High Performance Unstructured SpMM Computation Using Tensor Cores

Patrik Okanovic
ETH Zurich
Department of Computer Science
Zurich, Switzerland
patrik.okanovic@inf.ethz.ch

Grzegorz Kwasniewski
ETH Zurich
Department of Computer Science
Zurich, Switzerland
gkwasnie@inf.ethz.ch

Paolo Sylos Labini
Free University of Bozen-Bolzano
Faculty of Engineering
Bolzano, Italy
Paolo.SylosLabini@student.unibz.it

Maciej Besta
ETH Zurich
Department of Computer Science
Zurich, Switzerland
maciej.besta@inf.ethz.ch

Flavio Vella
University of Trento
Trento, Italy
flavio.vella@unitn.it

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

***Abstract*—High-performance sparse matrix–matrix (SpMM) multiplication is paramount for science and industry, as the ever-increasing sizes of data prohibit using dense data structures. Yet, existing hardware, such as Tensor Cores (TC), is ill-suited for SpMM, as it imposes strict constraints on data structures that cannot be met by unstructured sparsity found in many applications. To address this, we introduce (S)parse (Ma)trix Matrix (T)ensor Core-accelerated (SMaT): a novel SpMM library that utilizes TCs for unstructured sparse matrices. Our block-sparse library leverages the low-level CUDA MMA (matrix-matrix-accumulate) API, maximizing the performance offered by modern GPUs. Algorithmic optimizations such as sparse matrix permutation, further improve performance by minimizing the number of non-zero blocks. The evaluation on NVIDIA A100 shows that SMaT outperforms SotA libraries (DASP, cuSPARSE, and Magicube) by up to 125x (on average 2.6x). SMaT can be used to accelerate many workloads in scientific computing, large model training, inference, and others.**

***Index Terms*—Mathematics of computing, SpMM, Matrix Multiplication, Tensor Cores**

I. Introduction

The performance of dense matrix multiplication has steadily improved in recent years, with new architectures and libraries feeding the growing computational needs of deep learning. *MMA* units such as the Tensor Processing Unit (TPU) [1] and the Tensor Core (TC) [2] are designed to efficiently handle big volumes of *multiply-accumulate* operations, a fundamental workload in scientific computing. However, the performance of multiplying a *sparse* matrix with a dense one (SpMM), which is not only crucial for important High-Performance workloads [3] or graph and data analytics [4]–[8] but also constitutes a growing part of many modern workloads—most notably, Graph Neural Networks (GNNs) [9]–[13] and general Sparse Deep Learning [14], [15], is far from reaching hardware peak performance.

It is still unclear how to leverage the hardware (HW) and software (SW) machinery developed for dense matrix computations in applications operating on sparse matrices. Thus, the computational power of dense matrix units, which are common in most high-performance hardware configurations, still remains untapped. Thanks to the efficient use of memory hierarchies and matrix units, explicitly storing zeros and using dense representation can be faster than computations on sparse data structures, even on very sparse matrices. For example, on the NVIDIA HW/SW stack, the sparsity threshold for the supremacy of dense multiplication (cuBLAS) over sparse SpMM (cuSPARSE) lays as high as 99.9% [16], depending on the size of the matrix and its sparsity pattern. Multiplying sparse matrices as if they were dense, however, is inherently inefficient due to *padding*—explicitly stored zeros. As the size and sparsity of the matrices grow, padding increases the storage requirements and data movement costs compared to sparse storage, relegating these powerful routines to small matrices. Most importantly, padding reduces the utilization of matrix units, wastefully processing null elements.

A natural approach to reduce padding and increase the utilization of matrix units, and thus to make the SpMM based on dense matrix units faster and more memory-efficient, is *blocking*—tiling a sparse matrix into blocks, and only storing and processing the nonzero ones [17]. Blocking enables partitioning a sparse matrix into a collection of dense matrices—the nonzero blocks—which can then be multiplied using dense units. However, two main challenges make reaching peak hardware performance of blocked-sparse matrix multiplication still an open problem: a) **block density**: finding the optimal blocking that minimizes the amount of padding - the number of explicit zeros per each dense block that increase memory footprint and account for wasted arithmetic operations; and b) **hardware-aware implementation**: efficient streaming of consecutive blocks to fully utilize memory pipelines, hiding the load latency and saturating all TC units.§

While the problem of efficient utilization of MMA units for sparse matrices was addressed in previous research, existing work mostly focus on narrow use-cases. Magicube [18] is

SC24, November 17-22, 2024, Atlanta, Georgia, USA
979-8-3503-5291-7/24/$31.00 ©2024 IEEE

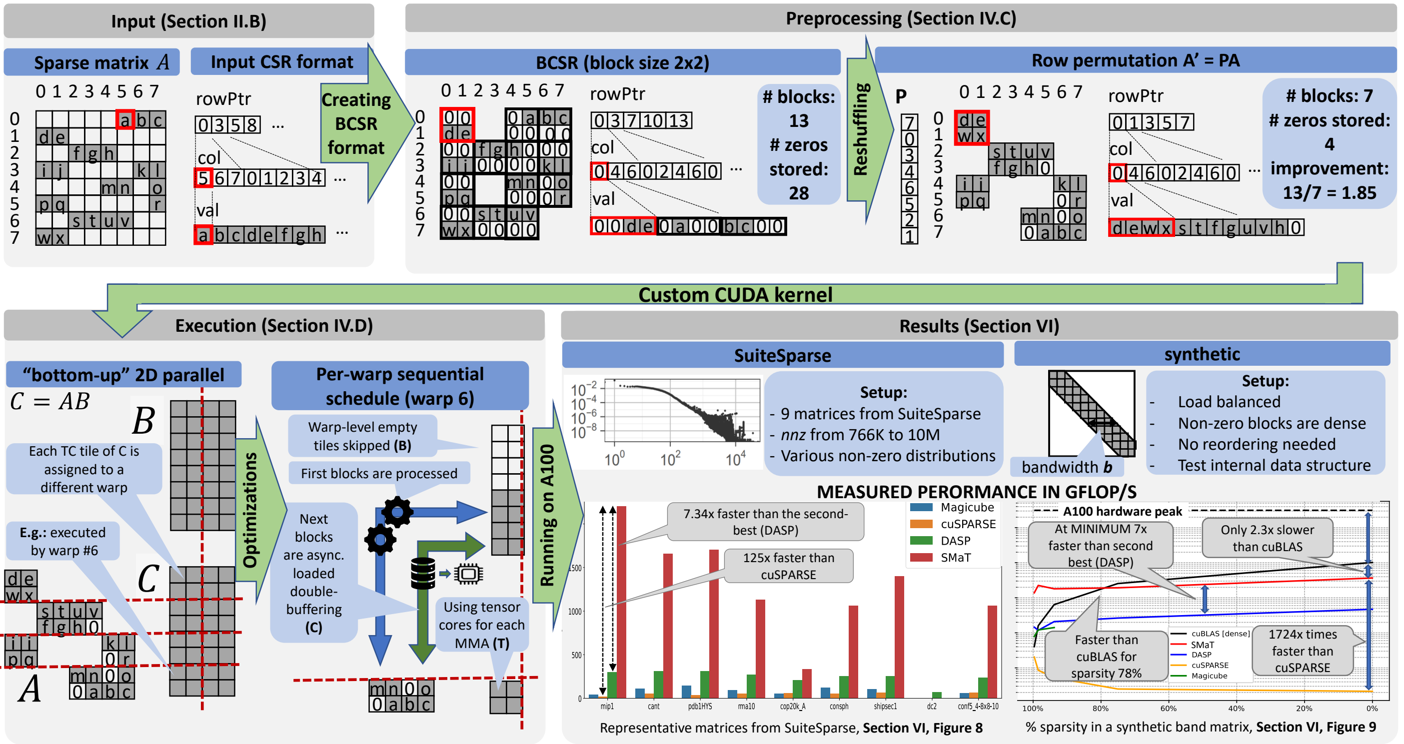


Fig. 1: A bird's-eye view of the entire SMaT's pipeline. SMaT performs SpMM on an input matrix in the CSR format stored in any precision supported by Tensor Cores. Then, it preprocesses the matrix to maximize the block density, minimize the total number of blocks, and maximize the load balance between rows. The preprocessing is done only once and the matrix is internally stored in the BCSR (Blocked-CSR) format. When the SpMM kernel is launched, an optimized CUDA kernel uses block-level bottom-up 2D parallelism to maximize the utilization of GPU hardware resources. The results - both on the SuiteSparse and on synthetic matrices confirm the universality of SMaT: it significantly outperforms remaining solutions in almost every test case: for very sparse and relatively dense matrices, for highly unstructured and for very regular matrices.

an SpMM library specifically designed for deep learning, requiring structured sparsity and supports only low-precision integers. DASP [19] supports only SpMV operation, which can be viewed as a special case of SpMM, with the dense matrix containing only a single column. NVIDIA's general-purpose cuSPARSE library [20] relies on a CSR format, but experiments show that it underperforms in many scenarios [18], [19], [21]. cuSPARSELt and VENOM [22] work only on fixed sparsity patterns.

In this work, we introduce *SMaT* — (S)parse (Ma)trix Matrix (T)ensor Core-accelerated library. It is a *general-purpose* SpMM library that works on unstructured sparse matrices in CSR format - arguably the most prevalent format for this purpose. It works with all data types supported by the MMA hardware units. The library first does the preprocessing permutation of the sparse matrix to *minimize the number of dense blocks*. Then, our highly-optimized low-level CUDA implementation *efficiently streams thread-level blocks*, overlapping computation with data movement and saturating all hardware TC units using bottom-up [23] 2D parallelism.

