Eager Pruning: Algorithm and Architecture Support for Fast Training of Deep Neural Networks

Jiaqi Zhang, Xiangru Chen, Mingcong Song, Tao Li
Department of Electrical and Computer Engineering, University of Florida, Gainesville, FL, USA
{jiaqizhang, cxr1994816}@ufl.edu, songmingcong@gmail.com, taoli@ece.ufl.edu

***Abstract*—Today's big and fast data and the changing circumstance require fast training of Deep Neural Networks (DNN) in various applications. However, training a DNN with tons of parameters involves intensive computation. Enlightened by the fact that redundancy exists in DNNs and the observation that the ranking of the significance of the weights changes slightly during training, we propose Eager Pruning, which speeds up DNN training by moving pruning to an early stage.**

Eager Pruning is supported by an algorithm and architecture co-design. The proposed algorithm dictates the architecture to identify and prune insignificant weights during training without accuracy loss. A novel architecture is designed to transform the reduced training computation into performance improvement. Our proposed Eager Pruning system gains an average of 1.91x speedup over state-of-the-art hardware accelerator and 6.31x energy-efficiency over Nvidia GPUs.

***Keywords*—Neural Network Training; Neural Network Pruning; Software-Hardware Co-Design**

1. Introduction

This is an era of big data and artificial intelligence. On one hand, the ubiquitous cameras, sensors and other data collectors provide access to extremely large datasets[1], enabling the machines to learn models even more accurate than human beings[2]. On the other hand, the tremendous data collection is valuable only when analyzed by deep and complex AI models. Among all the AI technologies, DNN is one of the most popular and promising[3]–[5].

Lots of DNN applications require lifetime learning to keep the models up to date. For example, the tweets sent by users are used to learn the real-time topics. The behaviors during Google search teach the engine to provide more precise results. Note that today's data is not only big but also fast[6].

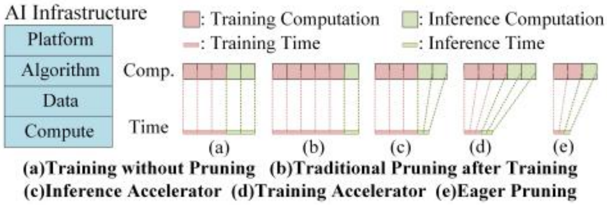


Figure 1. AI Infrastructure

6,000 tweets are sent and 40,000 Google search queries are processed every second[7]. To fully utilize the data, the speed of model adjustment should cater to that of data generation.

A sudden change in the circumstance is another situation where fast retraining is necessary. Take autonomous animal disease control as an example. When a new disease appears, the system needs to be retrained as soon as possible on those symptoms related to the new disease to identify the infected animals. A delay of retraining may lead to an explosion of the disease. Also consider the case where a self-driving car goes to a new place, where the traffic signs and surroundings look totally different. The car's recognition model has to make quick adjustments to prevent accidents.

Nevertheless, DNNs comprise a large quantity of parameters. To train them, three computation phases, i.e. forward propagation, backpropagation and kernel update, are repeated millions of times, making it both time and energy consuming. As the networks[8], [9] grow deeper for better representations, this problem gets more severe. Several works[10], [11] made efforts to speed up DNN training by extreme parallelization. However, their methods demand great computation resources like thousands of KNLs or GPUs, which is not available for general applications. In this work, we accelerate DNN training from another perspective.

Figure 1 shows the infrastructure of an AI computing stack[12]. The algorithm layer determines the required computation for training the networks while the compute layer determines the time consumed on the computation. This work accelerates DNN training by exploiting an algorithm and architecture co-design to reduce both the computation and the time spent on computation as shown in Figure 1(e).

It is widely recognized that considerable redundancy exists in DNNs. Prior works[13]–[15] remove insignificant weights from a completely trained model and retrain it. The sparse model after pruning and retraining is as accurate as the dense one. Enlightened by their success, we propose to move the pruning process to an earlier stage to reduce the computation involved during training. This is feasible because our study

ISCA '19, June 22–26, 2019, AUSTIN, TX, USA
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-6669-4/19/06…$15.00
https://doi.org/10.1145/3307650.3322263

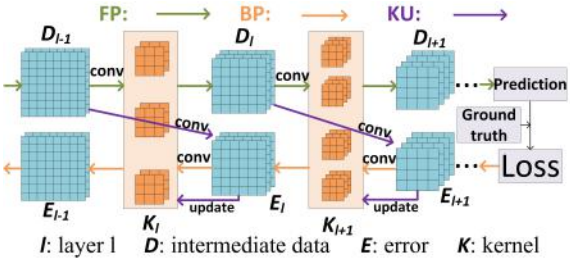
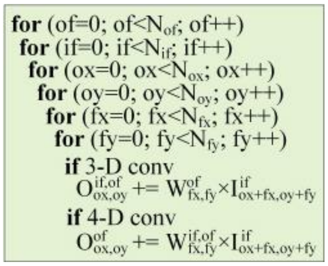


Figure 2. Three Phases in Training



```
for (of=0; of<$N_{of}$; of++)
  for (if=0; if<$N_{if}$; if++)
    for (ox=0; ox<$N_{ox}$; ox++)
      for (oy=0; oy<$N_{oy}$; oy++)
        for (fx=0; fx<$N_{fx}$; fx++)
          for (fy=0; fy<$N_{fy}$; fy++)
            if 3-D conv
              $O^{if,of}_{ox,oy}$ += $W^{of}_{fx,fy} \times I^{if}_{ox+fx,oy+fy}$
            if 4-D conv
              $O^{of}_{ox,oy}$ += $W^{if,of}_{fx,fy} \times I^{if}_{ox+fx,oy+fy}$
```

Figure 3. Pseudo Code for Convolution

Table 1. Convolution in Three Phases

$\boldsymbol{D_l}$: intermediate data of layer l, $\boldsymbol{E_l}$: error of layer l, $\boldsymbol{K_l}$: kernel of layer l, $\boldsymbol{TK_l}$: transposed kernel of layer l, $\Delta\boldsymbol{K_l}$: kernel update of layer l.

Phase	Input, filter and output of layer l			Dependency of layer l	Conv type
	In	Filter	Out		
FP	D_{l-1}	K_l	D_l	FP_{l-1}	4-D
BP	E_{l+1}	TK_{l+1}	E_l	BP_{l+1}	4-D
KU	D_{l-1}	E_l	ΔK_l	FP_{l-1}, BP_l	3-D

observes that the ranking of the significance of the weights changes slightly during training, implying if a weight is insignificant in the early iterations, it can be claimed with high confidence that this trend will hold to the end. Therefore, pruning can be applied as soon as the network is initialized with low risk. To this end, we propose Eager Pruning (EP). The algorithm for EP reduces training computation by 40% while maintaining the original accuracy.

Although the computation can be reduced by pruning during training, existing DNN accelerators are incapable of transforming this computation reduction into performance improvement. They unroll and exchange the nested loops in convolution to leverage the computation parallelism and data reuse by designing dedicated architectures with fixed unrolling scales and dataflows. However, the irregular and changeable sparsity in EP causes resource underutilization and the discrepancy in different training phases increases inefficiency of the existing accelerators.

To solve the challenges brought by EP to the traditional accelerators and fulfill its potential, we break all the fixed connections between the processing elements (PEs) so that each PE can be assigned independently. Since each kernel may have various size in EP, weights inside the kernels are distributed to different number of PEs resulting in a consecutive but irregular partition of the PEs. Besides, when the weights in PEs are replaced, the partition changes as well. To sum up the partial results within a kernel, we propose a novel Dynamically Reconfigurable Add and Collect Tree (DRACT) to collect the results. It can be configured based on the pattern of the sparse kernels loaded into PEs in real time. The EP dataflow in different training phases is also proposed.

EP is evaluated on eight popular DNNs and compared with five state-of-the-art accelerators and three Nvidia GPUs. Due to the high energy efficiency, EP brings the most value to time-sensitive applications with limited power budget.

The main contributions of our work are summarized as:

(1) We analyze the behavior of DNN weights during training and demonstrate the feasibility of Eager Pruning.

(2) We propose an algorithm that prunes DNNs during training, which on average reduces 40% training computation while maintaining the original accuracy on eight networks.

(3) We further propose and implement a DRACT-based architecture design to support Eager Pruning, which achieves an average of 1.91x speedup over state-of-the-art hardware design and 6.31x energy-efficiency over Nvidia GPUs.

The rest of this paper is organized as follows. Section 2 provides the motivation and proof of feasibility of EP. The algorithm for EP is introduced in Section 3. In Section 4, we propose the architecture support for EP. Section 5 presents the evaluation of our system. The related work is discussed in Sections 6. Section 7 concludes the paper.

2. Background and Motivation

2.1 Background on DNN Training

The kernels of neural networks are trained using the gradient descent algorithm[16]. Each iteration consists of three different phases: *forward propagation* (*FP*), *backpropagation* (*BP*) and *kernel update* (*KU*). Figure 2 shows the three phases in convolution layers. The computation in FP is the same as DNN inference. The output of a layer is convolved by the kernel to generate the output of the next layer. We call the outputs of hidden layers as *intermediate data*. The last layer outputs the classification prediction of the network. The first step in BP is to calculate the *training loss* using the prediction and the ground truth. Then this loss is backpropagated to produce the *errors*. As opposed to the intermediate data, the error of a layer is convolved with the transposed kernel to produce the error of the previous layer. Finally in KU, the kernel update of a layer is generated by convolving the intermediate data of the previous layer with the error of the current layer.

