ABSTRACT
The widespread popularity of deep neural networks (DNNs) has made it an important workload in modern datacenters. Training DNNs is both computation-intensive and memory-intensive. While prior works focus on training parallelization (e.g., data parallelism and model parallelism) and model compression schemes (e.g., pruning and quantization) to reduce the training time, choosing an appropriate data layout for input feature maps also plays an important role and is considered to be orthogonal to parallelization and compression in delivering the overall training performance. However, finding an optimal data layout is non-trivial since the preferred data layout varies depending on different DNN models as well as different pruning schemes that are applied. In this paper, we propose a simple-yet-effective data layout arbitration framework that automatically picks up the beneficial data layout for different DNNs under different pruning schemes. The proposed framework is built upon a formulated cache estimation model. Experimental results indicate that our approach is always able to select the most beneficial data layout and achieves the average training performance improvement with 14.3% and 3.1% compared to uniformly using two popular data layouts.

CCS CONCEPTS
- Software and its engineering → Compilers; - Computer systems organization → Multicore architectures.

KEYWORDS
deep neural networks, data layout, cache model

1 INTRODUCTION
In recent years, Deep Neural Networks (DNNs) gain momentum in a wide spectrum of applications, ranging from image classification to natural language processing. However, training DNN models is very time-consuming and challenging on modern computing platforms due to its significant computation intensity and memory intensity. For example, BERT training requires 81 hours on 16 Google TPUs [6] and it takes more than 40 days to train an AlphaGo Zero system [31]. Consequently, training DNN models is widely deployed in datacenters and becomes a significant fraction of the datacenter workload. However, as the success of the state-of-the-art DNN models relies on a large number of model parameters (e.g., layers) and batch sizes, model training remains very time-consuming and is one of the major bottlenecks that prevent the wide adoption of personalized DNNs.

To reduce training time, prior works have investigated parallelization [17, 25] (e.g., data parallelism and model parallelism) and model compression [3] (e.g., pruning and quantization) for DNN training on datacenters. In particular, pruning effectively accelerates training performance with negligible impact on training accuracy. The widely adopted pruning schemes include the shape pattern-based shape pruning, filter pruning, channel pruning, and kernel pruning [20, 27, 32]. While those optimizations effectively reduce the training time, choosing an appropriate data layout of input feature maps is also important and has received little attention in the literature. There are two popular layouts: NCHW and NHWC, where \( N \) represents the batch size, \( C \) represents the number of Channels, \( H \) represents the height, and \( W \) represents the width (discussed in detail in Section 2). In general, NHWC is known to provide better performance on CPUs because of vectorization, whereas NCHW is better on GPUs due to expensive reduction operations [1, 29]. However, in this paper, we find that the preferred data layout varies when different pruning schemes have been applied. This is because of the poor data locality exhibited in caches after pruning. To be more specific, caches are organized at cacheline granularity and the capability to take the spatial locality within cachelines can significantly affect the performance. After pruning the weights, certain portions of the values in the feature maps are
not used and can be skipped without loading to the cache. However, since caches are managed by hardware in cacheline granularity, those skipped values may mix with other values in the same cachelines, leading to cache thrashing and under-utilization of cachelines. On the other hand, a different layout may have those skipped values clustered in the same cachelines such that those cachelines are not accessed and are not loaded to the cache, reducing the probability of cache thrashing.

Motivated by this observation, in the paper, we propose a data layout arbitration framework that automatically picks up the beneficial data layout for training DNNs under different pruning schemes. The framework is built upon a proposed cache model that estimates the cache performance under different data layouts and pruning schemes. The cache model uses static parameters (e.g., input feature map dimensions, filter dimension, and pruning pattern) without the need of profiling, nor introducing any runtime overheads. The main contributions of this paper are as follows:

- It investigates the training execution time under different data layouts and different pruning schemes. The observation is that the preferred data layout can be different for different DNNs, even the same DNN with different pruning schemes being applied.
- It proposes a cache estimation model that is able to effectively predict the cache performance under different execution scenarios. The cache model undertakes the DNN model parameters, the data layout, and the pruning strategy to estimate the cache performance. The cache model is completely static without any runtime overheads and does not require any profiling.
- It implements an auto-arbitration framework based on the proposed cache model. The framework automatically selects the beneficial data layout to avoid loading pruned parameters into the cache, achieving significant training performance improvement without compromising the model accuracy.
- It evaluates the proposed work on 5 DNN models with 4 different pruning strategies. Experimental results indicate that our approach accurately selects the optimal layout without the need of trial-and-error searching. It achieves training speedup with an average of 14.3% and 3.1%, comparing to uniformly using NHWC layout and NCHW layout, respectively.

