To appear in 2020 IEEE International Symposium on High Performance Computer Architecture (HPCA).

Enabling Highly Efficient Capsule Networks Processing Through A PIM-Based Architecture Design

Xingyao Zhang
University of Houston
Houston, USA
xzhang55@uh.edu

Shuaiwen Leon Song
University of Sydney
Sydney, Australia
leonangel991@gmail.com

Chenhao Xie
Pacific Northwest National Laboratory
Richland, USA
fenahuhu@gmail.com

Jing Wang
Capital Normal University
Beijing, China
jwang@cnu.edu.cn

Weigong Zhang
Capital Normal University
Beijing, China
5591@cnu.edu.cn

Xin Fu
University of Houston
Houston, USA
xfu8@central.uh.edu

ABSTRACT

In recent years, the CNNs have achieved great successes in the image processing tasks, e.g., image recognition and object detection. Unfortunately, traditional CNN's classification is found to be easily misled by increasingly complex image features due to the usage of pooling operations, hence unable to preserve accurate position and pose information of the objects. To address this challenge, a novel neural network structure called Capsule Network has been proposed, which introduces equivariance through capsules to significantly enhance the learning ability for image segmentation and object detection. Due to its requirement of performing a high volume of matrix operations, CapsNets have been generally accelerated on modern GPU platforms that provide highly optimized software library for common deep learning tasks. However, based on our performance characterization on modern GPUs, CapsNets exhibit low efficiency due to the special program and execution features of their routing procedure, including massive unshareable intermediate variables and intensive synchronizations, which are very difficult to optimize at software level. To address these challenges, we propose a hybrid computing architecture design named *PIM-CapsNet*. It preserves GPU's on-chip computing capability for accelerating CNN types of layers in CapsNet, while pipelining with an off-chip in-memory acceleration solution that effectively tackles routing procedure's inefficiency by leveraging the processing-in-memory capability of today's 3D stacked memory. Using routing procedure's inherent parallellization feature, our design enables hierarchical improvements on CapsNet inference efficiency through minimizing data movement and maximizing parallel processing in memory. Evaluation results demonstrate that our proposed design can achieve substantial improvement on both performance and energy savings for CapsNet inference, with almost zero accuracy loss. The results also suggest good performance scalability in optimizing the routing procedure with increasing network size.

1. INTRODUCTION

Recently, machine learning has bloomed into rapid growth and been widely applied in many areas, including medical [1, 2], security [3], social media [4], engineering [5] and etc. Such explosive development is credited to the success of the neural network algorithms, especially the convolutional neural networks (CNNs), which are extremely suitable for the image processing tasks, e.g., image recognition and object detection [6, 7, 8, 9, 10]. However, recent studies have found that CNNs could easily misdetect important image features when image scenarios are more complex. This could significantly affects the classification accuracy [11, 12, 13]. As shown in Fig.1, when attempting to identify the lung cancer cells, traditional CNNs are bounded to the confined region, thus missing critical regions around the cell edges and leading to the wrong identification. This is because the pooling operations used in common CNNs apply happenstance translational invariance [11, 14] which limits the learning of rotation and proportional change, resulting in obtaining only partial features. With the increasing adoption of emerging applications (e.g., medical image processing and autonomous driving) into humans' daily life that have strict requirements on object detection's accuracy, wrong identification could be fatal in some cases. To address these challenges, a novel neural network called Capsule Network (CapsNet) has been proposed recently [14]. Fig.1 demonstrates the evolution from classic CNN identification to CapsNet identification. CapsNet abandons the usage of pooling operations in CNNs and introduces the concept of *capsule* which is any function that tries to predict the presence and the instantiation parameters of a particular object at a given location. The figure illustrates a group of capsules, each with a double-featured activation vector (i.e., probability of presence and pose, shown as the green and red arrows in Fig.1). Because of this added equivariance, CapsNet can accurately detect the cancer cells via precise classification according to the cell edges and body texture. According to the re-

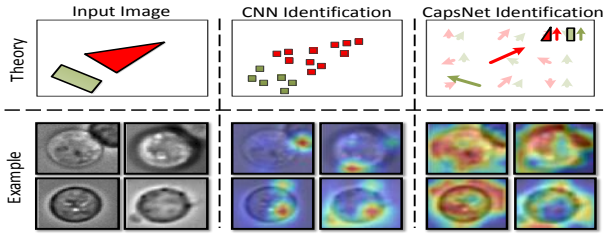

Figure 1: The comparison of neural networks in identifying lung cancer cells [17], where CapsNet outperforms the traditional CNN on detection accuracy. The heat maps indicate the detected features.

cent studies, CapsNets are increasingly involved in the human-safety related tasks, and on average outperform CNNs by 19.6% and 42.06% on detection accuracy for medical image processing[15, 16, 17, 18, 19, 20] and autonomous driving[21, 22, 23].

Because CapsNets execution exhibits a high percentage of matrix operations, state-of-the-art GPUs have become primary platforms for accelerating CapsNets by leveraging their massive on-chip parallelism and deeply optimized software library [24, 25]. However, processing efficiency of CapsNets on GPUs often cannot achieve the desired level for fast real-time inference. To investigate the root causes of this inefficiency, we conduct a comprehensive performance characterization on CapsNets' execution behaviors on modern GPUs, and observe that the computation between two consecutive capsule layers, called *routing procedure* (Sec.2.2), presents the major bottleneck. Through runtime profiling, we further identify that the inefficient execution of the routing procedure originates from (i) tremendous data access to off-chip memory due to the massive unshareable intermediate variables, and (ii) intensive synchronizations to avoid the potential write-after-read and write-after-write hazards on the limited on-chip storage. These challenges are induced by the unique features of the routing procedure execution, and cannot be addressed well via common NN optimization techniques [26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37] as well as software-level on-chip memory management (e.g., register manipulation or shared memory multiplexing).

To tackle CapsNets' significant off chip-memory access and intensive synchronization (induced by numerous aggregation operations in the routing procedure), we propose a processing-in-memory based hybrid computing architecture named *PIM-CapsNet*. At the highest level, PIM-CapsNet continues to utilize GPU's native on-chip units as the host for fast processing CNN-type of layers such as Convolution and Fully-connected included in CapsNet. Meanwhile, by leveraging batch execution, PIM-CapsNet pipelines the host GPU execution with an off-chip in-memory solution that can effectively accelerate CapsNet's routing procedure.

To enable the in-memory acceleration capability for the routing procedure (RP), we select one of the emerging 3D stacked technologies, i.e., hybrid memory cube (HMC)[38], for RP's design and integration. Because of HMC's high internal and external memory bandwidth, and its unique logic layer for easy integration of computation logic, it has become a promising PIM platform [39, 40, 41, 42, 43]. By leveraging HMC's unique architectural features, our PIM-CapsNet design mitigates the data-intensive RP into HMC to tackle its major challenges above. There are two key objectives for this in-memory design for RP: under the hardware design constraints, creating a hierarchical optimization strategy to enable (i) inter-vault workload balance and communication minimization (Sec.5.1 and 5.3) and (ii) intra-vault maximum parallel processing (Sec.5.2 and 5.3). Additionally, the in-memory optimizations should be generally applicable to different RP algorithms.

For objective (i), we further investigate RP's algorithm and identify an interesting feature: highly parallelizable in multi-dimensions. This makes RP's workloads have great potential to be concurrently executed across vaults without incurring significant communication overheads. We then create a modeling strategy by considering both per-vault workloads and inter-vault communication to guide dimension selection for parallelization, in order to achieve the optimal performance and power improvement. This also significantly reduces the original synchronization overheads through the conversion to aggregation within a vault. For objective (ii), we integrate multiple processing elements (PEs) into a vault's logic layer and explore the customized PE design to concurrently perform RP's specific operations across memory banks. Meanwhile, we propose a new address mapping scheme that effectively transfers many inter-vault level data requests to the intra-vault level and address the bank conflict issues of concurrent data requests. Furthermore, to reduce logic design complexity and guarantee performance, we use simple low-cost logic to approximate complex special functions with negligible accuracy loss. To summarize, this study makes the following contributions:

- We conduct a comprehensive characterization study on CapsNet inference on modern GPUs and identify its root causes for execution inefficiency.
- Based on the interesting insights from the characterization and further algorithm analysis, we propose a processing-in-memory based hybrid computing architecture named *PIM-CapsNet*, which leverages both GPU on-chip computing capability and off-chip in-memory acceleration features of 3D stacked memory to improve the overall CapsNet inference performance.
- To drastically reduce the identified performance bottlenecks, we propose several memory-level optimizations to enable minimal in-memory communication, maximum parallelization, and low design complexity and overhead.
- The experimental results demonstrate that for the overall CapsNet inference, our proposed *PIM- CapsNet* design outperforms the baseline GPU by 2.44x (upto 2.76x) on performance and 64.91% (upto 85.16%) on energy saving. It also achieves good performance scalability with increasing network size.

2. BACKGROUND: CAPSULE NETWORK

As shown in Fig.2, CapsNet inherits the convolutional (Conv) and fully connected (FC) layers from the standard CNNs, but introduces new layers (i.e., Caps layers) to realize the concept of "capsule" for better information representation. A capsule is a group of neurons (Fig.1)

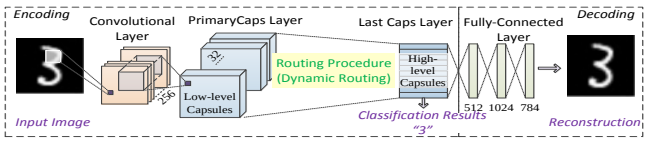

Figure 2: The computation diagram of CapsNet for MNIST [14].

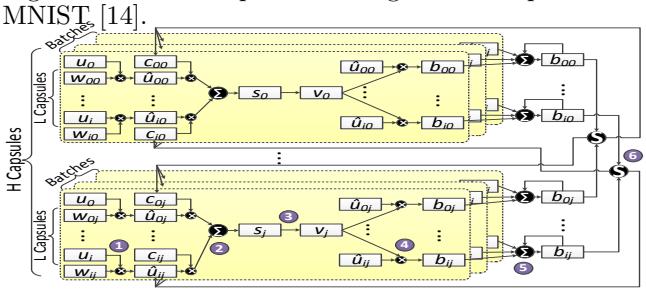

Figure 3: Dynamic Routing Procedure (RP) in CapsNet: describing computation flow and multiple ways for parallel processing.

whose activity vector represents instantiation parameters of a specific type entity (e.g., the location and pose of an object). It introduces equivariance which makes standard CNNs understand rotation and proportional change (e.g., in Fig.1). CapsNet significantly lifts the limitation of happenstance translational invariance of pooling operations applied in the traditional CNNs, thus being considered to be superior in image segmentation and object detection [11, 14].

