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, 2200012827@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 an NVIDIA RTX 4090 GPU with 24GB memory. 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.

Introduction

Recent years have highlighted the remarkable success of billion scale LLMs. Hand-to-hand with task performance improvement 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 common 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.

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.

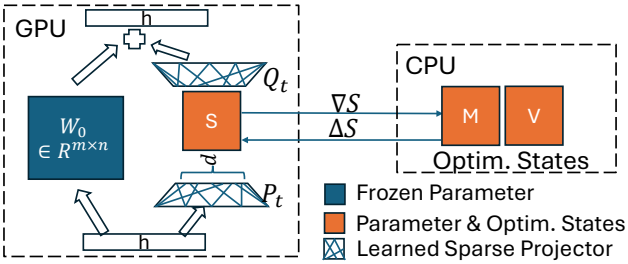


Figure 1: LSP-Offload

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 types of approaches. We present LSP-Offload (Fig. 1), a novel fine-tuning framework that (i) mitigates the bottlenecks in prior offloading approaches via a new approach for refactoring the offloading process and (ii) trains efficiently via a new approach to constraining 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-sparse projectors* (P_t and Q_t in Fig. 1), sparse embedding matrices that represent a subspace but whose size 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 boost 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, on 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 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 allows us to optimize on high-dimensional subspaces with constant memory and compute overhead.
- We verify that LSP-Offload can converge to the same accuracy as native training on the GLUE dataset. Also, on the instruction-tuning task, 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.

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 the intermediate results used in backward propagation. The optimizer states are used by the optimizer to update the parameters. Out of the three, memory for parameters (M_{param}) and for the optimizer state (M_{opt}) consume most of the memory. When trained with 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. These techniques (Zhang et al. 2023; Huang, Jin, and Li 2020; Rajbhandari et al. 2020, 2021; Ren et al. 2021) enable training the full model with inadequate GPU memory by utilizing non-GPU memory such as 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 onto 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). Our work is built on top of the Zero series offloading schedule to make it practical for single GPU training with minimal communication overhead.

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) recently 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 algorithms require extra memory and compute linear 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.

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.

Motivation

Numerical Analysis for Fine-tuning on a GPU

In this section, we motivate our work by analyzing the fundamental limits of vanilla offloading on a single commodity GPU. We use the example setting of training/fine-tuning a llama-7B model on an Nvidia RTX 4090 GPU (a commodity workstation GPU), which can provide only $24/(14 + 42 + 8) = 37.5\%$ of the required memory (Table 1).[1]

[1]A similar analysis, with the same general conclusions, can be done for the GPT2-1.3B model on a commodity laptop GPU, based on Table 5 in the Appendix.

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.

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	1.7ms/layer	3.5ms/layer	1ms/layer

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 type is represented by (Huang, Jin, and Li 2020; Zhang et al. 2023), which perform all compute on the GPU while swapping in and out memory on the fly. An example of this type of schedule is shown in Fig. 3.c. 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 5.33s communication per iteration, which adds 3.2x overhead compared to the GPU compute even if compute and communication are fully overlapped.

The second type of offloading schedule splits the workload across CPU and GPU. Among the forward pass (FWD), backward pass (BWD), and parameter update step (UPD), because of the CPU's limited computing power relative to the GPU, only UPD is suitable to run on the CPU. For example, assigning the FWD+BWD pass of just one layer to the CPU directly adds 4.9s overhead, which is already 3.21x the GPU compute. Moreover, offloading UPD to the CPU[2] 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. 3.a (Alg. 2 in the Appendix). In their schedule, M_{param} communication happens every iteration (gradients to CPU, deltas to GPU), which brings the communication overhead to 0.93s. When there is no overlap between CPU compute and GPU compute (Fig. 3.a), the training slowdown is 2.11x. 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, slowing down training by 2.14x.

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) algorithm for reduced overheads when offloading.

Case Study on Zero's Schedule

Moreover, prior offload schedules are suboptimal. Here we profile Zero-Offload's schedule for a more comprehensive view of its performance. We study two settings for profiling: (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.

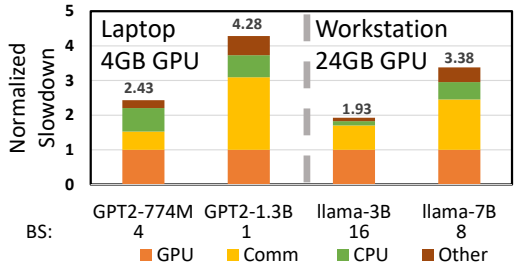


Figure 2: Normalized slowdown of Zero's schedule on laptop and workstation GPUs. The breakdown for communication (Comm) depicts the additional slowdown due to communication that is **not** overlapped with GPU compute. Similarly, the CPU compute and Other are additional non-overlapped overheads. The experiments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

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 because the maximum available batch size decreases. 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. 3.a shows Zero's standard training pipeline, which is sub-optimal 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 around the same scale as the GPU's compute. (ii) No overlap exists between the GPU-to-CPU communication and CPU-to-GPU communication, which implies that the full duplex PCIe channel is at least 50% underutilized. As a result, the per-iteration time for Zero's schedule is

$$T^{iter}_{Zero} = T_{FWD} + \max\{T_{BWD}, T^{\text{GPU-to-CPU}}_{Comm}\} + \max\{T_{UPD}, T^{\text{CPU-to-GPU}}_{Comm}\}. \quad (1)$$

[2]More specifically, the computation of ΔW to the CPU—applying these deltas to the model parameters remains on the GPU.

