Array Aware Training/Pruning: Methods for Efficient Forward Propagation on Array-based Neural Network Accelerators

Krishna Teja Chitty-Venkata and Arun K. Somani
Department of Electrical and Computer Engineering
Iowa State University, Ames, Iowa
krishnat@iastate.edu, arun@iastate.edu

***Abstract*—Due to the increase in the use of large-sized Deep Neural Networks (DNNs) over the years, specialized hardware accelerators such as Tensor Processing Unit and Eyeriss have been developed to accelerate the forward pass of the network. The essential component of these devices is an array processor which is composed of multiple individual compute units for efficiently executing Multiplication and Accumulation (MAC) operation. As the size of this array limits the amount of DNN processing of a single layer, the computation is performed in several batches serially leading to extra compute cycles along both the axes. In practice, due to the mismatch between matrix and array sizes, the computation does not map on the array exactly. In this work, we address the issue of minimizing processing cycles on the array by adjusting the DNN model parameters by using a structured hardware array dependent optimization. We introduce two techniques in this paper: Array Aware Training (AAT) for efficient training and Array Aware Pruning (AAP) for efficient inference. Weight pruning is an approach to remove redundant parameters in the network to decrease the size of the network. The key idea behind pruning in this paper is to adjust the model parameters (the weight matrix) so that the array is fully utilized in each computation batch. Our goal is to compress the model based on the size of the array so as to reduce the number of computation cycles. We observe that both the proposed techniques results into similar accuracy as the original network while saving a significant number of processing cycles (75%).**

***Keywords*—Array, Deep Neural Networks, Accelerators, Training, Pruning**

I. Introduction

Deep Learning has emerged as a major problem-solving tool covering a wide range of applications which includes image recognition, machine translation, speech-to-text conversion, etc. [1]. Deep Neural Network (DNN) is a collection of multiple layers with a varying number of neurons across different layers. Each layer computes a non-linear function of the weighted sum of inputs plus a bias ($y_j = b_j + \sum_{i=1}^{n} W_{ij} * x_i$) from the previous layer to produce an output which propagates to the adjacent layer. The processing in every layer translates into a matrix multiplication operation between weight and an activation matrix in which the former remains constant. The number of columns in the latter matrix corresponds to the number of data samples applied to the network, and the number of rows corresponds to the number of neurons.

Special purpose hardware architectures [2], [3] have gained popularity to efficiently implement the forward path (Matrix Multiplication) which is the underlying operation in these networks. Google's Tensor Processing Unit (TPU) [4] has been designed for Multi-layer Perceptrons (MLP) as the workload in their data-center was predominantly Dense or Fully Connected layers. On the other hand, MIT's Eyeriss [5] was highly optimized for the inference execution of Convolutional Neural Networks. The heart of these machines is a systolic array processor which is made up of a collection of multiple individual components called Processing Elements (PEs). Each PE is designed to perform a specific task and pass the intermediate results to its neighbouring PE(s) [6]. This structure is designed to exploit both pipelining and parallelism and can be extended horizontally or vertically.

As the forward propagation of DNN is common in both training and inferences phases, these devices can be used to accelerate both the stages. The forward pass computation time is mainly limited by the size of the underlying array processor. A matrix of size X*Y is blocked into multiple M*N sub-matrices along rows and columns to fit on the array of size M*N. The matrices, which are perfect multiples of the size of the array, would use all PEs effectively. However, the ones which are not, cause the device to use a partial set of PEs in the last sub-matrix, which leads to spending extra cycles along rows and columns. Therefore, choosing the number of neurons in each layer without being aware of the hardware specifications causes under-utilization of the array, thereby consuming extra cycles of computation. Therefore, in the first part of our work, we develop a hyperparameter-tuning method called Array Aware Training which is dependent on the array size to save processing cycles during training operation.

Large models with millions of parameters can guarantee good accuracy but at the higher cost of computation [7]. Pruning parameters of DNN accompanied by retraining can remove inherent redundancy, and reduce the size of the network. The magnitude-based pruning method [8], [9] is very effective in compressing the DNN model size significantly. However, this pruning is irregular in nature and therefore introduces considerable sparsity in the matrices. Since the systolic array design in TPU does not support sparse matrix implementations, irregular pruning cannot be leveraged properly on such architectures. In fact, this non-uniform pruning may not change the execution time of the naive network.

Node pruning techniques [10]–[12] can resolve the bottlenecks created by irregular pruning and can be efficiently implemented on CPU and GPU devices. However, these pruning methods do not take the array size of the accelerator array into consideration leading to a mismatch between network size and array size. In the second part of our paper, we develop a pruning algorithm viz. Array Aware Pruning based on the size of the systolic array size for efficient processing of the forward path of the network during inference. Some researchers [13]–[15] focused on designing specialized accelerators on top of compressed models or sparse matrices. We aim to compress the DNN model as per the array size to fit the matrices within the bounds of computing hardware. Our main contributions are as follows.

1) Array Aware Training (AAT): We propose a hyper-parameter tuning algorithm to save training time, which is cognizant of the underlying shape and size of the array.

2) Array Aware Pruning (AAP): We propose a pruning followed by a fine-tuning method with respect to the size and shape of the array in order to save processing cycles on the array during inference.

