NeuralHMC: An Efficient HMC-Based Accelerator for Deep Neural Networks

Chuhan Min
University of Pittsburgh
Pittsburgh, PA, U.S.A
chm114@pitt.edu

Jiachen Mao, Hai Li, Yiran Chen
Duke University
Durham, North Carolina, U.S.A
jiachen.mao, hai.li, yiran.chen@duke.edu

ABSTRACT

In Deep Neural Network (DNN) applications, energy consumption and performance cost of moving data between memory hierarchy and computational units are significantly higher than that of the computation itself. Process-in-memory (PIM) architecture such as Hybrid Memory Cube (HMC), becomes an excellent candidate to improve the data locality for efficient DNN execution. However, it's still hard to efficiently deploy large-scale matrix computation in DNN on HMC because of its coarse grained packet protocol. In this work, we propose *NeuralHMC*, the first HMC-based accelerator tailored for efficient DNN execution. Experimental results show that *NeuralHMC* reduces the data movement by 1.4× to 2.5× (depending on the DNN data reuse strategy) compared to *Von Neumann* architecture. Furthermore, compared to state-of-the-art PIM-based DNN accelerator, *NeuralHMC* can promisingly improve the system performance by 4.1× and reduces energy by 1.5×, on average.

KEYWORDS

Hybrid Memory Cube; processing-in-memory; simulation

ACM Reference Format:
Chuhan Min and Jiachen Mao, Hai Li, Yiran Chen. 2019. NeuralHMC: An Efficient HMC-Based Accelerator for Deep Neural Networks. In *ASPDAC '19: 24th Asia and South Pacific Design Automation Conference (ASPDAC '19), January 21–24, 2019, Tokyo, Japan.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3287624.3287642

1 INTRODUCTION

Deep Neural networks (DNNs) have demonstrated great potential in tasks such as object detection, recognition, and classification. In many benchmark suites [9, 15], DNN models even obtained an accuracy higher than the level that humans can achieve. However, a typical DNN contains thousands of network layers and hundreds of millions of parameters [12, 14]. Data movement between different DNN layers incurs large number of memory accesses.

To overcome the memory bottleneck during DNN execution, many DNN accelerators are proposed to improve *data reuse* [4, 11] and *data locality* [3]. To improve *data reuse*, an on-chip scratchpad memory is introduced in [3] to support data reuse of local accelerators. In [11], many processing elements (PEs) are organized as a systolic array to allow temporal data reuse among the PEs. To improve *data locality*, Processing-in-Memory (PIM) structure is recently adopted in DNN acceleration, eliminating the costly data movement between memory and computation host. PRIME [5], for example, utilizes resistive memory to store the data and performs the DNN executions directly on the local data.

In an attempt to further improve *data locality*, memory manufacturers have invented 3D-stacked memory where multiple layers of memory arrays are stacked on top of each other [6]. One prominent example is Hybrid Memory Cube (HMC), which was announced by Micron Technology in 2011 [6]. Inherited from the concept of PIM, some 3D-stacked memory architectures [6] also include a logic layer that can integrate general-purpose computational logic directly within main memory to take advantages of high internal bandwidth during computation.

Although HMC-based PIM designs significantly reduce data movements between the memory and the computation host, challenges still exist before applying them to DNN applications:

- HMC utilize a Network-on-Chip (NoC) to connect their internal structural elements. As pointed out in [7], inter-vault data movement overhead increases with the degree of computational parallelism.
- Unique features of HMC (e.g., packet-based protocol, unidirectional lane, internal queuing characteristics, etc.) largely constrain the memory bandwidth utilization.

In this work, we propose *NeuralHMC*, the first HMC-based accelerator for efficient DNN execution. Our major contributions are:

- We analyze data movement overhead of multiple NoC designs with different DNN data reuse strategies and adopt the optimal one in *NeuralHMC* for parallel multi-HMC scheme.
- We propose a weight sharing MAC to reduce weight data access and a packet scheduling method with pipelined decoder to maximize memory bandwidth utilization.
- We add multi-HMC support in HMC-MAC simulator and test *NeuralHMC* with respect to energy consumption and performance. Experimental results shows that *NeuralHMC* achieves both higher energy efficiency and better execution performance when compared with the state-of-the-art PIM accelerator design built with DDRx [4].

