Scaling Deep Learning on Multiple In-Memory Processors

Lifan Xu, Dong Ping Zhang, and Nuwan Jayasena

AMD Research, Advanced Micro Devices, Inc.

{lifan.xu, dongping.zhang, nuwan.jayasena}@amd.com

ABSTRACT

Deep learning methods are proven to be state-of-the-art in addressing many challenges in machine learning domains. However, it comes at the cost of high computational requirements and energy consumption. The emergence of Processing In Memory (PIM) with die-stacking technology presents an opportunity to speed up deep learning computation and reduce energy consumption by providing low-cost high-bandwidth memory accesses. PIM uses 3D die stacking to move computations closer to memory and therefore reduce data movement overheads. In this paper, we study the parallelization of deep learning methods on a system with multiple PIM devices. We select three typical layers: the convolutional, pooling, and fully connected layers from common deep learning models and parallelize them using different schemes. Preliminary results show we are able to reach competitive or even better performance using multiple PIM devices when comparing with traditional GPU parallelization.

1. INTRODUCTION

Deep learning has shown promising success in domains such as speech recognition, image classification, and natural language processing [1]. However, state-of-the-art deep learning models often contain a large number of neurons, resulting in millions or even billions of free parameters [2]. To train such complex models, tremendous computational resources, energy, and time are required. Recently, a significant amount of effort has been put into speeding up deep learning by taking advantage of high-performance systems. Despite that, AlexNet [3] takes more than five days to train on two GPUs; DistBelief [4] uses 16,000 cores to train a neural network in a few days; COTS HPC system [2] scales to neuron networks with over 11 billion parameters using a cluster of 16 GPU servers. While these prior works have successfully accelerated the computation by mapping the application to existing architectures, relatively little research has been done to evaluate the potential of emerging architecture designs, such as in-memory computing, to improve the performance and energy efficiency of deep learning.

This paper explores the potential of Processing In Memory (PIM)[1] implemented via 3D die stacking to improve the performance of deep learning. While PIM research has been active from time to time for a few decades, it has not been commercially viable due to manufacturing and economic challenges. However, recent advances in 3D die stacking technology make it possible to stack a logic die with one or more memory dies enabling a new class of PIM solutions. These solutions build on the same underlying 3D stacking technology used by recent memory technologies such as Hybrid Memory Cube (HMC) [5] and High Bandwidth Memory (HBM) [6]. It has been demonstrated that a broad range of applications can achieve competitive performance and much greater energy efficiency on viable 3D-stacked PIM configurations compared with a representative mainstream GPU [7].

In this study, we evaluate the performance of scaling deep learning models on a system with multiple PIM devices. In this system, the host is a high-performance, mainstream APU. This host is attached to several memory modules, each with PIM capabilities consisting of a small APU.

From the two most popular deep learning models, Convolutional Neural Network (CNN) and Deep Belief Network (DBN), we select three frequently used and representative layers: the convolutional layer, pooling layer, and fully connected layer. Across the multiple PIM devices, we parallelize these layers individually. Two parallelization schemes are evaluated, which are data parallelism and model parallelism. The data parallelism approach keeps a copy of the full neural network model on each device but partitions the input data into mini batches across them. We evaluate data parallelism on all three layers. The model parallelism approach partitions the neural network model and distributes one model partition to each device. We apply model parallelism to the fully connected layer, as the number of parameters of the neural network in this layer increases drastically as the network grows. Memory capacity can often be a limiting factor for fully connected layers. When the model is too large to fit into a PIM's memory, it is essential to partition the model across multiple PIMs using model parallelism.

Preliminary experiments show that by scaling deep learning models to multiple PIMs available in a system, we are able to achieve better or competitive performance compared with a high-performance host GPU in many cases across the different layers studied. We show that model parallelism consumes much less memory than data parallelism on fully connected layers, and it also reaches better performance when the number of input images per batch is small. However, as the batch size increases, data parallelism scales better due to the absence of synchronization and outperforms model parallelism.

2. DEEP LEARNING MODELS

[1]This paper uses PIM as an abbreviation interchangeably for process*ing* in memory and process*or* in memory depending on the context.

Deep Belief Networks (DBN) are constructed by chaining a set of Restricted Boltzmann Machines (RBM) [8]. We explain the details of RBM in Section 2.3. Here we show an example of a DBN trained for speech recognition in Fig. 1. The model takes as an input a spectral representation of a sound wave. The input is then processed by several RBMs where each RBM may contain a different number of hidden units. Finally, the DBN translates the input sound wave to text output.

