

# PIM-Opt: Demystifying Distributed Optimization Algorithms on a Real-World Processing-In-Memory System

Steve Rhyner<sup>1</sup> Haocong Luo<sup>1</sup> Juan Gómez-Luna<sup>2</sup> Mohammad Sadrosadati<sup>1</sup>  
Jiawei Jiang<sup>3</sup> Ataberk Olgun<sup>1</sup> Harshita Gupta<sup>1</sup> Ce Zhang<sup>4</sup> Onur Mutlu<sup>1</sup>

<sup>1</sup>ETH Zurich <sup>2</sup>NVIDIA <sup>3</sup>Wuhan University <sup>4</sup>University of Chicago

## Abstract

Modern *Machine Learning* (ML) training on large-scale datasets is a very time-consuming workload. It relies on the optimization algorithm *Stochastic Gradient Descent* (SGD) due to its effectiveness, simplicity, and generalization performance (i.e., test performance on unseen data). Processor-centric architectures (e.g., CPUs, GPUs) commonly used for modern ML training workloads based on SGD are bottlenecked by data movement between the processor and memory units due to the poor data locality in accessing large training datasets. As a result, processor-centric architectures suffer from low performance and high energy consumption while executing ML training workloads. *Processing-In-Memory* (PIM) is a promising solution to alleviate the data movement bottleneck by placing the computation mechanisms inside or near memory. Several prior works propose PIM techniques to accelerate ML training; however, prior works either do not consider *real-world* PIM systems or evaluate algorithms that are not widely used in modern ML training.

Our goal is to understand the capabilities and characteristics of popular distributed SGD algorithms on *real-world* PIM systems to accelerate data-intensive ML training workloads. To this end, we 1) implement several representative centralized parallel SGD algorithms, i.e., based on a central node responsible for synchronization and orchestration, on the *real-world* general-purpose UPMEM PIM system, 2) rigorously evaluate these algorithms for ML training on large-scale datasets in terms of performance, accuracy, and scalability, 3) compare to conventional CPU and GPU baselines, and 4) discuss implications for future PIM hardware. We highlight the need for a shift to an algorithm-hardware codesign to enable decentralized parallel SGD algorithms in real-world PIM systems, which significantly reduces the communication cost and improves scalability.

Our results demonstrate three major findings: 1) The general-purpose UPMEM PIM system can be a viable alternative to state-of-the-art CPUs and GPUs for many memory-bound ML training workloads, especially when operations and datatypes are natively supported by PIM hardware, 2) it is important to carefully *choose* the optimization algorithms that best fit PIM, and 3) the UPMEM PIM system does *not* scale approximately linearly with the number of nodes for many data-intensive ML training workloads. We open source all our code to facilitate future research at <https://github.com/CMU-SAFARI/PIM-Opt>.

## 1 Introduction

*Stochastic Gradient Descent* (SGD) [26, 157] is perhaps the most important and commonly deployed optimization algorithm for modern *Machine Learning* (ML) training [22, 23, 98, 111, 124, 176]. SGD is the

main building block of most centralized and decentralized optimization algorithms that have been introduced to accommodate the continuously increasing demand for scalability and high-performance training of ML models on large-scale datasets.

Training ML models on growing datasets [55, 190, 193] is a time-consuming task that demands both high computational power and memory bandwidth [42, 74, 75, 81]. The low data reuse during ML training on large-scale datasets leads to poor data locality. As a result, processor-centric architectures (e.g., CPU, GPU) commonly used by the ML community repeatedly need to move training samples between the processor and off-chip memory. This not only degrades performance [96] but is also a major source of the overall system’s energy consumption [17]. This phenomenon is referred to as the data movement bottleneck [135, 138, 139], which is common in data-intensive workloads. ML training is a prominent example of such workloads.

*Processing-In-Memory* (PIM) [69, 136–139, 169] is a promising way to alleviate the data movement bottleneck by placing the computation mechanisms inside or near memory units. PIM, an idea proposed several decades ago [101, 177], is a memory-centric computation paradigm that has recently gained traction in both academia [10–13, 16, 17, 31, 32, 36, 39, 46, 49, 53, 56, 63, 65, 69, 70, 72–74, 76, 77, 80, 81, 92, 93, 95, 99, 100, 102, 104–106, 110, 113, 115, 116, 121, 126, 135–139, 141, 149, 151, 154, 161, 169, 171, 172, 181, 186, 189, 196, 203] and industry; some commercial PIM systems and prototypes have recently been developed by industry [52, 103, 112, 119, 120, 142, 184, 185, 188].

Several prior works explore the effectiveness of using PIM for fundamentally improving ML training performance and energy efficiency [56, 63, 94, 97, 122, 128, 130, 161, 162, 171, 179, 181, 189]. However, none of these prior works provide a comprehensive evaluation on *real-world* general-purpose PIM architectures. To our knowledge, there is only one prior work [76, 77] on training and evaluating ML models on a *real-world* PIM system using *Gradient Descent* (GD) [155]. Unfortunately, GD-based algorithms are not widely used in modern ML. SGD [26, 157] is a simplification of GD: in each iteration, only stochastic gradients instead of the full gradient need to be computed [23]. Since stochastic gradients are, in general, significantly more efficient to compute compared to full gradients, SGD alleviates the computational bottleneck [176] of computing the full gradient by approximating the expected gradient with an unbiased estimate [78]. Variants of SGD such as mini-batch SGD [124] allow for parallelization by batching (in each iteration) the training samples whose gradients can be computed independently. We focus on popular SGD-based algorithms due to their effectiveness, simplicity, and generalization performance (i.e., test performance on unseen data) [209].

**Our goal** in this paper is to understand the capabilities and characteristics of popular distributed SGD algorithms on *real-world* PIM architectures to accelerate data-intensive ML training workloads. To do so, we implement and rigorously evaluate 12 representative ML training workloads, commonly used in the ML community, on the *real-world* UPMEM PIM architecture. We choose the general-purpose UPMEM PIM system for our study because it is commercially available [184, 185, 188]. First, we implement and investigate all combinations of 1) three centralized parallel SGD algorithms that specifically take into account the close resemblance of the UPMEM PIM system to a distributed system [31], *mini-batch Stochastic Gradient Descent with Model Averaging* (MA-SGD) [134, 212], *mini-batch Stochastic Gradient Descent Gradient Averaging* (GA-SGD) [48, 123], and the *communication-efficient* distributed *Alternating Direction Method of Multipliers* (ADMM) algorithm [25], 2) two popular and representative linear binary classification models, *Logistic Regression* (LR) [78] and *Support Vector Machine* (SVM) [21, 40, 78], and 3) two large-scale datasets, YFCC100M-HNfc6 and Criteo [5, 43]. Second, we rigorously evaluate all combinations of these algorithms, models, and datasets in terms of performance, accuracy, and scalability. Third, we compare the training speed and test set inference accuracy of the UPMEM PIM system to state-of-the-art CPU (2x AMD EPYC 7742 CPU 64-core processor [6]) and GPU (NVIDIA A100 [143]) baselines. Fourth, we discuss implications for future PIM hardware and highlight the need for a shift to an algorithm-hardware codesign perspective to accommodate decentralized optimization algorithms on *real-world* PIM systems by supporting direct communication across PIM nodes.

**Our results** demonstrate three major findings: 1) the UPMEM PIM system can be a viable alternative to state-of-the-art CPUs and GPUs for many data-intensive ML training workloads when operations and datatypes are natively supported by PIM hardware. For instance, for the YFCC100M-HNfc6 (Criteo) dataset, training SVM with GA-SGD on PIM is 1.94x faster (2.43x slower) compared to the CPU baseline, and 3.19x (10.65x) faster compared to mini-batch SGD on the GPU architecture while achieving similar accuracy. 2) It is important to carefully *choose* the optimization algorithm that best fits PIM. For example, for the YFCC100M-HNfc6 (Criteo) dataset, training SVM with the ADMM algorithm using PIM, we observe speedups of 3.19x (31.82x) compared to GA-SGD at the cost of a small reduction, i.e., 1.007x (1.014x), in test accuracy (AUC score [91]; see §4.4 for more details). 3) The UPMEM PIM system exhibits scalability challenges for many ML training workloads in terms of *statistical efficiency*, i.e., how many steps are needed until convergence [204]. For instance, in our strong scaling (see §5 for more details) experiments of the YFCC100M-HNfc6 (Criteo) dataset, training LR with ADMM using PIM, we observe speedups of 7.43x (3.85x) while the achieved test accuracy (AUC score) decreases from 95.46% (0.74) to 92.17% (0.718), as we scale the number of nodes from 256 to 2048. This reduction in accuracy is due to the fact that more nodes increase staleness as each node uses its own local model before synchronizing with the central node.<sup>1</sup>

This paper makes the following key contributions:

- To our knowledge, this paper is the first to implement, analyze, and train linear ML models on two large-scale datasets

<sup>1</sup>For a theoretical analysis of this phenomenon of how the number of nodes affects the convergence rate, we refer the reader to [208].

using *realistic* and *communication-efficient* distributed SGD algorithms on a *real-world* PIM system (i.e., UPMEM).

- We present the design space covering the design choices of various algorithms, models, and workloads for ML training on a state-of-the-art *real-world* PIM system.
- We demonstrate scalability challenges of the UPMEM PIM system in terms of *statistical efficiency*. We discuss implications for hardware design to accommodate decentralized optimization algorithms and highlight the need for a shift towards an algorithm-hardware codesign in the context of ML training using PIM.
- We open source all our code to facilitate future research at <https://github.com/CMU-SAFARI/PIM-Opt>.

## 2 Background & Motivation

We provide a brief introduction to linear models, *Machine Learning* (ML) training, regularization, and algorithms (§2.1). We describe the UPMEM *Processing-In-Memory* (PIM) system (§2.2), the first *real-world* general-purpose PIM hardware architecture that we perform ML training on. There are a number of works exploring a variety of approaches on PIM [1–3, 7, 8, 14–20, 27, 29, 30, 33, 44, 45, 50, 54, 58–60, 62, 64, 67, 68, 71, 79, 82, 84–86, 88–90, 107, 109, 117, 118, 129, 132, 133, 140, 144–148, 150, 153, 156, 163–168, 170, 173, 174, 180, 202, 205, 210, 211]. For general PIM background and discussion of many works in the field, we refer the reader to [69, 136, 138, 139]. We conclude this section with our motivation (§2.3).

### 2.1 Models, ML Training, Regularization & Algorithms

**Models.** Two of the most commonly trained linear binary classification models for convex optimization tasks are: 1) *Logistic Regression* (LR) with Binary Cross Entropy Loss (BCE) [78], and 2) *Support Vector Machines* (SVM) with Hinge Loss [21, 40, 78]. Each model consists of a linear layer and a bias.

**ML Training.** The goal of *Machine Learning* (ML) training is to find an optimal ML model

$$w^* = \operatorname{argmin}_{w \in \mathbb{R}^d} L(w) \text{ where } L(w) = \frac{1}{n} \sum_{i=1}^n l(x_i, y_i, w) \quad (1)$$

over a training dataset  $\mathcal{D} = \{(x_i, y_i)\}_{i=1}^n$  [78, 87, 98, 124]. Here,  $x_i \in \mathbb{R}^d$  is referred to as feature vector,  $y_i \in \mathbb{R}$  as label of the  $i^{\text{th}}$  training sample,  $n$  denotes the cardinality of  $\mathcal{D}$ , and  $l(x_i, y_i, w)$  is a loss function [98, 124]. For binary classification tasks, for LR, it is common to assign labels  $y_i \in \{0, 1\}$  to denote the membership of the  $i^{\text{th}}$  training sample to one of the two classes. In contrast, for SVM, the labels are  $y_i \in \{-1, 1\}$ .

**Regularization.** It is common to add a regularizer  $r(w)$  to prevent overfitting on the training dataset and to control model complexity. The objective function is obtained by setting

$$l(x_i, y_i, w) = l'(x_i, y_i, w) + \lambda r(w) \quad (2)$$

where  $l'(x, y, w)$  is a loss function and  $\lambda$  is the regularization parameter [124]. The regularization strategy of defining the regularizer as  $r(w) = \frac{1}{2} \|w\|_2^2$  is referred to as  $L^2$  regularization [78]. Another popular approach is to set  $r(w) = \|w\|_1$ , i.e., the sum of the absolute values of the model parameters, known as  $L^1$  regularization [78].

**Algorithms.** *Stochastic Gradient Descent* (SGD) [26, 157] is perhaps the most important and commonly deployed optimization algorithm for modern *Machine Learning* (ML) training [22, 23, 98, 111, 124, 176]. In each iteration, SGD computes a stochastic gradient and updates the model [158]. SGD is a simplification of GD [155]: in each iteration, only stochastic gradients instead of the full gradient need to be computed [23]. Since stochastic gradients are, in general, significantly more efficient to compute compared to full gradients, SGD alleviates the computational bottleneck [176] of computing the full gradient by approximating the expected gradient with an unbiased estimate [78]. Variants of SGD such as mini-batch SGD [124] allow for parallelization by batching the training samples in each iteration whose gradients can be computed independently.

We provide background on three widely used centralized optimization algorithms for training both LR and SVM: 1) *mini-batch Stochastic Gradient Descent with Model Averaging* (MA-SGD) [134, 212], 2) *mini-batch Stochastic Gradient Descent Gradient Averaging* (GA-SGD) [48, 123], and 3) *distributed Alternating Direction Method of Multipliers* (ADMM) [25]. These algorithms are based on a parameter server, i.e., a central node responsible for synchronization and updating the global model, and several workers among which the training dataset is evenly partitioned.

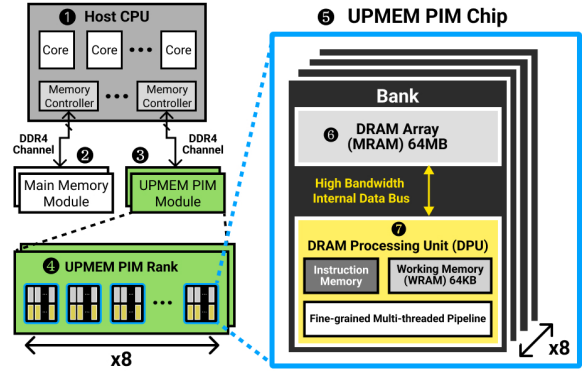
In MA-SGD, every worker trains a local model using the mini-batch SGD optimization algorithm independently and in parallel. Each worker processes several mini-batches, updating its local model before synchronization on the parameter server where the models are averaged. Then, the averaged model, i.e., the global model, is broadcast back to the workers, and each worker continues training with mini-batch SGD starting from the global model. There exists a *one-shot averaging* [134, 212] variant of the MA-SGD algorithm, where models are averaged after each worker has processed its entire partition of the training data. Although one-shot averaging reduces communication, it has been shown that increasing the model averaging frequency leads to a higher convergence rate [201, 206].

In contrast, GA-SGD distributes each batch among all workers. Each worker runs mini-batch SGD independently and in parallel, computes the gradients for a fraction of the batch, and communicates the gradient with the parameter server after *every* iteration, where the gradients are averaged and the global model is updated. Subsequently, the global model is communicated with the workers, and the next batch is processed. For both GA-SGD and MA-SGD, it is common to refer to one *global epoch* once the whole training dataset has been processed.

The distributed ADMM algorithm follows a *decomposition-coordination* procedure, dividing a convex optimization problem into smaller local subproblems distributed among workers [25, 98]. Each worker solves its subproblem, e.g., with mini-batch SGD until convergence. Next, the local models are communicated with the parameter server, where the global model and auxiliary variables that help lead the workers to a consensus are computed. Each worker continues training with its local model after the synchronization step. For ADMM, it is common to refer to one *global epoch* once the synchronization step on the parameter server has been completed.

