The Scheduler is also responsible for scheduling the fill-jobs onto the pipeline bubbles. The Scheduler has access to the fill-job profiles, partitioning algorithm, and bubble descriptions of every device. Using this information, the Scheduler is able to calculate any fill-job's throughput/processing-time on any device. The Scheduler exposes the scheduling policy by defining a function that takes as input a job's information (arrival time, processing-time on every possible device, and deadline) as well as the current state of all the Executors in the system, and outputs a score. When a device completes a fill-job, the Scheduler chooses which job to submit to the device by choosing the job which maximizes the score. This allows specifying a variety of different scheduling policies. For example, to specify a Shortest-Job-First policy the function can be defined as: $f(j,s,i) = \frac{1}{min(j.proc\_times)}$ where $j.proc\_times$ is a list containing the job's processing times on all devices, s is the current state of all Executors, and i is the index of the Executor which is to be filled. A more complex example is a policy that minimizes the makespan, which can be specified with the function: $f(j,s,i) = \frac{1}{max(j.proc\_times[i], s.rem\_times)}$ where $s.rem\_times$ is a list containing the remaining amount of time each Executor will be busy. This policy will minimize the maximum busy time across all Executors, thereby minimizing makespan. By defining the policy using weighted compositions of multiple functions, hierarchical policies can be defined that behave differently under different circumstances. For example, policies can be defined that prioritize proximity-to-deadline as a feature, but default to more standard policies (e.g. SJF, FIFO) when there are no jobs with deadlines.

Since the Scheduler knows how long the currently executing fill-jobs will take to complete, as well as the order in which the queued fill-jobs will be executed, users can query the Scheduler to know when a currently submitted fill-job is expected to complete or whether a fill-job's deadline can be met under current conditions. This can be used by a higher-level scheduler, which manages other resources in addition to the pipeline bubbles, to make scheduling decisions about which of its jobs can be submitted to the Scheduler.

3.4 Implementation

Our implementation is based on DeepSpeed (18) . Like many distributed training frameworks, DeepSpeed creates a process to manage each GPU and processes are grouped by their membership in tensor, data, and pipeline parallel communication groups. Each process uses a pipeline engine to execute a pipeline schedule. We augment the DeepSpeed pipeline engine and schedule with the instrumentation for bubble filling and main job offloading. Figure 4 shows an example of a pipeline schedule (implemented as a function) which bubble filling inserted. PIPEFILL additional creates a new Executor process for each GPU (which also uses DeepSpeed to execute the fill jobs). The main job training process uses IPC to signal it's corresponding Executor process when bubbles occur and the GPU is free for the Executor to run fill jobs. To support large-model fill-jobs with limited GPU free-memory, the Executor is enabled with fill-job configurations that use CPU-offloading and activation checkpointing. In particular, the Executor will consider using ZeRO-Offload(23) and ZeRO-Infinity(22) to offload optimizer states, gradients, activations, and parameters of the fill-job.

Main job pipeline schedule. We consider GPipe(9) and 1F1B(19) schedules for the main job. Both schedules exhibit two-phase bubble behavior: one bubble occurs between the drain of the previous minibatch iteration and the fill of the next iteration (fill-drain), and the other bubble occurs between the forward-pass pipeline saturation and the backward pass (fwd-bwd). The fill-drain bubble of both schedules is the same, but the fwd-bwd bubbles can be different. For GPipe, the fwd-bwd bubble duration is $(num\_stages - stage\_id - 1) * (t_{fwd} + t_{bwd})$ whereas for 1F1B its duration is $(num\_stages - stage\_id - 1) * t_{bwd} + max(0, num\_stages - stage\_id - m) * t_{fwd}$. 1F1B additionally has some non-contiguous bubbles (which PIPEFILL

size	model	# parameters	job type
S	EfficientNet(24)	117M	CV
S	Bert-base(4)	109M	NLP
M	Bert-large(4)	334M	NLP
M	Swin-large(16)	779M	CV
L	XLM-Roberta-XL(6)	2.8B	NLP

S: small **M**: medium **L**: large

Table 1. Fill job category.