The computation in all the phases is convolution. To avoid confusion, *kernel* in this paper specially refers to the parameters of the network that we intend to learn. *Filter* is the term used in a general convolution, which is the kernel in FP, the transposed kernel in BP and the error in KU. In convolution layers, the intermediate data and errors are 3-D and the kernels and updates are 4-D. Therefore, the computation in FP and BP is 4-D convolution while KU performs 3-D convolution. Figure 3 shows the nested loops in 4-D and 3-D convolutions, where N_{of} and N_{if} represent the number of output and input feature maps. N_{ox}, N_{oy} and N_{fx}, N_{fy} specify the size of each output feature map and filter respectively. Table 1 summarizes the input, filter and output of the three phases.

Note that the fully connected layers can be viewed as a simplified convolution layer, where the size of input and output is 1. RNN is a kind of fully connected neural network that takes the previous outputs as inputs of the current step. The kernels of RNNs are trained using Backpropagation

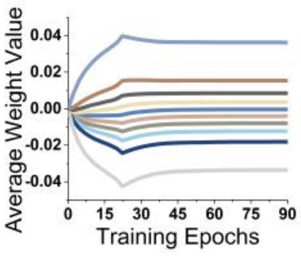


Figure 4. Weight Change

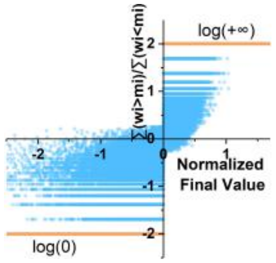


Figure 5. Crossing the Margin

Through Time[17]. In BPTT, the errors are backpropagated through time steps and the kernels are updated using all the results within a time window. By unrolling the time steps, the operations in RNNs can be transformed into the same operations in fully connected layers.

2.2 Motivation for Eager Pruning

From Figure 3, the required computation can be derived as:

$$Comp. = N_{of} \times N_{if} \times output\_size \times filter\_size \ (1).$$

The size of the kernel determines the filter size in FP and BP and the output size in KU. Therefore, all the phases will speed up if the network has fewer kernel weights.

Several works[13]–[15] demonstrate redundancy in DNNs. They remove the insignificant weights after training and then retrain the network, resulting in a compact but still accurate model. However, although the inference computation is reduced, their method requires extra efforts in training because pruning and retraining are performed after the original training (Figure 1(b)). For example, [13] spends 75 hours on training the original AlexNet and 173 hours on pruning and retraining.

Nevertheless, the idea motivates us to propose Eager Pruning (EP). If the non-contributing kernel weights can be identified earlier (i.e. as soon as training starts), training will significantly speed up. Not only is the computation for each iteration reduced, extra retraining step is also avoided as each iteration plays the role of retraining. Since there are inherently useless parameters, pruning them more eagerly does not harm the accuracy if they are correctly recognized.

2.3 Feasibility of Eager Pruning

In this work, the kernel weights with small values are viewed insignificant as in [13]. This section demonstrates that small weights are always small during the entire training process and exception happens with a very small probability.

First, the ranking of the significance of all the weights barely changes during training. Figure 4 illustrates this phenomenon when training AlexNet[18] on ImageNet[1] for 90 epochs. A snapshot is taken every two epochs. The weights are evenly divided into ten groups based on their final values. The average value of the weights in each group at each snapshot is depicted using a unique line. As shown, the ranking of the lines in Figure 4 is fixed as time elapses. This indicates that if a weight is found insignificant in early iterations, we can claim with high confidence that the trend will hold to the end.

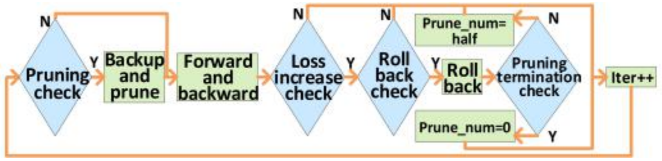


Figure 6. Eager Pruning Algorithm

Table 2. Terms Used in EP Algorithm

Parameters	
$iter$	the current iteration
$last\_rb\_iter$	the last iteration roll back performed
$smoothed\_loss$	the smoothed training loss
$last\_prune\_loss\_max$	the greatest loss of the last pruning period
$loss\_exceed\_times$	the times that the training loss exceeds
$prune\_fail\_times$	the times that roll back occurs at the same point
$prune\_num$	the quantity of weights pruned each time
Hyperparameters	
$prune\_interval$	the interval of pruning
$prune\_num_{max}$	the initial value of $prune\_num$
$over\_prune\_threshold$	the threshold to distinguish over-pruning
Condition Checks	
Pruning check	$(iter - last\_rb\_iter)$ % $prune\_interval$ ==0
Loss increase check	$smoothed\_loss > last\_prune\_loss\_max$
Roll back check	$loss\_exceed\_times > over\_prune\_threshold$
Pruning termination check	$prune\_fail\_times > 3$

Next, only few weights break the above-mentioned rule. 30000 weights randomly chosen from GoogLeNet layer inception_4d/5x5 are scattered in Figure 5. Here the smallest 50% weights are considered insignificant. The horizontal coordinate represents the final significance of the weight:

$$x_w = \log(w_f/m_f) \ (2),$$

where w_f and is the weight's final value and m_f is the median of the weights after training. The vertical coordinate indicates how often the weight appears significant during training. We count the times that each weight becomes greater and smaller than the median at each snapshot respectively and derive the vertical coordinate as:

$$y_w = \log(\sum(w_i > m_i) \,/\, \sum(w_i < m_i)) \ (3),$$

where w_i and m_i are the weight value and the median in the i^{th} snapshot respectively. In Figure 5, 96% of the weights fall into the first and third quadrants. $Log(0)$ and $log(+\infty)$ denote weights that never cross the significant-insignificant margin. Those weights in the second and fourth quadrants reside near the origin. The result suggests that only few weights close to the margin oscillate around during training.

We get similar results for all the layers of all the networks in these two experiments.

3. Algorithm for Eager Pruning

Section 2.3 proves that the insignificant weights can be identified more eagerly. This section introduces the policies and hyperparameters in EP.

3.1 Policies

The algorithm for EP is illustrated in Figure 6. Table 2 summarizes all the terms used to elaborate the algorithm.

Prune Instead of pruning all the insignificant weights at once, EP removes the smallest $prune\_num$ weights every

$prune\_interval$ iterations. From Section 3.2, it can be concluded that the smaller a weight is, the less likely it is going to cross the significant-insignificant margin. This guarantees EP always removes the weights with the highest possibility to be insignificant at the end. In addition, as shown in Figure 4, the weights stabilize as training process continues due to the decreasing learning rate[19]. Therefore, it is reasonably believed that by periodical pruning, the oscillating weights are removed with more care (pruned later when they are more stable).

Terminate pruning Pruning should be terminated at a proper time, otherwise the network will lose more parameters than desired, i.e. *over pruned*. Generally, when some weights are removed, there will be a sudden rise in the training loss and a sudden drop in the validation accuracy. To avoid extra computation for validation, we use the smoothed training loss (average of the last 100 losses) to recognize over pruning.

If the removed weights are insignificant, the loss only increases slightly and drops back quickly. On the contrary, when the loss considerably deteriorates and hardly recovers, over pruning occurs. *Pruning period* is defined as the iterations between the two adjacent pruning operations. In EP, the network is considered over pruned if the smoothed training loss appears to exceed the maximum of the last pruning period ($last\_prune\_loss\_max$) for a certain number of times in the current pruning period. $Loss\_exceed\_times$ records the times that the loss exceeds and the threshold for over pruning is $over\_prune\_threshold$. Therefore, if

$$loss\_exceed\_times > over\_prune\_threshold \quad (4)$$

is true, the network is over pruned.

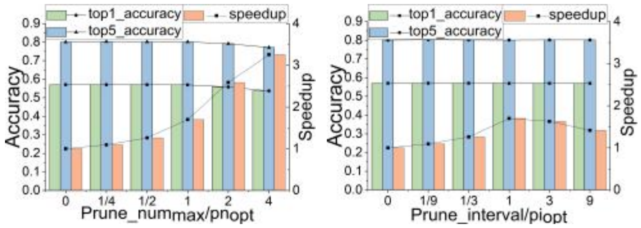
When over pruning occurs, if the incorrectly pruned weights are not added back, the network will never achieve the best accuracy. Therefore, EP is equipped with the *roll back* function to bring the network back to the state before over pruning. Besides, the pruning procedure does not cease immediately. The network has several chances to try different pruning quantities. Every time it fails to prune $prune\_num$ (initialized as $prune\_num_{max}$) weights, it rolls back and tries pruning $prune\_num/2$ weights instead. If several different quantities (we find 3 is an appropriate choice for popular networks) fail at the same point, the pruning procedure terminates directly to avoid more overhead. That is, $prune\_num$ is set to 0 when

$$prune\_fail\_times > 3 \quad (5),$$

where $prune\_fail\_times$ counts the times that roll back occurs at the same point.