SMaT, while being a general-purpose library, significantly outperforms both vendor-optimized cuSPARSE, as well as use-case-specific SotA solutions like Magicube and DASP. We compare against the DASP SpMV library by treating SpMM as a batched SpMV and we show that SMaT outperforms DASP for batch size as small as 4. Our empirical performance model (Section III) outlines our driving design choices: hardware-aware, data-movement-centric code design can bring more performance to contemporary GPU architectures than even sophisticated preprocessing algorithms aimed to reduce the number of blocks.

Experiments on synthetic matrices show up to 2,445 times improvement over cuSPARSE, with at least a minimum of 5.3x improvement compared to the second-best library (Section VI-C). Furthermore, we compared the performance of SMaT against cuBLAS: a vendor-optimized GEMM library for *dense* matrices, and show that contrary to previous findings [16] we outperform cuBLAS for sparsity regimes as low as 78%. Evaluated on the real-world matrices from cuSPARSE, we measure up to $8.6\times$ performance improvement (on average $4.8\times$). The high-level design together with representative results are presented in Figure 1.

To summarize, we provide the following contributions:

- SMaT: an end-to-end solution for general-purpose SpMM that supports unstructured sparsity and all data types supported by the TC units,

- A sparse matrix preprocessing permutation scheme that decreases the number of dense blocks by up to 2.5x,
- A high-performance implementation of the blocked-CSR (BCSR) SpMM on Tensor Cores using the low-level CUDA MMA API,
- An empirical performance model for SpMV in BCSR format that quantifies both the impact of preprocessing and the implementation optimizations,
- An evaluation demonstrating speedups of up to 125x over cuSPARSE and up to 7.3x over second-best among tested state-of-the-art SpMM routines: cuSPARSE, Magicube, and DASP.

II. Background

Throughout the paper, we consider a matrix-matrix multiplication $C = AB$ with $C \in \mathbb{F}^{M \times N}$, $A \in \mathbb{F}^{M \times K}$, and $B \in \mathbb{F}^{K \times N}$, for some ring $\mathbb{F}$. Furthermore, we assume matrix A to be *sparse*: denoting the number of non-zero elements in A as *nnz*, we have $1 - nnz/(M \cdot K) = \Omega(1)$. Matrix B, of size $K \times N$, is dense. The SpMV operator (sparse Matrix-Vector multiplication) can be seen as a special case with $N = 1$.

We first introduce fundamental concepts.

A. Hardware execution model

1) Execution model: The execution model leverages a hierarchy that includes threads, warps, and thread blocks. A thread is the smallest unit of execution. A warp is group of threads that are executed simultaneously by the GPU. The size of a warp is specific to the GPU architecture, with 32 being a common size in NVIDIA GPUs. Warps execute independently but can share data with other warps in the same thread block using shared memory. They have their own set of private registers and can be synchronized within their block.

2) Tensor Cores: Tensor Cores (TCs) are specialized processing units found in most modern NVIDIA GPUs. These cores are optimized for performing mixed-precision matrix multiply-and-accumulate operations very efficiently, which are fundamental to the training and inference. TCs perform matrix operations where inputs are typically in lower precision formats (like FP16, BF16, or INT8), but accumulate results in a higher precision format (like FP32) to maintain the precision of the computations. A typical operation performed by a TC involves multiplying two small matrices and adding the result to a third matrix, i.e., Fused Multiply-Add (FMA).

When a CUDA kernel that leverages TCs is executed, the warp scheduler on the GPU is responsible for directing warps to utilize these cores efficiently. The warp scheduler must manage the distribution of matrix operations across the available TCs, ensuring that the workload is evenly distributed and that TCs are kept busy to maximize throughput.

3) Memory model: The CUDA memory model provides a hierarchy of memory options for optimizing performance in NVIDIA GPUs. While this hierarchy is common across GPU generations [24], we provide the parameters for the A100-SXM4-40GB architecture [2], as this is our primary target in the Evaluation (Section V).

- Global memory: large shared space accessible by all threads, yet slower compared to other memory types. **40GB HBM2, bandwidth 1.5TB/s.**
- Shared memory: smaller but significantly faster memory shared by threads within a block, making it suitable for data that requires frequent access by cooperating threads. **Configurable, up to 164KB per SM, 32 banks with bandwidth 64b per clock cycle.**
- Registers: smallest and fastest memory units, private to each thread, ideal for storing frequently used variables within a single thread. **256KB per SM.**

In order to utilize TCs, data needs to be strategically moved from global memory to registers. To avoid bank conflicts and efficiently overlap computation with data movement, memory alignment and software pipelining play an important role by hiding data movement latency. Further details on the implementation can be found in Section IV-D.

B. Sparse matrix representation

1) Unstructured sparsity: Without any information about the topological structure of nonzeros, each individual element has to contain information not only about its value but also about its location, yielding $\Omega(nnz)$ additional memory overhead. CSR (Compressed Sparse Row), CSC (Compressed Sparse Column), and CO (Coordinate) are arguably the most common formats for storing unstructured sparsity, with the first being dominant. Despite the memory overhead for storing locations, they tend to have the smallest memory footprint, as no zero values are stored. That comes at the cost of latency (each value's location has to be individually decoded), load balance, and cache utilization.

2) Structured sparsity: Structured sparsity has recently gained significant attention due to its applications in deep learning and the introduction of Sparse Tensor Cores (SPTC) in NVIDIA's Ampere architecture. However, SPTCs only support the 2:4 format, which limits achievable sparsity ratios to 50%. This format requires every consecutive 4 elements to have 2 nonzero values, promising a 2× speedup. Castro et al. extended the algorithmic support for M:N sparsity [22] — among every N consecutive values, M are non-zeros. Low-rank decomposition [25] can be viewed as the lossy compression format, where a large matrix is represented as a product of lower-rank matrices. Its primary advantage is that the multiplicand matrices are usually dense and naturally fit for dense GEMM-optimized libraries and hardware.

In general, the performance of structured sparse matrices is higher due to lower position decoding overhead, streamed and vectorizable access patterns, and hardware support. However, in many applications, such as analysis of real-world graphs [10], higher-order patterns can be prohibitively expensive to discover or may be even non-existent [26].

3) Blocked format: While the unstructured format addresses each individual value in the matrix, the *blocked* format addresses the block granularity: submatrices of fixed size $h \times w$ [27]. Each block is assumed to be dense, that is, all $h \cdot w$ values are stored explicitly, even if some of them are

zeros. While each block can, in principle, start at any row and column offset, in this work, we assume a fixed block structure: each block offset (the row and column index of the top left value of the block) is a multiple of h or w, respectively. Given a matrix $A \in \mathbb{F}^{n \times m}$ with n rows and m columns, and given the rectangular block sizes h and w, we identify A with its blocked version $\mathbf{A}$ (omitting the dependency on h and w). Specifically, $\mathbf{A}$ is an $N \times M$ block matrix, where $N = \lceil \frac{n}{h} \rceil$ and $M = \lceil \frac{m}{w} \rceil$. Each of its elements $\mathbf{A}_{i,j}$ is a block in $\mathbb{F}^{h \times w}$, containing all entries $A_{k,l}$ s.t. $\lfloor \frac{k}{h} \rfloor = i$ and $\lfloor \frac{l}{w} \rfloor = j$.

Once a matrix has been blocked, it can be stored and processed in a blocked storage format, where only nonzero blocks (those with at least one nonzero) are stored explicitly. We refer to zero entries within a nonzero block as *padding*.

Storing and processing padding elements is one of the two main costs of blocked SpMM, the other being accessing the nonzero blocks themselves. The former grows with the size of the blocks, while the latter grows with their number. On the one hand of this trade-off lay purely sparse storage formats, such as the established Compressed Sparse Rows (CSR), which only store nonzero elements but need to access each one separately. On the other hand stand fully dense storage schemes, which store all zeros and nonzeros in one large block. Block storage formats, such as those considered in this paper, carefully balance between these two extremes.

III. Performance model

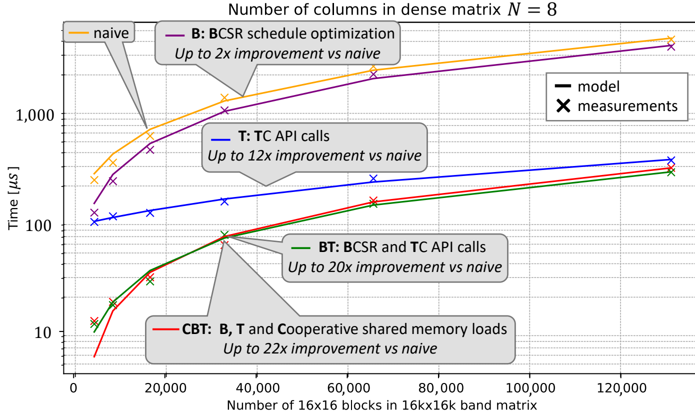


Fig. 2: Performance measurements vs. model (Equation 1) for various combinations of low-level optimizations: **C**: warp-cooperative asynchronous loading from global to shared memory using `memcpy_async`; **B**: using BCSR pointer array to skip empty-block evaluation in the inner loop; **T**: using TC MMA API (`MMA16816`)

We use a linear performance model

$$T_{tot} = T_e \cdot n_e + T_{init}, \tag{1}$$

where T_{tot} is the total runtime of the kernel, T_e is the execution time of a single compute instruction, n_e is the number of elementary computations, and T_{init} accounts for any startup, initialization, cache warm-up, and finalization overhead. Depending on the implementation, this compute instruction may be, e.g., a single scalar FMA (fused multiply-add), a vectorized operation (e.g., using AVX), or a Tensor Core MMA instruction.

We validate our model and fit the parameters on a $16k \times 16k$ band matrix with varying bandwidth (from 64 to 4096) times a tall-and-skinny dense matrix of size $16k \times 8$. We note that such a scenario does not reflect typical real-world sparse matrices, yet we want to isolate any effects of imperfect load balancing, non-zero distribution, or varying block fill-ins. In this synthetic benchmark, our model matches the measurements well (Figure 2).