does not fill), which makes the total bubble time the same for both schedules.

```
    def gpipe_steps(self):
        total_steps = 2 * (self.micro_batches + self.
             stages - 1)
        for step_id in range(total_steps):
            # Map steps to micro-batch id, fwd vs bwd
            micro_batch_id, is_forward = self.
                 _step_to_micro_batch(step_id)

            cmds = []
            # Exchange activations
            if is_forward:
                    SendRecvActivation()
            else:
                    SendRecvGrad()

            # Computation
            if self._valid_micro_batch(micro_batch_id):
                if is_forward:
                    ForwardPass()
                else:
                    BackwardPass()

            if micro_batch_id == self.micro_batches and
                  is_forward:
                FillBubble()

            # Model step at the end of the batch
            if step_id == total_steps - 1:
                ReduceGrads()
                OptimizerStep()
                FillBubble()

            yield cmds
```

Figure 4. Example python code for GPipe schedule function, and how pipeline bubble filling is inserted. **FillBubble** function uses IPC to signal the Executor to start running fill jobs.

4 Experimental Setup

Hardware and Simulator We use a cluster of 16 AWS EC2 *p3.16xlarge* instances to run small-scale experiments and to collect traces for large-scale simulation experiments. Each node contains 8 NVIDIA V100 GPUs, each equipped with 16GB HBM and 125 TFLOPS of peak compute. GPUs on the same machine are connected with NVLink, and separate machines are connected with 25 Gbps network bandwidth.

To evaluate our system on multiple large-scale settings, we create an event-driven simulator. Deep learning jobs have repetitive patterns, so an accurate simulator only needs to profile a pattern once to simulate the time and resources it takes to repeat that pattern. Our simulator relies on profiles of the main training jobs' pipeline instructions and the fill jobs' layers (under different configurations). The events in our simulator are the arrivals and completions of fill-jobs (since these are when the state of the system can change), and we simulate the time between these events using the profiled execution times and the job arrivals from the trace.

Main Jobs Our physical cluster experiments use a 5B parameter LLM training job as the main job, and are executed on 16 GPUs on separate machines (16 pipeline-stages, no tensor-parallelism). We also collect profiles of a 40B parameter LLM training job executed using 8-way tensor-parallelism (8 GPUs per machine) and 16-stage pipeline-parallelism (16 machines). The simulator main job has almost the same settings as the physical cluster job, only scaled up using tensor-parallelism ; consequently the bubble sizes are almost identical. We use the profiles of the 40B model training job to seed our simulator, which we use for sensitivity studies done in simulation.

Both main jobs use sequence length of 2048 tokens per sample, 2 samples per microbatch, and 1024 samples per minibatch (across all data-parallel replicas); both jobs use the Adam(13) optimizer. We use the GPipe schedule by default, unless otherwise specified. Data-parallel execution has been shown to be predictable(14), so we run only one data-parallel replica across all our experiments, varying the number of microbatches according to different data-parallel configurations.

Fill Jobs We create our fill-job traces in two steps. First, we construct a fill-job model distribution. We extract all model sizes and model types from the HuggingFace (HF) Model Hub(10); we filter for models uploaded in the last year with over 100K downloads. We find among these models, 71% have less than 3B parameters, so we filter out all models with greater than 3B parameters. Among the remaining models, we find 10.4% are CNNs (the remainder being transformer models). We choose a representative set of models shown in Table 1, and set sampling probabilities to each model to match the distribution of model sizes and types from the HF Model Hub.

For sampling job arrivals, we use public traces from Alibaba(28) collected on real GPU clusters. These traces provide arrival times, GPUs requested, service times, and quality-of-service for each job. We filter out jobs that have "latency-sensitive" quality-of-service, and we convert GPUs requested and service time to GPU-hours (by multiplying the two). We filter out jobs greater than 9 GPU-minutes for the physical cluster experiments (leaving 55% of all jobs) and 1 GPU-hour for the simulation experiments (leaving 81.6% of all jobs), and we bucket the remaining GPU-hours distribution according to the sampling probabilities of the models from Table 1 so that every job arrival in the trace is mapped to a specific model. For smaller models (<700M

parameters) we set the job to training or batch-inference with equal probability; for larger models we always set the job to batch-inference. To determine how many samples a job should process, we divide the job-size (in GPU-hours) by the max throughput the job-type can achieve when executed in isolation on one GPU. This yields a trace that contains job arrivals, job models, job category (training vs batch-inference), and job samples.