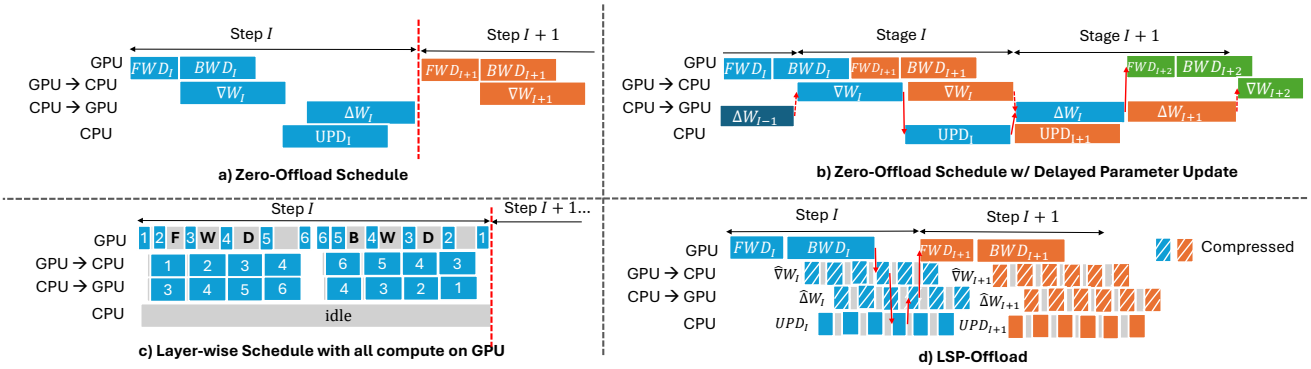


Figure 3: Comparison between current offloading pipelines and LSP-Offload's overlapped pipeline.

To mitigate the first issue, Zero proposed delayed parameter updates (Fig. 3.b), 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 hurt training accuracy. 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 motivates our design for a layer-wise schedule that enables maximal parallelism between CPU compute, GPU compute, CPU-to-GPU communication and GPU-to-CPU communication.

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 in the motivation section, 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.

Fig. 1 illustrates our approach. Following previous works (Hu et al. 2021; Zhao et al. 2024), we focus on matrix multiplication operations. Similar to LoRA and GaLore, we freeze the pre-trained weight matrix and optimize on a decomposed subspace. However, the rank of LoRA's and GaLore's optimization space is linearly growing with the extra GPU memory needed to store the projectors and the optimization states, 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, adding 119% and 208% GPU memory overhead compared to storing only the pre-trained model.

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.

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

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 Appendix).

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 non-zero 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 store only the $O((m+n)r)$ non-zero values of the projectors on the GPU. As a result, shown in Table 2, **LSP-Offload is able to optimize in a larger subspace while using less GPU memory than SOTA PEFT**. For our example setting, LSP-Offload only 2.015GB GPU memory when using $r = 4$.

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

$$\Delta W = P_1 S_1 Q_1^T + P_2 S_2 Q_2^T + ... + P_\tau S_\tau Q_\tau^T, \quad (2)$$

where $P_t, Q_t \in R^{n\times d}$ are periodically updated (d, r)-sparse projectors, and $S_t \in R^{d\times d}$ is a dense trainable matrix.

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

1: **HyperParam:** s: subspace size. d, r: d, r-sparse projectors. $\gamma \in \mathbb{R}^+$. $CheckFreq, \alpha$: check frequency and 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: $M \in \mathbb{R}^{s \times s} \leftarrow P^T P_{prev} M Q_{prev}^T Q$
9: $V \in \mathbb{R}^{s \times s} \leftarrow (P^T P_{prev})^2 V (Q_{prev}^T Q)^2$
10: **Return** P, Q
11: **Function** MAIN($\mathcal{M}$: Model, $\mathcal{D}$: Dataset, $W \in \mathbb{R}^{m \times n}$: Weights, $M, V \in \mathbb{R}^{s \times s}$: 1st, 2nd order optimizer state)
12: **for** $t \leftarrow 0$ to $\tau - 1$ **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** $t \bmod CheckFreq = 0$ **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**

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 section, the steps are further pipelined between layers to hide the latency. Next, we introduce several techniques to boost the training quality.

Learned sparse projectors. First, we boost the performance of the sparse projectors by a data-driven approach, which is a key contribution. Specifically, we initialize (d, r)-sparse projectors by randomly sampling the r non-zero positions for each row, and randomly sampling the non-zero values from $\mathcal{N}(0, 1/\sqrt{r})$. After that, we fit the projectors on the calibrated 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* $\boldsymbol{b}^{P,Q}(\Sigma) := PP^T \Sigma QQ^T - \Sigma$.

Denote the forward pass of the matrix multiplication operation as $Wx = (W_0 + PSQ^T)x$. We optimize the following problem for better projectors:

$$\min_{P,Q} \underbrace{\|\mathbf{b}^{P,Q}(\nabla_W)\|_F}_{\text{estimation error of gradient}} + \beta \cdot \underbrace{(\|P\|_F + \|Q\|_F)}_{\text{regularization}} \quad (3)$$

Compared to GaLore, which uses SVD decomposition as the projection matrix, we empirically find our data-driven approach has lower generalization error when using the same amount of extra GPU memory (Fig. 8 in the Appendix).

Updating the subspace. Secondly, we avoid the overhead of frequently training the projectors by optionally updating the subspace. Specifically, on a sub-sampled dataset, only when the gradient estimation bias exceeds a certain threshold α (line 3), do we switch to a new (learned) projector.