This model captures two crucial aspects of high-performance SpMM and their equal contribution: low-level implementation T_e and high-level algorithmic optimization n_e.

A. Single instruction time T_e

SMaT fully utilizes TC: thus, depending on the used precision, we treat a single MMA instruction as an elementary computation (e.g., for FP16, we use 16x8x16 MMA). Optimizing MMM in CUDA requires multiple stages of careful optimizations [28] — Figure 2 shows the impact of three representative steps. In Section IV-D, we discuss these optimization in detail.

B. Number of elementary computations n_e

The number of elementary computations n_e is the number of dense blocks in the BCSR format (Section II-B3). As mentioned in Section III-A, the size of the block $h \times w$ depends on the used precision. For a sparse matrix of size $N \times M$ holding $nnz \leq N \cdot M$ nonzero values, it can be seen that n_e is bounded by

$$\frac{nnz}{h \cdot w} \leq n_e \leq \min\left(\frac{N \cdot M}{h \cdot w}, nnz\right) \tag{2}$$

The lower bound is achieved when all blocks are fully packed and there is no zero fill-in. The upper bound represents the case where there is only a single non-zero per entire block, and the remaining $h \cdot w - 1$ elements are explicitly stored zeros. Section IV-C discusses how SMaT reduces the number of blocks n_e by matrix reordering via clustering rows and columns based on the similarity metrics.

Observation: We argue that low-level kernel optimizations can play a more important role than even an optimal preprocessing algorithm. While experimenting with different reshuffling algorithms we rarely observed a reduction in the total number of blocks greater than 3 times. On the other hand, just using the TC API increases performance by 10 times, with our optimized kernel outperforming a naive implementation 22 times (Figure 2).

IV. SMaT— (S)parse (Ma)trix Matrix (T)ensor Core-accelerated library

In this section we describe SMaT — a novel SpMM library designed to utilize TCs for unstructured sparse matrices.

A. Overview

Figure 1 displays the overview of SMaT. Initially, the sparse matrix A is read in the CSR format. Internally, the matrix is converted to the block format. Afterwards, in the preprocessing phase, the matrix is reshuffled to minimize the number of non-zero blocks with row permutations **A'** $= PA$. The permuted matrix is stored and passed to the execution phase, where our custom CUDA kernel performs the matrix multiplication.

B. Data Structures

Our implementation leverages the widely used blocked format for sparse matrices, BCSR. In Figure 1, we provide an example of the BCSR format, which consists of three arrays. Similarly to CSR, the BCSR array $rowPtr$ stores the offset pointers in the col array for each block row, and col holds the column index for each block. Since the blocks are stored as dense, every $h \cdot w$ consecutive values in array val represent one block, which may require filling some values with zeros (Figure 1, top left). Matrices stored in this format can directly be used as input for the MMA Units since the block dimensions h and w of BCSR match the dimensions of the MMA API calls. This, in turn, depends on the used precision: e.g., for FP16, we use the block size 16×8, corresponding to the M16N8K16 instruction (Listing 1).

It is worth noting that both DASP and cuSPARSE use the CSR format, while Magicube uses a Strided Row-major BCRS (SR-BCRS) format. The dense vectors in SR-BCRS are stored in a stride-wise row-major manner. If the number of dense vectors in the row is not a multiple-of-stride, zero vectors are padded for the last stride.

C. Preprocessing

While finding the optimal block-minimizing permutation is NP-hard, various heuristics exist. We tested various state-of-the-art reordering schemes that cluster the non-zero values together to minimize the number of blocks. Reverse Cuthill–McKee [29] minimizes the matrix bandwidth. Saad's algorithm [30] uses a similarity metric for clustering similar rows to increase the spatial locality. Çatalyürek et al. [31] uses hypergraph partitioning techniques to minimize the cut between the rows. Zhao et al. [32] uses the Gray code ordering to maximize data locality. Sylos Labini et al. [33] use Jaccard's similarity metric to determine the blocking. In our tests, Sylos Labini's algorithm provided the best reduction in the block count. Thus, we use this as our baseline preprocessing routine.

Sylos Labini's algorithm reduces the padding with zeros by clustering similar rows together. Intuitively, similar rows have many zero and nonzero values in the same columns. In order to measure the similarity of two rows v, w they use the Jaccard distance $J(v,w) = 1 - \frac{|v \cap w|}{|v \cup w|}$, which measures the ratio of padding to the total cluster area. Their algorithm iteratively performs the following greedy procedure: create a new cluster c and choose an unclustered row v; for all other unclustered rows w check whether the Jaccard distance $dist(w, pc)$ is less than a threshold distance (pc represents the union of rows in cluster c); merge all rows for which the distance is less than the threshold. This procedure is repeated until all rows belong to some cluster.

While Sylos Labini's algorithm performs only row permutation to avoid reshuffling the right-hand-size matrix B, we tested the impact of both row and column reordering to maximize the block reduction ratio. We note that while row permutation comes with a minimal cost (the entire computation schedule is similar up to the permutation of the result matrix), permuting columns causes the overhead of accessing B, so any reduction in the number of blocks must be large enough to compensate for this. Our experiments (Section VI-A) show that performing additional column permutation does not provide sufficient benefits. Therefore, in our implementation, SMaT performs row-only permutation as a block-densification preprocessing step.

It is worth noting, that in some special cases, such as band matrices, the sparse matrix is already structured to minimize the number of non-zero blocks. In this case, the permutation P becomes the identity matrix, and the sparsity pattern of the permuted matrix **A'** is the same as for the input matrix A.

D. Implementation

Algorithm 1 Pseudocode of the warp-level SMaT kernel

Require: $valuesBcsr, rowPtrBcsr, colIdxBcsR, B, C$

1: **for** $laneid = 0$ **to** 31 **in parallel do**
2: $\quad RC \leftarrow 0$
3: $\quad$ **for** $blocks$ **in** $bcsrVals[row]$ **do**
4: $\quad\quad RD \leftarrow RC$
5: $\quad\quad$ MEMCPY_ASYNC(A_shared, $bcsrVals[idxA]$)
6: $\quad\quad$ MEMCPY_ASYNC(B_shared, $B[idxB]$)
7: $\quad\quad$ LDMATRIX_X4(RA, A_shared)
8: $\quad\quad$ LDMATRIX_X2(RB, B_shared)
9: $\quad\quad$ HMMA16816(RD, RA, RB, RC)
10: $\quad C_shared \leftarrow RC$
11: $\quad C[idxC + laneid] \leftarrow C_shared$

```
1 #define HMMA16816(RD0, RD1, RA0, RA1, RA2, RA3,\
2     RB0, RB1, RC0, RC1) \
3     asm volatile("mma.sync.aligned.m16n8k16.row.col.\
4     f16.f16.f16.f16 \
5     {%0, %1}, {%2, %3, %4, %5}, {%6, %7}, {%8, %9};\n" \
6     : "=r"(RD0), "=r"(RD1)  \
7     : "r"(RA0), "r"(RA1), "r"(RA2), "r"(RA3), \
8       "r"(RB0), "r"(RB1), "r"(RC0), "r"(RC1))
```

Listing 1: mma.m16n8k16 Tensor Core instruction in FP16

```
1 #define LDMATRIX_X2(R0, R1, addr) \
2     asm volatile("ldmatrix.sync.aligned.x2.m8n8.shared.b16 \
3     {%0, %1}, [%2];\n" : "=r"(R0), "=r"(R1) : "r"(addr))
```

Listing 2: LDMATRIX_X2

```
#define LDMATRIX_X4(R0, R1, R2, R3, addr) \
    asm volatile("ldmatrix.sync.aligned.x4.m8n8.shared.b16 \
    {%0, %1, %2, %3}, [%4];\n" \
    : "=r"(R0), "=r"(R1), "=r"(R2), "=r"(R3) \
    : "r"(addr))
```

Listing 3: LDMATRIX_X4

We now describe selected most important optimization steps employed in our library. These steps are illustrated in Figure 2. Full code is publicly available on GitHub[1].

T: TC API We execute in half precision $mma.m16n8k16$ operation. Listing 1 shows the PTX code example we incorporate in our implementation for using TCs.

B: BCSR iteration We use arrays `rowPtr` and `colIdx` from Figure 1 in order to iterate only over non-zero blocks. Without these data structures, one needs to iterate over every block and check whether that block is non-zero. More details can be found in Algorithm 1 and Section IV-B.

C: collective loads `cuda::memcpy_async` achieves overlapping computation with data movement, allowing for faster transfer of data from global memory to registers. Section IV-E contains additional information.

Algorithm 1 presents the pseudocode for the CUDA kernel of SMaT. Each warp is responsible for the calculation of a submatrix of matrix **C** such that the dimensions correspond to the dimensions of the TC. As the dense matrix **B** has dimension $N << K$, most of its width of **B** is loaded in the memory. Afterwards, non-zero blocks using the BCSR format are loaded from the global memory into registers in an efficient way. For that, we use `cuda:memcpy:async` for loading into shared memory, and the PTX command `ldmatrix` shown in Listing 2 and 3 for loading into registers in the required format. Then the execution of the TCs is called with the command `HMMA16816`. Finally, the result stored in the registers is transferred back into global memory.

E. Asynchronous Data Loads

In order to hide the latency of transferring data from the global GPU memory to shared memory we utilize `cuda::memcpy_async`. These asynchronicity features enable overlapping computations with data movement, reducing total execution time. The `cudaMemcpyAsync` function allows data movement between CPU memory and GPU global memory to be overlapped with kernel execution. Similarly, the `cuda::memcpy_async` function allows data movement from GPU global memory to shared memory to be overlapped with thread execution. It is important to note that copying data from global to shared memory without `cuda::memcpy_async` is a two-step process. The process of copying data from global memory into registers and then from registers into shared memory is performed in multiple stages through the memory hierarchy. To avoid this, `cuda::memcpy_async` can be used to directly transfer data from global memory to shared memory using DMA engines without involving registers. This frees up the thread block from the task of moving data and allows registers to be used for computations.

