Evaluating Machine Learning Workloads on Memory-Centric Computing Systems

Juan Gómez-Luna[1] Yuxin Guo[1] Sylvan Brocard[2] Julien Legriel[2]
Remy Cimadomo[2] Geraldo F. Oliveira[1] Gagandeep Singh[1] Onur Mutlu[1]

[1]ETH Zürich [2]UPMEM

ABSTRACT

Training machine learning (ML) algorithms is a computationally intensive process, which is frequently memory-bound due to repeatedly accessing large training datasets. As a result, processor-centric systems (CPU, GPU) waste large amounts of energy and execution cycles due to the data movement between memory units and processing units. Memory-centric computing systems, i.e., systems with processing-in-memory (PIM) capabilities, can alleviate this data movement bottleneck.

Our goal is to understand the potential of general-purpose PIM architectures to accelerate ML training. To do so, we (1) implement several classic ML algorithms (namely, linear regression, logistic regression, decision tree, K-Means clustering) on a real-world general-purpose PIM architecture, (2) evaluate and characterize them in terms of accuracy, performance and scaling, and (3) compare to their counterpart state-of-the-art implementations on CPU and GPU. Our evaluation on a real memory-centric computing system with more than 2500 PIM cores shows that PIM greatly accelerates memory-bound ML workloads, when the necessary operations and datatypes are natively supported by PIM hardware. For example, our PIM implementation of decision tree is 27× faster than the CPU implementation on an 8-core Intel Xeon, and 1.34× faster than the GPU implementation on an NVIDIA A100. Our PIM implementation of K-Means clustering is 2.8× and 3.2× faster than CPU and GPU implementations, respectively. We provide several key observations, takeaways, and recommendations for users of ML workloads, programmers of PIM architectures, and hardware designers and architects of future memory-centric computing systems. We open-source all our code and datasets at https://github.com/CMU-SAFARI/pim-ml.

KEYWORDS

machine learning, processing-in-memory, regression, classification, clustering, benchmarking, memory bottleneck

1 INTRODUCTION

Machine learning (ML) algorithms [1–6] have become ubiquitous in many fields of science and technology due to their ability to learn from and improve with experience with minimal human intervention. These algorithms train by updating their model parameters in an iterative manner to improve the overall prediction accuracy. However, training ML algorithms is a computationally intensive process, which requires large amounts of training data [7–9]. Accessing training data in current processor-centric systems (e.g., CPU, GPU) requires costly data movement between memory and processors, which results in high energy consumption and a large percentage of the total execution cycles. This data movement can become the bottleneck of the training process, if there is not enough computation and locality to amortize its cost [10–15].

One way to alleviate the cost of data movement is *processing-in-memory* (*PIM*) [16–20], a data-centric computing paradigm that places processing elements near or inside the memory arrays [7, 18, 21–154]. Even though PIM was first proposed in the 1960s [21, 22], real-world PIM systems have only recently been manufactured [155–165]. The UPMEM PIM architecture [155–160] is the first PIM architecture to become commercially available.

Our **goal** in this work is to quantify the potential of general-purpose real-world PIM architectures for training of ML algorithms. To this end, we implement four representative classic machine learning algorithms (linear regression [166, 167], logistic regression [166, 168], decision tree [169], K-Means [170]) on a memory-centric system containing PIM-enabled memory, specifically the UPMEM PIM architecture [156–160].[1] Our PIM implementations of ML algorithms follow PIM programming recommendations in recent literature [157–159, 177]. We apply several optimizations to overcome the limitations of existing general-purpose PIM architectures (e.g., limited instruction set, relatively simple pipeline, relatively low frequency) and take full advantage of the inherent strengths of PIM (e.g., large memory bandwidth, low memory latency).

We evaluate our PIM implementations in terms of training accuracy, performance, and scaling characteristics on a real memory-centric system with PIM-enabled memory [156, 177, 178]. The system features 2,524 PIM cores running at 425 MHz, and 158 GB of DRAM memory. Our experimental real system evaluation provides new observations and insights, including the following:

- ML training workloads that show memory-bound behavior in processor-centric systems can greatly benefit from (1) fixed-point data representation, (2) quantization [179, 180], and (3) hybrid precision implementation [164, 181] (without much accuracy loss), in order to alleviate the lack of native support for floating-point and high-precision (i.e., 32- and 64-bit) arithmetic operations in the evaluated PIM system.
- ML training workloads that require complex activation functions (e.g., *sigmoid*) [182] can take advantage of *lookup tables* (*LUTs*) [107, 183, 184], instead of function approximation (e.g., Taylor series) [185], when PIM systems lack native support for those activation functions.

[1]We do *not* include neural networks in our study, since GPUs and TPUs [171] have a solid position as the preferred and highly optimized accelerators for them [98, 171–176] due to their extremely high floating-point performance. The UPMEM PIM architecture (used in this study) currently does not have native support for floating-point operations [155–160].

- Data can be placed and laid out such that accesses of PIM cores to their nearby memory banks are streaming, which enables better utilization of the internal PIM memory bandwidth.
- ML training workloads with large training datasets can greatly benefit from scaling the size of PIM-enabled memory with PIM cores attached to memory banks. Training datasets can remain in memory without being moved to the host processor (e.g., CPU, GPU) in every iteration of the training process. Even if PIM cores need to communicate intermediate results via the host processor, this communication overhead is tolerable.

We compare our PIM implementations of linear regression, logistic regression, decision tree, and K-Means to their state-of-the-art CPU and GPU counterparts. We observe that memory-centric systems with PIM-enabled memory can significantly outperform processor-centric systems for memory-bound ML training workloads, when the operations needed by the ML workloads are natively supported by PIM hardware (or can be replaced by efficient LUT implementations). We open-source all our PIM implementations of ML training workloads, training datasets, and evaluation scripts in our GitHub repository [186].

2 BACKGROUND AND MOTIVATION

2.1 Machine Learning Workloads

Machine learning (ML) [1–6] is a family of algorithms that learns a target function (or *model*) that best maps the input variables to an output variable. ML algorithms build (*train*) a model using the observed data (*training dataset*). The model is then used to make (*infer*) predictions or decisions.

Our **goal** in this study is to analyze how real-world general-purpose PIM architectures can accelerate training of representative ML algorithms, and generate insights and recommendations that are useful to programmers and architecture designers. We select four representative classic machine learning algorithms (linear regression, logistic regression, decision tree, K-Means clustering) from three of the subcategories of ML algorithms (regression, classification, clustering).

We employ the roofline model [187] to quantify the memory boundedness of the CPU versions of the four workloads. Fig. 1 shows the roofline model on an Intel Xeon E3-1225 v6 CPU [188] with Intel Advisor [189]. We observe from Fig. 1 that all of the CPU versions of the four workloads are in the memory-bound area of the roofline model (i.e., the shaded region on the left side of the intersection between the DRAM bandwidth roof and the peak compute performance roof). Hence, we confirm that the four workloads are limited by memory access. As a result, these ML workloads are potentially suitable for PIM.

2.2 Processing-in-Memory

Processing-in-memory (PIM) [7, 18, 21–154] is a computing paradigm that advocates for memory-centric computing systems, where processing elements (general-purpose cores and/or accelerators) are placed near or inside the memory arrays. PIM is a feasible solution to alleviate the *data movement bottleneck* [16–20], caused by (1) the need for moving data between memory units and compute units in processor-centric systems, which causes a huge performance loss

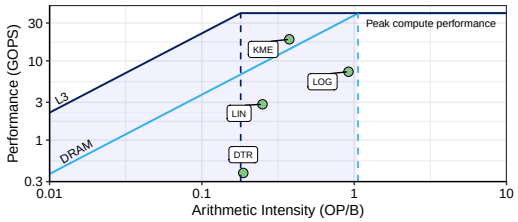

Figure 1: Roofline model for the CPU versions of four ML workloads (LIN: linear regression, LOG: logistic regression, DTR: decision tree, KME: K-Means clustering) on an Intel Xeon E3-1225 v6 CPU.

and energy waste, and worsened by (2) the increasing performance disparity between fast processor units and slow memory units.

Real-world PIM architectures are finally becoming a reality, with the commercialization of the UPMEM PIM architecture [156–158], and the announcement of Samsung HBM-PIM [161, 162], Samsung AxDIMM [163], SK Hynix AiM [164], and Alibaba HB-PNM [165]. These five real-world PIM systems have some important common characteristics, as depicted in Fig. 2. First, there is a host processor (CPU or GPU), typically with a deep cache hierarchy, which has access to (1) standard main memory, and (2) PIM-enabled memory. Second, the PIM-enabled memory chip contains multiple PIM processing elements (PIM PEs), which have access to memory (either memory banks or ranks) with higher bandwidth and lower latency than the host processor. Third, the PIM processing elements (either general-purpose cores, SIMD units, FPGAs, or specialized processors) run at only a few hundred megahertz, and have a small number of registers and relatively small (or no) cache or scratchpad memory. Fourth, PIM PEs may not be able to communicate directly with each other (e.g., UPMEM DPUs, HBM-PIM PCUs or AiM PUs in different chips), and communication between them happens via the host processor.

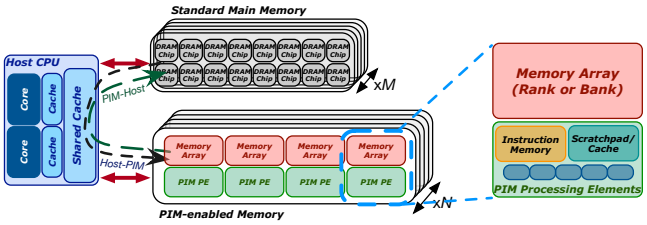

Figure 2: High-level view of a state-of-the-art processing-in-memory system. The host CPU has access to M standard memory modules and N PIM-enabled memory modules.

In our study, we use the UPMEM PIM architecture [155–159, 177, 178, 190]. This PIM architecture uses 2D DRAM arrays and combines them with general-purpose cores, called *DPUs*, on the same chip. In the current architecture generation (as of April 2023), there are 8 DPUs and 8 DRAM banks per chip, and 16 chips per DIMM (8 chips/rank). DPUs are relatively deeply pipelined and fine-grained multithreaded [191–193]. DPUs run software threads, called *tasklets*.

DPUs have a 32-bit RISC-style general-purpose instruction set [177]. They feature native support for 32-bit integer addition/-subtraction and 8-bit multiplication, but some complex operations (e.g., 32-bit integer multiplication/division) and floating-point operations are emulated [158, 159].

Each DPU has access to its own (1) 64-MB DRAM bank, called *MRAM*, (2) 24-KB instruction memory, and (3) 64-KB scratchpad memory, called *WRAM*. The host CPU can access the MRAM banks for copying input data (from main memory to MRAM, i.e., CPU-DPU) and retrieving results (from MRAM to main memory, i.e., DPU-CPU). Since there is no direct communication channel between DPUs, all inter-DPU communication takes place through the host CPU by using DPU-CPU and CPU-DPU data transfers.

Throughout this paper, we use generic terminology, since our implementation strategies are applicable to PIM systems like the generic one described in Fig. 2, and not exclusive of the UPMEM PIM architecture. Thus, we use the terms *PIM core*, *PIM thread*, *DRAM bank*, *scratchpad*, and *CPU-PIM/PIM-CPU transfer*, which correspond to DPU, tasklet, MRAM bank, WRAM, and CPU-DPU/DPU-CPU transfer in UPMEM's terminology [177].

3 ML TRAINING AND PIM IMPLEMENTATION

We select four widely-used machine learning workloads (linear regression, logistic regression, K-Means, and decision tree) as representative ones for our analysis of machine learning training on real-world PIM architectures. We consider them representative because they are diverse in terms of learning approach and application. They have also diverse computational characteristics (e.g., computation pattern, synchronization needs), as Table 1 shows.

Table 1: Machine learning workloads

Characteristic		Linear Regression	Logistic Regression	Decision Tree	K-Means
Short name		LIN	LOG	DTR	KME
Learning approach		Supervised			Unsupervised
Application		Regression	Classification		Clustering
Memory access pattern	Sequential	Yes	Yes	Yes	Yes
	Strided	No	No	No	No
	Random	No	No	No	No
Computation pattern	Operations	mul, add	mul, add, exp, div	compare, add	mul, compare, add
	Datatypes	float, int32_t	float, int32_t	float	int16_t, int64_t
Communication/ synchronization	Intra PIM Core	barrier	barrier	barrier, mutex	barrier, mutex
	Inter PIM Core	Yes	Yes	Yes	Yes

We do *not* include any neural network (or deep learning algorithm) or reinforcement learning (RL) algorithm in our study for two main reasons. First, training of neural networks (e.g., CNN, RNN, GAN) can generally benefit from large caches and register files in processor-centric computing systems, since they expose high temporal locality [7]. Together with their inherent data-level parallelism and very high floating-point operation intensity, they are a good fit for GPUs [173]. In fact, the state-of-the-art ML-targeted PIM architecture [161, 162] shows performance improvements for neural network inference (not training) and with small batch sizes. Second, RL [194] is an inherently sequential process, where an *agent* learns to make decisions by receiving a *reward* at timestep $t+1$ for an *action* that was performed at timestep t on an *environment*. As a result, RL does not appear as a natural fit for PIM systems with many parallel processing elements, such as the one depicted in Figure 2. For deep RL, the state-of-the-art approaches [195] accelerate only the neural network training part in RL (neural network training is out of the scope of our work as explained above).

