HMC-Based Accelerator Design For Compressed Deep Neural Networks

by

Chuhan Min

B.S. in Physics, Nanjing University, China, 2012

M.S. in Electrical Engineering, University of Pittsburgh, 2014

Submitted to the Graduate Faculty of

the Swanson School of Engineering in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

University of Pittsburgh

2019

UNIVERSITY OF PITTSBURGH

SWANSON SCHOOL OF ENGINEERING

This dissertation was presented

by

Chuhan Min

It was defended on

November 20, 2019

and approved by

HMC-Based Accelerator Design For Compressed Deep Neural Networks

Chuhan Min, PhD

University of Pittsburgh, 2019

Deep Neural Networks (DNNs) offer remarkable performance of classifications and regressions in many high dimensional problems and have been widely utilized in real-word cognitive applications. In DNN applications, high computational cost of DNNs greatly hinder their deployment in resource-constrained applications, real-time systems and edge computing platforms. Moreover, energy consumption and performance cost of moving data between memory hierarchy and computational units are higher than that of the computation itself. To overcome the memory bottleneck, data locality and temporal data reuse are improved in accelerator design. 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. Inherited from the concept of Process-In-Memory (PIM), some 3D-stacked memory architectures 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.

In this dissertation, we are going to investigate hardware/software co-design for neural network accelerator. Specifically, we introduce a two-phase filter pruning framework for model compression and an accelerator tailored for efficient DNN execution on HMC, which can dynamically offload the primitives and functions to PIM logic layer through a latency-aware scheduling controller.

In our compression framework, we formulate filter pruning process as an optimization problem and propose a filter selection criterion measured by conditional entropy. The key idea of our proposed approach is to establish a quantitative connection between filters and model accuracy. We define the connection as conditional entropy over filters in a convolutional layer, i.e., distribution of entropy conditioned on network loss. Based on the definition, different pruning efficiencies of global and layer-wise pruning strategies are compared, and two-phase pruning method is proposed. The proposed pruning method can achieve a reduction of 88% filters and 46% inference time reduction on VGG16 within 2% accuracy degradation.

Table of Contents

List of Tables

List of Figures

Preface

This dissertation is submitted in partial fulfillment of the requirements for Chuhan Min's degree of Doctor of Philosophy in Electrical and Computer Engineering. The work is original to the best of my knowledge, except where acknowledgement and reference are made to the previous work.

I would like to thank my advisor, Yiran Chen, for his mentorship throughout graduate school. I am so thankful to be a student of him and work with excellent colleagues at CEI lab. I would also like to thank those who have graciously served on my committees through the years. Even though I always came into these meetings in a nervous state of mind and emotion, I always left each meeting feeling empowered and enthusiastic about the next stage. I've been fortunate in having such encouraging members on my committees, so thank you all. Last but not least, my deepest gratitude to my parents for their unconditional love, support, and patience. Without them, I would have never been able to successfully finish my degree. It has been a great journey over these years and now has come to an end. It's time for a new and exciting adventure.

1.0 Introduction

1.1 Motivation

The development of Deep Neural Networks (DNNs) has shown remarkable progress in improving accuracy over the years. DNNs have demonstrated state-of-the-art performance and are widely adopted in many artificial intelligence (AI) applications from computer vision to speech recognition etc. However, the unprecedented accuracy comes at a cost of high computational complexity and intensive memory accesses. For example, the ResNet50 [82] with 50 convolutional layers needs over 95MB memory for storage and over 3.8 billion floating number multiplications when processing an image. After discarding some redundant weights, the network still works as usual but saves more than 75% of parameters and 50% computational time. The limitation hinders DNN model deployment on resource-constrained devices, such as smartphones and wearable gadgets.

Running state-of-the-art DNNs on current systems mostly relies on either general-purpose processors, ASIC designs, or FPGA accelerators, all of which suffer from data movements due to limited on-chip memory and data transfer bandwidth. Previous work [8] proposed to use dedicated memory block such as SRAM to store large size of network weights and input data. In the context of efficient DNN implementation, prior works employ a variety of techniques to perform DNN computation but the memory still takes up to 90% of the energy consumption according to [71].

Processing-In-Memory (PIM) is a promising solution to address the issue by implementing logic circuits within memory. Instead of sending a large size of data to the processing elements, PIM performs a part of the operations such as bit-wise manipulation inside memory to reduce data movements so that the overall application performance can be accelerated by avoiding memory access bottleneck. The challenges in adopting PIM as DNN accelerator are summarized as follows: (1) massive number of operations for storage/compute, (2) PIM features (e.g., packet-based protocol).

1.1.1 Challenge 1: massive number of operations for storage/compute

In recent years, deep neural networks have recently received lots of attention, been applied to different applications and achieved dramatic accuracy improvements in many tasks. These works rely on deep networks with millions or even billions of parameters, and the availability of GPUs with very high computation capability plays a key role in their success. For example, [47] achieved breakthrough results in the 2012 ImageNet Challenge using a network containing 60 million parameters with five convolutional layers and three fully-connected layers. Usually, it takes two to three days to train the whole model on ImagetNet dataset with a NVIDIA K40 machine. In architectures that rely only on fully-connected layers, the number of parameters can grow to billions [15]. As larger neural networks with more layers and nodes are considered, reducing their storage and computational cost becomes critical, especially for some real-time applications such as online learning and incremental learning. In addition, recent years witnessed significant progress in virtual reality, augmented reality, and smart wearable devices, creating unprecedented opportunities for researchers to tackle fundamental challenges in deploying deep learning systems to portable devices with limited resources (e.g. memory, CPU, energy, bandwidth). Efficient deep learning methods can have significant impacts on distributed systems, embedded devices, and FPGA for Artificial Intelligence. For devices like cell phones and FPGAs with only several megabyte resources, how to compact the models used on them is also important.

Achieving these goal calls for joint solutions from many disciplines, including but not limited to machine learning, optimization, computer architecture, data compression, indexing, and hardware design. In this paper, we review recent works on compressing and accelerating deep neural networks. The parameter pruning and sharing based methods explore the redundancy in the model parameters and try to remove the redundant and uncritical ones. Low-rank factorization based techniques use matrix/tensor decomposition to estimate the informative parameters of the deep CNNs. The approaches based on transferred/compact convolutional filters design special structural convolutional filters to reduce the parameter space and save storage/computation. The knowledge distillation methods learn a distilled model and train a more compact neural network to reproduce the output of a larger network.

1.1.2 Challenge 2: PIM features

One category of PIM is near-memory processing. The underlying principle is processing in proximity of memory by physically placing monolithic compute units (multi-core, GPU, FPGA, ASIC etc.) closer to monolithic memory to minimize data transfer cost. Another category of PIM is in-memory processing, i.e. processing inside memory which seamlessly embeds computation in memory array. For near-memory processing, computation is still performed in digital signals and power consumption is dominated by memory read operations. For in-memory processing, memory access and computation are combined, resulting mixed signal computation.

As the compute units become more tightly coupled with memory, one can exploit more fine-grained parallelism for better performance and energy efficiency. In-memory processing is expected to achieve highest bandwidth with significant power and latency reduction. However, since in-memory processing involves mixed signal processing, this option is less preferred considering:

- non-idealities of analog compute (sensitivity to process and temperature variations)
- conversion cost from analog to digital is expensive
- increase size of bit cell and increase size in peripheral will reduce storage density
- limited precision that can be stored in each storage element

Embedded DRAM (eDRAM) and 3D stack memory (SRAM, DRAM) are two promising technologies for the memory system of NN accelerators. In this work, 3D stack memory is preferred for its high parallelism. The two well-known realizations of 3D memory are Micron's Hybrid Memory Cube (HMC) [13] and JEDEC High Bandwidth Memory (HBM) [59]. Hybrid Memory Cube (HMC) vertically integrates multiple DRAM dies on top of a logic layer within a single package by leveraging low-capacitance through-silicon vias (TSVs). Nowadays, each 3D memory stack can use thousands of TSVs and provide an order of magnitude higher bandwidth (160 to 250 GBps) with 3 to 5 times lower access energy than DDR3 [6]. In addition, the large number of TSVs can be organized into multiple, independently-operated channels that exploit memory level parallelism. 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.

1.2 Dissertation Contribution

In this dissertation, we propose hardware/software co-design for neural network accelerator to handle the design challenges from massive number of operations for storage/compute and specific PIM features that limit performance.

Based on our observation on the different pruning efficiencies of global and layer-wise pruning strategies, we propose to combine these two strategies to achieve a higher compression ratio of the neural network compared to applying only one strategy in network pruning. The proposed pruning method can achieve a reduction of 88% filters and 46% inference time reduction on VGG16 within 2% accuracy degradation.

With compressed model, we propose *NeuralHMC*, the first HMC-based accelerator for efficient DNN execution. 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 [9].

We then extend the proposed accelerator on Generative Adversarial Network (GAN) with dynamic scheduling. In this work, we offload computational kernels to PIM module with parsed annotations from compilation time and use a SRAM buffer to reduce data movement. Also, we an asynchronous training pattern to achieve blob level parallelism. Our experimental results show that the proposed method can achieve $1.6\times$ speedup.

2.0 Two-Phase Filter Pruning Based on Conditional Entropy

Deep Convolutional Neural Networks (CNNs) offer remarkable performance of classifications and regressions in many high-dimensional problems and have been widely utilized in real-word cognitive applications. However, high computational cost of CNNs greatly hinder their deployment in resource-constrained applications, real-time systems and edge computing platforms. For example, smartphones nowadays cannot even run object classification with AlexNet [47] in real-time for more than an hour. In addition to accuracy, the design of modern CNNs is starting to incorporate new metrics to make it more favorable in real-world environments. The trend is to simultaneously reduce the overall CNN model size and/or simplify the computation while going deeper. This can be achieved either by pruning the weights of existing CNNs, i.e., making the filters sparse by setting some of the weights to zero, or by designing new CNNs with (1) highly bitwidth-reduced weights and operations (e.g., XNOR-Net [70]), or (2) compact layers with fewer weights [55, 83, 35, 82].

To deploy CNNs in resource-constrained devices, model compression has been a promising research topic in past decades. We notice that CNNs with a large scale usually have significant redundancy of their filters and feature channels, which offer a large compression and pruning space. To overcome this challenge, we propose a novel filter-pruning framework – two-phase filter pruning based on conditional entropy (2PFPCE), to compress CNN models and reduce inference time with minimum performance degradation. In our proposed method, we formulate filter pruning process as an optimization problem and propose a novel filter selection criteria measured by conditional entropy. Based on the assumption that the representation of neurons shall be evenly distributed, we also develop a maximum-entropy filter freeze technique that can reduce over-fitting. Two filter pruning strategies – global and layer-wise strategies, are compared. Our experiment result shows that combining these two strategies can achieve a higher neural network compression ratio than applying only each of them under the same accuracy drop threshold. Two-phase pruning, that is, combining both global and layer-wise strategies, achieves $\sim 10\times$ FLOPs reduction and 46% inference time reduction on VGG-16, with 2% accuracy drop.

2.1 Introduction

Deep Convolutional Neural Networks (CNNs) have been widely utilized in many applications and achieved remarkable success in computer vision [84], speech recognition [1], natural language processing [12], etc. Going deeper has been proven as an effective approach to improve the model accuracy in solving high-dimensional problems [5, 84]. However, when the network depth increases, the number of parameters of the neural network increases too.

Model compression techniques aim at reducing the storage and computational costs of deep neural networks (DNNs) [49, 17, 26, 91]. Network pruning is one important example of model compression techniques that can reduce the network complexity and suppress the over-fitting issue simultaneously. Han et al. [26, 27] proposed to reduce network parameters by pruning the weights with small magnitudes and then retrain the network in an iterative manner to maintain the overall accuracy. Majority of the pruned parameters is actually from fully connected layers. Since fully connected layers contribute to very small portion of the total floating point operations (FLOPS), e.g., less than 1% in VGG-16 [76], the overall computational cost reduction achieved by this method is very limited [91]. Moreover, the random distribution of the removed weights in memory hierarchy also incurs a high cache miss rate, which greatly harms the actual performance acceleration obtained in real systems [91]. Recently, more and more works focus on pruning convolutional layers to reduce computational cost in inference time [54, 60, 57]. Despite of the significant weight sparsity in fully connected layers, the non-structured random connectivity ignores the impact of cache and memory accesses as indicated in [91]. In some recent work on CNNs [84, 30], the fully connected layers are replaced by average pooling layers in order to build a deep architecture with hundreds of layers. The computational cost of the convolutional layers, hence, dominates the overall computational cost of CNNs when the networks become deeper. We note that CNNs with a large scale usually have significant redundancy of their filters and feature channels, which offer a large compression and pruning space.

We propose a **T**wo-**P**hase **F**ilter **P**runing framework based on **C**onditional **E**ntropy, referred to as *2PFPCE* in this chapter, to prune the filters of CNNs based on conditional entropy in a two-phase manner. The framework mechanism is as shown in Figure 1. The key idea of our proposed approach is to establish a quantitative connection between the filters and the model accuracy. We

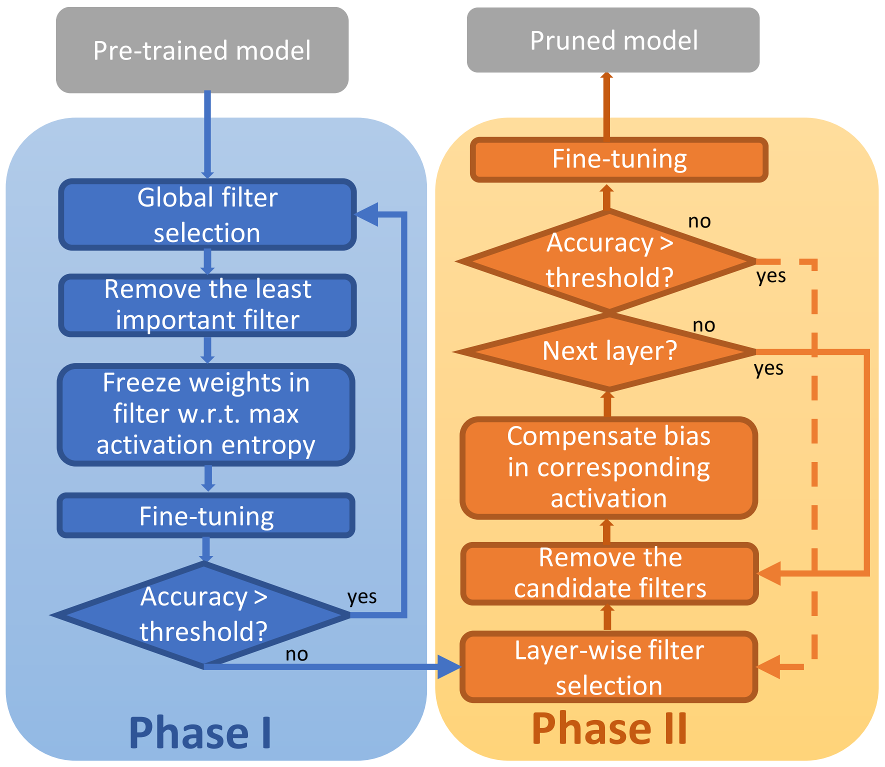

Pre-trained model
Global filter selection
Remove the least important filter
Freeze weights in filter w.r.t. max activation entropy
Fine-tuning
Accuracy > threshold?
yes
no
Phase I
Pruned model
Fine-tuning
Accuracy > threshold?
no
yes
Next layer?
no
yes
Compensate bias in corresponding activation
Remove the candidate filters
Layer-wise filter selection
Phase II

Figure 1: Basic scheme of our proposed 2PFPCE. This scheme show the network pruning process consisting phase I and II.

adopt global pruning as Phase I and layer-wise pruning as Phase II, respectively. In Phase I, the filters with the minimum conditional entropy is pruned filter-by-filter, followed by an iterative fine-tuning constrained by an accuracy drop threshold. In Phase II, the filters are pruned layer-by-layer in a greedy manner based on conditional entropy, followed by also a fine-tuning of the neural network constrained by the accuracy loss threshold. Our major contributions can be summarized as:

- We calculate the conditional entropy over the filters in a convolutional layer, i.e., the distribution of entropy conditioned on the network loss. We also propose to use conditional entropy as a criteria to select the filters to be pruned in our method:
- When pruning filter in global and layer-wise method, we observe a difference in accuracy/filter number at different pruning ratios. Based on our observation on the different pruning efficien-

cies of global and layer-wise pruning strategies, we propose to combine these two strategies to achieve a higher compression ratio of the neural network compared to applying only one strategy in network pruning.

- Based on the assumption that the information of the neurons in a layer shall be uniformly distributed, we propose a novel fine-tuning approach where the weights in a filter corresponding to the neuron with the maximum entropy is kept constant during the back-propagation to reduce over-fitting.

  Experimental results show that 2PFPCE can achieve a reduction of 88% filters on VGG16 with only 2% accuracy degradation. The data volume is decreased from 310784 bytes to 49165 bytes and the inference time is $\sim 54\%$ of the original model. Under the same accuracy drop threshold, our experiments show that applying a combination of both methods achieves higher compression ratio than applying either independently.

2.2 Related works

2.2.1 Model compression

The compression techniques of convolutional layers can be roughly categorized into the following three types according to their approximation levels: **Pruning** reduces the redundancy in parameters which are not sensitive to the performance at a level of weight and filter. Network pruning, which aims at reducing the connectivities of the network, is a classic topic in model compression and has been actively studied in the past years. Pruning has been performed at weight level [26, 91] and filter level [54, 60]. **Quantization** compresses the network by reducing the number of bits required to represent the weights [26]. Binarization [70] is an extreme case of quantization where each weight is represented using only 1-bit.

Convolution reconstruction divides convolution into subproblems based on organization of filters at layer level. Low rank approximation [16, 97, 85, 36] imitates convolutional operations by decomposing the weight matrix as a low rank product of two smaller matrices without changing the original number of filters. Based on the correlation between groups of filters, [11, 95] build a convolutional layer from a group of base filters. FFT convolution [89] designs a set of leaf filters with well-tuned in-register performance and reduces convolution to a combination of these filters by data and loop tiling.

Knowledge distillation [33] compresses an ensemble of deep networks (teacher network) into a student network with similar depth by applying a softened penalty of the teacher's output to the student. This compression method works at network level.

There is no golden rule to measure which one of the three kinds of approach is the best. In this work, we focus on filter pruning. There exist some heuristic criteria to evaluate the importance of each filter in the literature such as APoZ (Average Percentage of Zeros) [34], ℓ_1-norm [54] and Taylor expansion [60].

- APoZ (Average Percentage of Zeros) [34]: calculates the sparsity of each channel in output feature map as its importance score $\frac{\sum_k^N \sum_j^M f(O_{c,j}(k)=0)}{N \times M}$.
- ℓ_1-norm [54]: measure the relative importance of a filter in each layer by calculating the sum of its absolute weights $\sum |F_{i,j}|$, i.e., its ℓ_1-norm $\|F_{i,j}\|_1$.
- Taylor expansion [60]: approximate change in the loss function with accumulation of the product of the activation and the gradient of the cost function w.r.t. the activation $\left|\frac{1}{M} \sum_m \frac{\delta C}{\delta z_{l,m}^{(k)}} z_{l,m}^{(k)}\right|$.

Unlike the above mentioned criterion, we directly quantize contribution of each filter to accuracy via conditional entropy. The details will be discussed in the following section.

2.2.2 Information Plane

There is a fast-growing interest in understanding DNNs and this motivates our information guided pruning. [87] proposed to analyze DNNs in the *Information Plane*. The idea is to optimize the Information Bottleneck (IB) trade-off between compression and prediction, successively, for each layer.

Two properties of the IB are very important in the context of network pruning. The first is the necessity of redundancy during model training. According to [87], the Stochastic Gradient Decent (SGD) optimization has two different and distinct phases: empirical error minimization (ERM) and representation compression. In ERM, redundancy is necessary since the high non-convex optimization is hard to be solved by current technologies. Considering convergence rate, reducing model size after its training is more time efficient. The second is the conditional distribution of output y on $\widetilde{x}$, i.e., $p\left(y \mid \widetilde{x}\right)$, where $\widetilde{x}$ is the compact expression of input x following the Markov chain condition $Y \leftarrow X \leftarrow \widetilde{X}$. It is important to note that this not a modeling assumption and the quantization $\widetilde{x}$ is not used as a hidden variable in a model of the data. Hence, a network can be decomposed to a cascade of sub-networks using its compact input feature maps as input and original model's output as output.

On information plane, Mutual Information (MI) quantifies the average number of relevant bits that the input variable X contains about the label Y as:

$$\begin{aligned} I\left(X,Y\right) &= \sum_{(x,y)\in A} p(x,y)\log[\frac{p(x,y)}{p(x)p(y)}] \\ &= \sum_{(x,y)\in A} p(x,y)\log[\frac{p(x \mid y)}{p(x)}] \\ &= H(X) - H(X|Y). \end{aligned} \tag{2.1}$$

Minimal sufficient statistics I_{min} is defined as sufficient statistic with the least information. The connection between mutual information and minimal sufficient statistics is based on its invariance to invertible transformations:

$$I(X,Y) = I_{min}(\psi(X),\phi(Y)) \tag{2.2}$$

for any invertible functions ψ and ϕ. The invariance of the information measures to invertible transformations comes with a high computational cost. For deterministic functions, the mutual information is insensitive to the complexity of the function or the class of functions it comes from [74]. If we have no information on the structure or topology of X, there is no way to distinguish low complexity classes from highly complex classes by using the mutual information alone.

In this work, instead of utilizing noise insensitive MI criteria, we propose to adopt conditional entropy in terms of error probability in guessing a finitely-valued random variable X given another random variable Y.

2.3 Conditional entropy based compression

In this section, we first formulate compression as an optimization problem, then propose a conditional entropy based filter selection criteria and compare the statistical result of CIFAR10 on pre-trained VGG-16 model. Furthermore, we discuss the relationship of error probability and conditional entropy.

2.3.1 Problem formulation

In [74, 73], each layer is seen as a *single* random variable. And the output distribution is calculated by joining all neuron outputs in this layer. However, the same method might not be optimal due to loss of the structure feature on both the input and the feature map.

