Ratel: Optimizing Holistic Data Movement to Fine-tune 100B Model on a Consumer GPU

Changyue Liao*†, Mo Sun*†, Zihan Yang*†, Jun Xie†, Kaiqi Chen†, Binhang Yuan‡, Fei Wu†, Zeke Wang†
†Zhejiang University, China
‡HKUST, China

Abstract—**Nowadays, AI researchers become more and more interested in fine-tuning a pre-trained LLM, whose size has grown to up to over 100B parameters, for their downstream tasks. One approach to fine-tune such huge models is to aggregate device memory from many GPUs. However, this approach introduces prohibitive costs for most data scientists with a limited budget for high-end GPU servers. In this paper, we focus on LLM fine-tuning on a single consumer-grade GPU in a commodity server with limited main memory capacity, which is accessible to most AI researchers. In such a scenario, existing offloading-based methods fail to fine-tune an LLM efficiently due to a lack of holistic intra-server tensor movement management. To this end, we present Ratel, a low-cost, high-performance deep learning training framework that enables efficient 100B-scale model fine-tuning on a commodity server with a consumer-grade GPU and limited main memory capacity. The key idea is to add holistic offloading traffic as an optimization dimension for 1) active gradient offloading, and 2) holistic traffic-aware activation swapping mechanism. The experimental results show that 1) Ratel is the first to fine-tune a 175B model on an RTX 4090 and 256 GB main memory, 2) Ratel achieves 2.32× throughput than the state-of-the-art baselines when fine-tuning a small 13B model, and 3) Ratel enables a cheap low-end consumer GPU to have higher cost-effectiveness than a DGX-A100 cluster when fine-tuning a 175B model.**

I. Introduction

Large language models (LLMs) have achieved impressive accuracy in natural language processing jobs [1]–[5] and data management tasks [6], [7]. There is a strong demand for data scientists to fine-tune a pre-trained LLM to be used for downstream AI tasks [8], [9]. However, the model sizes of LLMs are growing fast. The largest open-source transformer models for fine-tuning in recent years have grown to over 100 billion (100B) parameters [10].[1] Fine-tuning a 100B model requires storing ~2.6 TB of temporary and persistent tensors at peak times, while the latest on-market data-center GPU has only up to 188 GB device memory [11].

One native approach to host huge models is to aggregate device memory from many data-center GPUs on high-end clusters like DGX platform [12] to fine-tune a 100B-scale model [13]–[30]. For example, it takes 32× $14177 NVIDIA A100 GPUs with 80 GB of device memory to fine-tune a model with 100B parameters, so accommodating a cluster of high-end GPUs introduces prohibitive costs for most data scientists with tight budgets.

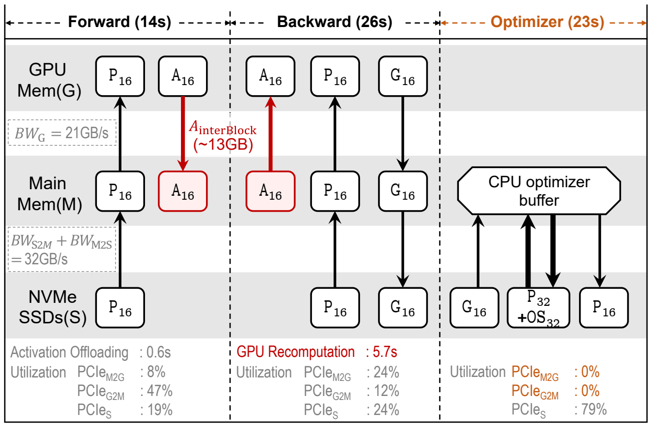
* Equal contribution.

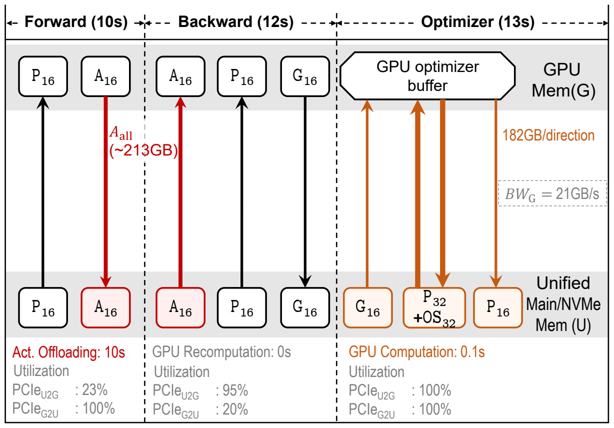
[1]Size of an LLM is defined by the number of parameters. We use "100B model" to represent a model with 100 billion parameters in this paper.

In this paper, we aim to explore whether it is feasible to efficiently *fine-tune a 100B-scale LLM on a single consumer-grade 4090 GPU ($1600, up to 24 GB device memory) with limited main memory capacity (256 GB)*. Such a solution would be attractive to researchers who seek to minimize LLM fine-tuning costs. To do so, the existing low-cost works [31]–[36] offload the tensors during the fine-tuning process from GPU memory to NVMe memory to maximize the trainable model size. However, we identify that these SSD-equipped systems suffer from two severe issues: low throughput and small maximum trainable model size.

- **Offloading Activation Tensors to SSDs.** LLM training consists of two types of tensors, namely activations and model states. Existing systems like FlashNeuron [37] only offload activations to SSDs and keep model states in GPU memory. We find that keeping model states in GPU memory severely limits the trainable model size. For example, FlashNeuron can only fine-tune a 1.55B model on RTX 4090, while fine-tuning a 175B model (a typical size of 100B-scale models [2], [38]) requires ~2.45 TB of GPU memory, which far exceeds the memory capacity of GPUs.
- **Offloading Model State Tensors to SSDs.** Existing systems like ZeRO-Infinity [39] and Colossal-AI [40] offload model states to NVMe SSDs [41]–[43] to enlarge the trainable model size. We identify that these systems have three issues in model fine-tuning, as shown in Figure 1a. First, these systems suffer from low GPU utilization, mainly because they execute the synchronous out-of-core CPU Adam[2] [44] in the optimizer stage where the GPU is idle. This stage takes 30%~60% of a training iteration. Second, these systems only offload the inter-transformer block activations (6% of total activations) to main memory and recompute the rest of the activations. Such an offloading method leads to 5.7 seconds (22% of the backward stage) of additional GPU recomputation overhead in the backward stage where PCIe bandwidth is underutilized. Third, these systems only offload activations to main memory, thus requiring a large amount of main memory to fine-tune an LLM. We estimate that ZeRO-Infinity requires ~1.1 TB main memory to fine-tune the 175B model, while most commodity servers equip only 128 GB~1 TB main memory.

[2]Here we refer to the out-of-core optimizer as optimizer executing on CPU instead of GPU (i.e., "in-core optimizer"). There are also works such as Angel-PTM [31] that present an asynchronous out-of-core optimizer. However, the asynchronous optimizer updating policy might affect model training convergence. Therefore, they are beyond the scope of this paper.


(a) ZeRO-Infinity: 1) executing the CPU optimizer in a separate stage where GPU and GPU-main memory link is idle, 2) only offloading inter-block activations and recomputing the others, which leads to 5.7s GPU recomputation in the backward stage thus causing only 24% PCIe utilization, 3) activations only offloaded to main memory thus limiting the trainable model size.


(b) G10: 1) transfering gradients, parameters and model states (182 GB per direction) between GPU and SSDs during the optimizer execution making GPU almost idle, and 2) offloading almost all activations to unified main/NVMe memory without recomputation thus causing ~213 GB of activation transfer when fine-tuning a 13B model.

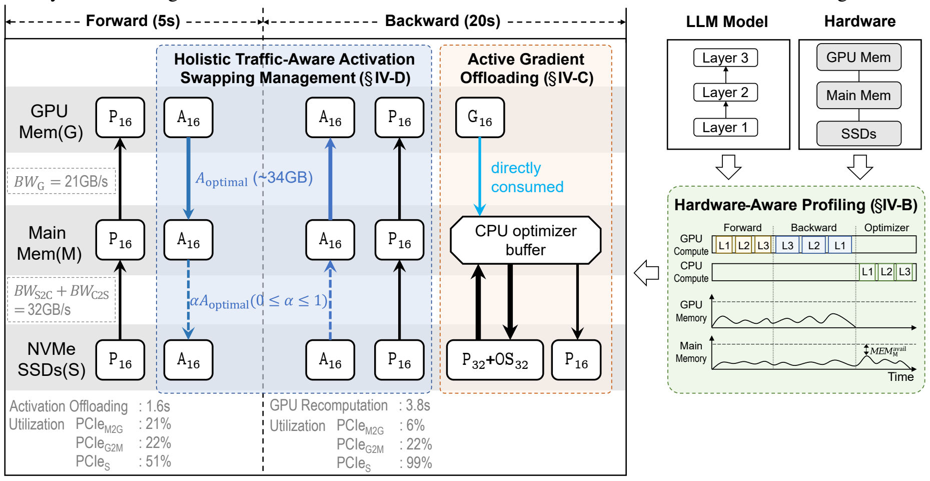

(c) Ratel: 1) directly consume gradients in the backward stage, and 2) determining an optimal amount of activations offloading to main memory and NVMe SSDs so that the total iteration time is minimum. When fine-tuning a 13B model, Ratel only offloads ~34 GB activation with 32% additional GPU computation in the backward stage.

Figure 1: Comparison of offloading-based systems. Bandwidth and overhead numbers are obtained on our evaluation server with 12 SSDs when fine-tuning a 13B model with a batch size of 32.

- **Naïvely Offloading Both Model State and Activation Tensors to SSDs.** Existing systems like G10 [45] offload both model states and activations to SSDs and execute the Adam optimizer on GPU. We identify that these systems have three issues, as shown in Figure 1b. First, executing the Adam optimizer on GPU requires transferring massive model states between GPU and NVMe SSDs, making the 0.1-second GPU computation wait for 13-second model state transfer. Second, these systems offload all the activations (213 GB when fine-tuning a 13B model with a batch size of 32) to SSDs, making 5.9-second GPU computation wait for 10-second activation transfer during the forward stage. Third, these systems rely on GPUDirect [46] technology that is not available for consumer-grade GPUs.

o In summary, they do not have holistic intra-server tensor management that impedes efficient fine-tuning on a 100B model when offloading activations or model states onto SSDs.

To this end, we present Ratel, a low-cost high-performance deep learning training framework that enables efficient 100B model fine-tuning on a commodity server with a consumer-grade GPU and limited main memory capacity. The key idea is to add holistic offloading traffic as an optimization dimension. As such, Ratel enables a commodity GPU without GPUDirect to efficiently fine-tune a huge model, whose size is limited by SSD capacity, rather than main memory/GPU memory size, when both model states and activations are offloaded to NVMe SSDs. To do so, Ratel consists of two innovations. First, for model states, Ratel presents the first active gradient offloading technology that enables the out-of-core CPU optimizer execution to directly consume the gradients from GPU to CPU, so as to hide CPU optimizer execution behind GPU computation. Second, for activations, Ratel proposes a holistic traffic-aware activation swapping management 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. To summarize, this paper makes the following contributions:

- We study the existing offloading strategies and identify the

issues of low throughput and small maximum trainable model size due to the lack of holistic intra-server tensor management.
- To optimize the holistic offloading traffic, we design an active gradient offloading technology and a holistic traffic-aware activation swapping management to overlap offloading and computation so as to maximize GPU utilization.
- We implement Ratel on the deep learning framework PyTorch [47]. Evaluations show that Ratel 1) is the first to fine-tune a 175B model on an RTX 4090 and 256 GB main memory, 2) achieves up to 2.32× throughput than the state-of-the-art baselines when fine-tuning a small 13B model, and 3) enables a cheap low-end consumer GPU to have higher cost-effectiveness than a DGX-A100 machine.

