Practical Offloading for Fine-Tuning LLM on Commodity GPU via Learned Sparse Projectors

Siyuan Chen[1*], **Zhuofeng Wang**[2], **Zelong Guan**[1], **Yudong Liu**[1], **Phillip B. Gibbons**[1]

[1] Carnegie Mellon University,
[2] Peking University
siyuanc3@andrew.cmu.edu, wangzf2003@stu.pku.edu.cn, zelongg@andrew.cmu.edu, yudongltech@gmail.com, gibbons@cs.cmu.edu

Abstract

Fine-tuning large language models (LLMs) requires significant memory, often exceeding the capacity of a single GPU. A common solution to this memory challenge is offloading compute and data from the GPU to the CPU. However, this approach is hampered by the limited bandwidth of commodity hardware, which constrains communication between the CPU and GPU, and by slower matrix multiplications on the CPU.

In this paper, we present an offloading framework, *LSP-Offload*, that enables near-native speed LLM fine-tuning on commodity hardware through *learned sparse projectors*. Our data-driven approach involves learning efficient sparse compressors that minimize communication with minimal precision loss. Additionally, we introduce a novel layer-wise communication schedule to maximize parallelism between communication and computation. As a result, our framework can fine-tune a 1.3 billion parameter model on a 4GB laptop GPU and a 6.7 billion parameter model on a 24GB NVIDIA RTX 4090 GPU. Compared to state-of-the-art offloading frameworks, our approach reduces end-to-end fine-tuning time by 33.1%-62.5% when converging to the same accuracy.

Code — https://github.com/gulang2019/LSP-Offload

1 Introduction

Recent years have highlighted the remarkable success of billion scale LLMs. Hand-to-hand with task performance improvements are the ever-growing model sizes and the strong demand for powerful computing resources that are available only in high-end clusters. Fortunately, fine-tuning provides everyday ML practitioners the accessibility to LLMs by allowing them to adapt a pre-trained model to downstream tasks using less onerous computational effort. However, fine-tuning's memory and compute demands are still daunting. For example, under a default fine-tuning configuration that uses the fp16 data type with the Adam optimizer (Kingma and Ba 2014), the memory footprint is $8 \times$ #Parameters bytes, which means top-notch commodity workstation GPUs (e.g., NVIDIA 4090 GPU and AMD 7900XTX with 24GB memory each) are able to hold only smaller LLMs (3B parameters). With commodity laptop GPUs (e.g., NVIDIA A1000 with 4GB memory), even 0.77B parameter LLMs do not fit.

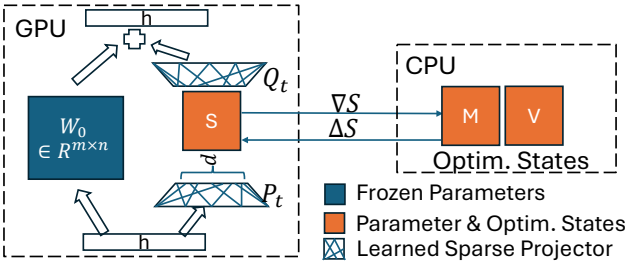


Figure 1: LSP-Offload

A variety of techniques have been proposed to reduce the memory demand during fine-tuning. A typical solution from system researchers is to offload part of the compute and memory from GPU to CPU, leveraging the fact that commodity laptop CPUs typically have 4x the memory of laptop GPUs and commodity workstation CPUs can provide 4TBs of memory (per socket). Although offloading is able to scale the trainable model size, large batch sizes are essential to remain efficient despite the limited PCIe bandwidth between CPU and GPU (Rajbhandari et al. 2021). In fact, we show that training with offloading is inherently bounded by either the CPU-GPU communication or the compute on CPU, especially on commodity hardware where the limited GPU memory dictates small batch sizes. Therefore, offloading itself can hardly save us from the scaling challenge.

Meanwhile, another promising method from ML researchers for memory-reduction is parameter-efficient fine-tuning (PEFT). The key idea of PEFT is to limit the trainable parameters to a carefully designed subspace (e.g., a low rank subspace (Hu et al. 2021; Zhao et al. 2024) or only part of the model (Guo, Rush, and Kim 2020)), so the GPU can train the model without offloading as long as it can hold the parameters and minimal optimizer states for the trainable parameters. However, though more memory-efficient, PEFT methods can suffer from slow convergence or sub-optimal training results due to their overly constrained space for parameter updates.

In this paper, we show how to mitigate the memory challenge by combining both approaches. We present LSP-Offload (Fig. 1), a novel fine-tuning framework that (i) mitigates bottlenecks in offloading approaches by a new approach to refactor the offloading process and (ii) trains efficiently by

*Correspondence Author.

a new approach to constrain the optimization space.

Specifically, to alleviate the compute pressure on the CPU as well as the communication overhead back-and-forth between CPU and GPU, we constrain the updates to happen on a periodically-changing subspace (S in Fig. 1). Because the updates from different subspaces are projected back and accumulate together in the original space, the model is able to update in the full-rank optimization space. State-of-the-art (SOTA) approaches (Hu et al. 2021; Zhao et al. 2024) for constraining the parameter update space suffer from linear memory and compute complexity that limits them from optimizing in large subspaces. We solve this problem by the introduction of (d, r)-*sparse projectors* (P_t and Q_t in Fig. 1), sparse embedding matrices that represent a subspace but whose memory consumption is independent of the subspace's size. In this way, given the same memory budget as PEFT, we are able to optimize in an arbitrary-size subspace. To further improve the compression quality of the subspace, we adopt a data-driven approach similar to (Liu et al. 2020) that adapts the subspace to the gradient matrices, which is empirically proven necessary for fast convergence.

Moreover, at the system level, we demonstrate that the SOTA offloading framework *Zero-Offload* (Rajbhandari et al. 2020) suffers from limited parallelism between communication and compute when running on commodity hardware. This is due to the limited GPU memory relative to the model size, which implies that only small batch sizes can be used during training. We improve Zero's schedule by performing fine-grained communication on the granularity of layers and communicating components of the gradient ahead of time. The new schedule enables us to explore the full parallelism between CPU compute, GPU compute, CPU-to-GPU communication, and GPU-to-CPU communication.