5 EVALUATION

We present the amount of GPU utilization recovered by PIPEFILL at different scales (Section 5.1); we then validate the accuracy of the simulator by comparing simulator results against physical cluster results (Section 5.1); we discuss how fill job characterization affects PIPEFILL's performance (Section 5.2); and we provide sensitive studies of pipeline schedule algorithm, fill-job scheduling policy, bubble duration and free memory (Section 5.3).

5.1 PIPEFILL Recovers GPU Utilization

Simulator Results To evaluate the GPU utilization recovered by PIPEFILL, we scale the 40B parameter LLM training job trace using DP up to 8K GPUs in our simulation. We measure the GPU utilization of filling inference jobs only, and filling both training and inference jobs. We use the GPU utilization without PIPEFILL as the baseline. To calculate the additional GPU FLOPS utilization recovered by PIPEFILL, we use the measured total FLOPs (floating-point operations) executed to complete the fill-jobs (from PyTorch profiling) and divide this by the simulated fill-job completion times (wall-clock time); we average this value across all GPUs across the duration of the main job.

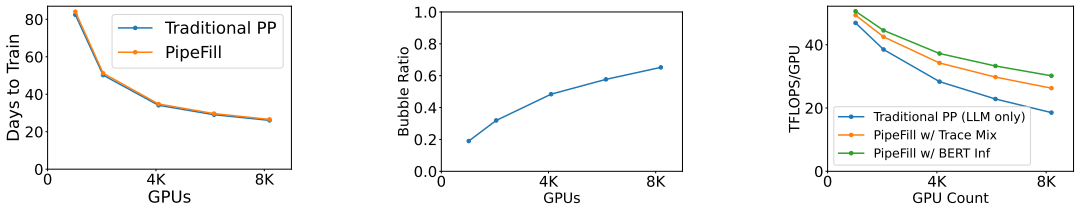
Figure 5 shows the results of main job training time, pipeline bubble ratio, and GPU utilization from using 1-8K GPUs. Even at low-scales (1K-2K GPUs), PIPEFILL improves GPU utilization by 5-10%. However, at higher scales PIPEFILL's potential is shown. Scaling the main job from 2K to 6K GPUs reduces training time from 50 days to 29 days; however, this results in a 40% drop in GPU utilization. With PIPEFILL, we are able to limit the drop in GPU utilization to $<$23%. At 4K GPUs (reducing main job training time by 16 days compared to 2K GPUs), PIPEFILL is able to get 89% of the GPU utilization of traditional PP at 2K GPUs; at 8K GPUs (reducing main job training time by 9 days compared to 4K GPUs), PIPEFILL is able to get 92% of the GPU utilization of traditional PP at 4K GPUs.

PIPEFILL's performance is even higher with a more bubble-friendly fill-job workload; in Figure 5 we also plot the GPU utilization recovered when filling with only BERT inference jobs. With this workload, PIPEFILL improves GPU utilization by 7.8-15.6% at low scales (1-2K GPUs). At 4K GPUs, PIPEFILL gets's 96.7% of the GPU utilization of traditional PP at 2K GPUs; and at 8K GPUs PIPEFILL exceeds the GPU utilization of traditional PP at 4K GPUs by 6.5%. These results show that PIPEFILL enables strong-scaling by an additional 2$\times$ with virtually no loss in GPU utilization, and at higher scales can even increase GPU utilization while strong-scaling. Additionally, due to the high bubble ratios and the relatively modest slowdowns experienced by the fill-jobs, the amount of GPUs worth of work being done by PIPEFILL using only the pipeline bubbles is notable. Generally, for a main job using C GPUs with a bubble ratio of B and fill-job relative performance of P, we can approximate the GPUs saved by filling as $C * B * P$. Depending on the workload, PIPEFILL can run 200-300 GPUs worth of fill-job when when the main job is using 2K GPUs, 600-900 GPUs worth of work when using 4K GPUs, and 1500-2600 GPUs when using 8K GPUs.

Physical cluster results. We confirm PIPEFILL's effectiveness and validate the fidelity of the simulator results by evaluating a subset of the settings on a small physical cluster with a 5B parameter LLM training job. We measure the free-memory quantity in the bubbles to be 4.5GB without main-job offloading; when we measured the free-memory of the larger training job, it was also 4.5GB, so we use this value in our simulator. We run the 5B parameter main job using 8 microbatches per minibatch per DP replica; this corresponds to using 64-way DP and results in a bubble ratio of 65%, which is comparable to the 8K GPU setting in Figure 5. We also use the full fill-job trace distribution for the physical cluster experiments, unless specified otherwise.