II. Background

LLM Training Stages. A model consists of L layers of mathematical functions $f_i\left(\boldsymbol{x}, \boldsymbol{P}_i\right), 1 \leq i \leq L$, where i denotes layer ID, $\boldsymbol{x}$ denotes input and $\boldsymbol{P}$ denotes its trainable parameters. The training procedure takes multiple training iterations to get the model converged. Each iteration consists of three stages:
- 1) Forward propagation, where the model takes training data $\boldsymbol{a}_0$ as input, computes the activation $\boldsymbol{a}_i = f_i(\boldsymbol{a}_{i-1}, \boldsymbol{P}_i)$ as indimediate values for each layer i successively and gets the loss value $\boldsymbol{l}_L = \boldsymbol{y} - \boldsymbol{a}_L$, where $\boldsymbol{y}$ denotes expected output.
- 2) Backward propagation, where the model performs two computations for each layer i in reverse to get gradients used for model updates: first, each layer takes the loss value $\boldsymbol{l}_i$ and computes the loss value delivered to the previous layer $\boldsymbol{l}_{i-1} = \nabla_{\boldsymbol{a}_{i-1}} f_i(\boldsymbol{a}_{i-1}, \boldsymbol{P}_i)^{\mathrm{T}} \boldsymbol{l}_i$; second, it computes the gradients of the layer $\boldsymbol{G}_i = \nabla_{\boldsymbol{P}_i} f_i(\boldsymbol{a}_{i-1}, \boldsymbol{P}_i)^{\mathrm{T}} \boldsymbol{l}_i$.
- 3) Optimizer execution, where the parameters are updated according to gradients, i.e., $\boldsymbol{P}^{\text{updated}} = o(\boldsymbol{G}, \boldsymbol{P})$, where o is the optimizer function. When training LLMs, Adam optimizer [44] is generally adopted to increase the model convergency, which introduces auxiliary optimizer states to smooth the parameter update process.

Notations. Table I lists notations used in the rest of the paper.

Memory Footprint. According to the training procedure introduced above, an LLM training iteration requires storing the following tensors: 1) Model states, including parameters $\mathtt{P}_{32}$, optimizer states $\mathtt{OS}_{32}$, gradients $\mathtt{G}_{16}$, and a low-precision parameter copy $\mathtt{P}_{16}$ for GPU computation; and 2) activations $\mathtt{A}_{16}$. Table II concludes the sizes and life cycles of the tensors within an iteration. The loss values are directly consumed after being produced, and thus are not discussed here.

Tensor Offloading. When no memory-saving techniques are applied, all the tensors are produced, stored, and consumed in GPU memory. Tensor offloading is a technique that moves part or all of the tensors from GPU memory to main memory or SSDs after the tensor is produced by GPU, and moves the tensor back to GPU memory before the tensor is consumed by GPU, so as to reduce the GPU memory footprint. The procedure of temporarily offloading a tensor is called swapping.

CPU Optimizer. Offloading optimizer execution to CPU [34] is a technology to reduce the PCIe traffic of GPU. The CPU optimizer eliminates the heavy parameter and optimizer state transfer between the GPU and main memory because when offloading the model states to NVMe SSDs, $\mathtt{P}_{32}$ and $\mathtt{OS}_{32}$ produced by the CPU optimizer in main memory are directly moved to SSDs. In contrast, $\mathtt{P}_{32}$ and $\mathtt{OS}_{32}$ produced by a GPU optimizer in GPU memory need to first move to main memory, then to SSDs.

Table I: List of notations.

Symbol	Definitions
T_{iter}	Elapsed time of a training iteration.
$T_{\text{f}}, T_{\text{b}}$	Elapsed time of the forward stage and the backward stage.
$T_{\text{f}}^{\text{G}}, T_{\text{f}}^{\text{G2M}}, T_{\text{f}}^{\text{M2G}}, T_{\text{f}}^{\text{S}}$	Elapsed time of GPU computation, GPU to main memory PCIe transfer, main memory to GPU PCIe transfer and SSD I/O during the forward stage.
$T_{\text{b}}^{\text{G}}, T_{\text{b}}^{\text{G2M}}, T_{\text{b}}^{\text{M2G}}, T_{\text{b}}^{\text{S}}$	Elapsed time of GPU computation, GPU to main memory PCIe transfer, main memory to GPU PCIe transfer and SSD I/O during the backward stage.
FLOP_{f}	Number of GPU floating point operations during the forward stage. The FLOP during the backward stage is thus $2\mathit{FLOP}_{\text{f}}$.
P	Number of parameters of a model.
A_{all}	Size of activations in bytes of a model.
$A_{\text{interBlock}}$	Size of inter-transformer block activations in bytes of a model.
THP_{G}	Peak GPU throughput in FLOPS measured.
BW_{G}	Maximum unidirectional PCIe bandwidth between GPU and main memory.
BW_{S2M}	Maximum SSD to main memory PCIe bandwidth measured.
BW_{M2S}	Maximum main memory to SSD PCIe bandwidth measured.
A_{G2M}	Activations size in bytes swapped from GPU.
$\mathit{MEM}_{\text{M}}^{\text{avail}}$	Minimum unallocated main memory in bytes during profiling stage.
α	Proportion of activations swapped to SSDs relative to A_{G2M}.
FLOP_{r}	Number of GPU floating point operations during recomputation.
OB	Activation offloading benefit of a layer, defined in Subsection IV-D.

Table II: Tensors in LLM fine-tuning.

Tensor	Produced During	Consumed During	Size
$\mathtt{P}_{32}$	optimizer (previous iteration)	optimizer (current iteration)	$4P$
$\mathtt{OS}_{32}$	optimizer (previous iteration)	optimizer (current iteration)	$8P$
$\mathtt{G}_{16}$	backward	optimizer	$2P$
$\mathtt{P}_{16}$	optimizer (previous iteration)	forward and backward (current iteration)	$2P$
$\mathtt{A}_{16}$	forward	backward	A_{all}

Activation Recomputation. Activation recomputation [48] is a memory-saving technique where only a subset of activations is kept in memory during forward propagation while others are discarded. During the backward propagation, when performing the backward propagation of a layer whose input activations are discarded, extra forward propagation from the last saved activation is performed to get the discarded activation. The extra forward propagation procedure is named recomputation.

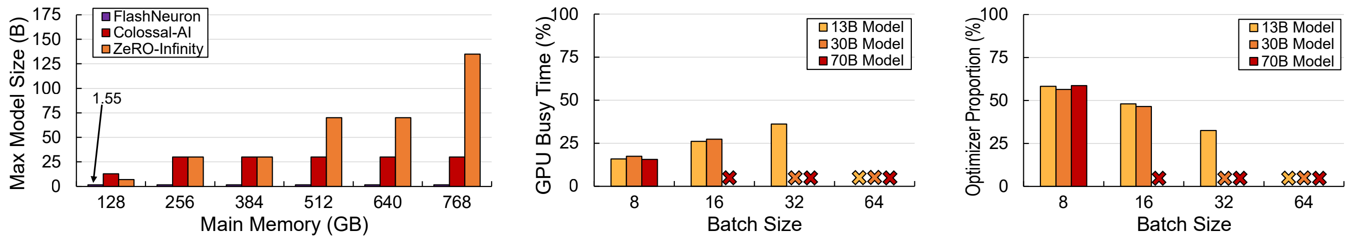
III. Motivation

In this section, we identify the issues of existing offloading-based works, which motivate the design of Ratel.

A. Issues of Approaches: Offloading Activations to SSDs

The existing works such as FlashNeuron [37] offload activations to SSDs to train a larger model. However, these systems keep the model states on GPU memory, thus severely limiting the maximum trainable model size.

To illustrate this issue, we implement a prototype of FlashNeuron and fine-tune LLMs on our evaluated server (Detailed implementation and server configurations shown in Section V-A). The experimental results in Figure 2a show that FlashNeuron even fails to fine-tune a 6B model, which is a common scale of today's pre-trained LLMs [9], [10], [38].



(a) Largest trainable model size of existing works under different main memory capacities.

(b) GPU utilization under different batch sizes in ZeRO-Infinity.

(c) Proportions of optimizer stage in a training step in ZeRO-Infinity.

Figure 2: Issues of SSD-offloading methods that motivate the design of Ratel. We perform the experiments on RTX 4090.

B. Issues of Approaches: Offloading Model States to SSDs

Systems like ZeRO-Infinity [39] and Colossal-AI [40] offload model states to SSDs to train a larger model. Meanwhile, they propose a CPU Adam [44] that efficiently executes the optimizer in the CPU. These systems perform badly in such a commodity server.[3] In the following, we identify three concrete issues of these systems.

1, Heavy Optimizer Execution Overhead. These systems execute CPU optimizer after the backward propagation of the entire model finishes, as Figure 1a shows. Thus, the CPU optimizer execution does not overlap with GPU computation, and the GPU is completely idle during the optimizer stage. When training in a high-end DGX cluster with many high-end CPUs, the CPU optimizer contributes a trivial time proportion. However, when fine-tuning on a commodity server with few CPUs, the optimizer execution takes a significant portion of the training time, leading to low GPU utilization during training.

To illustrate this, we quantitatively analyze the ratio of GPU busy time over the total elapsed time when fine-tuning different models using ZeRO-Infinity. Figure 2b shows that the GPU is busy during only 36% of an iteration, even when the model is relatively small (such as 13B) and the batch size is large enough to saturate GPU computing resources (such as 32). Colossal-AI achieves even lower GPU utilization (GPU is busy only for 12% of an iteration, not shown in the figure).

To show the overhead of the optimizer execution, we measure the time proportion of the optimizer stage in ZeRO-Infinity when fine-tuning different models in Figure 2c. The optimizer execution takes 30%~60% of the training step.

2, Excessive Activation Recomputation Overhead. These systems adopt a static activation management strategy. Specifically, they offload the inter-transformer block activations (12.5 GB for a 13B model with a batch size of 32) to main memory and recompute all the intra-transformer block activations (200 GB for a 13B model with a batch size of 32). Recomputing activations introduces additional GPU computation during backward propagation because GPU computation generally takes longer than PCIe tensor transfer when fine-tuning with a large batch size. For example, when fine-tuning a 13B model with a batch size of 32, the GPU backward propagation without recomputation takes 1.91× longer than SSD I/O time and 1.88× longer than the GPU-CPU activation and gradient transfer, thus GPU computation is the bottleneck. Therefore, this recomputation strategy incurs GPU computation overhead during the backward propagation.

[3]They are originally designed for high-end DGX servers rather than for a commodity server with a single consumer-grade GPU.

To illustrate this, we use ZeRO-Infinity to fine-tune a 13B model and break down its training time, as Figure 1a shows. The GPU-main memory tensor transfer $\texttt{PCIe}_{\texttt{G2M}}$ and main memory-SSD tensor transfer $\texttt{PCIe}_{\texttt{SSD}}$ only takes 3.18 and 6.25 seconds respectively during the backward stage, while GPU computation takes 17.6 seconds, indicating that excessive activation recomputation of ZeRO-Infinity introduces GPU computation overhead.