Therefore we adopt the procedure in [45, 44] to estimate the mutual information of each convolutional layer. The first step is to use the entropy of each layer's output as the measurement of the information flow. The *activation entropy* can be calculated using the function below where p_i denotes the probability of *i*-th filter in the feature map.

$$H_{C_n} = \sum_{i=0}^{n} p_i * \log p_i. \tag{2.3}$$

Considering the general scenario of a neural network whose operation is parametrized by a vector $\theta \in \Re^W$ (representing the weights), and whose input/output characteristics are described by a conditional probability distribution $p_\theta(x_{out} \mid x_{in})$. Here $x_{in} \in \Re^N$ and $x_{out} \in \Re^M$ denote input and output vectors, respectively. The performance of this network on a given input is measured by a loss function $\varepsilon(x_{in}, x_{out})$. If the probability of an input x_{in} is defined as $p(x_{in})$, the global error made by a network with parameter is given by

$$E_\theta = \sum^{x_{in}} \sum^{x_{out}} \epsilon(x_{in}, x_{out}) p_\theta(x_{out} \mid x_{in}) p(x_{in}). \tag{2.4}$$

We define $p_\theta(x) = p_\theta(x_{out} \mid x_{in}) p(x_{in})$, $x = (x_{in}, x_{out}) \in \Re^{N+M}$. It now combines both the parametrized properties of the network and the likelihood of the input data.

When recognizing neural network as a stack of sub-networks, the above definition holds true for each layer and thus error in convolutional layers can be calculated by:

$$E_{\theta}^{C} = \sum^{C_{in}} \sum^{x_{out}} \epsilon(C_{in}, x_{out}) p_{\theta}(x_{out} \mid C_{in}) p(C_{in}). \tag{2.5}$$

Here C_{in} denotes the input feature maps of the convolutional layer. In this way, we reorganize the compression problem as an optimization problem and minimize the distance $\left\| E_{\theta}^{C} - E_{\theta'}^{C'} \right\|$, where C'_{in} is a minimum subset of C_{in}.

We adopt an 1-dimensional binary vector σ: 1 indicating the filter is selected and 0 indicating the filter is discarded. Notation x_{σ} indicates the vector of selected features, that is, the full vector x projected onto the dimensions specified by σ. Notation $x_{\widetilde{\sigma}}$ is the complement, that is, the unselected features. The full feature vector can therefore be expressed as $x = \{x_{\sigma}, x_{\widetilde{\sigma}}\}$. As aforementioned, we assume the process p is defined by a subset of the features, so for some unknown optimal vector σ^{*}, $p\left(y \mid x\right) = p\left(y \mid x_{\sigma^{*}}\right)$. We approximate p using an hypothetical predictive model q, with two layers of parameters: σ representing which filters are selected and τ representing the parameters used to predict y. Our problem statement is to identify the minimal subset of features such that we maximize the conditional likelihood of the training labels w.r.t. these parameters. For i.i.d data $D = \left\{\left(x^{i}, y^{i}\right); i = 1..N\right\}$ the conditional likelihood of the labels given parameters $\{\sigma, \tau\}$ is

$$L\left(\sigma, \tau \mid D\right) = \prod_{i=1}^{N} q\left(y^{i} \mid x_{\sigma}^{i}, \tau\right). \tag{2.6}$$

The (scaled) conditional log-likelihood is

$$l = \frac{1}{N} \sum_{i=1}^{N} \log q\left(y^{i} \mid x_{\sigma}^{i}, \tau\right). \tag{2.7}$$

This is the error function we wish to optimize w.r.t. the parameters $\{\sigma, \tau\}$; the scaling term has no effect on the optima, but simplifies the exposition later. We now introduce the quantity $p\left(y \mid x_{\sigma}\right)$: this is the true distribution of the loss given the selected filters x_{σ}. Multiplying and dividing q by $p\left(y \mid x_{\sigma}\right)$, we can re-write the Eq. (2.7) as:

$$l = \frac{1}{N} \sum_{i=1}^{N} \log \frac{q\left(y^{i} \mid x_{\sigma}^{i}, \tau\right)}{p\left(y^{i} \mid x_{\sigma}^{i}\right)} + \frac{1}{N} \sum_{i=1}^{N} \log p\left(y^{i} \mid x_{\sigma}^{i}\right). \tag{2.8}$$

The second term in Eq. (2.8) can be similarly expanded to introduce the probability $p\left(y \mid x\right)$ term for next step. These are finite sample approximations, drawing data points i.i.d. w.r.t. the distribution. We use $E_{xy}\left\{.\right\}$ to denote statistical expectation. For the convenience, we negate the above to turn our maximization problem into a minimization, or:

$$\begin{aligned} -l \approx & E_{xy}\left\{\log \frac{p\left(y \mid x_{\sigma}\right)}{q\left(y \mid x_{\sigma}, \tau\right)}\right\} + E_{xy}\left\{\log \frac{p\left(y \mid x\right)}{p\left(y \mid x_{\sigma}\right)}\right\} \\ & - E_{xy}\left\{\log p\left(y \mid x\right)\right\}. \end{aligned} \tag{2.9}$$

In our experiments, we use the above training loss as the single variable. Then our problem statement is to identify the minimal subset of features such that we *maximize the conditional likelihood of the training loss w.r.t. these parameters*.

2.3.2 Filter selection algorithm

Our filter selection algorithm is illustrated in Algorithm 1. For each sample, we calculate the cross entropy loss and output activation corresponding to each filter. To achieve discrete statistical requirement, each parameter is multiplied by a factor of $1e4$ and quantized as 32-bit integer. For each filter, c_val denotes 1-D distribution on output activation and c_bins denote a 2-D statistics on output activation conditioned on the loss.

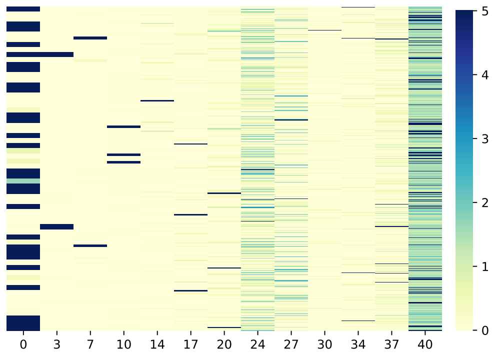
c_total is the number of activations per filter, i.e., samples in a dataset. act_ent denotes the entropy of feature map, providing the distribution of output activation across the dataset. Notice here zero activations are excluded because it's considered to contain no information w.r.t. the next layer. Given the probability of a specific output activation, ent_i denotes the entropy of output activation conditioned on the distribution of cross entropy loss. con_ent denotes the conditional entropy of a filter, which is an accumulation of ent_i. Then, the con_ent is sorted in ascending and the filters corresponding to the top-r con_ent are selected to be removed. In the above single-layer illustration, the layer to prune is predefined. We note that this can be generalized to multiple layers or the whole model.

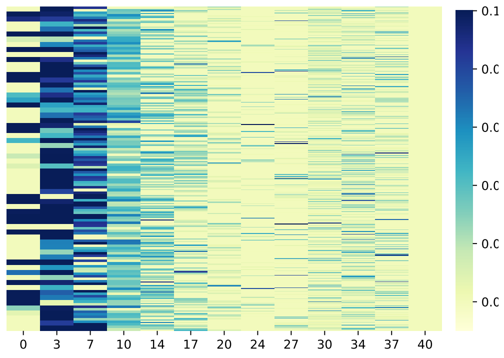
Algorithm 1 Filter selection algorithm.

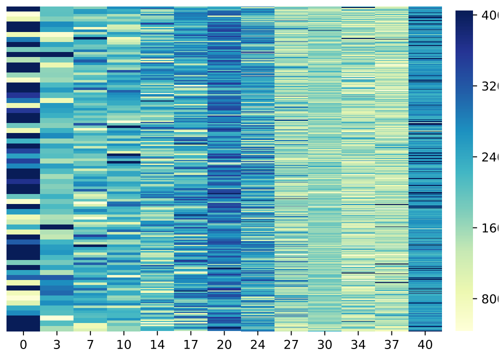
Input: a baseline model $\boldsymbol{M}$, convolutional layer to prune l, training dataset $\mathbf{x}_{train}$, number of filters to prune r

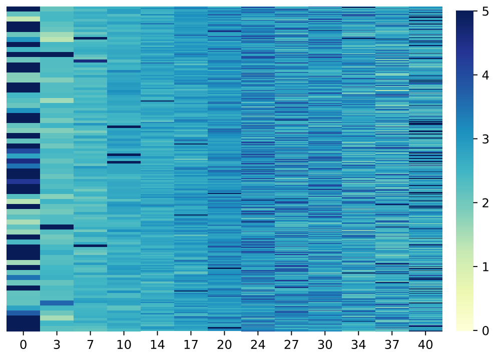
Output: candidate filter(s) to prune σ^r

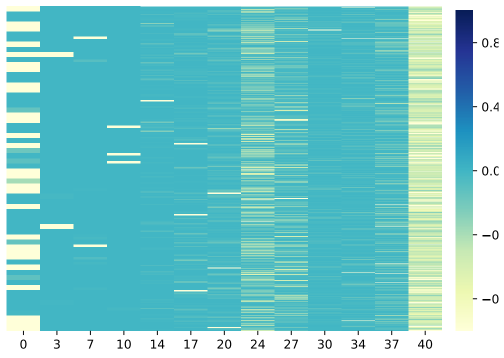
```
procedure CONDITIONALENTROPY_CALCULATION
    $eps\_h = 1e4$
    $criteria = CrossEntropy(reduce = false)$.
    for $batch\_idx$ in $batches$ do
        $output \leftarrow \boldsymbol{M}(\mathbf{x}_{train}[batch\_idx])$.
        $loss[batch\_size] \leftarrow criteria(output, target)$.
        $j = eps\_h * loss[batch\_size]$.
        for $k$ in $l$ do
            $i = eps\_h * fmap\_out^l[k]$.
            $ans[k].extend(i, j)$.
            $c\_bins[i][j] += 1$.
            $c\_val[i] += 1$.
    $c\_total = \sum_{i}^{c\_val} c\_val[i]$.
    $act\_ent = \sum_{j \neq 0}^{c\_val} -\frac{c\_val[j]}{c\_total} * \log \frac{c\_val[j]}{c\_total}$
    $ent_i = \frac{c\_val[i]}{c\_total} \sum_{j} -\frac{c\_bins[i][j]}{c\_val[i]} * \log \frac{c\_bins[i][j]}{c\_val[i]}$
    $con\_ent = \sum_{i} ent_i$
    sort $con\_ent$ of each filter in layer $l$ ascending
    add corresponding filter of top-$r$ $con\_ent \rightarrow \sigma^r$
```


(a) Maximum conditional entropy.


(b) Activation of maximum conditional entropy.


(c) Number of zero activations.


(d) Average conditional entropy of filter.


(e) Maximum conditional entropy filter ratio.

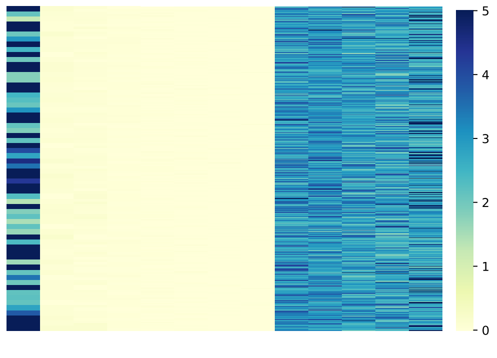

(f) Zero activation ratio.

Figure 2: Statistical result of VGG-16 on CIFAR10.

2.3.3 Statistical result of CIFAR10 on VGG-16

As one of the core concepts in convolutional networks, feature maps reveals huge amount of information about the information flow within the network. Entropy is a commonly used metric to measure the disorder or uncertainty in information theory. A larger entropy value means the system contains more information. In our filter pruning scenario, if a channel of activation tensor contains less information, its corresponding filter is then less important, thus could be dropped. Thus, we using statistical variable like conditional entropy, to measure the connection between feature map and error probability.

Figure 2 is the statistical result of filters in each convolutional layer of CIFAR10 on VGG-16. The x-axis indicate the index of convolution layer in the model. Figure 2a and Figure 2b shows the maximum ent_i and its corresponding output activation, respectively.

Figure 2c depicts the number of zero activations and Figure 2f shows the negative zero activation ratio. We observe zero activation mostly reside in the first and last several convolutional layers.

Figure 2d illustrates total conditional entropy of each filter. Figure 2e is the ratio of maximum conditional entropy to total conditional entropy and it's remains relatively uniform in different layers.

2.4 Experiments

2.4.1 Experiment setup

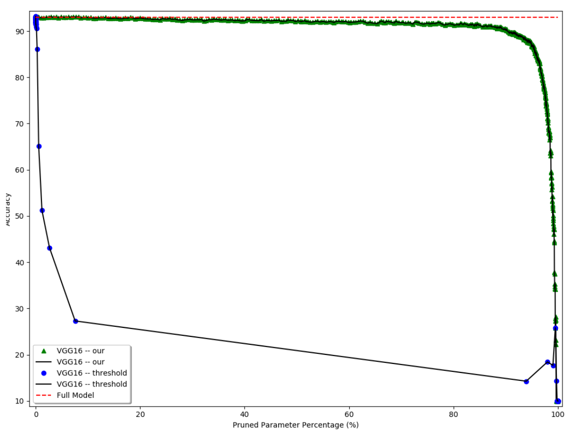
In the initial stage of our experiment, we assume entropy $H(T_n)$ can be used to measure information flowed in the neural network. To testify our assumption, we perform several experiments with MNIST datasets and a two-layered CNN. We added Gaussian noise to the picture and see how the entropy in neural networks would change as the power of noise changes. If the entropy effectively measures the information uncertainty in CNN, H_{T_1} and H_{T_2} should increase as the loss increases. The result is affirmative, that is, H_{T_1} and H_{T_2} do increase as expected. As summarized in Algorithm 1, the proposed Conditional Entropy Filter Selection algorithm can dynamically remove the filters. Specifically, the key is to keep updating the pruned filters in the training stage. Such an updating manner brings several benefits. It not only keeps the model capacity of the compressed deep CNN models as the same as the original models, but also avoids the greedy layer by layer pruning procedure, allowing pruning almost all layers at the same time. After each epoch, the $\ell2$-norm of all filters are computed for each weighted layer and used as the criterion of our filter selection strategy.

Here, we trained a VGG-16 [76] network with *tanh* activation function on CIFAR10 [46] and logged the output of each layers. The network has 13 convolution layers, each of which has 64 filters to 512 filters with a size of 3. The network converged at around 100 epochs, but we trained it for 400 epochs. The final accuracy is 99.6% on training set and 92.68% on test set. As we can observe from Figure 2d, conditional entropy in filters are almost uniformly distributed, except for the first convolutional layer where conditional entropy is slightly higher. Also, the number of zero activations shown in Figure 2c indicates the same fact that the first convolutional layer in VGG-16 is greatly redundant in CIFAR10 classification.

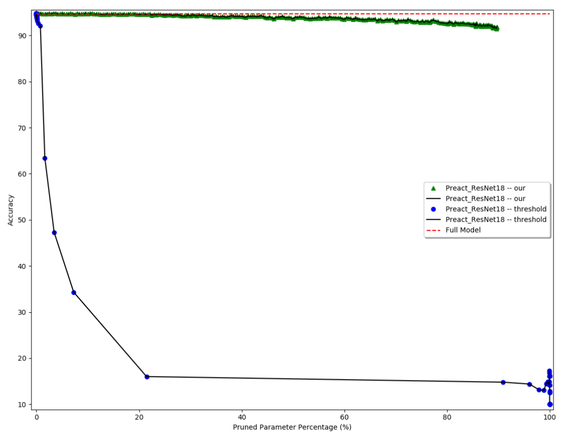
However, the philosophy that each neuron in network is interchangeable and informative equivalent [92] indicates a uniform distribution of information across the layers. As a result, filters are supposed to be pruned in a layer-wise manner where the percentage of the filters to be pruned in each layer should keep at a similar value, e.g. prune 16 filters in a convolutional layer with 64 filters, and pruning 32 filters of a convolutional layer with 128 filters, etc.

In our experiments, we evaluate the tradeoffs between accuracy and pruning ratio in global and layer-wise approaches, respectively. Inspired from above observations, we propose a two phase filter pruning framework based on conditional entropy, namely *2PFPCE*. The filter selection criteria is described in Algorithm 1. The procedure is shown in Figure 1.

In phase I, filters are pruned and fine-tuned iteratively until the accuracy drop reach the threshold (1%). The purpose is to remove the redundancy filters w.r.t. the dataset. Notice that dataset plays a crucial role during the model compression: The number of involved features in an 1000-category dataset is probably much larger than that in a 10-category dataset. After a layer is pruned, weights in the filter w.r.t the maximum activation entropy are kept constant and cannot be updated during fine tuning. Similar to dropout [80], this aims at penalizing any single neuron that may overly fitted to the dataset.



(a) Accuracy vs. pruning ratio of VGG16.



(b) Accuracy vs. pruning ratio of Preact-ResNet18.

Figure 3: Trade-offs between computation complexity and classification accuracy.

In phase II, filters are pruned in a layer-wise manner: in each iteration, a small portion filters (1/32 or 1/16) of each layer are pruned until the accuracy drop reach the threshold γ. The threshold γ is a hyper-parameter and can be adjusted to satisfy certain application constraint. To retain the information in pruned filters and avoid time consuming fine-tune, we update the bias term as the activation of maximum conditional entropy filter (see Figure 2b).

2.4.2 Global Pruning Approach

In this experiment, we evaluate the tradeoffs between computation complexity and classification accuracy in global pruning approach. To comprehensively understand global pruning, we test the accuracy of different pruning rates for Preact-ResNet18 and VGG-16.

As shown in Figure 3a and Figure 3b, the relationships between accuracy and pruning ratio are similar in both networks. Without fine-tuning, the accuracy in VGG-16 and ResNet-18 decrease significantly to 27% and 16% within a pruning ratio of 10% and 21%, respectively. As the pruning rate increases, the accuracy of the pruned model first rises above the baseline model and then drops approximately linearly. For the pruning rate before turning point, the accuracy of the pruned model is higher than that of the baseline model. This shows that our global pruning has a regularization effect on the neural network because global pruning reduces the over-fitting of the model. After the turning point, in VGG-16 the accuracy slowly decrease to 15% as the pruning ratio increases while in ResNet-18 the accuracy drop is within 1%. With fine-tuning, each network can remain its accuracy with a pruning ratio below 20% and then slowly degrade with a marginal accuracy drop between 1%-2% until the pruning ratio reaches 90%. The results also show that the performance of DNNs can recover from small disturb with fine tuning. The contribution of each filter is very similar in a converged convolutional neural network. Through the experiment results on VGG-16 and ResNet-18, we can conclude that our global pruning approach can prune about 40% and 80% filters with 1% and 2% accuracy compromises, respectively.

This result is reasonable, since pruning too many filters at one time would greatly harm the overall generalization ability of CNN models, making it hard to restore its original performance. In fact, if we adopt the same network architecture but train the model from scratch, we cannot obtain the same level accuracy.

2.4.3 Layer-wise Pruning Approach

Layer-wise pruning iterates over the filter selection, filter pruning and reconstruction steps. After the model converges, we can obtain a sparse model containing many *zero filters*. One *zero filter* corresponds to one feature map. The features maps that corresponding to those *zero filters* will always be zero during the inference procedure. There will be no influence on accuracy if we remove these filters as well as the corresponding feature maps.

Table 1a compares the obtained accuracy of different filter importance criterion under the same pruning ratios. For a fair comparison, the DNNs are pruned in a layer-wise approach and fine-tuned for the same number of epochs to recover the accuracy after pruning. At each pruning iteration, we remove a certain percentage of feature maps and then perform 20 minibatch SGD updates with batch-size = 32, momentum = 0.9, learning rate = 10^{-4}, and weight decay = 10^{-4}. The pruning ratio is guided by a feedback loop in fine tuning. As we can see from the table, our proposed conditional entropy filter selection criteria outperforms counterparts by achieving the highest accuracy on both pruning ratios of 25% and 50%.

Table 1b compares the obtained pruning ratios of filter that achieved by all methods at approximately the same error rate, e.g., within 2% accuracy loss. The baseline is L1-Norm which has a pruning ratio of 60% on both networks. Our experiments confirmed that conditional entropy based criterion can always achieve the highest pruning ratio with a marginal accuracy loss.

(a) Filter importance criteria vs. accuracy.

VGG16/CIFAR10: 92.98%				
Prune	L1-Norm	APoZ	Act. ent	Cond. ent
25%	92.86%	92.94%	92.93%	93.50%
50%	92.11%	92.02%	92.00%	92.76%
ResNet50/CIFAR10: 93.16%				
20%	94.42%	94.36%	94.33%	94.84%
50%	94.48%	94.25%	94.42%	94.44%

(b) Filter importance criteria vs. pruning ratio.

VGG16/CIFAR10: 92.98%				
Acc.	L1-Norm	APoZ	Act. ent	Cond. ent
91.0(±0.3)%	1.0×	0.88×	1.27×	1.32×
ResNet50/CIFAR10: 93.16%				
92.0(±0.2)%	1.0×	0.93×	0.96×	1.05×

Table 1: Comparison of filter importance criteria in layer-wise approach VGG-16/ResNet-50 on CIFAR10.

2.4.4 Stage Pruning Approach

We focus on reducing the number of convolutional feature maps and the total estimated floating point operations (FLOPs). During pruning we were measuring reduction in computations by FLOPs, which is a common practice [26]. Reduction in FLOPs result in monotonically decreasing inference time of the networks because of removing entire feature map from the layer. However, time consumed by inference depends on particular implementation of convolution operator, parallelization algorithm, hardware, scheduling, memory access pattern, etc. Unlike previous works where execution time is approximated through inaccurate proxy (e.g., FLOPS), in our experiments, we directly measure real-world inference time by executing the model on GPU (NVIDIA Titan Xp with CUDNN 8.0). The measured improvement of the inference time are summarized in Figure 4. A two-phase pruning approach is applied.