The rest of this paper is organized as follows: Section 2 introduces HMC architecture; Section 3 illustrates the motivation of *NeuralHMC*; Section 4 describes the design details of *NeuralHMC*; Section 5 shows the experimental setup and evaluation result; Section 6 concludes this work.

ASPDAC '19, January 21–24, 2019, Tokyo, Japan

ACM ISBN 978-1-4503-6007-4/19/01…$15.00
https://doi.org/10.1145/3287624.3287642

Table 1: HMC Specification.

	Configuration
Memory density	8GB (8 memory layer)
Memory per bank	16MB
# of external links	2, 4
Link lane speed (Gb/s)	12.5, 15, 25, 28, 30
# of quadrants	4
# of vaults/quadrant	8
# of partitions/vault	8
# of memory banks/partition	2
Max DRAM data bandwidth	320GB/s (2.56Tb/s)
Max vault data bandwidth	10GB/s (80Gb/s)
Max cubes connectable	8

2 BACKGROUND

2.1 Overview of HMC

Figure 1 illustrates a typical organization of HMC architecture. HMC consists of up to eight DRAM dies stacked on top of a logic die and vertically connected by 512 Through-Silicon Vias (TSVs). As shown in Figure 1, each layer in HMC is divided into 16 partitions and every partition is a vault with a corresponding vault controller in the logic layer. A vault employs a 32-byte DRAM data bus using 32 TSVs. A group of eight vaults composes a quadrant that is connected to a shared external full duplex serialized link. Our work adopts HMC2.0 specification as shown in Table 1.

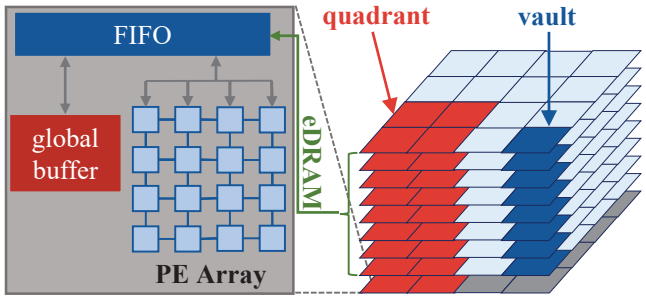

Figure 1: HMC module architecture.

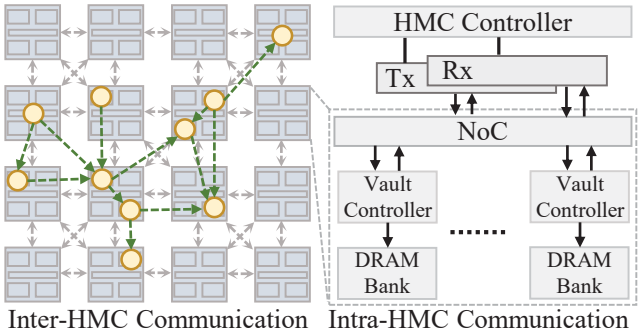

Figure 2: HMC communication.

2.2 HMC Communication

The HMC interface utilizes a packet-based communication protocol. An HMC has two or four external links to connect to other HMCs or hosts. Each independent link is connected to a quadrant that is internally connected to other quadrants, which routes the packets to their corresponding vaults. As a result of that, accessing a local vault in the same quadrant has a shorter latency than accessing a vault in another quadrant.

There exist two levels of communications in the whole HMC architecture, including (1) inter-HMC communication that is performed on the switch path and (2) intra-HMC communication that is handled by the HMC controller.

For inter-HMC communication, as depicted in the left part of Figure 2, each HMC has four serialized links with a packet-based protocol. Traditionally, the off-chip controller generates the packet of memory requests in a coarse-grained manner. Such a scheme, however, results in low communication bandwidth utilization and large performance degradation in multi-HMC environment because (1) the communication latency differs between the near and the distant quadrants and (2) packets have to be decoded before accessing the destination quadrant. A customized switch design is highly desired to reduce the overhead of inter-HMC data movement and to improve the scalability of multi-HMC.