3, Limited Trainable Model Size under Limited GPU/Main Memory Capacity. ZeRO-Infinity offloads activation only to main memory, as Figure 1a shows, and Colossal-AI does not offload activations to either main memory or SSDs. These systems do not offload activations to SSDs because it incurs additional pressure to SSD I/O and introduces additional design complexity. However, hosting activations in GPU and main memory limits the maximum trainable model size when fine-tuning in a server with limited GPU and main memory.

To illustrate this, we fine-tune LLMs of various sizes with the two systems in our evaluated server (detailed configurations see Subsection V-A). We set the batch size to 1 to minimize the effect of activations. Figure 2a shows the maximum trainable model size with the two systems. They fail to fine-tune a 175B model in our evaluated server with 768GB main memory.

C. Issues of Approaches: Naïvely Offloading Both Model States and Activations to SSDs

A recent work namely G10 [45] supports offloading both model states and activations to unified main/NVMe memory, which theoretically supports the 100B-scale model fine-tuning with scarce GPU and main memory. G10 does not consider activation recomputation in model fine-tuning and offloads almost all activations to SSDs. Besides, it executes the Adam optimizer on GPU, as most in-GPU model training systems do. We identify that G10 has three issues in LLM fine-tuning.

1, Heavy Model States Transfer Overhead. G10 executes the optimizer on GPU which introduces massive model state transfer (182 GB per direction for a 13B model) on the PCIe link during the optimizer execution, causing heavy PCIe transfer overhead.

To illustrate this, we simulate the performance of G10 when fine-tuning a 13B model on RTX 4090 with a batch size of 32 assuming the GPUDirect is available and the GPU computation and PCIe transfer are fully pipelined. Figure 1b shows the result. We observe that the GPU optimizer only takes 0.1 seconds during this stage, while the PCIe transfer takes 13 seconds (37% of the iteration time).

2, High Activation Transfer Overhead. G10 offloads all activations to main memory and then to SSDs without recomputation, which incurs massive activation transfer (213 GB for a 13B model with a batch size of 32) on GPU's PCIe port, which leads to low GPU utilization during forward propagation. To illustrate this, we break down the training time of the forward stage, where offloading activations takes 10 seconds, far beyond the GPU computation time (5.96s).

This performance overhead would be more severe if we intend to overlap optimizer execution and the backward propagation whose time is bounded by GPU computation, like some existing works [49]. Our corresponding simulation shows that PCIe transfer accounts for almost 100% of both forward and the overlapped backward-optimizer stage time, while GPU computation only accounts for 59% of the forward stage time and 69% of the overlapped backward-optimizer stage time, indicating that the PCIe transfer for model states and activations becomes the bottleneck throughout the whole training process!

3, GPUDirect Requirement. G10 deeply relies on GPUDirect for tensor offloading. However, consumer-grade GPUs do not support GPUDirect, thus G10 cannot run on consumer-grade GPUs.

IV. Design of Ratel

A. Design Overview

To address the issues of existing works, we present Ratel, a holistic tensor management system that enables efficient low-cost 100B model fine-tuning on a commodity server with a consumer-grade GPU and limited main memory capacity. The key idea is to add holistic tensor offloading management as an optimization dimension. As such, Ratel achieves high GPU utilization when fine-tuning a 100B model, even when offloading both model states and activations to NVMe SSDs.

To do so, Ratel consists of three main components: 1) hardware-aware profiling that collects essential data for model states and activation management (Subsection IV-B), 2) active gradient offloading that enables the out-of-core CPU optimizer execution to directly consume the gradients from GPU to CPU, so as to hide CPU optimizer execution behind GPU computation (Subsection IV-C), and 3) holistic traffic-aware activation swapping management that automatically determines the amount of swapping activations to further minimize epoch time (Subsection IV-D). Figure 1c illustrates an overview of Ratel.

B. Hardware-Aware Profiling

In the hardware-aware profiling stage, Ratel automatically gathers essential data from both model and hardware settings, which are required by the subsequent components.

Profiling Goals. The profiling stage aims to provide the minimum unallocated main memory $MEM_{\mathrm{M}}^{\mathrm{avail}}$, the total elapsed time of the forward stage T_{f} and backward stage T_{b}, number of model parameters P, size of model activations A_{all}, peak GPU throughput THP_{G}, maximum PCIe bandwidth between GPU and main memory BW_{G}, maximum SSD to main memory PCIe bandwidth BW_{S2M}, maximum main memory to SSD PCIe bandwidth BW_{M2S}, and the number of GPU floating point operations of each layer, which are required by the holistic traffic-aware activation swapping management.

Profiling Details. Ratel parses the PyTorch model definition during initialization to obtain P, A_{all}, and the number of GPU floating point operations of each model layer. In the profiling stage, Ratel only offloads inter-layer activations and recomputes the rest of activations, just like ZeRO-Infinity, so as to minimize the activation offloading overhead while ensuring the recomputation process does not exceed the GPU memory limit. Besides, Ratel offloads all activations and model states to NVMe SSDs at this stage without any further optimizations to accurately break down computation and communication costs. we record the computation time of each layer in the model during forward propagation so as to compute THP_{G}. To get BW_{G}, BW_{S2M}, and BW_{M2S}, Ratel gets the system topology from hardware settings during initialization, and monitors the PCIe traffic during the profiling stage, so as to estimate the PCIe bandwidth of each link.

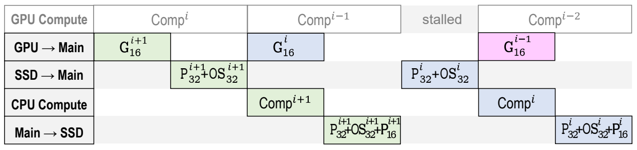
Profiling Overhead. We perform the hardware-aware profiling stage only in the first iteration, which takes about 2~3× times longer than that of a subsequent iteration. Fine-tuning an LLM requires thousands of iterations to converge, so the profiling overhead is negligible compared to the whole fine-tuning process.

C. Active Gradient Offloading

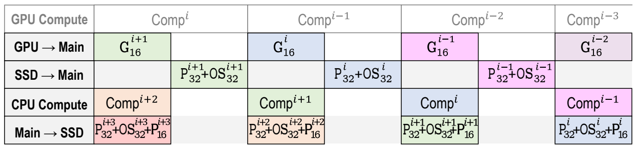
Inspired by *Active Messages* [50] that allows a sender to specify a user-level handler along with each message and requires a receiver to immediately call the handler on the message arrival with the message body as an argument, we present the active gradient offloading technology that allows the CPU to perform the out-of-core CPU optimizer execution (i.e., user-level handler) upon main memory receiving the offloaded gradients (message body) from GPU, rather than to further offload to SSDs. As such, Ratel has the opportunity to overlap the CPU optimizer execution with GPU backward propagation. In the following, we present the concrete challenge, followed by naïve active gradient offloading and optimized active gradient offloading.

Challenge. Model states of the 100B model are stored in low-bandwidth and high-latency SSDs, thus exploiting such an opportunity comes with a main challenge. In particular, how to enable the out-of-core CPU optimizer to efficiently consume the offloaded gradients from GPU during backward propagation, because they compete for limited PCIe bandwidth.

Naïve Active Gradient Offloading. Due to the stacked nature of transformer blocks, we assume the gradient tensors arrive at the CPU optimizer sequentially with a decreasing index during



(a) Naïve active gradient offloading.



(b) Optimized active gradient offloading.

Figure 3: Comparison of active gradient offloading designs.

the backward stage. When the gradient tensor i arrives at the main memory, its optimizer execution (user-defined handler) consists of three steps, as shown in Figure 3a. First, during the `SSD→Main` step, the SSD writes the corresponding model states of the model tensor i to main memory. Second, during the `CPU Compute` step, the CPU updates the model states of the tensor i and produces a 16-bit parameter copy. Third, during the `Main→SSD` step, the SSD reads the updated model states and 16-bit parameters back.

We can easily observe that the naïve active gradient offloading mechanism serializes three steps, such that the gradients are slowly consumed because expensive SSD accesses are involved in two steps.

Optimized Active Gradient Offloading. Our key observation is that GPU's backward propagation, in-core CPU optimizer (`CPU Compute`), and SSD I/O (`SSD→Main` and `Main→SSD`) utilize almost different computation and communication resources in a server, as shown in Figure 1c. Therefore, these three steps have the potential to overlap to maximize the utilization of GPU, PCIe, and CPU.

To this end, we are the first to present optimized active gradient offloading that overlaps the SSD I/O, in-core optimizer execution, and GPU's backward propagation to maximize GPU utilization. With optimized scheduling, `Main→SSD` of the tensor i is performed after `SSD→Main` of the tensor $(i-1)$, as shown in Figure 3b. By doing so, `Main→SSD` of the tensor i can be overlapped with `CPU Compute` of the tensor $(i-1)$, thus Ratel can overlap the CPU computation and SSD I/O during the out-of-core optimizer execution, increasing the PCIe bandwidth utilization between main memory and SSDs. As such, Ratel keeps synchronous model updating while minimizing the GPU's idle time.[4]

D. Holistic Traffic-Aware Activation Swapping Management

Swapping activations from GPU first to main memory then to SSDs incurs heavy PCIe traffic, while recomputing activations instead of swapping relieves PCIe traffic pressure at the cost of unnecessary GPU computation time. Existing activation swapping and recomputation works such as Capuchin [32] only consider GPU computation time of backward propagation and GPU-main memory PCIe traffic to determine the amount of swapping activations, assuming gradients, parameters, and model states are kept on the GPU memory.

Challenges. Ratel offloads both model states and activations to the SSDs and adopts the active gradient offloading, thus posing two unique challenges to the activation swapping design: 1) During the backward propagation, the GPU needs to swap back not only the activations but also the on-demand activations, incurring complex PCIe traffic. 2) Ratel adopts active gradient offloading that enables the CPU optimizer to overlap with backward propagation, so that the maximum execution time of backward propagation and the CPU optimizer serves to determine the swapping amount of the activations.

To address these challenges, Ratel proposes the holistic traffic-aware activation swapping management that automatically determines the amount of swapping activations and thus minimizes each iteration time. In the following, we describe Ratel's design in detail.

The goal of the activation swapping strategy is to find an A_{G2M} that minimizes the iteration time T_{iter}. The high-level observation is that T_{iter} is a convex function of A_{G2M}, so that the optimal A_{G2M} can be found by iterating activations to swap, computing the corresponding A_{G2M} and iteration time, and finding the inflection point where T_{iter} is minimized. We first discuss how we compute T_{iter} when given an A_{G2M}, then give proof that T_{iter} is convex, and finally show the concrete procedure of choosing activations to swap.