In summary, our paper makes the following contributions:

- We analyze LLM training on commodity hardware (both laptop and workstation) to show that current offloading workflows are fundamentally bounded by either the communication or the CPU's compute.
- We design LSP-Offload to enable near-native speed fine-tuning on commodity hardware. The system is built on the key idea of *learned sparse projectors*, which enables fine-tuning on high-dimensional subspaces with constant memory and compute overhead. We open source our framework at https://github.com/gulang2019/LSP-Offload.
- We verify that LSP-Offload converges to the same accuracy as native training on the GLUE dataset. For instruction-tuning, LSP-Offload reduces end-to-end fine-tuning time by 33.1% to 62.5% over SOTA offloading, when converging to the same accuracy. Moreover, LSP-Offload improves accuracy by 27.8% to 30% over SOTA PEFT approaches on the Alpaca and Humaneval datasets.

2 Background and Related Work

Memory breakdown for training large language models. Training a deep learning model requires memory for parameters, activations, and optimizer states. Activations include intermediate results used in backward propagation. The optimizer states are used by the optimizer to update the parameters. Of the three, the memory for parameters (M_{param}) and optimizer states (M_{opt}) consume most of the memory. When trained with the Adam optimizer and half precision, $M_{param} + M_{opt} \approx 8 \times$ #Parameters bytes, which easily exceeds the single GPU's memory for billion-scale models.

Memory offloading. Memory offloading techniques (Huang, Jin, and Li 2020; Rajbhandari et al. 2020, 2021; Ren et al. 2021; Zhang et al. 2023) enable training a full model on limited GPU memory by utilizing CPU memory or SSDs. Among these, Zero series are the SOTA approaches for fine-tuning large models. Zero-Offload (Ren et al. 2021) offloads the optimizer states and the update step to the CPU. Compared to other approaches that offload only the memory to CPU and do all computations on GPU, Zero-Offload achieves the optimal communication volume for full parameter training. Nevertheless, we found that Zero's training is severely bottlenecked by the communication (see Fig. 2).

Parameter-efficient fine-tuning. PEFT enables pre-trained models to rapidly adapt to downstream tasks with minimal extra memory required. LoRA (Hu et al. 2021) is among the most popular PEFT techniques by constraining the optimization onto a decomposed low-rank subspace. However, recent works (Lialin et al. 2023; Valipour et al. 2022) found LoRA is sensitive to hyperparameter tuning and can struggle with tasks requiring significant change to the base model. To break the low-dimensional constraint of LoRA, GaLore (Zhao et al. 2024) explores a similar idea to ours that periodically changes the subspace computed by singular-value-decomposition (SVD). However, both LoRA and GaLore have the limitation that their extra memory and compute requirements scale linearly with the subspace's size (rank), which inherently prevent them from tuning on a higher dimensional subspace. Our work mitigates this problem via novel subspace projectors whose compute and memory demands are independent of the subspace size, enabling us to achieve better model accuracy by tuning in a larger subspace. Moreover, a contemporary work (He et al. 2024) explores the similar idea to reduce memory overhead via PEFT but using a sparse matrix approach.

Other methods for memory-efficient training. Various approaches such as quantization (Dettmers et al. 2024) and gradient checkpointing (Chen et al. 2016) have been proposed to reduce the memory demand for training/fine-tuning LLMs. The quantization approach uses data types with fewer bits for training, and is fully compatible with our techniques (we use fp16 in our evaluations). Meanwhile, the gradient checkpointing technique trades computation for memory by recomputing activations during the backward pass. We include this technique in our implementation.

3 Motivation

Numerical Analysis for Fine-tuning on a GPU

We motivate our work by an analysis on the fundamental limits of vanilla offloading on a single commodity GPU. We use the example setting of fine-tuning a llama-7B model on a

Parameters	Optimizer State	Activations	CPU-GPU Bandwidth	#Layers	GPU Memory
14GB	42GB	8GB	10–20GB/s	32	24GB
FWD on CPU	BWD on CPU	UPD on CPU	FWD on GPU	BWD on GPU	UPD on GPU
1.61s/layer	3.30s/layer	0.06s/layer	12.2ms/layer	28.1ms/layer	1ms/layer

Table 1: Configurations and timings for training/fine-tuning the llama-7B Model (using fp16) on commodity workstation hardware—the Nvidia RTX 4090 GPU and AMD Ryzen Threadripper 3970X CPU. For UPD, we measure the fused Adam kernel with thread-level parallelism and SIMD optimizations. Bandwidth is the PCIe bandwidth with a pinned memory buffer.

commodity workstation GPU (Nvidia RTX 4090), which provides only $24/(14+42+8) = 37.5\%$ of the required memory (Tab. 1). (A similar analysis can be done for the GPT2-1.3B model on a commodity laptop GPU (Nvidia A100)–see the full version of our paper (Chen et al. 2024).)

Current offloading techniques can be categorized into two classes: (i) those that offload only memory to the CPU, and (ii) those that offload both memory and compute to the CPU. The first class is represented by (Huang, Jin, and Li 2020; Zhang et al. 2023), which perform all compute on the GPU while swapping in and out of GPU memory on the fly. An example of this type of schedule is shown in Fig. 3c. However, this type of offloading schedule is inherently bounded by the communication under the following observation:

Observation. *Training a model demanding M_{tot} memory on a GPU with only M_{gpu} memory, such that the GPU performs all the computation, requires $\geq M_{tot} - M_{gpu}$ of communication per iteration.*

For our setting, we need 2-4s communication per iteration ($64 - 40$ divided by 10-20). Since the per-iteration training time is 1.32s (32 layers $\times(.0122 + .0281 + .001)$), this adds 0.5x to 2.0x overhead compared to the GPU compute even if compute and communication are fully overlapped.

The second class divides the workload between the CPU and GPU. Because of the CPU's limited computing power, only the parameter update step (UPD) is suitable to run on the CPU. For example, assigning the FWD+BWD pass of *just one layer* to the CPU directly adds $1.61 + 3.30 = 4.91$s overhead, which is already 3.7x the *per-iteration* GPU compute. Moreover, offloading UPD to the CPU (more specifically, the computation of ΔW to the CPU—applying these deltas to the model parameters remains on the GPU) means that the 42GB optimizer state can reside on the CPU, enabling larger models like llama-7B to fit in the GPU memory.

