



USENIX

THE ADVANCED COMPUTING
SYSTEMS ASSOCIATION

Cocoon: A System Architecture for Differentially Private Training with Correlated Noises

Donghwan Kim and Xin Gu, *The Pennsylvania State University*; Jinho Baek, Timothy Lo, Younghoon Min, Kwangsik Shin, and Jongryool Kim, *SK hynix Inc.*; Jongse Park, *Korea Advanced Institute of Science and Technology (KAIST)*; Kiwan Maeng, *The Pennsylvania State University*

<https://www.usenix.org/conference/osdi26/presentation/kim-donghwan>

This paper is included in the Proceedings of the 20th USENIX Symposium on Operating Systems Design and Implementation.

July 13–15, 2026 • Seattle, WA, USA

ISBN 978-1-939133-55-7

Open access to the Proceedings of the 20th USENIX Symposium on Operating Systems Design and Implementation is sponsored by



جامعة الملك عبد الله
للعلوم والتقنية
King Abdullah University of
Science and Technology

Cocoon: A System Architecture for Differentially Private Training with Correlated Noises

Donghwan Kim¹, Xin Gu¹, Jinho Baek², Timothy Lo², Younghoon Min²,
Kwangsik Shin², Jongryool Kim², Jongse Park³, Kiwan Maeng¹

¹The Pennsylvania State University

²SK hynix Inc. ³KAIST

Abstract

Machine learning (ML) models memorize and leak training data, causing serious privacy issues to data owners. Training algorithms with differential privacy (DP) have been gaining attention as a solution. However, these algorithms add noise at each training iteration and degrade accuracy, limiting their real-world adoption. To improve accuracy, a new family of approaches adds carefully designed *correlated noises*, so that noises cancel out each other across iterations. We performed an extensive characterization study of these new mechanisms and show they incur non-negligible overheads when the model is relatively large or uses large embedding tables compared to the hardware capacity. Motivated by the analysis, we propose Cocoon, a framework for efficient training with correlated noises. Cocoon stores and processes the large noise history across CPU, GPU, and memory extension module, introduces optimizations for sparse embedding tables, and leverages to-be-commercialized near-memory processing (NMP) devices. On a real system with an FPGA-based NMP device prototype, Cocoon improves the performance by $1.23\text{--}10.82\times$.

1 Introduction

Machine learning (ML) models are vulnerable to memorization and leakage of their training data—sometimes by accident, or when model is subjected to an adversarial attack. This poses a serious *privacy risk* because ML models are often trained with sensitive user data (*e.g.*, medical data [108], audio [66] or keyboard inputs [82, 113], personal chat logs [103], *etc.*). The threat is not hypothetical but imminent: a recent study demonstrated the feasibility of attacking the popular ChatGPT [68] to extract its training data that contains sensitive information. Similar attacks have been shown to be possible for other models [9, 10, 28, 50]. The privacy risks can deter user participation in data collection and training, degrading the quality of real-world ML-based services [60].

Differential privacy (DP) [25] is one of the most principled and widely adopted approaches for mitigating such privacy

risks. DP theoretically bounds the information leakage of private data through its statistics. In the context of ML training, this translates to a mathematical bound on the probability that an adversary can successfully extract information of the training data from the trained model [36, 37, 39, 49, 70]. DP is not a premature research concept; already deployed in several real-world applications, including the US Census [3], Apple’s device analytics [86, 91], and Google Chrome’s behavior telemetry [26]. The use of DP for ML training is also gaining increased attention. For instance, Google publicly announced its use of DP for training all of their current and future smart keyboard (Gboard) models [101, 113], and also recently released VaultGemma [89], an LLM that was trained with DP from scratch. Microsoft has also limitedly deployed a DP-trained model on their SwiftKey keyboard [2].

Most DP training algorithms add Gaussian noise to the gradient at each training iteration to provide a theoretical privacy guarantee [1, 21]. This Gaussian noise accumulates over training and significantly degrades the model accuracy, which is the primary barrier to DP’s widespread adoption. To counteract the accuracy issue, a recent line of works [13–16, 48, 63–65, 79, 109] proposed to use noises that are *correlated* across iterations, allowing noises added in later iterations can partially cancel out earlier noises. *Correlated noise mechanisms* have demonstrated both theoretical [13, 55] and empirical [13–16, 48, 63–65, 79, 109] benefits over simply adding Gaussian noise. The mechanism has already been successfully deployed in real-world products [101, 113], and adopted into popular ML privacy libraries [32, 33]. Their rapid adoption send a strong signal of their importance.

Despite their growing popularity, the system implications of correlated noise have not been thoroughly studied. Existing literature mostly focused on accuracy rather than system efficiency, training relatively small models with a large number of TPUs/GPUs (*e.g.*, using 1024 machines to train a billion-parameter model [63]). In these *resource-abundant* environments, noise-related overheads are small. However, our study reveals that correlated noise incurs significant system overheads, including a memory footprint that exceeds the pa-

parameter size by over an order of magnitude, and causing up to $14.49\times$ runtime slowdown. Such costs are particularly severe for *economical* setups with fewer TPUs/GPUs, which is a configuration frequently studied to enable large model training on small and mid-sized entities [5, 83–85, 110]. This overhead arises because generating correlated noises involve storing and processing noise history from past iterations, which can scale up to hundreds of gigabytes or even terabytes. It becomes especially problematic for (1) models with large embedding tables and (2) large-scale, billion-parameter models (e.g., LLMs).

We introduce *Cocoon*, a framework for efficient at-scale DP training with correlated noises. Cocoon splits the large noise history between CPU, GPU, and (if present) additional memory extension module, and processes them in a distributed fashion to maximize performance. To mitigate managing noise history for large embedding tables, Cocoon precomputes and stores the noise for embedding tables in a compact format by exploiting their gradient sparsity (Section 4.2). Furthermore, when part of the noise history is offloaded to a memory extension module, we show that leveraging emerging near-memory processing (NMP) solutions [46, 87] can greatly improve performance and reduce the cost of deployment (Section 4.3). Our evaluation on a real system, with an FPGA-based industry prototype NMP device, showed $1.23\text{--}10.82\times$ speedup on various models and training scenarios. We summarize our contributions:

- We present an extensive system characterization study of emerging DP training methods that use correlated noises. Our analysis reveals that, on an economical setup where the number of GPUs is moderate, correlated noise generation can introduce non-negligible slowdown.
- We introduce Cocoon, a highly-optimized, PyTorch-based DP training library that uses correlated noises. Cocoon splits and processes noise history on CPU, GPU, and (if present) emerging NMP devices.
- Cocoon introduces a noise pre-computing and coalescing optimization that accelerates training with large embedding tables by $2.33\text{--}10.82\times$. With to-be-commonly-deployed NMP devices, Cocoon can achieve $1.23\text{--}2.32\times$ speedup over the baseline when training large models, which we demonstrate with a real industry prototype.

2 Background and Motivation

2.1 Differentially Private Training Algorithms

Differential privacy. Differential privacy [25] (DP) provides a robust guarantee that the outcome of a randomized mechanism does not change significantly when a single entry in the input dataset changes. When applied to ML training, DP ensures that the final trained model does not depend significantly on a single sample (or user [11, 19]) in the training corpus, which effectively limits an adversary’s capabilities. For

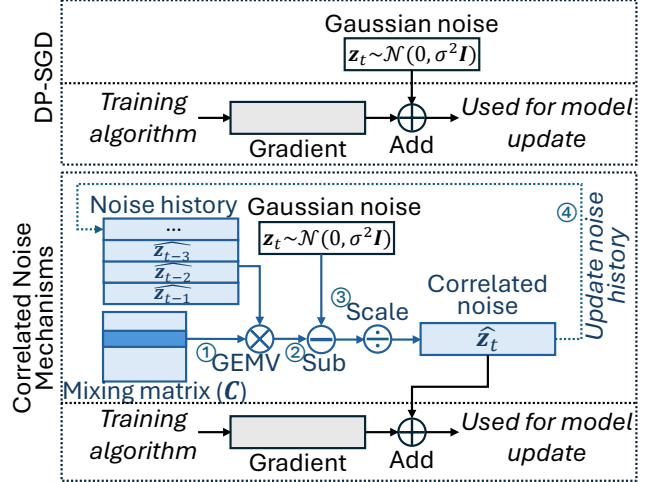


Figure 1: Independent Gaussian vs. correlated noise.

example, DP parameters can theoretically bound the attack success rate for adversaries attempting to infer whether a user participated during training [49, 70] or trying to directly reconstruct the original training data [36, 37, 39] from a DP-trained model. A model trained with DP can be used for multiple inferences over time, and it will not leak additional information beyond the privacy cost paid upfront during training. This is in contrast with inference-time DP techniques [31, 54, 94, 97], which leak more information about the corpus with more inferences [80, 97].

Training with Gaussian noise. The most common method to train a model with DP is using DP-SGD [1] or its variants. DP-SGD introduces three key modifications to regular SGD. First, the batch in each iteration is formed through random sampling with replacement. Second, unlike SGD, which calculates an average gradient across the batch, DP-SGD first calculates per-sample gradients, scales (clips) each one individually, and then averages the scaled gradients. Finally, an independently sampled Gaussian noise is added to the averaged gradient before it is applied to update the model [1]. While these steps provide stronger privacy guarantees, DP-SGD significantly compromises the trained model’s accuracy.

Several follow-up works have introduced optimizations to improve the model accuracy over the DP-SGD algorithm. Most of this research still relies on *independent Gaussian noise* and focuses on optimizing other components of DP-SGD (e.g., sampling [20], clipping [7, 67], hyperparameter [21], feature engineering [96], post-processing after noise addition [34, 112], optimizer [12, 29, 92, 93, 102], and architecture [21, 74]). These optimizations still suffer from accuracy degradation due to Gaussian noise. Figure 1 (top) highlights the independent Gaussian noise added to each gradient in DP-SGD.

Correlated noise mechanisms. Instead of adding independent Gaussian noise, a recent line of works [13–16, 48, 63–65, 79] proposed to add noise that is *correlated* across iterations. When generating correlated noises at iteration t , these mechanisms mix noises that were used in the previous $\hat{b} - 1$ iterations (*i.e.*, noise history), where $\hat{b}$ is called the *band size*.

Mathematically, correlated noises are generated as follows. For a model with m trainable parameters trained through n training iterations, let $\mathbf{z}_t \in \mathbb{R}^m$ be a Gaussian noise sampled at iteration t . Each noise is as large as the trainable parameters (m), and training a large model requires using a noise that is as large. For a specific *mixing matrix* $\mathbf{C} \in \mathbb{R}^{n \times n}$, the correlated noise at iteration t , $\hat{\mathbf{z}}_t \in \mathbb{R}^m$, can be calculated as:

$$\hat{\mathbf{z}}_t = (\mathbf{z}_t - \sum_{\tau=1}^{\min(t, \hat{b}-1)} \mathbf{C}[t, t-\tau] \hat{\mathbf{z}}_{t-\tau}) / \mathbf{C}[t, t]. \quad (1)$$

Figure 1 (bottom) visualizes how $\hat{\mathbf{z}}_t$ is calculated. First, a ① *weighted average* of the $\hat{b} - 1$ previous noises is performed. The result is ② subtracted from a newly-sampled Gaussian noise ($\mathbf{z}_t$) and ③ properly rescaled, which becomes the new correlated noise ($\hat{\mathbf{z}}_t$). As in DP-SGD, $\hat{\mathbf{z}}_t$ is added to the gradient, and the noisy gradient is used to update the model. At the same time, ④ $\hat{\mathbf{z}}_t$ is saved to the noise history, so that it can be used to generate future noises.

The weighting/rescaling factors at t -th iteration are defined by the t -th row of the mixing matrix $\mathbf{C}$, and the weighted averaging can be done through a *matrix-vector multiplication* (*GEMV*) between the stacked noise history (matrix of size $(\hat{b} - 1) \times m$) and the t -th row of $\mathbf{C}$ (each row is of size n but only has $\hat{b} - 1$ nonzero elements). $\mathbf{C}$ should be carefully designed to guarantee DP, and different prior works developed different designs of $\mathbf{C}$ [15, 48, 65]. When $\hat{b} = 1$ and $\mathbf{C} = \mathbf{I}$ (an identity matrix), this reduces to independent Gaussian noise.

