EFFICIENT GPU KERNELS FOR N:M-SPARSE WEIGHTS IN DEEP LEARNING

Bin Lin [1 2 *] **Ningxin Zheng** [1 *] **Lei Wang** [1 *] **Shijie Cao** [1] **Lingxiao Ma** [1] **Quanlu Zhang** [1] **Yi Zhu** [1] **Ting Cao** [1] **Jilong Xue** [1] **Yuqing Yang** [1] **Fan Yang** [1]

ABSTRACT

N:M sparsity is becoming increasingly popular for its potential to deliver high model accuracy and computational efficiency for deep learning. However, the real-world benefit of N:M sparsity is limited as there is a lack of dedicated GPU kernel implementations for general N:M sparsity with various sparsity ratios. In this work, we introduce *nmSPARSE*, a library of efficient GPU kernels for two fundamental operations in neural networks with N:M sparse weights: sparse matrix-vector multiplication (SpMV) and sparse matrix-matrix multiplication (SpMM). By exploiting the intrinsic balance characteristic of N:M sparsity, *nmSPARSE* kernels rearrange irregular computation and scattered memory accesses in sparse matrix multiplication into hardware-aligned regular computation and conflict-free memory accesses at runtime. When evaluated on NVIDIA A100 GPU, *nmSPARSE* kernels achieve up to $5.2\times$ speedup on SpMV and $6.0\times$ speedup on SpMM over the fastest baseline. End-to-end studies on transformer models demonstrate that using *nmSPARSE* outperforms other baselines.

1 INTRODUCTION

To reduce the model size of Deep Neural Networks (DNN) and accelerate model inference, weight pruning has been extensively studied in academia and industry (Han et al., 2015; Gale et al., 2020a; Zheng et al., 2022). The sparsity pattern in model weights plays a critical role as it fundamentally affects the trade-off between model accuracy and inference efficiency of a compressed model. Element-wise (EW) sparsity, which prunes weights individually, minimally impacts model accuracy but struggles to take advantage of commodity GPU due to irregular computation and scattered memory accesses (Mao et al., 2017; Gale et al., 2020a; Mishra et al., 2021). Vector-wise (VW) or block-wise (BW) sparsity, which prunes groups of weights with a coarser granularity, leads to efficient execution on GPU but suffers from deteriorated model accuracy. Such an accuracy-efficiency trade-off remains a longstanding challenge for sparse DNNs.

Recently, N:M sparsity has emerged as a promising alternative to achieve both high model accuracy and high inference efficiency (Cao et al., 2019; Mishra et al., 2021). N:M sparsity essentially imposes a balanced distribution on non-zero weights, e.g., element-wisely enforcing N non-zero elements in every M elements (EW-N:M sparsity). Previous works demonstrate such an N:M distribution has minimal or imperceptible impact on model accuracy on a wide range of common tasks and model architectures, and many algorithmic studies are striving to enhance the accuracy of N:M sparsity (Zhou et al., 2021; Sun et al., 2021; Pool & Yu, 2021; Oh et al., 2022; Holmes et al., 2022). Additionally, N:M sparsity allows for efficient implementation with customized hardware due to its more regular computation and memory access patterns (Cao et al., 2019). Commodity hardware like NVIDIA Ampere architecture even supports EW-2:4 sparsity, a special case of the more general N:M sparsity, in its Sparse Tensor Core (Mishra et al., 2021). The N:M sparsity concept can also be extended to VW and BW sparsity, where N non-zero vectors exist in every M vectors (VW-N:M sparsity) and N non-zero blocks exist in every M blocks (BW-N:M sparsity), respectively.

Despite the promising results, the real-world benefit of general N:M sparsity is limited. There is a lack of GPU kernels dedicated to general N:M sparsity with various sparsity ratios. *cuSPARSELt* leverages the Sparse Tensor Core in NVIDIA Ampere architecture (Mishra et al., 2021), but only supports EW-2:4 weight sparsity which restricts the balance window M to 4 and the sparsity ratio to 50%. Despite the excellent performance on 2:4 sparsity for SpMM, *cuSPARSELt* does not support the SpMV operation, which is critical for auto-regressive generative model inference. While various models or even layers usually have different levels of redundancy, thus favoring different sparsity ratios which are typically ranging from 50% to 90%. Moreover, although VW-N:M and BW-N:M can further boost inference efficiency, currently there is no dedicated GPU implementation to leverage them.

[*]Equal contribution [1]Microsoft Research Asia [2]Tsinghua University. Correspondence to: Shijie Cao <shijiecao@microsoft.com>, Lingxiao Ma <lingxiao.ma@microsoft.com>.

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

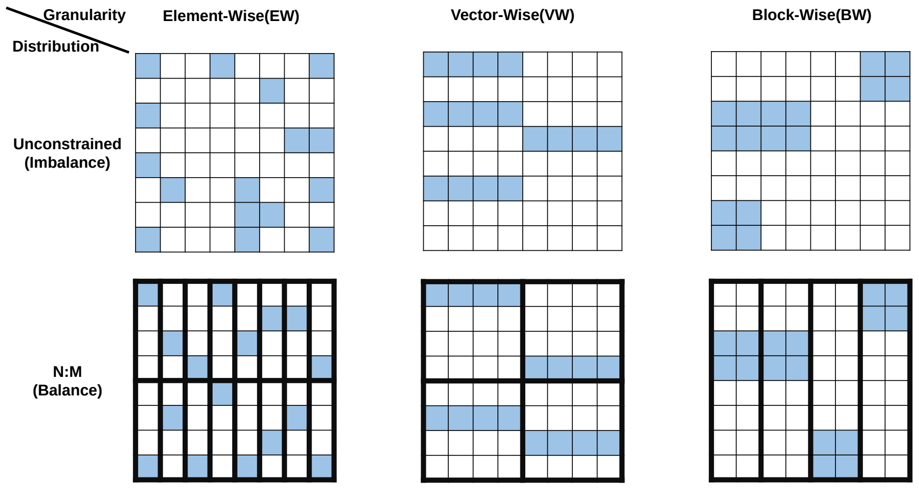


Figure 1. A unified representation of sparsity patterns with granularity and distribution. *nmSPARSE* implements GPU kernels for sparse weights with the N:M balanced distribution.

In this work, we present *nmSPARSE*, a highly-optimized GPU library of SpMV and SpMM kernels for general N:M sparsity. To the best of our knowledge, *nmSPARSE* is the first GPU library that supports general N:M sparse weights in DNN with various sparsity ratios. The key insight of *nmSPARSE* is to rearrange irregular computation and scattered memory accesses in sparse matrix multiplication into hardware-aligned regular computation and conflict-free memory accesses at runtime by leveraging the intrinsic balance distribution of N:M sparsity.

Specifically, *nmSPARSE* first proposes a condensed representation for general N:M sparsity to reduce the memory footprint and decoding overhead. Such a condensed representation encodes seemingly *irregular and sparse* nonzeros into a *regular and dense* matrix for efficient data loading and explicitly indicates the indexes to load demanded elements in the multiplied dense matrix. To address the challenge of irregular and scattered memory accesses to the multiplied dense matrix, *nmSPARSE* kernels leverage the banked shared memory to service concurrent memory requests from parallel threads and schedule memory requests to perfectly match the *conflict-free access pattern* and *conflict-free broadcast access pattern* to the shared memory. By extending the N:M distribution to VW/BW sparsity, *nmSPARSE* takes the advantage of both balanced distribution and larger granularity to offer superior performance with aligned memory accesses and Tensor Core support.