3.2 Hyperparameters

EP introduces $over\_prune\_threshold$, $prune\_interval$ and $prune\_num_{max}$ as new hyperparameters. Like other training hyperparameters (e.g. learning rate and maximal iterations), experience is needed when setting them. Our studies show little difference in over pruning recognition when $\boldsymbol{over\_prune\_threshold}$ is set over 30. In the experiment, 10 is used for networks that take less than 500,000 iterations to train and 20 for those take longer.



(a) $\boldsymbol{Prune\_num_{max}}$ (b) $\boldsymbol{Prune\_interval}$

Figure 7. Models for Choosing Hyperparameters (Alexnet)

$\boldsymbol{Prune\_interval}$ and $\boldsymbol{prune\_num_{max}}$ are the most important hyperparameters in EP. Intuitively, the smaller $prune\_interval$ is and the greater $prune\_num_{max}$ is, the more computation is reduced. However, pruning too eagerly hurts the accuracy and compression rate of the network. On one hand, when too many weights are pruned at each time, significant weights may be accidentally pruned when they are still oscillating, resulting in degraded accuracy. On the other hand, if the network is pruned at a high frequency, the remaining weights do not have enough time to be retrained before next pruning, so the network is considered over pruned. This causes the pruning procedure to terminate early, leading to low computation reduction.

Figure 7 shows how the two hyperparameters impact the accuracy and computation. Pi_{opt} and pn_{opt} are the optimal values for $prune\_interval$ and $prune\_num_{max}$. In Figure 7(a), $prune\_interval$ is set to pi_{opt} and $prune\_num_{max}$ varies from $pn_{opt}/4$ to $4pn_{opt}$. The accuracy starts to drop when $prune\_num_{max}$ grows larger than pn_{opt}. In Figure 7(b), $prune\_num_{max}$ is set to pn_{opt} while $prune\_interval$ varies from $pi_{opt}/9$ to $9pi_{opt}$. The most reduced computation is achieved at pi_{opt}. In our study, setting $prune\_interval$ as $1/100$ of the total iterations and $prune\_num_{max}$ as $1/50$ of the number of weights expected to be pruned is a conservative strategy. The pruning will terminate at the middle of training. To determine the expected pruning quantity, it is recommended to refer to off-the-shelf models trained on datasets of similar size.

4. Architecture Support for Eager Pruning

The proposed algorithm exhibits great potential in reducing computation for DNN training. Nevertheless, this benefit cannot be exploited by recently proposed accelerators[20]–[28] due to the irregular sparsity, imbalanced workloads in EP and the discrepancy in different training phases. In this section, we analyze the challenges brought by EP to existing designs and propose our architecture support for EP.

4.1 Challenges

Irregular sparsity requires flexible dataflow. To leverage the abundant parallelism and data reuse in convolutions, a common practice is to unroll the convolution loops in Figure 3. Figure 8(a) illustrates an example that unrolls the outputs within an output feature map[20]. Each cycle, a new filter weight is loaded and broadcast to all PEs.

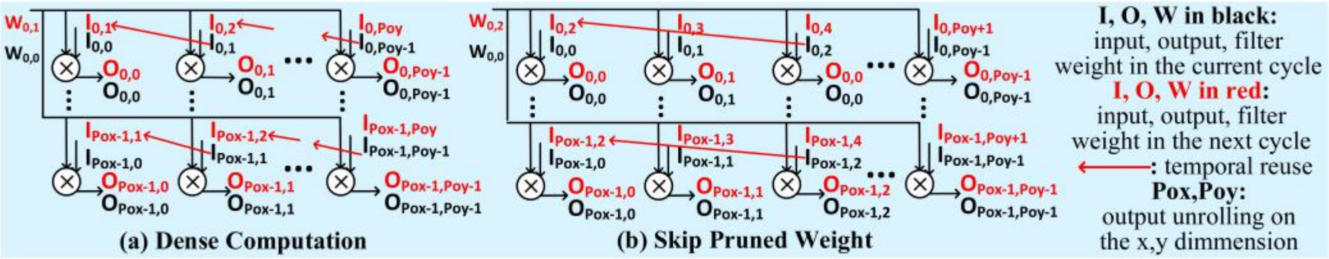


Figure 8. An Example of Data Reuse

Each PE is responsible for one output by locally accumulating the partial results. The index of the input required by each PE can be calculated as:

$$(ix, iy) = (fx + ox, fy + oy) \text{ (6)},$$

where (fx, fy), (ox, oy) and (ix, iy) are the indices of filter weight, output and input respectively. The black and red notations in Figure 8(a) represents the input, output and weight in two adjacent cycles. As can be seen, the current input of one PE is the same as the input of its neighbor in the next cycle. Therefore, the inputs can be temporally reused by different PEs in different time slots through shifting.

However, this model cannot deal with irregular sparsity of the filter. Assume that the weight $W_{0,1}$ is pruned. The performance can only be enhanced if the partial result of $W_{0,1}$ is skipped as shown in Figure 8(b). In this case, the inputs in the PEs should shift by 2 to realize temporal reuse. Similarly, when both $W_{0,1}$ and $W_{0,2}$ are pruned, inputs need to shift by 3. Since the filter sparsity in EP is irregular, the fixed shifting structure is not applicable. [21], [22] adopt other data reuse policies but the same problem exists. Flexflow[24] eliminates some of the datapaths between PEs to achieve reconfigurable hybrid unrolling and data reuse, but it still lacks the ability to skip computations irregularly.

Imbalanced workloads cause resource underutilization. Cambricon-X[26] is designed for sparse neural networks but also suffers from resource underutilization due to the fixed unrolling scale. It assigns T_m multipliers to each of the T_n outputs and the partial results are accumulated by an adder tree. However, the computation entailed for one output differs significantly from another in a sparse network. Idleness will occur in rows with lighter workloads.

Discrepancy in multiple phases yields inefficiency. SCNN[27] proposes a novel dataflow. Each PE (a group of multipliers) only loads the non-zero inputs and filter weights and performs the Cartesian products of them. This dataflow works well for DNN inference but is not efficient for training, especially in the KU phase.

In each layer, the MAC operations performed by SCNN is ($input\_size \times filter\_size$), but the MAC operations required for the output is ($output\_size \times filter\_size$). Therefore, the utilization can be derived as:

$$Util. = \frac{required\ op.}{performed\ op.} = \frac{output\_size}{input\_size} \text{ (7)}.$$

Based on Table 1, the output and input of KU are the kernel update and intermediate data respectively. The average kernel size of all the popular networks is 7 while the average size of a feature map is 1964. Such a considerable gap results in SCNN utilization less than 1%.

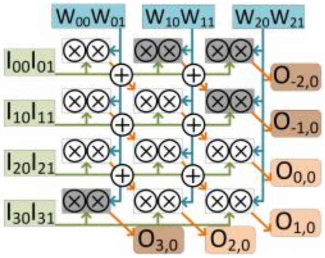


Figure 9. Dataflow in ScaleDeep

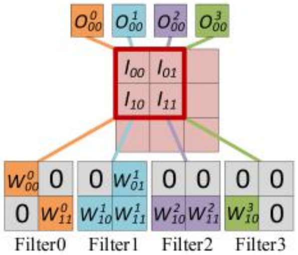


Figure 10. Sparse Filters

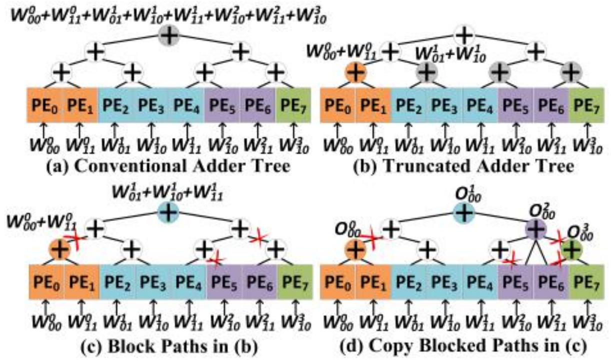
ScaleDeep[28] is proposed for DNN training and the workloads are mapped to different compute tiles based on their Bytes/FLOP rate. Nevertheless, the CompHeavy Tile, which is used for performing convolutions in the three phases, adopts a similar dataflow as in SCNN and also results in low utilization. In Figure 9, each row of inputs is multiplied by each row of the weights and the partial results are accumulated diagonally. Consider the output is 3 × 3. The dark multipliers do not generate any effectual partial results.

What's more, since SCNN and ScaleDeep compute all the pairs of inputs and filter weights, they are unable to skip the computation for the update of a pruned kernel weight. Note that the average MAC operations to calculate an update in KU is 770, which is much greater than the average MAC operations (i.e. 7) to calculate an output in FP. Therefore, a huge amount of time and energy will be wasted.

Solution The dataflow and the unroll scale of PEs are fixed in the existing designs. Even the flexible architectures designed for sparse DNNs still partition the compute resource into groups and remain some of the connections inside and between each group, resulting in underutilization. This problem can be solved by further decoupling and disconnecting the PEs to allow each of them to be allocated to any output to perform computation individually. A reconfigurable adder tree to dynamically collect and accumulate the results of the ungrouped PEs that are assigned to the same output will be introduced in Section 4.2.