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 $CheckFreq = 1$. Namely, $W_{t+1} = W_t - \eta P_t P_t^T \nabla f_{x_t}(W_t) Q_t Q_t^T, t = 1, 2, ..., T$, where P_t, Q_t are (d, r)-sparse projectors. We make following three assumptions.

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 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 matrices W.*

We show the following convergence rate of our algorithm—please see the Appendix 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 the sub-sampled dataset.

Theorem 1. *For any $\beta > 0, 0 < \delta < 1$, suppose f is a L-smooth function, Assumptions 1, 2, 3 hold, and that we check every iteration in Alg. 1 with the sub-sampled dataset $\mathcal{D}'$ 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 $1 - \delta$,*

$$\tau = \mathcal{O}(\frac{1}{\epsilon}) \cdot \frac{LF}{(1 - 2c^2\alpha^2)}$$

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. *Subspace quality, encoded in α, is critical for both final accuracy and the time to convergence.*

Remark 2. *The optional update approach enjoys logarithmic sample efficiency, meaning that the overhead of subsampling $\mathcal{D}'$ is low.*

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 *optimizer 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. 3.d (Alg. 3 in the Appendix presents pseudo-code). We split the GPU-to-CPU, CPU update, 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. Thus, in our schedule, the critical path of the training is characterized by

$$T^{iter}_{LSP} = \max\{T_{FWD} + T_{BWD} + T^{layer}_{Comm} + T^{layer}_{UPD}, T^{\text{GPU to CPU}}_{Comm}, T^{\text{CPU to GPU}}_{Comm}, T_{UPD}\}. \quad (4)$$

Compared to Eqn. 1, LSP-Offload is able to reduce 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 the llama-7B model.

In the Appendix we show how to avoid a deeper layer's workload from blocking a shallower layer's computation that executes earlier in the next iteration, improving performance.

Evaluation

We first verify the convergence of LSP-Offload on the GLUE dataset and then evaluate the end-to-end training performance on the instruction-tuning task. Detailed configurations for the experiments are described in the Appendix.

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 (Wang et al. 2018) dataset, which is a language understanding task set that is widely adopted for evaluating fine-tuning (Hu et al. 2021; Zhao et al. 2024). For hyper-parameters, we set both the rank of GaLore's projector and the non-zero entries per row in LSP to be 16, so that they use equal GPU memory. The projection space of LSP is set to 512. As both GaLore and LSP need additional profiling time, we make an end-to-end comparison that allows all candidates to train under an hour's time budget. LSP-Offload outperforms full parameter tuning, despite using only 253MB GPU memory vs. 747MB. LSP-Offload also achieves 1% higher average accuracy than GaLore. We attribute this to LSP-Offload's larger parameter update space (for the same GPU memory), which is 10x for this experiment. Moreover, Fig. 7 in the Appendix shows that LSP converges at the same iteration rate as full parameter tuning.

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 WizardCoder's (Luo et al. 2023) method on the laptop GPU (workstation GPU).We choose rank in LoRA, Galore, and r in LSP-Offload such that they use similar amount of memory below the GPU memory capacity. Detailed configurations are listed in the Appendix.

Table 3: Accuracy validation of LSP after 1 hour fine-tuning the pre-trained RoBertA-base model on GLUE.

	MNLI	SST2	MRPC	CoLA	QNLI	QQP	SST2	STS-B	Avg
Full Parameter	0.8111	**0.934**	0.866	0.55	0.904	0.808	**0.933**	0.884	0.8362625
GaLore (Rank=16)	**0.83**	0.92	0.88	0.567	0.881	**0.852**	0.92	0.9	0.84375
LSP (d=512, r=16)	0.814	0.917	**0.911**	**0.6165**	**0.9178**	0.8339	0.922	**0.91**	**0.855275**

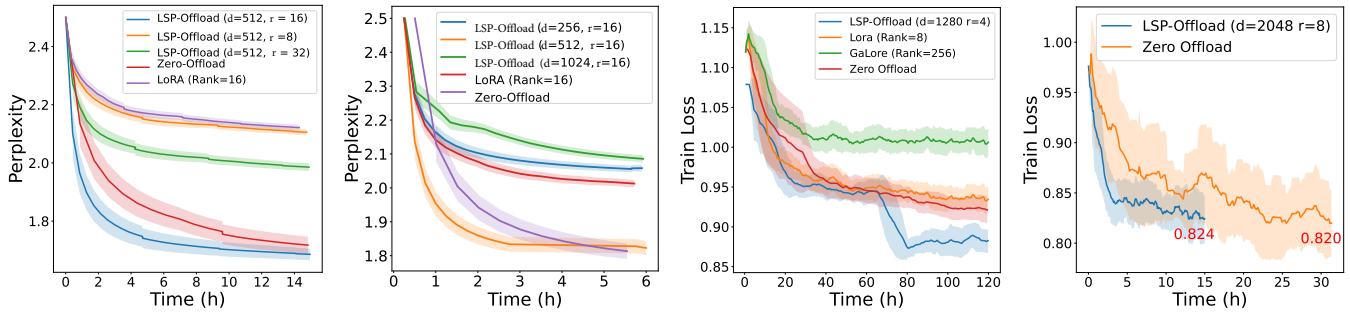
Table 4: Evaluation accuracy on the Humaneval dataset instruction after fine-tuning Deepseek-Coder-1.3B (top) and Deepseek-Coder-6.7b (bottom) with bfloat16 on the laptop GPU (top) and workstation GPU (bottom).