3.1 Linear Regression

Linear regression [166, 167] is a supervised learning algorithm where the predicted output variable has a linear relation with the input variable.

Algorithm Description. Linear regression obtains a linear model that predicts an output vector y from an input matrix X based on some coefficients or *weights*, vector w. We implement linear regression with *gradient descent* [196], as the optimization algorithm to find the minimum of the loss function. During training, we repeatedly refine the values of w based on the observed values y for the inputs in matrix X (row vectors x_i). In each iteration, we first calculate the predicted output for each row vector x_i, i.e., the dot product of x_i and w. Second, we calculate the gradient for the predicted output, i.e., the error of the predicted output with respect to the observed value y. Third, we update the weights w using the calculated gradient. We repeat the above process until convergence (i.e., the gradient of loss function is zero or close to zero).

PIM Implementation. Our PIM implementation of linear regression with gradient descent divides the training dataset (X) so that each PIM core is assigned an equal number of row vectors x_i. If the training dataset resides initially in the main memory of the host processor, we need to transfer the corresponding partitions of the training dataset to the local memories (e.g., DRAM banks) of the PIM cores. Inside a PIM core, we first distribute the assigned row vectors x_i across the running threads, which compute the dot products of row vectors and weights ($x_i \cdot w$). Second, each dot product result is compared to the observed value y to compute a partial gradient value. Third, we reduce partial gradient values, and return the results to the host. Finally, the host (1) performs final reductions, (2) updates the weights w, and (3) redistributes them to the PIM cores for the next training iteration.

We implement four different versions of linear regression with different input datatypes and optimizations:

- `LIN-FP32` trains with input datasets of 32-bit real values.
- `LIN-INT32` uses 32-bit fixed-point representation of input datasets. It uses 32-bit integer arithmetic.
- `LIN-HYB` is applicable to input datasets of limited value range that fit in 8 bits. The dot product result is 16-bit width, and the final gradient is represented in 32 bits. This hybrid implementation is motivated by the fact that real-world PIM cores only feature arithmetic units of limited precision. For example, UPMEM DPUs [177] run native 8-bit integer multiplication, but emulate 32-bit integer multiplication using *shift-and-add* instructions [158]. HBM-PIM [161] and AiM [164] have only 16-bit floating-point units.
- `LIN-BUI` replaces compiler-generated 16-bit and 32-bit multiplications with custom multiplications based on 8-bit built-in multiplication functions [178] (this optimization is specific to UPMEM PIM). This optimization, which is based on the assumption that input data is encoded in 8 bits, reduces the number of instructions

for each multiplication from 7 instructions (compiler-generated) to 4 (custom).

In §5, we evaluate all LIN versions in terms of accuracy (§5.1), performance for different numbers of threads per PIM core (§5.2), and performance scaling characteristics (§5.3). We also compare our LIN versions to custom CPU and GPU implementations of linear regression (§5.4), which use Intel MKL [197] and NVIDIA cuBLAS [198], respectively.

3.2 Logistic Regression

Logistic regression [166, 168] is a supervised learning algorithm used for classification, which outputs probability values for each input observation variable or vector. These values represent the likelihood of belonging to a certain class or event.

Algorithm Description. Logistic regression uses the *sigmoid* function to map predicted values (output vector y obtained from an input matrix X and a weights vector w) to probabilities. Our implementation of logistic regression uses gradient descent, same as our linear regression (§3.1). We implement the logistic regression algorithm in four steps. First, in the beginning of each training iteration, we obtain the dot product of row vectors x_i and weights w. Second, we apply the sigmoid function to the dot product results. Third, we calculate the gradient to evaluate the error of the predicted probability. Fourth, we update the weights w according to the gradients.

PIM Implementation. Our PIM implementation of logistic regression follows the same workload distribution pattern as our linear regression implementation. First, row vectors x_i are distributed across PIM cores and threads in each PIM core. Second, each thread computes the dot product of a row vector and the weights ($x_i \cdot w$), and applies the sigmoid function to the dot product result. Third, the thread computes partial gradient values. Fourth, partial gradient values from different threads are reduced, and the results are returned to the host. Finally, the host computes the final reductions, and updates the weights before redistributing them to the PIM cores.

We implement six different versions of logistic regression with different input datatypes and optimizations:

- LOG-FP32 trains with input datasets of real data (32-bit precision). If the PIM architecture does *not* support exponentiation (needed for sigmoid), this operation can be approximated by Taylor series [185]. This is true for the UPMEM PIM architecture.
- LOG-INT32 uses 32-bit fixed-point representation of input datasets. It uses 32-bit integer arithmetic, and Taylor series for the sigmoid function.
- LOG-INT32-LUT versions use a LUT per PIM core for sigmoid values, instead of Taylor series. The size of the LUT depends on the sigmoid boundary and the number of bits for the decimal part of the fixed-point representation. We take advantage of the fact that the sigmoid function is symmetric. Thus, for a sigmoid boundary of 20 and 10 bits for the decimal part, the size of the LUT is 20×1024 entries. To represent this range of values, we can fit the entries in 16 bits. As a result, the size of our LUT is 40 KB. This small size can comfortably reside in the small scratchpads/caches of PIM cores (e.g., 64-KB WRAM in the UPMEM PIM architecture). In §5.2, we compare a version that accesses the LUT directly from DRAM (e.g., MRAM in the UPMEM PIM architecture), called LOG-INT32-LUT (MRAM), and a version that accesses the LUT from the scratchpad, called LOG-INT32-LUT (WRAM).
- LOG-HYB-LUT is applicable to input datasets of a limited value range represented in 8 bits, same as LIN-HYB, and uses LUT-based sigmoid (LUT in scratchpad).
- LOG-BUI-LUT uses 8-bit built-in multiplication, same as LIN-BUI, and LUT-based sigmoid (LUT in scratchpad).

In §5, we evaluate all LOG versions in terms of training error rate (§5.1), performance for different numbers of threads per PIM core (§5.2), and performance scaling characteristics (§5.3). We also compare our LOG versions to custom CPU and GPU implementations of logistic regression (§5.4), which use Intel MKL [197] and NVIDIA cuBLAS [198], respectively.

3.3 Decision Tree

Decision trees [169] are tree-based methods for classification and regression. A decision tree partitions the feature space into *leaves*, with a simple prediction model in each leaf, typically a comparison to a threshold (e.g., an average value in regression, a majority class in classification).

Algorithm Description. The training process of a decision tree builds a binary-search tree, which represents the partitioning of the feature space. Each tree node splits the current rectangular subspace further based on a feature and a threshold. The prediction is later done by following the correct path in the tree, up to a leaf which contains the predicted value.

Two main steps of decision tree algorithms are:

(1) Split a tree leaf, thus creating two children connected to their parent node (i.e., the *old* leaf). A split is represented as a tuple $(l, f, thresh)$, where l is the tree leaf index, f is the feature index, and $thresh$ is the feature threshold. After a split, the left child contains the points p of the training set for which $p[f] <= thresh$, and the right child contains the points for which $p[f] > thresh$.
(2) Evaluate the quality of a leaf split. The quality of a split is measured with a score, e.g., the *Gini impurity* [169], a probability measure of a randomly chosen element being incorrectly labeled if it was randomly labeled.

PIM Implementation. Our PIM implementation of a decision tree partitions the training set into subsets of equal size, which the host processor transfers to the PIM cores. The host processor maintains the tree representation and makes splitting decisions, while the PIM cores compute partial Gini scores to evaluate the splits. The partial Gini scores computed by PIM cores are returned to the host and aggregated, in order to make splitting decisions based on the total Gini score.

The host maintains an active frontier of nodes, i.e., the current leaves of the tree. In each training iteration, the host decides whether (1) to split a leaf, an operation called *split commit*, or (2) to evaluate a split, an operation called *split evaluate*, or (3) to query the minimum and maximum values of a feature in a leaf, an operation called *min-max*. The minimum value (*min*) and the maximum value (*max*) are needed by the host to randomly select a candidate split threshold in the [*min*, *max*] interval. Then, the host sends commands (i.e., split commit, split evaluate, min-max) to the PIM cores. The host can send multiple commands at once (with the only

restriction that there must be at most one command per leaf), thus exploiting task-level parallelism in the PIM cores.

Inside a PIM core, a split evaluate command is also parallelized, as different PIM threads work on different batches of feature values. PIM threads move batches of feature values in the training datasets from the DRAM bank to the scratchpad (i.e., from MRAM to WRAM in UPMEM DPUs), compare them to the corresponding threshold, and update the partial Gini score accordingly. This operation has low arithmetic intensity, since only one floating-point comparison and one integer addition are needed. Consequently, a key point for performance is to load and handle multiple feature values at once, in order to hide the latency of accesses to DRAM banks (e.g., in UPMEM DPUs, the MRAM-WRAM transfers are handled by a DMA engine with a deterministic cost for each transfer [158]). Streaming memory accesses (using large MRAM-WRAM transfers) sustain higher memory bandwidth than fine-grained strided/random accesses (using short MRAM-WRAM transfers) [158]. In order to access memory in streaming during split evaluate operations, we lay out the training data in split commit operations as follows:

(1) Points are stored by features. If we denote $p_i[f]$ the value of feature f of point p_i, the first feature values are $p_0[0]p_1[0]...p_n[0]$, then $p_0[1]p_1[1]...p_n[1]$, etc.
(2) For all features, the feature values of points belonging to the same tree leaf are kept consecutive in memory. This means that for a leaf node l containing the subset of points $p_0^l, p_1^l, ..., p_k^l$, and a feature f, the values of $p_0^l[f]p_1^l[f]...p_k^l[f]$ are stored consecutively in memory. The same applies to the class values.

In §5, we evaluate DTR in terms of training accuracy (§5.1), performance for different numbers of threads per PIM core (§5.2), and performance scaling characteristics (§5.3). We also compare our DTR implementation to state-of-the-art CPU and GPU implementations of decision tree (§5.4). The CPU version is from Scikit-learn [199] and the GPU version is from RAPIDS [200].

3.4 K-Means Clustering

K-Means [170] is an iterative clustering method used to find groups, which have not been explicitly labeled, in a dataset.

Algorithm Description. A K-Means algorithm attempts to partition the dataset into K pre-defined distinct non-overlapping subgroups (*clusters*) where each data point belongs to only one group. Points within a cluster are meant be as similar (close) as possible while in comparison to points belonging to other clusters, their differences (*distance*) should be maximized. A cluster is identified by its *centroid*, a point with coordinates determined as the minimum total distance between itself and each point of the cluster. Our K-Means algorithm follows Lloyd's method [170].

PIM Implementation. Our PIM implementation of K-Means partitions the training set and distributes it evenly over the PIM cores. The host processor sets initial random values of the centroids and broadcasts them to all PIM cores. In successive iterations, (1) each PIM core assigns points of its part of the training set to the clusters, and then (2) the host adjusts the centroids based on the new assignment of points.

First, inside a PIM core, PIM threads evaluate which centroid is the nearest one to each point of the training set. Distance calculations are done using 16-bit integer arithmetic. Input data is quantized over a range of ±32767 (16-bit signed integers) to avoid overflowing when doing summations. Second, after finding the nearest centroid to a point, a PIM thread increments a counter and updates one accumulator per coordinate. The counter and the accumulators are associated to the corresponding cluster. Each per-coordinate accumulator contains the sum of values of the corresponding coordinate for all points belonging to a cluster. After all points are processed, each PIM core has partial sums of the coordinate values of the points in each cluster, and the number of points in each cluster. Third, the host processor then retrieves all per-cluster partial sums and counts from all PIM cores, and reduces them in order to compute the new coordinates of the centroids (calculated as the total sum of each coordinate divided by the total count). If these new centroid coordinates are far enough from the previous ones, they are sent over to the PIM cores for another iteration. The process continues until a centroid's coordinates converge to a local optimum, i.e., when the updated coordinates are within a threshold distance to the previous coordinates. The threshold distance used to check for convergence is the *Frobenius norm* [201]. Fourth, once a clustering is completed, the PIM cores compute the *inertia* (also known as *within-cluster sum-of-squares*) of the clustering for their assigned points, and the host processor sums them up. The entire K-Means algorithm is repeated with different random starting centroids. The host processor chooses the clustering with the lowest inertia as the final result.