## 2.2 UPMEM PIM System Architecture

Fig. 1 shows the high-level system organization of an UPMEM PIM-enabled system [74, 75, 81] and the hardware architecture of an UPMEM PIM chip. The system consists of a regular host CPU ①, conventional main memory modules ②, and UPMEM PIM memory modules ③. Each UPMEM PIM memory module contains two *ranks* ④. Each rank has 8 UPMEM PIM *chips* ⑤. Inside each chip, there are 8 *banks*. Each bank contains 1) a 64MB DRAM array called MRAM ⑥, and 2) a *general-purpose DRAM Processing Unit* (DPU) ⑦.



**Figure 1: High-level system organization of an UPMEM PIM-enabled system and the hardware architecture of an UPMEM PIM chip.**

The MRAM implements a standard JEDEC DDR4 DRAM interface that can be accessed by the host CPU. The DPU has an SRAM Instruction Memory, a 64KB SRAM Working Memory (WRAM), and an in-order fine-grained multi-threaded pipeline with 11 stages and supports 24 hardware threads. It implements a 32-bit RISC-based Instruction Set Architecture (ISA) with native support for 32-bit integer additions/subtractions and 8-bit integer multiplications. Other more complex arithmetic operations (e.g., integer divisions and floating-point operations) are emulated through software. The DPU does not have a cache but uses the WRAM as a scratchpad memory [184, 185].

Each DPU has *exclusive* access to its MRAM (with respect to other DPUs) through a high-bandwidth (up to 0.7GB/s per DPU) internal data bus [76, 81]. There are no direct communication channels among DPUs within an UPMEM PIM chip. All inter-DPU communications are done through the host CPU (i.e., the host CPU first gathers data from the DPUs' MRAM into the system's main memory and then distributes the data from the main memory to the DPUs' MRAM).

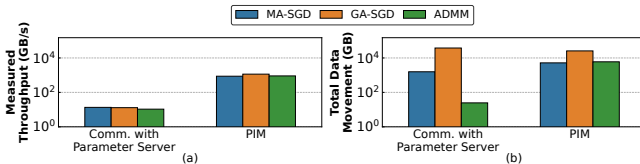
**PIM Programming and Execution Model.** DPU programs are written in the C programming language with the UPMEM SDK [187] and runtime libraries. The execution model of a DPU is based on the *Single-Program Multiple-Data* (SPMD) paradigm. Each DPU runs multiple (up to 24) software threads, called *tasklets*, which execute the same code but operate on different data. Each tasklet has its own control flow, independent from other tasklets. Tasklets are assigned to DPUs *statically* by the programmer during *compile-time*. Tasklets assigned to the same DPU share MRAM and WRAM [76, 81].

## 2.3 Motivation

*Stochastic Gradient Descent* (SGD) [26, 157] is one of the most important optimization algorithms and the basis of many distributed optimization algorithms. However, SGD is memory-bound [47, 108, 131, 191, 198], which poses a fundamental challenge for processor-centric architectures (e.g., CPU, GPU). SGD’s memory-boundedness is attributed to large training dataset size, leading to decreased cache efficiency and low data reuse of training samples during ML training, which results in performance degradation [34, 35, 198]. The increasing discrepancy in performance between fast processors and slow memory units exacerbates this problem [76]. PIM is a promising way to alleviate the data movement bottleneck and is a promising paradigm to efficiently execute ML training workloads.

There are several prior proposals on PIM acceleration for ML training [56, 63, 161, 171, 181, 189]. However, none of these prior works provide a comprehensive evaluation on *real-world* general-purpose PIM systems. To our knowledge, there is only one prior work [76, 77] on training and evaluating ML models on a *real-world* PIM system (i.e., UPMEM) using *Gradient Descent* (GD) [155]. In this work, we examine popular SGD-based algorithms due to their better effectiveness, simplicity, and generalization performance [209]. We specifically address the resemblance of the UPMEM PIM system to a distributed system [31] with the host CPU as the central node, i.e., the parameter server.

We first demonstrate that it is important to carefully *choose* the distributed optimization algorithm that best fits the UPMEM PIM system. To do so, we analyze key differences in data movement of distributed optimization algorithms on the UPMEM PIM system. Fig. 2 shows the per global epoch comparison of data movement for all distributed optimization algorithms we study, i.e., MA-SGD, GA-SGD, and ADMM, using the UPMEM PIM system with 2048 DPUs training an LR model on the Criteo dataset (see §3 and §4). Fig. 2(a) shows the per global epoch measured throughput between PIM and the parameter server (Comm. with Parameter Server, i.e., the measured throughput between UPMEM PIM memory modules and the host CPU over the DDR4 channels) and within PIM (PIM, i.e., the internal aggregated measured throughput between MRAM and WRAM). Fig. 2(b) shows the per global epoch total data movement between PIM and the parameter server (Comm. with Parameter Server, i.e., the absolute amount of data exchanged between the UPMEM PIM memory modules and the host CPU over DDR4 channels) and within PIM (PIM, i.e., the absolute amount of data transferred between MRAM and WRAM). For MA-SGD and ADMM, the batch size is 2K; for GA-SGD, it is 262K.



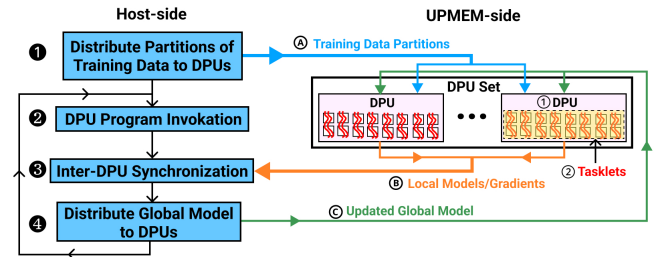
**Figure 2: Per global epoch comparison of measured throughput (a) and total data movement (b) for all studied algorithms (MA-SGD, GA-SGD, and ADMM) on the UPMEM PIM system for the Criteo dataset.**

We make two major observations. First, the throughput between PIM and the parameter server and within PIM is very large. For

instance, for LR, we observe that the throughput within PIM for MA-SGD/GA-SGD/ADMM is 64.55x/88.35x/85.22x higher than the throughput between PIM and the parameter server. This is because the bandwidth *within* PIM is much higher than the bandwidth between PIM and the parameter server. Hence, it is specifically crucial to minimize the communication between PIM and the parameter server to significantly reduce the total communication complexity. Second, the absolute data movement between PIM and the parameter server is very high. For instance, for LR, the algorithms MA-SGD (GA-SGD) exhibit 64.01x (1536.14x) higher absolute data movement for expensive communication between PIM and the parameter server per global epoch compared to ADMM. This observation shows that the communication patterns imposed by MA-SGD and GA-SGD lead to a communication bottleneck on the parameter server due to the large amount of data to be transferred between PIM and the parameter server. In contrast, ADMM’s *efficient* communication pattern alleviates this communication bottleneck by reducing the data movement over the low-bandwidth channels between PIM and the parameter server. We conclude that ADMM is a good fit for the UPMEM PIM system because it addresses the communication bottleneck on the parameter server.

## 3 UPMEM PIM System Implementation

Fig. 3 shows the high-level workflow of training ML models using distributed optimization algorithms on the UPMEM PIM system. First (❶ in Fig. 3), the host CPU *statically* partitions, assigns, and distributes the training data to the MRAM of the DPUs ❶ (i.e., each DPU receives a partition of the whole training data ❶) and assigns tasklets ❷ to DPUs. This training data transfer from the host to the DPUs happens *only once* throughout the entire training process. Second ❷, the host invokes the DPU program to run mini-batch SGD on every DPU. Third ❸, the host synchronizes all DPUs by collecting and aggregating the local models (MA-SGD and ADMM) or gradients (GA-SGD) ❸ from the DPUs in the host’s main memory to produce an updated global model ❹ (see §2.1). Fourth ❹, the host distributes the updated global model ❹ to each DPU and then invokes the DPU program ❷ again to keep on training. Steps ❷, ❸, and ❹ repeat until the training finishes (i.e., reaches a certain number of global epochs or achieves a certain level of accuracy).



**Figure 3: High-level workflow for distributed optimization algorithms on the UPMEM PIM system.**

**Data Partitioning.** For both MA-SGD and ADMM, each DPU’s partition consists of multiple mini-batches of the entire training data. For GA-SGD, each partition consists of a fraction of *all* the mini-batches of the training data (i.e., different DPUs get different fractions of the same mini-batches).



**Synchronization.** For MA-SGD, each DPU only processes one mini-batch from its assigned training data partition and updates its local model before synchronization (i.e., model averaging) on the host. Doing so leverages the fact that increasing model averaging frequency leads to a higher convergence rate at the cost of higher communication overhead [201, 206]. For GA-SGD, each DPU computes intermediate gradients from its assigned fraction of one mini-batch before synchronization (i.e., gradient averaging) on the host. A larger batch size leads to a higher number of samples to be processed by the DPU before synchronization (i.e., less communication overhead) with the host CPU. For ADMM, each DPU processes *all* assigned mini-batches and updates its local model for every mini-batch. ADMM only synchronizes the local models on the host *once* after the DPUs finish processing all their assigned training data, making it attractive for distributed ML due to the low communication overhead compared to MA-SGD and GA-SGD.

**Task Parallelism.** For all distributed optimization algorithms we study (MA-SGD, GA-SGD, and ADMM), we consider every DPU as a *worker*. For each DPU (worker), we use 16 tasklets collaboratively (i.e., parallelizing the dot product and transferring data between MRAM and WRAM) to implement the mini-batch SGD optimizer to fully utilize the multi-threaded pipeline and improve latency [81]. We evenly distribute features of the training samples and the corresponding model parameters among tasklets.

**LUT-based Methods.** Training of LR involves computing the exponential function to evaluate the sigmoid activation function. Since the UPMEM PIM system does not support transcendental functions, we use efficient LUT-based methods [61, 76, 95] for computation. LUTs are fast [61, 95] but incur significant storage overhead (in our case, 4MB of MRAM per DPU). However, allocating this fraction of MRAM for the LUT is necessary to enable the evaluation of the sigmoid activation function with high precision.

## 4 Methodology

In this section, we describe the system configurations (§4.1), CPU & GPU baseline implementations (§4.2), experiment implementation details of our UPMEM PIM system and baselines (§4.3), and datasets (§4.4) used in this paper.

### 4.1 System Configurations

Table 1 shows the system configuration of 1) the UPMEM PIM system [184, 185] with 20 UPMEM PIM memory modules (2560 DPUs), 2) the CPU baseline system [6] with 2x AMD EPYC 7742 64-core CPUs (in total 128 cores), and 3) the GPU baseline system [143] with an NVIDIA A100 GPU that we perform ML training on.

### 4.2 Baseline Implementations

**CPU Baseline Implementation.** We implement our CPU baselines using PyTorch [152]. We implement three distributed optimization algorithms, MA-SGD, GA-SGD, and ADMM, to train LR and SVM models, using the optimizers and communication libraries provided by PyTorch [152]. We consider each CPU thread as a *worker* in the distributed optimization algorithms.

**GPU Baseline Implementation.** We implement our GPU baselines using PyTorch [152]. We only implement mini-batch SGD on the GPU because PyTorch does not provide a way to limit the

Table 1: System Configurations

UPMEM PIM System	
<b>Processor</b>	2x Intel Xeon Silver 4215 8-core processor @ 2.50GHz
<b>Main Memory</b>	256 GB total capacity 4x64 GB DDR4 (RDIMMs)
<b>PIM-Enabled Memory</b>	160 GB total capacity 20x8 GB UPMEM PIM modules, 2560 DPUs, 2 ranks per module, 8 chips per rank, 8 DPUs per chip 350 MHz DPU clock frequency
CPU Baseline System	
<b>Processor</b>	2x AMD EPYC 7742 64-core processor @ 2.25GHz
<b>Main Memory</b>	1 TB total capacity 32x32 GB DDR4 (RDIMMs)
GPU Baseline System	
<b>Processor</b>	2x Intel Xeon Gold 5118 12-core processor @ 2.30GHz
<b>Main Memory</b>	512 GB total capacity 16x32 GB DDR4 (RDIMMs)
<b>GPU</b>	1x NVIDIA A100 (PCIe, 80 GB)

amount of GPU resources the kernels use, causing model averaging to be serialized on a single GPU. For fair comparison, we do not use a cluster of GPUs for our baseline because the UPMEM PIM system is a single-server node.<sup>2</sup>

### 4.3 Experiment Implementation Details

**Data Format.** Quantization is a popular approach used in the ML community to enable fixed-point operations [37, 51, 197]. We conduct ML training on the *real-world* UPMEM PIM system on quantized [38, 83, 183, 194] training data and models. Both data and models are represented using a 32-bit fixed-point format because the UPMEM PIM system does not natively support floating-point operations. We use the FP32 floating-point format for our CPU and GPU baselines because 1) CPUs and GPUs natively support it and 2) it provides higher accuracy.

**Hyperparameter Tuning.** We tune the learning rates and regularization terms for all workloads we evaluate. We open source all tested hyperparameters along with our complete codebase at <https://github.com/CMU-SAFARI/PIM-Opt>.

**Regularization.** We use standard regularization techniques to achieve lower generalization errors. For MA-SGD and GA-SGD, we add an  $L^2$  regularization term to the loss functions of the LR and SVM models. For ADMM, we include  $L^2$  regularization for the SVM loss function, while we include  $L^1$  regularization for LR. By including  $L^1$  regularization for LR ADMM, the dual optimization problem admits a closed-form solution similar to SVM ADMM with  $L^2$  regularization [25].

**Batch Size.** Given a batch of size  $b$ , for both MA-SGD and ADMM, each *worker* processes  $b$  samples in each iteration of mini-batch SGD. In contrast, for GA-SGD running on a system consisting of  $N$  *workers*, each *worker* processes  $b/N$  samples before the intermediate gradients are communicated with the parameter server. For simplicity, assume that  $b$  is divisible by the number of *workers*  $N$ . When training models on the YFCC100M-HNfc6 dataset, we consider batch sizes 8, 16, 32, and 64 for MA-SGD/ADMM and 4096 (4K), 8192 (8K), 16'384 (16K), and 32'768 (32K) for GA-SGD. When training models on Criteo, we consider batch sizes 1024 (1K), 2048

<sup>2</sup>A multi-GPU system can be compared with a multi-UPMEM PIM system, which we leave for future work.

(2K), 4096 (4K), and 8192 (8K) for MA-SGD/ADMM, and 131'072 (131K), 262'144 (262K), 524'288 (524K), and 1'048'576 (1048K) for GA-SGD. We use different batch sizes for Criteo due to its orders of magnitude larger number of samples in the training dataset (§4.4). For each experiment in §5 (except for the batch size sensitivity analysis), we tune the batch size to ensure high accuracy, high performance in terms of total training time, and fair comparison of algorithms & architectures.

**Initialization.** For both the UPMEM PIM system implementation and the CPU/GPU baselines, the training data and model weights *initially* reside in main memory. For the UPMEM PIM system and GPU experiments, the initialization phase includes transferring the data from the main memory to the PIM DRAM bank and the GPU global memory.

#### 4.4 Datasets

We consider two large-scale datasets, YFCC100M-HNfc6 and Criteo 1TB Click Logs (Criteo).

1) **YFCC100M-HNfc6** [5] consists of 97M samples with features extracted by a deep convolutional neural network from the YFCC100M multimedia dataset [182]. Each sample consists of 4096 floating-point dense features and a collection of tags. We randomly sample and shuffle data points with the tag *"outdoor"*, treating them as positive labels, and sample the same number of data points with the tag *"indoor"*, treating them as negative labels, turning this subset into a binary classification task. Although SGD randomly draws samples in theory, in practice, it is common to randomly shuffle the training dataset and sequentially process training samples at runtime, which generally is much faster [23]. We apply standard normalization to each feature column, and for our implementation on the UPMEM PIM system, we quantize the normalized dataset into a 32-bit fixed-point format. The total size of model parameters is 4KB.