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

Compared to Zero-Offload, LSP-Offload achieves faster convergence on Alpaca. As shown in Fig. 4a and 4b, LSP-Offload uses around 62.5% and 33.1% less time when converging to the same 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. Also, LSP-Offload converges to the perplexity of 1.63 after 12 hours, which is achieved by Zero-Offload after 20 hours. Moreover, as shown in Fig. 4c 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, for the Deepseek-Coder-6.7B model, LSP-Offload completes the fine-tuning for one epoch 2x faster than Zero-Offload while achieving close accuracy. When trained for 15 hours, LSP-Offload outperforms Zero-Offload on average accuracy by 2.4%.

In addition, LSP-Offload achieves 30% lower evaluation perplexity than LoRa on Alpaca (Fig. 4a), and outperforms GaLore across 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. As shown in the Appendix, this is due to its lower relative estimation bias.

Training throughput comparison. Fig. 5 compares training throughput for different configurations. When trained on a subspace of size $d = 512$, LSP-Offload achieves 2.03, 3.09, 2.04, 3.33 times higher training throughput than Zero-Offload for the 4 test cases listed Fig. 5. Compared to an idealized training that accounts for FWD+BWD+UPD on the GPU but with free communication and CPU compute, LSP-Offload



(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 4: End-to-end evaluation of LSP-Offload. Rolling average is applied for drawing each curve. The shaded area around the line shows the standard deviation.

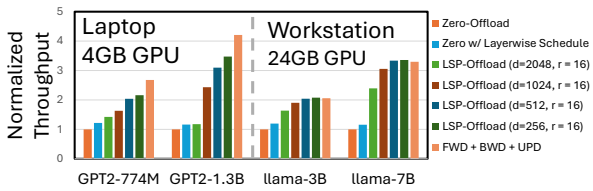


Figure 5: Training throughput comparison.

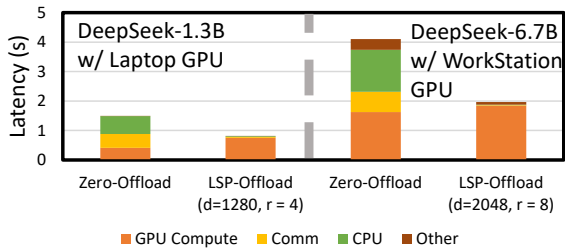


Figure 6: Breakdown for training 1 iteration of DeepSeek-Coder-1.3B (6.7B) with token batch size 384 = 1×384 (4096 = 4×1024) on the laptop GPU (workstation GPU). Only the non-overlapped part for Comm. and CPU are plotted.

only slows down on average 10.6%, 16.7%, 38.3% with the subspace of size 256, 512, 1024. Specifically, when trained on the workstation GPU with $d \leq 512$, LSP-Offload obtains 2% higher throughput as compared to idealized training due to its fully paralleled optimizer update step on the CPU. Lastly, applying our layer-wise schedule to Zero's schedule yields on average 18% increase in the throughput.

Hyperparameter sensitivity. We empirically compare the performance for different values of d (subspace size) and r (non-zeros per row). While smaller d limits the optimization space, we found too large d can lead to low accuracy because of over-fitting. Shown in Fig. 4b, $d = 512$ outperforms both 256 and 1024. But the training loss is 0.61 for $d = 1024$ and 0.72 for $d = 512$. In the Appendix, we further find that larger d and small r (e.g., $r = 4$) lead to lower estimation bias.

Training time breakdown. Fig. 6 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 paralleled, resulting in minimal non-overlapped overhead for communication and CPU compute.

Limitation

While efficient, LSP-Offload introduces a few hyperparameters that may need careful selection for the best performance (Fig. 4a, 4b). These include the selection of the (d, r)-sparse projector, the frequency of subspace updates, the threshold for these updates, and others. Additionally, the current prototype of LSP-Offload does not yet support quantization with 8-bit or lower data types, which is fully compatible with our approach and is planned for future work. Finally, while compression and decompression may introduce compute overhead on the GPU, this can be effectively mitigated by implementing specialized *sparse*-matrix multiplication kernels, which we also intend to address in future work.

Conclusion

In this paper, we observed that in the commodity setting, current offloading frameworks are fundamentally bottlenecked by the expensive communication and/or the CPU compute. Motivated by the PEFT method, we designed LSP-Offload to enable near-native speed fine-tuning by constraining the parameter update onto a subspace. Technically, we projected the gradient onto a subspace using a sparse projector, and boosted its performance by minimizing the empirical bias. Compared to the prior PEFT approaches (GaLore, LoRa), with the same amount of additional memory on GPU, we are able to optimize in much larger optimization spaces. In evaluation, we verified that LSP training can converge at the same rate with native training on the GLUE dataset. Also, in the end-to-end comparison with SOTA offloading framework Zero-Offload on the instruction-tuning task, LSP-Offload reduces fine-tuning time by 33.1% to 62.5% when converging to the same accuracy. Moreover, LSP-Offload improves accuracy by 27.8% to 30% over GaLore and LoRa on the Alpaca and Humaneval datasets.

References

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, 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.

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

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.

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

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.

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

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*.

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*.