In §5, we evaluate KME in terms of training quality (§5.1), performance for different numbers of threads per PIM core (§5.2), and performance scaling characteristics (§5.3). We also compare our KME implementation to state-of-the-art CPU and GPU implementations of K-Means (§5.4). The CPU version is from Scikit-learn [199] and the GPU version is from RAPIDS [200].

4 METHODOLOGY

We make our implementations of ML workloads for a real-world PIM system compatible with Scikit-learn [199], an open-source machine learning library, by deploying them as Scikit-learn estimator objects.

We run our experiments on a real-world PIM system [156] with 2,524 PIM cores running at 425 MHz, and 158 GB of DRAM memory. Table 2 shows the main characteristics of this PIM system. The table also includes characteristics of the CPU and the GPU that we use as baselines for comparison. We compare our PIM implementations of ML workloads to state-of-the-art CPU and GPU implementations of the same workloads in terms of performance and quality (§5.4). For linear and logistic regression, we implement CPU versions with Intel MKL [197] and GPU versions with NVIDIA cuBLAS [198]. For decision tree and K-Means, CPU versions are from Scikit-learn [199] and GPU versions from RAPIDS [200].

Table 3 presents the datasets that we use in different experiments. For analysis of PIM kernel performance and performance scaling (both weak and strong scaling) experiments (§5.2 and §5.3), we use synthetic datasets, since we can generate them as large as needed for the scaling experiments. For comparison to CPU and GPU (§5.4), we use state-of-the-art real datasets.

Table 2: Evaluated PIM system, baseline CPU and GPU

Metric		UPMEM PIM System [156]	Intel Xeon Silver 4215 CPU [202]	NVIDIA A100 GPU [203]
Processor Node		2x nm	14 nm	7 nm
Processor	Total Cores	2,524	8 (16 threads)	108 (6,912 SIMD lanes)
	Frequency	425 MHz	2.5 GHz	1.4 GHz
	Peak Performance	1,088 GOPS	40 GFLOPS⋆	19,500 GFLOPS
Main Memory	Capacity	158 GB	256 GB	40 GB
	Total Bandwidth	2145 GB/s	37.5 GB/s	1555 GB/s
TDP		280 W†	85 W	250 W

†$Estimated\ TDP = \frac{Total\ PIM\ cores}{PIM\ cores/DIMM} \times 14\ W/DIMM$ [156].
⋆$Estimated\ GFLOPS = 2.5\ GHz \times 8\ cores \times 2\ instructions\ per\ cycle$.

Table 3: Datasets

Datasets			Linear regression	Logistic regression	Decision tree	K-Means
Synthetic†	Strong Scaling	1 PIM Core	2^{11}; 16 (0.125 MB)	2^{11}; 16 (0.125 MB)	60 K; 16 (3.84 MB)	10 K; 16 (0.64 MB)
		256–2048 PIM Cores	6,291,456; 16 (384 MB)	6,291,456; 16 (384 MB)	153,600,000; 16 (9830 MB)	25,600,000; 16 (1640 MB)
	Weak Scaling	per PIM Core	2^{11}; 16 (0.125 MB)	2^{11}; 16 (0.125 MB)	2^{11}; 16 (38.4 MB)	10 K; 16 (6.4 MB)
Real			SUSY [204, 205]	Skin segmentation [206]	Higgs boson [204, 207]	Higgs boson [204, 207]

†$Format = Samples\ (dataset\ elements);\ Attributes\ (Size\ in\ MB)$.

4.1 ML Training Quality Metrics

We evaluate the training quality of the different versions of our ML workloads. We use synthetic datasets (with uniformly distributed random samples) and run the experiments on a single PIM core (i.e., an UPMEM DPU).

For LIN and LOG, the synthetic datasets contain samples (i.e., dataset elements, each with a number of attributes) with 4 decimal numbers, represented as 32-bit floating-point values. The fixed-point versions use the same datasets after quantization. The number of samples is 8,192 and the number of attributes is 16. We calculate the *training error rate* (lower is better) as the percentage of inference errors of a model for the same data the model was trained on.

For DTR, the synthetic dataset has 32-bit floating-point values. The data is *not* quantized. The number of samples is 600,000 and the number of attributes is 16. There are 4 informative attributes, 4 redundant attributes (a random linear combination of the informative attributes), and 8 random attributes. We evaluate the *training accuracy* (closer to 1 is better) on the same data the model was trained on.

For KME, the synthetic dataset has 32-bit floating-point values. The PIM version uses the same dataset after quantization. The number of samples is 100,000 and the number of attributes is 16. Because it is an unsupervised problem, we do *not* use accuracy as a metric. Instead, we use the *Calinski-Harabasz score* [208] to measure the absolute quality of the clustering with no knowledge of the ground truth used in the generation of the dataset. We also measure the similarity of the clusterings produced by the Scikit-learn version of the algorithm with the PIM implementation using the *adjusted Rand index* [209].

5 EVALUATION

5.1 ML Training Quality

5.1.1 Linear Regression (LIN). We evaluate the training error rate of our four versions of LIN for varying number of training iterations between 1 and 1000. We observe that the training error rate flattens after 500 iterations for the four versions. LIN-FP32 achieves a training error rate as low as 0.55% (same as the CPU version). This is the comparison point for the integer versions (i.e., LIN-INT32, LIN-HYB, LIN-BUI). The training error rate of the integer versions remains low (1.02% for LIN-INT32 and 1.29% for LIN-HYB and LIN-BUI) and close to that of the 32-bit floating-point version. LIN-HYB and LIN-BUI show the same behavior, since they use the same datatypes.

5.1.2 Logistic Regression (LOG). We evaluate the training error rate of our six versions of LOG for numbers of training iterations between 1 and 1000. The training error of LOG-FP32, which we use as the comparison point for the integer versions (i.e., LOG-INT32, LOG-INT32-LUT (MRAM), LOG-INT32-LUT (WRAM), LOG-HYB-LUT (WRAM), LOG-BUI-LUT (WRAM)), is almost flat after 100 iterations, and is as low as 1.20% after 1000 iterations (same as the CPU version). We observe that the training error rate of LOG-INT32 (2.42%) is higher than that of LOG-INT32-LUT (MRAM) and LOG-INT32-LUT (WRAM) (2.14%) The reason is that LOG-INT32 approximates exponentiation (hence, sigmoid) with Taylor series, while LOG-INT32-LUT (MRAM) and LOG-INT32-LUT (WRAM) store exact sigmoid values in a LUT. LOG-HYB-LUT (WRAM) and LOG-BUI-LUT (WRAM) increase the training error rate significantly (14.12%) due to the use of reduced-precision datatypes (i.e., 8- and 16-bit integers).

5.1.3 Decision Tree (DTR). We limit the tree depth to 10. The tree is built by splitting leaf nodes until no node can be split. A node cannot be split if (i) it holds fewer than two data points, (ii) it contains only points belonging to the same class, or (iii) its depth exceeds the maximum tree depth. To account for the effect of different synthetic datasets (with randomly generated samples) on both PIM and CPU implementations, we restart the algorithm 10 times, and average the resulting accuracies. We register a training accuracy (closer to 1 is better) of 0.90008 for the PIM implementation, against 0.90175 for the CPU version.

5.1.4 K-Means Clustering (KME). We perform a K-Means clustering with 16 clusters to match the dataset generation. The clustering iterates for a maximum of 300 iterations, or until the relative Frobenius norm between the cluster centers of two consecutive iterations is lower than 0.0001. In practice, the clustering always converges after less than 40 iterations on both the PIM and the CPU implementations. To account for the effect of synthetic datasets (with randomly generated samples), we average the metrics across 10 different runs with different random seeds. We register average Calinski-Harabasz scores [208] of 82200 for both the PIM and the CPU implementations. The adjusted Rand index [209] between the PIM and CPU clusterings is 0.999347 on average, showing that the clusterings are nearly identical despite the quantization.

5.2 Performance Analysis of PIM Kernels

5.2.1 Linear Regression (LIN). Fig. 3 shows the PIM kernel time of our four versions of LIN. The upper plot (Fig. 3(a)) represents the

PIM kernel time of LIN-FP32. The lower plot (Fig. 3(b)) shows the PIM kernel time of the integer versions.

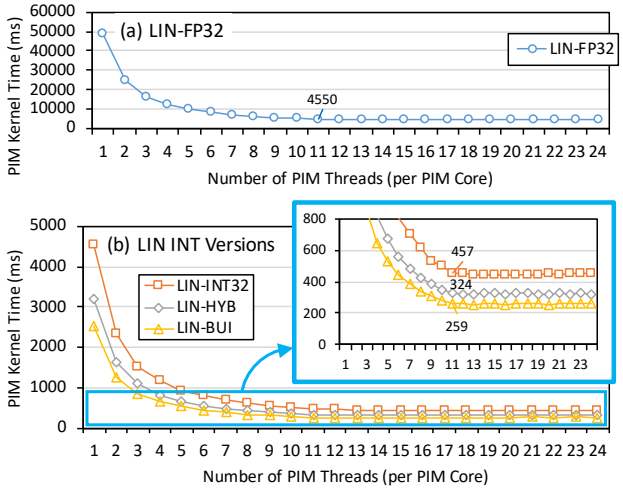

Figure 3: Execution time (ms) of four versions of linear regression using 1-24 PIM threads in 1 PIM core.

We make four observations. First, all LIN versions result in their best performance with 11 or more PIM threads. Eleven is the minimum number of PIM threads that keep the pipeline of the PIM core (i.e., UPMEM DPU) full [155, 158]. For this PIM core, a workload with performance saturation at 11 PIM threads can be considered a compute-bound workload, since the latency of instructions executed in the pipeline hides the latency of memory accesses [158].

Second, using fixed-point representation instead of floating-point (i.e., LIN-INT32 instead of LIN-FP32) reduces the kernel time by an order of magnitude. The PIM cores used in our evaluation do *not* natively support floating-point arithmetic. Thus, floating-point operations are emulated, since the PIM cores only have integer arithmetic units [155, 158].

Key Takeaway 1. *Workloads that require arithmetic operations or datatypes that are not natively supported by PIM cores run at low performance due to instruction emulation (e.g., floating-point operations in UPMEM PIM).*

Recommendation 1. *ML workloads (e.g., LIN, LOG) can employ fixed-point representation if PIM cores do not support floating-point operations (e.g., UPMEM PIM) without sacrificing much accuracy (§5.1).*

Third, LIN-HYB accelerates the PIM kernel by 41% over LIN-INT32. The speedup comes from the use of 8-bit integer multiplication, instead of the emulated 32-bit integer multiplication.

Recommendation 2. *Quantization can be used to take advantage of native hardware support, if PIM cores natively support only limited precision. For example, using hybrid precision after quantizing the training dataset can significantly improve performance.*

Fourth, LIN-BUI achieves an additional 25% speedup over LIN-HYB due to our custom multiplication operation (§3.1).

Recommendation 3. *Programmers (or better compilers) can optimize code at low-level to better leverage available native instructions and hardware (e.g, 8-bit integer multiplication in UPMEM DPUs). Our custom 16- and 32-bit integer multiplications (§3.1) significantly improve performance over compiler-generated code for quantized training datasets.*

5.2.2 Logistic Regression (LOG). Fig. 4 shows the PIM kernel time of our versions of LOG. Fig. 4(a) shows the results for the two versions (LOG-FP32, LOG-INT32) that estimate sigmoid based on Taylor series. Although the 32-bit integer version reduces the kernel time by 65% with respect to the 32-bit floating-point version, which uses emulated floating-point operations, the kernel time of both versions is very high due to the use of Taylor series, which require multiple iterations to achieve the necessary precision. Fig. 4(b) shows the PIM kernel time of the LUT-based versions.

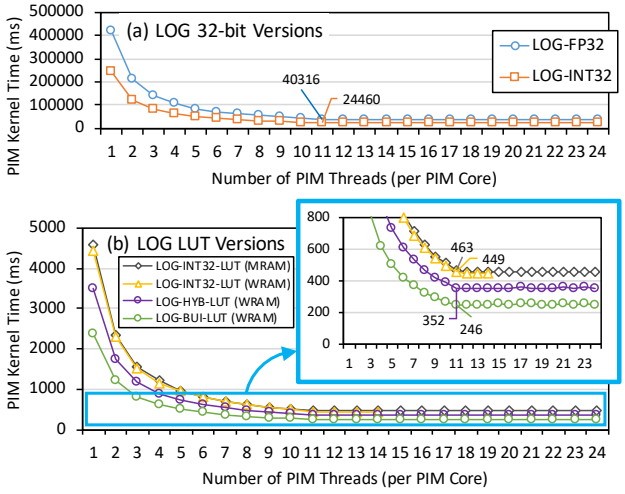

Figure 4: Execution time (ms) of six versions of logistic regression using 1-24 PIM threads in 1 PIM core.