2) **Criteo 1TB Click Logs (Criteo)** [43] preprocessed by LIB-SVM [57] consists of approximately 4.37 billion high-dimensional sparse samples with 1M features. Criteo is a popular click-through rate prediction dataset. Data points labeled *"click"* are treated as positive and *"no-click"* as negative labels. The dataset has been collected over 24 days and is highly imbalanced, with only 0.034% of data points being *"clicks"*. To construct the training dataset, we randomly sample and shuffle from day 0 to 22 while maintaining the ordering only among days. We use the entire day 23 as a test dataset for all our experiments. Each data point consists of a label and 39 categorical features representing a sparse embedding in a 1M-dimensional feature space. While data points only consist of 40 parameters, the models/gradients consist of 1M variables and, therefore, incur a significantly higher communication overhead compared to YFCC100M-HNfc6. We use the area under the receiver operating characteristics curve (AUC score) [91] to assess the generalization capabilities of models trained on Criteo due to its imbalanced data distribution. The total size of the model parameters is 4MB.

Table 2 summarizes the dataset configurations used in our experiments for the scaling analysis and comparison to the CPU and GPU baseline systems.

**Table 2: Dataset Configurations**

YFCC100M-HNfc6				
# Workers	# Training samples	Training size (GB)	# Test samples	Test size (GB)
256 DPUs	851'968	13.96	212'992	3.49
512 DPUs	1'703'936	27.92	425'984	6.98
1024 DPUs	3'407'872	55.83	851'968	13.96
2'048 DPUs	6'815'744	111.67	1'703'936	27.92
128 CPU threads	6'815'744	111.67	1'703'936	27.92
1 GPU	6'815'744	111.67	1'703'936	27.92
Criteo				
# Workers	# Training samples	Training size (GB)	# Test samples	Test size (GB)
256 DPUs	50'331'648	8.05	178'236'537	28.52
512 DPUs	100'663'296	16.11	178'236'537	28.52
1'024 DPUs	201'326'592	32.21	178'236'537	28.52
2'048 DPUs	402'653'184	64.42	178'236'537	28.52
128 CPU threads	402'653'184	64.42	178'236'537	28.52
1 GPU	402'653'184	64.42	178'236'537	28.52

## 5 Evaluation

We evaluate ML training of 1) dense models on the YFCC100M-HNfc6 dataset (§5.1), and 2) high-dimensional sparse models on the Criteo 1TB Click Logs dataset (§5.2). We do the following analyses.

**PIM Performance Breakdown.** To understand key characteristics of distributed ML on the UPMEM PIM system using 2048 DPUs, we break the training time of one global epoch down into 1) communication and synchronization between PIM and the parameter server (Comm./Sync. Para. Server, i.e., the fraction of training time for communicating gradients and models, and worker synchronization; see §3 for details), 2) PIM computation time (PIM Comp., i.e., the fraction of training time to execute arithmetic operations by UPMEM PIM processing units), and 3) PIM data movement time (i.e., the fraction of training time for data movement between MRAM and WRAM).

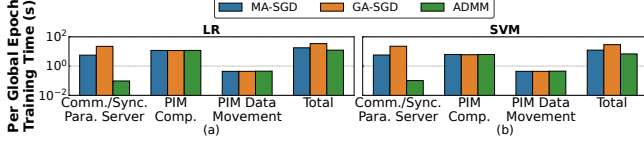
**Algorithm Selection.** To show that it is important to carefully choose the distributed optimization algorithm that best fits the UPMEM PIM system, we compare the total training time and the test accuracy (AUC score; see §4.4) to perform ML training on the dataset YFCC100M-HNfc6 (Criteo) for several combinations of models (i.e., LR, SVM), algorithms (i.e., MA-SGD, GA-SGD, ADMM, mini-batch SGD), and architectures (i.e., UPMEM PIM system, CPU baseline system, and GPU baseline system).

**Batch Size.** We study the impact of the batch size on performance in terms of total training time and the test accuracy (AUC score) to perform ML training on the dataset YFCC100M-HNfc6 (Criteo). We analyze several batch sizes on the UPMEM PIM system and the CPU baseline system. We only implement mini-batch SGD on the GPU because PyTorch does not provide a way to limit the amount of GPU resources the kernels use, causing model averaging to be serialized on a single GPU (see §4.2).

**Scaling.** We explore two different scaling variants to assess the impact of scaling on total training time (i.e., for 10 global epochs) and test accuracy (AUC score) on the UPMEM PIM system for the dataset YFCC100M-HNfc6 (Criteo). 1) *Weak Scaling.* We increase the number of DPUs from 256 to 2048 in our experiments while the training dataset size is increased from 13.96GB to 111.67GB for YFCC100M-HNfc6 (from 8.05GB to 64.42GB for Criteo). 2) *Strong Scaling.* We fix the training dataset size that fits on the smallest number of DPUs (i.e., 256). As we scale the number of DPUs from 256 to 2048 the training dataset remains unchanged, i.e., 13.96GB for YFCC100M-HNfc6 (8.05GB for Criteo).

## 5.1 Evaluation of YFCC100M-HNfc6

**PIM Performance Breakdown.** In Fig. 4, we show the training time for one global epoch (y-axis) and breakdown the training time into communication and synchronization between PIM and the parameter server (Comm./Sync. Para. Server), PIM computation time (PIM Comp.), and PIM data movement time (x-axis) for LR (Fig. 4(a)) and SVM (Fig. 4(b)). For MA-SGD and ADMM, we set the batch size to 8. For GA-SGD, we set the batch size to 4K.



**Figure 4: Per global epoch training time breakdown into Comm./Sync. Para. Server, PIM Comp., and PIM data movement time for LR (a) and SVM (b).**

**Obsv. 1.** Communication and synchronization between the parameter server and PIM is a bottleneck for MA-SGD/GA-SGD.

For instance, LR MA-SGD (GA-SGD) communication and synchronization between PIM and the parameter server requires 56.0x (223.3x) more time compared to ADMM. Here, we observe that *communication-efficient* optimization algorithms such as ADMM improve performance.

**Obsv. 2.** For all combinations of optimization algorithms and models, PIM computation takes more time than PIM data movement on the UPMEM PIM.

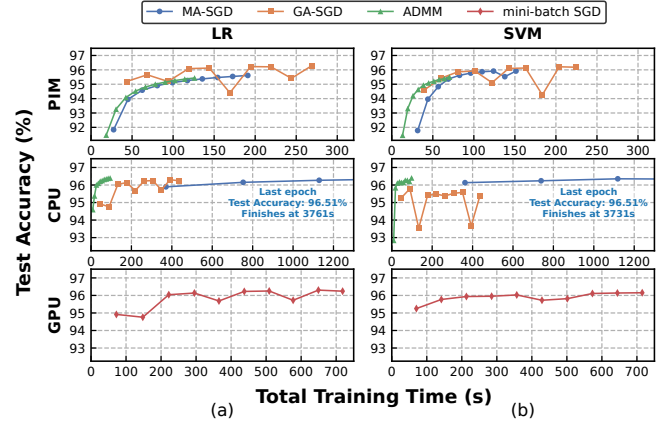
For instance, LR (SVM) MA-SGD on PIM spends 26.75x (14.05x) more time on computation than moving data between MRAM and WRAM. PIM spends less time on computation for SVM than LR because SVM exhibits lower computational complexity.

**Takeaway 1.** The UPMEM PIM is *less* suitable for ML models and optimization algorithms that require frequent communication and synchronization between PIM and the parameter server.

**Algorithm Selection.** In Fig. 5, we study the test accuracy (i.e., reached in the last global epoch; y-axis) and total training time (i.e., for 10 global epochs; x-axis) with LR (Fig. 5(a)) and SVM (Fig. 5(b)). The UPMEM PIM system with 2048 DPUs (top), the CPU baseline system (middle), and the GPU baseline system (bottom). For the algorithms MA-SGD and ADMM, we set the batch size to 8. For GA-SGD and mini-batch SGD, we set the batch size to 4K.

**Obsv. 3.** The difference in total training time between MA-SGD and ADMM is significantly lower on the UPMEM PIM compared to the CPU. GA-SGD is slower than ADMM for all configurations of LR, SVM, the UPMEM PIM, and the CPU.

For example, on the UPMEM PIM system (CPU baseline system), we observe a speedup of 1.51x (39.79x) with LR ADMM compared to LR MA-SGD. The higher speedup on the CPU baseline system is due to the smaller number of workers and, therefore, less communication overhead compared to the UPMEM PIM system. For instance, on the UPMEM PIM system (CPU baseline system), we observe speedups of 3.19x (4.45x) with SVM ADMM compared to SVM GA-SGD. This is a result of efficient communication for ADMM since local models are collected only after each DPU/CPU thread has



**Figure 5: Comparison of various models (LR (a) and SVM (b)), algorithms (MA-SGD, GA-SGD, ADMM, and mini-batch SGD), and architectures (PIM, CPU, and GPU). We study the test accuracy (at the last global epoch) and total training time (10 global epochs).**

processed its complete partition of the training dataset. In contrast, for GA-SGD, gradients are communicated in each iteration. This can cause a very large communication overhead, especially when training large-scale models.

**Obsv. 4.** GA-SGD on the UPMEM PIM outperforms GA-SGD on the CPU and mini-batch SGD on the GPU for both LR and SVM.

For LR (SVM), GA-SGD on the UPMEM PIM system achieves speedups of 1.62x (1.94x) over the CPU baseline system and 2.67x (3.19x) over the GPU baseline system running mini-batch SGD. A possible explanation for these speedups is that per CPU thread, there is not enough work before synchronization with the parameter server (Obsv. 3), and the batch size is too small on the GPU baseline system. The difference in the increase in training time between LR and SVM results from SVM’s lower computational complexity compared to LR, and therefore, in general, SVM is faster than LR on the UPMEM PIM system.

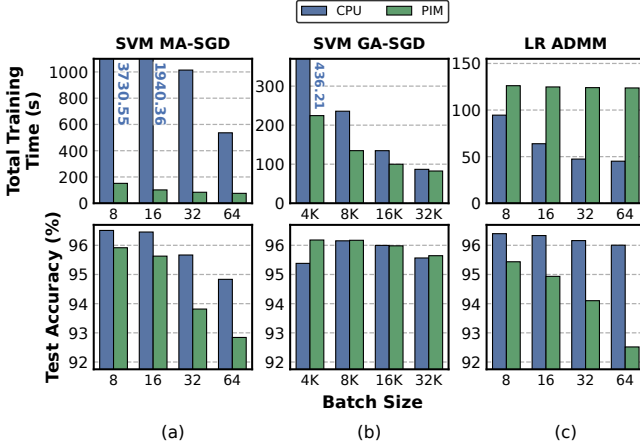
**Obsv. 5.** ADMM is faster on the UPMEM PIM for SVM compared to the CPU. For LR, the CPU is faster.

For SVM ADMM, we observe a speedup of 1.39x on the UPMEM PIM system compared to the CPU baseline system. In contrast, for LR ADMM, we notice a slowdown by a factor of 1.33x on the UPMEM PIM system compared to the CPU baseline system. This is expected since the training of SVM on the UPMEM PIM system requires less computation and no lookup to approximate the sigmoid function compared to LR.

**Takeaway 2.** The UPMEM PIM is a viable alternative to the CPU and the GPU for training small dense models on large-scale datasets.

**Batch Size.** In Fig. 6, we compare the total training time for 10 global epochs (y-axis; first row), the test accuracy reached in the last global epoch (y-axis; second row), and varying batch size (x-axis). We illustrate a fixed combination of the model and optimization algorithm for SVM MA-SGD (Fig. 6(a)), SVM GA-SGD (Fig. 6(b)), and LR ADMM (Fig. 6(c)). Each subplot compares the UPMEM PIM

system with 2048 DPUs and the CPU baseline system with 128 CPU threads for every batch size.



**Figure 6: Impact of batch size on total training time (10 global epochs) and test accuracy (at the last global epoch) for SVM MA-SGD (a), SVM GA-SGD (b), and LR ADMM (c).**

**Obsv. 6.** As batch size increases, for MA-SGD and ADMM, we observe a reduction in the total training time on the CPU. In contrast, on the UPMEM PIM, the reduction of total training time is less significant.

When batch size increases from 8 to 64, the total training time of SVM MA-SGD on the UPMEM PIM system decreases by 2.01x from 151.36s to 75.28s, compared to the CPU baseline system, where the total training time decreases by 6.96x from 3730.55s to 536.30s. This speedup is attributed to the fact that larger batch sizes directly result in less communication. The reason for the high speedup on the CPU baseline system is discussed in Obsv. 3. For LR ADMM, the total training time decreases by 1.02x on the UPMEM PIM system, compared to 2.10x on the CPU baseline system, respectively. This stems from the fact that the local model update on PIM is not significantly more compute-intensive and only requires reading the gradient into WRAM compared to a single gradient step. On the CPU baseline system, the slowdown likely stems from polluted caches due to the local model and the gradient to be loaded into the cache with an increased frequency of model updates for smaller batch sizes. We observe that the test accuracy decreases as batch size increases from 8 to 64, e.g., for SVM MA-SGD, the test accuracy decreases from 95.92% to 92.84% on the UPMEM PIM system and from 96.51% to 94.83% on the CPU baseline system. This decrease arises from the famous bias-variance tradeoff [66] when training ML models, i.e., we want to reduce variance by increasing the batch size until we observe a drop in accuracy. Note that SGD-based algorithms admit unbiased gradient estimates [78]. The discrepancy in test accuracy between the UPMEM PIM system and CPU baseline system stems from the quantization of the training data and the model and a significantly larger number of models on the UPMEM PIM system.

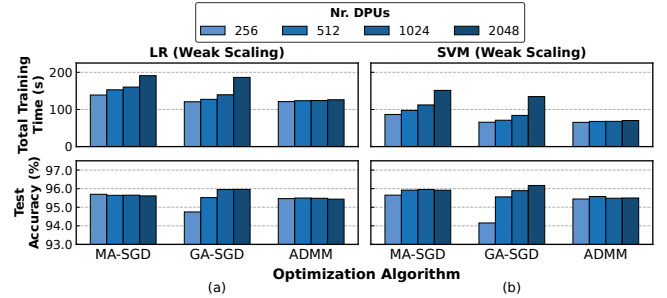
**Obsv. 7.** Both the UPMEM PIM and the CPU benefit from larger batch sizes for GA-SGD.

Only for GA-SGD, for both the UPMEM PIM system and the CPU baseline system, we observe a significant reduction in total training

time as we increase the batch size, while jointly, the test accuracy only slightly degrades. This behavior is explained by the reduction of communication for larger batch sizes since each DPU/CPU thread can process more samples before gradients need to be collected and the model is updated. GA-SGD’s test accuracy is less sensitive to increasing batch size because GA-SGD has only one model (see Obsv. 10 for more details).

**Takeaway 3.** The UPMEM PIM has benefits for 1) models that require smaller batch sizes to achieve high accuracy, and 2) algorithms that minimize inter-DPU communication via the parameter server.

**Scaling.** In Fig. 7, we study the weak scalability (i.e., the training dataset size increases proportionally as the number of DPUs increases) of using MA-SGD, GA-SGD, and ADMM (x-axis) to train LR (Fig. 7(a)) and SVM (Fig. 7(b)) models on the UPMEM PIM system. We plot the total training time for 10 global epochs (y-axis; first row), and the test accuracy reached in the last global epoch (y-axis; second row). For all combinations of the models and optimization algorithms, we increase the number of DPUs from 256 to 2048 and proportionally increase the total training dataset size from 13.96GB to 111.67GB. For MA-SGD and ADMM, we set the batch size to 8. For GA-SGD, we set the batch size to 8K.