For intra-HMC communication, the vault controllers are connected by an internal NoC, which is shown in the right part of Figure 2. At 1.25GHz execution frequency [16], HMC supplies a maximum bit-rate of 30 Gbit/s and 480 Gbit/s in transmission (Tx) and receive (Rx) directions, respectively, at each of the 16 link lanes. As a result, total 384 bits can be transferred between memory dies and switch per cycle. The TSV bit-width is assumed to be 32 bits (e.g., 32 TSV data lanes) and the bit-rate is 2.5 Gbit/s. Hence, the bandwidth of the TSV bus is 64 bits per cycle at the execution frequency of 1.25GHz.

In *NeuralHMC*, a packet scheduling scheme is introduced to improve the efficiency of HMC communication, which will be detailed in Section 4 and different external NoC designs are examined with DNN data reuse strategies in Section 5.

3 MOTIVATION

3.1 Dataflows in DNN Accelerators

It has been proven that the execution of a DNN is composed of many multiplications and additions, which can be accelerated using dedicated accelerators [17]. The dataflow graph for the DNN computations can be mapped onto a PE array in multiple ways, leading to different dataflow characteristics. We follow the taxonomy introduced in Eyeriss [4]. Eyeriss divides a DNN accelerator design into the following three key components:

- Weight Stationary (WS): In a WS accelerator, each PE fetches a unique weight from the global buffer (GB) and retains it until the PE completes all the calculations involving that weight. GB transfers input activations via a broadcast to each PE. The PEs may forward psums back to the GB (awaiting to be redistributed later), or accumulate them locally within the PE array.
- Output Stationary (OS): An OS accelerator maps an output pixel on to a PE in every iteration. Each PE fetches both

weights and input activations from the GB and accumulates partial sums internally. When the accumulation completes or an output activation is generated, each PE sends the output activation to the GB.

- Row Stationary (RS): A RS accelerator maps a row of partial sum calculations to a column of the PE array to facilitate data reuse of weights and input activations. Partial sums are accumulated by forwarding the computation along the column, and the PEs at the top of the column send the final output activations to the GB.

3.2 Potential of DNN Execution on HMC

A recent work named HMC-MAC [10] was proposed to offload multiply-accumulate (MAC) functions to logic layer in HMC without major modifications of HMC structure and control logic in the vault. According to their simulation result, the execution time of MAC operations is stabilized around 50ns. Because the MAC operation in [10] is carried out in parallel under the HMC-MAC architecture with the support of parallel vault operations, bank interleaving and data block accesses. According to HMC specification, up to 128KB of data, which is the product of the number of vaults (32), the number of banks (16) and the maximum block size (256B), can be processed in parallel in a HMC.

Such high efficiency of HMC execution inspires us to exploit the execution efficiency of HMC on DNN-related applications, which has rarely been exploited in previous arts. Take the specification of AlexNet [12] as an example, the numbers of parameters of each layer and the associated MAC operations are summarized in Table 2. Assume the bit width of the MAC is 16B (data type: long long), the computation time of 106M MAC ops in CONV1 layer is within a second (not including the data movement to the PEs). This observation shows that HMC-MAC is an effective PIM architecture to accelerate the MAC operations in DNN applications. When the scale of the neural networks increases, a larger parallelism can be exploited with a multi-HMC structure.

3.3 Challenge of DNN Execution on HMC

However, the above estimation in Section 3.2 is too optimistic and ignores the communication cost and data access latency from/to the PEs. As aforementioned in Section 2, HMC utilizes a packet-based protocol. The total data access latency largely depends on the packet processing and response generation steps in the HMC communication.

Dense matrix multiplication in DNN execution exhibits a high fine-grained parallelism and is computation-extensive, e.g., the ratio between computations and memory accesses is high. As illustrated in [10], to ensure one memory request can only access a single vault, one memory request is regenerated into multiple requests. In consequence, final result is the accumulation over multiple vaults per request. When a regenerated memory request arrives at the vault controller, this memory request is stored in the request buffer and then converted into a DRAM command. The vault controller

Table 2: AlexNet Architecture Overview.