We evaluate the performance of *nmSPARSE* kernels on sparse matrices with a large set of synthetic sizes and typical sizes from real models. Results on NVIDIA A100 GPU show that *nmSPARSE* achieves up to $5.2\times$ speedup on SpMV and $6.0\times$ speedup on SpMM. We have also integrated *nmSPARSE* to SparTA (Zheng et al., 2022), an end-to-end framework that supports DNN inference with sparsity. Our evaluation shows using *nmSPARSE* outperforms other baselines. By open sourcing *nmSPARSE*[1], we hope it can benefit efficient sparse model inference and motivate new innovations on N:M sparsity in both machine learning and system communities.

2 Background and Motivation

2.1 Weight Pruning and Sparsity Patterns

Weight pruning aims to find and remove redundant weights that have little impact on model accuracy. Pruning directly reduces the model memory footprint, while the ultimate goal is to achieve practical inference speedup on commodity hardware which is highly influenced by the sparsity pattern and sparsity ratio.

In this work, we provide *a unified representation of sparsity patterns with two abstractions*: **granularity** and **distribution**, as shown in Figure 1. Blue entries correspond to non-zero values, while white entries indicate zero values. In terms of granularity, we classify sparsity patterns into three categories: element-wise(EW), vector-wise(VW) and block-wise(BW), as shown in three columns respectively. As the granularity increases, the sparse model is easier to achieve speedup, but maintaining the same sparsity ratio while preserving model accuracy becomes increasingly challenging (Wen et al., 2016; Mao et al., 2017; Gray et al., 2017; Liu et al., 2022). In terms of distribution, we classify sparsity patterns into two types: Unconstrained(imbalance)-distributed and N:M(balance)-distributed, as shown in two rows. Each pattern in the second row can be seen as being

[1]`https://github.com/microsoft/SparTA/tree/nmsparse`

added an N:M constraint (N non-zero elements/groups in every M elements/groups) to its corresponding pattern in the first row, noted as EW-N:M, VW-N:M and BW-N:M sparsity patterns. Note that the N:M distribution is along the reduction dimension k in matrix multiplication as each output is calculated by multiplying and accumulating elements along the k dimension.

Intuitively, N:M sparsity is promising in maintaining model accuracy because it only slightly restricts the locality distribution of non-zero elements/groups while the distribution of non-zeros inside each window of size M is still unrestricted. Numerous algorithmic studies are actively recovering and enhancing the accuracy of N:M sparsity (Mishra et al., 2021; Zhou et al., 2021; Oh et al., 2022; Holmes et al., 2022; Zhang et al., 2022b). Meanwhile, N:M sparsity has shown great potential in achieving efficient parallel executions on customized hardware and GPU (Cao et al., 2019; Mishra et al., 2021). In summary, N:M distribution is a promising way to mitigate the trade-off between accuracy and speedup in sparse DNN.

2.2 Sparse Matrix Computation

As weights are pruned to be sparse, the most frequent and time-consuming operation of DNN inference changes from dense general matrix-matrix multiplication (GEMM) to sparse GEMM: SpMV and SpMM, the two most fundamental operations to support DNN inference with sparse weights.

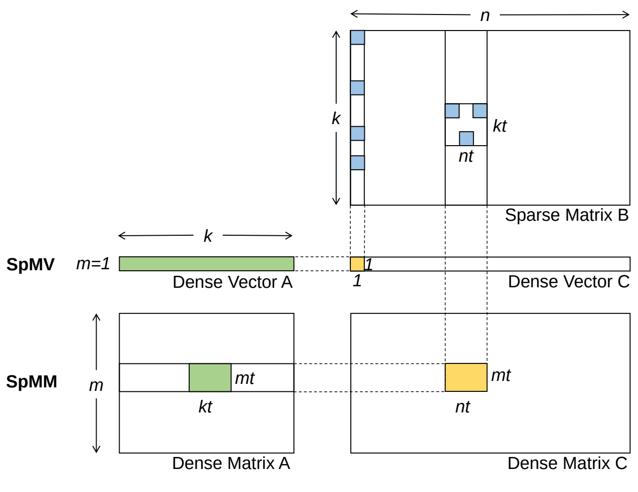


Figure 2. SpMV and SpMM denoted as $C_{m\times n} = A_{m\times k} \times B_{k\times n}$.

In the rest of this paper, we denote sparse matrix multiplication as $C = A \times B$, where B is the sparse weight matrix, A and C are the input and output dense matrix (as shown in Figure 2), following the conventional representation of GEMM in DNNs. The size of the sparse weight matrix B is $k \times n$, which transforms k-dimensional features into n-dimensional features. In SpMV, A is an input dense vector of size $1 \times k$, and C is an output dense vector of size $1 \times n$, each of which is a k-element dot product of A and its corresponding sparse column in B. In SpMM, A becomes a matrix of $m \times k$ and C becomes a matrix of $m \times n$. Each element in C with location (i,j) is the dot product of the i-th dense row in A and the j-th sparse column in B. To avoid ambiguity, we refer to lowercase n and m as the matrix dimensions, and uppercase N and M as the distribution configuration in N:M sparsity.

Challengs of sparse matrix multiplication on GPU. Though requiring less data loading and computation theoretically, sparse GEMM poses new challenges to efficient GPU execution compared to dense GEMM. First, sparse matrix formats encode the indexes of non-zero values, which necessitate decoding prior to computation. Decoding overheads could easily overshadow the benefit of reduced computation. Second, the unbalanced distribution of non-zeros might cause workload skew among parallel threads if kernels are not carefully designed. Last but not least, the irregularity in sparsity patterns leads to irregular computation and scattered memory accesses, which could decrease bandwidth utilization and eventually stall the parallel execution. Increasing the granularity is a straightforward way to address the aforementioned challenges, but sacrifices model accuracy.

New opportunities of N:M sparsity. Besides the almost negligible impact on model accuracy, N:M sparsity also provides fresh possibilities for efficient and highly parallel sparse GEMM implementations on GPUs. The intrinsic balance property of N:M sparsity ensures automatic and complete workload balance of matrix partitioning for parallel computation. Furthermore, the N:M distribution constraint on the sparse matrix B also limits the locality of memory accesses to the dense vector/matrix A for parallel threads during SpMV/SpMM on GPUs. This property allows kernels to take advantage of locality and attain more efficient data loading. Such a minor alteration in the sparsity pattern can result in a significant improvement in efficient kernel design, which will be explicated in the following section.

3 *nmSPARSE* KERNEL DESIGN

This section describes the critical design of *nmSPARSE* kernels to unleash the potential of N:M sparsity. *nmSPARSE* implements highly-optimized SpMV and SpMM GPU kernels for sparse weights that are pruned to satisfy N:M sparsity patterns. *nmSPARSE* aims to leverage the balance characteristic that N:M sparsity patterns inherently offer to tackle the irregularity challenges in sparse matrix multiplication, and ultimately maximize the utilization of GPU memory bandwidth and computing resources.

To reduce the memory footprint and decoding overhead, *nmSPARSE* first compresses N:M-sparse weight matrices to a condensed representation(§ 3.1). *nmSPARSE* leverages the banked shared memory in GPU architecture to rearrange irregular computation and scattered memory accesses into hardware-aligned regular computation and conflict-free memory accesses(§ 3.2). Especially for VW/BW-N:M sparsity, *nmSPARSE* takes the advantages of both balanced distribution and larger granularity to offer superior performance with aligned memory accesses and Tensor Core support(§ 3.3).

3.1 Condensed Representation of N:M sparsity

Various sparse matrix representations (or formats, e.g., CSC, CSR, COO, etc.) are adopted to reduce the memory footprint in sparse matrix computation by only storing non-zero values and their indices. However, existing sparse formats may not be the best option for storing sparse weights with N:M patterns and performing SpMV and SpMM operations on them. The main reason for this is that previous sparse matrix representations and the GPU kernels running on them are customized and optimized for applications with extremely sparse matrices (e.g. with sparsity ratio more than 95% or even 99%). In contrast, the sparsity ratios in DNNs are normally moderate (e.g., from 50% to 90%). Therefore, directly applying previous sparse matrix representations to N:M sparse weights will introduce storage and decoding overheads for efficient and parallel kernel implementations, resulting in low GPU resource utilization.