**Figure 7: Impact of weak scaling on total training time (10 global epochs) and test accuracy (at the last global epoch) for LR (a) and SVM (b).**

**Obsv. 8.** The UPMEM PIM has good weak scalability with ADMM but poor weak scalability with MA-SGD and GA-SGD in terms of total training time.

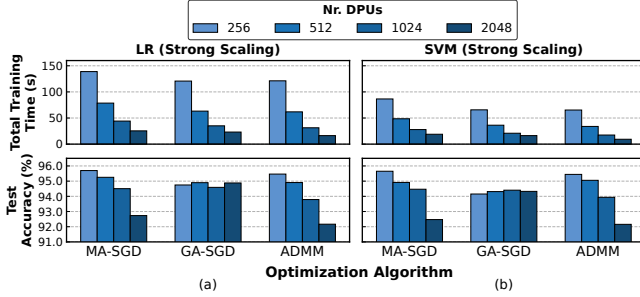
As an example, for SVM ADMM (MA-SGD), we observe an increase of total training time by 1.08x (1.75x), while the achieved test accuracy only changes very slightly as we scale from 256 to 2048 DPUs. The increase in training time is attributed to the slightly higher communication overhead for small, dense models as we scale the number of workers.

**Obsv. 9.** Among the algorithms we test, only GA-SGD’s test accuracy consistently increases when both the training dataset size and the number of DPUs increase.

For SVM GA-SGD, we observe an increase in total training time by 2.05x, while the achieved test accuracy increases from 94.15% to 96.17% as we scale the number of DPUs from 256 to 2048. The slowdown stems from higher communication overhead when training with more DPUs. For GA-SGD, when we increase the number of DPUs, each DPU processes fewer samples per batch, exacerbating the communication overhead.



In Fig. 8, we use the same experiment setting as in Fig. 7, except that we fix the training dataset size as we scale the number of DPUs (strong scaling).



**Figure 8: Impact of strong scaling on total training time (10 global epochs) and test accuracy (at the last global epoch) for LR (a) and SVM (b).**

**Obsv. 10.** The UPMEM PIM has good strong scalability in terms of total training time, but poor in accuracy.

As an example, for LR ADMM (MA-SGD), we observe a speedup of 7.43x (5.47x), while the achieved test accuracy decreases from 95.46% (95.70%) to 92.17% (92.73%), as we scale from 256 to 2048 DPUs. In contrast, for LR GA-SGD, we observe a speedup of total training time by 5.22x, while the achieved test accuracy only slightly improves as we scale the number of DPUs from 256 to 2048. Therefore, the *communication-efficient* ADMM algorithm achieves a higher speedup compared to MA-SGD and GA-SGD. The observed reduction in test accuracy for a larger number of DPUs directly corresponding to a larger number of models when training ML models with MA-SGD and ADMM is due to the fact that more workers increase staleness as each worker uses its own local model before synchronizing with the parameter server.<sup>3</sup> Other works also make this empirical observation that convergence becomes slower as the number of workers is scaled [195, 207]. However, these works consider a substantially smaller number of workers (i.e., up to 128).

**Takeaway 4.** The scalability potential of the UPMEM PIM for training small dense models is limited by its lack of direct inter-DPU communication.

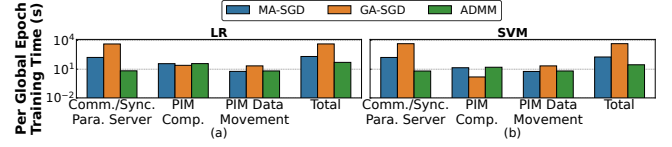
## 5.2 Evaluation of Criteo

**PIM Performance Breakdown.** In Fig. 9, we show the training time for one global epoch (y-axis) and breakdown the training time into communication and synchronization between PIM and the parameter server (Comm./Sync. Para. Server), PIM computation time (PIM Comp.), and PIM data movement time (x-axis) for LR (Fig. 9(a)) and SVM (Fig. 9(b)). For MA-SGD and ADMM, we set the batch size to 2K. For GA-SGD, we set the batch size to 262K.

**Obsv. 11.** Communication and synchronization between PIM and the parameter server is a bottleneck for MA-SGD/GA-SGD.

For instance, LR MA-SGD (GA-SGD) communication and synchronization between PIM and the parameter server requires 25.10x (640.35x) more time compared to ADMM. This coincides with Obsv. 1 for YFCC100M-HNfc6.

<sup>3</sup>For a theoretical analysis of this phenomenon of how the number of workers affects the convergence rate, we refer the reader to [208].



**Figure 9: Per global epoch training time breakdown into Comm./Sync. Para. Server, PIM Comp., and PIM data movement time for LR (a) and SVM (b).**

**Obsv. 12.** For both MA-SGD and ADMM, PIM computation takes more time than PIM data movement on the UPMEM PIM. For GA-SGD, PIM data movement takes more time than PIM computation on the UPMEM PIM.

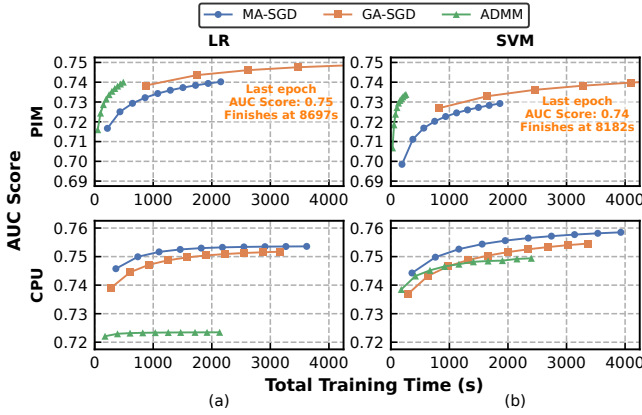
As an example, LR (SVM) MA-SGD on PIM spends 6.38x (2.44x) more time on computation than moving data between MRAM and WRAM. Compared to LR, SVM’s lower computational complexity causes it to spend less time doing computation. In contrast to our Obsv. 2 for YFCC100M-HNfc6, for Criteo, SVM GA-SGD spends 14.29x more time on moving data between MRAM and WRAM compared to computation on the PIM system. This is because the gradient update of GA-SGD requires sequentially reading the complete gradient into the WRAM and subsequently back to MRAM. For Criteo, we can take advantage of larger individual data transfers that are more efficient compared to YFCC100M-HNfc6. Therefore, most of the computation of updating the model is offloaded to the parameter server.

**Takeaway 5.** The UPMEM PIM is *less* suitable for training high-dimensional sparse models and optimization algorithms that require frequent communication and synchronization between PIM and the parameter server.

**Algorithm Selection.** In Fig. 10, we study the AUC score (i.e., reached in the last global epoch; y-axis) and total training time (i.e., for 10 global epochs; x-axis) with LR (Fig. 10(a)) and SVM (Fig. 10(b)) models. The UPMEM PIM system uses 2048 DPUs (first row), and the CPU baseline system uses 128 CPU threads (second row). For the algorithms MA-SGD and ADMM, we set the batch size to 2K. For GA-SGD and mini-batch SGD, we set the batch size to 524K. For the GPU baseline system, we only report a per batch speedup comparison to PIM with GA-SGD (Obsv. 15) because the training of Criteo’s high-dimensional sparse model with mini-batch SGD is prohibitively on the GPU baseline system.

**Obsv. 13.** ADMM outperforms MA-SGD for both LR and SVM on the UPMEM PIM in both total training time and AUC score. On CPU, ADMM outperforms MA-SGD in terms of total training time but reaches a lower AUC score.

Training on the sparse dataset Criteo, for LR (SVM) ADMM, we observe a speedup of 4.40x (7.24x) on the UPMEM PIM system and 1.70x (1.64x) on the CPU baseline system compared to MA-SGD. The reason for ADMM having a larger speedup compared to MA-SGD on the UPMEM PIM system than the CPU baseline is that the communication overhead of ADMM is much smaller compared to MA-SGD, which benefits the UPMEM PIM system more than the CPU. The reason for SVM to have a larger speedup than LR on the UPMEM PIM system is that SVM has a lower computational complexity compared to LR.



**Figure 10: Comparison of various models (LR (a) and SVM (b)), algorithms (MA-SGD, GA-SGD, and ADMM), and architectures (PIM and CPU). We study the AUC score (at the last global epoch) and total training time (10 global epochs).**

**Obsv. 14.** ADMM and MA-SGD significantly outperform GA-SGD for both LR and SVM on the UPMEM PIM in total training time with a negligible reduction in AUC score. On CPU, ADMM outperforms GA-SGD only in terms of total training time. In contrast to the UPMEM PIM system, on CPU, MA-SGD is slightly slower than GA-SGD but achieves a higher AUC score.

For instance, on the UPMEM PIM system (CPU baseline system), we observe speedups of 31.82x (1.41x) with SVM ADMM compared to SVM GA-SGD at the cost of a reduction of the AUC score by 1.014x (1.007x). The difference in the increase in training time between the UPMEM PIM system and CPU baseline system for GA-SGD is exacerbated because there are more workers on the PIM system, causing more intermediate gradients to be communicated over the slow channel between PIM and the parameter server. For SVM MA-SGD, we observe speedups of 4.39x at the cost of a reduction of the AUC score by 1.02x on the UPMEM PIM system compared to GA-SGD. In contrast, on the CPU baseline system, for SVM MA-SGD, we observe a slowdown of 1.16x and an increase of the AUC score by 1.01x. The increase in training time on the CPU baseline system for MA-SGD compared to GA-SGD is a result of that for MA-SGD, each CPU thread needs to read its gradient into cache and update the model followed by communication of the models, while for GA-SGD, the intermediate gradients are communicated directly, and only a single model is updated. Each CPU thread processes the same number of samples for MA-SGD and GA-SGD.

**Obsv. 15.** GA-SGD on the CPU outperforms GA-SGD on the UPMEM PIM and the GPU.

For LR (SVM), GA-SGD on the CPU baseline system achieves speedups of 2.75x (2.43x) over the UPMEM PIM system. This observation differs from our Obsv. 4 for YFCC100M-HNfc6. The reason is that the communication overhead is exacerbated for Criteo because of the larger model size (i.e., 4MB). For the GPU baseline system with mini-batch SGD, we only report a per batch speedup comparison to the UPMEM PIM system with GA-SGD. Training of Criteo’s high-dimensional sparse model is very slow on the GPU because only minimal computation is required for each sample. GA-SGD

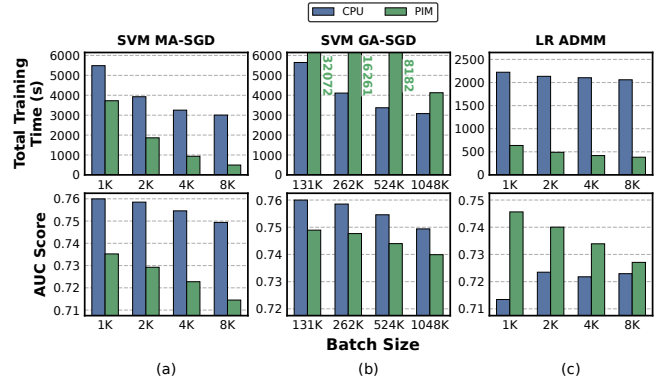
on the UPMEM PIM system achieves speedups of 10.65x per batch for SVM over the GPU running mini-batch SGD.

**Obsv. 16.** ADMM is faster on the UPMEM PIM for both LR and SVM compared to the CPU.

For LR (SVM) ADMM, we observe speedups of 4.36x (9.33x) on the UPMEM PIM system compared to the CPU baseline system. In contrast to the YFCC100M-HNfc6, we observe a speedup for LR when training on the Criteo dataset because, per sample, there is less computation.

**Takeaway 6.** The UPMEM PIM is a viable alternative to the CPU and the GPU for training high-dimensional sparse models on large-scale datasets.

**Batch Size.** In Fig. 11, we compare the total training time for 10 global epochs (y-axis; first row), the AUC score reached in the last global epoch (y-axis; second row), and varying batch size (x-axis). We illustrate a fixed combination of the model and optimization algorithm for SVM MA-SGD (Fig. 11(a)), SVM GA-SGD (Fig. 11(b)), and LR ADMM (Fig. 11(c)). Each subplot compares the UPMEM PIM system with 2048 DPUs and the CPU baseline system with 128 CPU threads for every batch size.



**Figure 11: Impact of batch size on total training time (10 global epochs) and AUC score (at the last global epoch) for SVM MA-SGD (a), SVM GA-SGD (b), and LR ADMM (c).**

**Obsv. 17.** As batch size increases, both the UPMEM PIM and the CPU exhibit performance improvement for MA-SGD. For ADMM, increasing the batch size only slightly improves performance for both the UPMEM PIM and the CPU.

When batch size increases from 1K to 8K, the total training time of SVM MA-SGD on the UPMEM PIM system (CPU baseline system) decreases by 7.53x (1.83x) from 3725.71s (5488.46s) to 494.98s (3004.8s). This is because larger batch sizes reduce the total amount of communication. The speedup on the UPMEM PIM system is higher compared to the CPU because the PIM system has more workers generating more communication, which benefits from the increase in batch size.

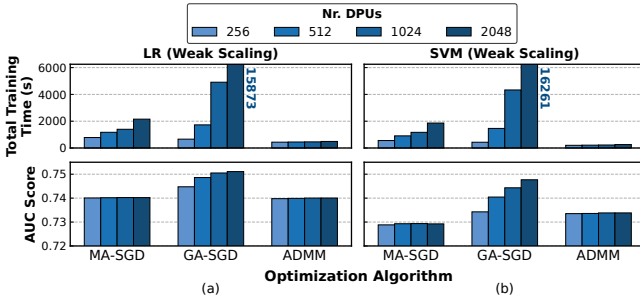
**Obsv. 18.** Both the UPMEM PIM and the CPU benefit from larger batch sizes for training high-dimensional sparse models with GA-SGD.

For SVM GA-SGD, for the UPMEM PIM system (CPU baseline system), we observe a reduction by 7.53x (1.83x) in total training time as we increase the batch size while for both the AUC score

only slightly degrades. This coincides with our Obsv. 7 for the YFCC100M-HNfc6 dataset.

**Takeaway 7.** When training high-dimensional sparse models, the UPMEM PIM has benefits for 1) models that are not sensitive to larger batch sizes, and 2) algorithms that require less inter-DPU communication via the parameter server.

**Scaling.** In Fig. 12, we study the weak scalability (i.e., the training dataset size increases proportionally as the number of DPUs increases) of using MA-SGD, GA-SGD, and ADMM (x-axis) to train LR (Fig. 12(a)) and SVM (Fig. 12(b)) models on the UPMEM PIM system. We plot the total training time for 10 global epochs (y-axis; first row) and the AUC score reached in the last global epoch (y-axis; second row). For all combinations of the models and optimization algorithms, we increase the number of DPUs from 256 to 2048 and proportionally increase the total training dataset size from 8.05GB to 64.42GB. For MA-SGD and ADMM, we set the batch size to 2K. For GA-SGD, we set the batch size to 262K.



**Figure 12: Impact of weak scaling on total training time (10 global epochs) and AUC score (at the last global epoch) for LR (a) and SVM (b).**

**Obsv. 19.** For high-dimensional sparse models, UPMEM PIM has good weak scalability with ADMM, but poor weak scalability with MA-SGD and GA-SGD in terms of total training time.