Offloading UPD to the CPU was first realized in Zero-Offload (Ren et al. 2021), whose schedule is displayed in Fig. 3a. In their schedule, $2M_{param}$ communication happens every iteration (gradients to CPU, deltas to GPU, both having the same size of Parameters), implying the communication time is 1.4-2.8s, which is already up to 2.1x the GPU compute time (1.29s). Moreover, the CPU compute can become the bottleneck for Zero's schedule. For the example setting, UPD on the CPU takes 1.92s per iteration. When there is no overlap between CPU compute and GPU compute (Fig. 3a), this adds 1.5x overhead compared to GPU compute.

This analysis shows that **training with offloading is computationally inefficient on modern commodity hardware due to fundamental bottlenecks in communication and/or CPU compute.** This motivates us to design a lossy (PEFT-style) algorithm for reduced overheads when offloading.

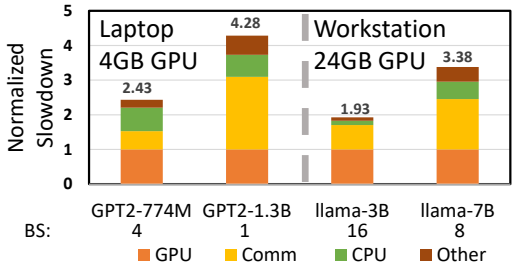


Figure 2: Normalized slowdown of Zero's schedule on laptop and workstation GPUs. The breakdown for communication (Comm) depicts the additional slowdown due to communication that is **not** overlapped with GPU compute. Similarly, the CPU compute and Other are additional non-overlapped overheads. The experiments are done using precision fp16, the largest batch sizes (BS) that fit, and gradient checkpointing.

Case Study on Zero's Schedule

Complementing our analysis, we study Zero-Offload in two settings: (i) training a GPT2 model on a 4GB laptop GPU, and (ii) training a llama model on a 24GB workstation GPU. The slowdown normalized by the GPU compute time is shown in Fig. 2. Under both configurations, Zero's schedule slows training by 1.93x to 4.28x, for the following two reasons.

Communication and CPU compute overhead. The primary source of overhead comes from the unavoidable high communication volume and slow CPU compute as demonstrated in our previous analysis. Shown in Fig. 2, although Zero is able to overlap part of the GPU/CPU compute with communication, the non-overlapped communication brings 0.61x to 2.09x added slowdown compared to the GPU compute time. For both the laptop and workstation GPUs, the situation is worse for the larger model due to the decrease in the largest batch size that fits. When training a 1.3B model on a 4GB GPU, the non-overlapped communication and CPU compute are 2.09x, 0.63x the GPU compute, respectively.

Limited parallelism between CPU and GPU, communication and compute. The second source of overhead comes from Zero's limited parallelism between compute and communication. Fig. 3a shows Zero's standard training pipeline, which is suboptimal for two reasons: (i) FWD and BWD on the GPU are not overlapped with the CPU's compute. This results in significant slowdown when the CPU compute is

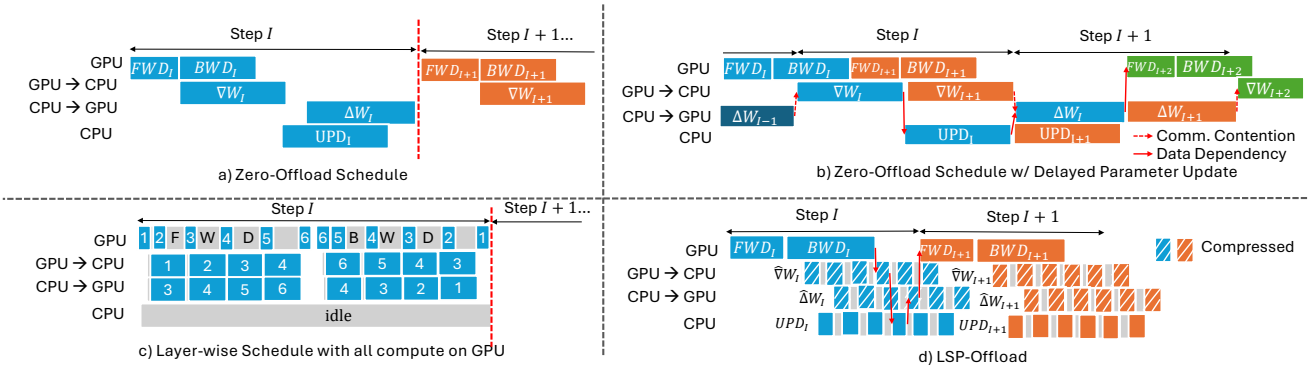


Figure 3: Comparison between current offloading pipelines and LSP-Offload's overlapped pipeline.

around the same scale as the GPU compute. (ii) There is no overlap between the GPU-to-CPU communication and the CPU-to-GPU communication, which implies that the full duplex PCIe channel is at least 50% underutilized.

To mitigate the first issue, Zero proposed delayed parameter updates (Fig. 3b), which use stale parameter values to calculate current gradients, allowing the CPU to perform the previous step's update at the same time the GPU performs the current step's forward and backward passes. Although increasing throughput, this method can affect the accuracy of training. Also, in order not to incur additional memory for buffering communication, the CPU-to-GPU communication and GPU-to-CPU communication cannot be parallelized.

These limitations inspire our design of a layer-wise scheduling strategy that maximizes parallelism between computation and communication. Unlike prior works that focus on parameter pulling or collective communication (Wang, Pi, and Zhou 2019) in distributed training (Lee et al. 2017), our approach applies layer-wise overlapping to offloading, achieving optimal parallelization across CPU and GPU computations and their communications.

4 LSP-Offload's Approach

In this section, we present LSP-Offload, a practical offloading framework for fine-tuning high-quality models efficiently under memory-constrained settings. We will introduce our training algorithm for mitigating the compute and communication overhead, and then illustrate our new schedule design for maximized parallelism in the offloading's schedule.

