Accelerating Transformer Pre-training with 2:4 Sparsity

Yuezhou Hu[1] Kang Zhao Weiyu Huang[1] Jianfei Chen[1] Jun Zhu[1]

Abstract

Training large transformers is slow, but recent innovations on GPU architecture give us an advantage. NVIDIA Ampere GPUs can execute a fine-grained 2:4 sparse matrix multiplication twice as fast as its dense equivalent. In the light of this property, we comprehensively investigate the feasibility of accelerating feed-forward networks (FFNs) of transformers in pre-training. First, we define a "flip rate" to monitor the stability of a 2:4 training process. Utilizing this metric, we propose three techniques to preserve accuracy: to modify the sparse-refined straight-through estimator by applying the masked decay term on gradients, to determine a feasible decay factor in warm-up stage, and to enhance the model's quality by a dense fine-tuning procedure near the end of pre-training. Besides, we devise two techniques to practically accelerate training: to calculate transposable 2:4 masks by convolution, and to accelerate gated activation functions by reducing GPU L2 cache miss. Experiments show that our 2:4 sparse training algorithm achieves similar convergence to dense training algorithms on several transformer pre-training tasks, while actual acceleration can be observed on different shapes of transformer block apparently. Our toolkit is available at `https://github.com/huyz2023/2by4-pretrain`.

1. Introduction

Pre-training large-scale transformers is hard, for its intensive computation and time-consuming process (Anthony et al., 2020). To accelerate training, sparsity-based methods have recently emerged as a promising solution, and one of the hardware-friendly sparse patterns is 2:4 sparsity. In a 2:4 sparse matrix, every four consecutive elements contain two zeros. Within a tensor core, a 2:4 sparse matrix multiplication (2:4-spMM) could be 2x faster than its dense equivalent on NVIDIA Ampere architecture GPUs.

Some works use 2:4 sparsity for accelerating training (Hubara et al., 2021; Lu et al., 2023; McDanel et al., 2022; Chmiel et al., 2023). However, they mainly target on convolutional neural networks (CNNs) (Hubara et al., 2021; McDanel et al., 2022), whose architecture, optimizer and training procedure are different from transformers. Whether these 2:4 sparse training methods are capable for transformers remains under-explored. In practice, we find two barriers: 1) **Low accuracy.** The hyperparameters in some accuracy preserving techniques for transformers vary significantly from that for CNNs, which is ineffective if transplanted directly. *Remarkably, simply halving the inner dimensionality of a feed-forward network can also reduce the same amount of computational cost, but provides better performance than most of proposed 2:4 sparse training methods.* 2) **Inefficiency.** All previous works on 2:4 training stay on simulation, and do not provide actual acceleration results. Besides, they don't focus on other key operations beyond matrix multiplication that affect the practical time cost, such as overheads of pruning and activation functions. They usually lead to substantial mismatches between simulation and actual acceleration performance.

In this work, we aim to propose an end-to-end acceleration method for pre-training transformers based on 2:4 sparsity. Here are our major contributions:

- We propose three accuracy-preserving techniques (two for masked decay and one for dense fine-tune) for 2:4 training. First, we propose to apply the masked decay on gradients rather than on weight. Second, we show that the feasible masked decay factor on transformers may be very small (100x smaller than it has been reported on CNNs) and devise a method to quickly determine an available decay factor. Besides, our analysis demonstrates that employing a dense fine-tuning stage at the end of pre-training, rather than at the beginning, can enhance the quality of transformers.
- We analyze practical factors affecting the 2:4 training speed of transformers, which is rarely considered by previous works. We identify two speed bottlenecks: pruning overhead and gated activation functions' overhead.

[1] Dept. of Comp. Sci. & Tech., Institute for AI, BNRist Center, Tsinghua-Bosch Joint ML Center, THBI Lab, Tsinghua University. Correspondence to: Jianfei Chen <jianfeic@tsinghua.edu.cn>.

Proceedings of the 41^{st} International Conference on Machine Learning, Vienna, Austria. PMLR 235, 2024. Copyright 2024 by the author(s).

We proposed kernel-level accelerated methods to address each of these bottlenecks.

- To the best of our knowledge, this is the first report on end-to-end acceleration on pre-training transformers (Figure 7, Table 11). Experiments show that transformers pre-trained using our proposed sparse training scheme are comparable or even superior in accuracy to those trained with dense training methods (Table 5, 6).

2. Related Work

Existing sparsity-based methods can be classified into two categories: accelerating inference and accelerating training. For training acceleration, they can be further grouped by whether 2:4 sparsity is involved.

Sparsity for Inference Acceleration Early methods include one-shot pruning (Han et al., 2015; 2016; Lee et al., 2018; Mishra et al., 2021). Later methods (Evci et al., 2021; Zhou et al., 2021; Lasby et al., 2023) suggest using dynamic sparse training (DST). Particularly, Zhou et al. (2021) proposes sparse-refined straight-through estimator (SR-STE) for 2:4 inference. Iterative magnitude-based pruning (IMP) methods (Chen et al., 2020; 2021; You et al., 2022), originated from the winning lottery ticket theory (Frankle & Carbin, 2019; Frankle et al., 2020), can also be viewed as a DST approach. All these methods only speedup the forward pass. They are insufficient to accelerate training.

2:4 Semi-Structured Sparsity for Training Acceleration Accelerating training by 2:4 sparsity is hard, because both the forward and backward passes need to be accelerated. On some GPUs involving sparse tensor cores, 2:4-spMMs perform 2x faster than dense GEMMs (Mishra et al., 2021; BUSATO & POOL). In light of this, (Hubara et al., 2021) firstly proposes a transposable N:M mask to accelerate both output activations and input gradients computation in backward pass. Zhang et al. (2023) improve transposable mask to bi-directional mask (Bi-Mask) to further boost mask diversity. To accelerate calculating weight gradient via 2:4-spMM, an unbiased minimum-variance estimator (MVUE) is introduced (Chmiel et al., 2023). In addition, Xu et al. (2022) also achieve fully sparse training of CNNs using spatial similarity. However, all these works do not report end-to-end training speedups on 2:4 sparse tensor cores, and they are built for CNNs. Practical 2:4 training acceleration on transformers has not been reported so far.

Other Structured Sparsity for Training Acceleration Structured sparsity means channel-wise pruning to dense networks. For instance, training a large model and then compressing it to be thinner or shallower seems effective (Li et al., 2020; Zhou et al., 2020), given a fixed accuracy requirement. However, it's not memory-efficient due to the larger model's redundancy. In addition, low-rank adaption proves to be an effective method to reduce fine-tuning costs (Hu et al., 2023), but it can't accelerate the pre-training.

3. Preliminary

In this section, we first present the mathematical formulations of dense training and fully sparse training. Afterward, we revisit the related methods which are helpful to achieve fully sparse training with 2:4 sparsity, including SR-STE (Zhou et al., 2021), transposable N: M mask (Hubara et al., 2021), and MVUE (Chmiel et al., 2023).

3.1. Dense Training

Problem Formulation Dense training solves an optimization problem $\min_{\mathbf{w}} \mathcal{L}(\mathbf{w})$, where $\mathcal{L}$ is a loss function, $\mathbf{w} \in \mathbb{R}^D$ is the collection of dense weights of all layers, flattened to a vector. The loss is optimized by gradient descent optimization algorithms such as SGD, Adam (Kingma & Ba, 2017) and AdamW (Loshchilov & Hutter, 2019).

GEMMs of a Linear Layer in Dense Training In each training step, a single linear layer performs three general matrix multiplications (GEMMs):

$$\mathbf{Z} = \mathbf{X}\mathbf{W}^\top, \quad \nabla_{\mathbf{X}} = \nabla_{\mathbf{Z}}\mathbf{W}, \quad \nabla_{\mathbf{W}} = \nabla_{\mathbf{Z}}^\top\mathbf{X}, \qquad (1)$$

where $\mathbf{X}$, $\mathbf{W}$ and $\mathbf{Z}$ are input activations, weights, and output activations, with shape $\mathbf{X}, \nabla_{\mathbf{X}} \in \mathbb{R}^{p\times q}$, $\mathbf{W}, \nabla_{\mathbf{W}} \in \mathbb{R}^{r\times q}$, and $\mathbf{Z}, \nabla_{\mathbf{Z}} \in \mathbb{R}^{p\times r}$. Here, the three GEMMs computes output activations, input activation gradients, and weight gradients, respectively. Without loss of generality, we assume the input $\mathbf{X}$ to be a 2D matrix rather than a 3D tensor. In the feed-forward networks of a transformer, this can be done by simply flattening the input tensors' first two axes, *i.e.*, axes of batch size and sequence length.

