Uniform Sparsity in Deep Neural Networks

Saurav Muralidharan [1]

Abstract

Deep neural networks are often highly over-parameterized, and weight pruning or sparsification can be an effective method for reducing both their memory footprints and inference latencies. Among existing pruning strategies, unstructured or fine-grained pruning typically achieves the highest compression ratios and lowest task errors; unfortunately, such irregular and non-uniform sparsity leads to significant load imbalance and consequently degraded performance on parallel architectures. Recent attempts to accelerate unstructured sparsity on GPUs have focused on the 90-99% sparsity regime, where most modern DNNs have been shown to lose considerable accuracy. In this paper, we introduce the *uniform sparsity pattern* that ensures a constant number of non-zero values per row of the sparse matrix, and thus lends itself well to efficient, load-balanced execution on modern parallel architectures. Uniform sparsity achieves useful speedups in both the moderate (50-90%) and high (90%+) sparsity regimes and performs similarly to unstructured sparsity in terms of accuracy. We describe how uniform sparsity is induced on DNN weights and present optimized kernels that accelerate uniform sparsity on GPUs. We evaluate uniform sparsity on a range of real-world networks and synthetic data, and demonstrate mean performance improvements of up to 62% over the NVIDIA cuSparse library at iso-accuracy settings.

1 Introduction

Modern deep neural networks (DNNs) are often over-provisioned for the specific task that they are trained to perform, and model compression techniques such as weight pruning have been shown to be effective at reducing their compute and memory requirements (Hoefler et al., 2021). Weight pruning, or pruning for short, involves the elimination of nonzero values from a trained model. Pruning is typically performed using some kind of thresholding, such as magnitude-based, and can be unstructured (prune any nonzero value) or structured (prune only blocks of nonzero values). The model is typically retrained or "fine-tuned" after pruning for additional training epochs to recover lost accuracy. Among the two kinds of pruning, unstructured pruning typically achieves the highest compression ratios while better retaining accuracy (Hoefler et al., 2021). Unfortunately, the irregular and non-uniform pattern of nonzeros in unstructured sparsity leads to significant data-dependent load imbalance among matrix rows and makes it difficult to accelerate the resulting sparse matrix operations (Merrill & Garland, 2016); this problem is exacerbated on wider parallel architectures such as modern GPUs that are particularly prone to under-utilization. In the absence of any structure, developers must rely on generic sparse matrix operations such as compressed sparse row (CSR) sparse matrix-dense matrix product (SpMM), which are known to achieve much lower throughput than their dense matrix counterparts on accelerators such as GPUs (Bell & Garland, 2009). Recent work has explored ways to accelerate such unstructured sparsity on GPUs (Gale et al., 2020; Chen et al., 2021); however, most of these approaches achieve meaningful speedups over vendor-optimized libraries such as NVIDIA cuSparse (Naumov et al., 2010) only at very high (90%+) sparsities, which is a regime where most neural networks lose considerable accuracy (Hoefler et al., 2021; Mishra et al., 2021). Here, sparsity is defined as the proportion of zero-valued elements to total elements.

To help us understand how to improve upon unstructured sparsity, we draw inspiration from the high-performance computing (HPC) literature, which has explored a number of sparse matrix formats and corresponding computation kernels (Bell & Garland, 2009; Filippone et al., 2017). In particular, the ELLPACK or ELL sparse matrix format (Bell & Garland, 2009) stores an $M \times K$ sparse matrix with at most K' nonzeros per row as two dense M-by-K' arrays: `ellval` of nonzero values and `ellidx` of column indices. All rows are zero-padded to length K'. Combined with the appropriate GPU kernels, ELL has been shown to outperform CSR for a number of sparse matrix operations such as sparse matrix-dense vector product (SpMV) (Bell & Garland, 2009) and sparse matrix-dense matrix product (SpMM) (Abu-Sufah & Ahmad, 2014). It is also well-known that sparse operations on ELL matrices perform best

[1]NVIDIA Corporation, Santa Clara, California, USA. Correspondence to: Saurav Muralidharan <sauravm@nvidia.com>.

Proceedings of the 6^{th} MLSys Conference, Miami Beach, FL, USA, 2023. Copyright 2023 by the author(s).

when the maximum number of nonzeros per row does not substantially differ from the average (Bell & Garland, 2009). Based on this observation, we introduce a sparsity pattern named *uniform sparsity*, where each row of a sparse matrix always contains the same number of nonzeros. This helps us eliminate the usual ELL trade-off, where most rows aren't K elements long, leading to wasted zero padding and/or large overflows.

To induce uniform sparsity on DNN weight matrices, we describe a simple algorithm based on global magnitude pruning and learning-rate rewinding (Renda et al., 2020). We evaluate the accuracy of both unstructured and uniform sparsity on a range of real-world DNNs drawn from domains such as natural language processing (NLP) and image classification, and demonstrate that uniform sparsity exhibits similar accuracy behavior to unstructured sparsity and thus can be used as a drop-in replacement for the latter. We also introduce a set of efficient sparse matrix-dense matrix product (SpMM) kernels for GPUs, adapted from existing ELL SpMM kernels (Bell & Garland, 2009; Abu-Sufah & Ahmad, 2014), that exploit the regular structure of uniformly-sparse matrices and outperform vendor-optimized CSR SpMM kernels. On an NVIDIA A100 GPU, our custom SpMM kernels achieve speedups of up to $1.62\times$ over NVIDIA cuSparse CSR SpMM on real-world weight matrices pruned to iso-accuracy-level sparsities (50-90%) and up to $13.73\times$ on representative synthetic data.

This paper makes the following contributions:

- It introduces the *uniform sparsity* pattern and describes a methodology for systematically inducing it on neural network weights.

- It demonstrates that uniform sparsity exhibits similar accuracy behavior to unstructured sparsity and thus can be used as a drop-in replacement for it on a range of real-world DNNs.

- It presents a set of high-performance GPU SpMM kernels optimized for uniform sparsity and demonstrates that they provide up to $1.62\times$ speedup over the vendor-optimized NVIDIA cuSparse CSR SpMM on real-world weight matrices pruned to iso-accuracy sparsity (50%-90%) and up to $13.73\times$ on representative synthetic data.

2 UNIFORM SPARSITY

In this section, we first provide a brief overview of weight pruning, followed by our algorithm for inducing uniform sparsity on DNN weights.