The rest of the paper is organized as follows. Section II provides a brief background on DNNs and pruning. In Section III, we discuss the different array architecture sizes along with the mapping policy of Dense and Conv. weights. While motivating the research problem in Section IV, Array Aware Training and Pruning are studied in Sections V and VI, respectively. The comparison of our proposed method with a well-known algorithm is studied in Section VII followed by Scalability Analysis in Section VIII and Conclusion in Section IX.

II. Background

A. Dense and Convolutional Layers

In a Fully Connected (FC), every neuron in one layer is connected to every other neuron in its adjacent layer. A typical Convolutional Neural Network (CNN) is a collection of multiple convolutional (Conv.) layers cascaded one-by-one, followed by an FC layer(s) [16]. Each Conv. layer is composed of multiple filters where each filter is a 2-D or a 3-D kernel that slides (convolves) over the input feature map (input activation) to produce the output feature map (output activation) [17]. The convolutional weights are used multiple times to generate the output feature maps, whereas the dense matrices are used only once to produce the output activations.

B. Pruning Deep Neural Networks

Pruning is an efficient method for inference as it removes the internal redundancy in the network, thereby making it smaller and memory-efficient [8]–[10], [18]. Every parameter that is chosen to be pruned in the weight matrix of a trained model is based on a predefined criteria. The resultant pruned matrix is retrained to retrieve the accuracy by forcing the pruned weights to remain at zero. This forcing of weights to remain at zero can be done either before updating weights or after backpropagation. A pruning mask matrix equal to the size of the weight matrix is created where every location is either 1, if the weight is not pruned, or 0 if pruned. The mask matrix is either multiplied with the gradient matrix before updating the model or with the updated weight matrix to achieve the pruning purpose.

C. Systolic Arrays and Dataflow

A systolic array is composed of several Processing Elements (PEs), organized as a 2-D grid, as shown in Fig. 1a. The weights and input activations are fed to the device through the respective memories attached to the array. Each PE is connected to a small set of its adjacent PEs that computes the product of the inputs followed by the addition of partial sum provided by the neighbouring PE in every clock cycle. Every column (i) in the array is operated in parallel which is delayed by one clock cycle with respect to column (i-1). Hence, 2N clock cycles are required to compute all the output partial sums over the entire array.

III. Hardware Architecture

In this section, we discuss the specifications of the array processors present in different versions of TPU [19] and Eyeriss, which acts as reference hardware in this paper. This paper predominantly focuses on a narrow set of architectures, i.e., TPUs and Eyeriss, as they are widely deployed and successful Neural Processing Unit (NPU) till date. Along with the architectural description, we also discuss the mapping policy, i.e. scheduling the weight matrices onto these arrays.

A. Hardware Specifications

The first version of TPU [4] has one systolic array of size 256*256 which is capable of producing 256 partial sums in a single computation cycle. The second version published by Google [19] has one systolic array of size 128*128 on every core. For simplicity, only one chip and one core are considered in this paper. The third version doubled the number of systolic arrays but retained the size of the array (128*128) from the previous generation, which is equivalent of one array with size (128*256). The first version of Eyeriss has one array with the dimensions of (12*14). Although our initial experiments are based on the sizes officially released by Google and MIT, we experimented our methods with varying sizes to check the scope of the proposed optimization algorithms.

B. Mapping Policy

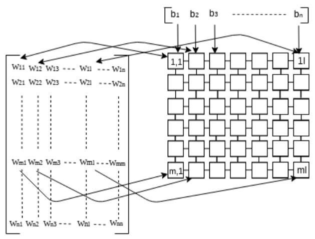
The mapping policy refers to identifying the physical location of an individual element in the dense or convolution matrix on the actual hardware. The TPU operates on Weight Stationary (WS) dataflow, where the weights are stored in the local register file, and the input activations are passed through the array in successive cycles. This kind of array size and dataflow is very well-suited for a Fully Connected (FC) layer which has a regular 2-D matrix multiplication. The Eyeriss hardware relies on Row Stationary (RS) aims to maximize the reuse and accumulation in the register for weights, input activation and partial sums. This kind of small array architecture and dataflow are optimized for Convolutional layers. Hence, in our experiments, we accelerate and design pruning algorithms for Dense networks on TPU and CNNs on Eyeriss. Initially, the mapping policy for one NxN array is studied and can be extended for a generic k NxN array.

1) NxN Array: We first discuss the policy for a Fully Connected layer and then develop a method for transforming convolutional kernels into a dense structure.

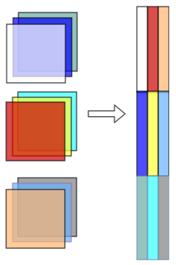
i) *Dense (FC) Matrix*: The row (physical_x) and column (physical_y) indices on the array are determined from the actual indices (i, j) of the matrix as:

$$\text{physical\_x}(i) = i\%N \text{ and } \text{physical\_y}(j) = j\%M$$

where (N*M) is the shape of the array. The bias parameter of the node (neuron) is assigned to the first MAC unit of the respective column. The complete procedure of mapping FC matrix on a single array is illustrated in Fig. 1a.