### Table 1: Parameters in cache model.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>N</td>
<td>Batch size</td>
<td>C</td>
<td>Number of Channels</td>
</tr>
<tr>
<td>H</td>
<td>Height of input fmap</td>
<td>W</td>
<td>Width of input fmap</td>
</tr>
<tr>
<td>H'</td>
<td>Height of output fmap</td>
<td>W'</td>
<td>Width of output fmap</td>
</tr>
<tr>
<td>S</td>
<td>Height of filter</td>
<td>S</td>
<td>Width of filter</td>
</tr>
<tr>
<td>STRIDE</td>
<td>Filters stride</td>
<td>cCAP</td>
<td>Cache capacity</td>
</tr>
<tr>
<td>n</td>
<td>Number of cachelines</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

2.2 Pruning Strategies

Pruning is a promising approach for DNNs compression and acceleration by eliminating redundant/unnecessary model parameters with negligible accuracy drop. In this paper, we consider four types of state-of-the-art structured pruning schemes: shape pruning, filter pruning, channel pruning, and kernel pruning [11, 12, 28, 30]. In shape pruning, certain portions of the filters are pruned. Note that, the portions are identical across all the filters. For example, in Figure 2(a), the bottom-left and the top-right value in the first channel, and the middle-right of the last channel for all filters are pruned. Figure 2(a) also shows the 2D flat weight matrix format representation of the filters after pruning. Figures 2(b) and (c) show filter pruning and channel pruning which prunes the entire filter(s)/channel(s). In the weight matrix format representation, filter pruning corresponds to reducing one row of the weight matrix and it is also termed as row pruning. Accordingly, channel pruning corresponds to reducing multiple consecutive columns in the weight matrix format. Finally, in Figure 2(d), we show kernel pruning. Unlike channel pruning and filter pruning, kernel pruning does prunes each filter separately. That is, it does not maintain a uniform pattern across the filters (as can be observed from the 2D weight matrix). The key advantage of structured pruning is that a full matrix will be maintained with dimension reduction, thereby facilitating hardware acceleration [27, 28].

2.3 Motivation

Although a general sense is that the NHWC provides better performance on CPUs, this may no longer hold true when the DNN model is pruned during training. This is because when some parameters are pruned, certain portions of the values in the feature maps will not be used in the computation. These values can be potentially skipped without loading to the cache to reduce the cache thrashing. However, since caches are managed at cacheline granularity, those skipped values may mix with other values in the same cachelines. As a result, those skipped values are still loaded to the cache, introducing cache thrashing. Fortunately, a different data layout can help mitigate this problem. Specifically, a different layout can have those skipped values clustered in the same cachelines such that those cachelines are not accessed and are not loaded to the cache, reducing the probability of cache thrashing. This observation motivates us to explore a layout arbitration framework that

---

1In this paper, we use the term “filter” and “weight” interchangeably.
automatically picks up the beneficial data layout for DNNs when different pruning schemes are applied.

3 OUR APPROACH
Our goal: The goal of this paper is to automatically determine the optimal data layout between NCHW and NHWC for pruned DNN training. There are two challenges to achieving the goal. First, the preferred data layout varies according to the type of DNN models and the pruning strategy applied. It requires a comprehensive model to estimates the performance benefits when different data layouts are applied during training. Second, pruning strategies are generally applied to the model parameters (i.e., weights). It is not intuitive how the pruned weights affect the data layout of feature maps. Motivated by these challenges, in this section, we propose a cache model that undertakes the static training configurations (i.e., input feature map parameters, pruning schemes, and pruning rate) to determine the beneficial data layout between NCHW and NHWC. All the symbols used in our cache model are listed in Table 1.

