HDnn-PIM: Efficient in Memory Design of Hyperdimensional Computing with Feature Extraction

Arpan Dutta
UC San Diego
adutta@ucsd.edu

Saransh Gupta
IBM Research
saransh@ibm.com

Behnam Khaleghi
UC San Diego
bkhaleghi@ucsd.edu

Rishikanth Chandrasekaran
UC San Diego
r3chandr@ucsd.edu

Weihong Xu
UC San Diego
wexu@ucsd.edu

Tajana Rosing
UC San Diego
tajana@ucsd.edu

Abstract

Brain-inspired Hyperdimensional (HD) computing is a new machine learning approach that leverages simple and highly parallelizable operations. Unfortunately, none of the published HD computing algorithms to date have been able to accurately classify more complex image datasets, such as CIFAR100. In this work, we propose HDnn-PIM, that implements both feature extraction and HD-based classification for complex images by using processing-in-memory. We compare HDnn-PIM with HD-only and CNN implementations for various image datasets. HDnn-PIM achieves 52.4% higher accuracy as compared to pure HD computing. It also gains 1.2% accuracy improvement over state-of-the-art CNNs, but with 3.63× smaller memory footprint and 1.53× less MAC operations. Furthermore, HDnn-PIM is 3.6×–223× faster than RTX 3090 GPU, and 3.7× more energy efficient than state-of-the-art FloatPIM [5].

CCS Concepts

• **Hardware**; • **Computer systems organization**;

Keywords

Hyperdimensional Computing, Processing-in-Memory, CNN, RRAM

ACM Reference Format:

Arpan Dutta, Saransh Gupta, Behnam Khaleghi, Rishikanth Chandrasekaran, Weihong Xu, and Tajana Rosing. 2022. HDnn-PIM: Efficient in Memory Design of Hyperdimensional Computing with Feature Extraction. In *Proceedings of the Great Lakes Symposium on VLSI 2022 (GLSVLSI '22), June 6–8, 2022, Irvine, CA, USA.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3526241.3530331

1 Introduction

Hyperdimensional (HD) computing [8, 16] is an emerging machine learning paradigm that has shown impressive efficiency gains in IoT domain benchmarks. HD encodes data into high-dimensional space, where it can apply simple logic and arithmetic operations on the hypervectors to carry out learning tasks. The simplicity of HD operations, its inherent parallelism, combined with robustness to noise [9, 11], makes it particularly appealing for learning at the edge. One of the key challenges of HD computing is its inability to get accurate results for more complex image analysis.

GLSVLSI '22, June 6–8, 2022, Irvine, CA, USA.

ACM ISBN 978-1-4503-9322-5/22/06.
https://doi.org/10.1145/3526241.3530331

Table 1: Accuracy comparison of HD, CNN, and HDnn.

Model↓ Dataset→	MNIST	CIFAR10	CIFAR100	Flowers
HD (RP) [7]	94%	26.9%	9%	19.6%
HD (non-linear) [18]	97%	45.5%	27.7%	31.5%
StocHD [13]	98%	N/A	N/A	N/A
CNN ([4])	99%	94.6%	78.7%	84.7%
HDnn ([4]-based FE)	99%	95.1%	78.3%	88.8%

Table 1 shows that the state of the art HD computing algorithms [7, 18] get poor accuracy for all but the simplest dataset, MNIST. Even more recently published work that includes feature extraction into HD [13] only provides MNIST results. To address this issue, we propose HDnn (HD and Neural Network), which leverages a few initial stages of the convolution-based feature extraction to enable HD to learn effectively on complex data, as show in Fig. 1. As Table 1 shows, HDnn is the only HD-based model that achieves state-of-the-art accuracy on more complex datasets such as Flowers, CIFAR10 and CIFAR100, while using 3.63× less memory vs. CNN [4]. Table 2 shows the accuracy-size trade off of cutting layers off the original CNN and adding HD classifier (ResNet34 baseline: [3, 4, 6, 3], accuracy: 77.41%, MAC: 1161.5M, Param: 21.33M).

There are a few recent publications that combine neural networks with HD computing. Early work on voice recognition shows that using a neural network after HD classification results in a smaller and just as accurate design as if only a neural network is used [6]. Another example is [12] that uses neural network feature extractors and classifiers for simple, non-image datasets, such as ISOLET at the significant increase in the overall area due to using two different neural networks, HD encoder and decoder. A few publications combine Spiking Neural Networks (SNN) with HD for event-based data classification[17], but again show results on only the simplest datasets such as MNIST. Moreover, these works can be orthogonal to HDnn as they focus on event/spike based data.

Quite a few publications have accelerated deep neural networks using processing-in-memory [1, 5, 14]. The majority of these work employ large and hard-to-scale mixed-signal circuits such as ADCs and DACs to achieve low latency operations [1, 14] which leads to a chip with a limited memory capacity incapable of processing large networks. *Digital PIM* uses logical operations on digital signals stored in memory cells to carry out computation [5, 15]. Such an architecture retains high memory density at the expense of slower atomic computations, which is less of a problem for HD due to its simple operations. Thus, we choose *digital in-memory* design for HDnn and compare it to state of the art digital PIM CNN [5].

