VENOM: A Vectorized N:M Format for Unleashing the Power of Sparse Tensor Cores

Roberto L. Castro*
roberto.lopez.castro@udc.es
CITIC
Universidade da Coruña
A Coruña, Spain

Andrei Ivanov
anivanov@inf.ethz.ch
Department of Computer Science
ETH Zürich
Zürich, Switzerland

Diego Andrade
diego.andrade@udc.es
CITIC
Universidade da Coruña
A Coruña, Spain

Tal Ben-Nun
talbn@inf.ethz.ch
Department of Computer Science
ETH Zürich
Zürich, Switzerland

Basilio B. Fraguela
basilio.fraguela@udc.es
CITIC
Universidade da Coruña
A Coruña, Spain

Torsten Hoefler
htor@inf.ethz.ch
Department of Computer Science
ETH Zürich
Zürich, Switzerland

ABSTRACT

The increasing success and scaling of Deep Learning models demands higher computational efficiency and power. Sparsification can lead to both smaller models as well as higher compute efficiency, and accelerated hardware is becoming available. However, exploiting it efficiently requires kernel implementations, pruning algorithms, and storage formats, to utilize hardware support of specialized sparse vector units. An example of those are the NVIDIA's Sparse Tensor Cores (SPTCs), which promise a 2× speedup. However, SPTCs only support the 2:4 format, limiting achievable sparsity ratios to 50%. We present the V:N:M format, which enables the execution of arbitrary N:M ratios on SPTCs. To efficiently exploit the resulting format, we propose *Spatha*, a high-performance sparse-library for DL routines. We show that Spatha achieves up to 37× speedup over cuBLAS. We also demonstrate a second-order pruning technique that enables sparsification to high sparsity ratios with V:N:M and little to no loss in accuracy in modern transformers.

KEYWORDS

Neural Networks, Pruning, GPGPU, CUDA, Sparse Tensor Cores

ACM Reference Format:
Roberto L. Castro, Andrei Ivanov, Diego Andrade, Tal Ben-Nun, Basilio B. Fraguela, and Torsten Hoefler. 2023. VENOM: A Vectorized N:M Format for Unleashing the Power of Sparse Tensor Cores. In *The International Conference for High Performance Computing, Networking, Storage and Analysis (SC '23), November 12–17, 2023, Denver, CO, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3581784.3607087

1 INTRODUCTION

The rapid progress of Deep Learning (DL) is revolutionizing Artificial Intelligence (AI) in areas such as Natural Language Processing (NLP). Large Language Models (LLMs) are at the forefront of modern NLP systems [5, 34]; however, their massive growth has led to unprecedented computational requirements [1, 12, 16, 30]. As a result, training transformers has become a dominant task in DL, with costs reaching millions of dollars and significant energy and carbon emissions [32]. Therefore, it is critical to improve their inference and training performance. One of the most widely used techniques for this purpose is network pruning [13], which removes the less significant weights to produce simpler and compressed, yet accurate models.

There is a plethora of pruning algorithms and sparse formats focused on accelerating tensor operations such as matrix-matrix multiplications (MMMs) by means of specialized hardware like Tensor Core Units (TCUs) [37]. While these algorithms and formats reduce the number of arithmetic operations and memory usage compared to their dense counterparts, achieving significant speedup on these accelerators while maintaining model accuracy is challenging [15]. Semi-structured pruning can yield practical speedups at moderate sparsity levels (e.g., $80 - 90\%$) [2, 3, 23]. However, the irregularity of the sparse input matrices still limits performance and makes difficult to reach the theoretical peak considering the reduction of the number of arithmetic operations [9].

Last generations of NVIDIA GPUs include Sparse Tensor Cores (SPTCs) that are specifically designed for sparse computation [25]. SPTCs promise to accelerate math operations by up to 2× at 50% sparsity. The data layout proposed to use SPTCs imposes strict constraints (i.e., 2:4 format, where every consecutive 4 elements have 2 nonzero values), but it reduces the irregularity of the sparse input w.r.t. other performance-aware sparse formats (e.g., vector-wise, block-wise). This makes the N:M format very suitable to execute on GPUs since it favors key aspects of the execution of tensor operations such as inter- and intra-warp load balance. However, there is an important limitation related to the usage of SPTCs and the 2:4 format: recent models like LLMs commonly have hundreds of millions to trillions of parameters, making it feasible to prune them to higher sparsity ratios with little or no loss in accuracy [19]. Unfortunately, there is currently no hardware support for executing arbitrary N:M formats with higher compression ratios, which limits the total achievable speedup.

*Corresponding author: Roberto L. Castro (roberto.lopez.castro@udc.es), Universidade da Coruña, CITIC, Computer Architecture Group, 15071 A Coruña, Spain

SC '23, November 12–17, 2023, Denver, CO, USA

ACM ISBN 979-8-4007-0109-2/23/11…$15.00
https://doi.org/10.1145/3581784.3607087

Recent research has explored the N:M format [4, 6]. However, these investigations have been limited to a theoretical perspective, such as network pruning, or have relied on CPU implementations due to a lack of hardware support for alternative N:M patterns on GPUs. To address these limitations, we propose the Vectorized N:M format, which we refer to as V:N:M[1]. This format introduces an abstraction layer over SPTCs, enabling the execution of alternative N:M formats and arbitrary sparsity ratios. The vectorization aspect is derived from the selection of vertical vectors of elements that are stacked together to provide the row-wise N:M pattern. This approach enables the conversion from generic N:M formats to the 2:4 that is accepted by SPTCs. To efficiently exploit the benefits of the V:N:M format, we propose Spatha[2], a template-based library dedicated to general matrix-matrix multiplication on half precision where one of the operands is sparse (SpMM). Spatha serves as an open-source alternative to cuSparseLt [26] and removes its 2:4 restriction. The main contributions of this paper are:

- A new sparse matrix format V:N:M which enables arbitrary N:M patterns on SPTCs.
- Highly optimized SpMM kernels to efficiently exploit the V:N:M format. Specifically, we propose a template-based implementation that can be tuned depending on the input dynamics, such as GEMM size or the V:N:M format configuration.
- A second-order pruning technique tailored for the V:N:M format and scalable to the dimensionality of LLMs. This technique allows the sparsification to high sparsity ratios with little to no loss in accuracy (e.g., $\sim$ 2% drop in BERT F1 score on the SQuAD dataset with 2:16 sparsity), which is required for the full exploitation of the V:N:M format.
- Spatha achieves unprecedented speedups w.r.t. its dense counterpart versions (e.g., cuBLAS) yielding up to 37× faster MMMs on matrices extracted from real-world DL models. Furthermore, Spatha implementation provides speedups of up to 1.38× over the vendor library for 2:4 sparsity, cuSparseLt.
- For end-to-end sparse LLMs inference, Spatha shows a GEMM time reduction of 11× at 2:32 sparsity on real-world models such as GPT-3.

2 BACKGROUND

This section presents the technical background of the paper, covering network pruning techniques and the Sparse Tensor Cores of NVIDIA GPUs.

2.1 Network pruning

In DL, pruning is a technique used to reduce memory usage, which can also reduce the computational load when combined with compressed storage formats and efficient sparse kernels. Pruning techniques can be categorized based on various criteria, such as the pruning strategy employed, or the granularity of the pruning.

Pruning schemes are often based on weight saliency metrics, which directly correlate with the expected impact on accuracy when those weights are removed from the network. Various methods exist to select the candidate weights for removal, including magnitude pruning [18], which selects weights with lower absolute values, and gradient-based methods that use the gradient applied to each weight to identify those that are trending towards to zero faster. Within the gradient-based methods, we can find first-order techniques based on the first-derivative information [31, 38], and second-order ones [7, 19, 21], which pursue to find the set of weights whose removal will generate a minimum loss increase in the network. Second-order methods have proven to be effective in pruning convolutional networks in the past, but they have recently been optimized for Large Language Models (LLMs) [19].

As for the granularity of the pruning, unstructured methods [11] remove weights individually, with gradual magnitude pruning (GMP) being the most commonly used variant [8]. On the other end of the granularity spectrum, structured methods [24, 35] prune complete components like layers, or heads, in the case of transformers networks [36]. In between, semi-structured methods prune groups of weights. These latter methods aim to balance performance and accuracy by defining specific formats that promote the exploitation of the underlying hardware more efficiently. These methods often imply the usage of tailored compressed storage formats and custom kernels [9, 20]. The N:M format, which enables the use of Sparse Tensor Cores (SPTCs) in NVIDIA GPUs, can be classified in this last group.

2.2 Sparse Tensor Cores of NVIDIA GPUs

The CUDA programming model organizes GPU kernels into three granularity levels: thread-blocks, warps, and threads. A thread block is composed of a set of warps, with warps being the basic scheduling unit in CUDA. Each warp consists of 32 threads.

NVIDIA GPUs consist of an array of Streaming Multiprocessors (SMs), with all SMs sharing the L2 cache, and a DRAM memory, also called Global Memory (GMEM). Each SM is divided in processing blocks, each one having a Register File (RF), a warp scheduler, and an L0 instruction cache. All the processing blocks within an SM share a L1 cache, which is partially used as Shared Memory (SMEM). Each processing block is also equipped with four types of units: Floating-Point Units (FPU), Tensor-Core Units (TCU), Int Units (ALU) and Special Function Units (SFU).

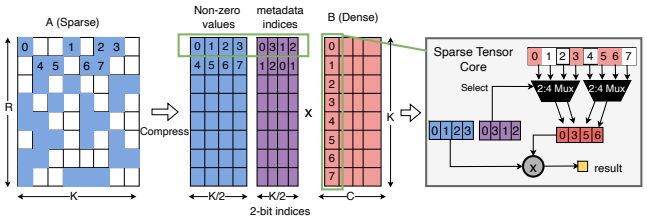

Figure 1: The 2:4 format and its mapping to SPTCs

Last generations of NVIDIA GPUs have extended their TCUs to also handle row-wise 2:4 sparsity. These updated TCUs include hardware support for sparse computation, and are referred to as Sparse Tensor Cores (SPTCs). To exploit SPTCs, the first argument in tensor operations must be stored in NVIDIA's N:M sparse format, where N represents the maximum number of non-zero elements in

[1]Pronounced "venom"

[2]SParse linear Algebra rouTines for High-performance Applications. The name is motivated by the analogy with the Cutlass library, with the accent on sparse computation - a sharp and efficient tool to cut through the complexity of sparse routines

a block of M values. Figure 1 illustrates this format. The left side of the figure shows an uncompressed sparse matrix following the row-wise 2:4 pattern. The compression of that $R \times K$ matrix requires two structures: (1) a $R \times K/2$ matrix representing the values of the non-zero elements, and (2) a metadata structure which contains the position of each nonzero value within each group of 4 values. Finally, Figure 1, right side, illustrates the mapping of a 2:4 sparse operation onto SPTCs. Notice that the metadata structure is also used by the hardware to select the corresponding elements in the dense matrix B and perform the Matrix Multiply-Accumulate (MMA) operation.