Layer name	CONV1	CONV2	CONV3	CONV4	CONV5	FC6	FC7	FC8
Parameter #	35K	307K	884K	1.3M	442K	37M	16M	4M
`# of MAC Operations`	106M	448M	150M	224M	150M	37M	16M	4M

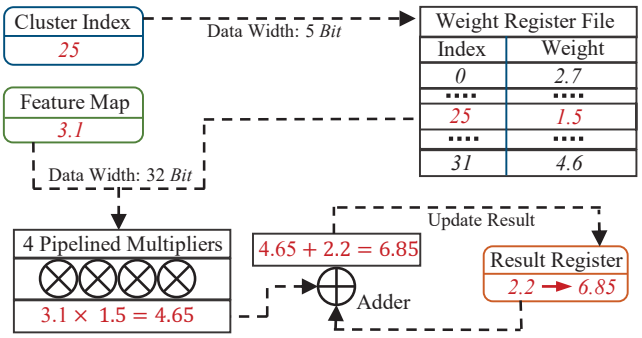

Figure 3: Weight sharing pipelined MAC design.

works as a conventional memory controller and issues this DRAM command. In addition, memory requests that access the same address are processed in the order of their arrival to guarantee the memory data coherency, similarly to the FR-FCFS scheduling. Constrained from the memory power consumption, there can exist only two active HMCs. As a result, the parallelism of packet decoder is limited. Hence, in Section 4.3, we optimize the multi-HMC by decoupling packet decoding and memory access with an always-on HMC.

4 ACCELERATOR DESIGN OF NEURALHMC

4.1 Weight Sharing Pipelined MAC Design

The size of weights in modern large-scale DNNs is growing fast, which accounts for a large amount of memory access and thus introduces long memory access latency. To tackle this problem, in this work, we innovatively adopt the weight sharing technology in [8] to our DNN accelerator design. Weight sharing quantifies the original DNN weights into several clusters and the clusters can be represented as cluster index, which is typically 5 bits for fully-connected layers and 8 bits for convolutional layers. Research shows that weight sharing incur no accuracy loss under most scenarios [8]. Such representation reduces the original 32 bits floating points weights to 5 or 8 bits cluster index, saving much memory consumption.

Figure 3 depicts the details of our proposed weight sharing pipelined MAC design. First, we use the cluster index to locate the quantized weights in the weight register file in O(1) time complexity. The weight register file is implemented in the logic layer and connected to the FIFO. Then, as shown in Figure 3, the feature map and the quantized weight are fed into the pipelined multipliers. Because multiplication require 4× of cycles required for adder, we leverage 4 multipliers to amortize the workload so that the whole MAC can generate one result for each cycle. After the available multiplier get the result, it will be added with the former accumulated results in an efficient way.

4.2 Asynchronous Packet Communication

The HMC controller uses three types of packets: flow control, request, and response packets. The packet control layout is shown in Figure 4. The communication between the host and the PIM is asynchronous with the assistant of flow control packet. As also demonstrated in Figure 4, the total size of register for instruction in the targeted PIM equals multiplication of register size, number of register files and number of active threads. The size of register

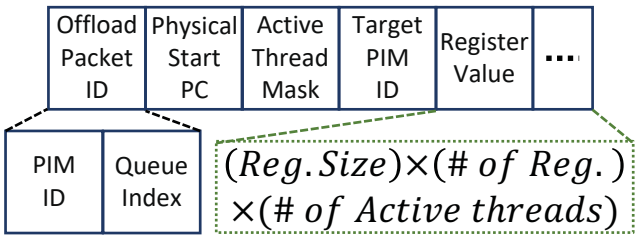

Figure 4: Flow control packet layout.

denotes the number of instructions in PIM instruction set. In this way, the flow packet can deliver instruction directly to PIM so long as the ISA is compatible between CPU and PIM.

A flow control packet is transmitted via a master-slave link. We aim to reduce data transferring cost between the CPU and the PIM and allow them to operate independently and simultaneously. When offloading computations to PIM, the flow packet is generated with the corresponding thread id of the CPU encapsulated in packet header. Figure 5 illustrates the asynchronous communication between the host and the PIM. Calculations on the CPU and the PIM are performed simultaneously in Figure 5. Furthermore, the data transmission between the CPU and the PIM can be hidden behind the calculations. After the computation completes, the PIM will update the flow packet in the tail and transmit it via the link layer to the CPU. Upon receiving the flow packet from PIM, the CPU checks the completion bit in the tail and thread id in the header.

The consecutive memory data of an HMC-MAC memory request may access more than one vault when the execution count is relatively big. In such a case, each memory request will regenerate its memory request as one memory request can access only one vault. Decoding a packet might introduce extra processing latency if it retrieves multiple memory requests in serial. The packet decoder in the off-chip controller is also enhanced to support packets: it firstly decodes the universal header to obtain the size of the packet (which indicates the number of the memory requests) as well as the request type; it then retrieves the address and granularity information of each memory request.