Table 2: ResNet34-based feature extractor on CIFAR100.

HDnn	Accuracy	MAC (M)	Parameter (M)
[3, 4, 6, 0]	77.07% (−0.34%)	951.6 (−18.1%)	8.56 (−59.9%)
[3, 4, 6, 1]	78.30% (+0.89%)	1010.4 (−13.0%)	12.29 (−42.4%)
[3, 3, 3, 1]	77.33% (−0.08%)	708.1 (−39.0%)	8.45 (−60.4%)

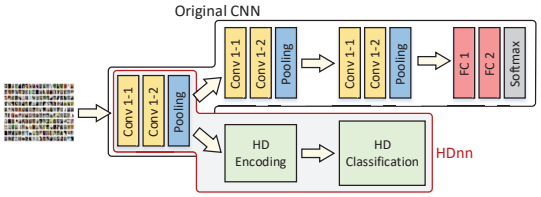

Figure 1: HDnn uses a few of the first convolution layers for feature extraction.

In summary, the major contributions in this work are:
(1) For the first time, we combine lightweight convolution-based feature extractor with HD computing for accurate yet efficient classification of complex data types.
(2) We propose a configurable PIM-based architecture to accelerate HDnn, which can run in HD- and DNN-only modes. HDnn-PIM achieves high throughput by allocating compute memories statically to layers based on their compute requirements. This flexibility is enabled by a novel 2-dimensional bus interconnect that leverages the locality of internal data transfers and reduces the wire length.
(3) HDnn-PIM innately supports residual connections found in new DNNs such as ResNet and Mobilenet, which alleviates the access to off-chip memory. It reduces the load on the bus for residual networks while also avoiding pipeline stalls (due to off-chip memory accesses) with a small area overhead.
(4) We identify the accumulation as a performance bottleneck of PIM architectures used for HD and DNNs. Accordingly, we propose a novel Compute Element (CE) that efficiently performs MAC operations by hiding their latency with memory read out. Accumulation in memory takes at least 41% of total MACs time given the fully parallel multiplication in memory. We reduce it to 'bitwidth number of cycles' in the CMOS domain using bit-serial Carry-Save Adder, leading to 1.66× faster MACs. HDnn-PIM's CE is also suited for HD, where it boosts HD's encoding by 5× speedup over [7].
(5) We design a Post Processing Network (PPN) that speeds up the accumulation for different (large or small) number of inputs after MAC operation (e.g., for on-the-fly accumulation or bypass of tiles outputs) by up to 4× compared to the typical in-memory addition.

2 HDnn Algorithmic Flow

Fig. 1 compares the HDnn and conventional CNN structure. HDnn comprises three steps: (1) realizing a feature extractor, (2) training of HD classifier, and (3) tuning the feature extractor.

(1) Feature extractor (FE): For FE, we rely on well-devised networks such as VGG, ResNet, and MobileNet. We cut these networks after a certain pooling layer and use the first convolution layers as FE. We extract the latent space representations by passing training images forward through the FE, which act as features for the HD encoder. The output of FE, per each image, is a manifold feature maps. We flatten it to a one-dimension feature vector using global averaging, which results in a compressed (usually 512 or 256, the same as the number of filters of the cutting layer) vector.

(2) HD training: The extracted input features act as training data for HD, which first encodes the data to high-dimensional space, e.g., to $\mathcal{D}$=4,000 dimensions. We use random projection (RP) [7] as it can be transformed to a matrix-vector multiplication as Eq. (1), where $\mathcal{P}$ is a $\mathcal{D} \times d$ binary projection matrix (d is the number of features per input), and $\mathcal{F}_d$ is the input feature extracted for an image. $\mathcal{H}$ is the binary encoding hypervector of an image.

$$H_{\mathcal{D}} = \texttt{sign}(\mathcal{P}_{\mathcal{D}\times d} \times \mathcal{F}_d) \quad (1)$$

Finally, HD adds all hypervectors of the same label to create the class of that label (e.g., 10 classed for CIFAR10 dataset) simply using $C^{\ell} = \sum_{i \in \ell} \mathcal{H}_i^{\ell}$. Inference is realized by comparing/searching the encoding hypervectors with the class hypervectors using cosine similarity or Hamming distance.

(3) FE tuning: For FE, we use pre-trained CNN networks that are not targeted for a classifier such as HD, especially when are cut. Therefore, after attaching HD to FE (see Fig. 1), we keep the HD fixed but retrain the FE to calibrate according to HD. The RP encoding of Eq. (1) uses `sign` which does not converge during backpropagation of retraining, hence, we replace it with `tanh` when training FE. After training HD, we quantize the class hypervectors to four bits without accuracy loss, especially the tuning step of FE adjusts based on the existing fixed HD.