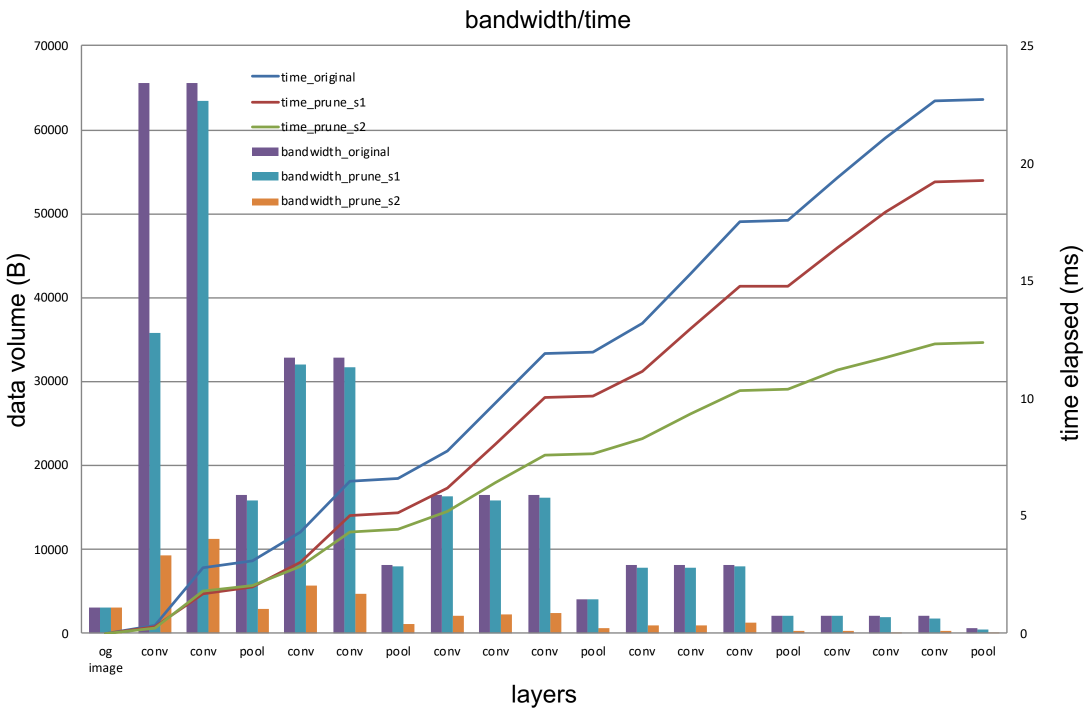


Figure 4: Stage pruning with 98% accuracy threshold.

Figure 4 shows that the two-phase pruning approach can achieve a pruning ratio of 88% within mere 2% accuracy loss. Here the adopted dataset is CIFAR10 with a minibatch size of 32. The inference time on the pre-trained VGG-16 is 22.69ms.

In Phase I, the number of filters decrease from 4224 to 3960 and total data volume reduces from 310784 bytes to 273658 bytes. The inference time reduces to 19.24ms. The removed filters are mostly from the first convolutional layer and this verifies our observation on redundancy: $\sim 60\%$ filters are removed from the first convolutional layer, reducing the total data memory requirement from 64M to 35M. Because of the high computing parallelization in GPU, the reduction in the inference time of a single layer is not as significant as that in data volume. In Phase II, the number of filters decrease from 3960 to 532 (or 87% pruning ratio) and total memory requirement from 273658 bytes to 49165 bytes . The inference time decreases to 12.34ms.

2.5 Summary

Inspired from the statistical result of information flow in neural networks, in this work, we propose to use conditional entropy as the filter selection criteria in filter pruning. For each sample in a dataset, we calculate the cross entropy loss and output activation corresponding to each filter. We then evaluate the filter selection criteria in a layer-wise pruning approach. The filters are pruned layer-by-layer in a greedy manner based on conditional entropy, followed by fine-tuning of the neural network constrained by the accuracy loss threshold. Experimental result shows that the performance of conditional based filter selection criteria outperforms the approaches based L1-Norm, APoZ and activation entropy. The proposed criteria can achieve 92.76% accuracy when pruning ratio is 50% and $\sim 93.5\%$ accuracy when pruning ratio is 25% on VGG16, while the baseline L1-norm results in 92.11% accuracy for 50% and 92.86% accuracy for 25% pruning ratio. To comply with the network information distribution, we adopt a two phase pruning framework which combines global approach with above layer-wise approach. The filters with the minimum conditional entropy are pruned filter-by-filter globally followed by layer-wise pruning. The above framework can achieve a pruning ratio of 87% within 2% accuracy drop of pre-trained VGG-16 model on CIFAR10.

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

As the traditional benefits for expanding the processing capability of computers through technology scaling has diminished with the end of Dennard scaling, limitations in traditional compute system, also known as "Memory Wall" [93] are being outpaced by the growth of data volume to the point where a new paradigm is needed. As such, Processing-in-Memory (PIM) has reignited interest among industry and academic communities, largely driven by the recent advances in technology (e.g., die stacking, emerging nonvolatile memory) and the ever-growing demand for large-scale data analytics. For instance, Hybrid Memory Cube (HMC) that stacks multiple DRAM dies on top of a CMOS logic layer using through-silicon-via (TSV) technology effectively addressed the previous limitations of implementing near-memory processing.

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. Optimizing data movement becomes crucial for these memory-intensive workloads. In these workloads, there exist some primitives and functions that overwhelm the overall data movements. Typically, each workload contains simple functions that contribute to a significant amount of the overall data movement. We investigate whether these functions are feasible to implement using PIM, given the limited area and power constraints of 3D memory.

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. *NeuralHMC* can dynamically offload the functions like multiply-accumulate (MAC) to PIM logic layer through a vault controller to boost the DNN execution on HMC. Experimental results show that *NeuralHMC* reduces the data movement by $1.4\times$ to $2.5\times$ (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 $4.1\times$ and reduces energy by $1.5\times$, on average.

3.1 Introduction

Deep Neural networks (DNNs) have demonstrated great potential in tasks such as object detection, recognition, and classification. In many benchmark suites [38, 67], 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 [47, 58]. 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* [9, 41] and *data locality* [8]. To improve *data reuse*, an on-chip scratchpad memory is introduced in [8] to support data reuse of local accelerators. In [41], 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 [10], 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 [13]. 3D stacking not only helps in providing a compact footprint, but it also reduces the latency of communication between circuits on different layers. One prominent example is Hybrid Memory Cube (HMC), which was announced by Micron Technology in 2011 [13]. Inherited from the concept of PIM, some 3D-stacked memory architectures [13] 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. With the HMC allowing to embed custom logic, novel non-von Neumann architectures can be accomplished, overcoming the performance gap while achieving a new path for scaling the computing performance. The logic layer in an HMC structure provides a convenient point where data transformations can be performed on the contents of the memory stack. Such transformations can filter data before the data are provided to a host processor for further processing, thereby reducing bandwidth requirements and latency at the interface between the host processor and memory. A broad spectrum of custom logic could be integrated into a mesh of HMCs, enabling holistic design-space explorations for computing systems in breadth and depth.

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 [25], 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 [9].

The rest of this paper is organized as follows: Section 3.2 introduces HMC architecture; Section 3.3 illustrates the motivation of *NeuralHMC*; Section 3.4 describes the design details of *NeuralHMC*; Section 3.5 shows the experimental setup and evaluation result; Section 3.6 concludes this work.

3.2 Background

3.2.1 Overview of HMC

Figure 5 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

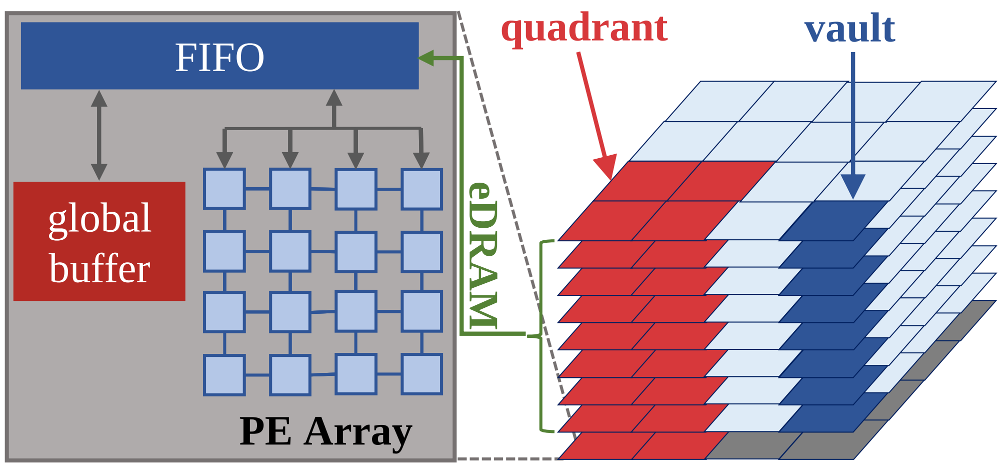


Figure 5: HMC module architecture.

(TSVs). As shown in Figure 5, 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 2.

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. Commands and data are transmitted in both directions across the link using a packet based protocol where the packets consist of 128-bit flow units called “FLITs.” These FLITs are serialized, transmitted across the physical lanes of the link, then re-assembled at the receiving end of the link. Three conceptual layers handle packet transfers:

- The physicallayer handles serialization, transmission, and deserialization.
- The link layer provides the low-level handling of the packets at each end of the link.
- The transaction layer provides the definition of the packets, the fields within the packets, and the packet verification and retry functions of the link.

Table 2: 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

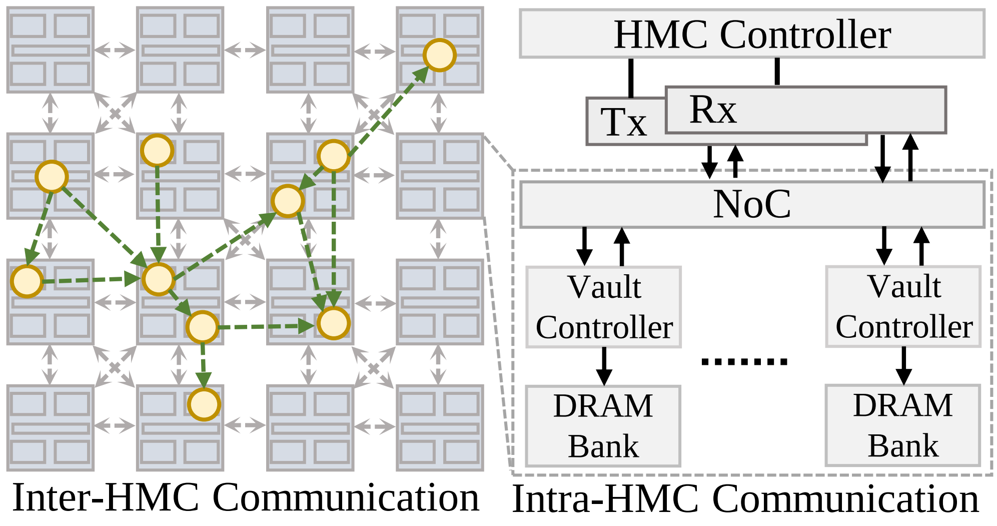


Figure 6: HMC communication.

3.2.2 HMC Communication

Two logical blocks exist within the link layer and transaction layer:

- The link master (LM), is the logical source of the link where the packets are generated and the transmission of the FLITs is initiated.
- The link slave (LS), is the logical destination of the link where the FLITS of the packets are received, parsed, evaluated, and then forwarded internally.

Requester: Represents either a host processor or an HMC link configured as a pass-thru link. Responder: Represents an HMC link configured as a host link. A responder transmits packets upstream to the requester. 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 6, 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 6. At 1.25GHz execution frequency [75], 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 3.4 and different external NoC designs are examined with DNN data reuse strategies in Section 3.5.

With multiple objectives for the HMC modeling tool set, three key aspects needed to be implemented. First, a holistic HMC simulator, capable of being adjusted easily regarding its internal components and properties. Second, a multi-HMC environment needed to be established that allowed to route HMC request and response packets to find a path through the HMC network to either store or load data or execute HMC commands. And last, significant simulation performance improvements were required, since single cycle or bandwidth accurate simulation models already tend to execute slowly or even poorly, whereas a multitude of these rapidly accumulate to an infeasible simulation. Consisting of multiple components such as the switch, the external links and internal communication, the quads, the vault controllers, vaults, and partitions and banks, one main objective for building the HMC simulator was to leverage actual freely available simulation tools. The initial simulator was based on the HMC-Sim that supported the HMC specification 2.0, since it offered the appealing opportunity to load 'custom memory cube' (CMC) operations and was already integrated into the SST full system simulation framework. As multiple aspects of HMC-Sim were either partially or barely working, such as the internal addressing of quads or vaults, the routing within a multi-HMC environment, or the internal processing of packets that does not rely on FLITs, the best piece i.e. in particular the vault handler was retrieved for the herewith established simulator. Moving forward, the simulator models HMCs according to the cutting-edge HMC specification 2.1, which allows for 4GB or 8GB cubes.

3.3 Motivation

3.3.1 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 [96]. 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 [9]. 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.

3.3.2 Potential of DNN Execution on HMC

A recent work named HMC-MAC [39] 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 [39] 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 [47] as an example, the numbers of parameters of each layer and the associated MAC operations are summarized in Table 3. 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.

3.3.3 Challenge of DNN Execution on HMC

However, the above estimation in Section 3.3.2 is too optimistic and ignores the communication cost and data access latency from/to the PEs. As aforementioned in Section 3.2, 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 [39], 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 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 3.4.3, we optimize the multi-HMC by decoupling packet decoding and memory access with an always-on HMC.

Table 3: 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

3.4 Accelerator Design of NeuralHMC

3.4.1 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 [26] 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 [26]. Such representation reduces the original 32 bits floating points weights to 5 or 8 bits cluster index, saving much memory consumption.

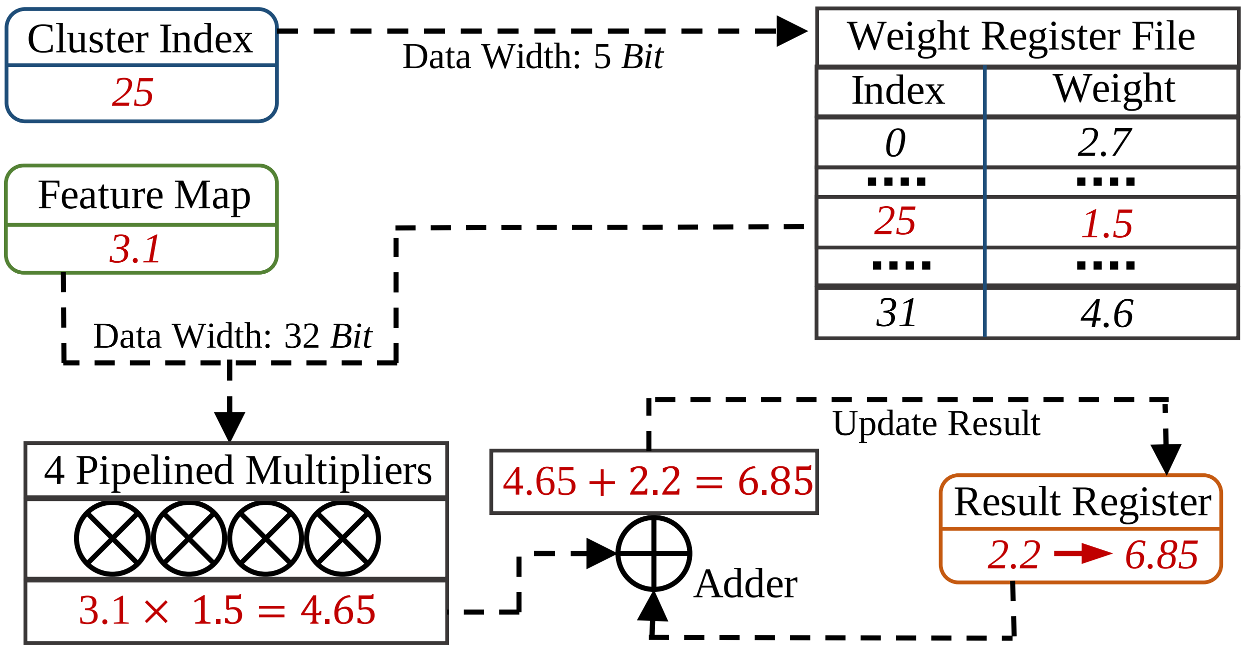


Figure 7: Weight sharing pipelined MAC design.

Figure 7 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 7, 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.

3.4.2 Asynchronous Packet Communication

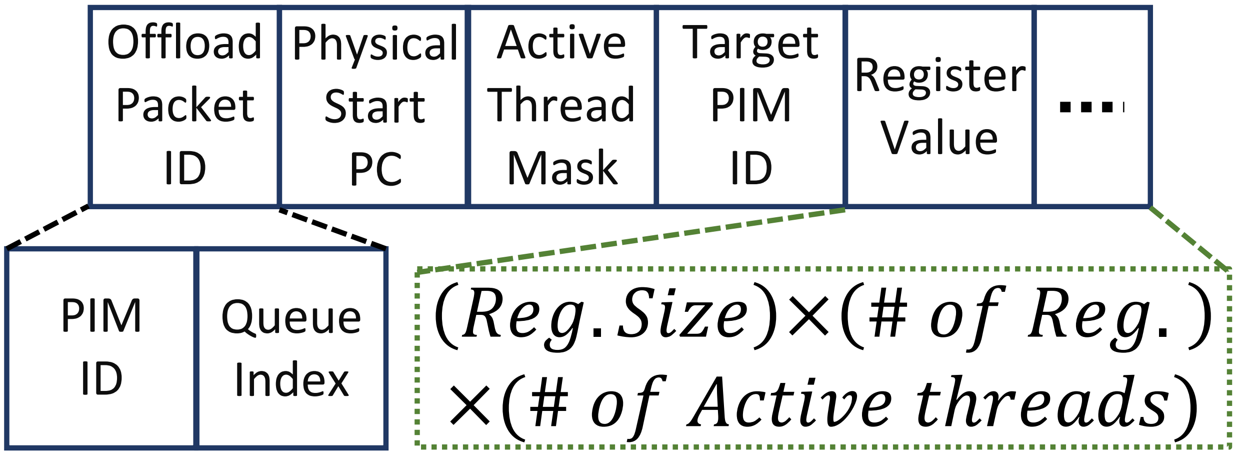


Figure 8: Flow control packet layout.

The HMC controller uses three types of packets: flow control, request, and response packets. The packet control layout is shown in Figure 8. The communication between the host and the PIM is asynchronous with the assistant of flow control packet. As also demonstrated in Figure 8, 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 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 9 illustrates the asynchronous communication between the host and the PIM. Calculations on the CPU and the PIM are performed simultaneously in Figure 9. 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.

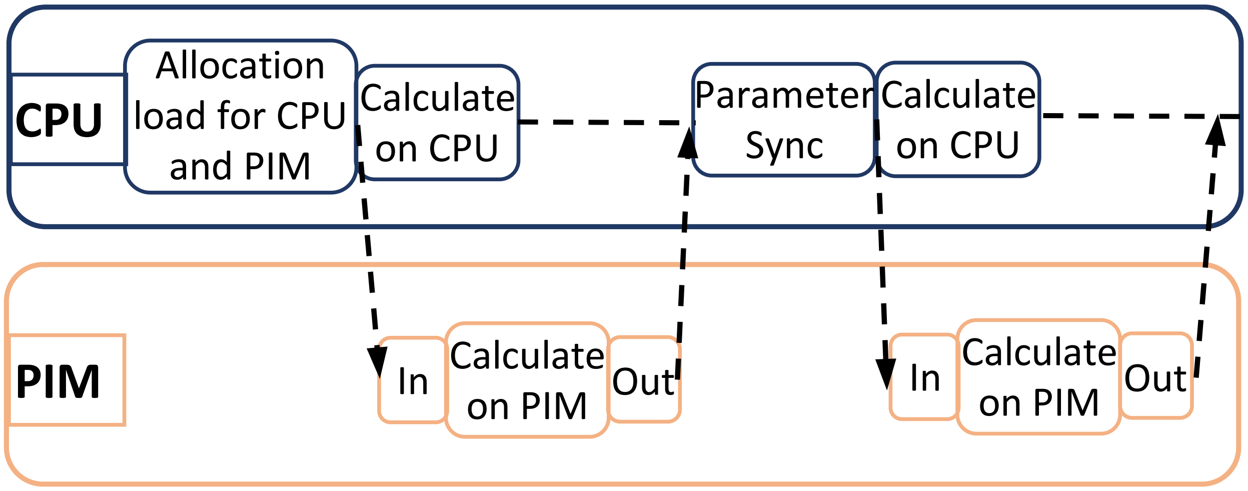


Figure 9: Asynchronous parameter communication between host and PIM.

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;
- A packet can be decoded before received completely in a pipeline manner: decoding the memory requests from the previous packet and receiving the next packet can be simultaneous.