2.1 CapsNet Structure

Fig.2 takes CapsNet-MNIST [14] as an example to illustrate a basic CapsNet structure. It contains two computation stages: encoding and decoding. The encoding stage is composed of Conv layers and Caps layers. The convolutional operations are first performed on the data mapping from neurons of the Conv layer to the capsules of the first Caps layer, which is defined as PrimeCaps layer. It is often followed by at least one other Caps layer. The data mapping between the capsules of the adjacent Caps layers is performed via the *routing procedure (RP)*. Finally, the last Caps layer produces the classification information towards categorization, with each capsule representing one category. Following the encoding stage, the decoding function includes multiple FC layers which attach to the last Caps layer for image reconstruction (i.e., improving model accuracy in training or plotting abstract features in inference) [14].

2.2 Essential Mechanism For Avoiding Feature Loss: Routing Procedure (RP)

The *routing procedure* (RP) is introduced to route the information from low-level capsules (or L capsules) to high-level capsules (or H capsules) without feature loss. There have been several routing algorithms used in routing procedure such as Dynamic Routing [14] and Expectation-Maximization routing[44]. In this work, we use the popular Dynamic Routing [14] as an example to explain the RP execution.

Fig.3 and Algorithm.1 demonstrate the computation flow and possible dimensions for parallelization. Given the kth batched input set, in order to generate jth H capsule, ith L capsule in this input set (u_i^k) first multiplies with the corresponding weight (W_{ij}) to generate the prediction vector ($\hat{u}_{j|i}$) (Fig.3 ❶):

$$\hat{u}^k_{j|i} = u^k_i \times W_{ij} \tag{1}$$

Then, these prediction vectors will multiply with their corresponding routing coefficients (c_{ij}) with results aggregated across all L capsules (Fig.3 ❷):

$$s^k_j = \textstyle\sum_i \hat{u}^k_{j|i} \times c_{ij} \tag{2}$$

The non-linear "squashing" function is then implemented on the aggregated results of s^k_j to produce the jth H capsule v_j (Fig.3 ❸):

$$v^k_j = \frac{||s^k_j||^2}{1+||s^k_j||^2} \frac{s^k_j}{||s^k_j||} \tag{3}$$

Note that the v^k_j can not be considered as the final value of jth H capsule unless the features of L capsules have been correctly inherited. The information difference between an L and H capsule can be quantified by the agreement measurement via the scalar production of the prediction vector $\hat{u}^k_{j|i}$ and the H capsule v^k_j (Fig.3 ❹), where the "0" output means the information is precisely inherited. In the case of a large divergence, the agreements will be accumulated into an intermediate variable b_{ij} (Fig.3 ❺), which will be used to update the routing coefficients via the "softmax" function (Fig.3 ❻).

$$b_{ij} = \textstyle\sum_k v^k_j \hat{u}^k_{j|i} + b_{ij} \tag{4}$$

$$c_{ij} = \frac{exp(b_{ij})}{\sum_k exp(b_{ik})} \tag{5}$$

The updated routing coefficients will then be integrated in Eq.(2) to start another iteration in this routing procedure (Fig.3 ❷).

The number of iterations is determined by the convergence of routing coefficients and set by programmers. Several recent studies indicate that the number of iterations increases for tasks with large datasets and categories [45, 46]. Once all the iterations complete, the features of L capsules should have already been routed to the H capsules and ready to proceed to the following layers.

Summary. Generally, the routing algorithms (e.g., Dynamic Routing, Expectation-Maximization Routing) share the similar execution pattern and exhibit several core features in CapsNet routing procedure: (1) The execution of the RP exhibits strong data dependency and needs to be sequentially processed. (2) The procedure adopts all-to-all computation, which routes all the

Algorithm 1 Dynamic Routing Procedure

Input: L capsules u, weight matrix W
Output: H capsules v
1: for all L capsule i & all H capsule j from all input set k:
  $\hat{u}^k_{j|i} \leftarrow u^k_i \times W_{ij}$ ▷ Eq.1
2: for all L capsule i & all H capsule j:
  $b_{ij} \leftarrow 0$ ▷ Initialize Routing Coeffcients
3: **for** Routing iterations **do**
4: for all L capsule i:
    $c_{ij} \leftarrow softmax(b_{ij})$ ▷ Eq.5
5: for all H capsule j from all input set k:
    $s^k_j \leftarrow \sum_i \hat{u}^k_{j|i} \times c_{ij}$ ▷ Eq.2
6: for all H capsule j from all input set k:
    $v^k_j \leftarrow squash(s^k_j)$ ▷ Eq.3
7: for all L capsule i & H capsule j:
    $b_{ij} = \sum_k v^k_j \hat{u}^k_{j|i} + b_{ij}$ ▷ Eq.4
8: **end for**
9: **Return** v

Table 1: CapsNet Benchmark Configurations

Net-work	Dataset	Configuration			
		BS	L Caps	H Caps	Iter
Caps-MN1	MNIST[47]	100	1152	10	3
Caps-MN2		200	1152	10	3
Caps-MN3		300	1152	10	3
Caps-CF1	CIFAR10[48]	100	2304	11	3
Caps-CF2		100	3456	11	3
Caps-CF3		100	4608	11	3
Caps-EN1	EMNIST_Letter[49]	100	1152	26	3
Caps-EN2	EMNIST_Balanced[49]	100	1152	47	3
Caps-EN3	EMNIST_By_Class[49]	100	1152	62	3
Caps-SV1	SVHN[50]	100	576	10	3
Caps-SV2		100	576	10	6
Caps-SV3		100	576	10	9

L capsules to the H capsules and forms aggregation in all possible dimensions. (3) The procedure produces a large amount of intermediate variables. (4) The routing procedure is iteratively processed to generate the dynamic coefficients to pass the feature information. We will discuss these features in relation to performance characterization of CapsNet in Sec.3, and our optimizations on Dynamic Routing in the following Sections can be easily applied to other routing algorithms with simple adjustment.

3. CHARACTERIZATION AND ANALYSIS

While the CapsNet starts gaining popularity from both academia and industry, the performance characterization of CapsNet on modern high-performance platforms is largely neglected. Given that GPU has become the major platform for executing CapsNet due to high computation capability and deep optimization on matrix operations (which CapsNet has a large amount of), we adopt some of the state-of-the-art NVIDIA GPU platforms to conduct a comprehensive characterization towards the execution behaviors of various CapsNets listed in Table 1. We use 4 different datasets and corresponding 12 CapsNets with CapsNet-MNIST like structure (Sec.2.1), and different configurations on batch size (BS in Table 1), L capsules, H capsules and routing iteration number. These CapsNets' inference are processed via PyTorch framework [51] with the latest deep learning library (i.e., CuDNN [52]), which already enables the state-of-the-art CNN optimizations [53].

3.1 Overall Evaluation for CapsNet Inference

Fig.4 demonstrates the performance breakdown of each layer to the overall CapsNet execution. It shows that, across different CapsNet configurations, the routing procedure (RP) accounts for an average of 74.62% of the entire inference time, becoming the major performance bottleneck. we further conduct detailed analysis on the performance results of CapsNets on GPU and make the following observations:

Observation 1: ineffectiveness of batched execution. One common strategy to accelerate CNNs is to conduct batched execution for improving hardware utilization, especially when input dataset is large. However, it cannot improve the RP's performance during inference. As Fig.4 illustrates, with the increasing of batch size (i.e., Caps-MN1 → Caps-MN3),the overall CapsNet inference time increases; meanwhile, the RP proportion also expands with batch size.

Observation 2: sensitivity to network scaling. Table 1 shows the network size (formed by a combination of L capsules, H capsules and routing iterations) for each individual case, e.g., Caps-SV1 being the smallest. The red curve in Fig.4 also demonstrates that the

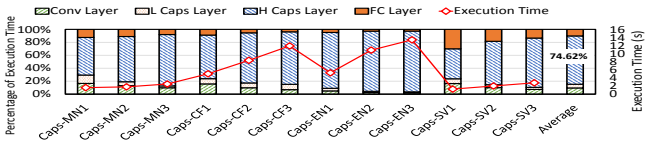

Figure 4: The overall execution time breakdown of CapsNets on GPU across different layers. Red line represents the actual inference time.

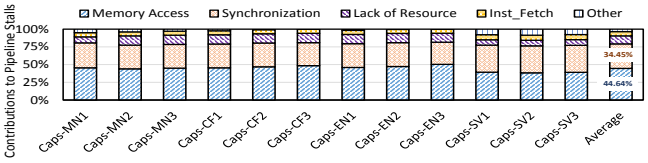

Figure 5: The breakdown for pipeline stall cycles during RP execution on Tesla P100 Pascal GPU.

overall inference time and RP's percentage generally increases when scaling up the network size (e.g., comparing Caps-MN1, Caps-CF1, Caps-EN1 and Caps-SV1). This implies that RP's execution time is sensitive to network size as well.

To summarize, using the highly optimized deep learning library, the RP execution time on GPU can not be effectively reduced through the general CNN optimization techniques such as batch execution. Moreover, it exhibits a certain level of sensitivity to network scaling. Both of these factors make the RP execution a dominating performance bottleneck for CapsNet inference, especially with a growing size and complexity of future CapsNet structures[45, 46].

3.2 Root Causes for Inefficient RP Execution

To understand the root causes of RP's inefficiency on GPU, we use NVprofiler [54] to collect runtime GPU stats for comprehensive analysis. We observe two root causes for poor RP execution efficiency on GPU-like architectures:

(1) Significant Off-Chip Memory Accesses: We profile the utilization of several major GPU function units during RP execution on a NVIDIA Tesla P100 GPU. We observe that the arithmetic logic unit (ALU) is lightly utilized (i.e., on average only 38.6% across the investigated benchmarks) while the load/store unit (LDST) is heavily stressed with an average utilization of 85.9%. This implies that CapsNets' RP phase fails to effectively leverage the GPU strong computation capability and is severely limited by the intensive off-chip memory access. We further investigate the factors that may contribute to GPU pipeline stalls during RP, including the off-chip memory access, barrier synchronization, lack of resource, etc. Fig.5 profiles the individual contribution of each major factor to the overall pipeline stall cycles. As can be seen, the majority of the pipeline stalls are induced by the memory access (i.e., on average 44.64%). This further confirms that RP performance is significantly limited by the off-chip memory access. This is caused by a combination of massive intermediate variables from RP execution and limited on-chip storage. Fig.6(a) illustrates the ratio of RP's intermediate variables' size to on-chip memory sizes of different generations of NVIDIA GPUs. As can be seen, the size of RP's intermediate variables far exceeds the GPU on-

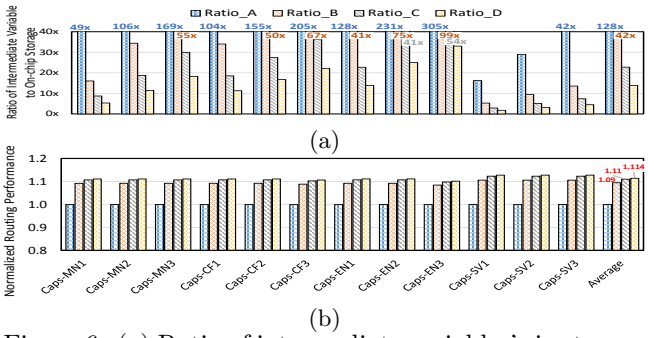

Figure 6: (a) Ratio of intermediate variables' size to on-chip storage of different GPUs; (b) the impact of on-chip storage sizes of state-of-the-art GPUs on RP's execution. A: 1.73MB (K40m), B: 5.31MB (Tesla P100), C: 9.75MB (RTX2080Ti), D: 16MB (Tesla V100).

