Accelerating DNN Training with Structured Data Gradient Pruning

Bradley McDanel
Franklin & Marshall College

Helia Dinh
Franklin & Marshall College

John Magallanes
Franklin & Marshall College

Abstract—Weight pruning is a technique to make Deep Neural Network (DNN) inference more computationally efficient by reducing the number of model parameters over the course of training. However, most weight pruning techniques generally does not speed up DNN training and can even require more iterations to reach model convergence. In this work, we propose a novel Structured Data Gradient Pruning (SDGP) method that can speed up training without impacting model convergence. This approach enforces a specific sparsity structure, where only N out of every M elements in a matrix can be nonzero, making it amenable to hardware acceleration. Modern accelerators such as the Nvidia A100 GPU support this type of structured sparsity for 2 nonzeros per 4 elements in a reduction. Assuming hardware support for 2:4 sparsity, our approach can achieve a 15-25% reduction in total training time without significant impact to performance. Source code and pre-trained models are available at https://github.com/BradMcDanel/sdgp.

I. INTRODUCTION

Deep Neural Networks (DNNs) are now widely used for many applications, such as computer vision, speech recognition, and natural language processing. However, their large number of parameters and associated computational complexity makes DNN training expensive. Most prior research on reducing DNN computation costs focus on DNN inference. For instance, quantization [18] can reduce the number of bits required to represent model parameters, allowing for more efficient storage and faster execution. Weight pruning [6] is another technique that can reduce the number of model parameters, by setting some parameters to zero during training. However, while these approaches can lead to more efficient inference, they often make DNN training less efficient. For instance, weight pruning often requires a longer training regime to reach convergence after multiple rounds of pruning [5].

In this work, we propose a novel structured data gradient pruning (SDGP) technique to speed up DNN training. Our approach is based on the observation that modern hardware accelerators, such as the Nvidia A100 GPU, support a specific type of structured sparsity, where only 2 out of every 4 elements in a matrix can be nonzero. We enforce this sparsity structure on the activations gradients during training, in order to achieve a 2× reduction in operations required to compute the input gradients for the previous layer. Figure 1 provides an overview of how SDGP prunes data gradients. During backpropagation, the output data gradients from the preceding layer are pruned by setting some elements to zero, before computing the input data gradients for the layer below. This approach reduces the number of matrix operations required to compute the input gradients, which can speed up training. We find that using SDGP achieves comparable performance (i.e., classification accuracy) to using dense data gradients.

In order for SDGP to minimally impact training stability, the pruned gradients must still closely approximate the original gradients. To this end, we evaluate multiple variants of SDGP that use different ranking mechanism to determine which elements in a group should be pruned. Borrowing from prior work on structured weight pruning [30], we propose three ranking schemes: (1) random pruning, (2) a greedy ranking scheme that prunes the smallest magnitude elements, and (3) a version of the greedy scheme that adjusts the remaining nonzero elements to better approximate the original distribution.

The A100 only supports a 50% sparsity ratio (with n = 2 and m = 4). In this work, we also evaluate higher sparsity ratios than supported by the A100 (e.g., 87.5% sparse with n = 2 and m = 16) in order to see how the proposed approach scales with increased sparsity. Generally, higher sparsity ratios lead to worse performance, but further reduce the training time. In section IV-C, we provide a Time-To-Accuracy [3] (TTA) on the relative performance a model trained with a given sparsity ratio can achieve in a fixed time budget.

We evaluate SDGP on a number of deep neural networks, including a 9-layer ResNet [7] (denoted ResNet-9) on CIFAR-10 [12] and ResNet-18, ResNet-50, and RegNet [26] on ImageNet [4]. Across all networks, we find that SDGP can speed up training by up to 15-25% without any significant loss in accuracy. Additionally, since SDGP only prunes data...
gradients, it can be easily integrated into existing deep learning frameworks without impacting how users construct or train model (similar to mixed-precision training [22]).

The novel contributions of the paper are:

- Structured data gradient pruning (SDGP) which speeds up training without impacting model convergence.
- A detailed evaluation of how different ranking schemes used by SDGP affect the pruning process and the final accuracy of the models.
- Efficient CUDA kernels for implementing SDGP that introduce minimal overhead during backpropagation.

The rest of the paper is organized as follows: Section II provides background on DNN training and discusses related work on structured pruning. Section III presents the SDGP algorithm and outlines how pruning is performed on data gradients, and Section IV compares SDGP to conventional DNN training and analyzes the impact of the sparsity ratio on model performance and runtime. Finally, we discuss future work and summarize the results in Section V.