Precision	Format	Supported shapes
fp32	1:2	$k8, k16$
half (fp16)	***2:4***	k32, k16
uint8	2:4	$k32, k64$
uint4	2:4	$k64, k128$

Table 1: Matrix Shapes for *mma.sp* on SPTCs. M and N dimensions are fixed to 16 and 8, respectively ($m16n8$)

SPTCs can be accessed in CUDA using the NVPTX API which includes the *mma.sp* instruction. SPTCs support various shapes of this instruction depending on the data precision (Table 1). This instruction multiplies a $m \times k$ matrix by a $k \times n$ matrix, where $m = 16$, $n = 8$ are fixed dimensions, and k represents the sparsified dimension which can vary in size. This paper focuses on half precision kernels. Instruction shapes define the sizes of the left-hand-side (LHS) and the right-hand-side (RHS) operands as inputs to TCUs. For example, $k = 32$ implies that the LHS operand has a shape of $m \times k = 16 \times 32$ while the RHS is $k \times n = 32 \times 8$. It is important to note that the LHS is 50% sparse, meaning that its real size will be $16 \times \mathbf{16}(32/2)$. NVIDIA's notation for this instruction is $m16n8k32$.

3 THE V:N:M FORMAT

This section presents the new V:N:M format, which enables pruning to arbitrary N:M ratios retaining the use of SPTCs, which are designed to support only 2:4 patterns natively.

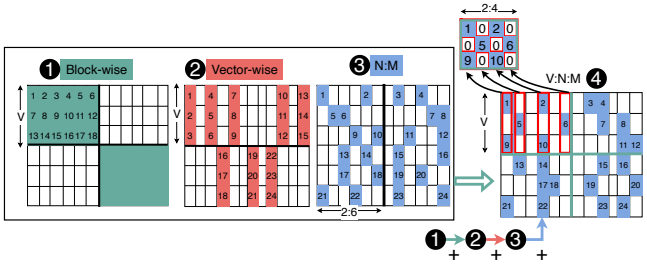

Figure 2: The V:N:M pruning procedure

Sparse compression formats are of great significance in many HPC areas other than DL. However, the characteristics of the sparse matrices in DL workloads differ from those in other areas in several aspects [9]: (1) the sparsity level is generally much lower, (2) the number of non-zeros per row is higher and (3) the load imbalance is more pronounced. To address these challenges, ad-hoc solutions for DL workloads have been developed in two different planes: compression formats and pruning techniques, often interlinked. They seek the efficient exploitation of the hardware during the execution of tensor operations in DL workloads.

A new area of research is focused on enhancing control over the distribution of non-zero elements in sparse matrices. This involves, for example, selecting 2D dense groups with size $v \times v$ (Figure 2, ❶) or 1D groups of length v, either row-wise or column-wise ❷. The aim is to create sparse matrices that are more regular, making them more suitable for efficient execution on GPUs. Block-based pruning techniques (❶ and ❷) are particularly useful on improving data reuse on L1 cache or registers during the multiplication of sparse matrices. Furthermore, optimized sparse formats, which compress their data, can be designed to facilitate traversal for the access patterns that arise during matrix multiplication [23, 28].

On the one hand, ❶ can be overly aggressive in dropping blocks of elements, leading to a significant reduction in accuracy as the sparsity level increases. On the other hand, ❷ offers more flexibility and enables higher sparsification ratios. However, using small vector lengths is a limiting factor to prevent accuracy loss (e.g., $v \leq 8$). Furthermore, in these approaches, the different number of elements per row can generate load imbalance and inherent negative effects such as thread divergence, inefficient memory transactions and low occupancy ratios.

The N:M format ❸ provides an alternative that overcomes most of the weaknesses of other performance-aware methods. Moreover, NVIDIA GPUs recently included hardware support for this format, but it is limited to 2:4. This paper introduces the new V:N:M format ❹ which combines block-wise storage, and vector-wise and N:M pruning to enable the exploitation of SPTCs for arbitrary N:M patterns, leveraging higher compression ratios and reducing further the number of arithmetic operations required in MMMs.

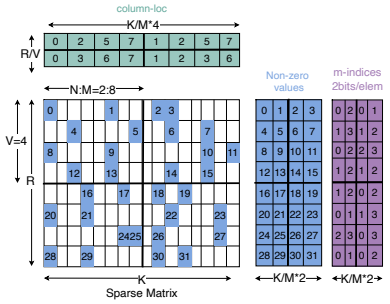

Figure 3: The V:N:M compression format

Figure 2, illustrates how this approach starts by partitioning the original dense matrix in blocks of $V \times M$ elements (block-wise). Then, the four most significant columns of each block are selected (vector-wise pruning), and for each row of four elements in a block, the two most meaningful weights are kept (2:4 pruning). These two levels of pruning (vector-wise and N:M) enable the exploitation of SPTCs for matrices with arbitrary levels of sparsity, as the vector-wise pruning stage diversifies the sparsity level, and N:M pruning imposes the restrictions required later by SPTCs. That is, in ❹,

the SPTC vector is 2:4, but it belongs to a 6-columns row, where 2 columns were fully pruned. It is actually an implementation of a 2:6 sparsity pattern that it is mapped onto SPTCs as the required 2:4.

Finally, the data is represented using a new block-wise compression format shown in Figure 3. As for the NVIDIA 2:4 layout (Figure 1), the format requires an array with the non-zero values, and a 2-bit metadata index per non-zero (*m-indices*). Notice that now, each 2-bit metadata index refers to one of the 4 columns that we have selected in each block and not to each column of the original dense input matrix (see ❹ in Figure 2). Furthermore, the size of these two structures depends on the M value, more specifically their shape now is $R \times K/M \times 2$. This format requires a third structure *column-loc* of size $R/V \times K/M \times 4$, that indicates which 4 columns (out of M) of each block were selected in the vector-wise pruning stage. Thus, while CSR storage overhead can reach up to 200% [25], V:N:M can be represented with 2-bit *m-indices* metadata per value, and up to 8-bit *column-loc* metadata per value every V rows. For $V = 128$, the overhead is $\sim 13.28\%$.

4 SPATHA: A HIGH-PERFORMANCE SPARSE LIBRARY FOR SPARSE MMM

This section provides an in-depth description of the sparse kernel implementation associated to the V:N:M format, Spatha. The Sparse Matrix-Matrix multiplication (SpMM) is an important workload in DL that serves as the sparse counterpart to Matrix-Matrix Multiplication (MMM). This routine is widely used in various components of modern DL models. For instance, in the forward pass of a pruned model, the sparse weight matrix is multiplied by a dense activation matrix. Similarly, in transformers, the self-attention operation is performed by multiplying a sparse attention weight matrix by a dense one. Thus, optimizing this routine is crucial to improve the efficiency and the performance of our models.

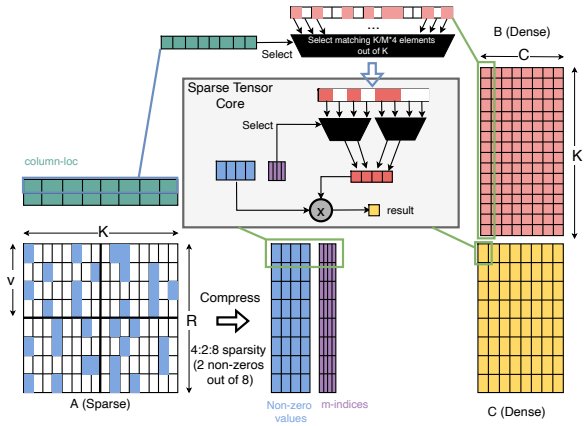

Figure 4: Mapping a 4:2:8 format onto Sparse Tensor Core (only native support to 2:4 format)

Figure 4 shows an example of how the new V:N:M format (4:2:8 in the figure) is mapped onto SPTCs, which natively only support the 2:4 format. It shows how the SPTC is fed with the appropriate values from a row of the sparse matrix and a column of the dense matrix. The LHS operand is a $R \times K/4$ dense matrix after having been pruned with sparsity of 75% (2:8). This pruning reduces the required multiply-and-add operations by 4 (from 16 to 4), but also halves the rows loaded from the dense matrix B (selected by the values contained in *column-loc*).

4.1 Kernel design

The design of an efficient CUDA kernel mostly depends on **three main stages**: (1) the efficient loading of the data to the top levels of the memory hierarchy (i.e., GMEM->SMEM->RF), (2) the computation, and (3) the storage of the results (i.e. RF->SMEM->GMEM). Figure 5 covers **stage 1**, particularly the data movement from GMEM to RF, which is divided into 3 steps (❶$_1$-❶$_3$). Figure 6 focuses on **stage 2**, and shows how the data in the RF is mapped onto SPTCs in three steps (❷$_1$-❷$_3$). Finally, Figure 8 illustrates how **stage 3** is performed (steps ❸$_1$-❸$_2$).

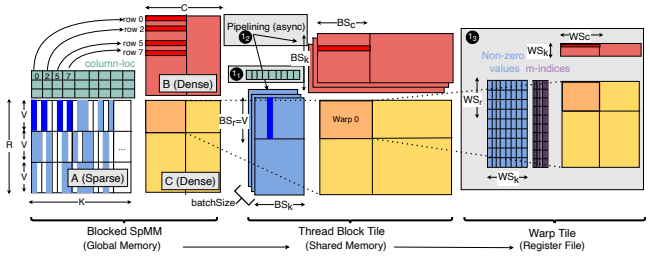

Figure 5: Thread-Block Tile and Warp tile view (stage 1)

Spatha is designed as a template-based library, where several parameters can be tuned depending on the input properties. Considering a $R \times K \times C$ GEMM problem, these parameters are: the thread-block tile size ($BS_r \times BS_k \times BS_c$), the warp tile size ($WS_r \times WS_k \times WS_c$), the mma instruction shape ($MMA_r \times MMA_k \times MMA_c$) and the level of memory pipelining (*batchSize*).

4.1.1 Stage 1-Data loading. Figure 5 shows the Spatha procedure to load the operands from GMEM onto RF. There are two dimensions to be taken into account: the data location (i.e., GMEM, SMEM, and RF), and the scope of this data from the NVIDIA programming model perspective (i.e., thread-block, and warp). Step ❶$_1$ loads the *column-loc* structure from GMEM to SMEM with a two-level prefetching strategy. Note that the *column-loc* information is used to select the rows of B to be loaded from GMEM (Figure 5, left side) to SMEM (step ❶$_2$). Pre-fetching this information breaks the data dependency with the activation matrix. Furthermore, *column-loc* is small, so it is convenient to load the information of multiple tiles together to maximize memory bandwidth. Next, step ❶$_2$ loads the corresponding A and B tiles from GMEM to SMEM. Each thread-block is responsible for an output block of size $BS_r \times BS_c$. More specifically, $BS_r = V$, so each thread-block will load **only** the rows of B selected by the *column-loc* structure. In order to avoid memory stalls due to data dependencies with the next steps, we pipelined step ❶$_2$ with step ❶$_3$ and stage ❷ (computation) taking advantage of CUDA asynchronous copies. The pipelining degree depends on the *batchSize* variable previously mentioned. Finally, in ❶$_3$, each warp is responsible for an output block of size $WS_r \times WS_c$, so

the corresponding tiles are loaded from SMEM to RF. Emphasize that all the previously mentioned memory transactions have been optimized to use 128-bit instructions. At this point, we also load directly to the RF the *m-indices* information.