[1]https://github.com/spcl/smat

V. Evaluation

We note that, to the best of our knowledge, there is no high-performance library for unstructured SpMM using tensor cores that would simultaneously work on real-world, highly sparse (>90%), and irregular matrices while utilizing Tensor Cores efficiently. Existing solutions tend to either focus on the former (highly sparse unstructured matrices in the CSR format [20]) or the latter (utilizing TC for relatively dense, small, structured matrices in very low precision commonly found in Machine Learning (ML) [18], [19], [22], [34]). We compare SMaT's performance against existing solutions that can be employed in such a scenario, exposing their weaknesses on a large set of both real-world and synthetic matrices of varying size, structure, and sparsity. Furthermore, we showcase the effectiveness of the preprocessing permutation of sparse matrices.

We also test different solutions for our preprocessing step to minimize the number of blocks. Our experiments show, that Sylos Labini's algorithm (Section IV-C) performs the best for our test matrices. We evaluate two variants: the original version proposed by the authors that permutes only the rows, and our experiment on both row and column permutation.

A. Comparison Targets

As comparison targets, we use Magicube, which is a high-performance sparse-matrix library for low-precision integers on Tensor cores optimized for ML scenarios. For a fair comparison, we evaluate the Magicube[2] mixed precision int16 since the throughput is equal to fp16 on TC. Although DASP[3] focuses on SpMV, we consider it as a batched vector algorithm by iteratively performing SpMV — for this reason, we focus mostly on tall-and-skinny dense matrices. Finally, we compare against a cuSPARSE implementation of SpMM using the CSR format[4].

B. Hardware Infrastructure

We run our experiments on the Swiss National Computing Center's Ault compute cluster. Each node is equipped with a single NVIDIA A100-SXM4-40GB GPU, and AMD EPYC 7742 @ 2.25GHz CPU. The A100 driver version is 530.30.02.

C. Software Stack

All experiments were executed using the GCC 12.3.0 compiler, NVIDIA nvcc v12.0, NVIDIA cuSPARSE v12.0, NVIDIA CUDA Toolkit v12.0, Python 3.9, and the following Python libraries: Pandas, Matplotlib, Numpy, Scipy, and Seaborn.

[2]https://github.com/Shigangli/Magicube

[3]https://github.com/SuperScientificSoftwareLaboratory/DASP

[4]https://github.com/NVIDIA/CUDALibrarySamples/blob/master/cuSPARSE

Domain	Name	Size	*nnz*	Sparsity
optimization	mip1	66K×66K	10.4M	99.76%
quantum chem.	conf5_4-8x8	49K×49K	1.9M	99.92%
2D/3D mesh	cant	62K×62K	4M	99.89%
weighted graph	pdb1HYS	36K×36K	4.3M	99.67%
fluid dynamics	rma10	46.8K×46.8K	2.3M	99.89%
2D/3D mesh	cop20k_A	121K×121K	2.6M	99.98%
2D/3D mesh	consph	83K×83K	6M	99.91%
structural	shipsec1	140K×140K	7.8M	99.96%
circuit simulation	dc2	116K×116K	766K	99.99%

TABLE I: Selected matrices from the SuiteSparse

D. *Tested Scenarios*

As mentioned in Section V-A, neither DASP nor Magicube are optimized for large SpMM. Magicube, which is optimized for relatively small matrices found in ML, has a large memory footprint. This significantly limits the number of matrices we can run all libraries on. We consider two different origins of matrices:

- **SuiteSparse Collection** [35]: to make a fair comparison against DASP, we use the same representative set of matrices as used by Lu et al. [19]. These matrices come from 7 application domains, and are considered a broad representation of different types of sparse matrices. Out of 21 matrices used by DASP, Magicube can support only 9 of them (Table I).
- **Synthetic band matrices**: we generate a series of band matrices of variable bandwidth, scaling their sparsity from 99.7% all the way to dense matrices (0% sparsity), These matrices serve as the empirical evidence for the claims made in Section III. ***Motivation:*** Testing general-purpose unstructured SpMV/SpMM routines on highly regular matrices is common among the HPC benchmarks. HPCG [36] – one of the main benchmarks to rank supercomputers – tests this scenario: high performance SpMV on a matrix originating from a 3D grid computation.

E. *Methodology*

We first execute a warm-up run for all the experiments to obtain reliable measurements of the execution time on the GPU. Afterward, for each method, we run the kernel call 10 times and report the arithmetic mean. We observe that the variance in time measurements is very low: across all experiments, the Coefficient of Variation $CV = \sigma/\mu = 0.0182$ (geometric mean across all the runs). For example, SMaT wallclock time on `cop20k_A` is 0.125ms, and the variance is 1ns. Therefore, when plotting the measurements, we do **not include the confidence intervals** nor the standard deviation, as it would be unreadably small and would only clutter the display.

VI. Results

A. *Preprocessing reordering*

We start by analyzing the impact of reordering matrices on computation time for representative matrices from the SuiteSparse Collection. Figure 4 demonstrates the importance of reordering for SMaT. As the performance positively correlates to the number of blocks in the reordering, this confirms the performance model introduced in Section III. We observe that some input matrices are already well-structured: e.g., `conf5_4-8x8`, which originates from the quantum chemistry simulations, is a sparse band matrix – the locality of interactions between particles put all nonzeros relatively close to the diagonal. In this case, the Jaccard's similarity metric used by our preprocessor incorrectly reshuffles the rows, *increasing* the number of blocks. We evaluate the same setting for DASP (Figure 5), cuSPARSE (Figure 7), and Magicube (Figure 6). In general, the reordering decreased the number of BCSR blocks in 6 our of 9 matrices, with the block count reduction ranging from $1.3\times$ (`cant`) up to $2.4\times$ (`cop20k_A`). This significantly translates to the final performance solution, as discussed in Section VI-B, confirming the importance of this preprocessing step. The distribution of the number of blocks per row in BCSR for the input matrices, after the row permutation, and after row and column permutation for all tested matrices can be found in Figure 3.

B. *SuiteSparse*

As stated in Section V-D, we measure the performance in GFLOP/s on 9 matrices with the non-zero count ranging from $766k$ to $10.4M$ and the number or rows ranging from $M = 36k$ to $M = 140k$. While SMaT supports much bigger matrices as well, Magicube is designed to optimize ML workloads, which host much smaller matrices. In our experiments Magicube's internal preprocessing and representation runs out of memory for larger matrices from SuiteSparse. Therefore, we include only matrices that are supported by all comparison targets. Furthermore, since we are using batched SpMV to simulate SpMM for DASP, we keep the batch dimension small ($N = 8$): large values of N would make a batched-SpMV less competitive compared to the direct SpMM routines, as discussed in Section VI-D.

The dimensions and number of non-zero elements of the benchmark matrices are shown in Table I. Across 9 matrices SMaT is better on average $7.71\times$ (geometric mean) than the respective baselines. Compared to baselines SMaT is: $2.60\times$ faster (up to $7.34\times$) than DASP, $10.78\times$ faster (up to $51.23\times$) than Magciube, $16.32\times$ faster (up to $125.48\times$) than cuSPARSE. The results are presented in Figure 8.

Best case scenario. SMaT has the largest performance improvement compared to baselines on the `mip1` matrix. This use-case emphasizes the importance of good preprocessing that simultaneously minimizes the number of blocks *and* the row load-imbalance. As discussed in Section IV-C, our preprocessing routine is especially effective for `mip1`, reducing the total number of blocks by 1.8 times, but more importantly, the standard deviation of block count per row from 146.2 to 17.4. This improves the load balance by 8.4 times (Figure 3), which is crucial for an efficient parallel schedule. Interestingly, other libraries seem not to be able to take full advantage of this property (Figures 5, 6, 7). Since they also employ their internal preprocessing algorithms, we expect that in this use-

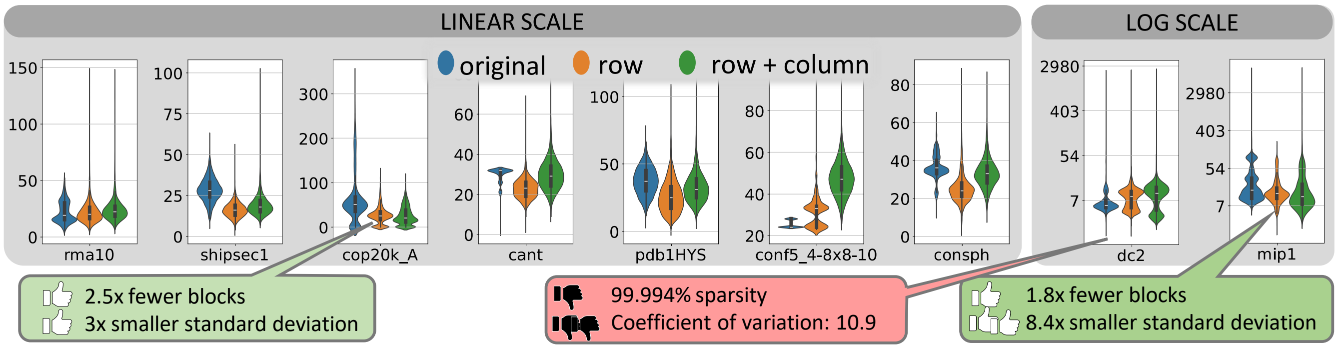


Fig. 3: Distribution of the blocks count per row in the BCSR format in the input matrix (original), after the row reordering, and after row and column reordering for the test matrices. For `cop20k_A`, row reordering reduces the number of BCSR blocks by 2.5x and the standard deviation by 3x. For `mip1`, while the reduction of the total block count is slightly smaller (1.8x), the standard deviation is reduced by 8.4x, significantly improving the load balance for our 2D parallel schedule. Matrix `dc2` is the most adversarial for SMaT: with its extreme sparsity and power-law distribution of nonzeros per row, the runtime cannot utilize tensor cores, and the warp-level static schedule generates high load imbalance on SMs.