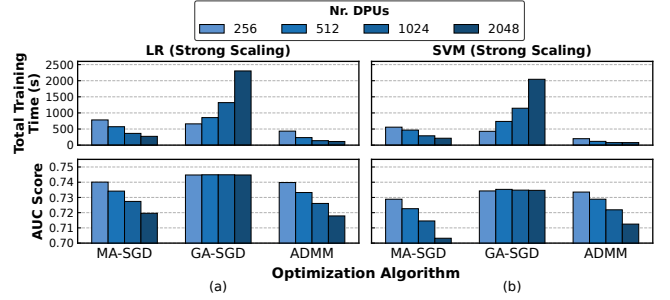
For example, for SVM ADMM (MA-SGD), we observe an increase of total training time by 1.28x (3.34x), while the achieved AUC score changes very slightly as we scale from 256 to 2048 DPUs. The difference in the increase in training time between ADMM and MA-SGD is higher compared to YFCC100M-HNfc6 (i.e., see Obsv. 8) because the communication overhead is exacerbated for Criteo’s larger high-dimensional sparse model.

**Obsv. 20.** Among the algorithms we test, only GA-SGD’s AUC score consistently increases when both the training dataset size and the number of DPUs increase.

For SVM GA-SGD, we observe an increase of total training time by 37.82x, while the achieved AUC score increases by 1.02x as we scale the number of DPUs from 256 to 2048. The observations follow the same reasoning as in Obsv. 9 for the YFCC100M-HNfc6 dataset.

In Fig. 13, we use the same experiment setting as in Fig. 12, except that we fix the training dataset size as we scale the number of DPUs (strong scaling).

**Obsv. 21.** For high-dimensional sparse models, the UPMEM PIM has good strong scalability in terms of total training time, but poor in terms of AUC score.



**Figure 13: Impact of strong scaling on total training time (10 global epochs) and AUC score (at the last global epoch) for LR (a) and SVM (b).**

For example, for LR ADMM (MA-SGD), we observe a speedup of 3.85x (2.87x), while the achieved AUC score decreases from 0.74 (0.74) to 0.718 (0.72), as we scale from 256 to 2048 DPUs. In contrast, for LR GA-SGD, we observe an increase in total training time by 3.49x, while the achieved AUC score changes only very slightly as we scale the number of DPUs from 256 to 2048. The smaller speedup of ADMM and MA-SGD, and even a slowdown for GA-SGD compared to YFCC100M-HNfc6 (i.e., see Obsv. 10), is a result of the larger models in Criteo that induce more communication overhead between the DPUs and the parameter server. The observed reduction of the AUC score follows the same line of reasoning as in the elaborations after Obsv. 10 for the YFCC100M-HNfc6 dataset.

**Takeaway 8.** The scalability potential of the UPMEM PIM for training high-dimensional sparse models is limited by its lack of direct inter-DPU communication.

## 6 Implications for PIM Hardware Design

Our evaluation (§5) demonstrates that a *real-world* PIM system (i.e., UPMEM) can be a viable alternative to state-of-the-art processor-centric architectures (e.g., CPU, GPU) for memory-bound ML training workloads involving large-scale datasets if 1) the optimization algorithms are carefully chosen to fit the PIM architecture, and 2) the arithmetic operations and data types are natively supported by the PIM hardware.

From our observations and analyses, we argue the most fundamental architectural design changes that future PIM architectures (including both UPMEM and other devices that do not have on-chip interconnect between PIM processing units like Samsung HBM-PIM [112], SK Hynix AiM [119]) should make to enable fast and efficient large-scale ML training is to enable more efficient communication among PIM processing units (e.g., DPUs in UPMEM) by adding interconnects and/or shared memory. Extending the UPMEM PIM system with on-chip interconnects (i.e., direct inter-DPU communication) enables the implementation of decentralized optimization algorithms (e.g., [9, 24, 28]). For instance, decentralized parallel SGD algorithms [114, 125, 175] are a promising solution to overcome scalability challenges of the *real-world* PIM system because of their two major advantages over its centralized counterparts: 1) decentralized optimization algorithms significantly reduce communication on the busiest node, and 2) decentralized parallel SGD theoretically provides approximately linear speedup in terms of computational complexity as we scale the number of nodes [125].

Without such on-chip interconnects, the advantages of using decentralized optimization algorithms to accelerate large-scale ML training workloads [114, 125, 127] cannot be leveraged and distributed ML workloads are significantly limited because of the high synchronization and communication overheads of centralized optimization algorithms, as our Takeaways 4 and 8 show.

For training workloads with even larger data and model sizes (e.g., training modern large language models with transformers on internet-scale text corpora [41]), we propose to further reduce the synchronization and communication overheads of PIM architectures by enabling the separate allocation of the training data from the model into memory units. This is because the model is frequently accessed and updated during ML training. In contrast, the training data is less frequently accessed as it is common to train such models only for a few epochs [200] to reduce cost.

We posit that a shift towards an algorithm-hardware codesign perspective is necessary in the context of ML training using PIM due to the high complexity of the design space, including algorithms, models, training, distributed system topology, and hardware design. With this paper, we hope to spark a data-driven discussion and further research that can truly unleash the full potential of PIM on ML training workloads.

## 7 Related Work

To our knowledge, this paper is the first to implement and rigorously evaluate distributed *Stochastic Gradient Descent* (SGD) algorithms on a *real-world* PIM system. We describe other related works on the UPMEM PIM system, PIM for ML training and inference, and distributed optimization algorithms.

**UPMEM PIM System.** Several prior works characterize and overview the UPMEM PIM architecture [52, 74, 81, 141, 154, 186]. Other works explore a variety of algorithms and applications on the UPMEM PIM system, such as compilers and programming models [31, 105], libraries [70, 95], simulation frameworks [92, 93], bioinformatics [32, 53, 115, 116], security [80, 99], analytics and databases [11–13, 100, 126], and ML training and inference [46, 72, 73, 76, 77, 110, 196, 203]. No prior work examines distributed SGD algorithms commonly used for data-intensive ML training workloads on the UPMEM PIM system.

**PIM for ML Training and Inference.** There are several works on PIM acceleration for ML training [56, 63, 161, 171, 181, 189]. However, none of these works use *real-world* PIM systems. Another body of work [10, 16, 17, 36, 39, 49, 65, 102, 103, 106, 112, 113, 119–121, 142, 149, 151, 172] focuses on accelerating ML inference using PIM, showing the effectiveness of PIM at mitigating the data movement bottleneck in ML inference.

**Distributed Optimization Algorithms.** Various works [4, 178, 192, 199] focus on algorithmically alleviating the communication overhead of centralized optimization algorithms since the parameter server has been identified to be the key bottleneck in distributed ML. Other works develop decentralized optimization algorithms to minimize communication among many nodes [114, 125, 127]. We believe that such algorithmic optimization combined with enhanced PIM hardware (see §6) can fundamentally improve ML training performance on large-scale datasets.

## 8 Conclusion

We evaluate and train ML models on large-scale datasets with centralized optimization algorithms on a *real-world* PIM system (i.e., UPMEM). We show that it is important to carefully *choose* the distributed optimization algorithm that fits the *real-world* PIM system and analyze tradeoffs. We demonstrate that commercial general-purpose PIM systems can be a viable alternative to processor-centric CPU and GPU architectures for many ML training workloads on large-scale datasets. Our results demonstrate the necessity of adapting PIM architectures to enable decentralized parallel SGD algorithms to overcome scalability challenges for many distributed ML training workloads.

## Acknowledgments

We thank the anonymous reviewers of PACT 2024 for feedback. We thank the SAFARI Research Group members for providing a stimulating intellectual environment. We thank UPMEM for providing hardware resources to perform this research. We acknowledge the generous gifts from our industrial partners, including Google, Huawei, Intel, and Microsoft. This work is supported in part by the Semiconductor Research Corporation (SRC), the ETH Future Computing Laboratory (EFCL), the European Union’s Horizon programme for research and innovation [101047160 - BioPIM], and the AI Chip Center for Emerging Smart Systems, sponsored by InnoHK funding, Hong Kong SAR (ACCESS).

## References

- [1] J. Ahn, S. Hong, S. Yoo, O. Mutlu, and K. Choi, “A Scalable Processing-In-Memory Accelerator for Parallel Graph Processing,” in *ISCA*, 2015.
- [2] J. Ahn, S. Yoo, O. Mutlu, and K. Choi, “PIM-Enabled Instructions: A Low-Overhead, Locality-Aware Processing-In-Memory Architecture,” in *ISCA*, 2015.
- [3] B. Akin, F. Franchetti, and J. C. Hoe, “Data Reorganization in Memory Using 3D-Stacked DRAM,” in *ISCA*, 2015.
- [4] D. Alistarh, D. Grubic, J. Li, R. Tomioka, and M. Vojnovic, “QSGD: Communication-Efficient SGD via Gradient Quantization and Encoding,” in *NeurIPS*, 2017.
- [5] G. Amato, F. Falchi, C. Gennaro, and F. Rabitti, “YFCC100M-HNfc6: A Large-scale Deep Features Benchmark for Similarity Search,” in *SISAP*, 2016.
- [6] AMD, “AMD EPYC 7742,” <https://www.amd.com/en/support/downloads/drivers.html/processors/epyc/epyc-7002-series/amd-epyc-7742.html>, 2019.
- [7] B. Asgari, R. Hadidi, J. Cao, S.-K. Lim, and H. Kim, “FAFNIR: Accelerating Sparse Gathering by Using Efficient Near-Memory Intelligent Reduction,” in *HPCA*, 2021.
- [8] H. Asghari-Moghaddam, Y. H. Son, J. H. Ahn, and N. S. Kim, “Chameleon: Versatile and practical near-DRAM acceleration architecture for large memory systems,” in *MICRO*, 2016.
- [9] T. C. Aysal, M. E. Yildiz, A. D. Sarwate, and A. Scaglione, “Broadcast Gossip Algorithms for Consensus,” *IEEE Transactions on Signal processing*, 2009.
- [10] E. Azarkhish, D. Rossi, I. Loi, and L. Benini, “Neurostream: Scalable and Energy Efficient Deep Learning with Smart Memory Cubes,” *TPDS*, 2017.
- [11] A. Baumstark, M. A. Jibril, and K.-U. Sattler, “Accelerating Large Table Scan Using Processing-In-Memory Technology,” *Datenbank-Spektrum*, 2023.
- [12] —, “Adaptive Query Compilation with Processing-in-Memory,” in *ICDEW*, 2023.
- [13] A. Bernhardt, A. Koch, and I. Petrov, “pimDB: From Main-Memory DBMS to Processing-In-Memory DBMS-Engines on Intelligent Memories,” in *DaMoN*, 2023.
- [14] M. Besta, R. Kanakagiri, G. Kwasniewski, R. Ausavarungrun, J. Beránek, K. Kanellopoulos, K. Janda, Z. Vónarburg-Shmária, L. Gianinazzi, I. Stefan *et al.*, “SISA: Set-Centric Instruction Set Architecture for Graph Mining on Processing-in-Memory Systems,” in *MICRO*, 2021.
- [15] A. Boroumand, “Practical Mechanisms for Reducing Processor–Memory Data Movement in Modern Workloads,” Ph.D. dissertation, Carnegie Mellon University, 2020.
- [16] A. Boroumand, S. Ghose, B. Akin, R. Narayanaswami, G. F. Oliveira, X. Ma, E. Shiu, and O. Mutlu, “Google Neural Network Models for Edge Devices:



Analyzing and Mitigating Machine Learning Inference Bottlenecks,” in *PACT*, 2021.