Iteration Time Computation. T_{iter} in Ratel is the sum of the forward stage time T_{f} and the backward stage time T_{b}, as shown in Equation 1.

$$T_{\mathrm{iter}} = T_{\mathrm{f}} + T_{\mathrm{b}} \tag{1}$$

We first evaluate T_{f}. When the GPU computation and PCIe tensor transfers are fully overlapped, the forward stage time is the maximum among $T_{\mathrm{f}}^{\mathrm{G}}$, $T_{\mathrm{f}}^{\mathrm{G2M}}$, $T_{\mathrm{f}}^{\mathrm{M2G}}$, and $T_{\mathrm{f}}^{\mathrm{S}}$, which can be expressed by Equation 2. Note that the GPU-CPU PCIe link is duplex, while the SSD is simplex, thus we need to compute GPU-CPU transfer time and CPU-GPU PCIe transfer time separately but consider the SSD I/O time as a whole.

$$\begin{aligned} T_{\mathrm{f}} &= \max\left(T_{\mathrm{f}}^{\mathrm{G}}, T_{\mathrm{f}}^{\mathrm{G2M}}, T_{\mathrm{f}}^{\mathrm{M2G}}, T_{\mathrm{f}}^{\mathrm{S}}\right) \\ &= \max\left(\frac{FLOP_{\mathrm{f}}}{THP_{\mathrm{G}}}, \frac{A_{\mathrm{G2M}}}{BW_{\mathrm{G}}}, \frac{2P}{BW_{\mathrm{G}}}, \frac{2P}{BW_{\mathrm{S2M}}} + \frac{\alpha A_{\mathrm{G2M}}}{BW_{\mathrm{M2S}}}\right) \end{aligned} \tag{2}$$

The last component of T_{f} includes the amount of activations that are swapped to SSDs αA_{G2M}. We next describe how αA_{G2M} is decided. When an activation is swapped in Ratel, it is accommodated by either main memory or SSDs. Ratel decides the amount of activations that are accommodated by main memory, $A_{\mathrm{G2M}} - \alpha A_{\mathrm{G2M}}$, based on the peak main memory usage gathered in the profiling stage. The main memory is first used for storing parameters that are prefetched from SSDs, and model states that are used for optimizer execution, while the rest of the main memory is used for accommodating activations. Therefore, the amount of activations accommodated by SSDs can be expressed as shown in

[4]Active gradient offloading might be confused with the "one-step delayed update" optimization of ZeRO-Offload [34]. The one-step delayed update postpones the optimizer execution of iteration i until the forward propagation of iteration $(i+1)$, thus introducing the parameter staleness and affecting the model training convergence. In contrast, Ratel does not introduce parameter staleness because forward/backward propagation reads the updated models.

Equation 3. Therefore, the forward stage time T_{f} can be further expressed by Equation 4.

$$\alpha A_{\mathrm{G2M}} = A_{\mathrm{G2M}} - MEM_{\mathrm{M}}^{\mathrm{avail}} \tag{3}$$

$$T_{\mathrm{f}} = \max\left(\frac{FLOP_{\mathrm{f}}}{THP_{\mathrm{G}}}, \frac{A_{\mathrm{G2M}}}{BW_{\mathrm{G}}}, \frac{2P}{BW_{\mathrm{G}}}, \frac{2P}{BW_{\mathrm{S2M}}} + \frac{A_{\mathrm{G2M}} - MEM_{\mathrm{CPU}}^{\mathrm{avail}}}{BW_{\mathrm{M2S}}}\right) \tag{4}$$

Then we evaluate T_{b}. Similar to the forward stage, the backward stage time can be expressed by Equation 5. Note that we do not consider the CPU Adam execution time because its time is shorter than reading/writing the optimizer states from/to SSDs.

$$\begin{aligned} T_{\mathrm{b}} &= \max\left(T_{\mathrm{b}}^{\mathrm{G}}, T_{\mathrm{b}}^{\mathrm{G2M}}, T_{\mathrm{b}}^{\mathrm{M2G}}, T_{\mathrm{b}}^{\mathrm{S}}\right) \\ &= \max\left(\frac{2FLOP_{\mathrm{f}} + FLOP_{\mathrm{r}}}{THP_{\mathrm{G}}}, \frac{2P}{BW_{\mathrm{G}}}, \frac{2P + A_{\mathrm{G2M}}}{BW_{\mathrm{G}}}, \frac{14P + \alpha A_{\mathrm{G2M}}}{BW_{\mathrm{S2M}}} + \frac{14P}{BW_{\mathrm{M2S}}}\right) \end{aligned} \tag{5}$$

Since $FLOP_{\mathrm{r}}$ can be computed by accumulating the computation quantity of layers that need to be recomputed, we can compute the iteration time once A_{G2M} is provided.

Proving Convexity of Iteration Time. We first list the mathematical theorems used in our proofs.

Theorem 1: The sum of convex functions is convex.

Theorem 2: The maximum of convex functions is convex.

Theorem 3: A linear function is a convex function.

Theorem 4: $f(y) = af(x) + b$ is convex to x if $f(x)$ is convex to x and $a > 0$.

To prove that the iteration time T_{iter} is convex to A_{G2M}, we prove that T_{f} and T_{b} is convex separately, thus proving T_{iter} is convex according to Theorem 1.

We first prove that T_{f} is convex to A_{G2M}. According to Equation 4, the first and the third components are independent of A_{G2M}, while the second and the last components are a linear function of A_{G2M}. Thus, all four components are convex according to Theorem 3. Therefore we conclude that T_{f} is a convex function of A_{G2M} according to Theorem 2.

Then we prove that T_{b} is convex to A_{G2M}. The second component of Equation 5 is independent of A_{G2M}, while the third and the last components are a linear function of A_{G2M}, thus the last three components are convex functions of A_{G2M} according to Theorem 3.

To prove that the first component $T_{\mathrm{b}}^{\mathrm{G}}$ is also convex, we first assume that activations of a layer can be partially offloaded while its recomputation overhead is proportional to discarded activations.[5] Next, we introduce Ratel's activation swapping order that is necessary for the proving process. For each layer's activations, Ratel assigns different swapping priorities. A layer's activation is more suitable for swapping rather than recomputing if it requires 1) more time to recompute or 2) less time to swap. Since the recomputing time of a layer is proportional to its operation quantity, and the offloading time is proportional to the activation size, we define the offloading benefit of a layer (OB_{layer}) as its floating point operations in recomputation ($FLOP_{\mathrm{layer}}$) over its activation tensor volume (A_{layer}), as shown in Equation 6.

[5]This assumption is only for interpolation, and we don't actually offload part of a layer's activation in reality.

$$OB_{\mathrm{layer}} = \frac{FLOP_{\mathrm{layer}}}{A_{\mathrm{layer}}} \tag{6}$$

A layer's activations that have higher offloading benefits have higher priority in swapping rather than recomputing.

Let A_i denote the activation size of the layer i, $FLOP_i$ denote its number of operations required in recomputation, and OB_i denote its offloading benefit. For each i that satisfies $\sum_{k=1}^{i} A_k \le A_{\mathrm{G2M}} \le \sum_{k=1}^{i+1} A_k$ (That is, A_{G2M} includes the first i layers and part of layer $(i+1)$), the recomputation overhead of a model can be expressed by Equation 7.

$$\begin{aligned} FLOP_{\mathrm{r}} &= FLOP_{\mathrm{f}} - \sum_{m=1}^{i} FLOP_{\mathrm{m}} - FLOP_{\mathrm{i+1}} \times \frac{A_{\mathrm{G2M}} - \sum_{n=1}^{i} A_n}{A_{i+1}} \\ &= FLOP_{\mathrm{f}} - \sum_{m=1}^{i} OB_m A_m - OB_{i+1} A_{i+1} \times \frac{A_{\mathrm{G2M}} - \sum_{n=1}^{i} A_n}{A_{i+1}} \end{aligned} \tag{7}$$

Thus the derivative of $FLOP_{\mathrm{r}}$ is expressed by Equation 8.

$$\frac{\mathrm{d}FLOP_{\mathrm{r}}}{\mathrm{d}A_{\mathrm{G2M}}} = -OB_{i+1} \tag{8}$$

Since OB_i is an decreasing function of i, $\frac{\mathrm{d}FLOP_{\mathrm{r}}}{\mathrm{d}A_{\mathrm{G2M}}}$ is an increasing function of A_{G2M}, thus $FLOP_{\mathrm{r}}$ is convex. Therefore, $T_{\mathrm{b}}^{\mathrm{G}}$ is also convex according to Theorem 4. According to Theorem 2, we conclude that the backward stage time T_{b} is a convex function of A_{G2M}.

Accordingly, the sum of T_{f} and T_{b}, which is the iteration time T_{iter}, is convex according to Theorem 1.

Concrete Procedure of Activation Swapping Strategy. Now we describe the concrete procedure of the activation swapping strategy. From the convexity of T_{iter}, we deduce three possible cases for the iteration time with regard to the offloaded activation size.

Case 1: The iteration time increases as A_{G2M} increases, indicating that the PCIe transfer is the training bottleneck even without recomputation. In this case, it is better to reduce the offloaded activation size as long as the recomputation does not exceed the GPU memory capacity. In Ratel, we choose $A_{\mathrm{interBlock}}$ as the minimum safe swapped activation amount by default.

Case 2: The iteration time decreases as A_{G2M} increases, indicating that GPU computation is the training bottleneck even when offloading all activations. In this case, all activations should be offloaded (i.e., $A_{\mathrm{G2M}} = A_{\mathrm{all}}$).

Case 3: As A_{G2M} increases, the iteration time decreases when A_{G2M} is smaller than a A_{optimal}, and increases when A_{G2M} is larger than the A_{optimal}, thus the A_{optimal} is the optimal offloaded activation size.

From the analysis, we can find the A_{optimal} by computing the iteration time when iterating different A_{G2M} and detecting the inflection point of T_{iter} with regard to A_{G2M} (Case 3). If no inflection point is detected, Ratel decides A_{G2M} by matching the pattern of T_{iter} to Cases 1 and 2. Algorithm 1 describes the details of this procedure.

Algorithm 1: Finding Optimal Activation Swapping Strategy.

```
Data: layer_list: List of all layers in the LLM.
Data: swap_list: List of swapped activations.
swap_list ← [];
$T_{\text{min}} \leftarrow \infty$;
$A_{\text{G2M}} \leftarrow 0$;
$FLOP_{\text{r}} \leftarrow FLOP_{\text{f}}$ ;            // Full recomputation
$i \leftarrow 0$;
layer_list.sortByOffloadingBenefit();
for layer in layer_list do
    $A_{\text{G2M}} \leftarrow A_{\text{G2M}}$+layer.actSize;
    $FLOP_{\text{r}} \leftarrow FLOP_{\text{r}}$−layer.flop;
    $T_{\text{iter}} \leftarrow$ computeIterTime($A_{\text{G2M}}$, $FLOP_{\text{r}}$);
    if $T_{\text{iter}} \geq T_{\text{min}}$ then
        if $A_{\text{G2M}} \geq A_{\text{interBlock}}$ then
            break ;
            // Ensure $A_{\text{G2M}} \geq A_{\text{interBlock}}$ to avoid OOM
        end
    else
        $T_{\text{min}} \leftarrow T_{\text{iter}}$;
    end
    swap_list.append(layer.activation)
end
```

```python
model = Model(config)




loss = nn.MSELoss()


optimizer = torch.optim.Adam()

for epoch in range(num_epochs):
    model.train()
    for batch_idx, (data, target)\
        in enumerate(train_loader):
        output = model(data)
        loss = loss(output, target)
        loss.backward()
        optimizer.step()
```

(a) PyTorch.

```python
with Ratel_init():
    # Apply profiling stage
    model = Model(config)

# Inject hooks for data prefetching and
# pipelined data transfer.
Ratel_hook(model)

loss = nn.MSELoss()

# Apply optimizer in active gradient offloading.
optimizer = Ratel_Optimizer(torch.optim.Adam())

for epoch in range(num_epochs):
    model.train()
    for batch_idx, (data, target)\
        in enumerate(train_loader):
        output = model(data)
        loss = loss(output, target)
        loss.backward()
        # optimizer.step()
```

(b) Ratel. Highlighted functions are APIs provided by Ratel.

Figure 4: User interface comparison of PyTorch and Ratel.