Appendix

Zero's Schedule

Algorithm 2: Zero-Offload's Pseudo-code

Input: $\mathcal{M}$: GPU model, $\mathcal{D}$: Dataset, S: Optimizer state on CPU, W: Weights
for $t \leftarrow 1$ to τ **do**
  Sample $x \sim \mathcal{D}$
  $l \leftarrow \mathcal{M}.forward(x)$ {FWD on GPU}
  $\nabla_W l \leftarrow SendToCPU(\mathcal{M}.backward(l))$ {Paralleled BWD on GPU and Gradient Offload}
  $\Delta W \leftarrow SendToGPU(Update(\nabla_W l, S))$ {Paralleled Update on CPU and Delta Upload}
  $W \leftarrow W + \eta_t \cdot \Delta W$ {on GPU, learning rate η_t}
end for

Properties of (d, r)-Sparse Projectors

Property 1 (commutativity)**.** *For a distribution $\mathcal{D}$ on $R^{m\times n}$ matrices, P, Q the sparse projector,*

$$E_{\Sigma\sim\mathcal{D}}[P^T(\Sigma)Q] = P^T(E[\Sigma])Q \tag{5}$$

$$E_{\Sigma\sim\mathcal{D}}[\boldsymbol{b}^{P,Q}(\Sigma)] = \boldsymbol{b}^{P,Q}(E[\Sigma]) \tag{6}$$

Proof of Theorem 1

Our proof adapts the analysis in (Stich 2020).

Before proving Theorem 1, we list the lemmas used in the proof.

Lemma 1 (Matrix Chernoff)**.** *Let $M_1, ..., M_t$ be independent matrix valued random variables such that $M_i \in \mathbb{R}^{d_1\times d_2}$ and $\mathbb{E}[M_i] = 0$. If $\|M_i\|_2 \le \gamma$ holds almost surely for all $i \in \{1, ..., t\}$, then for every $\epsilon > 0, 0 < \delta < 1$, when $t > \frac{8\gamma^2}{3\epsilon^2}\log(\frac{d_1+d_2}{\delta})$,*

$$Pr(\|\frac{1}{t}\Sigma_i M_i\|_2 > \epsilon) \le \delta$$

Lemma 2. *For sparse projector P, Q, under Assumption 2, we can bound the bias by the empirical bias on a random sub-sampled dataset S of size $\mathcal{O}(\frac{8\gamma^2}{3\epsilon^2}\log\frac{m+n}{\delta})$ with probability at least $1-\delta$,*

$$\|\boldsymbol{b}^{P,Q}(\nabla f(W))\|_2 \le \|\boldsymbol{b}^{P,Q}(\nabla f_S(W))\|_2 + \epsilon,$$

where $f_S(W) := \Sigma_{x\sim S} f_x(W)/|S|$.

Proof. For data $x \in S$, let $M_x = \mathbf{b}^{P,Q}(\nabla f_x(W)) - \mathbf{b}^{P,Q}(\nabla f(W))$. By the commutativity of the bias (eqn. 6), $\mathbb{E}[M_x] = \mathbf{0}$. Under Assumption 2, $\|M_x\| < \gamma$. Also,

$$\begin{aligned}
\frac{1}{|S|}\Sigma_{x\in S} M_x &= \frac{1}{|S|}\Sigma_{x\in S}(\mathbf{b}^{P,Q}(\nabla f_x(W)) - \mathbf{b}^{P,Q}(\nabla f(W))) \\
&= \frac{1}{|S|}\Sigma_{x\in S}(\mathbf{b}^{P,Q}(\nabla f_x(W))) - \mathbf{b}^{P,Q}(\nabla f(W)) \\
&= \mathbf{b}^{P,Q}(\frac{1}{|S|}\Sigma_{x\in S}(\nabla f_x(W))) - \mathbf{b}^{P,Q}(\nabla f(W)) \\
&= \mathbf{b}^{P,Q}(\nabla f_S(W)) - \mathbf{b}^{P,Q}(\nabla f(W)).
\end{aligned}$$

By Matrix Chernoff (lemma 1), we have that for $|S| > \frac{8\gamma^2}{3\epsilon^2}\log\frac{m+n}{\delta}$,

$$Pr(\|\mathbf{b}^{P,Q}(\nabla f_S(W)) - \mathbf{b}^{P,Q}(\nabla f(W))\|_2 > \epsilon) \le \delta.$$

Therefore, with probability $1-\delta$,

$$\begin{aligned}
\|\mathbf{b}(\nabla f(W))\|_2 &\le \|\mathbf{b}^{P,Q}(\nabla f_S(W))\|_2 + \\
&\quad \|\mathbf{b}^{P,Q}(\nabla f_S(W)) - \mathbf{b}^{P,Q}(\nabla f(W))\|_2 \\
&\le \|\mathbf{b}^{P,Q}(\nabla f_S(W))\|_2 + \epsilon
\end{aligned}$$

□

Theorem 2. *(Stich 2020) For any $\epsilon > 0, 0 < \delta < 1$, suppose f is an L-smooth function[3], and for any weight matrices W, $\|\boldsymbol{b}^{P,Q}(\nabla f(W))\| \le m\nabla f(W) + \psi$, where $0 < m < 1, \psi > 0$, and stepsize $\eta = \frac{1}{L}$. Denote $F := \mathbb{E}[f(W_0)] - f^*$, Then with probability $1-\delta$,*

$$\tau = \mathcal{O}(\frac{1}{\epsilon})\cdot\frac{LF}{(1-m)}$$

iterations are sufficient to obtain $\min_{t\in[T]}\mathbb{E}\|\nabla f(W_t)\|^2 = \mathcal{O}(\epsilon + \frac{\psi}{1-m})$.

Now, we prove Theorem 1.