First, we evaluate whether the recovered GPU utilization and low overhead to the main job predicted by the simulator is observed in a physical environment. In Figure 7c, we vary the percentage of the bubble duration that PIPEFILL's Executor's attempt to fill. We find that the overhead to the main job is $<$2% for up to 68% of the bubble duration filled by the Executor; at higher fill percentages, the overhead to the main job can be substantial (though the total GPU FLOPS utilization continues to increase). We find the threshold of bubble filling without main job overhead to be around this value for all fill job types. Also at 68% we see that the TFLOPS/GPU recovered is around 7.39; this is within 5% of the TFLOPS predicted by the simulator at the same bubble ratio. This is because, in our simulator results, the Fill Job Executors fill the same percentage of the bubble duration by default.

Next, we evaluate whether the types of fill-job being run affect the main job overhead. In Figure 6, we take two very different job types from our trace: batch-inference with XLM (the largest model) and training with EfficientNet (the smallest model and the only CNN). We fix the percentage of the bubble duration filled by the Executor at 68%, and vary the fill-jobs from being all XLM to all EfficientNet; we find that the overhead to the main job does not vary significantly.



(a) Main Job Training v.s. # GPUs (b) Bubble Ratio v.s. # GPUs (c) GPU Utilization v.s. # GPUs

Figure 5. Simulator results of running a 40B LLM training job using 1-8K GPUs.

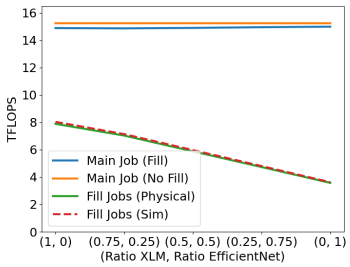


Figure 6. Simulator and physical cluster results of running a 5B LLM with varying distributions of fill job types.

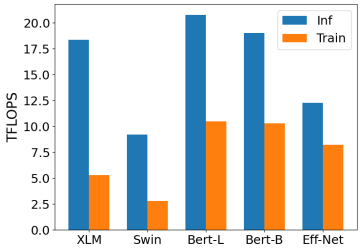
This shows that the overhead to the main job is independent of the types of fill-jobs being executed; instead it is only affected by the percentage of the bubble duration being filled. Figure 6 also plots the fill-job recovered-FLOPS predicted by the profile-based simulator and observed in physical execution—the maximum error of the simulator is $<2\%$.

5.2 Fill job characterization

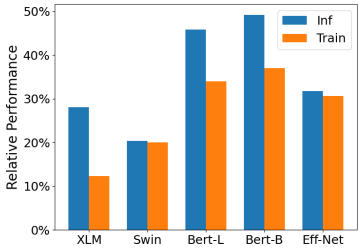
This subsection discusses how fill job characterization affects PIPEFILL's performance. This study helps understand the tradeoffs in which workloads are used as fill jobs. In the experiments, we evaluate training and inference of five different models as fill jobs. We measure the GPU FLOPS utilization they are able to achieve during their execution as fill-jobs and compare to the GPU FLOPS utilization they achieve when run in isolated resources. Here we divide the FLOPs (floating-point operations) executed to complete the fill-jobs by the total duration that they are executed (sum of all bubble durations used to complete the fill-job). This is in contrast to dividing by the wall-clock completion time, which we did in section 5.1 in order to understand the performance of the fill-jobs when they are executing (as opposed to the FLOPS utilization they can recover).

GPU FLOPS. Different fill jobs are able to utilize the GPU FLOPS to varying degrees; there are several reasons for this, some related to the jobs' fundamental characteristics and some related to the bubble constraints. In Figure 7a we plot the GPU FLOPS that each model and each job type (i.e., training vs. batch inference) is able to utilize on average during its execution; for comparison, the main job is able to utilize 60 TFLOPS when it is executing. Our first observation is that batch inference jobs are able to reach higher FLOPS utilization than training jobs; this is because inference jobs have low memory requirements and thus can use higher batch sizes under the free memory constraints of the bubbles than training jobs can. Among training jobs, large-model training jobs have particularly poor performance; this is because the much larger activation footprint of these models requires CPU-offloading of the activations. When comparing models, we see that Swin and EfficientNet perform particularly poorly. The Swin model is a non-uniform vision-transfomer model that uses a specialized attention operator; the memory-overhead of the larger layers limit the batch size, which further hurts the GPU utilization of the smaller layers, and the specialized attention operator is not well-optimized in our implementation. The EfficientNet model is small compared to the other models, but since it is a CNN it has particularly large activation sizes; the low free-memory in the bubbles limits the batch size that we can use, and since the model is small, the batch sizes that fit in the free-memory are not large enough to reach high GPU utilization.