II. BACKGROUND AND RELATED WORK

In this section, we first review the computation of DNN training in Section II-A. Then, in Section II-B, we review hardware accelerators that support sparse computations. Finally, we discuss related work on structured weight pruning in Section II-C.

A. Computation of DNN Training

Each DNN training iteration consists of two stages: a forward pass and a backward pass. In the forward pass, the input data is propagated forward through all layers of the network, and a loss function is used to compute the error of the network with respect to some target. In backward pass, the computed error is used to update weights by propagating it backwards through the network.

Figure 2 provides a computational view of a single training iteration. On the left, we see a 3-layer DNN, with the forward pass shown using black arrows and the backward pass shown using red arrows. During the forward pass, each fully connected or convolutional layer computes a data (activation) tensor that becomes the input for the next layer. During the backward pass, a weight gradient (dw/W) and data gradient (dL/dO) is computed for each layer. The right side of Figure 2 illustrates the computations of a single layer during these forward and backward passes as matrix multiplications. For simplicity, we assume the convolutional kernel size is 1×1. In general, im2col [1] can be used to map convolution into matrix multiplication. In this work, we propose to prune the output data gradients dL/dO before computing the input data gradients dL/dW using SDGP. This pruning operation is performed on every layer in the network in each training iteration.

B. Sparse Hardware Accelerators

The majority of prior work on reducing DNN training and inference runtime has a focus on leveraging sparsity present in weights and activations during forward propagation [21], [34], [31], [2]. TensorDash [21] automatically skips multiplications with zero values in the activation data, which commonly occurs when using certain activation functions such as ReLU [24]. Eager Pruning [34] and Procrustes [31] co-designing DNN training to fit a given hardware platform. Column combining [14] adds a structured sparsity constraint that allows only n nonzeros per m elements in a DNN weight matrix and designed a corresponding systolic [13] to efficiently support sparse structured matrix multiplication. The Nvidia A100 sparse tensor cores introduce a similar N:M sparsity constraint with a different underlying hardware architecture [25].

C. Structured Pruning Techniques

DNN pruning is an extensively studied technique for reducing DNN model size and runtime [6], [23], [27], [9], [23], [17], [5], [19], [36], [8]. As stated earlier, the majority of pruning techniques are applied to weight tensors over the course of training to make models smaller and more efficient when deployed at runtime for inference.

Multiple pruning functions have been proposed to determine the importance of the DNN weights, including magnitude-based pruning [6] and gradient-based approaches [16], [23], [27]. Much of the initial work on weight pruning did not add any structured constraints, such n:m sparsity, when determining which elements to prune. Generally, such unstructured pruning techniques lead to irregular distributions on nonzero weights, making efficient hardware implementation difficult. By comparison, structured pruning techniques [30], [36], [32], [8], [20] add additional constraints that lead to sparse tensors that can be more efficiently processed. Typically, more rigid sparsity structures, such as pruning entire filters, leads to sub-optimal performance (e.g., classification) compared to unstructured pruning but lead to significantly faster inference. The N:M structured sparsity has achieved significantly less...
attention that these prior methods. Zhou et al. proposed to enforce weight tensors with the N:M sparsity pattern at the start of training [35]. In [10], the authors apply a transposable structured pruning mask to the weights to support efficient computation during both forward and backward passes.

In this work, we forgo adding any sparsity constraint to weight tensors and instead enforce N:M structured sparsity on the gradients. Therefore, this approach could potentially be used with any of the proposed weight pruning approaches described above to achieve further savings.

Ye et al. also explore pruning activation gradients [33] to reduce computation during backpropagation. However, they use an unstructured pruning algorithm making it difficult to realize the theoretical benefits of training during pruning.

III. STRUCTURED DATA GRADIENT PRUNING

In this section, we provide an overview of how structured data gradient pruning (SDGP) is used during DNN training. First, in Section III-A, we show how SDGP is applied to data gradient tensors. Next, Section III-B, gives an overview of the different structured pruning we use in our evaluation. Finally, in Section III-C, we provide intuition for why SDGP can be applied without significantly impacting model performance.

A. SDGP Algorithm

Algorithm 1 shows how structured data gradient pruning is applied to a gradient data tensor \( D \). The \( n \) and \( m \) parameters determine the number of nonzeros per group and group size, respectively. The pruning function \( P \) (discussed next in Section III-B) dictates which \( n \) element across the group \( m \) are kept and prunes the other \( m - n \) elements. The algorithm works by simply iterating across all groups, applying \( P \) to each group, and saving the pruned version of the group at the corresponding location in the output data gradient tensor \( \hat{D} \).