4.2 DRACT

In our design, a single PE multiplies one input with one filter weight. The function of DRACT is to dynamically collect and accumulate the partial results of the same output. This is done by configuring the DRACT based on the weights loaded into PEs in real time.


$W_{00}^0+W_{11}^0+W_{01}^1+W_{10}^1+W_{11}^1+W_{10}^2+W_{11}^2+W_{10}^3$
$W_{00}^0+W_{11}^0$
$W_{01}^1+W_{10}^1$
$W_{01}^1+W_{10}^1+W_{11}^1$
O_{00}^0
O_{00}^1
O_{00}^2
O_{00}^3
PE_0 PE_1 PE_2 PE_3 PE_4 PE_5 PE_6 PE_7
W_{00}^0 W_{11}^0 W_{01}^1 W_{10}^1 W_{11}^1 W_{10}^2 W_{11}^2 W_{10}^3

(a) Conventional Adder Tree (b) Truncated Adder Tree

(c) Block Paths in (b) (d) Copy Blocked Paths in (c)

Figure 11. Prototype of DRACT (Inputs Omitted)

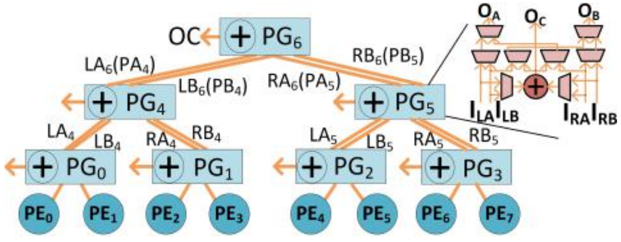


Figure 12. Structure of DRACT

Prototype of DRACT Suppose there are four sparse filters convolving one input as shown in Figure 10. They generate four different outputs. The non-zero weights are loaded into eight PEs consecutively in Figure 11. To produce the outputs, the PEs need to be partitioned into four irregular groups $[PE_0, PE_1]$, $[PE_2, PE_3, PE_4]$, $[PE_5, PE_6]$, $[PE_7]$ and the partial results of each group should be summed up. A conventional binary adder tree can only generate one output at the top of the tree (Figure 11(a)). Even if the tree is truncated and produces $(W_{00}^0 + W_{11}^0)$ at the first-level adder, obtaining $(W_{01}^1 + W_{10}^1 + W_{11}^1)$ is impossible because W_{11}^1 is added by W_{10}^2 first (Figure 11(b)). One solution is to block the path of PE_5, preventing it from reaching to PE_4 (Figure 11(c)). However, W_{10}^2 is not able to reach out to perform addition with W_{11}^2 in this way. This drives us to copy the paths between PE_5 and PE_6 so that they can connect to each other (Figure 11(d)).

Structure of DRACT Since the sparsity of the filters changes during training, the result of every PE may be blocked from its left or right siblings but needed on the other side. Therefore, all the paths in the adder tree are duplicated. The path forwards a data only when the data falls into a group that contains data not in this path's subtree. Since PEs are partitioned consecutively, if the subtree's data belongs to more than two groups, the data of the middle groups never need to reach out because the subtree provides sufficient connections for all the internal additions. Therefore, doubling the paths is enough regardless of the number of PEs.

Figure 12 presents a DRACT connecting eight PEs. Each adder is embedded into a *path gate (PG)*. The PG has two inputs, I_{LA} and I_{LB}, from its left child, two inputs, I_{RA} and I_{RB}, from its right child and three outputs. O_A and O_B are sent upward and serve as the left or right inputs of its parent. The other output is collected by the *output collector (O_C)* and then sent to the output buffer. Note that the bottom paths are not doubled because there is at most one data from each child.

Table 3. Outputs of PG

Case	LA	LB	RA	RB	PA	PB	O_A	O_B	O_C
1 input 1 output	1	0	0	0	1	0	I_{LA}		
	0	0	1	0	1	0	I_{RA}		
2 input 0 output	1	0	1	0	0	0			$I_{LA} + I_{RA}$
2 input 1 output	1	0	1	0	1	0	$I_{LA} + I_{RA}$		
2 input 2 output	1	0	1	0	1	1	I_{LA}	I_{RA}	
3 input 1 output	1	1	1	0	1	0	I_{LA}		$I_{LB} + I_{RA}$
	1	0	1	1	1	0	I_{RB}		$I_{LA} + I_{RA}$
3 input 2 output	1	1	1	0	1	1	I_{LA}	$I_{LB} + I_{RA}$	
	1	0	1	1	1	1	$I_{LA} + I_{RA}$	I_{RB}	
4 input 2 output	1	1	1	1	1	1	I_{LA}	I_{RB}	$I_{LB} + I_{RA}$

Table 4. DRACT Configuration

PE	F	L	FC	LC	BO	Out PG
0	1	0	LA4,LB4,LA6,LB6		L0	
1	0	1		LA4,LB4	R0	0
2	1	0	RA4,RB4		L1	
3	0	0		RA4,RB4,LA6,LB6	R1	
4	0	1	LA5,LB5,RA6,RB6		L2	6
5	1	0		LA5,LB5	R2	
6	0	1	RA5,RB5		L3	5
7	1	1		RA5,RB5,RA6,RB6	R3	3

For each path in DRACT, one *datapass bit* is utilized to denote whether it is in use. Specifically, LA (or $LB/RA/RB$) is set to 1 only when the child is sending data from I_{LA} (or $I_{LB}/I_{RA}/I_{RB}$). PA and PB represent if the PG is sending data to its parent through O_A and O_B. PA and PB are equivalent to the parent's LA and LB (or RA and RB).

A series of selectors in PG are controlled by the datapass bits to dynamically connect the inputs and outputs. Note that the data is sent to O_B only when O_A is occupied. Based on this principle, all the possible combinations of the datapass bits and the outputs of each case are listed in Table 3. When the same number of inputs and outputs are in use, the inputs are simply forwarded. When the number of outputs is less than that of the inputs by one, two neighboring inputs, one from each child, are added up and forwarded. When there are two more inputs than outputs, the result of addition is sent to O_C.

Configuration of DRACT The datapass bits of DRACT are reconfigured in real time when new weights are loaded into the PEs. Each weight is accompanied with two *partition bits*. The partition bits indicate if this weight is the first or the last non-zero weight of the filter. For example, [(1,0), (0,1), (1,0), (0,0), (0,1), (1,0), (0,1), (1,1)] corresponds to the partition of $[PE_0, PE_1]$, $[PE_2, PE_3, PE_4]$, $[PE_5, PE_6]$, $[PE_7]$. The partition bits of the weight that resides in a PE are used to set the datapass bits of two paths.

When a PE is the first or last external node of a path's subtree, this path is configurable by the PE. If the PE is the first external node and the weight in the PE is not the first of the filter, this PE must be connected to the outside of this path's subtree. In this case, the datapass bit of the path is set to 1 to forward data to a higher level. Similarly, if the PE is

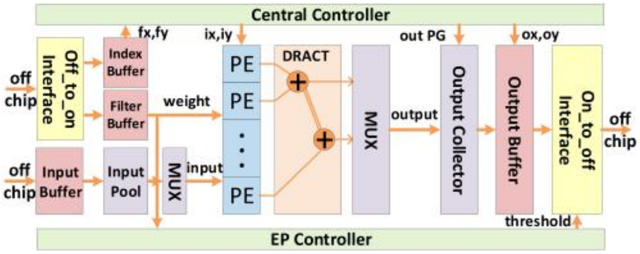


Figure 13. Eager Pruning System Overview

the last external node of the path's subtree and the weight is not the last of the filter, the datapass bit is also set to 1. Note that LA/RA should be set first. If LA/RA is 1, then set LB/RB.

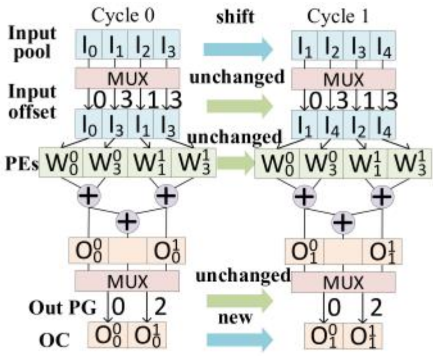
Table 4 gives an example of DRACT configuration assuming the partition is $[PE_0, PE_1]$, $[PE_2, PE_3, PE_4]$, $[PE_5, PE_6]$, $[PE_7]$. Each row lists the configuration performed by one PE. F/L refers to whether the weight is the first/last non-zero one of the filter. FC/LC lists the datapass bits of the paths configurable by this PE because this PE is the first/last node of the subtree. BO represents the special case of bottom paths, where the PEs are both the first and last node of their subtrees. Datapass bits set to 1 are marked as red. Those already set to 1 by other PEs are marked as green. *Out PG* denotes which PG is generating outputs for the output collector. It is the highest-level PG whose input path is set as 1 by each filter. Not surprisingly, the result of the DRACT configuration is the same as in Figure 11(d).

