Adding NVMe SSDs to Enable and Accelerate 100B Model Fine-tuning on a Single GPU

Changyue Liao*, Mo Sun*,
Zihan Yang*, Kaiqi Chen
Zhejiang University, China

Binhang Yuan
HKUST, China

Fei Wu, Zeke Wang
Zhejiang University, China

Abstract

Recent advances in large language models have brought immense value to the world, with their superior capabilities stemming from the massive number of parameters they utilize. However, even the GPUs with the highest memory capacities, currently peaking at 80GB, are far from sufficient to accommodate these vast parameters and their associated optimizer states when conducting stochastic gradient descent-based optimization. One approach to hosting such huge models is to aggregate device memory from many GPUs. In particular, it takes 32 NVIDIA A100 GPUs to fit a model with 100 billion parameters for fine-tuning. However, this approach introduces prohibitive costs for most academic researchers, who always have a limited budget for many high-end GPU servers. In this paper, we focus on huge model fine-tuning on a single, even low-end, GPU in a commodity server, which is accessible to most AI researchers. In such a scenario, the state-of-the-art work ZeRO-Infinity suffers from two severe issues when running in a commodity server: 1) low GPU utilization due to inefficient swapping, and 2) limited trainable model size due to CPU memory capacity. The underlying reason is that ZeRO-Infinity is optimized for running on high-end GPU servers.

To this end, we present *Fuyou*, a low-cost training framework that enables efficient 100B huge model fine-tuning on a low-end server with a low-end GPU and limited CPU memory capacity. The key idea is to add the SSD-CPU communication as an optimization dimension and thus carefully co-optimize computation and data swapping from a systematic approach to maximize GPU utilization. To do so, Fuyou consists of three innovations. First, we propose a synchronous out-of-core CPU optimizer that overlaps with backward propagation to maximize the GPU utilization. Second, we propose a GPU-CPU-SSD fully-pipelined activation swapping mechanism to allow for a significantly larger model fine-tuning. Third, we present an automatic activation swapping management to automatically determine the optimal amount of swapping activations so as to minimize the epoch time. The experimental results show that 1) Fuyou is able to fine-tune 175B GPT-3 on a consumer GPU RTX 4090 with high GPU utilization, while ZeRO-Infinity fails to fine-tune; and 2) when training a small GPT-3 13B model, Fuyou achieves 156 TFLOPS on an RTX 4090 GPU while ZeRO-Infinity only achieves 45 TFLOPS.

1 Introduction

Large language models (LLMs) have drawn the world's attention due to their impressive accuracy in various natural language processing jobs [5, 8, 39, 46, 58] including various data management tasks [12, 54]. Along with advances in Transformer models are their fast-growing model sizes; in recent years, the model size of dense transformer models has grown from 1.5B (GPT-2 [39]) to 540B (PaLM [7]). However, even the GPUs with the highest memory capacities, currently peaking at 80GB, are far from sufficient to accommodate these vast parameters and their associated optimizer states when conducting stochastic gradient descent-based optimization. One approach to hosting such huge models is to aggregate device memory from many GPUs [24, 25, 47, 59]. For example, it takes 32 NVIDIA A100 GPUs to fit a model with 100 billion parameters for training. However, training such huge models from scratch requires millions of GPU hours and thus introduces prohibitive costs for most academic researchers, who always have a limited budget for many high-end GPU servers. Fortunately, a pre-trained model could be used in various downstream AI tasks via fine-tuning [34, 53]. In this paper, we focus on huge model fine-tuning on a single, even low-end, GPU in a commodity server, which is accessible to most AI researchers.

Existing methods [29, 37, 40, 45, 55, 60] exploit heterogeneous storage to train an LLM, the state-of-the-art method ZeRO-Infinity [41] utilizes GPU, CPU, and NVMe memory to fine-tune huge models on high-end GPU servers. In particular, ZeRO-Infinity offloads parameters, gradients, and optimizer states from GPU memory to CPU memory and even to NVMe storage, and offloads activations to host memory if necessary, thereby enabling the fine-tuning of huge models under limited GPU memory. ZeRO-Infinity performs weight updates on the CPU so as to reduce the massive data transfer of optimizer states. Even though existing works allow a huge model fine-tuning on a high-end GPU server, they still suffer from two severe issues when fine-tuning on a consumer GPU RTX 4090 of a commodity server.[1]

- **Limited Maximum Trainable Model Size.** ZeRO-Infinity fails to fine-tune a 65B model when the host memory capacity is smaller than 512 GB.

---

*Equal contribution.
[1]Two issues remain on a A100-80G GPU.

- **Low GPU Utilization.** Even with a sufficient amount of host memory, ZeRO-Infinity achieves only 26% GPU utilization when fine-tuning a 65B model.

The underlying reason is that the existing works are originally designed for many high-end GPU servers such as DGX-2 with high-end GPUs and huge CPU memory, rather than for a commodity server. Furthermore, fine-tuning on many high-end servers does not really need to offload activations and optimizer states to SSDs. In a nutshell, we **first** identify two unique technical issues that prevent the existing offloading works such as ZeRO-Infinity from achieving high performance when fine-tuning a huge model on a low-end server with a low-end GPU and limited CPU memory capacity.

1, Serializing Synchronous Out-of-core Optimizer and Backward Propagation. The existing works such as ZeRO-Infinity rely on CPUs to implement synchronous out-of-core optimizer whose states are materialized into SSDs, such that ZeRO-Infinity is able to fine-tune a larger model. However, these works do not overlap the out-of-core optimizer with backward propagation to preserve model synchronization. As such, ZeRO-Infinity needs a significant amount of time to update optimizer states. For example, the CPU optimizer consumes up to 70% of the total training time.[2]

2, Activations Only Offloaded to CPU Memory, not Further to SSDs. The existing works such as ZeRO-Infinity are designed to run on many high-end servers, and thus only offload activations to host memory, not further to SSDs, because these high-end servers have the sufficient large aggregated memory capacity to accommodate activations. However, such an offloading mechanism incurs high pressure on host memory capacity in a commodity server, because the host memory is also shared by other offloaded objects such as optimizer. Therefore, the existing works do not consider offload activations to SSDs so as to allow a larger model fine-tuning.

It becomes common wisdom that offloading optimizer states or activations to SSDs when fine-tuning a 100B model yields a significantly lower GPU utilization [41, 49]. In this paper, we ask:

Can we fine-tune a 100B model with a low-end GPU in a commodity server while keeping high GPU utilization?

To this end, we present Fuyou, a low-cost training framework that enables efficient 100B huge model fine-tuning on a low-end server with a low-end GPU and limited CPU memory capacity. The key idea is to add SSDs as an optimization dimension for efficient activation swapping and synchronous out-of-core optimizer that overlaps with backward propagation. In particular, Fuyou consists of three innovations.

---
[2]The asynchronous approach such as Angel-PTM [29] presents an out-of-core optimizer which is overlapped with backward propagation, however, it adopts an asynchronous optimizer updating policy that could affect model training convergence. Therefore, they are beyond the scope of this paper.

- **Synchronous Out-of-core CPU Optimizer Overlapped with Backward Propagation.** In order to maximize GPU utilization when fine-tuning on a single GPU, we propose a synchronous out-of-core CPU optimizer that overlaps with backward propagation so as to remove the optimizer stage, where the CPU updates the optimizer states and the GPU is entirely idle. At the same time, Fuyou does not compromise any training convergence rate due to its synchronous model update.
- **GPU-CPU-SSD Fully-Pipelined Activation Swapping.** In order to maximize the trainable model size, we propose a GPU-CPU-SSD fully-pipelined activation swapping technology that enables efficient data swapping between GPU memory, CPU memory, and NVMe SSDs to enable a commodity server to fine-tune a huge model, whose size is limited by SSD capacity, rather than CPU/GPU memory size.
- **Automatic Activation Swapping Management.** The existing swapping and recomputation works such as Capuchin [37] only consider GPU PCIe traffic and activation recomputation overhead to determine the amount of swapping activations such that the PCIe communication time is roughly equal to the backward propagation time because these works do not overlap optimizer and backward propagation. However, Fuyou overlaps the synchronous out-of-core CPU optimizer with backward propagation and thus poses a new challenge to Fuyou on how to determine the exact amount of swapping activations given that 1) the maximal time of backward propagation time and optimizer time could be used to swap activations, and 2) activation swapping and out-of-core CPU optimizer compete for precious SSD bandwidth and GPU PCIe bandwidth. To this end, we propose an automatic activation swapping management mechanism to automatically determine the amount of swapping activations such that the epoch time is minimized when training on a single GPU in a commodity server. The key contribution of automatic activation swapping management is to build a cost model to roughly predict the epoch time given a certain amount of swapping activations. Given the cost model, Fuyou considers all the possible amounts of swapping activations, estimates their corresponding epoch times, and finally chooses the smallest estimation cost.

We implement Fuyou on the popular deep learning framework PyTorch [35]. We evaluate Fuyou on either NVIDIA A100-80GB [32] or RTX 4090 [33] in a commodity server. When fine-tuning a GPT-3 175B model, Fuyou achieves 87 TFLOPS (53% of peak FLOPs[3]) on 4090 and 172 TFLOPS on A100-80GB (86% of peak FLOPs), while ZeRO-Infinity and Colossal-AI fail to fine-tune. When fine-tuning a GPT-3 13B model on RTX 4090, Fuyou reaches up to 3.47× TFLOPS compared to ZeRO-Infinity.