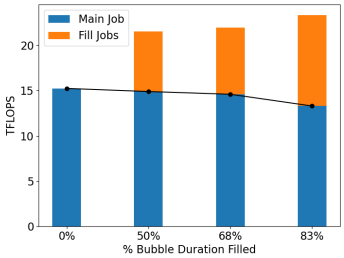
Fill job slowdown. TFLOPS recovered lets us compare the GPU utilization recovered across fill-job types, but we would also like to know the slowdown experienced by the fill jobs relative to their performance if they were run on exclusive GPUs. This analysis lets us approximate how many GPUs can be saved during the duration of the main job by filling its bubbles with certain fill-job types. In Figure 7b, we again see that the slowdown varies substantially across fill-job types. As expected, all fill-jobs experience substantial slowdown due to several factors that put fill-job execution at a disadvantage compared to exclusive execution: 1) the fill-jobs can only use a fraction of the GPU memory (about 25%) which can necessitate CPU-offloading and limit batch-sizes, 2) the fill-job execution is interrupted every time a bubble ends, introducing unavoidable inefficiencies in the Executor's plan, and 3) because the fill-job execution can only run for a short period of time, each bubble, it often can only run a single iteration of a subset of the model, which is not enough to warmup the GPU caches.



(a) GPU TFLOPS v.s. fill job types



(b) Job slowdown v.s. fill job types



(c) TFLOPS of 5B LLM on physical cluster, varying filled bubble durations.

Figure 7. Fill job characterization and physical cluster results

However, we see that these factors affect different fill-job types to varying degrees. In particular, we see that although XLM inference recovers similar TFLOPS as BERT inference, it experiences more slowdown; this is because XLM requires aggressive CPU-offloading, but because the model is large it can still submit enough computation work to keep the GPU busy. We hypothesize that on newer hardware-systems that have higher bandwidth between CPU and GPU memory (e.g., newer PCIe generations, NVLink-C2C), the fill-job slowdown from offloading could be substantially lower. Regardless, most of the fill-job workloads we evaluate experience around 30% of exclusive execution.

5.3 Sensitivity studies

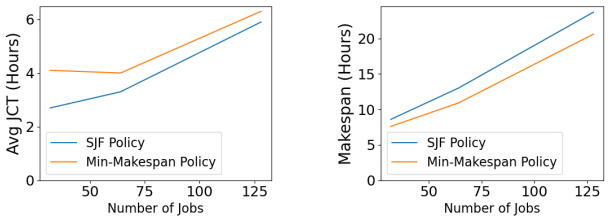
Main job pipeline schedule. We compare PIPEFILL with the main job using a GPipe schedule to using a 1F1B schedule, using the same main job as the simulator in section 5.1 and using the full fill-job trace. We vary the number of GPUs from 2K (18.9% bubble ratio) to 16K (78.9% bubble ratio). We find that the at smaller scales PIPEFILL recovers 20% more GPU utilization when the main job uses GPipe, but at larger scales the gap closes to 5%. This is because 1F1B contains some non-contiguous bubbles that are not within the fill-drain bubble or the fwd-bwd bubble, which PIPEFILL does not fill; at larger scales these non-contiguous bubbles become a smaller proportion of the total bubbles. We also simulate 1F1B-interleave schedule (20), and find it recovers 10-30% less TFLOPS than 1F1B; note that interleaving increases PP communication and can reduce main job efficiency if there is insufficient network bandwidth.

Fill-job scheduling policy. PIPEFILL allows the scheduling policy to be configured by the user; this section evaluates two possible policies. In Figures 8a and 8b, we implement a Shortest-Job-First policy and a Makespan-Minimizing policy. We see that the SJF policy is able to achieve lower average JCTs, especially at lower loads where completion time is not as dominated by queueing time. Conversely, the Makespan-Minimizing policy is able to reduce makespan, especially at higher loads where maximizing fill-job efficiency has a larger impact.