Generality and importance of correlated noise. As Figure 1 shows, correlated noise can almost serve as a drop-in replacement for independent Gaussian noise¹. Thus, making the correlated noise generation efficient is likely to benefit a wide range of DP training algorithms that currently use independent Gaussian noise. Several recent works have demonstrated that correlated noise has theoretical [13, 55] and empirical [13–16, 48, 63–65, 79, 109] benefits over independent Gaussian noise. Google recently adopted correlated noise in its production (Gboard training [101, 113]) and has been exploring using it for larger models [63, 79], which strongly signals its practical importance. A recent 212-page report [79] summarizes the latest efforts around correlated noise. Its generality and importance motivate our study.

¹When using correlated noise, it puts a restriction on what data sampling method can be used, but other parts of the training do not need to change.

2.2 Training of Embedding Tables

An embedding table is a trainable data structure that converts categorical features into a dense vector. It is commonly used in deep learning recommendation models (DLRMs) or large language models (LLMs). Especially in DLRMs, embedding tables are very tall and dominate the model size [38, 62, 71]. During each training iteration, only rows (entries) of the table corresponding to the present feature values of the input are used, which leads to several unique behaviors. First, the training speed grows sub-linearly with its size compared to other models, because only a tiny subset is used even when the entire table is large. Second, unused entries in each iteration have zero gradients. Still, DP training requires noise addition to these zero gradients for privacy [62, 72]. Third, only the entries accessed in each iteration contribute to the gradient calculation at that iteration. As we will show later, the first characteristic leads to a unique overhead when using correlated noises, and the latter two will be leveraged by Cocoon’s optimizations (Section 4.2).

2.3 CXL and Near-memory Processing (NMP)

Compute express link (CXL) is an open industry interconnect standard based on PCIe, which allows high-speed loads and stores for memory expansion modules plugged into the PCIe slot. Memory expansion module connected through CXL, or *CXL memory*, has emerged as a way to expand memory capacity. We study the use of such CXL memory to expand the device memory capacity when the noise history is too large to be hosted on main memory.

While CXL memory provides larger memory, data access through CXL is much slower than the main memory of the CPU. Several recent works [35, 51, 53, 78, 88, 107] have proposed to put additional compute units in the CXL controller to reduce data movement through PCIe, and some have made it into (near-)real products [46, 87]. In Section 4.3, we show how such emerging *near-memory processing* (NMP) *CXL memory* products can accelerate Cocoon.

3 Characterization of Correlated Noise

To study the system overheads of correlated noise, we trained various models with independent Gaussian noise and correlated noise. We used a correlated noise mechanism called BandMF [15]. Our study can be generalized to other mechanisms, because other mechanisms only differ in the derivation of mixing matrix $\mathbf{C}$ (Section 2.1), and are computationally equivalent. Our study highlights that correlated noise mechanisms experience non-negligible memory (Section 3.1) and compute (Section 3.2) overheads.

For the study, we trained popular models from prior DP training literature on a dual-socket Intel Xeon Gold 6330 CPU with 256GB DRAM and 8x NVIDIA A5000 GPUs, which

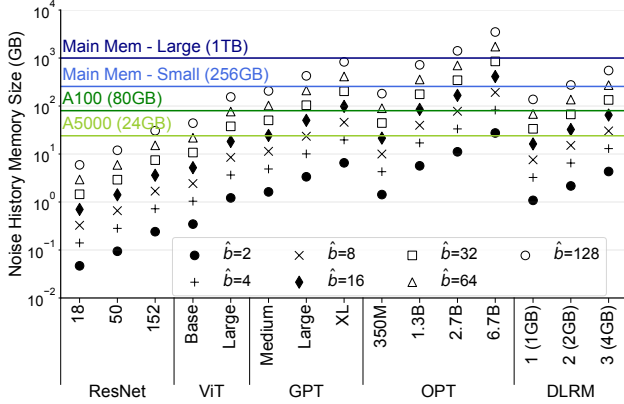


Figure 2: Noise history size of various ML models and $\hat{b}$.

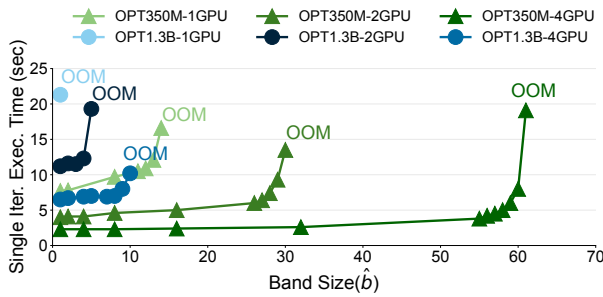


Figure 3: Training time of OPT [111] on 1–4 A5000 GPUs.

represents an economical setup of a small/mid-sized entity. The models we trained include: ResNet [41], ViT [24], LLMs (GPT [81] and OPT [111]), and DLRM [71]. Measurements were done on our custom DP training code with correlated noise support, which we built as part of our Cocoon library (Section 4). We only highlight the most interesting results.

3.1 Memory Overhead

Generating correlated noise involves storing $\hat{b} - 1$ past noises, each of which is as large as the number of trainable parameters (m). This introduces several capacity and performance issues.

Capacity issue. Figure 2 summarizes the memory footprint of the noise history for various models and $\hat{b}$, along with common GPU memory and main memory (CPU DRAM) sizes. The footprint frequently exceeds the GPU/CPU memory, and the noise history must be offloaded to main memory or secondary memory (e.g., CXL memory or SSD) to run the training at all in these cases. Performance implications of these fallbacks are discussed in Section 3.2. Some works [63, 79] found such offloading to be unnecessary, but this was because they had enough GPUs/TPUs, unlike our economical setup.

Takeaway 1: The noise history can become larger than the aggregate GPU/TPU memory or even the main memory.

Performance issue. Even when the entire noise history fits into the GPU, the performance can degrade if too little memory is left for training. Figure 3 shows the training latency of two OPT models [111], where different lines correspond to different models and GPU counts. It can be seen that the training time increases with $\hat{b}$ for all setups, until an out-of-memory (OOM) error is triggered. This is because if too little memory is left, training must use smaller microbatches, underutilizing the GPU [98].

Takeaway 2: Even when storing the entire noise on the GPU is possible, it may degrade the training performance.

Why not re-generate noises? Instead of storing past noises, prior work [48] considered only storing the seed and re-generating noises on every iteration. However, doing so requires re-generating all the noises from the beginning ($\hat{z}_1, \dots, \hat{z}_t$) and not just the past $\hat{b} - 1$ noises, because each noise generation recursively requires its past $\hat{b} - 1$ noises. This incurs $O(n^2)$ overhead for n training iterations, which becomes too large unless n is very small. Another prior work [79] similarly observed that this approach scales poorly with n .

3.2 Compute Overhead

The weighted averaging of prior noises (GEMV between the noise history and the mixing vector) also incurs a computational overhead. We mainly consider economical setups where part of the noise history is stored in main/secondary memory.

For the parts hosted outside the GPU, we studied two options for processing them: (1) *GPU-GEMV* performs GEMV only on the GPU. While GEMV is fast on a GPU, this design requires additional data transfer from the main/secondary memory to GPU through the slow PCIe bus. (2) *CPU-GEMV* performs GEMV on the CPU for the subset of the noise history stored in main/secondary memory, and only sends the result to the GPU. While CPU’s GEMV is slower, this design enjoys the higher bandwidth between the CPU and main memory, compared to the PCIe bus. Also, CPU-side GEMV can happen *in parallel* with GPU-side training and can be partially or completely hidden.

Overheads of DLRM. Correlated noises incur a unique overhead on DLRMs due to their large embedding tables (Section 2.2). Figure 4 shows the training time of three DLRMs with different embedding dimensions (DLRM-1/2/3). We only characterized a single-GPU setup because the single-iteration latency was too small (near 100ms) to make data parallel training effective. GPU-GEMV’s latency consists of the GPU-side training and GEMV (“Train (GPU)”), and data transfer from the main memory (“Transfer (Main Mem)”). CPU-GEMV’s latency is governed by the slower of the two parallel tasks shown in side-by-side bars: the GPU-side training, and the CPU-side GEMV (“GEMV (CPU)”) plus the

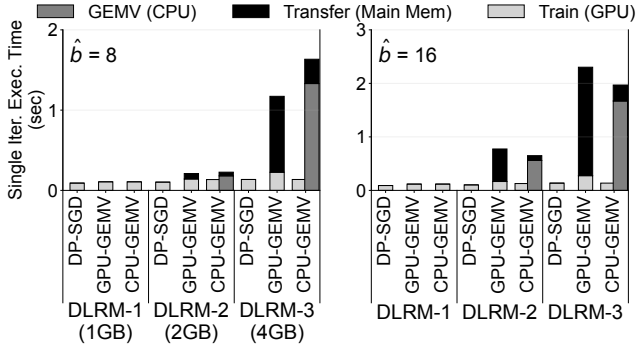


Figure 4: Training time breakdown for DLRM.

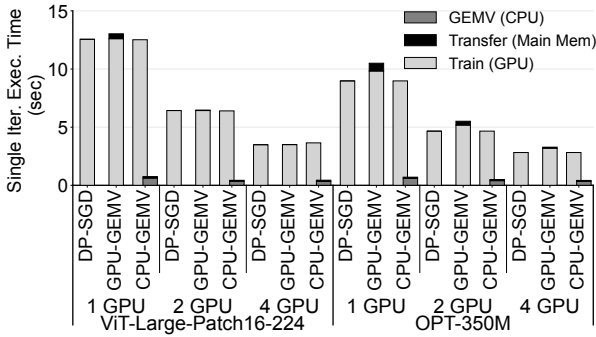


Figure 5: Training time breakdown when the noise history entirely fits into main memory.

transfer of the GEMV result to the GPU (“Transfer (Main Mem)”).

In general, GPU-GEMV is better when the model and/or $\hat{b}$ is small, and CPU-GEMV outperforms when they are larger. However, except for uninteresting cases where the entire noise history fits into GPU (DLRM-1), both baselines still incurred non-negligible slowdown. Even the better baseline between the two incurred $2.03\text{--}8.62\times$ ($\hat{b}=8$) and $6.28\text{--}14.49\times$ ($\hat{b}=16$) slowdown. This was due to the noise-related overheads growing much more linearly with m compared to the training time, which grows sub-linearly with m (Section 2.2).

Takeaway 3: For DLRM, correlated noises incur significant slowdown compared to using independent noises.

Overheads of non-DLRM. Models other than DLRM experienced similar trends: the exact type mattered less, and their absolute size (m) mattered more. Specifically, the behavior depended on whether the noise history overflowed the main memory and involved secondary memory.

Figure 5 shows the result from training ViT-L and OPT-350M with $\hat{b}=16$. In these setups, the entire noise fits into the main memory. The slowdown was only 0.6–18.2% for GPU-GEMV, and there was negligible slowdown for CPU-GEMV. Other small models (CNNs, ViTs, and sub-billion LLMs) showed similar behaviors, which we omit.

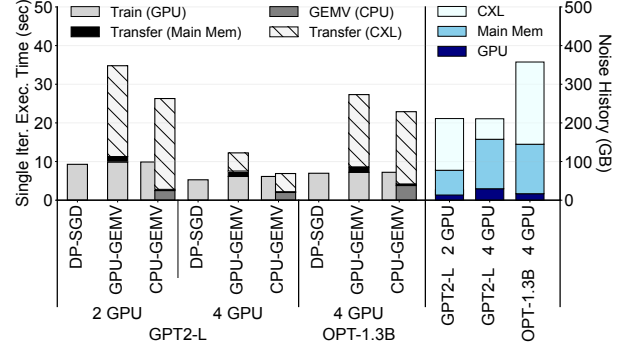


Figure 6: Training time breakdown with noise history partially stored in CXL memory.

