A Fast and Energy Efficient Computing-in-Memory Architecture for Few-Shot Learning Applications

Dayane Reis*, Ann Franchesca Laguna*, Michael Niemier, and Xiaobo Sharon Hu
Department of Computer Science and Engineering
University of Notre Dame, Notre Dame, IN, USA, 46556
E-mails: {dreis, alaguna, mniemier, shu}@nd.edu
*These authors contributted equally to this work.

***Abstract*—Among few-shot learning methods, prototypical networks (PNs) are one of the most popular approaches due to their excellent classification accuracies and network simplicity. Test examples are classified based on their distances from class prototypes. Despite the application-level advantages of PNs, the latency of transferring data from memory to compute units is much higher than the PN computation time. Thus, PNs performance is limited by memory bandwidth. Computing-in-memory addresses this bandwidth-bottleneck problem by bringing a subset of compute units closer to memory. In this work, we propose a CiM-PN framework that enables the computation of distance metrics and prototypes inside the memory. CiM-PN replaces the computationally intensive Euclidean distance metric by the CiM-friendly Manhattan distance metric. Additionally, prototypes are computed using an in-memory mean operation realized by accumulation and division by powers of two, which enables few-shot learning implementations where "shots" are powers of two. The CiM-PN hardware uses CMOS memory cells, as well as CMOS peripherals such as customized sense amplifiers, carry-look-ahead adders, in-place copy buffers and a logarithmic shifter. Compared with a GPU implementation, a CMOS-based CiM-PN achieves speedups of 2808x/111x and energy savings of 2372x/5170x at iso-accuracy for the prototype and nearest-neighbor computation, respectively, and over 2x end-to-end speedup and energy improvements. We also gain 3-14% accuracy improvement when compared to existing non-GPU hardware approaches due to the floating-point CiM operations.**

***Index Terms*—Few-shot learning, prototypical networks, computing-in-memory**

I. Introduction

Traditional neural networks cannot quickly adapt to new tasks without extensive retraining with a large amount of information. On the contrary, humans leverage acquired knowledge to quickly adapt to new situations. The task of learning how to learn – or *meta-learning* [1] – leverages past experiences to learn new tasks. *Few-shot learning* [1], [2], a form of meta-learning, uses just a few examples to learn an unseen class. A common approach called *metric-based few-shot learning* [3], is to learn a metric space in which classes can be distinguished using a distance metric.

In this work, we focus on hardware support for few-shot learning tasks that employ prototypical networks (PNs) [3], a commonly-used metric-based few-shot learning model. Per Fig. 1, in prototypical learning, a neural network is used to convert an example into a feature vector in an embedding space. To learn a new class, the feature embeddings of the training examples are averaged to form a prototype (e.g., c_1, c_2, and c_3 are different prototypes in Fig. 1). During inference, a query x is compared to these prototypes using a distance metric to determine x's class. These steps require substantial amounts of data transfer between the memory and processing units. Moreover, PN operations become increasingly memory-bandwidth limited as the number of prototypes

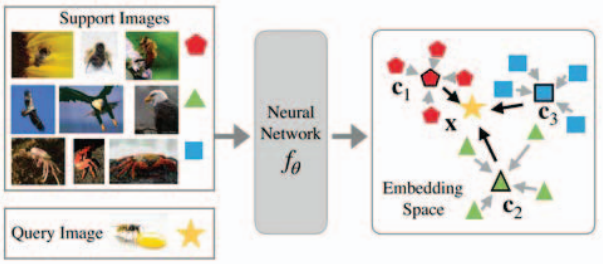


Fig. 1: Prototypical Network projects the images to an embedding space using neural network f_θ where the embedding of query image is compared to the prototype of the support images

increases. Computing-in-memory (CiM) architectures could circumvent the memory bandwidth bottleneck which degrades performance and increases energy consumption.

CiM paradigms [4]–[7] aim to overcome the "memory wall" [8], as data can be processed without moving it to a central processing unit (CPU). CiM enables energy and time-efficient computation within the boundaries of memory due to significantly fewer data transfers [9]. Additionally, CiM can take advantage of memory's inherently high internal bandwidth by processing data in parallel within different arrays. As such, CiM can be beneficial to a wide range of applications across different domains, including few-shot learning [6], [10], [11].