Proof. We analyze with some $\delta_0 > 0$ and $\beta > 0$. From lemma 2, under the Assumption 2, we know that for $|S| > \frac{8\gamma^2}{3\beta^2}\log\frac{m+n}{\delta_0}$, with probability $1-\delta_0$,

$$\|\mathbf{b}^{P,Q}(\nabla f(W))\|_2 \le \|\mathbf{b}^{P,Q}(\nabla f_S(W))\|_2 + \beta.$$

Also, because $\mathbb{E}[\nabla f_S(W)] = \nabla f(W)$ (lemma 1), under the Assumption 2, we have for $|S| > \frac{8\gamma^2}{3\beta^2}\log(\frac{d_1+d_2}{\delta_0})$, with probability $1-\delta_0$,

$$\|\nabla f_S(W) - \nabla f(W)\|_2 \le \beta$$

We bound the bias for every parameter update step,

$$\begin{aligned}
&\|\mathbf{b}^{P,Q}(\nabla f(W))\|_F \\
&\quad\text{by Assumption 3} \le c\|\mathbf{b}^{P,Q}(\nabla f(W))\|_2 \\
&\quad\text{with prob } 1-\delta_0 \le c\|\mathbf{b}^{P,Q}(\nabla f_S(W))\|_2 + c\beta \\
&\quad\text{by Assumption 1} \le c\alpha\|\nabla f_S(W)\|_2 + c\beta \\
&\quad\text{with prob } 1-\delta_0 \le c\alpha\|\nabla f(W)\|_2 + c\beta + c\alpha\beta \\
&\qquad\qquad \le c\alpha\|\nabla f(W)\|_F + c\beta + c\alpha\beta
\end{aligned}$$

Thus, with probability $1-2\delta_0$,

$$\|\mathbf{b}^{P,Q}(\nabla f(W))\|_F^2 \le 2c^2\alpha^2\|\nabla f(W)\|_F^2 + 2c^2\beta^2(1+\alpha)^2.$$

By plugging this into Theorem 2 for all steps from 1 to τ, we have that for $|S| > \frac{8\gamma^2}{3\beta^2}\log(\frac{d_1+d_2}{\delta_0})$, with probability $1-2\tau\delta_0$,

$$\tau = \mathcal{O}(\frac{1}{\epsilon})\cdot\frac{LF}{(1-2c^2\alpha^2)}$$

iterations are sufficient to obtain $\min_{t\in[\tau]}\mathbb{E}\|\nabla f(W_t)\|^2 = \mathcal{O}(\epsilon + \frac{2c^2\beta^2(1+\alpha)^2}{1-2c^2\alpha^2})$. Setting $\delta_0 = \frac{\delta}{2\tau}$ concludes the proof. □

[3] A function f: $\mathbb{R}^d \to \mathbb{R}$ is an L-smooth function if it is differentiable and there exists a constant $L > 0$ such that $f(\mathbf{y}) \le f(\mathbf{x}) + \langle\nabla f(\mathbf{x}), \mathbf{y}-\mathbf{x}\rangle + \frac{L}{2}\|\mathbf{y}-\mathbf{x}\|^2$.

Table 5: Configurations and timings for training/fine-tuning the GPT2-1.3B Model (using fp16) on commodity laptop hardware—the Nvidia A1000 GPU (4GB) and Intel Core-i7 12800H CPU (32GB). 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.

Parameters	Optimizer State	Activations	CPU-GPU Bandwidth	#Layers	GPU Memory
2.6GB	7.8GB	0.5GB	10–15GB/s	40	4GB
FWD on CPU	**BWD on CPU**	**UPD on CPU**	**FWD on GPU**	**BWD on GPU**	**UPD on GPU**
0.16s/layer	0.27s/layer	0.08s/layer	4.5ms/layer	8.7ms/layer	7.9ms/layer

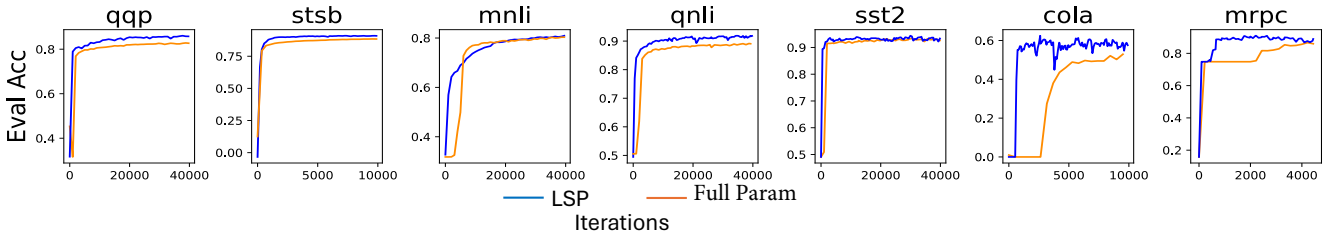


Figure 7: Convergence Validation of LSP by finetuning pre-trained RoBertA-base model on GLUE.