Efficient and High-quality Offloading via Learned Sparse Projectors

As discussed before, on commodity hardware, the large optimization space combined with limited communication bandwidth causes offloading with a standard training algorithm to result in significant communication and compute overheads. To mitigate this problem, our key insight is to assist the offloading algorithm by using PEFT to configure the size of the optimization subspace, but to do so using novel techniques that avoid the pitfalls of prior PEFT.

	LoRA	GaLore	LSP-Offload
Weight Matrix Trainable Parameters	$W + AB^T$ $A, B \in R^{m\times r, n\times r}$	$W + A_t B_t^T$ $B_t \in R^{n\times r}$	$W + P_t S_t Q_t^T$ $S_t \in R^{d\times d}$
GPU Memory Rank(Optim. Space)	$mn + \beta(m+n)r$ r	$mn + (m+\beta n)r$ $\gamma_1 r\tau$	$mn + (m+n)r$ $\gamma_2 d\tau$

Table 2: Comparison between different fine-tuning approaches, where n, d, r are tensor dimensions satisfying $n \gg d \gg r$. $W \in R^{m\times n}$ is the frozen pre-trained weight matrix. $\beta \geq 1$ is the scale factor for storing the optimizer state ($\beta = 3$ for Adam), τ is the number of updates on the subspace, and $\gamma_1, \gamma_2 \in (0, 1]$ are scaling factors that adjust the rank based on how the individual subspaces interact when added together. LSP-Offload both reduces GPU memory and increases the optimization space rank.

Fig. 1 illustrates our approach. Following previous work (Hu et al. 2021; Zhao et al. 2024), we focus on matrix multiplication operations. Similarly to LoRA and GaLore, we freeze the pre-trained weight matrix and optimize on a decomposed subspace. However, LoRA's and GaLore's extra GPU memory to store the projectors and the optimization states grows *linearly* with the rank of their optimization spaces, preventing them from optimizing in a sufficiently large subspace. E.g., as shown in (Zhao et al. 2024), fine-tuning a 1B model with a hidden size of 2048 on a rank-512 subspace in half precision requires 4.38GB for LoRA and 6.17GB for GaLore, 2.2x and 3.1x the GPU memory needed for just the model.

To overcome this limitation, we made the key innovation to design the projector as sparse matrices, decoupling the dependence between the GPU memory overhead and the rank of the optimization space. Specifically, we use (d, r)-sparse projectors as the template projector (see the properties of this projector in the full version (Chen et al. 2024)).

Definition 1 ((d, r)-Sparse Projector)**.** *We define the projection bases $P \in \mathbb{R}^{m\times d}, Q \in \mathbb{R}^{n\times d}$ as (d, r)-sparse projectors if both P, Q have r nonzero values per row.*

As shown in Fig. 1, by using (d, r)-sparse projectors to replace the dense projectors, we project the weights on a $d \times d$ dimensional subspace. Meanwhile, the sparsity allows us to

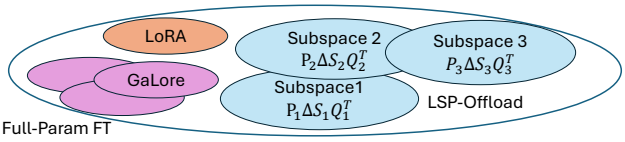


Figure 4: Illustration of the optimization spaces for LoRA, GaLore, and LSP-Offload within the full-parameter space.

store only the $O((m+n)r)$ non-zero values of the projectors on the GPU. This brings LSP-Offload two benefits:

- LSP-Offload is capable of optimizing in a larger subspace while using less GPU memory than SOTA PEFT. For our 2GB model example setting, LSP-Offload requires only 15MB extra GPU memory when using $r = 4$.
- LSP-Offload's optimization space scales linearly with the parameter size. LSP-Offload optimizes in subspaces of size $O(d^2)$, with d set to $n/2$ to hide communication overhead. This results in a scaling of $O(n^2)$ as the model size grows, outperforming LoRA and Galore's $O(n \times r)$ scaling, especially when $n \gg r$ for large models.

In all, the optimization space for a matrix multiplication operation with pre-trained matrix $W_0 \in R^{m\times n}$ constrains as

$$\Delta W = P_1 S_1 Q_1^T + P_2 S_2 Q_2^T + ... + P_\tau S_\tau Q_\tau^T, \quad (1)$$

where $P_t \in R^{m\times d}, Q_t \in R^{n\times d}$ are periodically updated (d, r)-sparse projectors, and $S_t \in R^{d\times d}$ is a dense trainable matrix. As illustrated in Fig. 4, LSP-Offload optimizes in a larger subspace than LoRA and GaLore for the same GPU memory overhead, underscoring LSP-Offload's efficiency.

Training algorithm. The above design leads to the LSP-Offload's core training algorithm listed in Alg. 1. In every iteration, the gradient is projected onto a subspace (line 15) before transferred to the CPU. The weight delta is then computed on CPU by optimizing on the subspace (line 16) before transferred back to GPU and projected to the original space (line 17). This way, both communication and compute complexity for offloading is reduced from $O(m \cdot n)$ to $O(d^2)$, which guarantees our algorithm's efficiency. Moreover, we optionally update the subspace (lines 18-21) by checking its quality. (In the next subsection, the steps are further pipelined between layers to hide latencies.)