We make five observations. First, the performance of all LOG versions saturates at 11 PIM threads, for the same reason as LIN versions. Second, LOG-INT32-LUT (MRAM) results in a speedup of 53× over LOG-INT32. This demonstrates the benefit of converting computation to memory accesses using LUTs in PIM architectures. Third, there is very little speedup (3%) coming from placing the LUT in the scratchpad (WRAM of UPMEM DPUs). The LUT query is only one memory access and its cost is negligible compared to the rest of computation. Fourth, the use of 8-bit integer multiplication allows LOG-HYB-LUT (WRAM) to outperform LOG-INT32-LUT (WRAM) by 28%. Fifth, the custom multiplication used by LOG-BUI-LUT (WRAM) provides an extra 43% speedup over LOG-HYB-LUT (WRAM).

Recommendation 4. *Programmers can convert computation to memory accesses in PIM architectures by keeping pre-calculated operation results (e.g., LUTs, memoization) in memory.*

5.2.3 Decision Tree (DTR). Fig. 5(a) shows the PIM kernel time of DTR. We make three observations. First, the performance of DTR and KME saturates at 11 PIM threads, for the same reason as LIN versions.

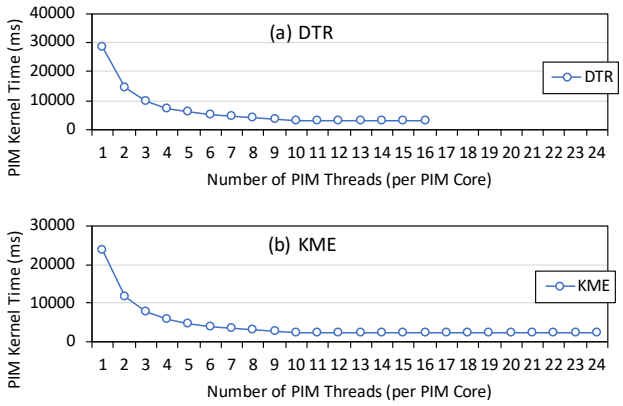

Figure 5: Execution time (ms) of decision tree (a) using 1-16 PIM threads in 1 PIM core and K-Means clustering (b) using 1-24 PIM threads in 1 PIM core.

Second, the optimized data layout of DTR (§3.3) ensures that data is accessed at maximum bandwidth and, thus, the pipeline latency hides the latency of memory accesses.

Recommendation 5. *For data structures of more than one dimension, programmers can optimize the data layout in a way that memory accesses are in streaming, thus exploiting higher sustained bandwidth.*

Third, for DTR, the maximum possible number of PIM threads is 16 due to the usage of the local scratchpad memory in the PIM core. The amount of memory needed by each PIM thread limits the maximum number of PIM threads to 16.

5.2.4 K-Means Clustering (KME). Fig. 5(b) shows the PIM kernel time of KME. The performance of KME saturates at 11 PIM threads, for the same reason as LIN versions.

Key Takeaway 2. *ML training workloads (e.g., linear regression, logistic regression, decision tree, K-Means) that are bound by memory access due to their low arithmetic intensity in processor-centric systems (e.g., CPU, GPU) behave as compute-bound when running on PIM cores.*

Recommendation 6. *Maximize the utilization of PIM cores by keeping their pipeline fully busy. For example, in the UPMEM PIM architecture [155], which has fine-grained multithreaded scalar cores, we recommend to schedule 11 or more PIM threads, which is the minimum number of PIM threads to saturate the pipeline throughput.*

5.3 Performance Scaling

We evaluate performance scaling characteristics of our ML workloads using weak scaling and strong scaling experiments. For weak scaling (§5.3.1), we run experiments on 1 rank (from 1 to 64 PIM cores). Our goal is to evaluate how the performance scales with the number of PIM cores for a fixed problem size per processing element. For strong scaling (§5.3.2), we run experiments on 32 ranks (from 256 to 2,048 PIM cores). Our goal is to evaluate how the performance of our ML workloads scales with the number of PIM cores for a fixed problem size.

5.3.1 Weak Scaling. Fig. 6 shows weak scaling results on 1-64 PIM cores for all versions of our ML workloads. Each bar presents the total execution time broken down into (1) execution time of the PIM kernel (i.e., PIM Kernel), communication time between the host CPU and the PIM cores (i.e., CPU-PIM and PIM-CPU times), and communication time between PIM cores (i.e., Inter PIM Core). We make the following observations from the figure.

First, we observe linear scaling of the PIM kernel time of all LIN versions, all LOG versions, and DTR. However, the PIM kernel time of KME reduces as we increase the number of PIM cores. This is caused by the fact that the K-Means algorithm on average converges with fewer iterations on a larger dataset. The PIM kernel time per iteration does scale linearly.

Second, the fraction of total execution time spent on communication between the host CPU and the PIM cores (i.e., CPU-PIM and PIM-CPU[2] times) and between PIM cores (i.e., Inter PIM Core) is negligible compared to the PIM kernel time for all versions. For all LIN versions, all LOG versions, DTR, and KME, the sum of CPU-PIM, Inter PIM Core, and PIM-CPU times takes less than 7% of the total execution time.

5.3.2 Strong Scaling. Fig. 7 shows strong scaling results on 256-2,048 PIM cores for all versions of our ML workloads. Each bar (left y-axis) presents the total execution time broken down into (1) execution time of the PIM kernel (i.e., PIM Kernel), communication time between the host CPU and the PIM cores (i.e., CPU-PIM and PIM-CPU times), and communication time between PIM cores (i.e., Inter PIM Core). Each red line (right y-axis) represents the speedup of a PIM kernel normalized to the performance of 256 PIM cores. We make the following observations.

First, we observe that the PIM kernel time scales linearly with the number of PIM cores. The speedup of 2,048 PIM cores over 256 PIM cores is between 6.37× and 7.98×.

Second, the overhead of communication between PIM cores (i.e., Inter PIM Core) is tolerable for all ML workloads. The largest fraction of Inter PIM Core over the total execution time is 36% for KME with 2,048 PIM cores. Even so, 2,048 PIM cores provide the lowest total execution time of KME.

Third, the communication time between the host CPU and the PIM cores (i.e., CPU-PIM and PIM-CPU times) represents a negligible fraction of the total execution time of all ML workloads.

Key Takeaway 3. *Memory-bound ML training workloads, which need large training datasets, benefit from large PIM-enabled memory with many PIM cores. Even if PIM cores need to communicate via the host processor (e.g., in UPMEM PIM), the amount of data movement needed for intermediate results is minimal with respect to the size of the whole dataset.*

5.4 Comparison to CPU and GPU

We compare our implementations of ML workloads on a PIM system to CPU and GPU implementations of the same workloads

[2]DTR and KME do *not* need final PIM-CPU transfer. For DTR, the reason is that the tree is built iteratively on the host side, and the algorithm ends when the CPU declares termination on the tree build. For KME, the CPU is in charge of the final cluster assignment once convergence has been declared.

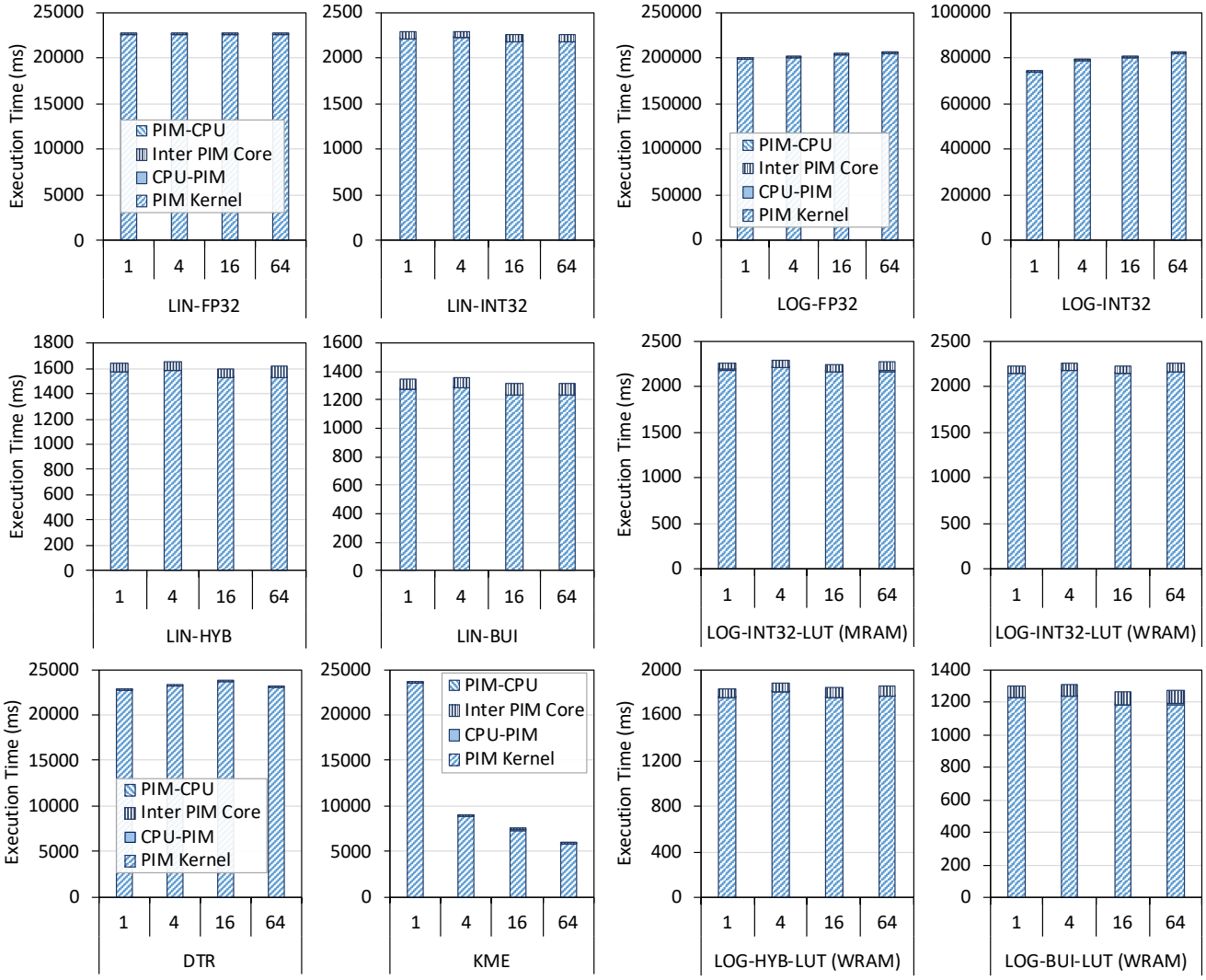

Figure 6: Execution time (ms) of ML workloads on 1, 4, 16, and 64 PIM cores using weak scaling. Inside a PIM core, we use the best-performing number of PIM threads (§5.2).

in terms of performance and quality.[3] For linear and logistic regression, we implement CPU versions with Intel MKL [197] and GPU versions with NVIDIA cuBLAS [198]. For decision tree and K-Means, CPU versions are from Scikit-learn [199] and GPU versions from RAPIDS [200].

For the PIM system performance measurements, we use the best-performing number of PIM cores. Inside a PIM core, we use the best-performing number of PIM threads (§5.2). We include the time spent in the PIM cores ("PIM Kernel"), the time spent for inter-PIM-core synchronization ("Inter PIM"), and the time spent in the initial CPU-PIM and the final PIM-CPU transfers ("CPU-PIM", "PIM-CPU"). For the GPU performance measurements, we include the kernel time ("GPU Kernel"), and the initial CPU-GPU and the final GPU-CPU transfer times ("CPU-GPU", "GPU-CPU"). The results that we show in this section correspond to the best configurations in terms of CPU threads (for the CPU versions), GPU threads per block and thread blocks (for the GPU versions), and PIM cores and PIM threads (for the PIM versions). We open-source all configurations for reproducibility [186].

5.4.1 Linear Regression (LIN). Fig. 8 shows the execution times of `LIN` versions on PIM, CPU, and GPU with the SUSY dataset [205]. We apply symmetric quantization [179, 180] to evaluate our integer versions. We make four observations. First, `LIN-FP32` is heavily burdened by the use of floating-point arithmetic, which is not natively supported by the PIM system we use in our evaluation [158]. Despite that, `LIN-FP32` is 13% faster than the CPU version. Second, `LIN-INT32` is 8.5× faster than `LIN-FP32`. This is the result of using natively supported instructions (even though 32-bit integer multiplication is emulated in the UPMEM PIM architecture [158]). Third,

[3]See Table 2 for a description of our baseline CPU and GPU architectures.