As previously shown in Table 1, while even a more sophisticated HD encoding (non-linear, which uses floating-point projection matrix) has 56.7% less average accuracy than CNN, HDnn improves the accuracy by 1.34% with smaller number of parameters and operations, which will be further elaborated in Section 4.

3 HDnn-PIM Architecture

3.1 Mapping HDnn on HDnn-PIM Architecture

Fig. 2 demonstrates the overall data flow and mapping of HDnn on HDnn-PIM. We compile the given network to find out the mapping details of the layers on the supertiles of HDnn-PIM. The architectural details are elaborated later in Section 3.2. We assign supertiles based on the compute requirements of a given layer to balance and hence maximize the throughput. We divide the input image of the layer into multiple smaller images that are split across the supertiles allocated to that layer. Each supertile has the same coordinates among the input's feature maps to process the outputs independently. The weights are preloaded into the supertiles before feeding the inputs. As shown in Fig. 2❶, the weights are copied across the supertiles of a layer.

Inside a supertile, inputs are reused among the tiles while the weights are distributed among the tiles (Fig. 2❷). A tile implements either convolution of the given input and filter, or matrix-matrix multiplication. Inside each tile there are CEs that perform the actual computations, i.e., the matrix-matrix multiplication breaks down into matrix-vector operation inside the CEs. According to Fig. 2❸, the input matrix is split into rows, where each CE takes care of a row, but the weight matrix is copied into all the CEs of a tile. As shown in Fig. 2❸ and ❹, each CE uses column-parallel multiplication in memory and bit-serial parallel CMOS accumulation to produce the partial sums or outputs. To perform the matrix-vector efficiently, the input is replicated and the weight matrix is flattened to realize it as a vector-vector multiplication, suited for digital PIM.

The FE module of HDnn-PIM generates the output features that are flattened, average pooled and forwarded as inputs to the HD module. The flattened input vector is multiplied by a binary projection matrix to realize encoding. This matrix-vector multiplication is performed in an array of CEs as described above. Based on whether the HD is in training or inference mode, the appropriate operation (i.e., accumulation with class hypervectors or search) is carried out.

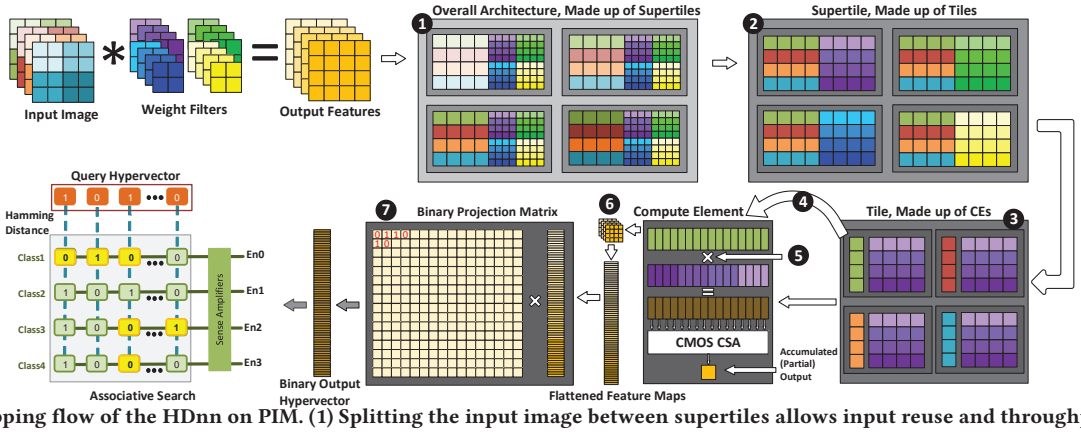

Figure 2: Mapping flow of the HDnn on PIM. (1) Splitting the input image between supertiles allows input reuse and throughput balancing, and avoids communication between the supertiles that implement the same layer. (2) Tiles of a supertile reuse/share the input on different filters for parallel output production and can perform matrix-matrix multiplication. (3) Compute Elements of a tile split the inputs and reuse the weights to generate output pixel and can perform matrix-vector multiplication. (4) Replication of inputs to perform matrix-vector multiplications for all rows of a matrix in parallel. (5) Column-parallel multiplication in memory. (6) Output of the supertiles passed to the final layer. (7) Random projection encoding using CE array to implement matrix-vector multiplication.

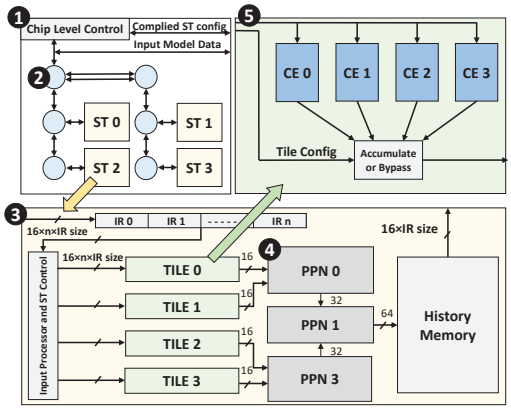

Figure 3: HDnn-PIM tiled architecture. ST: Supertile; PPN: Post Processing Network; CE: Compute Element.