When the model size and $\hat{b}$ get larger, noise history must be partially offloaded to secondary memory which incurs a non-negligible slowdown. We only focused on CXL memory due to its relatively better read speed. Slowdown will be higher for slower alternatives like SSD. Figure 6 shows the training time breakdown (left), along with the information on where different portions of the noise history are stored (right). When training GPT2-L with 2 GPUs, 63% of the noise history was placed in CXL memory, leading to $2.83\text{--}3.75\times$ slowdown. When more GPUs are added (GPT2-L with 4 GPUs), some noises could move from CXL memory to GPU memory, improving the slowdown to $1.30\text{--}2.31\times$. With a larger OPT-1.3B, the slowdown again increases to $3.28\text{--}3.91\times$.

Takeaway 4: For non-DLRM models, the overhead is small if the noise history can fit into main memory. When the noise history overflows main memory, data traffic to secondary memory adds significant latency.

When CPU is highly utilized. We additionally note that CPU-GEMV may incur a larger slowdown when the CPU suffers from resource contention. As an illustration, we trained OPT-350M with $\hat{b} = 64$ on 4 GPUs, and varied the number of cores used by the CPU-side GEMV. We observed that the training starts to slow down if not enough CPU cores can be dedicated to GEMV. With only 4–7% of the cores, the training slowed down by $1.52\text{--}2.77\times$.

Takeaway 5: CPU-GEMV may incur additional overhead when the CPU is highly congested.

4 The Cocoon System

We introduce Cocoon, a framework for efficient DP training with correlated noise. Cocoon stores and processes large noise history across GPU, CPU, and CXL memory device with NMP to maximize performance on an economical setup (Figure 7). For large embedding tables of DLRM, Cocoon provides a dedicated optimization that pre-computes and stores correlated noises in a coalesced format (Section 4.2). When

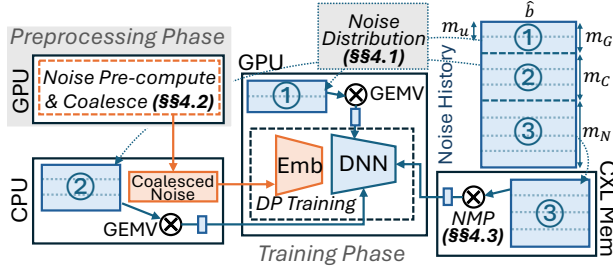


Figure 7: Overview of Cocoon. Cocoon distributes the large noise history across ① GPU, ② CPU, and ③ CXL memory, and performs GEMV near each memory unit to minimize data movement (Section 4.1). Cocoon introduces a preprocessing phase for large embedding tables (Section 4.2), and GEMV inside the CXL memory is done through NMP (Section 4.3).

the CXL memory with NMP exists, Cocoon offloads part of the GEMV to the NMP device to further improve performance (Section 4.3). Cocoon is built on top of Amazon’s fastDP [8] and tested on a node with multiple GPUs and an industry prototype CXL NMP device. Cocoon can leverage other NMP devices as long as they support GEMV [46, 87].

Threat model. DP training methods provide guarantees against an adversary who can access (1) the final model and (2) all the intermediate gradients. Cocoon assumes the same adversary except for its embedding table optimization (Section 4.2), where it assumes a weaker-but-practical adversary who can access only the final model and not the intermediate gradients. Assuming an attacker with full knowledge about the intermediate gradients is an artifact of DP-SGD’s underlying math and is often considered excessively strong [70]. The weaker-but-practical adversary we assume for the embedding table optimization reflects the most common real-world threat, such as attackers attempting to extract training data from ML services via APIs [68] or through open-sourced weights [9, 10], where intermediate gradients are unavailable. Many other works adopt this same practical adversary model [17, 27, 62, 69, 70, 72, 99].

4.1 Distributing Noise with Profiling

Cocoon splits the $\hat{b} \times m$ noise history in its parameter (m) dimension and distributes slices of size $\hat{b} \times m_G$, $\hat{b} \times m_C$, and $\hat{b} \times m_N$ to GPU, CPU, and NMP device, respectively ($m_G + m_C + m_N = m$). The goal is to minimize the overall training latency while ensuring none of the devices experience an out-of-memory error. Cocoon achieves this through simple profiling, whose cost can be amortized across multiple training iterations.

The goal is solve the following optimization problem:

$$\begin{aligned} \min_{m_G, m_C, m_N} \quad & \max \left(\text{Lat}_{\text{train-G}} + \text{Lat}_{\text{gemv-G}}(m_G, \hat{b}), \right. \\ & \left. \text{Lat}_{\text{gemv-C}}(m_C, \hat{b}), \text{Lat}_{\text{gemv-N}}(m_N, \hat{b}) \right) \\ \text{s.t.} \quad & m_G + m_C + m_N = m, \\ & \text{Mem}_{\text{total-G}} \geq \text{Mem}_{\text{train-G}} + \text{Size}(m_G \times \hat{b}) + \text{Buf}, \\ & \text{Mem}_{\text{total-C}} \geq \text{Mem}_{\text{train-C}} + \text{Size}(m_C \times \hat{b}), \\ & \text{Mem}_{\text{total-N}} \geq \text{Size}(m_N \times \hat{b}), \end{aligned}$$

where $\text{Lat}_{\text{train-G}}$ refers to the single training iteration latency on the GPU, $\text{Lat}_{\text{gemv-X}}(m_X, \hat{b})$, $X \in \{G, C, N\}$ is the GEMV latency on GPU (G), CPU (C), and NMP (N) with the given noise history, $\text{Mem}_{\text{total-X}}$, $X \in \{G, C, N\}$ refers to each device’s total memory capacity, and $\text{Mem}_{\text{train-X}}$, $X \in \{G, C\}$ is the peak memory usage of GPU and CPU for training (NMP is not used for training). Buf is a space reserved in GPU for receiving the GEMV result from CPU and NMP plus avoiding out-of-memory (OOM) due to noise in the estimation, and is a design parameter that depends on the implementation. In short, Cocoon minimizes the overall latency, which is bottlenecked by the slowest among the three devices (GPU training + GEMV, CPU GEMV, and NMP GEMV), while ensuring no out-of-memory.

Finding an optimal split requires accurately estimating each term. $\text{Size}(m_X \times \hat{b})$ and $\text{Mem}_{\text{total-X}}$ can be known exactly offline, and $\text{Lat}_{\text{train-G}}$ and $\text{Mem}_{\text{train-X}}$ can be upper-bounded by running a few training iterations. $\text{Lat}_{\text{gemv-X}}$ cannot be easily estimated because the GEMV latency is not necessarily linear with the matrix size, especially when the matrix is small. We simplify the optimization goal by defining a minimum unit of allocation, m_u , that perfectly divides m , and choose m_G , m_C , and m_N to be a multiple of m_u . This simplifies the GEMV latency estimation because if m_u is large enough, GEMV becomes memory-bottlenecked and almost linearly scales with the number of m_u processed in each device, i.e., $\text{Lat}_{\text{gemv-X}}(m_X, \hat{b}) \approx \frac{m_X}{m_u} \times \text{Lat}_{\text{gemv-X}}(m_u, \hat{b})$. Measuring $\text{Lat}_{\text{gemv-X}}(m_u, \hat{b})$ for $X \in \{G, C, N\}$ once and plugging the approximation back in the optimization objective gives us a linear objective that can be solved to get m_G , m_C , and m_N . Our implementation also receives GEMV results from CPU and NMP devices to the GPU in m_u -granularity, which means that Buf should be at least $2m_u$ to temporarily hold GEMV results sent from CPU/NMP.

As an example, consider training OPT-1.3B with $\hat{b}=64$ on 4×A5000 GPUs (24GB each, $\text{Mem}_{\text{total-G}}=96\text{GB}$ across 4 GPUs), 128GB DRAM ($\text{Mem}_{\text{total-C}}=128\text{GB}$), and a 256GB NMP device ($\text{Mem}_{\text{total-N}}=256\text{GB}$). The noise of the $m \approx 1.4\text{B}$ model is sharded across the GPUs, CPU, and NMP memory. Training takes about 72GB across 4 GPUs ($\text{Mem}_{\text{train-G}}=72\text{GB}$) and 22GB of CPU DRAM

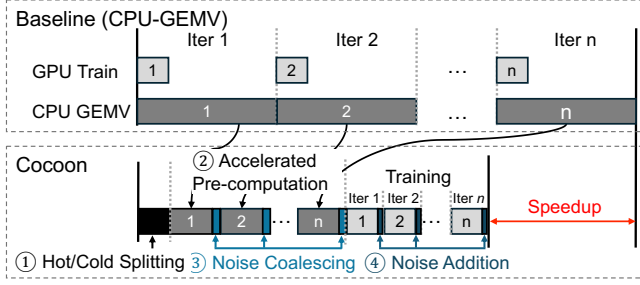


Figure 8: Cocoon’s embedding table optimization. Cocoon replaces the per-iteration CPU-GEMV bottleneck (top) with accelerated noise pre-computation (bottom).

($Mem_{train-C}=22GB$). If we heuristically set $m_G=m_u$ and $B_{uf}=1.5GB$ per GPU, solving the above optimization problem gives $m_u=m_G \approx 71M$, meaning that m_u divides the parameter m into 20 equal slices. GPUs hold one slice across 4 GPUs, CPU holds 5, and NMP can hold the remaining 14. Varying B_{uf} had little effect on the overall performance, as the GPU-side overhead is mostly due to training, which is independent of B_{uf} . Thus, a reasonable value that would not cause OOM suffices.

4.2 Optimizations for Embedding Tables

Figure 8 summarizes Cocoon’s optimization for embedding tables. For embedding tables, Cocoon ① splits the table entries by access frequency into either hot or cold, ② efficiently pre-computes all the correlated noises to be used for the cold entries, ③ coalesces and stores the noises in a compact format, and ④ runs training using the pre-computed noises.

Noise pre-computing. Instead of performing GEMV on each iteration (Figure 8, top), Cocoon *pre-computes* correlated noises for all the future iterations of the embedding tables before the actual training (Figure 8, bottom). The pre-computed results cannot be reused across jobs for privacy, so pre-computing must be done efficiently to not bottleneck the entire training. While pre-computing performs the same amount of GEMV as the baselines, it can be done much faster due to two benefits. Compared to CPU-GEMV, pre-computing can use the faster GPU, which idles before the training starts. Compared to GPU-GEMV, Cocoon maximizes data reuse inside the GPU and minimizes data transfer through PCIe with *noise tiling*.

Figure 9 explains noise tiling. Between consecutive iterations, the most recently updated $\hat{b}-2$ out of $\hat{b}-1$ rows of the noise history are reused (Figure 9, left). However, the reused data ($(\hat{b}-2) \times m$) is often too large to be kept inside the GPU, and GPU-GEMV must spill it to main memory between iterations. Instead, Cocoon splits the noise history into smaller tiles and performs noise pre-computing for each tile, while

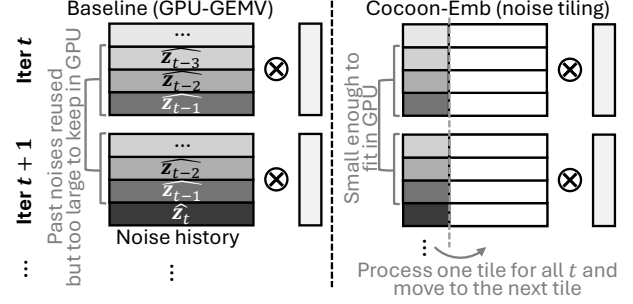


Figure 9: Cocoon’s noise tiling for embedding tables. Cocoon (right) tiles the noise history to fit within GPU memory during pre-computation, avoiding the memory spilling required by the baseline (left) when data exceeds GPU capacity.

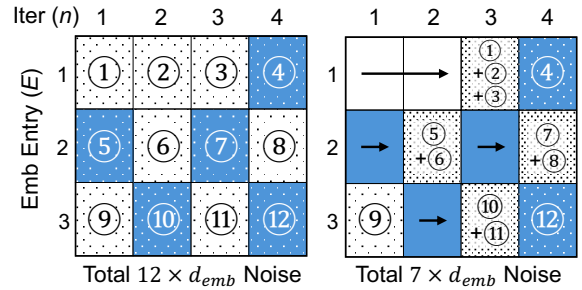


Figure 10: Noise coalescing (d_{emb} : embedding dimension). Cocoon (right) reduces storage overhead (12 to 7 units) by aggregating noise until an entry is accessed (filled blue), replacing the per-iteration baseline (left).

choosing the tile size so that the reused data always fits inside the GPU (Figure 9, right). After generating all future noises from one tile, Cocoon moves on to the next tile. Noise tiling is only possible during pre-computing, where we can choose to compute noises for all future iterations for one tile before computing the next tile. GPU-GEMV cannot benefit from noise tiling, as it must immediately compute the entire noise (*i.e.*, for all the tiles) before proceeding to the next iteration.