Previous CiM implementations of few-shot learning algorithms [6], [10]–[12] primarily focus on the distance calculations required to identify nearest neighbors. For instance, [10] implements a ℓ^∞ distance metric to perform a nearest neighbor search using a Ternary Content Addressable Memory (TCAM). While TCAMs enable distance metric calculations between a query and all memory entries to be performed in parallel, application-level classification accuracy when using TCAMs with ℓ^∞ distance metrics is significantly lower than cases where cosine or Euclidean (ℓ^2 norm)) are employed [3], [6]. To address this problem. [10] uses combinations of ℓ^∞ and ℓ^1 (Manhattan) distance metrics to improve accuracy for nearest neighbor calculations. While an ℓ^∞ metric can be calculated via TCAM arrays, the implementation of a Manhattan distance metric is supported by the use of an in-memory adder in a general-purpose CiM design. Although higher accuracies can be obtained with this method (assuming a fixed point data representation), the calculation of combined distance metrics as proposed in [6] requires two types of CiM hardware, i.e. a TCAM and a CiM random-access memory (RAM). Crossbar arrays [11] can implement dot products but to compute for cosine distances, the divisions must still be outsourced to a special functional unit. There are also accuracy degradations attributed to using fixed-point precision instead

978-3-9819263-4-7/DATE20/©2020 EDAA

of floating point precision. Floating point representation and operations are not supported in [6], [10]–[12].

In this work, we propose a unified CiM framework that can support both fixed and floating point precision. Our CiM-PN framework enables the calculation of (i) *prototype representations* (i.e., the ability to compute representative prototypes for different classes) and (ii) *nearest neighbors* based on the Manhattan distance metric for few-shot classification of query images. The core computations for PN are enabled by our proposed CiM architecture through in-memory copy buffers, carry-look-ahead adders (CLA) and logarithmic shifters. To compute the prototype representation and the Manhattan distance metric in the memory, we introduce the in-memory mean (*iM-Mean*) and in-memory Manhattan (*iM-NearestNeighbor*) operations, which can be realized by a sequence of subtractions, divisions, and additions in the memory with CiM hardware.

To evaluate the impact of our CiM architecture for PNs, we compare classification accuracies for the Omniglot [13] and mini-ImageNet [2], [14] datasets to a GPU-based PN that uses the Euclidean distance metric. The Euclidean distance has been shown to produce more accurate classification accuracies with PNs than cosine distance [3]. To evaluate our CiM approach, we employ the BSIM-CMG FinFET model from [15] with a 14nm technology node in HSPICE simulations to quantify the impact of our CiM hardware components, including the memory array, peripherals, CLA, logarithmic shifter, and write/copy buffers. Compared with a GPU implementation that uses a classic Euclidean-squared distance for classification, our Manhattan distance-based CiM-PN achieves a speed-up of 2808x/111x and energy improvement of 2372x/5170x at iso-accuracy for the prototype and nearest-neighbor computations respectively for a single query, and end-to-end speedups and energy improvements of over 2x.

II. Background

To support subsequent discussions, PNs are discussed in Sec. II-A. Prior work that employs CiM-based solutions for few-shot learning applications is discussed in Sec. II-B.

A. Prototypical Network for Few-Shot Learning

Traditional neural networks cannot learn new classes without extensive retraining. One-shot or few-shot learning aims to solve this problem and aims to learn using only one or a few examples. For example, in an N-way K-shot classification, the network must determine in which of the N different classes the query image q belongs to using only K training examples for each class.

Prototypical networks [3] aim to do this by learning an embedding (Fig. 1) in which all examples in a given class clusters to a single prototype . The prototype is computed using the mean of all embedded support points. The prototypes are computed using the mean of all embedded support points. The prototypes are then compared to the embedding of the query using a weighted nearest neighbor using a distance metric, such as cosine distance and Euclidean-squared distance [3].

In prototypical networks [3], a neural network is used to obtain feature embeddings $f(x)$ of the support set and the query. The neural network proposed in [3] is composed of four convolutional boxes, each with 64-filter 3×3 convolution, a batch normalization layer, a ReLU nonlinearity, a 2×2 max-pooling layer, and a fully-connected layer. The output of the last layer forms the embedding vector that is used to compute for the prototypes. The output embedding vector for Omniglot is 64-dimensions and is 3200-dimensions for miniImageNet.

The training examples for each class are averaged into a single prototype $\mathbf{c}_k$.

$$\mathbf{c}_k = \frac{1}{K} \sum_{(\mathbf{x}_i, y_i) \in S_k} f(\mathbf{x}_i) \tag{1}$$

where S_k represents the support set of class k and $\mathbf{x}_i, y_i$ are the embedding vectors and label respectively. The prototypes are then compared using a distance metric such as cosine distance or Euclidean-squared distance metric. The Euclidean-squared metric has shown higher accuracies [3] for PN. Hence we use Euclidean-squared metric as our baseline. In this work, we replace the Euclidean-squared distance metric with a more hardware friendly Manhattan distance metric which removes the square operation.