4.1.2 Stage 2-Computation. When all the data is loaded in the RF, stage ❷ starts, which performs the Matrix Multiply-Accumulate (*mma.sp*) on this data using SPTCs. Figure 6 shows a detailed view of stage ❷, depicting how the data in the RF is mapped onto SPTCs to be executed. Each warp has to break down the warp tile into instruction tiles, which depends on the instruction shapes available on SPTCs, in this example *m16n8k32*. The first step $❷_1$, selects $MMA_k = 16$ elements from the warp tile and maps this data to SPTCs following step $❷_2$ layout. This layout represents the LHS fragment to the *mma.sp* instruction. That means that, if $WS_r = 32$, we will need to iterate twice over the rows of A's warp tile. Similarly, the next step maps the B's warp tile information into SPTCs following step $❷_3$ layout, which represents the RHS fragment to the *mma.sp* instruction. At this point, the *mma.sp* instruction is executed.

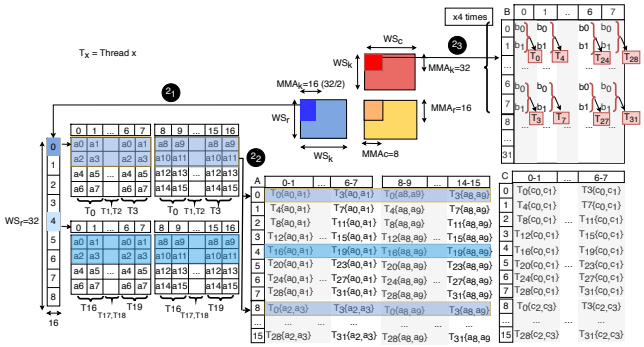

Figure 6: SPTCs view

Storage order. Related to stage ❶ and ❷, we propose a specific order to store the non-zero values and the *m-indices* structure of the V:N:M format, which merges, once again, the block-wise and the N:M principles. This order is represented in Figure 7, and it seeks to optimize the data traversal during the data loading and computation. In this representation, half of the non-zero structure shows the access pattern followed to store the data, while the other half shows how the second half-warp is mapped into this structure. This storage order enables 128-bit memory transactions, ensures memory coalescence, and can dispense with the *ldmatrix* instruction, which is known to cause bank conflicts and can require more SMEM transactions to sequentially serve the memory access [33].

4.1.3 Stage 3-Result storage. Once the product is calculated, we have to write the output tiles back to GMEM (stage ❸). This requires storing the intermediate partial results in SMEM. On NVIDIA GPUs, shared memory is partitioned into banks, each one of 32 bits. Each bank can only address one position at a time, so if a quarter-warp (128-bit instructions) tries to access the same bank, the instruction will be serialized. This effect is known as bank conflict. An example of thread mapping to SMEM with $BS_c = 64$ is shown in Figure 8.

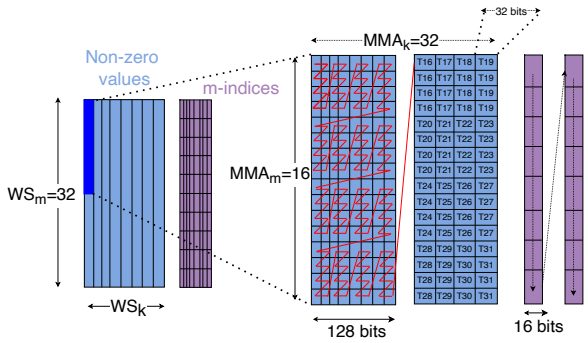

Figure 7: Storage order

The left side of the figure shows how the threads in a warp are mapped to SMEM banks during the storage of their partial results (step $❸_1$). These stores are performed with 128-bit instructions. Padding elements have been added to avoid bank conflicts. In this specific example, each thread has accumulated 8 partial results ($BS_c/MMA_c = 64/8$), so the thread mapping is repeated 8 times, meaning that each thread needs 8 iterations to store its partial results. Each color represents a quarter-warp, so we can see that each group of 8 consecutive threads accesses a different memory bank in the same iteration.

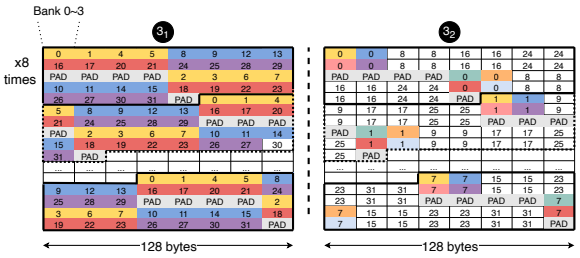

Figure 8: Conflict-free accesses for output tiles on SMEM

The right side of Figure 8 shows step $❸_2$, that is, the SMEM thread mapping designed to read the previously stored intermediate results, and finally, write them back to GMEM. The loads from SMEM and the stores to GMEM are performed with 128-bit instructions. Once again, each thread will need to access SMEM 8 times to read all the data. We have colored the accesses related to the first quarter-warp, what depicts a conflict-free layout.

Ablation study - Spatha performance and column-loc overhead. In Figure 9, we present the results of a microbenchmark study on matrices of fixed outer dimensions (corresponding to the size of one $BERT_{large}$ weight linear layer), but varying the inner (sparsified) one, K ($1024 \times K \times 4096$). The study was conducted using different sparsity levels, specified by different N:M combinations (from 2:10 to 2:100), while the vector size V was kept constant at 128. Furthermore, to measure the effect of using the *column-loc* mechanism, we tested the performance with and without this structure. In the latter we used fixed indexes to simulate an ideal situation with no memory accesses. These experiments are performed on an NVIDIA RTX 3090 GPU, equipped with SPTCs. The

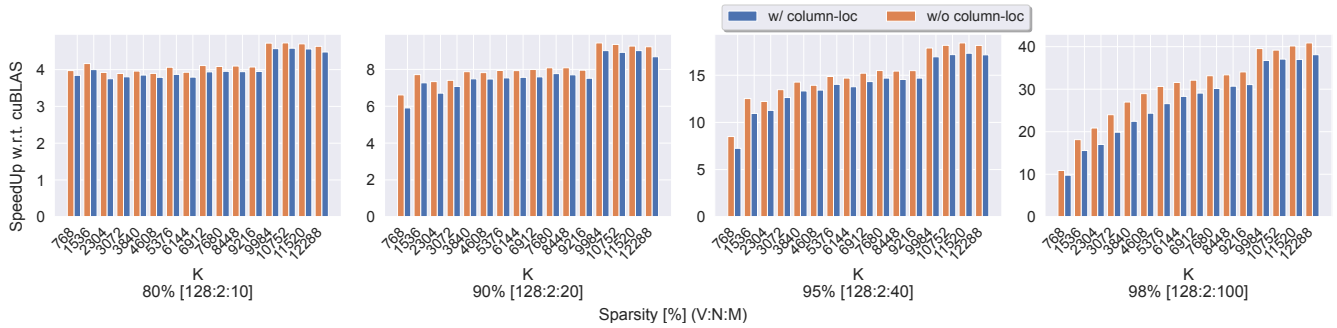

Figure 9: Ablation study of column-loc with different sizes of the inner K dimension and different V:N:M formats ($BERT_{large}$)

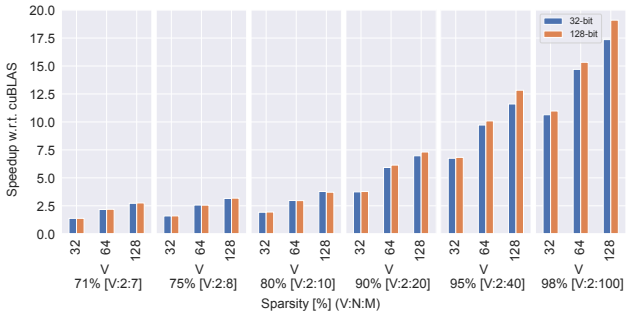

Figure 10: Scaling study of wide shared memory stores for different V:N:M configurations

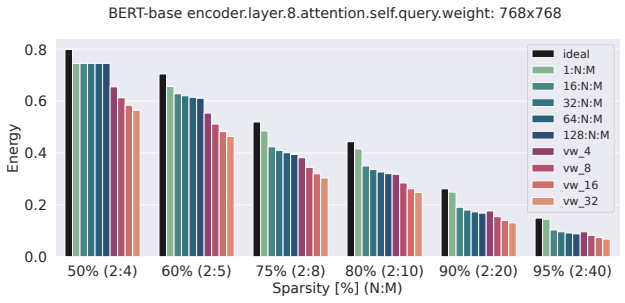

Figure 11: Energy evaluation study on the V:N:M format

results show that Spatha achieves speedups for sparse computation, approaching theoretical peak performance for a given sparsity level considering the operation count reduction w.r.t. the dense counterpart version. This effect becomes more pronounced as the GEMM problem size increases, as it tend to have higher arithmetic intensity. For instance, at a sparsity level of 80% (2:10 format), the speedup is approximately 4.5×, where 5× is the ideal scenario. Then, the speedups reported are 8.5×, 17.5×, and 37× for sparsity levels of 90% (2:20), 95% (2:40) and 98% (2:100), whose theoretical caps are 10×, 20× and 50×, respectively. It can be observed that, for every sparsity ratio, the *column-loc* structure's overhead has a negligible effect on the overall time, despite being a software approach to support arbitrary N:M ratios. However, the impact of *column-loc* becomes slightly more noticeable when dealing with 2:100 sparsity, which is not practical for DL applications in real-world scenarios.

Scaling study - Impact of V and output layout format. The V variable in our V:N:M format can be used to define trade-offs between performance and accuracy in the same way that the block-size in block-wise pruning, for example. To study this, we performed a second ablation study on one matrix from $BERT_{large}$ (size $1024 \times 4096 \times 4096$). Figure 10 shows the performance results of Spatha on this matrix using three different vector lengths: 32, 64 and 128. This test is conducted for different sparsity levels, in practice, the test explores different configurations of the V:N:M values. Furthermore, in order to study the impact of the previously proposed layout for writing back results (Figure 8), it is compared the effect of using such layout, enabling 128-bit SMEM stores instead of 32-bit ones. As we can see in Figure 10, the difference in terms of speedups between the three selected vector lengths is noticeable, the value of V being conditioned by the accuracy loss. The effect of using 128-bit stores instead of 32-bit ones is noticeable in this problem size, bringing up to a 2× difference in the final speedup. We performed a similar ablation test for a matrix of a GPT-3 model (size 36864×12288×4096) and the effect of using 128-bit stores was attenuated, as the weight of the output phase in the total execution time is smaller.

5 ENERGY EVALUATION OF V:N:M

DL pruning techniques aim to achieve the highest possible sparsity levels in the pruned models while ensuring little to no loss in accuracy. This becomes especially challenging when the target sparse format requires a specific pruning scheme, and when high sparsity levels are targeted. In these scenarios, the percentage of non-zero values is low, and their location is heavily influenced by the format. Therefore, it is crucial to demonstrate the effectiveness of new sparse formats, to ensure its applicability with minimal or no impact on accuracy.