Layer-wise Scheduling

Algorithm 3: Layer-wise Scheduling

Hyperparameter: $TransitionLayer$: prevlayer to change the schedule mode from FirstComeFirstServe to LastComeFirstServe. Others are same as Alg. 1.
for $t \leftarrow 0$ to $\tau - 1$ **do**
  Sample $(x_0, y) \sim \mathcal{D}$
  for l in $layers$ **do**
    Wait for event e_l {prevforward pass happens after the parameter gets updated}
    $x_l \leftarrow \text{forward}(x_{l-1}, l, W_l)$
  end for
  $grad = \text{loss}(x_l, y)$
  $mode \leftarrow$ FCFS
  for l in reversed($layers$) **do**
    if l is $TransitionLayer$ **then**
      $mode \leftarrow$ LCFS
    end if
    $grad, \nabla_{W_l} \leftarrow \text{backward}(grad, x_l, l, W_l)$
    $\hat{\nabla}_{W_l} \leftarrow P_l^T \nabla_{W_l} Q_l$
    AsyncMemcpy$(mode, \hat{\nabla}_{W_l}, GPU2CPU)$
    AsyncExecOnCPU$(mode, \Delta_{W_l} \leftarrow Update(\hat{\nabla}_{W_l}))$
    AsyncMemcpy$(mode, \Delta_{W_l}, CPU2GPU)$
    AsyncExecOnGPU$(mode, W_l \leftarrow W_l - \eta_t P_l \Delta_{W_l} Q_l^T, CPU2GPU)$
  end for
end for

Avoiding blocking. To avoid the deeper layer's workload from blocking the shallower layer's computation that executes earlier in the next iteration, we use a heuristic to switch between two schedule mode: $FirstComeFirstServe$ (FCFS) and $LastComeFirstServe$ (LCFS). When the backward pass begins, FCFS is used first for parallelizing GPU compute and offloading. As the backward pass proceeds, we change the Schedule to LCFS which helps shallower layers get ready for the next pass. We set the switch point to be $TransitionLayer = \#Layer - \frac{T_{BWD} - (T^{layer}_{Offload} + T^{layer}_{Upload} + T^{layer}_{UPD})}{\max\{T^{layer}_{Offload}, T^{layer}_{Upload}, T^{layer}_{UPD}\}}$, which is the deepest layer that may block the computation of the first layer.

Implementation

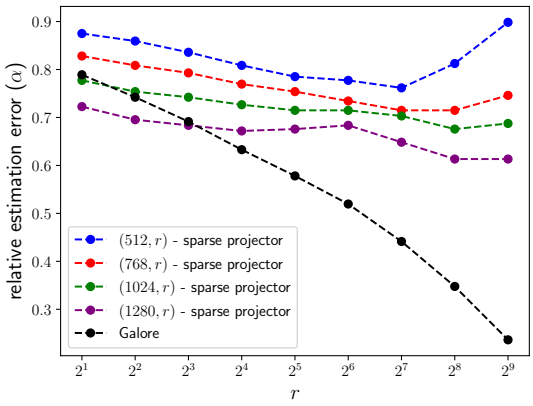
We prototyped LSP-Offload as a Python library built on top of Pytorch. LSP-Offload can automatically detect each matrix multiplication module and replace it with the offloaded version. To achieve best performance, we implemented the fused Adam kernel in Zero-Offload to accelerate the parameter update on the CPU. Also, we used a pinned memory buffer on the CPU to enable fast communication, and used CUDA streams for paralleled communication and computation. Moreover, gradient checkpoint is enabled to reduce the activation memory.

Experiment Configurations and Further Results

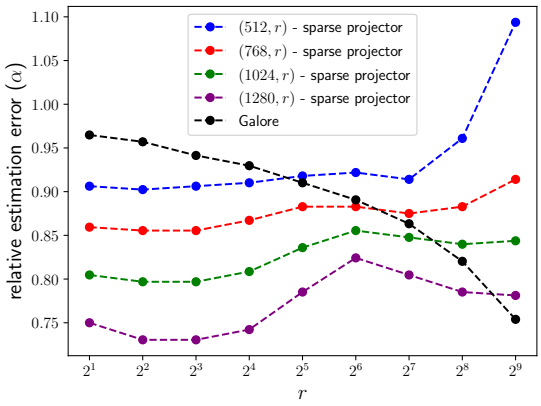
As noted in the Evaluation section, our **laptop GPU** setup is an Intel Core-i7 12800H CPU (32GB) laptop with an Nvidia A1000 Laptop GPU (4GB), and our **workstation GPU** setup is an AMD Ryzen Threadripper 3970X CPU (252GB) with an Nvidia RTX 4090 GPU (24 GB).

For all experiments, the random seed is set to a fixed value shown in the code.

GLUE Experiment For the GLUE experiment, we use a batch size of 16 and a learning rate of 5e-5 for LSP-Offload and Full Parameter fine-tuning. For LSP-Offload, we update the subspace at the beginning of each epoch or every 1000 iterations ($CheckFreq = 1000$) and set α, the threshold for updating projectors, to 0.3. All experiments are limited to train for 1 hour.



(a) Relative estimation train error on ∇_W



(b) Relative estimation test error on ∇_{W_t}

Figure 8: Relative estimation train/test error of different projectors. ∇_W and ∇_{W_t} are sampled from different inputs with batch size of 128. (d, r)-sparse projectors are learned on the ∇_W, and the SVD of GaLore is also performed on it.

As shown in Fig. 7, for all cases in GLUE, Alg. 1 is able to converge at a comparable rate *per iteration* as full parameter fine-tuning, despite its use of lossy compression (learned sparse projectors). Since full parameter fine-tuning suffers from significantly slower iteration times than LSP, the convergence rate *per hour* is slower than LSP. As Tab. 3 showed, this results in LSP achieving 0.855 average accuracy compared to Full Parameter's 0.836 after 1 hour.