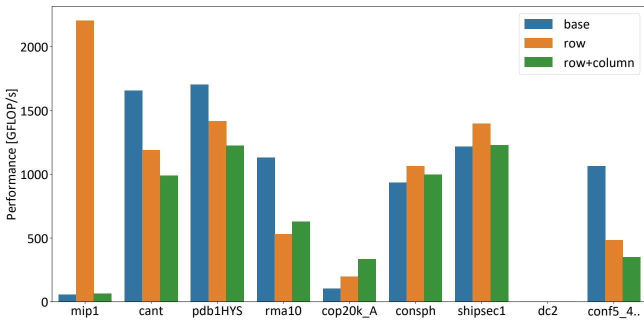


Fig. 4: Reordering effect on the performance of SMaT on 9 representative matrices from SuiteSparse.

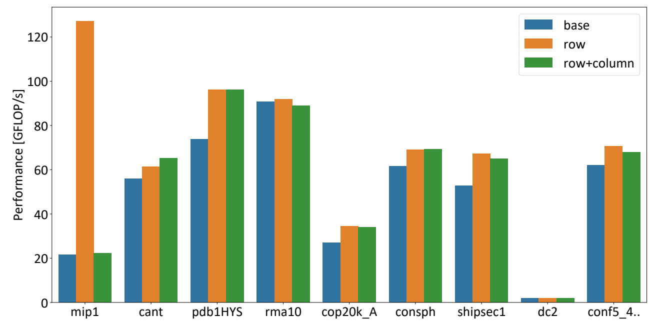


Fig. 6: Reordering effect on the performance of Magicube on 9 representative matrices from SuiteSparse.

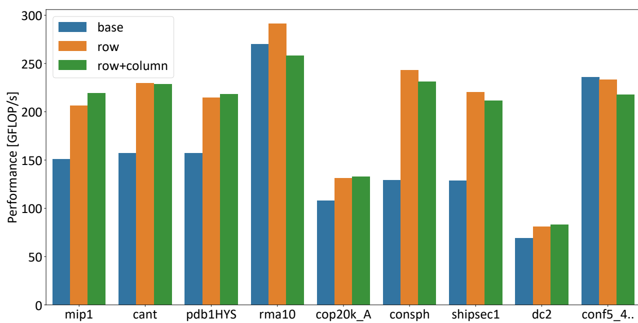


Fig. 5: Reordering effect on the performance of DASP on 9 representative matrices from SuiteSparse.

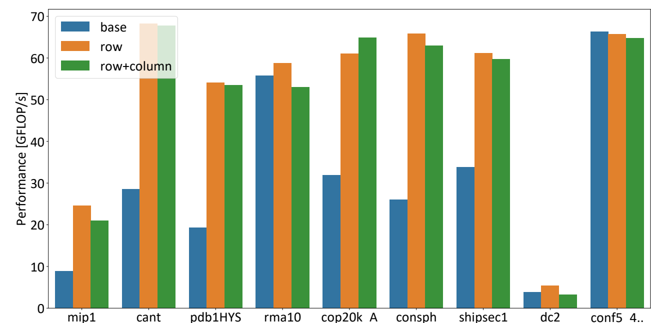


Fig. 7: Reordering effect on the performance of cuSPARSE on 9 representative matrices from SuiteSparse.

case their preprocessor actually *increases* the load imbalance. However, a detailed analysis of this aspect is out of scope of this work.

Worst case scenario. We observe that SMaT does not always perform the best. `dc2` is the sparsest among tested matrices (sparsity 99.994%) with a very high row imbalance: the standard deviation of the number of blocks per row is 169.9, while the arithmetic mean is 15.65. This makes `dc2` especially

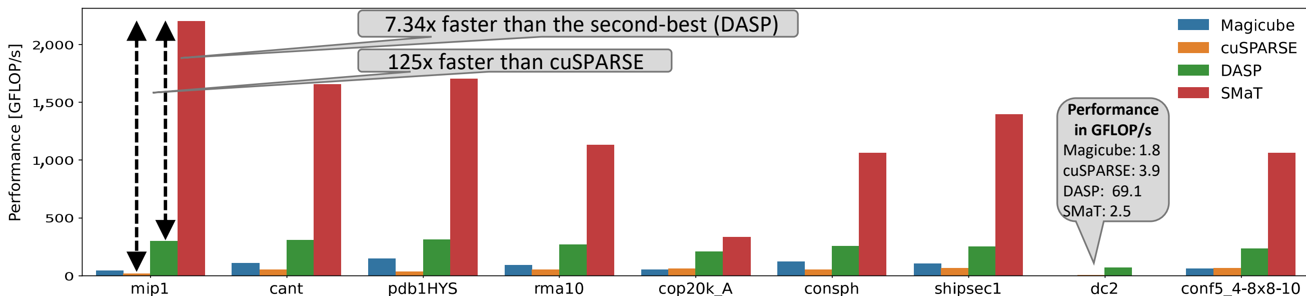


Fig. 8: Performance comparison on 9 representative matrices from SuiteSparse.

ill-suited for SMaT's execution model with static 2D parallel schedule, with most of the blocks containing only a single non-zero value, heavily underutilizing tensor cores, achieving only 2.5 GFLOP/s. This is the use-case where either DASP row-packing algorithm benefits most (69.1 GFLOP/s) or even a non-blocked CSR used in cuSPARSE (3.9 GFLOP/s).

C. Synthetic Matrices

Formally, an $n \times n$ matrix $\mathbf{A} = (a_{i,j})$ is a band matrix if all elements are zero outside a diagonally bordered band of width b: $a_{i,j} = 0 \quad \text{if} \quad j < i - b \quad \text{or} \quad j > i + b; \quad b \geq 0$. We evaluate performance on band matrices to measure the dependence of SMaT on the number of blocks n_e while isolating the randomness of the sparse matrix structure. Furthermore, for band matrices, blocks in BCSR format are already dense, so no further reordering is necessary. Hence, we do not incorporate the effect of the reordering while evaluating the kernel.

We perform our tests on matrix A of size $16,384 \times 16,384$ with varying the bandwidth from $b = 64$ all the way up to $b = 16,384$, effectively making the matrix dense. This allows us to measure a very important performance factor in sparse computations: **at what sparsity threshold a sparse library can outperform a highly-optimized dense library, if the matrix is explicitly padded with zeros?** Previous experiments [16] suggest that this can be as high as 99%. While a band matrix is definitely *not* a fair representation of an unstructured sparse matrix, we show that in this artificial scenario, SMaT can be competitive with cuBLAS.

Note: We measure the performance of cuBLAS only once - for a dense matrix $16,384 \times 16,384$. We then report cuBLAS performance as the *effective* FLOP/s, that is, we scale it by the fraction of nonzeros.

We first measure the performance of $C = AB$ with the number of columns $N = 8$ and present the results in Figure 9a. SMaT is up to $1,724\times$ faster than cuSPARSE and is at minimum $7\times$ faster than the second best method DASP. We notice that SMaT outperforms cuBLAS for sparsity $\geq 78\%$ and is only $2.3\times$ slower than cuBLAS in the dense case.

For $N = 128$, the difference between SMaT and the baselines rises (Figure 9b). As N grows, cuBLAS performance increases and reaches closer to the hardware peak in the dense setting. Nevertheless, SMaT performs better than cuBLAS for sparsity $\geq 96\%$. Compared with other methods, SMaT is at minimum $5.3\times$ faster than the second best method, Magicube, and gets up to $2,445\times$ faster than cuSPARSE.

D. Scaling Matrix Dimensions

Figure 10 demonstrates the relationship between the compute time and the outer dimension N of the dense matrix $\mathbf{B}$. The performance evaluation is conducted on the sparse matrix *cop20K_A* from Table I.

As N grows, SMaT exhibits the best performance among all the baselines. While DASP and cuSPARSE exhibit a degradation in performance as N increases, Magicube, similar to SMaT, shows a slow increase. Although DASP remains the fastest for $N = 1$, i.e., for SpMV.

For larger dimensions, such as $N = 1,000$, SMaT outperforms the baseline methods in terms of execution time. Specifically, SMaT takes 6.98ms to execute, while Magicube takes 12.13ms, DASP takes 29.70ms, and cuSPARSE takes 60.07ms. This means that SMaT is $1.73\times$, $4.24\times$, and $8.60\times$ faster than the baseline methods, respectively.

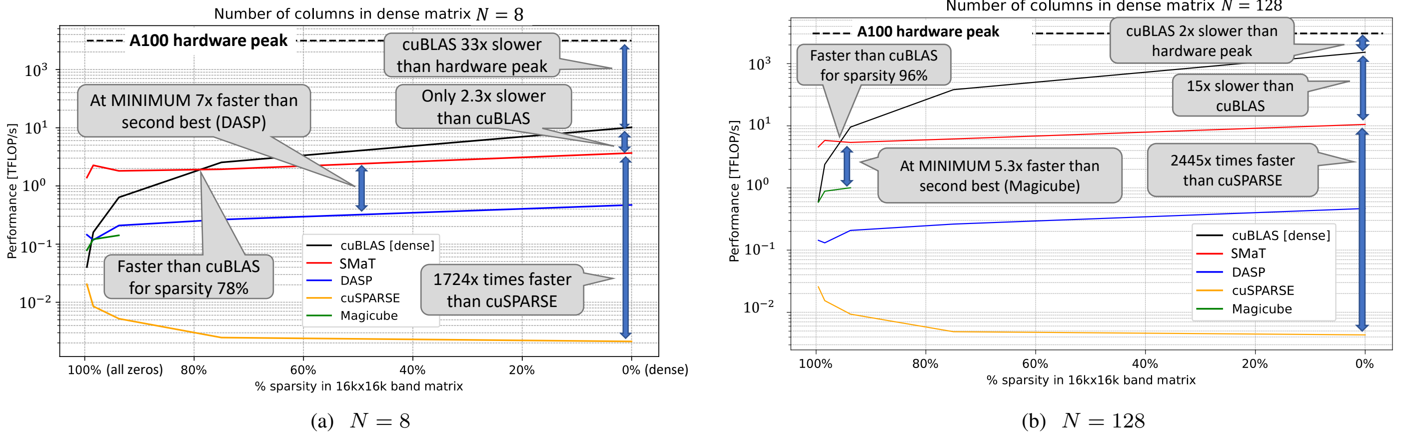
E. Distribution of the Number of Blocks per Row

The reordering experiments reveal an important property of our 2D parallel BCSR schedule. By using a fixed parallel grid and assigning one block of the output matrix per warp (Figure 1), SMaT's schedule is sensitive to the highly skewed distribution of blocks per row — some warps can have significantly more non-zero blocks to process than the others. This can be viewed, for example, for matrix `cant`: while the reordering decreases the mean number of blocks per row from 29.6 to 22.8, it *increases* its standard deviation from 4 to 5.2 (Figure 3). This results in the actual decrease in SMaT's performance (Figure 4). In contrast, the remaining libraries benefit from the reduced block count despite the increased load imbalance (Figures 5, 6, and 7).