In Figure 3, we present the condensed representation dedicated to N:M sparsity patterns in *nmSPARSE*, similar to formats in (Cao et al., 2019) and (Mishra et al., 2021). The top three are the dense represented EW-, VW- and BW-N:M sparse matrices, and the bottom three show their condensed representations accordingly. As the N:M distribution is vertical (along the k dimension), we condense non-zeros vertically, as shown by the blue arrow. Once the non-zero values are condensed, the resulting data array is guaranteed to be a standard 2D array with the same number of entries in each row and column. The index array records the positions of non-zero elements or groups inside the balance window of size M that they belong to. Furthermore, the metadata of granularity and distribution (abbreviated as Gran. and Dist. in Figure 3) is stored separately.
The condensed sparsity representation offers three benefits. First, loading non-zeros of a tile or an entire column of the sparse matrix is efficient, which is important to parallel computing across tiles (for SpMM) and columns (for SpMV). Second, storage for indices is reduced. As the index array only records non-zeros' positions inside each balance window of size M, each index occupies up to a maximum $\log_2 M$ bits. Third, the index is decoding-friendly. Indices of non-zeros in sparse matrix B correspond to the addresses of elements that need to be loaded in dense matrix A. These read addresses can be generated through an element-wise operation on the index array.

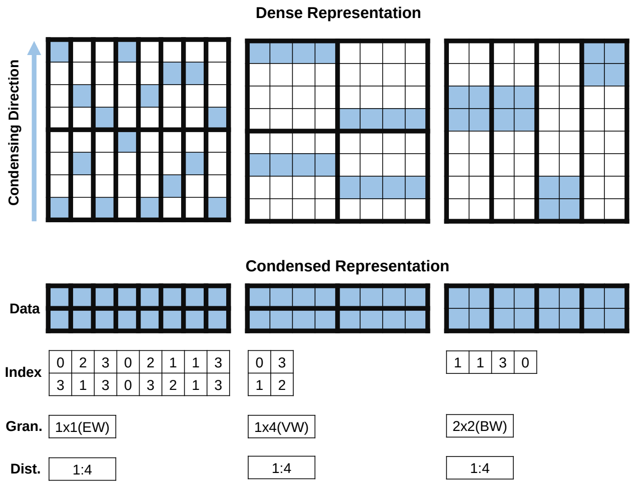


Figure 3. Condensed representation of weight matrices with EW-/VW-/BW-N:M sparsity patterns.

3.2 EW-N:M Sparsity

EW sparsity has the most irregular pattern, resulting in irregular and scattered memory accesses as the biggest challenge. In dense matrix multiplication, loading two input matrices (both A and B) is aligned and sequential. While in sparse matrix multiplication, though requiring fewer loads and computations as many elements are zeros in the sparse matrix, how to efficiently load demanded elements and skip unnecessary elements in both A and B matrices is essential to truly unleash the potential afforded by sparsity. Thanks to the balance intrinsic of N:M sparsity and the condensed representation shown in Figure 3, loading tiles or entire rows of sparse matrix B is aligned and sequential. However, the locations of demanded elements of dense matrix A are irregular. Frequent irregular memory accesses will significantly reduce the memory bandwidth utilization, eventually stalling parallel execution.

In *nmSPARSE*, we leverage the *shared memory* in GPU architecture to achieve efficient memory access to demanded elements in vector/matrix A for SpMV/SpMM operations on EW-N:M sparsity. Shared memory is divided into equally sized memory banks that can be accessed simultaneously, in order to achieve high memory bandwidth for concurrent memory requests from parallel threads. In NVIDIA GPUs, as parallel threads are scheduled and executed in warps consisting of 32 concurrent threads, the number of banks in shared memory is also 32. Therefore, efficiently leveraging shared memory is critical to the bandwidth utilization and final performance of SpMV and SpMM kernels.

The only performance issue with shared memory is the *bank conflict*. When multiple threads in a warp request addresses that map to the same memory bank, a bank conflict occurs. The hardware can only respond to conflicting requests sequentially, decreasing the effective bandwidth. In contrast, if the memory access addresses of 32 threads in a warp map to 32 distinct memory banks, then they can be served simultaneously, yielding no bank conflicts and maximum bandwidth utilization (noted as conflict-free access). A special case is when multiple threads in a warp access the same bank but with exactly the same address, this can be served with a broadcast mechanism supported in hardware, which is not regarded as a bank conflict (noted as conflict-free broadcast access).

Combining the advantage of both the banked shared memory of GPU and the balanced distribution of N:M sparsity, we design and implement SpMV and SpMM GPU kernels for EW-N:M sparsity with scheduled memory requests to eliminate bank conflicts in shared memory. Notably, our SpMV and SpMM kernels perfectly match the conflict-free access pattern and conflict-free broadcast access pattern to the shared memory respectively.

3.2.1 *SpMV: Conflict-Free Access*

A straightforward approach to parallelizing SpMV is that each thread is assigned to compute one output element which is the dot product of the dense input vector and a sparse column. However, this limits the degree of parallelism to the number of columns in the sparse matrix, which can potentially lead to an under-utilization of the GPU due to fewer threads executing in parallel. Furthermore, as the non-zeros' locations of different sparse columns are random and unrestricted, concurrent accesses to random locations of the dense vector will cause conflicts and reduce bandwidth utilization.

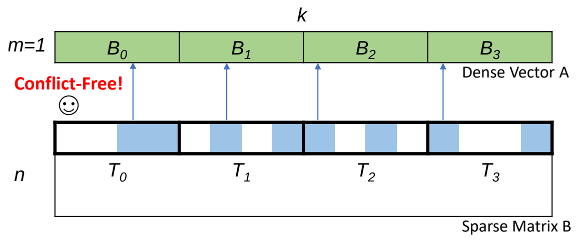


Figure 4. Conflict-free access pattern to the shared memory in SpMV kernels. *T*: threads in a warp and *B*: banks in the shared memory.

The SpMV design in *nmSPARSE* further partitions each sparse column into smaller sub-columns to exploit more parallelism as the column size (k) is sufficiently large in model weights and naturally partitioned into balanced regions of size M in N:M sparsity patterns. Our SpMV kernel achieves such inter-column and intra-column parallelism with conflict-free access to shared memory through subtly organizing vector data and mapping parallel threads. A simple example of parallel threads accessing to distinct memory banks is diagrammed in Figure 4. Note that the sparse matrix is transposed for a better illustration. Each thread (*T*) is assigned to compute the dot product of a sub-column. The workload across threads is naturally balanced given the intrinsic balance characteristic of EW-N:M sparsity. When the dense vector A is loaded from global memory to shared memory, vector A is partitioned according to the partitioning of sparse columns and stored in distinct memory banks (*B*), as shown by green chunks in the figure. By such a data organization and thread mapping, memory requests from different threads in a warp are guaranteed to access different memory banks, therefore bank conflicts are eliminated. Figure 5 shows the pseudo-code for our SpMV kernel.

```
__global__ void SpMV(float *A, float *B, int *B_IDX, float *C) {
    __shared__ float A_shared[SHARE_TILE_A_LEN];
    float A_reg;
    float B_reg;
    float C_reg[MINIBATCH] = 0;
    // Load A_tile from global memory to shared memory.
    LoadTile(A_shared, A);
    // Main loop.
    for(int i = 0; i < BANK_NUM * (1-SPARSITY); i++){
        LoadReg(B_reg, B);
        // Get indices for loading scattered elements in A_tile
        access_idx = GetIndex(B_IDX);
        for(j = 0; j < MINIBATCH; j++){
            LoadRegWithIdx(A_reg, A_shared, access_idx);
            CalculOnReg(A_reg, B_reg, C_reg);
        }
    }
    Store(C, C_reg);
}
```

Figure 5. CUDA pseudo-code for SpMV.