2.1 Background: Weight Pruning

For a machine learning task such as language modeling, assume we are given a trained *reference* model $\overline{\mathbf{w}} = \operatorname{argmin}_{\mathbf{w}} L(\mathbf{w})$, where $L()$ denotes a *loss function* (e.g., cross-entropy on a given training set), and $\mathbf{w} \in \mathbb{R}^P$. Model compression refers to finding a smaller model Θ that can be applied to the same task and ideally achieves the same accuracy as $\overline{\mathbf{w}}$. Weight pruning, or pruning for short, is a model compression technique that eliminates or "prunes" nonzero values from $\overline{\mathbf{w}}$ to obtain Θ. For a pruned model, *sparsity* is defined as the proportion of zero-valued elements to total elements. A common pruning technique is global magnitude-based pruning (GMP) that eliminates all nonzero values in a model that fall below a certain magnitude threshold (Hoefler et al., 2021). Zeroing out weights often results in considerable accuracy loss compared to the reference model $\overline{\mathbf{w}}$, and there is typically a subsequent *retraining* or *fine-tuning* step to help recover some of the lost accuracy (Renda et al., 2020; Hoefler et al., 2021).

Pruning can be either unstructured (prune any nonzero value) or structured (prune only *blocks* of nonzero values), with unstructured pruning typically achieving the highest compression ratios while better retaining accuracy (Hoefler et al., 2021). Operations on unstructured sparse matrices, however, achieve very poor hardware utilization due to data-dependent load imbalance among matrix rows (Merrill & Garland, 2016). On the other hand, while structured patterns such as filter (He et al., 2018; Luo et al., 2017) and block (Gray et al., 2017) sparsity are more suited to efficient parallel execution, they often suffer from significantly worse accuracy degradation than unstructured sparsity (Hoefler et al., 2021).

2.2 Inducing Uniform Sparsity

To achieve more efficient load-balanced execution of sparse DNNs on modern parallel architectures while still providing similar accuracy characteristics to unstructured sparsity, we introduce *uniform sparsity*. Uniform sparse matrices contain the same number of nonzero values in each row. Figure 1 illustrates our algorithm for pruning a dense weight matrix to 50% uniform sparsity. We first sort the elements in each row in decreasing order of magnitude and then compute the *aggregate value* of each column using a function such as `max` or `sum`. While a number of sophisticated aggregation functions have been explored in the weight pruning literature (Molchanov et al., 2019; Crowley et al., 2018), we use the ℓ^2-norm aggregation function in this paper due to its simplicity and ease of implementation. Next, the columns corresponding to the lowest-magnitude aggregated values are removed based on a threshold. Given a target sparsity for the model, we compute a single *global* magnitude threshold value using all prunable weights in the network. From

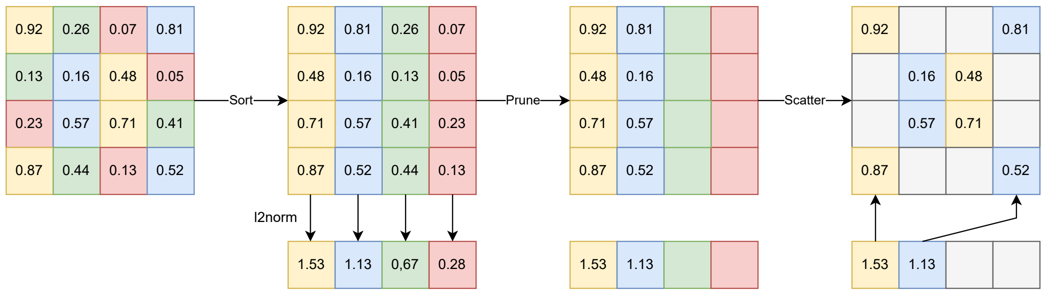


Figure 1: Uniform pruning illustration for 50% sparsity. The ℓ^2 norm is computed over nonzeros shown in the same color (column-wise). Nonzeros with the lowest corresponding ℓ^2 norm values are pruned away.

our experiments, we noticed that using column-wise aggregation with a global, network-wide magnitude threshold resulted in better accuracy recovery than directly pruning the lowest-magnitude values in each row (i.e., local layer-wise pruning). In the Figure, we use a target sparsity of 50% and thus prune away two of the four columns (green and red). The last step of the process is to scatter the remaining nonzero values back to their original locations. We implement the code for uniform-sparse pruning using Python and the PyTorch framework (Paszke et al., 2019).

Learning Rate Rewinding: Once uniform sparsity is applied to DNN weights, we perform retraining or fine-tuning of the model using learning rate rewinding (LR rewinding) (Renda et al., 2020). LR rewinding is a simple accuracy recovery algorithm that works as follows: assume that we start with a reference (dense) model $\overline{\mathbf{w}}$ pruned to obtain a uniform sparse model Θ. If $\overline{\mathbf{w}}$ was originally trained for T epochs, and the learning rate during training was varied from lr_0 in epoch 0 to lr_{T-1} in epoch $T-1$, then LR rewinding involves retraining the pruned model Θ for K additional epochs, where $0 < K < T$ and the learning rate is set (or "rewound") to that of epoch $T-K$. In this paper, we set $K = T$, which implies full retraining of Θ using the original learning rate schedule that goes from lr_0 to lr_{T-1}. While a number of more complex accuracy recovery algorithms exist in the literature (Carreira-Perpinán, 2017; Zhang et al., 2018), LR rewinding, despite being relatively simpler, has been shown to provide state-of-the-art accuracies on a number of large DNNs (Renda et al., 2020). It is possible to further improve the accuracy recovery performance of LR rewinding using complementary strategies such as iterative magnitude pruning (Renda et al., 2020). In this paper, we only perform a single rewinding step with fixed masks and leave the exploration of iterative approaches to future work.

3 GPU-ACCELERATED SPMM

Modern machine learning frameworks such as PyTorch (Paszke et al., 2019) and TensorFlow (Abadi et al., 2016) use matrix multiplication (GEMM) to implement a number of important deep learning operators, such as fully-connected layers, convolutions, and self-attention modules. For instance, the GEMM operation corresponding to the forward propagation (inference) computation of a fully-connected layer in PyTorch is:

$$Y = X \cdot W^T \tag{1}$$

where X represents the batched input with dimensions $N \times K$ (N is the batch size), W is the weight matrix with dimensions $M \times K$, and Y is the batched output matrix with dimensions $N \times M$. The data layout for the three matrices X, Y and W depends on the specific framework being used; for instance, both PyTorch and TensorFlow store matrices in row-major layout (Paszke et al., 2019; Abadi et al., 2016). The GEMM operation in Eq. 1 is typically offloaded to vendor-optimized BLAS libraries such as NVIDIA cuBLAS or cuSparse (Naumov et al., 2010) on GPUs. Since a number of BLAS libraries expect a column-major layout, Eq. 1 is often rewritten as follows:

$$Y^T = W \cdot X^T \tag{2}$$

by interpreting the values of all three matrices above in column-major order, we can avoid explicit transposition of the X and Y matrices (note that W would still be transposed).