4.3 Data Access

Figure 13 illustrates an overview of the proposed system for Eager Pruning. The entire system is governed by the central controller. All the PEs are ungrouped and connected by the DRACT. There are four multi-bank on-chip buffers. Three of them support input, filter and output accesses and the other is used to store the indices of unpruned kernel weights. All the buffers are dual-port so the writing of the new data and reading of the old data are pipelined.

Filter The kernels are stored off-chip. Each weight has one mask bit indicating if it is pruned. CSC format[23] is not employed due to the overhead when sparsity is low. In FP and BP, the weights and mask bits are loaded from off-chip through the off-chip-to-on-chip interface. Informed of the kernel shape by the central controller, the interface calculates the index and partition bits of unpruned weights using a pair of counters. Then the weights are put into the filter buffer and the indices are put into the index buffer. The filter access in KU is realized by buffer redirecting, which is introduced in Section 4.4. The multi-bank filter buffer can feed a number of PEs per cycle. By doubling the weight registers in the PEs, weight loading can be overlapped with computation.

Input The *input window* is defined as a patch of input feature map required to produce an output. It is the same size of the filter. For example, the input window of $O_{ox,oy}$ is a square $[I_{ox,oy}, \ldots, I_{ox+2,oy+2}]$ if the filter is 3×3. In EP system, the inputs are first loaded into the input buffer and then the data within the input windows of all the outputs being computed is sent to the input pool.



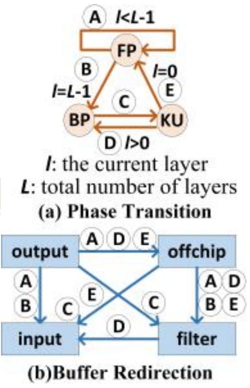


Figure 14. Data Indexing

Figure 15. Phase Transition

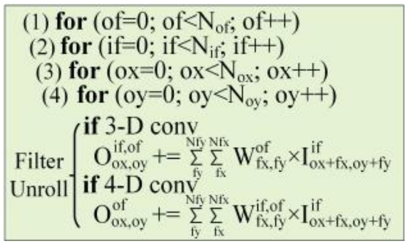
When new weights are loaded into PEs, the weight indices are sent to the central controller. Note that in FP and BP, the indices are fetched from the index buffer. In KU, the indices are dense and directly generated by the central controller based on the filter shape. A group of adders in the central controller calculate the input offset for each PE. The input offset is calculated using Equation(6) with the weight index and output index. The PEs are connected to the input pool by the Fat-tree[29] and the input indexing is implemented as in Cambricon-X[26]. Once the input is located, the PE keeps receiving inputs from the same position until the weights are replaced. The elements in the input pool are shifted to keep the input window as shown in Figure 14.

Output Each output buffer bank has an output collector. The PGs are connected to the output collectors by the Fat-tree. When the filter weights are loaded into the PEs, the DRACT informs the Out PG for each output, as mentioned in Section 4.2. Then the output collectors select the required outputs as the PEs select the inputs. Shown in Figure 14, the indexing of the outputs is also fixed because the partition of the PEs is fixed until the weights are reloaded. In FP and BP, the output buffer is redirected to serve other purposes, as introduced in Section 4.4. In KU, the updated sparse kernels are transformed into the pattern with a mask bit at the on-chip-to-off-chip interface and sent to off-chip.

4.4 Dataflow

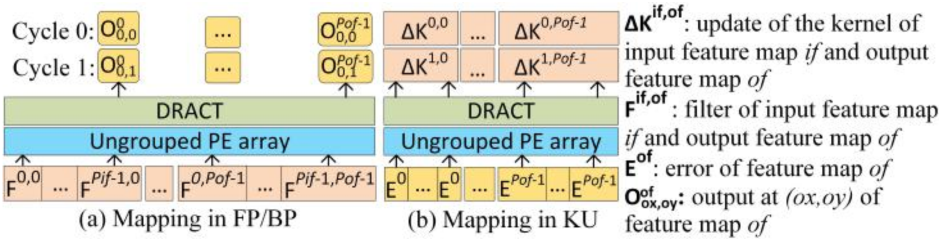
In SGD[19], the training samples are grouped into mini batches. Each sample goes through the three phases to generate the kernel updates respectively. Then the kernels are updated by the average values.

Phase Transitions The system processes one layer at a time. The central controller dictates the transition between the phases following the state machine in Figure 15(a). Based on the data dependencies in Table 1, FP of all the layers is performed first. When FP of the last layer finishes, the system transits to an alternation of BP and KU. Specifically, once the error of a layer is generated, the system starts to compute the kernel update for that layer. When KU of the first layer is done, the system transits to FP to process another sample.



```
(1) for (of=0; of<$N_{of}$; of++)
  (2) for (if=0; if<$N_{if}$; if++)
    (3) for (ox=0; ox<$N_{ox}$; ox++)
      (4) for (oy=0; oy<$N_{oy}$; oy++)
Filter Unroll
        if 3-D conv
          $O^{if,of}_{ox,oy} += \sum_{fy}^{Nfy}\sum_{fx}^{Nfx} W^{of}_{fx,fy} \times I^{if}_{ox+fx,oy+fy}$
        if 4-D conv
          $O^{of}_{ox,oy} += \sum_{fy}^{Nfy}\sum_{fx}^{Nfx} W^{if,of}_{fx,fy} \times I^{if}_{ox+fx,oy+fy}$
```

Figure 16. Loop Unrolling



$\Delta K^{if,of}$: update of the kernel of input feature map *if* and output feature map *of*
$F^{if,of}$: filter of input feature map *if* and output feature map *of*
E^{of}: error of feature map *of*
$O^{of}_{ox,oy}$: output at *(ox,oy)* of feature map *of*

Figure 17. Computation Mapping

When phase transition happens, the input, filter and output buffers are redirected so that off-chip access is reduced. For example, in FP, the output of layer l is the input of layer $l+1$, so the input and output buffers can be simply switched when proceeding to the next layer. Figure 15(b) illustrates the buffer redirections in the case of each phase transition.

Computation Mapping In our design, the filters are always completely unrolled as in Figure 11. Since FP/BP and KU perform different convolutions, the unrolling of the other four loops in Figure 16 is different in FP/BP and UD.

The mapping of **FP/ BP** is shown in Figure 17(a). Loops (1) and (2) are unrolled by P_{of} and P_{if} respectively. Filters of P_{if} input feature maps corresponding to the same output feature map are loaded into consecutive PEs to produce one output. The filters of P_{of} feature maps are unrolled so the outputs of P_{of} feature maps can be calculated at the same time. Loops (3) and (4) are not unrolled, so only one output of a feature map is generated each cycle. The output index of each cycle is the number of cycles. It takes $N_{ox} \times N_{oy}$ cycles to generate an output feature map.

The input pool holds the input windows of all the unrolled input feature maps. Since the input window is of the same size as the dense filter, P_{if} is chosen as the maximal that satisfies:

$$P_{if} \times dense\_filter\_size \leq input\_pool\_size \ (8).$$

P_{of} is limited by the number of PEs and the number of output collectors. It is chosen as the maximal that satisfies both:

$$\sum_{0}^{Pof-1}\sum_{0}^{Pif-1} \#nzw \leq \#PEs \ (9),$$

$$P_{of} \leq \#output\_collectors \ (10),$$

where $\#nzw$ is the number of non-zero weights in a filter.

KU generates the kernel updates. Therefore, the output of KU is sparse. If only one output is generated for each output feature map in each cycle, idleness will occur for those with less weights to update. This drives us to unroll the outputs within a feature map (loops (3) and (4)) in Figure 17(b). The filters (errors) are duplicated to generate the outputs (updates) within one feature map. Besides, loop (1) is also unrolled by P_{of}, so the updates of P_{of} kernels are generated at the same time. Note that in the 3-D convolution in KU, the filters are shared by different input feature maps, so the errors are kept until the updates of all the input feature maps are generated.

4.5 EP Controller

The EP controller in Figure 13 connects to the output buffer to fulfill the algorithm for EP. It activates pruning according to the training iterations and smooths the training loss to adjust $prune\_num$. When a pruning is activated, EP controller tries to obtain the threshold of the smallest $prune\_num$ kernel weights by guessing. Each time the weights are loaded from the filter buffer to PEs in FP/BP, they are also sent to the EP controller. The EP controller counts the number of weights under the threshold and adjusts the threshold if the guess is wrong. It is acceptable if the correct guess appears later because the weights are not updated until a batch has been processed. When the method of bisection is applied, it takes at most 8 iterations (one guess per FP/BP) to obtain the threshold if 16-bit fixed-points are used, which is negligible compared with $prune\_interval$.

The threshold is sent to the on-chip-to-off-chip interface. When the kernel weights are updated, the mask bit of those under the threshold will be set to 1. These pruned weights will not be loaded into the filter buffer in the following iterations.