3.2. Fully Sparse Training with 2:4 Sparsity

GEMMs can be accelerated with structured sparsity. Particularly, 2:4 sparsity (Mishra et al., 2021) is a semi-structured sparsity pattern supported on NVIDIA Ampere architectures. A 2:4 sparse matrix partitions its elements into groups of four numbers, where each group has exactly two zeros. Depending on the direction of partition, there are row-wise 2:4 sparse matrix and column-wise 2:4 sparse matrix; see Appendix A.1. With such sparsity, a GEMM $\mathbf{C} = \mathbf{A}\mathbf{B}$ can be accelerated by 2x with the 2:4-spMM kernel if either $\mathbf{A}$ is row-wise 2:4 sparse, or $\mathbf{B}$ is column-wise 2:4 sparse.

To accelerate training, each GEMM in Equation (1) should have one 2:4 sparse operand. In general, weights and output activation gradients are selected to be pruned due to relatively lower pruning-induced loss (Chmiel et al., 2023).

That is,

$$\mathbf{Z} = \mathbf{X}S_{wt}(\mathbf{W}^\top), \quad (2)$$

$$\nabla_{\mathbf{X}} = \nabla_{\mathbf{Z}} S_w(\mathbf{W}), \quad (3)$$

$$\nabla_{\mathbf{W}} = S_z(\nabla_{\mathbf{Z}}^\top)\mathbf{X}. \quad (4)$$

In Equations (2) to (4), S_{wt}, S_w, and S_z represent the pruning functions of $\mathbf{W}^\top$, $\mathbf{W}$, and $\nabla_{\mathbf{Z}}^\top$. They take dense matrices as input, and outputs 2:4 sparse matrices. By intuition, a pruning function picks out the 2 elements with the max magnitudes in the adjoining 4 elements and zero out the rest. With hardware support, computing Equations (2) to (4) can be theoretically 2x faster than Equation (1). This method use 2:4-spMMs for all matrix multiplications in forward and backward propagation, so we call it *fully sparse training* (FST). Note that Equation (4) contains a straight-through estimator (STE), which we will explain later.

Transposable Masks Hubara et al. (2021) suggest that a weight matrix and its transpose can be simply pruned by multiplying binary masks, *i.e.*,

$$S_{wt}(\mathbf{W}^\top) = \mathbf{W}^\top \odot \mathbf{M}_{wt}, \quad S_w(\mathbf{W}) = \mathbf{W} \odot \mathbf{M}_w,$$

where $\mathbf{M}_{wt}, \mathbf{M}_w \in \{0,1\}^{p\times q}$ are 2:4 sparse, and $\odot$ is element-wise product. To utilize 2:4-spMM, the two binary masks should be mutually transposable:

$$\mathbf{M}_{wt} = \mathbf{M}_w^\top, \quad (5)$$

which they call as transposable masks (same as our defination in Section 5.1). In this manner, the backward pass share the same sparse weight matrix with the forward pass. The authors also propose a 2-approximation method for generating such masks with claimed low computational complexity.

Minimum-Variance Unbiased Estimator Chmiel et al. (2023) propose to calculate the 2:4 sparse masks of neural gradients by MVUE, *i.e.*,

$$S_z(\nabla_{\mathbf{Z}}^\top) = \mathrm{MVUE}(\nabla_{\mathbf{Z}}^\top). \quad (6)$$

Compared to the commonly used minimum square error estimation, MVUE guarantees unbiasedness and minimizes the variance of the sparsified gradients, which is more favorable for promoting the convergence of training.

3.3. Optimization Strategies for Sparse Training

The optimization of a sparse network is difficult as it has non-differentiable pruning functions. The optimization objective can be formulated as $\min_{\mathbf{w}} \mathcal{L}(\tilde{\mathbf{w}})$. The network makes prediction with a sparse weight vector $\tilde{\mathbf{w}} = \mathbf{m}(\mathbf{w}) \odot \mathbf{w}$, where the mask $\mathbf{m}(\mathbf{w}) \in \{0,1\}^D$ is the concatenation of masks for each layer. If a layer is not sparsified, then the corresponding mask is an all-one matrix. Computing the gradient is tricky since the mask $\mathbf{m}$ is dynamically computed based on the dense weight $\mathbf{w}$: by chain rule we have $\nabla_{\mathbf{w}}\mathcal{L}(\tilde{\mathbf{w}}) = \frac{\partial \tilde{\mathbf{w}}}{\partial \mathbf{w}} \nabla_{\tilde{\mathbf{w}}}\mathcal{L}(\tilde{\mathbf{w}})$, where $\frac{\partial \tilde{\mathbf{w}}}{\partial \mathbf{w}}$ is a Jacobian matrix. However, $\tilde{\mathbf{w}}$ is not differentiable with $\mathbf{w}$ since it includes a non-differentiable mask-computing-function $\mathbf{m}(\cdot)$ in it. Thus, it takes some skills to estimate the gradients and update the parameters.

STE As $\tilde{\mathbf{w}}$ is an approximation of $\mathbf{w}$, a straight-through estimator (STE, Bengio et al. (2013)) directly passes the gradient of $\tilde{\mathbf{w}}$ to $\mathbf{w}$:

$$\nabla_{\mathbf{w}}\mathcal{L}(\tilde{\mathbf{w}}) \leftarrow \nabla_{\tilde{\mathbf{w}}}\mathcal{L}(\tilde{\mathbf{w}}). \quad (7)$$

SR-STE There is a problem with STE: only a portion of the weights in a layer participate in the forward calculation, but all the weights receive gradients. This indicates that the gradients associated with masked weights[1] might be inaccurate. To suppress those inaccurate gradients, Zhou et al. (2021) proposes sparse-refined straight-through estimator (SR-STE) which adds a decay term when updating:

$$\mathbf{w}_t \leftarrow \mathbf{w}_{t-1} - \gamma(\nabla_{\mathbf{w}}\mathcal{L}_t(\tilde{\mathbf{w}}_{t-1}) + \lambda_W(\overline{\mathbf{m}(\mathbf{w}_{t-1})}) \odot \mathbf{w}_{t-1}), \quad (8)$$

where γ stands for the learning rate, λ_W is the decay factor, and $\overline{\mathbf{m}(\mathbf{w}_{t-1})}$ denotes the logical not operation of $\mathbf{m}(\mathbf{w}_{t-1})$. This decay term alleviates the change of weight mask. With SR-STE, the optimization target becomes

$$\min_{\mathbf{w}} \mathcal{L}(\tilde{\mathbf{w}}) + \tfrac{\lambda_W}{2}\|\mathbf{w} \odot \overline{\mathbf{m}(\mathbf{w})}\|_2^2. \quad (9)$$

4. Accuracy Preserving Techniques

While the methods reviewed in Section 3 can successfully perform FST on small-scale models such as ResNet and DenseNet, it is not clear whether they can be directly applied to pre-train large transformers. It is challenging for FST to preserve the accuracy of dense training, since the weights and masks need to be learned jointly, which is a non-differentiable, combinatorial optimization problem. Moreover, unlike inference acceleration methods, FST has no pre-trained dense model to start with. In this section, we propose three practical techniques to improve the convergence of FST for transformers: transformer-specific masked decay, Fast decay factor determination and dense fine-tuning.

4.1. Flip Rate: Stability of Training

Inspired by previous work (Zhou et al., 2021; You et al., 2022), we define a "flip rate" to measure how frequently the mask vector changes after one optimizer step. This metric could be used to monitor whether the network connection is stable during training.

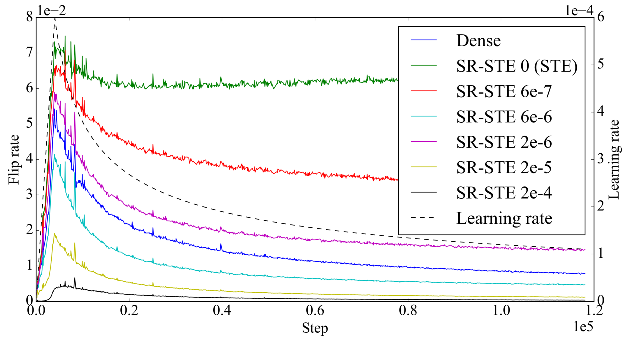

Figure 1. Flip rates change throughout the training of different λ_W on Transformer-base. Note that these models utilize an identical learning rate schedule.