The energy evaluation metric (magnitude preservation) [22, 29] is employed to measure the flexibility of a format by comparing the total magnitude of the model (sum of the individual weights) before and after pruning to a specific format. Let us assume a well-optimized dense model $w^* \in \mathbb{R}^d$, where d is the total number of weights. We wish to prune w^* to a target sparsity $s \in (0, 1]$ by

zeroing out $s \times d$ weights. The result is a sparse model $w \in \mathbb{R}^{s \times d}$. The energy metric is defined as follows:

$$energy = \frac{\sum_{i=0}^{s \times d} |w_i|}{\sum_{i=0}^{d} |w_i^*|}$$

This metric yields a normalized score between $0 \sim 1$, the higher the better.

Figure 11 presents the energy evaluation study for a weight tensor extracted from an encoder layer of BERT_{base}. This figure compares three weight selection policies: unstructured (ideal), V:N:M with different V values, and vector-wise pruning with several vector lengths l (vw_l). The evaluation is done for different sparsity levels, whose value in the V:N:M format is controlled by the N:M ratio.

Unstructured magnitude-pruning represents the ideal non-zero selection policy, as it does not impose any restrictions on the location of non-zeros. Vector-wise pruning can accelerate sparse routines on GPUs. However, if the vector length is > 8, it can significantly reduce the accuracy [2, 3, 23]. The results demonstrate that the V:N:M format occupies an intermediate position between unstructured and vector-wise pruning. Moreover, it is highly robust to changes in the vector length, allowing the usage of $V = 128$ while consistently preserving more energy than vw_8 and vw_4.

Additionally to the previous conclusions, independently of the selected pruning method, we can also see the tremendous impact on the energy of magnitude-based weight selection policies. At 50% of sparsity, unstructured pruning already lost 20% of the original dense matrix energy. At the other side, at 95% only 20% of the original energy remain in the pruned dense matrix. Thus, we can conclude that, in order to achieve moderate to high sparsity ratios in models with the dimensionality of BERT, more sophisticated pruning methods must be used. Second-order pruning offers an alternative to these problems.

6 SECOND-ORDER PRUNING

Magnitude-based pruning techniques provide a straightforward approach to reducing the size of our models without requiring model evaluation for weight selection. However, while magnitude pruning can be effective at moderate sparsity levels, it becomes more challenging to select the "least significant" weights to remove when aiming for high sparsity ratios, and this can significantly impact network accuracy.

In contrast, second-order pruning methods offer a more sophisticated approach to select weight candidates for removal, by considering the difference in loss relative to the current model. Hence, they target to find the set of weights whose removal will generate a minimum loss increase. In this context, the Hessian matrix is a key component of second-order pruning methods which represents the matrix of second-order derivatives of the loss function w.r.t. the weights, mathematically expressed as $H = \nabla_w^2 L$, for a twice-differentiable loss L. The Fisher matrix is very similar to the Hessian matrix but in the probabilistic setting, used to estimate the curvature of the loss function around the current value. As a result, this approximation allows to identify the weight parameters that have less impact in the loss function, and therefore are candidates to be pruned [13].

6.1 The V:N:M format in 2nd order methods

This section introduces a new second-order pruning method based on [19] and tailored for the V:N:M format. This type of approach yields state-of-the-art results in LLMs for unstructured and semi-structured (block) compression.

Let us assume we have a well-optimized dense model $w^* \in \mathbb{R}^d$, where d is the total number of weights. Our target is to identify a set of weights Q that we can prune with a minimum loss increase. The following saliency score function is defined to rank groups of weights [19]:

$$\rho_Q = \frac{1}{2}(E_Q w^*)^T (E_Q \widehat{F}^{-1}(w^*) E_Q^T)^{-1} E_Q w^*$$

where,

- $\widehat{F}^{-1}(w) \in \mathbb{R}^{d \times d}$ is the Fisher matrix.
- $E_Q \in \mathbb{R}^{|Q| \times d}$ is a matrix composed of the corresponding canonical basis vectors for a set of Q weights.

Thus, the set of canonical basis vectors E_Q depends on the specific sparse format we are using. For instance, in 2:4 sparsity, the canonical vectors are:

$$E_Q = [[1,1,0,0], [1,0,1,0], [1,0,0,1], [0,1,1,0], [0,1,0,1], [0,0,1,1]]$$

As observed, E_Q encompasses all possible correlations between 2 weights, in a set of 4 elements. In general, for an N:M format, this approach requires evaluating $\binom{M}{N}$ combinations to determine the best one, which can turn into an intractable combinatorial problem. Furthermore, in the V:N:M format, the addition of a new dimension V amplifies the complexity as it requires finding the optimal set of $V \times N$ weights, leading to a combinatorial explosion.

To address these challenges, we adopt a similar approach as [19] between sets of Q elements, which involves disregarding correlations between rows within $V \times M$ blocks. This simplification drops the number of combinations to evaluate. Additionally, to mitigate combinatorial issues that may still arise within $1 \times M$ groups, we propose a **pair-wise approach** where correlations are calculated between pairs of elements, that is:

$$E_Q = [[1, 0], [0, 1], [1, 1]]$$

The overhead of this pair-wise approach represents < 1% of the sparsification process. Then, depending on the N and M values, we can modulate the complexity of the problem to be solved by dynamically selecting the m-combinatorial or the pair-wise approach.

6.1.1 Gradual pruning definition. The N:M format prunes a model to a target sparsity $s \in (0, 1]$. Typically, the $s \times d$ weights are removed in one step (one-shot pruning). For 50% (2:4) sparsity, this approach can be applied in most cases and the models still recover the original accuracy. However, for higher sparsity ratios, one-shot pruning reduces severely the model performance and makes hard to recover the original accuracy using additional fine-tuning steps. This negative effect on accuracy also happens in second-order methods, where one-shot pruning can result in worse Taylor approximations of the function. We propose a structure decay scheduler for the V:N:M format, which performs N:M pruning across different β steps, for increasing sparsity levels. This scheduler starts with a high initial value of $N_0 >> N_\beta$ (lower sparsity), where N_β is our target N value, and gradually decreases N (conversely

increasing sparsity) until it reaches the N target value. This gradual pruning approach mitigates the adverse effects on network accuracy and improves accuracy recovery in subsequent fine-tuning stages.

7 EVALUATION

We evaluate the performance on an NVIDIA RTX 3090 GPU of the Ampere architecture equipped with SPTCs. We compare the performance of Spatha with different sparse libraries (cuSparseLt, cuSparse, CLASP, and Sputnik) and also with a dense counterpart version (cuBLAS). We build our benchmarks on matrices from the DLMC dataset [10], and real-world LLMs. Additionally to these micro benchmarks, we also conduct a case study on real-world applications. At this point, we demonstrate the proposed second-order pruning technique, and we benchmark the end-to-end performance of Spatha on different LLM models (BERT, GPT-2, and GPT-3).

7.1 Comparison with existing dense and sparse libraries

Firstly, we evaluate our baseline implementation for 1:2:4 sparsity (50%). Since higher N:M ratios will depend on this baseline's performance, it is crucial to have good speedup results in this configuration. We selected cuBLAS GEMM as our dense counterpart, and for exploiting the 2:4 format on SPTCs, we used the cuSparseLt SpMM implementation, which represents the reference library on this format. Our experiments involve varying sizes of a $R \times K \times C$ GEMM problem, where R and C are predetermined values from two BERT's weight linear layers (768 and 4096 for BERT$_{base}$, 1024 and 4096 for BERT$_{large}$). The inner dimension K of the product, which is the sparsified one, is variable in these experiments. Note that the inner dimension is usually scaled up to enhance the network accuracy. For instance, GPT-3 uses a hidden size of 12288 [1]. Figure 12 reports the performance of the three contending implementations (cuBLAS, cuSparseLt and Spatha) and the speedups of the selected sparse libraries w.r.t. cuBLAS. The results show that the performance of the sparse implementation improves with the GEMM size, as larger GEMMs tend to have larger arithmetic intensity. In these microbenchmarks, BERT$_{large}$ matrices (right side) increase the computation intensity w.r.t. BERT$_{base}$ (left). Notably, for larger GEMM sizes, the performance of cuSparseLt and Spatha is similar and approaches the peak for 2:4 sparsity, 2×. However, our implementation shows better performance on smaller sizes, which constitutes an interesting feature, since Spatha can probably cover a more variety of network architectures. Overall, Spatha achieves up to 1.38× speedup over the vendor library, cuSparseLt.

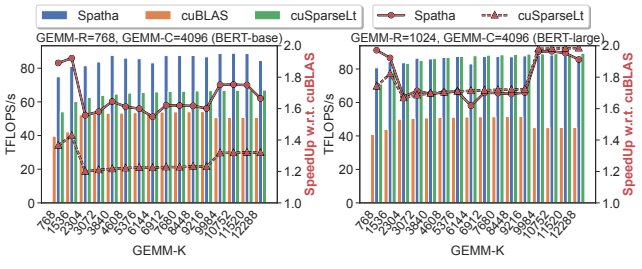

Figure 12: Baseline performance at 50% sparsity (2:4 format)

Figures 13 and 14 compare the performance of Spatha to other dense and sparse libraries for higher sparsity levels ranging from 50 ∼ 98%. In this context, cuSparseLt SpMM is the reference library to exploit the 2:4 format on SPTCs. However, cuSparseLt's results are *strictly limited* to 50% sparsity and cannot be executed for the entire sparsity range because it only supports 2:4 sparse matrices. Since there are no SpMM GPU implementations for arbitrary N:M sparsity levels, we have considered the following third-party libraries for half-precision: Sputnik [9], cuSparse [27], and CLASP [2] which extends vectorSparse [3] to the latest generations of NVIDIA GPU architectures. Concerning semi-structured sparse matrices, [2] focuses on the column-vector format, which supports vector lengths $l = 2, 4$ and 8. Additionally, cuSparse introduces the Blocked-ELL format to improve the performance of block-wise sparse matrices, for which we considered the same block lengths (2, 4, 8) as [2, 3]. For non-structured sparsity, [9] represents the SOTA implementation over cuSparse+CSR, excluded here for simplicity.

Figure 13 benchmarks are built using the DLMC dataset [10]. Spatha, CLASP, and cuSparse configurations are referenced in the rows with the notation V:N:M, vw_l, and bw_l, respectively. In the absence of sparse matrices compatible with the N:M format, we have generated the requisite matrices by pruning the original dense models [8]. Most of the dense matrices in this dataset are too small (e.g., 64×64, from ResNet-50) to fill the GPU resources. This translates into low arithmetic intensities, hindering peak performance. As Figure 13 shows, when we increase the arithmetic intensity by means of C (batch size), cuSparseLt approaches the 2× peak [25] (still 1.05× on average for $C = 2048$). However, Sputnik, cuSparse and CLASP performance shows important limitations. Existing SpMM kernels are mostly designed considering small models, where the LHS operand can be a tiny matrix. That influences the SpMM design, since data can be loaded directly into registers, but these design decisions represent a serious scalability problem. However, Spatha is able to scale with the problem size, reaching speedups of up to ∼ 25× (98% sparsity). The scalability property is critical with the increasing models size, especially on LLMs.