Learned sparse projectors. Further, we boost the performance of the sparse projectors with a data-driven approach. Specifically, we initialize the (d, r)-sparse projectors by randomly sampling the r nonzero positions for each row and randomly sampling the nonzero values from $\mathcal{N}(0, 1/\sqrt{r})$. Random sampling ensures an unbiased estimation gradient with good approximation properties, as supported by the JL lemma (Kane and Nelson 2014). After that, we fit the projectors on the calibration dataset to minimize the following estimation bias on the gradient:

Definition 2 (Estimation Bias)**.** *For a (d, r)-sparse projector P, Q and a matrix $\Sigma \in R^{m\times n}$, the estimation bias is $\boldsymbol{b}^{P,Q}(\Sigma) := PP^T\Sigma QQ^T - \Sigma$.*

Algorithm 1: LSP-Offload's fine-tuning with learned sparse projectors [simplified version without layer-wise scheduling]

1: **HyperParam:** d, r: (d, r)-sparse projectors. $CheckFreq, \alpha$: check frequency, threshold for updating projectors.
2: **Function** MAYBEUPDATE(∇_W: the gradient, P_{prev}, Q_{prev}: previous projectors, M, V: optimizer state)
3: **if** $\|\mathbf{b}^{P,Q}(\nabla_W)\|_F / \|\nabla_W\|_F \leq \alpha$ **then**
4: **Return** P_{prev}, Q_{prev}
5: $P, Q \leftarrow Initialize(d, r)$
6: Minimize $loss := \|\mathbf{b}^{P,Q}(\nabla_W)\|_F + \beta \cdot (\|P\|_F^2 + \|Q\|_F)$ until $\|\mathbf{b}^{P,Q}(\nabla_W)\|_F / \|\nabla_W\|_F \leq \alpha$ or Timeout.
7: {Project previous M and V tensors to new subspace}
8: $M \in \mathbb{R}^{d\times d} \leftarrow P^T P_{prev} M Q_{prev}^T Q$
9: $V \in \mathbb{R}^{d\times d} \leftarrow (P^T P_{prev})^2 V (Q_{prev}^T Q)^2$
10: **Return** P, Q
11: **Function** MAIN($\mathcal{M}$: Model, $\mathcal{D}$: Dataset, $W \in \mathbb{R}^{m\times n}$: Weights, $M, V \in \mathbb{R}^{d\times d}$: 1st, 2nd order optimizer state, $P \in \mathbb{R}^{m\times d}, Q \in \mathbb{R}^{n\times d}$ the sparse projectors)
12: **for** $t \leftarrow 1$ to T **do**
13: Sample $x \sim \mathcal{D}$
14: $\nabla_W \leftarrow forwardBackward(\mathcal{M}, x)$ {FWD+BWD on GPU}
15: $grad \leftarrow SendToCPU(P^T \nabla_W Q)$ {Compress on GPU and gradient offload}
16: $\Delta_W \leftarrow SendToGPU(Update(grad))$ {UPD on CPU and delta upload}
17: $W \leftarrow W - \eta_t P \Delta_W Q^T$ {Decompress, apply deltas on GPU}
18: **if** $(t-1) \bmod CheckFreq = 0$ **then**
19: $\nabla_W \leftarrow$ gradient on sampled subset $\mathcal{D}' \subset \mathcal{D}$.
20: $P, Q \leftarrow$ MAYBEUPDATE(∇_W, P, Q, M, V)
21: **end if**
22: **end for**

Particularly, we optimize the following problem for better projectors:

$$\min_{P,Q} \underbrace{\|\mathbf{b}^{P,Q}(\nabla_W)\|_F}_{\text{estimation error of gradient}} + \beta \cdot \underbrace{(\|P\|_F + \|Q\|_F)}_{\text{regularization}} \quad (2)$$

Compared to GaLore, which uses SVD decomposition as the projection matrix, we empirically find that our data-driven approach has a lower generalization error when using the same amount of extra GPU memory (Fig. 6b).

Convergence analysis of Alg. 1. For dataset $\mathcal{D}$, weight matrix $W \in R^{m\times n}$, we consider minimizing $f(W) = \Sigma_{x\sim\mathcal{D}} f_x(W)/|\mathcal{D}|$ using Alg. 1 with $CheckFreq = 1$. That is, $W_{t+1} = W_t - \eta P_t P_t^T \nabla f_{x_t}(W_t) Q_t Q_t^T, t = 1, 2, \ldots, T$, where P_t, Q_t are (d, r)-sparse projectors. We derive the convergence theorem based on L-smooth functions, which indicate convexity and smoothness and are widely used in prior work (Ajalloeian and Stich 2020; Garrigos and Gower 2023).

Assumption 1 (Effectiveness of the subspace)**.** *The relative error on the subspace is kept under α in Alg. 1.*

Assumption 2 (Bounded bias)**.** *There exists $\gamma > 0$, such that for any weight W and $x \sim \mathcal{D}$, $\|\boldsymbol{b}^{P_t,Q_t}(\nabla f_x(W))\| < \gamma, \|\nabla f_x(W)\| < \gamma$.*

Assumption 3 (Sparse bias)**.** *There exists a constant* $0 < c < \frac{1}{\sqrt{2}\alpha}$*, such that* $\|\boldsymbol{b}^{P_t,Q_t}(\nabla f(W))\|_F < c\|\boldsymbol{b}^{P_t,Q_t}(\nabla f(W))\|_2$ *holds for any weight matrix* W*.*

We show the following convergence rate of our algorithm—see the full version (Chen et al. 2024) for the proof. The key idea is that a small gradient estimation error on the full dataset, which drives convergence, can be inferred from a bounded gradient estimation error on sub-sampled datasets.

Theorem 1. *For any* $\beta > 0$ *and* $0 < \delta < 1$*, suppose that* f *is an* L*-smooth function, Assumptions 1, 2, 3 hold and that we check every iteration in Alg. 1 with the subsampled data set* $\mathcal{D}'$ *of size* $\mathcal{O}(\frac{8\gamma^2}{3\beta^2}\log\frac{(m+n)T}{\delta})$*, and stepsize* $\eta = \frac{1}{L}$*. Denote* $F := \mathbb{E}[f(W_0)] - f^*$*. Then with probability* $1-\delta$*,* $T = \mathcal{O}(\frac{1}{\epsilon}) \cdot \frac{LF}{(1-2c^2\alpha^2)}$ *iterations are sufficient to obtain* $\min_{t\in[T]} \mathbb{E}\|\nabla f(W_t)\|^2 = \mathcal{O}(\epsilon + \frac{2c^2\beta^2(1+\alpha)^2}{1-2c^2\alpha^2})$.