3.2 HDnn-PIM Architecture

Fig. 3 shows the HDnn-PIM's tiled architecture. The tiles are grouped into supertiles that allows input reuse among tiles and increases the supported input channels and filter size. Fusion of the tiles becomes necessary when a tile cannot hold all the data necessary to generate an output feature (i.e., the $k \times k \times n_i$ filter and the corresponding convolution window on the input). At most, all the tiles of a supertile can fuse to support a maximum input size, governed $k \times k \times n \leq c \times n_{CE} \times n_{tile}$, where k is the filter size, n_i is the number of input channels, c is the memory columns per CE, n_{CE} is the number of CEs per tile, and n_{tile} is the number of tiles per supertile. A supertile has the following components.

Input Registers (IR) act as primary inputs to the tiles in the supertile. IRs are divided into sets (IR 1 to IR n in Fig. 3❸) where each set delivers a convolution window of a input channel to the tiles (tiles share the same inputs). The maximum window that can be written to the tile at an instance is equal to the size of the sets. Larger windows may need multiple passes to write a window. The number of input channels that can be simultaneously written is equal to the number of sets.

Input Processor (IP) in a supertile takes the IRs as inputs and generates memory addresses to fill the CE memories. Based on the configuration of the supertile, IP concatenates IR data into rows of 1,024 elements that can be written in parallel to the CE memories. Row addresses are generated such that enough space is left for in-memory multiplication. If the layer mapped to the supertile is preceded by pooling, the IP produces addresses to store the corresponding inputs below one another for in-memory pooling.

Post Processing Network (PPN) enables the fusion of tiles, and routes the tiles output to the appropriate supertile input (of the next layer). PPN comprises an activation unit and an accumulator, so applies activation functions to the accumulated outputs of the tiles, as well. It connects the tiles in a binary tree structure as shown in Fig. 3❹. The accumulations in PPNs depends on the number of fused tiles. Fusing happens when the input channels required to produce an output pixel span multiple tiles due to memory limit. The tiled architecture allows parallel accumulation/activation when the input sizes are small and multiple of them fit in a tile.

History Memory (HM) is an additional 1,024×1,024 memory block that stores the history of the outputs, and is required because of the fine-grained computation by the compute pipeline. In a pipelined architecture, the current layer needs to provide the input window of the next layer. Hence, in a sliding window mechanism, the next layer will be lagging by at least one column worth of convolution windows. Therefore, this history needs to be stored so the the next layer can accomplish its window. This storage increases when a residual connection is introduced, where a layer may send data several layers ahead and requires to match the exact latency of the data that is passing through all the intermediate layers to reach the residue destination. Thus, this storage becomes indispensable to avoid large power hungry register banks at the output.

Feature Extraction (FE) tile has multiple CEs that perform MACs in parallel. Each tile is equipped with its own top level accumulator to accumulate the partial sums of all the CEs (this is different from the CSA adder of each CE directly connected to the memory block). The output of each CE is one accumulated value. This allows us to avoid data dependence between CEs and keep the tile architecture simple. We consecutively store a convolution window (of all the channels) associated with an output feature, and then move on to store the window corresponding to the next output feature. This allows of maximum use of the available memory size.

The HDnn-PIM architecture is designed such that supertiles that process the same layer have no data dependency with each

other. This increases the throughput of processing a given layer by avoiding data transfers between supertiles that all compute a certain layer. All supertiles of a layer work in parallel.

3.3 Supertile Data Management

HDnn-PIM architecture statically assigns supertiles to different layers (to be executed pipelined) proportionally based on the compute opportunity they present. The bus interconnect establishes data transfer based on the destination information of each supertile. Data transfer for the next computation is exerted simultaneously with the current computation to throttle the cost of data transfer.

Supertile Mapping: For the mapping of layers on supertiles, the compiler takes the network specifications as input and generates the configuration of HDnn-PIM. The key aspects taken into consideration for a convolution layer for assignment are the filter size k, number of input/output channels n_i/n_o, input dimensions d_I, and strides along the x and y directions (s_x and s_y). These parameters are used to calculate the compute intensity ($CI = \frac{d_I^2}{s_x \times s_y}$) and configure the supertiles accordingly. The compiler ensures the entire network fits into the CE memories. Otherwise, the network is processed as batches of layers. Parallelism can also be extracted by computing multiple convolution windows of the same input feature in parallel (i.e., multiple filters over multiple window, as in Fig. 2❶). The number of windows to be processed in parallel for a given layer is the ratio of its CI to the layer with minimum CI. The same procedure is used when the memories are limited. However, now one CE may hold multiple sets of inputs and weights. Multiplication is then performed sequentially for all sets in a CE. This avoids off-chip accesses at the cost of throughput. By using CI, the data rate mismatch due to different strides is also mitigated.