Two optimizations are conducted to reduce the decoding latency:

- Multiple memory requests can be decoded in parallel while the offset of each memory request in a packet can be efficiently calculated in advance because the size of the ADDR field in the packet tail is fixed;

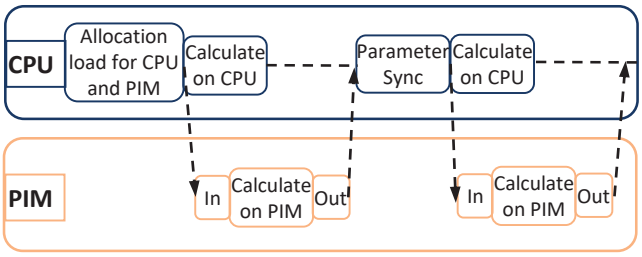

Figure 5: Asynchronous parameter communication between host and PIM.

Algorithm 1 Packet Scheduling Algorithm

procedure PACKET SCHEDULING ALGORITHM (*Global_Cycle, Request_Time, Active_HMC, Sleep_HMC*)

1: **while** $Req\_Queue \neq \emptyset$ and $Active\_HMC < HMC\_Cap$ **do**
2: $Global\_Cycle++$.
3: **for** *Req* in *Req_Queue* **do**
4: $Req\_Waiting++$
5: **if** exist $Req\_Waiting > Starvation\_Thd$ **then**
6: $Victim\_HMC \leftarrow$ dequeue *Req*
7: wakeup(*Victim_HMC*)
8: $Sleep\_HMC \leftarrow Sleep\_HMC - Victim\_HMC$
9: $Active\_HMC \leftarrow Active\_HMC \cup Victim\_HMC$
10: **if** $Req\_Queue \neq \emptyset$ and $\exists Free\_HMC \in Active\_HMC$ **then**
11: $Victim\_HMC \leftarrow$ LRU(*Free_HMC*)
12: sleep(*Victim_HMC*)
13: $Active\_HMC \leftarrow Active\_HMC - Victim\_HMC$
14: $Sleep\_HMC \leftarrow Sleep\_HMC \cup Victim\_HMC$
Return =0

- A packet can be decoded before it is received completely in a pipeline manner: decoding the memory requests received from the previous packet and receiving the next packet can be performed at the same time.

4.3 Packet Scheduling Algorithm

In a traditional multi-HMC system, the memory power budget allows only two active HMCs at the same time [6]. We denote the ready packet as the packet destined to the active HMC. The starvation time is defined as the waiting time for the next ready packet in queue.

Assuming there are two packets – one is destined to HMC 0 and the other destined to HMC 1. HMC 0 is initially active while HMC 1 is in sleep mode. After the completion of the first packet, the second memory request must wait until the power manager turns off HMC 0 and then turns on HMC 1. In this situation, the decoding process is serialized, incuring long memory access latency.

Therefore, we proposed Algorithm 1 in *NeuralHMC* to deal with the aforementioned situation. In the above situation, when the request queue is drained and the waiting time for the next ready request (Req_waiting in Algorithm 1) exceeds the starvation time (Starvation_Thd in Algorithm 1), the second packet is issued to active HMC 0. The header of the next not-ready packet (CUBID) is decoded in HMC 0 and then the sleep of HMC 0 is issued. In the next cycle, the activation of the target HMC (HMC 1) is issued. During the activation of the target HMC, the packet is forwarded to HMC 2 that is always active and the packet body is decoded. The data in HMC 1 is then retrieved once it's activated. In this way, the packet decoding and the memory access are decoupled from each other, the packet decoding is pipelined with the activation/sleep of HMC.

In traditional HMC-MAC implementation, the multi-request packets are regenerated as multiple fine-grained packets to perform accumulation. This packet regeneration process incurs additional computation cost. In our DNN accelerator design, we optimize this

Table 3: Power Consumption

Component	Power
DRAM power	$P_{dram}(B) = 7.9W + B \times 21.5Ws/GB$
Cube power	$P = P_{dram}(B) + K \times 165pJ \times f$

packet regeneration process by avoiding across vault memory access in the first place when composing a packet in on-chip HMC controller. By keeping a page table of starting address of each vault in on-chip controller, the maximum number of MAC operation is computed given the start address of the memory operand.