Figure 14 benchmarks are built using sparse matrices from weight-pruned linear layers extracted from BERT. Spatha, cuSparse, and CLASP configurations have been referenced in the columns of Figure 14 with the same notation as Figure 13. The first row of Figure 14 shows the speedup results on sparse matrices extracted from BERT$_{base}$ while the second one reports that performance on BERT$_{large}$. The y-axis is represented in a logarithmic scale to make the results more readable. As we saw in Figure 13, existing implementations outperform the dense counterpart version at sparsity levels above 80% when the matrices are small. However, when we evaluate these implementations on medium or big matrices extracted from larger models (e.g., LLMs), the performance is even worse, and they only outperform cuBLAS at sparsity levels above 90%. Thus, their kernel design presents scalability issues when the problem size grows. The fact that Spatha reaches 2× speedup at 50% sparsity enables the achievement of high speedups as the sparsity increases, yielding up to 27× in BERT-like matrices. We can also appreciate that the best performance in our implementation is reached with the augmentation of arithmetic intensity, peaking for BERT_*large* with batch size 16. A discussion of how these speedups affect the global LLMs latency will be presented next.

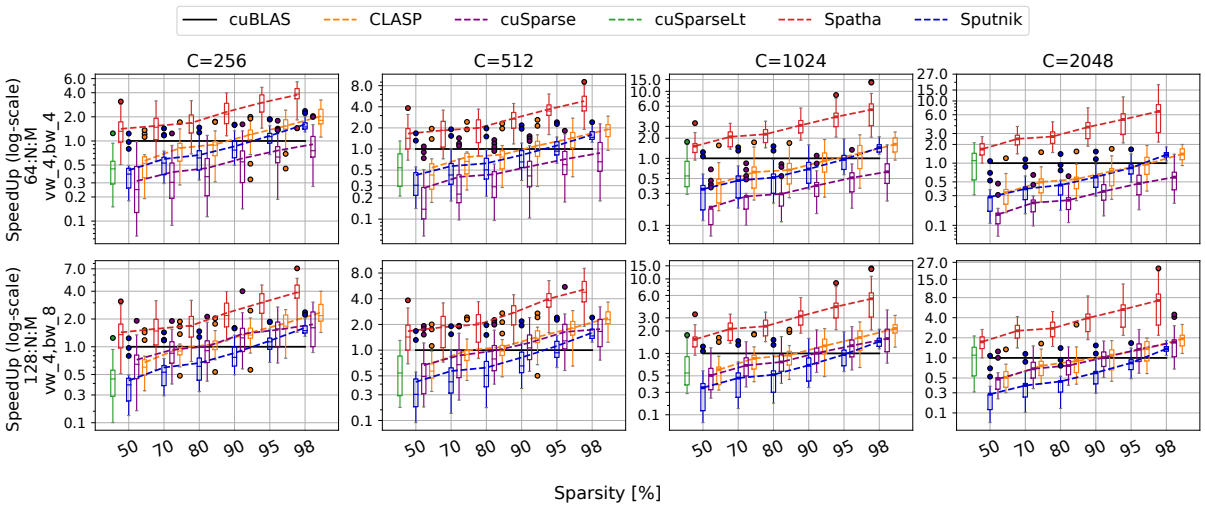

Figure 13: Speedup results on sparse matrices from the neural networks on the DLMC dataset. The problem size is $A_{R\times K} \times B_{K\times C}$ where A is the sparse matrix. The R and K sizes are given in the DLMC dataset, while C is selected from $\{256, 512, 1024, 2048\}$

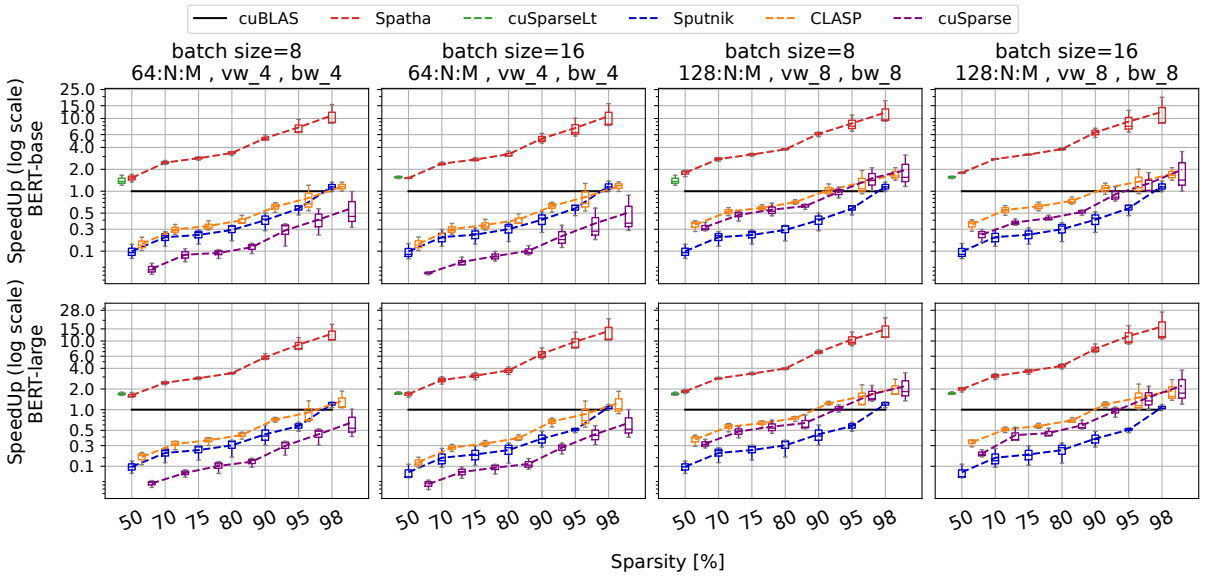

Figure 14: Speedup results on BERT$_{base}$ and BERT$_{large}$ with sequence length=512. The notation V:N:M represents the vector length V used on Spatha, while vw_l and bw_l represents the vector length l used on CLASP and cuSparse, respectively. The N:M pattern related to each of the considered sparsity levels are in ascending order of sparsity: 2:4, 2:7, 2:8, 2:10, 2:20, 2:40 and 2:100

7.2 Case study: sparse LLMs

LLMs have revolutioned the NLP field with their unrivaled performance in various domains. Nowadays, these models are widely used in everyday technologies, such as ChatGPT. Transformer LLMs typically consist of multiple transformer layers with self-attention.

There are two major sub-components inside a transformer architecture: the multi head attention (MHA), and the fully connected feed forward network (FFN). At a higher level, the model size is determined by different configurable components, such as the head dimension, the number of heads and the number of layers, depending on the specific architecture used.

This case study focuses on weight pruning, and explores the on computational speedups achievable with Spatha. In LLMs weight tensors are present in Linear Layers, which can be found in both the MHA and the MLP sub-components. Figure 15 illustrates a pruned MHA where four GEMM instructions are converted to SpMMs by sparsifying the corresponding weight tensors. In this study we demonstrate the efficiency of Spatha on different LLMs. However, it

is important to note that without an efficient implementations of the SpMM instruction, the final performance of the pruned model can significantly decrease compared to the dense counterpart version.

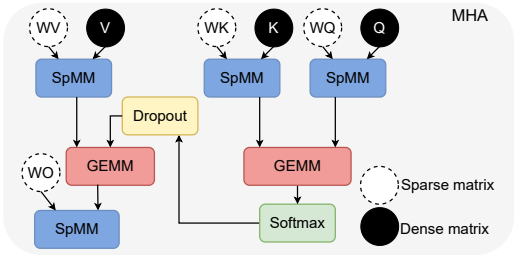

Figure 15: Simplified view of a pruned MHA

7.2.1 Second-order pruning at LLMs scale. We used our 2nd order pruning approach following the V:N:M format to demonstrate its applicability to the size of LLM models. Specifically, we focused on BERT$_{base}$, one of the most commonly used LLMs, which comprises 12 transformer layers with 110M parameters. As per community standards [19], we pruned the encoder's weights of the model (85M). We evaluate the performance on the SQuAD v.1.1 task, which is a widely-used benchmark to measure model compression. Table 2 shows the F1 score metric for different pruning techniques including: the original N:M one-shot pruning [19], vector-wise pruning with dense vertical vectors of size 8 (vw_8), and our gradual pruner for plain N:M (1:N:M), and V:N:M with V size of 64 and 128.

LLMs have been shown to be susceptible to minor model perturbations that can cause model collapse [17]. However, in these experiments we considered 75% and 87.5% sparsity levels, represented by 2:8 and 2:16 ratios, respectively, to demonstrate that our pruning approach produce robust results on this kind of networks.

Sparsity	N:M [19]	1:N:M	64:N:M	128:N:M	vw_8
75% (2:8)	88.22	88.61	88.47	87.94	88.55
87.5% (2:16)	85.95	87.73	86.50	85.01	86.90

Table 2: F1 score of BERT$_{base}$ on the SQuADv1.1. Dense model F1=88.43

As we can see, 1:N:M, 64:N:M and vw_8 slightly improve the original model accuracy at 2:8 sparsity, while the 128:N:M format presents a 0.005% accuracy loss. For 2:16 sparsity, these four methods suffer a slight accuracy loss. Specifically, the plain 1:2:16 format is able to recover 99% of the original accuracy, while 64:2:16 and vw_8 pruning recover 98%. In these terms, the 128:2:16 approach is slightly more restrictive but is still able to recover 96% of the original accuracy. Finally, regarding the N:M baselines, 1:N:M gradual pruning shows more robust to sparsity than one-shot N:M [19].

7.2.2 Integration with Pytorch. In order to perform the end-to-end evaluations, we have streamlined the adoption of Spatha into the PyTorch training pipeline by integrating it with the STen library [14]. This integration allows for easy addition of sparsity to existing models such as BERT and GPT with just a few lines of code. Users can specify a list of weights to be made sparse in their custom models, making the process straightforward. To facilitate this, we have defined a `VNMSparsifier` class that performs pruning while adhering to the V:N:M format constraints. Additionally, we have introduced a `VNMTensor` class that serves as a container for tensors in the V:N:M format. When using SpMM with `VNMTensor`, STen automatically dispatches it to the efficient implementation in Spatha. A pseudocode example of this integration is shown in Listing 1.

```
import sten
import spatha

@sten.register_sparsifier_implementation(
    sparsifier=spatha.VNMSparsifier,
    inp=torch.Tensor, out=spatha.VNMTensor)
def torch_tensor_to_vnm(sparsifier, tensor, grad_fmt):
    return sten.SparseTensorWrapper \
        .wrapped_from_dense(
            spatha.vnm_sparsifier(
                sparsifier.n, sparsifier.m,
                sparsifier.v, tensor),
            tensor, grad_fmt)

class Spmm(torch.nn.Module):
    def __init__(self, original: torch.nn.Linear):
        self.bias = original.bias
        w = original.weight.wrapped_tensor
        self.values   = w.values
        self.columns  = w.columns
        self.metadata = w.metadata
    def forward(self, input):
        return spatha.spmm(self.values, self.columns,
            self.metadata, input, self.bias, ...)
```

Listing 1: Pseudocode example of using Spatha and the V:N:M sparsifier

7.2.3 Sparse Inference. We benchmark the end-to-end performance of Spatha on the inference task for different real-world LLM models: BERT (336M), GPT2-large (774M), and GPT-3 (175B), obtained from HuggingFace. Since GPT-3 is not a public trained model, we have created a model with the same configuration than this LLM. The target of this experiment is measuring time performance, thus, we initialized the weights of the GPT-3 model with random values. The time results on BERT and GPT2-large have been obtained over the inference of the entire model, while the results of GPT-3 were obtained by measuring the inference time of a single encoder to fit it on a single GPU.