Pooling uses simple logic and arithmetic operations and is performed in memory. As explained for the Input Register (IR) functionality, the output of the layer preceding pooling is sent to the memories allocated to subsequent layer. The pooling windows is stored vertically in the memory columns of the next layer. Once the output is obtained, the inputs are no longer required, so the intermediate storage of the in-memory multiplications are released for MAC operations. Large pooling windows that cannot fit in the memory can be processed in multiple iterations.

Residual connections are challenging due to data transfer to multiple layers ahead, which requires the latency of the computation to be matched such that the right inputs are added before activation. This is realized by storing larger histories in the HM for layers that are the producer of the residue and feed the values as required to the consumer. The producer lag is fixed based on the specification of the intermediate layers and the supertile mapping.

Bus Interconnect (BI), shown in Fig. 3❷, acts as the backbone of HDnn-PIM that enables all the mapping of supertiles and pipelining between the network layers. The flexibility of using CI to map layers to the supertiles can only be achieved if all the supertiles can communicate with the other supertiles. Each supertile has an ID, and each register set in a supertile has a local register ID. This is used to uniquely identify each supertile and its register sets. After mapping all the layers to the supertiles, the compiler generates a source and a destination ID for each supertile. When data is being sent by a supertile, this metadata is used by the BI to route it to the proper destination. Each node of the BI acts as a switch that compares the destination ID to the threshold of that node (e.g., IDs 0–31 for the first branch and so on) and routes the data to the appropriate branch. Once the data reaches the destination, the supertile uses the register ID to store the value in the right register set.

Dual bus structure is used because for larger designs the resources of such structure scale linearly due to the type of data transfer involved. Due to the natural staggering of computation among different layers, the memory blocks assigned to the same layer do not receive and send data at the same time. This significantly reduces the maximum bandwidth requirement of the top bus (limiting it to the transfer of the largest layer currently mapped).

3.4 Mapping HD to HDnn-PIM Architecture

HD Encoding in HDnn-PIM: HD encoding is implemented as a matrix (binary)-vector multiplication (VMM). The core CE remains the same as for the FE. The data access patterns in VMM and convolution are different. Thus, we use different FE and HD units on the chip to simplify the FE control logic and mapping. For simple VMM, the tile hierarchy inside the supertiles, the BI, and the HM are not required. We create multiple copies of the input feature vector (output of the cutting layer) that are stored in different supertiles, same as the weights of the FE. After multiplication in memory and accumulating all the products in the supertile, one dimension of the encoding hypervector is generated. The number of outputs that can be generated concurrently is equal to the the number of supertiles of the HD component, which is configured to match the FE throughput. The results of the VMM (encoding) is stored in a separate supertile. The dimensions of the hypervector, which are multi-bit, are then reduced to 1-bit using thresholding.

HD Inference and Training in HDnn-PIM: For inference, we encode the vector using the above architecture to obtain the query hypervector. The class is determined using search in memories by comparing the query hypervector to the stored class hypervectors. This associative search can be performed in-memory as demonstrated in [2]. For HD training all class hypervectors are obtained through FE and encoding followed by accumulating the vectors of the same class. Therefore, we get one vector per image from the FE, that we encode and add it to the class vector to train the HD classifier. We obtain all the class hypervectors once all the images have passed through this procedure. For retraining, we perform inference on each image in the training set and compare the predicted class with the expected class. If they do not match, we add the query hypervector to the expected class's hypervector and subtract it from the predicted class's hypervector. The location of the class hypervectors remain the same memory blocks where they were first stored. All training additions are performed in memory.

3.5 HDnn-PIM Compute Element

The CE is the lowest level component of HDnn-PIM, comprising a 1,024×1,024 single-bit ReRAM crossbar capable of in-memory arithmetic. Fig. 4 shows the organization of data in the memory crossbar. Two sets of 1,024 16-bit numbers are stored in adjacent columns, where each number is stored vertically in a single column and 16 rows. The first (second) 16 rows store the first (second) set. The two sets act as multipliers and multiplicands that are being multiplied down the column to produce the product. Therefore, all 1,024 multiplications are performed in parallel. The two inputs, for instance, could be the weights and input feature windows that need

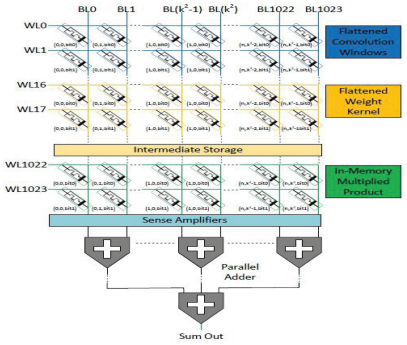

Figure 4: CE crossbar and associated bit-serial parallel CSA adder.

to be multiplied for convolution. The two inputs and the product together occupy 48 rows of the crossbar. The remaining (976) bits are used to store another input or act as scratchpads that store the intermediate signals that are generated during the product.