5 EXPERIMENTS

5.1 Experimental Setup

The power model adopted in *NeuralHMC* is based on [1], which can be summarized in Table 3. Here B is the requested bandwidth, K is the number of clusters and f is the clock frequency. Experiments in this section are performed on an a cycle-accurate simulator HMC-MAC [10], which has been modified to adopt multi-HMC backend in HMC-Sim [13]. In the multi-HMC simulator, an undirected graph depicts the link connections among the HMCs and the host. The memory trace file is generated by Gem5 [2] – a full system memory simulator, then loaded into HAC-MAC by the trace loader module. The activating/sleep latency is set to $2\mu s$ and the timing configuration of the simulator is shown in Table 4.

5.2 Evaluation of Single-HMC in NeuralHMC

In single-HMC schemes, we evaluate our proposed weight-sharing pipelined MAC optimization in HMC-MAC.

Figure 6 depicts the scalability of the single-HMC's throughput when MAC ops/fmap increases. We evaluated 3 neural network layer types in DNN, including convolution layer (CONV), fully-connected layer (FC) and 1x1 convolution layer (1x1-CONV). With the MAC operations per feature map scaling, the throughput does not scale in linear. In CONV, FC and 1x1-CONV, number of MAC operations per output ranges from 100 to 1000. The saturation of the throughput when the MAC ops/fmap increases is because the computation parallelism is also constrained by the number of vaults (i.e. MAC units).

Figure 7 is the breakdown of the operations in AlexNet executions, including the MAC operations, the data accesses from global buffer (GB) to PE and the data access from PE to PE. The results of three data reuse strategies – WR, RS, and OS are all included. Note that only RS involves the data transfer from PE to PE. Because the partial sums are accumulated by forwarding the computation along the column. The accumulation result is transferred to GB per MAC

Table 4: HMC timing configuration.

tCK	0.8ns
tRAS, tRCD, tRRD, tRC, tRP	27, 13, 4, 10, 10
tCCD, tRTP, tWTR, tWR, tRFC	4, 7, 10, 74, 24
tRTRS, tCMD, tXP, tRP, tRC	1, 1, 4, 10, 40
RL, WL, BL	13, 3, 1

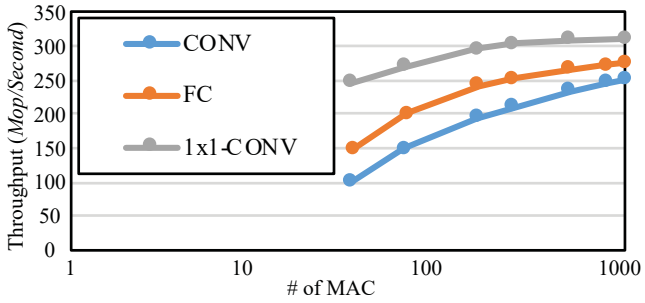

Figure 6: Throughput vs. MAC ops per feature map.

operation. From Figure 7, we find that RS has the smallest number of overall operations across most convolution layers among three data reuse strategies. Because both WS and OS involves parameter multi-cast and uni-cast, while in RS only accumulation result is transferred to GB.

Figure 8 shows the NoC latencies in different NoC topology with WS and RS. Bus achieves the lowest latency in WS among all three topology. It is because bus is very efficient in broadcasting and WS repeats broadcasting the activations to PEs in convolutions. In RS, however, crossbar exhibits the lowest latency. In RS, the NoC latency is generally longer than that in WS in spite of a relatively high data reuse level. The Mesh performance is non-optimal in all cases because it needs to serialize all the scatter traffic.

5.3 Evaluation of Multi-HMC in NeuralHMC

Figure 9 shows the speedup and energy consumption of an 8-HMCs architecture. Our baseline is Eyeriss [4] – an spatial DNN accelerator with 128 processing elements. We use RS data reuse strategy

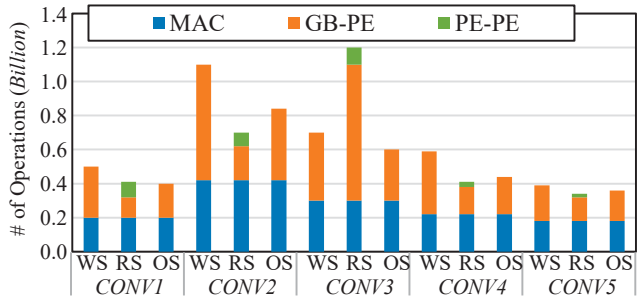

Figure 7: Operation breakdown in AlexNet.