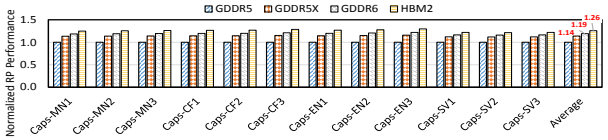

Figure 7: The impact of memory bandwidth on the overall RP performance. GDDR5:288GB/s (K40m), GDDR5X: 484GB/s (GTX 1080Ti), GDDR6: 616GB/s (RTX 2080Ti), HBM2: 897GB/s (Tesla V100)

chip storage. Given the iterative computation pattern of RP, these variables (e.g., $\hat{u}, s_j, v_j, b_{ij}, c_{ij}$) need to be periodically loaded into GPU cores from the off-chip memory due to the limited on-chip storage. Moreover, the intermediate variables are not sharable among different input batches, which also explains the ineffectiveness of batched execution as we observed in Sec.3.1. Due to the large data volume and lack of temporal value similarity from these variables, software-level schemes such as register manipulation and shared memory multiplexing are also not very effective.

(2) Intensive Synchronization: Fig.5 also indicates that the frequent barrier synchronization is the second major contributor (i.e., on average 34.45%) to the pipeline stalls. These synchronization overheads are induced by the *__syncthread()* calls, which coordinate shared memory accesses for all threads in one thread block. There are two major factors causing the frequent synchronization during the RP execution: (i) the RP execution contains numerous aggregation operations (e.g., Eq.(2)), inducing massive data communication between the threads through shared memory; (ii) the size of the intermediate variables far exceeds the shared memory size, leading to frequent data loading from the global memory. Thus, *__syncthread()* calls occur frequently to avoid the potential write-after-write and write-after-read hazards.

To address these issues above, we attempt to apply two naive solutions: scaling up the on-chip and off-chip memory capacity. Fig.6(b) shows the impact of on-chip memory sizes of different generations of NVIDIA GPUs on RP execution. We can observe that increasing on-chip memory size can help alleviate the challenges above but not very effective, e.g., only up to an average of 14% performance improvement for the 16MB V100. This is because the nonsharable intermediate variables' size from RP still far exceeds the current GPUs' on-chip storage, shown in Fig.6(a). Similarly, Fig.7 shows that only increasing memory bandwidth from 288GB/s GDDR5 to 897 GB/s HBM slightly improves the overall RP's performance by an average of 26%. This indicates that higher off-chip memory bandwidth can only solve a small part of the problem but itself does not reduce the high intensity of the off-chip memory accesses. Therefore, to significantly improve CapsNet's inference performance, we need a customized solution to address the root causes for RP's inefficient execution.

4. BASIC IDEA: PROCESSING-IN-MEMORY + PIPELINING

As discussed previously, we aim to address CapsNets' significant off chip-memory access caused by massive unshareable intermediate variables and intensive synchronization due to numerous aggregation operations in RP. Meanwhile, we also want to utilize the excellent core computing capability provided by modern GPUs for deep learning's matrix operations. Thus, we propose a hybrid computing engine named *"PIM-CapsNet"*, shown in Fig.8. It utilizes GPU's native on-chip units as the host for fast processing layers such as Conv and FC, while pipelining with an off-chip in-memory acceleration that effectively tackles RP's inefficient execution. For example, since multiple input sets are generally batched together to be concurrently processed in RP to avoid the local optimal solution of the routing coefficients [55], host processors can start processing Conv/FC operations from the different batches of the input sets while waiting for RP's results from in-memory processing on the current batch, forming an execution pipeline. Furthermore, the in-memory accelerators of our PIM-CapsNet design can hierarchically improve RP's execution efficiency by minimizing data movement and maximizing parallel processing. Note that our proposed design is a general optimization solution that is applicable to different routing algorithms used in RP.

To build our in-memory acceleration capability for RP, we resort to one of the emerging 3D stacked technologies, i.e., hybrid memory cube (HMC)[38], which has become a promising PIM platform [39, 40, 41, 42, 43]. The major reason to replace current GPU's off-chip memory (e.g., GDDRX or HBMs) with HMC in our design is their lack of logic layer for in-memory integration. As illustrated in Fig.9, HMC stacks several DRAM dies on the CMOS logic layer as a cube (specification 2.1 [56]). The memory cube connects with the host processor (i.e., GPU cores in our case) via the fully-duplex links which can provide up to 320GB/s of external memory bandwidth. Additionally, the logic layer is split into 32 sub-memory controllers with each communicates its local DRAM banks through Through-Silicon Vias (TSVs) which together provide an internal memory bandwidth of 512GB/s [38, 41]. The sub-memory controller and its local DRAM partitions (each partition contains multiple DRAM banks) form the *vault* architecture, as highlighted in the red dashed box. The logic layer receives system commands and routes memory access to different vaults. A crossbar is integrated in the logic layer to support the communication between SerDes links and vaults. Note that relatively simple computation logic can be integrated onto HMC's

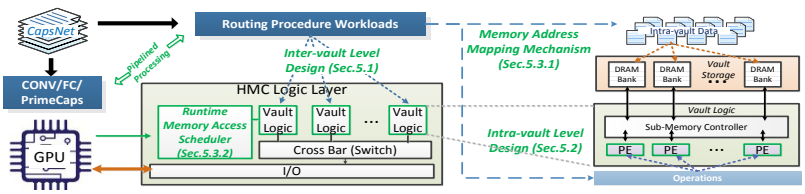

Figure 8: The overview of PIM-CapsNet Design.

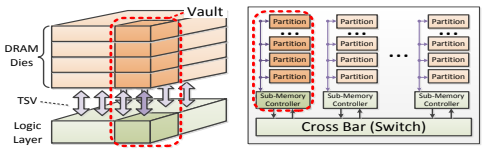

Figure 9: Left: HMC Organization. Right: HMC Block Diagram. The Red dashed box marks the vault structure.

Table 2: Possible Parallelizable Dimensions

	Batch (B-dimension)	Low-level Caps (L-dimension)	High-level Caps (H-dimension)
Eq.1	x	x	x
Eq.2	x		x
Eq.3	x		x
Eq.4		x	x
Eq.5		x	

logic layer which can directly access data from vaults via memory controllers and benefit from large internal memory bandwidth. This layer is very suitable for integrating in-memory accelerators for RP execution. Next, we will discuss the detailed PIM-CapsNet design.

5. PIM-CAPSNET ARCHITECTURE DESIGN

There are two **key design objectives** in PIM-CapsNet: (i) maintaining workload balance with minimal data communication at *inter-vault level*(Sec.5.1 and Sec.5.3), and (ii) maximizing parallel processing at *intra-vault level* but within architecture design constraints (Sec.5.2 and Sec.5.3).

5.1 Inter-Vault Level Design

A typical HMC design adopts crossbar switch to support the inter-vault communication [56], and the overall HMC performance power/thermal efficiency can be drastically impacted by the increasing data movements across vaults. Employing a more complicated data flow management (e.g., network-on-chip) on the HMC logical layer would further exacerbate the thermal issue. In our proposed design, we leverage the unique features of the RP execution and intelligently distribute workloads across vaults with minimal inter-vault communication.

5.1.1 RP's Unique Feature: Highly Parallelizable in Multi-Dimensions

As an interesting feature, the operations of RP equations (Sec.2.2) are highly parallelizable. For example, in Eq.(1), the vector-matrix multiplications for all the low- to high-level (L-H) capsule pairs are independent. We define such independent operations on L capsules as parallelism in the L-dimension, while the independent operations on H capsules are defined as parallelism in the H-dimension. Additionally, if operations corresponding to different batches are independent, they are defined as parallelism in the B-dimension. Thus, we make the following key observations:

Observation I: *Operations of each equation in the RP can be partitioned into multiple independent sub-operations on at least one of the three dimensions (L, H, or B), suggesting highly parallelizable feature.*

Table 2 further demonstrates which possible dimensions these five equations of the dynamic routing procedure can be parallelized through. Based on it, we also make the second key observation:

Observation II: *All the RP equations cannot be concurrently executed through the same dimension.*

Observation I indicates that the inter-vault data communication can be reduced by parallelizing the independent sub-operations of each equation on *one chosen dimension* and then distributing them across HMC vaults. By doing so, the major communication across vaults is only required by the aggregation operations at that dimension (aggregations required by the other two dimensions will be performed locally within vaults), which is relatively low. Observation II, however, emphasizes that the inter-vault communication can not be fully eliminated for RP as none of the three dimensions can support the parallelization for all the RP equations. When distributing the entire RP workloads on a certain dimension, some related data have to be reloaded to another designated vault for aggregation operations.

5.1.2 Parallelization and Workload Distribution

Since distributing the workloads on different dimensions leads to different communication overheads and execution patterns, it is important to apply an intelligent workload distributor to achieve the optimal design for power and performance. To minimize the inter-vault communication, *our distributor only distributes the RP workload on a single chosen dimension.* Fig.10 illustrates an example of the RP execution flow when distributing on the B-dimension. As it shows, the RP operations of Eq.1,2,3 (❶❷❸) exhibit parallelism along the B-dimension. Additionally, the multiplication operations of Eq.4 (❹) (i.e., $v_j^k \times \hat{u}_{j|i}^k$) can also be parallelized along the B-dimension. These parallelizable workloads can be divided into snippets (i.e. green blocks) and distributed across the HMC vaults. Note that typical CapsNet workloads will generate way more snippets than the number of vaults in the HMC (e.g., up to 32 vaults in HMC Gen3).