Remark 1. *The relative error (*α *in Assumption 1) is critical both for the final accuracy and for the time to convergence.*

Remark 2. *The logarithmic sample efficiency in our optional update indicates low overhead for subsampling* $\mathcal{D}'$*.*

Layer-wise Schedule for Maximal Parallelism

At the system level, we propose a new scheduling approach that addresses both issues in Zero's schedule, based on the observation that *optimization update steps for different layers are independent*. This allows us to overlap GPU computation, CPU-GPU communication in both directions, and parameter updates on the CPU across different layers. The key idea and its benefits are illustrated in Fig. 3d (see the full version (Chen et al. 2024) for pseudocode). We split the GPU-to-CPU, CPU update, and CPU-to-GPU communication into small blocks to unlock the parallelism between layers without the accuracy loss of Zero's use of stale parameter values. We parallelize the CPU's and GPU's compute by executing the deeper layers' update step on CPU while doing the backward pass of shallower layers on GPU. We also parallelize the double-sided communication by executing deeper layer's upload step while doing the shallower layer's offload step. Compared to Zero-Offload, LSP-Offload reduces the CPU's involvement in the critical path from the entire parameter update step to the update for only one layer, a 32x improvement for llama-7B. We show in the full version (Chen et al. 2024) how to avoid a deeper layer's workload from blocking a shallower layer's computation that executes earlier in the next iteration.

5 Evaluation

We first verify the convergence of LSP-Offload on the GLUE dataset and then evaluate the end-to-end training performance on instruction-tuning. Detailed configurations for the experiments are described in the full version (Chen et al. 2024).

Accuracy validation of LSP-Offload on GLUE. Tab. 3 summarizes the accuracy of LSP-Offload for fine-tuning the pre-trained RoBertA-base (Liu et al. 2019) (117M) model on the GLUE dataset (Wang et al. 2018), which is a set of language understanding tasks that is widely adopted to evaluate fine-tuning (Hu et al. 2021; Zhao et al. 2024). We measure the best accuracy within one hour of training for end-to-end comparison given potential overheads. As shown in Tab. 3, LSP-Offload outperforms full parameter tuning by 1.9% accuracy, despite using only 253MB GPU memory vs. 747MB. Furthermore, the full version (Chen et al. 2024) shows that LSP-Offload converges at the same rate as the full parameter tuning. Compared to Galore, LSP-Offload achieves 1.2% higher accuracy. We attribute this to LSP-Offload's larger parameter update space (for the same GPU memory), which is 10x for this experiment.

	MNLI	SST2	MRPC	CoLA	QNLI	QQP	SST2	STS-B	Avg
Full Parameter	81.11	**93.4**	86.6	55.0	90.4	80.8	**93.3**	88.4	83.63
GaLore (Rank=16)	**83.0**	92.0	88.0	56.7	88.1	**85.2**	92.0	90.0	84.38
LSP (d=512, r=16)	81.4	91.7	**91.1**	**61.65**	**91.78**	83.39	92.0	**91.0**	**85.53**

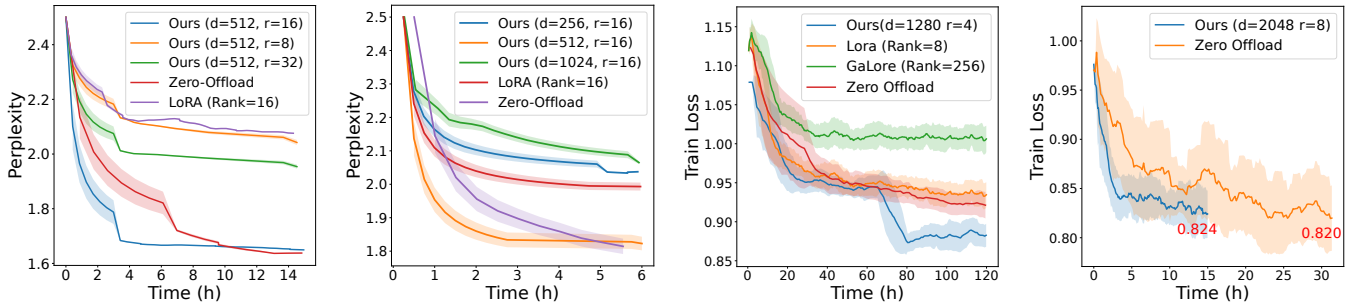
Table 3: Accuracy (%) Comparison after 1 hour fine-tuning the pre-trained RoBertA-base model on GLUE.

	Mem	Time	python	java	cpp	js	ts	php	Avg.
Zero-Offload	3.3	120	**57.93**	37.97	39.75	**52.80**	47.17	**40.99**	45.5
LoRA (Rank=8)	3.6	120	43.29	41.77	35.40	41.61	43.40	31.68	39.3
GaLore (Rank=256)	7.9	120	39.63	36.08	31.68	34.78	40.88	36.02	36.4
LSP (d=1280, r=4)	3.6	120	55.49	**42.41**	**40.99**	50.31	**48.43**	38.51	**45.6**
Zero-Offload	16.8	15	73.78	61.39	**64.60**	66.46	64.15	58.39	64.8
Zero-Offload	16.8	30	**75.00**	**64.56**	61.49	**70.81**	65.41	62.73	**66.7**
LSP (d=2048, r=8)	17.0	15	74.39	62.66	61.49	66.46	**67.30**	**65.84**	66.4

Table 4: Evaluation accuracy (%) on the Humaneval dataset instruction after fine-tuning Deepseek-Coder-1.3B (top) and Deepseek-Coder-6.7b (bottom) with bfloat16 on the laptop GPU (top) and workstation GPU (bottom). Memory is measured in GB of used GPU memory, time is measured in hours.

End-to-end evaluation. Next, we evaluate the end-to-end performance of LSP-Offload for instruction-tuning. We perform our evaluation using four settings: (1) fine-tuning the GPT2-774M model on the Alpaca dataset (Taori et al. 2023) on a laptop with Nvidia A1000 Laptop GPU (4GB) and Intel Core-i7 12800H CPU (32GB), (2) fine-tuning the Llama-3B model on Alpaca on a workstation with Nvidia RTX 4090 GPU (24 GB) and AMD Ryzen Threadripper 3970X CPU (252GB), and (3,4) fine-tuning the Deepseek-Coder-1.3B model (Deepseek-Coder-6.7B model) on an open-source code instruction dataset generated using Wizard-Coder' method (Luo et al. 2023) on the laptop GPU (workstation GPU). We choose the r in LSP-Offload and the ranks in LoRA and GaLore such that they all use similar amounts of memory below the GPU memory capacity.