3.1 Overview
Figure 3 shows a high-level workflow of our proposed data layout arbitration framework. Overall, the cache model (i.e., predictor) uses static parameters (e.g., input feature map dimensions, filter dimension, pruning pattern, and hardware configuration) as inputs to estimate the cache performance under different data layouts and pruning schemes. This cache model is statically offline without the need for profiling, nor does it introduce any runtime overheads. The cache model helps to find the beneficial data layout. Taking the cache model prediction outputs as the configuration for pruned-DNN training, our framework automatically identifies the pruned parameters based on the index, avoiding loading the pruned data to cache and unnecessary computations. In the following section, we will introduce our cache model in detail.

3.2 Cache Model
3.2.1 Total Number of Memory Accesses. To estimate the cache miss rate, we first need to calculate the total number of memory requests during the course of convolution computation. Specifically, each value in the output feature map involves convolution of the input feature map and the filter. Let us assume the output feature map dimension is $H'$ height and $W'$ width, we have

$$\text{Output Height (H')} = \frac{H - R}{\text{Stride}} + 1$$

(1)

$$\text{Output Width (W')} = \frac{W - S}{\text{Stride}} + 1$$

(2)

where the $H$ and $W$ represent the dimension of the input feature map, and $R$ and $S$ represent the filter dimension. The Stride represents the convolution stride when applying filters on the input.
feature map. As a result, one can have the total number of memory accesses calculated as:

$$Total Access = N \times H' \times W' \times R \times S \times C$$

$$= N \times \left(\frac{H - R}{Stride + 1}\right) \times \left(\frac{W - S}{Stride + 1}\right) \times R \times S \times C$$  \hspace{1cm} (3)

3.2.2 Cache Capacity. Let us assume that the cache in the system has \( n \) cachelines with \( l \) cacheline size. Then, the cache capacity is:

$$Cache Capacity (cCAP) = l \times n$$  \hspace{1cm} (4)

3.2.3 NCHW data layout. Now, let us first construct the model when the feature map is stored as NCHW layout in memory. In this case, the width \( W \) is the fastest variance dimension of the array, then height \( H \) and channel \( C \). We assume the width of the feature map is larger than one cacheline (i.e., \( W > l \)).

In the ideal scenario, the cache capacity is large enough to hold the entire working set (including input feature maps and the filters) of the convolution process. In such a case, there exist only cold misses to bring the data to cache from memory. However, in practice, cache capacity is generally much smaller than the size of the input feature maps. Recall our discussion about convolution in Section 2, a filter is applied to the input feature map in a "sliding stride" manner. That is, the filter is applied to the sub-portions of the input feature map left-to-right and top-to-bottom based on the stride. We call one left-to-right convolution as a row and one top-to-bottom as a column. Depending on the stride and cache capacity, it can happen that when the filter moves to the second row of the feature map, the cachelines containing input feature data have been already evicted from the cache and the same cachelines have to be loaded back to the cache, leading to poor cache performance. In the worst case, all the cachelines that are reused during the column convolution have been evicted by the time reuse happens. Specifically, the amount of data involved in each row convolution can be calculated as \((C \times R \times W)\) by using the number of channels \((C)\), the height of the filter \((R)\), and the width of the feature map \((W)\). Depending on the cache capacity, several scenarios can occur. First, if the cache capacity is smaller than \( C \times R \times W - C \times l \), the cache cannot hold the last \( C \times l \) data for one row convolution. As a result, the cacheline that contains the data accessed by the first row convolution might have been evicted from the cache when the second row convolution reuses the same data, leading to additional cache misses for the second row convolution.