Due to the aggregation requirements on all the dimensions during RP execution, there are always some workloads that cannot be processed in parallel on the selected dimension, leading to workload imbalance. For instance, the remaining operations of the RP procedure in Fig.10 (i.e., the purple blocks), including partial Eq.4 (❺) and Eq.5 (❻) cannot be divided into snippets due to the lack of parallelism on the B-dimension. Additionally, these workloads usually require data to be aggregated in one place, making it hard to utilize all the HMC computation resources. Furthermore, the data size requested by the aggregation operations can be large, which may cause massive inter-vault communication and even the crossbar stalls. To reduce the inter-vault communication and increase hardware utilization, we propose to pre-aggregate the partial aggregation operations inside each vault for the corresponding allocated snippets. For example, when the RP workloads are distributed on B-dimension, the pre-aggregation can

Table 3: Parameters for Modeling Inter-Vault Data Movement

Symbol	Description
I	DR iteration number
N_B	Scale for B-dimension i.e. batch size
N_L	Scale for L-dimension i.e. number of low-level capsules
N_H	Scale for H-dimension i.e. number of high-level capsules
N_{vault}	Number of Vault
C_L	Dimension of low-level capsule i.e. number of scaler per low-level capsule
C_H	Dimension of high-level capsule i.e. number of scaler per high-level capsule
$SIZE_x$	Data size of a variable or packet head&tail

be performed for Eq.(4) to combine the b_{ij}^k from the snippets assigned to a specific vault before performing global inter-vault aggregation.

Guiding Distribution via an Execution Score: To achieve the optimal power/thermal and performance results, we propose a metric called execution score S to guide workload distribution, which quantitatively estimates the RP execution efficiency under a given workload distribution. S considers workload assignment to each vault, inter-vault communication overheads, device-dependent factors and inter-vault memory bandwidth. S is modeled as $S = 1/(\alpha E + \beta M)$.

where E represents the largest workloads distributed to a single vault (since even distribution across vaults is typically not feasible), which can be quantified based on the amount of allocated operations. M represents the amount of inter-vault data movement. Both E and M are affected by the distribution strategy and the model configuration. Finally, α and β represent the device-dependent coefficients, determined by HMC frequency and inter-vault memory bandwidth, respectively.

The calculation of S is independent of the online parameters, thus, the distribution strategy can be determined off-line before the actual inference. Then, the in-vault operations according to this selected distribution dimension will be generated by compiler and the corresponding workloads will be assigned into each vault via a hardware scheduler at runtime. We now demonstrate how to model S via estimating E and M on the three distribution dimensions.

Distribution on B-dimension: As Fig.10 illustrates, the largest workload assigned to a single vault (E) consist of the workload snippets including ❶❷❸❹ and the partial operations ❺❻. With our optimizations, the single vault can get at most $\frac{\lceil \log_2(N_{vault}) \rceil}{N_{vault}}$ of the unparallelizable operations, where N_{vault} represents number of the HMC vaults. Using parameters shown in Table 3, E can be modeled as follows:

$$\begin{aligned} E_B = \lceil \tfrac{N_B}{N_{vault}} \rceil \times N_L \times N_H \times C_H \times (2C_L - 1) + I \times [\lceil \tfrac{N_B}{N_{vault}} \rceil \times \\ N_H \times C_H \times (2N_L - 1) + \lceil \tfrac{N_B}{N_{vault}} \rceil \times N_H \times (3C_H + 19) + \\ \lceil \tfrac{N_B}{N_{vault}} \rceil \times N_L \times N_H \times (2C_H - 1) + \tfrac{\lceil \log_2(N_{vault}) \rceil}{N_{vault}} + 4 \times C_H] \end{aligned} \quad (6)$$

Since $N_L \gg 1$, the above equation can be simplified as:

$$E_B = \lceil \tfrac{N_B}{N_{vault}} \rceil \times N_L \times N_H \times [(4I - 1)C_H + 2C_L C_H - I] \quad (7)$$

The inter-vault data communication consists of sending pre-aggregated b_{ij} from all the vaults to a single vault and scattering c_{ij} across all the vaults. The data transmission is in the form of packets with the head and tail size represented as $SIZE_{pkt}$. Therefore, the amount of data movements M can be represented as:

$$\begin{aligned} M_B = I \times [(N_{vault} - 1) \times N_L \times N_H \times (SIZE_{b_{ij}} + SIZE_{pkt}) \\ + (N_{vault} - 1) \times N_L \times N_H \times (SIZE_{c_{ij}} + SIZE_{pkt})] \end{aligned} \quad (8)$$

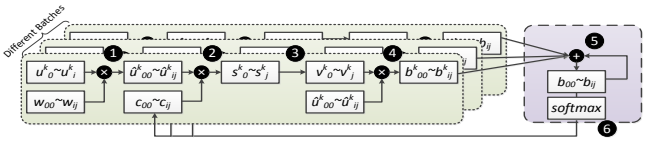

Figure 10: The execution diagram for the RP procedure with B-dimension distribution. The workloads in green blocks can be split across vaults, but workloads in purple blocks cannot be distributed via B-dimension.

Distribution on L-dimension: As Table 2 illustrates, the RP operations of Eq.1,4,5 can be divided into workload snippets on the L-dimension. Besides, partial operations from Eq.2 (i.e., $\hat{u}_{j|i}^k \times c_{ij}$) also exhibit parallelism on the L-dimension. Thus, E can be represented as:

$$E_L = N_B \times \lceil \tfrac{N_L}{N_{vault}} \rceil \times N_H \times [2I(2C_H - 1) + C_H(2C_L - 1)] \quad (9)$$

The inter-vault communication contains data all-reducing for of s_j and broadcasting v_j^k:

$$\begin{aligned} M_L = I \times [N_B \times (N_{valut} - 1) \times N_H \times (SIZE_{s_j^k} + SIZE_{pkt}) \\ + N_B \times (N_{vault} - 1) \times N_H \times (SIZE_{v_j^k} + SIZE_{pkt})] \end{aligned} \quad (10)$$

Distribution on H-dimension: As Table 2 presents, only Eq.5 cannot be parallelized on this dimension. Hence, E can be represented as

$$E_H = N_B \times N_L \times \lceil \tfrac{N_H}{N_{vault}} \rceil \times C_H \times [2C_L - 1 + 2I] \quad (11)$$

The inter-vault communication contains data all-reducing for of b_{ij} and broadcasting c_{ij}:

$$\begin{aligned} M_H = I \times [(N_{vault} - 1) \times N_L \times (SIZE_{b_{ij}} + SIZE_{pkt}) \\ + N_L \times (SIZE_{cij} + SIZE_{pkt})] \end{aligned} \quad (12)$$

5.2 Intra-Vault Level Design

In this section, we propose the intra-vault level design that effectively processes the sub-operations of each equation that are allocated to a vault. We target the design for IEEE-754 single precision (FP32) format, which provides sufficient precision range for CapsNet workloads [14]. Our design can also fit other data formats with minor modifications.

5.2.1 Intra-Vault Level Workload Distribution

In a basic HMC design, the logical layer of each vault contains a sub-memory controller to handle the memory access. In order to conduct RP specific computation, we introduce 16 processing elements (PEs) into each vault. This design overhead has been tested to satisfy both area and thermal constraints for HMC [43] (see detailed overhead analysis in Sec.6.5). These PEs (integrated onto the logic layer) are connected to the sub-memory controller in the vault, shown in Fig.11(left). Note that the number of parallel sub-operations on certain dimension is generally the orders of magnitude higher than the number of vaults in HMC. In other words, many parallel sub-operations will be allocated to the same vault. Hence they can be easily distributed on the same dimension and concurrently processed via the PEs without introducing additional communication overheads. There

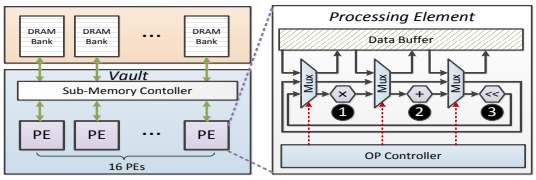

Figure 11: Intra-vault level Architecture Design.

may exist some extreme cases that the number of parallel sub-operations allocated to the vault is smaller than the number of PEs, leading to low PE utilization. Since most equations in RP can be parallelized on more than one dimension, the workloads can then be distributed along a different dimension which can produce enough parallelism to fully utilize the PE resources.

5.2.2 Customized PE Design

There have been some studies [43, 42, 57] that integrate adders and multipliers onto the HMC logic layer to perform multiply-accumulation (MAC) which is an essential operation for deep learning (e.g., CNN). However, CapsNet's RP execution involves other operations beyond MAC. As discussed in Sec.2.2, among the five RP equations, Eq.1, Eq.2 and Eq.4 can be translated into MAC operations. But the other operations including Eq.3 and Eq.5 involve more complex functions such as division (Eq.3), inverse square-root (Eq.3) and exponential functions (Eq.5) that require complicated logic design, resulting in large performance and power/thermal overheads [58, 59].

Operation Approximation: To gain high performance while maintaining low hardware design complexity, we propose approximation to simplify these operations with negligible accuracy loss. For simplifying division and inverse square-root functions of the FP32, we apply bit shifting [60], which is widely adopted in graphics processing domain [61, 62, 63]. For exponential function, we approximate the operation as follows:

The original exponential function can be transformed in the form of power function with the base as 2 [64]:

$$e^x = 2^{log_2(e)\times x} = 2^y = 2^{\lfloor y \rfloor}(1 + 2^{y-\lfloor y \rfloor} - 1) \quad (13)$$

where $\lfloor y \rfloor$ is the integer part of y, and $y - \lfloor y \rfloor$ is the decimal part.

Fig.12(a) illustrates an example of representation transfer. First, the ep of both A and B will be subtracted from the bias b to get their real exponents (i.e., $ep - b$), as shown in Fig.12(a)❶&❸. Then, the most significant $ep-b+1$ bits of Ⓐ's significand (i.e., $1 + fraction$) will be chunked and filled into the least significant $ep-b+1$ bits of Ⓒ's exponent field, with the remaining exponent bits in Ⓒ filled by zeros, as shown in Fig.12(a)❷. We conduct s similar operation to transfer Ⓑ to the Ⓒ's fraction. Since Ⓑ is a fraction value, its exponent is a non-positive number. Ⓑ's significand will be logical shift right by $|ep - b|$ bits and then its most significant 23 bits are filled into Ⓒ's fraction field, as shown in Fig.12(a)❹.