Algorithm 2 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
```

3.4.3 Packet Scheduling Algorithm

In a traditional multi-HMC system, the memory power budget allows only two active HMCs at the same time [13]. 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 2 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 2) exceeds the starvation time (Starvation_Thd in Algorithm 2), 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 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.

3.5 Experiments

3.5.1 Experimental Setup

The power model adopted in *NeuralHMC* is based on [3], which can be summarized in Table 4. 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 [39], which

Table 4: Power Consumption

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

has been modified to adopt multi-HMC backend in HMC-Sim [51]. 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 [7] – a full system memory simulator, then loaded into HAC-MAC by the trace loader module. The activating/sleep latency is set to 2μs and the timing configuration of the simulator is shown in Table 5.

Table 5: 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

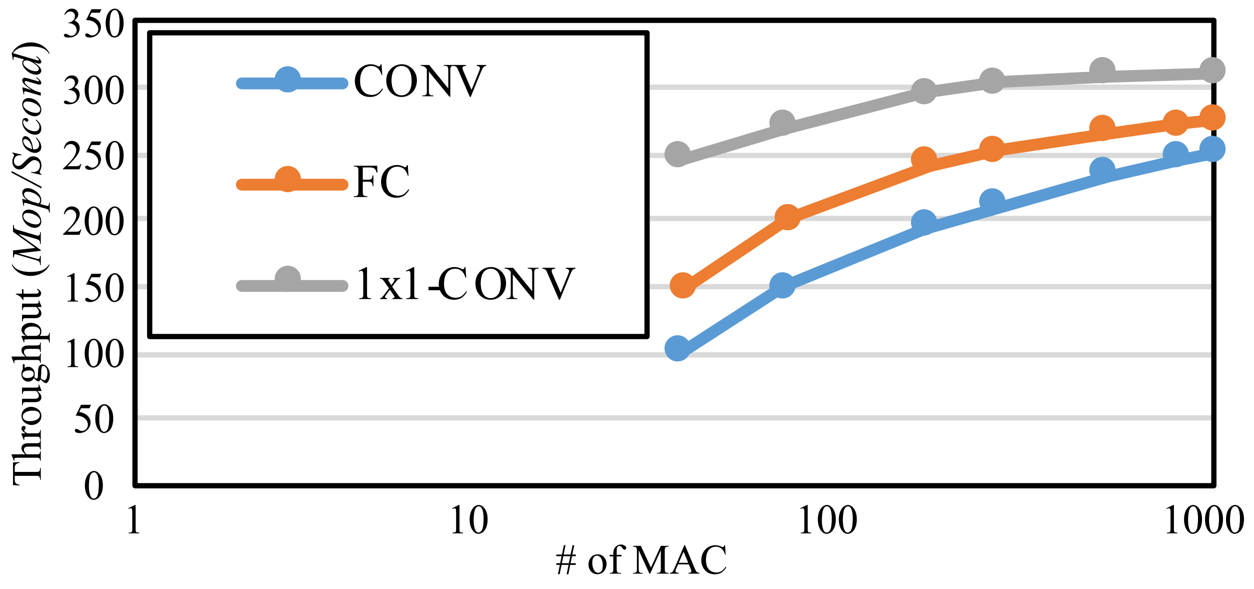


Figure 10: Throughput vs. MAC ops per feature map.

3.5.2 Evaluation of Single-HMC in NeuralHMC

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

Figure 10 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 11 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 operation. From

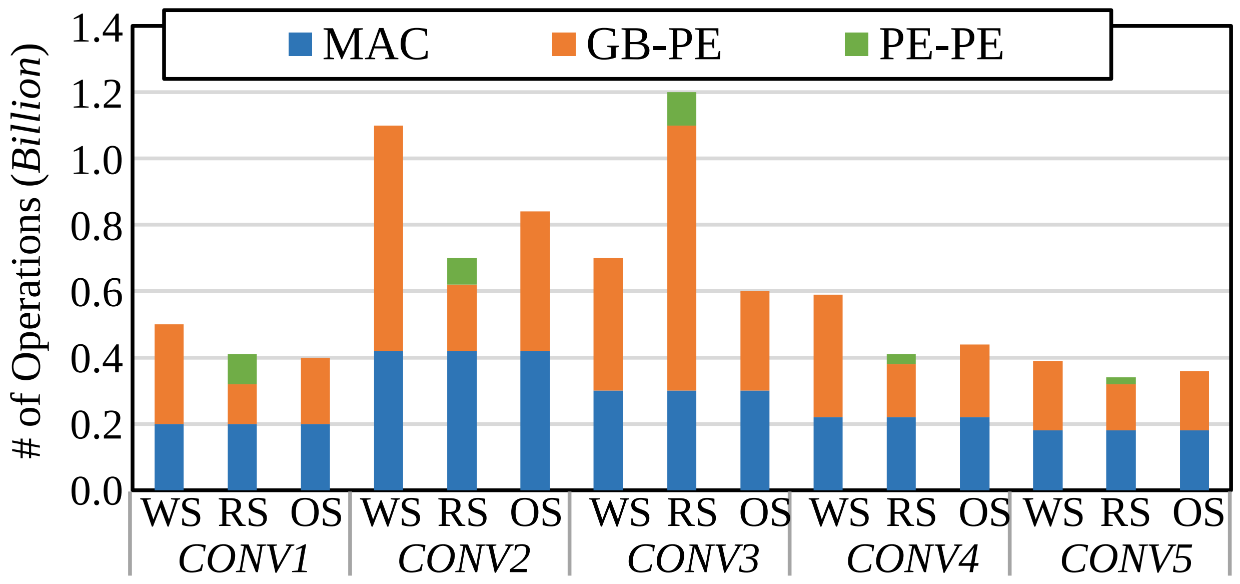


Figure 11: Operation breakdown in AlexNet.

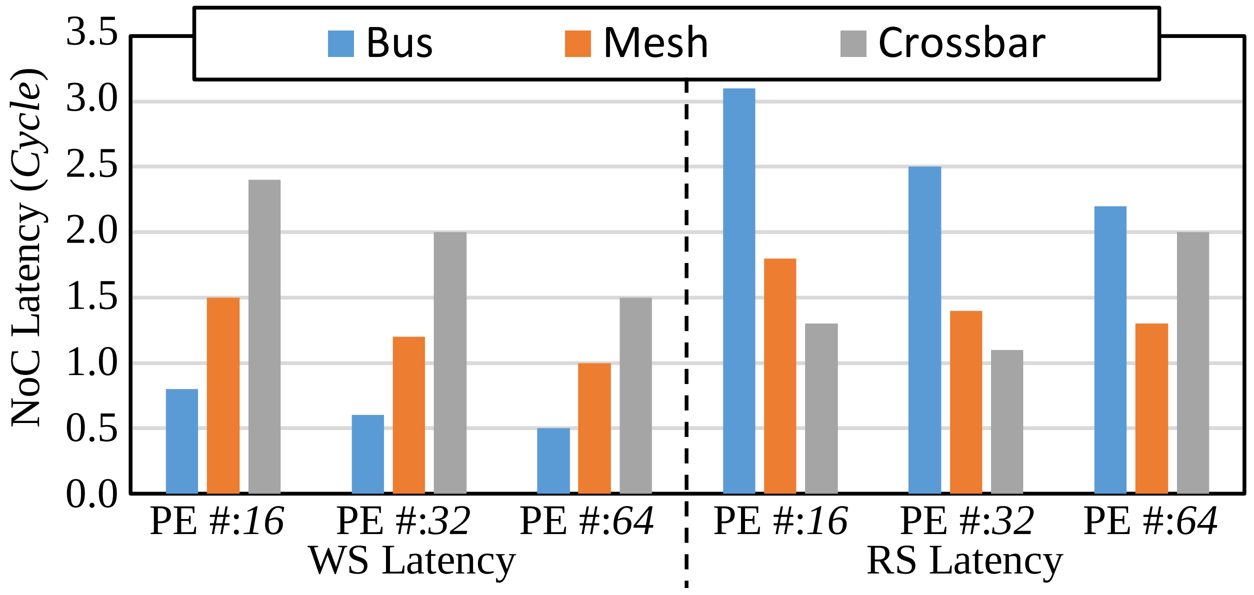


Figure 12: NoC latencies with RS and WS.

Figure 11, 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 12 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.

3.5.3 Evaluation of Multi-HMC in NeuralHMC

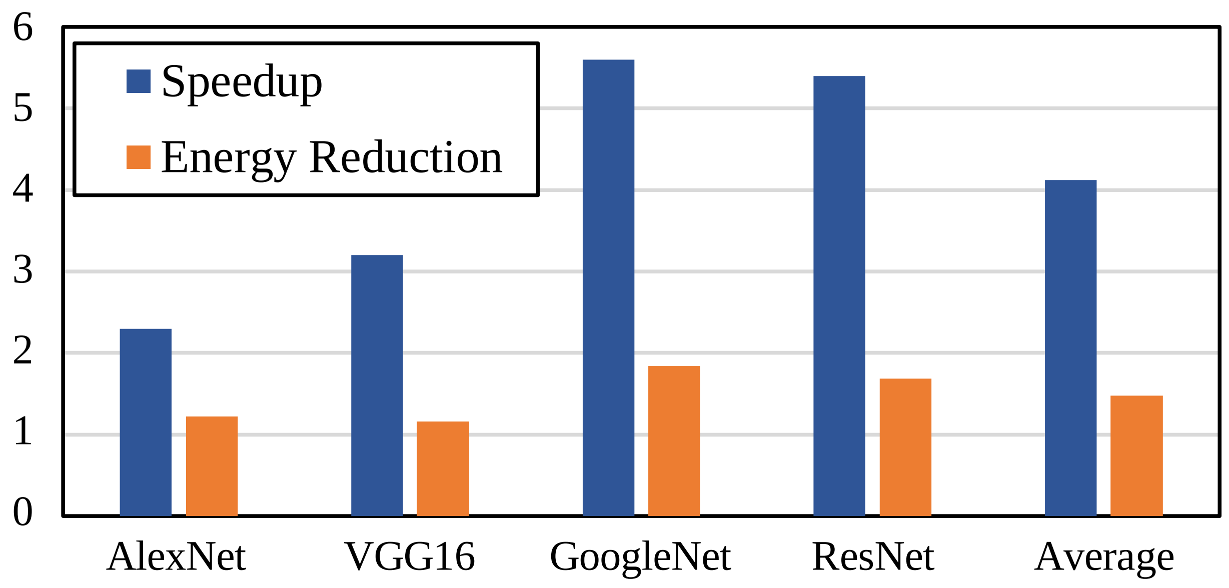


Figure 13: Total speedup and energy reduction.

Figure 13 shows the speedup and energy consumption of an 8-HMCs architecture. Our baseline is Eyeriss [9] – an spatial DNN accelerator with 128 processing elements. We use RS data reuse strategy 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$\times$ and energy consumption reduction is up to 1.85$\times$.

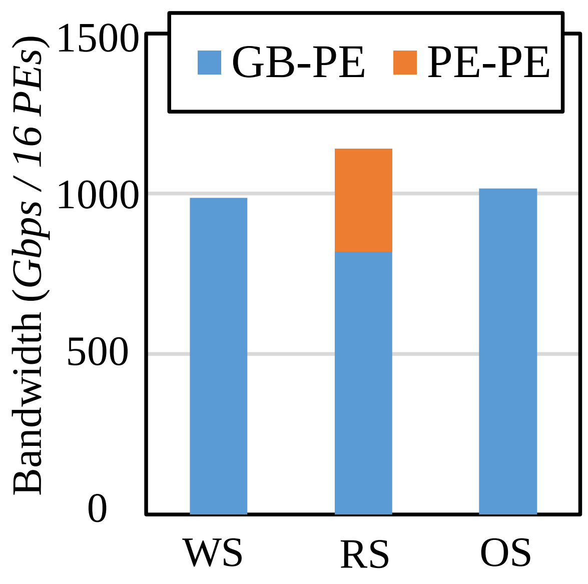


Figure 14: Inter-vault bandwidth breakdown.

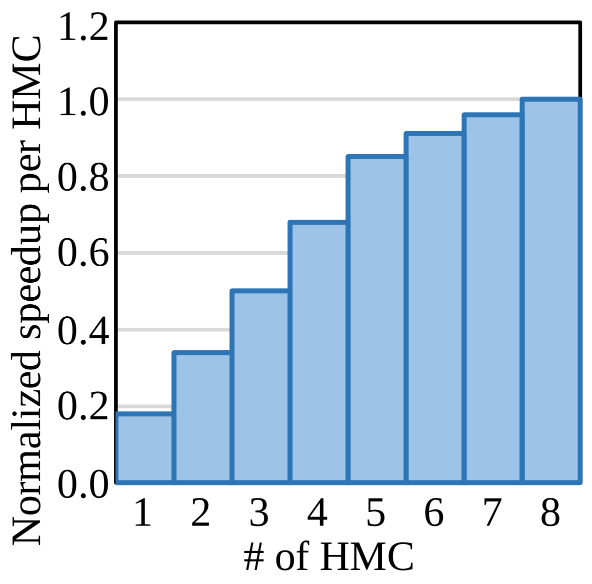


Figure 15: Performance of scalability.

In all benchmarks shown in Figure 13, 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 $4.1\times$ and an energy reduction of $1.5\times$.

We evaluate the inter-vault bandwidth of the three data reuse dataflow in ResNet-50 as shown in Figure 14. 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 15. As we can observe in Figure 15, 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.

3.6 Summary

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 $4.1\times$ in speedup and $1.5\times$ 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.

4.0 DSGAN: Acceleration of Generative Adversarial Networks (GAN) based on Dynamic Scheduling of Kernel Offloading

Wide applications of Generative Adversarial Network (GAN) have inspired great interest in acceleration of adversarial training. However, the asynchronization of loss calculation between adversarial phases limits training efficiency of GAN on traditional neural network accelerators. In this work, we propose a new accelerator design, namely, DSGAN that can dynamically schedule kernel offloading to process-in-memory modules to minimize the data movement and synchronization overheads in GAN training. Experimental results show that the DSGAN can achieve $1.6\times$ speedup compared to the state-of-art compiler-based acceleration techniques on multicore systems.

4.1 Introduction

Deep Neural networks (DNNs) is usually composed of multiple layers so that they can be trained to extract complex features from raw input data. DNNs, hence, have been widely utilized in many cognitive applications such as computer vision [37], speech recognition [32] and natural language processing [67]. Since adding more layers and more parameters has been proven as an efficient way to improve the accuracy of DNNs, executing the modern DNN models consumes a large amount of computing resources [14].

Since the operation of DNNs is primarily based on data movement in the network model, many DNN accelerators were proposed to optimize the memory accesses [8] and data reuse [9, 41] of DNN executions. In [8], for example, an on-chip scratchpad memory is introduced to support data reuse of local accelerators. In [41], many processing elements (PEs) are organized as a systolic array to allow temporal data reuse among the PEs and alleviate the reliance on memory bandwidth. Recently, Processing-in-Memory (PIM) structure was also adopted by DNN acceleration to eliminate the costly data movement between the pipeline and the memory. PRIME [10], for example, uses resistive memory to both store the data and perform the DNN executions on the data at local.

Another factor that greatly hinders the applications of DNNs is the availability of labeled training examples, which are needed to train the DNN for specific functions. Generative adversarial network (GAN) [18], therefore, is recently proposed to generate "fake" training examples through the coordination of two DNN models: *generator* and *discriminator*: Based on the feedback given by the discriminator, the generator is able to generate fake examples from a uniform noise distribution; Eventually, the generated fake examples cannot be distinguished by the discriminator and can be used to support unsupervised or semi-supervised learning [68, 79, 63].

Execution of GAN generates a memory usage pattern very different from that of conventional DNN: For example, training data fetching is one of the major contributors to memory traffic in a DNN accelerator [14]. In GAN, however, half of the inputs to the discriminator is generated by the generator at runtime, making data fetching very inefficient. As another example, the adversarial training procedure includes four different phases, each of which involves different pipeline stages or inputs and leads to various memory access patterns.

We note that utilizing PIM to accelerate GAN naturally solves the above challenges in memory access optimization as the execution is performed near the stored training examples and the generated examples. In this work, we propose DSGAN – a GAN accelerator based on dynamic scheduling of computation kernel offloading to PIM modules. DSGAN is implemented on Hybrid Memory Cube (HMC) – a recently proposed PIM architecture with enhanced capability of logic-die. Compared to the prior arts, our major contributions are:

- We analyze the training procedure of GAN and demonstrate that a uniform pipeline involves pipeline frequently stalls due to the asychrononization of loss calculation. In specific, feed forward propagation of discriminator is always $1.5\times$ network layer cycles ahead of generator before backpropagation can proceed. Based on our analysis, we proposed DSGAN to better support different computational phases of GAN by exploiting the limiting data dependency;
- We propose to offload computational kernels to PIM module with parsed annotations from compilation time and use a SRAM buffer to reduce data movement. We also propose to use an asynchronous training pattern to achieve blob level parallelism;
- We evaluate DSGAN on Sniper [31] with 32 cores CPU as host and 32 vaults on DCGAN in Caffe framework and compare it with a state-of-the-art compiler-based optimization [86]. Our experimental results show that DSGAN can achieve $1.6\times$ speedup.

The rest of this paper is organized as follows: Section 4.2 gives preliminary about GAN and introduces traditional CNN accelerator design and its limitations;Section 4.3 describes our proposed DSGAN architecture; Section 4.4 shows the experimental setup and the evaluation; Section 4.5 concludes this work.

4.2 Background

4.2.1 Generative Adversarial Networks (GANs)

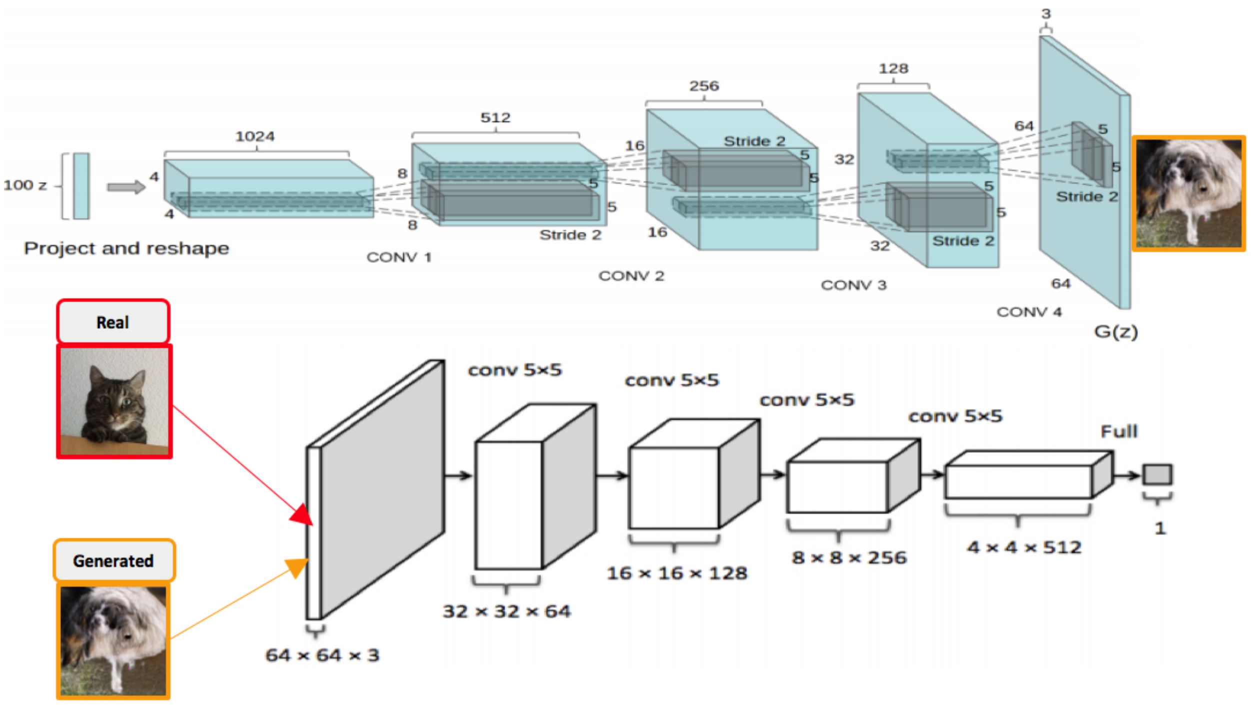


Figure 16: Architecture of DCGAN.

Generative Adversarial Networks (GANs) are a class of generative models that perform training process through coordination between a generator network G and a discriminator network D. As a basic variation of GAN, DCGAN [68] has been widely adopted in image processing applications [94]. Figure 16 depicts a typical DCGAN where the discriminator network D is a convolutional neural network (CNN) and the generator network G is a fractionally strided CNN. The D is trained to distinguish the samples generated by the *G* from the training data by taking the samples from the *G* as negative instances and the samples from the real data as positive instances. Here the *G* tries to "trick" the *D*by generating fake samples from a uniform noise distribution. Through a series of fractionally strided convolution layers, the 1D noise distribution is projected to a high dimensional latent space to produce a sample image.

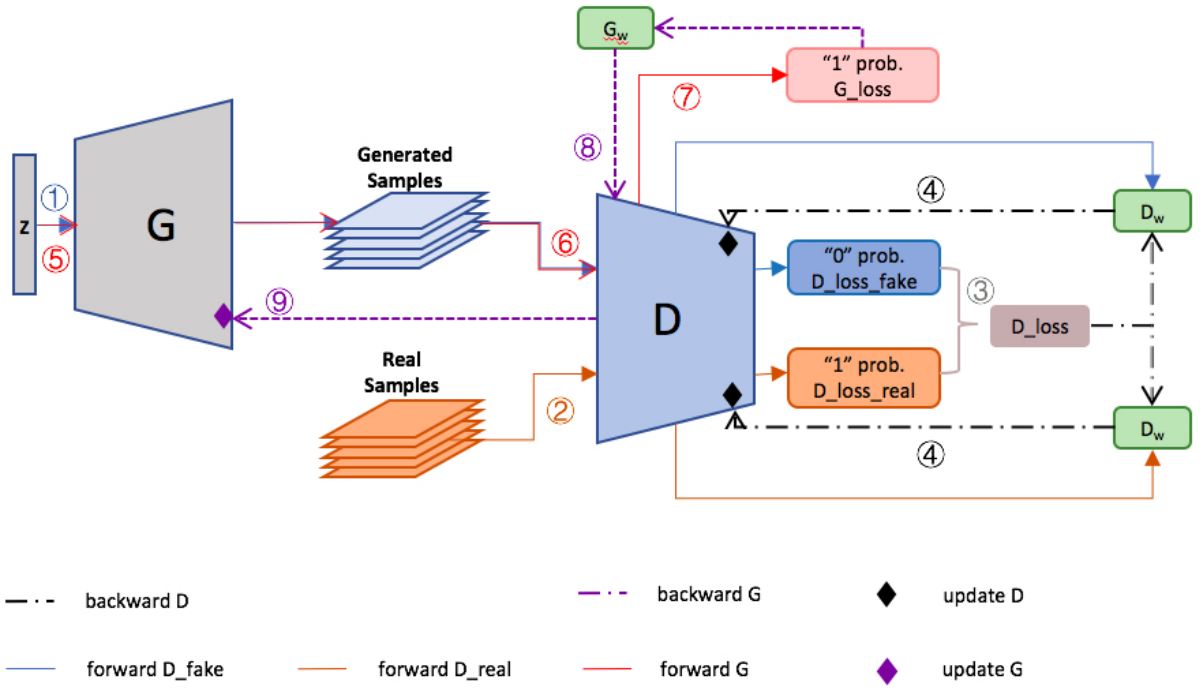


Figure 17: Training process of DCGAN.

The training process of DCGAN can be formulated as a $min - max$ game of a differentiable objective and solved greedily by iteratively performing gradient descent steps to improve the *G* and the *D* to reach a Nash equilibrium [18]. As shown in Figure 17, the training process of DCGAN can be divided into 2 phases and 9 stages:

1. Phase 1: During training of the *D*, the *G* only process forward pass and no backpropagation is applied;
2. Phase 2: In opposite, train the *G* and freeze the **D**.

We will discuss the details of training and its limitations on parallel computing in Section 4.3.1.

Algorithm 3 A transposed convolution on CPU.