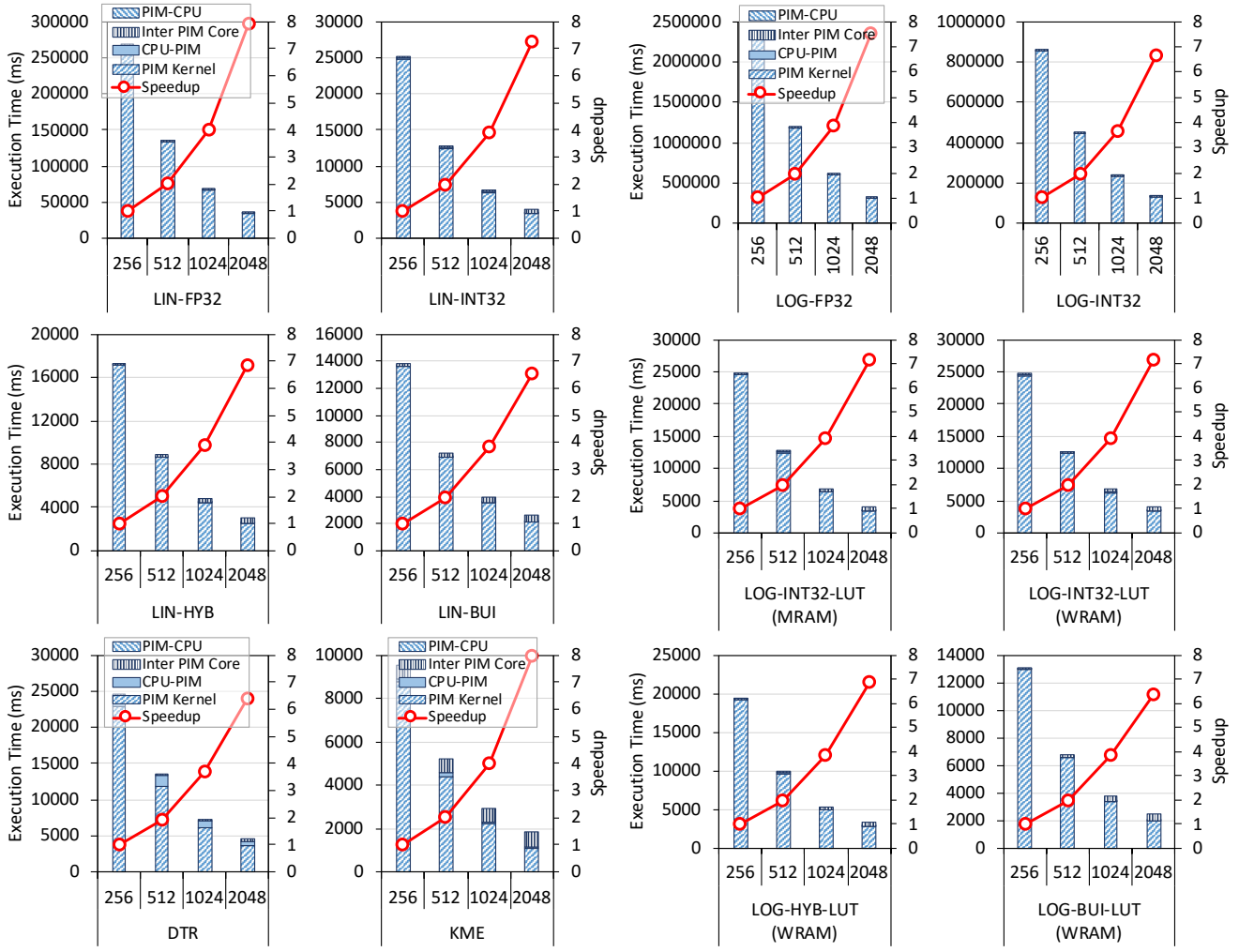

Figure 7: Execution time (ms) of ML workloads on 256, 512, 1,024, and 2,048 PIM cores using strong scaling (left y-axis), and speedup of the PIM kernel normalized to the performance of 256 PIM cores (right y-axis). Inside a PIM core, we use the best-performing number of PIM threads (Section 5.2).

LIN-HYB and LIN-BUI further improve performance. The kernel time of LIN-HYB is 10% lower than that of LIN-INT32 due to the use of hybrid precision. Our custom multiplication in LIN-BUI reduces the kernel time by an additional 4%. Fourth, the GPU version is 4.1× faster than LIN-BUI, since the A100 (1) has much higher compute throughput than the PIM system that we use in our experiments, and (2) its memory bandwidth is only 39% lower than the bandwidth of the PIM system (Table 2).

5.4.2 Logistic Regression (LOG). Fig. 9 shows the execution times of LOG versions on PIM, CPU, and GPU with the Skin segmentation dataset [206]. We make four observations. First, LOG-FP32 and LOG-INT32 PIM versions are almost 10× slower than the CPU version. The reason is the high cost of sigmoid estimation with Taylor series due to their iterative nature (as mentioned in §5.2.2). Second, LOG-INT32 is 17% faster than LOG-FP32 due to the faster integer arithmetic [158]. Third, replacing Taylor series

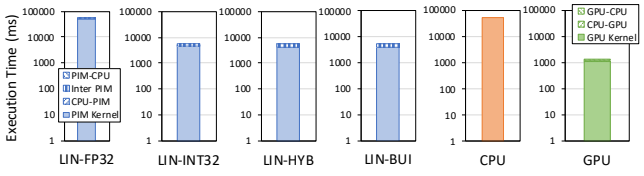

Figure 8: Execution time (ms) of LIN on PIM, CPU, and GPU with the SUSY dataset. For all PIM versions (LIN-*), the best-performing number of PIM cores is 2,524.

with the use of LUTs (in LOG-INT32-LUT (MRAM), LOG-INT32-LUT (WRAM), LOG-HYB-LUT (WRAM), and LOG-BUI-LUT (WRAM)) to compute sigmoid accelerates the PIM versions by almost two orders of magnitude. For example, LOG-INT32-LUT (WRAM) is 3.3× and LOG-BUI-LUT (WRAM) is 3.9× faster than the CPU version. Fourth,

even though the GPU version is significantly faster than all PIM versions (e.g., 16.5× faster than LOG-BUI-LUT (WRAM)), the gap between GPU and PIM is greatly reduced by using appropriate optimizations in PIM codes (e.g., LUTs, custom multiplication).

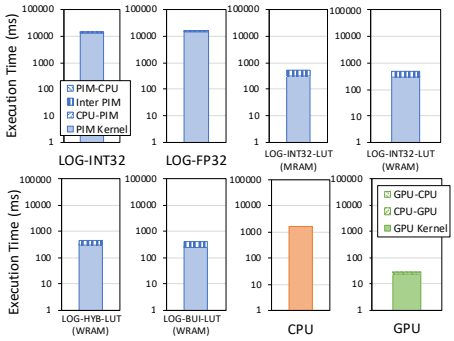

Figure 9: Execution time (ms) of LOG on PIM, CPU, and GPU with the Skin segmentation dataset. For the PIM versions (LOG-*), the best-performing number of PIM cores is 2,524 for LOG-FP32 and LOG-INT32, 320 for LOG-INT32-LUT (MRAM) and LOG-INT32-LUT (WRAM), and 256 for LOG-HYB-LUT (WRAM) and LOG-BUI-LUT (WRAM).

We evaluate the training error rate of all versions of LIN (with the SUSY dataset) and LOG (with the Skin segmentation dataset). We make two observations. First, the training error rates of the floating-point versions (i.e., LIN-FP32, LOG-FP32) is the same as that of the CPU and GPU versions. Second, the training error rates of the PIM versions of LIN and LOG that use quantized datasets are greater than those of the CPU and GPU versions, but they may still be acceptable (i.e., < 20% for LIN and < 9% for LOG) for some applications [210–216].

5.4.3 Decision Tree (DTR). Fig. 10(a) shows the execution times of DTR versions on PIM, CPU, and GPU with the Higgs boson dataset [207]. We make two observations. First, the PIM version of DTR outperforms the CPU version and the GPU version by 27× and 1.34×, respectively. Since DTR mostly uses comparison operations (e.g., comparing a feature value to a threshold), the PIM version can take advantage of the large internal bandwidth of the PIM system without being burdened by other costly arithmetic operations. Second, 70% of the execution time of the GPU version of DTR is spent on moving data between the host CPU and the GPU, while only 27% of the execution time of the PIM version is due to communication between the host CPU and the PIM cores or between PIM cores. The fact that the host CPU and the PIM cores are connected through memory channels is an advantage over the GPU, which uses PCIe bus, as the memory channels provide higher bandwidth.

We evaluate the training accuracy of DTR versions on PIM, CPU, and GPU. We observe that the accuracy of our PIM version (0.65635) is very similar to the accuracy of the CPU version (0.65581), and only slightly smaller than that of the GPU version (0.70462).

5.4.4 K-Means Clustering (KME). Fig. 10(b) shows the execution times of KME versions on PIM, CPU, and GPU with the Higgs boson dataset [207]. We observe that the PIM version of KME is 2.8× faster

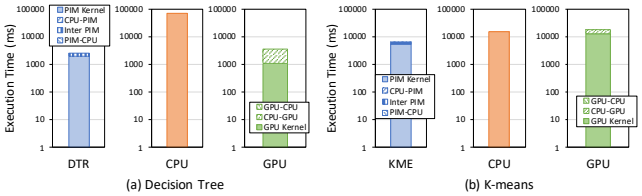

Figure 10: Execution time (ms) of DTR (a) and KME (b) on PIM, CPU, and GPU with the Higgs boson segmentation dataset. For the PIM versions (DTR, KME), the best-performing number of PIM cores is 1,024 for DTR and 2,524 for KME.

than the CPU version and 3.2× faster than the GPU version. Similar to DTR, KME does *not* use costly arithmetic operations but mainly 16-bit integer arithmetic.

We evaluate the similarity of the clusterings (given by the adjusted Rand index) produced by KME versions on PIM, CPU, and GPU. The adjusted Rand index between the PIM version and the CPU version is 0.999985, while the adjusted Rand index between the GPU version and the CPU version is significantly lower (0.758579).

Key Takeaway 4. *Memory-bound ML workloads that require mainly operations natively supported by the PIM architecture (e.g., 32-bit integer addition/subtraction in UPMEM PIM), such as decision tree and K-Means clustering, leverage the large PIM bandwidth, and perform better than their CPU and GPU counterparts.*

6 RELATED WORK

To our knowledge, this is the first work that *comprehensively* evaluates the benefits of a *real* general-purpose processing-in-memory (PIM) system for ML training workloads. We briefly summarize prior works on PIM acceleration of Deep Learning (DL) and other ML algorithms.

PIM for DL inference. Many prior works focus on accelerating DL inference using different PIM solutions. This includes both proposals from academia [47, 73, 98, 136, 150, 151, 217–224] and industry [161–165], targeting various types of DL models, including CNNs [47, 73, 98, 136, 150, 151, 161, 162, 217, 219–222], RNNs [136, 164, 224], and recommendation systems [163, 165, 218, 223]. Our work differs from such works since we focus on classic ML algorithms (i.e., regression, classification, clustering) using a real-world general-purpose PIM architecture (i.e., UPMEM PIM [156]).

PIM for DL training. Other works leverage PIM techniques to accelerate DL training [225–237]. These works mainly utilize the analog computation capabilities (e.g., for matrix vector multiplication) of non-volatile memories (NVMs) to implement training of deep neural networks [225–228, 230, 232, 234, 236]. In contrast, executing DL training using DRAM-based PIM architectures is challenging, since the area and power constraints of such architectures lead to performance bottlenecks when executing key operations (e.g., multiplication) required during training [238].

PIM for other ML algorithms. Few related prior works [80, 239–243] propose solutions for ML algorithms other than DL inference and training. Such works leverage different memory technologies (e.g., 3D-stacked DRAM [80, 239, 242], ReRAM [241], SRAM [240,

243]) to accelerate ML workloads such as linear regression [239–242], logistic regression [239, 241], support vector machines [239], and K-nearest neighbors [240, 243]. None of these works provide comprehensive implementation and evaluation of ML algorithms using a real processing-in-memory architecture.

7 CONCLUSION

Machine learning training frequently becomes memory-bound in processor-centric systems due to repeated accesses to large training datasets. Memory-centric systems (i.e., systems with processing-in-memory (PIM) capabilities) can overcome this memory boundedness.

We implement several representative classic machine learning algorithms on a real-world general-purpose PIM architecture with the aim of understanding the potential of memory-centric systems for ML training. We evaluate our PIM implementations on a memory-centric computing system with more than 2500 PIM cores in terms of accuracy, performance, and scaling characteristics, and compare to state-of-the-art implementations for CPU and GPU.