Comparison with Zero-Offload. Compared to Zero-Offload, LSP-Offload achieves faster convergence while achieving similar convergence accuracy. For the instruction-tuning task, LSP-Offload uses around 62.5% (Fig. 5a) and 33.1% (Fig. 5b) less time when converging to similar accuracy. E.g., when training on the Laptop GPU, LSP-Offload achieves the evaluation perplexity of 1.82 after 2 hours of training, while reaching the same perplexity takes 4.5 hours with Zero-Offload. Moreover, as shown in Fig. 5c and Tab. 4, within the 120 hour training budget, LSP-Offload trains 1.97x more epochs than Zero-Offload, resulting in lower training losses. Similarly, shown in Fig. 5d, for the Deepseek-Coder-



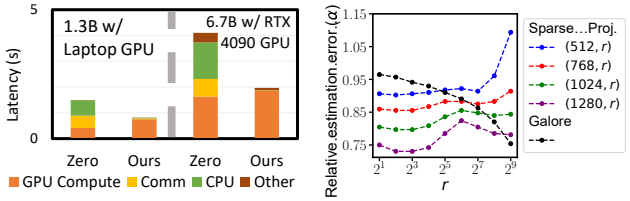
(a) Evaluation perplexity of fine-tuning GPT2-774M w/ the laptop GPU.

(b) Evaluation perplexity of fine-tuning Llama-3B w/ the workstation GPU.

(c) Simulated training loss of fine-tuning Deepseek-Coder-1.3B w/ the laptop GPU.

(d) Simulated training loss of fine-tuning Deepseek-Coder-6.7B w/ workstation GPU for one epoch.

Figure 5: End-to-end evaluation of LSP-Offload. Rolling average is applied. Shading depicts the standard deviation.



(a) Breakdown for training 1 iteration on the coding task.

(b) Estimation bias w/ Deepseek-1.3B Model on validation set.

Figure 6: Analysis on Coding task.

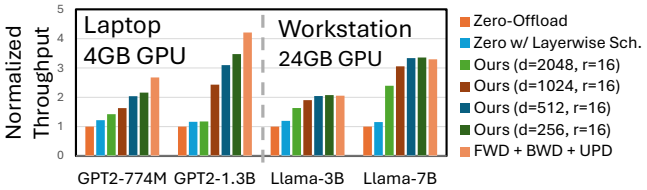


Figure 7: Ablation on training throughput.

6.7B model , LSP-Offload completes the fine-tuning for an epoch 2x faster than Zero-Offload while achieving close accuracy (0.820 vs. 0.824). When trained for 15 hours, LSP-Offload outperforms Zero-Offload on average accuracy by 2.4%.

Comparison with PEFT approaches. LSP-Offload achieves 30% lower evaluation perplexity than LoRA in Alpaca (Fig. 5a), and outperforms GaLore in all coding tasks with 27.8% higher average accuracy on the Humaneval (Chen et al. 2021; Cassano et al. 2023) dataset (Tab. 4), even if GaLore trains 60% more epochs than LSP-Offload.

Training time breakdown. Fig. 6a shows the time breakdown of LSP-Offload for training a single iteration. Compared to Zero-Offload, LSP-Offload cuts 50% the per-iteration latency by reducing the wall-clock time of CPU compute and communication. Because of the layer-wise parallel schedule, the communication and compute on both CPU and GPU are fully in parallel, resulting in minimal non-overlapped overhead for communication and CPU compute.

Hyperparameters. We measured the estimation bias across different configurations on the Deepseek Coding task. As shown in Fig. 6b, larger d with r at 4 or 8 minimizes estimation bias, leading to faster and higher quality convergence. Therefore, it is advisable to set larger ds as long as the communication is hidden from the compute with smaller rs.

Ablation study. Fig. 7 shows an ablation study on training throughput with different techniques. Training throughput is measured by the number of training iterations executed in unit time. First, by adding layer-wise scheduling (blue columns), we improve Zero-Offload (leftmost column) throughput by 18%. After that, we apply LSP-Offload with different configurations. Compared to a native training setup (rightmost column) where only FWD, BWD, and UPD operations are performed on the GPU without CPU computation or communication, LSP-Offload incurs an average slowdown of just 10.6%, 16.7% for subspace sizes of 256, 512 respectively.

6 Limitation

LSP-Offload introduces a list of hyperparameters for the selection of the (d, r)-sparse projector, the frequency of subspace updates, the threshold for these updates, and others. We refer the reader to the full version (Chen et al. 2024) for more empirical insights into their selection.

7 Conclusion

In this paper, inspired by PEFT methods, we developed LSP-Offload to enable near-native speed fine-tuning by constraining parameter updates to a subspace. Using a sparse projector and minimizing empirical bias, LSP-Offload optimizes in larger spaces than GaLore and LoRA for the same GPU memory size. In the GLUE data set, LSP-Offload achieves convergence at the same rate as native training. Compared to zero-offload, it reduces the fine-tuning time by 33. 1% - 62. 5% in instruction-tuning tasks while maintaining accuracy. Furthermore, it improves accuracy by 27. 8% - 30% in the Alpaca and Humaneval datasets over GaLore and LoRA.

Acknowledgments

This work was supported by National Science Foundation grant CNS-2211882 and by the member companies of the Wasm Research Center and PDL consortium. We thank Aashiq Muhamed, David Woodruff, and Dave Anderson for the discussion on this work, as well as anonymous reviewers, for providing valuable feedback.

References

Ajalloeian, A.; and Stich, S. U. 2020. Analysis of SGD with biased gradient estimators. *arXiv preprint arXiv:2008.00051*.

Cassano, F.; Gouwar, J.; Nguyen, D.; Nguyen, S.; Phipps-Costin, L.; Pinckney, D.; Yee, M.-H.; Zi, Y.; Anderson, C. J.; Feldman, M. Q.; Guha, A.; Greenberg, M.; and Jangda, A. 2023. MultiPL-E: A Scalable and Polyglot Approach to Benchmarking Neural Code Generation. *IEEE Transactions on Software Engineering*, 49(7): 3675–3691.