Another component of the CE is a bit-serial parallel CSA adder. The in-memory addition does not allow this operation optimally since, first, all the columns of the array perform a multiplication and no space is left along the rows (where the products are stored) to perform accumulation. Second, if we change the data arrangement to carry out in-memory accumulation, it is not possible to perform 1,024 16-bit accumulation in parallel in one crossbar. To maintain this parallelism, the bit-serial CSA adder is used. Since the products are stored along the columns, a bit of the same significance of all the numbers can be read out simultaneously and accumulated. This operation is performed 16 times using the CSA adder to produce the accumulated output. Such PIM-CMOS hybrid structure allows the architecture to extract high parallelism by performing more multiplications, which is the most expensive operation. In addition, accumulation is also performed in parallel while reading the data out. Hence, it is hidden within the memory read out latency.

4 Evaluation

4.1 Experimental Setup

We implemented an operation-level simulator in Python to analyze the HDnn-PIM which models its architecture, considering the size of operations, data mapping, memory size, and network parameters. The simulator uses the performance and energy consumption values obtained from circuit-level evaluations in 45 nm technology in Cadence Virtuoso. The memory cell characteristics are derived from VTEAM memristor model [10]. We calibrate the model to represent the device characteristics used in [5]. The resultant memory cell has R_{OFF} and R_{ON} as 10 MΩ and 10 kΩ respectively, with a device switching delay of 1.1 ns (PIM's design cycle time).

In our experiments we used CIFAR10/100 and Flowers datasets on VGG-16, MobileNetV2, and ResNet-18/34 networks. HDnn truncates the networks and uses as feature extractors (FEs), which is followed by a HD-based classifier with $\mathcal{D}$=4,000 dimensions. We compare HDnn-PIM with the state-of-the-art PIM accelerators FloatPIM [5] (digital) and ISAAC [14] (analog). We also compare the HDnn-PIM performance with Intel Xeon Gold 6140 CPU and Nvidia RTX 3090 GPU using PyTorch implementation of the HDnn.

4.2 HDnn Accuracy Analysis

For feature extraction with VGG-16, we cut at layer L17 (out of 44 layers including batch normalization). For ResNet-18, we cut off the last fully connected layer and six preceding convolution layers.

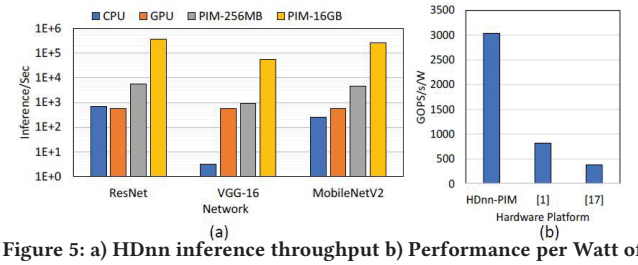

Figure 5: a) HDnn inference throughput b) Performance per Watt of HDnn-PIM vs. FloatPIM[5] and ISAAC[14]

We cut the MobileNetV2 layers after the fourth bottleneck layer but we preserve its last fully connected layer. All analyses are done for inference using models trained with 16-bit fixed point representation. Our analysis shows that HDnn is on average 51.0%, 49.1%, and 57.3% more accurate than the *mere HD model* on CIFAR100, CIFAR10, and Flowers datasets, respectively (see Table 1).

Fig. 6(a) shows the HDnn accuracy, and Fig. 6(c), and 6(d) show the MAC and parameter reduction of HDnn using the aforementioned trimmed CNNs as the feature extractor, compared to the original CNN. HDnn increases the accuracy by 1.2% on average compared to CNN models for image classification, while reducing the number of MACs and model parameters by 34.8% and 72.5% respectively. Note that we also compared HDnn with the CNN that is cut at the same layer followed by a fully connected layer (instead of HD). Compared to such trimmed ResNet-18 configuration, HDnn achieves 1.8% higher accuracy on CIFAR10 dataset, indicating the effectiveness of HD to gain better insight from data.

4.3 HDnn-PIM vs State of the Art

CPU and GPU: Fig. 5(a) shows the inference throughput of HDnn models on different platforms. Table 3 summarizes the speedup and energy efficiency gains over GPU relative to other state of the art works (HD only model and models without references are run on HDnn-PIM). Our high performance design (PIM-16GB) achieves 223× higher throughput than RTX 3090 GPU, while consuming less memory than the GPU which has 24GB of RAM. The area efficient version (PIM-256MB) is yet 3.6× better than GPU. Comparing with CPU, our high performance (area-efficient) design achieved 13,796× (219×) higher throughput.