Figure 16 shows the end-to-end evaluation results on the inference of these models. As we have seen in the previous micro benchmark experiments, increase the arithmetic intensity of the MMMs improves the utilization of the GPU resources, and also the final performance of the SpMM. We configured the three models to the larger configuration possible before achieving out-of-memory issues. In the case of BERT$_{large}$, this implied the selection of a batch size (bs) of 32. For GPT2-large, the bs is 8, and in the case of GPT-3, it is 1. However, bs only affect the C dimension of the GEMM problem ($R \times K \times C$), while the two others, R and K, depend on the model characteristics. Regarding these sizes, BERT has smaller weight tensor sizes (the ones sparsified) than GPT2-large, while GPT-3 is formed by weight tensors much larger than the two other models.

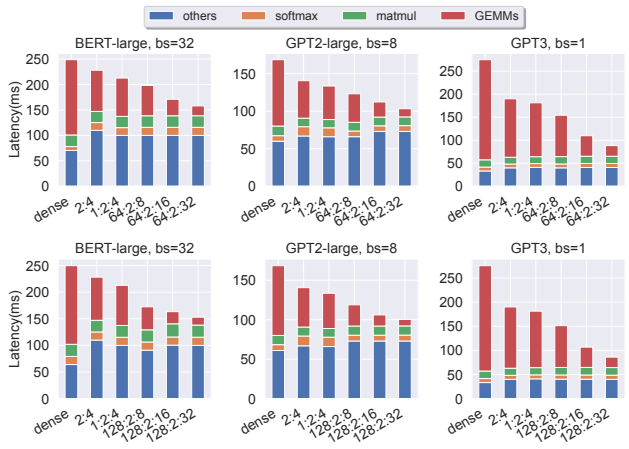

Figure 16: Latency of LLMs inference using cuBlas (dense), cuSparseLt (2:4), and Spatha (V:N:M, with V in $\{1, 64, 128\}$)

Due to the previously described reasons, we can see that the best performance is obtained in the case of GPT-3, where the GEMM computation contributes to around 80% of the total execution time.

In the case of BERT, tensor contraction time is improved up to 9.95×, while in terms of the whole model, the end-to-end latency is improved up to a 72%. For GPT2-large, the GEMM time is improved in 10.84×, since some weight tensors are slightly bigger, but the total GEMM time is around 50%, so the general improvement is limited by this factor. However, when we move to GPT-3, the tensor time contraction is improved up to 11×, but the GEMM time represents a much higher percentage, meaning a time reduction of up to 3.20× of the total execution time of a GPT-3 encoder.

8 RELATED WORK

Semi-structured pruning techniques are a hot research topic. The column-vector-sparse-encoding [3] seeks to accelerate sparse kernels, and it achieves a speedup between 1.71× and 7.19× over cuSPARSE without exploiting SPTCs, and limited to the Volta architecture. The same authors target the SPTCs in [4] proposing DFSS, a dynamic N:M sparse attention mechanism and a tailored implementation of the sparse kernels, but limited to the 2:4 format. The unaligned group-level pruning proposed in [22] increases the accuracy of this kind of semi-structured pruning techniques by providing additional flexibility.

*N*VIDIA cuSparse [27] is a library from NVIDIA that implements several linear algebra routines for sparse matrices stored in different compressed formats (COO, CSR and Blocked-Ellpack). It was originally created to target scientific applications. The *cu*SparseLt[26] library from NVIDIA adds support for the exploitation of Sparse-Tensor Cores (SPTCs) following the N:M format, and giving support to 1:2 ad 2:4 sparsity patterns (50% of sparsity).

*S*putnik [9] library has been specifically designed for DL workloads. It uses only the CSR compressed format, and it focuses on gaining flexibility on the scheduling of workloads by defining a one-dimensional tiling scheme. This library evolved to *V*ector-Sparse [3] adding support for the exploitation of Tensor-Core Units. It is based on using semi-structured 1D pruning, and a special compressed format called Column-Vector Sparse Encoding. As a continuation, *C*LASP [2] offers an SPMM implementation which extends the support of Vector-Sparse to the Ampere architecture.

In the same line, *M*agicube [23] is an implementation of the SPMM and SDDMM routines for quantized sparse matrices. The kernels are complemented with en efficient online method to transpose the dense matrix.

9 DISCUSSION

a) *Spatha application to other tasks.* The integration of the Spatha library into STen, and the implementation of a specific 2nd order pruning technique to exploit the V:N:M format, enables distributed sparse training as a direct application of the previously mentioned contributions. Furthermore, notice that the Spatha library represents a tool to perform general Sparse Matrix-Matrix Multiplications, so can be extended to other domains other than DL.

b) *Distributed deep learning systems.* In this work, we have focused on large-scale models based on LLMs. However, the Spatha library represents a generic tool for sparse MMMs. To achieve efficient large-scale DL on distributed systems, data, operator, and pipeline parallelism are often combined. In this context, Spatha can serve as a third-party implementation to accelerate the execution of these operators in the backend, and mitigate the computation bottleneck on these systems.

10 CONCLUSION

This paper opens the possibility to use Sparse Tensor Cores (SPTCs) for arbitrary sparsity levels and N:M patterns. In order to do so, we defined a new sparse format (V:N:M), a new library to efficiently exploit the proposed kernel (Spatha), and a second-order pruning technique that demonstrated the applicability of the proposed format on real-world deep learning models. The experiments show that this three-fold approach yields up to a 37× speedup over cuBLAS at the kernel level. Furthermore, the proposed pruning technique offers a solution scalable to the dimensionality of LLMs, and is able to achieve high sparsity ratios with minimum impact in loss (∼ 2% at 2:16 sparsity on BERT models). Finally, we demonstrate the performance on end-to-end sparsity, achieving speedups on GPT-3 encoder of up to 3.20× at 2:32 sparsity, what is translated into a tensor contraction improvement of up to 11×.

ACKNOWLEDGMENTS

This research was supported by the Ministry of Science and Innovation of Spain (grants PID2019-104184RB-I00 and PID2022-136435NB-I00, funded by MCIN/AEI/ 10.13039/501100011033, PID2022 also funded by "ERDF A way of making Europe", EU), the Ministry of Education (predoctoral grant of Roberto L. Castro, FPU19/03974), by Xunta de Galicia under the Consolidation Program of Competitive Reference Groups (ED431C 2021/30), and ERC grant PSAP, no. 101002047. We also acknowledge the support from CITIC, funded by Xunta de Galicia and FEDER funds of the EU (Centro de Investigación de Galicia accreditation 2019-2022, ED431G 2019/01). Finally, we thank the Swiss National Supercomputing Center (CSCS) and the Centro de Supercomputación de Galicia (CESGA) for the use of their computers.

REFERENCES

[1] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language Models Are Few-Shot Learners. In *Proceedings of the 34th International Conference on Neural Information Processing Systems* (Vancouver, BC, Canada) *(NIPS'20)*. Curran Associates Inc., Red Hook, NY, USA, Article 159, 25 pages.

[2] Roberto L. Castro, Diego Andrade, and Basilio B. Fraguela. 2023. Probing the Efficacy of Hardware-Aware Weight Pruning to Optimize the SpMM Routine on Ampere GPUs. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques* (Chicago, Illinois) *(PACT '22)*. Association for Computing Machinery, New York, NY, USA, 135–147. https://doi.org/10.1145/3559009.3569691

[3] Zhaodong Chen, Zheng Qu, Liu Liu, Yufei Ding, and Yuan Xie. 2021. 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* (St. Louis, Missouri) *(SC '21)*. Association for Computing Machinery, New York, NY, USA, Article 78, 14 pages. https://doi.org/10.1145/3458817.3476182

[4] Zhaodong Chen, Zheng Qu, Yuying Quan, Liu Liu, Yufei Ding, and Yuan Xie. 2023. Dynamic N:M Fine-Grained Structured Sparse Attention Mechanism. In *Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming* (Montreal, QC, Canada) *(PPoPP '23)*. Association for Computing Machinery, New York, NY, USA, 369–379. https://doi.org/10.1145/3572848.3577500

[5] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2019, Minneapolis, MN, USA, June 2-7, 2019, Volume 1 (Long and Short Papers)*, Jill Burstein, Christy Doran, and Thamar Solorio (Eds.). Association for Computational Linguistics, Minneapolis, Minnesota, 4171–4186. https://doi.org/10.18653/v1/n19-1423

[6] Elias Frantar and Dan Alistarh. 2023. SparseGPT: Massive Language Models Can Be Accurately Pruned in One-Shot. arXiv:2301.00774 [cs.LG]

[7] Elias Frantar, Eldar Kurtic, and Dan Alistarh. 2021. M-FAC: Efficient Matrix-Free Approximations of Second-Order Information. arXiv:2107.03356 [cs.LG]

[8] Trevor Gale, Erich Elsen, and Sara Hooker. 2019. The State of Sparsity in Deep Neural Networks. arXiv:1902.09574 [cs.LG]

[9] Trevor Gale, Matei Zaharia, Cliff Young, and Erich Elsen. 2020. Sparse GPU Kernels for Deep Learning. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '20)*. IEEE Press, Atlanta, Georgia, Article 17, 14 pages.

[10] Google Research. 2020. Deep Learning Matrix Collection. Retrieved March 26, 2023 from https://github.com/google-research/google-research/tree/master/sgk

[11] Song Han, Huizi Mao, and William J. Dally. 2016. Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding. arXiv:1510.00149 [cs.CV]

[12] Joel Hestness, Sharan Narang, Newsha Ardalani, Gregory Diamos, Heewoo Jun, Hassan Kianinejad, Md. Mostofa Ali Patwary, Yang Yang, and Yanqi Zhou. 2017. Deep Learning Scaling is Predictable, Empirically. arXiv:1712.00409 [cs.LG]

[13] Torsten Hoefler, Dan Alistarh, Tal Ben-Nun, Nikoli Dryden, and Alexandra Peste. 2021. Sparsity in Deep Learning: Pruning and Growth for Efficient Inference and Training in Neural Networks. *J. Mach. Learn. Res.* 22, 1, Article 241 (jan 2021), 124 pages.