Now, let us formulate the cache misses. We first calculate the total number of cachelines that miss in the cache during convolution. For one row of the convolution, the number of cacheline misses can be calculated as \((N \times C \times R \times W)/l\). Since the entire convolution has \( H - R + 1 \) rows, the total number of misses is \((N \times C \times R \times W) \times (H - R + 1)/l\). Therefore, the miss rate in this scenario is:

$$When \, cCAP < C \times R \times W - C \times l, \ and \ l < W,$$

$$Miss Rate = \frac{N \times C \times R \times W \times (H - R + 1)}{l \times TotalAccess}$$  \hspace{1cm} (5)

Where the \( TotalAccess \) is captured in Eq 3.

While Eq 5 captures the unfortunate scenarios where the reuses in the subsequent row convolutions miss the cache upon reuse, those reuses can be captured by the cache with a larger capacity. In such cases, fewer cachelines are needed to be loaded since some of the cachelines still reside in the cache when the reuses happen. Specifically, if the cache capacity is larger than one row of convolution, there are no conflict misses. That is:

$$\begin{align*}
When \, cCAP \geq C \times R \times W - C \times l, \\
cCAP < N \times C \times H \times W, \ and \ l < W, \\
Miss Rate = \frac{N \times C \times H \times W}{1 \times TotalAccess} \hspace{1cm} (6)
\end{align*}$$

3.2.4 NHWC data layout. Recall our discussion in Section 2, unlike NCHW data layout, channel is the fastest variance dimension in the NHWC format. Let us call the convolution along the channel dimension as “aisles”. Similar to the case of NCHW format, additional cache misses occur when the convolution operates from one aisle to another if the cache capacity is not enough to hold the whole data of one aisle convolution. Specifically, when the cache capacity \((cCAP)\) is smaller than \( R \times S \times C - R \times l \), and the cacheline size \( l \) is smaller than the number of channels \((C)\), i.e., a single cacheline is not able to hold all the values in channel, the cachelines reused by the subsequent aisles have already been evicted from the cache upon reuse, leading to extra cache misses. To be more concrete, the product of weightâ€™s height, weightâ€™s width, and the number of weight channels \((R \times S \times C)\) equals to the number of total data in the feature map accessed in one aisle. When cache capacity is smaller than \( R \times S \times C - R \times l \), it cannot hold the last \( R \times l \) data for one aisle. Thus, it evicts the oldest cachelines based on LRU policy to make room for the \( R \) cachelines. These replaced cacheline contain those that will be accessed in the next aisle of convolution, leading to additional capacity misses for future computations.

To formulate this scenario, the cacheline misses for one aisle equals to the product of \((N \times R \times S \times C)/l\). Consider all the aisles, the total number of cacheline misses is the product of per-aisle misses, the distance that the weights move along with the height of the feature map \((H - R + 1)\), and the distance that the weights move along with the width of the feature map \((W - S + 1)\). Therefore, the miss rate can be calculated as:

$$\begin{align*}
When \, cCAP < R \times S \times C - R \times l, \ and \ l \leq C, \\
Miss Rate = \frac{N \times R \times S \times C \times (H - R + 1) \times (W - S + 1)}{l \times TotalAccess} \hspace{1cm} (7)
\end{align*}$$

where the \( TotalAccess \) is derived from Eq 3.

It can happen that the cacheline size \( l \) is larger than channel size, especially at the beginning of deep neural networks (e.g., generally three-channels of input images and three-channels of weights). In such a scenario, the cacheline size \( l \) is greater than the number of channels \((C)\), and data at different channels and different heights but the same width may reside in the same cacheline. For the worst case, cacheline blocks that will be accessed first in the new row of convolution have already been replaced by cacheline blocks that will not be used in the recent future. When loading these blocks back to the cache, other blocks that will be reused in the next aisle need to be replaced by the time reuse happens.

We use \( 1/2 \) to represent the percent of data of all channels with the same height and width that reside a cacheline. In this case, \( 1/2 \) is always smaller than 1 since \( l > C \). The number of cachelines which stores data in one aisle is \( \left\lceil \frac{W}{l} \times R \times S \right\rceil \). The total number of cacheline misses is a product of \( \left\lceil \frac{W}{l} \times R \times S \right\rceil \), the distance the weight moves along with the height of the feature map \((H - R + 1)\), and the distance the weight moves along with the width of the feature
We now use several examples to explain the pruned pattern in input weights, a significant subset of values in the feature maps will not be accessed during convolution. As a result, such “pruned” values can bring significant under-utilization of cachelines. In this section, we use several examples to explain the pruned pattern in input feature maps and show the cache inefficiencies caused by weight pruning.