Previous PIM designs: None of the existing PIM-based HD accelerators evaluate their design on complex image datasets such as CIFAR100 or Flowers [3, 9], instead focusing primarily on MNIST. They either use an inferior HD encoding as compared to the baseline encoding used in HDnn [3, 9] or do not evaluate on complex image datasets [13]. Hence, HDnn-PIM is at least 52.4% more accurate on average than existing HD-PIM designs for Flower, CIFAR10, and CIFAR100 datasets (Section 4.2).

We compare HDnn-PIM with digital-PIM FloatPIM DNN accelerator [5], and analog-PIM ISAAC [14]. We use VGG-16 for comparison since these works are not capable of running networks with residual connections. Our evaluations show that HDnn-PIM has an energy-efficiency of 3,036 GOPS/s/W for 16-bit operations, which is 3.7× higher than FloatPIM [5] (Fig. 5(b)). By balancing throughput,

Table 3: Performance-Power normalized to GPU RTX 3090.

Model	Speedup	Energy Efficiency
HD (RP) [7]	133x	1215x
CNN (VGG-16)	84x	1202x
CNN (VGG-16) [5]	100x	325x
HD+FE [13]	14x	259x
HDnn (VGG-16-based FE)	104x	1213x

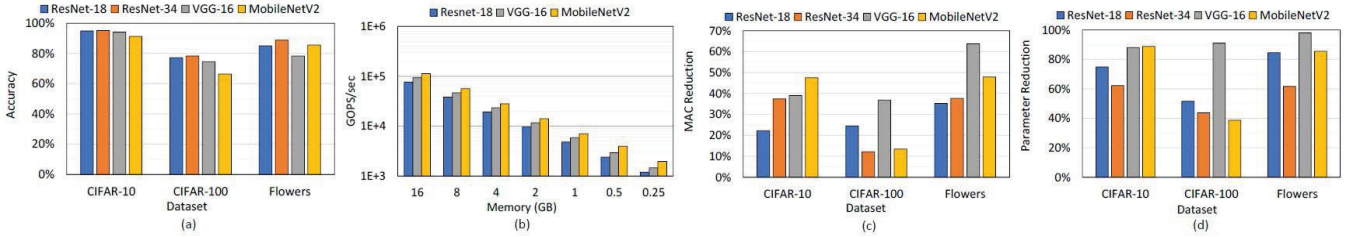

Figure 6: (a) HDnn accuracy for different datasets leveraging different CNNs as FEs, (b) Performance scaling of HDnn-PIM, (c) HDnn MAC reduction using different FEs, (d) HDnn parameter reduction using using different FEs for various datasets.

HDnn-PIM ensures optimum use of the given resources. HDnn-PIM has per-area performance of 275.5 GOPS/s/mm^2 which is 8.2% (7.7×) less than FloatPIM's low-power (high-power) version. This is because, unlike FloatPIM, we consider the data-flow and mapping overheads of all DNN layers in HDnn-PIM. This includes, but is not limited to, computation and data mapping requirements in case of pooling, residual connections, and larger strides. HDnn-PIM is 7.9× more energy-efficient than ISAAC [14] as it does not use power-hungry mixed-signal circuits such as DAC/ADCs. Since ISAAC merely performs convolution and does not handle other DNN layers, it is 1.7× better in per-area performance.

4.4 HDnn-PIM Performance-Area Tradeoff

We analyse the impact of scaling the memory size by considering memory sizes ranging from 256 MB to 16 GB using the same HDnn networks described in Section 4.2. As shown in Fig. 6(b), the performance increases linearly with the available memory. This indicates the scalability of HDnn-PIM to fit multiple networks on different configurations without impacting performance drastically.

We also discuss the performance-efficient and area-efficient versions of HDnn-PIM. The performance-efficient design has 8,192 supertiles with 16 tiles each (total 16 GB). All multiplication operations in the pipeline happen in parallel which provides high computation throughput. For the area-efficient design, we use lower memory size which consumes lower power due to reduced parallelism. We choose a design with 128 supertiles with 16 tiles each (total 256 MB), in which we perform compact data mapping and store multiple rows of weights and inputs in a CE. This results in less parallel multiplications but can fit the network on a significantly smaller chip. Our evaluations show that the performance-efficient (area-efficient) HDnn-PIM has a throughput of 93.3 TOPS/s (1.5 TOPS/s) with an end-to-end latency of 0.14 ms (8.9 ms) for VGG-16, 76.7 TOPS/s (1.2 TOPS/s) with 0.03 ms (2.1 ms) latency for ResNet-18, and 112.3 TOPS/s (1.9 TOPS/s) with 0.13 ms (7.6 ms) latency for MobileNetV2. The higher latency of MobileNet versus ResNet-18 is due to its 1×1 convolutions that make the memory blocks underutilized. Also, MobileNet has higher image size reduction due to the pooling and stride-2 convolutions, requiring more resources for throughput balancing (e.g., a 2×2 pooling requires four times of resources to be allocated to the previous layer).