Chen, M.; Tworek, J.; Jun, H.; Yuan, Q.; de Oliveira Pinto, H. P.; Kaplan, J.; Edwards, H.; Burda, Y.; Joseph, N.; Brockman, G.; Ray, A.; Puri, R.; Krueger, G.; Petrov, M.; Khlaaf, H.; Sastry, G.; Mishkin, P.; Chan, B.; Gray, S.; Ryder, N.; Pavlov, M.; Power, A.; Kaiser, L.; Bavarian, M.; Winter, C.; Tillet, P.; Such, F. P.; Cummings, D.; Plappert, M.; Chantzis, F.; Barnes, E.; Herbert-Voss, A.; Guss, W. H.; Nichol, A.; Paino, A.; Tezak, N.; Tang, J.; Babuschkin, I.; Balaji, S.; Jain, S.; Saunders, W.; Hesse, C.; Carr, A. N.; Leike, J.; Achiam, J.; Misra, V.; Morikawa, E.; Radford, A.; Knight, M.; Brundage, M.; Murati, M.; Mayer, K.; Welinder, P.; McGrew, B.; Amodei, D.; McCandlish, S.; Sutskever, I.; and Zaremba, W. 2021. Evaluating Large Language Models Trained on Code. arXiv:2107.03374.

Chen, S.; Guan, Z.; Liu, Y.; and Gibbons, P. B. 2024. Practical Offloading for Fine-Tuning LLM on Commodity GPU via Learned Sparse Projectors. *arXiv preprint arXiv:2406.10181*.

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

Dettmers, T.; Pagnoni, A.; Holtzman, A.; and Zettlemoyer, L. 2024. QLORA: Efficient Finetuning of Quantized LLMs. *Advances in Neural Information Processing Systems*, 36.

Garrigos, G.; and Gower, R. M. 2023. Handbook of convergence theorems for (stochastic) gradient methods. *arXiv preprint arXiv:2301.11235*.

Guo, D.; Rush, A. M.; and Kim, Y. 2020. Parameter-efficient transfer learning with diff pruning. *arXiv preprint arXiv:2012.07463*.

He, H.; Li, J. B.; Jiang, X.; and Miller, H. 2024. Sparse matrix in large language model fine-tuning. *arXiv preprint arXiv:2405.15525*.

Hu, E. J.; Shen, Y.; Wallis, P.; Allen-Zhu, Z.; Li, Y.; Wang, S.; Wang, L.; and Chen, W. 2021. LoRA: Low-rank adaptation of large language models. *arXiv preprint arXiv:2106.09685*.

Huang, C.-C.; Jin, G.; and Li, J. 2020. 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*, 1341–1355.

Kane, D. M.; and Nelson, J. 2014. Sparser johnson-lindenstrauss transforms. *Journal of the ACM*, 61(1): 1–23.

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

Lee, S.; Jha, D.; Agrawal, A.; Choudhary, A.; and Liao, W.-k. 2017. Parallel deep convolutional neural network training by exploiting the overlapping of computation and communication. In *2017 IEEE 24th international conference on high performance computing (HiPC)*, 183–192. IEEE.

Lialin, V.; Muckatira, S.; Shivagunde, N.; and Rumshisky, A. 2023. ReLoRA: High-Rank Training Through Low-Rank Updates. In *Workshop on Advancing Neural Network Training: Computational Efficiency, Scalability, and Resource Optimization (WANT@ NeurIPS 2023)*.

Liu, S.; Liu, T.; Vakilian, A.; Wan, Y.; and Woodruff, D. 2020. A framework for learned CountSketch.

Liu, Y.; Ott, M.; Goyal, N.; Du, J.; Joshi, M.; Chen, D.; Levy, O.; Lewis, M.; Zettlemoyer, L.; and Stoyanov, V. 2019. RoBERTa: A Robustly Optimized BERT Pretraining Approach. *arXiv preprint arXiv:1907.11692*.

Luo, Z.; Xu, C.; Zhao, P.; Sun, Q.; Geng, X.; Hu, W.; Tao, C.; Ma, J.; Lin, Q.; and Jiang, D. 2023. WizardCoder: Empowering Code Large Language Models with Evol-Instruct. *arXiv preprint arXiv:2306.08568*.

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

Rajbhandari, S.; Ruwase, O.; Rasley, J.; Smith, S.; and He, Y. 2021. 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*, 1–14.

Ren, J.; Rajbhandari, S.; Aminabadi, R. Y.; Ruwase, O.; Yang, S.; Zhang, M.; Li, D.; and He, Y. 2021. ZeRO-Offload: Democratizing Billion-Scale Model Training. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, 551–564.

Taori, R.; Gulrajani, I.; Zhang, T.; Dubois, Y.; Li, X.; Guestrin, C.; Liang, P.; and Hashimoto, T. B. 2023. Stanford Alpaca: An Instruction-following LLaMA model. https://github.com/tatsu-lab/stanford_alpaca.

Valipour, M.; Rezagholizadeh, M.; Kobyzev, I.; and Ghodsi, A. 2022. Dylora: Parameter efficient tuning of pre-trained models using dynamic search-free low-rank adaptation. *arXiv preprint arXiv:2210.07558*.

Wang, A.; Singh, A.; Michael, J.; Hill, F.; Levy, O.; and Bowman, S. R. 2018. GLUE: A multi-task benchmark and analysis platform for natural language understanding. *arXiv preprint arXiv:1804.07461*.

Wang, S.; Pi, A.; and Zhou, X. 2019. Scalable distributed DL training: Batching communication and computation. In *Proceedings of the AAAI Conference on Artificial Intelligence*, 5289–5296.

Zhang, H.; Zhou, Y. E.; Xue, Y.; Liu, Y.; and Huang, J. 2023. G10: Enabling An Efficient Unified GPU Memory and

Storage Architecture with Smart Tensor Migrations. *arXiv preprint arXiv:2310.09443*.

Zhao, J.; Zhang, Z.; Chen, B.; Wang, Z.; Anandkumar, A.; and Tian, Y. 2024. GaLore: Memory-efficient LLM training by gradient low-rank projection. *arXiv preprint arXiv:2403.03507*.