To our knowledge, our work is the first one to evaluate training of machine learning algorithms on a real-world PIM architecture. We show that PIM systems can greatly outperform CPUs and GPUs for memory-bound ML training workloads when the PIM processing elements have native support for the arithmetic operations and datatypes required by the ML training workloads. Compared to CPUs, PIM systems feature significantly higher memory bandwidth and many more parallel processing elements, the number of which scales with memory capacity. Compared to GPUs, PIM systems benefit from higher host-accelerator bandwidth given that PIM processing elements are connected to the host CPU via memory channels (as opposed to PCIe like GPUs). We believe that our work shows great promise for PIM systems as widely-used accelerators for ML training workloads, and this promise can materialize in future PIM systems with more mature architectures, hardware, and software support.

ACKNOWLEDGMENTS

We thank the anonymous reviewers of ASPLOS 2023 and ISPASS 2023 for feedback. We acknowledge the generous gifts provided by our industrial partners, including ASML, Facebook, Google, Huawei, Intel, Microsoft, and VMware. We acknowledge support from the Semiconductor Research Corporation and the ETH Future Computing Laboratory.

A longer version of this paper is available in arXiv [244]. A much shorter version of this paper appears as an invited paper at the 2022 IEEE Computer Society Annual Symposium on VLSI (ISVLSI) [245].

REFERENCES

[1] A. Géron, *Hands-on Machine Learning With Scikit-Learn, Keras, and TensorFlow: Concepts, tools, and techniques to build intelligent systems*, 2019.
[2] E. Alpaydin, *Introduction to Machine Learning*, 2020.
[3] I. Goodfellow *et al.*, *Deep Learning*, 2016.
[4] M. Mohri *et al.*, *Foundations of Machine Learning*, 2018.
[5] S. Shalev-Shwartz and S. Ben-David, *Understanding Machine Learning: From Theory to Algorithms*, 2014.
[6] S. Raschka and V. Mirjalili, *Python Machine Learning: Machine Learning and Deep Learning with Python, Scikit-Learn, and TensorFlow 2*, 2019.
[7] G. F. Oliveira *et al.*, "DAMOV: A New Methodology and Benchmark Suite for Evaluating Data Movement Bottlenecks," *IEEE Access*, 2021.
[8] M. Wang *et al.*, "A survey on large-scale machine learning," *IEEE Transactions on Knowledge and Data Engineering*, 2020.
[9] C. Dünner *et al.*, "Snap ml: A hierarchical framework for machine learning," *Advances in Neural Information Processing Systems*, vol. 31, 2018.
[10] X. Xie *et al.*, "CuMF_SGD: Parallelized Stochastic Gradient Descent for Matrix Factorization on GPUs," in *HPDC*, 2017.
[11] C. De Sa *et al.*, "Understanding and Optimizing Asynchronous Low-precision Stochastic Gradient Descent," in *ISCA*, 2017.
[12] H. Kim *et al.*, "GradPIM: A Practical Processing-in-DRAM Architecture for Gradient Descent," in *HPCA*, 2021.
[13] D. Mahajan *et al.*, "Tabla: A Unified Template-based Framework for Accelerating Statistical Machine Learning," in *HPCA*, 2016.
[14] B. Peng *et al.*, "HarpGBDT: Optimizing Gradient Boosting Decision Tree for Parallel Efficiency," in *CLUSTER*, 2019.
[15] M. A. Bender *et al.*, "K-Means Clustering on Two-Level Memory Systems," in *MEMSYS*, 2015.
[16] O. Mutlu *et al.*, "Processing Data Where It Makes Sense: Enabling In-Memory Computation," *MicPro*, 2019.
[17] O. Mutlu *et al.*, "A Modern Primer on Processing in Memory," *Emerging Computing: From Devices to Systems - Looking Beyond Moore and Von Neumann*, 2021, https://arxiv.org/pdf/2012.03112.pdf.
[18] S. Ghose *et al.*, "Processing-in-Memory: A Workload-Driven Perspective," *IBM JRD*, 2019.
[19] V. Seshadri and O. Mutlu, "In-DRAM Bulk Bitwise Execution Engine," arXiv:1905.09822 [cs.AR], 2020.
[20] O. Mutlu *et al.*, "Enabling Practical Processing in and near Memory for Data-Intensive Computing," in *DAC*, 2019.
[21] H. S. Stone, "A Logic-in-Memory Computer," *IEEE TC*, 1970.
[22] W. H. Kautz, "Cellular Logic-in-Memory Arrays," *IEEE TC*, 1969.
[23] D. E. Shaw *et al.*, "The NON-VON Database Machine: A Brief Overview," *IEEE Database Eng. Bull.*, 1981.
[24] P. M. Kogge, "EXECUBE - A New Architecture for Scaleable MPPs," in *ICPP*, 1994.
[25] M. Gokhale *et al.*, "Processing in Memory: The Terasys Massively Parallel PIM Array," *IEEE Computer*, 1995.
[26] D. Patterson *et al.*, "A Case for Intelligent RAM," *IEEE Micro*, 1997.
[27] M. Oskin *et al.*, "Active Pages: A Computation Model for Intelligent Memory," in *ISCA*, 1998.
[28] Y. Kang *et al.*, "FlexRAM: Toward an Advanced Intelligent Memory System," in *ICCD*, 1999.
[29] K. Mai *et al.*, "Smart Memories: A Modular Reconfigurable Architecture," in *ISCA*, 2000.
[30] R. C. Murphy *et al.*, "The Characterization of Data Intensive Memory Workloads on Distributed PIM Systems," in *Intelligent Memory Systems*. Springer.
[31] J. Draper *et al.*, "The Architecture of the DIVA Processing-in-Memory Chip," in *SC*, 2002.
[32] S. Aga *et al.*, "Compute Caches," in *HPCA*, 2017.
[33] C. Eckert *et al.*, "Neural Cache: Bit-serial In-cache Acceleration of Deep Neural Networks," in *ISCA*, 2018.
[34] D. Fujiki *et al.*, "Duality Cache for Data Parallel Acceleration," in *ISCA*, 2019.
[35] M. Kang *et al.*, "An Energy-Efficient VLSI Architecture for Pattern Recognition via Deep Embedding of Computation in SRAM," in *ICASSP*, 2014.
[36] V. Seshadri *et al.*, "Ambit: In-Memory Accelerator for Bulk Bitwise Operations Using Commodity DRAM Technology," in *MICRO*, 2017.
[37] V. Seshadri *et al.*, "Buddy-RAM: Improving the Performance and Efficiency of Bulk Bitwise Operations Using DRAM," arXiv:1611.09988 [cs:AR], 2016.
[38] V. Seshadri *et al.*, "Fast Bulk Bitwise AND and OR in DRAM," *CAL*, 2015.
[39] V. Seshadri *et al.*, "RowClone: Fast and Energy-Efficient In-DRAM Bulk Data Copy and Initialization," in *MICRO*, 2013.
[40] S. Angizi and D. Fan, "Graphide: A Graph Processing Accelerator Leveraging In-DRAM-computing," in *GLSVLSI*, 2019.
[41] J. Kim *et al.*, "The DRAM Latency PUF: Quickly Evaluating Physical Unclonable Functions by Exploiting the Latency–Reliability Tradeoff in Modern DRAM Devices," in *HPCA*, 2018.
[42] J. Kim *et al.*, "D-RaNGe: Using Commodity DRAM Devices to Generate True Random Numbers with Low Latency and High Throughput," in *HPCA*, 2019.
[43] F. Gao *et al.*, "ComputeDRAM: In-Memory Compute Using Off-the-Shelf DRAMs," in *MICRO*, 2019.
[44] K. K. Chang *et al.*, "Low-Cost Inter-Linked Subarrays (LISA): Enabling Fast Inter-Subarray Data Movement in DRAM," in *HPCA*, 2016.
[45] X. Xin *et al.*, "ELP2IM: Efficient and Low Power Bitwise Operation Processing in DRAM," in *HPCA*, 2020.
[46] S. Li *et al.*, "DRISA: A DRAM-Based Reconfigurable In-Situ Accelerator," in *MICRO*, 2017.
[47] Q. Deng *et al.*, "DrAcc: A DRAM Based Accelerator for Accurate CNN Inference," in *DAC*, 2018.
[48] N. Hajinazar *et al.*, "SIMDRAM: A Framework for Bit-Serial SIMD Processing Using DRAM," in *ASPLOS*, 2021.