Noise coalescing. Without any optimization, the size of the entire pre-computed noises to be used over n training iterations is $m \times n$, which is too large to store. Cocoon uses a technique called *noise coalescing* to solve this issue. As discussed in Section 2.2, only a few entries in embedding tables are used at each iteration, and unused entries do not contribute to that iteration’s gradient. Thus, we do not need to accurately update (*i.e.*, add proper noise) all the entries in every iteration. Instead, it is sufficient to add an equivalent, *aggregated* or *coalesced* noises, as long as they are added before an entry is accessed.

Figure 10 shows a toy example of an embedding table with three entries trained over four iterations. Colored boxes indicate in which iteration each entry is accessed, and dotted

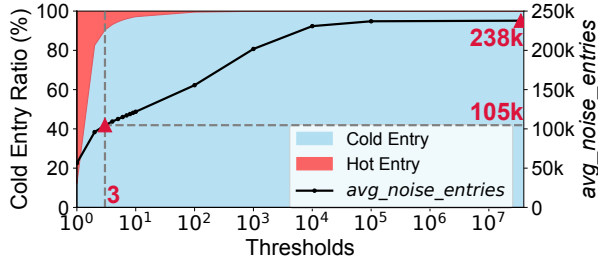


Figure 11: Hot/Cold threshold analysis. Lowering the threshold to 3 reduces *avg_noise_entries* (memory overhead) by $2.3\times$ compared to not using hot/cold splitting.

numbered boxes indicate when noises are added to each entry’s gradient. For example, entry 1 is only accessed in the 4th iteration. Without coalescing (Figure 10, left), noises must be added to all the entries in all the iterations. Instead, noise coalescing (Figure 10, right) only adds an equivalent, aggregated noise right before each entry is accessed or training ends. For example, no noise is added to entry 1 in iterations 1–2, and an equivalent noise (①+②+③) is added at the end of iteration 3. During pre-computing, Cocoon merges and only stores the aggregated noise (e.g., stores ①+②+③ instead of storing three noises separately). In our toy example, only 7 (instead of 12) aggregated noises need to be stored. The benefit is much larger for real models.

Implementing noise coalescing requires knowing exactly when each entry will be accessed during training. This can be known by using a random batch sampler with the same random seed both during pre-computing and training. Cocoon stores the coalesced noise (Figure 10, right) in a compressed sparse column (CSC) format. Cocoon does not pre-compute noises for the rest of the model (e.g., MLP layers) and simply uses GPU-GEMV/CPU-GEMV as they are small.

Hot/cold splitting. The size of the coalesced noise is directly proportional to *avg_noise_entries*, the average number of entries that need noise to be added in each iteration ($avg_noise_entries = \frac{7}{4}$ in our example), which is correlated with the average access frequency of each entry. Typically, most entries are scarcely accessed, but a few “hot” entries [71] drive up *avg_noise_entries*. To reduce *avg_noise_entries*, Cocoon classifies each entry as either “hot” or “cold” and only pre-computes and coalesces noise for cold entries. Hot entries, just like MLP layers, rely on CPU-GEMV/GPU-GEMV. As there are usually only a few hot entries [71], the additional overhead is moderate. We use a simple threshold to classify entries as hot or cold based on their access frequency.

Figure 11 illustrates the relationship between the threshold and the *avg_noise_entries* for Criteo Kaggle [47] dataset. The dataset has 39 million samples, and the model used for this dataset has 33 million unique embedding table entries. A lower threshold labels more entries as hot. For example,

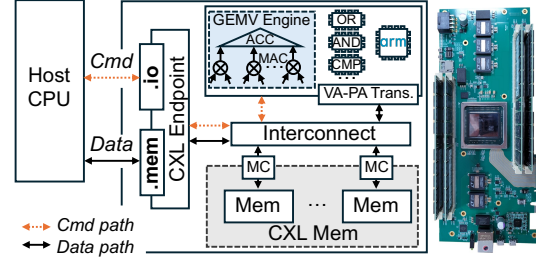


Figure 12: NMP prototype. FPGA-based CXL board featuring an integrated GEMV engine for near-memory acceleration.

using 3 as a threshold labels 7% of the entries as hot, lowering *avg_noise_entries* from 238K to 105K ($2.3\times$ memory reduction), compared to not using hot/cold splitting. We empirically choose the threshold to balance the memory overhead and additional GEMV overhead.

4.3 Leveraging Near-Memory Processing

When the noise history is too large and must be partially stored in CXL memory, Cocoon leverages a near-memory processing (NMP) device if available to greatly reduce the data movement overhead. In particular, we demonstrate the benefit by adopting an FPGA-based industry prototype CXL memory device with NMP capabilities, provided through our industry partner. The benefit of adopting an NMP device will become larger for cases where the CPU is heavily utilized.

NMP device overview. Figure 12 shows the prototype device from our industry partner. The prototype is implemented as an add-in card (AIC)-type custom board that integrates a CXL controller and an NMP engine into a Xilinx Versal (VP1502) FPGA (Figure 12, right). The board is equipped with DDR4 mounted in DIMM slots. The device receives commands from the CPU via CXL.io and data via CXL.mem, and can act either as normal CXL memory or perform GEMV with data in its memory. The hardware is equipped with MAC and ACC (accumulation) hardware IP, which Cocoon uses to run the desired GEMV. The prototype additionally contains the complete set of basic logic blocks required to compose SQL and ML operators, which Cocoon did not use. The hardware performs a simple virtual-to-physical address translation through saving and looking up memory offset for each matrix and using it to locate them in CXL memory. When there are multiple jobs running on the host, their commands are queued and processed in a first-come-first-served fashion.

Integration to Cocoon. Figure 13 illustrates how correlated noise generation is partially offloaded to the NMP device. The $\hat{b} - 1$ past noises have the size of a $(\hat{b} - 1) \times m$ noise history matrix. Noise used at step t is stored at the $(t \bmod (\hat{b} - 1))$ -th row, updating the rows in a circular manner (i.e., storing

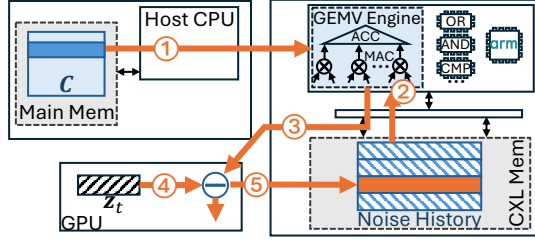


Figure 13: Correlated noise generation with NMP. CPU offloads GEMV to NMP (①–③) with Gaussian noise generation (④) to produce correlated noise that updates the noise history and model (⑤).

noise history in a ring buffer). At each iteration, the CPU ① passes an appropriate mixing vector to the NMP device, ② initiates GEMV between the mixing vector and the noise history, and ③ the GEMV result sent to GPU. ④ the GPU generates Gaussian noise (noise generation can be done by others as well for perfect parallelism, but we use the GPU due to its high throughput). Finally, ⑤ the generated noise updates the noise history and model. These process can be done in parallel while the GPU performs training. Cocoon pre-normalizes the mixing vector ($C[t, t - \tau]$ in Equation 1) and the Gaussian noise (z_t) by the (t, t) -th entry of C prior to GEMV to avoid later scaling. As the noise history table is updated in a circular fashion, the mixing vector must also be properly reordered, which is done statically before training.

Cocoon sends the mixing vector once to the NMP device, stores it in an internal buffer, and reuses it m times. For reasonably large models, this amortizes the vector transfer cost. The GEMV operation is implemented with MAC and ACC (accumulation) hardware IP, and the memory bandwidth is maximized through memory-channel interleaving. The prototype device we used achieved 47.9GB/s peak GEMV throughput. The performance may improve in the future with faster DRAM technologies (*e.g.*, DDR5). While the current implementation detail is specific to the prototype we used, we believe similar integration will be possible with other NMP devices [53, 78, 88], as long as they allow GEMV. Coherence is controlled by flushing at the end of every iteration.

5 Evaluation

5.1 Experimental Setup

Hardware. Most of our evaluation was done on a dual-socket Intel Xeon Gold 6330 CPU with 256GB DRAM and 8 NVIDIA RTX A5000 GPUs. When needed, we ran additional experiments on a more powerful dual-socket AMD EPYC 7763 CPU server with 1TB of DRAM and 8 NVIDIA A100 (80GB) GPUs. We additionally used the industry prototype NMP device. Unless noted otherwise, DLRMs were trained

on a single GPU without NMP, and LLMs were trained on four GPUs with NMP. CPU-side GEMV used Intel MKL (Intel) and OpenBLAS (AMD) in Pytorch (v.2.4.0).

Due to its early-stage engineering, our NMP prototype exhibited a suboptimal memcpy throughput of 5–7GB/s. This is significantly lower than what a typical CXL memory can achieve (32GB/s for gen5 8x). To *emulate* the end-to-end performance of a mature system, we separately measured the NMP device’s overhead and analytically scaled the memcpy overhead assuming a 22GB/s memcpy throughput, which was measured from a similar in-house CXL memory. During training, we run a separate process that adds the estimated NMP device overhead to the training latency, while the rest of the CPU/GPU runs training end-to-end. Although the end-to-end performance is emulated when NMP is involved, except for the memcpy performance, all the overheads were from real measurements. When NMP is not involved, our results are from direct end-to-end measurement. We assume all resources (CPU cores, PCIe bandwidth, CXL memory capacity, *etc.*) are divided evenly across GPUs.

Datasets/models. For DLRM, we used the Criteo Kaggle dataset [47] and the architecture from [71]. We additionally generated synthetic datasets to study the impact of varying the number of embedding entries and data skewness. Synthetic datasets were generated by first ensuring all embedding entries are accessed at least once, and generating the remainder such that the entry accesses follow a Zipfian distribution with a varying α . For non-DLRMs, we used ImageNet [22]-sized dummy data and E2E dataset [73], although the datasets do not impact performance, and architectures from TorchVision/HuggingFace.

Hyperparameters. The training batch size B and band size $\hat{b}$ crucially influence the correlated noise overheads. We used the following values from the literature: $B = 1024$ for vision and language models [6, 8], $B = 65536$ for DLRMs [20], and $\hat{b}=2-256$ [15, 29, 64]. The impact of these hyperparameters is additionally evaluated in the sensitivity studies.

5.2 Performance Improvements: DLRM

5.2.1 End-to-End Training Time

Figure 14 compares the DLRM training time of Cocoon with the baselines. All the bars are normalized to the training time of DP-SGD. For $\hat{b} > 8$, Cocoon consistently outperforms the baselines. Compared to the better baseline, Cocoon improves the overall training time by $2.46-4.87\times$ for $\hat{b} > 8$. The speedup generally increases with $\hat{b}$ ($2.46\times$ for $\hat{b} = 16$ and $4.87\times$ for $\hat{b} = 64$). The breakdown shows that pre-computing dominates the Cocoon-Emb latency.

When $\hat{b} < 8$, the entire noise history fits into the GPU. For these *trivial cases*, the training time of both baselines becomes identical (we only show GPU-GEMV) and close to DP-SGD, and our optimization simply adds unnecessary overheads.

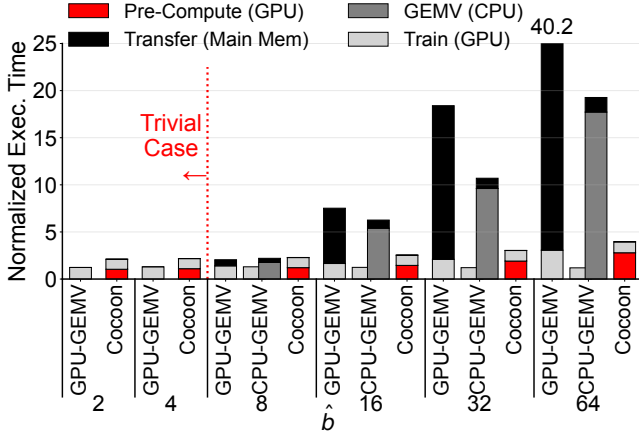


Figure 14: Normalized training time of Cocoon with DLRM. When the entire noise history fits into GPU memory (trivial case; $\hat{b}=2-4$), our optimization can be turned off.