5. EVALUATION

This section evaluates EP's performance in the algorithm and compute layers respectively and explore the overall benefits of our system. EP is applied to eight popular networks, LeNet(LN)[30], AlexNet(AN)[18], GooLeNet (GLN)[8], NIN[31], Resnet-32(R32), Resnet-50(R50)[32], Finetuning(FT)[33] and LSTM[34] (1024 memory cells), on different datasets in Table 5. FT is a case of transfer learning, where CaffeNet[35] is first trained on all the non-animal samples in ImageNet and then finetuned on animal samples.

5.1 Algorithm Validation

We modify Caffe[35] and Kaldi[36] to implement the algorithm for EP in the CNN and RNN training respectively. The networks are trained on the training datasets and validated on the validation datasets. EP is applied to only convolution layers in CNNs because more than 90% of the computation comes from these layers[27]. In LSTM, EP is applied to all the weight matrices. The results are listed in Table 5. On average, the algorithm compresses the models by 2.57x and reduces 40% training computation while maintaining the original accuracy. The accuracy of traditional training only achieves 95% of that of EP within the same time.

To distinguish our work (Figure 1(e)) from the traditional pruning method (Figure 1(b)), Figure 18 depicts the accuracy and remaining weight ratio in the process of training AlexNet using EP (green lines) and the traditional pruning which prunes after the original training iterations (orange lines). The

Table 5. Algorithm Validation

BS: batch size, $\boldsymbol{Pn_{max}}$: $prune\_num_{max}$, $\boldsymbol{pi}$: $prune\_interval$, **original**: the accuracy of the original training without pruning, **baseline**: the accuracy achieved by the traditional training when EP training finishes, **CR**: the rate that the model is compressed with EP, **RC**: the ratio of computation reduced during training. *Phone recognition accuracy.

Net-work	Dataset	Training Hyperparameters				Validation Accuracy(%)			CR	RC(%)
		Iterations	BS	$\boldsymbol{Pn_{max}}$	$\boldsymbol{Pi}$	Original	EP	Baseline		
AN	ImageNet[1]	4.5×10^5	256	29,358	4,500	57.01	57.00	55.48	2.61x	41.19
GLN	ImageNet	5×10^6	32	120,076	100,000	65.70	64.72	62.80	3.43x	43.25
NIN	ImageNet	4.5×10^5	64	128,443	4,500	58.76	57.60	54.82	1.56x	26.87
R50	ImageNet	2.4×10^6	32	281,459	24,000	72.99	72.72	69.61	2.37x	40.13
LN	MNIST[30]	1×10^4	64	80	200	99.18	99.05	98.76	1.85x	38.06
R32	Cifar10[52]	1×10^5	128	6,122	1,000	91.67	91.80	87.22	2.98x	44.86
FT	Animals	2.5×10^5	256	29,358	2,500	65.83	65.53	63.97	2.14x	34.63
LSTM	TIMIT[53]	4×10^5	32	52,072	4,000	79.95*	78.24*	70.80*	3.58x	52.40

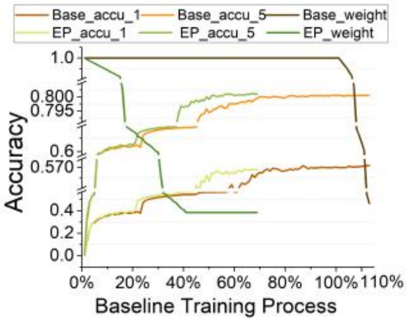


Figure 18. Training Process of EP

time axis is normalized to the time to complete baseline training without pruning. As shown, EP accomplishes training and produces a compressed model in less than 70% of the original training time, while traditional pruning starts at the 100% training time and takes more than 110% in total.

5.2 Architecture Configuration

Our architecture design is prototyped on an Xilinx VCU118 Evaluation board which includes two sets of five 512MB DDR4 SDRAM and an Xilinx UltraScale+ XCVU9P FPGA. The system runs at 200MHz and the data precision implemented is 16-bit.

To alleviate the indexing overhead, the resources are evenly partitioned into four sub-systems to calculate four groups of input feature maps. In the experiment, we notice that although the number of pruned weights differs between different kernels, the total number of pruned weights of an input channel is balanced. Therefore, the sub-systems will finish the work synchronously.

In each sub-system, four 1.5MB dual-port buffers are utilized to store inputs, filters, outputs and indices, each supporting a peak data access of 64B per cycle. The input pool of a sub-system contains 8×8 elements to hold the input windows. Each sub-system has 512 PEs and 32 output collectors. The resource utilization of our design is listed in Table 6 and the total dynamic on-chip power is 7.3W.

EP is compared with five state-of-the-art DNN accelerators, FlexFlow(FF)[24], Cambricon-X(CX)[26], SCNN[27], ScaleDeep(SD)[28] and ESE[37]. Each layer of the eight networks is mapped to the prototype architectures individually. Note that all the designs use at least 2048 DSPs as their PEs and the same data buffers as in our design.

FF has 64 rows, each with 32 PEs computing for one output. Output feature maps and the outputs within each feature map are unrolled vertically. The input feature maps and the filters are unrolled horizontally. We try a wide range of the unrolling factors from 1 to 64 and count the cycles needed to complete each layer under different unrolling policies. To boost performance, FF is applied with per layer optimization, i.e. we sum up the fewest cycles to complete each layer among all the unrolling policies.

The 16×16 PE array in the original CX design is duplicated by 8, resulting in 128 rows of PEs. 16 PEs in a row serve one output. A 16×16 input pool is applied for input indexing as in our design. The unrolling policy and dataflow of the implemented CX is also similar as our design in the three training phases. The difference is that CX assigns a fixed number of 16 PEs to each output.

Table 6. Utilization of FPGA Resource

Resource	LUT	Flip-Flop	Block RAM	DSP
Usage	733,450	289,879	1,539	2,064
Available	1,182,240	2,364,480	1,800	6,840

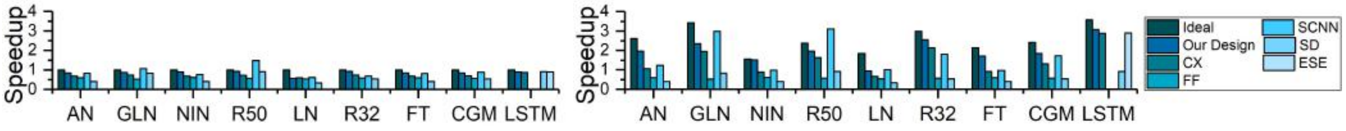
SCNN is made up of 128 tiles, each unrolling 4 non-zero inputs and 4 non-zero filter weights. Since the buffer provides at most 128 weights and each tile consumes 4 weights at a time, the tiles are divided into 32 groups and those within each group share the weights spatially. The inputs of each group are stationary until all the Cartesian products are useless. Then the new inputs are fed in and weights are fetched from the beginning again. The input sparsity is not considered in EP but SCNN is outstanding for its ability to skip zeros in both weights and inputs. Therefore, we multiply the performance of SCNN by 2 to estimate the real speedups.

SD has 6 rows and 4 columns of CompHeavy tiles (CHT). Each CHT consists of 8×3 4-lane 2D-PE arrays. The function of MemHeavy tiles (MHT) in convolution is realized by an array of accumulators connected to the outputs of CHTs. As discussed in [28], the 2D-PE with 4 lanes are split into four 1-lane PEs for the fully connected layers in FP/BP. Fully connected layers in the KU phase is performed by the MHT. An array of 128 multipliers is applied, which is determined by the buffer bandwidth.

ESE is evaluated on LSTM only. As mentioned in Section 2.1, the operations in LSTM are indeed the same as in fully connected layers. Since each weight is used by only one output, a new weight is loaded by each PE in every cycle. In this case, the number of PEs is determined by the bandwidth of the filter buffer, which is 128 in our design. Therefore, ESE is deployed with 16 channels and each channel has 8 PEs. When comparing to ESE, the other models also use 128 PEs.

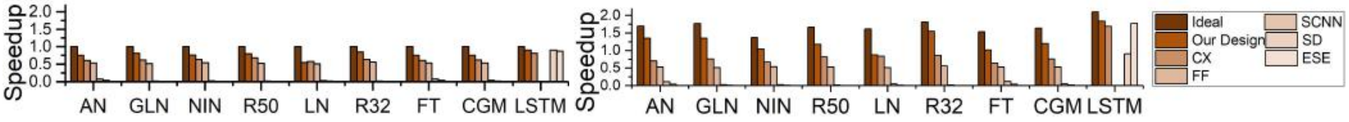
5.3 Accelerator Performance

We first evaluate the performance on one iteration of dense and sparse inference and training. Figures 19 and 20 show the results on inference and training respectively. The ideal cases refer to the performance when all the PEs work without



(a) Dense Inference **(b) Sparse Inference**

Figure 19. Performance on Inference



(a) Dense Training **(b) Sparse Training**

Figure 20. Performance on Training

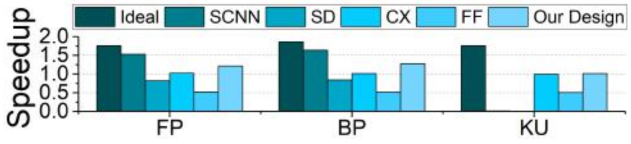


Figure 21. Training Breakdown

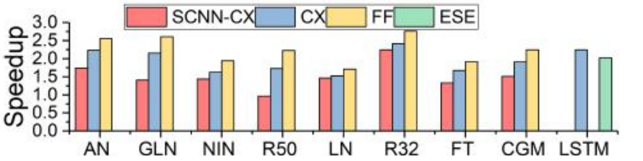


Figure 22. Overall Benefits

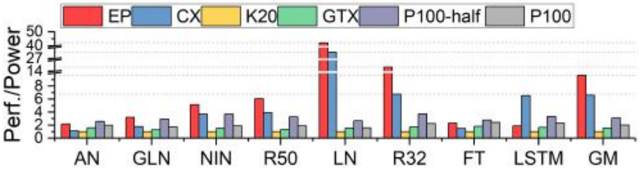


Figure 23. Energy Efficiency

idleness. The bars are normalized to the ideal case of dense computation. CGM is the geometric mean of the CNNs.