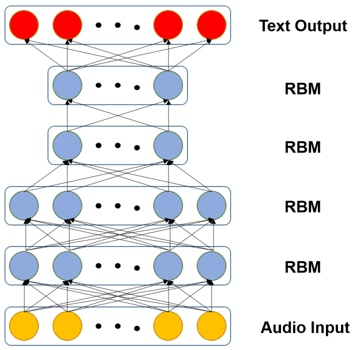

Figure 1: DBN on speech recognition

Unlike DBN, CNN may consist of multiple different layers. The most basic ones are convolutional, pooling, and fully connected layers. The fully connected layer has effectively the same characteristics as the RBM. The details of each layer are discussed in the following subsections. Here we show an example of a CNN model trained for digit recognition in Fig. 2. The input to this model is an image containing one hand-written digit. The input is first processed by the convolutional layer where each filter outputs one feature map. The feature maps are downsampled by the max pooling layer. Outputs from the pooling layer are then processed by the fully connected layer. The final output layer contains 10 neurons where each neuron represents one digit. The neuron with the highest probability is the prediction result of the input.

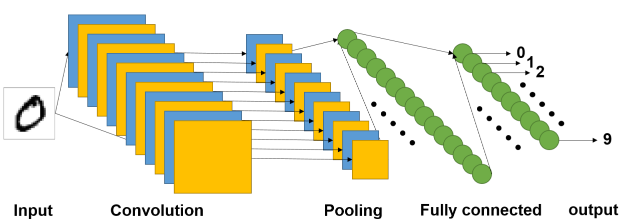

Figure 2: CNN on digit recognition

Traditionally, deep learning applications consist of two phases: training and prediction. The training phase contains forward propagation and backward propagation for weight updates [9, 10]. In forward propagation, input images are processed through all layers in the deep learning model with initial weights. In backward propagation, error is computed based on the model output. The error is then propagated back through all layers and used to update the weights for each layer. The prediction phase contains only the forward propagation using the weights learned in the training phase. This paper focuses on the three common layers in the forward propagation: convolutional, pooling, and fully connected, as they are key to both the training and prediction phases.

2.1 Convolutional Layer

The Convolutional (conv) layer is the core building block of CNN. The input of this layer is a batch of images and each image has 3 dimensions including width, height, and depth (or channels). The conv layer applies one or several convolutional filters (or kernels) to each 3D input volume. The filters are spatially small along the width and height dimensions, but they have the same depth as the input volume. Although it is not required, practitioners usually set the filter to have the same size along width and height dimensions in practice and call this hyperparameter **filter size**. During the forward propagation, each 3D filter is applied by sliding it across the width and height dimensions of each input volume, producing a 2D feature map of that filter. Each time we slide the filter across the input, we compute the dot product between the entries of the filter and the 3D sliding window. The hyperparameter **stride** defines how far we slide the filter. Assuming stride as 1, we slide the filter by only 1 spatial unit for the next convolution. Also, a technique called zero-padding can be applied to add zeros surrounding the input volume, so the filter can be applied to the border elements of the input.

Fig. 3 shows an example of 2D convolution. The input is 4×4 with zero padding. The filter size is 3, and the output is also 4×4 because we set stride size to be 1. The red window slides along the width and height of the input. Dot products between entries in the input red window and filter are performed and output to the resulting feature map.

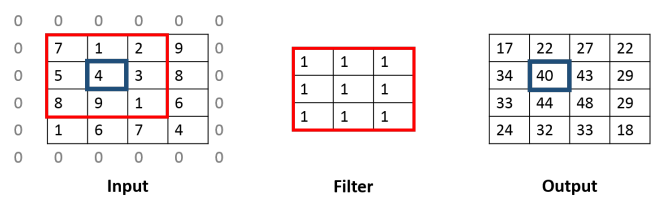

Figure 3: 2D Convolution example.

2.2 Pooling Layer

Input to pooling layer is usually the output from conv layer after an element-wise non-linear transformation. The pooling layer is used to reduce the spatial size of the input through downsampling. By doing so, the amount of parameters and computation can be greatly reduced and can also help alleviate overfitting. The most common pooling operation in CNN is max pooling. It slides a 2D window along the width and height of the input on every channel. Each window outputs a max value of the elements within the window. Therefore, the output of the pooling layer is spatially downsampled on width and height but remains the same depth as the input. Similar to the conv layer, the output size depends on the choices of kernel size and stride. Fig. 4 shows an example of performing max pooling on a 4×4 input with a filter size of 2 and stride size of 2. The maximum value of each window in the input is the output in the resulting feature map.