---
[3]Peak FLOPs is measured by benchmarking a transformer block, which might be different from theoretical FLOPs.

Table 1. Comparison of Fuyou with different solutions for large-scale model fine-tuning.

	Activation Offloading		Optimizer Stage		
	to CPU memory	to SSD	Out-of-core	Overlapped w/ Backward Stage	Synchronous
vDNN++ [48]	✓	✗	✗	✗	✓
SwapAdvisor [15]	✓	✗	✗	✗	✓
Beaumont et al. [4]	✓	✗	✗	✗	✓
STR [61]	✓	✗	✗	✗	✓
Capuchin [37]	✓	✗	✗	✗	✓
SuperNeurons [55]	✓	✗	✗	✗	✓
DeFiNES [23]	✓	✗	✗	✗	✓
L2L [38]	✓	✗	✗	✗	✓
ZeRO-Offload [44]	✓	✗	✗	✓	✗
ZeRO-Infinity [41]	✓	✗	✓	✗	✓
STRONGHOLD [49]	✓	✗	✗	✓	✓
Angel-PTM [29]	✓	✗	✓	✓	✗
Fuyou	✓	✓	✓	✓	✓

2 Background

2.1 Deep Learning Training

Training Stages. A deep learning model consists of multiple layers of mathematical functions. To get the model converged, the training procedure takes multiple training iterations. Each iteration consists of three stages:

- 1) Forward stage, where the model takes training data as input and computes the error values. Each layer gets the output activations of the previous layer and delivers the output activations to the next layer.
- 2) Backward stage, where the error values are propagated from the last layer to the first layer to compute the gradients. Each layer gets the error values from the next layer, computes the gradient of each parameter according to the error values and input activations, and delivers the output error values to the previous layer.
- 3) Optimizer stage, where the parameters are updated according to gradients. For LLMs, to increase the model convergency, Adam optimizer [19] is generally adopted. In Adam optimizer, auxiliary optimizer states are introduced to smooth the parameter update process.

Memory Footprint. In deep learning training, memory usage mainly consists of two components: 1) Model states, including parameters, gradients, and optimizer states. Gradients are produced in the backward stage and consumed in the optimizer stage, while parameters and optimizer states are kept throughout the training process. The size of model states is only proportional to the model size. 2) Intermediate values, namely activations. Activations are produced in the forward stage and consumed in the backward stage. The size of activations is decided by model size, batch size, and sequence length.

Activation Checkpointing. Activation checkpointing is a mechanism to reduce the memory footprint in deep learning training. When activation checkpointing is applied, during the forward stage, only a subset of activations is saved, namely checkpoints, while others are discarded. During the backward stage, when performing the backward propagation of a layer whose input activations are discarded, extra forward propagation from the last checkpoint is performed to get the discarded activation. The extra forward propagation is called recomputation.

Activation Swapping. Activation swapping is another mechanism for memory saving. Since the activations are produced in the forward stage and consumed in the backward stage, when activation swapping is applied, during the forward stage, activations are swapped out from GPU memory after being produced, and during the backward stage, they are swapped into GPU memory before being consumed. Activation swapping can be combined with the checkpointing mechanism, where activations are either swapped out or discarded after being produced during the forward stage. In this case, activation swapping trades off communication volume for recomputation overhead.

2.2 Optimizations of ZeRO-Offload and ZeRO-Infinity

ZeRO-Infinity [41] is the state-of-the-art training method utilizing heterogeneous storage to train large models. It's integrated into DeepSpeed [42], an optimized deep learning library specifically for large-scale models. Besides ZeRO-Infinity, DeepSpeed also integrates ZeRO-Offload [44], an optimization method that offloads model states to the CPU memory. In this subsection, we will introduce the optimizations of these two methods.

Memory Management Optimizations. To enable larger model size with limited GPU memory, ZeRO-Offload offloads model states to CPU memory, while ZeRO-Infinity further offloads model states to NVMe SSDs. For activations, both ZeRO-Offload and ZeRO-Infinity adopt activation checkpointing and activation swapping to reduce the GPU memory footprint of activations. The two methods only checkpoint activations between transformer blocks, while users
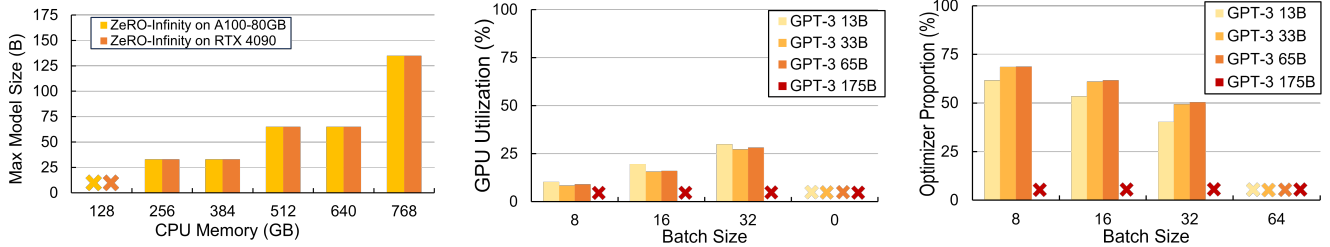
(a) Largest model under different CPU memory limits. **(b)** GPU utilization under different batch sizes. **(c)** Proportions of optimizer stage in a training step.

Figure 1. The two issues of ZeRO-Infinity motivates the design of Fuyou. We perform the experiments on A100-80GB GPU.

can set how many transformer blocks between checkpoints. ZeRO-Offload retains checkpoints in GPU memory, while ZeRO-Infinity further offloads checkpoints to CPU memory. Checkpoints are not offloaded to SSDs in both two methods. **Optimizer Stage Optimizations.** In ZeRO-Offload and ZeRO-Infinity, the forward and backward stages are executed on GPU while the optimizer stage is executed on CPU. ZeRO-Offload and ZeRO-Infinity are originally based on ZeRO distributed training strategy [40], and thus shard optimizer states across multiple nodes, with each node only updating a portion of the model parameters and obtaining updated parameters from other nodes through collective communication. Therefore, each node only performs part of the parameter update during the optimizer stage, reducing the computational pressure on the CPU. Furthermore, to further hide computational overhead on the CPU, ZeRO-Infinity claims to provide a "one-step delayed parameter update" mechanism, which overlaps the optimizer stage with the forward and backward stages of the next iteration. However, with this mechanism, parameter update is asynchronous with forward and backward stages, which affects the model convergence and, thus is not preferred by most data scientists. Moreover, the open-source implementation of the DeepSpeed library doesn't provide the delayed parameter update function for ZeRO-Offload.

3 Motivation

ZeRO-Infinity is originally designed for high-end DGX-2 [31] servers, rather than for a commodity server with a single GPU. Therefore, ZeRO-Infinity works badly in a commodity server with only one GPU. In the following, we identify two concrete severe issues that prevent ZeRO-Infinity from allowing efficient fine-tuning of a huge model on a commodity server with a single GPU as below:

3.1 Supporting Limited Trainable Model Size under Limited CPU Memory Capacity

ZeRO-Infinity fails to fine-tune a 175B model on a commodity server with limited CPU memory capacity. To quantitatively validate the effect of CPU memory capacity on ZeRO-Infinity, we intend to fine-tune GPT-3 [5] models of different sizes

on our server, whose detailed configurations are shown in Subsection 5.1. The batch size is 1 to minimize its effect.

Figure 1a illustrates the maximum trainable model size of ZeRO-Infinity under different CPU memory sizes, where the storage space is 48TB, far beyond the sufficient storage space to accommodate the whole training. We observe that the maximum trainable model that ZeRO-Infinity can fine-tune is highly constrained by the CPU memory capacity. For example, ZeRO-Infinity can only fine-tune a 65B model with 512GB CPU memory. The underlying reason is that ZeRO-Infinity can only offload activations to CPU memory, rather than further to NVMe SSDs. Such offloading causes high pressure on CPU memory, which is shared by other intermediate objects.

3.2 Low GPU Utilization when Fine-tuning a Small Model on a Single GPU

We quantitatively analyze the GPU utilization when fine-tuning a small model on a single GPU A100-80G.[4] Figure 1b illustrates the ratio of GPU busy time over the total elapsed time within one iteration when varying the batch size. We observe that the GPU utilization is only 28% even when the batch size used is relatively large (such as 32). The underlying main reason is two-fold:

- **Heavy Weight Update Overhead.** To accommodate larger models with limited GPU memory capacity, ZeRO-Infinity stores FP32 optimizer states on SSDs and performs the weight updates on the CPU. However, ZeRO-Infinity updates weights and optimizer states once after a forward propagation stage and a backward propagation stage, indicating that the CPU optimizer stage does not overlap with forward and backward propagation, where GPU computation occurs. Throughout the weight update stage, the GPU is idle, with no communication or computation tasks being executed on the GPU. In distributed training, ZeRO-Infinity evenly distributes optimizer states across all machines. By aggregating memory bandwidth and SSD-to-CPU bandwidth from many nodes, the CPU Adam [19]