5 Conclusion

We proposed HDnn, a novel approach of feature extraction for HD computing for complex datasets such as images, and HDnn-PIM, the first PIM based HDnn accelerator. We evaluated HDnn on a variety of feature extractors and datasets. Our evaluation shows that HDnn achieves 52.4% higher accuracy as compared to HD computing without feature extraction. It also gains 1.2% accuracy over state-of-the-art CNNs, but at 72.46% (3.63×) lower memory footprint and 34.83% (1.53×) fewer MACs. Furthermore, HDnn-PIM is 223× faster than RTX 3090 GPU and 3.7× more energy efficient than the state-of-the-art PIM accelerator for DNNs.

Acknowledgements

This work was supported by TSMC, in part by CRISP, one of six centers in JUMP (an SRC program sponsored by DARPA), SRC Global Research Collaboration (GRC) grant, and NSF grants #1911095, #1826967, #2100237, and #2112167.

References

[1] Ping Chi, Shuangchen Li, et al. 2016. Prime: A novel processing-in-memory architecture for neural network computation in reram-based main memory. *ACM SIGARCH Computer Architecture News* 44, 3 (2016), 27–39.
[2] Amirali Ghofrani et al. 2016. Associative memristive memory for approximate computing in gpus. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems* 6, 2 (2016), 222–234.
[3] Saransh Gupta, Mohsen Imani, and Tajana Rosing. 2018. Felix: Fast and energy-efficient logic in memory. In *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 1–7.
[4] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep Residual Learning for Image Recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.
[5] Mohsen Imani, Saransh Gupta, Yeseong Kim, and Tajana Rosing. 2019. Floatpim: In-memory acceleration of deep neural network training with high precision. In *46th International Symposium on Computer Architecture (ISCA)*. IEEE, 802–815.
[6] Mohsen Imani, Deqian Kong, Abbas Rahimi, and Tajana Rosing. 2017. VoiceHD: Hyperdimensional Computing for Efficient Speech Recognition. In *2017 IEEE International Conference on Rebooting Computing (ICRC)*. 1–8.
[7] Mohsen Imani, Justin Morris, et al. 2019. Bric: Locality-based encoding for energy-efficient brain-inspired hyperdimensional computing. In *56th Annual Design Automation Conference*. 1–6.
[8] Pentti Kanerva. 2009. Hyperdimensional computing: An introduction to computing in distributed representation with high-dimensional random vectors. *Cognitive computation* 1, 2 (2009), 139–159.
[9] Geethan Karunaratne, Manuel Le Gallo, Giovanni Cherubini, Luca Benini, Abbas Rahimi, and Abu Sebastian. 2019. In-memory hyperdimensional computing. *arXiv preprint arXiv:1906.01548* (2019).
[10] Shahar Kvatinsky, Misbah Ramadan, et al. 2015. VTEAM: A general model for voltage-controlled memristors. *IEEE Transactions on Circuits and Systems II: Express Briefs* 62, 8 (2015), 786–790.
[11] Justin Morris, Kazim Ergun, et al. 2021. HyDREA: Towards More Robust and Efficient Machine Learning Systems with Hyperdimensional Computing. In *2021 Design, Automation Test in Europe Conference Exhibition (DATE)*. 723–728.
[12] Mahdi Nazemi, Amirhossein Esmaili, et al. 2020. SynergicLearning: Neural Network-Based Feature Extraction for Highly-Accurate Hyperdimensional Learning. In *International Conference On Computer Aided Design (ICCAD)*. 1–9.
[13] Prathyush Poduval, Zhuowen Zou, Hassan Najafi, et al. 2021. StocHD: Stochastic Hyperdimensional System for Efficient and Robust Learning from Raw Data. In *ACM/IEEE Design Automation Conference (DAC)*. 1195–1200.
[14] Ali Shafiee, Anirban Nag, et al. 2016. ISAAC: A convolutional neural network accelerator with in-situ analog arithmetic in crossbars. *ACM SIGARCH Computer Architecture News* 44, 3 (2016), 14–26.
[15] Nishil Talati, Saransh Gupta, Pravin Mane, and Shahar Kvatinsky. 2016. Logic design within memristive memories using memristor-aided loGIC (MAGIC). *IEEE Transactions on Nanotechnology* 15, 4 (2016), 635–650.
[16] Anthony Thomas, Sanjoy Dasgupta, and Tajana Rosing. 2020. Theoretical Foundations of Hyperdimensional Computing. *arXiv:2010.07426* (2020).
[17] Zhuowen Zou, Haleh Alimohamadi, Farhad Imani, Yeseong Kim, and Mohsen Imani. 2021. Spiking Hyperdimensional Network: Neuromorphic Models Integrated with Memory-Inspired Framework. arXiv:2110.00214 [cs.NE]
[18] Zhuowen Zou, Yeseong Kim, M. Hassan Najafi, and Mohsen Imani. 2021. ManiHD: Efficient Hyper-Dimensional Learning Using Manifold Trainable Encoder. In *2021 Design, Automation Test in Europe Conference Exhibition (DATE)*. 850–855.