- [17] A. Boroumand, S. Ghose, Y. Kim, R. Ausavarungnirun, E. Shiu, R. Thakur, D. Kim, A. Kuusela, A. Knies, P. Ranganathan *et al.*, “Google Workloads for Consumer Devices: Mitigating Data Movement Bottlenecks,” in *ASPLOS*, 2018.
- [18] A. Boroumand, S. Ghose, G. F. Oliveira, and O. Mutlu, “Polynesia: Enabling Effective Hybrid Transactional/Analytical Databases with Specialized Hardware/Software Co-Design,” *arXiv:2103.00798*, 2021.
- [19] A. Boroumand, S. Ghose, M. Patel, H. Hassan, B. Lucia, R. Ausavarungnirun, K. Hsieh, N. Hajinazar, K. T. Malladi, H. Zheng *et al.*, “CoNDA: Efficient Cache Coherence Support for Near-Data Accelerators,” in *ISCA*, 2019.
- [20] A. Boroumand, S. Ghose, M. Patel, H. Hassan, B. Lucia, K. Hsieh, K. T. Malladi, H. Zheng, and O. Mutlu, “LazyPIM: An Efficient Cache Coherence Mechanism for Processing-in-Memory,” *ICAL*, 2016.
- [21] B. E. Boser, I. M. Guyon, and V. N. Vapnik, “A Training Algorithm for Optimal Margin Classifiers,” in *COLT*, 1992.
- [22] L. Bottou, “Large-Scale Machine Learning with Stochastic Gradient Descent,” in *COMPSTAT*, 2010.
- [23] —, *Stochastic Gradient Descent Tricks*. Springer, 2012.
- [24] S. Boyd, A. Ghosh, B. Prabhakar, and D. Shah, “Gossip Algorithms: Design, Analysis and Applications,” in *INFOCOM*, 2005.
- [25] S. Boyd, N. Parikh, E. Chu, B. Peleato, J. Eckstein *et al.*, “Distributed Optimization and Statistical Learning via the Alternating Direction Method of Multipliers,” *Foundations and Trends® in Machine Learning*, 2011.
- [26] S. P. Boyd and L. Vandenberghe, *Convex Optimization*. Cambridge University Press, 2004.
- [27] D. S. Cali, G. S. Kalsi, Z. Bingöl, C. Firtina, L. Subramanian, J. S. Kim, R. Ausavarungnirun, M. Alser, J. Gomez-Luna, A. Boroumand *et al.*, “GenASM: A High-Performance, Low-Power Approximate String Matching Acceleration Framework for Genome Sequence Analysis,” in *MICRO*, 2020.
- [28] R. Carli, F. Fagnani, P. Frasca, and S. Zampieri, “Gossip Consensus Algorithms via Quantized Communication,” *Automatica*, 2010.
- [29] K. K. Chang, “Understanding and Improving the Latency of DRAM-based Memory Systems,” Ph.D. dissertation, Carnegie Mellon University, 2017.
- [30] K. K. Chang, P. J. Nair, D. Lee, S. Ghose, M. K. Qureshi, and O. Mutlu, “Low-Cost Inter-Linked Subarrays (LISA): Enabling fast inter-subarray data movement in DRAM,” in *HPCA*, 2016.
- [31] J. Chen, J. Gómez-Luna, I. El Hajj, Y. Guo, and O. Mutlu, “SimplePIM: A Software Framework for Productive and Efficient Processing-in-Memory,” in *PACT*, 2023.
- [32] L.-C. Chen, C.-C. Ho, and Y.-H. Chang, “UpPipe: A Novel Pipeline Management on In-Memory Processors for RNA-seq Quantification,” in *DAC*, 2023.
- [33] P. Chi, S. Li, C. Xu, T. Zhang, J. Zhao, Y. Liu, Y. Wang, and Y. Xie, “PRIME: A Novel Processing-In-Memory Architecture for Neural Network Computation in ReRAM-based Main Memory,” in *ISCA*, 2016.
- [34] W.-S. Chin, Y. Zhuang, Y.-C. Juan, and C.-J. Lin, “A Fast Parallel Stochastic Gradient Method for Matrix Factorization in Shared Memory Systems,” *TIST*, 2015.
- [35] —, “A Learning-Rate Schedule for Stochastic Gradient Methods to Matrix Factorization,” in *PAKDD*, 2015.
- [36] S. Cho, H. Choi, E. Park, H. Shin, and S. Yoo, “McDRAM v2: In-Dynamic Random Access Memory Systolic Array Accelerator to Address the Large Model Problem in Deep Neural Networks on the Edge,” *IEEE Access*, 2020.
- [37] J. Choquette, W. Gandhi, O. Giroux, N. Stam, and R. Krashinsky, “NVIDIA A100 Tensor Core GPU: Performance and Innovation,” in *MICRO*, 2021.
- [38] E. Chung, J. Fowers, K. Ovtcharov, M. Papamichael, A. Caulfield, T. Massengill, M. Liu, D. Lo, S. Alkalay, M. Haselman *et al.*, “Serving DNNs in Real Time at Datacenter Scale with Project Brainwave,” in *MICRO*, 2018.
- [39] A. S. Cordeiro, S. R. dos Santos, F. B. Moreira, P. C. Santos, L. Carro, and M. A. Alves, “Machine Learning Migration for Efficient Near-Data Processing,” in *PDP*, 2021.
- [40] C. Cortes, “Support-Vector Networks,” *Machine Learning*, 1995.
- [41] R. Cotterrell, A. Svete, C. Meister, T. Liu, and L. Du, “Formal Aspects of Language Modeling,” *arXiv:2311.04329*, 2023.
- [42] B. Cottier, “Trends in the Dollar Training Cost of Machine Learning Systems,” 2023. [Online]. Available: <https://epochai.org/blog/trends-in-the-dollar-training-cost-of-machine-learning-systems>
- [43] Criteo AI Lab, “Criteo 1TB Click Logs Dataset,” <https://ailab.criteo.com/download-criteo-1tb-click-logs-dataset/>, 2014.
- [44] G. Dai, T. Huang, Y. Chi, J. Zhao, G. Sun, Y. Liu, Y. Wang, Y. Xie, and H. Yang, “GraphH: A Processing-in-Memory Architecture for Large-Scale Graph Processing,” *TCAD*, 2018.
- [45] G. Dai, Z. Zhu, T. Fu, C. Wei, B. Wang, X. Li, Y. Xie, H. Yang, and Y. Wang, “DIMining: Pruning-Efficient and Parallel Graph Mining on Near-Memory-Computing,” in *ISCA*, 2022.
- [46] P. Das, P. R. Sutradhar, M. Indovina, S. M. P. Dinakarrao, and A. Ganguly, “Implementation and Evaluation of Deep Neural Networks in Commercially Available Processing in Memory Hardware,” in *SOCC*, 2022.
- [47] C. De Sa, M. Feldman, C. Ré, and K. Olukotun, “Understanding and Optimizing Asynchronous Low-Precision Stochastic Gradient Descent,” in *ISCA*, 2017.
- [48] O. Dekel, R. Gilad-Bachrach, O. Shamir, and L. Xiao, “Optimal Distributed Online Prediction Using Mini-Batches,” *JMLR*, 2012.
- [49] Q. Deng, L. Jiang, Y. Zhang, M. Zhang, and J. Yang, “DrAcc: a DRAM based Accelerator for Accurate CNN Inference,” in *DAC*, 2018.
- [50] A. Denzler, G. F. Oliveira, N. Hajinazar, R. Bera, G. Singh, J. Gómez-Luna, and O. Mutlu, “Casper: Accelerating Stencil Computations Using Near-Cache Processing,” *IEEE Access*, 2023.
- [51] T. Dettmers, M. Lewis, Y. Belkada, and L. Zettlemoyer, “GPT3.int8(): 8-bit Matrix Multiplication for Transformers at Scale,” in *NeurIPS*, 2022.
- [52] F. Devaux, “The true Processing in Memory accelerator,” in *HCS*, 2019.
- [53] S. Diab, A. Nassereldine, M. Alser, J. Gómez Luna, O. Mutlu, and I. El Hajj, “A Framework for High-throughput Sequence Alignment using Real Processing-in-Memory Systems,” *Bioinformatics*, 2023.
- [54] M. Drumond, A. Daglis, N. Mirzadeh, D. Ustiugov, J. Picorel, B. Falsafi, B. Grot, and D. Pnevmatikatos, “The Mondrian Data Engine,” 2017.
- [55] C. Dünner, T. Parnell, D. Sarigiannis, N. Ioannou, A. Anghel, G. Ravi, M. Kandasamy, and H. Pozidis, “Snap ML: A Hierarchical Framework for Machine Learning,” in *NeurIPS*, 2018.
- [56] H. Falahati, P. Lotfi-Kamran, M. Sadrosadati, and H. Sarbazi-Azad, “ORIGAMI: A Heterogeneous Split Architecture for In-Memory Acceleration of Learning,” *arXiv:1812.11473*, 2018.
- [57] R.-E. Fan, “LIBSVM Data: A Collection of Benchmarks for Support Vector Machine Research,” <https://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/>.
- [58] A. Farmahini-Farahani, J. H. Ahn, K. Morrow, and N. S. Kim, “NDA: Near-DRAM Acceleration Architecture Leveraging Commodity DRAM Devices and Standard Memory Modules,” in *HPCA*, 2015.
- [59] I. Fernandez, C. Giannoula, A. Manglik, R. Quisilant, N. M. Ghiasi, J. Gómez-Luna, E. Gutierrez, O. Plata, and O. Mutlu, “MATSA: An MRAM-based Energy-Efficient Accelerator for Time Series Analysis,” *IEEE Access*, 2024.
- [60] I. Fernandez, R. Quisilant, E. Gutiérrez, O. Plata, C. Giannoula, M. Alser, J. Gómez-Luna, and O. Mutlu, “NATSA: A Near-Data Processing Accelerator for Time Series Analysis,” in *ICCD*, 2020.
- [61] J. D. Ferreira, G. Falcao, J. Gómez-Luna, M. Alser, L. Orosa, M. Sadrosadati, J. S. Kim, G. F. Oliveira, T. Shahroodi, A. Nori *et al.*, “pLUTo: Enabling Massively Parallel Computation in DRAM via Lookup Tables,” in *MICRO*, 2022.
- [62] F. Gao, G. Tziantzioulis, and D. Wentzlaff, “ComputeDRAM: In-Memory Compute Using Off-the-Shelf DRAMs,” in *MICRO*, 2019.
- [63] M. Gao, G. Ayers, and C. Kozyrakakis, “Practical Near-Data Processing for In-Memory Analytics Frameworks,” in *PACT*, 2015.
- [64] M. Gao and C. Kozyrakakis, “HRL: Efficient and flexible reconfigurable logic for near-data processing,” in *HPCA*, 2016.
- [65] M. Gao, J. Pu, X. Yang, M. Horowitz, and C. Kozyrakakis, “TETRIS: Scalable and Efficient Neural Network Acceleration with 3D Memory,” in *ASPLOS*, 2017.
- [66] S. Geman, E. Bienenstock, and R. Doursat, “Neural Networks and the Bias/Variance Dilemma,” *Neural Computation*, 1992.
- [67] N. M. Ghiasi, M. Sadrosadati, H. Mustafa, A. Gollwitzer, C. Firtina, J. Eudine, H. Mao, J. Lindegger, M. B. Cavlak, M. Alser, J. Park, and O. Mutlu, “MegIS: High-Performance, Energy-Efficient, and Low-Cost Metagenomic Analysis with In-Storage Processing,” in *ISCA*, 2024.
- [68] N. M. Ghiasi, N. Vijaykumar, G. F. Oliveira, L. Orosa, I. Fernandez, M. Sadrosadati, K. Kanellopoulos, N. Hajinazar, J. G. Luna, and O. Mutlu, “ALP: Alleviating CPU-Memory Data Movement Overheads in Memory-Centric Systems,” *IEEE Transactions on Emerging Topics in Computing*, 2022.
- [69] S. Ghose, A. Boroumand, J. S. Kim, J. Gómez-Luna, and O. Mutlu, “Processing-In-Memory: A Workload-Driven Perspective,” *IBM Journal of Research and Development*, 2019.
- [70] C. Giannoula, I. Fernandez, J. G. Luna, N. Koziris, G. Goumas, and O. Mutlu, “SparseP: Towards Efficient Sparse Matrix Vector Multiplication on Real Processing-In-Memory Architectures,” *POMACS*, 2022.
- [71] C. Giannoula, N. Vijaykumar, N. Papadopoulou, V. Karakostas, I. Fernandez, J. Gómez-Luna, L. Orosa, N. Koziris, G. Goumas, and O. Mutlu, “SynCron: Efficient Synchronization Support for Near-Data-Processing Architectures,” in *HPCA*, 2021.
- [72] C. Giannoula, P. Yang, I. F. Vega, J. Yang, Y. X. Li, J. G. Luna, M. Sadrosadati, O. Mutlu, and G. Pekhimenko, “Accelerating Graph Neural Networks on Real Processing-In-Memory Systems,” *arXiv:2402.16731*, 2024.
- [73] K. Gogineni, S. S. Dayapule, J. Gómez-Luna, K. Gogineni, P. Wei, T. Lan, M. Sadrosadati, O. Mutlu, and G. Venkataramani, “SwiftRL: Towards Efficient Reinforcement Learning on Real Processing-In-Memory Systems,” *arXiv:2405.03967*, 2024.
- [74] J. Gómez-Luna, I. El Hajj, I. Fernandez, C. Giannoula, G. F. Oliveira, and O. Mutlu, “Benchmarking Memory-Centric Computing Systems: Analysis of Real Processing-In-Memory Hardware,” in *IGSC*, 2021.
- [75] —, “Benchmarking a New Paradigm: Experimental Analysis and Characterization of a Real Processing-In-Memory System,” *IEEE Access*, 2022.
- [76] J. Gómez-Luna, Y. Guo, S. Brocard, J. Legriel, R. Cimadomo, G. F. Oliveira, G. Singh, and O. Boroumand, “An Experimental Evaluation of Machine Learning