---

[4]We choose A100-80G, rather than 4090, because 4090 has higher compute power while A100-80G has higher IO bandwidth such as memory, and thus A100 has less opportunity to be bounded by IO.
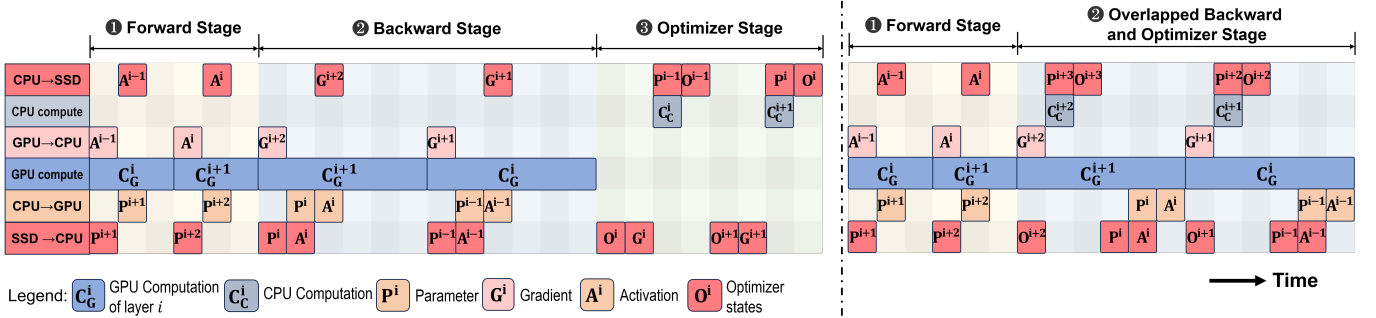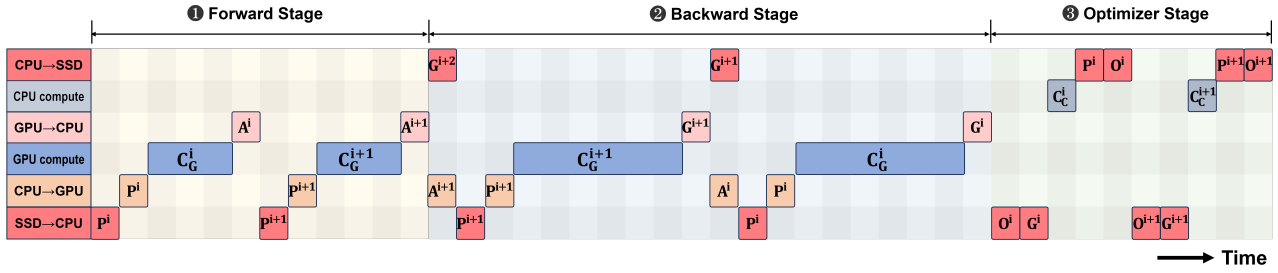
(a) ZeRO-Infinity: 1) serializing optimizer and backward propagation, and 2) almost serializing GPU compute and PCIe communication.

(b) Fuyou w/o overlapping: 1) overlapping GPU compute and SSD-CPU-GPU communication.

(c) Fuyou = Fuyou w/o overlapping + 2) overlapping optimizer and backward propagation.

Figure 2. Comparison of Fuyou and ZeRO-Infinity.

contributes a trivial time proportion of each iteration. However, when training with only one GPU within a server, updating the complete set of optimizer parameters can be a highly time-consuming task. Figure 1c shows the time proportion of the CPU optimizer stage to the training step in ZeRO-Infinity. We observe that the optimizer stage takes 40%~70% of the training step, during which the GPU is completely idle. This significantly affects the achievable GPU utilization rate.

- **Almost Serial Execution of Computation and Communication during Forward and Backward Propagation.** Figure 2a illustrates the concrete dataflow of ZeRO-Infinity that trains a 13B model on an A100-80G GPU with a batch size of 32. From the NVIDIA Nsight™ Systems [30], we observe that during forward and backward propagation, the GPU kernel does not overlap with CPU-GPU and CPU-SSD communications too much, because ZeRO-Infinity does not optimize the communication-computation overlapping when gradients and parameters are offloaded to SSDs. For example, P^i (SSD to CPU), P^i (CPU to GPU), C_G^i (GPU), and A^i (GPU to CPU) are serialized during the forward propagation.

4 Design of Fuyou

4.1 Design Overview

To address these issues of ZeRO-Infinity, we present Fuyou, a low-cost training framework that enables efficient 100B
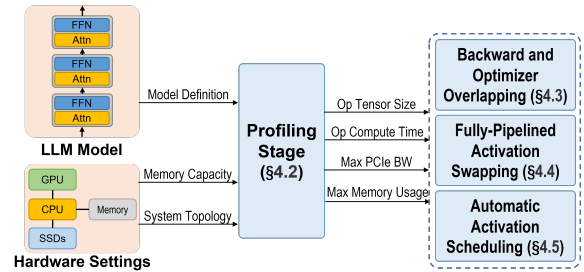

Figure 3. Fuyou Overview.

huge model fine-tuning on a low-end server with a low-end GPU. The key idea is to add the SSD-CPU communication as an optimization dimension for pipelining and thus carefully co-optimize computation and data swapping from a systematic approach to maximize GPU utilization and the model size that Fuyou can fine-tune. Fuyou consists of four main components: 1) A profiling stage that collects essential data for Fuyou's automatic swapping management (Subsection 4.2), 2) Synchronous out-of-core CPU optimizer overlapped with backward propagation which avoids GPU being idle during optimizer stage while not compromising training convergence rate (Subsection 4.3), 3) A fully pipelined activation swapping mechanism that enables fully-pipelined GPU-CPU-SSD two-level activation swapping which enables fine-tuning a larger model size (Subsection 4.4), and 4) An automatic activation scheduling strategy that automatically determines the amount of swapping activations to further

minimize epoch time (Subsection 4.5). Figure 3 illustrates the overall structure of Fuyou.

4.2 Profiling Stage

In the profiling stage, Fuyou gathers essential data from both model and hardware settings for further optimizations.

Profiling Setting. In this stage, Fuyou offloads all activations and model states to NVMe SSDs without enabling any optimizations, thus all computations and communications are executed in serial. As such, we can get roughly accurate predictions on the computation/communication cost of each layer.

Profiling Goals. In this stage, Fuyou will produce the following information. First, it takes the PyTorch model definition during initialization. During runtime, it parses each operator via PyTorch hooks and then gets the sizes of activations and parameters for each operator. Besides, it records the computation time of each operator during forward computation. Second, it gets the system topology and memory capacity from hardware settings during initialization, monitors the maximum PCIe bandwidth of each PCIe link, and maximum CPU memory and GPU memory usage.

4.3 Backward Propagation and Optimizer Overlapping

In order to maximize GPU utilization, we propose a synchronous out-of-core CPU optimizer that overlaps with backward propagation. Our optimizer is based on ZeRO-Infinity's synchronous out-of-core CPU optimizer. In this subsection, we will explain the opportunity and our concrete design to make the optimizer overlap with the backward stage.

Overlapping Opportunity. The model training process in Fuyou on one GPU involves the following computation and communication resources: GPU computation (R1), CPU computation (R2), CPU-to-GPU communication (R3), GPU-to-CPU communication (R4), and SSD I/O (R5). Note that SSD I/O is simplex so only one direction can be utilized at the same time. During backward propagation, R1, R3, R4, R5 are utilized while R2 is spare. During optimizer stage, R2, R5 are utilized while R1, R3, R4 are spare. Except for SSD I/O, these two stages utilize completely different resources. This leaves the potential for overlapping the two stages to accelerate the training process.

Moreover, overlapping backward and optimizer stages can reduce the overall SSD I/O as well. Without overlapping, during backward propagation, when GPU computes gradients, they need to be temporarily stored in SSDs until being used for the optimizer stage. When the two stages are overlapped, gradients produced in backward propagation can be directly consumed by the optimizer process, without having to be stored in SSDs. Therefore, overlapping backward and optimizer stages is beneficial for all cases. When SSD I/O is the system bottleneck throughout the two stages, which occurs when the batch size and the number of SSDs are both small,

overlapping the two stages saves SSD I/O for gradients, thus reducing the overall training time. When SSD I/O is not the bottleneck, the two stages have no conflict in computation and communication resources, so overlapping the two stages can naturally reduce the overall training time.

Concrete Design. Figure 2c illustrates an example of overlapping the two stages. At initialization, the main training process launches a CPU subprocess for optimizer computation. The two processes are completely decoupled aside from necessary synchronizations. Synchronizations are done via CUDA events provided by PyTorch. When performing computation tasks on the GPU, the corresponding operator's optimizer state is asynchronously prefetched to the CPU. After the gradient computation is completed on the GPU and offloaded to the CPU memory, the CPU asynchronously performs the Adam computation, while the GPU continues to execute the computation for the next operator. In this example, the execution time for the overlapped backward-optimizer stage is not significantly increased compared to the individual backward stage.

Fuyou also tends to improve the parallelism within the optimizer process. In Fuyou, weight updates are performed in parameter groups. In a serialized setting, the whole workflow is divided into three steps: 1) Reading optimizer states of group i from the SSDs, 2) Updating optimizer states of group i, 3) Writing the updated data of group i back to the SSDs. In this case, CPU computation and SSD I/O are serialized. In Fuyou, we adopt a delayed write-back strategy, i.e., the write-back of group i is performed after the update of group $i-1$ is completed. By doing so, step 2 can be overlapped with steps 1 and 3, thereby better utilizing CPU computation and SSD I/O resources.

4.4 Fully Pipelined Activation Swapping

Figure 2b illustrates an example of Fuyou's pipelined execution strategy. During forward and backward propagation, Fuyou intends to overlap GPU computation and PCIe communication (SSD-CPU and CPU-GPU) to its best. During the optimizer stage, Fuyou overlaps CPU computation and SSD accesses as well. As such, this strategy ensures maximum GPU utilization during forward and backward propagation, therefore solving ZeRO-Infinity's serial execution issue.