When the weight matrix W is sparse, it is typically represented using a sparse matrix format such as compressed sparse row (CSR) and Eq. 1 is implemented using a sparse matrix-dense matrix product (SpMM) instead of a GEMM. While a number of optimized CSR SpMM implementations exist for GPUs (Merrill & Garland, 2016; Naumov et al., 2010; Gale et al., 2020), they all suffer from data-dependent performance degradation due to irregular row lengths in W, which is especially severe on wider parallel architectures and at lower precisions such as float16. The HPC literature has explored a number of sparse matrix formats and associated computation kernels to address the limitations of CSR. In particular, the ELLPACK or ELL sparse matrix format stores the $M \times K$ sparse matrix W with at most K' nonzeros per row as two dense M-by-K' matrices: `ellval` of nonzero values and `ellidx` of column indices (Bell & Garland, 2009). All rows are zero-padded to length K'. When the maximum number of nonzeros per row does not substantially differ from the average, sparse kernels that utilize the ELL format have been shown to significantly outperform CSR-based kernels (Bell & Garland, 2009; Abu-Sufah & Ahmad, 2014). To handle largely uniform matrices with a small number of very long rows, special formats such as HYB have also been proposed, where most of the nonzeros in the matrix are stored using the ELL format with minimal zero-padding, and long-running rows are stored in CSR (Bell & Garland, 2009). With uniform sparsity, we are able to reuse the ELL format while eliminating the need for both zero-padding and special handling of overflowing rows, yielding a perfectly regular structure amenable to load-balanced parallel execution.

ELL Kernel Design: we design and implement optimized GPU kernels in CUDA for the SpMM operation shown in Eq. 2 (column-major). Here, we assume that W is uniform sparse and stored in the ELL format as described above, with the corresponding `ellval` and `ellidx` matrices stored in column-major order. Recall that `ellval` and `ellidx` have dimensions $M \times K'$, and storing both of these matrices in column-major order helps us access elements along the un-pruned M dimension in coalesced fashion on GPUs. Since we use the ELL format, existing GPU kernels for ELL sparse matrix-dense vector product (SpMV) and SpMM helped provide a starting point for our work (Bell & Garland, 2009; Abu-Sufah & Ahmad, 2014). In particular, Abu et al. describe an ELL SpMM kernel that assigns a single thread for processing an entire row of W (Abu-Sufah & Ahmad, 2014); such a parallelization strategy, however, severely under-utilizes modern GPUs such as the NVIDIA A100, which can process significantly more resident threads in parallel than there are rows in W.

Figure 2 illustrates the thread decomposition used by our SpMM kernel, which we denote as `USpMM`. As shown in the Figure, each thread block of the kernel is of size `WarpSize` $\times$ `NumWarps`, where `WarpSize` is 32 and `NumWarps` is the number of warps per block. Each thread block iteratively processes a tile of size `WarpSize` $\times$ `NumWarps` of W before moving onto the next tile along the K' dimension. Each thread loads a single element from the `ellval` and `ellidx` matrices, along with `VSize` elements from a row of X before multiplying them together. `NumWarps`, or the number of warps per block, is computed by multiplying `VSize` with a factor `NWarp`, which in turn controls the width of each thread block. Both `VSize` and `NWarp` are tunable parameters whose values significantly impact the work performed by each thread and consequently the total kernel runtime. We describe the effects of varying

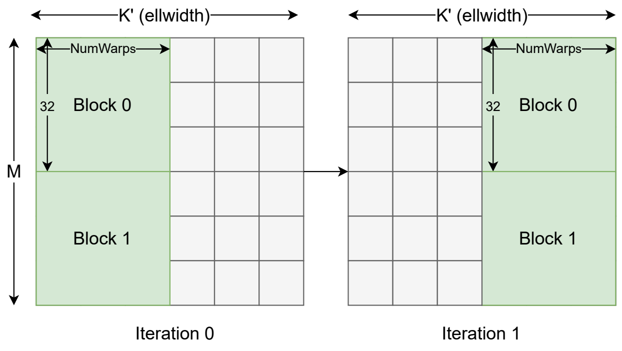


Figure 2: Thread decomposition of the `USpMM` kernel. Each block of threads iteratively processes a tile of size `32`$\times$`NumWarps` items in W.

these tunable parameters in more detail in Section 4.3. The kernel is launched with a grid size of (M / `WarpSize`) $\times$ (N / `VSize`). We provide the full code of our FP16 `USpMM` kernel in Listing 1; the full-precision FP32 version is similar. As shown in lines 24-37, the threads in each warp cooperatively load a partial column of `ellval` and `ellidx` (lines 26 and 27), and multiply each element of W with `VSize` elements in the X matrix, as shown on lines 28-35. Recall that column-major storage of W results in coalesced loads of its column elements. Intermediate partial results are stored in the shared memory array `sh_c`, as shown on lines 38-40. Finally, the partial results across threads in a block are accumulated (lines 44-46) and written out to the result matrix Y on line 48.