Table 1. Training results of different λ_W on Transformer-base. As λ_W increases from 0 to 2e-4, accuracy first rises and then drops, which means that λ_W should be neither too big nor too small to reach the optimal results.

λ_W	AVG EPOCH LOSS	VAL LOSS	TEST BLEU
DENSE	4.558	3.978	26.15
0 (STE)	4.76	4.164	24.98
6E-7	4.684	4.079	25.68
6E-6	4.626	4.033	25.81
2E-6	4.64	4.041	25.94
2E-5	4.642	4.049	25.74
2E-4	4.662	4.06	25.62

Definition 4.1. Suppose $\mathbf{w}_t$ is a D-dimensional weight vector at time t, and the flip rate r_t is defined as the change in proportion of the mask vector after an optimizer step: $r_t = \|\mathbf{m}(\mathbf{w}_t) - \mathbf{m}(\mathbf{w}_{t-1})\|_1 / D \in [0, 1]$. The larger r_t is, the more unstable the network connections become.

You et al. (2022) suggest that a sparse neural network acts differently in different training phases. In the early phase of training, it eagerly explores different connection modes, which means the masks vector change rapidly over time. Later, the masks gradually become stable, and the network turns itself to fine-tune weight values. In terms of flip rate, we hypothesize that

A healthy training process comes with the flip rate r_t rising at the beginning of training and then gradually fading to 0.

We measure flip rate change for dense training, STE and SR-STE with different λ_W in Figure 1. For dense training, we compute the flip rate by pruning the dense weight in each iteration, despite the pruned weight is never used for training. In terms of flip rate, dense training is healthy: its r_t exactly increases first before declines. If a training process consistently has higher flip rate than dense training, which we call as "flip rate explosion", it may suffer from a loss in final accuracy due to unstable training; see Table 1. In practice, STE suffers from a flip rate explosion, while SR-STE takes effect by "freezing" masks of weights: by adding a decay term, it decrease the number of flips. This inhibition effect is related to the decay factor of SR-STE: the larger λ_W is, the stronger the inhibition of flips is, and the smaller flip rate goes.

In this section, all methods we propose involve our ultimate principle: *the peak of the curve should be sufficiently high to fully explore different connection modes, and the tail should be sufficiently low for the optimization process to converge.*

4.2. Transformer-Specific Masked Decay

Based on our insights on flip rate, we propose a method to suppress the frequent change of masks during FST for transformers, which we call *masked decay*.

Unlike Equation (8) which imposes regularization directly on weights, we propose to add masked decay on gradients, *i.e.*,

$$\mathbf{g}_t \leftarrow \nabla_{\mathbf{w}} \mathcal{L}_t(\tilde{\mathbf{w}}_{t-1}) + \lambda_W (\overline{\mathbf{m}(\mathbf{w}_{t-1})} \odot \mathbf{w}_{t-1}). \quad (10)$$

On SGD, applying decay on weights and on gradients are equivalent, but on popular optimizers like Adam and AdamW they aren't. Specifically, Adam updates weights by

$$\mathbf{w}_t \leftarrow \mathbf{w}_{t-1} - \frac{\gamma(\beta_1 \mathbf{u}_{t-1} + (1-\beta_1)\mathbf{g}_t)}{(1-\beta_1^t)(\sqrt{\hat{\mathbf{v}}_t} + \epsilon)} \quad (11)$$

where $\mathbf{u}$ and $\mathbf{v}$ are the first and second order momentum of $\mathbf{w}$. Compared to Equation (8), the masked decay regularization term in Equation (10) would be later normalized by $\sqrt{\hat{\mathbf{v}}_t} + \epsilon$ in Equation (11), before it is subtracted from weights. In this way, each dimension receives a different intensity of decay ("masked decay"). More specifically, weights with larger gradients get smaller decay intensity, and vice versa.

In FST, we periodically prune weights by their magnitudes. STE may cause the network to fall into such "dilemma points", where a portion of pruned weights and unpruned weights have nearly the same L1 norm. Thus, the network consistently oscillate between two possible masks $\mathbf{m}_1$ and $\mathbf{m}_2$, and is unlikely to jump out the dilemma itself. To illustrate this, we split each weight matrix by small 4×4 blocks. We count each block's cumulative flip number and measure the "L1 norm gap" by $g_i = \|\mathbf{m}_1 \odot \mathbf{w}_i\|_1 - \|\mathbf{m}_2 \odot \mathbf{w}_i\|_1$, where $\mathbf{w}_i$ is the i-th 4×4 weights, $\mathbf{m}_1 \odot \mathbf{w}_i$ and $\mathbf{m}_2 \odot \mathbf{w}_i$ have the first and second largest L1-norm among different pruning binary masks. The selected mask is most likely to oscillate between $\mathbf{m}_1$ and $\mathbf{m}_2$, especially when g_i is small. In STE, there exists more 4×4 blocks

[1] Unlike some relevant literature, we use "masked weights" and "pruned weights" to denote the weights that are set to 0.

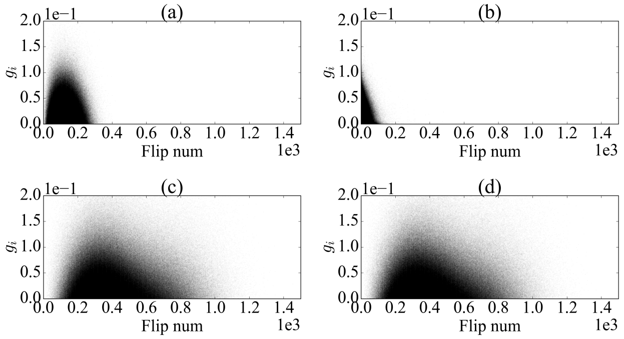

Figure 2. Scatter plots of cumulative flip number and L1 norm gap g_i on every 4×4 block. All results are selected on Transformer-base, with epoch=20. (a) shows the result of dense model. (b)-(d) shows that of masked decaying on gradients, no decaying, and masked decaying on weights. Also, we do it on purpose to choose an extremely large λ_W for SR-STE.

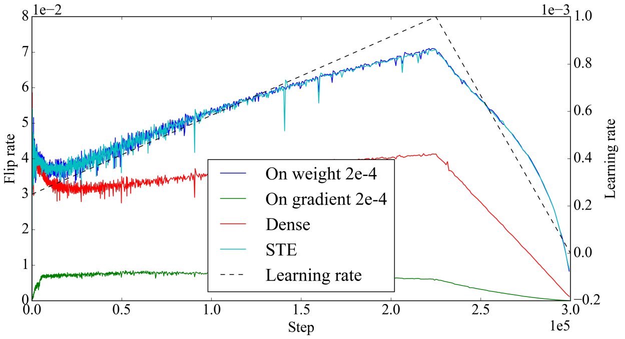

Figure 3. Applying masked decay on weights takes no effect to inhibit flip rate on BERT-base (compared to applying directly on gradient).

Table 2. Optimal λ_W for multiple models.

Model		Optimal λ_W
ResNet18 (Zhou et al., 2021)		2e-4
BERT-base		6e-6
Transformer-base		1e-6
DeiT-tiny		2e-3
GPT-2	124M	6e-5
	350M	2e-4
	774M	2e-4
	1558M	6e-5

with high flip num and low "L1 norm gap"; see Figure 2. This results in overall flip rate explosion of STE.

On these occasions, we argue that an evenly masked decay applied on weights is insufficient to save the training from such "traps". The weights don't differentiate themselves after an update, so masks may oscillate back. By normalizing the weight gradients with $\sqrt{\hat{\mathbf{v}}_t} + \epsilon$, our masked decay amplifies the regularization strength for the dimension with smaller gradient, pushing it towards zero. Then, the regularized dimension can no longer compete with other dimensions. So we effectively break the tie and push the training process out of the trap, towards a "healthier" state.

The comparison results between our masked decay defined in Equation (10) and the conventional counterpart in Equation (8) are shown in Figure 3. Results show that applying masked decay on weights takes no effect to inhibit flip rate explosion of STE, while applying on gradients works fine.