3.2.2 *SpMM: Conflict-Free Broadcast Access*

Tiling is a widely-adopted approach for efficient GEMM implementation on GPU to reduce global memory accesses by taking advantage of the shared memory. Tiling-based kernels implement GEMM by partitioning the output matrix into tiles, which are then assigned to thread blocks. Each thread block computes the output tile (C_tile) by stepping through the k dimension in tiles, loading the input tile of A and B matrices (A_tile, B_tile), and multiplying and accumulating them into the output.

When applying tiling to SpMM, the challenge arises in how to efficiently load the demanded elements in A_tiles and B_tiles. Loading non-zeros in the B_tile is straightforward, thanks to the balance distribution of N:M sparsity and its condensed representation. The SpMM kernel in *nmSPARSE* achieves efficient accesses to demanded elements in A_tile by storing A_tile in shared memory and mapping threads to columns in B_tile. Coupled with the N:M balance distribution in EW-N:M sparsity, our kernel design matches the conflict-free broadcast access pattern to shared memory. While removing such an N:M constraint will cause bank conflicts. Figure 6 illustrates these two cases with examples and Figure 7 shows the pseudo-code for our SpMM kernel.

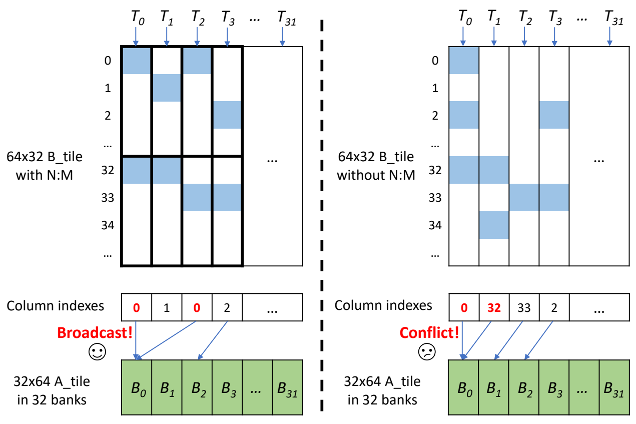


Figure 6. Conflict-free access pattern to the shared memory in SpMM kernels.T: threads in a warp and B: banks in the shared memory.

The left of Figure 6 shows an example of computing an SpMM tile of EW-N:M sparsity in *nmSPARSE*. In this example, the dense A_tile is with the shape of 32x64, and the sparse B_tile is with the shape of 64x32. The N:M configuration in B_tile is 1:32 meaning 1 non-zero element in every 32 elements vertically. Elements in A_tile are directly stored in shared memory in a row-major manner. B_tile is illustrated in dense form, but non-zeros are stored in the condensed format. As parallel threads are mapped to columns in B_tile, we list the column indices of the non-zeros in each column, as shown in the middle. The column indexes also indicate the read addresses of parallel threads to the shared memory. When executing 32 threads in a warp, the read addresses of 32 threads are guaranteed to be restricted in a range of 32, for example [0,32). That is to say, although multiple concurrent threads may access the same bank, their addresses are guaranteed to be the same. In this example, 2 threads access exactly the same address (0) inside the same bank (Bank0). The broadcast mechanism can perfectly solve this conflict.

In contrast, the right of Figure 6 shows a corresponding example of EW-unconstrained sparsity. Because of the lack of the N:M balance constraint, the read addresses and their ranges inside a warp are not restricted. For example, 2 threads access different entries (0 and 32) inside the same bank (Bank 0). This is where a bank conflict occurs and can not be solved by broadcast.

3.3 VW/BW-N:M sparsity

Applying the N:M distribution to VW/BW sparsity constructs VW/BW-N:M sparsity, which is endowed with the efficiency advantages of both balanced distribution and larger

```
__global__ void SpMM(float *A, float *B, int *B_IDX, float *C) {
    __shared__ float A_shared[SHARE_TILE_A_LEN];
    __shared__ float B_shared[SHARE_TILE_B_LEN];
    float A_reg[REG_TILE_A_LEN];
    float B_reg[REG_TILE_B_LEN];
    float C_reg[REG_TILE_A_LEN][REG_TILE_B_LEN] = 0;
    // ThreadBlock loop.
    for(; nnz > 0; nnz -= kBlockItemsK) {
        // Load tile from global memory to shared memory.
        LoadTile(A_shared, A);
        LoadTile(B_shared, B);
        // Register loop.
        for(int i = 0; i < K_BLOCK_TILE_LEN * (1-SPARSITY); i++){
            LoadReg(B_reg, B_shared);
            // Get indices for loading scattered elements in A_tile
            access_idx = GetIndex(B_IDX);
            LoadRegWithIdx(A_reg, A_shared, access_idx);
            CalculOnReg(A_reg, B_reg, C_reg);
        }
    }
    Store(C, C_reg);
}
```

Figure 7. CUDA pseudo-code for SpMM.

granularity. *nmSPARSE* also implements SpMM kernels for VW- and BW-N:M sparsity by leveraging the above-mentioned advantages to offer superior performance.

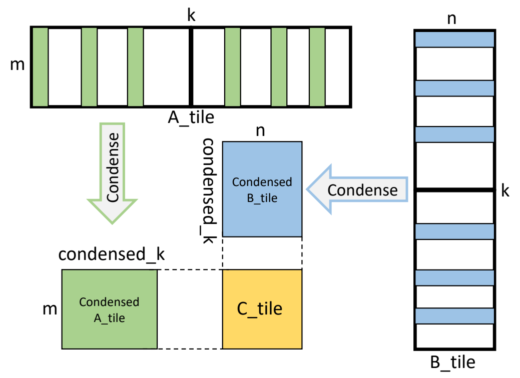


Figure 8. VW/BW-N:M sparsity reduces data loads of A matrix and enables leveraging Tensor Cores.

More regular computation and more continuous memory accesses make GPU kernels more capable of saturating the memory bandwidth and computing resources. Taking VW-N:M sparsity as an example shown in Figure 8, it further improves the tiling friendliness of SpMM. First, as the non-zeros are aligned to vectors (in blue) in B_tile, the demanded elements in A_tile are corresponding vectors (in green) as well. Compared to the tiling scheme in EW-N:M sparsity, memory accesses to A_tile are aligned, and not the entire A_tile needs to be loaded. Second, because of the balance distribution in the sparse matrix B, the size of condensed B_tile/A_tile remains the same for each output C_tile. Furthermore, the regular tiling scheme can be easily mapped to the dedicated matrix unit in hardware. (e.g. TensorCore in Nvidia GPU) to maximize the speed of matrix multiplication.

4 Implementation

Pruning. The implementation of our pruning algorithm is based on ASP (Nvidia; Pool & Yu, 2021), an open-source pruning library to generate sparse networks developed by Nvidia. In order to support generating general N:M sparse weights, we made 3 major extensions to ASP. 1) The original ASP only implements pruning with 2:4 sparsity and fails to set M larger than 10, we extend it to support arbitrary N:M settings. $M = 32$ is a good setting in practice because it can easily align with the hardware banks to achieve good performance and cover a wide range of sparsity ratios from 3% to 97%. When $M > 32$, it becomes difficult to avoid bank conflicts, so the performance will be impacted negatively. 2) We extend ASP to support VW- and BW-N:M sparsity for *nmSPARSE* to achieve higher speedup. 3) We further enable layer-wise sparsity ratio configuration for ASP because various DNN layers favor different sparsity ratios.