The above two transfers can be considered as bit shifting on the significand (i.e., $1+fraction$) in FP32 format with the distance and direction determined by the real exponent (i.e., $ep - b$). As illustrated in Fig.12, the Ⓑ's exponent can increase to match the exponent of Ⓐ with its significand bits logically shifting right. By doing this, the faction representation can be described as Ⓓ in Fig.12(b). Given the matched real exponent value (i.e., $ep - b$), the two representations, i.e., Ⓐ and Ⓓ in Fig.12(b), now share the identical bit shifting operations. Additionally, Ⓐ and Ⓓ correspond to ExpResult's (i.e.,exponential function's result) integer and fraction, respectively. There is no overlapping between their fraction bits. Thus, these two representations can be combined (i.e., Ⓐ OR Ⓓ) followed by a unified bit shifting operation on the significand. Note that the exponent matching procedure (Ⓑ→Ⓓ) could over chuck several least significant bits which would originally be mapped into the ExpResult.

Since the exponent matching and combination of two FP32 numbers can be simply considered as a FP32 addition, we can treat the ExpResult computation as an addition of the exponent and fraction representations (i.e., $\lfloor y \rfloor + b + 2^{y-\lfloor y \rfloor} - 1$) followed by the bit shifting operations. Note that the power of 2 function causes high complexity in both execution and logic design, we propose to simplify it as follows:

The above polynomial can be expanded as $(y+2^{y-\lfloor y \rfloor} - (y-\lfloor y \rfloor)+b-1)$; then, the average value Avg of $(2^{y-\lfloor y \rfloor} - (y - \lfloor y \rfloor))$ can be achieved via integrating the polynomial over $(y - \lfloor y \rfloor) \in [0, 1)$, which is fixed and can be obtained offline. With y represented by x, the exponential function can be described as follows:

$$ExpResult \simeq BS(log_2(e) \times x + Avg + b - 1) \quad (14)$$

where the BS is bit shifting operations with information from $ep - b$; and $log_2(e)$ is a constant that is computed offline.

Accuracy Recovery: Under the worst case scenari, othere might be several lowest significand bits chucked when mapping from Ⓓ to Ⓒ. It may cause some accuracy loss. To minimize the bit chucking impact, we analyze 10,000 exponential executions to collect the value differences between the approximated and original results. During the approximation execution, the accuracy loss will be recovered via enlarging the results by the mean percentage of the value difference. Note that our accuracy recovery scheme only introduces one additional multiplication operation during the inference, which guarantees the high performance and the low design complexity compared to other exponential approximation methodology, e.g., applying look-up tables [59]. Sec.6.5 shows the detailed accuracy analysis.

Final PE Structure: According to the discussion above, the special functions can be simplified as a combination of addition, multiplication, and bit shifting operations. Thus, our intra-vault PE employs adders, multipliers, and bit-shifters to construct these special functions, as shown in Fig.11. Specifically, our PE enables the flow configuration via the multiplexer (MUX) to support different types of operations. For example, PE execution flow ❶❷ is for MAC operations; ❸❷❶❷❶ is for inverse square-root operations; and ❶❷❷❸ is for exponential function.

5.3 Contention Reduction in CapsNet Design

In this section, we discuss strategies to combat the

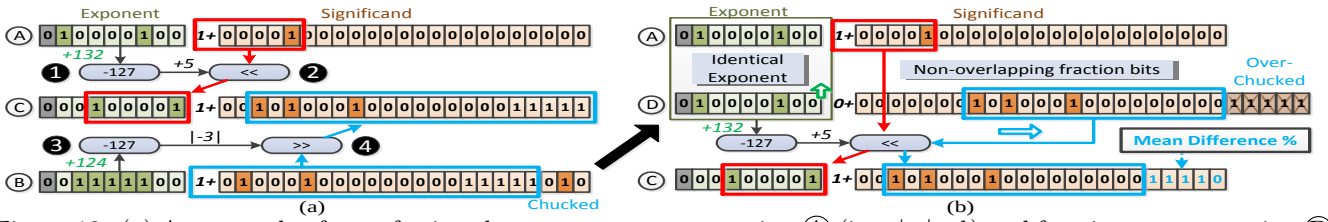

Figure 12: (a) An example of transferring the exponent representation Ⓐ (i.e., $\lfloor y \rfloor + b$) and fraction representation Ⓑ (i.e., $2^{y-\lfloor y \rfloor} - 1$) to the exponential function's result Ⓒ in FP32 format. (b) Combining the exponent representation Ⓐ and the fraction representation (i.e., Ⓓ that transferred from Ⓑ), and applying a unified bit shifting to obtain the exponential function's result Ⓒ.

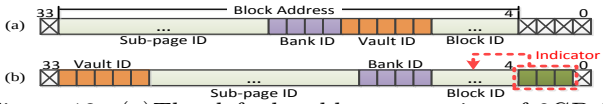

Figure 13: (a)The default address mapping of 8GB in HMC Gen3; (b) Our address mapping.

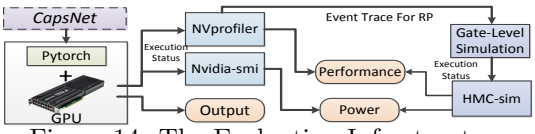

Figure 14: The Evaluation Infrastructure.

memory-level contention in our PIM-CapsNet design.

5.3.1 *Memory Address Mapping Mechanism*

In the default HMC design, memory access granularity is 16 bytes which is defined as a block, and the sequential interleaving mapping mechanism is applied to benefit from better bandwidth utilization. Moreover, MAX block is introduced to define the maximum number of blocks that one bank can provide at a time. Its size can be set to 32B, 64B, 128B, or 256B [38]. In this study, we redefine MAX block as a subpage in order to differentiate it from block, and one subpage is composed of multiple blocks. Fig.13(a) illustrates the default address mapping from HMC Gen3 specifications [38]. The lowest 4 bits and the highest 1 bit are ignored, and the remaining bits describe the block address. From its lower to higher bits, a block address is composed of several fields: the block ID in the sub-page (the number of bits is determined by the sub-page size), the 5-bit vault ID for 32 vaults, the 4-bit Bank ID for 16 banks per vault, and the sub-page ID in the bank. As can be seen, sub-pages with consecutive addresses will be first spread sequentially to different vaults and then different DRAM banks.

Note that our inter-vault level design requires consecutive blocks allocated into one vault to avoid high inter-vault communication. This can be easily addressed by moving up the vault ID field to the highest field of the block address (as shown in Fig.13(b)) so that vault ID remains unchanged during the intra-vault memory address mapping. However, at the intra-vault level, PEs process their workloads in parallel and concurrently generate data requests, which may result in serious bank conflicts and vault request stalls (VRS).

Interestingly, we observe that most of the concurrent PE requests assigned to the same bank actually visit different blocks. Based on this, we propose a new memory addressing scheme to distribute these blocks to different banks, in order to significantly alleviate bank conflicts and decrease the VRS. However, simply distributing blocks to different banks could further increase the VRS as one PE may request multiple consecutive blocks at a time. Because in this case these blocks will reside in multiple banks, it leads to multiple accesses to these banks, resulting in higher bank conflicts. To ensure the consecutive blocks required by one PE are stored in the same bank, our scheme will dynamically determine the sub-page size according to the size of the requested data. As shown in Fig.13(b), we leverage bit 1 $\sim$ bit 3 in the lowest 4 ignored bits as the indicator to determine the sub-page size for the data requests, where range "000" $\sim$ "100" represents the sub-page size from 16B $\sim$ 256B. Given that the data requests from PEs and host GPU need to be allocated into different banks, the indicator bits are dynamically assigned by the page table during the virtual-physical address translation according to the storage requested by each variable involved in each execution thread.

5.3.2 *Identifying Memory Access Priority*

During CapsNet execution, resource contention from concurrent data requesting to the same vault from both host GPU and PEs could occur. Although the Conv/FC layers exhibit much better data locality than the RP, they may still need to periodically request data from the HMC. By default, the priority of a request is determined by the arrival time. But this can cause stalls if both sides are requesting to access the same bank in a vault, which may occur more frequently after applying our new address mapping.

To address this issue, we propose a runtime memory access scheduler (RMAS) to dynamically identify the priority of memory requests from both the host GPU and vault PEs. We first observe that, with our address mapping mechanism, consecutive data are likely to be distributed across the banks within a vault instead of being scattered over all the vaults. Thus, it is highly likely that each consecutive data request from the host will only access a single or few vaults at a time instead of all the vaults. This provides opportunities for some vaults to serve the GPU memory requests in rotation without affecting the overall HMC execution.

To quantify the impact of vault access priority, the runtime scheduler (RMAS) first collects the information of the issued operations by HMC and the number of PE requests (Q) from the vault request queues in HMC. It also collects the information of the issued operations from the host GPU about which and how many of vaults the operations are requesting from the HMC (defined as n_{max}). The collected information above from RMAS

is associated with the HMC performance overhead if granting priority to access from either side. Thus, the performance impact of serving the two types of access requests from HMC and GPU can be quantified via the following overhead functions:

$$\kappa = \gamma_v \times n_h \times \overline{Q} + \gamma_h \times \frac{n_{max}}{n_h} \quad (15)$$

where κ represents the quantified performance impact; γ_v and γ_h represent the impact factors that are determined by the issued operations' type from HMC and host GPU, e.g., memory intensive operations corresponds to a large γ as their performance is more sensitive to the memory bandwidth than the computation intensive operations; $\overline{Q}$ is the average number of PEs' requests in the request queues from the targeted vaults; n_h is the number of vaults that are granted with access priority from the host GPU, which is in the range of $[0, n_{max}]$. If n_h is "0", all the target vaults will first process current HMC PEs requests before processing the GPU requests; while if n_h is n_{max}, all the target vaults will grant priority to the GPU requests. To achieve the minimal impact (i.e. $\min(\kappa)$), n_h should be equal to $\left|\sqrt{\frac{n_{max}\times\gamma_h}{\overline{Q}\times\gamma_v}}\right|$, where $n_h \in [0, n_{max}]$. The RMAS will then send the control signals to n_h vaults that will give host GPU higher priority to access. Note that a vault with smaller Q has higher priority to be chosen by our RMAS to further reduce the impact on execution.