The loss function is then computed using the negative probability distribution $-\log p(x)$ using a softmax function of the distances. The distances are minimized by a stochastic gradient descent. During inference, the softmax can be replaced by a nearest neighbor calculation. We focus on inference for PNs, and the softmax function is not included in the implementation.

The prototypical network can be implemented using additions and subtraction. The division operation in Eq. 1 is determined by the number of shots (number of training examples per class). Typically, few-shot learning networks are benchmarked using 1-shot or 5-shot tasks. In deployment, this may not be the case and the number of shots may vary.

B. Related Work: CiM Solutions to Few-shot Learning

In recent years, different approaches have emerged to address the memory-bandwidth problem associated with the so-called memory-wall problem [8]. Near-memory processing (NMP) and CiM are two computing paradigms that aim to bring computation closer to (or even inside) the memory. With NMP, processing units are placed onto the same chip as the memory, thus reducing the latency and energy costs associated with data transfers [16]–[18]. A downside of NMP is that the integration of density-optimized memory, and performance-optimized processing units on the same die may not be cost-effective due to fabrication challenges [19]. As an alternative, CiM leverages changes in either the structure of memory cells or peripheral circuits to perform in-memory computation at the array level – i.e., without data transfers to an external processing unit [4], [5], [7]. CiM designs can be general-purpose or application specific (as in the case of few-shot learning). Previous CiM hardware, such as crossbar-arrays and TCAM solutions for few-shot learning focus on implementing Memory Augmented Neural Networks (MANNs), where a memory module is used as a cache for the embedding operations. MANNs are similar in structure to PNs. These CiM hardware approaches for few-shot learning are discussed below.

1) Crossbar Arrays: Crossbar arrays can perform matrix multiplications and are hence suitable for calculating dot products [20]. As such, the XMANN [11] paper proposed to use crossbar arrays to perform distance calculation searches for

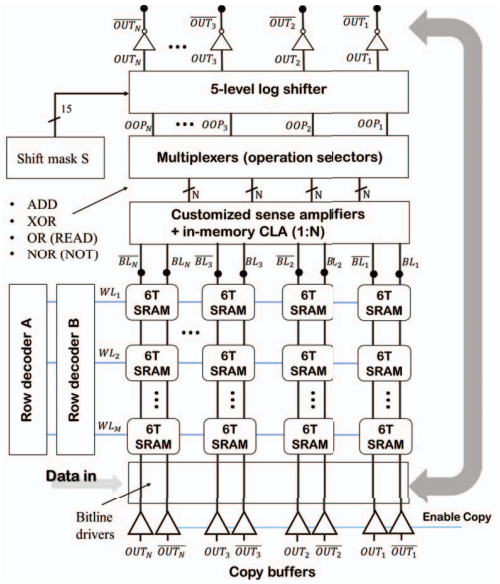


Fig. 2: Computing-in-Memory Prototypical Network (CiM-PN) array

MANNs [1] and neural turing machines (NTM) [21] (networks that are also used for few-shot learning). Traditionally, cosine similarity has been used in neural networks to search the memory or compare samples or prototypes to solve a one-/few-shot learning problems. XMANN [11] proposed a crossbar friendly, modified similarity metric to replace the original cosine similarity. The modified cosine distance replaces the product of the ℓ^2 norms of the query and the memory entries by the sum of the ℓ^1 norms of the query and the memory entries. To utilize the crossbar for cosine distance, the ℓ^1 norms of the memory entries are computed by performing a dot product of the memory entries with a vector of all $1's$. A special functional unit is then used to perform additions and division.

2) Ternary Content Addressable Memories: Content addressable memories or CAMs are memory devices that perform parallel associative Hamming distance searches in memory between a query vector and the memory entries. This approach is suitable for a locality sensitive hashing search approach [12]. *Locality sensitive hashing (LSH)* is an algorithmic technique that aims to produce similar hashes with nearby points. TCAMs have also been used to calculate ℓ^∞ and ℓ^1 distance metrics using a range encoding scheme [6], [10]. TCAMs allow the presence of wildcard bits, i.e. the don't care states, that facilitates the search over different ranges [22].

III. CiM-based Prototypical Networks

We propose a CiM-based prototypical network (CiM-PN) framework, which comprises CiM hardware implementation and algorithm mapping. In Sec. III-A, we discuss the CiM hardware, consisting of memory cells, a customized sense amplifier performing bitwise logic, a carry-look-ahead (CLA) adder and a logarithmic shifter to support computations for PNs. Furthermore, we demonstrate CiM-based operations with integers and FLP operations that can be performed with our CiM hardware. Finally, we explain how these operations can be used for inference.