Inference Figures 19(a) and (b) illustrate the performance on dense and sparse inference. SCNN (0.88x and 1.72x on dense and sparse inference) and our design (0.84x and 1.86x) have the best and similar average performance. Our design does not perform well on LN because LN has only one input feature map so only one of the four sub-systems is busy at the first layer. The performance of SCNN on AN, NIN and R32 are not as high as the other cases because they contain more layers where inputs are significantly larger than outputs.

Training Figures 20(a) and (b) show the average speedup on one iteration of training without and with EP algorithm respectively. Figure 21 presents the speedup breakdown on the three phases when training GLN with EP. As discussed in Section 4.1, the utilization of SCNN and SD in KU is very low due to the great gap between the size of inputs and outputs. Therefore, the training speedup of SCNN and SD is dominated by KU, which is nearly 0 (0.05x and 0.03x) in Figure 21. Consequently, their speedups on training are negligible compared with other accelerators. Our design gains the highest speedup on training. The edge over the others is even more conspicuous in the sparse training thanks to the flexible connections. In Figure 20(b), our architecture for EP is the only one with > 1 speedup (1.19x CGM).

The difference in the performance of different accelerators on LSTM is not as obvious as CNNs because there is abundant parallelism but the number of PEs is limited by the filter buffer bandwidth.

5.4 Overall Benefits

To further investigate the overall benefits brought by our proposed algorithm and architecture support for EP, Figure 22 shows the end-to-end speedup of Eager Pruning system over the dense training on other architectures. The combination of our EP algorithm and accelerator achieves an average of 1.91x and 2.25x speedup over CX and FF respectively. We found that SCNN works the best in FP/BP and CX works the best in KU. If a hybrid architecture of SCNN and CX is applied to train by pipelining each phase, EP still wins by 1.51x.

5.5 Energy Efficiency

To evaluate the energy efficiency of EP, we train the networks from scratch without pruning on CX and three GPUs. The GPUs are selected from three generations of Nvidia, Tesla K20c (Kepler), Geforce GTX Titan X (Maxwell) and Tesla P100-16GB (Pascal). All the implementations use the same training batch size and iterations as listed in Table 5. P100 is evaluated in both the single-precision and half-precision modes. The power of each implementation is measured using WattsUp[38]. In Figure 23, the performance/power value is normalized to K20. GPUs can efficiently process large networks with big batch sizes, e.g. AN and FT. However, their energy efficiency on LN and R32 is not good as the utilization is low. They also perform better on LSTM due to sufficient memory and bandwidth. On average, our design achieves 1.46x, 3.10x, 6.17x and 9.66x energy-efficiency over CX, P100, GTX and K20.

Figure 24 shows the potential ability of EP, CX and P100 (half-precision) to satisfy the latency requirements of diverse applications[3]–[5], [39], [40] on different platforms. The power budgets vary from several to thousands of watts on mobile, IoT, desktop and datacenter devices. The high-power GPUs cannot be deployed to mobile ends and IoT nodes.

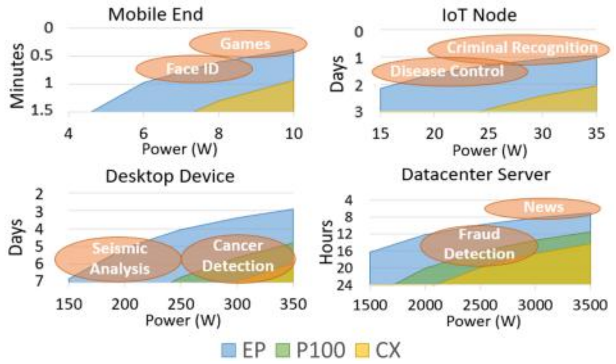


Figure 24. Application Latency on Various Platforms

The mobile ends are used to perform simple tasks where the dataset is small or the required accuracy is low. We use the latency to train LN on MNIST to represent this kind of applications. IoT nodes are usually adopted for detecting a subset of objects. In this case, the accuracy needs to be guaranteed although the training data and time are limited. The most suitable example is R32 on Cifar10. R50 is trained on ImageNet on desktop and datacenter devices to obtain high accuracy on extreme large datasets. In Figure 24, EP overlaps with all the bubbles while CX and the GPU cannot meet the requirements of time-sensitive application. For example, users may abandon the app to transfer images into a customized art style in more than 30 seconds[41] but CX can only finish the training in 60 seconds. In a datacenter, only our design can learn a model on the latest news in one night (8 hours). As for applications which can be run by all the implementations, EP meets the latency requirement with the lowest power budgets. For example, EP can learn a cancer detection model in 7 days with less than 150 watts and train a model for fraud detection in 24 hours with less than 1500 watts. By comparing the four platforms, it can be concluded that EP brings the most value to applications with high latency requirement but low power budget.

6. Related Work

Many works are proposed to scale down DNN models. The methods[42]–[44] that leverage precision reduction and encoding to decrease the size of each parameter are orthogonal to our work. For those making efforts to reduce the number of parameters, [13], [14], [37] prune the parameters with small absolute values in DNNs and get no loss in accuracy after retraining. [15], [45] perform group pruning to reduce irregularity. All these methods only benefit the inference tasks as shown in Figure 1(b). [46] introduces a regularization that encourages low rank of parameter matrix during training. It is easier to generate more compact models but does not reduce the computation during training. [47] compresses the weights into regular patterns and reduces computation using FFT during training but it requires transforming the kernels into block-circulant patterns.

Traditionally, DNNs are accelerated by unrolling the nested loops, like the methods used in [20]–[22]. They rely on fixed unrolling strategies and delicate connections between PEs to exploit data reuse. This makes them unable to handle irregular computation patterns in sparse DNNs. [23], [37] skip arbitrary zeros in filters but are only proposed to accelerate fully connected layers. [24]–[26] eliminate some data paths between PEs to allow higher flexibility but still result in resource underutilization because the unrolling scale is fixed while the workloads of EP are imbalanced. [27] skips all the zeros in both filter weights and inputs but its dataflow is inefficient in KU. [48] also proposes a flexible adder tree but it cannot be reconfigured in real time and the dataflow to perform different phases in sparse training is unclear.

For DNN training, [28] is not equipped with computation skipping function and adopts dataflow which is not efficient for KU. [49], [50] target on memory optimization during training and [51] is proposed to alleviate the overhead of distributed DNN training.

7. Conclusions

Eager Pruning is an algorithm and architecture co-design for fast training of DNNs. Based on the observation that the ranking of the significance of the weights changes slightly during training, we propose to move pruning to an earlier stage to reduce training computation. To transform the reduced computation into performance improvement, we also propose a DRACT-based architecture to support Eager Pruning. Thanks to the high energy-efficiency, Eager Pruning shows great potential in enabling applications with high latency requirement and limited power budget.

Acknowledgment

This work is supported in part by NSF grants 1900713, 1822989, 1822459, 1527535, 1423090, and 1320100.

References

[1] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. ImageNet: A large-scale hierarchical image database. In CVPR, 2009.

[2] Andrej Karpathy. What I learned from competing against a ConvNet on ImageNet: http://karpathy.github.io/2014/09/02/what-i-learned-from-competing-against-a-convnet-on-imagenet/.

[3] Henry A. Rowley, Shumeet Baluja, and Takeo Kanade. Neural Network-Based Face Detection, TPAMI, 1998.

[4] Zhaohui Zhang, Xinxin Zhou, Xiaobo Zhang, LizhiWang, and PengweiWang. A Model Based on Convolutional Neural Network for Online Transaction Fraud Detection, Secur. Commun. Networks, 2018.

[5] Carey E. Floyd, Joseph Y. Lo, A. Joon Yun, Daniel C. Sullivan, and Phyllis J. Kornguth. Prediction of breast cancer malignancy using an artificial neural network, Cancer, 1994.

[6] Big Data vs. Fast Data: https://www.voltdb.com/why-voltdb/big-data/.

[7] Internet Live Stats: http://www.internetlivestats.com.

[8] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going Deeper With Convolutions. In CVPR, 2015.

[9] Karen Simonyan and Andrew Zisserman. Very Deep Convolutional Networks for Large-Scale Image Recognition. In arXiv:1409.1556, 2015.

[10] Takuya Akiba, Shuji Suzuki, and Keisuke Fukuda. Extremely Large Minibatch SGD: Training ResNet-50 on ImageNet in 15 Minutes. In arXiv:1711.04325, 2017.