E. Framework Integration

We implement Ratel on the top of the popular deep-learning framework PyTorch [47]. Ratel provides a set of wrappers to hide the implementation details, so that users can enable efficient model fine-tuning via Ratel with only a few lines of code changes. Figure 4 shows the user interface comparison between PyTorch and Ratel. Ratel performs the profiling stage via the `Ratel_init` wrapper. Compared to PyTorch, Ratel removes the optimizer execution from the serialized execution flow. Besides, Ratel's hooking the operators in the model enables the automatic activation management without explicit code change by users.

V. Evaluation

A. Experimental Setup

Evaluated Machine. We perform all the experiments on a server whose configurations are summarized in Table III.

Workloads. We choose a series of decoder-only models for our experiments. The hyperparameter choice of the models follows GPT-3 [4] and open-source pre-trained models like OPT [2] and are listed in Table IV. We simply randomly initialize model parameters and datasets for evaluations that do not require model convergence. We train the models in mixed precision that is widely adopted in LLM fine-tuning. In our experiments, the sequence length is set to 1024 and the vocabulary size is 50257.

Table III: Configurations of the evaluation server.

CPU	Dual Intel Xeon Gold 5320 CPU @ 2.20GHz
Main Memory	768 GB 3200MHz DDR4
PCIe	PCIe Gen 4
GPU	NVIDIA GeForce RTX 3090/4080/4090
SSD	12× 3.84TB Intel P5510 SSDs
CUDA Toolkit	11.8
PyTorch	2.0.0+cu118

Table IV: LLM for evaluation.

Model Size	#Layers	#Heads	Hidden Dimension
6B	28	32	4096
13B	40	40	5120
30B	48	56	7168
70B	80	64	8192
135B	88	88	11264
175B	96	96	12288
276B	112	112	14336
412B	128	128	16384

Baseline Configurations. We choose the following open-source baselines for evaluation.

The first baseline is ZeRO-Infinity [39] and ZeRO-Offload [34] from DeepSpeed. The former offloads model states to SSDs while the latter offloads model states to main memory. Both systems swap the inter-transformer block activations to main memory and recompute the intra-block activations. We evaluate with DeepSpeed version 0.9.3 and disable the one-step delayed optimizer of ZeRO-Offload since it introduces parameter staleness.

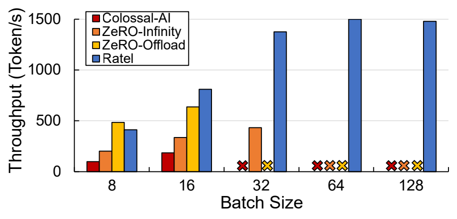
The second baseline is Colossal-AI [40], a popular billion-scale model training solution. Colossal-AI keeps the inter-block activations in GPU memory and recomputes the intra-block activations. We evaluate with Colossal-AI version 0.3.5 and enable the Gemini memory manager [51], [52].

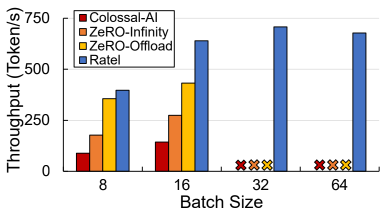
The third baseline is FlashNeuron [37], which only offloads activations to SSDs. We implement a prototype of FlashNeuron using POSIX file API instead of GPUDirect to offload activations to main memory, and then to SSDs, so that FlashNeuron can run on our consumer-grade GPUs.

B. Maximum Trainable Model Size

We first compare the maximum trainable model size of Ratel and the baselines by fine-tuning the models on three consumer-grade GPUs, namely RTX 4090, 3090 (24GB device memory), and 4080 (16 GB device memory), with different main memory capacities. We set the batch size to 1 to minimize effect of the batch size. To limit main memory capacity, we pin a certain amount of memory so that the evaluated systems cannot utilize the pinned memory. We further disable Linux swap partition.

Figure 6 illustrates the comparison results. Ratel is able to fine-tune significantly larger models than the baselines under any GPU and main memory capacities, because Ratel fully leverages the memory capacities of main memory and GPU to its best by holistically offloading model states and activations. Ratel enables the fine-tuning of a 276B model under 768 GB


(a) Fine-tuning 13B model on RTX 4090


(b) Fine-tuning 13B model on RTX 3090

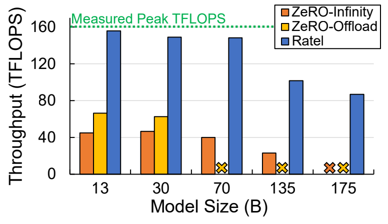

(c) Throughput vs model size on RTX 4090

Figure 5: End-to-end GPU throughput comparison between Ratel and baselines with different batch sizes.

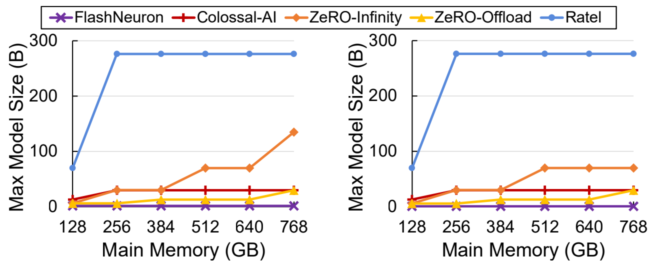

(a) on RTX 4090 and 3090 (b) on RTX 4080

Figure 6: Maximum trainable model size of Ratel and baselines under different main memory capacities.

main memory on RTX 4090, which is 2.04× larger than that of ZeRO-Infinity. Ratel succeeds in training a 175B model even with only 256 GB main memory and RTX 4080, which is reachable by most researchers.

C. *End-to-End Throughput Comparison*

Throughput w.r.t. Batch Size. To demonstrate the efficiency of Ratel, we first compare the end-to-end training throughput of Ratel and the three baselines. We employ Ratel and the baselines to fine-tune the 13B model on both RTX 4090 and 3090 with different batch sizes.

Figure 5a shows the throughput when fine-tuning the 13B model on RTX 4090. We observe that Ratel achieves 2.32×, 3.46×, and 8.02× higher throughput over ZeRO-Offload, ZeRO-Infinity, and Colossal-AI, respectively. The figure does not include FlashNeuron which fails to fine-tune the model on RTX 4090, because it only offloads activation checkpoints to SSDs while keeping massive model states in GPU memory, thus requiring much larger GPU memory space than the 24GB memory capacity of RTX 4090.

Figure 5b shows the throughput when fine-tuning the 13B model on RTX 3090. Ratel achieves 1.57×, 2.48×, and 4.72× improvements over ZeRO-Offload, ZeRO-Infinity, and Colossal-AI, respectively, showing a similar trend as on 4090.

Throughput w.r.t. Model Size. We compare the maximum TFLOPS of Ratel, ZeRO-Infinity, and ZeRO-Offload fine-tuning different models on 4090, as shown in Figure 5c, where the green line indicates the peak FLOPS measured by benchmarking a transformer block inside the GPU without any PCIe traffic.

We observe that Ratel achieves 90%~95% of peak FLOPS when the model size is smaller than 70B, while the baselines achieve only 40% at most. Ratel maintains a relatively small 53% of peak FLOPS when fine-tuning a 175B model, because a single layer of a large model has a large size of parameters and activations, thus the allowable batch size is small to fit in limited GPU memory. However, this is still significantly higher than ZeRO-Infinity at its maximum FLOPS.

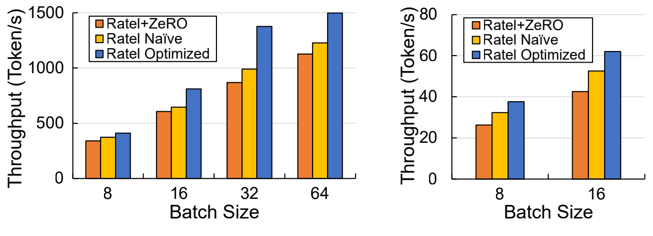

(a) Fine-tuning 13B Model (b) Fine-tuning 175B Model

Figure 7: Effect of active gradient offloading.

Conclusion. Ratel can fine-tune the 175B model on RTX 4090 while the baselines cannot. Besides, Ratel achieves significantly higher throughput than the baselines, indicating that Ratel enables efficient fine-tuning on large-scale models.

D. *Effect of Active Gradient Offloading*

To demonstrate the benefits of active gradient offloading (Subsection IV-C), we test Ratel with three implementations: 1) Ratel Optimized, the implementation with the optimized active gradient offloading, 2) Ratel Naïve that implements a naïve active gradient offloading, and 3) Ratel+ZeRO that does not overlap backward and optimizer execution as ZeRO-Infinity does. All implementations follow the same training procedure except the gradient offloading strategy.

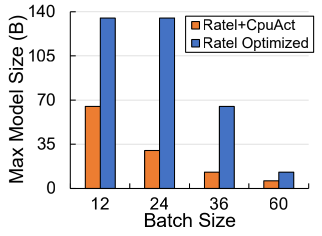
We test the implementations by fine-tuning the 13B and 175B models on RTX 4090 GPU, as shown in Figure 7. We make two observations.

First, the optimized active gradient offloading generally achieves higher performance gain than the naïve active gradient offloading. For example, Ratel Optimized achieves 1.22× throughput than Ratel Naïve and 1.33× throughput than Ratel+ZeRO when fine-tuning 13B model with a batch size of 64. This is because the optimized active gradient offloading completely overlaps CPU computation and SSD I/O, thus minimizing the GPU's idle time.

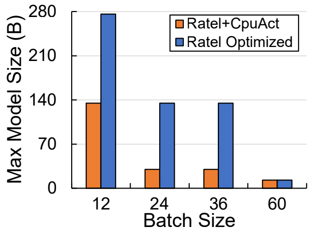
Second, the throughput gain of the active gradient offloading over serializing backward stage and the CPU optimizer drops when the batch size is too small (e.g., 8), because GPU backward propagation costs significantly less time than CPU optimizer, thus resulting in fewer overlapping opportunities.

E. *Effect of Holistic Traffic-Aware Activation Management*

To demonstrate the benefit of the holistic traffic-aware activation swapping management (Subsection IV-D), we first show the benefit of swapping activations to SSDs rather than



(a) With 128 GB main memory.



(b) With 256 GB main memory.

Figure 8: Effect of swapping activations to SSDs.

to main memory in maximum trainable model size, then show the throughput gain of the activation management strategy.

Benefit of Swapping Activations to SSDs. To show the benefit of swapping activations to both main memory and SSDs rather than only to main memory, we evaluate Ratel (Ratel Optimized) and an implementation Ratel+CpuAct, which follows the same training procedure as Ratel except that Ratel+CpuAct swaps activations only to main memory rather than to SSDs. We measure the maximum trainable model sizes of two implementations fine-tuning on RTX 4090 with different main memory and batch sizes.

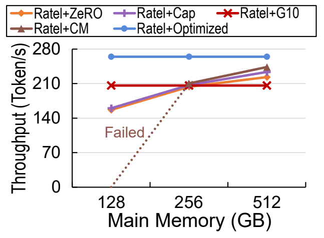
Figure 8 illustrates the comparison results. We observe that 1) swapping activations to SSDs significantly enlarges the trainable model size in the single-GPU commodity server with scarce main/GPU memory. For example, Ratel Optimized can fine-tune 2×~5× larger model than Ratel+CpuAct with 128 GB main memory. 2) The difference in trainable model size is not significant when the batch size is too large, e.g., the maximum model size of two implementations is the same with 256 GB main memory and batch size of 60, because when the batch size is too large, the maximum trainable model size is bounded by accommodating activations of a single layer in limited GPU memory capacity, rather than main memory.