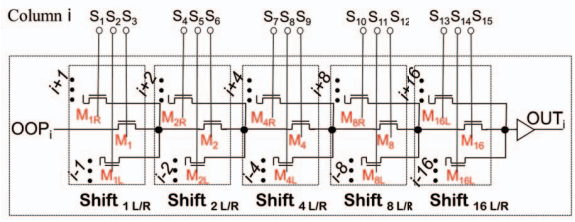


Fig. 3: Log shifter in CiM-PN. Bits S_1 to S_{15} forms the shift mask. This shifter supports left and right shifts of up to 31 bits, i.e., achieves a single-step multiplications (divisions) by powers of two (up to 2^{31}). Bit shifts higher than 31 bits are possible with multiple steps, i.e., by performing a smaller shift, writing the result to memory, and performing a new shift.

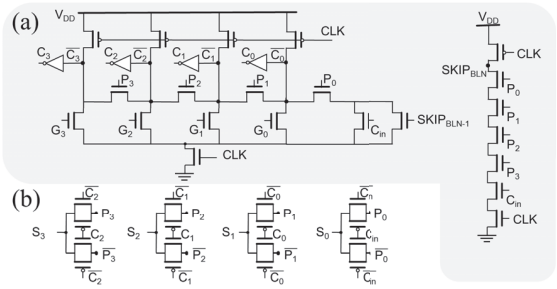


Fig. 4: In-memory carry-look-ahead adder implemented in CiM-PN (iM-CLA). (a) Manchester carry and carry skip circuits. (b) Sum circuits.

A. CiM Hardware for Prototypical Networks

Our memory space is composed of multiple memory arrays of $M \times N$ size such as the one depicted in Fig. 2. In these arrays, we store (i) the feature vectors of images in the support set (which are used in the prototype computation), and (ii) prototypes computed in previous episodes that are used to compare a query image to prototypes via a distance metric calculation. Computation for PNs is supported in each of the CiM arrays by a set of customized peripheral circuits.

1) Customized sense amplifiers: CiM-PN leverages/modifies the sense amplifier circuits proposed in [4], which can only compute bitwise NOR and AND operations between two words stored in memory. Specific modifications include: **(i)** added extra CMOS circuitry, including inverters, to compute OR and NAND functions by inverting NOR and AND outputs generated by the sense amplifiers, and **(ii)** an AND gate that uses the results of NOR and AND logic to compute XOR (which is used for addition operations). As in [4], reads and NOT operations can still be performed by selecting one word instead of two at read time. Reads (NOTs) share a common output with OR (NOR) operations. When computing any type of operation in-memory, we must input its *opcode*, i.e., an operator selection, to multiplexers that provides the operation result as inputs to the next CiM component, (e.g., the logarithmic shifter).

2) iM Logarithmic Shifter: CiM-PN uses left and right shifts for several of its operations. For instance, left shifts enable multiplying a number by powers of two, which is needed, e.g., for floating point additions/subtractions in CiM-PN. Furthermore, right shifts (divisions) are used in the computation of prototypes (to perform means). A logarithmic (log) shifter [23] of 5 levels is used in CiM-PN (iM-shifter) to

enable CiM-PN operations. Although log shifters are common building blocks in digital design, to the best of our knowledge, there are no existing implementations of this circuit in the context of a CiM architecture. With the log shifter depicted in Fig. 3, up to 31 right (left) shifts, i.e., divisions (multiplications) by powers of two (up to 2^{31}), can be performed in a single step in the memory. Divisions (multiplications) by 2^{32} or greater may require extra steps. For example, to divide (multiply) by 2^{32}, we first perform a division (multiplication) by 2^{16}, write this result to memory using copy buffers, and then we perform another division (multiplication) by 2^{16}. Compared to other circuits such as the barrel shifter [23], a log shifter enables more flexibility in terms of possible combinations of shifts while occupying less area.

The iM-shifter circuit has N inputs, i.e., the same as the number of columns of a CiM array. These inputs correspond to the outputs of the operation selectors (OOP_0 to OOP_N in Fig 2). The log shifter circuit for one column i is depicted in Fig. 3. The shift mask S, a 15-bit number, determines whether no shift, a right shift, or a left shift occurs in each level of the iM-shifter. For instance, in Fig. 3, the shift mask '010010010010010', activates transistors M_1,M_2,M_4,M_8,M_{16} (and no other transistors). As a consequence, the value of iM-shifter output is equal to the value of iM-shifter input. In the case where we have a '010010010**100**010' value for the shift mask (for a 2 bit right shift) we activate the transistor M_{2R} instead of M_2 in the aforementioned sequence. Therefore, we perform a 2 bit right-shift as we read the word, which is equivalent to a division by $2^2 = 4$. After the read, we can store this shifted result in the memory using copy buffers.