2.3 Fully Connected Layer

The fully connected layer in CNN can be treated as the RBM used in DBN. An RBM is an energy-based generative model that consists of two layers: a layer of

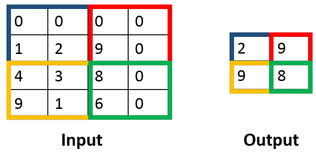

Figure 4: Max pooling example.

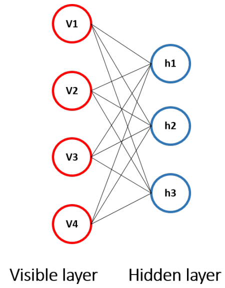

Figure 5: An example of RBM.

visible units v, and a layer of hidden units h. The units in different layers are fully connected with no connections among units in the same layer. Figure 5 shows a very small RBM with 4 units in the visible layer and 3 units in hidden layer for illustration purpose. In total, there are 4×3 edges in the network. Weights associated with these edges are represented as a 4×3 weight matrix. In CNN, input of fully connected layer is usually the output of pooling layer. Each 3D volume from the pooling layer can be unrolled to a large 1D vector. The dimension of the 1D vector equals the visible layer size of the RBM. By unrolling all 3D volumes from the pooling layer, we are then able to represent them as a 2D matrix, which we then multiply with the weight matrix to derive the output of the fully connected layer.

3. PIM ARCHITECTURE

Fig. 6 shows our system organization consisting of one host and four PIM stacks. Each PIM stack has a logic die containing the in-memory processor and memory (DRAM) dies on top of it.

Both the host and in-memory processors are Accelerated Processing Units (APU). Each APU consists of CPU and GPU cores on the same silicon die which enables the execution of both CPU- and GPU-oriented general-purpose code on either the host or PIM. Selecting an APU as the in-memory processor lowers the barrier to adoption and allows the use of existing rich sets of development tools for CPUs and GPUs. For this evaluation, we focus on the GPU execution units of the host and the PIM APUs.

The in-memory processor in each memory stack has high-bandwidth access to the memory stacked on it at a peak of 320 GB/s. The capacity of each memory stack is 4 GB. The host also has direct access to the memory stacked atop the PIM devices but at a reduced bandwidth, as those accesses must be over a board-level memory interface. We model the host memory interface on Hybrid Memory Cube (HMC) [5] at 160 GB/s peak bandwidth per memory stack (i.e., $1/2$ the internal bandwidth), which results in aggregate host bandwidth of 640 GB/s across the four memory stacks. In order to model more mainstream hosts with lower memory bandwidth, we also evaluate designs where the host has $1/4$ and $1/8$ the internal bandwidth per memory stack.

We model a unified address space among the host and PIM devices that allows direct access from any PIM device to any memory within the system. Access to remote memory (i.e., memory in other PIM stacks) by PIM devices is modeled at $1/8$ the intra-stack bandwidth per stack.

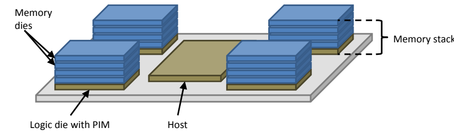

Figure 6: A node with four PIM stacks.

4. PIM PERFORMANCE MODEL

A key challenge for memory systems research is the need for evaluating realistic applications with large data sets that can be prohibitive to run on cycle-level simulators. This issue is exacerbated as PIM expands the design space that must be explored. Therefore we perform our evaluations using a model that analyzes performance on existing hardware and uses machine learning techniques to predict the performance on future system organizations [11].

The model is constructed by executing a sufficiently large number of diverse kernels (the training set) on native GPU hardware and characterizing their execution through performance counters. Each hardware parameter that we are interested in scaling for future systems is varied to identify the performance sensitivity of each kernel to that hardware parameter. We then use a clustering algorithm to identify groups of kernels with similar scaling characteristics. Once the model is constructed, we are able to run a new kernel at a single hardware configuration, and use its performance counters as a signature to map it to one of the clusters formed during model construction. The cluster identifies the scaling characteristics of the kernel, which is used to predict the performance for future machine configurations of interest, including PIM. The accuracy of this approach has been shown to be comparable to cycle-level models for exploring the design space of key architectural parameters such as compute throughput and memory bandwidth [11].