GPU kernels. We implement our SpMV and SpMM kernels for N:M sparsity with different granularities respectively. For VW-N:M sparsity with the granularity 64 and BW-N:M sparsity with the granularity 64x64, we implement their kernels based on cutlass by leveraging the high-performance MMA building block. In cases where the m dimension of SpMM is very small, as seen in low batch size and auto-regressive scenarios, its characteristics are more similar to that of SpMV, with low arithmetic intensity and high memory intensity. Therefore, we have observed that selecting the SpMV implementation leads to faster results. Our best practice is to use SpMV kernels directly to implement SpMM if $m \leq 8$.

End-to-end model inference. In order to support end-to-end model inference with N:M sparsity, we integrate *nmSPARSE* to SparTA (Zheng et al., 2022), an end-to-end framework to support DNN inference with sparsity. We also integrate *cuSPARSELt* and other baseline libraries into SparTA to make a direct and fair comparison.

5 Evaluation

We evaluate *nmSPARSE* kernels on both operator benchmarks and end-to-end models by comparing them with state-of-the-art dense and sparse libraries and DNN compilers. Our findings are as follows: 1) The EW-N:M kernels of *nmSPARSE* achieve up to $5.2\times$ and $2.1\times$ speedup for SpMV and SpMM operators respectively over the fastest baselines. 2) With the increase of granularity, *nmSPARSE* kernels can further achieve up to $6.0\times$ speedup for SpMM operators over the fastest baselines. 3) End-to-end studies on Transformer demonstrate that *nmSPARSE* outperforms other baselines.

Evaluation Setup. Our evaluation is on an Azure NC24ads_A100_v4 VM equipped with 24 AMD EPYC 7V13 CPU cores and an NVIDIA Tesla A100-PCIE-80GB GPU, installed with Ubuntu 20.04 LTS and CUDA 11.3.

For operator benchmarks, we compare *nmSPARSE* with state-of-the-art dense and sparse libraries, including cuBLAS and cuBLASLt, two vendor-specific libraries for dense matrix computation from NVIDIA; cuSPARSE and cuSPARSELt, two vendor-specific sparse matrix libraries from NVIDIA; and Sputnik (Gale et al., 2020b), a state-of-the-art library of sparse linear algebra kernels for deep learning. Note that cuSPARSELt is implemented with the Sparse Tensor Core in Ampere architecture. For baseline libraries, we opt for the CSR format for the element-wise and vector-wise sparse matrices, and the Block-ell format for the block-wise sparse matrices. We further add SparTA (Zheng et al., 2022), a state-of-the-art end-to-end DNN compiler for model sparsity; Rammer (Ma et al., 2020), a state-of-the-art open-source DNN compiler that *nmSPARSE* and SparTA are integrated with; and TensorRT (v8.5), a vendor-specific inference engine from NVIDIA, in the end-to-end comparison.

5.1 Operator Benchmarks

Operator shapes and sparsity ratios. The operator configurations of the benchmarks are from both real-world models and synthetic ones. Specifically, we extract the matrices in the widely-used BERT-Large model (Devlin et al., 2018) and the state-of-the-art open-source pre-trained language model OPT (Zhang et al., 2022a). In the synthetic configuration set, we sample some square shapes ranging from 1024 to 8192. To satisfy the requirements in various deep learning scenarios (i.e., online inference, offline inference, and training), we evaluate *nmSPARSE* with different batch sizes (i.e., m = 1, 256, 1024, and 4096). Detailed shape configurations are shown in Appendix A. We use shapes with m = 1 to evaluate SpMV kernels and other shapes with larger m to evaluate SpMM kernels. To be consistent with the relatively moderate sparsity ratios found in deep learning models, we pick three typical sparsity ratios for all benchmark evaluations: 50%, 75%, and 90%.

5.1.1 SpMV

Figure 9 shows the performance comparisons for the 8 SpMV operators in our benchmark. Comparing to the dense matrix computation library cuBLAS, *nmSPARSE* achieves speedup in all cases. Specifically, *nmSPARSE* achieves a $1.4\times$ speedup on average (up to $1.9\times$) in 50% sparsity ratio, a $2.1\times$ speedup on average (up to $2.3\times$) in 75% sparsity ratio, and a $3.5\times$ speedup on average (up to $5.1\times$) in 90% sparsity ratio. Comparing with cuSPARSE, *nmSPARSE* achieves $21.2\times$, $16.3\times$ and $11.3\times$ on average (up to $42.4\times$,

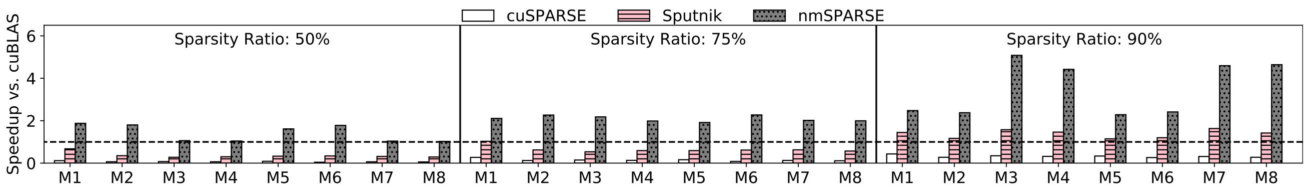


Figure 9. Speedup of *nmSPARSE* on CUDA Cores for SpMV operators with different sizes.

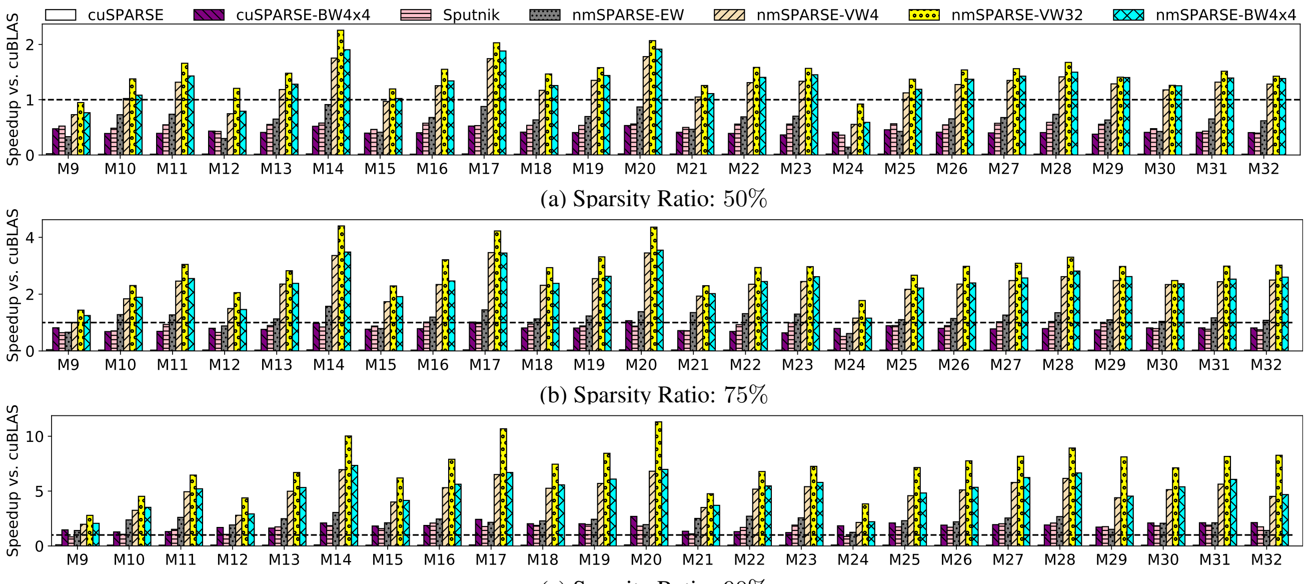


Figure 10. Speedup of *nmSPARSE* on CUDA Cores for SpMM operators with different sizes at 50%, 75%, and 90% sparsity ratio.