3) iM Carry-Look-Ahead Adder: An in-memory implementation of a carry-look-ahead adder (iM-CLA) is proposed to perform fast addition of two words stored in different rows in memory (without data transfers to an external compute unit). In PNs, addition latency can be critical to total computation time, as many operations require sums to be obtained as part of the functions that they execute. Different from in-memory adders employed in prior work [5], [7], an iM-CLA does not suffer from high addition latency imposed by long carry propagation times. However, given an n-bit iM-CLA adder in which the individual bits are accessed by index, a downside of traditional CLA designs is the large area required to embed the carry circuits of the $(i-1)^{th}$ bit in each $(i)^{th}$ bit. To allow for an area-efficient and fast implementation of a CLA adder in the memory, we employ a Manchester (dynamic logic) carry circuit with 4 stages, referred to as iM-CLA block (Fig. 4(a)). To perform addition on long words, e.g., 32 or 64-bit, we cascade multiple iM-CLA blocks. The carry circuit shown in Fig. 4(a)) needs the generate (G) and propagate (P) inputs for each bit, which are results of in-memory bitwise operations between two words A and B, i.e., $G_i = A_iB_i$, $P_i = A_i \oplus B_i$. The bitwise operations are computed with the sense amplifier described in Sec. III-A1. We combine the Manchester carry with a carry skip circuit to minimize any effects of RC delay and improve our addition performance as in [24]. The sum (Fig. 4(b)) requires the same inputs as the carry, (i.e., bitwise operations $G_i = A_iB_i$, $P_i = A_i \oplus B_i$) and can be implemented with a pass-transistor-logic based XOR function.

4) iM Accumulator: In-memory writes are facilitated by copy buffers as those proposed in [4]. When the result of an operation, (e.g., addition) is finished, we set the enable signals of copy buffers to '1', and select a destination address (by activating its corresponding wordline). As the output of copy buffers are directly connected to the bitlines, an in-place writing of the result to the selected memory destination will occur. By leveraging copy buffers and iM-CLAs, the in-memory accumulate operation is a two-step process of "adds and writes". For instance, in order to accumulate results of n steps of computations (or to obtain a summation of a set of n numbers), we must perform the desired operation (or simply read each number from our set), and add it to the content already stored in the same address in the memory.

B. Mapping Prototypical Network Computations to CiM

Prototypical network inference can be divided into two parts: (1) computation of prototypes, which is the mean of the feature embeddings and (2) search of the nearest prototype c_i from a given query x. Previous hardware implementations for one-shot learning use fixed point representation and are not as accurate as floating point operations (FLOPS) [10], [12]. We leverage the use of CiM operations to implement floating point (FLP) addition and subtraction to compute for the prototype from the support embedding vectors and the nearest prototype as explained below. The neural network are trained using the maximum number of shots and are only constrained to 2^k-shots during inference.

1) Floating point operations: FLP operations show significantly higher accuracy than fixed point operations in few-shot learning applications. FLP representation is divided into the mantissa and exponent parts. To perform an additions or subtraction of FLP numbers, the exponents must have the same value. This requires the mantissa to be shifted if the original exponents are not the same. To minimize rounding errors, given two numbers, the mantissa of the number $\mathbf{a}$ with the larger exponent, is shifted to the left. The exponent of $\mathbf{a}$ is then changed to the smaller value. The length of this shift can be determined by a subtraction and a 2's complement operation (if necessary). After the exponents are aligned, the addition or subtraction operation on the mantissa can be performed. After the addition or subtraction operation, the mantissa is then normalized to represent the proper FLP format again using iM-addition/iM-subtraction and iM-Shift operations. The iM-Subtraction operation is implemented by performing a NOT and Add with a carry-in of 1.

2) iM-Mean operation: Prototypes are computed by obtaining the mean of the samples, which requires addition and division. When the number of samples are powers of 2, the division operation can be replaced by an iM-subtraction operation of the exponent which lowers the number of operations required, and improves the latency and energy of computing the mean. The mean operation can then be implemented using an accumulator (Section III-A4) along with the normalization of exponents to better support in memory distance calculations.

3) iM-NearestNeighbor: We replace the classic Euclidean-squared distance that is typically employed in PNs with the Manhattan distance. The Manhattan distance can be divided into three stages: (i) the subtraction of the query from the prototype implemented, (ii) absolute value, and (iii) summation. These are implemented using a FLP iM-subtraction, a NOT if the value is negative, and FLP iM-accumulate (with carry-in of 1 if the value is negative). The minimum distance is then computed by comparison using subtractions until the minimum distance is found.