Benefit of the Activation Management Strategy. To validate the effectiveness of the activation management strategy, we evaluate Ratel on the 70B model with five implementations: 1) Ratel+Optimized that uses holistic traffic-aware activation management to swap activations, 2) Ratel+ZeRO that statically swaps the inter-layer activations of each transformer block to main memory and recompute the rest, 3) Ratel+Cap that smartly keeps, recomputes or swaps the activations to CPU by profiling the overhead of activation swapping and recomputation as proposed by Capuchin [32], 4) Ratel+G10 that smartly keeps or swaps activations to SSDs based on inactive time measurement as proposed by G10 [45], and 5) Ratel+CM that smartly recomputes or offloads activations to main memory with a cost-model and MILP solver proposed by Checkmate [53]. All the implementations offload model states to SSDs and execute optimizer in CPU, which is necessary for 70B model fine-tuning.

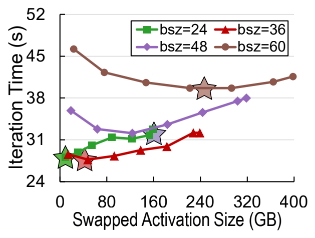
Figure 9a shows the throughput comparison and Table V shows the respective batch sizes. We observe that 1) the performance of all baselines except G10 drops when with less main memory capacity because these systems swap activations only to main memory, thus limiting the achievable batch

Table V: Batch size adopted by different activation management strategies fine-tuning the 70B model.

Main Memory Size	128 GB	256 GB	512 GB
Ratel+DS	16	24	32
Ratel+Cap	16	24	32
Ratel+G10	32	32	32
Ratel+CM	Failed	24	32
Ratel+Optimized	32	32	32



(a) Throughput of Ratel integrating different activation management strategies.



(b) Iteration time of Ratel with different amounts of swapped activations. Stars are predicted optimal size.

Figure 9: Effect of activation management strategy.

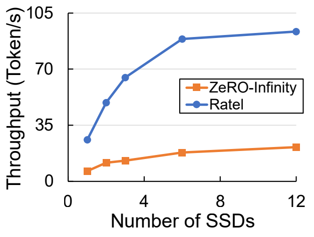
size with scarce memory capacity. In contrast, Ratel achieves steady throughput by offloading activations to SSDs; and 2) With the same batch size (e.g., the batch size is 32 with 512 GB main memory), Ratel achieves higher throughput than all the baselines, because Ratel's offloading strategy is holistic, considering the traffic from both activations and model states.

Further, we show that Ratel predicts the optimal amount of swapped activations. To illustrate this, we test the iteration time of Ratel when fine-tuning the 13B model with different amounts of swapped activation. Figure 9b illustrates the results with batch sizes of 24, 36, 48, and 60, where stars indicate the predicted optimal amount of swapped activations. We observe that 1) for all batch sizes, Ratel's iteration time model produces nearly optimal predictions according to the experimental results. 2) The iteration time increases as the swapped activation amount increases at a batch size of 24, which fits case 1 deduced in Subsection IV-D. Meanwhile, the trend of iteration time with regard to swapped activation amount fits well with deduced case 3 at batch sizes of 36, 48, and 60, showing the correctness of Ratel's iteration time model and the preciseness of the profiling stage.

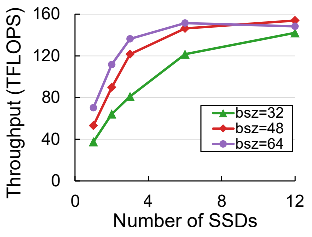
F. Effect of the Number of SSDs

We study Ratel's throughput w.r.t. number of SSDs. We evaluate the maximum training throughput of Ratel and ZeRO-Infinity when fine-tuning the 135B model (the largest model ZeRO-Infinity can fine-tune) on RTX 4090 with different numbers of SSDs. We adopt the largest batch size the two systems can fine-tune. Figure 10a illustrates the effect of the number of SSDs on the achievable throughput.

We make three observations. First, Ratel achieves near linear scalability when the SSD number increases from 1 to 3, indicating that SSD I/O is the training bottleneck in this case and Ratel aggregates the bandwidth of multiple SSDs well. Second, Ratel's throughput gain is small as the SSD number increases from 6 to 12. This is because the system bottleneck with adequate I/O bandwidth has shifted to GPU computation and GPU-main memory PCIe transfer. Third, ZeRO-Infinity's

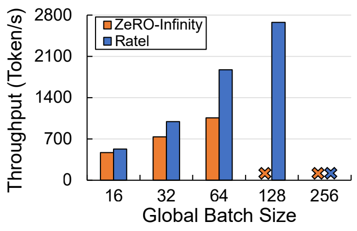


(a) Maximum throughput of Ratel and ZeRO-Infinity when fine-tuning 135B model.

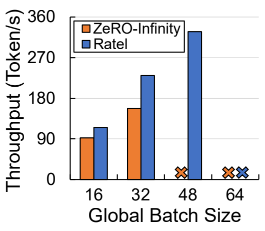


(b) Throughput of Ratel when fine-tuning 13B model with different batch sizes.

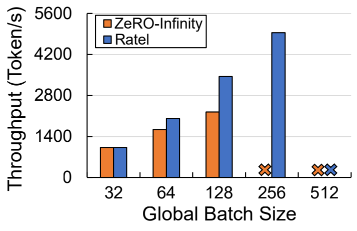
Figure 10: Effect of the Number of SSDs.



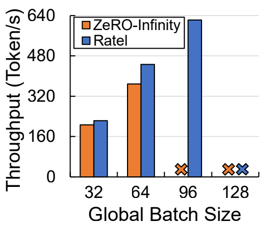
(a) 13B model with 2 GPUs.



(b) 70B model with 2 GPUs.



(c) 13B model with 4 GPUs.



(d) 70B model with 4 GPUs.

Figure 11: Throughput comparison between Ratel and ZeRO-Infinity on 4-GPU machine.

throughput grows slowly as the number of SSDs increases. This is because ZeRO-Infinity almost serializes GPU computation, CPU computation, and SSD access, thus the I/O bandwidth of SSDs is not well utilized.

To further study the throughput characteristics w.r.t. SSD number, we measure the training throughput of Ratel when fine-tuning the 13B model on RTX 4090 with different SSD numbers and batch sizes. Figure 10b illustrates the result.

We make two observations. First, Ratel's throughput is maximized under sufficient cheap SSDs. With more than 6 SSDs for batch sizes of 32 and 48 or 12 SSDs for batch size of 64, Ratel achieves the near maximum throughput. Second, Ratel needs fewer SSDs to reach maximum throughput when using a larger batch size. For example, It requires 12 SSDs to achieve 135 TFLOPS with a batch size of 32, while requiring 6 and 3 SSDs for batch sizes 48 and 64 respectively.

G. *Performance on Multi-GPU Server*

Many data scientists might own a commodity server with multiple consumer-grade GPUs. We show that Ratel's optimizations also work for the multi-GPU servers. We evaluate the training throughput of Ratel and ZeRO-Infinity on a multi-GPU server, whose configurations are the same as the single 4090 server, except that the multi-GPU server equips 4 NVIDIA RTX 4090 GPUs (The maximum supported number within the server's power supply). We use the two systems to

Table VI: Diffusion models for evaluation.

Model Size	#Layers	#Heads	Hidden Dimension
0.67B	28	16	1152
0.90B	30	16	1280
1.4B	32	16	1536
10B	28	32	4096
20B	40	40	5120
40B	48	56	7168

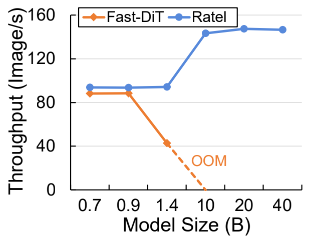


Figure 12: Throughput on diffusion models: Ratel vs. Fast-DiT.

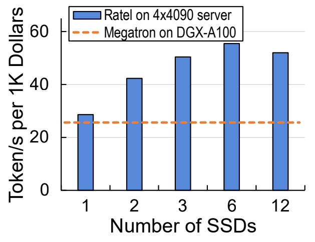


Figure 13: Comparison of Cost-effectiveness: Ratel vs. Megatron on DGX-A100.

fine-tune the 13B model and the 70B model (The largest model ZeRO-Infinity can fine-tune[6]) at different global batch sizes. Figure 11 shows the global throughput of the two systems fine-tuning on 2 and 4 RTX 4090 GPUs.

The experimental result shows that Ratel achieves higher throughput than ZeRO-Infinity. Ratel achieves 2.21× and 1.69× throughput than ZeRO-Infinity when fine-tuning the 13B and 70B models on 4 GPUs respectively. The underlying reason is two-fold. First, Ratel offloads the activations to SSDs, thus allowing the fine-tuning with a larger batch size. Second, Ratel considers holistic offloading traffic as an optimization dimension, and thus achieves higher throughput than ZeRO-Infinity even with the same batch size. In conclusion, Ratel still benefit the fine-tuning on a multi-GPU server.

H. *Performance on Other Large-Scale Models*

Large-scale models not only exist in language models (LMs) but also in models for tasks such as vision and image generation [54]–[57]. We take fine-tuning diffusion models as an example to show that Ratel's optimizations are beneficial for more general large-scale deep-learning models.

To evaluate performance on large diffusion models, we adopt the model architecture of the DiT-XL/2 [54] model and scale the layer number, attention head number, and hidden dimension of the backbone, as listed in Table VI. The input image size is 512×512. We compare the throughput of Ratel and Fast-DiT [58], the state-of-the-art open-source training framework for DiT models. Figure 12 illustrates the result. We make two observations.

First, Ratel enables fine-tuning much larger models than Fast-DiT because Ratel offloads activations and model states to main memory and SSDs. In contrast, Fast-DiT keeps the tensors in GPU memory.

Second, Ratel achieves higher throughput than Fast-DiT when fine-tuning the same model. The reason is two-fold:

[6]Even though ZeRO-Infinity can fine-tune the 135B model with a single RTX 4090, it can only fine-tune the 70B model on the multi-GPU server because of the additional GPU and main memory overhead introduced by multi-GPU synchronization and multiprocessing.

Table VII: Estimated price of components.

Machines and Components	Price ($)
DGX-A100 server with 8 A100-80G NVLink GPUs	200,000 [59]
Commodity 4U server, without GPUs and SSDs	14,098 [60]
NVIDIA RTX 4090	1,600 [61]
Intel P5510 SSD	308 [60]

1) Fast-DiT suffers from low throughput due to small trainable batch size as the model size grows (e.g., 1.4B), while Ratel allows the fine-tuning with high batch size; and 2) Ratel's activation management strategy reduces the fine-tuning time compared to Fast-DiT's static activation swapping strategy, thus enabling Ratel to achieve higher throughput even when both two systems fine-tune at the same large batch size.

I. Cost-Effectiveness Comparison

To show the cost-effectiveness of utilizing holistic offloading in improving training throughput, we compare the throughput over the server price of Ratel on the 4× RTX 4090 GPU server and a baseline: Megatron-LM [62] on an NVLink-enhanced DGX-A100 [12] server using tensor parallelism. Megatron-LM does not rely on data offloading. We fine-tune the 30B model (the largest model Megatron-LM can fine-tune on the DGX machine) on the two systems. The prices of server components are estimated as shown in Table VII. We evaluate Ratel on RTX 4090 GPUs with different SSD numbers when fine-tuning the 30B model.