Such trivial cases can be easily detected by comparing the noise history size with the GPU memory capacity, and our optimization can be turned off.

5.2.2 Sensitivity Study

Figure 15 shows the speedup of Cocoon over the better baseline between CPU-GEMV and GPU-GEMV, while varying different dimensions of the model and dataset. Again, bars located at the left side of the red vertical line are trivial cases, and our embedding table optimizations can be turned off.

Model size. Figures 15a and 15b show the speedup of Cocoon while varying the model size, adjusting the embedding dimension (d_{emb}) or the number of embedding entries. The speedup improves with the model size: if we compare the bars at $\hat{b} = 32$, the speedup improves from $3.51\times$ to $6.27-6.35\times$ when the size is doubled, and reduced to $1.37\times$ when halved. This is because larger models must offload more noise history to main memory and penalize the baselines more severely. Cocoon will become more effective as models grow.

Batch size. Figure 15c shows that the speedup decreases with an increasing batch size. When considering $\hat{b} = 32$, the peak speedup increased from $3.51\times$ with $B = 64K$ to $4.79\times$ with $B = 32K$, and reduced to $2.57\times$ with $B = 128K$. This is because the correlated noise generation overhead (which Cocoon optimizes) stays the same regardless of the batch size, while the training latency (which Cocoon cannot optimize) becomes larger with bigger B . While not shown, Doubling the entries accessed per sample (*i.e.*, pooling factor [71]) has nearly the same effect as doubling the batch size.

Skewness. Figure 15d shows the speedup of Cocoon when the access frequency of each embedding entry experiences different skewness. The skewness was controlled by varying α of the Zipfian distribution of our synthetic dataset. Interestingly, the skewness had only a minor effect on training time.

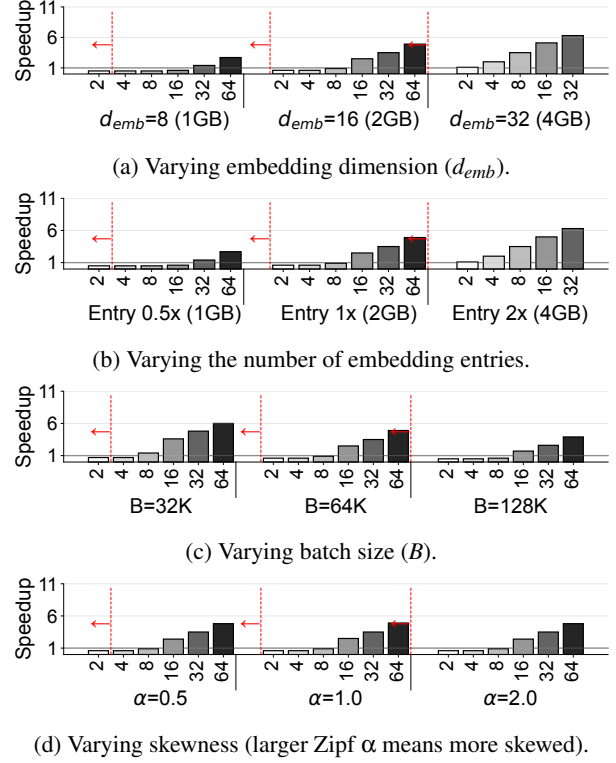


Figure 15: Speedup of Cocoon under various models and datasets. Numbers below each bar indicates $\hat{b}$.

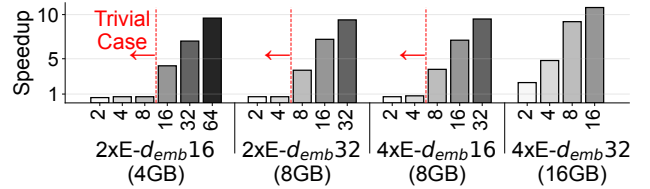


Figure 16: Speedup of Cocoon with various model sizes on an A100 GPU. Numbers below each bar indicates $\hat{b}$.

We will later show that skewness is a critical factor in the memory footprint in Section 5.3.

Hardware. Figure 16 shows the speedup of Cocoon on more powerful A100 GPUs. As A100 has more memory, we used larger models (4–16GB) with a larger number of embedding entries (2–4× larger, denoted as 2×E/4×E) and bigger embedding dimensions ($d_{emb}=16-32$). Cocoon achieved a speedup of $2.33-10.82\times$ for non-trivial cases, which is generally larger than the results on the A5000. Cocoon’s speedup is larger on A100 because its computation (GPU-side GEMV and training) is much faster than on the A5000, but overheads related to correlated noises (GPU-main memory data transfer, CPU-side GEMV), which Cocoon optimizes, are similar.

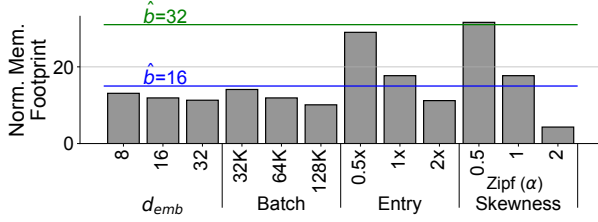


Figure 17: Memory footprint of coalesced noise normalized by the model size. Memory footprint of a noise history without pre-computing for different $\hat{b}$ are in horizontal lines.

5.3 Memory Overhead: DLRM

5.3.1 Overall Memory Overhead

Unlike CPU-GEMV and GPU-GEMV, which incur a fixed $O(\hat{b}m)$ memory overhead, the overhead of Cocoon with its embedding table optimization depends on the effectiveness of its noise coalescing. In the worst case, Cocoon must hold all the pre-computed noises ($O(nm)$ with n iterations), which can be much larger than that of CPU-GEMV/GPU-GEMV (usually, $n \gg \hat{b}$). However, the actual memory overhead is much less thanks to noise coalescing.

Figure 17 evaluates the memory footprint of the coalesced noise of Cocoon while varying the embedding dimension (d_{emb}), batch size, number of embedding entries, and entry access distribution skewness. The bars are normalized to the model size m . For this figure, we used $n = 1800$ (three epochs using Criteo Kaggle [47] dataset with $B = 64K$), so the worst-case overhead is $1800 \times m$. However, the actual memory overhead is only $4.3\text{--}31.6\times$, which is *less* than the memory overhead of the baselines in many cases (shown in horizontal lines for $\hat{b}=16$ and $\hat{b}=32$). The memory overhead of Cocoon is independent of $\hat{b}$ and only depends on the entry access pattern, while the overheads of the baselines grow linearly with $\hat{b}$.

5.3.2 Sensitivity Study

Figure 17 shows how different models and datasets affect the efficacy of noise coalescing. It can be seen that the efficacy decreases with reducing d_{emb} and batch size, but the effect is small. Conversely, decreasing the number of embedding entries and using datasets with less skewed patterns significantly increases the memory overhead. This meets our expectation because noise coalescing works better when batched samples are mostly accessing the same entries, which leads to lower *avg_noise_entries*. Decreasing the number of entries has a similar effect to reduced skewness, because the accesses are hashed into the remaining entries.

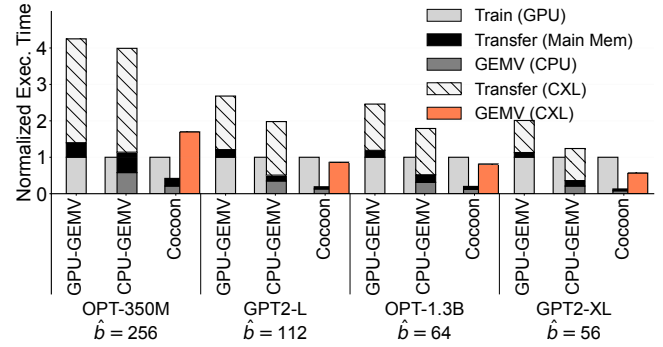


Figure 18: End-to-end normalized training time of Cocoon +NMP and the baselines. Model sizes are in an ascending order. $\hat{b}$ is chosen, so that over 200GB of the noise history is offloaded to CXL memory.

Table 1: Hardware cost and power estimate (GPT2-XL, $\hat{b}=64$).

1 HW Unit: 4×NVIDIA A5000 (4×\$2K), 1× Intel Xeon Gold 6330 (1×\$2.1K), 8×16GB DDR4 3200MHz (8×\$0.15K)			
	GPU-Only	CPU-GEMV	Cocoon
HW Unit	10	3	1 + NMP [78]
Cost (\$)	113K	33.9K	18.3K
Peak Power (KW)	11.6	3.5	1.2
Normalized Throughput	1	0.33	0.13

5.4 Performance Improvement with NMP

5.4.1 End-to-End Training Time

Figure 18 plots the training time and breakdown for the baselines and Cocoon with the NMP hardware prototype, when the models are large enough to involve CXL memory. Cocoon has three bars side-by-side, indicating the GPU-side training, CPU-side GEMV, and the GEMV happening inside the NMP device, all happening in parallel. $\hat{b}$ values are chosen for each model to ensure over 200GB of the noise history is offloaded to CXL memory, to avoid trivial, uninteresting setups.

Figure 18 shows that Cocoon consistently outperforms the baselines, achieving **1.23–2.32×** speedup compared to the better baseline. Cocoon achieves high speedup by eliminating the large data transfer overhead between the CXL memory and CPU/GPU (“Transfer (CXL)”), while incurring a moderate GEMV overhead inside the CXL controller (“GEMV (CXL)”). When the GEMV overhead of Cocoon is less than the training time and can be completely hidden (GPT2-L, OPT-1.3B, GPT2-XL), training becomes the critical path. Our noise distribution strategy minimizes the overall latency, trying to put the minimum noise history on the bottlenecked device.

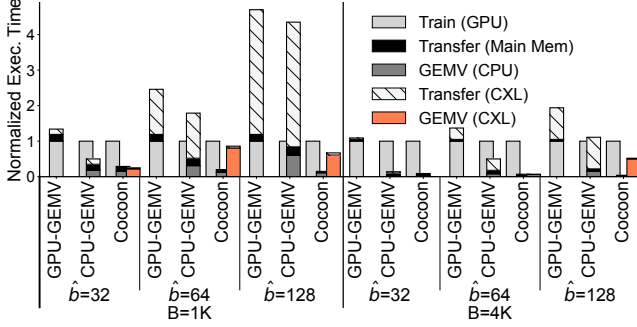


Figure 19: End-to-end normalized training time of Cocoon+NMP and the baselines with OPT-1.3B. $\hat{b} = 128$ number assumes a hypothetical CXL device with an expanded memory capacity, as it does not fit into our current CXL prototype.

5.4.2 Power and Cost

Table 1 compares the estimated hardware cost and peak power of Cocoon with the NMP device against two alternative setups (GPU-Only, CPU-GEMV), which do not involve CXL memory and instead scale out the GPU (GPU-Only) or CPU (CPU-GEMV) to hold the large noise history. GPU-Only represents prior works’ setups [63, 79] where enough GPUs hold the entire noise history. We assume GPT2-XL training with $\hat{b}=64$ (~413GB of noise history). As there can be many different ways to scale out, we simplify the estimation by considering our current single-socket configuration (4x A5000 + 1x Intel Xeon Gold + 128GB memory) as one hardware unit, and estimating how many such units are needed to support noise generation. As our prototype NMP device does not have a public power/cost number, we use numbers from prior work that used a similar device [78]. The device from [78] is more powerful than our prototype, so the power/cost numbers are likely to be an *upper bound*. We do not model the extra cost (e.g., network, power supply) of scaling out beyond a single node. All of these simplistic estimations *favor other baselines*, and Cocoon’s actual benefit is likely to be higher.