F. Results Summary

Our results consistently show significant improvement over existing solutions. Several observations stand out from our experiments:

- DASP, which is a highly optimized SpMV library, often performs a single SpMV *slower* than SMaT performs a tall-and-skinny SpMM with $N = 8$,
- for relatively dense matrices, cuSPARSE performance is low (up to 2,445 times lower than SMaT). We also



(a) $N = 8$

(b) $N = 128$

Fig. 9: Measured performance of multiplying a synthetic band matrix $16k \times 16k$ with dense matrix $16k \times N$. Bandwidth b varies from $b = 64$ to $b = 16k$. Corresponding sparsity ranges from 99.7% to 0% (fully dense matrix).

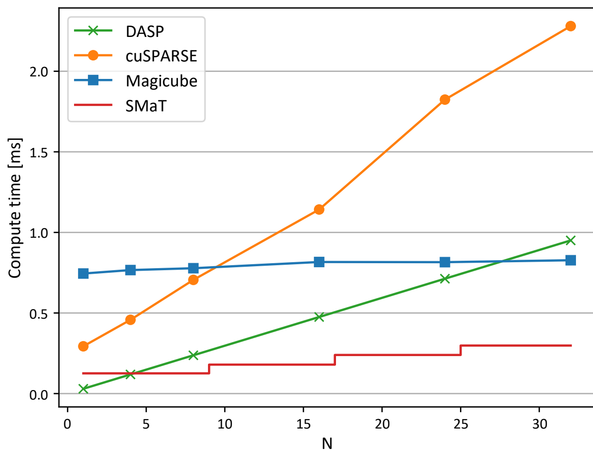


Fig. 10: Wall-clock time of SpMM $AB = C$, where A is the sparse matrix *cop20k_A* and B is a tall-and-skinny dense matrix with varying number of columns N.

observe that its performance *drops* for denser matrices (Figure 9),

- Magicube is a highly-specialized library for ML: while being the second-fastest in the band matrix experiment (Figure 9b), it quickly runs out of memory for larger values of N, making it unsuitable for general-purpose SpMM kernels,
- Sparse matrix column permutation preprocessing does not significantly reduce the number of blocks in BCSR format.

VII. Related Work

GEMM Dense matrix multiplication is an active field of research for decades. Cannon's algorithm [37] was the first distributed 2D parallel algorithm for square matrices. van de Geijn et al. extended it to non-square matrices in the SUMMA algorithm [38]. Agarwal et al. presented the distributed 3D algorithm that parallelizes also the reduction dimension [39]. Kwasniewski et al. introduced COSMA [23] – a communication optimal 2.5D parallel GEMM library for both CPUs and GPUs. Research in this area varies depending on the hardware platforms for which it is designed (GPU or CPU), the type of sparsity it addresses (structured or unstructured), and the precision levels it targets, including formats such as fp16, fp32, fp64, int8, and other precision schemes.

Tensor Cores In addition to GEMM and its applications such as machine learning [22], Tensor cores are successfully employed to enhance basic operators such as scan and reduction [40], stencil computation [41], and FFT [42].

Reordering Reordering and blocking have been explored to accelerate parallel multiplication, particularly in SpMV [43] and SpMM [44]. Row-reordering is often used to promote efficient memory hierarchy usage by enhancing locality [45]. This involves moving rows with similar nonzero structures closer together to locally densify blocks and reduce cache misses during SpMV or SpMM [46].

SpMV One line of research focuses on multiplying sparse matrices with dense vectors (SpMV). Researchers have explored the trade-offs between balancing workload [47], [48], data locality [49], and format generation [50], [51]. SpMV overview studies can be found in [52].

SpMM Gao et al. presents an overview of the existing research on sparse matrix multiplication (SpGEMM) [53]. NVIDIA has developed cuSPARSE [20] and cuSPARSELt [54] for sparse matrix multiplication. cuSPARSE is capable of performing unstructured SpMM for sparsity above 95% on CUDA Cores, while cuSPARSELt uses Tensor Cores and utilizes 2:4 structured sparsity. Gale et al. design Sputnik [55], a GPU kernel to accelerate sparse matrix operations in neural networks. Chen et al. present vectorSparse [56], proposing column-vector-sparse-

encoding for SpMM. Li et al. develop Magicube [18], a high-performance library for low-precision integers on Tensor Cores focusing on deep learning as well. Furthermore, Li et al. propose the SR-BCRS format, based on the CS format, which is suitable for Tensor Cores. While Magicube is for structured sparse formats, Xue et al [57]. introduce a method for unstructured sparsity in fp16 precision.

VIII. Conclusions

This paper introduces SMaT, a novel general-purpose SpMM library that utilizes Tensor Core hardware units to accelerate Sparse Matrix-Matrix Multiplication (SpMM). SMaT first employs modern state-of-the-art preprocessing techniques to significantly reduce the number of blocks in BCSR format, as well as improve the load balance for the bottom-up 2D parallel schedule. The optimized CUDA implementation uses low-level API to fully utilize Tensor Cores, asynchronous shared memory loads, and warp-level memory alignment. We develop the empirical performance model that quantifies the benefit of both the preprocessing and the kernel optimizations.

We perform a comprehensive performance study on both real-world sparse matrices from SuiteSparse as well as synthetic matrices. Evaluated on NVIDIA Ampere GPUs, we show the supremacy of SMaT compared to state-of-the-art libraries: Magicube, DASP, and cuSPARSE. We measure up to 7.3x speedup vs the second-best library (2.6x on average), with up to 125x speedup over vendor-optimized cuSPARSE.

Furthermore, while scaling the sparsity ratio on the synthetic matrices for tall-and-skinny SpMM, we outperform cuBLAS for sparsity as low as 78%, outperforming cuSPARSE by up to 2,445 times. The presented results confirm that SpMM libraries do not need to be highly specialized to narrow use cases for fixed data types, sparsity patterns, and matrix sizes.

Acknowledgments

This research is carried out in the frame of the “UrbanTwin: An urban digital twin for climate action, assessing policies and solutions for energy, water and infrastructure” project with the financial support of the ETH-Domain Joint Initiative program in the Strategic Area Energy, Climate and Sustainable Environment. This work received EuroHPC-JU funding with support from the European Union’s Horizon 2020 program and the European Research Council under grant agreement PSAP, number 101002047, as well as from the European Union – NextGenerationEU project. We also wish to acknowledge the support from the DEEP-SEA project under grant agreement number 955606. This research is partially funded by the European Union – NextGenerationEU. The authors would like to thank the Swiss National Supercomputing Centre (CSCS) for access and support of the computational resources.

References

[1] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers *et al.*, “In-datacenter performance analysis of a tensor processing unit,” in *Proceedings of the 44th Annual International Symposium on Computer Architecture*. ACM, 2017, pp. 1–12.

[2] H. Abdelkhalik, Y. Arafa, N. Santhi, and A.-H. A. Badawy, “Demystifying the nvidia ampere architecture through microbenchmarking and instruction-level analysis,” in *2022 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 2022, pp. 1–8.

[3] Z. Bai, J. Demmel, J. Dongarra, A. Ruhe, and H. van der Vorst, *Templates for the solution of algebraic eigenvalue problems: a practical guide*. SIAM, 2000.

[4] J. R. Gilbert, S. Reinhardt, and V. B. Shah, “A unified framework for numerical and combinatorial computing,” *Computing in Science & Engineering*, vol. 10, no. 2, pp. 20–25, 2008.

[5] M. Besta, F. Marending, E. Solomonik, and T. Hoefler, “Slimsell: A vectorizable graph representation for breadth-first search,” in *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2017, pp. 32–41.

[6] M. Besta, M. Podstawski, L. Groner, E. Solomonik, and T. Hoefler, “To push or to pull: On reducing communication and synchronization in graph computations,” in *Proceedings of the 26th International Symposium on High-Performance Parallel and Distributed Computing (HPDC)*, 2017, pp. 93–104.

[7] E. Solomonik, M. Besta, F. Vella, and T. Hoefler, “Scaling betweenness centrality using communication-efficient sparse matrix multiplication,” in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (ACM/IEEE Supercomputing)*, 2017, pp. 1–14.

[8] M. Besta, D. Stanojevic, J. D. F. Licht, T. Ben-Nun, and T. Hoefler, “Graph processing on fpgas: Taxonomy, survey, challenges,” *arXiv preprint arXiv:1903.06697*, 2019.

[9] Z. Wu, S. Pan, F. Chen, G. Long, C. Zhang, and S. Y. Philip, “A comprehensive survey on graph neural networks,” *IEEE transactions on neural networks and learning systems*, vol. 32, no. 1, pp. 4–24, 2020.