Figure 13 illustrates the comparison results. We observe that Ratel on RTX 4090 achieves at most 2.17× cost-effectiveness over Megatron-LM on a DGX-A100 machine. This shows that for large-scale training, Ratel enables a commodity GPU to achieve higher cost-effectiveness than high-end data-center clusters that do not rely on offloading to train a huge model. Here cost-effectiveness decreases when Ratel's number of SSDs is increased from 6 to 12 because adding SSDs beyond the optimal number of SSDs has only a small performance gain but raises costs.

VI. Related Works

Offloading Optimization in Data Management Tasks. Prior works [41], [63]–[68] study the characteristics of modern storage devices such as SSDs and provide guidelines for optimizing the data offloading in data management tasks.

Following these works, existing practices have optimized the storage I/O of several data management fields such as buffer management [69]–[73], indexing [74], [75], query scheduling [73], [76], and transaction logging [76], [77] of rational databases, or other fields such as distributed databases [78], [79], object databases [80], key-value stores [81]–[85], vector data processing [86], graph data processing [87]–[94], and information retrieval [95].

Compared to these systems, Ratel targets the LLM fine-tuning tasks that have different application characteristics.

Tensor Management Methods for LLM Fine-tuning. Many existing works [48], [53], [96]–[101] consider the optimal recomputation strategies while keeping the rest of activations and the entire model states on GPU, thus these works fail to fine-tune even a 1B model. In contrast, Ratel is the first to schedule activation offloading and activation recomputation under holistic activations and model states offloading.

Many existing works [35], [36], [102]–[111] offloads the activations to main memory to train models that cannot fit in GPU memory. FlashNeuron [37] further offloads activations to NVMe SSDs. However, these systems do not offload model states, thus they even fail to fine-tune a 6B model, and only considering the optimal activation swapping and recomputation is not optimal when offloading both activations and model states. Further, some works [32], [112]–[114] offloads both activations and model states to main memory, and G10 [45] further offloads these tensors to SSDs. However, these systems execute optimizer in GPU, thus incurring heavy model state transfer overhead on PCIe interconnect as discussed in Subsection III-C. In contrast, Ratel considers both activation management and model states offloading to SSDs as optimization dimensions, thus enabling 100B-scale LLM fine-tuning on a single GPU while keeping efficiency.

Some prior works [34], [49], [52], [115], [116] introduce CPU Adam and offload model states to main memory so as to enlarge the trainable model sizes of LLMs and ZeRO-Infinity [39] offloads the model states to SSDs. However, these systems do not holistically manage activations swapping, activation swapping, and model state offloading. In particular, they either do not adopt activation recomputation thus incurring heavy activation transfer overhead on PCIe interconnect, or only adopts a naive activation recomputation strategy. Angel-PTM [31] adopts asynchronous weight updates which leads to parameter staleness. In contrast, Ratel presents the first active gradient offloading that overlaps CPU's SSD accesses, optimizer execution, and GPU's backward propagation to maximize GPU utilization. In contrast, Ratel is the first to propose holistic traffic-aware activation swapping management that achieves optimal activation management under large model sizes, as shown in Subsection V-E.

VII. Conclusion

In this paper, we propose Ratel, a low-cost high-performance training framework that enables efficient 100B huge model fine-tuning on a low-end server with a low-end GPU and limited main memory capacity. The key idea is to add holistic offloading traffic as an optimization dimension. The experimental results show that 1) Ratel is the first to fine-tune a 175B model on an RTX 4090 and 256 GB main memory, and 2) Ratel enables a cheap low-end consumer GPU to have higher cost-effectiveness than a DGX-A100 cluster when fine-tuning a 175B model. Ratel's artifact is available at https://github.com/RC4ML/LoHan.

Acknowledgement. The work is supported by the following grants: the National Key R&D Program of China (Grant No. 2022ZD0119301), the National Natural Science Foundation of China under the grant number (62472384, 62441605, U24A20326), Starry Night Science Fund of Zhejiang University Shanghai Institute for Advanced Study (SN-ZJU-SIAS-0010). Zeke Wang is the corresponding author.

References

[1] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," in *NAACL-HLT*, 2019.

[2] S. Zhang, S. Roller, N. Goyal, M. Artetxe, M. Chen, S. Chen, C. Dewan, M. Diab, X. Li, X. V. Lin, T. Mihaylov, M. Ott, S. Shleifer, K. Shuster, D. Simig, P. S. Koura, A. Sridhar, T. Wang, and L. Zettlemoyer, "Opt: Open pre-trained transformer language models," *arXiv preprint*, 2022.

[3] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever, "Language models are unsupervised multitask learners," *OpenAI blog*, 2019.

[4] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei, "Language models are few-shot learners," in *NeurIPS*, 2020.

[5] T. L. Scao, A. Fan, C. Akiki, E. Pavlick, S. Ilić, D. Hesslow, R. Castagné, A. S. Luccioni, F. Yvon, M. Gallé, J. Tow, A. M. Rush, S. Biderman, A. Webson, P. S. Ammanamanchi, T. Wang, B. Sagot, N. Muennighoff, A. V. del Moral, O. Ruwase, R. Bawden, S. Bekman, A. McMillan-Major, T. Wolf, I. Beltagy, H. Nguyen, L. Saulnier, S. Tan, P. O. Suarez, V. Sanh, H. Laurençon, Y. Jernite, J. Launay, M. Mitchell, and C. Raffel, "Bloom: A 176b-parameter open-access multilingual language model," *arXiv preprint*, 2022.

[6] I. Trummer, "The case for nlp-enhanced database tuning: towards tuning tools that" read the manual"," in *VLDB*, 2021.

[7] R. C. Fernandez, A. J. Elmore, M. J. Franklin, S. Krishnan, and C. Tan, "How large language models will disrupt data management," in *VLDB*, 2023.

[8] L. Ouyang, J. Wu, X. Jiang, D. Almeida, C. Wainwright, P. Mishkin, C. Zhang, S. Agarwal, K. Slama, A. Ray, J. Schulman, J. Hilton, F. Kelton, L. Miller, M. Simens, A. Askell, P. Welinder, P. F. Christiano, J. Leike, and R. Lowe, "Training language models to follow instructions with human feedback," in *NeurIPS*, 2022.

[9] H. Touvron, L. Martin, K. Stone, P. Albert, A. Almahairi, Y. Babaei, N. Bashlykov, S. Batra, P. Bhargava, S. Bhosale, D. Bikel, L. Blecher, C. C. Ferrer, M. Chen, G. Cucurull, D. Esiobu, J. Fernandes, J. Fu, W. Fu, B. Fuller, C. Gao, V. Goswami, N. Goyal, A. Hartshorn, S. Hosseini, R. Hou, H. Inan, M. Kardas, V. Kerkez, M. Khabsa, I. Kloumann, A. Korenev, P. S. Koura, M.-A. Lachaux, T. Lavril, J. Lee, D. Liskovich, Y. Lu, Y. Mao, X. Martinet, T. Mihaylov, P. Mishra, I. Molybog, Y. Nie, A. Poulton, J. Reizenstein, R. Rungta, K. Saladi, A. Schelten, R. Silva, E. M. Smith, R. Subramanian, X. E. Tan, B. Tang, R. Taylor, A. Williams, J. X. Kuan, P. Xu, Z. Yan, I. Zarov, Y. Zhang, A. Fan, M. Kambadur, S. Narang, A. Rodriguez, R. Stojnic, S. Edunov, and T. Scialom, "Llama 2: Open foundation and fine-tuned chat models," *arXiv preprint*, 2023.

[10] LlamaTeam, "Thellama3herdofmodels," https://ai.meta.com/research/publications/the-llama-3-herd-of-models/, 2024.

[11] NVIDIA, "Nvidia h200 tensor core gpu," https://www.nvidia.com/en-us/data-center/h200/, 2023.

[12] ——, "Nvidia dgx platform," https://www.nvidia.com/en-us/data-center/dgx-platform/, 2023.

[13] 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," in *VLDB*, 2022.

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

[15] 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 *OSDI*, 2022.

[16] 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," in *VLDB*, 2022.

[17] M. Li, D. G. Andersen, J. W. Park, A. J. Smola, A. Ahmed, V. Josifovski, J. Long, E. J. Shekita, and B.-Y. Su, "Scaling distributed machine learning with the parameter server," in *OSDI*, 2014.

[18] T. Um, B. Oh, B. Seo, M. Kweun, G. Kim, and W.-Y. Lee, "Fastflow: Accelerating deep learning model training with smart offloading of input data pipeline," in *VLDB*, 2023.

[19] X. Miao, Y. Shi, Z. Yang, B. Cui, and Z. Jia, "Sdpipe: A semi-decentralized framework for heterogeneity-aware pipeline-parallel training," in *VLDB*, 2023.

[20] X. Nie, X. Miao, Z. Wang, Z. Yang, J. Xue, L. Ma, G. Cao, and B. Cui, "Flexmoe: Scaling large-scale sparse pre-trained model training via dynamic device placement," *PACMMOD*, 2023.

[21] Y. Guo, Z. Zhang, J. Jiang, W. Wu, C. Zhang, B. Cui, and J. Li, "Model averaging in distributed machine learning: a case study with apache spark," *VLDBJ*, 2021.

[22] Y. Huang, T. Jin, Y. Wu, Z. Cai, X. Yan, F. Yang, J. Li, Y. Guo, and J. Cheng, "Flexps: Flexible parallelism control in parameter server architecture," in *VLDB*, 2018.

[23] J. Jiang, B. Cui, C. Zhang, and L. Yu, "Heterogeneity-aware distributed parameter servers," in *SIGMOD*, 2017.

[24] J. Jiang, F. Fu, T. Yang, and B. Cui, "Sketchml: Accelerating distributed machine learning with data sketches," in *SIGMOD*, 2018.

[25] X. Miao, X. Nie, Y. Shao, Z. Yang, J. Jiang, L. Ma, and B. Cui, "Heterogeneity-aware distributed machine learning training via partial reduce," in *SIGMOD*, 2021.

[26] S. S. Sandha, W. Cabrera, M. Al-Kateb, S. Nair, and M. Srivastava, "In-database distributed machine learning: demonstration using teradata sql engine," in *VLDB*, 2019.

[27] Y. Zhang, F. Mcquillan, N. Jayaram, N. Kak, E. Khanna, O. Kislal, D. Valdano, and A. Kumar, "Distributed deep learning on data systems: a comparative analysis of approaches," in *VLDB*, 2021.

[28] N. Band, "Memflow: Memory-aware distributed deep learning," in *SIGMOD*, 2020.

[29] K. Nagrecha, "Model-parallel model selection for deep learning systems," in *SIGMOD*, 2021.

[30] X. Miao, Z. Jia, and B. Cui, "Demystifying data management for large language models," in *SIGMOD*, 2024.

[31] 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," in *VLDB*, 2023.

[32] 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 *ASPLOS*, 2020.

[33] 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 *HPCA*, 2023.

[34] 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 *ATC*, 2021.

[35] 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 *MICRO*, 2016.

[36] 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 *PPoPP*, 2018.

[37] 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 *FAST*, 2021.

[38] A. Q. Jiang, A. Sablayrolles, A. Mensch, C. Bamford, D. S. Chaplot, D. de las Casas, F. Bressand, G. Lengyel, G. Lample, L. Saulnier, L. R. Lavaud, M.-A. Lachaux, P. Stock, T. L. Scao, T. Lavril, T. Wang, T. Lacroix, and W. E. Sayed, "Mistral 7b," *arXiv preprint*, 2023.