```
for $i_c = 0 to I_C - 1$ do
    for $i_h = 0$ to $I_H - 1$ do
        for $i_w = 0$ to $I_W - 1$ do
            for $o_c = 0$ to $O_C - 1$ do
                for $k_h = 0$ to $K - 1$ do
                    for $k_w = 0$ to $K - 1$ do
                        $o_h \leftarrow S \times i_h + k_h - P$
                        $o_w \leftarrow S \times i_w + k_w - P$
                        $out[o_c][o_h][o_w] \leftarrow in[i_c][i_h][i_w] \times \ kernel[o_c][i_c][k_h][k_w]$
=0
```

4.2.2 Transposed Convolution

In DCGAN, fractionally strided convolution or so called transposed convolution is a upsampling procedure that adds zeros between each input in the feature maps with zero padding and computes the extended feature maps and the kernel.

It has been proven that CNN can be decomposed into multiplications and additions that can be accelerated using dedicated accelerators like [96]. However, transposed convolution arithmetic involves adding columns and rows of zeros to the input and incurs overlapping regions between the output blocks to be summed together. The overlap region scales with dimension of the input feature maps and creates a chessboard-like pattern. In practice, the input feature map is divided into blocks, and the PEs read each block from an off-chip memory and process the transposed convolution on each block. The computation results are then stored back to the off-chip memory. Considering the sparse structure of the input feature map and the overlapping summation, the involved memory accesses this procedure can be very inefficient. The pseudo code of a transposed convolution on CPU can be found at Algorithm 3. As we shall show in Section 4.3.5, we propose to perform partly loop unrolling and resize the tensor shape with the nearest interpolation to skip strided holes in the transposed convolution.

4.2.3 Kernel Offloading to PIM

Figure 5 shows a typical organization of PIM architecture. Memory is organized into vaults and each vault is controlled by a vault controller. The PIM cores are usually ISA-compatible with the processor cores but do not have large cache and dedicated ILP techniques for implementation simplicity. Performing computation in the memory can substantially eliminate memory traffic and improve the computation efficiency. The PIM architecture adopted in this work – Hybrid Memory Cube (HMC) in this work shares the similar architecture and also implements a weight buffer to reduce the weight access latency.

Due to the limited shared cache in PIM, application level offloading benefits only for irregular or random data access pattern, for instance, BFS [62]. Figure 18 demonstrate the tradeoffs in the offloading scheduling in a PIM architecture when different granularity and concurrency of kernel executions, which is often constrained by the data dependency between the tasks.

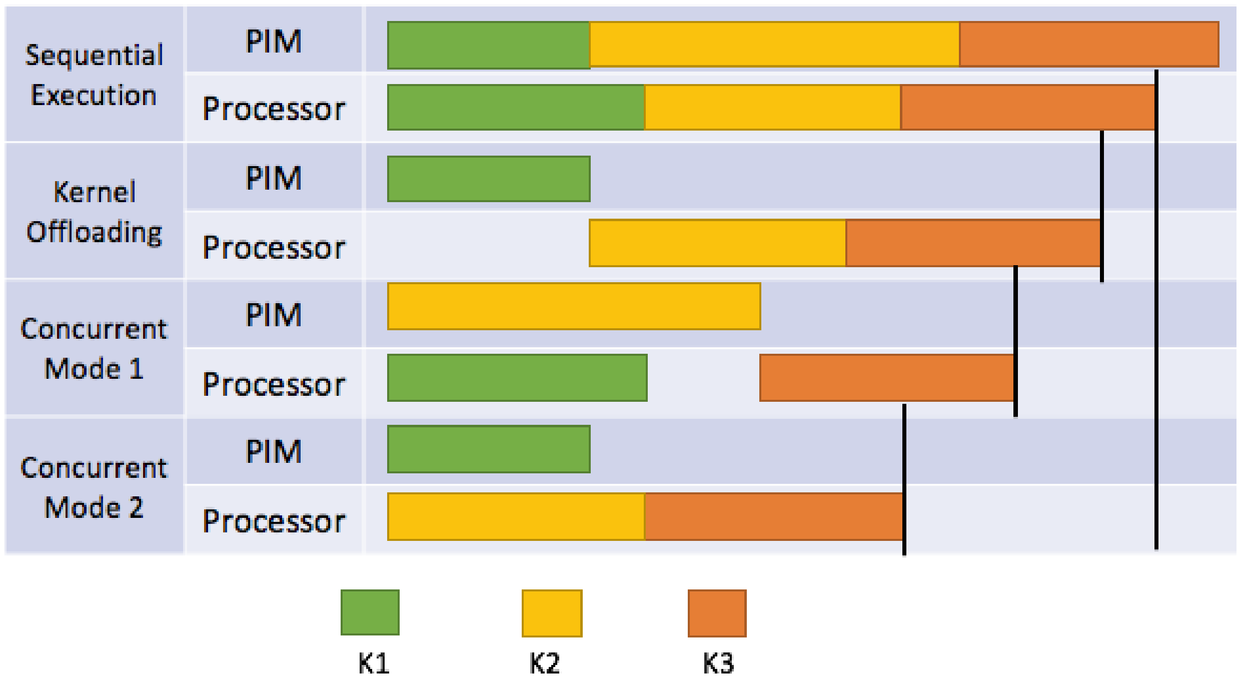


Figure 18: The influences of offloading granularity and concurrency of kernel executions.

4.3 DSGAN Architecture

In this section, we introduce our DSGAN architecture for GAN acceleration through the exploration on the limitation of computation parallelism.

4.3.1 GAN's Parallelism Limitation

As shown in Figure 17, the training process alternates between two phases until reaches convergence. The loss function *D_loss_fake* is first calculated based on a true label in stage ① which is '0' for generated images. The error and partial derivatives are computed and stored though backpropagation is deferred. Again, in stage ②, the loss function of discriminator *D_loss_real* is calculated based on the true label, which is '1' for real images this time. After the summation of *D_loss_real* and *D_loss_fake* (both are cross entropy loss functions) in ③, error and partial derivatives backpropagate and the propagation is constrained to *D* as in ④. In the training of *G*, which is shown as ⑤-⑨, the loss function is calculated based on a false label, which is '1' for the generated images. The error and partial derivatives backpropagate to update the weights in *G* while the weights in *D* is fixed.

The complex training procedure above introduces two limitations in GAN execution:

1. Limited parallelism that resulting from the synchronization requirement for the computations of *D_loss* (①-③); *G_loss*(①-⑤).

2. Increased memory requirement for the intermediate results (e.g. partial update and error) storage due to the deferred backpropagation.

Previous work [10] use a *Map_Topology ()* function to map the topology of the neural network to full function (FF) arrays. However, the mapping process is constrained by *directed acyclic computation graph* and can not achieve a uniform pipeline when taking the discriminator and generator as a whole network. If discriminator and generator are mapped to FF arrays separately, then the training procedure involve two dependent pipelines which stall from stage ② to stage ⑥.

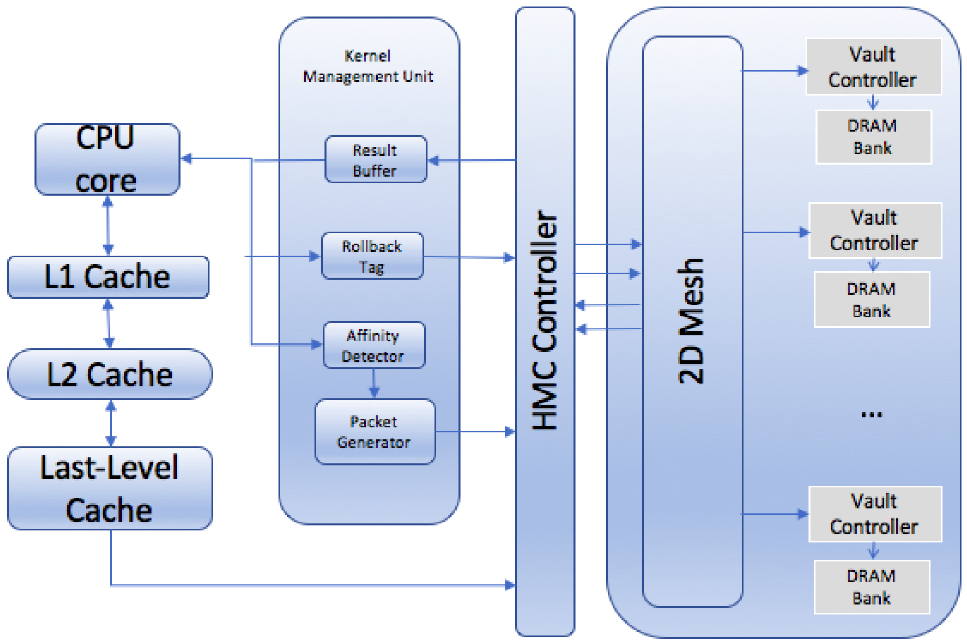


Figure 19: DSGAN overview.

4.3.2 Overview of DSGAN

Figure 19 gives an overview of our architecture. The baseline memory technology is Hybrid Memory Cube (HMC) [66].

An affinity detector identifies kernels with high data locality and executes them in the corresponding vault. We decide the memory vault that an kernel has affinity with by using the following equation:

$$affinity = \frac{kernel\_id}{N_{threads\_per\_vault}} \bmod N_{vaults} \tag{4.1}$$

$N_{threads\_per\_vault}$ is the number of threads that can run concurrently in one vault. For example, if one vault has four PEs and each of which can run six threads, $N_{threads\_per\_vault}$ is 24. When N is the number of memory vaults and T is the total number of kernels, T/N kernel have the same affinity. With the affinity information, whenever an PE is available, instead of assigning any unscheduled kernel to it, the scheduler picks one that has affinity to that memory vault. On the other hand, where the data should be located can also be computed, as the affinity-based scheduling algorithm already determines where the computation will be performed. The equations to compute $chunk\_size$ and $vault\_id$ are as follows:

$$chunk\_size = min(4KB, B \times N_{blocks\_per\_vault}) \tag{4.2}$$

$$vault\_id = \frac{(virtual\_addr - virtual\_start\_addr)}{(chunk\_size)} \bmod N_{vaults} \tag{4.3}$$

The chunk size is bounded by 4KB, when the result is not a multiple of page size, misaligned pages will be allocated in the next consecutive memory vault. In this scenario, whether to localize or distribute each memory object is based on its anticipated access pattern and the affinity-based scheduling algorithm guides kernel to the memory stack where the data they access is located to reduce PIM-to-PIM communications.

The computation overhead of affinity detector depends on the kernel size. Traditionally, PIM targets execution of medium sized computation kernels having less than a few kilobytes of instructions. To reduce the affinity detector overhead, we further partition kernel into subcomputation segment and offload these ”sub-kernels” instead. To accommodate the abundant the variable computing phases in GAN, we deploy a weight buffer in HMC logic die and utilize SRAM buffer in eDRAM to store intermediate results in each pass.

To further parallelize kernel level execution, a kernel level data dependence data graph is required to decide which kernel can execute in parallel. Since most existing compilers targeting array-based applications already run dependence analysis for loop kernel, what we concern here is actually across kernel dependency.

4.3.3 Inter-Kernel Dependency

Despite the compile time awareness of intra-kernel dependency, inter-kernel dependency is examined to maintains the logical behavior. We use a state machine to speculate inter-kernel dependency. At the PIM kernel completes, the PIM core transmits a compressed signatures with all addresses that the PIM core reads or writes to host processor. If a conflict is detected, a roll-back signal is sent to corresponding PIM core. Otherwise, the read/write operation is committed. Thanks to the sub-kernel partitioning, we may perform a commit after each sub-kernel completes.

First, we lower the probability of conflicts as the transmitted signal contains fewer read and write operations, which reduces the detection window. Second, the overhead of rollback operation is alleviated, because the PIM kernel only need to rollback to the conflict sub-kernel checkpoint instead of the begin of the kernel. Third, the signal size shrinks as less addresses are kept.

4.3.4 Kernel Scheduling Algorithm

The kernel request is queued to PEs in Algorithm 4. The host processor is responsible for generating different kernels to initiate the offloaded execution on the PIM while transferring register data and generating memory requests on behalf of the PIM. When *OFLD.BEG* instruction is executed on an affinity detector at the beginning of an kernel, the affinity detector generates an offload command packet with all information needed for initiation of offloaded execution on the PIM, including the register values that need to be transferred.

An asynchronous method is applied in host to PIM level communication. We aim to reduce data transferring cost between CPU and PIM and allow them to calculate independently and simultaneously. One thread is created on the CPU and the signal mechanism is support by PIM device driver. After allocating load for CPU and PIM, the host will set the calculation signal. Calculation on CPU and PIM will be carried out simultaneously and the data transmission between CPU and PIM is hidden. After training of every n batches, the PIM will set the synchronization signal and parameters will be synchronized between host and PIM.

Algorithm 4 Dynamic kernel management.