(a) Mapping 2D Matrix on Array



(b) Linearization of Conv. Matrices

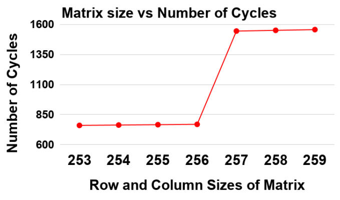
Fig. 1: Weight Mapping

ii) *Convolutional Matrix*: The weights in the Conv. layer neither have the conventional 2-D matrix structure, which is suitable to be mapped on a 2-D systolic array nor do they posses the traditional matrix multiplication operation. Hence, we propose a data shaping approach to convert filters into a dense-like matrix (2-D matrix). Consider K set of Conv. Filters with dimensions (M, N, C, K) where C represents the number of channels and (M, N, C) is the 3-D representation of each filter. The Conv. filters are converted to a regular matrix of dimensions (M*N*C, K), as shown in Fig. 1b. The weights of every individual channel contributing to one output pixel in the output feature map reside in a single column. By this transformation, the j^{th} output feature map is computed along the $(j\%N)^{th}$ column of the array. The Dense matrix mapping policy is applied after the elongation of kernel weights. The input and output activations are also linearized along with the weight matrices of the Conv. layer.

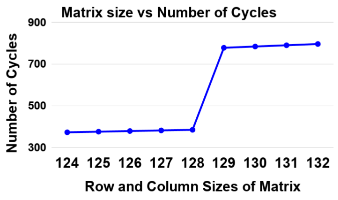
2) k NxN Arrays: The weight mapping algorithm is similar to one NxN array except for the division of workload between the symmetric processors. For k processors, an FC or conv. matrix is divided approximately into k parts along the column, such that they are decomposed into k sub-matrices and stored in the same memory device. Each sub-matrix is accelerated on an individual array which follows the same strategy as described earlier on its respective array. This way of splitting matrices avoids communication among the arrays since the output activations are computed along the column.

IV. The Problem

In this section, we examine the processing cycles (computation + data movement cycles) of weight matrices whose size is close to the multiples of array size to demonstrate the need of our proposed methods. In our preliminary experimentation, we performed acceleration of varying sizes of FC matrices and Conv. weights using an open-source cycle accurate DNN-based systolic array simulator, SCALE-sim [20], which captures the combined effect of data movement and compute cycles. We focused on FC matrix sizes (row and column) around 256 on TPU Version 1 (256*256) and 128 on Version 2 (128*128) using weight stationary dataflow scheme. From the results depicted in Fig. 2, we observe that the number of processing cycles almost doubled when the weight matrix exceeds the array size even by one row/column. A significant jump in the number of cycles can be observed at matrix size of 257 and 129 on Version 1 and 2 respectively. This trend continues for all the weight matrices around the multiples of array size.



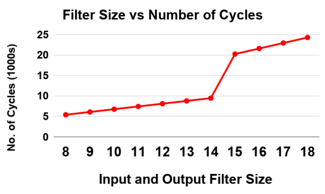
(a) Varying Matrix sizes on V1



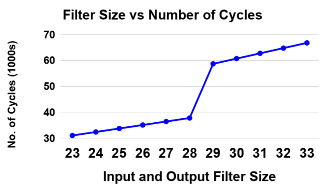
(b) Varying Matrix sizes on V2

Fig. 2: Variable Matrix Sizes on TPU Versions 1 and 2 (Sizes: 256*256, 128*128)

We also capture the processing cycles of a (3, 3, x, x) convolutional filter on a (32, 32, x) input feature map where x indicates the varying filter size on Eyeriss Version 1 (14*12 array), as illustrated in Fig. 3. As the simulator does not support Row Stationary strategy, we relax the dataflow of Eyeriss to Output Stationary throughout the paper. A surge is observed clearly in the processing cycles by having even one extra Filter beyond the size of the array, which is of interest to us. Our goal in this paper is to optimize the DNN architecture to bring it down to the multiples of array size to save processing cycles.



(a) Varying Filter sizes around 14



(b) Varying Filter sizes around 28

Fig. 3: Variable Filter Sizes on Eyeriss (Size: 14*12)

V. Array Aware Training (AAT) for efficient Training

The initial version of TPU [4] supported only the inference phase of DNNs. However, realizing the importance of DNN training, later architectures [19] started offering support for training. The training process begins with initializing random weights and iterating several times until the global minimum is obtained. Hence, it is very difficult to determine the redundant parameters when the model isn't trained. However, a simple process of optimizing the hyperparameters or randomly initializing them as per the size of the array can save a significant number of processing cycles during training operation. In order to accomplish this, we propose Array Aware Training, where the main idea is to train the networks from scratch based on the size of the underlying array. The inputs to this algorithm are the network parameters (original Hyperparameters) of the model, along with the dimensions (row and column) of individual systolic array. The output is a trained network where the dimensions of the weight matrices are the nearest (floor) multiples of the row and column sizes of the array.