[39] S. Rajbhandari, O. Ruwase, J. Rasley, S. Smith, and Y. He, "Zero-infinity: Breaking the gpu memory wall for extreme scale deep learning," in *SC*, 2021.

[40] Z. Bian, H. Liu, B. Wang, H. Huang, Y. Li, C. Wang, F. Cui, and Y. You, "Colossal-ai: A unified deep learning system for large-scale parallel training," in *ICPP*, 2023.

[41] G. Haas and V. Leis, "What modern nvme storage can do, and how to exploit it: High-performance i/o for high-performance storage engines," in *VLDB*, 2023.

[42] A. Lerner and P. Bonnet, "Not your grandpa's ssd: The era of co-designed storage devices," in *SIGMOD*, 2021.

[43] I. Petrov, G. G. Almeida, A. P. Buchmann, and U. Gräf, "Building large storage based on flash disks," in *ADMS@VLDB*, 2010.

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

[45] 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 *MICRO*, 2023.

[46] NVIDIA, "Nvidia gpudirect: Enhancing data movement and access for gpus," https://developer.nvidia.com/gpudirect, 2011.

[47] 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 Autodiff Workshop*, 2017.

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

[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 *SC*, 2022.

[50] T. Eicken, D. Culler, S. Goldstein, and K. Schauser, "Active messages: A mechanism for integrated communication and computation," in *ISCA*, 1992.

[51] J. Fang and Y. You, "Meet gemini: The heterogeneous memory manager of colossal-ai," https://colossalai.org/docs/advanced_tutorials/meet_gemini/, 2022.

[52] 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," *TPDS*, 2022.

[53] 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," in *MLSys*, 2020.

[54] W. Peebles and S. Xie, "Scalable diffusion models with transformers," in *ICCV*, 2023.

[55] A. Dosovitskiy, L. Beyer, A. Kolesnikov, D. Weissenborn, X. Zhai, T. Unterthiner, M. Dehghani, M. Minderer, G. Heigold, S. Gelly, J. Uszkoreit, and N. Houlsby, "An image is worth 16x16 words: Transformers for image recognition at scale," in *ICLR*, 2021.

[56] Z. Liu, Y. Lin, Y. Cao, H. Hu, Y. Wei, Z. Zhang, S. Lin, and B. Guo, "Swin transformer: Hierarchical vision transformer using shifted windows," in *ICCV*, 2021.

[57] M. D. M. Reddy, M. S. M. Basha, M. M. C. Hari, and M. N. Penchalaiah, "Dall-e: Creating images from text," *UGC Care Group I Journal*, 2021.

[58] C. Jin and S. Xie, "Fast-dit: Fast diffusion models with transformers," https://github.com/chuanyangjin/fast-DiT, 2024.

[59] Y. Feng, M. Xie, Z. Tian, S. Wang, Y. Lu, and J. Shu, "Mobius: Fine tuning large-scale models on commodity gpu servers," in *ASPLOS*, 2023.

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

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

[62] 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 *SC*, 2021.

[63] A. Kroviakov, P. Kurapov, C. Anneser, and J. Giceva, "Heterogeneous intra-pipeline device-parallel aggregations," in *DaMoN*, 2024.

[64] A. Lerner and G. Alonso, "Data flow architectures for data processing on modern hardware," in *ICDE*, 2024.

[65] F. Maschi and G. Alonso, "The difficult balance between modern hardware and conventional cpus," in *DaMoN*, 2023.

[66] G. Alonso, N. Ailamaki, S. Krishnamurthy, S. Madden, S. Sivasubramanian, and R. Ramakrishnan, "Future of database system architectures," in *SIGMOD*, 2023.

[67] L. Von Merzljak, P. Fent, T. Neumann, and J. Giceva, "What are you waiting for? use coroutines for asynchronous i/o to hide i/o latencies and maximize the read bandwidth!" in *ADMS@VLDB*, 2022.

[68] J. Wei and X. Zhang, "How much storage do we need for high performance server," in *ICDE*, 2022.

[69] Y. Lv, B. Cui, B. He, and X. Chen, "Operation-aware buffer management in flash-based systems," in *SIGMOD*, 2011.

[70] J. Do, D. Zhang, J. M. Patel, and D. J. DeWitt, "Fast peak-to-peak behavior with ssd buffer pool," in *ICDE*, 2013.

[71] B. Lee, M. An, and S.-W. Lee, "Lru-c: Parallelizing database i/os for flash ssds," in *VLDB*, 2023.

[72] T. I. Papon and M. Athanassoulis, "Aceing the bufferpool management paradigm for modern storage devices," in *ICDE*, 2023.

[73] J. Li, H.-W. Tseng, C. Lin, Y. Papakonstantinou, and S. Swanson, "Hippogriffdb: Balancing i/o and gpu bandwidth in big data analytics," in *VLDB*, 2016.

[74] R. Thonangi and J. Yang, "On log-structured merge for solid-state drives," in *ICDE*, 2017.

[75] Z. Wang, L. Shou, K. Chen, and X. Zhou, "Bushstore: Efficient b+ tree group indexing for lsm-tree in non-volatile memory," in *ICDE*, 2024.

[76] J. Chu, Y. Tu, Y. Zhang, and C. Weng, "Latte: A native table engine on nvme storage," in *ICDE*, 2020.

[77] M. Haubenschild, C. Sauer, T. Neumann, and V. Leis, "Rethinking logging, checkpoints, and recovery for high-performance storage engines," in *SIGMOD*, 2020.

[78] T. Ziegler, C. Binnig, and V. Leis, "Scalestore: A fast and cost-efficient storage engine using dram, nvme, and rdma," in *ICDE*, 2022.

[79] X. Fan, S. Yan, Y. Huang, and C. Weng, "Tengine: A native distributed table storage engine," in *ICDE*, 2024.

[80] S. Shedge, N. Sharma, A. Agarwal, M. Abouzour, and G. Aluç, "An extended ssd-based cache for efficient object store access in sap iq," in *ICDE*, 2022.

[81] Y. Chai, Y. Chai, X. Wang, H. Wei, N. Bao, and Y. Liang, "Ldc: a lower-level driven compaction method to optimize ssd-oriented key-value stores," in *ICDE*, 2019.

[82] C. Duffy, J. Shim, S.-H. Kim, and J.-S. Kim, "Dotori: A key-value ssd based kv store," in *VLDB*, 2023.

[83] J. Do, I. L. Picoli, D. Lomet, and P. Bonnet, "Better database cost/performance via batched i/o on programmable ssd," *VLDBJ*, 2021.

[84] Y. Wang, J. Yuan, S. Wu, H. Liu, J. Chen, C. Ma, and J. Qin, "Leaderkv: Improving read performance of kv stores via learned index and decoupled kv table," in *ICDE*, 2024.

[85] Y. Wang, J. He, K. Sun, Y. Dong, J. Chen, C. Ma, A. C. Zhou, and R. Mao, "Boosting write performance of kv stores: An nvm-enabled storage collaboration approach," in *ICDE*, 2024.

[86] Y. Huang, X. Fan, S. Yan, and C. Weng, "Neos: A nvme-gpus direct vector service buffer in user space," in *ICDE*, 2024.

[87] T. I. Papon, "Enhancing data systems performance by exploiting ssd concurrency & asymmetry," in *ICDE*, 2024.

[88] Y. Park, S. Min, and J. W. Lee, "Ginex: Ssd-enabled billion-scale graph neural network training on a single machine via provably optimal in-memory caching," in *VLDB*, 2022.

[89] M.-S. Kim, K. An, H. Park, H. Seo, and J. Kim, "Gts: A fast and scalable graph processing method based on streaming topology to gpus," in *SIGMOD*, 2016.

[90] J. B. Park, V. S. Mailthody, Z. Qureshi, and W.-m. Hwu, "Accelerating sampling and aggregation operations in gnn frameworks with gpu initiated direct storage accesses," in *VLDB*, 2024.

[91] J. Sun, Z. Shi, L. Su, W. Shen, Z. Wang, Y. Li, W. Yu, W. Lin, F. Wu, J. Zhou, and B. He, "Helios: Efficient distributed dynamic graph sampling for online gnn inference," in *PPoPP*, 2025.

[92] J. Sun, M. Sun, Z. Zhang, J. Xie, Z. Shi, Z. Yang, J. Zhang, F. Wu, and Z. Wang, "Hyperion: Optimizing ssd access is all you need to enable cost-efficient out-of-core gnn training," in *ICDE*, 2025.

[93] M. Zhang, J. Sun, Q. Hu, P. Sun, Z. Wang, Y. Wen, and T. Zhang, "Torchgt: A holistic system for large-scale graph transformer training," in *SC*, 2024.

[94] J. Sun, L. Su, Z. Shi, W. Shen, Z. Wang, L. Wang, J. Zhang, Y. Li, W. Yu, J. Zhou, and F. Wu, "Legion: Automatically pushing the envelope of Multi-GPU system for Billion-Scale GNN training," in *ATC*, 2023.

[95] J. Wang, C. Lin, Y. Papakonstantinou, and S. Swanson, "Evaluating list intersection on ssds for parallel i/o skipping," in *ICDE*, 2021.

[96] J. Feng and D. Huang, "Optimal gradient checkpoint search for arbitrary computation graphs," in *CVPR*, 2021.

[97] A. Gruslys, R. Munos, I. Danihelka, M. Lanctot, and A. Graves, "Memory-efficient backpropagation through time," in *NIPS*, 2016.

[98] 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," *TOMS*, 2024.

[99] O. Beaumont, J. Herrmann, G. Pallez, and A. Shilova, "Optimal memory-aware backpropagation of deep join networks," *Philosophical Transactions of the Royal Society A*, 2020.

[100] M. Kusumoto, T. Inoue, G. Watanabe, T. Akiba, and M. Koyama, "A graph theoretic framework of recomputation algorithms for memory-efficient backpropagation," in *NeurIPS*, 2019.

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

[102] O. Beaumont, L. Eyraud-Dubois, and A. Shilova, "Efficient combination of rematerialization and offloading for training dnns," in *NeurIPS*, 2021.

[103] Z. Zong, L. Lin, L. Lin, L. Wen, and Y. Sun, "Str: Hybrid tensor re-generation to break memory wall for dnn training," *TPDS*, 2023.

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

[105] 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," *TACO*, 2018.

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

[107] S. Shriram, A. Garg, and P. Kulkarni, "Dynamic memory management for gpu-based training of deep neural networks," in *IPDPS*, 2019.

[108] O. Beaumont, L. Eyraud-Dubois, and A. Shilova, "Optimal gpu-cpu offloading strategies for deep neural network training," in *Euro-Par*, 2020.

[109] 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 *HPCA*, 2021.

[110] 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 *HPCA*, 2023.

[111] J. Jung, J. Kim, and J. Lee, "Deepum: Tensor migration and prefetching in unified memory," in *ASPLOS*, 2023.

[112] C.-C. Huang, G. Jin, and J. Li, "Swapadvisor: Pushing deep learning beyond the gpu memory limit via smart swapping," in *ASPLOS*, 2020.

[113] X. Nie, X. Miao, Z. Yang, and B. Cui, "Tsplit: Fine-grained gpu memory management for efficient dnn training via tensor splitting," in *ICDE*, 2022.

[114] 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 *ICML*, 2022.

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

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