```
p,q denotes the candidate PEs, h is the affinity offload queue
if p = Idle $\wedge$ p.queue $\neq 0 \wedge$ h.queue $\neq 0$ then
    partition kernel into $sub\_kernels$
    for each sub_kernel in $sub\_kernels(h.queue)$ do
        if $\sum time(sub\_kernel, h) + executed(sub\_kernel, p) > \sum time(sub\_kernel, p)$
then
            offload kernel to p; h.dequeue
        else offload kernel to q; h.dequeue
```

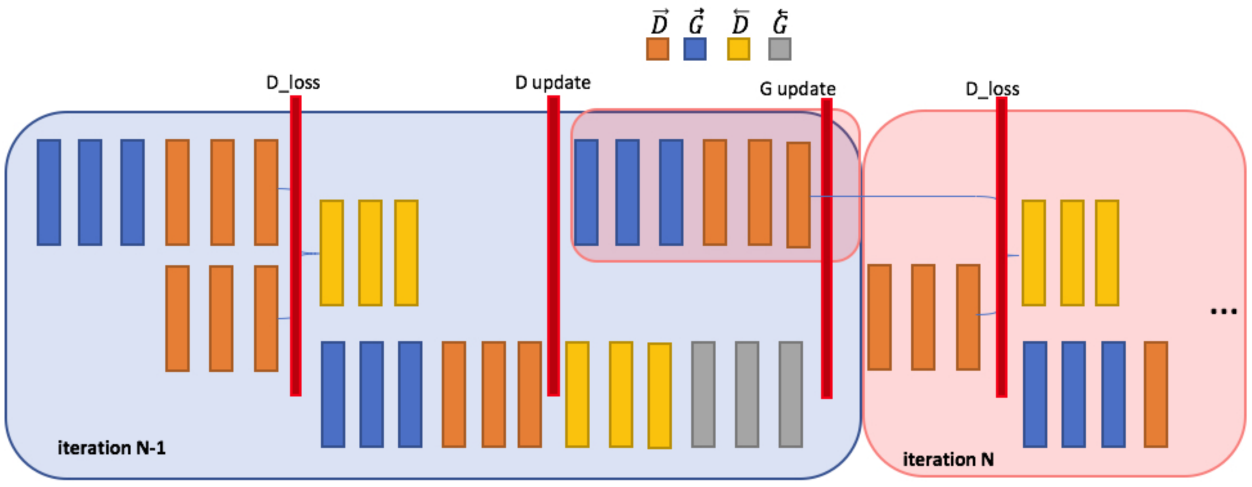


Figure 20: Asynchronous pipeline flow for parallelism.

To achieve more parallelism, an asynchronous pipeline is proposed as shown in Figure 20. The red block in the figure denotes the data dependency between discriminator and generator. In this scenario, *D_loss_fake* is one iteration behind the original DCGAN but can parallel pipeline by number of layers per iteration.

4.3.5 Transposed Convolution Optimization

To avoid the overlapping summation problem described in Sec 4.2.2, we first take a block in the output space and determine which inputs are needed to calculate the values in the block. Then, for each block, the input is fractionally strided convoluted and the appropriate output is extracted. This is done sequentially until values have been computed for the entire output space. However, when the kernel size is not divided by the stride, the input is not traversed sufficiently. An approach to overcome this is to separate out upsampling to a higher resolution to compute feature [64]. We recast output along one dimensional into two variables with nearest-neighbor (NN) interpolation, then do a convolution to compute features. In this way, instead of using the output space to determine which input blocks to upsampling and thus eliminating the need for the additional summation operations.

Based on the time prediction model in [65], we build our own sub-kernel execution time model. For inter-kernel dependency, we utilize GAN layer wise execution to profile the dependency graph which involves typical training procedures: (1) CNN forward propagation; (2) Transposed CNN forward propagation; (3) CNN backward propagation; (4) Transposed CNN backward propagation.

4.4 Experimental Setup and Evaluation

Experiments in this section are performed on a Sniper [31] multicore simulator to model the processor simulating the same affinity detector process. We modified Sniper to support fine-grained (cycle-by-cycle) multithreading for the HMC cores and TLB management. We then use the gem5 [7] architectural simulator in full-system mode, using the x86 ISA to implement proposed kernel offloading mechanism. To emulate the high in-memory bandwidth within HMC for the PIM cores, the DRAM model is modified in gem5. DRAMSim2 [72] is applied to model the memory timing. Below is the system configuration for simulation. We examine how the DSGAN compares with prior compiler optimized approach [86]. We show results of kernel commit and sub-kernel commit normalized to processor only baseline in Figure 22. The system configuration is summarized in Table 6.

4.4.1 Execution Time Model

We use DCGAN workloads with different number of layers from medium to large scale with several hundreds of MBytes memory footprints to evaluate DSGAN. The neural network model obtained from the training phase is sent to Caffe [40], with notations and variables added for tracking the timing and other parameters of each layer. A typical five layer DCGAN and its classify bin of execution time is summarized in Table 7. According to [4], a software-stack has been developed for application to view PIM as a standard accelerator with a dynamic binary offloading mechanism designed to detect the code at run-time. The PIM device driver provides a low overhead and high-performance communication mechanism between parallel programming APIs and PIM.

Table 6: System configuration.

Component	Configuration
`Core`	16 tiles (each tile 2 cores)
`L1 I/D-Cache`	Private, 64KB, 4/8-way, 64B blocks
`L2 Cache`	Private, 1MB, 8-way, 64B blocks
`L3 Cache`	Shared, 64MB, 16-way, 64B blocks
`TLB`	32 entries, 2 MB page, 200-cycle miss penalty
`Main Memory`	32GB, 8HMCs
`HMC`	8GB, 32 vaults, 512 DRAM banks
`Timing Parameters`	t$_{CK} = 1.6ns, t_{RAS} = 22.4ns$
	t$_{RCD} = t_{CAS} = t_{RP} = 11.2ns, t_{WR} = 14.4ns$
`weight buffer`	64KB

Table 7: Classification of predicted execution time.

Layer	Weight Shape	Output Shape	Cycles	Classification Bins
D conv_1	$5^2x256x512$	$4x4x512$	$819,200$	3
D conv_2	$5^2x128x256$	$8x8x256$	$819,200$	3
D conv_3	$5^2x128x256$	$8x8x256$	$819,200$	3
D conv_4	5^2x3x64	$32x32x64$	$76,800$	2
G dconv_1	$5^2x128x3$	$64x64x3$	$76,800$	2
G dconv_2	$5^2x256x128$	$32x32x128$	$13,107,200$	4
G dconv_3	$5^2x512x256$	$16x16x256$	$13,107,200$	4
G dconv_4	$5^2x1024x512$	$8x8x512$	$13,107,200$	4

In kernel offloading, the host processor parses the binary computation kernel and dynamically offloads *.text* and *.rodata* sections to PIM's memory map. In [65], kernel affinity is predicted via a regression model using three categories of metrics before it starts execution. In our work, the performance the evaluated by the kernel efficiency, which we define as the the fraction of peak theoretical FLOP can be achieved by pipeline and data reuse. The baseline is a CPU-only solution. We also evaluate two different NDP solutions: one is a compiler based optimization [86] on state-of-the-art many core system with 36 tiles of Intel Knights's Landing (KNL) [78] interconnected 2D mesh NoC, another is the 2D baselines using a 16×16 PE array with a 1024byte register file per PE and a 576kB global buffer.

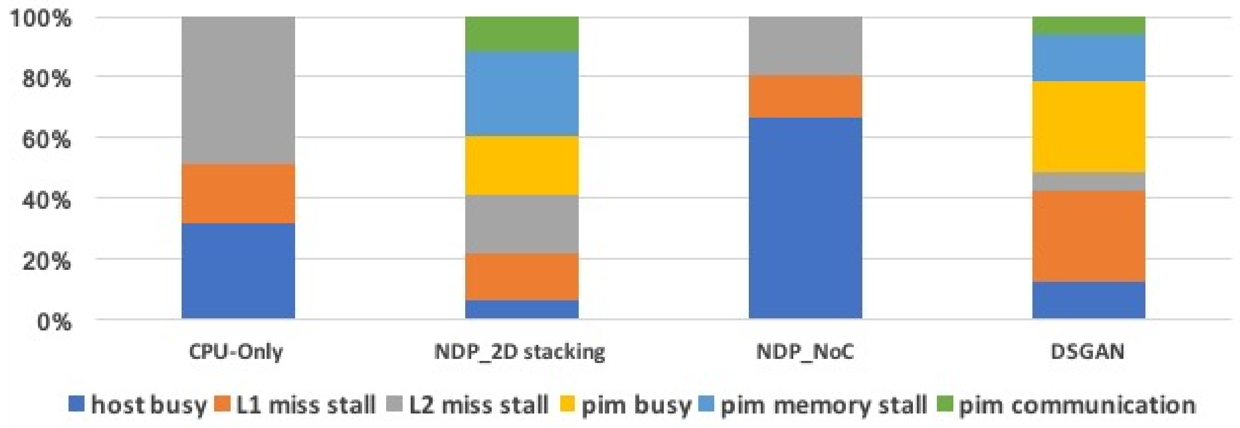


Figure 21: Execution time breakdown.

Figure 21 illustrates simulation results for this application. The baseline sequential execution is classified into several components. The host busy category accounts for the time spent executing instructions. The L1 and L2 miss stall categories represent time spent waiting for memory accesses to be satisfied from either the L2 cache or main memory. Note that additional coherence overhead is charged as L1 and L2 cache misses in the host when PIMs are used; by flushing data from the cache prior to PIM computations, extra cache misses in the host may occur in later host computation. This cache miss effect due to flushing is not significant in the programs presented here because the irregular accesses in the PIM computations were polluting the host cache when executed on the host. Additional categories show time spent in the PIMs, including PIM-to-PIM communication overhead and time spent in local memory stalls. As the results in figure indicate, the original application suffers significantly from poor cache locality in CPU mode with overall L1 and L2 cache miss rates of up to 30% and 50%, while by adopting NDP method, the cache miss rate is significantly reduced with a average of 15% in 2D memory stacking PIM and even lower with DSGAN with weight buffer.

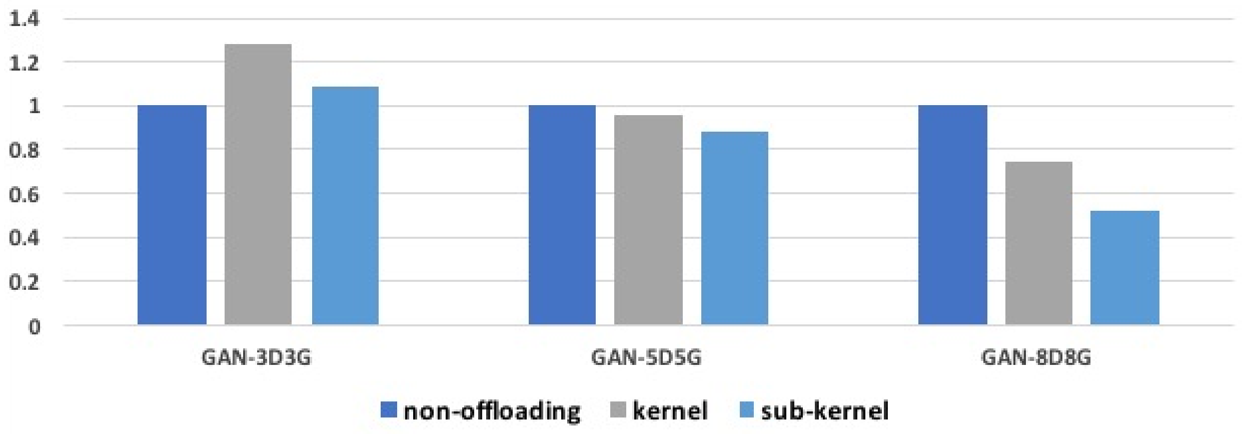


Figure 22: Speedup of kernel granularity.

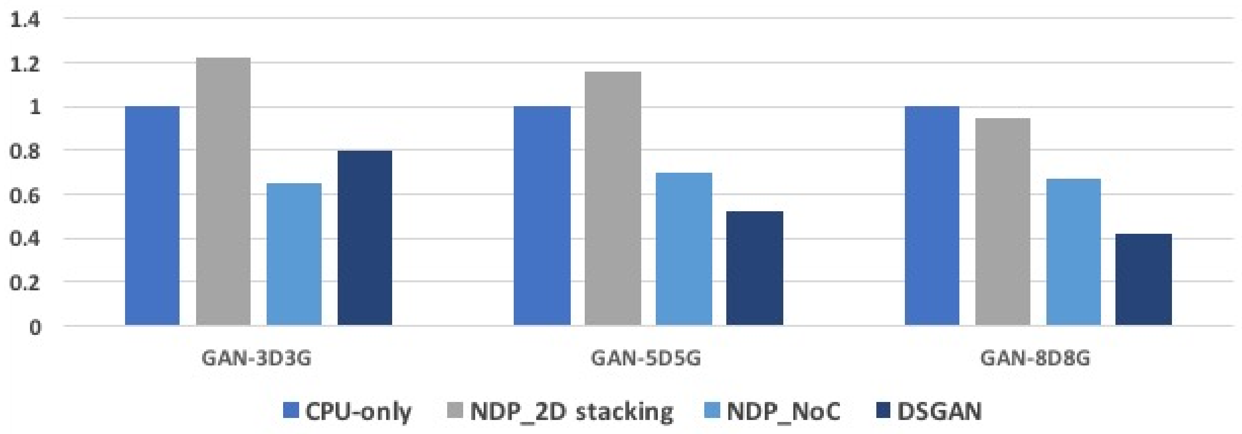


Figure 23: Concurrent kernel computation performance.

Figure 22 shows the offloading granularity issue. The longer the pipeline, the further the distance intermediate results are reused. For DSGAN, the computation of per batch per feature map convolution is atomically scheduled and intermediate results is stored on the same memory vault instead of a traditional bank interleaved distribution. Figure 23 shows the kernel ratio utilization for a 32-core architecture (with 32 processor cores and 32 PIM cores) across different phases of training process of GAN. Before the training of *G* started (annotated in compiler), the host processor inserts a packet signal to check for the commit of cross entropy loss and make the weights in SRAM invalid. As we can see from the simulation results, the DSGAN can achieve of 1.6$\times$ speedup compared to NoC with data movement optimization on a 16-layer DCGAN.

4.5 Summary

In this work, we utilize affinity-based kernel offloading to accelerate GAN, in which pipeline is deferred by loss synchronization. By adopting co-location of computation and data in HMC memory stacks, the intermediate results including partial gradient and error, are stored in SRAM buffer of the same vault where the computation takes place. In this way, despite the long data reuse distance, gradient synchronization in backpropagation does not incur long memory access latency. Moreover, we refined the training to be asynchronous for one iteration but achieve more parallelism than the original GAN. Our simulation result demonstrates 1.6$\times$ speedup compared with the compiler based Nearest-Data-Processing (NDP) system.

5.0 Extending the Lifetime of Object-based NAND Flash Device with STT-RAM/DRAM Hybrid Buffer

NAND flash memory has achieved remarkable success in modern storage systems for its advantages in cost, capacity and non-volatility. However, its potential is not brought into full play due to its architectural limitation in block-based storage system. A major limitation of NAND flash memory is erase-before-program characteristics. It incurs write amplification, severely degrading system performance and endurance. An object-based NAND flash storage system is proposed to overcome the limit by offloading the storage management layer to device. Unlike block-based storage device, object-based NAND flash device (ONFD) stores user data and metadata in different chucks. Previous works reveal that metadata update substantially contributes to write amplification in object-based NAND flash device (ONFD). Besides, write amplification severely degrades the performance and endurance of ONFD by causing redundant page write and garbage collection (GC). Also, in object-based NAND flash device (ONFD), power failure may cause inconsistency between object data and object metadata and impair the system reliability.

In this work, we propose a design to alleviate write amplification and handle power failure in ONFD. By adopting non-volatile STT-RAM as a complement buffer to existing on-device DRAM buffer, redundant page writes of per-object indices are minimized and data recovery mechanism can be implemented with acceptable circuit overhead. To further reduce the overhead of metadata update in ONFD, we propose a hybrid buffer scheme (HBS) by utilizing the lower latency and byte-addressable characteristics of the promising emerging non-volatile memory STT-RAM. Our HBS proposes to store ONFD metadata with highest cost in a complement STT-RAM buffer to reduce write amplification. Considering limited size of STT-RAM, we propose a hybrid buffer management technique to maximize effective memory utilization. In addition, by leveraging non-volatility of STT-RAM, our HBS can also substantially reduce data recovery overhead and complexity upon power failure. Experiment results show that the proposed design can achieve up to 15% performance improvement with average 34% endurance extension compared to the state-of-the-art works.

5.1 Introduction

Thanks to high density, low power consumption and good scalability, NAND flash memory has been widely adopted as μs-class storage media in various storage systems, ranging from consumer electronic devices to high-end servers. A NAND flash cell is a floating gate transistor with programmable threshold voltage. The stored values are represented by different voltage levels. In NAND flash memory, data write is realized by program operation. Before re-programming, erase operation has to be applied to NAND flash memory to remove the stored data. Besides long latency, the program/erase (P/E) operation wears out the NAND flash cells, eventually leading to cell failure. Program operation is performed in unit of page while erase operation is applied in unit of block. Due to these inconsistent operation units and erase-before-program characteristics, valid pages of a dirty block needs to migrate to clean blocks before the dirty one is erased during garbage collection. To emulate the logical disk, handle the out-of-place update, and prolong the system lifetime under limited memory endurance, a flash translation layer (FTL) is implemented in NAND flash based storage system. FTL maintains the mapping information between the logic and physical address spaces. To accommodate the out-of-place update feature of flash cells, FTL also carries out the garbage collection which recycles the invalid pages for the upcoming write requests. The adoption of block-based interface constrains the optimization of device performance. During garbage collection, valid pages of the dirty blocks migrates to clean blocks before these blocks are erased. As a result, a page write request from the file system usually incurs two ore more writes to the NAND flash memory and more data is written to the NAND flash memory than requested. The scenario is denoted as *write amplification*. Write amplification causes severe system performance and endurance degradation. In addition, the block-based interface also results in the functionality overlap in the file system layer and FTL as shown in Figure 24. The former maintains the mapping relation between the file data and LBA, while the latter contains the mapping table of LBA and the physical addresses. The extra mapping table incurs significant storage overhead [98].

Unfortunately, the traditional block-based storage model cannot efficiently reduce write amplification. Previous work [88] reveals that write amplification can be reduced by isolating hot and cold data to different blocks. However, the traditional block-based storage model only passes logic block address to the NAND flash storage device, which cannot accurately identify hot data [42].

To effectively mitigate the write amplification, a refactored design of I/O architecture, object-based NAND flash storage model has been proposed [56].

In the object-based NAND flash storage model, data is stored in unit of object instead of logic block. The high-level semantics of objects, such as object size and type, are delivered to object-based NAND flash device (ONFD). Hot data can be identified with high-level semantics more accurately. As a result, the efficiency of garbage collection is improved and write overhead is reduced [42]. Despite architectural optimization and substantial performance improvement, there still exists exploration space to reduce write amplification in ONFD.

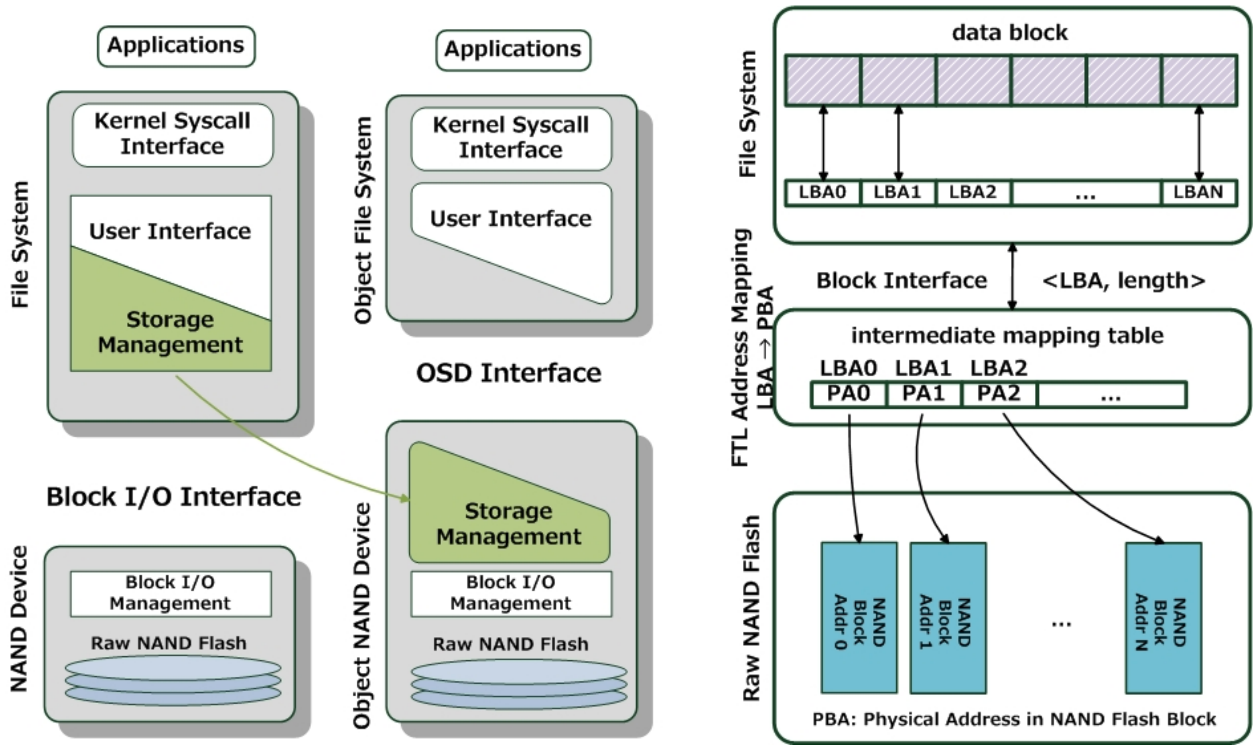


Figure 24: Comparision of block-based and object-based storage models

As shown in [20], one major cause of write amplification is byte-level metadata update. Due to inconsistent unit of NAND flash program operation and metadata update, partial page updates can invoke up to 20× of write amplification. By separating data from metadata, optimization based on data type and their access patterns can be explored. The issue of operation inconsistency can be addressed by the emerging memory technology in recent decades, such as spin-transfer torque memory (STT-RAM). Compared with NAND flash memory, STT-RAM has the advantage of nearly unlimited endurance, ns-level access latency and byte-addressability [53]. By leveraging the byte-addressable characteristics of STT-RAM, we proposed a STT/DRAM hybrid buffer scheme (HBS) to further reduce write amplification in ONFD in this work. The main contribution of this chapter is summarized as follows:

- We proposed to differentiate ONFD metadata types and store the metadata with highest performance cost in STT-RAM to minimize write amplification of NAND flash memory.
- Due to limited size of STT-RAM, we proposed a buffer management technique to maximize the memory utilization and extend endurance by minimizing data migration during power failure.
- We also implemented a module in ObjNandSim to mimic the behavior of ONFD with hybrid buffer.

We employed various types of real-world workloads to evaluate efficiency of our proposed HBS. The experimental results show that compared with DRAM-only buffer scheme, our proposed HBS scheme can achieve about 34% endurance improvement.

The following of the chapter is organized as follows: Section 5.2 introduces the background knowledge; Section 5.3 presents the motivation of our work; Section 5.4 describes details of the proposed design; Section 5.5 introduces the simulation platform and presents the simulation results; Section 5.6 concludes the work.

5.2 Background

5.2.1 Object-based NAND Flash Device (ONFD)

The object-based NAND flash storage model [42] is shown in Figure 25. In this model, *object* serves as the basic storage unit. An *object*, uniquely identified by an 64-bit object ID, denotes a variable-sized data container, e.g., a file in this chapter. Each object includes *data* and *attributes*. The object data contains variable-length file data. The object attributes include file metadata such as ownership and access control. The object-based storage model includes an object file system and an object-based NAND flash device (ONFD). The object file system only maintains the name space. The ONFD manages the storage of objects.

In the ONFD, the object data is accessed via object metadata, which includes both per-object indices and onodes. Like the object data, the per-object indices and the onodes are also stored in the NAND flash memories. The per-object index, with a tree structure, maintains the physical addresses of the object data. The onode contains the object attributes and the address of the per-object index root node page. The object attributes have different length and therefore, the onode has a variable length too. For example, in the file system, every file has a fix-sized inode structure to store the file attributes. Some objects have extended attributes that need additional storage space. The physical addresses of the onodes are maintained in a global index. The global index, which is arranged with a B+ tree data structure, resides in the DRAM for fast access.

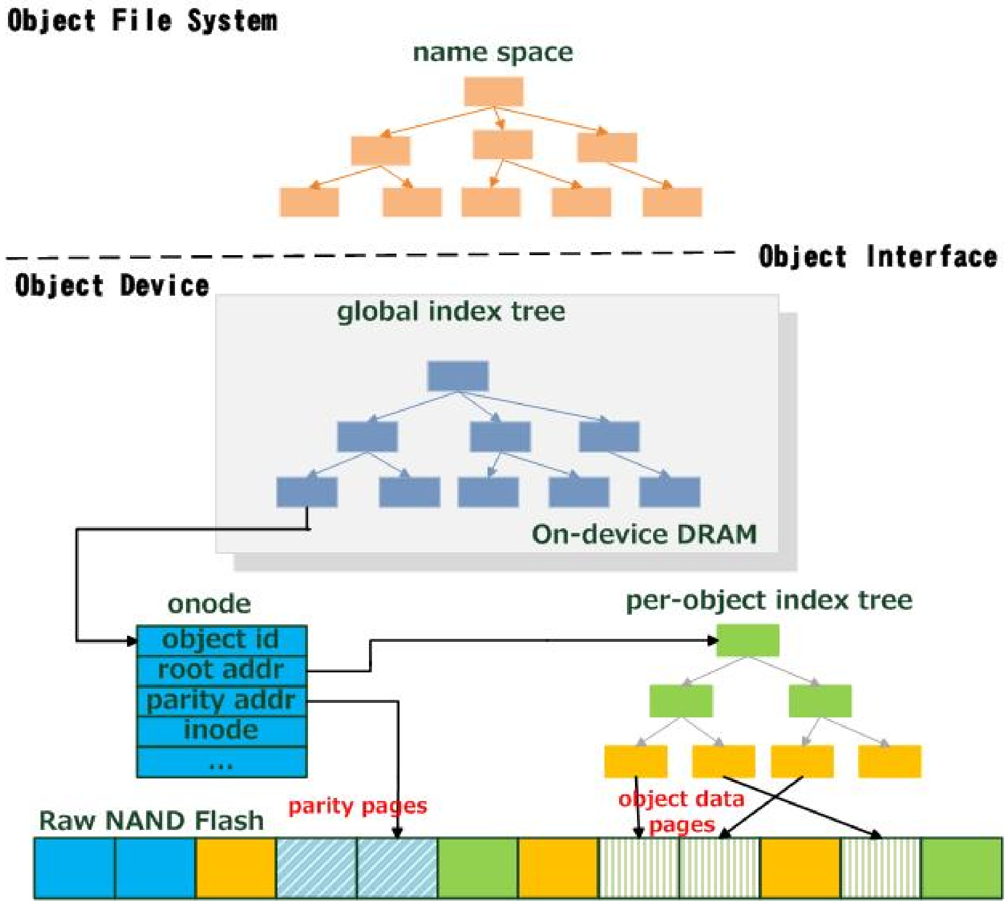


Figure 25: The architecture of object-based storage system.

5.2.2 STT-RAM Characteristics

STT-RAM is one of most promising byte addressable non-volatile memories. With the advantages of non-volatility and density, the STT-RAM is emerging as an alternative of DRAM. In addition, without any leakage current for data retention, STT-RAM can eliminate significant refresh power and reduce the system power consumption. In addition, STT-RAM has the access latency within the same magnitude of DRAM and much faster than NAND flash memory. However, the write performance of STT-RAM is degraded because the current required for writing into an MTJ is notably higher than that needed for reading from it. By integrating DRAM with STT-RAM, we can take advantage of both RAMs to 1) easily handle the data recovery during power failure, 2) avoid unnecessary writes to STT-RAM by adopting lazy update policy, 3) reduce the area budget with the dense STT-RAM cells.

5.2.3 Related Works

To reduce partial page update and cascading update induced data migration, a broad class of solutions are proposed for architectural optimization of object-based NAND flash system. J. Guo et al. proposed multi-level garbage collection and B+ table tree to mitigate onode incurred partial page update and internal node page incurred cascade update [20]. Y.Lu et al. proposed lazy back-pointer index and compact update to reduce object data incurred partial page update from file system layer [56]. However, the lazy update scheme significantly increases memory consumption. Our proposed scheme can work with these previous works to further improve write performance. Y. Kang et al. proposed a object-based storage class memory (SCM) device model [42]. To evaluate the efficiency of the proposed SCM device, a simulation infrastructure needs to be setup. There are several simulators for NAND flash storage systems and object storage device. Fastsim [89] is one of the most popular NAND flash based solid disk drive simulator.

Researchers and system designers are also dedicated to exploring the space of architectural optimization by adopting non-volatile memory in NAND flash based storage systems. C. Sun et al. proposed to buffer data in SCM to minimize partial page update [81]. J. Guo et al. proposed to store flash translation layer (FTL) metadata in phase change memory (PCM) to minimize power failure protection cost [24]. J. Kim et al. proposed to store file system metadata and FTL metadata in PCM to avoid unnecessary data write and reduce main memory consumption [43]. Due to architectural difference, our HBS stores different types of system metadata in STT-RAM from these prior works. Despite a similar motivation as J. Kim's work, our work is different from the prior works at the aspect of optimization approach. Compared with PCM, STT-RAM has limited density despite lower access latency and higher endurance. To increase memory utilization, our HBS differentiates object metadata and buffers the metadata with most performance cost in STT-RAM.

There are other works [90, 23, 19, 22] proposed to improve performance of NAND flash storage system. Several prior works have proposed the use of STT-RAM to reduce energy consumption in on-chip cache or as a substitute of embedded DRAM [69, 99, 77]. While [48] evaluated STT-RAM as an alternative to DRAM in main memory.

5.3 Motivations

Write amplification shortens system endurance and incurs write performance degradation. Despite architectural optimization, write amplification still causes non-negligible write overhead in ONFD. One cause of write amplification is *per-object index cascading update*. The per-object indices are implemented with an B+ tree. The internal node page maintains physical addresses of child nodes (i.e. per-object index entries) and offset. The leaf node page records index entries indicating the address of the object data with format <offset, length, address>. When a node is updated, its indirect node is updated due to the direct pointer update. If a leaf node page is updated, due to erase-before-program issue, its direct index page should also be updated, leading to per-object index cascading update. Such cascading updates can be avoided by using a node address table [20, 50]. Each node page has a virtual address and a physical address. The internal node entry stores the virtual address of its child node pages. Similarly, to further avoid frequent onode update, virtual address of the index root page is stored in onode. When a leaf node page address is updated, only the new physical address is updated in node address table. In this way, cascade update can be effectively reduced. A per-object index cache may further reduce updates of the per-object index pages, especially when the write pattern is sequential. For fast accesses, updated leaf node page and its related internal node pages are cached. when the size of all buffered pages is higher than a per-defined threshold, the tree node pages of least-recently accessed object is evicted in a leaf-to-root sequence to the NAND flash memory. If the per-object index root page is updated, onode is committed only after internal node pages. However, write cost of per-object index and onode is still high due to its small size.

Similar to block-based NAND flash device, ONFD also poses a reliability challenge to system designers. In a typical ONFD, DRAM is adopted to buffer the frequently accessed metadata and data such as the node address table and per-object index table. To comply with data integrity during power failures, super-capacitors are usually used as backup power supply to back up a large amount of data from DRAM to NAND devices. However, according to Arrhenius law lifetime model, the capacitance loss of super-capacitors under 60°can be as high as 30% within five years [24]. The capacitance degradation severely impairs the reliability of the entire storage device: With DRAM-only buffer scheme, power failure may cause permanent loss of the node address table and per-object index cache and lead to inconsistency between object data and metadata.

5.4 STT-RAM/DRAM Hybrid Buffer Scheme

A STT-RAM/DRAM Hybrid Buffer Scheme (HBS) is proposed to further mitigate write amplification by utilizing byte addressability of STT-RAM. HBS also utilizes non-volatility of STT-RAM to simplify data recovery upon power failure.

The overall architecture of HBS is shown in Figure 26. Similar to the ONFD architecture described in Section 5.2.1, our HBS also adopts three data types for object management: onode, per-object index and object data. Unlike the baseline ONFD, the placement of object data depends on data size. If the total size of inode and data are less than one physical page, inode and data are stored together without per-object index; otherwise, object inodes, per-object internal indices and object data are stored in different chucks.

The hybrid management layer implements flash management policies like page allocation, garbage collection and wear-leveling. It also manipulates received requests in a data type differentiating strategy.

5.4.1 Design Overview

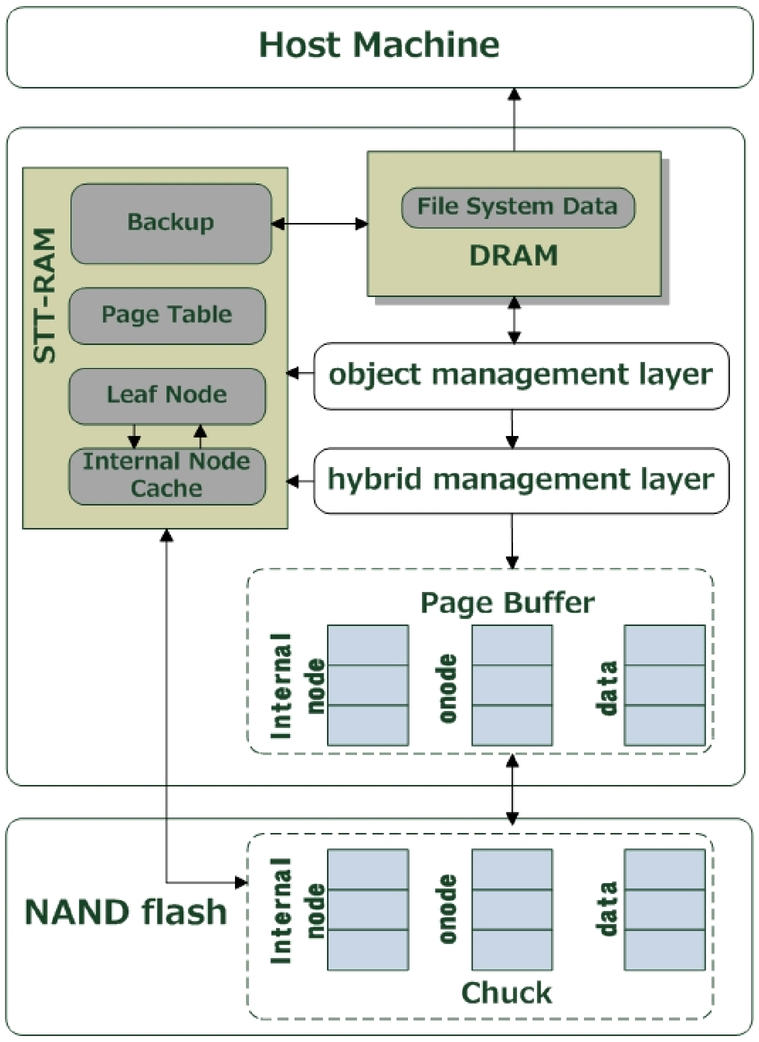


Figure 26: The architecture of proposed scheme

For onode and object data, the object-based interface allows byte addressing. In NAND flash memory, data is accessed in unit of a page. Hence, the management layer has to handle sub-page write: 1) If sub-page write is performed to existing data, read-before-write operation is performed as in [81]. 2) If sub-page write is performed to new data, the new data is first stored in page-size buffer. DRAM is used as a page buffer before data are flushed into NAND flash array.

For per-object indices, a page table is used to address the cascading update problem resulted from a chain of writes [50]. A STT-RAM is partitioned into three regions: metadata storage, per-object index entry cache and backup region. The buffer management technique to maximize the memory utilization and extend endurance by minimizing data migration during power failure.

5.4.2 Hybrid Buffer Module

Metadata storage contains a B+ tree page table and leaf nodes in page granularity. The idea is using the concept of virtual address to prevent recursive update. The mapping between physical address and virtual address is stored in a page table. Figure 27 is an example of tree node address table. When a leaf node entry in virtual address (VA) 5 is updated, clean data with updated data is migrated to the new physical address (PA) 14. Data in PA 4 becomes dirty and its mapping to VA 5 is invalidated. By updating the VA to PA mapping table, recursive updates in B+ tree is avoided.

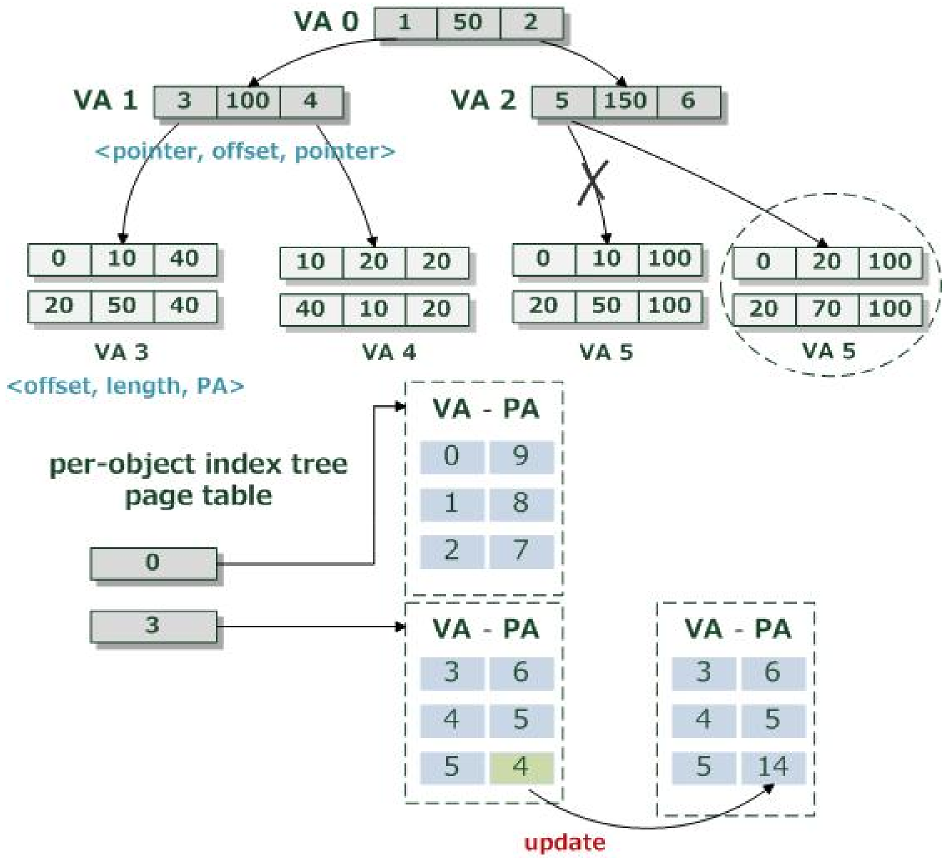


Figure 27: The per-object index table tree and an example of node update

Despite reduced updates of the internal node pages and the onode pages, object data updates still generate considerable write accesses to leaf node pages, especially for write-intensive workloads. An intuitive solution is to migrate leaf node pages to STT-RAM for better write performance and faster access. To estimate the approximate capacity requirement for leaf node storage in STT-RAM, we utilize a real world benchmark TPC-C. The benchmark involves 40GB of object data and the maximum file size is 14GB. There are 600 objects in total and the per-object index tree depth is 4. The page size is 8KB. Considering the tree depth, there are a maximum of 2^{20} per-object index leaf node entries and about 2^{21} per-object index nodes in a balanced tree. In the page table, a reasonable assumption of virtual address is 3B which is 1B less than physical address for space optimization. Thus, the approximate size of the page table is 14MB. A leaf node entry contains offset, length, physical address with 4B size each. Thus, the leaf node storage occupies about 12MB, which is acceptable. This estimation is not sufficient for reasons below: (1) the benchmark size is too small when compared to real world file system; (2) the leaf node number depends heavily on benchmark characteristics, for example, benchmarks with a large amount of small size user data tend to have more per-object index leaf nodes. However, we still take the validity of leaf node storage in STT-RAM: (1) per-object index tree with more leaf nodes tends to have a higher tree depth; (2) file systems with a majority of small files are uncommon and most desktops share a similar file size distribution [2]. As a result, leaf node updates are in-place update and cascading update is minimized. Scenarios still incur an cascading update include node insertion and deletion.

When power failure occurs, the DRAM data which has not been flushed into the NAND flash array will be copied to the STT-RAM. In the proposed design, the size of data migration is minimized by levering the non-volatility of STT-RAM. Because capacitance budget is directly influenced by the size of the DRAM data that has to be moved, the reduction of capacitance budget enables the possibility of replacing the super-capacitors with the more reliable regular capacitors.

5.4.3 Hybrid Buffer Management

For read request, the onode address is retrieved from the global index tree and the per-object index with offset is retrieved from corresponding onode page in NAND flash. Before accessing the NAND flash chuck, internal node cache is referred to for fast accesses. On a cache hit, search for subsequent tree data path until a cache miss or a leaf node address is achieved. On a cache miss, a page-based popularity list is updated and NAND flash is accessed for the requested address.

The page popularity list is maintained in backup when power on and will be invalidated immediately if there is a power failure. On a cache miss, we increment the page popularity by 1. Internal node cache updates in LRU policy when the most popularity page count reaches a threshold.

For write request, sub-page write algorithm is shown below. The old object data is invalidated first on updates to existing data. The new object data is wrapped into object operation and inserted to the data queue for process. Once the object data is flushed into NAND flash, the per-object indices are updated accordingly. The per-object index tree adopts a lazy update algorithm. Upon node insertion/deletion, only the prior pointer is updated. The dirty internal nodes data is invalidated first. The new index is inserted into the internal node queue. After the internal node page is flushed into NAND flash, the page table in STT-RAM is updated. The onodes are updated in a similar approach. In write-before-read scenario, both the onode/per-object index page in NAND and the onode/per-object index data queue are searched for the updated indices.

Algorithm 5 Read algorithm for HBS