**Algorithm 1** Structured Data Gradient Pruning (SDGP)

**Arguments:** \( \frac{dL}{dD} \) output data gradient tensor, \( n \) nonzeros, \( m \) group size, \( P \) pruning function (described in Section III-B)

1: function SDGP\( (\frac{dL}{dD}, n, m, P) \)
2: \( \frac{dL}{dD} \leftarrow \text{zeros}_{\text{like}}(\frac{dL}{dD}) \) \( \triangleright \) holds pruned \( \frac{dL}{dD} \)
3: \( k \leftarrow ||\frac{dL}{dD}|| \)
4: for \( i \leftarrow 0 \) to \( k \) do
5: \( \frac{dL}{dD}[im : im + m] \leftarrow P(\frac{dL}{dD}[im : im + m], n) \)
6: end for
7: return \( \frac{dL}{dD} \)
8: end function

Note that Algorithm 1 assumes that the pruning dimension is the last dimension in the tensor (meaning that the stride between elements in the group is 1). This simplified the implementation and leads to a more efficient CUDA implementation (see https://github.com/BradMcDanel/sdgp/blob/main/kernels/prune_kernel.cu). For convolutional layers, we use \( N \) as the pruning dimension (as depicted in Figure 2b) which is commonly referred to as the channel dimension. Therefore, in our implementation, we require the channel dimension to be the final dimension. This is generally not an issue as using channels as the final dimension is a commonly used memory layout when using Tensor Cores in Nvidia A100 GPUs [25].

Figure 3 shows how SDGP is applied across this pruning dimension of the output data gradients \( \frac{dL}{dD} \) before computing the input data gradients \( \frac{dL}{dD} \). In our CUDA implementation, each thread is responsible for determine which elements to prune in a given group. The implementation supports a maximum group size of \( m = 32 \). Since \( \frac{dL}{dD} \) is also used to compute the weight gradients \( \frac{dL}{dD} \), SDGP creates a copy of \( \frac{dL}{dD} \) to store the pruned data gradient tensor \( \frac{dL}{dD} \).

B. Structured Pruning Functions

In this section, we present multiple pruning functions used in conjunction with SDGP. Algorithm 1 requires a pruning function \( P \) to perform the pruning for each group. In this work, we evaluate three pruning function, which can be summarized as follows:

- **Random**: Randomly prunes \( m - n \) elements per group.
- **Magnitude**: Sorts the elements in a group by magnitude and prunes the smallest \( m - n \) elements per group.
- **Rescaled Magnitude**: Uses magnitude pruning, then rescales the remaining \( n \) elements such that \( ||D||_1 = ||\hat{D}||_1 \).

Figure 4 shows the result of each pruning function applied to an input dense gradient tensor. For simplicity, only 3 rows, representing batch and spatial dimensions (i.e., width and height for images), and 4 columns, representing channels, are depicted. In practice, one of the these pruning algorithms will be applied to each \( m \) element group across all network layers. As discussed later in Section IV-A, we find that rescaling the remaining nonzeros stabilize training and leads to slightly higher accuracy compared to simple magnitude pruning.

To support magnitude pruning at the group-level, we must first sort all \( m \) elements in the group by their magnitude and then select the top \( n \) to keep. For instance, for the group \([1, 7, -3, 2]\), we first rank the elements by magnitude: \([7, -3, 2, 1]\). Then, we prune the smallest \( m - n \). So, for \( n = 2\),

---

Fig. 3. (left) The computation required to compute the data gradients \( \frac{dL}{dD} \) for a convolutional layer. (right) How SDGP is applied to the output data gradients \( \frac{dL}{dD} \) before performing the data gradient computation. Only a single group at a fixed \( B, W, \) and \( H \) position is shown.

Fig. 4. Shows the result of each pruning function applied to an input dense gradient tensor. For simplicity, only 3 rows, representing batch and spatial dimensions (i.e., width and height for images), and 4 columns, representing channels, are depicted.
the group would become $[7, -3, 0, 0]$. Importantly, we must store the original positions of each element in the group before ranking in order to restore the remaining elements to their original position: $[0, 7, -3, 0]$. All of these steps are performed by a single CUDA thread that is responsible for a given group.

### C. Motivating SDGP via Data Gradient Distribution

A conventional argument from weight pruning methodologies is that smaller weights contribute less to the resulting activation values, making them less useful than larger weights from a computational standpoint [6]. In this section, we similarly argue that extremely small gradients will have minimal impact on the resulting weight gradients and therefore convergence of the model.