TABLE I: Latency and energy values of individual CiM operations

Operation	Latency	Energy
READ/NOT	0.465 ns	1.78 pJ
WRITE	0.153 ns	0.257 pJ
ADD	1.36 ns	9.90 pJ
SUB	1.83 ns	11.7 pJ
(L/R) SHIFT	0.465 ns	1.78 pJ

TABLE II: GPU and CiM accuracy for 2^k prototypical networks using Omniglot and miniImageNet Dataset

Dataset	Hardware	5-way 1-shot	5-way 2-shot	5-way 4-shot	5-way 5-shot
Omniglot	GPU	99.33	99.71	99.56	99.70
	CiM	98.35	99.17	99.56	N/A
miniImageNet	GPU	53.92	61.95	68.74	69.48
	CiM	52.05	61.04	67.83	N/A

IV. Evaluation

We employ the BSIM-CMG FinFET model from [15] with a 14nm technology node in HSPICE simulations to test and evaluate our CiM hardware components, such as the memory array, peripherals, CLA, logarithmic-shifter, and write/copy buffers. Furthermore, we also consider a GPU-based PN-baseline and measure accuracy, power, and time of PN computations assuming the MiniImageNet and Omniglot datasets. Below, we present and discuss our results, and compare a CiM-PN with a GPU-based implementation as well as TCAM [6], [10] and crossbar-based approaches [11].

A. Array Level Evaluation

Using SPICE simulations, we determine the latency and energy of CiM-based operations that support PN operations. We consider CiM-PN based implementation that leverages multiple 64×64 arrays. Table I reports the results for one array and are used for subsequent system-level evaluations

B. System Level Evaluation

For the the system level evaluation, we use an Nvidia Titan GPU with 2688 CUDA cores and 288.4 GB/s memory bandwidth for GPU comparison. The neural network is similar to the one used in [3].

1) Accuracy: Table II summarizes accuracies of GPU and CiM based PNs. The CiM implementation obtained an accuracy closes to the Omniglot dataset [13], and a small (less than 1%) difference with the mini-ImageNet dataset [2], [14].

Given the 5-way 5-shot problem, we compare the accuracies of the GPU, TCAM+CiM [6], [10], XMANN crossbar [11] and our CiM-PN implementations using the distance metrics discussed in [6], [10], [11] as shown in Table III. The distance metric is used in both training and inference. Since CiM-PN can only implement 4-shot and 8-shot cases, we use these accuracies for comparison. For the 8-shot case, three of the 5 examples are used twice in the iM-mean operation. The experimental results show that for both cases of CiM-PN (4-shot and 8-shot) achieve more comparable accuracy to the GPU with 3-14% improvement compared to the TCAM+CiM [6], [10] approach and crossbar XMANN approach [11]. The PN-CiM 8-shot using Manhatan distance has comparable accuracy with the GPU using Manhattan distance. This significant improvement can be attributed to the use of FLP precision Manhattan distance with our in-memory calculation as opposed to fixed point.

TABLE III: 5-way miniImageNet classification for various technologies, given 5 training samples per class. *The 4-shot CiM-PN removes one sample per class. **The 8-shot CiM-PN uses three of the 5 samples per class twice.

Hardware	Distance Metric	Accuracy
GPU (5-shot)	Euclidean (L_2)	69.48
GPU (5-shot)	Manhattan (L_1)	68.91
TCAM+CiM [6], [10] (5-shot)	$L\infty + L_1$	53.84
XMANN Crossbar [11] (5-shot)	Modified Cosine	64.84
PN-CiM (4-shot)*	Manhattan (L_1)	67.83
PN-CiM (8-shot)**	Manhattan (L_1)	68.90

2) Latency and Energy: Energy and latency evaluations of the GPU-based PN are obtained using the Nvidia Profiler (NVProf), NVVP (Nvidia Visual Profiler) and Nvidia-Management-System Management Interface (nvidia-smi) and are summarized in Fig. 5 and 6. CiM-PN obtains a 2808x speedup and 2372x energy improvement for a single episode and 1650x speedup and 89x energy reduction for a parallel implementation of 1000 episodes of the prototype calculation. For a single episode, the GPU does not maximize its parallellism capabilities, with a batch operation it maximizes its parallelism capabilities and reduces the memory overhead by latency hiding [25]. When comparing the CiM implementation of the nearest prototype implementation to a Euclidean-squared distance given a GPU implementation we obtained a 111x speedup and 5170x energy improvement for a single episode, and a 154x speedup and 457x energy improvement for a parallel implementation of 1000 episodes. The 1000 episodes case illustrates expected improvements when both the GPU and CiM achieved their maximum capacities. Since the GPU can also implement Manhattan distance, we also compared the delay and energy of the GPU and CiM-PN based on this metrics and found slightly smaller (but still substantial per the figures) improvements in terms of delay and energy compared to using Euclidean Distance in the GPU.