Table 1 shows that Cocoon can realize training with much less power and hardware cost. This is because instead of using expensive GPU/CPU memory, Cocoon can use CXL NMP memory to partially host and process noise history, which is expected to be much cheaper and lower power than CPU/GPU [78]. However, as the throughput also increases with the hardware units, the baselines also achieve better throughput. If you compare the ratio between the normalized throughput numbers and the cost/power, Cocoon is similar to CPU-GEMV, while being much more cost/power-efficient than GPU-Only. This is because GPU-GEMV introduces too many GPUs, and GPUs can become underutilized. The result indicates that scaling out CPU/GPU is better when high throughput is the main goal, while Cocoon with NMP can be useful if one has a limit on the power/cost budget.

5.4.3 Sensitivity Study

Model size. Figure 18 also shows that smaller models (models to the left) achieve more speedup than larger models (models to the right), when the noise history size inside the CXL memory is similar. This is because the training time ultimately becomes the major bottleneck and limits achievable speedup in larger models.

Band size. Figure 19 plots the end-to-end training time and breakdown for OPT-1.3B while varying $\hat{b}$. When using $\hat{b} = 128$, the noise history is too large to fit into our CXL memory, and we cannot currently run it on our CXL prototype. For $\hat{b} = 128$, we provide an analytical projection assuming a hypothetical future device with expanded memory capacity, but otherwise identical characteristics (GEMV throughput, interconnect bandwidth, etc.). This additional datapoint shows how the idea can scale to larger CXL memory devices in the future to support even larger models and band sizes. When $\hat{b}$ is small (e.g., $\hat{b}=32$), Cocoon and CPU-GEMV both perform similarly to DP-SGD, because the training time dominates and all the other overheads are hidden. With larger $\hat{b}$, both the baseline and Cocoon start to incur slowdown, and Cocoon outperforms the best baseline by $2.26\times$.

Batch size. Figure 19 shows that the speedup decreases when we increase the batch size from $B=1K$ to $B=4K$. This is because the training time increases with larger batch sizes, and the rest of the overheads can be hidden behind this increased training time. With $B=4K$, Cocoon showed speedup over the best baseline only when $\hat{b}=64$; otherwise, both Cocoon and the baseline performed on-par. The result indicates that Cocoon’s benefit will decrease when using larger batches, but would still show speedup when $\hat{b}$ is large enough.

5.5 Performance Improvement: DLRM+NMP

Our current evaluation only covers the use of NMP with LLMs, not DLRMs. This is because our NMP prototype’s limited GEMV throughput was insufficient to benefit DLRM, whose single-iteration latency was on the order of hundreds of milliseconds. Currently, all the GEMV for DLRM (during both pre-compute and actual training) is done on the GPU to operate near this timescale, as detailed in Section 4.2. However, DLRM can also benefit from NMP if its GEMV throughput improves in the future. For example, [78] claimed that building an NMP device with TB/s of internal bandwidth is possible. If such a high bandwidth is indeed possible, we can use the NMP device to accelerate pre-computing of the noise history for the embedding tables. If we assume the throughput from [78], our analytical projection shows that DLRM training would be accelerated by an additional $2.4\times$.

6 Related Work

DP-SGD for large models. In the earlier days of DP training research, it was thought that DP training only works well for small models [21]. Recently, many studies have shown that DP training can work well for larger foundation models [4, 7, 8, 11, 19, 40, 43, 61, 64, 98, 104, 105, 112] and DLRMs [18, 20, 23, 30, 62, 72] as well. Google recently released an LLM that was fully trained from scratch with DP [89].

System optimizations for DP training. Several works studied how DP-SGD can be made faster through optimizing the software [6, 8, 58, 95] and hardware [75]. These works mainly focused on efficiently calculating the per-example gradient and are orthogonal to this work. There are also works that studied how to accelerate DP-SGD for DLRMs [30, 62, 72]. LazyDP [62], the closest to our work, also leveraged the fact that one can defer adding noise until an entry is accessed and add an equivalent, aggregated noise. However, [62] relies on the fact that the sum of independent Gaussians is also a Gaussian, and it cannot be extended to (non-independent) correlated noises. Our optimization for DLRM is entirely different from LazyDP. The others [30, 72] modify the DP algorithm itself, affecting privacy and accuracy.

Near/in-memory processing. Near-memory processing (NMP) runs memory-intensive workloads closer to memory. Prior works explored running compute inside the CXL controller (e.g., for LLM inference [35, 78], vector database search [53, 88], and DLRM inference [51, 107]), DIMM (e.g., for DLRM [52, 56, 77] and database operations [52]), network switches [44, 45], and SSD controllers [90, 100]. Processing-in-memory (PIM) embeds compute logic directly in the memory hardware to enjoy even higher bandwidth [42, 57, 59, 76, 106]. Cocoon leverages NMP on a CXL controller for correlated noise generation.

6.1 Applicability to Large-scale Training

While our evaluation focuses on single-node configurations (economical setups), the idea of Cocoon naturally extends to multiple nodes, because correlated noise generation of each model dimension is independent (noise generation for different dimensions can be easily parallelized across nodes). However, if additional nodes provide sufficient aggregate CPU/GPU memory, the relative benefit of the CXL NMP device diminishes. We leave the exploration of these scaled-out setups as future work.

7 Conclusion

DP training with correlated noise is an emerging technique whose system implications have yet to be thoroughly studied. We conducted a systematic study of the new technique and found several major bottlenecks when applied to DLRMs and billion-parameter models on an economical setup. We

introduced Cocoon, a framework for efficient DP training with correlated noise. When baseline approaches fail to deliver competitive performance, hardware/software designs of Cocoon can deliver $1.23\text{--}10.82\times$ speedup.

Acknowledgments

This work was partly supported by the U.S. National Science Foundation under award No. CNS-2349610 and CCF-2529883. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

References

- [1] Martin Abadi, Andy Chu, Ian Goodfellow, H Brendan McMahan, Ilya Mironov, Kunal Talwar, and Li Zhang. Deep learning with differential privacy. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2016.
- [2] Abdelrahman Abouelenin, Mohamed Abdelrehim, Raffy Fahim, Amr Hendy, and Mohamed Afify. Privacy-preserving transformers: SwiftKey’s differential privacy implementation. In *arxiv.org*, 2025.
- [3] John M Abowd. The US Census Bureau adopts differential privacy. In *Proceedings of the ACM SIGKDD international conference on knowledge discovery & data mining*, 2018.
- [4] Sebastian Rodriguez Beltran, Marlon Tobaben, Joonas Jälkö, Niki Loppi, and Antti Honkela. Towards efficient and scalable training of differentially private deep learning. In *arxiv.org*, 2024.
- [5] Zhiqi Bu, Justin Chiu, Ruixuan Liu, Yu-Xiang Wang, Sheng Zha, and George Karypis. Zero redundancy distributed learning with differential privacy. In *ICLR Workshop on Pitfalls of limited data and computation for Trustworthy ML*, 2023.
- [6] Zhiqi Bu, Jialin Mao, and Shiyun Xu. Scalable and efficient training of large convolutional neural networks with differential privacy. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2022.
- [7] Zhiqi Bu, Yu-Xiang Wang, Sheng Zha, and George Karypis. Automatic Clipping: Differentially private deep learning made easier and stronger. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2023.
- [8] Zhiqi Bu, Yu-Xiang Wang, Sheng Zha, and George Karypis. Differentially private optimization on large

- model at small cost. In *Proceedings of the International Conference on Machine Learning (ICML)*, 2023.
- [9] Nicholas Carlini, Florian Tramer, Eric Wallace, Matthew Jagielski, Ariel Herbert-Voss, Katherine Lee, Adam Roberts, Tom Brown, Dawn Song, Ulfar Erlingsson, Alina Oprea, and Colin Raffel. Extracting training data from large language models. In *Proceedings of the USENIX Security Symposium*, 2021.
 - [10] Nicolas Carlini, Jamie Hayes, Milad Nasr, Matthew Jagielski, Vikash Sehwal, Florian Tramer, Borja Balle, Daphne Ippolito, and Eric Wallace. Extracting training data from diffusion models. In *Proceedings of the USENIX Security Symposium*, 2023.
 - [11] Zachary Charles, Arun Ganesh, Ryan McKenna, H. Brendan McMahan, Nicole Mitchell, Krishna Pillutla, and Keith Rush. Fine-tuning large language models with user-level differential privacy. In *arxiv.org*, 2024.
 - [12] Jay Chooi, Kevin Cong, Russell Li, and Lillian Sun. DP-AdamW: Investigating decoupled weight decay and bias correction in private deep learning. In *arxiv.org*, 2025.
 - [13] Christopher A Choquette-Choo, Krishnamurthy Dvijotham, Krishna Pillutla, Arun Ganesh, Thomas Steinke, and Abhradeep Thakurta. Correlated noise provably beats independent noise for differentially private learning. In *arxiv.org*, 2023.
 - [14] Christopher A. Choquette-Choo, Arun Ganesh, Saminul Haque, Thomas Steinke, and Abhradeep Guha Thakurta. Near-exact privacy amplification for matrix mechanisms. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2025.
 - [15] Christopher A Choquette-Choo, Arun Ganesh, Ryan McKenna, H Brendan McMahan, John Rush, Abhradeep Guha Thakurta, and Zheng Xu. (Amplified) Banded Matrix Factorization: A unified approach to private training. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2023.
 - [16] Christopher A. Choquette-Choo, H. Brendan McMahan, Keith Rush, and Abhradeep Thakurta. Multi-epoch matrix factorization mechanisms for private machine learning. In *Proceedings of the International Conference on Machine Learning (ICML)*, 2023.
 - [17] Christopher A. Choquette-Choo, Florian Tramèr, Nicholas Carlini, and Nicolas Papernot. Label-only membership inference attacks. In *Proceedings of the International Conference on Machine Learning (ICML)*, 2021.
 - [18] Lynn Chua, Qiliang Cui, Badih Ghazi, Charlie Harrison, Pritish Kamath, Walid Krichene, Ravi Kumar, Pasin Manurangsi, Nicolas Mayoraz, Hema Venkata Krishna Giri Narra, Steffen Rendle, Amer Sinha, Avinash V. Varadarajan, and Chiyuan Zhang. Training differentially private ad prediction models with semi-sensitive features. In *Proceedings of the Workshop on Data Mining for Online Advertising (AdKDD)*, 2024.
 - [19] Lynn Chua, Badih Ghazi, Yangsibo Huang, Pritish Kamath, Ravi Kumar, Daogao Liu, Pasin Manurangsi, Amer Sinha, and Chiyuan Zhang. Mind the privacy unit! User-level differential privacy for language model fine-tuning. In *Conference on Language Modeling (COLM)*, 2024.
 - [20] Lynn Chua, Badih Ghazi, Pritish Kamath, Ravi Kumar, Pasin Manurangsi, Amer Sinha, and Chiyuan Zhang. Scalable dp-sgd: Shuffling vs. poisson subsampling. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2024.
 - [21] Soham De, Leonard Berrada, Jamie Hayes, Samuel L Smith, and Borja Balle. Unlocking high-accuracy differentially private image classification through scale. In *arxiv.org*, 2022.
 - [22] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. ImageNet: A large-scale hierarchical image database. In *Proceedings of the Conference on Computer Vision and Pattern Recognition (CVPR)*, 2009.
 - [23] Carson Denison, Badih Ghazi, Pritish Kamath, Ravi Kumar, Pasin Manurangsi, Krishna Giri Narra, Amer Sinha, Avinash V. Varadarajan, and Chiyuan Zhang. Private ad modeling with DP-SGD. In *Proceedings of the Workshop on Data Mining for Online Advertising (AdKDD)*, 2023.
 - [24] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, Jakob Uszkoreit, and Neil Houlsby. An image is worth 16x16 words: Transformers for image recognition at scale. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2020.
 - [25] Cynthia Dwork. Differential privacy. In *International Colloquium on Automata, Languages, and Programming (ICALP)*, 2006.
 - [26] Úlfar Erlingsson, Vasyl Pihur, and Aleksandra Korolova. Rappor: Randomized aggregatable privacy-preserving ordinal response. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2014.