[10] S. Abadal, A. Jain, R. Guirado, J. López-Alonso, and E. Alarcón, “Computing graph neural networks: A survey from algorithms to accelerators,” *ACM Comput. Surv.*, vol. 54, no. 9, oct 2021. [Online]. Available: https://doi.org/10.1145/3477141

[11] M. Besta and T. Hoefler, “Parallel and distributed graph neural networks: An in-depth concurrency analysis,” *IEEE Transactions on Pattern Analysis and Machine Intelligence (TPAMI)*, 2023.

[12] J. Bazinska, A. Ivanov, T. Ben-Nun, N. Dryden, M. Besta, S. Shen, and T. Hoefler, “Cached operator reordering: A unified view for fast gnn training,” *arXiv preprint arXiv:2308.12093*, 2023.

[13] M. Besta, P. Renc, R. Gerstenberger, P. Sylos Labini, T. Chen, L. Gianinazzi, F. Scheidl, K. Szenes, A. Carigiet, P. Iff, G. Kwasniewski, R. Kanakagiri, C. Ge, S. Jaeger, J. Was, F. Vella, and T. Hoefler, “High-performance and programmable attentional graph neural networks with global tensor formulations,” in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (ACM/IEEE Supercomputing)*, 2023.

[14] T. Hoefler, D. Alistarh, T. Ben-Nun, N. Dryden, and A. Peste, “Sparsity in deep learning: Pruning and growth for efficient inference and training in neural networks,” *The Journal of Machine Learning Research*, vol. 22, no. 1, pp. 10 882–11 005, 2021.

[15] B. Liu, M. Wang, H. Foroosh, M. Tappen, and M. Penksy, “Sparse convolutional neural networks,” in *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2015, pp. 806–814.

[16] Z. Wang, “Sparsert: Accelerating unstructured sparsity on gpus for deep learning inference,” in *Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT ’20. New York, NY, USA: Association for Computing Machinery, 2020, p. 31–42. [Online]. Available: https://doi.org/10.1145/3410463.3414654

[17] S. Filippone, V. Cardellini, D. Barbieri, and A. Fanfarillo, “Sparse matrix-vector multiplication on gpgpus,” *ACM Trans. Math. Softw.*, vol. 43, no. 4, Jan. 2017.

[18] S. Li, K. Osawa, and T. Hoefler, “Efficient quantized sparse matrix operations on tensor cores,” ser. SC ’22. IEEE Press, 2022.

[19] Y. Lu and W. Liu, “Dasp: Specific dense matrix multiply-accumulate units accelerated general sparse matrix-vector multiplication,” in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2023, pp. 1–14.

[20] M. Naumov, L. Chien, P. Vandermersch, and U. Kapasi, “Cusparse library,” in *GPU Technology Conference*, 2010.

[21] Z. Chen, Z. Qu, L. Liu, Y. Ding, and Y. Xie, “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*, 2021, pp. 1–14.

[22] R. L. Castro, A. Ivanov, D. Andrade, T. Ben-Nun, B. B. Fraguela, and T. Hoefler, "Venom: A vectorized n: M format for unleashing the power of sparse tensor cores," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2023, pp. 1–14.

[23] G. Kwasniewski, M. Kabić, M. Besta, J. VandeVondele, R. Solcà, and T. Hoefler, "Red-blue pebbling revisited: near optimal parallel matrix-matrix multiplication," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2019, pp. 1–22.

[24] T. M. Aamodt, W. W. L. Fung, T. G. Rogers, and M. Martonosi, *General-purpose graphics processor architectures*. Springer, 2018.

[25] X. Yu, T. Liu, X. Wang, and D. Tao, "On compressing deep models by low rank and sparse decomposition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2017, pp. 7370–7379.

[26] M. Besta, Z. Vonarburg-Shmaria, Y. Schaffner, L. Schwarz, G. Kwasniewski, L. Gianinazzi, J. Beranek, K. Janda, T. Holenstein, S. Leisinger *et al.*, "Graphminesuite: Enabling high-performance and programmable graph mining algorithms with set algebra," *arXiv preprint arXiv:2103.03653*, 2021.

[27] R. Eberhardt and M. Hoemmen, "Optimization of block sparse matrix-vector multiplication on shared-memory parallel architectures," in *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 2016, pp. 663–672.

[28] "How to optimize a cuda matmul kernel for cublas-like performance: a worklog," https://siboehm.com/articles/22/CUDA-MMM, accessed: 2024-03-28.

[29] W.-H. Liu and A. H. Sherman, "Comparative analysis of the cuthill–mckee and the reverse cuthill–mckee ordering algorithms for sparse matrices," *SIAM Journal on Numerical Analysis*, vol. 13, no. 2, pp. 198–213, 1976. [Online]. Available: https://doi.org/10.1137/0713020

[30] Y. Saad, "Finding exact and approximate block structures for ilu preconditioning," *SIAM Journal on Scientific Computing*, vol. 24, 09 2001.

[31] U. Çatalyürek, K. Devine, M. Faraj, L. Gottesbüren, T. Heuer, H. Meyerhenke, P. Sanders, S. Schlag, C. Schulz, D. Seemaier, and D. Wagner, "More recent advances in (hyper)graph partitioning," *ACM Comput. Surv.*, vol. 55, no. 12, mar 2023. [Online]. Available: https://doi.org/10.1145/3571808

[32] H. Zhao, T. Xia, C. Li, W. Zhao, N. Zheng, and P. Ren, "Exploring better speculation and data locality in sparse matrix-vector multiplication on intel xeon," in *2020 IEEE 38th International Conference on Computer Design (ICCD)*, 2020, pp. 601–609.

[33] P. S. Labini, M. Bernaschi, W. Nutt, F. Silvestri, and F. Vella, "Blocking sparse matrices to leverage dense-specific multiplication," in *2022 IEEE/ACM Workshop on Irregular Applications: Architectures and Algorithms (IA3)*, 2022, pp. 19–24.

[34] D. Kim and J. Kim, "Analysis of several sparse formats for matrices used in sparse-matrix dense-matrix multiplication for machine learning on gpus," in *2022 13th International Conference on Information and Communication Technology Convergence (ICTC)*. IEEE, 2022, pp. 629–631.

[35] S. P. Kolodziej, M. Aznaveh, M. Bullock, J. David, T. A. Davis, M. Henderson, Y. Hu, and R. Sandstrom, "The suitesparse matrix collection website interface," *Journal of Open Source Software*, vol. 4, no. 35, p. 1244, 2019. [Online]. Available: https://doi.org/10.21105/joss.01244

[36] P. R. Luszczek, D. H. Bailey, J. J. Dongarra, J. Kepner, R. F. Lucas, R. Rabenseifner, and D. Takahashi, "The hpc challenge (hpcc) benchmark suite," in *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, vol. 213, no. 10.1145, 2006, p. 1.

[37] L. E. Cannon, *A cellular computer to implement the Kalman filter algorithm*. Montana State University, 1969.

[38] R. A. van de Geijn and J. Watts, "SUMMA: Scalable universal matrix multiplication algorithm," *Concurrency: Practice and Experience*, vol. 9, no. 4, pp. 255–274, 1997.

[39] R. C. Agarwal, S. M. Balle, F. G. Gustavson, M. Joshi, and P. Palkar, "A three-dimensional approach to parallel matrix multiplication," *IBM Journal of Research and Development*, vol. 39, no. 5, pp. 575–582, 1995.

[40] A. Dakkak, C. Li, J. Xiong, I. Gelado, and W.-m. Hwu, "Accelerating reduction and scan using tensor core units," in *Proceedings of the ACM International Conference on Supercomputing*, ser. ICS '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 46–57. [Online]. Available: https://doi.org/10.1145/3330345.3331057

[41] X. Liu, Y. Liu, H. Yang, J. Liao, M. Li, Z. Luan, and D. Qian, "Toward accelerated stencil computation by adapting tensor core unit on gpu," in *Proceedings of the 36th ACM International Conference on Supercomputing*, ser. ICS '22. New York, NY, USA: Association for Computing Machinery, 2022. [Online]. Available: https://doi.org/10.1145/3524059.3532392

[42] B. Li, S. Cheng, and J. Lin, "tcfft: A fast half-precision fft library for nvidia tensor cores," in *2021 IEEE International Conference on Cluster Computing (CLUSTER)*, 2021, pp. 1–11.

[43] E. Cuthill, *Several Strategies for Reducing the Bandwidth of Matrices*. Boston, MA: Springer US, 1972, pp. 157–166. [Online]. Available: https://doi.org/10.1007/978-1-4615-8675-3_14

[44] J. Pichel, D. Heras, J. Cabaleiro, and F. Rivera, "Performance optimization of irregular codes based on the combination of reordering and blocking techniques," *Parallel Computing*, vol. 31, no. 8, pp. 858–876, 2005. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167819105000803

[45] P. S. Labini, M. Bernaschi, W. Nutt, F. Silvestri, and F. Vella, "Blocking sparse matrices to leverage dense-specific multiplication," in *2022 IEEE/ACM Workshop on Irregular Applications: Architectures and Algorithms (IA3)*, 2022, pp. 19–24.

[46] K. Lakhotia, S. Singapura, R. Kannan, and V. Prasanna, "Recall: Reordered cache aware locality based graph processing," in *2017 IEEE 24th International Conference on High Performance Computing (HiPC)*, 2017, pp. 273–282.

[47] J. Aliaga, H. Anzt, T. Grützmacher, E. S. Quintana-Ortí, and A. Tomás, "Compression and load balancing for efficient sparse matrix-vector product on multicore processors and graphics processing units," *Concurrency and Computation: Practice and Experience*, vol. 34, 07 2021.