6. EVALUATIONS

6.1 Experimental Setup

Fig.14 shows our evaluation infrastructure to evaluate PIM-CapsNet. We first employ Pytorch [51], a popular open-source machine learning framework that supports dynamic computation graphs, to execute the CapsNet on our host GPU. We then design a physical-simulator cooperated platform which takes the execution status of CapsNet provided by Pytorch to obtain the detailed performance and energy results when applying PIM-CapsNet. From the physical side, we adopt the Nvidia Tesla P100 [65] as our host processor to evaluate performance and energy consumption of the Conv/PrimeCaps/FC layers of CapsNet. The detailed execution time and power information for these layers are captured by using NVprofiler [54] and Nvidia-smi [66]. From the simulator side, we leverage NVprofile to collect the event trace from host and pass it to a modified HMC-sim 3.0 [67] to simulate the computing and memory accesses in HMC. Considering that HMC-sim 3.0 cannot provide precise execution cycles and power information for the logic layer design, we conduct a gate-level simulation on Cadence [68] to measure the execution latency and power consumption for our logic design (PE). We then integrate the gate level results and our PIM design in HMC-sim to obtain the performance and energy consumption of RP execution. Finally, since the execution of CapsNet is pipelined on the host processor and HMC, we combine the results from both sides via overlapping the execution time and accumulating the energy consumption. The detailed platform configurations are shown in Table 4. In this work, we choose 12 different types of CapsNets as our benchmarks which are introduced in Sec.3 and shown in Table 1.

Table 4: Platform Specifications

Host Processor (GPU)	
Shading Unit	3584 @ 1190MHz
On-chip Storage	L1Cache/Shared: 24KB x 56
	L2 Cache: 4MB
Default Memory	HBM, 8GB, 320GB/s
HMC	
Capacity	8 GB, 32 Vault, 16 Banks/Vault
Bandwidth	Extl:320 GB/s, Intl: 512GB/s
No. of PEs per Vault	16
Frequency	312.5MHz

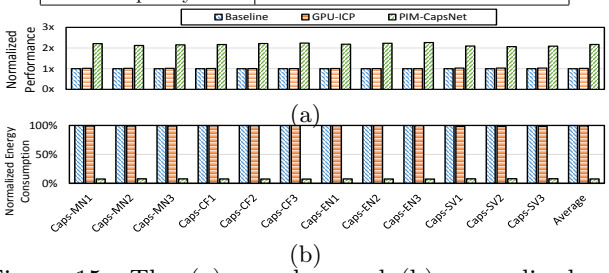

Figure 15: The (a) speedup and (b) normalized energy consumption of PIM-CapsNet on the RP procedure comparing with the GPU-based design.

To evaluate the effectiveness of PIM-CasNet, we compare with the following design scenarios: (1) Baseline: the state-of-the-art GPU accelerated CapsNet execution with the HBM memory (320GB/s). (2) GPU-ICP: the GPU accelerated CapsNet execution with ideal cache replacement policy. (3) PIM-CapsNet: our combined inter-vault level and intra-vault level design for RP acceleration with the new memory addressing scheme and RMAS scheme (4) PIM-Intra: our PIM-CapsNet without inter-vault level design, and the memory addressing scheme does not optimize the inter-vault data distribution. (5) PIM-Inter: our PIM-CapsNet without intra-vault level design, and the memory addressing scheme does not support the intra-vault bank conflict optimization. (6) RMAS-PIM and (7) RMAS-GPU: Our PIM-CapsNet with the naive memory access scheduling, which always grants HMC PEs higher priority than GPU for RMAS-PIM, and always grants GPU higher priority than HMC PEs for RMAS-GPU. (8) All-in-PIM: the HMC accelerated CapsNet execution, including RP and other layers' execution.

6.2 Effectiveness of PIM-CapsNet

6.2.1 Performance and Energy for RP execution

We first evaluate the effectiveness of PIM-CapsNet on RP execution. Fig.15 illustrates the normalized performance and energy consumption of our PIM-CapsNet design compared with the GPU-based design (i.e., The Baseline and GPU-ICP) for the RP execution. From the figure, we first observe that the GPU-ICP only outperforms Baseline by 1.14% on performance and 0.77% on energy during RP execution. This is because RP requires a large number of intermediate variables which exceed the on-chip storage. As a result, the cache policy improvements can barely reduce the off-chip memory accesses. Second, PIM-CapsNet outperforms Baseline by 117% on performance by addressing the large number of memory access as well as the intensive synchronizations. Third, from Fig.15(b), we observe PIM-CapsNet saves 92.18% on energy comsumpting comparing to Baseline. This is because the entire working power of our PIM design is much lower then host

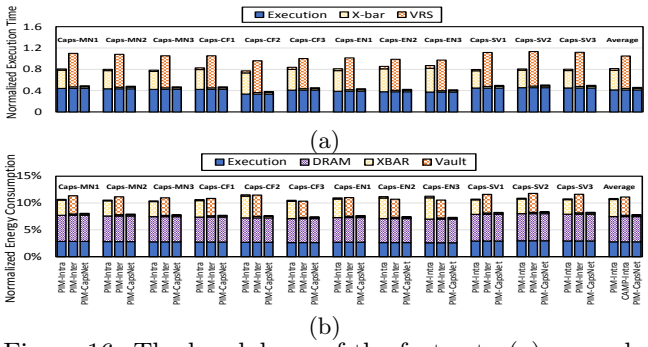

Figure 16: The breakdown of the factor to (a) normalized performance and (b) normalized energy consumption when performing RP execution on different PIM designs.

and PIM-CapsNet is able to reduce a huge number of data movements between host and HMC. Moreover, we observe that PIM-CapsNet can achieve better performance and energy saving for RP execution in larger size CapsNet, e.g. 2.27× speedup and 92.52% energy saving of accelerating Caps-EN3 compared with 2.09× speedup and 91.90% energy saving of accelerating Caps-SV1. This implies that PIM-CapsNet exhibits the scalability in optimizing the RP execution with the ever-increasing network size.

6.2.2 *Effectiveness of Intra-Vault and Inter-Vault Level Designs*

To better understanding the effectiveness of our intra-vault and inter-vault level designs, we compare PIM-CapsNet with other PIM design scenarios for the RP execution only. Fig.16(a) illustrates the evaluation results of normalized performance with the breakdown factors for different PIM designs. From the figure, we have several observations. First, even though PIM-Intra achieves 1.22× speedup over Baseline, the inter-vault communication overheads contribute on averages 45.24% to the overall performance. This is because the PIM-intra design induces massive concurrent data transfer between the centralized computation units in the logic layer and the DRAM banks in vaults, leading to high crossbar utilization and serious stalls. Second, PIM-Inter decreases the performance by 4.73% compared with the Baseline. Compared with PIM-Intra, the inter-vault communication overheads have been significantly reduced in PIM-Inter, but the vault request stalls (VRS) grow which contribute on average 57.91% to the execution time due to the serious bank conflicts within the vault. Finally, PIM-CapsNet improves the performance about 127.83%/76.62% on average comparing to PIM-Inter/PIM-Intra by reducing both inter-vault communications and VRS. From the energy perspective, as Fig.16(b) shows, these PIM designs achieve high energy saving compared with Baseline by executing RP on energy-efficient PIM design and our PIM-CapsNet on average outperforms the PIM-Inter/PIM-Intra by 4.81%/4.52% respectively on energy saving.

6.3 Overall Performance and Energy

Fig.17 shows the normalized speedup and energy of entire CapsNet execution. First, to evaluate the effectiveness of the pipelined execution between the host and the HMC, we compare our design with all in GPUs (i.e., Baseline), All-in-PIM. From the figure, we observe that the All-in-PIM causes the 47.59% performance drop compared to the Baseline. This is because we mainly focuses on the RP procedure optimization at minimal cost in HMC, and this low-cost design choice can hardly achieve the best performance for Conv/FC layers. But it reduces the energy consumption by 71.09%, which exhibits better execution efficiency (i.e. performance /energy consumption) compared with Baseline. Fig.17 Further plots the performance and energy consumption of the pipelined design with native memory access schedulers (i.e., RMAS-PIM and RMAS-GPU). We observe that these naive schedulers can not achieve the best performance and energy saving compared with PIM-CapsNet due to the memory access stalls. In summary, our design outperforms Baseline on both performance (i.e., on average 2.44×) and energy saving (i.e., 64.91%), and it even achieves higher performance improvement compared with the acceleration for RP execution only, which benefits from the pipelined execution.

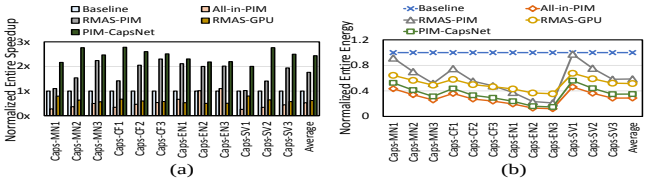

Figure 17: The (a) speed up and (b) normalized energy consumption when processing entire CapsNet with different designs.

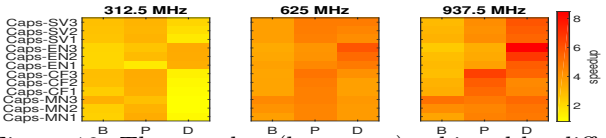

Figure 18: The speedup (heat map) achieved by different workloads distribution dimensions (X-axis) under different HMC execution frequency. Red means better improvement.

6.4 Sensitivity to PE Frequency Scaling

We conduct the sensitivity analysis on inter-vault level workload distributions in our PIM-CapsNet when different frequency is adopted in the PEs, e.g., 312.5MHz, 625MHz and 937.5MHz. Note the frequency scaling will be controlled under a certain range without violating the HMC thermal constraint. Fig.18 shows the speedup achieved by selecting different distribution dimensions (i.e., B-dimension, L-dimension, and H-dimension) under the above three different frequencies. The darker color indicates the higher speedups (e.g., the red color indicates the substantial speedups while the yellow color means trivial speedups).

It is obvious that PIM-CapsNet can achieve better improvement with higher execution frequency. We also notice that the selection of the distribution dimension changes with frequency scaling. For example, for Caps-SV3, the L-dimension distribution can achieve the best performance under 312.5MHz execution frequency; but the H-dimension distribution achieves the best performance when frequency increases to 937.5MHz. This indicates that the dimension selection is affected by not only the network configurations, but also the hardware configurations, e.g. processing frequency.