[14] Andrei Ivanov, Nikoli Dryden, Tal Ben-Nun, Saleh Ashkboos, and Torsten Hoefler. 2023. STen: Productive and Efficient Sparsity in PyTorch. arXiv:2304.07613 [cs.LG]

[15] Andrei Ivanov, Nikoli Dryden, Tal Ben-Nun, Shigang Li, and Torsten Hoefler. 2021. Data Movement Is All You Need: A Case Study on Optimizing Transformers. In *Proceedings of Machine Learning and Systems*, A. Smola, A. Dimakis, and I. Stoica (Eds.), Vol. 3. San Jose, CA, USA, 711–732.

[16] Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B. Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. 2020. Scaling Laws for Neural Language Models. arXiv:2001.08361 [cs.LG]

[17] Olga Kovaleva, Saurabh Kulshreshtha, Anna Rogers, and Anna Rumshisky. 2021. BERT Busters: Outlier Dimensions that Disrupt Transformers. arXiv:2105.06990 [cs.CL]

[18] Eldar Kurtic and Dan Alistarh. 2022. GMP*: Well-Tuned Gradual Magnitude Pruning Can Outperform Most BERT-Pruning Methods. arXiv:2210.06384 [cs.CL]

[19] Eldar Kurtic, Daniel Campos, Tuan Nguyen, Elias Frantar, Mark Kurtz, Benjamin Fineran, Michael Goin, and Dan Alistarh. 2022. The Optimal BERT Surgeon: Scalable and Accurate Second-Order Pruning for Large Language Models. arXiv:2203.07259 [cs.CL]

[20] François Lagunas, Ella Charlaix, Victor Sanh, and Alexander M Rush. 2021. Block Pruning For Faster Transformers. arXiv:2109.04838 [cs.LG]

[21] Yann LeCun, John Denker, and Sara Solla. 1989. Optimal Brain Damage. In *Proceedings of Neural Information Processing Systems (NIPS'89)*, D. Touretzky (Ed.), Vol. 2. Morgan-Kaufmann, Denver, Colorado.

[22] Kwangbae Lee, Hoseung Kim, Hayun Lee, and Dongkun Shin. 2020. Flexible Group-Level Pruning of Deep Neural Networks for on-Device Machine Learning. In *Proceedings of the 23rd Conference on Design, Automation and Test in Europe* (Grenoble, France) *(DATE '20)*. EDA Consortium, San Jose, CA, USA, 79–84.

[23] Shigang Li, Kazuki Osawa, and Torsten Hoefler. 2022. Efficient Quantized Sparse Matrix Operations on Tensor Cores. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC '22)*. IEEE Press, Dallas, Texas, Article 37, 15 pages.

[24] Paul Michel, Omer Levy, and Graham Neubig. 2019. Are Sixteen Heads Really Better than One?. In *Proceedings of Neural Information Processing Systems (NIPS'19)*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett (Eds.), Vol. 32. Curran Associates, Inc., Vancouver, Canada.

[25] Asit Mishra, Jorge Albericio Latorre, Jeff Pool, Darko Stosic, Dusan Stosic, Ganesh Venkatesh, Chong Yu, and Paulius Micikevicius. 2021. Accelerating Sparse Deep Neural Networks. arXiv:2104.08378 [cs.LG]

[26] NVIDIA. 2020. Exploiting NVIDIA Ampere Structured Sparsity with cuSPARSELt. Retrieved March 26, 2023 from https://developer.nvidia.com/blog/exploiting-ampere-structured-sparsity-with-cusparselt/

[27] NVIDIA. 2023. The cuSparse Library. Retrieved March 26, 2023 from https://docs.nvidia.com/cuda/cusparse/index.html

[28] Ali Pinar and Michael T. Heath. 1999. Improving Performance of Sparse Matrix-Vector Multiplication. In *Proceedings of the 1999 ACM/IEEE Conference on Supercomputing* (Portland, Oregon, USA) *(SC '99)*. Association for Computing Machinery, New York, NY, USA, 30–es. https://doi.org/10.1145/331532.331562

[29] Jeff Pool and Chong Yu. 2021. Channel Permutations for N:M Sparsity. In *Proceedings of Neural Information Processing Systems (NIPS'21)*, M. Ranzato, A. Beygelzimer, Y. Dauphin, P.S. Liang, and J. Wortman Vaughan (Eds.), Vol. 34. Curran Associates, Inc., Virtual-only Conference, 13316–13327.

[30] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. 2019. Language models are unsupervised multitask learners. *OpenAI blog* 1, 8 (2019), 9.

[31] Victor Sanh, Thomas Wolf, and Alexander Rush. 2020. Movement pruning: Adaptive sparsity by fine-tuning. *Proceedings of Neural Information Processing Systems (NIPS'20)* 33 (2020), 20378–20389.

[32] Emma Strubell, Ananya Ganesh, and Andrew McCallum. 2019. Energy and Policy Considerations for Deep Learning in NLP. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*. Association for Computational Linguistics, Florence, Italy, 3645–3650. https://doi.org/10.18653/v1/P19-1355

[33] Wei Sun, Ang Li, Tong Geng, Sander Stuijk, and Henk Corporaal. 2023. Dissecting Tensor Cores via Microbenchmarks: Latency, Throughput and Numeric Behaviors. *IEEE Transactions on Parallel and Distributed Systems* 34, 1 (jan 2023), 246–261. https://doi.org/10.1109/tpds.2022.3217824

[34] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. 2017. Attention is All you Need. In *Proceedings of Neural Information Processing Systems (NIPS'17)*, I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett (Eds.), Vol. 30. Curran Associates, Inc., Long Beach, CA, USA. https://proceedings.neurips.cc/paper_files/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf

[35] Elena Voita, David Talbot, Fedor Moiseev, Rico Sennrich, and Ivan Titov. 2019. Analyzing Multi-Head Self-Attention: Specialized Heads Do the Heavy Lifting, the Rest Can Be Pruned. arXiv:1905.09418 [cs.CL]

[36] Ziheng Wang, Jeremy Wohlwend, and Tao Lei. 2020. Structured Pruning of Large Language Models. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Association for Computational Linguistics, Online, 6151–6162. https://doi.org/10.18653/v1/2020.emnlp-main.496

[37] Da Yan, Wei Wang, and Xiaowen Chu. 2020. Demystifying Tensor Cores to Optimize Half-Precision Matrix Multiply. In *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, New Orleans, Louisiana, 634–643. https://doi.org/10.1109/IPDPS47924.2020.00071

[38] Qingru Zhang, Simiao Zuo, Chen Liang, Alexander Bukharin, Pengcheng He, Weizhu Chen, and Tuo Zhao. 2022. PLATON: Pruning Large Transformer Models with Upper Confidence Bound of Weight Importance. In *Proceedings of the 39th International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 162)*, Kamalika Chaudhuri, Stefanie Jegelka, Le Song, Csaba Szepesvari, Gang Niu, and Sivan Sabato (Eds.). PMLR, Baltimore, Maryland, 26809–26823.

Appendix: Artifact Description/Artifact Evaluation

ARTIFACT DOI

https://doi.org/10.5281/zenodo.8084447

ARTIFACT IDENTIFICATION

Sparse Tensor Cores (SPTCs) have been recently incorporated in modern architectures of NVIDIA GPUs, enabling the execution of sparse matrices on Tensor Core Units. However, an important limitation is that they only support sparse matrices following the 2:4 format, hence 50% sparse matrices. This article proposes (1) the Vectorized N:M format, abbreviated as V:N:M, which enables the execution of arbitrary N:M patterns and sparsity levels on SPTCs. First, to effectively manage the V:N:M format, the computational artifacts provide a class that specifically addresses it. This class simplifies the management of V:N:M sparse data and facilitates its conversion to other matrix representations (both sparse and dense) for benchmarking and testing purposes.

As a result of the new V:N:M format, this article has the following set of additional contributions:

(2) A highly optimized template-based library called Spatha which implements the sparse matrix-matrix multiplication (SpMM), and efficiently exploits the V:N:M format. Artifacts demonstrate that Spatha can achieve speedups of up to 37× over the vendor library for dense MMMs (i.e., cuBLAS). Furthermore, they also show that Spatha provides speedups of up to 1.38× over the vendor library for 2:4 sparsity (i.e., cuSparseLt).
(3) A second-order pruning technique targeting the V:N:M format scalable to the dimensionality of LLMs, which allows the sparsification to high-sparsity ratios. Artifacts demonstrate that LLMs such as BERT, can target high sparsity levels with minimum increase in loss.
(4) The integration of Spatha on real-world LLMs for end-to-end inference. The computational artifacts show that Spatha can achieve a GEMM time reduction of 11× at 2:32 sparsity on real-world models such as GPT-3.

The software architecture of the computational artifacts is structured as follows: hyperref

- A centralized benchmarking tool (located in `src/`) for testing, evaluating, and comparing existent SpMM implementations. It provides a user-friendly interface that supports a range of sparse formats (e.g., CSR, CVS, N:M, V:N:M), GEMM implementations (e.g. cuBLAS, cuBLASLt), and SpMM implementations (e.g. CLASP, Sputnik, Spatha, cuSparseLt).
- A set of sparse libraries (located in `include/`) that implement the SpMM routine, which includes Spatha. The software architecture design of Spatha is described in-depth in the article.
- A set of scripts to reproduce microbenchmarking evaluations (located in `benchmark/`), with the corresponding python scripts to represent the results obtained (located in `plot/`)
- The integration of the proposed 2nd order technique into the SparseML library which allows to easily sparsify neural networks by means of pruning recipes (located in `sparseml/`).
- The integration of Spatha into the Pytorch pipeline in order to benchmark the end-to-end sparse inference performance on real-world models (located in `end2end/`)

The computational artifacts provided can execute all the experiments described in this article and contains scripts to simplify this process, enabling the easy reproducibility of all the results.

The source code is publicly available at https://github.com/UDC-GAC/venom

REPRODUCIBILITY OF EXPERIMENTS

The experiment workflow can be divided into three main parts: (1) kernel benchmarks, (2) application benchmarks for LLMs inference, and (3) application benchmarks for LLMs sparsification.

(1) All kernel benchmarks were run on a single NVIDIA RTX 3090 GPU. Kernel benchmarks represent an in-depth evaluation of the V:N:M format and the performance of the Spatha library. To obtain accurate timing, each kernel has been executed over 100 iterations, preceded by 10 iterations of warm-up. Different kernel configurations are executed for each problem size to finally select the best one depending on the input dynamics. All the proposed benchmarks produce the time in ms, formated as a CSV file that can be post-processed later to generate the corresponding plots (see `plot/`). The detailed content of these benchmarks is the following:

- **Spatha performance and overhead of the column-loc structure** (`benchmark/run_ablation1.sh`). This microbenchmark analyzes the general performance of the Spatha library for different V:N:M configurations and different problem sizes. Furthermore, it also studies the overhead of the column-loc structure in the total execution time by reproducing an ideal scenario where no memory accesses to this structure are performed. The corresponding plot shows the performance of the previous configurations in terms of speedup w.r.t. cuBLAS. This microbenchmark demonstrates that Spatha can achieve speedups of up to 37× over the vendor library for dense MMM, and that the overhead of the column-loc structure is negligible in the total time. The results of this microbenchmark are related to *Figure 9* in the article. The execution time to reproduce this experiment is about 1 hour.
- **Impact of V and wider memory storage instructions** (`benchmark/run_ablation2.sh`). This microbenchmark shows the impact of varying the value of V in the proposed format. Furthermore, it also considers different M values, or equivalently, different sparsity levels, and two kinds of SMEM storage instructions: 32 and 128-bit. The results are depicted in terms of speedup over the vendor library for dense computation (i.e., cuBLAS). The results show that the larger the value of V, the better the performance.