Bubble durations and free memory. Main job characteristics affect the pipeline bubble durations and free-memory; for example, a deeper pipeline or a wider main job model (with longer forward and backward times) can increase the bubble durations. Meanwhile, a larger main job model could also reduce the bubble free-memory. Here we analyze the effects of these factors on PIPEFILL's effectiveness.

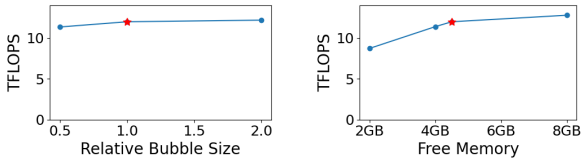
In Figure 9a, we scale the bubble size by equally scaling the main job model width and depth. We scale the original main job from section 5.1, from 50% to 200% of the original model size; we fix the free memory at 4.5GB. We see little difference in the recovered TFLOPS, though shrinking the bubble duration by 50% reduced TFLOPS by 5.3%.

In Figure 9b, we fix the main job model size (and thus the bubble duration) and vary the free-memory from 2GB to 8GB. We find free-memory to have larger impact on recovered TFLOPS, though with diminishing returns: 4GB recovers 30% more TFLOPS than 2GB, but 8GB only recovers 12.2% more TFLOPS than 4GB.



(a) Average JCT (b) Makespan

Figure 8. Sensitivity study of fill job schedule policy.



(a) TFLOPS v.s. bubble sizes (b) TFLOPS v.s. free mem

Figure 9. Sensitivity study of bubble size and free memory.

6 Related Work

Pipeline optimizations. There are many prior works on increasing pipeline-parallel efficiency. Chimera (15) proposes bidirectional pipelines to reduce pipeline bubbles at the cost of increasing the memory overhead on each device. In practice, it is not possible due to limited GPU memory for large LLM training jobs. Megatron-3D (20) proposes interleaved pipelines, which requires the number of microbatches to be a multiple of the number of pipeline stages. It has limited applicability, since minibatch sizes are fixed, as large-model training is scaled up using data parallelism the number of microbatches per stage decreases quickly to be less than the number of pipeline stages. Alpa (33), FlexFlow (11), Dapple (5) aim to search for optimal pipeline partition configuration for the training, which cannot eliminate bubbles. As discussed, Bamboo (25) and Pipefischer (21) fill pipeline bubbles with work that is dependant on the main job. Hydro (8) fills bubbles with hyperparameter-tuning jobs (in particular, trials which are not critical and can be early-terminated) for small models which requrie very little memory and can complete training iterations within a single bubble.

Resource sharing. Many prior works have identified and addressed the data-center GPU under-utilization issue by GPU-sharing. AntMan (29) provides the elasticity for DL training jobs to scale up and down for better efficiency. Salus (30) puts multiple DL jobs on the same device to improve the utilization. PipeSwitch (2) allows time-sharing of clusters for inference jobs with training jobs, when user demands of inference job is at valley. REEF (7) enables kernel-level preemption and concurrent execution for sharing GPUs with multiple inference jobs. PilotFish (31) exploits the spare resources on Cloud gaming platform for DL training. Muri (32) interleaves the usages of multiple hardware resources (e.g., network, GPU, etc.) among multiple DL jobs. These prior works do not address the pipeline bubbles of large model training like LLMs with tens-of-billions parameters.

7 Conclusion

PipeFill fills the pipeline bubbles of huge DNN training jobs with *other* jobs, significantly reducing the traditional GPU utilization penalty associated with extreme scale-out for such jobs. Experiments confirm that PipeFill can increase GPU utilization by up to 63% when LLM training is scaled-out, with $<$2% increase in LLM training time.

References

[1] Aws trainium. https://aws.amazon.com/machine-learning/trainium/, (accessed December 2023).

[2] 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 20)*, pages 499–514, 2020.

[3] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.

[4] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, Minneapolis, Minnesota, 2019. Association for Computational Linguistics.

[5] 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 '21, page 431–445, New York, NY, USA, 2021. Association for Computing Machinery.