Table 5: Accuracy Validations for PIM-CapsNet

	Caps-MN1	Caps-MN2	Caps-MN3	Caps-CF1	Caps-CF2	Caps-CF3
Origin	99.75%	99.75%	99.75%	89.40%	90.03%	90.43%
w/o Accuracy Recovery	99.73%	99.74%	99.73%	89.15%	89.91%	90.08%
w/ Accuracy Recovery	99.75%	99.75%	99.75%	89.37%	90.02%	90.39%
	Caps-EN1	Caps-EN2	Caps-EN3	Caps-SV1	Caps-SV2	Caps-SV3
Origin	88.74%	85.01%	82.36%	96.70%	95.90%	95.90%
w/o Accuracy Recovery	88.19%	84.16%	81.64%	95.08%	95.92%	95.92%
Recovery	88.69%	84.96%	82.34%	96.42%	95.90%	95.90%

6.5 Overhead Analysis

Accuracy Analysis: Table 5 shows the absolute CapsNet accuracy after applying the approximations and accuracy recovery methodologies on our PIM-CapsNet designs as discussed in Sec.5.2.2. As it shows, the approximations cause on average 0.35% accuracy loss while the accuracy recovery method can achieve no accuracy loss for most of the cases, and only induces on average 0.04% accuracy difference across our benchmarks.

We also observe the slight accuracy boost for Caps-SV2 and Caps-SV3 when we increase the RP iteration numbers for PIM-CapsNet without accuracy recovery. This is because the noise induced by the approximation enhances the robustness of feature mapping given the enough RP iterations[69].

Area Analysis: Our PIM-CapsNet introduces 16 PEs and operation controller into each HMC vault architecture, with one RMAS module located in the logic layer. Based on our gate-level simulations, our logic design for 32 vaults and the RMAS together incurs $3.11mm^2$ area overheads under the $24nm$ process technology, which only occupy 0.32% HMC logic surface area.

Thermal Analysis: Note that our logic design raises the total power of the HMC, which could cause thermal issues for the 3D stacked memory architecture. We observe the average power overhead of our logic design is $2.24W$, which is below the maximum power overhead (i.e., $10W$ thermal design power (TDP) [70]) the HMC can tolerate.

7. RELATED WORKS

PIM-Based NN Acceleration: There have been multiple studies focus on exploring PIM-based neural network accelerators [71, 72, 73, 74, 75, 76]. For example, [72] employs the ReRAM-based PIM to conduct efficient neural network execution. However, the massive intermediate variables involved in the RP could induce both performance and energy overheads in the ReRAM design for frequent value updating. Besides, [43, 77, 42] propose the CNN accelerator design using 3D stack PIM technique. Since the execution pattern of the RP procedure are far different from CNN layers, previous logic layer designs of 3D stacked memory exhibit low efficiency on the RP execution. To our best knowledge, this is the first work that leverages HMC to explore a customized architectural design for efficient CapsNet acceleration.

Workload Distributions: Several studies have explored the workload distribution for efficient neural network execution [78, 79, 80, 81, 82]. For example, [80] proposes to distribute the parallel workloads of convolutional layer among the computation units within a single device; and [82] splits the convolutional execution and distribute the workloads into multiple devices. Their methods have achieved significant CNN accelerations via greatly improving the resource utilization. Since the execution of the RP procedure is much more complicated than the convolutional layer and involves strong execution dependence, these studies are not applicable to the CapsNet acceleration.

Exponential Approximation: There are also some works on approximation for exponential function [58, 59, 83, 84]. For example, [59] leverages the lookup table to conduct the fast exponential execution, which causes substantial performance and area overheads. [84] accelerates exponential function via software optimizations, which are hard to be implemented via the simple logic design. In this work, we conduct the efficient acceleration for exponential function (Sec.5.2.2) with low design complexity and low power overhead.

8. CONCLUSIONS

In recent years, the CapsNet has outperformed the CNN on the image processing tasks and becomes increasing popular in many areas. In this study, we propose a processing-in-memory based hybrid computing design called PIM-CapsNet, which leverages both GPU core computing capability and the special features of the hybrid memory cube to effectively process the data-intensive RP operations, achieving substantial improvements on both performance and energy saving. To drastically reduce the identified performance bottlenecks of RP, we propose several memory-level optimizations (e,g., multi-dimensional parallelism selection, intelligent workload distribution, complex logic approximation and new address mapping mechanism) based on RP's program and execution features to enable minimal in-memory communication, maximum parallelization, and low design complexity and overhead. Evaluation results demonstrate that our proposed design achieves significant performance speedup and energy savings over the baseline GPU design, for both the RP phase and overall CapsNet inference. It also achieves good performance scalability with increasing network size.

9. REFERENCES

[1] I. Kononenko, "Machine learning for medical diagnosis: history, state of the art and perspective," *Artificial Intelligence in medicine*, vol. 23, no. 1, pp. 89–109, 2001.

[2] B. J. Erickson, P. Korfiatis, Z. Akkus, and T. L. Kline, "Machine learning for medical imaging," *Radiographics*, vol. 37, no. 2, pp. 505–515, 2017.

[3] A. L. Buczak and E. Guven, "A survey of data mining and machine learning methods for cyber security intrusion detection," *IEEE Communications Surveys & Tutorials*, vol. 18, no. 2, pp. 1153–1176, 2016.

[4] J. Grimmer, "We are all social scientists now: How big data, machine learning, and causal inference work together," *PS: Political Science & Politics*, vol. 48, no. 1, pp. 80–83, 2015.

[5] J. Kober, J. A. Bagnell, and J. Peters, "Reinforcement learning in robotics: A survey," *The International Journal of Robotics Research*, vol. 32, no. 11, pp. 1238–1274, 2013.

[6] S. Ren, K. He, R. Girshick, and J. Sun, "Faster r-cnn: Towards real-time object detection with region proposal networks," in *Advances in neural information processing systems*, pp. 91–99, 2015.

[7] F. N. Iandola, S. Han, M. W. Moskewicz, K. Ashraf, W. J. Dally, and K. Keutzer, "Squeezenet: Alexnet-level accuracy with 50x fewer parameters and< 0.5 mb model size," *arXiv preprint arXiv:1602.07360*, 2016.

[8] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich,

"Going deeper with convolutions," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 1–9, 2015.

[9] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.

[10] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 770–778, 2016.

[11] G. E. Hinton, A. Krizhevsky, and S. D. Wang, "Transforming auto-encoders," in *International Conference on Artificial Neural Networks*, pp. 44–51, Springer, 2011.

[12] G. Hinton, "Where do features come from?," *Cognitive science*, vol. 38, no. 6, pp. 1078–1101, 2014.

[13] M. Jaderberg, K. Simonyan, A. Zisserman, *et al.*, "Spatial transformer networks," in *Advances in neural information processing systems*, pp. 2017–2025, 2015.

[14] S. Sabour, N. Frosst, and G. E. Hinton, "Dynamic routing between capsules," in *Advances in neural information processing systems*, pp. 3856–3866, 2017.

[15] A. Jiménez-Sánchez, S. Albarqouni, and D. Mateus, "Capsule networks against medical imaging data challenges," in *Intravascular Imaging and Computer Assisted Stenting and Large-Scale Annotation of Biomedical Data and Expert Label Synthesis*, pp. 150–160, Springer, 2018.

[16] P. Afshar, K. N. Plataniotis, and A. Mohammadi, "Capsule networks for brain tumor classification based on mri images and course tumor boundaries," *arXiv preprint arXiv:1811.00597*, 2018.

[17] A. Mobiny and H. Van Nguyen, "Fast capsnet for lung cancer screening," in *International Conference on Medical Image Computing and Computer-Assisted Intervention*, pp. 741–749, Springer, 2018.

[18] A. S. Lundervold and A. Lundervold, "An overview of deep learning in medical imaging focusing on mri," *Zeitschrift für Medizinische Physik*, 2018.

[19] X. Zhang and S. Zhao, "Blood cell image classification based on image segmentation preprocessing and capsnet network model," *Journal of Medical Imaging and Health Informatics*, vol. 9, no. 1, pp. 159–166, 2019.

[20] L. Wang, R. Nie, R. Xin, J. Zhang, and J. Cai, "sccapsnet: a deep learning classifier with the capability of interpretable feature extraction, applicable for single cell rna data analysis," *bioRxiv*, p. 506642, 2018.

[21] A. D. Kumar, "Novel deep learning model for traffic sign detection using capsule networks," *arXiv preprint arXiv:1805.04424*, 2018.

[22] J. Kronenberger and A. Haselhoff, "Do capsule networks solve the problem of rotation invariance for traffic sign classification?," in *International Conference on Artificial Neural Networks*, pp. 33–40, Springer, 2018.

[23] M. Pöpperl, R. Gulagundi, S. Yogamani, and S. Milz, "Capsule neural network based height classification using low-cost automotive ultrasonic sensors," *arXiv preprint arXiv:1902.09839*, 2019.

[24] D. A. Lopez, *Evolving GPU-Accelerated Capsule Networks*. PhD thesis, 2018.

[25] A. Yusuf and S. Alawneh, "A survey of gpu implementations for hyperspectral image classification in remote sensing," *Canadian Journal of Remote Sensing*, pp. 1–19, 2018.

[26] A. Stoutchinin, F. Conti, and L. Benini, "Optimally scheduling cnn convolutions for efficient memory access," *arXiv preprint arXiv:1902.01492*, 2019.

[27] C. Li, Y. Yang, M. Feng, S. Chakradhar, and H. Zhou, "Optimizing memory efficiency for deep convolutional neural networks on gpus," in *SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 633–644, IEEE, 2016.

[28] M. Song, Y. Hu, H. Chen, and T. Li, "Towards pervasive and user satisfactory cnn across gpu microarchitectures," in *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 1–12, IEEE, 2017.

[29] Z. Chen, L. Luo, W. Quan, Y. Shi, J. Yu, M. Wen, and C. Zhang, "Multiple cnn-based tasks scheduling across shared gpu platform in research and development scenarios," in *2018 IEEE 20th International Conference on High Performance Computing and Communications; IEEE 16th International Conference on Smart City; IEEE 4th International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*, pp. 578–585, IEEE, 2018.

[30] X. Zhang, C. Xie, J. Wang, W. Zhang, and X. Fu, "Towards memory friendly long-short term memory networks (lstms) on mobile gpus," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 162–174, IEEE, 2018.

[31] Y. Chen, T. Luo, S. Liu, S. Zhang, L. He, J. Wang, L. Li, T. Chen, Z. Xu, N. Sun, *et al.*, "Dadiannao: A machine-learning supercomputer," in *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 609–622, IEEE Computer Society, 2014.

[32] Z. Du, R. Fasthuber, T. Chen, P. Ienne, L. Li, T. Luo, X. Feng, Y. Chen, and O. Temam, "Shidiannao: Shifting vision processing closer to the sensor," in *ACM SIGARCH Computer Architecture News*, vol. 43, pp. 92–104, ACM, 2015.

[33] X. Zhou, Z. Du, Q. Guo, S. Liu, C. Liu, C. Wang, X. Zhou, L. Li, T. Chen, and Y. Chen, "Cambricon-s: Addressing irregularity in sparse neural networks through a cooperative software/hardware approach," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 15–28, IEEE, 2018.