4.3. Fast Decay Factor Determination

The determination of the decay factor λ_W in Equation (10) is non-trivial: if λ_W is excessively large, then the "peak" of the flip rate curve is not high enough; if λ_W is too small, the "tail" of the curve is not low enough. Both do not provide a healthy training process. Besides, we find that λ_W values for CNNs and other small-scale networks differ significantly from those for transformers, while on transformers, optimal λ_W can span up to three orders of magnitude (Table 2).

As pre-training large transformers is costly, grid searching for λ_W with the final accuracy is impractical, so it is vital to determine a feasible λ_W as quickly as possible. To quickly determine λ_W, here we propose a test-based method:

1) **Grid search on the warm-up stage of training.** For each λ_W value in a candidate set, sample a corresponding flip rate of the sparse network from a small number of training steps. Note that sampling in early training stage is enough to obtain a representative flip rate specific to a sparse network.
2) **Comparison with the dense counterparts.** Suppose r_{t_0} to be the standard flip rate on the dense network at time t_0 and r'_{t_0} to be the sparse network's flip rate. Their ratio is $\mu = {r'_{t_0}}/{r_{t_0}}$. We suggest that a feasible λ_W should have $\mu \in [0.60, 0.95]$ and the sparse network may suffer from an accuracy drop if $\mu \geq 1$.

4.4. Dense Fine-Tuning

To better improve accuracy, we suggest using a "dense fine-tuning" procedure at the end of training. Formally, we select a switch point t_s. FST is performed while $t \leq t_s$, and dense training is switched to if $t > t_s$.

Why Choose Dense Fine-Tuning Instead of Dense Pre-training? While previous work (Han et al., 2017) suggest to switch between sparse and dense training stages, some recent works like STEP (Lu et al., 2023) utilize dense pre-training rather than dense fine-tuning, which means a dense network is initially trained for a period of time before being switched to a sparse one. However, we argue that dense pre-training is meaningless in our FST process. As described in

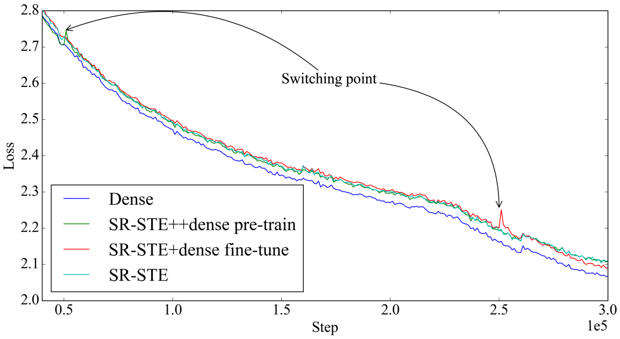

Figure 4. Dense fine-tuning versus dense pre-training on BERT-base

Section 4.1, the peak of the flip rate curve should be sufficiently high to explore connection modes, so what matters most to the flip rate is the magnitudes of weights, which are the key to determine if connections are built or demolished. In this regard, both FST and dense pre-training are capable of delivering proper gradient magnitudes, so dense pre-training is a waste. The precise gradients are generally more necessary in the later stages of training, where the flip rate of the dense network comes to its tail. Figure 4 visualizes the loss curve of pre-training BERT-base, where dense pre-train obtains nearly the same result as the naive SR-STE method. From this, we propose the following insight:

If dense pre-training of t_α steps provides slight improvement of accuracy, then moving the t_α dense steps to the end gives far more improvement than dense pre-training.

As for the specific position of the switch point in training, STEP (Lu et al., 2023) suggests that the dense pre-training occupy 10% to 50% of the total steps. Likewise, we determine that our dense fine-tuning takes up the last $1/6$ of total steps for balance training efficiency and accuracy.

5. Training Acceleration Techniques

For transformers, the forward pass of FST involves pruning weights in FFNs with transposable 2:4 masks and then performing normal forward propagation. During backward propagation in FST, the gradients of input activations and weight gradients in FFNs are derived by Equation (3) and (4), respectively. Note that we also utilize MVUE to prune gradients of output activations, *i.e.*, Equation (6). Compared to dense training, our FST replaces all the GEMMs in FFNs with 2:4-spMMs that theoretically perform 2x faster than their dense counterparts on GPUs within sparse tensor cores.

In addition to speeding up the most time-consuming GEMMs in FFNs, there are three major operations that also have non-negligible impacts on training speed:

1) **Pruning.** In FST, pruning includes two steps: finding a mask that satisfies the 2:4 sparse patterns and then enforcing the mask to the corresponding dense matrices. In our case, we find that the time cost of finding transposable masks is time-consuming.
2) **Activation functions.** In transformers, SwiGLU and GEGLU (Shazeer, 2020) are popular. These two activation functions involve a gate mechanism to regulate activations. This mechanism easily induces the GPU L2 cache misses, thus decreasing the computing speed.
3) **Updating optimizer states.** The excessive update frequency can introduce additional time overheads.

Below, we show our methods to accelerate these operations, the main workflow of which is shown in Appendix B.

5.1. Fast Computation of Transposable Masks

Problem Formulation We aim to find such a mask matrix $\mathbf{M} \in \{0,1\}^{r\times q}$ for every $\mathbf{W} \in \mathbb{R}^{r\times q}$ in the FFN layer that 1) each adjoining 4×4 block contains 8 non-zero positions; each row and column in the block occupies 2 non-zero elements exactly; 2) $\max_{\mathbf{M}} \|\mathbf{M}\odot\mathbf{W}\|_1$. Then $\mathbf{M}$ would be our targeting *transposable mask*.

As described in Equation (5), both a transposable mask itself and its transposition conform to the format of 2:4 sparsity. Previous 2-approximation algorithm (Hubara et al., 2021) consists of two steps: sort elements, and pick elements out of the array. They claim that the procedure has less computational complexity. However, in practice, the sorting and picking process contains too many jumps in its control flow, and may be fatal to modern GPU architecture. To make full use of the GPUs' parallel computation capability (SIMD and SIMT), we convert the transposable mask-search process into a convolution operation which traverse all the masks to obtain the optimal one in three steps:

1) Create a convolutional kernel in the shape of $4\times 4\times n_t$, where n_t denotes the number of transposable masks. In the case of 2:4 sparsity, mask diversity $n_t = 90$. These mask blocks for 2:4 sparsity can be selected by exhaustively inspecting all potential masks offline.
2) Calculate the index matrix via Algorithm 1. The index matrix denotes which 4×4 mask in the convolutional kernel is the optimal mask that retains most of the weight norms after being applied to weights.

Algorithm 1 transposable mask search

```
**Input:** mask pattern $\mathbf{m}'$, weight matrix $\mathbf{W}$
1. $\mathbf{W} = \mathrm{abs}(\mathbf{W})$
2. $out = \mathrm{conv2d}(\mathbf{W}, \mathbf{m}', stride = 4, padding = 0)$
3. $index = \mathrm{argmax}(out, dim = 2)$
**return** $index$
```

3) Replace all the elements in the index matrix by the corresponding 4×4 block, which is the desired mask.

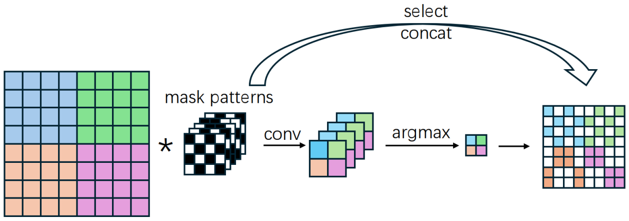

Figure 5. Transposable mask search

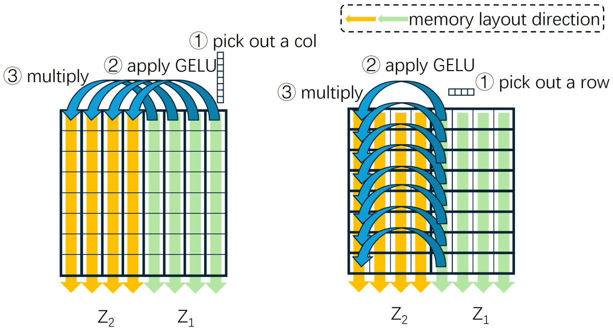

Figure 6. left: adapted method; right: intuitive method

Table 3. Throughput of two transposable search kernels on RTX3090 (TB/s).