The design of the deeply pipelined pipeline strategy is not trivial. The main challenge is to determine when to prefetch data and how much data to prefetch. Insufficient prefetching results in the serialization of communication and computation, while excessive prefetching introduces unnecessary pressure on GPU memory, thus limiting the trainable model size.

To efficiently implement the execution strategy without compromising the trainable model size, we propose a GPU-memory-aware FIFO prefetching mechanism. With the peak GPU memory utilization acquired in the profiling stage Fuyou

allocates the rest of the GPU memory spaces for the prefetching parameters and activations. Therefore, Fuyou creates a FIFO buffer for storing parameters, activations, and gradients which could be used for the pipelined communication. Whenever the FIFO buffer is empty, Fuyou prefetches activations and parameters of the next layers so as to maximize GPU utilization.

This design solves the two problems. First, it determines when to prefetch since data required by the current module can simply be retrieved from the prefetch queue. Second, it resolves the issue of the prefetch data volume, as we maximize prefetching within the constraints of available GPU memory. Initially, we determine the size of the data prefetching queue within the GPU. Subsequently, based on the ratio of the GPU-CPU bandwidth to the SSD-CPU bandwidth, we ascertain the size of the data prefetching queue in the CPU.

Moreover, to make efficient use of CPU storage resources and make the system more flexible, Fuyou dynamically decides the offloading location of activation. When the CPU memory resources are sufficient to store activations, activations are stored in the CPU memory instead of offloaded to SSDs to reduce the SSD I/O pressure.

4.5 Automatic Activation Scheduling

We utilize activation checkpointing to reduce memory usage and further offload activation checkpoints to SSDs to free up storage space on GPUs and CPUs. Since activation recomputation brings overhead in GPU computation, to minimize the recomputation overhead, we propose an automatic activation swapping management mechanism, which automatically determines the amount of swapping activations.

Notations. Notations in this subsection are listed below. N_{SSD} is the number of SSDs used, h is the hidden dimensions of the model, l is the number of layers, b is the batch size, s is the sequence length, and p is the total parameter count. These values are decided by training settings. Besides, BW_{GPU} is the PCIe bandwidth between GPU and CPU, T_f is the execution time of the forward stage, T_f^{comp} is the GPU compute time during the forward stage, T_o^{comp} is the CPU compute time for the optimizer, BW_{S2C} is the bandwidth from a single SSD to CPU, and BW_{C2S} is the bandwidth from CPU to a single SSD, $Tput_f$ is the GPU throughput in FLOPS during the forward stage. These values are acquired during the profiling stage. D_f denotes the communication volume of activation checkpoints from GPU to SSD during the forward propagation stage, while D_{b+o} denotes the checkpoint communication volume from SSD to GPU during the overlapped backward-optimizer stage. Since D_f and D_{b+o} are equivalent, in the following text we only discuss D_f.

How many activations need to be swapped? Our optimization goal is to select an appropriate D_f to minimize the total time of the entire training phase T_{iter}, which can be

expressed in Equation 1.

$$T_{\text{iter}} = T_f + T_{b+o} \tag{1}$$

For the forward stage, the execution time is the maximum among the actual time for forward computation on the GPU T_f^{com}, the data communication time between GPU and CPU T_f^{GPU}, and the data communication time between SSD and CPU T_f^{SSD}. This can be expressed by Equation 2.

$$T_f = \max\left(T_f^{\text{comp}}, T_f^{\text{GPU}}, T_f^{\text{SSD}}\right) \tag{2}$$

Here, T_f and T_f^{comp} is measured during the profile stage. Communication times T_f^{GPU} and T_f^{SSD} can be estimated by communication volume divided by bandwidth. Here, the data size of fp16 parameters in the SSD-CPU-GPU path is $2p$. Communication in GPU is duplex, thus the communication time between GPU and CPU T_f^{GPU} is the maximum in two directions, which can be estimated by Equation 3. Communication in SSD is simplex, thus the communication time between SSD and CPU T_f^{SSD} is the sum of two directions, which can be estimated by Equation 4.

$$T_f^{\text{GPU}} = \max\left(\frac{2p}{BW_{\text{GPU}}}, \frac{D_f}{BW_{\text{GPU}}}\right) \tag{3}$$

$$T_f^{\text{SSD}} = \frac{2p}{BW_{\text{S2C}}N_{\text{SSD}}} + \frac{D_f}{BW_{\text{C2S}}N_{\text{SSD}}} \tag{4}$$

For the overlapped backward-optimizer stage, the execution time is the maximum among the computation time on the GPU T_b^{comp}, the optimizer execution time on the CPU T_o^{comp}, the data communication time between GPU and CPU T_{b+o}^{GPU}, and the data communication time between SSD and CPU T_{b+o}^{SSD}, which can be expressed by Equation 5.

$$T_{b+o} = \max\left(T_b^{\text{comp}}, T_o^{\text{comp}}, T_{b+o}^{\text{GPU}}, T_{b+o}^{\text{SSD}}\right) \tag{5}$$

Here, T_o^{comp} can be measured during the profile stage. Similar to the forward stage, the communication times T_{b+o}^{GPU} and T_{b+o}^{SSD} can be estimated by the communication volume divided by bandwidth. During the overlapped backward and optimizer stage, fp16 parameters are transferred in the SSD-CPU-GPU path, fp16 gradients are transferred from GPU to CPU, fp32 model states are read from SSD to CPU, while the updated fp32 model states and fp16 parameters are written from CPU to SSD. Therefore, the communication times can be estimated by Equation 6 and 7.

$$T_{b+o}^{\text{GPU}} = \max\left(\frac{2p}{BW_{\text{GPU}}}, \frac{2p + D_f}{BW_{\text{GPU}}}\right) \tag{6}$$

$$T_{b+o}^{\text{SSD}} = \frac{12p + 2p + D_f}{BW_{\text{S2C}}N_{\text{SSD}}} + \frac{12p + 2p}{BW_{\text{C2S}}N_{\text{SSD}}} \tag{7}$$

As for GPU computation time for backward stage T_b^{comp}, it equals the time for backward propagation plus the time for recomputation. The backward propagation time can be estimated as two times the forward time $2 \times T_{\text{fw}}^{\text{com}}$. Let $RC(D_f)$

be the time for recomputation. Since more activations are swapped, less time is required for recomputation, $RC(D_f)$ is a decreasing function of D_f. Therefore, T_b^{comp} can be estimated as Equation 8.

$$T_b^{comp} = 2 \times T_f^{comp} + RC(D_f) \qquad (8)$$

From the above analysis, T_f^{comp} and T_o^{comp} are independent of D_f. T_f^{comp} is related to the model size and batch size, while T_o^{comp} is only related to the model size. For T_f^{GPU}, T_f^{SSD}, T_{b+o}^{GPU}, and T_{b+o}^{SSD}, increasing D_f increases execution time. Besides, increasing D_f will decrease the execution time of T_b^{comp}.

On the other hand, the amount of activation checkpoint data is constrained by GPU memory capacity. Too few checkpoints can lead to an excessive number of temporary intermediate variables generated during backward propagation, risking memory overflow. To implement an adaptive swap scheduling strategy while avoiding memory overflow, we set the initial value of D_f to user-determined D_{start} during the profile stage. By default, D_{start} is set to apply one activation checkpoint for each transformer block, which is the strategy adopted by ZeRO-Infinity. This initial strategy doesn't lead to a significant communication overhead, since the total parameter size for a transformer block is $12 \times h \times h$ bytes while saving the activation for each transformer block only requires $b \times s \times h$ bytes GPU space. For large LLMs, h is often large, thus the activation size is small compared to the parameter size.

After initializing D_f, the automatic scheduling engine adaptively iterates D_f for each training iteration. We attempt to reduce the overall training time by increasing D_f, as decreasing D_f from its initial value carries the risk of memory overflow. However, we can only reduce the overall training time by swapping more activations when GPU backward propagation is the bottleneck for the overlapped backward and optimizer stage, i.e., $T_{b+o} = T_b^{comp}$. This usually occurs in scenarios with larger batch sizes. In other cases, increasing the swap of activations leads to an increase in overall training time. Besides, the upper bound of our overall training time benefit T_{MAX} can be computed by Equation 9.

$$T_{max} = T_b^{comp} - \max(T_{b+o}^{GPU}, T_{b+o}^{SSD}) \qquad (9)$$

Therefore, the upper bound for D_f can be defined as Equation 10 shows.

$$D_{max} = T_{max} \times \min(BW_{GPU}, BW_{C2S}N_{SSD}, BW_{S2C}N_{SSD}) \quad (10)$$

Since increasing D_f will cause both T_{b+o}^{GC} and T_{b+o}^{SC} to increase, and may potentially increase the overall time of the forward phase. Therefore, we can get the constraint of D_f as Equation 11 shows.

$$D_{start} \leq D_f \leq D_{MAX} \qquad (11)$$

Table 2. Benefit of Activation Block Swap.