For an (X*Y) matrix that is mapped on an (M*N) array, the number of computation cycles is calculated by considering k*j where k = $\lceil X/M \rceil$ and j = $\lceil (Y/N) \rceil$. If X and Y are not exactly divisible by M and N respectively, computing cycles can be saved by using an X' = $\lfloor X/M \rfloor$*M and Y' = $\lfloor Y/N \rfloor$*N sized weight matrix and eliminating the rest of the columns and rows. Hence, instead of (k x j) cycles of computation, the array processing happens only in (k-1) x (j-1) cycles saving (k+j-1) cycles. In the following subsections, we explain the number of parameters to be chosen to train in each layer as per the algorithm.

1) Choosing Weights in FC Layer: For a symmetric array, the same number of weights are processed in a single cycle along both directions. Also, the number of outputs in the previous layer determines the input connections to the next layer. Hence, modifying the column size (number of neurons) in the (i-1)th layer FC weight matrix as per the requirement (nearest integer multiple of the array) results in the adjustment of the number of rows in the ith layer FC matrix. However, for more than one array with the same size, the effective number of rows and columns becomes N and k*N respectively. A mismatch could happen if the number of columns in (i-1)th layer and the number of rows in ith layer are chosen to be nearest multiples of k*N, N, respectively. Thus, the number of neurons in every layer is chosen to be closest (floor) integer multiple of the number of columns (k*N) across all the arrays to satisfy the requirement of matrix multiplication.

2) Choosing Parameters in Conv. Layer: In this context, we consider the relationship between the given number of input convolutional kernels to the optimized number of kernels to be chosen during the training process. As per our kernel mapping policy, the weights of one convolutional channel (2D or 3D) of a layer lies along one entire column, and different channels utilize different columns. For a conv. layer with dimensions (M$_i$, K$_i$, L$_i$, C$_i$) in ith layer and (M$_{i-1}$, K$_{i-1}$, L$_{i-1}$, C$_{i-1}$) in (i-1)th layer, size of each channel (L$_i$) in ith layer is equal to total number of channels (C$_{i-1}$) in (i-1)th layer. Therefore, initially we choose the number of channels (L'$_{i-1}$) in the (i-1)th layer to be the closest integer multiple of the number of processing columns (k*N) as per Eq. 1.

Algorithm 1 Array Aware Training

```
Input: Hyperparameters, No. of Arrays, Size of the Array
Output: Trained Neural Network
0: $H$: Set of Hyperparameters
0: $N$: Size of each Array, $k$: Number of Arrays
0: procedure TRAINING($H$, $N$, $k$)
0:     for i ← 1 to Total Number of Layers do
0:         if Operation == Conv. then
0:             w$_c$ ← number_of_channels(H[i])
0:             q$_c$ ← $\lfloor w_c/(N*k) \rfloor$
0:             if q$_c$ > 1 then
0:                 number_of_channels(H[i]) ← $q_c$*N*k
0:             end if
0:         end if
0:         if Operation == Dense then
0:             w$_c$ ← number_of_neurons(H[i])
0:             q$_c$ ← $\lfloor w_c/(N*k) \rfloor$
0:             if q$_c$ > 1 then
0:                 number_of_neurons(H[i]) ← q$_c$*N*k
0:             end if
0:         end if
0:     end for
0:     Train the weights based on updated Hyperparameters
0:     return $Weights$
0: end procedure=0
```

$$L' = \lfloor L/(k*N) \rfloor * k * N \tag{1}$$

Thus, reducing the number of channels in the (i-1)th layer will reduce the size of each channel in ith layer. Then, the number of channels (L"$_{i-1}$) in (i-1)th layer are further optimized based on a single channel dimension (M, K, L) in ith layer as per Eq. 2.

$$L'' = \lfloor (M_i\text{*K}_i\text{*L}_i)/\text{N} \rfloor * N \tag{2}$$

Hence, Intra-Kernel (across different channels of CNN) optimization minimizes the computation cycles across the columns, whereas Inter-Kernel (within the same channel) optimization minimizes the computation cycles across the rows. This kind of reduction applies to both symmetric and non-symmetric arrays. We applied the proposed method on several well-known networks from [21] on MNIST, ConvNet [22], Network-in-Network [7], AlexNet [1] and Vgg16 [23]. We named networks in [21] to be MNIST-1 (1000-500-10), MNIST-2 (1500-1000-500-10), MNIST-3 (2000-1500-1000-500-10) and MNIST-4 (2500-2000-1500-1000-500-10) where numbers indicate the number of neurons in each layer. Tensorflow [24] has been used to train the DNNs on Nvidia Tesla V100 GPU cards which reside on our high performance computing cluster.

TABLE I: Speedup of Dense Networks on TPU

Network	Version 1	Version 2	Version 3
MNIST-1	1.77	1.30	1.43
MNIST-2	1.56	1.22	1.15
MNIST-3	1.41	1.17	1.15
MNIST-4	1.32	1.13	1.09

Tables I and II illustrate the speedup while using AAT method on MLP networks on TPU and CNNs on Eyeriss with a negligible drop in classification accuracy. The performance enhancement is under the assumption that the first version of TPU [4] and Eyeriss [5] architecture support training (the official papers mention these hardware units as an inference engine).