Method / Input	2-approx		Ours	
	fp16	fp32	fp16	fp32
3072×768	18.5	36.4	69.2	104.7
4096×1024	22.5	38.4	91.9	131.5
5120×1280	22.6	44.4	91	128.2
1024×1600	22.8	44.8	95	134.5
8192×2048	23	45.1	99.4	142.9
16384×4096	23.2	45.4	100.1	144.8
30768×8192	23.2	45.5	100.9	145.1

Table 4. Throughput of two GEGLU implementations on RTX3090 with fp16 column-major input tensors (TB/s).

Method / Input	Intuitive	Ours
$32 \times 512 \times 768$	18.4	55.5
$32 \times 512 \times 1024$	19.9	55.7
$32 \times 512 \times 1280$	18.2	55.9
$32 \times 512 \times 1600$	18.4	55.9
$32 \times 512 \times 2048$	19.5	56
$32 \times 512 \times 4096$	11.8	56.1
$32 \times 512 \times 8192$	12.1	56.2

Notably, step (1) is executed offline. Step (2) and (3) are frequently performed during FST. The workflow of our method is shown in Figure 5. Compared to the 2-approximation algorithm, our method is up to about 5 times faster (Table 3).

5.2. Acceleration of Gated Activation Functions

Activation functions with gated mechanisms are widely used in transformers such as GLM (Du et al., 2022) and LLaMA (Touvron et al., 2023). Typical gated activation functions involve SwiGLU and GEGLU. The bottleneck of such activation functions is that the gate operations easily incur GPU L2 cache miss. Take GEGLU as an example: $\text{GEGLU}(\mathbf{X}, \mathbf{U}, \mathbf{V}, \mathbf{b}, \mathbf{c}) = \text{GELU}(\mathbf{X}\mathbf{U}^\top + \mathbf{b}) \odot (\mathbf{X}\mathbf{V}^\top + \mathbf{c})$, where $\mathbf{X} \in \mathbb{R}^{p \times q}, \mathbf{U}, \mathbf{V} \in \mathbb{R}^{r \times q}, \mathbf{b}, \mathbf{c} \in \mathbb{R}^r$. In practice, this function is composed of three steps:

1) Concatenate $\mathbf{U}$ and $\mathbf{V}$ into a new weight matrix $\mathbf{W} \in \mathbb{R}^{2r \times q}$, and $\mathbf{b}, \mathbf{c}$ into a new bias vector $\mathbf{d} \in \mathbb{R}^{2r}$.
2) Directly calculate $\mathbf{Z} = \mathbf{X}\mathbf{W}^\top + \mathbf{d} \in \mathbb{R}^{p \times 2r}$ as a compressed matrix.
3) Split the $\mathbf{Z}$ in the second dimension into $\mathbf{Z_1}, \mathbf{Z_2} \in \mathbb{R}^{p \times r}$. Calculate $\text{GELU}(\mathbf{Z_1}) \odot \mathbf{Z_2}$.

Different from dense model, where output activations are row-major matrices, in FST, the output activations are column-major; see Appendix A.2. This property results in the third step being extremely time-consuming if conventionally $\mathbf{Z}$ is accessed along the row dimension. To illustrate, Figure 6 shows that in a column-major matrix $\mathbf{Z}$, accessing along the column accords with array layout. Thus, adjacent elements loaded into the GPU cache can be probably hit. By contrast, accessing along the row does not fully utilize the efficiency of GPU cache. In light of this, we carefully implement a GEGLU kernel where elements are accessed along the column dimension. In this way, GEGLU is performed 5 times faster than the naive counterpart; see Table 4.

5.3. Other Implementation Details

Reducing Updating Frequency We find that a 2:4 mask doesn't change a lot after one optimization step, and it is not necessary to update a mask frequently. For the sake of efficiency, we update the transposable masks of weights every l optimizer steps. We usually take $l = 40$ in practice.

Utilities For 2:4-spMMs, we use CUTLASS (Thakkar et al., 2023). Other GPU kernels are implemented in Triton, including transposable mask search kernel, pruning kernel, MVUE kernel, GEGLU kernel, and masked decay kernel.

6. Experiments

In this section, we validate the proposed training speedup methods on several transformers, including BERT (Devlin et al., 2019), GPT-2 (Radford et al., 2019), Transformer-

[2] Results reported in the original paper; see `https://github.com/facebookresearch/deit/blob/main/README_deit.md`.

[3] DeiT-base dense model using the original recipe.

Table 5. GLUE scores of different 2:4 training methods with BERT.

Method	Loss	Avg score	CoLA	MNLI	mnliextra	MRPC	QNLI	QQP	RTE	SST-2	STS-B
Dense	2.0669	79.8 ± 0.4	45.3 ± 1.1	82.6 ± 0.2	83.4 ± 0.1	78.8 ± 1.7/86.1 ± 1	89.3 ± 0.2	90.3 ± 0.1/87.1 ± 0	55.8 ± 0.9	91 ± 0.5	83.7 ± 1/83.7 ± 1
Half	2.1280	77.9 ± 0.4	37.2 ± 1.3	82.4 ± 0.1	83 ± 0.3	75.1 ± 1.4/84.2 ± 0.7	88.8 ± 0.3	89.9 ± 0.1/86.6 ± 0.1	51.2 ± 2.4	92.1 ± 0.5	82.1 ± 0.5/82.3 ± 0.4
STEP	2.1179	77.7 ± 0.1	40.4 ± 1.4	82.2 ± 0.1	82.8 ± 0.1	74.5 ± 0.7/83.5 ± 0.4	88.3 ± 0.4	90.2 ± 0.1/87 ± 0.1	50.8 ± 2.1	92.3 ± 0.3	79.7 ± 1.2/80.7 ± 0.6
Bi-Mask	2.1176	77.7 ± 0.3	38.3 ± 0.7	82.3 ± 0.1	83 ± 0.1	74.3 ± 0.7/83 ± 0.6	88.3 ± 0.3	90.2 ± 0.1/86.9 ± 0.1	53.1 ± 1.4	90.9 ± 0.3	80.9 ± 0.7/81.7 ± 0.4
Ours	**2.0968**	**79.6 ± 0.6**	**44.4 ± 1.9**	**82.6 ± 0.2**	**83 ± 0.1**	**80.9 ± 0.7/87.4 ± 0.4**	**88.4 ± 0.3**	**90.3 ± 0.1/87 ± 0.1**	**54.3 ± 1**	**91.2 ± 0.4**	**82.9 ± 2.1/83 ± 1.7**

Table 6. GLUE scores with different model sizes on GPT-2 models.

Params	Method	Val loss	Avg Score	CoLA	MNLI	MRPC	QNLI	QQP	RTE	SST-2	STS-B	WNLI
124M	Dense	2.907	73.9 ± 1.1	44.6 ± 0.9	82 ± 0.1	78.3 ± 1.3/84.8 ± 1	88.4 ± 0.2	90 ± 0	86.5 ± 0/61.3 ± 1.5	91.9 ± 0.2	77.3 ± 3.2/77.9 ± 2.9	24.3 ± 7.1
	Ours	**2.952**	**74.3 ± 0.5**	**44.8 ± 1.3**	**81.5 ± 0.2**	**77.5 ± 1.8/84.2 ± 1.3**	**87.8 ± 0.1**	**89.5 ± 0.1**	**85.9 ± 0.1/66 ± 1**	**90.6 ± 0.4**	**80 ± 0.8/80.3 ± 0.5**	**23.9 ± 6.4**
350M	Dense	2.618	76.3 ± 0.1	54.3 ± 0.4	85.1 ± 0.1	80.7 ± 1/86.6 ± 0.7	90.7 ± 0.1	91 ± 0.1	87.8 ± 0.1/64.9 ± 1.7	93.5 ± 0.4	81.7 ± 1.2/82.2 ± 0.8	17.6 ± 3.2
	Ours	**2.688**	**77.1 ± 0.2**	**51.8 ± 1.8**	**84.3 ± 0.1**	**80.6 ± 1.3/86.5 ± 0.8**	**90.4 ± 0.2**	**90.7 ± 0.1**	**87.5 ± 0.1/66.7 ± 1.3**	**93.3 ± 0.4**	**83.4 ± 1.1/83.5 ± 1.1**	**26.4 ± 4**
774M	Dense	2.493	76.2 ± 0.4	57.5 ± 2	86.1 ± 0.1	80.3 ± 1.3/86.4 ± 0.9	91.4 ± 0.2	91.1 ± 0.1	88 ± 0.1/67.7 ± 2.6	94.6 ± 0.4	77.3 ± 3.3/78.4 ± 2.9	15.1 ± 2.3
	Ours	**2.564**	**77.1 ± 0.4**	**55.9 ± 0.9**	**85.6 ± 0.2**	**81.2 ± 0.6/87 ± 0.4**	**91.4 ± 0.1**	**91 ± 0.1**	**87.8 ± 0.1/71.5 ± 0.7**	**94.2 ± 0.4**	**81.8 ± 1.3/82.3 ± 1.2**	**15.8 ± 1.2**
1558M	Dense	2.399	76.5 ± 0.5	55.3 ± 2	87 ± 0.1	79 ± 1/85.3 ± 0.8	91.8 ± 0.3	91.3 ± 0.1	88.3 ± 0.1/73.3 ± 2	95.9 ± 0.3	78.5 ± 2.4/79.2 ± 2.5	13 ± 1.3
	Ours	**2.489**	**77.1 ± 0.5**	**56.4 ± 3**	**86.6 ± 0.1**	**80 ± 0.4/86.1 ± 0.3**	**91.9 ± 0.1**	**91.4 ± 0.1**	**88.4 ± 0.1/75 ± 1.8**	**95.2 ± 0.4**	**80.6 ± 1.1/81.1 ± 1.3**	**12.7 ± 1.1**

Table 7. SQuAD scores on GPT-2 models.