Layer	Act Shape	FLOP	ST	SBF Ratio
Linear_qkv	$(b, s, 3h)$	$6bsh^3$	$3t_s$	1
Linear_htoh	(b, s, h)	$2bsh^3$	t_s	1
Linear_hto4h	$(b, s, 4h)$	$8bsh^3$	$4t_s$	1
Linear_4htoh	(b, s, h)	$8bsh^3$	t_s	4

Which activations to be swapped? We further compute the optimal D_f by analyzing the activations to be swapped. A transformer block contains four layers, namely Linear_qkv, Linear_htoh, Linear_hto4h and Linear_4htoh, whose output activation shape and FLOPs are listed in Table 2. <mark>For minimizing the swapping overhead, our optimization goal is to hide swap time behind recomputation time as much as possible.</mark>

Since the swap time (ST) is proportional to the activation size, we define the swap time of Linear_htoh as unit swap time t_s, thus the swap times of each layer can be calculated as ST in Table 2. According to the optimization goal, we can define the Swap Benefit Factor (SBF) of each layer as Equation 12 shows.

$$SBF = \frac{FLOP}{ST} \qquad (12)$$

The ratio of SBF of each layer is listed in Table 2. According to SBF of layers, we adopt a Prioritized Activation Swapping strategy to pick activations for swapping. During the profiling stage, all layers are pushed into two queues, where the high-priority queue contains Linear_4htoh layers while the low-priority queue contains other layers. So far, we have a concrete order of layers to swap activations.

To find the optimal D_f and corresponding layers to swap, we iterate the layers to swap. Let the activation size of the layer in bytes be S_{layer}. For each layer, swapping the layer subtracts the T_b^{comp} by $Tput_f \times FLOP_{layer}$, adds the T_{b+o}^{GPU} by S_{layer}/BW_{GPU}, and adds the T_{b+o}^{SSD} by $S_{layer}/BW_{S2C}N_{SSD}$. Therefore, we can compute the new iteration time T_{iter} by Equation 1. By iteratively computing T_{iter} when swapping i layers with the highest priority, we can pick the optimal i that meets Equation 11 while with minimum T_{iter}. The first i layers with the highest priority are thus the layers for swapping.

5 Evaluation

5.1 Experimental Setup

Evaluation Machine. We perform all experiments on a server, whose configurations are summarized in Table 3.
Workloads. We choose the GPT-3 model for our evaluation experiments, which is a typical 100B-level LLM. We adopt the same hyperparameters of GPT-3 13B and GPT-3 175B in the GPT-3 paper [5]. We set a series of custom configurations to evaluate Fuyou on more diverse model sizes, as shown in Table 4. We follow LLaMA [52] to choose the hyperparameters of GPT-3 33B and GPT-3 65B, and follow GPT-3 175B to proportionally extend the hyperparameters of GPT-3 135B,

Table 3. Configurations of our server.

CPU	Intel Xeon Gold 5320 CPU @ 2.20GHz
CPU Memory	768 GB 3200MHz DDR4
PCIe	PCIe Gen 4
GPU	NVIDIA A100 80GB
	NVIDIA Geforce RTX 4090
SSD	12× 3.84TB Intel P5510 SSDs
CUDA Toolkit	11.8
PyTorch	2.0.0+cu118

Table 4. Models for evaluation.

Model	#Layers	#Heads	Hidden Dimension
GPT-3 13B	40	40	5120
GPT-3 33B	60	52	6656
GPT-3 65B	80	64	8192
GPT-3 135B	88	88	11264
GPT-3 175B	96	96	12288
GPT-3 276B	112	112	14336
GPT-3 412B	128	128	16384
GPT-3 805B	160	160	20480

GPT-3 276B, GPT-3 412B and GPT-3 805B. In all evaluation experiments, the sequence length is set to 1024.

Baseline Configurations. To evaluate the effectiveness of Fuyou, we choose three open-source baselines. The first baseline is ZeRO-Infinity [41], the currently widely adopted open-source heterogeneous training system. The second baseline is ZeRO-Offload [44], which offloads the model states to CPU memory instead of SSDs, so ZeRO-Offload can only fine-tune much smaller models. For ZeRO-Infinity and ZeRO-Offload, we run our experiments with Deepspeed's official examples [26]. The release version we use is 0.9.3. We set the activation checkpoint granularity to each transformer block and offload checkpoints to CPU memory. Both baselines perform the optimizer stage on the CPU. ZeRO-Infinity offloads parameters and optimizer states to SSDs, whereas ZeRO-Offload offloads them to CPU Memory.

The third baseline is Colossal-AI, a popular billion-scale model training solution. We evaluate Colossal-AI with the official docker release of version 0.3.0 based on the official GPT-2 example [51]. For Colossal-AI, checkpoints are set for each transformer block, parameters and gradients are offloaded to the CPU, optimizer states are offloaded to the SSDs, and the optimizer stage is completed on the CPU. We do not offload activation checkpoints in Colossal-AI because it does not support.

5.2 Maximum Trainable Model Size

We first validate the maximum trainable model size of Fuyou over ZeRO-Infinity.[5] We train GPT-3 models on both A100-80GB and RTX 4090 with different CPU memory capacities.

[5]We do not compare with Colossal-AI and ZeRO-Offload because they support smaller trainable model sizes than ZeRO-Infinity.
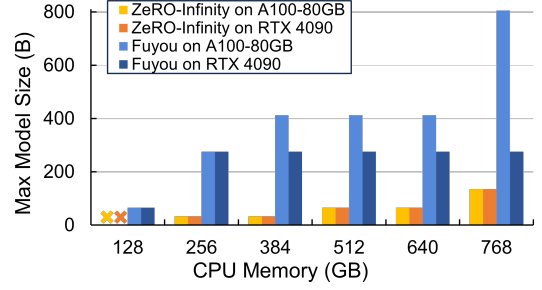
Figure 4. Maximum trainable model size of Fuyou and baselines under different CPU memory limits.

We set the batch size to 1 to minimize its effect. To limit CPU capacity, we pin a certain amount of memory so that both Fuyou and ZeRO-Infinity cannot utilize the pinned memory. Linux swap partition is disabled in our evaluations. Figure 4 illustrates the results. Here we have three observations.

First, Fuyou is able to fine-tune significantly larger models than ZeRO-Infinity under any CPU and GPU memory capacities, because Fuyou can fully leverage the memory capacities of CPU and GPU while ZeRO-Infinity cannot. Under 768 GB CPU memory, Fuyou enables the fine-tuning of 805B and 276B models on A100-80GB and RTX 4090, 5.96× and 2.04× larger than that of ZeRO-Infinity, respectively.

Second, the CPU memory capacity limits the largest model size of ZeRO-Infinity, because the maximum trainable model size with ZeRO-Infinity is the same under the same CPU memory limit, where A100-80GB has 80 GB GPU memory while RTX 4090 has only 24GB. Furthermore, ZeRO-Infinity fails to train the 13B model with 128 GB CPU memory on both A100-80GB and RTX 4090. In contrast, Fuyou succeeds in training a 65B model even with only 128 GB CPU memory and RTX 4090, which is reachable by most researchers.

Third, Fuyou can fine-tune larger models on A100-80 GB than that on RTX 4090, when CPU memory capacity is no less than 384 GB, indicating that 24GB GPU memory of RTX 4090 becomes the new bottleneck in this case. This is because a larger model brings a larger intermediate value size within a layer, which is not offloaded to CPU and SSDs, bringing high GPU memory requirements.

5.3 End-to-end Throughput Comparison

To demonstrate the efficiency of Fuyou, we compare the end-to-end training throughput of Fuyou and the three baselines. We employ Fuyou and baselines to fine-tune GPT-3 13B and 175B on both A100-80GB and RTX 4090 with different batch sizes.

Figure 5b illustrates the throughput of Fuyou and baselines when fine-tuning the 13B model on A100-80GB. Fuyou achieves at most 202 TFLOPS, which is 2.46×, 3.42×, and 6.73× improvements over ZeRO-Offload, ZeRO-Infinity, and Colossal-AI at their highest throughput respectively. With a batch size of 8, ZeRO-Offload achieves higher throughput
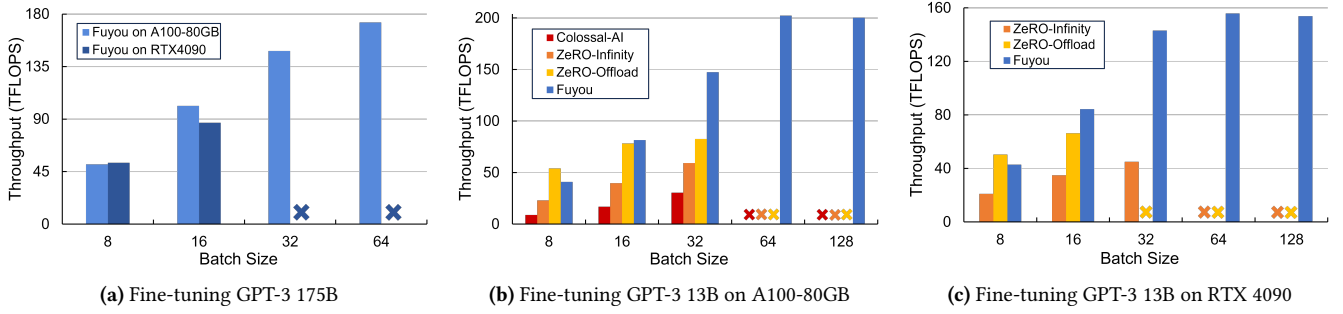
(a) Fine-tuning GPT-3 175B   (b) Fine-tuning GPT-3 13B on A100-80GB   (c) Fine-tuning GPT-3 13B on RTX 4090

Figure 5. End-to-end GPU throughput comparison between Fuyou and baselines with different batch sizes.
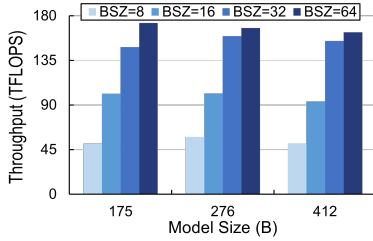
Figure 6. End-to-end GPU throughput of Fuyou fine-tuning extreme large GPT-3 models on A100-80GB.
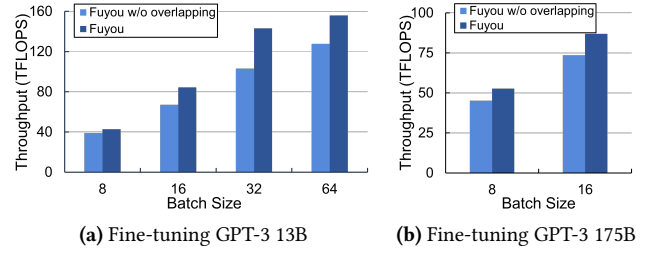

(a) Fine-tuning GPT-3 13B   (b) Fine-tuning GPT-3 175B

Figure 7. Effect of backward and optimizer overlapping.

than ZeRO-Infinity. This is reasonable since ZeRO-Offload doesn't offload optimizer and activations to SSDs, while in Fuyou with a small batch size CPU-SSD communication of the optimizer stage takes a large proportion of an iteration.