Vectorization: we further optimize the performance of our `USpMM` kernel by using vectorized loads and stores. Vectorized memory accesses reduce the total number of instructions and latency, and improve bandwidth utilization. In our vectorized kernel, which we denote as `USpMM-Vec`, we use CUDA's vectorized data types to load multiple values in the `ellval` and `ellidx` matrices at once. Since we perform vectorized loads along the M dimension, the kernel grid size is reduced by a factor of `VecSize`, or the vector width, to (M / (`WarpSize` * `VecSize`)) $\times$ (N / `VSize`). Additionally, the partial results accumulated in the shared memory array `sh_c` of size `VSize` are now split into `VecSize` parts, one for each vector element loaded from W. In this paper, we set `VecSize` to 2, and leave the exploration of larger vector widths to future work. As we demonstrate in Section 4.3, the `USpMM-Vec` kernel outperforms `USpMM` in a number of performance regimes, such as for higher values of N (batch size).

```
template<int VSize, int NumWarps>
__global__ void uniform_spmm_fp16(
    const int m, const int n, const int k,
    const int ellwidth, const half* ellval,
    const int* ellidx, const half* X, half* Y) {

    constexpr int WarpSize = 32;
    const int tid = threadIdx.x+blockIdx.x*blockDim.x;
    const int warpid = threadIdx.x / WarpSize;
    const int lane = threadIdx.x % WarpSize;

    int mpad = ((m + WarpSize-1)/WarpSize)*WarpSize;
    // Index into m, split into WarpSize-sized blocks
    int crow = (blockIdx.x*WarpSize)%mpad+tid%WarpSize;
    int ccol_base = (tid / (mpad * NumWarps)) * VSize;

    __shared__ float sh_c[VSize * NumWarps * WarpSize];
#define sh_c_(X, Y)  sh_c[(X)+(Y)*NumWarps*WarpSize]

    float ctmp[VSize];
    #pragma unroll
    for(int i=0; i<VSize; ++i) ctmp[i] = 0.f;

    if(crow < m) {
        for(int x=warpid; x<ellwidth; x+=NumWarps) {
            float aval = ellval[x*m + crow];
            int   brow = ellidx[x*m + crow];
            #pragma unroll
            for(int i=0; i<VSize; ++i) {
                int bcol = ccol_base + i;
                if(bcol < n) {
                    float bval = X[bcol*k + brow];
                    ctmp[i] += aval * bval;
                }
            }
        }
    }
    #pragma unroll
    for(int i=0; i<VSize; ++i)
        sh_c_(threadIdx.x, i) = ctmp[i];
    __syncthreads();
    if(warpid < VSize) {
        float cval = 0.f;
        #pragma unroll
        for(int i=0; i<NumWarps; ++i)
            cval += sh_c_(lane + i*WarpSize, warpid);
        int ccol = ccol_base + warpid;
        if(crow<m && ccol<n) Y[ccol*m + crow] = cval;
    }
}
```

Listing 1: `USpMM` CUDA kernel code

4 Evaluation

We conduct extensive experiments and fully analyze the accuracy and runtime behavior of uniform sparsity on both synthetic data and two real-world deep neural networks.

4.1 Methodology

We evaluate the accuracy of unstructured and uniform sparsity across a range of sparsities for two real-world machine learning tasks:

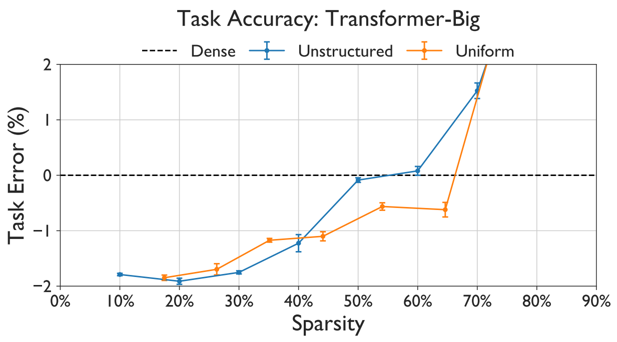
(1) Machine translation using Transformer-Big: we use the Transformer-Big model proposed by Vaswani et al. (Vaswani et al., 2017) for English-to-German machine translation on the WMT'14 en2de dataset. The model contains 213 million parameters and achieves a BLEU score of 28.4. We start with the pre-trained models available in FairSeq (Ott et al., 2019).

(2) Image classification using ResNet50: we use the ResNet50 network (He et al., 2016) trained on the challenging ImageNet task (Deng et al., 2009); specifically, the ILSVRC 2012 version. The model contains 65 million parameters and achieves a top-1 accuracy of 77.71%. We use the ResNet50v1.5 implementation provided by NVIDIA's Deep Learning Examples (DLE) repository (NVIDIA, 2022).

Accuracy Experiments: to prune each model, we use ℓ^2-norm as the column aggregation function, and compute a single global (network-wide) magnitude threshold for pruning given a target sparsity value (see Section 2 for more details). A weight tensor corresponding to a 2D convolution layer with dimensions (C_{out}, C_{in}, H, W)[1] is flattened to a 2D matrix with dimensions $(C_{out}, C_{in} * H * W)$ before pruning. We retrain each model using learning rate rewinding (Renda et al., 2020) for accuracy recovery. Also, as noted in Section 2, we only retrain each model once (i.e., we don't perform iterative rewinding). We run all our retraining experiments 3 times with different random seeds and report the mean task error and corresponding standard deviations. Retraining is performed on an NVIDIA DGX-1 node with $8\times$ NVIDIA V100 GPUs, each containing 32GB of memory.

Runtime Experiments: we collect runtime numbers on an NVIDIA A100 GPU with 6912 cores and 40GB of memory. The NVIDIA CUDA toolkit version is 11.3 with cuBLAS version 11.5 and cuSparse version 11.6. We evaluate the SpMM computation given in Eq. 2 (column-major) in 16-bit floating-point precision using our `USpMM` and `USpMM-Vec` kernels, and compare their performance to cuSparse COO and CSR SpMM baselines. The values for tunable parameters `NWarp` and `VSize` are set to $(2, 8)$ and $(1, 8)$ for the `USpMM` and `USpMM-Vec` kernels, respectively; these parameters are described in more detail in Section 3 and their effect on performance is evaluated in Section 4.3. In addition to pruned DNN weights, we also evaluate the runtime performance

[1]C_{out} and C_{in} denote the number of output and input channels, respectively, while H and W denote the filter dimensions.



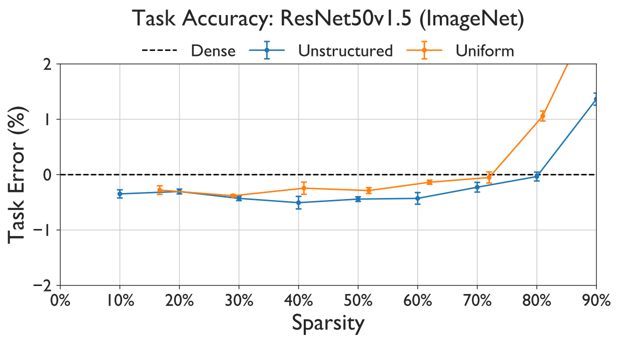


Figure 3: Task error vs. sparsity for each model. Uniform sparsity exhibits similar accuracy behavior to unstructured sparsity for all models.

of our kernels on representative synthetic data; we describe this in more detail in Section 4.3.