- Training on a Real Processing-In-Memory System,” *arXiv:2207.07886*, 2022.
- [77] —, “Evaluating Machine Learning Workloads on Memory-Centric Computing Systems,” in *ISPASS*, 2023.
- [78] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. The MIT Press, 2016.
- [79] B. Gu, A. S. Yoon, D.-H. Bae, I. Jo, J. Lee, J. Yoon, J.-U. Kang, M. Kwon, C. Yoon, S. Cho *et al.*, “Biscuit: A Framework for Near-Data Processing of Big Data Workloads,” in *ISCA*, 2016.
- [80] H. Gupta, M. Kabra, J. Gómez-Luna, K. Kanellopoulos, and O. Mutlu, “Evaluating Homomorphic Operations on a Real-World Processing-In-Memory System,” in *IISWC*, 2023.
- [81] J. Gómez-Luna, I. E. Hajj, I. Fernandez, C. Giannoula, G. F. Oliveira, and O. Mutlu, “Benchmarking a New Paradigm: An Experimental Analysis of a Real Processing-In-Memory Architecture,” *arXiv:2105.03814*, 2021.
- [82] N. Hajinazar, G. F. Oliveira, S. Gregorio, J. D. Ferreira, N. M. Ghiasi, M. Patel, M. Alser, S. Ghose, J. Gómez-Luna, and O. Mutlu, “SIMDRAM: An End-to-End Framework for Bit-Serial SIMD Computing in DRAM,” in *ASPLOS*, 2021.
- [83] S. Han, H. Mao, and W. J. Dally, “Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding,” *arXiv:1510.00149*, 2015.
- [84] M. Hashemi, K. Hubaib, E. Ebrahimi, O. Mutlu, and Y. N. Patt, “Accelerating Dependent Cache Misses with an Enhanced Memory Controller,” in *ISCA*, 2016.
- [85] M. Hashemi, O. Mutlu, and Y. N. Patt, “Continuous Runahead: Transparent Hardware Acceleration for Memory Intensive Workloads,” in *MICRO*, 2016.
- [86] S. M. Hassan, S. Yalamanchili, and S. Mukhopadhyay, “Near Data Processing: Impact and Optimization of 3D Memory System Architecture on the Uncore,” in *MEMSYS*, 2015.
- [87] T. Hastie, R. Tibshirani, and J. Friedman, *The Elements of Statistical Learning*. Springer, 2009.
- [88] J. M. Herruzo, I. Fernandez, S. González-Navarro, and O. Plata, “Enabling Fast and Energy-Efficient FM-index Exact Matching using Processing-Near-Memory,” *The Journal of Supercomputing*, 2021.
- [89] K. Hsieh, E. Ebrahimi, G. Kim, N. Chatterjee, M. O’Connor, N. Vijaykumar, O. Mutlu, and S. W. Keckler, “Transparent Offloading and Mapping (TOM): Enabling Programmer-Transparent Near-Data Processing in GPU Systems,” in *ISCA*, 2016.
- [90] K. Hsieh, S. Khan, N. Vijaykumar, K. K. Chang, A. Boroumand, S. Ghose, and O. Mutlu, “Accelerating Pointer Chasing in 3D-Stacked Memory: Challenges, Mechanisms, Evaluation,” in *ICCD*, 2016.
- [91] J. Huang and C. X. Ling, “Using AUC and Accuracy in Evaluating Learning Algorithms,” *IEEE Transactions on Knowledge and Data Engineering*, 2005.
- [92] B. Hyun, T. Kim, D. Lee, and M. Rhu, “Pathfinding Future PIM Architectures by Demystifying a Commercial PIM Technology,” *arXiv:2308.00846*, 2023.
- [93] —, “Pathfinding Future PIM Architectures by Demystifying a Commercial PIM Technology,” in *HPCA*, 2024.
- [94] M. Imani, S. Gupta, Y. Kim, and T. Rosing, “FloatPIM: In-Memory Acceleration of Deep Neural Network Training with High Precision,” in *ISCA*, 2019.
- [95] M. Item, J. Gómez-Luna, G. F. Oliveira, M. Sadrosadati, Y. Guo, and O. Mutlu, “TransPimLib: Efficient Transcendental Functions for Processing-in-Memory Systems,” in *ISPASS*, 2023.
- [96] A. Ivanov, N. Dryden, T. Ben-Nun, S. Li, and T. Hoefler, “Data Movement is All You Need: A Case Study on Optimizing Transformers,” in *MLSys*, 2021.
- [97] H. Jiang, X. Peng, S. Huang, and S. Yu, “CIMAT: A Transpose SRAM-based Compute-In-Memory Architecture for Deep Neural Network On-Chip Training,” in *MEMSYS*, 2019.
- [98] J. Jiang, S. Gan, Y. Liu, F. Wang, G. Alonso, A. Klimovic, A. Singla, W. Wu, and C. Zhang, “Towards Demystifying Serverless Machine Learning Training,” in *SIGMOD*, 2021.
- [99] G. Jonatan, H. Cho, H. Son, X. Wu, N. Livesay, E. Mora, K. Shivdikar, J. L. Abellán, A. Joshi, D. Kaeli *et al.*, “Scalability Limitations of Processing-in-Memory using Real System Evaluations,” *POMACS*, 2024.
- [100] H. Kang, Y. Zhao, G. E. Blleloch, L. Dhulipala, Y. Gu, C. McGuffey, and P. B. Gibbons, “PIM-trie: A Skew-resistant Trie for Processing-in-Memory,” in *SPAA*, 2023.
- [101] W. H. Kautz, “Cellular Logic-in-Memory Arrays,” *IEEE Transactions on Computers*, 1969.
- [102] L. Ke, U. Gupta, B. Y. Cho, D. Brooks, V. Chandra, U. Diril, A. Firoozshahian, K. Hazelwood, B. Jia, H.-H. S. Lee *et al.*, “RecNMP: Accelerating Personalized Recommendation with Near-Memory Processing,” in *ISCA*, 2020.
- [103] L. Ke, X. Zhang, J. So, J.-G. Lee, S.-H. Kang, S. Lee, S. Han, Y. Cho, J. H. Kim, Y. Kwon *et al.*, “Near-Memory Processing in Action: Accelerating Personalized Recommendation With AXDIMM,” *MICRO*, 2021.
- [104] A. A. Khan, J. P. C. De Lima, H. Farzaneh, and J. Castrillon, “The Landscape of Compute-Near-Memory and Compute-In-Memory: A Research and Commercial Overview,” *arXiv:2401.14428*, 2024.
- [105] A. A. Khan, H. Farzaneh, K. F. Friebe, C. Fournier, L. Chelini, and J. Castrillon, “CINM (Cinnamon): A Compilation Infrastructure for Heterogeneous Compute In-Memory and Compute Near-Memory Paradigms,” *arXiv:2301.07486*, 2022.
- [106] B. Kim, J. Chung, E. Lee, W. Jung, S. Lee, J. Choi, J. Park, M. Wi, S. Lee, and J. H. Ahn, “MViD: Sparse Matrix-Vector Multiplication in Mobile DRAM for Accelerating Recurrent Neural Networks,” *IEEE Transactions on Computers*, 2020.
- [107] D. Kim, J. Kung, S. Chai, S. Yalamanchili, and S. Mukhopadhyay, “Neurocube: A Programmable Digital Neuromorphic Architecture with High-Density 3D Memory,” in *ISCA*, 2016.
- [108] H. Kim, H. Park, T. Kim, K. Cho, E. Lee, S. Ryu, H.-J. Lee, K. Choi, and J. Lee, “GradPIM: A Practical Processing-in-DRAM Architecture for Gradient Descent,” in *HPCA*, 2021.
- [109] J. S. Kim, D. Senol, H. Xin, D. Lee, S. Ghose, M. Alser, H. Hassan, O. Ergin, C. Alkan, and O. Mutlu, “GRIM-Filter: Fast Seed Location Filtering in DNA Read Mapping Using Processing-in-Memory Technologies,” *arXiv:1708.04329*, 2017.
- [110] S. Y. Kim, J. Lee, Y. Paik, C. H. Kim, W. J. Lee, and S. W. Kim, “Optimal Model Partitioning with Low-Overhead Profiling on the PIM-based Platform for Deep Learning Inference,” *TODAES*, 2024.
- [111] D. P. Kingma, “Adam: A Method for Stochastic Optimization,” *arXiv:1412.6980*, 2014.
- [112] Y.-C. Kwon, S. H. Lee, J. Lee, S.-H. Kwon, J. M. Ryu, J.-P. Son, O. Seongil, H.-S. Yu, H. Lee, S. Y. Kim *et al.*, “25.4 A 20nm 6GB Function-In-Memory DRAM, Based on HBM2 with a 1.2TFLOPS Programmable Computing Unit Using Bank-Level Parallelism, for Machine Learning Applications,” in *ISSCC*, 2021.
- [113] Y. Kwon, Y. Lee, and M. Rhu, “TensorDIMM: A Practical Near-Memory Processing Architecture for Embeddings and Tensor Operations in Deep Learning,” in *MICRO*, 2019.
- [114] G. Lan, S. Lee, and Y. Zhou, “Communication-Efficient Algorithms for Decentralized and Stochastic Optimization,” *Mathematical Programming*, 2020.
- [115] D. Lavenier, R. Cimadomo, and R. Jodin, “Variant Calling Parallelization on Processor-in-Memory Architecture,” in *BBM*, 2020.
- [116] D. Lavenier, C. Deltel, D. Furodet, and J.-F. Roy, “BLAST on UPMEM,” Ph.D. dissertation, INRIA Rennes-Bretagne Atlantique, 2016.
- [117] D. Lee, J. So, M. Ahn, J.-G. Lee, J. Kim, J. Cho, R. Oliver, V. C. Thummala, R. s. JV, S. S. Upadhyay *et al.*, “Improving In-Memory Database Operations with Acceleration DIMM (AxDIMM),” in *DaMoN*, 2022.
- [118] J. H. Lee, J. Sim, and H. Kim, “BSSync: Processing Near Memory for Machine Learning Workloads with Bounded Staleness Consistency Models,” in *PACT*, 2015.
- [119] S. Lee, K. Kim, S. Oh, J. Park, G. Hong, D. Ka, K. Hwang, J. Park, K. Kang, J. Kim *et al.*, “A 1nm 1.25V 8Gb, 16Gb/s/pin GDDR6-based Accelerator-in-Memory supporting 1TFLOPS MAC Operation and Various Activation Functions for Deep-Learning Applications,” in *ISSCC*, 2022.
- [120] S. Lee, S.-h. Kang, J. Lee, H. Kim, E. Lee, S. Seo, H. Yoon, S. Lee, K. Lim, H. Shin *et al.*, “Hardware Architecture and Software Stack for PIM Based on Commercial DRAM Technology,” in *ISCA*, 2021.
- [121] Y. S. Lee and T. H. Han, “Task Parallelism-Aware Deep Neural Network Scheduling on Multiple Hybrid Memory Cube-based Processing-in-Memory,” *IEEE Access*, 2021.
- [122] B. Li, J. R. Doppa, P. P. Pande, K. Chakrabarty, J. X. Qiu, and H. Li, “3D-ReG: A 3D ReRAM-based Heterogeneous Architecture for Training Deep Neural Networks,” *JETC*, 2020.
- [123] M. Li, D. G. Andersen, A. J. Smola, and K. Yu, “Communication Efficient Distributed Machine Learning with the Parameter Server,” 2014.
- [124] M. Li, T. Zhang, Y. Chen, and A. J. Smola, “Efficient Mini-Batch Training for Stochastic Optimization,” in *SIGKDD*, 2014.
- [125] X. Lian, C. Zhang, H. Zhang, C.-J. Hsieh, W. Zhang, and J. Liu, “Can Decentralized Algorithms Outperform Centralized Algorithms? A Case Study for Decentralized Parallel Stochastic Gradient Descent,” in *NeurIPS*, 2017.
- [126] C. Lim, S. Lee, J. Choi, J. Lee, S. Park, H. Kim, J. Lee, and Y. Kim, “Design and Analysis of a Processing-in-DIMM Join Algorithm: A Case Study with UPMEM DIMMs,” *PACMOD*, 2023.
- [127] J. Liu, C. Zhang *et al.*, “Distributed Learning Systems with First-Order Methods,” *Foundations and Trends® in Databases*, 2020.
- [128] J. Liu, H. Zhao, M. A. Ogleari, D. Li, and J. Zhao, “Processing-In-Memory for Energy-Efficient Neural Network Training: A Heterogeneous Approach,” in *MICRO*, 2018.
- [129] Z. Liu, I. Calciu, M. Herlihy, and O. Mutlu, “Concurrent Data Structures for Near-Memory Computing,” in *SPAA*, 2017.
- [130] Y. Luo and S. Yu, “Benchmark Non-Volatile and Volatile Memory Based Hybrid Precision Synapses for In-Situ Deep Neural Network Training,” in *ASP-DAC*, 2020.
- [131] D. Mahajan, J. Park, E. Amaro, H. Sharma, A. Yazdanbakhsh, J. K. Kim, and H. Esmaeilzadeh, “TABLA: A Unified Template-based Framework for Accelerating Statistical Machine Learning,” in *HPCA*, 2016.
- [132] N. Mansouri Ghiasi, J. Park, H. Mustafa, J. Kim, A. Olgun, A. Gollwitzer, D. Senol Cali, C. Firtina, H. Mao, N. Almadhoun Alser *et al.*, “GenStore: A High-Performance In-Storage Processing System for Genome Sequence Analysis,” in *ASPLOS*, 2022.

- [133] H. Mao, M. Alser, M. Sadrosadati, C. Firtina, A. Baranwal, D. S. Cali, A. Manglik, N. A. Alser, and O. Mutlu, "GenPIP: In-Memory Acceleration of Genome Analysis via Tight Integration of Basecalling and Read Mapping," in *MICRO*, 2022.
- [134] R. McDonald, K. Hall, and G. Mann, "Distributed Training Strategies for the Structured Perceptron," in *NAACL HLT*, 2010.
- [135] O. Mutlu, "Intelligent Architectures for Intelligent Computing Systems," *DATE*, 2021.
- [136] —, "Memory-Centric Computing," in *DAC*, 2023.
- [137] O. Mutlu, S. Ghose, J. Gómez-Luna, and R. Ausavarungnirun, "Enabling Practical Processing in and near Memory for Data-Intensive Computing," in *DAC*, 2019.
- [138] —, "Processing Data Where It Makes Sense: Enabling In-Memory Computation," *Microprocessors and Microsystems*, 2019.
- [139] —, "A Modern Primer on Processing in Memory," in *Emerging Computing: From Devices to Systems: Looking Beyond Moore and Von Neumann*. Springer, 2022.
- [140] L. Nai, R. Hadidi, J. Sim, H. Kim, P. Kumar, and H. Kim, "GraphPIM: Enabling Instruction-Level PIM Offloading in Graph Computing Frameworks," in *HPCA*, 2017.
- [141] J. Nider, C. Mustard, A. Zoltan, J. Ramsden, L. Liu, J. Grossbard, M. Dashti, R. Jodin, A. Ghiti, J. Chauzi *et al.*, "A Case Study of Processing-in-Memory in off-the-Shelf Systems," in *USENIX*, 2021.
- [142] D. Niu, S. Li, Y. Wang, W. Han, Z. Zhang, Y. Guan, T. Guan, F. Sun, F. Xue, L. Duan *et al.*, "184QPS/W 64Mb/mm<sup>2</sup> 3D Logic-to-DRAM Hybrid Bonding with Process-Near-Memory Engine for Recommendation System," in *ISSCC*, 2022.
- [143] NVIDIA, "NVIDIA A100 Tensor Core GPU Architecture. White Paper," <https://images.nvidia.com/aem-dam/en-zz/Solutions/data-center/nvidia-ampere-architecture-whitepaper.pdf>, 2020.
- [144] A. Olgun, J. G. Luna, K. Kanellopoulos, B. Salami, H. Hassan, O. Ergin, and O. Mutlu, "PiDRAM: A Holistic End-to-end FPGA-based Framework for Processing-in-DRAM," *TACO*, 2022.
- [145] G. F. Oliveira, A. Boroumand, S. Ghose, J. Gómez-Luna, and O. Mutlu, "Heterogeneous Data-Centric Architectures for Modern Data-Intensive Applications: Case Studies in Machine Learning and Databases," in *ISVLSI*, 2022.
- [146] G. F. Oliveira, J. Gómez-Luna, L. Orosa, S. Ghose, N. Vijaykumar, I. Fernandez, M. Sadrosadati, and O. Mutlu, "DAMOV: A New Methodology and Benchmark Suite for Evaluating Data Movement Bottlenecks," *IEEE Access*, 2021.
- [147] G. F. Oliveira, A. Kohli, D. Novo, J. Gómez-Luna, and O. Mutlu, "DaPPA: A Data-Parallel Framework for Processing-In-Memory Architectures," *arXiv:2310.10168*, 2023.
- [148] G. F. Oliveira, A. Olgun, A. G. Yağlıkcı, F. N. Bostancı, J. Gómez-Luna, S. Ghose, and O. Mutlu, "MIMDRAM: An End-to-End Processing-Using-DRAM System for High-Throughput, Energy-Efficient and Programmer-Transparent Multiple-Instruction Multiple-Data Computing," in *HPCA*, 2024.
- [149] J. Park, B. Kim, S. Yun, E. Lee, M. Rhu, and J. H. Ahn, "TRiM: Enhancing Processor-Memory Interfaces with Scalable Tensor Reduction in Memory," in *MICRO*, 2021.
- [150] J. Park, R. Azizi, G. F. Oliveira, M. Sadrosadati, R. Nadig, D. Novo, J. Gómez-Luna, M. Kim, and O. Mutlu, "Flash-Cosmos: In-Flash Bulk Bitwise Operations Using Inherent Computation Capability of NAND Flash Memory," in *MICRO*, 2022.
- [151] N. Park, S. Ryu, J. Kung, and J.-J. Kim, "High-throughput Near-Memory Processing on CNNs with 3D HBM-like Memory," *TODAES*, 2021.
- [152] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimeshin, L. Antiga *et al.*, "PyTorch: An Imperative Style, High-Performance Deep Learning Library," in *NeurIPS*, 2019.
- [153] A. Pattnaik, X. Tang, A. Jog, O. Kayiran, A. K. Mishra, M. T. Kandemir, O. Mutlu, and C. R. Das, "Scheduling Techniques for GPU Architectures with Processing-In-Memory Capabilities," in *PACT*, 2016.
- [154] B. Peccerillo, M. Mannino, A. Mondelli, and S. Bartolini, "A Survey on Hardware Accelerators: Taxonomy, Trends, Challenges, and Perspectives," *Journal of Systems Architecture*, 2022.
- [155] B. T. Polyak, *Introduction to Optimization*. Optimization Software, 1987.
- [156] S. H. Pugsley, J. Jests, H. Zhang, R. Balasubramanian, V. Srinivasan, A. Buyuktosunoglu, A. Davis, and F. Li, "NDC: Analyzing the Impact of 3D-Stacked Memory+Logic Devices on MapReduce Workloads," in *ISPASS*, 2014.
- [157] H. Robbins and S. Monro, "A Stochastic Approximation Method," *The Annals of Mathematical Statistics*, 1951.
- [158] S. Rudner, "An Overview of Gradient Descent Optimization Algorithms," *arXiv:1609.04747*, 2016.
- [159] SAFARI Research Group, "PIM-Opt Artifact — GitHub Repository," <https://github.com/CMU-SAFARI/PIM-Opt>, 2024.
- [160] —, "PIM-Opt Artifact — Zenodo Repository," <https://doi.org/10.5281/zenodo.12747665>, 2024.
- [161] J. Saikia, S. Yin, Z. Jiang, M. Seok, and J.-s. Seo, "K-Nearest Neighbor Hardware Accelerator Using In-Memory Computing SRAM," in *ISLPED*, 2019.
- [162] F. Schuiki, M. Schaffner, F. K. Gürkaynak, and L. Benini, "A Scalable Near-Memory Architecture for Training Deep Neural Networks on Large In-Memory Datasets," *IEEE Transactions on Computers*, 2018.
- [163] V. Seshadri, "Simple DRAM and Virtual Memory Abstractions to Enable Highly Efficient Memory Systems," *arXiv:1605.06483*, 2016.
- [164] V. Seshadri, K. Hsieh, A. Boroum, D. Lee, M. A. Kozuch, O. Mutlu, P. B. Gibbons, and T. C. Mowry, "Fast Bulk Bitwise AND and OR in DRAM," *ICAL*, 2015.
- [165] V. Seshadri, Y. Kim, C. Fallin, D. Lee, R. Ausavarungnirun, G. Pekhimenko, Y. Luo, O. Mutlu, P. B. Gibbons, M. A. Kozuch *et al.*, "RowClone: Fast and Energy-Efficient In-DRAM Bulk Data Copy and Initialization," in *MICRO*, 2013.
- [166] V. Seshadri, D. Lee, T. Mullins, H. Hassan, A. Boroumand, J. Kim, M. A. Kozuch, O. Mutlu, P. B. Gibbons, and T. C. Mowry, "Buddy-RAM: Improving the Performance and Efficiency of Bulk Bitwise Operations Using DRAM," *arXiv:1611.09988*, 2016.
- [167] —, "Ambit: In-Memory Accelerator for Bulk Bitwise Operations Using Commodity DRAM Technology," in *MICRO*, 2017.
- [168] V. Seshadri and O. Mutlu, "Simple Operations in Memory to Reduce Data Movement," in *Advances in Computers*. Elsevier, 2017.
- [169] —, "In-DRAM Bulk Bitwise Execution Engine," *arXiv:1905.09822*, 2019.
- [170] T. Shahroodi, G. Singh, M. Zahedi, H. Mao, J. Lindegger, C. Firtina, S. Wong, O. Mutlu, and S. Hamdioui, "Swordfish: A Framework for Evaluating Deep Neural Network-based Basecalling using Computation-In-Memory with Non-Ideal Memristors," in *MICRO*, 2023.
- [171] C. F. Shelor and K. M. Kavi, "Reconfigurable Dataflow Graphs for Processing-In-Memory," in *ICDCN*, 2019.
- [172] H. Shin, D. Kim, E. Park, S. Park, Y. Park, and S. Yoo, "McDRAM: Low Latency and Energy-Efficient Matrix Computations in DRAM," *TCAD*, 2018.
- [173] G. Singh, D. Diamantopoulos, C. Hagleitner, J. Gomez-Luna, S. Stuijk, O. Mutlu, and H. Corporaal, "NERO: A Near High-Bandwidth Memory Stencil Accelerator for Weather Prediction Modeling," in *FPL*, 2020.
- [174] G. Singh, J. Gómez-Luna, G. Mariani, G. F. Oliveira, S. Corda, S. Stuijk, O. Mutlu, and H. Corporaal, "NAPEL: Near-Memory Computing Application Performance Prediction via Ensemble Learning," in *DAC*, 2019.
- [175] B. Sirb and X. Ye, "Consensus Optimization with Delayed and Stochastic Gradients on Decentralized Networks," in *Big Data*, 2016.
- [176] S. U. Stich, J.-B. Cordonnier, and M. Jaggi, "Sparsified SGD with Memory," in *NeurIPS*, 2018.
- [177] H. S. Stone, "A Logic-in-Memory Computer," *IEEE Transactions on Computers*, 1970.
- [178] N. Ström, "Scalable Distributed DNN Training using Commodity GPU Cloud Computing," *Sixteenth Annual Conference of the International Speech Communication Association*, 2015.
- [179] H. Sun, Z. Zhu, Y. Cai, X. Chen, Y. Wang, and H. Yang, "An Energy-Efficient Quantized and Regularized Training Framework for Processing-In-Memory Accelerators," in *ASP-DAC*, 2020.
- [180] W. Sun, Z. Li, S. Yin, S. Wei, and L. Liu, "ABC-DIMM: Alleviating the Bottleneck of Communication in DIMM-based Near-Memory Processing with Inter-DIMM Broadcast," in *ISCA*, 2021.
- [181] Z. Sun, G. Pedretti, A. Bricalli, and D. Ielmini, "One-Step Regression and Classification with Cross-Point Resistive Memory Arrays," *Science Advances*, 2020.
- [182] B. Thomee, D. A. Shamma, G. Friedland, B. Elizalde, K. Ni, D. Poland, D. Borth, and L.-J. Li, "YFCC100M: The New Data in Multimedia Research," *Communications of the ACM*, 2016.
- [183] Y. Umuroglu, N. J. Fraser, G. Gambardella, M. Blott, P. Leong, M. Jahre, and K. Visser, "FINN: A Framework for Fast, Scalable Binarized Neural Network Inference," in *FPGA*, 2017.
- [184] UPMEM, "Product Sheet UPMEM," 2022.
- [185] UPMEM, "UPMEM Processing In-Memory (PIM)," UPMEM PIM Tech Paper, 2022.
- [186] UPMEM, "UPMEM PIM Platform for Data-Intensive Applications," in *ABUMPIMP*. Symposium as part of Euro-Par, 2023.
- [187] —, "UPMEM SDK, Version 2023.2.0," <https://sdk.upmem.com/2023.2.0/>, 2023.
- [188] —, "UPMEM Website," <https://www.upmem.com>, 2024.
- [189] J. Vieira, N. Roma, P. Tomás, P. Ienne, and G. Falcao, "Exploiting Compute Caches for Memory Bound Vector Operations," in *SBAC-PAD*, 2018.
- [190] P. Villalobos, J. Sevilla, L. Heim, T. Besiroglu, M. Hobbhahn, and A. Ho, "Will we run out of Data? An Analysis of the Limits of scaling datasets in Machine Learning," *arXiv:2211.04325*, 2022.
- [191] J. Wang, W. Wang, and N. Srebro, "Memory and Communication Efficient Distributed Stochastic Optimization with Minibatch Prox," in *COLT*, 2017.
- [192] J. Wang, Y. Lu, B. Yuan, B. Chen, P. Liang, C. De Sa, C. Re, and C. Zhang, "CocktailSGD: Fine-Tuning Foundation Models over 500Mbps Networks," in *ICML*, 2023.
- [193] M. Wang, W. Fu, X. He, S. Hao, and X. Wu, "A Survey on Large-Scale Machine Learning," *IEEE Transactions on Knowledge and Data Engineering*, 2020.
- [194] Z. Wang, K. Kara, H. Zhang, G. Alonso, O. Mutlu, and C. Zhang, "Accelerating Generalized Linear Models with MLWeaving: A One-Size-Fits-All System for Any-Precision Learning," in *VLDB*, 2019.
- [195] M. Wortsman, S. Gururangan, S. Li, A. Farhadi, L. Schmidt, M. Rabbat, and A. S. Morcos, "Lo-fi: Distributed Fine-tuning Without Communication,"