5. DEEP LEARNING ON MULTIPLE PIMS

The key challenge in implementing deep learning algorithms on a system with multiple PIMs is partitioning the data among the memory stacks and dispatching the scoped compute kernels to the PIMs corresponding to the data partitions to exploit the high memory bandwidth available from each PIM to its local memory stack. Due to the high parallelism and throughput requirements of deep learning algorithms, we focus on the GPU execution units of the host and PIM APUs.

5.1 Data Parallelism and Model Parallelism

We explore two approaches to parallelize deep learning models on multiple PIM GPUs: data parallelism and model parallelism. In data parallelism, the input batch of images is partitioned across PIMs. Each PIM GPU gets a subset of the data and works on the full model. In model parallelism, the neural network model is partitioned across PIM GPUs. Each GPU works on one partition of the model using the full input batch.

The advantage of data parallelism is that each PIM gets a copy of the model, allowing each one to operate completely independently on its data without any inter-PIM communication. This is often desirable in cases where the model fits within the memory capacity of a single stack and the capacity overhead of replicating the model on all PIM stacks is acceptable. Further, by increasing batch size, data parallelism can be made arbitrarily scalable and efficient. For example, if there are 8 PIMs and the batch size is 256, then each PIM gets 32 input images which may result in low GPU usage. If we increase the batch size to 1024, then each PIM can get 128 input images and higher GPU usage. However, increasing the batch size can increase response time for latency-critical prediction tasks and adversely affect convergence rates in model training. Therefore, the batch sizes are typically set to be hundreds. For example, AlexNet uses a batch size of 128 and VGG nets [12] use 256. In this paper, we evaluate batch sizes up to 1024.

The advantage of model parallelism is to enable training and prediction with much larger deep learning models. For example, COTS HPC system trains a network with more than 11 billion parameters which requires about 82 GB memory. Such a model is too large to fit into one single node using data parallelism, and thus needs to be partitioned using model parallelism. However, inter-PIM communication is inherent in model parallelism. As the model is partitioned across PIMs, each PIM can only compute a subset of neuron activities. They need synchronization to get the full neuron activities. In Fig. 7, we show how to partition the RBM example from Fig. 5 across two PIMs. PIM1 gets visible unit $v1$ and $v2$ while PIM2 gets visible unit $v3$ and $v4$. When computing the neuron activities of $h1$, $h2$, and $h3$, PIM1 only computes the contributions from $v1$ and $v2$ and PIM2 only computes the contributions from $v3$ and $v4$. However, the full activities come from all units in the visible layer; therefore the contributions from PIM1 and PIM2 are summed together to generate the correct result.

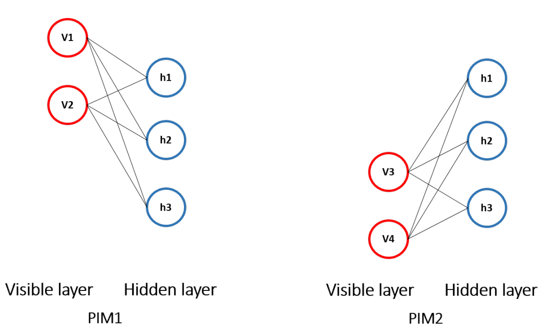

Figure 7: Model Partitioning of the RBM example shown in Fig. 5 across two PIMs.

5.2 Convolutional Layer Parallelization

In deep learning models, conv layers cumulatively contain most of the computation, e.g., 90% to 95%, but only a small fraction of the parameters, e.g., 5% [13]. As we focus only on the prediction phase of deep learning in this study (i.e., there is no backward propagation and no weights update) the two parallelization schemes result in the same amount of computation for conv layers. Suppose there are I input images, F filters, and the image size is S by S. With a stride of 1, the total number of convolutions is $I \times F \times S \times S$. For data parallelism across N PIMs, each PIM is responsible for $(I/N) \times F \times S \times S$ convolutions because input data is partitioned. For model parallelism, each PIM is assigned $I \times (F/N) \times S \times S$ convolutions because the set of filters is partitioned. Therefore, the amount of computation is the same for each PIM GPU no matter which parallelization scheme is applied. Further, due to the small size of the model parameter set, memory capacity pressure is not a factor in conv layers. Therefore, for simplicity, this paper only evaluate data parallelism on conv layer. Given N PIM devices, the input batch of images is evenly partitioned to N mini batches. Each mini batch is assigned to one PIM and then propagates forward independently.