Params	Method	EM	F1
124M	Dense	67.6	78.8
	Ours	**67.5**	**78.5**
350M	Dense	73.2	83.6
	Ours	**71.9**	**82.4**
774M	Dense	74.3	84.9
	Ours	**74.3**	**84.6**

Table 8. Experimental results for DeiT.

Size	Method	Acc@1	Acc@5
DeiT-tiny	Original[2]	72.2	91.1
	Dense[3]	72.9	91.6
	Ours	**70.4**	**90.1**
DeiT-small	Original	79.9	90.5
	Dense	79.9	94.5
	Bi-Mask	77.6	-
	Ours	79.2	94.8
DeiT-base	Original	81.8	95.6
	Dense	81.0	95.0
	Ours	**81.3**	**95.4**

Table 9. Experimental results for Transformer-base.

Method	Avg epoch loss	Test BLEU	Val BLEU	Val loss
Dense	4.558	26.15	26.56	3.982
Half	4.659	26.12	26.36	4.041
STEP	4.692	25.27	25.85	4.082
Ours	**4.649**	**26.48**	**26.78**	**3.977**

base for machine translation (Vaswani et al., 2023), and DeiT (Touvron et al., 2021b). For BERT, we use Cramming (Geiping & Goldstein, 2022) to pre-train a 16-layer BERT model with the sequence length of 512 on the C4 dataset (Raffel et al., 2019). For GPT-2, we use nanoGPT (Karpathy, 2023) to pre-train GPT-2 124M, 355M, 774M, and 1.5B on OpenWebText (Gokaslan & Cohen, 2019). Both BERT and GPT-2 models are estimated on GLUE (Wang et al., 2018). For DeiT (Touvron et al., 2021a), we pre-train DeiT-tiny on ImageNet-1K dataset (Deng et al., 2009). Besides, we use fairseq (Ott et al., 2019) to train Transformer-base on the WMT 14 En-De dataset (Bojar et al., 2014) and measure the BLEU (Papineni et al., 2002) score of the trained model.

Of note, we use n to denote the length of sequences, d to denote the input and output dimensions of each transformer block, d_{ff} to denote the inner dimensions of the FFNs in each transformer block, h to denote the number of heads, and N to denote the micro-batch size on each device. The pre-training and evaluation scripts are publicly available at `https://github.com/thu-ml/2by4-pretrain-acc-examples`.

6.1. Accuracy Results

To investigate the effect of different 2:4 sparse training methods, we pre-train a sparse BERT-base model on the C4 dataset using two sparse training methods: STEP (Lu et al., 2023) and Bi-Mask (Zhang et al., 2023). Besides, we also pre-train a dense BERT-base and a 'Half' BERT-base for comparison. Of note, 'Half' denotes a smaller yet still dense BERT-base model. To create Half model, we simply reduce the d_{ff} of each FFN layer in the original BERT-base by half while maintaining the original value of d. Theoretically, this adjustment halves the floating operations (FLOPs) of the original FFN layer as well. Except for the FFN layers, the shapes of the rest layers remain unaltered.

All the pre-trained models are measured on GLUE benchmark (WNLI excluded). Surprisingly, Table 5 shows that despite having identical FLOPs, the 2:4-sparse BERT-base trained with STEP and Bi-Mask shows inferior average scores compared to the Half model. The Half model attains

Table 10. Experimental results of masked decay, MVUE, and dense fine-tuning (FT) with BERT-Base. For decay term, we use both techniques in Sections 4.2 and 4.3.

MASKED DECAY	MVUE	DENSE FT	LOSS	AVG SCORE
✗	✗	✗	2.1553	77.6 ± 0.2
✓	✗	✗	2.1096	79.2 ± 0.2
✓	✓	✗	2.1172	78.4 ± 0.3
✓	✗	✓	2.0896	79.4 ± 0.2
✓	✓	✓	**2.0968**	$\mathbf{79.6 \pm 0.6}$

Table 11. Actual pre-train speed up on the whole network.

PARAMETERS	BATCH SIZE	SPEEDUP
124M	16	1.18
350M	8	1.2
774M	4	1.21

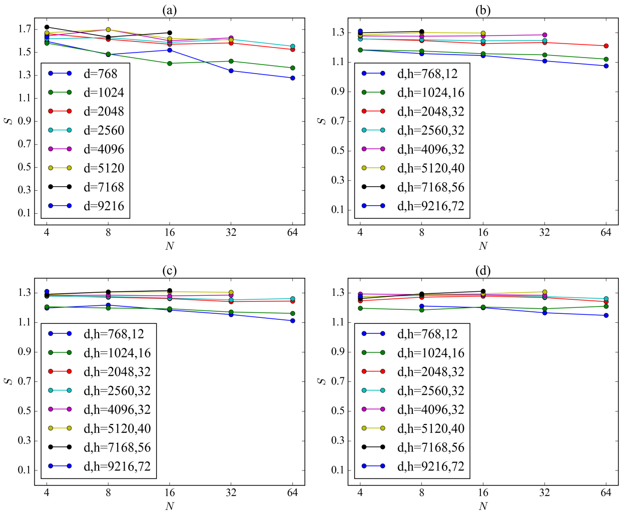

Figure 7. Result of acceleration ratio S of different batch sizes and embedding Sizes. (a) shows the acceleration of a FFN layer. (b)-(d) shows the acceleration of a transformer block when $n = 2048, 1024, 512$.

an average score of 77.9 on GLUE tests, while STEP and Bi-Mask only reach 77.7 due to the weaknesses in MRPC, QNLI, and STSB. By comparison, BERT-base trained in our proposed training method achieves 79.6 on GLUE, which significantly outperforms other sparse training methods and is comparable with the dense baseline, *i.e.*, 79.8.

Besides, we pre-train GPT-2 models with proposed methods. Table 6 and 7 shows that our method for model sizes of 124M, 350M, 775M and 1558M achieves lossless scores compared with dense baselines. Similarly, DeiT and Transformer-base trained with our method also reach comparable results to dense training; see Table 8 and 9. For GPT-2 and BERT, the training loss curves are sketched in Appendix C.

Ablation Study We aim to investigate the effect of masked decay, MVUE and dense fine-tuning introduced in Section 4.2, 3.2, and 4.4. The 16-layer BERT-base is used for ablation study. Results in Table 10 show that: 1) The dense fine-tuning procedure helps to improve accuracy on GLUE by 2 points at most ; 2) MVUE leads to insignificant, controllable accuracy loss; 3) By combining all these techniques together, 2:4 sparse training for transformers achieves comparable accuracy results as dense training.

6.2. Speedup Results

The training acceleration techniques proposed in Section 5 are evaluated using GPT-2 models and RTX3090 GPUs. FP16 mixed precision training is used on all models. The practical speedups of a single FFN layer, a single transformer block, and the entire network, compared to their respective dense counterparts, are reported. All the measured datum contain both forward and backward propagation.

Feed-forward Network Layers For a single FFN layer, we fix $n = 2048$ and change d. Results in Figure 7 show that a FFN layer can be accelerated up to 1.7x faster than its corresponding dense layer.

Transformer Block We measure the acceleration ratio of a transformer block when $n = 512, 1024, 2048$. Results in Figure 7 show that in most cases, a transformer block can be accelerated to 1.3x faster via 2:4 sparsity. To illustrate this, a detailed profile result is given in Appendix D.