The roofline model in Fig. 7 illustrates the GPU performance as compared to CiM-PN model for the combined prototype calculation and nearest prototype calculation. The flat red line on the right side represents the GPU compute limit, while the sloped line shows the bandwidth limit with respect to the number of FLOPS/byte. CiM-PN roofline is flat because it only has a compute limit and there are no memory

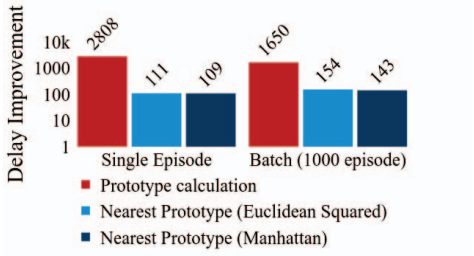


Fig. 5: Normalized Speedup of using CiM-PN compared to GPU using Euclidean Squared and Manhattan distance

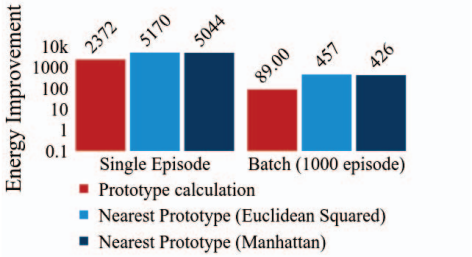


Fig. 6: Normalized Energy Improvement of using CiM-PN compared to GPU using Euclidean Squared and Manhattan distance

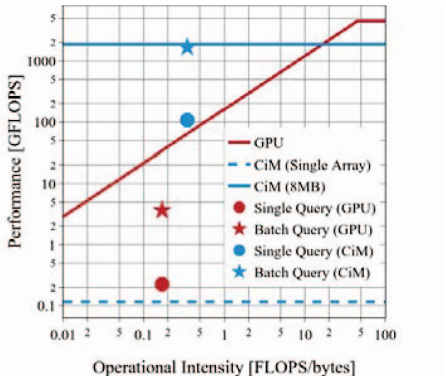
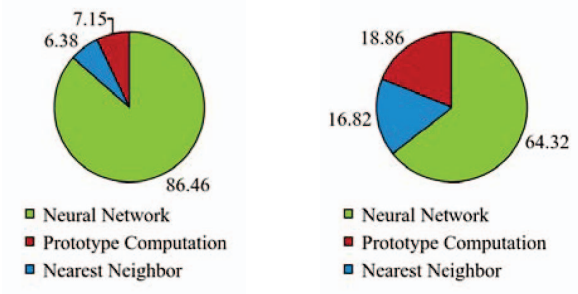


Fig. 7: Roofline model based evalaution.



(a) 5-way 4-shot (b) 100-way 4-shot

Fig. 8: Execution Time Distribution

transfers. The GPU prototypical network implementation lies below the memory bandwidth and is hence memory-bandwidth limited. The CiM-PN shows better performance than the GPU because of parallelism and the absence of memory transfer.

3) End-to-End evaluation: Fig. 8 shows the percentage of computations of the neural network and the calculation of the nearest-prototype for the GPU-based implementation. The prototypical approach generates the feature vectors of the query to be compared to the examples and/or prototypes stored in memory. With lower number of ways (classes), the neural network dominates. However, the percentage of computation associated with the neural network is reduced as the number of ways or classes increases as there are more support vectors that need to be fetched from the memory. For a 5-way classification task, we obtain an overall improvement of 1.2x for both latency and energy, while a 2.1x delay and energy improvement is obtained assuming a 100-way classification task. This improvement increases as the number of ways further increases. Moreover, research on emerging devices for accelerating neural networks can further improve the speed and energy of this component of the system.

V. Conclusion

We introduce a CiM architecture to support PNs for few-shot learning problems, addressing the memory-bandwidth bottleneck. CiM peripherals including logarithmic shifter, carry-look-ahead adder and accumulator are proposed to implement an iM-Mean operation and iM-NearestNeighbor operations. Speedups of 2808x and 111x, and energy improvement of 2372x and 5170x are obtained for *iM-Mean* and *iM-NearestNeighbor* as compared to the GPU implementation at iso-accuracy. Compared to a TCAM+CiM, we obtain a 14% accuracy improvement, and a 3% accuracy improvement compared to a crossbar approach. Improvements in accuracy primarily come from the use of floating point precision that are supported by CiM-PN as opposed to fixed point representation from the TCAM [6], [10] and crossbar [11] approaches.