Figure 5c shows the throughput when fine-tuning the 13B model on RTX 4090. The figure does not include Colossal-AI since Colossal-AI fails to train the model on RTX 4090. Fuyou achieves 156 TFLOPS, which is 2.36× and 3.47× improvements over ZeRO-Offload and ZeRO-Infinity. Colossal-AI fails to run because Colossal-AI does not offload activation checkpoints, thus requiring larger GPU memory space than the 24GB memory capacity of RTX 4090.

Figure 5a shows Fuyou's throughput when fine-tuning a 175B model on A100-80GB and RTX 4090. All three baselines fail to fine-tune the 175B model under our hardware settings. On A100-80GB, Fuyou achieves a throughput of 173 TFLOPS, while on the smaller 13B model, it maintains 86% of this high throughput. On RTX 4090, Fuyou achieves a throughput of 86 TFLOPS, while on the 13B model, it maintains 55% of this throughput. Due to the GPU memory limitation, the supported batch size is relatively small compared to fine-tuning a 13B model, which limits the GPU throughput. This leaves the potential for further optimizations. However, compared to ZeRO-Infinity's throughput training the 13B model on RTX 4090, which is only 45 TFLOPS, this is still a considerable throughput.

Figure 6 show Fuyou's throughput when fine-tuning larger GPT-3 models on A100-80GB. With a batch size of 64, Fuyou achieves 168, 163 TFLOPS fine-tuning 276B and 412B models

respectively. This is not a significant drop compared to fine-tuning the 175B model.

In summary, Fuyou is able to fine-tune GPT-3 175B on RTX 4090 while the baselines aren't. When fine-tuning the same model on the same GPU, Fuyou achieves significantly higher throughput than the baselines, indicating that Fuyou enables efficient fine-tuning on large-scale models.

5.4 Effect of Backward and Optimizer Overlapping

To validate the effectiveness of overlapping backward and optimizer stages (Subsection 4.3), we compare Fuyou with Fuyou w/o overlapping, an implementation that disables backward and optimizer overlap optimization. We test Fuyou with Fuyou w/o overlapping fine-tuning GPT-3 13B and 175B on RTX 4090 GPU. Figure 7 illustrates the comparison results.

Fuyou achieves higher throughput than that without overlapping at all batch sizes due to the backward and optimizer overlapping mechanism. When fine-tuning GPT-13B on RTX 4090, compared to Fuyou w/o overlapping, Fuyou achieves 1.09×, 1.25×, 1.38× and 1.22× higher throughput when the batch sizes are 8, 16, 32, and 64, respectively. When fine-tuning GPT-175B, Fuyou achieves 1.16× and 1.18× higher throughput when the batch sizes are 8 and 16, respectively. The throughput gain drops when batch size is either too small or too large because in these cases backward propagation and optimizer stage have significantly different execution times, thus resulting in fewer overlapping opportunities.
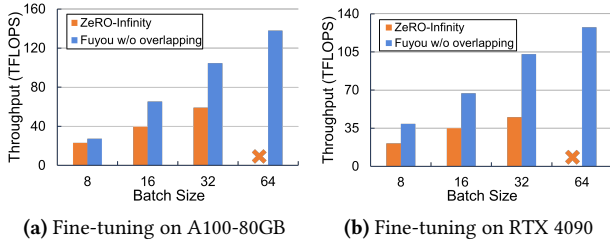
(a) Fine-tuning on A100-80GB   (b) Fine-tuning on RTX 4090

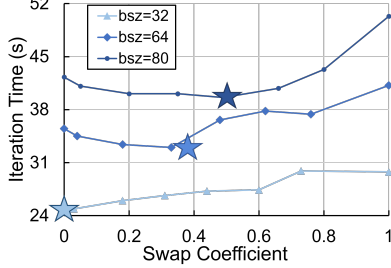**Figure 8.** Effect of pipelined activation swapping.

Figure 9. Iteration time of Fuyou fine-tuning GPT-3 13B on A100-80GB using different recomputation strategies. Stars are predicted optimal swap coefficients.

5.5 Effect of Pipelined Activation Swapping

To validate the effectiveness of the pipelined activation swapping (Subsection 4.4). We test Fuyou w/o overlapping and ZeRO-Infinity fine-tuning the GPT-3 13B on A100-80GB and RTX 4090 GPU with different batch sizes. Figure 8 illustrates the comparison results.

Fuyou w/o overlapping outperforms ZeRO-Infinity both on A100-80GB and RTX 4090. On A100-80GB, Fuyou w/o overlapping achieves 1.66×, 1.88× and 1.97× throughput at batch sizes of 8, 16 and 32, respectively, while on RTX 4090, Fuyou w/o overlapping achieves 1.85×, 1.92× and 2.28× throughput at batch sizes of 8, 16 and 32, respectively. The throughput gain is due to two reasons. First, we adopt the deeply pipelined execution strategy which overlaps GPU computation and PCIe communication. Second, ZeRO-Infinity has a performance issue because it uses pageable memories to store activations instead of pinned ones, which slows down the activation transmission between GPU and CPU.

5.6 Effect of Activation Swapping Management

To validate the effectiveness of the activation swapping management (Subsection 4.5), we test Fuyou with different activation swapping strategies fine-tuning GPT-3 13B on A100-80GB with 12 SSDs. Batch size is set to 32, 64, and 80. For activation swapping strategies, we define swap coefficient as the data volume ratio of activations to be swapped over all intra-transformer block activations. We test different swap coefficients and measure the training time of one iteration. Figure 9 illustrates the result, where stars indicate the optimal swap coefficient predicted by the automatic activation swapping management mechanism.
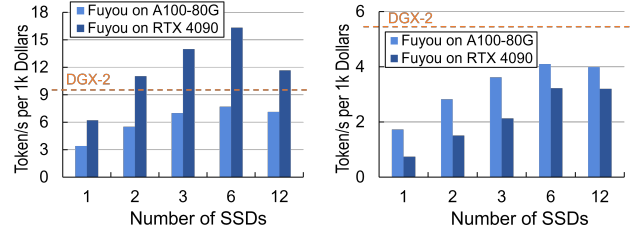
(a) Throughput over GPU+SSD prices.(b) Throughput over machine prices.

Figure 10. Comparison of throughput per 1000 dollars between Fuyou and Megatron-LM on DGX-2 when fine-tuning GPT-3 175B.

For the batch size of 32, the predicted swap coefficient is 0 because in this case the execution time for overlapped backward and optimizer stage is bounded by communication, increasing swapped activations doesn't help in reducing training time. For the batch size of 64 and 80, Fuyou provides a positive predicted swap coefficient. For the three batch sizes, Fuyou's automatic swapping mechanism produces nearly optimal predictions according to the experimental results.

5.7 Cost-Effectiveness Comparison

To show the cost-effectiveness of utilizing cheap SSDs in improving training throughput, we compare the cost-effectiveness of Fuyou with Megatron-LM [27] on NVLink-enhanced DGX-2 [31] nodes using tensor parallelism. Megatron-LM does not rely on data offloading. We choose the comparison metric to be throughput in token/s over price in dollars. The price of a machine and its components are estimated as Table 5 shows. We evaluate Fuyou both on A100-80GB and RTX 4090 with different SSD numbers. The evaluated model we use is GPT-3 175B to maximize the swapping overhead.

Table 5. Estimated price of server and components.

Machines and Components	Price ($)
DGX-2 server with 8 A100-80G NVLink GPUs	200,000 [11]
Commodity 4U server without GPUs and SSDs	14,098 [50]
NVIDIA A100-80GB	14,177 [50]
NVIDIA RTX 4090	1,600 [33]
Intel P5510 SSD	308 [50]

We first compare the throughput over the total price of GPUs[6] and SSDs in a server. Figure 10a illustrates that Fuyou on RTX 4090 achieves at most 1.70× cost-effectiveness over Megatron-LM. This shows that for large-scale training, by

---

[6]DGX-2 servers feature more powerful A100 SXM GPUs instead of A100 PCIe in commodity servers, which are bundled with whole-machine solutions. For Megatron-LM, we use the price of an A100 PCIe GPU as that of a GPU in DGX server.

offloading data to SSDs, a commodity GPU can still achieve comparable cost-effectiveness to high-end data-center clusters that do not rely on offloading to train a huge model. Besides, when the number of SSDs is no more than 6, the cost-effectiveness of Fuyou increases as the number of SSDs increases. This indicates the efficiency of system topology redesign. In particular, only increasing the number of cheap SSDs is an economical solution to significantly increase GPU utilization under Fuyou. When the SSD number increases from 6 to 12, the cost-effectiveness drops. This is because 12 SSDs in this case are larger than the optimal SSD number, and the performance gain from increasing the number of SSDs is diminishing.