TABLE II: Performance Improvement of CNNs on Eyeriss

Network	Speedup	% of Cycle Reduction
ConvNet	1.47	32%
NiN	1.18	15%
AlexNet	1.19	16%
Vgg16	1.10	9%

Large networks yield smaller performance improvement compared to smaller networks for the same array size, although the total number of cycles saved is more than the cycles saved in the case of small networks. For MNIST-1 network, 7,567 cycles are saved for a single input sample on a 256*256 array. If the array is used as a training device, in the course of one epoch, 378.3 million processing cycles can be saved for 50,000 data samples, thereby saving 3.78 billion cycles throughout the training process (if the number of epochs = 10), resulting in 43.69% improvement. Therefore, the Array Aware Training technique is very effective.

VI. Array Aware Pruning (AAP) for Efficient Inference

In the previous AAT method, the weight matrices were initialized and trained based on the size of the systolic array. Even though the AAT method saves a significant number of cycles during training, its scope of improvement during inference is minimal as we are removing only one batch among multiple number of batches. Also, if a DNN model is originally trained on a different sized array or hardware (Eg. GPU), the weight matrices may not be a perfect multiple of the array which is currently being used for inference. To overcome the difficulties of during inference, we propose ”Array Aware Pruning (AAP)” to prune multiple batches being processed on the array. As the proposed optimization algorithm is systolic-dimension-dependent, the inputs are trained weight matrices along with the shape of every array available. As the AAP algorithm relies on node pruning of DNNs, the first step is to understand the optimization mechanism to prune redundant nodes in the network. We use L-1 norm (similar to the one in [25]) of an individual Filter (Neuron) as a criterion to prune the redundant parameters in the network. The routine for the L-1 norm-based symmetric pruning is given in Algorithm 2.

Algorithm 2 Pruning

```
Input: Weights, Number of Nodes to be Pruned
Output: Prune Mask
0: $W$: Weights,
0: $P$: Number of Nodes to be Pruned, $Mask$: Prune Mask
0: procedure PRUNING($W$, $P$, $Mask$)
0:     $S_i \leftarrow$ Sum of absolute weights of Filter (Neuron) i
0:     Prune P Neurons which have least magnitude in $W$
0:     $Mask \leftarrow$ Update the Prune Mask
0:     return $W$, $Mask$
0: end procedure=0
```

The function calculates the significance of the node in each layer by measuring its sum of absolute weights. As smaller network parameters tend to exhibit less importance in the case of [9], the nodes with relatively smaller or low L-1 norms demonstrate weak values in the output activations. The Pruning function (Algorithm 2) accepts Weight (W) matrices and Number of pruning nodes (P) as inputs, and prunes P filters/neurons which have the least magnitude. The overall Array Aware Pruning method (Algorithm 3) works in the following manner:

1) The first step in the AAP algorithm converts every weight matrix in the network to be perfect multiples of the row and column sizes of the systolic array. This step is performed as the trained model may not be in accordance with the array size. Hence, we calculate the number of nodes (P_1) to be pruned in each layer such that the dimensions of the weight matrix are the nearest (floor) multiples of the array (N). Prune P_1 nodes in the corresponding layer based on the procedure mentioned in the Pruning routine and fine-tune the model to recover the accuracy.

2) At the beginning of this step, the dimensions of every matrix in the model should be a perfect multiple of the underlying array size. In the next stage, calculate the total number of batches (B_i = Number of Nodes (w_n)/Array Size (N)) required to fit the nodes of layer i.

3) In any pruning algorithm, the weights or nodes have to be removed iteratively followed by a retraining step. Pruning a large number of parameters at once causes the model to lose its accuracy. Hence, in every pruning iteration i, (i*x)% (provided by the user; we consider x = 5) of the total number of batches are pruned in each layer which is indicated by B_x = round((i*x*B_i)/100). The effective number of nodes to be pruned (P_2) is given by

$$P_2 = (B_i - B_x) * N \tag{3}$$

4) Repeat this iterative pruning operation followed by re-training the entire network until a significant accuracy loss is

incurred. The output of this algorithm is a pruned model with accuracy being close to the naive network.

Algorithm 3 Array-Aware Pruning

```
Input: Trained Model, Size of the Array
Output: Pruned Neural Network
0: $W$: Trained Weights, $N$: Size of the Array
0: procedure ARRAY AWARE PRUNING($W$, $N$, $x$)
0:   for i $\leftarrow$ 1 to Total Number of Layers do
0:     Mask[i] $\leftarrow$ Ones(Shape of the layer i)
0:   end for
0:   for i $\leftarrow$ 1 to Total Number of Layers do
0:     $w_n$ $\leftarrow$ Number of Nodes in layer i
0:     $P_1$ $\leftarrow$ $w_n$%N
0:     W[i], Mask[i] $\leftarrow$ Pruning(W[i], $P_1$, Mask[i])
0:     Number of Nodes in layer i $\leftarrow$ $w_n$ - $P_1$
0:     Retrain the Network based on Prune Mask
0:   end for
0:   for i $\leftarrow$ 1 to Total Number of Layers do
0:     $w_n$ $\leftarrow$ Number of Nodes in layer i
0:     $B_i$ $\leftarrow$ $w_n$/N
0:   end for
0:   for iter in [1, 2, 3, 4....] do
0:     for i $\leftarrow$ 1 to Total Number of Layers do
0:       $B_x$ $\leftarrow$ round((iter*x*$B_i$)/100)
0:       $P_2$ $\leftarrow$ ($B_i$ - $B_x$)*N
0:       W[i], Mask[i] $\leftarrow$ Pruning(W[i], $P_2$, Mask[i])
0:       Retrain the Network based on Prune Mask
0:     end for
0:     if Significant Accuracy Loss then
0:       break
0:     end if
0:   end for
0:   return Pruned Weights
0: end procedure=0
```