[49] S. H. S. Rezaei *et al.*, "NoM: Network-on-Memory for Inter-Bank Data Transfer in Highly-Banked Memories," *CAL*, 2020.
[50] Y. Wang *et al.*, "FIGARO: Improving System Performance via Fine-Grained In-DRAM Data Relocation and Caching," in *MICRO*, 2020.
[51] M. F. Ali *et al.*, "In-Memory Low-Cost Bit-Serial Addition Using Commodity DRAM Technology," in *TCAS-I*, 2019.
[52] S. Li *et al.*, "Pinatubo: A Processing-in-Memory Architecture for Bulk Bitwise Operations in Emerging Non-Volatile Memories," in *DAC*, 2016.
[53] S. Angizi *et al.*, "PIMA-Logic: A Novel Processing-in-Memory Architecture for Highly Flexible and Energy-efficient Logic Computation," in *DAC*, 2018.
[54] S. Angizi *et al.*, "CMP-PIM: An Energy-efficient Comparator-based Processing-in-Memory Neural Network Accelerator," in *DAC*, 2018.
[55] S. Angizi *et al.*, "AlignS: A Processing-in-Memory Accelerator for DNA Short Read Alignment Leveraging SOT-MRAM," in *DAC*, 2019.
[56] Y. Levy *et al.*, "Logic Operations in Memory Using a Memristive Akers Array," *Microelectronics Journal*, 2014.
[57] S. Kvatinsky *et al.*, "MAGIC—Memristor-Aided Logic," *IEEE TCAS II: Express Briefs*, 2014.
[58] A. Shafiee *et al.*, "ISAAC: A Convolutional Neural Network Accelerator with In-situ Analog Arithmetic in Crossbars," in *ISCA*, 2016.
[59] S. Kvatinsky *et al.*, "Memristor-Based IMPLY Logic Design Procedure," in *ICCD*, 2011.
[60] S. Kvatinsky *et al.*, "Memristor-Based Material Implication (IMPLY) Logic: Design Principles and Methodologies," *TVLSI*, 2014.
[61] P.-E. Gaillardon *et al.*, "The Programmable Logic-in-Memory (PLiM) Computer," in *DATE*, 2016.
[62] D. Bhattacharjee *et al.*, "ReVAMP: ReRAM based VLIW Architecture for In-memory Computing," in *DATE*, 2017.
[63] S. Hamdioui *et al.*, "Memristor Based Computation-in-Memory Architecture for Data-intensive Applications," in *DATE*, 2015.
[64] L. Xie *et al.*, "Fast Boolean Logic Papped on Memristor Crossbar," in *ICCD*, 2015.
[65] S. Hamdioui *et al.*, "Memristor for Computing: Myth or Reality?" in *DATE*, 2017.
[66] J. Yu *et al.*, "Memristive Devices for Computation-in-Memory," in *DATE*, 2018.
[67] C. Giannoula *et al.*, "SynCron: Efficient Synchronization Support for Near-Data-Processing Architectures," in *HPCA*, 2021.
[68] I. Fernandez *et al.*, "NATSA: A Near-Data Processing Accelerator for Time Series Analysis," in *ICCD*, 2020.
[69] D. S. Cali *et al.*, "GenASM: A High-Performance, Low-Power Approximate String Matching Acceleration Framework for Genome Sequence Analysis," in *MICRO*, 2020.
[70] J. S. Kim *et al.*, "GRIM-Filter: Fast Seed Location Filtering in DNA Read Mapping Using Processing-in-Memory Technologies," *BMC Genomics*, 2018.
[71] J. Ahn *et al.*, "PIM-Enabled Instructions: A Low-Overhead, Locality-Aware Processing-in-Memory Architecture," in *ISCA*, 2015.
[72] J. Ahn *et al.*, "A Scalable Processing-in-Memory Accelerator for Parallel Graph Processing," in *ISCA*, 2015.
[73] A. Boroumand *et al.*, "Google Workloads for Consumer Devices: Mitigating Data Movement Bottlenecks," in *ASPLOS*, 2018.
[74] A. Boroumand *et al.*, "CoNDA: Efficient Cache Coherence Support for near-Data Accelerators," in *ISCA*, 2019.
[75] G. Singh *et al.*, "NAPEL: Near-memory Computing Application Performance Prediction via Ensemble Learning," in *DAC*, 2019.
[76] H. Asghari-Moghaddam *et al.*, "Chameleon: Versatile and Practical Near-DRAM Acceleration Architecture for Large Memory Systems," in *MICRO*, 2016.
[77] O. O. Babarinsa and S. Idreos, "JAFAR: Near-Data Processing for Databases," in *SIGMOD*, 2015.
[78] P. Chi *et al.*, "PRIME: A Novel Processing-In-Memory Architecture for Neural Network Computation In ReRAM-Based Main Memory," in *ISCA*, 2016.
[79] A. Farmahini-Farahani *et al.*, "NDA: Near-DRAM acceleration architecture leveraging commodity DRAM devices and standard memory modules," in *HPCA*, 2015.
[80] M. Gao *et al.*, "Practical Near-Data Processing for In-Memory Analytics Frameworks," in *PACT*, 2015.
[81] M. Gao and C. Kozyrakis, "HRL: Efficient and Flexible Reconfigurable Logic for Near-Data Processing," in *HPCA*, 2016.
[82] B. Gu *et al.*, "Biscuit: A Framework for Near-Data Processing of Big Data Workloads," in *ISCA*, 2016.
[83] Q. Guo *et al.*, "3D-Stacked Memory-Side Acceleration: Accelerator and System Design," in *WoNDP*, 2014.
[84] M. Hashemi *et al.*, "Accelerating Dependent Cache Misses with an Enhanced Memory Controller," in *ISCA*, 2016.
[85] M. Hashemi *et al.*, "Continuous Runahead: Transparent Hardware Acceleration for Memory Intensive Workloads," in *MICRO*, 2016.
[86] K. Hsieh *et al.*, "Transparent Offloading and Mapping (TOM): Enabling Programmer-Transparent Near-Data Processing in GPU Systems," in *ISCA*, 2016.
[87] D. Kim *et al.*, "Neurocube: A Programmable Digital Neuromorphic Architecture with High-Density 3D Memory," in *ISCA*, 2016.
[88] G. Kim *et al.*, "Toward Standardized Near-Data Processing with Unrestricted Data Placement for GPUs," in *SC*, 2017.
[89] J. H. Lee *et al.*, "BSSync: Processing Near Memory for Machine Learning Workloads with Bounded Staleness Consistency Models," in *PACT*, 2015.
[90] Z. Liu *et al.*, "Concurrent Data Structures for Near-Memory Computing," in *SPAA*, 2017.
[91] A. Morad *et al.*, "GP-SIMD Processing-in-Memory," *ACM TACO*, 2015.
[92] L. Nai *et al.*, "GraphPIM: Enabling Instruction-Level PIM Offloading in Graph Computing Frameworks," in *HPCA*, 2017.
[93] A. Pattnaik *et al.*, "Scheduling Techniques for GPU Architectures with Processing-in-Memory Capabilities," in *PACT*, 2016.
[94] S. H. Pugsley *et al.*, "NDC: Analyzing the Impact of 3D-Stacked Memory+Logic Devices on MapReduce Workloads," in *ISPASS*, 2014.
[95] D. P. Zhang *et al.*, "TOP-PIM: Throughput-Oriented Programmable Processing in Memory," in *HPDC*, 2014.
[96] Q. Zhu *et al.*, "Accelerating Sparse Matrix-Matrix Multiplication with 3D-Stacked Logic-in-Memory Hardware," in *HPEC*, 2013.
[97] B. Akin *et al.*, "Data Reorganization in Memory Using 3D-Stacked DRAM," in *ISCA*, 2015.
[98] M. Gao *et al.*, "Tetris: Scalable and Efficient Neural Network Acceleration with 3D Memory," in *ASPLOS*, 2017.
[99] M. Drumond *et al.*, "The Mondrian Data Engine," in *ISCA*, 2017.
[100] G. Dai *et al.*, "GraphH: A Processing-in-Memory Architecture for Large-scale Graph Processing," *IEEE TCAD*, 2018.
[101] M. Zhang *et al.*, "GraphP: Reducing Communication for PIM-based Graph Processing with Efficient Data Partition," in *HPCA*, 2018.
[102] Y. Huang *et al.*, "A Heterogeneous PIM Hardware-Software Co-Design for Energy-Efficient Graph Processing," in *IPDPS*, 2020.
[103] Y. Zhuo *et al.*, "GraphQ: Scalable PIM-based Graph Processing," in *MICRO*, 2019.
[104] P. C. Santos *et al.*, "Operand Size Reconfiguration for Big Data Processing in Memory," in *DATE*, 2017.
[105] W.-M. Hwu *et al.*, "Rebooting the Data Access Hierarchy of Computing Systems," in *ICRC*, 2017.
[106] M. Besta *et al.*, "SISA: Set-Centric Instruction Set Architecture for Graph Mining on Processing-in-Memory Systems," in *MICRO*, 2021.
[107] J. D. Ferreira *et al.*, "pLUTo: In-DRAM Lookup Tables to Enable Massively Parallel General-Purpose Computation," *arXiv:2104.07699 [cs.AR]*, 2021.
[108] A. Olgun *et al.*, "QUAC-TRNG: High-Throughput True Random Number Generation Using Quadruple Row Activation in Commodity DRAMs," in *ISCA*, 2021.
[109] S. Lloyd and M. Gokhale, "In-memory Data Rearrangement for Irregular, Data-intensive Computing," *Computer*, 2015.
[110] D. G. Elliott *et al.*, "Computational RAM: Implementing Processors in Memory," *IEEE Design & Test of Computers*, 1999.
[111] L. Zheng *et al.*, "RRAM-based TCAMs for pattern search," in *ISCAS*, 2016.
[112] J. Landgraf *et al.*, "Combining Emulation and Simulation to Evaluate a Near Memory Key/Value Lookup Accelerator," 2021.
[113] A. Rodrigues *et al.*, "Towards a Scatter-Gather Architecture: Hardware and Software Issues," in *MEMSYS*, 2019.
[114] S. Lloyd and M. Gokhale, "Design Space Exploration of Near Memory Accelerators," in *MEMSYS*, 2018.
[115] S. Lloyd and M. Gokhale, "Near Memory Key/Value Lookup Acceleration," in *MEMSYS*, 2017.
[116] M. Gokhale *et al.*, "Near Memory Data Structure Rearrangement," in *MEMSYS*, 2015.
[117] R. Nair *et al.*, "Active Memory Cube: A Processing-in-Memory Architecture for Exascale Systems," *IBM JRD*, 2015.
[118] A. C. Jacob *et al.*, "Compiling for the Active Memory Cube," Tech. rep. RC25644 (WAT1612-008). IBM Research Division, Tech. Rep., 2016.
[119] Z. Sura *et al.*, "Data Access Optimization in a Processing-in-Memory System," in *CF*, 2015.
[120] R. Nair, "Evolution of Memory Architecture," *Proceedings of the IEEE*, 2015.
[121] R. Balasubramonian *et al.*, "Near-Data Processing: Insights from a MICRO-46 Workshop," *IEEE Micro*, 2014.
[122] Y. Xi *et al.*, "In-Memory Learning With Analog Resistive Switching Memory: A Review and Perspective," *Proceedings of the IEEE*, 2020.
[123] K. Hsieh *et al.*, "Accelerating Pointer Chasing in 3D-Stacked Memory: Challenges, Mechanisms, Evaluation," in *ICCD*, 2016.
[124] A. Boroumand *et al.*, "LazyPIM: An Efficient Cache Coherence Mechanism for Processing-in-Memory," *CAL*, 2016.
[125] C. Giannoula *et al.*, "SparseP: Towards Efficient Sparse Matrix Vector Multiplication on Real Processing-In-Memory Systems," *arXiv preprint arXiv:2201.05072*, 2022.
[126] C. Giannoula *et al.*, "Towards Efficient Sparse Matrix Vector Multiplication on Real Processing-in-Memory Architectures," in *SIGMETRICS*, 2022.
[127] A. Denzler *et al.*, "Casper: Accelerating stencil computation using near-cache processing," *arXiv preprint arXiv:2112.14216*, 2021.
[128] A. Boroumand *et al.*, "Polynesia: Enabling Effective Hybrid Transactional/Analytical Databases with Specialized Hardware/Software Co-Design,"