We also compare the throughput over the price of the whole server, as shown in Figure 10b. Fuyou achieves 75% cost-effectiveness, compared to that of Megatron-LM.[7] The underlying reason is that the server itself takes the majority of the total cost under Fuyou. Since a typical commodity 4U server can contain at most 8 GPUs, increasing the number of commodity GPUs can achieve higher cost-effectiveness with proper optimizations. We leave this to our future work.

6 Related Works

To our knowledge, Fuyou is the first framework to enable efficient fine-tuning of extremely large-scale models using only one desktop GPU card. Table 1 summarizes the difference between Fuyou and some of the previous works. In this section, we further discuss previous researches that propose advancements in large-scale DNN training.

Offloading Model States and Activations to CPU Memory. Offloading has been a widely studied approach to reducing the memory footprint of the DNN model training process. Among these works, vDNN [45], TFLMS [22], LayRub [18], Zhang et al. [57], vDNN++ [48], Beaumont et al. [2, 4], Capuchin [37], TSplit [28], POET [36], STR [61] and Sentinel [43] support offloading activations to CPU memory. SuperNeurons [55], L2L [38], ZeRO-Offload [44], PatrickStar [9], and Elixir [16] support offloading model states to CPU memory. SwapAdvisor [15] and DeFiNES [23] support offloading both activations and model states to CPU memory. All these works support neither in-SSD activation offloading nor the out-of-core optimizer. In contrast, Fuyou proposes in-SSD activation offloading and efficient out-of-core synchronous optimizer, thus enabling a much larger model scale in a single GPU than in previous works.

SSD-Offloading Frameworks. Some existing works offload model states to NVMe SSDs which enable large-scale model training on a single GPU. Among these works, Flash-Neuron [1] uses GPUDirect and DPDK to offload activations to SSD, however, it does not support model state offloading

and out-of-core optimizer. G10 [56] uses GPUDirect Storage to offload model states and activation to SSDs, however, it performs optimizer on GPU thus leading to heavy network pressure between GPU and SSDs. ZeRO-Infinity [41] supports an out-of-core optimizer with synchronous weight update, however, it does not overlap the optimizer stage with backward propagation, which limits the model training efficiency. STRONGHOLD [49] in theory supports model states offloading to SSDs, but with low performance, because it is positioned as a fallback mechanism when CPU memory is not enough. Angel-PTM [29] supports an out-of-core optimizer which is overlapped with backward propagation, however, it adopts asynchronous weight update which affects model training convergence. In summary, all these works don't support out-of-core synchronous optimizer stages that overlap with the backward propagation, which is beneficial for fine-tuning on a single GPU. In contrast, Fuyou proposes an out-of-core synchronous optimizer while enabling optimizer overlapping with the backward stage, which ensures the maximum trainable model size while maintaining GPU utilization.

Activation Checkpointing Strategies. Chen et al. [6], Re-forwarding [10], Gruslys et al. [13], Herrmann et al. [14], Beaumont et al. [3], Kusumoto et al. [21], Checkmate [17] and DTR [20] focus on finding optimal activation checkpointing strategies to reduce memory footprint during training. Further, Beaumont et al. [4], Capuchin [37], TSplit [28], and POET [36] consider the optimal checkpointing strategies under the activation offloading scenario, while SuperNeurons [55] adopts an LRU-based activation checkpointing and offloading strategy when both model states and activations are only offloaded to CPU memory for CNN models. However, all these works only target scenarios without offloading or when activations are offloaded to CPU memory. In contrast, Fuyou is the first to schedule activation swapping and recomputation with CPU-SSD two-level offloading which has more complex PCIe traffic from a systematic view.

7 Conclusion

In this paper, we propose Fuyou, a low-cost training framework that enables efficient 100B huge model fine-tuning on a low-end server with a low-end GPU and limited CPU memory capacity. The key idea is to add SSD-CPU communication as an optimization dimension and thus carefully co-optimizer computation and data swapping from a systematic approach to maximize GPU utilization. To achieve this, first, we propose a synchronous out-of-core CPU optimizer that overlaps with backward propagation to maximize the GPU utilization. Second, we propose a GPU-CPU-SSD fully pipelined activation swapping mechanism to allow for a significantly larger model fine-tuning. Third, we present an automatic activation swapping management to automatically determine the optimal amount of swapping activations so as to minimize

---

[7] Our evaluation does not count the prices of network devices for the DGX-2 cluster, because we do not know the exact prices. Since Fuyou is evaluated for a single-GPU-training scenario, Fuyou does not need network devices.

the epoch time. We implement Fuyou based on PyTorch and show that Fuyou achieves 87 and 172 TFLOPS when fine-tuning GPT-3 175B on 4090 and A100-80GB respectively while ZeRO-Infinity and Colossal-AI fail to train. Besides, Fuyou reaches up to 3.42× and 6.73× TFLOPS compared to ZeRO-Infinity and Colossal-AI when fine-tuning GPT-3 13B on A100-80GB.

References

[1] J. Bae, J. Lee, Y. Jin, S. Son, S. Kim, H. Jang, T. J. Ham, and J. W. Lee. Flashneuron: Ssd-enabled large-batch training of very deep neural networks. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*, pages 387–401, 2021.

[2] O. Beaumont, L. Eyraud-Dubois, and A. Shilova. Optimal gpu-cpu offloading strategies for deep neural network training. In *European Conference on Parallel Processing*, pages 151–166. Springer, 2020.

[3] O. Beaumont, J. Herrmann, G. Pallez, and A. Shilova. Optimal memory-aware backpropagation of deep join networks. *Philosophical Transactions of the Royal Society A*, 378(2166):20190049, 2020.

[4] O. Beaumont, L. Eyraud-Dubois, and A. Shilova. Efficient combination of rematerialization and offloading for training dnns. *Advances in Neural Information Processing Systems*, 34:23844–23857, 2021.

[5] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.

[6] T. Chen, B. Xu, C. Zhang, and C. Guestrin. Training deep nets with sublinear memory cost. *arXiv preprint arXiv:1604.06174*, 2016.

[7] A. Chowdhery, S. Narang, J. Devlin, M. Bosma, G. Mishra, A. Roberts, P. Barham, H. W. Chung, C. Sutton, S. Gehrmann, et al. Palm: Scaling language modeling with pathways. *arXiv preprint arXiv:2204.02311*, 2022.

[8] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[9] J. Fang, Z. Zhu, S. Li, H. Su, Y. Yu, J. Zhou, and Y. You. Parallel training of pre-trained models via chunk-based dynamic memory management. *IEEE Transactions on Parallel and Distributed Systems*, 34(1):304–315, 2022.

[10] J. Feng and D. Huang. Optimal gradient checkpoint search for arbitrary computation graphs. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 11433–11442, 2021.

[11] Y. Feng, M. Xie, Z. Tian, S. Wang, Y. Lu, and J. Shu. Mobius: Fine tuning large-scale models on commodity gpu servers. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pages 489–501, 2023.

[12] R. C. Fernandez, A. J. Elmore, M. J. Franklin, S. Krishnan, and C. Tan. How large language models will disrupt data management. *Proceedings of the VLDB Endowment*, 16(11):3302–3309, 2023.

[13] A. Gruslys, R. Munos, I. Danihelka, M. Lanctot, and A. Graves. Memory-efficient backpropagation through time. *Advances in neural information processing systems*, 29, 2016.

[14] J. Herrmann, O. Beaumont, L. Eyraud-Dubois, J. Hermann, A. Joly, and A. Shilova. Optimal checkpointing for heterogeneous chains: how to train deep neural networks with limited memory. *arXiv preprint arXiv:1911.13214*, 2019.

[15] C.-C. Huang, G. Jin, and J. Li. Swapadvisor: Pushing deep learning beyond the gpu memory limit via smart swapping. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 1341–1355, 2020.

[16] H. Huang, J. Fang, H. Liu, S. Li, and Y. You. Elixir: Train a large language model on a small gpu cluster. *arXiv preprint arXiv:2212.05339*, 2022.

[17] P. Jain, A. Jain, A. Nrusimha, A. Gholami, P. Abbeel, J. Gonzalez, K. Keutzer, and I. Stoica. Checkmate: Breaking the memory wall with optimal tensor rematerialization. *Proceedings of Machine Learning and Systems*, 2:497–511, 2020.

[18] H. Jin, B. Liu, W. Jiang, Y. Ma, X. Shi, B. He, and S. Zhao. Layer-centric memory reuse and data migration for extreme-scale deep learning on many-core architectures. *ACM Transactions on Architecture and Code Optimization (TACO)*, 15(3):1–26, 2018.

[19] D. P. Kingma and J. Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[20] M. Kirisame, S. Lyubomirsky, A. Haan, J. Brennan, M. He, J. Roesch, T. Chen, and Z. Tatlock. Dynamic tensor rematerialization. *arXiv preprint arXiv:2006.09616*, 2020.