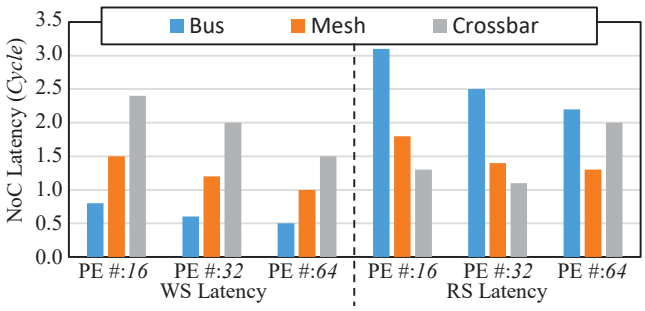

Figure 8: NoC latencies with RS and WS.

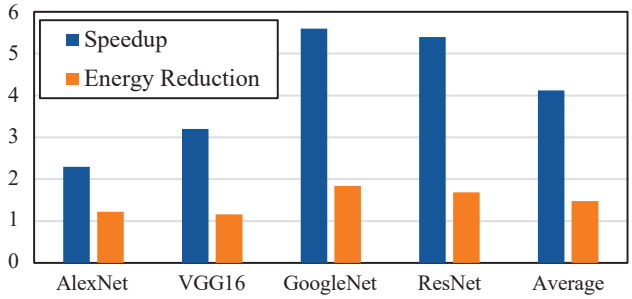

Figure 9: Total speedup and energy reduction.

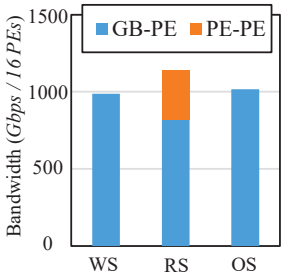

Figure 10: Inter-vault bandwidth breakdown.

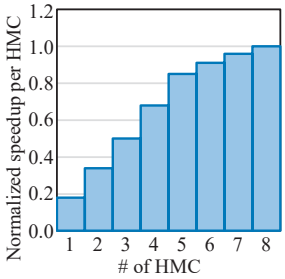

Figure 11: Performance of scalability.

and crossbar NoC topology in both baseline and proposed accelerator. In above four benchmarks (i.e., AlexNet, VGG16, GoogleNet, ResNet-50), the speedup of GoogleNet can be up to 5.6× and energy consumption reduction is up to 1.85×.

In all benchmarks shown in Figure 9, the energy consumption reduction is not as high as speedup. The reason lies in the constraint of the MAC logic in TSV and the control logic in vault controller. Even for the AlexNet, our proposed architecture still outperforms the Eyeriss implementation in both speed and energy consumption. Thanks to our *NeuralHMC* accelerator architecture and weight sharing MAC, an average speedup over all benchmarks is 4.1× and an energy reduction of 1.5×.

We evaluate the inter-vault bandwidth of the three data reuse dataflow in ResNet-50 as shown in Figure 10. The PE-PE bandwidth is 300Gops by assuming there are 16PEs (1 HMC) in RS. The GB-PE bandwidth is 1000Gops in both WS and OS. The performance scalability of multi-HMC is illustrated in Figure 11. As we can observe in Figure 11, the speedup scales in linear when the number of the HMCs is smaller than 5. With proposed packet scheduling, the decoding process is pipelined and the performance degradation is offset with an active HMC. However, keep increasing the number of HMCs will not maintain linear performance improvement. This is mainly because the parameters are not evenly distributed in multiple HMCs.

6 CONCLUSION

In this work, we propose a neural network accelerator based on processing-in-memory multi-HMC called *NeuralHMC*. *NeuralHMC* is optimized to perform MAC operation in DNN application. The optimizations include: (1) using a weight sharing MAC to reduce weight data access, (2) packet scheduling in multi-HMC architecture to pipeline packet decoding, (3) avoiding packet regeneration in vault controller by calculating the maximum MAC count in on-chip controller. Experimental results show that our proposed *NeuralHMC* can improve the system performance by 4.1× in speedup and 1.5× in energy reduction compared to the DNN accelerator design built with conventional low-power DRAM memory. Furthermore, our *NeuralHMC* outperforms the baseline for all the DNN architectures and neural network layer types, showing excellent generality for different DNN implementations.