End-to-end Acceleration Finally, we test the practical speedups of training GPT-2 models. Results in Table 11 show that our training method conducts up to 1.2x faster than the dense training on a single RTX3090.

7. Conclusions

In this study, we are the first to propose accelerating the pre-training of transformers by 2:4 sparsity. We analyze the limitations of previous 2:4 training methods, including the impropriety in choosing positions and determining values of the masked decay factor, speed bottleneck incurred by computing transposable masks and gated activation functions. We propose a series of techniques to tackle them. Our training method is validated on DeiT, BERT, Transformer-base and GPT-2 models. In particular, we have attained 1.2x end-to-end training acceleration for the GPT-2 774M model without losing its accuracy.

Acknowledgements

We would like to thank Ziteng Wang, Bingrui Li and Haocheng Xi for valuable discussions and help on the training large transformers. This work was supported by the National Key Research and Development Program of China (No. 2021ZD0110502), NSFC Projects (Nos. 62376131, 62061136001, 62106123, 62076147, U19A2081, 61972224), Tsinghua Institute for Guo Qiang, and the High Performance Computing Center, Tsinghua University. J.Z is also supported by the XPlorer Prize.

Impact Statement

Our proposed efficient algorithm can be used to accelerate pre-training large-scale transformers like GLM (Du et al., 2022), LLaMA (Touvron et al., 2023), etc. Recently, large transformers have exhibited remarkable efficacy in various fields such as natural language processing, computer vision, and speech recognition. However, the pre-training stage of large transformers is computationally intensive and time-consuming. For instance, pre-training a GPT-4 can span several months, even using a supercomputer equipped with thousands of GPUs. Thus, acceleration approaches are necessary. Our fully sparse training approach of transformers can potentially accelerate the FFN layers of a model by theoretical 2x faster, without loss of accuracy. Thus, it can be potentially used to save energy and reduce carbon footprint. But this work can also be used to accelerate baleful software, like software that generates malicious contents, which may have a negative impact on human society.

References

Anthony, L. F. W., Kanding, B., and Selvan, R. Carbontracker: Tracking and predicting the carbon footprint of training deep learning models, 2020.

Bengio, Y., Léonard, N., and Courville, A. Estimating or propagating gradients through stochastic neurons for conditional computation, 2013.

Bojar, O., Buck, C., Federmann, C., Haddow, B., Koehn, P., Leveling, J., Monz, C., Pecina, P., Post, M., Saint-Amand, H., Soricut, R., Specia, L., and Tamchyna, A. Findings of the 2014 workshop on statistical machine translation. In *WMT@ACL*, 2014. URL `https://api.semanticscholar.org/CorpusID:15535376`.

BUSATO, F. and POOL, J. Exploiting nvidia ampere structured sparsity with cusparselt [online]. 2020 [visited on 2021-10-10].

Chen, T., Frankle, J., Chang, S., Liu, S., Zhang, Y., Wang, Z., and Carbin, M. The lottery ticket hypothesis for pre-trained bert networks, 2020.

Chen, X., Cheng, Y., Wang, S., Gan, Z., Wang, Z., and Liu, J. Earlybert: Efficient bert training via early-bird lottery tickets, 2021.

Chmiel, B., Hubara, I., Banner, R., and Soudry, D. Minimum variance unbiased n:m sparsity for the neural gradients. In *The Eleventh International Conference on Learning Representations*, 2023. URL `https://openreview.net/forum?id=vuD2xEtxZcj`.

Deng, J., Dong, W., Socher, R., Li, L.-J., Li, K., and Fei-Fei, L. Imagenet: A large-scale hierarchical image database. In *2009 IEEE Conference on Computer Vision and Pattern Recognition*, pp. 248–255, 2009. doi: 10.1109/CVPR.2009.5206848.

Devlin, J., Chang, M.-W., Lee, K., and Toutanova, K. Bert: Pre-training of deep bidirectional transformers for language understanding, 2019.

Du, Z., Qian, Y., Liu, X., Ding, M., Qiu, J., Yang, Z., and Tang, J. Glm: General language model pretraining with autoregressive blank infilling, 2022.

Evci, U., Gale, T., Menick, J., Castro, P. S., and Elsen, E. Rigging the lottery: Making all tickets winners, 2021.

Frankle, J. and Carbin, M. The lottery ticket hypothesis: Finding sparse, trainable neural networks, 2019.

Frankle, J., Dziugaite, G. K., Roy, D. M., and Carbin, M. Stabilizing the lottery ticket hypothesis, 2020.

Geiping, J. and Goldstein, T. Cramming: Training a language model on a single gpu in one day, 2022.

Gokaslan, A. and Cohen, V. Openwebtext corpus. `http://Skylion007.github.io/OpenWebTextCorpus`, 2019.

Han, S., Pool, J., Tran, J., and Dally, W. J. Learning both weights and connections for efficient neural networks, 2015.

Han, S., Mao, H., and Dally, W. J. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding, 2016.

Han, S., Pool, J., Narang, S., Mao, H., Gong, E., Tang, S., Elsen, E., Vajda, P., Paluri, M., Tran, J., Catanzaro, B., and Dally, W. J. Dsd: Dense-sparse-dense training for deep neural networks, 2017.

Hu, Z., Lan, Y., Wang, L., Xu, W., Lim, E.-P., Lee, R. K.-W., Bing, L., and Poria, S. Llm-adapters: An adapter family for parameter-efficient fine-tuning of large language models. *arXiv preprint arXiv:2304.01933*, 2023.

Hubara, I., Chmiel, B., Island, M., Banner, R., Naor, S., and Soudry, D. Accelerated sparse neural training: A provable and efficient method to find n:m transposable masks, 2021.

Karpathy, A. nanogpt. https://github.com/karpathy/nanoGPT/, 2023.

Kingma, D. P. and Ba, J. Adam: A method for stochastic optimization, 2017.

Lasby, M., Golubeva, A., Evci, U., Nica, M., and Ioannou, Y. Dynamic sparse training with structured sparsity, 2023.

Lee, N., Ajanthan, T., and Torr, P. H. Snip: Single-shot network pruning based on connection sensitivity. *arXiv preprint arXiv:1810.02340*, 2018.

Li, Z., Wallace, E., Shen, S., Lin, K., Keutzer, K., Klein, D., and Gonzalez, J. Train big, then compress: Rethinking model size for efficient training and inference of transformers. In *International Conference on machine learning*, pp. 5958–5968. PMLR, 2020.

Loshchilov, I. and Hutter, F. Decoupled weight decay regularization, 2019.

Lu, Y., Agrawal, S., Subramanian, S., Rybakov, O., Sa, C. D., and Yazdanbakhsh, A. Step: Learning n:m structured sparsity masks from scratch with precondition, 2023.

McDanel, B., Dinh, H., and Magallanes, J. Accelerating dnn training with structured data gradient pruning, 2022.

Mishra, A., Latorre, J. A., Pool, J., Stosic, D., Stosic, D., Venkatesh, G., Yu, C., and Micikevicius, P. Accelerating sparse deep neural networks, 2021.

Ott, M., Edunov, S., Baevski, A., Fan, A., Gross, S., Ng, N., Grangier, D., and Auli, M. fairseq: A fast, extensible toolkit for sequence modeling. In *Proceedings of NAACL-HLT 2019: Demonstrations*, 2019.

Papineni, K., Roukos, S., Ward, T., and Zhu, W. J. Bleu: a method for automatic evaluation of machine translation. 10 2002. doi: 10.3115/1073083.1073135.

Radford, A., Wu, J., Child, R., Luan, D., Amodei, D., and Sutskever, I. Language models are unsupervised multitask learners. 2019. URL https://api.semanticscholar.org/CorpusID:160025533.

Raffel, C., Shazeer, N., Roberts, A., Lee, K., Narang, S., Matena, M., Zhou, Y., Li, W., and Liu, P. J. Exploring the limits of transfer learning with a unified text-to-text transformer. *arXiv e-prints*, 2019.

Shazeer, N. Glu variants improve transformer, 2020.

Thakkar, V., Ramani, P., Cecka, C., Shivam, A., Lu, H., Yan, E., Kosaian, J., Hoemmen, M., Wu, H., Kerr, A., Nicely, M., Merrill, D., Blasig, D., Qiao, F., Majcher, P., Springer, P., Hohnerbach, M., Wang, J., and Gupta, M. CUTLASS, January 2023. URL https://github.com/NVIDIA/cutlass.