Figure 4 shows the example of applying kernel pruning under two different data layouts. We use the red box to denote one cacheline. Note that, the cacheline size can vary based on different platform configurations. In Figure 4(a), the input feature maps are stored as NCHW layout in the memory. The pruned values in the filters are labeled using white color, whereas the rest non-pruned values are labeled using blue colors. During convolution, the computation involves those pruned weights are skipped since the value of those weights is 0s. As a result, values at the corresponding location in the feature map will not be used in the intermediate steps of computation. However, since the cachelines are managed hardware and the minimum granularity in cache swapping is cacheline, it can happen that some cachelines are severely under-utilized because of pruning. In the example in Figure 4(a), the first, third, and last channels of filter one were pruned. Thus, values in the first, third, and last channels of the input feature maps are not used during convolution. Therefore, NCHW data layout allows most of the skipped values to be clustered in the same cacheline (as shown by the red block). Those cachelines are not loaded to the cache avoiding cache thrashing. Meanwhile, those cachelines that are loaded to the cache have a high utilization as most of the data in the cacheline are not pruned. On the contrary, when the input feature maps are stored in NHWC data layout (shown in Figure 4(b)), the cacheline utilization is lower.

Figure 5 shows the example of applying shape pruning under two different data layouts. In the example in Figure 5(a), the input feature maps are stored in NCHW layout. The shape pruned in every filter is the left-bottom corner and the top-right corner. Therefore, the intermediate convolution on feature maps will skip the values denoted using the white area. A cacheline is represented using the red box. In this case, every cacheline will include the skipped values, leading to poor cache performance. On the contrary, when the input feature maps are stored as NHWC layout as shown in Figure 5(b), one can find contiguous skipped values clustered in the same cacheline (as denoted by the red box in the intermediate convolution). This means that these cachelines are not loaded to the cache, yielding less cache thrashing and better performance. However, we cannot simply assume that shape pruning always prefers NHWC layout. It is also affected by the model parameters (e.g., feature map dimensions, filter dimensions, different shapes, etc). Our cache model takes all these variances into consideration and always makes the prediction of a beneficial layout.

### 3.3 Example

With the proposed cache model and equations, we now use several examples to show how pruning affects cache utilization under different data layouts of input feature maps. In particular, we consider the four pruning strategies discussed in Section 2. The key observation is that, after applying different pruning strategies on weights, a significant subset of values in the feature maps will not be accessed during convolution. As a result, such “pruned” values can bring significant under-utilization of cachelines. In this section, we use several examples to explain the pruned pattern in input feature maps and show the cache inefficiencies caused by weight pruning.

Figure 4 shows the example of applying kernel pruning under two different data layouts. We use the red box to denote one cacheline. Note that, the cacheline size can vary based on different platform configurations. In Figure 4(a), the input feature maps are stored as NCHW layout in the memory. The pruned values in the filters are labeled using white color, whereas the rest non-pruned values are labeled using blue colors. During convolution, the computation involves those pruned weights are skipped since the value of those weights is 0s. As a result, values at the corresponding location in the feature map will not be used in the intermediate steps of computation. However, since the cachelines are managed hardware and the minimum granularity in cache swapping is cacheline, it can happen that some cachelines are severely under-utilized because of pruning. In the example in Figure 4(a), the first, third, and last channels of filter one were pruned. Thus, values in the first, third, and last channels of the input feature maps are not used during convolution. Therefore, NCHW data layout allows most of the skipped values to be clustered in the same cacheline (as shown by the red block). Those cachelines are not loaded to the cache avoiding cache thrashing. Meanwhile, those cachelines that are loaded to the cache have a high utilization as most of the data in the cacheline are not pruned. On the contrary, when the input feature maps are stored in NHWC data layout (shown in Figure 4(b)), the cacheline utilization is lower.