28.0× and 16.8×) when the sparsity ratio is 50%, 75% and 90%, respectively. Comparing with Sputnik, *nmSPARSE* achieves 4.0×, 3.3× and 2.5× on average (up to 5.2×, 4.1× and 3.3×) when the sparsity ratio is 50%, 75% and 90%, respectively. Because SpMV is primarily a memory-bound operation due to its nature of low arithmetic intensity, we did not see significant kernel speedup with coarse-grained sparse matrices.

5.1.2 SpMM

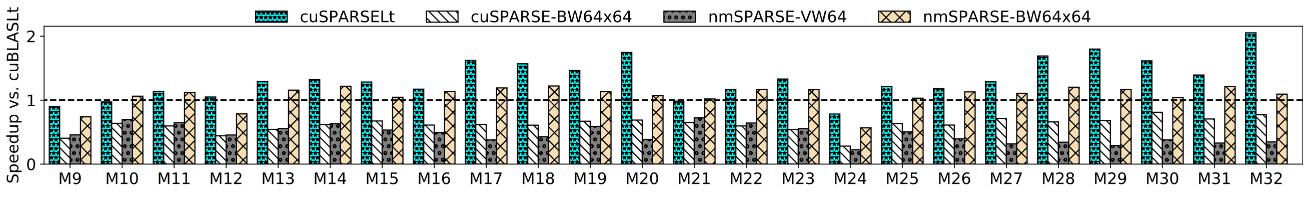
We evaluated *nmSPARSE* on both FP32 precision which uses the CUDA Cores and INT8 precision which can leverage the Tensor Cores.

CUDA Core Figure 10 shows the performance comparisons for the 24 SpMM operators in our benchmark on FP32 CUDA Cores. For cuSPARSE, we evaluated both the fine-grained sparsity (marked as cuSPARSE) and the 4x4 block-wise sparsity (cuSPARSE-BW4x4). For *nmSPARSE*, we evaluated the EW-sparsity (*nmSPARSE*-EW), the VW-sparsity with a granularity of 4 (*nmSPARSE*-VW4) and 32 (*nmSPARSE*-VW32), and the BW-sparsity with a granularity of 4x4 (*nmSPARSE*-BW4x4).

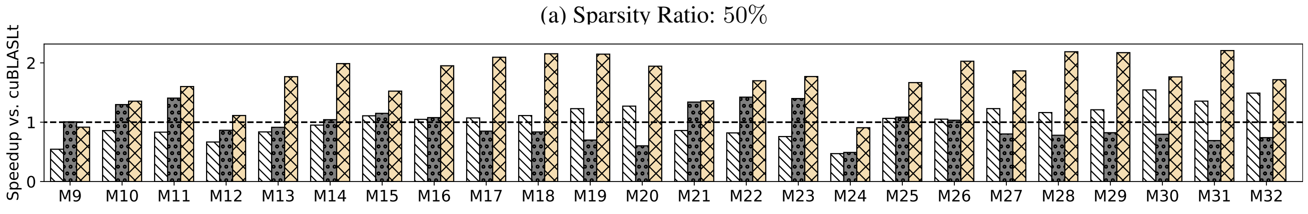
Compared to cuBLAS, *nmSPARSE* can achieve speedups in most cases of each sparse ratio. When comparing with the state-of-the-art baselines (i.e., cuSPARSE, cuSPARSE-BW4x4 and Sputnik), *nmSPARSE*-EW can achieve 1.2×, 1.4× and 1.3× on average (up to 1.7×, 1.9× and 2.1×) over the fastest one of these baselines when the sparsity ratio is 50%, 75% and 90%, respectively. With the increase of the granularity, *nmSPARSE* can achieve higher speedups. Specifically, *nmSPARSE*-VW4 can achieve 2.4×, 2.7× and 2.7× on average (up to 3.3×, 3.8× and 3.8×) over the fastest one of these baselines when the sparsity ratio is 50%, 75% and 90%, while *nmSPARSE*-VW32 can further achieve 2.9×, 3.4× and 4.0× on average (up to 3.9×, 4.7× and 6.0×) over the fastest baseline when the sparsity ratio is 50%, 75% and 90%. Besides, *nmSPARSE*-BW4x4 can achieve 2.5×, 2.8× and 2.8× on average (up to 3.5×, 4.0× and 4.0×) over the fastest baseline when the sparsity ratio is 50%, 75% and 90%.

Tensor Core Figure 11 shows the performance comparisons for the 24 SpMM operators in our benchmark on INT8 Tensor Cores. All the baselines and *nmSPARSE* leverages the hardware-specialized Tensor Cores in GPU. Moreover, cuSPARSELt leverages the Sparse Tensor Core in Ampere architecture.

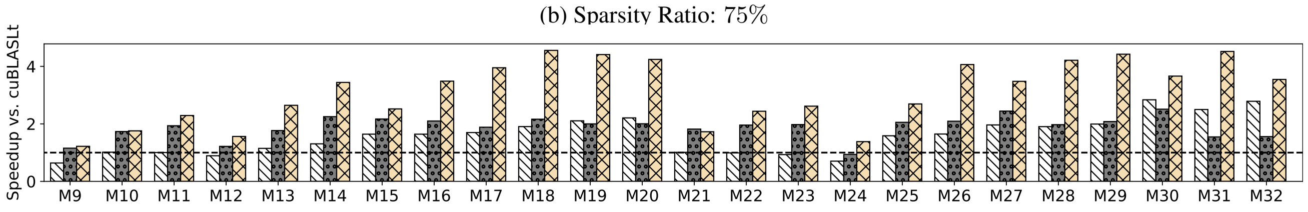
With 50% sparsity ratio, cuSPARSELt performs the best



(a) Sparsity Ratio: 50%



(b) Sparsity Ratio: 75%



(c) Sparsity Ratio: 90%

Figure 11. Speedup of *nmSPARSE* on Tensor Cores for SpMM operators with different sizes at 50%, 75%, and 90% sparsity ratio.

Kernels	cuSPARSE	cuSPARSE-BW4x4	Sputnik	nmSPARSE-EW	nmSPARSE-VW4	nmSPARSE-VW32	nmSPARSE-BW4x4
latency(ms)	6.79	0.26	0.31	0.24	0.12	0.08	0.11
dram_utilization(%)	0.15	0.64	3.26	3.19	7.58	9.39	7.81
shared_utilization(%)	0.71	33.88	10.45	45.88	52.05	47.83	46.82
fp_unit_utilization(%)	2.87	23.77	18.72	21.79	43.13	66.2	44.59

Table 1. Kernel profiling results of operator shape M25 at a sparsity ratio of 90%.

on average because it executes on the hardware-customized Sparse Tensor Core in Ampere architecture. But its limitation is also only supporting the 2:4 sparsity with a 50% sparsity ratio. *nmSPARSE* can outperform cuSPARSELt with larger sparsity ratios. Comparing with cuSPARSE-BW64x64, nmSPARSE-BW64x64 can achieve $1.8\times$, $1.7\times$ and $2.0\times$ on average (up to $2.2\times$, $2.3\times$ and $2.8\times$) when the sparsity ratio is 50%, 75% and 90%.