- [27] Vitaly Feldman, Ilya Mironov, Kunal Talwar, and Abhradeep Thakurta. Privacy Amplification by Iteration. In *IEEE Annual Symposium on Foundations of Computer Science (FOCS)*, 2018.
- [28] Matt Fredrikson, Somesh Jha, and Thomas Ristenpart. Model inversion attacks that exploit confidence information and basic countermeasures. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2015.
- [29] Arun Ganesh, Brendan McMahan, and Abhradeep Thakurta. On design principles for private adaptive optimizers. In *arxiv.org*, 2025.
- [30] Badih Ghazi, Yangsibo Huang, Pritish Kamath, Ravi Kumar, Pasin Manurangsi, Amer Sinha, and Chiyuan Zhang. Sparsity-preserving differentially private training of large embedding models. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2023.
- [31] Antonio Ginart, Laurens van der Maaten, James Zou, and Chuan Guo. Submix: Practical private prediction for large-scale language models. In *arxiv.org*, 2022.
- [32] Google. TensorFlow Privacy. 2025.
- [33] Filip Granqvist, Congzheng Song, Áine Cahill, Rogier van Dalen, Martin Pelikan, Yi Sheng Chan, Xiaojun Feng, Natarajan Krishnaswami, Vojta Jina, and Mona Chitnis. pfl-research: simulation framework for accelerating research in private federated learning. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2025.
- [34] Xin Gu, Gautam Kamath, and Zhiwei Steven Wu. Choosing public datasets for private machine learning via gradient subspace distance. In *IEEE Conference on Secure and Trustworthy Machine Learning (SaTML)*, 2025.
- [35] Yufeng Gu, Alireza Khadem, Sumanth Umesh, Ning Liang, Xavier Servot, Onur Mutlu, Ravi Iyer, and Reetuparna Das. PIM is all you need: A CXL-enabled GPU-free system for large language model inference. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operation Systems (ASPLOS)*, 2025.
- [36] Chuan Guo, Brian Karrer, Kamalika Chaudhuri, and Laurens Van der Maaten. Bounding training data reconstruction in private (deep) learning. In *Proceedings of the International Conference on Machine Learning (ICML)*, 2022.
- [37] Chuan Guo, Alexandre Sablayrolles, and Maziar Sanjabi. Analyzing privacy leakage in machine learning via multiple hypothesis testing: A lesson from fano. In *Proceedings of the International Conference on Machine Learning (ICML)*, 2023.
- [38] Udit Gupta, Samuel Hsia, Vikram Saraph, Xiaodong Wang, Brandon Reagen, Gu-Yeon Wei, Hsien-Hsin S. Lee, David Brooks, and Carole-Jean Wu. DeepRecSys: A system for optimizing end-to-end at-scale neural recommendation inference. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2020.
- [39] Jamie Hayes, Borja Balle, and Saeed Mahloujifar. Bounding training data reconstruction in DP-SGD. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2023.
- [40] Jiyan He, Xuechen Li, Da Yu, Huishuai Zhang, Janardhan Kulkarni, Yin Tat Lee, Arturs Backurs, Nenghai Yu, and Jiang Bian. Exploring the limits of differentially private deep learning with group-wise clipping. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2023.
- [41] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep Residual Learning for Image Recognition. In *Proceedings of the Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016.
- [42] Yintao He, Haiyu Mao, Christina Giannoula, Mohammad Sadrosadati, Juan Gómez-Luna, Huawei Li, Xiaowei Li, Ying Wang, and Onur Mutlu. PAPI: Exploiting dynamic parallelism in large language model decoding with a processing-in-memory-enabled computing system. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operation Systems (ASPLOS)*, 2025.
- [43] Yuzheng Hu, Fan Wu, Ruicheng Xian, Yuhang Liu, Lydia Zakynthinou, Pritish Kamath, Chiyuan Zhang, and David A. Forsyth. Empirical privacy variance. In *Proceedings of the International Conference on Machine Learning (ICML)*, 2025.
- [44] Wenqin Huangfu, Krishna T Malladi, Andrew Chang, and Yuan Xie. Beacon: Scalable near-data-processing accelerators for genome analysis near memory pool with the cxl support. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, 2022.
- [45] Pingyi Huo, Anusha Devulapally, Hasan Al Maruf, Minseo Park, Krishnakumar Nair, Meena Arunachalam, Gulsum Gudukbay Akbulut, Mahmut Taylan Kandemir, and Vijaykrishnan Narayanan. Pifs-rec: Process-in-fabric-switch for large-scale recommendation system inferences. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, 2024.

- [46] SK Hynix. CMM-Ax. https://www.linkedin.com/posts/sk-hynix_sk-hynix-cmm-ax-activit-y-7262426491148222465-sAbH/, 2024.
- [47] Kaggle. Criteo Display Advertising Challenge, 2014.
- [48] Peter Kairouz, Brendan McMahan, Shuang Song, Om Thakkar, Abhradeep Thakurta, and Zheng Xu. Practical and private (deep) learning without sampling or shuffling. In *Proceedings of the International Conference on Machine Learning (ICML)*, 2021.
- [49] Peter Kairouz, Sewoong Oh, and Pramod Viswanath. The composition theorem for differential privacy. In *Proceedings of the International Conference on Machine Learning (ICML)*, 2015.
- [50] Sanjay Kariyappa, Chuan Guo, Kiwan Maeng, Wenjie Xiong, G. Edward Suh, Moinuddin K. Qureshi, and Hsien-Hsin S. Lee. Cocktail party attack: Breaking aggregation-based privacy in federated learning using independent component analysis. In *Proceedings of the International Conference on Machine Learning (ICML)*, 2023.
- [51] Liu Ke, Udit Gupta, Carole-Jean Wu, Benjamin Youngjae Cho, Mark Hempstead, Brandon Reagen, Xuan Zhang, David Brooks, Vikas Chandra, Utku Diril, Amin Firoozshahian, Kim Hazelwood, Bill Jia, Hsien-Hsin S. Lee, Meng Li, Bert Maher, Dheevatsa Mudigere, Maxim Naumov, Martin Schatz, Mikhail Smelyanskiy, and Xiaodong Wang. Recnmp: Accelerating personalized recommendation with near-memory processing. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2020.
- [52] Liu Ke, Xuan Zhang, Jinin So, Jong-Geon Lee, Shin-Haeng Kang, Sukhan Lee, Songyi Han, YeonGon Cho, Jin Hyun Kim, Yongsuk Kwon, KyungSoo Kim, Jin Jung, Ilkwon Yun, Sung Joo Park, Hyunsun Park, Joonho Song, Jeonghyeon Cho, Kyomin Sohn, Nam Sung Kim, and Hsien-Hsin S. Lee. Near-memory processing in action: Accelerating personalized recommendation with axdim. *IEEE Micro*, 2022.
- [53] Seoyoung Ko, Hyunjeong Shim, Wanju Doh, Sungmin Yun, Jinin So, Yongsuk Kwon, Sang-Soo Park, Si-Dong Roh, Minyong Yoon, Taeksang Song, and Jung Ho Ahn. COSMOS: A CXL-based full in-memory system for approximate nearest neighbor search. In *IEEE Computer Architecture Letters*, 2025.
- [54] Tatsuki Koga, Ruihan Wu, Zhiyuan Zhang, and Kamalika Chaudhuri. Privacy-preserving retrieval-augmented generation with differential privacy. In *arxiv.org*, 2025.
- [55] Anastasiia Koloskova, Ryan McKenna, Zachary Charles, John Rush, and H Brendan McMahan. Gradient descent with linearly correlated noise: Theory and applications to differential privacy. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2023.
- [56] Youngeun Kwon, Yunjae Lee, and Minsoo Rhu. TensorDIMM: A practical near-memory processing architecture for embeddings and tensor operations in deep learning. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, 2019.
- [57] Hyojung Lee, Daehyeon Baek, Jimyong Son, Jieun Choi, Kihyo Moon, and Minsung Jang. PAISE: PIM-accelerated inference scheduling engine for transformer-based LLM. In *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*, 2025.
- [58] Jaewoo Lee and Daniel Kifer. Scaling up differentially private deep learning with fast per-example gradient clipping. *Proceedings on Privacy Enhancing Technologies*, 2021.
- [59] Sukhan Lee, Shin-haeng Kang, Jaehoon Lee, Hyeonsu Kim, Eojin Lee, Seungwoo Seo, Hosang Yoon, Seungwon Lee, Kyoungwan Lim, Hyunsung Shin, Jinhyun Kim, O Seongil, Anand Iyer, David Wang, Kyomin Sohn, and Nam Sung Kim. Hardware architecture and software stack for PIM based on commercial DRAM technology : Industrial product. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2021.
- [60] Kif Leswing. Facebook says Apple iOS privacy change will result in \$10 billion revenue hit this year. <https://www.cnbc.com/2022/02/02/facebook-says-apple-ios-privacy-change-will-cost-10-billion-this-year.html>, 2022.
- [61] Xuechen Li, Florian Tramèr, Percy Liang, and Tatsunori Hashimoto. Large language models can be strong differentially private learners. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2022.
- [62] Juntaek Lim, Youngeun Kwon, Ranggi Hwang, Kiwan Maeng, Edward Suh, and Minsoo Rhu. LazyDP: Co-designing algorithm-software for scalable training of differentially private recommendation models. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operation Systems (ASPLOS)*, 2024.
- [63] Ryan McKenna. Scaling up the banded matrix factorization mechanism for differentially private ML. In *arxiv.org*, 2024.

- [64] Ryan McKenna, Yangsibo Huang, Amer Sinha, Borja Balle, Zachary Charles, Christopher A. Choquette-Choo, Badih Ghazi, George Kaissis, Ravi Kumar, Ruiho Liu, Da Yu, and Chiyuan Zhang. Scaling laws for differentially private language models. In *arxiv.org*, 2025.
- [65] H. Brendan McMahan, Zheng Xu, and Yanxiang Zhang. A hassle-free algorithm for private learning in practice: Don’t use tree aggregation, use BLTs. In *arxiv.org*, 2024.
- [66] Assaf Hurwitz Michaely, Xuedong Zhang, Gabor Simko, Carolina Parada, and Petar Aleksic. Keyword spotting for Google assistant using contextual speech recognition. In *IEEE Automatic Speech Recognition and Understanding Workshop (ASRU)*, 2017.
- [67] Milad Nasr, Saeed Mahloujifar, Xinyu Tang, Prateek Mittal, and Amir Houmansadr. Effectively using public data in privacy preserving machine learning. In *Proceedings of the International Conference on Machine Learning (ICML)*, 2023.
- [68] Milad Nasr, Javier Rando, Nicholas Carlini, Jonathan Hayase, Matthew Jagielski, A Feder Cooper, Daphne Ippolito, Christopher A Choquette-Choo, Florian Tramèr, and Katherine Lee. Scalable extraction of training data from aligned, production language models. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2025.
- [69] Milad Nasr, Reza Shokri, and Amir Houmansadr. Comprehensive privacy analysis of deep learning: Passive and active white-box inference attacks against centralized and federated learning. In *IEEE Symposium on Security and Privacy (SP)*, 2019.
- [70] Milad Nasr, Shuang Song, Abhradeep Thakurta, Nicolas Papernot, and Nicholas Carlin. Adversary instantiation: Lower bounds for differentially private machine learning. In *IEEE Symposium on Security and Privacy (SP)*, 2021.
- [71] Maxim Naumov, Dheevatsa Mudigere, Hao-Jun Michael Shi, Jianyu Huang, Narayanan Sundaraman, Jongsoo Park, Xiaodong Wang, Udit Gupta, Carole-Jean Wu, Alisson G. Azzolini, Dmytro Dzhulgakov, Andrey Mallevich, Ilia Cherniavskii, Yinghai Lu, Raghuraman Krishnamoorthi, Ansha Yu, Volodymyr Kondratenko, Stephanie Pereira, Xianjie Chen, Wenlin Chen, Vijay Rao, Bill Jia, Liang Xiong, and Misha Smelyanskiy. Deep learning recommendation model for personalization and recommendation systems. In *arxiv.org*, 2019.
- [72] Lin Ning, Steve Chien, Shuang Song, Mei Chen, Yunqi Xue, and Devora Berlowitz. EANA: Reducing privacy risk on large-scale recommendation models. In *Proceedings of the ACM Conference on Recommender Systems (RecSys)*, 2022.
- [73] Jekaterina Novikova, Ondřej Dušek, and Verena Rieser. The E2E dataset: New challenges for end-to-end generation. In *Proceedings of the 18th Annual SIGdial Meeting on Discourse and Dialogue*, 2017.
- [74] Nicolas Papernot, Abhradeep Thakurta, Shuang Song, Steve Chien, and Úlfar Erlingsson. Tempered sigmoid activations for deep learning with differential privacy. In *Proceedings of the AAAI Conference on Artificial Intelligence*, 2021.
- [75] Beomsik Park, Ranggi Hwang, Dongho Yoon, Yoonhyuk Choi, and Minsoo Rhu. DiVa: An accelerator for differentially private machine learning. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, 2022.
- [76] Jaehyun Park, Jaewan Choi, Kwanhee Kyung, Michael Jaemin Kim, Yongsuk Kwon, Nam Sung Kim, and Jung Ho Ahn. AttAcc! unleashing the power of pim for batched transformer-based generative model inference. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operation Systems (ASPLOS)*, 2024.
- [77] Jaehyun Park, Byeongho Kim, Sungmin Yun, Eojin Lee, Minsoo Rhu, and Jung Ho Ahn. TRiM: Enhancing processor-memory interfaces with scalable tensor reduction in memory. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, 2021.
- [78] Sang-Soo Park, KyungSoo Kim, Jinin So, Jin Jung, Jonggeon Lee, Kyoungwan Woo, Nayeon Kim, Younghyun Lee, Hyungyo Kim, Yongsuk Kwon, Jinhyun Kim, Jieun Lee, YeonGon Cho, Yongmin Tai, Jeonghyeon Cho, Hoyoung Song, Jung Ho Ahn, and Nam Sung Kim. An LPDDR-based CXL-PNM platform for TCO-efficient inference of transformer-based large language models. In *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*, 2024.
- [79] Krishna Pillutla, Jalaj Upadhyay, Christopher A. Choquette-Choo, Krishnamurthy Dvijotham, Arun Ganesh, Monika Henzinger, Jonathan Katz, Ryan McKenna, H. Brendan McMahan, Keith Rush, Thomas Steinke, and Abhradeep Thakurta. Correlated noise mechanisms for differentially private learning. In *arxiv.org*, 2025.