5.3 Pooling Layer Parallelization

For the pooling layers, there are no model parameters. Therefore, we can only apply data parallelism. However, depending on the parallelization scheme applied on the previous conv layer, the conv layer may have different groupings of the same output resulting in different input groupings to the pooling layer. This does not affect the correctness of the application. For example, consider an input to the previous conv layer with eight images and four filters. In model parallelism across two PIM GPUs, each PIM outputs eight images where each image has two feature maps. In data parallelism, each PIM outputs four images where each image has four feature maps. Nevertheless, the total amount of computation for each PIM stays the same for the subsequent pooling layer.

5.4 Fully Connected Layer Parallelization

In contrast to the conv layers, fully connected layers contain a small part of the computation, e.g., 5% to 10%, but the majority of the model parameters, e.g., 95% [13]. Fully connected layers can choose to deploy whichever parallelization scheme was used in previous layers. However, if the model is too large to fit into each PIM's memory, then model parallelism is required to train and predict at that large scale. Therefore, we evaluate both data and model parallelism on fully connected layer. Please note that, by applying model parallelism on fully connected layer, synchronization is needed at the end. As shown in Fig. 7, hidden layer activities from PIM1 and PIM2 need to be summed together to get the correct hidden layer activities. In our implementation, this reduction across PIMs happens on host. The host accesses each PIM's memory, performs the reduction, and writes the results back to, for example, PIM0. The other PIMs can then fetch the results from PIM0.

6. RESULTS

To show the potential of scaling deep learning algorithms on multiple PIMs, we evaluate three representative layers: the convolutional, pooling, and fully connected layers. We first run these layers with varying application parameters on an AMD Radeon HD 7970 GPU with 32 compute units and 3 GB device memory. During each run, we profile and collect the performance counters of all kernels. We then scale the performance on the native hardware to multiple desired host and PIM configurations applying the methodology described in Section 4.

6.1 PIM configurations

Table 1: Host and PIM configurations

	Host_4_160	Host_4_320	Host_4_640	Host_8_320	Host_8_640	Host_8_1280	4PIM	8PIM
Number of CUs	32	32	32	64	64	64	64	128
Engine Frequency (MHz)	1300	1300	1300	1300	1300	1300	650	650
Total DRAM BW (GB/s)	160	320	640	320	640	1280	1280	2560
DRAM BW/stack (GB/s)	40	80	160	40	80	160	320	320
Number of DRAM stacks	4	4	4	8	8	8	4	8

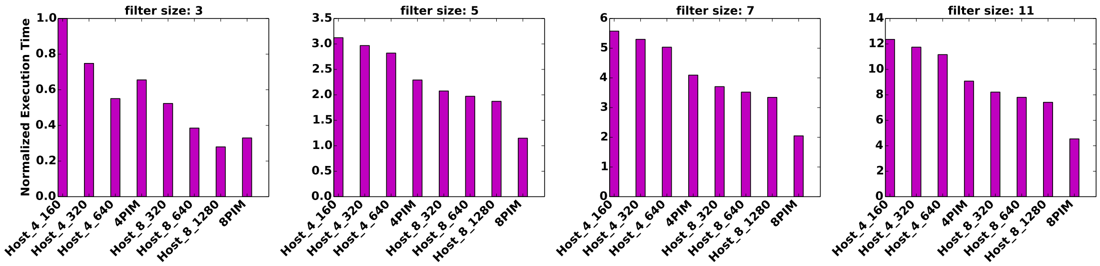

Figure 8: Convolutional layer results (normalized to *Host_4_160* with filter size 3).

In our experiments, we set the memory bandwidth to 320 GB/s and peak computation throughput to 650 GFLOPS for each PIM device. Two node organizations are explored. The first node organization has a host and four PIM stacks shown in Fig. 6. The second one has a host and eight PIM stacks. The objective is to compare the performance of deep learning parallelization on multiple PIM devices against the performance of the host GPU. For fair comparison, we set the peak FLOPS of the host GPU equal to the aggregate peak FLOPS of all PIM devices. The host accesses the memory stacks at lower bandwidth than in-stack memory access from the PIMs. Three host bandwidths are evaluated: 1/2, 1/4, and 1/8 of the in-stack PIM bandwidth per memory stack. However, the host can simultaneously access all the memory stacks.