[6] Naman Goyal, Jingfei Du, Myle Ott, Giri Anantharaman, and Alexis Conneau. Larger-scale transformers for multilingual masked language modeling. In *Proceedings of the 6th Workshop on Representation Learning for NLP (RepL4NLP-2021)*. Association for Computational Linguistics, 2021.

[7] 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 22)*, pages 539–558, 2022.

[8] 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.

[9] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Mia Xu Chen, Dehao Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V. Le, Yonghui Wu, and Zhifeng Chen. *GPipe: Efficient Training of Giant Neural Networks Using Pipeline Parallelism*. 2019.

[10] Hugging Face Inc. Hugging face model hub. https://huggingface.co/models, 2024. Accessed: 2024-04-20.

[11] Zhihao Jia, Matei Zaharia, and Alex Aiken. Beyond data and model parallelism for deep neural networks. *Proceedings of Machine Learning and Systems*, 1:1–13, 2019.

[12] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th annual international symposium on computer architecture*, pages 1–12, 2017.

[13] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015.

[14] Shen Li, Yanli Zhao, Rohan Varma, Omkar Salpekar, Pieter Noordhuis, Teng Li, Adam Paszke, Jeff Smith, Brian Vaughan, Pritam Damania, and Soumith Chintala. Pytorch distributed: Experiences on accelerating data parallel training. *CoRR*, abs/2006.15704, 2020.

[15] 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 '21, New York, NY, USA, 2021. Association for Computing Machinery.

[16] Ze Liu, Yutong Lin, Yue Cao, Han Hu, Yixuan Wei, Zheng Zhang, Stephen Lin, and Baining Guo. Swin transformer: Hierarchical vision transformer using shifted windows. In *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*, 2021.

[17] Sam McCandlish, Jared Kaplan, Dario Amodei, and OpenAI Dota Team. An empirical model of large-batch training. *arXiv preprint arXiv:1812.06162*, 2018.

[18] Microsoft. microsoft/deepspeed, October 2024.

[19] 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 '19. Association for Computing Machinery, 2019.

[20] Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, Amar Phanishayee, and Matei Zaharia. Efficient large-scale language model training on gpu clusters using megatron-lm. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '21. Association for Computing Machinery, 2021.

[21] Kazuki Osawa, Shigang Li, and Torsten Hoefler. Pipefisher: Efficient training of large language models using pipelining and fisher information matrices. *Proceedings of Machine Learning and Systems*, 5, 2023.

[22] Samyam Rajbhandari, Olatunji Ruwase, Jeff Rasley, Shaden Smith, and Yuxiong He. Zero-infinity: breaking the gpu memory wall for extreme scale deep learning. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '21. Association for Computing Machinery, 2021.

[23] Jie Ren, Samyam Rajbhandari, Reza Yazdani Aminabadi, Olatunji Ruwase, Shuangyan Yang, Minjia Zhang, Dong Li, and Yuxiong He. ZeRO-Offload: Democratizing Billion-Scale model training. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. USENIX Association, 2021.

[24] Mingxing Tan and Quoc Le. Efficientnet: Rethinking model scaling for convolutional neural networks. In *International conference on machine learning*, pages 6105–6114. PMLR, 2019.

[25] 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 23)*, pages 497–513, Boston, MA, April 2023. USENIX Association.

[26] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*, 2023.

[27] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti

Bhosale, et al. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*, 2023.

[28] Qizhen Weng, Lingyun Yang, Yinghao Yu, Wei Wang, Xiaochuan Tang, Guodong Yang, and Liping Zhang. Beware of fragmentation: Scheduling {GPU-Sharing} workloads with fragmentation gradient descent. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*, pages 995–1008, 2023.

[29] 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 *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation*, OSDI'20. USENIX Association, 2020.

[30] Peifeng Yu and Mosharaf Chowdhury. Salus: Fine-grained gpu sharing primitives for deep learning applications. *arXiv preprint arXiv:1902.04610*, 2019.

[31] 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 *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 217–232, 2022.

[32] Yihao Zhao, Yuanqiang Liu, Yanghua Peng, Yibo Zhu, Xuanzhe Liu, and Xin Jin. Multi-resource interleaving for deep learning training. In *Proceedings of the ACM SIGCOMM 2022 Conference*, SIGCOMM '22. Association for Computing Machinery, 2022.

[33] 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 22)*. USENIX Association, 2022.