Touvron, H., Cord, M., Douze, M., Massa, F., Sablayrolles, A., and Jegou, H. Training data-efficient image transformers & amp; distillation through attention. In *International Conference on Machine Learning*, volume 139, pp. 10347–10357, July 2021a.

Touvron, H., Cord, M., Douze, M., Massa, F., Sablayrolles, A., and Jégou, H. Training data-efficient image transformers & distillation through attention, 2021b.

Touvron, H., Lavril, T., Izacard, G., Martinet, X., Lachaux, M.-A., Lacroix, T., Rozière, B., Goyal, N., Hambro, E., Azhar, F., et al. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*, 2023.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., and Polosukhin, I. Attention is all you need, 2023.

Wang, A., Singh, A., Michael, J., Hill, F., Levy, O., and Bowman, S. R. Glue: A multi-task benchmark and analysis platform for natural language understanding. In *BlackboxNLP@EMNLP*, 2018. URL https://api.semanticscholar.org/CorpusID:5034059.

Xu, W., He, X., Cheng, K., Wang, P., and Cheng, J. Towards fully sparse training: Information restoration with spatial similarity. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 36, pp. 2929–2937, 2022.

You, H., Li, C., Xu, P., Fu, Y., Wang, Y., Chen, X., Baraniuk, R. G., Wang, Z., and Lin, Y. Drawing early-bird tickets: Towards more efficient training of deep networks, 2022.

Zhang, Y., Luo, Y., Lin, M., Zhong, Y., Xie, J., Chao, F., and Ji, R. Bi-directional masks for efficient n:m sparse training, 2023.

Zhou, A., Ma, Y., Zhu, J., Liu, J., Zhang, Z., Yuan, K., Sun, W., and Li, H. Learning n:m fine-grained structured sparse neural networks from scratch, 2021.

Zhou, D., Ye, M., Chen, C., Meng, T., Tan, M., Song, X., Le, Q., Liu, Q., and Schuurmans, D. Go wide, then narrow: Efficient training of deep thin networks. In *International Conference on Machine Learning*, pp. 11546–11555. PMLR, 2020.

A. 2:4-spMM

A.1. 2:4 Sparsity

Examples of row-wise, column-wise and transposable 2:4 sparse matrix are shown in Figure 8. Note that transposable 2:4 sparsity aligns with both row-wise and column-wise 2:4 sparsity.

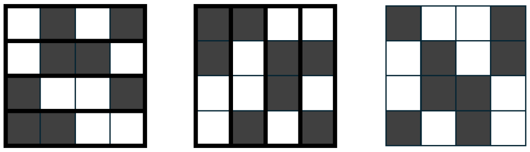

Figure 8. Row-wise 2:4, column-wise and transposable 2:4 sparse matrix.

A.2. Array Layout

The array layout of different types of matrix multiplications are listed in Table 12, which explains why output activations and activation gradients are column-major matrices in FST.

Table 12. Array layout of $\mathbf{MN}$. Here S denotes that the matrix is in row-wise 2:4 sparsity, R denotes row-major dense matrix, and C denotes column-major dense matrix.

M \ N	S	$S^\top$	R	C
S	✗	✗	R	R
$S^\top$	✗	✗	✗	✗
R	✗	C	R	R
C	✗	C	R	R

B. Workflow

The main workflow of a single linear layer in FST process is depicted in Figure 9.

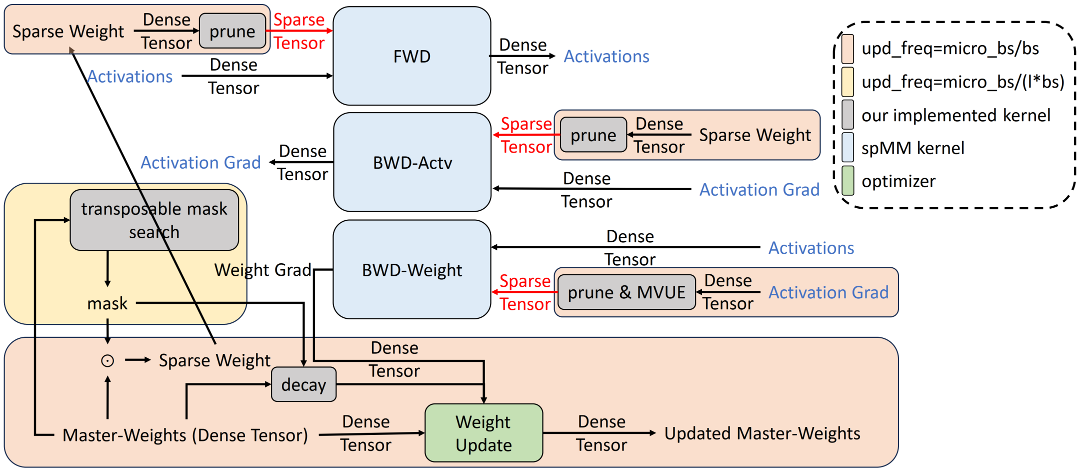

Figure 9. 2:4 sparse training iteration for a layer on a single batch.

C. Training Loss Curve

For BERT-base and GPT-2, we depict training loss curve in Figure 10.

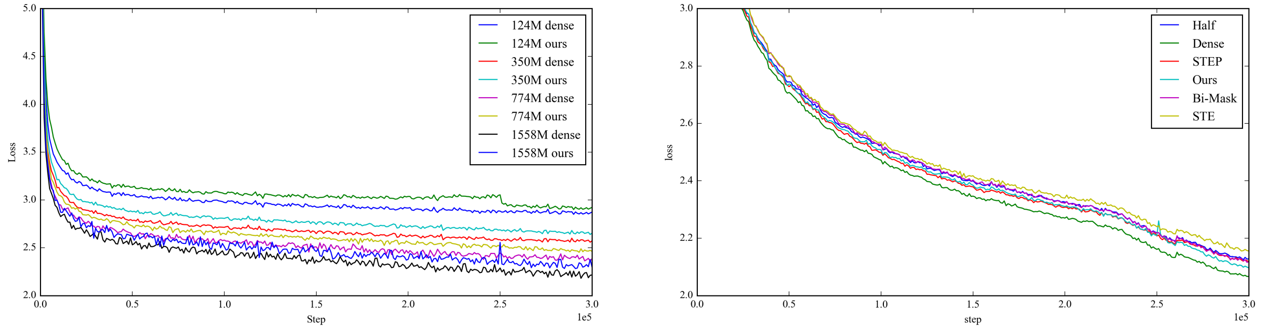

Figure 10. Left: train loss of GPT-2; right: train loss of BERT.

D. Profiling result

To explain how we reach 1.3x block speedup, we profile our code and break down the time costs as shown in the table below; see Table 13.

Table 13. Time costs of each part of our network and the dense model in one iteration per layer. m denotes the accumulation steps over micro batches. Our method is evaluated on GPT-2, with batch size 16, sequence length 1024, embedding dimension 1024 and heads number 16.

				Dense (ms/exec)	Sparse (ms/exec)	Acceleration ratio S	Frequency (exec/iter)
FFN	Linear	FWD	GEMM	12173.8	7305.78	1.666324472	-
		BWD	GEMM	23295	14080.82	1.654378083	-
			MVUE+prune	0	171.4	-	-
			Total	23295	14252.22	1.634482207	-
		Total		**35468.8**	**21558**	**1.645273216**	-
	Others[4]	FWD		167	118.17	-	-
		BWD		65.5	20.03	-	-
		Total		232.5	138.2	-	-
	Total	FWD		12340.8	7423.95	1.662295678	-
		BWD		23360.5	14272.25	1.636777663	-
		Total		35701.3	21696.2	1.645509352	-
Others		FWD		6874.3	7090.55	-	-
		BWD		13920.7	14117.45	-	-
		Total		20795	21208	-	-
Total		FWD		19215.1	14514.5	1.323855455	-
		BWD		37281.2	28389.7	1.313194574	-
		Total		**56496.3**	**42904.2**	**1.316801152**	-
Masked decay				0	45.2	-	$\frac{1}{m}$
Prune weights				0	320.3	-	$\frac{1}{m}$
Transposable mask search				0	634.8	-	$\frac{1}{40m}$

[4] All functions in FFN except linear layers, *i.e.*, activation function and dropout.