ACKNOWLEDGMENTS

This research was funded in part by National Science Foundation 1615475, 1725456 and Department of Energy DE-SC0018064.

REFERENCES

[1] Junwhan Ahn, Sungjoo Yoo, and Kiyoung Choi. 2014. Dynamic power management of off-chip links for hybrid memory cubes. In *Proceedings of the 2014 ACM/EDAC/IEEE Design Automation Conference*. 1–6.
[2] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R Hower, Tushar Krishna, Somayeh Sardashti, et al. 2011. The gem5 simulator. *ACM SIGARCH Computer Architecture News* 39, 2 (2011).
[3] Tianshi Chen, Zidong Du, Ninghui Sun, Jia Wang, Chengyong Wu, Yunji Chen, and Olivier Temam. 2014. Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning. *ACM Sigplan Notices* 49, 4 (2014).
[4] Yu-Hsin Chen, Tushar Krishna, Joel S Emer, and Vivienne Sze. 2017. Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks. *IEEE Journal of Solid-State Circuits* 52, 1 (2017).
[5] Ping Chi, Shuangchen Li, Cong Xu, Tao Zhang, Jishen Zhao, Yongpan Liu, Yu Wang, and Yuan Xie. 2016. Prime: A novel processing-in-memory architecture for neural network computation in reram-based main memory. In *Proceedings of the 43rd International Symposium on Computer Architecture*. 27–39.
[6] Hybrid Memory Cube Consortium et al. 2015. HMC specification 2.0.
[7] Ramyad Hadidi, Bahar Asgari, Jeffrey Young, Burhan Ahmad Mudassar, Kartikay Garg, Tushar Krishna, and Hyesoon Kim. 2018. Performance Implications of NoCs on 3D-Stacked Memories: Insights from the Hybrid Memory Cube. In *Proceedings of the 2018 IEEE International Symposium on Performance Analysis of Systems and Software*. 99–108.
[8] Song Han, Huizi Mao, and William J Dally. 2015. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149* (2015).
[9] Phillip Isola, Jun-Yan Zhu, Tinghui Zhou, and Alexei A Efros. 2017. Image-to-Image Translation with Conditional Adversarial Networks. In *Proceedings of the 2017 IEEE Conference on Computer Vision and Pattern Recognition*. 5967–5976.
[10] Dong-Ik Jeon, Kyeong-Bin Park, and Ki-Seok Chung. 2018. HMC-MAC: Processing-in Memory Architecture for Multiply-Accumulate Operations with Hybrid Memory Cube. *IEEE Computer Architecture Letters* 17, 1 (2018).
[11] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. 2017. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*. 1–12.
[12] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*. 1097–1105.
[13] John D Leidel and Yong Chen. 2016. Hmc-sim-2.0: A simulation platform for exploring custom memory cube operations. In *Proceedings of the 2016 IEEE International Parallel and Distributed Processing Symposium Workshops*. 621–630.
[14] Jiachen Mao, Xiang Chen, Kent W Nixon, Christopher Krieger, and Yiran Chen. 2017. Modnn: Local distributed mobile computing system for deep neural network. In *Proceedings of the 2017 Design, Automation & Test in Europe Conference & Exhibition*. 1396–1401.
[15] Yao Qian, Yuchen Fan, Wenping Hu, and Frank K Soong. 2014. On the training aspects of deep neural network (DNN) for parametric TTS synthesis. In *Proceedings of the 2014 IEEE International Conference on Acoustics, Speech and Signal Processing*. 3829–3833.
[16] Patrick Siegl, Rainer Buchty, and Mladen Berekovic. 2017. A bandwidth accurate, flexible and rapid simulating multi-HMC modeling tool. In *Proceedings of the 2017 International Symposium on Memory Systems*. 71–82.
[17] Chen Zhang, Peng Li, Guangyu Sun, Yijin Guan, Bingjun Xiao, and Jason Cong. 2015. Optimizing fpga-based accelerator design for deep convolutional neural networks. In *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 161–170.