[48] A. Ashari, N. Sedaghati, J. Eisenlohr, S. Parthasarath, and P. Sadayappan, "Fast sparse matrix-vector multiplication on gpus for graph applications," in *SC '14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2014, pp. 781–792.

[49] C. Alappat, A. Basermann, A. R. Bishop, H. Fehske, G. Hager, O. Schenk, J. Thies, and G. Wellein, "A recursive algebraic coloring technique for hardware-efficient symmetric sparse matrix-vector multiplication," *ACM Trans. Parallel Comput.*, vol. 7, no. 3, jun 2020. [Online]. Available: https://doi.org/10.1145/3399732

[50] A. Benatia, W. Ji, Y. Wang, and F. Shi, "Sparse matrix format selection with multiclass svm for spmv on gpu," in *2016 45th International Conference on Parallel Processing (ICPP)*, 2016, pp. 496–505.

[51] Z. Du, J. Li, Y. Wang, X. Li, G. Tan, and N. Sun, "Alphasparse: Generating high performance spmv codes directly from sparse matrices," 2022.

[52] G. Goumas, K. Kourtis, N. Anastopoulos, V. Karakasis, and N. Koziris, "Performance evaluation of the sparse matrix-vector multiplication on modern architectures," *The Journal of Supercomputing*, vol. 50, pp. 36–77, 2009.

[53] J. Gao, W. Ji, F. Chang, S. Han, B. Wei, Z. Liu, and Y. Wang, "A systematic survey of general sparse matrix-matrix multiplication," *ACM Comput. Surv.*, vol. 55, no. 12, mar 2023. [Online]. Available: https://doi.org/10.1145/3571157

[54] Nvidia, "cusparselt documentation," https://docs.nvidia.com/cuda/cusparselt, 2024.

[55] T. Gale, M. Zaharia, C. Young, and E. Elsen, "Sparse gpu kernels for deep learning," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '20. IEEE Press, 2020.

[56] Z. Chen, Z. Qu, L. Liu, Y. Ding, and Y. Xie, "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*, ser. SC '21. New York, NY, USA: Association for Computing Machinery, 2021. [Online]. Available: https://doi.org/10.1145/3458817.3476182

[57] Z. Xue, M. Wen, Z. Chen, Y. Shi, M. Tang, J. Yang, and Z. Luo, "Releasing the potential of tensor core for unstructured spmm using tiled-csr format," in *2023 IEEE 41st International Conference on Computer Design (ICCD)*. Los Alamitos, CA, USA: IEEE Computer Society, nov 2023, pp. 457–464. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/ICCD58817.2023.00076

Appendix: Artifact Description/Artifact Evaluation

Artifact Description (AD)

I. Overview of Contributions and Artifacts

A. Paper's Main Contributions

C_1 A sparse matrix preprocessing permutation scheme that decreases the number of dense blocks by up to 2.5x.

C_2 A high-performance implementation of the blocked-CSR (BCSR) SpMM on Tensor Cores using the low-level CUDA MMA API.

C_3 SMaT: an end-to-end solution for general-purpose SpMM that supports unstructured sparsity and all data types supported by the TC units.

B. Computational Artifacts

A_1 https://doi.org/10.5281/zenodo.13305901

Artifact ID	Contributions Supported	Related Paper Elements
A_1	C_1, C_2, C_3	Figure 2-10

II. Artifact Identification

A. Computational Artifact A_1

Relation To Contributions

Artifact A_1 represents general-purpose SpMM library that works on unstructured sparse matrices. The artifact implements the contributions stated in Chapter I-A.

Expected Results

Execution time comparison of SMaT with DASP, cuSPARSE, and Magicube. Furthermore:

- DASP often performs a single SpMV *slower* than SMaT performs a tall-and-skinny SpMM with $N = 8$,
- cuSPARSE performance is low for relatively dense matrices (up to 2,445 times lower than SMaT),
- Magicube while being the second-fastest in the band matrix experiment, quickly runs out of memory for larger values of N, making it unsuitable for general-purpose SpMM kernels,
- Sparse matrix column permutation preprocessing does not significantly reduce the number of blocks in BCSR format.

Expected Reproduction Time (in Minutes)

The total estimated time for reproducing artifact A_1 is 70 min. Compilation of artifact takes 2 min. Downloading selected matrices from the SuiteSparse Matrix Collection takes 10 min. Generating synthetic matrices runs for 25 min or more. Preprocessing takes 2-3 min per matrix depending on matrix size. Since we do not reorder synthetic band matrices, the total preprocessing for 9 matrices is 25 min or more. Finally, running SMaT multiple times on all matrices takes 5 min.

Artifact Setup (incl. Inputs)

Hardware: We run our experiments on the Swiss National Computing Center's Ault compute cluster. Each node is equipped with a single NVIDIA A100-SXM4-40GB GPU, and AMD EPYC 7742 @ 2.25GHz CPU. The A100 driver version is 530.30.02.

Software: All experiments were executed using the GCC 12.3.0 compiler, NVIDIA nvcc v12.0, NVIDIA cuSPARSE v12.0, NVIDIA CUDA Toolkit v12.0, Python 3.9, and the following Python libraries: Pandas, Matplotlib, Numpy, Scipy, and Seaborn.

Datasets / Inputs: We consider two different origins of matrices:

- **SuiteSparse Collection**: These matrices come from 7 application domains, and are considered a broad representation of different types of sparse matrices. The matrices can be downloaded at: `http://sparse.tamu.edu/`. In the artifact we provide a script for downloading that can be called using:
```
$ python download_suitesparse.py
```
- **Synthetic band matrices**: we generate a series of band matrices of variable bandwidth, scaling their sparsity from 99.7% all the way to dense matrices (0% sparsity). To generate the necessary matrices one needs to execute:
```
$ python generate_matrices.py
```

Installation and Deployment: We provide a file with all the requirements for a `conda` environment. To install and use the environment:

```
$ conda env create -f smat_env.yml
$ sudo apt-get install libgflags-dev
$ conda activate smat
```

Artifact Execution

We identify the following tasks:

T_1 download SuiteSparse matrices

T_2 generate synthetic band matrices

T_3 preprocess the matrices

T_4 run SMaT, DASP, cuSPARSE, and Magicube

First, we prepare the input matrices by executing tasks T_1 and T_2, which can be run in parallel. Afterward, we need to preprocess and reorder the matrices. Finally, we can perform SpMM using SMaT on the processed matrices. Hence, task dependencies are the following: $T_1, T_2 \to T_3 \to T_4$.

Artifact Analysis (incl. Outputs)

We provide a Jupyter notebook for recreating all the figures in the paper. After the results have been saved it is necessary to change the path pointing in the notebook and run all notebook elements in *plotting.ipynb*.

Artifact Evaluation (AE)

For reproducing the results we use the Chameleon node with A100_NVLink 844736d7-5a65-47ce-a54c-fd82dea47069 (F1BG3Q3) launched with CC-Ubuntu20.04-CUDA image.

A. Computational Artifact A_1

Artifact Setup (incl. Inputs)

Installing and preparing our library is explained in Section II-A. For compiling the code for SMaT run the following:

```
$ cd src/cuda_hgemm
$ source compile.sh
```

For comparison towards baselines, we use the exact same libraries as proposed by the authors. For using DASP and cuSPARSE, we follow the instructions from https://github.com/SuperScientificSoftwareLaboratory/DASP. After changing the directory to *baselines/DASP* compile the code with the following command:

```
$ make half
```

For Magicube we create a separate environment as recommended by the authors at https://github.com/Shigangli/Magicube. To achieve that, execute the following commands:

```
$ conda create --name py38_sc22 python=3.8
$ conda activate py38_sc22
$ pip install torch==1.9.1+cu111
torchvision==0.10.1+cu111
torchaudio==0.9.1 -f https://download.
pytorch.org/whl/torch_stable.html
$ pip install -r requirements.txt
$ cd baselines/Magicube/baselines
$ bash setup.sh
```

Artifact Execution

After downloading the dataset and creating synthetic band matrices as described in II-A, we perform reordering of all SuiteSparse matrices. Firstly, compile the library for reordering:

```
$ cd preprocess
$ make all
```

For further details check: https://github.com/HicrestLaboratory/SPARTA. Secondly, change the path to point to the location of SuiteSparse matrices inside *reorder.py*, and then execute the command:

```
$ python reorder.py
```

When the script finishes this will populate the same folder with reordered SuiteSparse matrices.

DASP & cuSPARSE After changing the directory to *baselines/DASP* and setting the path in the script to point to the location of matrices run:

```
$ source run_dasp_cusparse.sh
```

This will perform SpMM for all the matrices for DASP and cuSPARSE and save the results in file *results_dasp_cusparse.csv*

Magicube Activate the Magicube environment only for this with:

```
$ conda activate py38_sc22
```

Magicube expects the input matrices to be in a different format than the rest of the methods. First, change the directory to *baselines/Magicube/SpMM/SpMM*, set the path to the folder with matrices inside the script, and create the appropriate format for Magicube by executing:

```
$ python magicube_format.py
```

Second, set the path in the script *run_magicube.py* to point to the location of matrices that have the appropriate format for Magicube *path/magicubeFormat*:

```
$ python run_magicube.py
```

This will run all the experiments for Magicube and save the results in file *results_magicube.out*. Lastly, return to the original environment with:

```
$ conda activate smat
```

SMaT Change the directory to *src/cuda_hgemm*, set the path in the script to point to the location of matrices, and run:

```
$ source run_smat.sh
```

This will run all the experiments for SMaT and store the results in file *results_smat.csv*.

Artifact Analysis (incl. Outputs)

After the experiments are finished we provide a Jupyter Notebook *plotting.ipynb* with all the necessary code for reproducing the figures in the paper and analyzing the results.

Firstly, the notebook parses the output files containing all the results:

- *results_dasp_cusparse.csv*,
- *results_magicube.out*,
- *results_smat.csv*.

Secondly, after the results are parsed we provide sections for reproducing Figures 2-10.