4.2 Task Errors

Figure 3 compares the task errors of uniform sparsity to unstructured sparsity for the Transformer-Big and ResNet50 models. The *task error* quantifies the relative accuracy difference between a pruned model and the baseline (dense) model in percentage points, and helps us more uniformly compare network architectures from different modalities such as computer vision and NLP (which may use different absolute accuracy metrics). A task error of 0 or less signifies that the model has lost no accuracy due to pruning. We compute the task error for each pruned model after retraining for sparsities ranging from 10% to 90%. The retraining step is extremely time- and resource-intensive, and requires anywhere from 1 to 8 hours of compute time per (model, sparsity) pair on our DGX-1 node with 8$\times$ NVIDIA V100 GPUs; consequently, we are only able to evaluate the models at a 10% sparsity granularity. For both models, we notice from the Figure that uniform sparsity performs on par with or better than unstructured sparsity for a wide range of sparsity values. Assuming a maximum task error tolerance of 1% [2], the pruned uniform-sparse Transformer-Big and ResNet50 models achieve a maximum sparsity of 65% and 81%, respectively, before they begin to lose accuracy. This is on par with the 60% and 90% maximum sparsities we observe for unstructured sparsity.

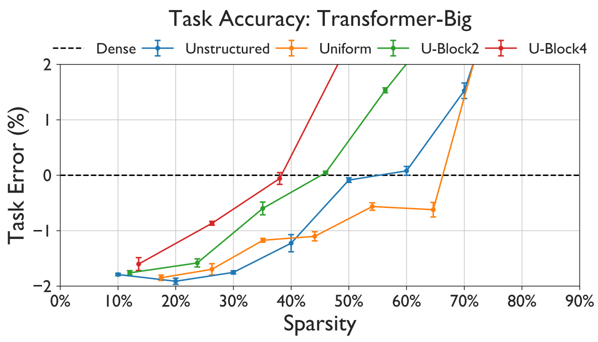


Figure 4: Task errors for blocked-uniform sparsity patterns on Transformer-Big. U-BlockN represents blocked uniform sparsity with block size of N. Accuracy degradation gets significantly worse at higher block sizes.

Improving Accuracy: more sophisticated weight importance estimation criteria such as ones based on Taylor or Fisher importance (Molchanov et al., 2019; Crowley et al., 2018) and/or accuracy recovery strategies such as L-C (Carreira-Perpinán, 2017) and ADMM (Zhang et al., 2018) can help us achieve better accuracies; in this paper, we use learning rate rewinding (Renda et al., 2020) along with a simple ℓ^2-norm aggregation function and global magnitude pruning due to its simplicity and ease of implementation.

Blocked Uniform Sparsity: To understand how uniform sparsity composes with other structured patterns such as block sparsity (Gray et al., 2017), we also implemented a blocked uniform sparsity pattern; here, the pruned matrix has a constant number of *nonzero blocks* per row of blocks. The implementation for this pattern is similar to uniform sparsity, with an additional block-wise aggregation step (using a function such as ℓ^2-norm) prior to intra-row sorting and column-wise aggregation. Such a pattern would theoretically enable us to represent W using a blocked-ELL format, and take advantage of optimized blocked-ELL SpMM kernels from libraries such as NVIDIA cuSparse (Naumov et al., 2010) to achieve speedups over dense GEMM at large-enough block sizes. Figure 4 shows a task error comparison between regular (block size of 1) uniform sparsity and its blocked counterpart for block sizes of 2 and 4 for the Transformer-Big model. We notice that the blocked versions achieve maximum iso-accuracy sparsities of only

[2] slight variations in accuracy may arise across runs due to differing starting conditions at training (e.g., the random seed).

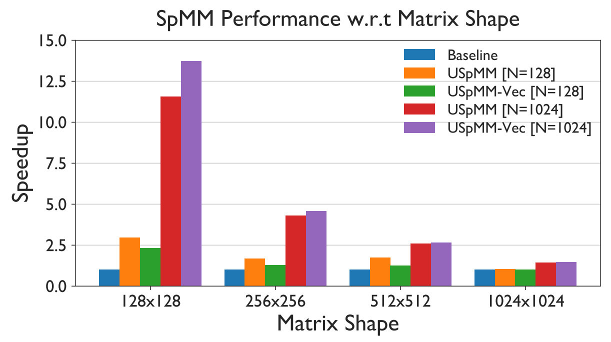


Figure 5: Speedup of our `USpMM` kernels w.r.t. cuSparse CSR, plotted against matrix shape. N represents batch size and is either 128 or 1024. Sparsity of each matrix is fixed to 60%. `USpMM` and `USpMM-Vec` outperform cuSparse CSR SpMM at all matrix and batch sizes.

46% and 38% for block sizes of 2 and 4, respectively; this is in contrast to the 65% and 60% iso-accuracy sparsities achieved by regular uniform sparsity and unstructured sparsity, respectively. Due to this significantly higher accuracy loss, we did not explore this avenue any further.

4.3 Runtime Performance

We measure the runtime performance of our `USpMM` and `USpMM-Vec` kernels on both synthetic data and pruned weights of real-world networks, and compare it to that of COO and CSR SpMM kernels provided with the NVIDIA cuSparse library (Naumov et al., 2010).

Performance on Synthetic Data: we generate a synthetic dataset based on weight matrices drawn from real-world networks such as ResNet50 (He et al., 2016), Transformer (Vaswani et al., 2017) and BERT (Devlin et al., 2018) to study the performance landscape of our kernels in more depth. The dataset covers a variety of problem sizes (M and K), batch sizes (N) and sparsities.

The first set of data we generate pertains to problem sizes or weight matrix dimensions; here, we fix the sparsity to a conservative value of 60%, and prune a set of dense matrices ranging in dimension from 128×128 to 1024×1024. We evaluate performance on two batch sizes: 128 and 1024. These matrix and batch sizes are commonly encountered in both convolutional neural networks and attention-based natural language processing (NLP) models (He et al., 2016; Vaswani et al., 2017; Devlin et al., 2018). Figure 5 shows the results of this experiment. Here, the x-axis represents the problem size, and the y-axis shows the speedups that our `USpMM` and `USpMM-Vec` kernels obtain over cuSparse CSR SpMM. From the Figure, we notice that both our kernels outperform cuSparse CSR (depicted as the "baseline" bar in the Figure) for all matrix and batch sizes, and achieve speedups of up to $13.73\times$. Among the two SpMM variants, we notice that `USpMM` and `USpMM-Vec` tend to perform better at smaller and larger batch sizes, respectively. Further, the largest speedups are obtained for smaller matrix dimensions - we believe that this is primarily due to the values we choose for the `NWarp` and `VSize` parameters, which may be further tuned for specific matrix and batch sizes for better performance. We discuss tuning opportunities in more detail later in this section.