5.1.3 *Microbenchmark*

To validate the speedup of *nmSPARSE*, we conducted a microbenchmark using M25 as a representative operator shape with a sparsity ratio of 90%. We profiled different kernels using *Nsight Compute* and analyzed DRAM utilization, shared memory utilization, and single precision floating point unit utilization to demonstrate the speedup. As shown in Table 1, *nmSPARSE* kernels achieve higher memory throughput utilization and floating point unit utilization.

5.2 Application Study on Transformer

We choose the Transformer as our application study of end-to-end accuracy and latency because Transformer-based models (Vaswani et al., 2017) have achieved state-of-the-art results on language (Devlin et al., 2018), image (Liu et al., 2021), and speech tasks (Chen et al., 2022).

Experimental setup. We prune a pre-trained bert-large model on the SQuAD-1.1 dataset with various N:M sparsity patterns and sparsity ratios, to verify the pruning effectiveness of N:M sparsity and evaluate the end-to-end latency speedup of our *nmSPARSE* kernels under different settings. As the first layers are sensitive to model accuracy, we set the first 4 out of 24 layers to be dense. To make a direct comparison, we prune the weight matrices of the rest layers with all sparsity patterns represented in Figure 1 (EW v.s. EW-N:M, VW v.s. VW-N:M, BW v.s. BW-N:M). For each sparsity pattern, we prune the model with a sparsity ratio of 50%, 75%, and 90% respectively. We finetune for 3 epochs for all settings.

Pruning effectiveness of N:M sparsity. Unlike the 2:4 sparsity supported by Nvidia Ampere architecture, general N:M sparsity does not restrict the configuration of N and M. Thus we first assess the pruning effectiveness of different N and M under the same sparsity ratio. Taking EW-N:M as an example, results in Table 2 demonstrate that varying N:M settings has no obvious impact on model accuracy as long as the sparsity ratio remains constant. Intuitively, increasing the window size M will relax the locality constraint of non-zeros, indicating better model accuracy. But this needs rigorous theoretical analysis by machine learning experts.

Previous studies demonstrate that adding N:M distribution to EW sparsity has a minimal impact on model accuracy (Zhou et al., 2021; Sun et al., 2021; Zhang et al., 2022b). As we extend EW-N:M sparsity to VW-N:M and BW-N:M sparsity, we evaluate the F1 scores of EW-, VW- and BW-N:M sparsity and compare with their original sparsity patterns without N:M distribution under 3 sparsity ratios. In this experiment, we set M = 32. As Table 3 shows, results of VW and BW sparsity are consistent with that of EW sparsity: no obvious or deterministic impact of adding N:M distribution on model accuracy is observed.

N:M	**2:4**	**4:8**	**8:16**	**16:32**
F1	90.59	90.81	90.76	90.95
N:M	**1:4**	**2:8**	**4:16**	**8:32**
F1	88.80	89.57	89.79	90.09

Table 2. F1 scores under different N:M settings. Under a fixed sparsity ratio, different N:M settings have no obvious impact on model accuracy.

Sparsity Ratio	50%	75%	90%
EW	90.72	90.10	86.23
EW-N:M	90.95	90.09	82.02
VW4	89.26	83.61	79.88
VW4-N:M	90.432	87.26	79.91
VW32	88.57	80.28	79.53
VW32-N:M	89.52	81.48	79.66
VW64	88.56	80.07	79.49
VW64-N:M	89.09	81.22	78.18
BW4x4	87.77	79.91	79.76
BW4x4-N:M	89.58	81.80	79.72
BW64x64	87.46	80.06	79.85
BW64x64-N:M	87.75	79.82	79.96

Table 3. Comparing F1 scores between with and without N:M distribution for EW, VW, and BW sparsity. Adding N:M distribution has no obvious or deterministic impact on model accuracy.

End-to-end speedup. Figure 12 shows the end-to-end comparison results on FP32 precision where all the systems except SparTA+cuSPARSELt use CUDA Cores for execution. TensorRT and Rammer achieved the same performance over each sparsity ratio because they treat these models as dense models for execution. *nmSPARSE* can achieve speedups over TensorRT and Rammer in each sparsity ratio. With the 50% sparsity ratio, SparTA+cuSPARSELt achieved the best performance due to the Sparse Tensor Core. However, it cannot support the 75% and the 90% sparsity ratio. *nmSPARSE* can outperform SparTA+cuSPARSELt with higher sparsity ratio. SparTA achieved the same performance as Rammer because its policy falls back to the dense execution. Thanks to the N:M sparsity, *nmSPARSE* can outperform SparTA and SparTA+Sputnik in each sparsity ratio. With the increase of the granularity, *nmSPARSE* can achieve higher speedups.

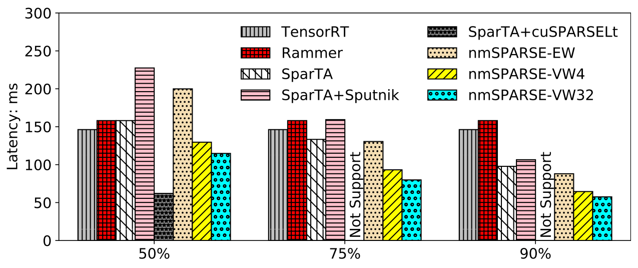


Figure 12. End-to-end speedup of *nmSPARSE* on Transformer at 50%, 75%, and 90% sparsity ratio.

6 Related Work

Pruning algorithm for N:M sparsity. Recently, N:M sparsity has received increased attention for its advantage in both high model accuracy and computational efficiency. (Mishra et al., 2021) follows the basic train, prune and fine-tune approach to generate 2:4 sparse models, and maintains accuracy over a wide range of models and tasks. Later, many pruning algorithms in the ML community have been proposed to enhance model accuracy for more general N:M sparsity. (Pool & Yu, 2021) introduce channel permutations to maximize the accuracy of N:M sparsity. (Sun et al., 2021) propose a layer-wise scheme for N:M sparsity to achieve higher accuracy than the uniform-sparsity scheme. (Oh et al., 2022) apply N:M sparisty to image restoration tasks and outperform previous pruning methods significantly. Although pruning algorithms for N:M sparsity are not the focus of our work, better pruning algorithms can achieve higher sparsity ratios, thereby amplifying the speedup of *nmSPARSE* kernels.

Execution hardware for N:M sparsity. To our best knowledge, hardware support for general N:M sparsity is limited. Nvidia Ampere architecture introduces Sparse Tensor Core (Mishra et al., 2021) to support 2:4 sparsity which is a particular form of general N:M sparsity. (Cao et al., 2019) propose a customized FPGA accelerator for N:M sparsity. (Yao et al., 2019) attempt to accelerate EW-N:M sparsity on GPU, but only achieve speedup on limited shapes.

7 Conclusion

This work presents *nmSPARSE*, a GPU library of SpMV and SpMM kernels for sparse DNN inference with general N:M sparsity patterns and various sparsity ratios. *nmSPARSE* addresses the longstanding challenges of irregular computation and scattered memory accesses in sparse matrix multiplications by leveraging the intrinsic balance characteristic of N:M sparsity. We hope *nmSPARSE* can not only benefit efficient DNN inference in the system community but also stimulate more research on enhancing the accuracy of N:M sparsity in the machine learning community.

References

Cao, S., Zhang, C., Yao, Z., Xiao, W., Nie, L., Zhan, D., Liu, Y., Wu, M., and Zhang, L. Efficient and effective sparse lstm on fpga with bank-balanced sparsity. In *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 63–72, 2019.

Chen, S., Wang, C., Chen, Z., Wu, Y., Liu, S., Chen, Z., Li, J., Kanda, N., Yoshioka, T., Xiao, X., et al. Wavlm: Large-scale self-supervised pre-training for full stack speech processing. *IEEE Journal of Selected Topics in Signal Processing*, 16(6):1505–1518, 2022.

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.