- arXiv:2210.11948*, 2022.
- [196] Y. Wu, Z. Wang, and W. D. Lu, "PIM-GPT: A Hybrid Process-in-Memory Accelerator for Autoregressive Transformers," *arXiv:2310.09385*, 2023.
  - [197] H. Xi, C. Li, J. Chen, and J. Zhu, "Training Transformers with 4-bit Integers," in *NeurIPS*, 2023.
  - [198] X. Xie, W. Tan, L. L. Fong, and Y. Liang, "CuMF\_SGD: Parallelized Stochastic Gradient Descent for Matrix Factorization on GPUs," in *HPDC*, 2017.
  - [199] H. Xu, C.-Y. Ho, A. M. Abdelmoniem, A. Dutta, E. H. Bergou, K. Karatsenidis, M. Canini, and P. Kalnis, "Compressed Communication for Distributed Deep Learning: Survey and Quantitative Evaluation," Technical Report, 2020.
  - [200] F. Xue, Y. Fu, W. Zhou, Z. Zheng, and Y. You, "To Repeat or Not To Repeat: Insights from Scaling LLM under Token-Crisis," in *NeurIPS*, 2024.
  - [201] H. Yu, S. Yang, and S. Zhu, "Parallel Restarted SGD with Faster Convergence and Less Communication: Demystifying Why Model Averaging Works for Deep Learning," in *AAAI*, 2019.
  - [202] İ. E. Yüksel, Y. C. Tuğrul, A. Olgun, F. N. Bostancı, A. G. Yağlıkcı, G. F. Oliveira, H. Luo, J. Gómez-Luna, M. Sadrosadati, and O. Mutlu, "Functionally-Complete Boolean Logic in Real DRAM Chips: Experimental Characterization and Analysis," in *HPCA*, 2024.
  - [203] N. Zarif, "Offloading Embedding Lookups to Processing-In-Memory for Deep Learning Recommender Models," Master's thesis, University of British Columbia, 2023.
  - [204] C. Zhang and C. Ré, "DimmWitted: A Study of Main-Memory Statistical Analytics," *arXiv:1403.7550*, 2014.
  - [205] D. Zhang, N. Jayasena, A. Lyashevsky, J. L. Greathouse, L. Xu, and M. Ignatowski, "TOP-PIM: Throughput-Oriented Programmable Processing in Memory," in *HPDC*, 2014.
  - [206] J. Zhang, C. De Sa, I. Mitliagkas, and C. Ré, "Parallel SGD: When does averaging help?" *arXiv:1606.07365*, 2016.
  - [207] Z. Zhang, J. Jiang, W. Wu, C. Zhang, L. Yu, and B. Cui, "MLlib": Fast Training of GLMs Using Spark MLlib," in *ICDE*, 2019.
  - [208] F. Zhou and G. Cong, "On the Convergence Properties of a K-step Averaging Stochastic Gradient Descent Algorithm for Nonconvex Optimization," *arXiv:1708.01012*, 2017.
  - [209] P. Zhou, J. Feng, C. Ma, C. Xiong, S. C. H. Hoi *et al.*, "Towards Theoretically Understanding Why SGD Generalizes Better Than ADAM in Deep Learning," in *NeurIPS*, 2020.
  - [210] Q. Zhu, T. Graf, H. E. Sumbul, L. Pileggi, and F. Franchetti, "Accelerating Sparse Matrix-Matrix Multiplication with 3D-Stacked Logic-in-Memory Hardware," in *HPEC*, 2013.
  - [211] Y. Zhuo, C. Wang, M. Zhang, R. Wang, D. Niu, Y. Wang, and X. Qian, "GraphQ: Scalable PIM-based Graph Processing," in *MICRO*, 2019.
  - [212] M. Zinkevich, M. Weimer, L. Li, and A. Smola, "Parallelized Stochastic Gradient Descent," in *NeurIPS*, 2010.



## A Artifact Appendix

### A.1 Abstract

Our artifact [159, 160] contains the source code and scripts needed to reproduce our results, including all figures in the paper. We provide: 1) source code to preprocess the YFCC100M-HNfc6 [5] and Criteo 1TB Click Logs [43] datasets preprocessed by LIBSVM [57], 2) the source code to perform experiments on the UPMEM PIM System, 3) the source code of the CPU and GPU baseline implementations, and 4) the source code to postprocess and evaluate results. We provide Python scripts and a Jupyter Notebook to analyze and plot the results.

### A.2 Artifact check-list (meta-information)

Parameter	Value
Program	C programs Python3 scripts/Jupyter Notebook Shell scripts
Compilation	gcc (Debian 8.3.0-6) 8.3.0 GNU Make 4.2.1
Run-time environment	Debian GNU/Linux 10 (buster) (UPMEM PIM System) Ubuntu 22.04.1 LTS (CPU Baseline System) Ubuntu 22.04.3 LTS (GPU Baseline System) Python 3.10.6 slurm-wlm 21.08.5 tmux 2.8+
Hardware	2x Intel Xeon Silver 4215 8-core processor @ 2.50GHz, 20x8 GB UPMEM PIM modules (UPMEM PIM System) 2x AMD EPYC 7742 64-core processor @ 2.25GHz (CPU Baseline System) 2x Intel Xeon Gold 5118 12-core processor @ 2.30GHz, 1x NVIDIA A100 (PCIe, 80 GB) (GPU Baseline System)
Output	Data and execution logs in plain text and plots in pdf and png format
Metrics	Runtime, Test Accuracy, AUC Score, Binary Cross Entropy Loss, and Hinge Loss
Experiment workflow	Preprocess datasets, perform experiments on UPMEM PIM System, run experiments on CPU Baseline System and GPU baseline, postprocess results, and run analysis scripts on the results
Disk space requirement	≈ 12TB
Workflow preparation time	≈ 3 days to preprocess YFCC100M-HNfc6 dataset ≈ 20 hours to preprocess Criteo 1TB Click Logs dataset ≈ 2 days to perform experiments using the UPMEM PIM System on the YFCC100M-HNfc6 dataset ≈ 1 week to perform experiments using the UPMEM PIM System on the Criteo 1TB Click Logs dataset ≈ 16 hours to perform experiments using the CPU Baseline System on the YFCC100M-HNfc6 dataset
Experiment completion time	≈ 8 hours to perform experiments using the CPU Baseline System on the Criteo 1TB Click Logs dataset ≈ 1 hour to perform experiments using the GPU Baseline System on the YFCC100M-HNfc6 dataset ≈ 12 hours to postprocess CPU and GPU Baseline System results ≈ 2 days to postprocess UPMEM PIM System results ≈ 1 hour to aggregate results, and reproduce plots
Publicly available?	Zenodo ( <a href="https://doi.org/10.5281/zenodo.12747665">https://doi.org/10.5281/zenodo.12747665</a> ) Github ( <a href="https://github.com/CMU-SAFARI/PIM-Opt">https://github.com/CMU-SAFARI/PIM-Opt</a> )
Code licenses	MIT

### A.3 Description

**A.3.1 How to access** The artifact is available on Zenodo with DOI <https://doi.org/10.5281/zenodo.12747665>. The live Github repository is at <https://github.com/CMU-SAFARI/PIM-Opt>.

**A.3.2 Hardware dependencies** Our hardware dependencies are listed in Table 1. For preprocessing of the datasets (§A.5.1) and postprocessing of the results (§A.5.4), the same hardware configuration as the CPU Baseline System is used.

#### A.3.3 Software dependencies

- gcc (Debian 8.3.0-6) 8.3.0, GNU Make 4.2.1
- UPMEM SDK, version 2023.2.0 [187]
- tar (GNU tar) 1.34

- Zip 3.0
- Python 3.10.6
- pip packages pandas, numpy, scipy, scikit-learn, matplotlib, seaborn, torch, coloredlogs
- slurm-wlm 21.08.5
- tmux 2.8+
- CUDA 11.7

**A.3.4 Datasets** In this paper, we use two large-scale datasets:

- YFCC100M-HNfc6 [5] can be requested at <http://www.deepfeatures.org/index.html>. For preprocessing, one needs to download the file `yfcc100m_autotags.bz2` from the original YFCC100M dataset [182], which can be requested at <https://www.multimediacommons.org>.
- Criteo 1TB Click Logs [43] preprocessed by LIBSVM [57] can be accessed by running:

```
$ wget -t inf https://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/binary/criteo_tb.svm.tar.xz
$ tar -xJvf criteo_tb.svm.tar.xz
```

### A.4 Installation

To reproduce our results, no extra installation steps are required besides installing the dependencies described in §A.3.3. We recommend using a terminal multiplexer (e.g., `tmux`) to ensure that experiments are completed without interruption.

### A.5 Experiment workflow

We describe the steps and commands to reproduce our results, including all figures in the paper, in this section. Note that we assume the use of `slurm` workload manager on a cluster. Readers with other workload managers should modify the scripts to fit their own environment.

**A.5.1 Preprocessing Datasets** The preprocessing of the datasets YFCC100M-HNfc6 and Criteo is initialized by running the commands:

```
$ cd preprocessing
$ DATA_ROOT=<path-to-data>
$ PARTITION=<name-of-slurm-partition>
$ NODE=<name-of-slurm-node>
$ ./run_preprocessing.sh ${DATA_ROOT} ${PARTITION} ${NODE} &
```

**A.5.2 UPMEM PIM System Experiments** To perform the experiments on the UPMEM PIM system, readers can run the command:

```
$ cd upmem_ml_coding/UPMEM
$ DATA_ROOT=<path-to-data>
$ ./run_upmem_experiments.sh ${DATA_ROOT} &
```

**A.5.3 CPU and GPU Baseline Experiments** The baseline experiments are launched by running the commands:

```
$ cd baseline
$ DATA_ROOT=<path-to-data>
$ PARTITION=<name-of-slurm-partition>
$ NODE_CPU=<name-of-slurm-cpu-node>
$ NODE_GPU=<name-of-slurm-gpu-node>
$ ./run_baseline_experiments.sh ${DATA_ROOT} ${PARTITION}
${NODE_CPU} ${NODE_GPU} &
```

**A.5.4 Postprocessing Results** Before continuing, the experiments in §A.5.2 and §A.5.3 must be completed. To continue with the postprocessing of the UPMEM PIM system results, i.e., computing metrics such as AUC Score, place the UPMEM PIM system results into the directory `/results`. Next, please run the commands:

```
$ cd postprocessing
$ DATA_ROOT=<path-to-data>
$ PARTITION=<name-of-slurm-partition>
$ NODE=<name-of-slurm-node>
$ ./run_postprocessing_Criteo.sh ${DATA_ROOT} ${PARTITION}
${NODE} &
```

**A.5.5 Reproducing Figures** Please navigate to the directory `/paper_plots`, open the Jupyter Notebook `paper_plots.ipynb`, and select Run All or if you prefer, you can click through the Jupyter Notebook cell by cell. The generated figures can be viewed at `/paper_plots/output` in pdf and png format.

## A.6 Evaluation and expected results

Running the experiments described in (§A.5) is sufficient to reproduce all of our results (Fig. 2, Fig. 4, Fig. 5, Fig. 6, Fig. 7, Fig. 8, Fig. 9, Fig. 10, Fig. 11, Fig. 12, and Fig. 13).