In the following discussion, we analyze the pruning patterns of dense and convolutional layers on different versions. We pool all the systolic arrays with k=1 into one category and study them together as the hardware is symmetric along their rows and columns. The weight matrices are set in such a way that row size corresponds to the number of neurons in the previous layer, and the column size corresponds to the number of neurons present in the current layer. Hence, the column size of (i-1)th layer matrix should match the row size in the ith layer matrix to execute the matrix multiplication operation. This theory stands as a basis for structured pruning of weight matrices based on the values of k and N.

NxN Array: Due to the inherent symmetrical nature of these versions along their axes, pruning any column in the (i-1)th layer FC matrix will naturally remove their respective row in the ith dense layer from the computation. Therefore, the network still retains the fully-connected structure. Hence, the number of batches to be pruned along the row and column of a matrix are calculated according to the value of N (Array size). In the case of CNNs, pruning any channel in (i-1)th layer removes the corresponding filter across all the channels in ith layer.

k NxN Arrays: The operating row (N) and column (k*N) sizes do not match with each other in the case of k NxN arrays. Also, pruning columns of (i-1)th dense layer matrix w.r.t N*k and rows of ith layer dense matrix w.r.t N leads to a mismatch while performing matrix multiplication. For this reason, the indices beyond which parameters have to be pruned along both directions are determined based on k*N as any number divisible by k*N is also divisible by N. For this reason, the network still has a fully connected nature, and matrices are still in the form of a regular 2-D matrix in all the versions, which are suitable on a uniform array. The same strategy applied to CNN weights in the case of one NxN array is applied here where the channels of convolutional layers are pruned with respect to k*N.

TABLE III: Speedup using AAP method on Dense networks

Network	Version 1	Version 2	Version 3
MNIST-1	4.4	10.4	7.1
MNIST-2	8.6	22.3	15.1
MNIST-3	15	41.2	27.1
MNIST-4	23.7	67.9	45.4

The speedup of Dense networks on different versions of TPU is reported in Table III. Unlike the AAT method where large models are barely impacted, AAP algorithm significantly improves the performance of large networks. The speedup of the number of cycles saved for various CNNs on Eyeriss hardware is demonstrated in Table IV. The speedup and cycles saved are with respect to one input image over a single iteration.

TABLE IV: Performance Improvement of CNNs on Eyeriss

Network	Cycles Saved	Speedup	% of Cycle Reduction
ConvNet	188976	6.33	84%
NiN	793946	4.31	76%
AlexNet	120588524	3.7	74%
Vgg16	85665522	5.5	81%

VII. Comparison with other methods

Irregular Pruning: The magnitude-based non-uniform pruning method proposed in [8], [9] is very effective in compressing the DNN model size significantly. However, the unpruned weights are mapped on the same locations according to the mapping strategy as discussed above with zeros being mapped on the PEs at the pruned locations. In this case, even if one weight exists in the last computing batch, the device takes the same number of cycles to process the outputs corresponding to those rows or columns. The uniform systolic array design in TPU does not support sparse matrix implementations and incurs irregular memory accesses. Hence, this kind of pruning cannot be leveraged effectively on such array architectures.

Regular Pruning: As our method falls under the category of “Structured Pruning”, it is important to compare our algorithm with the well-known node pruning techniques. We consider the example of Scalpel pruning which resolved the bottlenecks of irregular pruning. This method does not considers the size of the array while pruning. Even if one node is present in the last batch, the array still consumes one whole computation cycle. We make sure that every layer is reduced in terms of nodes to utilize the array effectively.

Network-in-Network: We demonstrate the advantage of our AAP technique by considering the example of NiN which has a Filter architecture as mentioned in the first row of Table V. The Pruning algorithm [10] does not impose any criteria on any layer and converts the naive model into a pruned version as per the second row of Table V. If this model is accelerated on Eyeriss architecture which has an array size of 14*12, the equivalent network size is given as per the third row in the table. This is because the number of nodes is not in synchronous with the array length causing the last computation batch of the matrix to be under-utilized, which is almost equal to the fully occupied array. Also, it is evident from the layer 4 of Traditional pruning that no performance benefit is obtained because of the equivalent conversion.

TABLE V: Number of Nodes Remaining after Pruning NiN

Method/Layer	1	2	3	4	5	6	7	8
Naive Network	192	160	96	192	192	192	192	192
Traditional Pruning [10]	117	81	62	183	123	148	91	54
Equivalent on Eyeriss	120	84	72	192	132	156	96	60
AAP	108	84	48	108	108	108	108	108

However, by employing our AAP method, where we enforce strict criteria on every layer to reduce its size, we can clearly notice that all the weights are pruned with respect to array size. Hence, performance benefit can be obtained from all the layers except the output layer. The speedup of the AAP method on ConvNet, NiN and AlexNet networks are compared with [10] in Table VI. It can be clearly noticed that pruning as per array size is beneficial over the traditional node pruning technique.