The number of parallel operations that CiM can support is proportional to the size of the memory. Commercially available SRAM sizes are in the range of 8 to 16MB, with 8MB being more common. In future work, memories based on emerging devices such as Resistive Random-Access Memory, Spin-Transfer Torque Magnetic Random-Access Memory, or Ferroelectric Field Effect Transistors-based Random-Access Memory will be studied in the context of CiM architectures achieve higher density and low associated leakage and should improve scaling.

Acknowledgments

This work was supported in part by ASCENT, one of six centers in JUMP, a Semiconductor Research Corporation (SRC) program sponsored by DARPA.

References

[1] A. Santoro, et al. Meta-Learning with Memory-Augmented Neural Networks. In *ICML*, pages 1842–1850, 2016.
[2] O. Vinyals, et al. Matching Networks for One Shot Learning. In *NeurIPS*, pages 3630–3638, 2016.
[3] J. Snell, et al. Prototypical Networks for Few-shot Learning. In *NIPS*, pages 4080–4090, 2017.
[4] S. Aga, et al. Compute caches. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, Feb 2017.
[5] D. Reis, et al. Computing in Memory with FeFETs. In *ISLPED*, pages 24:1–24:6, New York, NY, USA, 2018. ACM.
[6] A. F. Laguna, et al. Ferroelectric FET Based In-Memory Computing for Few-Shot Learning. In *GLSVLSI*, pages 373–378, 2019.
[7] S. Jain, et al. Computing in Memory With Spin-Transfer Torque Magnetic RAM. *TVLSI*, PP(99):1–14, 2017.
[8] W. A. Wulf et al. Hitting the memory wall: implications of the obvious. *ACM SIGARCH computer architecture news*, 23(1):20–24, 1995.
[9] A. Sebastian, et al. Temporal correlation detection using computational phase-change memory. *Nature Communications*, 8(1):1115, 2017.
[10] A. F. Laguna, et al. Design of hardware-friendly memory enhanced neural networks. In *DATE*, pages 1583–1586, 2019.
[11] A. Ranjan, et al. X-MANN: A Crossbar based Architecture for Memory Augmented Neural Networks. In *DAC*, page 130, 2019.
[12] K. Ni, et al. Ferroelectric Ternary Content Addressable Memory for One-Shot Learning. *To appear in Nature Electronics*, In press.
[13] B. M. Lake, et al. Human-level concept learning through probabilistic program induction. *Science*, 350(6266):1332–1338, 2015.
[14] S. Ravi et al. Optimization as a Model for Few-Shot Learning. In *ICLR*, 2017.
[15] J. P. Duarte, et al. BSIM-CMG: Standard FinFET compact model for advanced circuit design. In *ESSCIRC*, pages 196–201, Sep. 2015.
[16] D. Patterson, et al. Intelligent RAM (IRAM): chips that remember and compute. In *ISSCC*, 1997.
[17] E. Riedel, et al. Active disks for large-scale data processing. *Computer*, 34(6), 2001.
[18] R. Nair, et al. Active Memory Cube: A processing-in-memory architecture for exascale systems. *IBM J Res Dev.*, 59(2/3):17:1–17:14, 2015.
[19] D. Patterson, et al. A case for intelligent RAM. *IEEE Micro*, 17(2), 1997.
[20] M. Hu, et al. Dot-Product Engine for Neuromorphic Computing: Programming 1T1M Crossbar to Accelerate Matrix-Vector Multiplication. In *DAC, 2016*, 2016.
[21] A. Graves, et al. Neural Turing Machines. *CoRR*, abs/1410.5401, 2014.
[22] A. Bremler-Barr, et al. Encoding Short Ranges in TCAM Without Expansion: Efficient Algorithm and Applications. *IEEE/ACM Transactions on Networking*, 26(2):835–850, April 2018.
[23] J. Wawrzynek. Eecs150 - digital design. http://www-inst.eecs.berkeley.edu/~cs150/sp12/agenda/lec/lec21-db3.pdf, 2012. [Online; accessed 15-Sept-2019].
[24] A. Tyagi. A reduced-area scheme for carry-select adders. *IEEE Transactions on Computers*, 42(10):1163–1170, 1993.
[25] H. Sato, et al. I/O chunking and latency hiding approach for out-of-core sorting acceleration using GPU and flash NVM. In *2016 IEEE International Conference on Big Data, BigData 2016, Washington DC, USA, December 5-8, 2016*, pages 398–403, 2016.