[11] Yang You, Zhao Zhang, Cho-Jui Hsieh, James Demmel, and Kurt Keutzer. ImageNet Training in Minutes. In arXiv:1709.05011, 2018.

[12] Sriram Subramanian. Modern AI Stack & AI as a Service Consumption Models: https://medium.com/clouddon/modern-ai-stack-ai-service-consumption-models-f9957dce7b25.

[13] Song Han, Jeff Pool, John Tran, and William Dally. Learning both Weights and Connections for Efficient Neural Network. In NIPS, 2015.

[14] Song Han, Huizi Mao, and William J. Dally. Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding. In arXiv:1510.00149, 2015.

[15] Jiecao Yu, Andrew Lukefahr, David Palframan, Ganesh Dasika, Reetuparna Das, and Scott Mahlke. Scalpel: Customizing DNN Pruning to the Underlying Hardware Parallelism. In ISCA, 2017.

[16] Backpropagation In Convolutional Neural Networks: http://www.jefkine.com/general/2016/09/05/backpropagation-in-convolutional-neural-networks/.

[17] BackPropagation Through Time: https://pdfs.semanticscholar.org/c77f/7264096cc9555cd0533c0dc28e909f9977f2.pdf.

[18] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. In NIPS, 2012.

[19] Sebastian Ruder. An overview of gradient descent optimization algorithms. In arXiv:1609.04747, 2017.

[20] Srimat Chakradhar, Murugan Sankaradas, Venkata Jakkula, and Srihari Cadambi. A dynamically configurable coprocessor for convolutional neural networks. In ISCA, 2010.

[21] Zidong Du, Robert Fasthuber, Tianshi Chen, Paolo Ienne, Ling Li, Tao Luo, Xiaobing Feng, Yunji Chen, and Olivier Temam. ShiDianNao: shifting vision processing closer to the sensor. In ISCA, 2015.

[22] Chen Zhang, Peng Li, Guangyu Sun, Yijin Guan, Bingjun Xiao, and Jason Cong. Optimizing FPGA-based Accelerator Design for Deep Convolutional Neural Networks. In FPGA, 2015.

[23] Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A. Horowitz, and William J. Dally. EIE: Efficient Inference Engine on Compressed Deep Neural Network. In ISCA, 2016.

[24] Wenyan Lu, Guihai Yan, Jiajun Li, Shijun Gong, Yinhe Han, and Xiaowei Li. FlexFlow: A Flexible Dataflow Accelerator Architecture for Convolutional Neural Networks. In HPCA, 2017.

[25] Yu-Hsin Chen, Tushar Krishna, Joel S. Emer, and Vivienne Sze. Eyeriss: An Energy-Efficient Reconfigurable Accelerator for Deep Convolutional Neural Networks. In Journal of Solid-State Circuits, 2017.

[26] Shijin Zhang, Zidong Du, Lei Zhang, Huiying Lan, Shaoli Liu, Ling Li, Qi Guo, Tianshi Chen, and Yunji Chen. Cambricon-X: An accelerator for sparse neural networks. In MICRO, 2016.

[27] Angshuman Parashar, Minsoo Rhu, Anurag Mukkara, Antonio Puglielli, Rangharajan Venkatesan, Brucek Khailany, and Joel Emer. SCNN: An Accelerator for Compressed-sparse Convolutional Neural Networks. In ISCA, 2017.

[28] Swagath Venkataramani, Ashish Ranjan, Subarno Banerjee, Dipankar Das, Sasikanth Avancha, Ashok Jagannathan, Ajaya Durg, Dheemanth Nagaraj, Bharat Kaul, Pradeep Dubey, and Anand Raghunathan. ScaleDeep: A Scalable Compute Architecture for Learning and Evaluating Deep Networks. In ISCA, 2017.

[29] Charles E. Leiserson. Fat-trees: Universal networks for hardware-efficient supercomputing, Trans. Comput., 1985.

[30] Yann LeCun, L.D. Jackel, B. Boser, J.S. Denker, H.P. Graf, I. Guyon, D. Henderson, R.E. Howard, and W. Hubbard. Handwritten digit recognition: applications of neural network chips and automatic learning, Commun. Mag., 1989.

[31] Min Lin, Qiang Chen, and Shuicheng Yan. Network in network. In ICLR, 2014.

[32] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep Residual Learning for Image Recognition. In CVPR, 2016.

[33] Jason Yosinski, Jeff Clune, Yoshua Bengio, and Hod Lipson. How transferable are features in deep neural networks? In NIPS, 2014.

[34] Haşim Sak, Andrew Senior, and Françoise Beaufays. Long Short-Term Memory Recurrent Neural Network Architectures for Large Scale Acoustic Modeling. In INTERSPEECH, 2014.

[35] 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 ACM Multimedia, 2014.

[36] Daniel Povey, Arnab Ghoshal, Gilles Boulianne, and Lukas Burget. The Kaldi Speech Recognition Toolkit, 2011.

[37] Song Han, Junlong Kang, Huizi Mao, Yiming Hu, Xin Li, Yubin Li, Dongliang Xie, Hong Luo, Song Yao, Yu Wang, Huazhong Yang, and William (Bill) J. Dally. ESE: Efficient Speech Recognition Engine with Sparse LSTM on FPGA. In FPGA, 2017.

[38] WattsUp: https://www.wattsupmeters.com/.

[39] Linqi Huang, Jun Lic, Hong Haob, and Xibing Lia. Micro-seismic event detection and location in underground mines by using Convolutional Neural Networks (CNN) and deep learning, Tunn. Undergr. Sp. Technol., 2018.

[40] Bin Liu, Yun Zhang, DongJian He, and Yuxiang Li. Identification of Apple Leaf Diseases Based on Deep Convolutional Neural Networks, Symmetry (Basel)., 2018.

[41] Jon Jordan. Opinion: Why your game needs to load within 30 seconds: https://www.pocketgamer.biz/monetizer/59041/opinion-why-your-game-needs-to-load-within-30-seconds/.

[42] Jorge Albericio, Alberto Delmás, Patrick Judd, Sayeh Sharify, Gerard O'Leary, Roman Genov, and Andreas Moshovos. Bit-pragmatic deep neural network computing. In MICRO, 2017.

[43] Hardik Sharma, Jongse Park, Naveen Suda, Liangzhen Lai, Benson Chau, Joon Kyung Kim, Vikas Chandra, and Hadi Esmaeilzadeh. Bit Fusion: Bit-Level Dynamically Composable Architecture for Accelerating Deep Neural Networks. In ISCA, 2018.

[44] Animesh Jain, Amar Phanishayee, Jason Mars, Lingjia Tang, and Gennady Pekhimenko. Gist: Efficient Data Encoding for Deep Neural Network Training. In ISCA, 2018.

[45] Xuda Zhou, Zidong Du, Qi Guo, Shaoli Liu, Chengsi Liu, Chao Wang, Xuehai Zhou, Ling Li, Tianshi Chen, and Yunji Chen. Cambricon-S: Addressing Irregularity in Sparse Neural Networks through A Cooperative Software/Hardware Approach. In MICRO, 2018.

[46] Jose M. Alvarez and Mathieu Salzmann. Compression-aware Training of Deep Networks. In arXiv:1711.02638, 2017.

[47] Caiwen Ding, Siyu Liao, Yanzhi Wang, Zhe Li, Ning Liu, Youwei Zhuo, Chao Wang, Xuehai Qian, Yu Bai, Geng Yuan, Xiaolong Ma, Yipeng Zhang, Jian Tang, Qinru Qiu, Xue Lin, and Bo Yuan. CirCNN: accelerating and compressing deep neural networks using block-circulant weight matrices. In MICRO, 2017.

[48] Hyoukjun Kwon, Ananda Samajdar, and Tushar Krishna. MAERI: Enabling Flexible Dataflow Mapping over. DNN Accelerators via Reconfigurable Interconnects. In ASPLOS, 2018.

[49] Minsoo Rhu, Mike O'Connor, Niladrish Chatterjee, Jeff Pool, and Stephen W. Keckler. Compressing DMA Engine: Leveraging Activation Sparsity for Training Deep Neural Networks. In HPCA, 2018.

[50] Youngeun Kwon and Minsoo Rhu. Beyond the Memory Wall: A Case for Memory-centric HPC System for Deep Learning. In MICRO, 2018.

[51] Youjie Li, Jongse Park, Mohammad Alian, Yifan Yuan, Zheng Qu, Peitian Pan, Ren Wang, Alexander Gerhard Schwing, Hadi Esmaeilzadeh, and Nam Sung Kim. A Network-Centric Hardware/Algorithm Co-Design to Accelerate Distributed Training of Deep Neural Networks. In MICRO, 2018.

[52] Alex Krizhevsky. Learning Multiple Layers of Features from Tiny Images, 2009.

[53] John S. Garofolo, Lori F. Lamel, William M. Fisher, Jonathan G. Fiscus, David S. Pallett, Nancy L. Dahlgren, and Victor Zue. TIMIT Acoustic-Phonetic Continuous Speech Corpus LDC93S1. In Linguistic Data Consortium, 1993.