- [80] Natalia Ponomareva, Sergei Vassilvitskii, Zheng Xu, Brendan McMahan, Alexey Kurakin, and Chiyuan Zhang. How to DP-fy ML: A practical tutorial to machine learning with differential privacy. In *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, 2023.
- [81] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners. *OpenAI blog*, 2019.
- [82] Rahul Raguram, Andrew M. White, Yi Xu, Jan-Michael Frahm, Pierre Georgel, and Fabian Monrose. On the privacy risks of virtual keyboards: Automatic reconstruction of typed input from compromising reflections. *IEEE Transactions on Dependable and Secure Computing*, 2013.
- [83] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. ZeRO: Memory optimizations toward training trillion parameter models. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2020.
- [84] Samyam Rajbhandari, Olatunji Ruwase, Jeff Rasley, Shaden Smith, and Yuxiong He. ZeRO-Infinity: Breaking the GPU memory wall for extreme scale deep learning. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2021.
- [85] Jie Ren, Samyam Rajbhandari, Reza Yazdani Aminabadi, Olatunji Ruwase, Shuangyan Yang, Minjia Zhang, Dong Li, and Yuxiong He. ZeRO-Offload: Democratizing billion-scale model training. In *USENIX Annual Technical Conference (USENIX ATC)*, 2021.
- [86] Apple Machine Learning Research. Understanding aggregate trends for Apple intelligence using differential privacy. <https://machinelearning.apple.com/research/differential-privacy-aggregate-trends>, 2025.
- [87] Samsung. Near memory solutions for the AI era. <http://semiconductor.samsung.com/news-events/tech-blog/near-memory-solutions-for-the-ai-era/>, 2023.
- [88] Joonseop Sim, Soohong Ahn, Taeyoung Ahn, Seungyong Lee, Myunghyun Rhee, Jooyoung Kim, Kwangsik Shin, Donguk Moon, Euihoek Kim, and Kyoung Park. Computational CXL-memory solution for accelerating memory-intensive applications. *IEEE Computer Architecture Letters*, 2022.
- [89] Amer Sinha, Thomas Mesnard, Ryan McKenna, Daogao Liu, Christopher A. Choquette-Choo, Yangsibo Huang, Da Yu, George Kaissis, Zachary Charles, Ruibo Liu, Lynn Chua, Pritish Kamath, Pasin Manurangsi, Steve He, Chiyuan Zhang, Badih Ghazi, Borja De Balle Pigem, Prem Eruvbetine, Tris Warkentin, Armand Joulin, and Ravi Kumar. VaultGemma: A differentially private Gemma model. In *arxiv.org*, 2025.
- [90] Xuan Sun, Hu Wan, Qiao Li, Chia-Lin Yang, Tei-Wei Kuo, and Chun Jason Xue. RM-SSD: In-storage computing for large-scale recommendation inference. In *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*, 2022.
- [91] Jun Tang, Aleksandra Korolova, Xiaolong Bai, Xueqiang Wang, and Xiaofeng Wang. Privacy loss in Apple’s implementation of differential privacy on MacOS 10.12. In *arxiv.org*, 2017.
- [92] Qiaoyue Tang and Mathias Lécuyer. DP-Adam: Correcting DP bias in Adam’s second moment estimation. In *arxiv.org*, 2023.
- [93] Qiaoyue Tang, Frederick Shpilevskiy, and Mathias Lécuyer. DP-AdamBC: Your DP-Adam is actually DP-SGD (unless you apply bias correction). In *Proceedings of the AAAI Conference on Artificial Intelligence*, 2024.
- [94] Xinyu Tang, Richard Shin, Huseyin A Inan, Andre Manoel, Fatemehsadat Miresghallah, Zinan Lin, Sivakanth Gopi, Janardhan Kulkarni, and Robert Sim. Privacy-preserving in-context learning with differentially private few-shot generation. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2024.
- [95] David Tastuggine and Ilya Mironov. Introducing Opacus: A high-speed library for training pytorch models with differential privacy. <https://ai.meta.com/blog/introducing-opacus-a-high-speed-library-for-training-pytorch-models-with-differential-privacy/>, 2020.
- [96] Florian Tramer and Dan Boneh. Differentially private learning needs better features (or much more data). In *arxiv.org*, 2020.
- [97] Laurens van der Maaten and Awni Hannun. The trade-offs of private prediction. In *arxiv.org*, 2020.
- [98] Liangyu Wang, Junxiao Wang, Jie Ren, Zihang Xiang, David E. Keyes, and Di Wang. FlashDP: Memory-efficient and high-throughput DP-SGD training for large language models. In *NeurIPS 2024 Workshop on Adaptive Foundation Models*, 2024.

- [99] Lauren Watson, Chuan Guo, Graham Cormode, and Alexandre Sablayrolles. On the importance of difficulty calibration in membership inference attacks. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2022.
- [100] Mark Wilkening, Udit Gupta, Samuel Hsia, Caroline Trippel, Carole-Jean Wu, David Brooks, and Gu-Yeon Wei. RecSSD: Near data processing for solid state drive based recommendation inference. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operation Systems (ASPLOS)*, 2021.
- [101] Zheng Xu, Yanxiang Zhang, Galen Andrew, Christopher A. Choquette-Choo, Peter Kairouz, H. Brendan McMahan, Jesse Rosenstock, and Yuanbo Zhang. Federated learning of Gboard language models with differential privacy. In *The Annual Meeting of the Association for Computational Linguistics: Industry Track (ACL)*, 2023.
- [102] Ruoxing Yang. DP-Adam-AC: Privacy-preserving fine-tuning of localizable language models using Adam optimization with adaptive clipping. In *arxiv.org*, 2025.
- [103] Da Yu, Peter Kairouz, Sewoong Oh, and Zheng Xu. Privacy-preserving instructions for aligning large language models. In *Proceedings of the International Conference on Machine Learning (ICML)*, 2024.
- [104] Da Yu, Saurabh Naik, Arturs Backurs, Sivakanth Gopi, Huseyin A. Inan, Gautam Kamath, Janardhan Kulkarni, Yin Tat Lee, Andre Manoel, Lukas Wutschitz, Sergey Yekhanin, and Huishuai Zhang. Differentially private fine-tuning of language models. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2022.
- [105] Yaodong Yu, Maziar Sanjabi, Yi Ma, Kamalika Chaudhuri, and Chuan Guo. ViP: A differentially private foundation model for computer vision. In *Proceedings of the International Conference on Machine Learning (ICML)*, 2024.
- [106] Sungmin Yun, Kwanhee Kyung, Juhwan Cho, Jaewan Choi, Jongmin Kim, Byeongho Kim, Sukhan Lee, Kyomin Sohn, and Jung Ho Ahn. Duplex: A device for large language models with mixture of experts, grouped query attention, and continuous batching. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, 2024.
- [107] Sungmin Yun, Hwayong Nam, Kwanhee Kyung, Jaehyun Park, Byeongho Kim, Yongsuk Kwon, Eojin Lee, and Jung Ho Ahn. CLAY: CXL-based scalable NDP architecture accelerating embedding layers. In *Proceedings of the International Conference on Supercomputing (ICS)*, 2024.
- [108] Angela Zhang, Lei Xing, James Zou, and Joseph C Wu. Shifting machine learning for healthcare from development to deployment and from models to data. *Nature Biomedical Engineering*, 2022.
- [109] Jiaojiao Zhang, Linglingzhi Zhu, Dominik Fay, and Mikael Johansson. Locally differentially private on-line federated learning with correlated noise. In *IEEE Transactions Signal Processing*, 2025.
- [110] Liang Zhang, Bingcong Li, Kiran Koshy Thekumparampil, Sewoong Oh, and Niao He. DPZero: private fine-tuning of language models without backpropagation. In *Proceedings of the International Conference on Machine Learning (ICML)*, 2024.
- [111] Susan Zhang, Stephen Roller, Naman Goyal, Mikel Artetxe, Moya Chen, Shuohui Chen, Christopher Dewan, Mona Diab, Xian Li, Xi Victoria Lin, Todor Mihaylov, Myle Ott, Sam Shleifer, Kurt Shuster, Daniel Simig, Punit Singh Koura, Anjali Sridhar, Tianlu Wang, and Luke Zettlemoyer. OPT: Open pre-trained transformer language models. In *arxiv.org*, 2022.
- [112] Xinwei Zhang, Zhiqi Bu, Borja Balle, Mingyi Hong, Meisam Razaviyayn, and Vahab Mirrokni. DiSK: Differentially private optimizer with simplified kalman filter for noise reduction. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2025.
- [113] Yuanbo Zhang, Daniel Ramage, Zheng Xu, Yanxiang Zhang, Shumin Zhai, and Peter Kairouz. Private federated learning in Gboard. In *arxiv.org*, 2023.

A Artifact Appendix

Abstract

Our artifact is available on GitHub. The repository contains our implementation of Cocoon, a system for differentially private training with correlated noise, along with scripts to reproduce the experiments in this paper. This codebase extends `fastDP` with noise distribution, generation, CPU offload pipelines, and noise preprocessing for embedding.

Scope

This artifact allows readers to validate the performance claims in the paper. Figures 3, 5, 14, 15 and 18 are scripted runs, and all other figures can be reproduced with appropriate changes. The getting-started script checks correctness and throughput on a single GPU in less than 30 minutes.

Contents

The repository consists of five directories: `fastDP/` contains the core library including the engine and noise worker. `examples/` holds experiment scripts for LLM, DLRM, and image classification. `benchmark/` provides a getting-started script with correctness checks and throughput benchmarks. `pfl/` contains BandMF matrix solvers from pfl-research. `train_utils/` provides shared training utilities.

Hosting

The artifact is available on GitHub (<https://github.com/psu-paws/Cocoon>), `osdi26` branch. The `main` branch of the GitHub repository will be actively updated.

Requirements

All main experiments were run on a machine with two sockets of $4 \times$ NVIDIA RTX A5000 24GB GPUs and an Intel Xeon Gold 6330 CPU with 128 GB DRAM. Parameters must be tuned to match the machine's topology and memory configuration.