Figure 5 (top row) shows four neighboring data gradient channels for the second convolutional layer of ResNet-18 trained on ImageNet after the first epoch of training. The red pixels denote exact zero values that have potential to be skipped under the structured pruning constraint. The reason that almost 25% of values in these data gradients are zero is due to the use of 16-bit floating point representations, which is supported natively by the A100 and roughly halves the training time compared to more conventional 32-bit floating point.

The remaining three rows of Figure 5 show the impact of applying the different pruning algorithms proposed in Section III-B on the top row. For all settings, two nonzeros $n = 2$ are kept across each group of $m = 4$ elements. As the pruning is performed across the channel dimension, only two of the four elements at each spatial position can be nonzero.

Random pruning often prunes high magnitude regions, which lead to lower performance as we discuss in Section IV. Magnitude and rescaled magnitude pruning always prune the same low-magnitude values. However, conventional magnitude pruning biases the distribution towards zero as it only sets values to zero. By comparison, the rescaling operation used by the rescaled magnitude pruning function preserves the $L^1$-norm across batch and spatial dimensions per channel.

### IV. Evaluation

In this section, we evaluate the performance of SDGP on a variety of DNNs and datasets to illustrate its robustness across domains. First, in Section IV-A, we compare the performance of conventional DNN training (using dense activation gradients) to SDGP with the various pruning methods outlined in Section III-B. Then, in Section IV-B, we analyze the impact of the degree of sparsity in SDGP (e.g., $m = 4$ versus $m = 8$ for a fixed $n = 2$) on accuracy. Finally, in Section IV-C, we estimate the reduction in training time for a variety of SDGP sparsity settings.

We evaluate SDGP on CIFAR-10 using a 9-layer version of ResNet [7] and ImageNet [4] using ResNet-18, ResNet-50, and RegNetX-400MF [26]. We use the FFCV [15] dataloader and training regime to train all networks. For CIFAR-10 and ImageNet, all networks are training for 150 and 88 epochs, respectively. Following FFCV, we employ a cyclic learning rate [28], a resolution scaling regime, label smoothing [29], weight decay, and several other regularization techniques. These techniques are applied equally across all SDGP settings to provide a fair comparison. All models were trained using either 1 or 4 Nvidia A100 GPUs. A complete set of training settings are provided in configuration scripts in the code repository. SDGP is applied to all convolution layers in each training iteration.

#### A. Comparison with Conventional DNN Training

In this section, we compare the classification accuracy of conventional DNN training against SDGP using the different pruning techniques outlines in Section III-B. Figure 6 show the average softmax cross entropy loss for the training dataset over
all training epochs for CIFAR-10 and ImageNet. For all SDGP settings, number of nonzeros per group \( n = 2 \) and group size \( m = 4 \). This configuration matches the structured sparsity supported by the sparse tensor cores in the Nvidia A100 [25]. The SDGP (Random) loss curves are generally higher than the dense model. By comparison, SDGP (Magnitude) and SDGP (Res. Magnitude) have similar loss curves to the baseline (Dense) setting.

Figure 7 shows the top-1 validation accuracy for the same models depicted in Figure 6. The SDGP (Random) settings achieve the lowest classification accuracy. This is likely due to the fact that randomly pruning gradients leads to many larger important gradients from being pruned more often, which can be seen in Figure 5 (second row). By comparison, both magnitude-based methods, SDGP (Magnitude) and SDGP (Rescaled Magnitude), maintain a similar classification accuracy as the baseline setting throughout the course of training. Note that the \( n = 2 \) and \( m = 4 \) structure sparsity supported by the A100 is a relatively conservative setting in terms of achieved sparsity (50\%). In Section IV-B, we explore the trade-off between sparsity and classification accuracy.

Table I shows the final top-1 validation accuracy for all evaluated DNNs. Generally, we see that SDGP is able to achieve similar accuracy as the baseline setting across all DNNs. This suggests that users may be able to use SDGP as a drop-in replacement without noticing a significant impact in performance. The benefit of pruning data gradients, as opposed to pruning weights/data, is that the resulting model does not require any additional considerations both during training and when deploying the model for inference. Even though the relative reduction in training time is modest (15-25\%), it makes no modifications to the model and minimal changes to the training framework.

As discussed earlier in Section III-C, rescaling the remaining gradients makes it so that pruning does not pull the average towards zero. We speculate that this may also have some interaction with Batch Normalization [11], as it uses mean and variance statistics across each batch. The rescaling operation ensures that the variance of data gradients is preserved across each sample and feature map.