arXiv:2103.00798 [cs.AR], 2021.
[129] A. Boroumand *et al.*, "Polynesia: Enabling Effective Hybrid Transactional Analytical Databases with Specialized Hardware Software Co-Design," in *ICDE*, 2022.
[130] G. Singh *et al.*, "FPGA-based Near-Memory Acceleration of Modern Data-Intensive Applications," *IEEE Micro*, 2021.
[131] G. Singh *et al.*, "Accelerating Weather Prediction using Near-Memory Reconfigurable Fabric," *ACM TRETS*, 2021.
[132] J. M. Herruzo *et al.*, "Enabling Fast and Energy-Efficient FM-Index Exact Matching Using Processing-Near-Memory," *The Journal of Supercomputing*, 2021.
[133] L. Yavits *et al.*, "GIRAF: General Purpose In-Storage Resistive Associative Framework," *IEEE TPDS*, 2021.
[134] B. Asgari *et al.*, "FAFNIR: Accelerating Sparse Gathering by Using Efficient Near-Memory Intelligent Reduction," in *HPCA*, 2021.
[135] A. Boroumand *et al.*, "Google Neural Network Models for Edge Devices: Analyzing and Mitigating Machine Learning Inference Bottlenecks," *arXiv preprint arXiv:2109.14320*, 2021.
[136] A. Boroumand *et al.*, "Google Neural Network Models for Edge Devices: Analyzing and Mitigating Machine Learning Inference Bottlenecks," in *PACT*, 2021.
[137] A. Boroumand, "Practical Mechanisms for Reducing Processor-Memory Data Movement in Modern Workloads," Ph.D. dissertation, Carnegie Mellon University, 2020.
[138] G. Singh *et al.*, "NERO: A Near High-Bandwidth Memory Stencil Accelerator for Weather Prediction Modeling," in *FPL*, 2020.
[139] V. Seshadri and O. Mutlu, "Simple Operations in Memory to Reduce Data Movement," in *Advances in Computers, Volume 106*, 2017.
[140] S. Diab *et al.*, "High-throughput Pairwise Alignment with the Wavefront Algorithm using Processing-in-Memory," *arXiv preprint arXiv:2204.02085*, 2022.
[141] S. Diab *et al.*, "High-throughput Pairwise Alignment with the Wavefront Algorithm using Processing-in-Memory," in *HICOMB*, 2022.
[142] D. Fujiki *et al.*, "In-Memory Data Parallel Processor," in *ASPLOS*, 2018.
[143] Y. Zha and J. Li, "Hyper-AP: Enhancing Associative Processing Through A Full-Stack Optimization," in *ISCA*, 2020.
[144] O. Mutlu, "Memory Scaling: A Systems Architecture Perspective," *IMW*, 2013.
[145] O. Mutlu and L. Subramanian, "Research Problems and Opportunities in Memory Systems," *SUPERFRI*, 2014.
[146] H. Ahmed *et al.*, "A Compiler for Automatic Selection of Suitable Processing-in-Memory Instructions," in *DATE*, 2019.
[147] S. Jain *et al.*, "Computing-in-Memory with Spintronics," in *DATE*, 2018.
[148] N. M. Ghiasi *et al.*, "GenStore: A High-Performance and Energy-Efficient In-Storage Computing System for Genome Sequence Analysis," in *ASPLOS*, 2022.
[149] G. F. Oliveira *et al.*, "DAMOV: A New Methodology and Benchmark Suite for Evaluating Data Movement Bottlenecks," *arXiv:2105.03725 [cs.AR]*, 2021.
[150] S. Cho *et al.*, "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.
[151] H. Shin *et al.*, "McDRAM: Low latency and energy-efficient matrix computations in DRAM," *IEEE TCADICS*, 2018.
[152] P. Gu *et al.*, "iPIM: Programmable In-Memory Image Processing Accelerator using Near-Bank Architecture," in *ISCA*, 2020.
[153] D. Lavenier *et al.*, "Variant Calling Parallelization on Processor-in-Memory Architecture," in *BIBM*, 2020.
[154] V. Zois *et al.*, "Massively Parallel Skyline Computation for Processing-in-Memory Architectures," in *PACT*, 2018.
[155] F. Devaux, "The True Processing In Memory Accelerator," in *Hot Chips*, 2019.
[156] UPMEM, "UPMEM Website," https://www.upmem.com, 2023.
[157] UPMEM, "Introduction to UPMEM PIM. Processing-in-memory (PIM) on DRAM Accelerator (White Paper)," 2018.
[158] J. Gómez-Luna *et al.*, "Benchmarking a New Paradigm: An Experimental Analysis of a Real Processing-in-Memory Architecture," arXiv:2105.03814 [cs.AR], 2021.
[159] J. Gómez-Luna *et al.*, "Benchmarking a New Paradigm: Experimental Analysis and Characterization of a Real Processing-in-Memory System," *IEEE Access*, 2022.
[160] J. Gómez-Luna *et al.*, "Benchmarking Memory-Centric Computing Systems: Analysis of Real Processing-In-Memory Hardware," in *IGSC*, 2021.
[161] Y.-C. Kwon *et al.*, "25.4 A 20nm 6GB Function-In-Memory DRAM, Based on HBM2 with a 1.2 TFLOPS Programmable Computing Unit Using Bank-Level Parallelism, for Machine Learning Applications," in *ISSCC*, 2021.
[162] S. Lee *et al.*, "Hardware Architecture and Software Stack for PIM Based on Commercial DRAM Technology: Industrial Product," in *ISCA*, 2021.
[163] L. Ke *et al.*, "Near-Memory Processing in Action: Accelerating Personalized Recommendation with AxDIMM," *IEEE Micro*, 2021.
[164] S. Lee *et al.*, "A 1ynm 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.
[165] D. Niu *et al.*, "184QPS/W 64Mb/mm2 3D Logic-to-DRAM Hybrid Bonding with Process-Near-Memory Engine for Recommendation System," in *ISSCC*, 2022.
[166] D. A. Freedman, *Statistical Models: Theory and Practice*, 2009.
[167] X. Yan and X. Su, *Linear Regression Analysis: Theory and Computing*, 2009.
[168] D. W. Hosmer Jr *et al.*, *Applied Logistic Regression*, 2013.
[169] S. Suthaharan, "Decision Tree Learning," in *Machine Learning Models and Algorithms for Big Data Classification*, 2016.
[170] S. P. Lloyd, "Least Squares Quantization in PCM," *IEEE Transactions on Information Theory*, 1982.
[171] N. P. Jouppi *et al.*, "Ten Lessons from Three Generations Shaped Google's TPUv4i: Industrial Product," in *ISCA*, 2021.
[172] D. B. Kirk *et al.*, *Programming Massively Parallel Processors, 3rd Edition, Chapter 16 - Application Case Study: Machine Learning*. Morgan Kaufmann, 2017.
[173] S. Chetlur *et al.*, "cuDNN: Efficient Primitives for Deep Learning," *arXiv preprint arXiv:1410.0759*, 2014.
[174] M. Abadi *et al.*, "Tensorflow: A System for Large-scale Machine Learning," in *OSDI*, 2016.
[175] Run:AI, "Best GPU for Deep Learning," https://www.run.ai/guides/gpu-deep-learning/best-gpu-for-deep-learning/, 2021.
[176] N. P. Jouppi *et al.*, "In-Datacenter Performance Analysis of a Tensor Processing Unit," in *ISCA*, 2017.
[177] UPMEM, "UPMEM User Manual. Version 2023.1.0," 2023.
[178] UPMEM, "UPMEM Software Development Kit (SDK)." https://sdk.upmem.com, 2023.
[179] N. Zmora *et al.*, "Achieving FP32 Accuracy for INT8 Inference Using Quantization Aware Training with NVIDIA TensorRT," https://developer.nvidia.com/blog/achieving-fp32-accuracy-for-int8-inference-using-quantization-\aware-training-with-tensorrt/.
[180] A. Gholami *et al.*, "A Survey of Quantization Methods for Efficient Neural Network Inference," in *Low-Power Computer Vision*.
[181] NVIDIA, "NVIDIA H100 Tensor Core GPU Architecture. White Paper," https://nvdam.widen.net/s/9bz6dw7dqr/gtc22-whitepaper-hopper, 2022.
[182] J. Han and C. Moraga, "The Influence of the Sigmoid Function Parameters on the Speed of Backpropagation Learning," in *IWANN*, 1995.
[183] Q. Deng *et al.*, "LAcc: Exploiting Lookup Table-based Fast and Accurate Vector Multiplication in DRAM-based CNN Accelerator," in *DAC*, 2019.
[184] M. Gao *et al.*, "DRAF: A Low-power DRAM-based Reconfigurable Acceleration Fabric," in *ISCA*, 2016.
[185] E. W. Weisstein, "Taylor Series," https://mathworld.wolfram.com/TaylorSeries.html, 2004.
[186] SAFARI Research Group, "PIM Machine Learning Training Benchmarks," https://github.com/CMU-SAFARI/pim-ml.
[187] S. Williams *et al.*, "Roofline: An Insightful Visual Performance Model for Multicore Architectures," *CACM*, 2009.
[188] Intel, "Intel Xeon Processor E3-1225 v6," https://ark.intel.com/content/www/us/en/ark/products/97476/intel-xeon-processor-e3-1225-v6-8m-cache-3-30-ghz.html, 2017.
[189] Intel, "Intel Advisor," 2020.
[190] J. Nider *et al.*, "A Case Study of Processing-in-Memory in off-the-Shelf Systems," in *USENIX ATC*, 2021.
[191] B. J. Smith, "A Pipelined, Shared Resource MIMD Computer," in *ICPP*, 1978.
[192] B. J. Smith, "Architecture and Applications of the HEP Multiprocessor Computer System," in *SPIE, Real-Time signal processing IV*, 1981.
[193] J. E. Thornton, *CDC 6600: Design of a Computer*, 1970.
[194] R. S. Sutton and A. G. Barto, *Reinforcement learning: An Introduction*, 2018.
[195] H. Cho *et al.*, "Fa3c: Fpga-accelerated deep reinforcement learning," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2019.
[196] B. T. Polyak, *Introduction to Optimization*, 1987.
[197] Intel, "Intel oneAPI Math Kernel Library (MKL)," https://www.intel.com/content/www/us/en/developer/tools/oneapi/onemkl.html, 2022.
[198] NVIDIA, "CUDA Basic Linear Algebra Subroutine (cuBLAS) Library," https://docs.nvidia.com/cuda/cublas/index.html, 2022.
[199] F. Pedregosa *et al.*, "Scikit-learn: Machine Learning in Python," *JMLR*, 2011.
[200] RAPIDS, "cuML: K-Means Clustering," https://docs.rapids.ai/api/cuml/stable/api.html, 2022.
[201] E. W. Weisstein, "Frobenius Norm. From MathWorld – A Wolfram Web Resource," http://mathworld.wolfram.com/FrobeniusNorm.html, last visited on 1/4/2022.
[202] Intel, "Intel Xeon Silver 4215 Processor," https://ark.intel.com/content/www/us/en/ark/products/193389/intel-xeon-silver-4215-processor-11m-cache-2-50-ghz.html, 2019.
[203] 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.
[204] P. Baldi *et al.*, "Searching for Exotic Particles in High-energy Physics with Deep Learning," *Nature Communications*, 2014.
[205] A. D. Rajen Bhatt, "SUSY Dataset. UCI Machine Learning Repository." [Online]. Available: http://archive.ics.uci.edu/ml/datasets/SUSY
[206] A. D. Rajen Bhatt, "Skin Segmentation Dataset. UCI Machine Learning Repository." [Online]. Available: https://archive.ics.uci.edu/ml/datasets/skin+

segmentation
[207] D. Dua and C. Graff, "UCI Machine Learning Repository," 2017. [Online]. Available: http://archive.ics.uci.edu/ml
[208] T. Caliński and J. Harabasz, "A Dendrite Method for Cluster Analysis," *Communications in Statistics-theory and Methods*, 1974.
[209] L. Hubert and P. Arabie, "Comparing Partitions," *Journal of Classification*, 1985.
[210] A. Wijaya and A. Bisri, "Hybrid Decision Tree and Logistic Regression Classifier for Email Spam Detection," in *ICITEE*, 2016.
[211] M.-w. Chang *et al.*, "Partitioned Logistic Regression for Spam Filtering," in *SIGKDD*, 2008.
[212] B. K. Dedeturk and B. Akay, "Spam Filtering Using a Logistic Regression Model Trained by an Artificial Bee Colony Algorithm," *Applied Soft Computing*, 2020.
[213] N. Sivasankari and S. Kamalakkannan, "Detection and Prevention of Man-in-the-middle Attack in IoT Network Using Regression Modeling," *Advances in Engineering Software*, 2022.
[214] J. Á. Martín-Baos *et al.*, "IoT-based Monitoring of Air Quality and Traffic Using Regression Analysis," *Applied Soft Computing*, 2022.
[215] A. Akbar *et al.*, "Predictive Analytics for Complex IoT Data Streams," *IEEE Internet of Things Journal*, 2017.
[216] A. A. Sarangdhar and V. Pawar, "Machine Learning Regression Technique for Cotton Leaf Disease Detection and Controlling Using IoT," in *ICECA*, 2017.
[217] E. Azarkhish *et al.*, "Neurostream: Scalable and Energy Efficient Deep Learning with Smart Memory Cubes," *TPDS*, 2017.
[218] Y. Kwon *et al.*, "TensorDIMM: A Practical Near-Memory Processing Architecture for Embeddings and Tensor Operations in Deep Learning," in *MICRO*, 2019.
[219] L. Ke *et al.*, "RecNMP: Accelerating Personalized Recommendation with Near-Memory Processing," in *ISCA*, 2020.
[220] A. S. Cordeiro *et al.*, "Machine Learning Migration for Efficient Near-Data Processing," in *PDP*, 2021.
[221] 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.
[222] N. Park *et al.*, "High-Throughput Near-Memory Processing on CNNs with 3D HBM-Like Memory," *TODAES*, 2021.
[223] J. Park *et al.*, "TRiM: Enhancing Processor-Memory Interfaces with Scalable Tensor Reduction in Memory," in *MICRO*, 2021.
[224] B. Kim *et al.*, "MViD: Sparse Matrix-Vector Multiplication in Mobile DRAM for Accelerating Recurrent Neural Networks," *IEEE Transactions on Computers*, 2020.
[225] 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.
[226] H. Sun *et al.*, "An Energy-Efficient Quantized and Regularized Training Framework for Processing-in-Memory Accelerators," in *ASP-DAC*, 2020.
[227] Y. Luo and S. Yu, "Accelerating Deep Neural Network In-Situ Training with Non-Volatile and Volatile Memory Based Hybrid Precision Synapses," *IEEE Transactions on Computers*, 2020.
[228] B. Li *et al.*, "3D-ReG: A 3D ReRAM-Based Heterogeneous Architecture for Training Deep Neural Networks," *JETC*, 2020.
[229] J.-W. Su *et al.*, "A 28nm 64Kb Inference-Training Two-Way Transpose Multibit 6T SRAM Compute-in-Memory Macro for AI Edge Chips," in *ISSCC*, 2020.
[230] M. Imani *et al.*, "FloatPIM: In-Memory Acceleration of Deep Neural Network Training with High Precision," in *ISCA*, 2019.
[231] H. Jiang *et al.*, "CIMAT: A Transpose SRAM-Based Compute-in-Memory Architecture for Deep Neural Network On-Chip Training," in *MEMSYS*, 2019.
[232] M. Cheng *et al.*, "TIME: A Training-in-Memory Architecture for RRAM-Based Deep Neural Networks," *TCAD*, 2018.
[233] J. Liu *et al.*, "Processing-in-Memory for Energy-Efficient Neural Network Training: A Heterogeneous Approach," in *MICRO*, 2018.
[234] M. J. Marinella *et al.*, "Multiscale Co-Design Analysis of Energy, Latency, Area, and Accuracy of a ReRAM Analog Neural Training Accelerator," *JETCAS*, 2018.
[235] F. Schuiki *et al.*, "A Scalable Near-Memory Architecture for Training Deep Neural Networks on Large In-Memory Datasets," *IEEE Transactions on Computers*, 2019.
[236] R. Hasan *et al.*, "A Fast Training Method for Memristor Crossbar Based Multi-Layer Neural Networks," *Analog Integrated Circuits and Signal Processing*, 2017.
[237] P. Gu *et al.*, "DLUX: A LUT-Based Near-Bank Accelerator for Data Center Deep Learning Training Workloads," *TCAD*, 2020.
[238] G. F. Oliveira, "DAMOV: A New Methodology and Benchmark Suite for Evaluating Data Movement Bottlenecks," https://people.inf.ethz.ch/omutlu/pub/DAMOV-Bottleneck-Analysis-and-DataMovement-Benchmarks_arxiv21-talk.pptx, video available at https://youtu.be/GWideVyo0nM, 2021, SAFARI Live Seminar, 22 July 2021.
[239] H. Falahati *et al.*, "ORIGAMI: A Heterogeneous Split Architecture for In-Memory Acceleration of Learning," *arXiv:1812.11473 [cs.LG]*, 2018.
[240] J. Vieira *et al.*, "Exploiting Compute Caches for Memory Bound Vector Operations," in *SBAC-PAD*, 2018.
[241] Z. Sun *et al.*, "One-Step Regression and Classification with Cross-Point Resistive Memory Arrays," *Science Advances*, 2020.
[242] C. F. Shelor and K. M. Kavi, "Reconfigurable Dataflow Graphs for Processing-in-Memory," in *ICDCN*, 2019.
[243] J. Saikia *et al.*, "K-Nearest Neighbor Hardware Accelerator Using In-Memory Computing SRAM," in *ISLPED*, 2019.
[244] J. Gómez-Luna *et al.*, "An Experimental Evaluation of Machine Learning Training on a Real Processing-in-Memory System," *arXiv preprint arXiv:2207.07886*, 2022.
[245] J. Gómez-Luna *et al.*, "Machine Learning Training on a Real Processing-in-Memory System," in *ISVLSI*, 2022.