Instruction Fine-tuning on Alpaca For the instruction fine-tuning experiments on the Alpaca data set, we use a batch size of 4 for the GPT2-774M model and 16 for the Llama-3B model, which are the largest without exceeding the laptop GPU and workstation GPU memory, respectively. We set the learning rate to be the best among {1e-4, 1e-5, 1e-6}, which is 1e-4 for LSP-Offload and 1e-5 for both LoRA and Zero-Offload. For LSP-Offload, $CheckFreq = 1000$ and $\alpha = 0.5$.

Instruction Fine-tuning with DeepSeek-Coder For the code instruction fine-tuning experiments, we use the gradient accumulate technique to simulate large batch sizes by averaging gradients from multiple small batches before updating the weights.

For the DeepSeek-Coder-1.3B model, we chose a simulated batch size of 128, max sequence length of 1024, the AdamW optimizer, max epoch number of 5, and the Cosine learning rate scheduler. The rank of LoRa is selected as the maximum value that the GPU memory can accommodate. The LoRa alpha (α) for LoRa (Rank=8) is 32. Because GaLore does not perform well with ranks that are small enough to fit into the GPU memory, we chose GaLore's rank to be 256, which uses 7.9GB memory. The alpha (α) for GaLore (Rank=256) is the default value of 0.25 in their library. We tried different learning rate from 1e-5 to 2e-4 and the learning rate is set to be the optimal value across multiple experiments. We found that the learning rate of 1e-4 performed well across different settings and used it in our final experiments. The evaluation accuracy of fine-tuning DeepSeek-Coder-1.3B on the Humaneval dataset shown in Tab. 4(top) are the scores corresponding to the checkpoint after approximately 120 hours of training on the laptop GPU.

For the DeepSeek-Coder-6.7B model, we chose a simulated batch size of 64, max sequence length of 1024, the AdamW optimizer, max epoch number of 1, and the Cosine learning rate scheduler with minimal learning rate. We found that the learning rate of 1e-4 and minimal learning rate of 5e-5 performs well across different learning rate settings. The evaluation accuracy of fine-tuning DeepSeek-Coder-6.7B shown in Tab. 4(bottom) are the scores corresponding to the checkpoint after approximately 15 hours of training on the workstation GPU. The table also shows the accuracy of Zero-Offload after 30 hours of training.

For the DeepSeek-Coder experiments, due to the extended training time required with offloading, we simulate the training process by:

1. Training on a GPU with sufficient memory: This allows us to obtain the training performance (e.g., training loss, evaluation score, etc.) as a function of training steps.

2. Profiling the average time per training step with offloading under maximum supported token batch size under the memory limit.

3. Combining the results: We merge the performance data from step 1 and scale the token batch size in step 2 to match the accumulated token batch size per update step in step 1, to map training performance against training time.

This approach enables us to simulate the training performance over time as if offloading were being used, without needing to actually train with offloading. Specifically, for both LSP-Offload and Zero-Offload, on the DeepSeek-Coder-1.3B (6.7B) experiment, we chose the token batch size as $384 = 1 \times 384$ ($4096 = 4 \times 1024$) respectively. The per-iteration profile result is shown in Fig. 6.

LSP-Offload's Hyperparameters

LSP-Offload introduces some new hyperparameters: including the number of non-zero values r in each row of the $(d,\ r)$-sparse projector, the size d of the subspace, and the frequency $CheckFreq$ and α threshold of the projector update. Among them, d and r have a great influence on the effect of the projector.

We tested the $estimated\ bias$ with our learned sparse projectors and with the orthogonal projectors used in GaLore on the DeepSeek-Coder-1.3B fine-tuning task. The orthogonal projectors are the spectrum of ∇_W calculated via Singular Value Decomposition (SVD):

$$\begin{aligned} &\nabla_W = USV^T \approx \sum_{i=1}^{r} s_i u_i v_i^T \\ &P = [u_1, u_2, \ldots, u_r], \quad Q = [v_1, v_2, \ldots, v_r]^T \end{aligned} \tag{7}$$

where r is also called the rank of the orthogonal projectors. We will use GaLore(r) to indicate GaLore run with rank=r. Recall from Tab. 2 that for the same r, GaLore(r) uses more GPU memory than LSP-Offload with (d, r)-sparse projectors. The results are shown in Fig. 8.

Although Fig. 8a shows that GaLore(r) has a lower *training* error than (d, r)-sparse projectors for the same r when $r \geq 16$, (d, r)-sparse projectors generalize much better: their *test* errors are much lower than GaLore's. For example, Fig. 8b shows that $(1280, r)$-sparse projectors achieve lower test error than GaLore(r) for the same r for all $r \leq 256$. We attribute this to (d, r)-sparse projectors' decoupling of the subspace size d and the non-zero values r (compared to GaLore's sole use of a rank r), such that we can optimize in a large subspace with minimal extra GPU memory.

Secondly, we found the performance of the learned (d, r)-sparse projectors does not necessarily improve as r increases. On the contrary, selecting a relatively small r, such as 4 or 8, tends to result in better generalization for this fine-tuning task (while also reducing GPU memory usage and shortening iteration times).

At the same time, the effectiveness of the projector improves as the subspace size d increases. Therefore, it is advisable to select the largest possible subspace size that does not cause an unacceptable decrease in training performance or exceed the GPU memory capacity.

Reproducibility Checklist

This paper does not use any novel datasets. Thus the novel dataset related answers are all NA.

All datasets used in this paper are public, thus the answer to the question regarding un-public datasets is NA.

Given the long-experiment times and corresponding resource costs, we run experiments for the best-found configuration once. Thus, we did not apply statistical tests to the result.