### B. Impact of Sparsity on Accuracy

While the A100 only supports a \((n = 2\) nonzero, \(m = 4\) group size\) structured sparsity pattern, we are interested in seeing how varying \(n\) and \(m\) impact final model accuracy. Since the 2:4 sparsity setting leads to a 2× speed-up when using the sparse tensor cores compared to using the dense tensor cores, we assume that even sparser settings would lead to additional reductions in computation time.

Figure 8 shows how the classification accuracy of CIFAR-10 using ResNet-9 changes as a function of \(n\) and \(m\). All settings are evaluated using the Rescaled Magnitude pruning function. As the sparsity ratio \((r = \frac{m}{n})\) increases, the classification accuracy decreases as expected. However, there are two interesting insights we can take from this data.

First, for a fixed sparsity ratio (e.g., 2), we see that using a larger group size \(m\) leads to a higher classification accuracy. This is due to the added flexibility in which nonzero elements are selected when using a larger group size. For example, picking the largest \(n = 2\) nonzeros for \(m = 4\) is less flexible...
backward weight gradients). Adding all the times up, we see a convolution layer (forward pass, backward data gradients, to Figure 2). Each grouping shows the total running time for layer (1 in forward pass and 2 in backward pass corresponding to Figure 2). Each grouping shows the total running time for a convolution layer (forward pass, backward data gradients, backward weight gradients). Adding all the times up, we see that computing data gradients takes approximately one-third of the total running time (33.17% in this case for ResNet-18). Thus, applying SDGP with \( n = 2 \) and \( m = 4 \) (sparsity ratio of 2) should lead to a 16.6% reduction in total runtime. Similarly, a sparsity ratio of 4 would lead to a 24.9% runtime reduction.

Figure 9 shows the running time for all convolutions per layer (1 in forward pass and 2 in backward pass corresponding to Figure 2). Each grouping shows the total running time for a convolution layer (forward pass, backward data gradients, backward weight gradients). Adding all the times up, we see that computing data gradients takes approximately one-third of the total running time (33.17% in this case for ResNet-18). Thus, applying SDGP with \( n = 2 \) and \( m = 4 \) (sparsity ratio of 2) should lead to a 16.6% reduction in total runtime. Similarly, a sparsity ratio of 4 would lead to a 24.9% runtime reduction.

Figure 10 shows the estimated reducing in training time to meet a target accuracy 94.5% for different SDGP sparsity settings on CIFAR-10. Since the Nvidia A100 does not support N:M sparsity settings other than \( n = 2 \), \( m = 4 \), we estimated these run times by reducing the fraction of time spend performing the data gradient computations based on the SDGP sparsity ratio. For instance, for SDGP\((n = 4, m = 32)\), which has a sparsity ratio of \(32/4 = 8\), we compute the reduced data gradient time by taking the original percentage of data gradient computation time (33.31%) and dividing it by the sparsity ratio to get the new computation percentage for data gradients \(33.31%/8 = 4.16\%\). We then rescale the original training time recorded per epoch to account for this reduction.

After estimating the runtime for each SDGP setting, we observe that the highest sparsity setting of SDGP\((n = 4, m = 32)\) outperforms the other settings. Specifically, it is able to achieve a relative speedup of 27% compared to the less aggressive sparsity setting of SDGP\((n = 16, m = 32)\) for the same TTA. This suggests that it may be useful to have more fine-grained hardware support for varying levels of structured sparsity (e.g., 1:8, 2:8, 4:8), in order to enable an efficient trade-off between training time and accuracy.

V. FUTURE WORK AND CONCLUSION

In this paper, we proposed a novel structured data gradient pruning (SDGP) technique to speed up DNN training. The structured pruning performed by SDGP is naturally supported by modern hardware accelerators like the Nvidia A100 GPU. SDGP enables a 15-25% reduction in DNN training time without a significant impact to classification performance. We evaluated SDGP on multiple CNNs (ResNet-18, ResNet-50, and RegNetX-400MF) on ImageNet and demonstrate that it can achieve comparable performance to the dense baseline (within 0.3% across all models).

We also investigated the impact of higher structured sparsity ratios on the reduction in training time and impact to performance. For some tasks, we believe that adding support for multiple structured sparsity settings will faster training time while maintaining acceptable accuracy. For instance, training a CIFAR-10 model using SDGP\((n = 4, m = 32)\) leads an additional 27% speed up over SDGP\((n = 2, m = 4)\) while for a 0.4% reduction in accuracy.

Once structured sparsity is better supported by modern deep learning frameworks, we hope that SDGP can be easily to add to existing training pipelines in a similar fashion to FP16 support. As illustrated, in some cases, this would lead to a relatively significant reduction in training time without much effort for the user (no modifications to learning rate, model structure, or other hyperparameters).