They also demonstrate that the usage of 128-bit instructions improves the final performance and that this improvement is more noticeable when the output writing is more relevant in the total time, for instance, when the sparsity level increases and the problem gets smaller. The results of this microbenchmark are related to *Figure 10* in the article. The execution time to reproduce this experiment is about 5 minutes.

- **Baseline performance** (`benchmark/run_baseline.sh`) This microbenchmark shows the performance of the baseline SpMM implementation at 2:4 sparsity using different problem sizes. It compares the performance of Spatha with the vendor library implementations for both dense (i.e., cuBLAS), and sparse MMM following the 2:4 format (i.e., cuSparseLt). This microbenchmark shows the baseline performance of Spatha and cuSparseLt in terms of speedup and TFLOPs/s w.r.t. cuBLAS. The results demonstrate that Spatha achieves a competitive performance w.r.t. cuSparseLt at larger problem sizes, approaching the 2× peak performance over cuBLAS. Furthermore, on smaller problem sizes, Spatha also provides speedups of up to 1.38× over cuSparseLt. The results of this microbenchmark are related to *Figure 12* in the article. The execution time to reproduce this experiment is about 20 minutes.
- **Comparison with other libraries for MMMs** (`benchmark/run_spmm_spatha.sh`). This benchmark compares the performance of Spatha with other dense (e.g., cuBLAS) and sparse libraries (e.g., CLASP, Sputnik, cuSparseLt) on different problem sizes and different sparsity levels. Furthermore, it allows to tune different sparse format configurations such as V in the V:N:M format (Spatha), or the vector length in the vector-wise format (CLASP). The performance of each library is represented in terms of speedup w.r.t. cuBLAS. The results show that Spatha achieves unprecedented performance results, obtaining up to 27× speedup over the vendor library for dense computation. The results of this benchmark are related to *Figure 13* in the article. The execution time to reproduce this experiment is about 3 hours

(2) All application benchmarks for LLMs inference have been conducted after streamlining the adoption of Spatha into the PyTorch pipeline by integrating it with the STen library. All the benchmarks were run on a single NVIDIA RTX 3090 GPU. The models to benchmark have been obtained from Huggingface, and the scripts provided allow to easily add new models to the evaluation. In order to obtain accurate timing results, each inference process has been repeated over 30 times, preceded by a warm-up stage. Pytorch profiling tools have been used to study the impact of each LLM component separately, which also allows to analyze how the Spatha library performs in the general process.

- **LLM inference latency** (`end2end/run_inference.sh`) This benchmark evaluates the inference latency of different sparse LLMs using Spatha, and compares it with the performance of the original Pytorch dense models. The script provided produces the times in ms, formatted as a CSV that can be used later to plot the results (`plot/run_inference.py`). The results show that the larger the impact of the GEMM instruction in the model, and the higher the arithmetic intensity, the better the performance of the sparse inference. In that sense, models such as GPT-3, composed of large matrices, can achieve speedups of up to 11× in terms of GEMM, and 3.20× in the whole encoder inference at 2:32 sparsity. The results of this benchmark are related to *Figure 15* in the article. The execution time to reproduce this experiment is about 10 minutes

(3) Application benchmarks for LLMs sparsification focus on demonstrating the applicability of the V:N:M format on real-world DL models. The evaluation workflow related to LLMs sparsification is the following: url

- **Energy evaluation study** (`benchmark/energy.py`) This test performs an energy evaluation study over different pruning approaches and sparsity levels applied to weight linear layers of real-world LLMs. The following pruning algorithms are considered in the evaluation: magnitude pruning, vector-wise pruning, N:M pruning, and V:N:M pruning. Furthermore, for vector-wise and V:N:M pruning, the experiment evaluates different algorithm configurations that involve the vector length, and the V value, respectively. Energy is calculated as a value between $0 \sim 1$, the higher the better. The results show by means of a bar plot that V:N:M offers an intermediate solution between magnitude and vector-wise pruning, and that this method is robust to large V values, allowing the usage of $V = 128$ and be consistently better than vector-wise with a vector length of 8. The results of this benchmark are related to *Figure 11* in the article. The execution time to reproduce this experiment is about 6 minutes.
- **LLMs sparsification with second-order pruning** This experiment represents a fork of the SparseML library extended to the V:N:M format, which enables the sparsification of neural networks using pruning recipes. We can divide the contributions to this library into three parts: (a) pruning algorithms (`sparseml/sparseml/pytorch/sparsification/pruning`), (b) pruning recipes (`sparseml/integrations/huggingface-transformers/recipes`), and (c) pruning scripts (`sparseml/integrations/huggingface-transformers/scripts`). Specifically, (a) contains all the pruning algorithms implemented and used in this article, (b) defines different pruning recipes to sparsify BERT models using the previous algorithms, and (c) contains a set of scripts that configure and launch the sparsification process. The result of each experiment is a sparse network tuned on the SQUAD v.1.1. task, which is a widely used task to measure model compression performance. The model accuracy is reported in terms of F1. Experiments demonstrate that the proposed 2nd order pruning technique can achieve high-sparsity ratios with minimum increase in loss. The results of this benchmark are related to *Table 2* in the article. Each sparsification process was executed with 3 RTX 3090 GPUs. In this terms, considering that the amount of time to reproduce the results can be high, we provide three different scripts

to alleviate this: (1) `sparseml/sparseml_SS1.sh` that contains a subset of the experiments with the most aggressive configurations using the pair-wise version of the sparsifier (about 4 days), (2) `sparseml/sparseml_SS2.sh` that contains all the sparsity-format configurations but relaxed with pair-wise version of the sparsifier (about 10 days), and (3) `sparseml/sparseml_SS3.sh` that contains all the sparsity-format configurations and performs the exhaustive search process (about 25 days).

ARTIFACT DEPENDENCIES REQUIREMENTS

We ran all the experiments on NVIDIA RTX 3090 GPU, which is also the recommended hardware to run the artifact. This GPU architecture has 82 Streaming Multiprocessor (SM) elements that share a 6MB L2 cache and 24GB of DRAM. Each SM share a 128KB L1 cache that can be partially used as Shared MEMory (SMEM). The software environment is:

- Operating system and version: Ubuntu 20.04.2 LTS
- Compiler and version: GCC 9.4.0
- CUDA Toolkit 11.5 or 11.7
- cuSparseLt v.0.3.0
- Python 3.10
- PyTorch 1.13.1
- cmake 3.16.3

The method provided to reproduce the results involves a pre-configured software environment within a docker container. For GPU execution, `nvidia-docker` installation is necessary, and it greatly simplifies the process.

Input data (DL models and datasets) is automatically downloaded from public and extensively used repositories such as HuggingFace.

The source code of VENOM is publicly available at: https://github.com/UDC-GAC/venom
Persistent ID [docker-container]: https://doi.org/10.5281/zenodo.8084447
Persistent ID [source-code]: https://doi.org/10.5281/zenodo.8085824

ARTIFACT INSTALLATION DEPLOYMENT PROCESS

Reproduction of the artifact with container. Please, follow the next steps to reproduce results with docker container:

Step 1: Download and run the container
Option 1: download an already-built docker image
wget https://zenodo.org/record/8084447/files/venom_container.tar.gz
Run the container by:
docker load -i venom_container.tar.gz
docker run -it –gpus all venom_container
Option 2: build the container from scratch
git clone –recurse-submodules git@github.com:UDC-GAC/venom.git && cd venom
docker build -t venom_container .
docker run -it –gpus all –name <your_container_name> venom_container
Step 2: Compile and run the experiments

Compilation is already inlined in the scripts provided, so you can jump directly to (1) if you plan to follow the artifact scripts. However, the instructions to build and install the code are the following:

Build and install the centralized benchmarking tool:

```
mkdir build && cd build
# about 1 minute
cmake .. -DCMAKE_BUILD_TYPE=Debug -DCUDA_ARCHS="86"
-DBASELINE=OFF -DIDEAL_KERNEL=OFF -DOUT_32B=OFF &&
make -j 16
```

Three compiling options are defined to build the following kernel versions:

- -DBASELINE: baseline Spatha implementation for 2:4 sparsity
- -DIDEAL_KERNEL: Spatha N:M implementation without column-loc structure overhead (ideal situation)
- -DOUT_32B: Spatha N:M implementation with 32-bit storage instructions. By default 128-bit instructions are used.

Build and install VENOM as a Python module:

```
cd end2end
# about 1 minute
./install.sh
```

(1) To reproduce the results on Fig 9

```
cd /projects/venom/
# about 1 hour
./benchmark/run_ablation1.sh
python plot/run_ablation1.py
```

(2) To reproduce the results on Fig 10

```
cd /projects/venom/
# about 5 minutes
./benchmark/run_ablation2.sh
python plot/run_ablation2.py
```

(3) To reproduce the results on Fig 12

```
cd /projects/venom/
# about 20 minutes
./benchmark/run_baseline_a.sh
./benchmark/run_baseline_b.sh
python plot/run_baseline_a.py
python plot/run_baseline_b.py
```

(4) To reproduce the results on Fig 13

```
cd /projects/venom/
# about 2 hours
./benchmark/run_spmm_spatha.sh
python plot/run_spmm_spatha.py
```

(5) To reproduce the results on Fig 15

```
conda activate end2end
# about 10 minutes
./end2end/run_inference.sh
python3 plot/run_inference.py
```

(6) To reproduce the results on Fig 11

```
conda activate end2end
```

```
python3 benchmark/energy.py
```

(7) Since reproducing results on Table 2 can take a significant amount of time, we provide three different scripts to alleviate this process.

```
conda activate sparseml_artf
cd sparseml
# Script that contains a subset of the experiments with the most aggressive configurations using the pair-wise version of the sparsifier
# about 4 days
./sparseml_SS1.sh
# Script that contains all the sparsity-format configurations but relaxed with pair-wise version of the sparsifier
# about 10 days
./sparseml_SS2.sh
# Script that contains all the sparsity-format configurations and performs the exhaustive search process
# about 25 days
./sparseml_SS3.sh
```

Note: each script in *integrations/huggingface-transformers/scripts* has two execution possibilities. Please, uncomment the first line if you want to use a single-GPU, or the second one with the total number of GPUs available for multiple-GPU execution.

```
# single-GPU
CUDA_VISIBLE_DEVICES=0 python3.10 src/sparseml/transformers/question_answering.py
# multi-GPU (3 in this example)
python3.10 -m torch.distributed.launch –nproc_per_node=3 src/sparseml/transformers/question_answering.py
```

Step 3: check plots

```
cd /projects/venom/result
scp *.pdf username@hostmachine:/host/path/target
```