Next, we analyze how performance varies w.r.t. changing sparsities. For this, we generate a set of 40 uniform sparse matrices with (M, N, K) dimensions set to $(128, 128, 128)$ (referred to as the small problem size) or $(1024, 1024, 1024)$ (large problem size) and sparsities ranging from 5% to 95% in increments of 5%. Figure 6 shows the results of this experiment. As shown in the Figure, at a moderate sparsity of 60%, our kernels obtain a speedup of $2.74\times$ (`USpMM`) and $1.50\times$ (`USpMM-Vec`) over cuSparse CSR SpMM for the small and large problem sizes, respectively. These speedups increase to $2.32\times$ (`USpMM-Vec`) and $3.29\times$ (`USpMM-Vec`), respectively, for a sparsity of 95%. By providing meaningful speedups in both moderate and high-sparsity regimes, our SpMM kernels are useful on a wide range of pruned DNNs that have varying accuracy characteristics.

Finally, we study how performance changes with varying batch sizes (N). For this experiment, we use a fixed problem size of 1024×1024 at a conservative 60% uniform sparsity. This is a common weight matrix dimension found in attention-based networks such as Transformer-Big. Accordingly, we vary the value of N from 128 to 16384 corresponding to an effective batch size range of 1 to 128 (assuming a length of 128 tokens per sequence). Figure 7 shows the results for this experiment. While our SpMM kernels perform on par with CSR SpMM at the lowest batch size of 1, they quickly start outperforming the latter at higher batch sizes. At the largest batch size of 128, we obtain a speedup of $1.62\times$ over cuSparse CSR SpMM using the `USpMM-Vec` kernel.

Network-wide Runtime Performance: Table 1 shows the mean inference throughput achieved by our `USpMM` and `USpMM-Vec` kernels on uniform sparse weight matrices from the two DNNs we consider. Inference throughputs are computed at iso-accuracy sparsities, which we define as the maximum sparsity value beyond which task error goes higher than 1% (we provide a 1% error tolerance to account for varying starting conditions in training such as the choice of seed value). The first, second and third columns of the Table list the name of the network, the corresponding dataset, and the number of parameters in the network, respectively. The fourth column lists the batch size (N) we use to evaluate each model - the batch size of 1024 shown for Transformer-

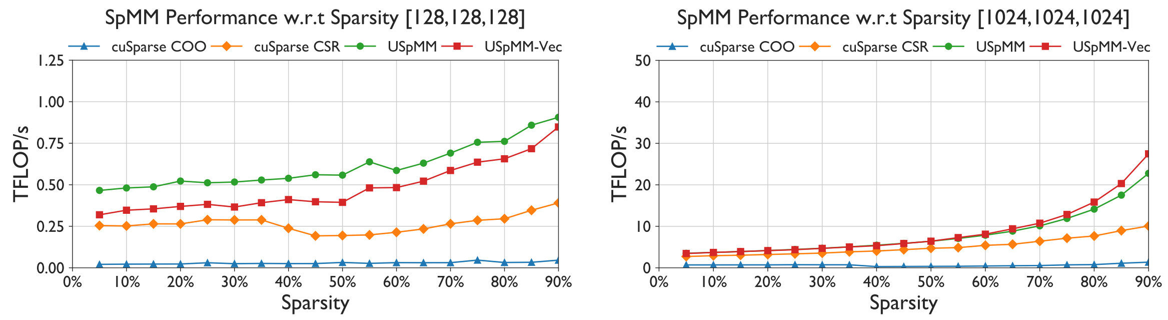


Figure 6: Performance comparison of cuSparse COO and CSR SpMM with uniform SpMM kernels w.r.t varying matrix sparsity. We use (M, N, K) values of $(128, 128, 128)$ (left) and $(1024, 1024, 1024)$ (right).

Model	Dataset	Params	N	Sparsity	Error (%)		Speedup	
					Unif.	Unstr.	USpMM	USpMM-Vec
Transformer-Big	WMT'14 en2de	213M	1024	65%	−0.74	0.11	1.49×	**1.62×**
ResNet50v1.5	ILSVRC 2012	25.5M	128	81%	0.97	−0.01	**1.31×**	1.01×

Table 1: Speedups over cuSparse CSR SpMM for two real-world networks pruned to the sparsity values shown above. Our SpMM kernels achieve speedups of up to $1.62\times$ while never being slower. Unif. and Unstr. stand for uniform and unstructured, respectively. N refers to batch size. Speedups are averaged (geomean) over all prunable weight matrices in each network.

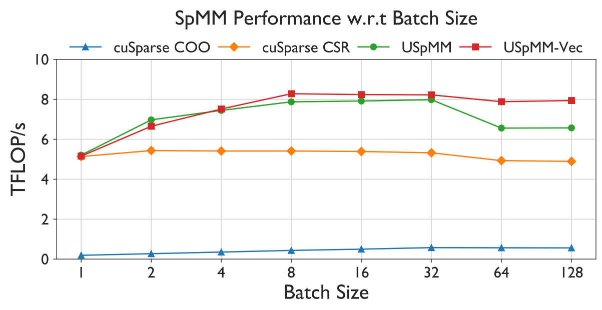


Figure 7: Performance of SpMM kernels w.r.t. batch size. Here, $M = K = 1024$, and $N = B \times \texttt{seqlen}$, where B is the batch size shown on the x-axis, and `seqlen` is 128. Sparsity of the matrix is fixed at 60%.

Big correspond to an effective batch size of 8, assuming a typical sequence length of 128 tokens. Iso-accuracy sparsities for each model are listed in column 5, and we notice that they lie in the moderate sparsity regime (50-90%) - it is thus important for any specialized SpMM kernels that execute these sparse DNNs to provide meaningful speedups over baseline in this range, as opposed to higher sparsities where these models often lose considerable accuracy. The last two columns of the table list the speedups achieved by our `USpMM` and `USpMM-Vec` kernels over cuSparse CSR SpMM. To compute these values, we take a geometric mean of the individual speedups attained for each pruned weight matrix in the network. Our kernels achieve speedups of up to $1.62\times$ over cuSparse CSR SpMM across all models. We notice that the non-vectorized `USpMM` kernel achieves higher speedups than the vectorized version for the ResNet50 model, which contains smaller weight matrices and uses smaller batch sizes, while the vectorized versions dominate in the larger NLP models. Neither of our kernels perform worse than cuSparse CSR SpMM in any setting, even when the wrong kernel is selected for the job (eg., `USpMM-Vec` for ResNet50).