Table 1 shows the configurations we used for host and PIM. In total, there are six host configurations; three of them have four PIMs and the other three have eight PIMs. The host configurations are named in the pattern of *Host_N_B* where N is the number of PIM stacks and B is the total memory bandwidth available to the host. For example, *Host_4_640* means this host has 4 PIM stacks and 640 GB/s memory bandwidth in total. Two PIM configurations are listed as *4PIM* and *8PIM* in the table.

6.2 Results on Convolutional Layer

We first explore data parallelism on conv layer. The size of a single input image is 256 by 256 in width and height. The number of channels per image is 16. The number of images per input batch is 256. The number of filters to 16. The depth of each filter is set to be 16 but different filter sizes including 3, 5, 7, and 11 are evaluated. We select these filter sizes because they were used in stat-of-the-art deep learning models. For example, AlexNet uses filter sizes 11, 5, and 3. VGG nets set the filter size to be 3. The stride size is set to be 1 for all cases for simplicity. For 4-PIM and 8-PIM configurations, the input batch is partitioned into mini batches, taking the data parallelism approach. Each PIM is assigned one mini batch and compute the convolution independently. For all host configurations, since there is only one GPU, no data partition is needed.

Fig. 8 shows the normalized execution time for the conv layer. Each of the four subfigures shows results obtained using a particular filter size. In these subfigures, the Y-axis shows execution time normalized to *Host_4_160* with a filter size of 3. The X-axis lists the different configurations: six host design points, 4-PIM design, and 8-PIM design. When filter size is 3, the execution times of *4PIM* and *8PIM* are slightly worse than the *Host_4_640* and *Host_8_1280* respectively. However, they do outperform the other host configurations. When the filter size is increased to 5, 7 or 11, running conv layer on multiple PIMs is faster than on all the host configurations. The observation fits our expectation because larger filter size means more memory access per convolution. The high memory bandwidth provided by PIM stacks makes it beneficial to run larger convolutions on PIM devices.

6.3 Results on Pooling Layer

The pooling layer is also evaluated with data parallelism as it has very localized compute pattern and there is no neural network model. We again use the single image size of 256 width by 256 height in pixels with 16 channels. Batch size is set to 256. There are various pooling operations used in deep learning applications. However, their computational and memory access characteristics are very similar. Hence we pick the commonly used max pooling for our evaluation. The max pooling operation is performed using a 2D window at each channel of the input image. Large filers are typically not used in pooling because too much information can be lost. For example, AlexNet uses a filter size of 3 for pooling, VGG nets use 2 as the filter size, and a filter size of 5 was used in COTS HPC system. As a result, we evaluate filter sizes 2, 3, 4, and 5. The latter two are added to evaluate how the performance changes as the filter size increases. The stride size is set to the filter size for simplicity.

Fig. 9 shows the normalized execution time of the pooling layer using different filter sizes on the proposed configurations. We pick the execution time from *Host_4_160* using a filter size of 2 as the baseline for normalization. When filter size is small (e.g., 2), the performance of multiple PIM stacks is competitive with the host. As the filter size increases, more significant performance improvement is observed on the two PIM configurations. This observation is similar to conv layer results as it also benefits from the high memory bandwidth of the PIMs. As filter size increases, more memory accesses are needed

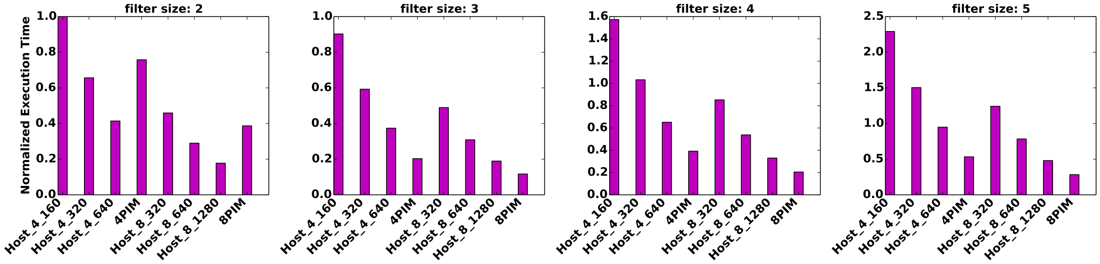

Figure 9: Pooling layer results (normalized to *Host_4_160* with filter size 2).