TABLE VI: Speedup of Various Networks on Eyeriss

Network	Traditional Pruning	AAP
ConvNet	1.7	6.33
NiN	1.9	4.31
AlexNet	1.3	3.7

VIII. Scalability Analysis

In this section, we discuss the scalability of the proposed AAP algorithm on different sizes of the systolic array to check the extent of this method. In our initial experiments, we considered only the array sizes reported in TPU and Eyeriss papers with the assumption that these hardware architectures are already optimized for different DNNs. However, in this study, we monitor the effects of changing the array size in TPU-like and Eyeriss-like device settings. Along with this study, we also consider the optimal array size to balance the pruning and execution time on the hardware.

TPU: We examined the performance of the naive unoptimized network, baseline pruned model, and our optimized MLP network after applying our proposed AAP method over different array sizes of 32*32, 64*64, 128*128 (TPU Version 2), 128*256 (TPU Version 3) and 256*256 (TPU Version 1). It is obvious from Fig. 4 (in log scale) that the naive network and baseline node pruned model requires more processing cycles on small array sizes like 32*32 as compared to a large array like 256*256. This is due to the fact that huge network size in the case of original networks benefits from large array sizes. Also, traditional node pruned networks are not in accordance with the shape of the array leading to extra cycles along row and column. However, with the AAP algorithm in place, the weight matrices are pruned as per the underlying array reducing the additional cycles caused due to mismatch of the array and network size. From the graphs of AAP, it can be noted that 128*128 and 128*256 sized arrays show better performance than 256*256 array in terms of execution time. The baseline technique on 256*256 array and AAP method on 128*128 array deliver the same performance in terms of the number of cycles. Hence, MLPs can be accelerated on 128*128 array with AAP technique compared to 256*256 array with [10], thereby decreasing the area of the matrix multiplication unit by four times.

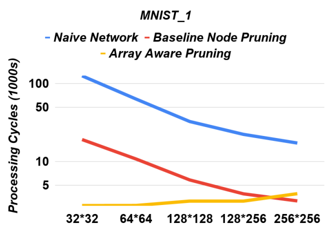
Eyeriss: As the CNNs are optimized for small array sizes and output stationary dataflow, we assess the performance on dimensions starting from 6*6, 7*7 to 14*14. In the majority of the cases, the execution time decreases with an increase in the size of the array. It is evident from Fig. 5 (in log scale) that our proposed AAP solution benefits over traditional methods over a wide range of array shapes. Different CNNs are optimized for different sizes of the array. ConvNet and Vgg16 are optimized at array dimensions of 12*12 while NiN and AlexNet deliver their best performance at 14*14.

IX. Conclusion

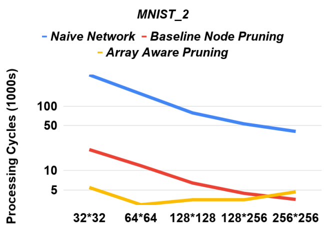
We addressed the problem of optimizing the size of DNN weight matrices to suit the hardware specifications for efficient forward propagation on array-based neural network accelerators. We initially proposed Array Aware Training (AAT) to remove the last computation batch from the network and train from scratch. This AAT method enhances training time and provides up to 1.5 times speedup on the selected networks. We also propose Array Aware Pruning for efficient inference, which prunes nodes based on the size of the array architecture. This method provides significant speedup during inference as low as 3.7 times on the benchmarks used in the experiments. We also compared our algorithm with a well-known node pruning method and demonstrated that it works for a wide range of array sizes.

Acknowledgements

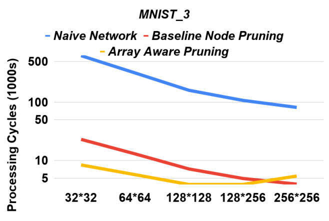
The research reported in this paper is supported by the Philip and Virginia Sproul Professorship at Iowa State University.



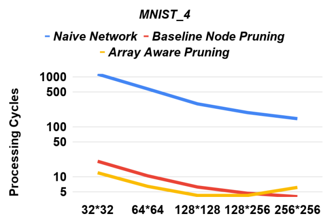
(a) MNIST-1



(b) MNIST-2

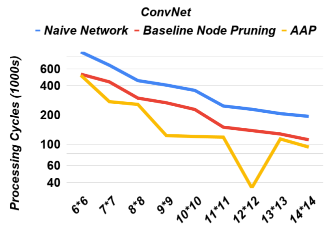


(c) MNIST-3

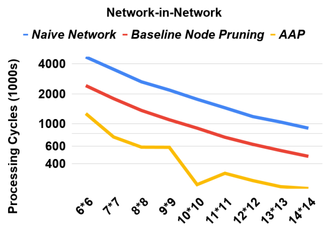


(d) MNIST-4

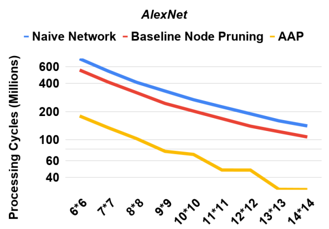
Fig. 4: Dense Networks on Varying Array Sizes of TPU



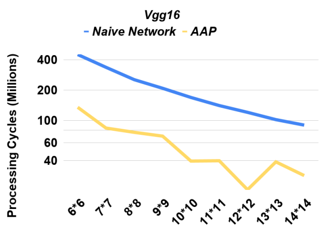
(a) ConvNet



(b) NiN



(c) AlexNet



(d) Vgg16

Fig. 5: CNNs Networks on Varying Array Sizes of Eyeriss Architecture

Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s).