Performance Tuning Opportunities: our SpMM implementations present a number of performance tuning opportunities. As we note earlier in this section, the `USpMM` and `USpMM-Vec` kernels both exhibit differing performance characteristics; while one appears to be more suited for smaller problem sizes and relatively lower sparsities (`USpMM`), the other (`USpMM-Vec`) performs better on larger problem sizes and higher sparsities. In this paper, we do not develop a heuristic to automatically select among the two; we note that this is an interesting input-sensitive autotuning problem, since the best kernel to use depends on input dataset characteristics such as problem and batch sizes, and sparsity values. We plan to explore the use of existing input-adaptive code variant selection systems (Muralidharan et al., 2014; Tillet & Cox, 2017) to automate this selection in the future.

Similarly, the selection of tunable parameters `NWarp` and `VSize` in our SpMM implementation has a significant impact on performance (see Section 3 for more details). To demonstrate this, we plot the performance of `USpMM` for a

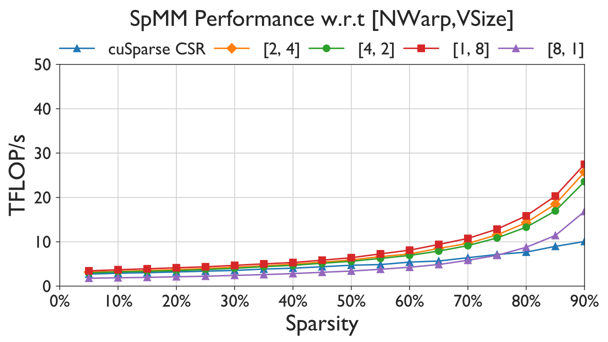


Figure 8: Performance of the vectorized uniform kernel for various values of `[NWarp,VSize]`. Here, M=N=K=1024.

problem size of $M = N = K = 1024$ across varying sparsities and four different configurations of (`NWarp`,`VSize`). Figure 8 shows the results of this experiment. For the specific problem and batch sizes above, we notice considerable variation in performance as the values of both these tunable parameters are changed. At the highest (95%) sparsity, the performance difference between the lowest and highest performing configurations is 14.48 TFlop/s. In this paper, we fix the values of `NWarp` and `VSize` to $(1, 8)$ and $(2, 8)$ for `USpMM` and `USpMM-Vec`, respectively, based on our experiments. However, we note that there is plenty of scope for tuning these values further using input-adaptive tuning strategies.

Other opportunities for improving performance include using a larger vectorization width (we use a width of 2 for `USpMM-Vec`) and using 16-bit column indices for `ellidx`. We leave the exploration of these strategies to future work.

5 Related Work

Pruning away blocks of nonzero values from a model (i.e., structured pruning) has been shown to be more amenable to performance optimization than unstructured or fine-grained pruning (Hoefler et al., 2021). Examples of structured pruning include filter (Luo et al., 2017; He et al., 2018), neuron (Hu et al., 2016), and vector/block pruning (Gray et al., 2017; Chen et al., 2021). While performant, structured sparsity comes at a cost: significantly degraded model accuracy, especially at higher sparsities (Hoefler et al., 2021). To achieve a better balance between fine- and coarse-grained pruning, recent work has explored less aggressive constraints on the sparsity pattern. For instance, NVIDIA Ampere-generation GPUs include *sparse tensor cores* that accelerate the 2:4 sparsity pattern; here, at most two out of four contiguous elements in W may be non-zero, resulting in a constant 50% sparsity (Mishra et al., 2021). Similarly, Yao et al. (Yao et al., 2019) present balanced sparsity, which provides a more generalized version of Ampere sparsity (here, each row of W is split into multiple equal-sized blocks and each block has the same number of nonzeros), and Liu et al. (Liu et al., 2022) introduce the density bound block (DBB) sparsity pattern and a corresponding systolic-array-based CNN accelerator. While less constrained sparsity patterns often achieve better performance, they are not perfect; for instance, 2:4 and DBB sparsity require specialized hardware support (with 2:4 resulting in a constant sparsity of 50%), while balanced sparsity was originally evaluated on relatively smaller networks such as a 2-layer LSTM model and its accuracy behavior on larger networks is unknown.

A separate body of work has also examined the problem of accelerating unstructured-sparse DNNs on parallel architectures (Gale et al., 2020; Wang, 2020). Gale et al. (Gale et al., 2020) introduce the Sputnik library of SpMM and sampled dense–dense matrix multiplication (SDDMM) GPU kernels optimized for the CSR format, while Wang et al. (Wang, 2020) present an inspector-executor-based code generator targeting unstructured sparsity. Such approaches have been shown to outperform older versions of the NVIDIA cuSparse library, especially at higher (90%+) sparsity regimes.

The HPC literature has explored a number of sparse matrix formats and corresponding linear algebra kernels tailored for specific applications and sparsity patterns (Bell & Garland, 2009; Abu-Sufah & Ahmad, 2014; Filippone et al., 2017). However, as Gale et al. demonstrate, the matrix dimensions, sparsity degrees, and sparsity patterns encountered in HPC applications differ significantly from those found in machine learning (Gale et al., 2020), making their direct use impractical and/or inefficient for the latter. Nevertheless, they can still act as excellent building blocks - in our case, we reuse the well-known ELL matrix format (Bell & Garland, 2009) to represent weight matrices and adapt and extend existing GPU ELL SpMV and SpMM implementations to build our own optimized kernels.

6 Conclusions

This paper has introduced uniform sparsity, a pattern which ensures a constant number of nonzero values per row of the sparse matrix, and thus lends itself well to efficient, load-balanced execution on modern parallel architectures. We have provided a simple algorithm for inducing uniform sparsity on real-world DNNs, and demonstrated that uniform sparsity exhibits similar accuracy behavior to unstructured sparsity, thus making it suitable as a drop-in replacement for the latter. We have also described our optimized GPU SpMM kernels tailored for uniform sparsity and demonstrated that they achieve speedups of up to $1.62\times$ over NVIDIA cuSparse CSR SpMM on real-world DNN weight matrices pruned to iso-accuracy-level sparsities (50-90%) and up to $13.73\times$ on representative synthetic data. With the initial framework in place, we now plan to explore more sophisticated accuracy recovery strategies for uniform-sparse models, and additional performance optimization opportunities for our SpMM kernels.