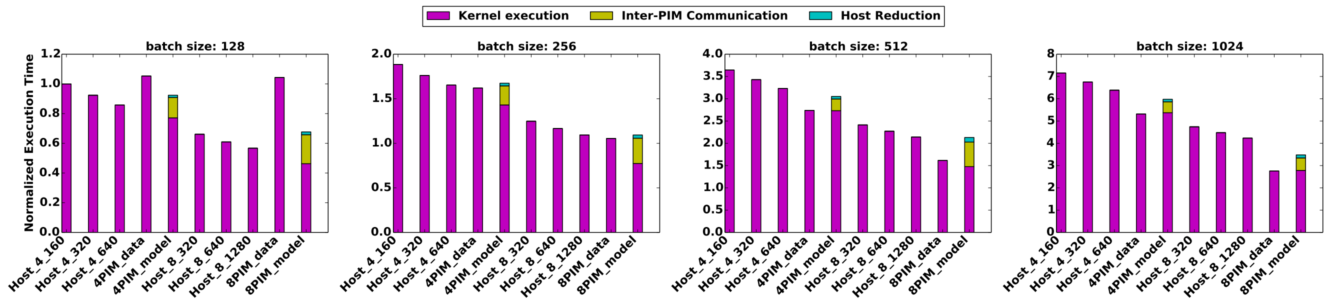

Figure 10: Fully Connected layer results (normalized to *Host_4_160* with batch size 128).

for each pooling operation.

6.4 Results on Fully Connected Layer

We evaluate both data and model parallelism on fully connected layer. We first compute the memory consumption per PIM for these two parallelization schemes. The results are shown in Fig. 11. In this figure, the solid lines are for data parallelism and the dashed lines represent model parallelism. The red lines are for the 4-PIM configuration and the green lines are for the 8-PIM configuration. Different markers correspond to different input batch sizes. This figure shows that data parallelism consumes substantially more memory per PIM for one fully connected layer. Our evaluation shows that varying the number of PIMs or batch sizes does not change the memory consumption significantly for data parallelism, as the large number of model parameters in the fully connected layer consumes most of the memory and is replicated on each memory module. However, in model parallelism, the model parameters are partitioned among the memory modules, resulting in moderate growth in memory capacity demand with both model size and input batch size. Further, with model parallelism, adding more PIMs reduces the memory pressure per PIM. In theory, if the batch size is large enough, model parallelism can consume similar amount of memory as data parallelism. However, as batch size is typically small in reality (e.g., 128 or 256), model parallelism is preferred in fully connected layer due to lower memory consumption.

We then evaluate the execution time for both data and model parallelization schemes using 4-PIM and 8-PIM configurations and compare them with host executions. Fig. 10 records the obtained results, normalized to *Host_4_160* with a batch size of 128. The four subfigures correspond to four different batch sizes: 128, 256, 512, and 1024. The layer size is fixed to be 4096 which was used in AlexNet and VGG nets. For each subfigure, the Y-axis records normalized execution time and the X-axis lists different configurations. Please note that, *4PIM_data* means data parallelism on 4-PIM configuration. Similarly, *8PIM_model* stands for model parallelism on 8-PIM configuration. Because synchronization is needed in model parallelism, we use different colors to represent different components of execution time in *4PIM_model* and *8PIM_model*. Purple represents OpenCL kernel execution time on PIM GPU, which excludes the reduction procedure that runs on the host GPU. The green segment represents the reduction across PIMs performed on host GPU, including memory access to all PIM stacks and writing the final results back to the first PIM in the configuration (PIM0). Yellow represents the time that all the other PIMs copy the reduction result from PIM0 to their local memory stacks. Please note that for PIM to PIM memory copy, we assume the memory bandwidth is 1/8 of the local in-stack memory access bandwidth. Therefore, for remote PIM memory access, the bandwidth is configured at 40 GB/s. This figure shows that model parallelism performs better than data parallelism and is comparable to host execution when batch size is small, e.g., 128. However, when we increase the batch size, model parallelism loses its advantage to data parallelism due to synchronization cost. However, with large batch sizes, both data and model parallelism on multiple PIM stacks outperform host execution.