References

[1] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in neural information processing systems*, 2012, pp. 1097–1105.

[2] T. Chen, Z. Du, N. Sun, J. Wang, C. Wu, Y. Chen, and O. Temam, "Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning," *ACM Sigplan Notices*, vol. 49, no. 4, pp. 269–284, 2014.

[3] W. Lu, G. Yan, J. Li, S. Gong, Y. Han, and X. Li, "Flexflow: A flexible dataflow accelerator architecture for convolutional neural networks," in *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2017, pp. 553–564.

[4] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers *et al.*, "In-datacenter performance analysis of a tensor processing unit," in *Computer Architecture (ISCA), 2017 ACM/IEEE 44th Annual International Symposium on*. IEEE, 2017, pp. 1–12.

[5] Y.-H. Chen, J. Emer, and V. Sze, "Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks," in *ACM SIGARCH Computer Architecture News*, vol. 44, no. 3. IEEE Press, 2016, pp. 367–379.

[6] H.-T. Kung, "Why systolic architectures?" *IEEE computer*, vol. 15, no. 1, pp. 37–46, 1982.

[7] M. Lin, Q. Chen, and S. Yan, "Network in network," *arXiv preprint arXiv:1312.4400*, 2013.

[8] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding," *arXiv preprint arXiv:1510.00149*, 2015.

[9] S. Han, J. Pool, J. Tran, and W. Dally, "Learning both weights and connections for efficient neural network," in *Advances in neural information processing systems*, 2015, pp. 1135–1143.

[10] J. Yu, A. Lukefahr, D. Palframan, G. Dasika, R. Das, and S. Mahlke, "Scalpel: Customizing dnn pruning to the underlying hardware parallelism," in *ACM SIGARCH Computer Architecture News*, vol. 45, no. 2. ACM, 2017, pp. 548–560.

[11] P. Molchanov, S. Tyree, T. Karras, T. Aila, and J. Kautz, "Pruning convolutional neural networks for resource efficient inference," *arXiv preprint arXiv:1611.06440*, 2016.

[12] S. Anwar, K. Hwang, and W. Sung, "Structured pruning of deep convolutional neural networks," *ACM Journal on Emerging Technologies in Computing Systems (JETC)*, vol. 13, no. 3, pp. 1–18, 2017.

[13] A. Parashar, M. Rhu, A. Mukkara, A. Puglielli, R. Venkatesan, B. Khailany, J. Emer, S. W. Keckler, and W. J. Dally, "Scnn: An accelerator for compressed-sparse convolutional neural networks," in *ACM SIGARCH Computer Architecture News*, vol. 45, no. 2. ACM, 2017, pp. 27–40.

[14] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally, "Eie: efficient inference engine on compressed deep neural network," in *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2016, pp. 243–254.

[15] S. Zhang, Z. Du, L. Zhang, H. Lan, S. Liu, L. Li, Q. Guo, T. Chen, and Y. Chen, "Cambricon-x: An accelerator for sparse neural networks," in *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Press, 2016, p. 20.

[16] T. N. Sainath, A.-r. Mohamed, B. Kingsbury, and B. Ramabhadran, "Deep convolutional neural networks for lvcsr," in *2013 IEEE international conference on acoustics, speech and signal processing*. IEEE, 2013, pp. 8614–8618.

[17] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2015, pp. 1–9.

[18] Y. LeCun, J. S. Denker, and S. A. Solla, "Optimal brain damage," in *Advances in neural information processing systems*, 1990, pp. 598–605.

[19] "Versions of tpu," https://cloud.google.com/tpu/docs/system-architecture.

[20] A. Samajdar, Y. Zhu, P. Whatmough, M. Mattina, and T. Krishna, "Scale-sim: Systolic cnn accelerator," *arXiv preprint arXiv:1811.02883*, 2018.

[21] D. C. Cireşan, U. Meier, L. M. Gambardella, and J. Schmidhuber, "Deep big multilayer perceptrons for digit recognition," in *Neural networks: tricks of the trade*. Springer, 2012, pp. 581–598.

[22] A. Krizhevsky, "cuda-convnet," https://code.google.com/archive/p/cuda-convnet/, 2012.

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

[24] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard *et al.*, "Tensorflow: a system for large-scale machine learning." in *OSDI*, vol. 16, 2016, pp. 265–283.

[25] H. Li, A. Kadav, I. Durdanovic, H. Samet, and H. P. Graf, "Pruning filters for efficient convnets," *arXiv preprint arXiv:1608.08710*, 2016.