### 4 EVALUATION

#### 4.1 Evaluation Methodology

We use five DNN models with the aforementioned four pruning schemes to evaluate our approach. The five DNN models include VGG16, ResNet18, MobileNet, DenseNet, and ResNet50. We use Cifar-10 dataset [15] to train each network with 200 epochs. We use Tensorflow v1.14.0 [1] to train each network with 200 epochs. The pruning rate is set to 4× according to previous paper [21]. All models are trained using Tensorflow v1.14.0 [1]. Table 2 list the training parameters for all the five DNN models. Note that, we only prune the convolutional layers.

We use an internal server to evaluate our approach. The system is equipped with Intel(R) Xeon(R) Silver 4114 CPU working at 2.20GHz. The system has three-level caches and the cacheline size is 64 bytes. Specifically, the level 1 cache (L1) consists of 32 KB 8-way associative instruction cache and 32 KB 8-way set associative data cache. The level 2 cache (L2) is 1 MB with 16-way set associativity. The level 3 cache (L3) is a 13.75 MB non-inclusive shared cache. We test our proposed arbitration framework using the aforementioned four pruning schemes. The performance results are end-to-end training execution times.
Table 2: Training parameters.

<table>
<thead>
<tr>
<th>DNN models</th>
<th>VGG16</th>
<th>ResNet18</th>
<th>MobileNetV2</th>
<th>ResNet50</th>
<th>DenseNet121</th>
</tr>
</thead>
<tbody>
<tr>
<td>Batch Size</td>
<td>128</td>
<td>128</td>
<td>128</td>
<td>128</td>
<td>128</td>
</tr>
<tr>
<td>Training Epochs</td>
<td>200</td>
<td>200</td>
<td>200</td>
<td>200</td>
<td>200</td>
</tr>
<tr>
<td>Original Model Size</td>
<td>14.53G</td>
<td>10.65G</td>
<td>2.16G</td>
<td>22.38G</td>
<td>6.55G</td>
</tr>
<tr>
<td>Pruned Model Size</td>
<td>4.01G</td>
<td>2.67G</td>
<td>0.55G</td>
<td>5.61G</td>
<td>1.65G</td>
</tr>
<tr>
<td>Pruning Rate</td>
<td>4x</td>
<td>4x</td>
<td>4x</td>
<td>4x</td>
<td>4x</td>
</tr>
</tbody>
</table>

Figure 6: Overall training execution time.

Figure 7: Cache miss rates of three-level caches.

4.2 Results

Figure 6 plots the execution time of training 5 DNN models under different pruning schemes and different data layouts. Since kernel pruning is not applicable to MobileNetV2 and channel pruning is not applicable to MobileNetV2, DenseNet121, and ResNet50. Figure 6 only shows results for the rest pruning schemes for those models.

One can make the following observations. First, our cache model is always able to predict the beneficial data layout for different DNNs and different pruning schemes. Compared to uniformly using NHWC layout and NCHW layout, our approach improves the average training performance with 14.3% and 3.1%, respectively. Second, all experimental results of filter pruning, channel pruning, and kernel pruning recommend NCHW layout as it provides lower training time. Recall our discussion in Section 3.3, NCHW layout avoids loading those skipped data in feature maps to cache, leading to better performance. Channel pruning is a special case of kernel pruning since we prune the same channels for different filters. Filter pruning is a special case of channel pruning because we randomly prune filters. Therefore, these three pruning strategies share a similar observation and have consistent data layout preference. Third, for shape pruning, it shows diverse results. While VGG16, ResNet18, ResNet50, and DenseNet121 prefers NHWC format, MobileNetV2 is an exception and prefers NCHW format. The reason behind this is twofold. First, MobileNetV2 uses a different pruning shape compared to the example in Figure 5. Second, the dimension of feature maps is comparably smaller than other networks. Consequently, one cacheline is able to capture more than one entire channels. Therefore, those skipped values will also be included in the cachelines.