ACKNOWLEDGMENTS

This material is based upon work supported by DARPA under Contract No. HR0011-18-3-0007. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the U.S. Government. Distribution Statement "A" (Approved for Public Release, Distribution Unlimited).

REFERENCES

Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., et al. TensorFlow: A System for Large-Scale Machine Learning. In *12th USENIX symposium on operating systems design and implementation (OSDI 16)*, pp. 265–283, 2016.

Abu-Sufah, W. and Ahmad, K. On implementing sparse matrix multi-vector multiplication on GPUs. In *2014 IEEE Intl Conf on High Performance Computing and Communications, 2014 IEEE 6th Intl Symp on Cyberspace Safety and Security, 2014 IEEE 11th Intl Conf on Embedded Software and Syst (HPCC, CSS, ICESS)*, pp. 1117–1124. IEEE, 2014.

Bell, N. and Garland, M. Implementing sparse matrix-vector multiplication on throughput-oriented processors. In *Proceedings of the conference on high performance computing networking, storage and analysis*, pp. 1–11, 2009.

Carreira-Perpinán, M. A. Model Compression as Constrained Optimization, with Application to Neural Nets. Part I: General Framework. *arXiv preprint arXiv:1707.01209*, 2017.

Chen, Z., Qu, Z., Liu, L., Ding, Y., and Xie, Y. Efficient Tensor Core-Based GPU Kernels for Structured Sparsity under Reduced Precision. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '21, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450384421. URL `https://doi.org/10.1145/3458817.3476182`.

Crowley, E. J., Turner, J., Storkey, A., and O'Boyle, M. Pruning neural networks: is it time to nip it in the bud? 2018.

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. Ieee, 2009.

Devlin, J., Chang, M.-W., Lee, K., and Toutanova, K. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. *arXiv preprint arXiv:1810.04805*, 2018.

Filippone, S., Cardellini, V., Barbieri, D., and Fanfarillo, A. Sparse matrix-vector multiplication on GPGPUs. *ACM Transactions on Mathematical Software (TOMS)*, 43(4): 1–49, 2017.

Gale, T., Zaharia, M., Young, C., and Elsen, E. Sparse GPU kernels for deep learning. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–14. IEEE, 2020.

Gray, S., Radford, A., and Kingma, D. P. GPU Kernels for Block-sparse Weights. *arXiv preprint arXiv:1711.09224*, 3, 2017.

He, K., Zhang, X., Ren, S., and Sun, J. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 770–778, 2016.

He, Y., Kang, G., Dong, X., Fu, Y., and Yang, Y. Soft filter pruning for accelerating deep convolutional neural networks. *arXiv preprint arXiv:1808.06866*, 2018.

Hoefler, T., Alistarh, D., Ben-Nun, T., Dryden, N., and Peste, A. Sparsity in Deep Learning: Pruning and growth for efficient inference and training in neural networks. *arXiv preprint arXiv:2102.00554*, 2021.

Hu, H., Peng, R., Tai, Y.-W., and Tang, C.-K. Network trimming: A data-driven neuron pruning approach towards efficient deep architectures. *arXiv preprint arXiv:1607.03250*, 2016.

Liu, Z.-G., Whatmough, P. N., Zhu, Y., and Mattina, M. S2TA: Exploiting Structured Sparsity for Energy-Efficient Mobile CNN Acceleration. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pp. 573–586, 2022. doi: 10.1109/HPCA53966.2022.00049.

Luo, J.-H., Wu, J., and Lin, W. Thinet: A filter level pruning method for deep neural network compression. In *Proceedings of the IEEE international conference on computer vision*, pp. 5058–5066, 2017.

Merrill, D. and Garland, M. Merge-Based Sparse Matrix-Vector Multiplication (SpMV) Using the CSR Storage Format. *SIGPLAN Not.*, 51(8), feb 2016. ISSN 0362-1340. doi: 10.1145/3016078.2851190. URL `https://doi.org/10.1145/3016078.2851190`.

Mishra, A., Latorre, J. A., Pool, J., Stosic, D., Stosic, D., Venkatesh, G., Yu, C., and Micikevicius, P. Accelerating sparse deep neural networks. *arXiv preprint arXiv:2104.08378*, 2021.

Molchanov, P., Mallya, A., Tyree, S., Frosio, I., and Kautz, J. Importance estimation for neural network pruning. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 11264–11272, 2019.

Muralidharan, S., Shantharam, M., Hall, M., Garland, M., and Catanzaro, B. Nitro: A framework for adaptive code variant tuning. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, pp. 501–512. IEEE, 2014.

Naumov, M., Chien, L., Vandermersch, P., and Kapasi, U. cuSparse Library. In *GPU Technology Conference*, 2010.

NVIDIA. NVIDIA Deep Learning Examples. 2022. URL `https://github.com/NVIDIA/DeepLearningExamples`.

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. *arXiv preprint arXiv:1904.01038*, 2019.

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Kopf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J., and Chintala, S. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In Wallach, H., Larochelle, H., Beygelzimer, A., d'Alché-Buc, F., Fox, E., and Garnett, R. (eds.), *Advances in Neural Information Processing Systems 32*, pp. 8024–8035. Curran Associates, Inc., 2019.

Renda, A., Frankle, J., and Carbin, M. Comparing rewinding and fine-tuning in neural network pruning. In *International Conference on Learning Representations*, 2020. URL `https://openreview.net/forum?id=S1gSj0NKvB`.

Tillet, P. and Cox, D. Input-aware auto-tuning of compute-bound HPC kernels. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–12, 2017.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., and Polosukhin, I. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

Wang, Z. SparseRT: Accelerating unstructured sparsity on GPUs for deep learning inference. *arXiv preprint arXiv:2008.11849*, 2020.

Yao, Z., Cao, S., Xiao, W., Zhang, C., and Nie, L. Balanced sparsity for efficient DNN inference on GPU. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, pp. 5676–5683, 2019.

Zhang, T., Zhang, K., Ye, S., Li, J., Tang, J., Wen, W., Lin, X., Fardad, M., and Wang, Y. Adam-ADMM: A unified, systematic framework of structured weight pruning for DNNs. *arXiv preprint arXiv:1807.11091*, 2018.