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, 2020a.

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, 2020b.

Gray, S., Radford, A., and Kingma, D. P. Gpu kernels for block-sparse weights. *arXiv preprint arXiv:1711.09224*, 3:2, 2017.

Han, S., Pool, J., Tran, J., and Dally, W. Learning both weights and connections for efficient neural network. *Advances in neural information processing systems*, 28, 2015.

Holmes, C., Zhang, M., He, Y., and Wu, B. Compressing pre-trained transformers via low-bit nxm sparsity for natural language understanding. *arXiv preprint arXiv:2206.15014*, 2022.

Liu, Z., Lin, Y., Cao, Y., Hu, H., Wei, Y., Zhang, Z., Lin, S., and Guo, B. Swin transformer: Hierarchical vision transformer using shifted windows. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pp. 10012–10022, 2021.

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. IEEE, 2022.

Ma, L., Xie, Z., Yang, Z., Xue, J., Miao, Y., Cui, W., Hu, W., Yang, F., Zhang, L., and Zhou, L. Rammer: Enabling holistic deep learning compiler optimizations with rtasks. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pp. 881–897. USENIX Association, November 2020. ISBN 978-1-939133-19-9. URL https://www.usenix.org/conference/osdi20/presentation/ma.

Mao, H., Han, S., Pool, J., Li, W., Liu, X., Wang, Y., and Dally, W. J. Exploring the granularity of sparsity in convolutional neural networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops*, pp. 13–20, 2017.

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.

Nvidia. Automatic sparsity (asp). URL https://github.com/NVIDIA/apex/tree/master/apex/contrib/sparsity.

Oh, J., Kim, H., Nah, S., Hong, C., Choi, J., and Lee, K. M. Attentive fine-grained structured sparsity for image restoration. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 17673–17682, 2022.

Pool, J. and Yu, C. Channel permutations for n: m sparsity. *Advances in Neural Information Processing Systems*, 34: 13316–13327, 2021.

Sun, W., Zhou, A., Stuijk, S., Wijnhoven, R., Nelson, A. O., Corporaal, H., et al. Dominosearch: Find layer-wise fine-grained n: M sparse schemes from dense neural networks. *Advances in Neural Information Processing Systems*, 34: 20721–20732, 2021.

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.

Wen, W., Wu, C., Wang, Y., Chen, Y., and Li, H. Learning structured sparsity in deep neural networks. *Advances in neural information processing systems*, 29, 2016.

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, S., Roller, S., Goyal, N., Artetxe, M., Chen, M., Chen, S., Dewan, C., Diab, M., Li, X., Lin, X. V., Mihaylov, T., Ott, M., Shleifer, S., Shuster, K., Simig, D., Koura, P. S., Sridhar, A., Wang, T., and Zettlemoyer, L. Opt: Open pre-trained transformer language models, 2022a. URL https://arxiv.org/abs/2205.01068.

Zhang, Y., Lin, M., Lin, Z., Luo, Y., Li, K., Chao, F., Wu, Y., and Ji, R. Learning best combination for efficient n: M sparsity. *arXiv preprint arXiv:2206.06662*, 2022b.

Zheng, N., Lin, B., Zhang, Q., Ma, L., Yang, Y., Yang, F., Wang, Y., Yang, M., and Zhou, L. Sparta: Deep-learning model sparsity via tensor-with-sparsity-attribute. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pp. 213–232, Carlsbad, CA, July 2022. USENIX Association. ISBN 978-1-939133-28-1. URL https://www.usenix.org/conference/osdi22/presentation/zheng-ningxin.

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. *arXiv preprint arXiv:2102.04010*, 2021.

A Operator shapes in our benchmark

	M1	M2	M3	M4	M5	M6	M7	M8
m	1	1	1	1	1	1	1	1
k	1024	2048	4096	8192	1024	4096	5120	20480
n	1024	2048	4096	8192	4096	1024	20480	5120
	M9	M10	M11	M12	M13	M14	M15	M16
m	256	1024	4096	256	1024	4096	256	1024
k	1024	1024	1024	2048	2048	2048	4096	4096
n	1024	1024	1024	2048	2048	2048	4096	4096
	M17	M18	M19	M20	M21	M22	M23	M24
m	4096	256	1024	4096	256	1024	4096	256
k	4096	8192	8192	8192	1024	1024	1024	4096
n	4096	8192	8192	8192	4096	4096	4096	1024
	M25	M26	M27	M28	M29	M30	M31	M32
m	1024	4096	256	1024	4096	256	1024	4096
k	4096	4096	5120	5120	5120	20480	20480	20480
n	1024	1024	20480	20480	20480	5120	5120	5120

Table 4. Operator shapes in our benchmark.

B Artifact Appendix

B.1 Abstract

This artifact contains the source code of nmSPARSE, a library of efficient GPU kernels for N:M sparse weights in deep learning, along with the docker file and running scripts to reproduce the main evaluation results presented in Figure 9, Figure 10, Figure 11 and Figure 12.

B.2 Artifact check-list (meta-information)

- **Algorithm: Sparse matrix-vector multiplication (SpMV) and sparse matrix-matrix multiplication (SpMM)**
- **Compilation: NVCC**
- **Hardware: We use Azure NC24ads_A100_v4 VM equipped with an NVIDIA A100-PCIE-80GB GPU**
- **Execution: Linux shell scripts**
- **Metrics: Latency**
- **Output: Execution latency**
- **How much time is needed to prepare workflow (approximately)?: About 1 hour to build the docker image**
- **How much time is needed to complete experiments (approximately)?: About 10 hours to finish experiments**
- **Publicly available?: Yes**
- **Code licenses?: MIT license**
- **Workflow framework used?: nnfusion(https://github.com/microsoft/nnfusion), SparTA(https://github.com/microsoft/SparTA)**

B.3 Description

B.3.1 How delivered

The artifact is hosted at https://github.com/microsoft/SparTA/tree/nmsparse_artifact.
To get the code, please git clone the SparTA repository and checkout to the *nmsparse_artifact* branch.

B.3.2 Hardware dependencies

All experiments are performed on a single NVIDIA A100 GPU.

B.3.3 Software dependencies

Please use docker to build *image/Dockefile* and run a docker container to set up the required environment.

B.4 Installation

To set up the environment, please first clone the code and build the docker image based on the Dockerfile we provided. Listing 1 shows the commands to set up the experiment environment.

Listing 1. Commands to set up the environment

```
# get the Dockerfile
git clone -b nmsparse_artifact https://github.
    com/microsoft/SparTA.git
# build the docker image and start a container
cd SparTA/image
sudo docker build . -t artifact
sudo docker run -it --gpus all --shm-size 16G
    artifact
```

B.5 Experiment workflow

We provide scripts to run experiments in Figure 9, 10, 11 and 12 respectively. Listing 2 shows the commands to run experiments.

Listing 2. Commands to run experiments

```
# get source codes and scripts in the docker
    container
mkdir workspace && cd workspace
git clone -b nmsparse_artifact https://github.
    com/microsoft/SparTA.git
conda activate artifact
# navigate to src directory
cd ./SparTA/src
# run SpMV experiment in Figure9
cd Figure9
bash run_baseline.sh
bash run_nmsparse.sh
# run SpMM on CudaCore experiment in Figure10
cd Figure10
bash run_baseline.sh
bash run_nmsparse.sh
# run SpMM on TensorCore experiment in Figure11
cd Figure11
bash run_baseline.sh
bash run_nmsparse.sh
# run end2end experiment in Figure12
cd Figure12
bash run.sh
```

B.6 Evaluation and expected result

Once all scrips finished running, results will be exported to *xxx_results.txt* in each folder. Execution latency of various kernels with different shapes and sparsity ratios can be obtained.