To further understand the performance gains, we show the cache miss rates of three-level caches in Figure 7. We used vTune to profile the training and obtain the miss rates. Comparing Figure 7 and the first two bars in Figure 6, one can observe that the L1 cache miss rates consistently match with the execution times. We also show the estimated L1 cache miss rates in Figure 8. Specifically, we use L1 cache capacity in our model and predict the cache miss rates under different data layouts. As one can observe, though the absolute values have a large discrepancy compared to the value in Figure 7, the trend predicted matches with Figure 7. That is, our model is always able to predict the beneficial data layout for different DNNs under different pruning schemes.

It is also important to emphasize that, the cache model based data layout prediction does not affect the training accuracy at all. We observe the exact same training accuracy in both NHWC and NCHW formats.

5 RELATED WORK

DNN parallelization: Previous works investigated data parallelism optimizations to improve the training performance [2, 10, 13, 17–19, 23, 26]. Kumar et al. [16] scaled the ML models to 4k-chip Google TPU-v3 machines and explored model parallelism to address the scaling limitations in data parallelism. They also optimized communication, investigated distributed evaluation of training metrics, and improved host input processing scaling. Krizhevsky et al. [14] proposed hybrid parallelism where data parallelism is used for convolutional/pooling layers and model parallelism is used for fully-connected layers. Goyal et al. [8] developed a three-step all-reduce operation to optimize communication across parallel devices. Their approach also overlaps gradient synchronization with backward propagation. Pal et al. [25] explored hybrid parallelization to overcome the statistical efficiency losses introduced by data-parallel at scale. Jia et al. [13] developed a mechanism to partition the tensor along multiple dimensions and then searches for the best parallelization strategy for each partition.

DNN pruning: Pruning is a widely-used approach in modern DNN models to significantly reduce DNN execution time by removing unnecessary computation and memory access while maintaining the accuracy. At a high level, various pruning strategies can be classified as unstructured pruning [4, 5, 7, 9], structured pruning [11, 12, 28, 30], and pattern-based pruning [22, 24]. Although unstructured pruning has the advantage of maintaining accuracy, it brings sparsity and irregularity in weight matrices, and as a result, extra indices are used to index the non-zero weights in the sparse matrix storage format (e.g., CSR format). In contrast, the major advantage of structured pruning is that a full matrix is maintained with dimension reduction, thereby facilitating hardware acceleration. Pattern-based pruning is guided by compilers but requires specific pre-determined patterns in order to leverage the hardware parallelization.
While most of the parallelization strategies focus on computation scheduling and placement, the optimized data layout brings further performance improvements by improving the cache performance. More importantly, the beneficial layout choice is no longer obvious when different pruning strategies are being applied. As such, the proposed auto-arbitration framework built upon the cache model can be combined with existing parallelization strategies and pruning schemes to further boost the training performance.

6 CONCLUSION AND FUTURE WORKS

Training DNN models is an important workload in datacenters and it is time-consuming and resource-demanding. While parallelization (e.g., data parallelism and model parallelism) strategies and pruning schemes effectively reduce the training time, the data layout of the input feature maps also plays a significant role in shaping the overall training performance. In this paper, we propose a data layout arbitration framework that is built upon a formulated cache model to estimate the impact of different data layouts of input feature maps, especially under different pruning schemes. Experimental results on five DNN models and four different pruning schemes indicate that our approach achieves an average of 14.3% and 3.1% training time reduction, comparing to uniformly using NHWC layout and NCHW layout, respectively.

While the cache model in this paper is effective in capturing the beneficial data layout between NCHW and NHWC, it also paves at least two research avenues. First, it is possible that neither NCHW nor NHWC provides the optimal training performance and a transformed new layout (e.g., NHWFC) is preferred. This is because, i) neither data layout may not be able to completely eliminate the unused entries in the cachelines, leading to cacheline under-utilization and cache flushing, and ii) different computation kernels in training prefer different layouts. To this end, we plan to expand the model to identify optimal data layouts in training. Second, we are expanding the model to GPUs with tensor cores support. The observation is that the preferred layout on GPUs and tensor cores is different from on the CPUs.

ACKNOWLEDGMENTS

This work is supported in part by NSF grants #2011146, a startup funding from the University of Pittsburgh, and the Pitt momentum seed grant.

REFERENCES