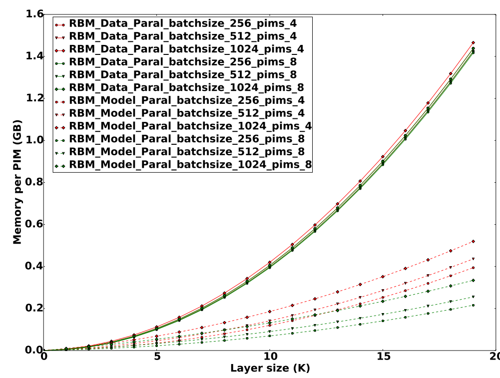

Figure 11: Memory consumption per PIM for data parallelism and model parallelism on fully connected layer.

7. CONCLUSION AND FUTURE WORK

In this paper, we evaluate the performance of deep learning models on PIM devices. We study three types of layers from CNN and DBN, which are two of the most popular forms of deep learning models. The fully connected layer is parallelized across multiple PIM devices using data parallelism, which partitions the input set, and model parallelism, which partitions the model parameter set. Our results show that memory capacity requirements of data parallelism increase much more rapidly than for model parallelism as the model size increases. Further, we show that model parallelism preforms better at small input batch sizes while data parallelism performs better as input batch size increases. We parallelize convolutional and pooling layers across multiple PIM devices using data parallelism. We also vary key parameters for each of the layers over commonly used ranges of values.

Our results show that PIM achieves competitive or better performance compared to a high-performance host GPU across a variety of system and model parameter ranges. This is an extremely promising result as this allows deep learning models to be ported to PIM with no loss of performance and yet realize the significant energy efficiency improvements that have been demonstrated for PIM in past studies. In the future, we plan to perform detailed evaluations of energy efficiency of deep learning on PIM.

8. ACKNOWLEDGMENTS

We would like to thank Joe Greathouse for discussion on simulation methodology. We also thank Junli Gu for discussion on deep learning in general and her feedback on the manuscript.

9. REFERENCES

[1] I. Arel, D. Rose, and T. Karnowski, "Deep machine learning - a new frontier in artificial intelligence research [research frontier]," *Computational Intelligence Magazine, IEEE*, vol. 5, pp. 13–18, Nov 2010.

[2] A. Coates, B. Huval, T. Wang, D. J. Wu, and A. Y. Ng, "Deep learning with cots hpc systems."

[3] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in Neural Information Processing Systems 25* (F. Pereira, C. Burges, L. Bottou, and K. Weinberger, eds.), pp. 1097–1105, Curran Associates, Inc., 2012.

[4] J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, Q. V. Le, M. Z. Mao, M. Ranzato, A. W. Senior, P. A. Tucker, K. Yang, and A. Y. Ng, "Large scale distributed deep networks," in *Advances in Neural Information Processing Systems 25: 26th Annual Conference on Neural Information Processing Systems 2012. Proceedings of a meeting held December 3-6, 2012, Lake Tahoe, Nevada, United States.*, pp. 1232–1240, 2012.

[5] J. T. Pawlowski, "Hybrid memory cube: breakthrough dram performance with a fundamentally re-architected dram subsystem," in *Proceedings of the 23rd Hot Chips Symposium*, 2011.

[6] "High bandwidth memory dram." https://www.jedec.org/standards-documents/docs/jesd235. Accessed: 2010-09-30.

[7] 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*, HPDC '14, (New York, NY, USA), pp. 85–98, ACM, 2014.

[8] G. E. Hinton, S. Osindero, and Y.-W. Teh, "A fast learning algorithm for deep belief nets," *Neural Comput.*, vol. 18, pp. 1527–1554, July 2006.

[9] Y. Bengio, "Learning deep architectures for ai," *Foundations and trends® in Machine Learning*, vol. 2, no. 1, pp. 1–127, 2009.

[10] J. Schmidhuber, "Deep learning in neural networks: An overview," *CoRR*, vol. abs/1404.7828, 2014.

[11] G. Wu, J. Greathouse, A. Lyashevsky, N. Jayasena, and D. Chiou, "GPGPU performance and power estimation using machine learning," in *21st IEEE Symp. on High Performance Computer Architecture*, 2015.

[12] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *CoRR*, vol. abs/1409.1556, 2014.

[13] A. Krizhevsky, "One weird trick for parallelizing convolutional neural networks," *CoRR*, vol. abs/1404.5997, 2014.