```
 1: procedure NAND_READ_OBJ(OID, OFFSET, LEN)
 2:     read onode from flash memory
 3: STT-RAM:
 4:     while internal node cache hit & node != leaf do
 5:         retrieve child node
 6:     if node $\neq$ leaf then
 7:         popularity list entry increment
 8:         if most popular index count $>$ threshold then
 9:             LRU replacement in internal node cache
10:         end if
11:     end if
12: NAND flash:
13:     while address $\in$ per-object index data chuck do
14:         look up per-object index
15:     read object data
```

Algorithm 6 Write algorithm for HBS

```
1:  procedure NAND_WRITE_OBJ(OID, OFFSET, LEN)
2:      read onode and per-object index as in Algorithm 5
3:      if <offset,len> $\in$ flash memory then
4:          remaining data $\rightarrow$ $data\_queue$
5:          invalid out-of-date data pages
6:      end if
7:      data to write with oid $\rightarrow$ $data\_queue$
8:  $handle\_data\_queue$:
9:      while $data\_queue \neq$ NULL do
10:         dequeue $data\_queue \Rightarrow \tau$
11:         if $dirty\_chucks >$ threshold then
12:             $recycle\_dirty\_chuck()$
13:         end if
14:         if <offset,len> of $\tau \notin$ leaf node then
15:             flush $\tau$ to flash memory
16:             if $avaliable\_leaf\_page = 0$ then
17:                 $nand\_write\_inode()$
18:             end if
19:             insert leaf node entry
20:         else
21:             flush $\tau$ to flash memory
22:             update page table in STT-RAM with new PA
23:         end if
```

5.4.4 Software Implementation

ObjNandSim implements the device architecture described in Section 5.2.1. To facilitate code modification, ObjNandSim provides a simple API for access to object data and metadata (inode) in the OSD. The API functions are listed as follows:

- $nand\_write\_obj()$, $nand\_read\_obj()$: write and read object data;
- $nand\_delete\_obj()$, $nand\_truncate\_obj()$: delete or truncate object; and
- $nand\_write\_inode()$, $nand\_read\_inode()$: write and read object inode.

Since ObjNandSim provides sufficient modularity and extend-ability, we are able to implement HBS by:

- making modifications to B+ tree operations by differentiating between internal node *struct bplus_block* and leaf node *struct index_block*
- adding a LRU cache for internal node updates in meta-operation $nand\_write\_inode()$
- managing hybrid buffer in object operation *struct obj_op* and $nand\_handle\_queue\_op()$

5.5 Evaluation

In this section, we evaluate object-based NAND flash storage with hybrid buffer using trace driven simulations. For benchmarks in different access patterns, we measure the overall average I/O response time, page write and erase statistics to compare with the DRAM-only scheme.

5.5.1 Simulation Setup

In our experiments, we adopt the same simulation platform as in [21]. The object file system, object-based storage(OSD) initiator and the simulator run on the host machine equipped with Intel Quad-Core Xeon 5-2609 v2 (10MB 2GHz) processor and 128GB RAM. The OSD initiator and the simulator are connected via the Gigabit Ethernet. In addition, the clean chunk threshold is set 100 to quickly initiate garbage collection.

By default, the ObjNandSim is configured with device capacity 64 GB with 16 NAND flash memory dies; the channel number and the chunk size are 16 and 2MB, respectively. The characteristics of these workload traces are shown in Table 8. Benchmark entries are replayed to generate input to the simulation platform. In this simulation platform, we adopt EXOFS in Linux kernel 3.2.67 as the object file system [28]. The OSD initiator is an existing Linux kernel library which generates OSD request packets according to SCSI OSD command sets [61]. Finally, OSD requests packets are sent to the OSD via open-iSCSI, an iSCSI initiator.

Table 8: Workload characteristics

Workload	TPC-C [52]	iPhoto [29]	Pages [29]	iMovie [29]
Pattern	random access	write intensive	small data access	sequential write access
Avg. request size (KB)	15.7	12	0.5	21
Avg. file size (KB)	67321	233	163	283
Write access	24.7%	90.3%	91%	60%
Write seq.	19%	36%	1%	92%
Read seq.	14%	15%	51%	18%

5.5.2 Overall Performance Improvement

We implement STT-RAM/DRAM HBS and supplement buffer management on ObjNandSim. The performance evaluation results are shown in Figure 28-30. Compared to the default ObjNandSim, HBS can achieves up to 15% average response time improvement and an average of 7% of all benchmarks. The workload of iPhoto has the maximum response time reduction because of a significant page write reduction. Even under read-intensive TPC-C, there is$\tilde{1}$0% response time reduction due to page write reduction in HBS. However, the workload of Pages show little improvement in the overall efficiency. It is reasonable because per-object index write is completely eliminated when object data and onode are small and reside in one page without per-object index. The page write is reduced by 13% on average. Workloads of TPC-C and iPhoto have similar write

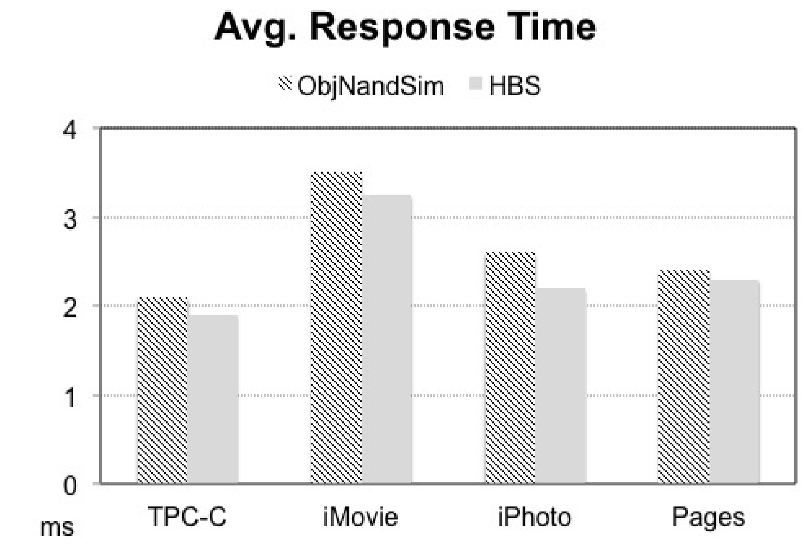


Figure 28: Average write response time

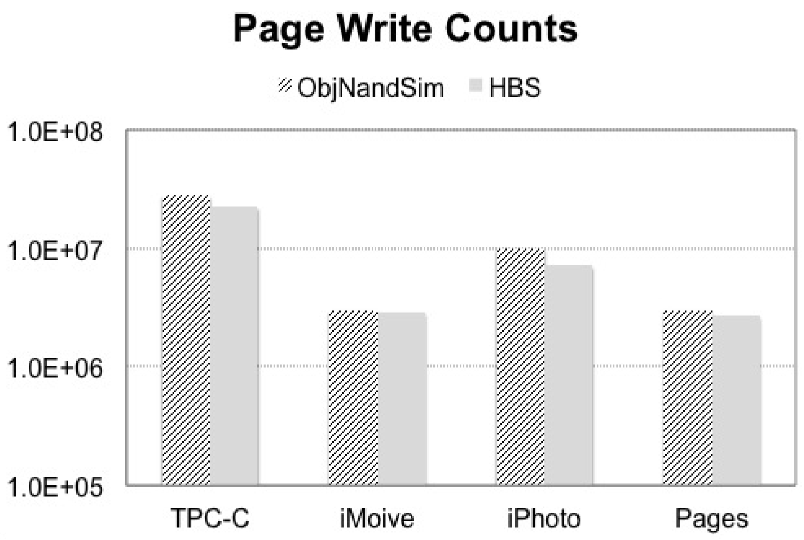


Figure 29: Total page write counts

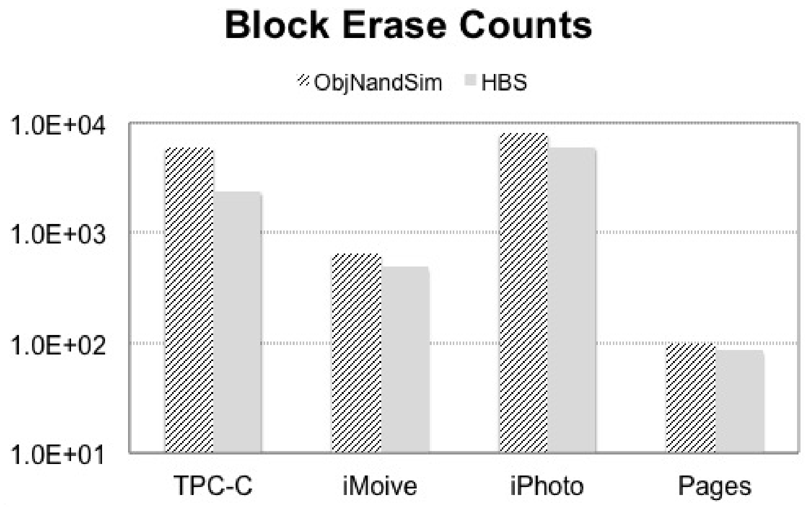


Figure 30: Total block erase counts

request count and average write request size. In our simulation results, page write count improvement of iPhoto (28%) outperforms TPC-C (20%). This may result from the better sequentiality in write access in iPhoto. The per-object index entry cache has higher hit rate with sequential access pattern. The average block erasure reduction is 31%. Due to reduction of page write and block erasure, HBS improves the system endurance by 34% on average for a normalized write request count of 1.0E+06 compared to ObjNandSim.

5.6 Summary

In an object-based NAND flash device, per-object indices are much more frequently updated than object data and can be considered as "hot" data. A major cause of write amplification is cascading update within per-object index. A page table is deployed to map the frequently updated physical address to the constant virtual address, therefore, most of the internal node page updates can be avoided. Moreover, a LRU cache is implemented to further reduce data migration. The cache buffers the updated leaf node and its corresponding internal node entries. Updated internal node entries are committed on cache replacement and merged to internal node page before eviction to NAND flash device. For leaf nodes, we estimate its approximate size via a real-world benchmark and propose to place all leaf nodes in STT-RAM for fast and frequent access. The per-object index page table, LRU cache and leaf nodes are reside in STT-RAM for data and metadata consistency during power failure. Finally, the rest of the 256MB STT-RAM is used as backup for DRAM, considering the size of data to be migrated and P/E latency of NAND flash device. By adopting STT-RAM as backup, super-capacitors can be replaced by normal capacitors for reliability. The experimental results show that, compared with the state-of-art work, our proposed design can reduce page write count by 13% with 34% endurance improvement.

6.0 Conclusions

Thanks to low energy and high throughput properties of the HMC, near-memory processing has been a good candidate for neural network accelerator design. Despite efforts in exploring model compression and architectural optimization, the system designers and researchers are still faced with challenges from both aspects: 1) massive number of operations for storage/compute, 2) specific PIM features.

In this dissertation, we first propose a filter pruning strategy to reduce computation and memory accesses. We propose to use conditional entropy as the filter selection criteria in filter pruning. For each sample in a dataset, we calculate the cross entropy loss and output activation corresponding to each filter. We then evaluate the filter selection criteria in a layer-wise pruning approach. Experimental result shows that the performance of conditional based filter selection criteria outperforms the approaches based L1-Norm, APoZ and activation entropy. The proposed criteria can achieve 92.76% accuracy when pruning ratio is 50% and $\sim 93.5\%$ accuracy when pruning ratio is 25% on VGG16, while the baseline L1-norm results in 92.11% accuracy for 50% and 92.86% accuracy for 25% pruning ratio. To comply with the network information distribution, we adopt a two phase pruning framework which combines global approach with above layer-wise approach. The filters with the minimum conditional entropy are pruned filter-by-filter globally followed by layer-wise pruning. The above framework can achieve a pruning ratio of 87% within 2% accuracy drop of pre-trained VGG-16 model on CIFAR10.

With compressed model, we then 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 $4.1\times$ in speedup and $1.5\times$ 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.

As an extension of HMC based accelerator, we utilize affinity-based kernel offloading to accelerate GAN, in which pipeline is deferred by loss synchronization. By adopting co-location of computation and data in HMC memory stacks, the intermediate results including partial gradient and error, are stored in SRAM buffer of the same vault where the computation takes place. In this way, despite the long data reuse distance, gradient synchronization in backpropagation does not incur long memory access latency. Moreover, we refined the training to be asynchronous for one iteration but achieve more parallelism than the original GAN. Our simulation result demonstrates $1.6\times$ speedup compared with the compiler based Nearest-Data-Processing (NDP) system.

In future work, to make HMC accelerator practically viable, there are other challenges that need to be addressed, including virtual memory support to ensure a unified address space, memory/cache coherence, fault tolerance, security and privacy, thermal and power constraints, compatibility with modern programming models, etc.

Bibliography

[1] Ossama Abdel-Hamid, Abdel-rahman Mohamed, Hui Jiang, and Gerald Penn. Applying convolutional neural networks concepts to hybrid nn-hmm model for speech recognition. In *Acoustics, Speech and Signal Processing (ICASSP), 2012 IEEE International Conference on*, pages 4277–4280. IEEE, 2012.

[2] Nitin Agrawal, William J Bolosky, John R Douceur, and Jacob R Lorch. A five-year study of file-system metadata. In *Proceedings of the 5th USENIX conference on File and Storage Technologies*, pages 3–3, 2007.

[3] Junwhan Ahn, Sungjoo Yoo, and Kiyoung Choi. Dynamic power management of off-chip links for hybrid memory cubes. In *Proceedings of the 2014 ACM/EDAC/IEEE Design Automation Conference*, pages 1–6, 2014.

[4] Erfan Azarkhish, Davide Rossi, Igor Loi, and Luca Benini. Design and evaluation of a processing-in-memory architecture for the smart memory cube. In *International Conference on Architecture of Computing Systems*, pages 19–31. Springer, 2016.

[5] Jimmy Ba and Rich Caruana. Do deep nets really need to be deep? In *Advances in neural information processing systems*, pages 2654–2662, 2014.

[6] Rajeev Balasubramonian, Jichuan Chang, Troy Manning, Jaime H Moreno, Richard Murphy, Ravi Nair, and Steven Swanson. Near-data processing: Insights from a micro-46 workshop. *IEEE Micro*, 34(4):36–42, 2014.

[7] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R Hower, Tushar Krishna, Somayeh Sardashti, et al. The gem5 simulator. *ACM SIGARCH Computer Architecture News*, 39(2), 2011.

[8] Tianshi Chen, Zidong Du, Ninghui Sun, Jia Wang, Chengyong Wu, Yunji Chen, and Olivier Temam. Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning. *ACM Sigplan Notices*, 49(4), 2014.

[9] Yu-Hsin Chen, Tushar Krishna, Joel S Emer, and Vivienne Sze. Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks. *IEEE Journal of Solid-State Circuits*, 52(1), 2017.

[10] Ping Chi, Shuangchen Li, Cong Xu, Tao Zhang, Jishen Zhao, Yongpan Liu, Yu Wang, and Yuan Xie. 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*, pages 27–39, 2016.

[11] Taco Cohen and Max Welling. Group equivariant convolutional networks. In *International conference on machine learning*, pages 2990–2999, 2016.

[12] Ronan Collobert and Jason Weston. A unified architecture for natural language processing: Deep neural networks with multitask learning. In *Proceedings of the 25th international conference on Machine learning*, pages 160–167. ACM, 2008.

[13] Hybrid Memory Cube Consortium et al. Hmc specification 2.0, 2015.

[14] William Dally. High-performance hardware for machine learning. *NIPS Tutorial*, 2015.

[15] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Marc'aurelio Ranzato, Andrew Senior, Paul Tucker, Ke Yang, et al. Large scale distributed deep networks. In *Advances in neural information processing systems*, pages 1223–1231, 2012.

[16] Misha Denil, Babak Shakibi, Laurent Dinh, Nando De Freitas, et al. Predicting parameters in deep learning. In *Advances in neural information processing systems*, pages 2148–2156, 2013.

[17] Yunchao Gong, Liu Liu, Ming Yang, and Lubomir Bourdev. Compressing deep convolutional networks using vector quantization. *arXiv preprint arXiv:1412.6115*, 2014.

[18] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial nets. In *Advances in neural information processing systems*, pages 2672–2680, 2014.

[19] Jie Guo, Zhijie Chen, Danghui Wang, Zili Shao, and Yiran Chen. DPA: A data pattern aware error prevention technique for NAND flash lifetime extension. In *2014 19th Asia and South Pacific Design Automation Conference*, pages 592–597, 2014.

[20] Jie Guo, Chuhan Min, Tao Cai, and Yiran Chen. A design to reduce write amplification in object-based NAND flash devices. In *Proceedings of the 11th IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, pages 5:1–5:10, 2016.

[21] Jie Guo, Chuhan Min, Tao Cai, Hai Li, and Yiran Chen. ObjNandSim: Object-based nand flash device simulator. In *Proceedings of the 5th Non-Volatile Memory Systems and Applications Symposium*, pages 1–6, 2016.

[22] Jie Guo, Wujie Wen, Jingtong Hu, Danghui Wang, Hai Li, and Yiran Chen. FlexLevel: A novel nand flash storage system design for LDPC latency reduction. In *Proceedings of the 52Nd Annual Design Automation Conference*, pages 194:1–194:6, 2015.

[23] Jie Guo, Wujie Wen, Yaojun Zhang Li, Sicheng Li, Hai Li, and Yiran Chen. DA-RAID-5: A disturb aware data protection technique for NAND flash storage systems. In *Proceedings of the 2013 Design, Automation & Test in Europe Conference & Exhibition*, pages 380–385, 2013.

[24] Jie Guo, Jun Yang, Youtao Zhang, and Yiran Chen. Low cost power failure protection for MLC NAND flash storage systems with PRAM/DRAM hybrid buffer. In *Proceedings of the 2013 Design, Automation & Test in Europe Conference & Exhibition*, pages 859–864, 2013.

[25] Ramyad Hadidi, Bahar Asgari, Jeffrey Young, Burhan Ahmad Mudassar, Kartikay Garg, Tushar Krishna, and Hyesoon Kim. 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*, pages 99–108, 2018.

[26] Song Han, Huizi Mao, and William J Dally. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149*, 2015.

[27] Song Han, Jeff Pool, John Tran, and William Dally. Learning both weights and connections for efficient neural network. In *Advances in neural information processing systems*, pages 1135–1143, 2015.

[28] B Harrosh and B Halevy. The linux exofs object-based pnfs metadata server, 2009.

[29] Tyler Harter, Chris Dragga, Michael Vaughn, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. A file is not a file: understanding the I/O behavior of apple desktop applications. *ACM Transactions on Computer Systems*, 30(3):10, 2012.

[30] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[31] Wim Heirman, Trevor Carlson, and Lieven Eeckhout. Sniper: Scalable and accurate parallel multi-core simulation. In *8th International Summer School on Advanced Computer Architecture and Compilation for High-Performance and Embedded Systems (ACACES-2012)*, pages 91–94. High-Performance and Embedded Architecture and Compilation Network of Excellence (HiPEAC), 2012.

[32] Geoffrey Hinton, Li Deng, Dong Yu, George E Dahl, Abdel-rahman Mohamed, Navdeep Jaitly, Andrew Senior, Vincent Vanhoucke, Patrick Nguyen, Tara N Sainath, et al. Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups. *IEEE Signal Processing Magazine*, 29(6):82–97, 2012.

[33] Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. Distilling the knowledge in a neural network. *arXiv preprint arXiv:1503.02531*, 2015.

[34] Hengyuan Hu, Rui Peng, Yu-Wing Tai, and Chi-Keung Tang. Network trimming: A data-driven neuron pruning approach towards efficient deep architectures. *arXiv preprint arXiv:1607.03250*, 2016.

[35] Forrest N Iandola, Song Han, Matthew W Moskewicz, Khalid Ashraf, William J Dally, and Kurt Keutzer. Squeezenet: Alexnet-level accuracy with 50x fewer parameters and¡ 0.5 mb model size. *arXiv preprint arXiv:1602.07360*, 2016.

[36] Yani Ioannou, Duncan Robertson, Jamie Shotton, Roberto Cipolla, and Antonio Criminisi. Training cnns with low-rank filters for efficient image classification. *arXiv preprint arXiv:1511.06744*, 2015.

[37] Phillip Isola, Jun-Yan Zhu, Tinghui Zhou, and Alexei A Efros. Image-to-image translation with conditional adversarial networks. *arXiv preprint arXiv:1611.07004*, 2016.

[38] Phillip Isola, Jun-Yan Zhu, Tinghui Zhou, and Alexei A Efros. Image-to-image translation with conditional adversarial networks. In *Proceedings of the 2017 IEEE Conference on Computer Vision and Pattern Recognition*, pages 5967–5976, 2017.

[39] Dong-Ik Jeon, Kyeong-Bin Park, and Ki-Seok Chung. Hmc-mac: Processing-in memory architecture for multiply-accumulate operations with hybrid memory cube. *IEEE Computer Architecture Letters*, 17(1), 2018.

[40] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast

feature embedding. In *Proceedings of the 22nd ACM international conference on Multimedia*, pages 675–678. ACM, 2014.

[41] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, pages 1–12, 2017.

[42] Yangwook Kang, Jingpei Yang, and Ethan L Miller. Object-based SCM: An efficient interface for storage class memories. In *Proceedings of the 2011 IEEE 27th Symposium on Mass Storage Systems and Technologies*, pages 1–12, 2011.

[43] Jin Kyu Kim, Hyung Gyu Lee, Shinho Choi, and Kyoung Il Bahng. A PRAM and NAND flash hybrid architecture for high-performance embedded storage subsystems. In *Proceedings of the 8th ACM International Conference on Embedded Software*, pages 31–40, 2008.

[44] Artemy Kolchinsky and Brendan D Tracey. Estimating mixture entropy with pairwise distances. *Entropy*, 19(7):361, 2017.

[45] Alexander Kraskov, Harald Stögbauer, and Peter Grassberger. Estimating mutual information. *Physical review E*, 69(6):066138, 2004.

[46] Alex Krizhevsky. Learning multiple layers of features from tiny images. Technical report, Citeseer, 2009.

[47] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[48] Emre Kültürsay, Mahmut Kandemir, Anand Sivasubramaniam, and Onur Mutlu. Evaluating STT-RAM as an energy-efficient main memory alternative. In *Proceedings of the 2013 IEEE International Symposium on Performance Analysis of Systems and Software*, pages 256–267, 2013.

[49] Yann LeCun, John S Denker, and Sara A Solla. Optimal brain damage. In *Advances in neural information processing systems*, pages 598–605, 1990.

[50] Changman Lee, Dongho Sim, Jooyoung Hwang, and Sangyeun Cho. F2FS: A new file system for flash storage. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies*, pages 273–286, 2015.

[51] John D Leidel and Yong Chen. 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*, pages 621–630, 2016.

[52] Scott T. Leutenegger and Daniel Dias. A modeling study of the TPC-C benchmark. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, pages 22–31, 1993.

[53] Hai Li and Yiran Chen. An overview of non-volatile memory technology and the implication for tools and architectures. In *Proceedings of the 2009 Design, Automation & Test in Europe Conference & Exhibition*, pages 731–736, 2009.

[54] Hao Li, Asim Kadav, Igor Durdanovic, Hanan Samet, and Hans Peter Graf. Pruning filters for efficient convnets. *arXiv preprint arXiv:1608.08710*, 2016.

[55] Min Lin, Qiang Chen, and Shuicheng Yan. Network in network. *arXiv preprint arXiv:1312.4400*, 2013.

[56] Youyou Lu, Jiwu Shu, and Weimin Zheng. Extending the lifetime of flash-based storage through reducing write amplification from file systems. In *Proceedings of the 11th USENIX Conference on File and Storage Technologies*, pages 257–270, 2013.

[57] Jian-Hao Luo and Jianxin Wu. An entropy-based pruning method for cnn compression. *arXiv preprint arXiv:1706.05791*, 2017.

[58] Jiachen Mao, Xiang Chen, Kent W Nixon, Christopher Krieger, and Yiran Chen. Modnn: Local distributed mobile computing system for deep neural network. In *Proceedings of the 2017 Design, Automation & Test in Europe Conference & Exhibition*, pages 1396–1401, 2017.

[59] JEDEC Standard High Bandwidth Memory. Dram specification. *Standard JESD235A*, 2015.

[60] Pavlo Molchanov, Stephen Tyree, Tero Karras, Timo Aila, and Jan Kautz. Pruning convolutional neural networks for resource efficient inference. *arXiv preprint arXiv:1611.06440*, 2016.

[61] David Nagle, ME Factor, Sami Iren, Dalit Naor, Erik Riedel, Ohad Rodeh, and Julian Satran. The ANSI T10 object-based storage standard and current implementations. *IBM Journal of Research and Development*, 52(4/5):401–411, 2008.

[62] Lifeng Nai and Hyesoon Kim. Instruction offloading with hmc 2.0 standard: A case study for graph traversals. In *Proceedings of the 2015 International Symposium on Memory Systems*, pages 258–261. ACM, 2015.

[63] Augustus Odena. Semi-supervised learning with generative adversarial networks. *arXiv preprint arXiv:1606.01583*, 2016.

[64] Augustus Odena, Vincent Dumoulin, and Chris Olah. Deconvolution and checkerboard artifacts. *Distill*, 1(10):e3, 2016.

[65] A. Pattnaik, X. Tang, A. Jog, O. Kayiran, A. K. Mishra, M. T. Kandemir, O. Mutlu, and C. R. Das. Scheduling techniques for gpu architectures with processing-in-memory capabilities. In *2016 International Conference on Parallel Architecture and Compilation Techniques (PACT)*, pages 31–44, Sept 2016.

[66] J Thomas Pawlowski. Hybrid memory cube: breakthrough dram performance with a fundamentally re-architected dram subsystem. In *Hot Chips*, volume 23, 2011.

[67] Yao Qian, Yuchen Fan, Wenping Hu, and Frank K Soong. 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*, pages 3829–3833, 2014.

[68] Alec Radford, Luke Metz, and Soumith Chintala. Unsupervised representation learning with deep convolutional generative adversarial networks. *arXiv preprint arXiv:1511.06434*, 2015.

[69] Mitchelle Rasquinha, Dhruv Choudhary, Subho Chatterjee, Saibal Mukhopadhyay, and Sudhakar Yalamanchili. An energy efficient cache design using spin torque transfer (STT) RAM. In *Proceedings of the 16th ACM/IEEE international symposium on Low Power Electronics and Design*, pages 389–394, 2010.

[70] Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi. Xnor-net: Imagenet classification using binary convolutional neural networks. In *European Conference on Computer Vision*, pages 525–542. Springer, 2016.

[71] B. Reagen, P. Whatmough, R. Adolf, S. Rama, H. Lee, S. K. Lee, J. M. Hernández-Lobato, G. Wei, and D. Brooks. Minerva: Enabling low-power, highly-accurate deep neural network accelerators. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pages 267–278, June 2016.

[72] Paul Rosenfeld, Elliott Cooper-Balis, and Bruce Jacob. Dramsim2: A cycle accurate memory system simulator. *IEEE Computer Architecture Letters*, 10(1):16–19, 2011.

[73] Andrew Michael Saxe, Yamini Bansal, Joel Dapello, Madhu Advani, Artemy Kolchinsky, Brendan Daniel Tracey, and David Daniel Cox. On the information bottleneck theory of deep learning. In *International Conference on Learning Representations*, 2018.

[74] Ravid Shwartz-Ziv and Naftali Tishby. Opening the black box of deep neural networks via information. *arXiv preprint arXiv:1703.00810*, 2017.

[75] Patrick Siegl, Rainer Buchty, and Mladen Berekovic. A bandwidth accurate, flexible and rapid simulating multi-hmc modeling tool. In *Proceedings of the 2017 International Symposium on Memory Systems*, pages 71–82, 2017.

[76] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[77] Clinton W Smullen, Vidyabhushan Mohan, Anurag Nigam, Sudhanva Gurumurthi, and Mircea R Stan. Relaxing non-volatility for fast and energy-efficient STT-RAM caches. In *Proceedings of 2011 IEEE 17th International Symposium on High Performance Computer Architecture*, pages 50–61, 2011.

[78] Avinash Sodani. Knights landing (knl): 2nd generation intel® xeon phi processor. In *Hot Chips 27 Symposium (HCS), 2015 IEEE*, pages 1–24. IEEE, 2015.

[79] Jost Tobias Springenberg. Unsupervised and semi-supervised learning with categorical generative adversarial networks. *arXiv preprint arXiv:1511.06390*, 2015.

[80] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *The Journal of Machine Learning Research*, 15(1):1929–1958, 2014.

[81] Chao Sun, Kousuke Miyaji, Koh Johguchi, and Ken Takeuchi. SCM capacity and NAND over-provisioning requirements for SCM/NAND flash hybrid enterprise SSD. In *Proceedings of the 5th IEEE International Memory Workshop*, pages 64–67, 2013.

[82] Christian Szegedy, Sergey Ioffe, Vincent Vanhoucke, and Alexander A Alemi. Inception-v4, inception-resnet and the impact of residual connections on learning. In *Thirty-First AAAI Conference on Artificial Intelligence*, 2017.

[83] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *Computer Vision and Pattern Recognition (CVPR)*, 2015.

[84] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1–9, 2015.

[85] Cheng Tai, Tong Xiao, Yi Zhang, Xiaogang Wang, et al. Convolutional neural networks with low-rank regularization. *arXiv preprint arXiv:1511.06067*, 2015.

[86] Xulong Tang, Orhan Kislal, Mahmut Kandemir, and Mustafa Karakoy. Data movement aware computation partitioning. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 730–744. ACM, 2017.

[87] Naftali Tishby and Noga Zaslavsky. Deep learning and the information bottleneck principle. In *Information Theory Workshop (ITW), 2015 IEEE*, pages 1–5. IEEE, 2015.

[88] Benny Van Houdt. Performance of garbage collection algorithms for flash-based solid state drives with hot/cold data. *Performance Evaluation*, 70(10):692–703, 2013.

[89] Nicolas Vasilache, Jeff Johnson, Michael Mathieu, Soumith Chintala, Serkan Piantino, and Yann LeCun. Fast convolutional nets with fbfft: A gpu performance evaluation. *arXiv preprint arXiv:1412.7580*, 2014.

[90] Danghui Wang, Jie Guo, Kai Bu, and Yiran Chen. Reduction of data prevention cost and improvement of reliability in MLC NAND flash storage system. In *2014 International Conference on Computing, Networking and Communications*, pages 259–263, 2014.

[91] Wei Wen, Chunpeng Wu, Yandan Wang, Yiran Chen, and Hai Li. Learning structured sparsity in deep neural networks. In *Advances in Neural Information Processing Systems*, pages 2074–2082, 2016.

[92] Ross S Williamson, Maneesh Sahani, and Jonathan W Pillow. The equivalence of information-theoretic and likelihood-based methods for neural dimensionality reduction. *PLoS computational biology*, 11(4):e1004141, 2015.

[93] Wm A Wulf and Sally A McKee. Hitting the memory wall: implications of the obvious. *ACM SIGARCH computer architecture news*, 23(1):20–24, 1995.

[94] Raymond Yeh, Chen Chen, Teck Yian Lim, Mark Hasegawa-Johnson, and Minh N Do. Semantic image inpainting with perceptual and contextual losses. *arXiv preprint arXiv:1607.07539*, 2016.

[95] Shuangfei Zhai, Yu Cheng, Zhongfei Mark Zhang, and Weining Lu. Doubly convolutional neural networks. In *Advances in neural information processing systems*, pages 1082–1090, 2016.

[96] Chen Zhang, Peng Li, Guangyu Sun, Yijin Guan, Bingjun Xiao, and Jason Cong. Optimizing fpga-based accelerator design for deep convolutional neural networks. In *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 161–170, 2015.

[97] Xiangyu Zhang, Jianhua Zou, Xiang Ming, Kaiming He, and Jian Sun. Efficient and accurate approximations of nonlinear convolutional networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 1984–1992, 2015.

[98] Yiying Zhang, Leo Prasath Arulraj, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. De-indirection for flash-based SSDs with nameless writes. In *Proceedings of the 10th USENIX conference on File and Storage Technologies*, pages 1–1, 2012.

[99] Ping Zhou, Bo Zhao, Jun Yang, and Youtao Zhang. Energy reduction for STT-RAM using early write termination. In *Proceedings of the 2009 International Conference on Computer-Aided Design*, pages 264–268, 2009.