[21] M. Kusumoto, T. Inoue, G. Watanabe, T. Akiba, and M. Koyama. A graph theoretic framework of recomputation algorithms for memory-efficient backpropagation. *Advances in Neural Information Processing Systems*, 32, 2019.

[22] T. D. Le, H. Imai, Y. Negishi, and K. Kawachiya. Tflms: Large model support in tensorflow by graph rewriting. *arXiv preprint arXiv:1807.02037*, 2018.

[23] L. Mei, K. Goetschalckx, A. Symons, and M. Verhelst. Defines: Enabling fast exploration of the depth-first scheduling space for dnn accelerators through analytical modeling. In *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 570–583. IEEE, 2023.

[24] X. Miao, Y. Wang, Y. Jiang, C. Shi, X. Nie, H. Zhang, and B. Cui. Galvatron: Efficient transformer training over multiple gpus using automatic parallelism. *Proceedings of the VLDB Endowment*, 16(3), 2022.

[25] X. Miao, H. Zhang, Y. Shi, X. Nie, Z. Yang, Y. Tao, and B. Cui. Het: Scaling out huge embedding model training via cache-enabled distributed framework. *Proceedings of the VLDB Endowment*, 15(2):312–320, 2022.

[26] Microsoft. Megatron-deepspeed github repository, 2021. URL https://github.com/microsoft/Megatron-DeepSpeed.

[27] D. Narayanan, M. Shoeybi, J. Casper, P. LeGresley, M. Patwary, V. Korthikanti, D. Vainbrand, P. Kashinkunti, J. Bernauer, B. Catanzaro, A. Phanishayee, and M. 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, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450384421. doi: 10.1145/3458817.3476209. URL https://doi.org/10.1145/3458817.3476209.

[28] X. Nie, X. Miao, Z. Yang, and B. Cui. Tsplit: Fine-grained gpu memory management for efficient dnn training via tensor splitting. In *2022 IEEE 38th International Conference on Data Engineering (ICDE)*, pages 2615–2628. IEEE, 2022.

[29] X. Nie, Y. Liu, F. Fu, J. Xue, D. Jiao, X. Miao, Y. Tao, and B. Cui. Angel-ptm: A scalable and economical large-scale pre-training system in tencent. *arXiv preprint arXiv:2303.02868*, 2023.

[30] NVIDIA. Nvidia nsight systems, 2018. URL https://developer.nvidia.com/nsight-systems.

[31] NVIDIA. Nvidia dgx-2, 2019. URL https://www.nvidia.cn/data-center/dgx-2/.

[32] NVIDIA. Nvidia a100, 2020. URL https://www.nvidia.com/en-us/data-center/a100/.

[33] NVIDIA. Geforce rtx 4090, 2022. URL https://www.nvidia.com/en-us/geforce/graphics-cards/40-series/rtx-4090/.

[34] L. Ouyang, J. Wu, X. Jiang, D. Almeida, C. Wainwright, P. Mishkin, C. Zhang, S. Agarwal, K. Slama, A. Ray, et al. Training language models to follow instructions with human feedback. *Advances in Neural Information Processing Systems*, 35:27730–27744, 2022.

[35] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer. Automatic differentiation in

pytorch. In *NIPS 2017 Autodiff Workshop: The Future of Gradient-based Machine Learning Software and Techniques*, 2017.

[36] S. G. Patil, P. Jain, P. Dutta, I. Stoica, and J. Gonzalez. POET: Training neural networks on tiny devices with integrated rematerialization and paging. In K. Chaudhuri, S. Jegelka, L. Song, C. Szepesvari, G. Niu, and S. Sabato, editors, *Proceedings of the 39th International Conference on Machine Learning*, volume 162 of *Proceedings of Machine Learning Research*, pages 17573–17583. PMLR, 17–23 Jul 2022. URL https://proceedings.mlr.press/v162/patil22b.html.

[37] X. Peng, X. Shi, H. Dai, H. Jin, W. Ma, Q. Xiong, F. Yang, and X. Qian. Capuchin: Tensor-based gpu memory management for deep learning. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 891–905, 2020.

[38] B. Pudipeddi, M. Mesmakhosroshahi, J. Xi, and S. Bharadwaj. Training large neural networks with constant memory using a new execution algorithm. *arXiv preprint arXiv:2002.05645*, 2020.

[39] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, I. Sutskever, et al. Language models are unsupervised multitask learners. *OpenAI blog*, 1 (8):9, 2019.

[40] S. Rajbhandari, J. Rasley, O. Ruwase, and Y. He. Zero: Memory optimizations toward training trillion parameter models. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–16. IEEE, 2020.

[41] S. Rajbhandari, O. Ruwase, J. Rasley, S. Smith, and Y. 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*, pages 1–14, 2021.

[42] J. Rasley, S. Rajbhandari, O. Ruwase, and Y. 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*, pages 3505–3506, 2020.

[43] J. Ren, J. Luo, K. Wu, M. Zhang, H. Jeon, and D. Li. Sentinel: Efficient tensor migration and allocation on heterogeneous memory systems for deep learning. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 598–611. IEEE, 2021.

[44] J. Ren, S. Rajbhandari, R. Y. Aminabadi, O. Ruwase, S. Yang, M. Zhang, D. Li, and Y. He. {ZeRO-Offload}: Democratizing {Billion-Scale} model training. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 551–564, 2021.

[45] M. Rhu, N. Gimelshein, J. Clemons, A. Zulfiqar, and S. W. Keckler. vdnn: Virtualized deep neural networks for scalable, memory-efficient neural network design. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–13. IEEE, 2016.

[46] T. L. Scao, A. Fan, C. Akiki, E. Pavlick, S. Ilić, D. Hesslow, R. Castagné, A. S. Luccioni, F. Yvon, M. Gallé, et al. Bloom: A 176b-parameter open-access multilingual language model. *arXiv preprint arXiv:2211.05100*, 2022.

[47] M. Shoeybi, M. Patwary, R. Puri, P. LeGresley, J. Casper, and B. Catanzaro. Megatron-lm: Training multi-billion parameter language models using model parallelism, 2020.

[48] S. Shriram, A. Garg, and P. Kulkarni. Dynamic memory management for gpu-based training of deep neural networks. In *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 200–209. IEEE, 2019.

[49] X. Sun, W. Wang, S. Qiu, R. Yang, S. Huang, J. Xu, and Z. Wang. Stronghold: fast and affordable billion-scale deep learning model training. In *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–17. IEEE, 2022.

[50] Supermicro. Supermicro sys-420gp-tnr dual xeon scalable 4u gpu superserver, 2023. URL https://store.supermicro.com/us_en/4u-gpu-superserver-sys-420gp-tnr.html.

[51] H.-A. Tech. Colossal examples, 2021. URL https://github.com/hpcaitech/ColossalAI/tree/main/examples/language/gpt/gemini.

[52] H. Touvron, T. Lavril, G. Izacard, X. Martinet, M.-A. Lachaux, T. Lacroix, B. Rozière, N. Goyal, E. Hambro, F. Azhar, et al. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*, 2023.

[53] H. Touvron, L. Martin, K. Stone, P. Albert, A. Almahairi, Y. Babaei, N. Bashlykov, S. Batra, P. Bhargava, S. Bhosale, et al. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*, 2023.

[54] I. Trummer. The case for nlp-enhanced database tuning: towards tuning tools that" read the manual". *Proceedings of the VLDB Endowment*, 14(7):1159–1165, 2021.

[55] L. Wang, J. Ye, Y. Zhao, W. Wu, A. Li, S. L. Song, Z. Xu, and T. Kraska. Superneurons: Dynamic gpu memory management for training deep neural networks. In *Proceedings of the 23rd ACM SIGPLAN symposium on principles and practice of parallel programming*, pages 41–53, 2018.

[56] H. Zhang, Y. Zhou, Y. Xue, Y. Liu, and J. Huang. G10: Enabling an efficient unified gpu memory and storage architecture with smart tensor migrations. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 395–410, 2023.

[57] J. Zhang, S. H. Yeung, Y. Shu, B. He, and W. Wang. Efficient memory management for gpu-based deep learning systems. *arXiv preprint arXiv:1903.06631*, 2019.

[58] S. Zhang, S. Roller, N. Goyal, M. Artetxe, M. Chen, S. Chen, C. Dewan, M. Diab, X. Li, X. V. Lin, et al. Opt: Open pre-trained transformer language models. *arXiv preprint arXiv:2205.01068*, 2022.

[59] L. Zheng, Z. Li, H. Zhang, Y. Zhuang, Z. Chen, Y. Huang, Y. Wang, Y. Xu, D. Zhuo, E. P. Xing, J. E. Gonzalez, and I. Stoica. Alpa: Automating inter- and Intra-Operator parallelism for distributed deep learning. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 559–578, Carlsbad, CA, July 2022. USENIX Association. ISBN 978-1-939133-28-1. URL https://www.usenix.org/conference/osdi22/presentation/zheng-lianmin.

[60] Q. Zhou, H. Wang, X. Yu, C. Li, Y. Bai, F. Yan, and Y. Xu. Mpress: Democratizing billion-scale model training on multi-gpu servers via memory-saving inter-operator parallelism. In *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 556–569. IEEE, 2023.

[61] Z. Zong, L. Lin, L. Lin, L. Wen, and Y. Sun. Str: Hybrid tensor regeneration to break memory wall for dnn training. *IEEE Transactions on Parallel and Distributed Systems*, 2023.