[34] M. Mahmoud, K. Siu, and A. Moshovos, "Diffy: a déjà vu-free differential deep neural network accelerator," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 134–147, IEEE, 2018.

[35] Y. Kwon and M. Rhu, "Beyond the memory wall: A case for memory-centric hpc system for deep learning," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 148–161, IEEE, 2018.

[36] Y. Li, J. Park, M. Alian, Y. Yuan, Z. Qu, P. Pan, R. Wang, A. Schwing, H. Esmaeilzadeh, and N. S. Kim, "A network-centric hardware/algorithm co-design to accelerate distributed training of deep neural networks," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 175–188, IEEE, 2018.

[37] C. Deng, S. Liao, Y. Xie, K. Parhi, X. Qian, and B. Yuan, "Permdnn: Efficient compressed deep neural network architecture with permuted diagonal matrices," in *MICRO*, 2018.

[38] J. Jeddeloh and B. Keeth, "Hybrid memory cube new dram architecture increases density and performance," in *2012 symposium on VLSI technology (VLSIT)*, pp. 87–88, IEEE, 2012.

[39] S. F. Yitbarek, T. Yang, R. Das, and T. Austin, "Exploring specialized near-memory processing for data intensive operations," in *2016 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 1449–1452, IEEE, 2016.

[40] J. Ahn, S. Yoo, O. Mutlu, and K. Choi, "Pim-enabled instructions: a low-overhead, locality-aware processing-in-memory architecture," in *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*, pp. 336–348, IEEE, 2015.

[41] J. Ahn, S. Hong, S. Yoo, O. Mutlu, and K. Choi, "A scalable processing-in-memory accelerator for parallel graph processing," *ACM SIGARCH Computer Architecture News*, vol. 43, no. 3, pp. 105–117, 2016.

[42] J. Liu, H. Zhao, M. A. Ogleari, D. Li, and J. Zhao, "Processing-in-memory for energy-efficient neural network training: A heterogeneous approach," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 655–668, IEEE, 2018.

[43] D. Kim, J. Kung, S. Chai, S. Yalamanchili, and

S. Mukhopadhyay, "Neurocube: A programmable digital neuromorphic architecture with high-density 3d memory," in *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pp. 380–392, IEEE, 2016.

[44] G. E. Hinton, S. Sabour, and N. Frosst, "Matrix capsules with em routing," 2018.

[45] E. Xi, S. Bing, and Y. Jin, "Capsule network performance on complex data," *arXiv preprint arXiv:1712.03480*, 2017.

[46] S. S. R. Phaye, A. Sikka, A. Dhall, and D. Bathula, "Dense and diverse capsule networks: Making the capsules learn better," *arXiv preprint arXiv:1805.04001*, 2018.

[47] Y. LeCun, L. Bottou, Y. Bengio, P. Haffner, *et al.*, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[48] A. Krizhevsky and G. Hinton, "Learning multiple layers of features from tiny images," tech. rep., Citeseer, 2009.

[49] G. Cohen, S. Afshar, J. Tapson, and A. van Schaik, "Emnist: an extension of mnist to handwritten letters," *arXiv preprint arXiv:1702.05373*, 2017.

[50] Y. Netzer, T. Wang, A. Coates, A. Bissacco, B. Wu, and A. Y. Ng, "Reading digits in natural images with unsupervised feature learning," 2011.

[51] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer, "Automatic differentiation in pytorch," 2017.

[52] S. Chetlur, C. Woolley, P. Vandermersch, J. Cohen, J. Tran, B. Catanzaro, and E. Shelhamer, "cudnn: Efficient primitives for deep learning," *arXiv preprint arXiv:1410.0759*, 2014.

[53] "Optimizations To Accelerate Deep Learning Training on NVIDIA GPUs." `https://devblogs.nvidia.com/new-optimizations-accelerate-deep-learning-training-gpu/`.

[54] "Nvidia Profiler." `https://docs.nvidia.com/cuda/profiler-users-guide/index.html`.

[55] R. Mukhometzianov and J. Carrillo, "Capsnet comparative performance evaluation for image classification," *arXiv preprint arXiv:1805.11195*, 2018.

[56] "HMC Specification 2.1." `http://hybridmemorycube.org/files/SiteDownloads/HMC-30G-VSR_HMCC_Specification_Rev2.1_20151105.pdf`.

[57] D.-I. Jeon, K.-B. Park, and K.-S. Chung, "Hmc-mac: Processing-in memory architecture for multiply-accumulate operations with hybrid memory cube," *IEEE Computer Architecture Letters*, vol. 17, no. 1, pp. 5–8, 2018.

[58] D. De Caro, N. Petra, and A. G. Strollo, "High-performance special function unit for programmable 3-d graphics processors," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 56, no. 9, pp. 1968–1978, 2009.

[59] J. Partzsch, S. Höppner, M. Eberlein, R. Schüffny, C. Mayr, D. R. Lester, and S. Furber, "A fixed point exponential function accelerator for a neuromorphic many-core system," in *2017 IEEE International Symposium on Circuits and Systems (ISCAS)*, pp. 1–4, IEEE, 2017.

[60] C. Lomont, "Fast inverse square root," *Tech-315 nical Report*, vol. 32, 2003.

[61] M. Robertson, *A brief history of invsqrt*. Department of Computer Science & Applied Statistics, 2012.

[62] A. Zoglauer, S. E. Boggs, M. Galloway, M. Amman, P. N. Luke, and R. M. Kippen, "Design, implementation, and optimization of megalib's image reconstruction tool mimrec," *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, vol. 652, no. 1, pp. 568–571, 2011.

[63] L. Middendorf and C. Haubelt, "A programmable graphics processor based on partial stream rewriting," in *Computer Graphics Forum*, vol. 32, pp. 325–334, Wiley Online Library, 2013.

[64] W. Kahan, "Ieee standard 754 for binary floating-point arithmetic," *Lecture Notes on the Status of IEEE*, vol. 754, no. 94720-1776, p. 11, 1996.

[65] "Nvidia Tesla P100 Whitepaper." `https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf`.

[66] "NVIDIA-smi." `https://developer.nvidia.com/nvidia-system-management-interface`.

[67] J. D. Leidel and Y. Chen, "Hmc-sim: A simulation framework for hybrid memory cube devices," *Parallel Processing Letters*, vol. 24, no. 04, p. 1442002, 2014.

[68] "Gate-Level Simulation Methodology - Cadence." `https://www.cadence.com/content/dam/cadence-www/global/en_US/documents/tools/system-design-verification/gate-level-simulation-wp.pdf`.

[69] P. Koistinen and L. Holmström, "Kernel regression and backpropagation training with noise," in *Advances in Neural Information Processing Systems*, pp. 1033–1039, 1992.

[70] D. Zhang, N. Jayasena, A. Lyashevsky, J. L. Greathouse, L. Xu, and M. Ignatowski, "Top-pim: throughput-oriented programmable processing in memory," in *Proceedings of the 23rd international symposium on High-performance parallel and distributed computing*, pp. 85–98, ACM, 2014.

[71] P. Chi, S. Li, C. Xu, T. Zhang, J. Zhao, Y. Liu, Y. Wang, and Y. Xie, "Prime: A novel processing-in-memory architecture for neural network computation in reram-based main memory," in *ACM SIGARCH Computer Architecture News*, vol. 44, pp. 27–39, IEEE Press, 2016.

[72] A. Shafiee, A. Nag, N. Muralimanohar, R. Balasubramonian, J. P. Strachan, M. Hu, R. S. Williams, and V. Srikumar, "Isaac: A convolutional neural network accelerator with in-situ analog arithmetic in crossbars," *ACM SIGARCH Computer Architecture News*, vol. 44, no. 3, pp. 14–26, 2016.

[73] F. Chen, L. Song, and Y. Chen, "Regan: A pipelined reram-based accelerator for generative adversarial networks," in *2018 23rd Asia and South Pacific Design Automation Conference (ASP-DAC)*, pp. 178–183, IEEE, 2018.

[74] H. Mao, M. Song, T. Li, Y. Dai, and J. Shu, "Lergan: A zero-free, low data movement and pim-based gan architecture," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 669–681, IEEE, 2018.

[75] B. K. Joardar, B. Li, J. R. Doppa, H. Li, P. P. Pande, and K. Chakrabarty, "Regent: A heterogeneous reram/gpu-based architecture enabled by noc for training cnns," in *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 522–527, IEEE, 2019.

[76] H. Falahati, P. Lotfi-Kamran, M. Sadrosadati, and H. Sarbazi-Azad, "Origami: A heterogeneous split architecture for in-memory acceleration of learning," *arXiv preprint arXiv:1812.11473*, 2018.

[77] M. Gao, J. Pu, X. Yang, M. Horowitz, and C. Kozyrakis, "Tetris: Scalable and efficient neural network acceleration with 3d memory," in *ACM SIGARCH Computer Architecture News*, vol. 45, pp. 751–764, ACM, 2017.

[78] Y. Shen, M. Ferdman, and P. Milder, "Maximizing cnn accelerator efficiency through resource partitioning," in *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, pp. 535–547, IEEE, 2017.

[79] J. Guo, S. Yin, P. Ouyang, L. Liu, and S. Wei, "Bit-width based resource partitioning for cnn acceleration on fpga," in *2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pp. 31–31, IEEE, 2017.

[80] S. Wang, G. Ananthanarayanan, Y. Zeng, N. Goel, A. Pathania, and T. Mitra, "High-throughput cnn inference on embedded arm big. little multi-core processors," *arXiv preprint arXiv:1903.05898*, 2019.

[81] C. Lo, Y.-Y. Su, C.-Y. Lee, and S.-C. Chang, "A dynamic deep neural network design for efficient workload allocation

in edge computing," in *2017 IEEE International Conference on Computer Design (ICCD)*, pp. 273–280, IEEE, 2017.

[82] S. Dey, A. Mukherjee, A. Pal, and P. Balamuralidhar, "Partitioning of cnn models for execution on fog devices," in *Proceedings of the 1st ACM International Workshop on Smart Cities and Fog Computing*, pp. 19–24, ACM, 2018.

[83] A. C. I. Malossi, Y. Ineichen, C. Bekas, and A. Curioni, "Fast exponential computation on simd architectures," *Proc. of HIPEAC-WAPCO, Amsterdam NL*, 2015.

[84] F. Perini and R. D. Reitz, "Fast approximations of exponential and logarithm functions combined with efficient storage/retrieval for combustion kinetics calculations," *Combustion and Flame*, vol. 194, pp. 37–51, 2018.