ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Using Sparse Tensor Cores for Sparse Matrix–Matrix Multiplication

Bachelor Thesis

A. Mistry

April 30, 2025

Advisors: Prof. Dr. T. Hoefler, P. Okanovic, Dr. G. Kwasiniewski
Department of Computer Science, ETH Zürich

Abstract

This thesis presents an empirical investigation into the performance optimization of sparse matrix multiplication on NVIDIA GPUs using sparse Tensor Cores. While sparse Tensor Cores offer theoretical throughput improvements for matrices with 2:4 sparsity patterns, their effectiveness in real-world applications depends critically on matrix structure. This work implements and evaluates multiple optimization strategies including sparse Tensor Core utilization, pipelining, tiling, and the separation of dense and sparse processing. Through comprehensive benchmarking on both synthetic and real-world matrices from the SuiteSparse collection, we demonstrate that sparse Tensor Cores can achieve over 2× throughput improvement for compliant matrices. Our analysis reveals that while many real-world matrices fail to meet the stringent 2:4 sparsity requirements (e.g., the SuiteSparse `cant` matrix with only 7.8% qualifying blocks), others contain sufficient compliant blocks to benefit substantially from our optimizations. The novel `mmaOBTS_large_separate` kernel, which eliminates runtime branching through preprocessing-based segregation of dense and sparse tiles, successfully combines sparse Tensor Cores with pipelining optimizations and achieves the highest performance across all qualifying matrices. For real-world matrices with significant 2:4 sparse content (such as `mip1` with 33.8% compliance), our combined approach outperforms traditional methods, while pipelining alone remains the most robust optimization for matrices with limited 2:4 sparsity. This work demonstrates that sparse Tensor Cores, when properly integrated with complementary optimizations, provide meaningful acceleration for a subset of real-world applications, offering insights for future developments in GPU sparse computation.

Contents

Chapter 1

Introduction

The problem of matrix multiplication has gathered a large amount of attention in recent years due to the exponential growth of machine learning applications. While substantial research focus has been directed toward optimizing dense matrix multiplication, sparse matrix multiplication (SpMM) has emerged as an increasingly critical operation. This growing importance stems from advancements in sparse transformers, mixture of experts (MoE) models, and Graph Neural Networks (GNNs). Furthermore, sparse matrix multiplication has long been fundamental in scientific computing domains such as Finite Element Methods (FEM) and Computational Fluid Dynamics (CFD), as well as in data-intensive applications like Graph Analytics.

Building on the work of Okanovic et al. [1], we ask: can 2:4-sparse Tensor Cores, in combination with other optimizations, deliver $\geq$1.5× speed-up over the state-of-the-art dense-core BCSR SpMM on matrices with $\geq$30% sparse-compatible tiles? To answer this, we explore a broader range of matrix configurations and evaluate how effectively Sparse Tensor Cores can be leveraged to improve computational efficiency under realistic sparsity conditions.

While this work focuses exclusively on NVIDIA GPUs, AMD has introduced comparable hardware support for sparse matrix multiplication through its Sparse Matrix Fused Multiply-Accumulate (SMFMA) instructions in the CDNA architecture [2]. These instructions impose similar structural sparsity constraints—such as fixed 2:4 sparsity patterns—making the techniques presented in this thesis potentially applicable to AMD hardware as well. However, NVIDIA was chosen as the target platform due to its mature CUDA ecosystem, superior developer tooling, and broader adoption in scientific computing and machine learning workloads. This also enables direct comparison with prior libraries and research, which predominantly evaluate performance on NVIDIA GPUs.

1.1 The GPU

The GPU is designed for extreme parallelism, which is reflected in its hardware architecture (see Figure 1.1). While CPUs consist of a relatively limited number of sophisticated cores, GPUs are composed of numerous Streaming Multiprocessors (SMs) [3]. Contemporary deep learning GPUs, such as the NVIDIA A100 80GB used in the benchmarks for this work, feature 108 SMs [4]. Each SM incorporates various storage and compute units, including Arithmetic Logic Units (ALUs)—also referred to as CUDA cores—that support different numerical formats, and specialized Tensor Cores, which will be discussed in greater detail later [3]. The parallel processing capabilities of GPUs make them particularly well-suited for matrix multiplication operations, as these computations can be effectively distributed across the many processing elements. This architectural advantage enables GPUs to achieve substantially higher throughput for the computationally intensive workloads characteristic of modern machine learning and scientific computing applications.

The extreme throughput of the GPU makes efficient memory flow optimization critical to maintaining high utilization of its compute units. To achieve this, GPU programming frameworks like CUDA allow programmers significantly greater control over caching mechanisms compared to traditional CPU programming.

At the lowest level is the DRAM, also referred to as global memory in CUDA. This ranges from up to 24GB on consumer cards to as much as 128GB on modern NVIDIA deep learning GPUs. Global memory serves as the initial repository for data transferred from the host's main memory. The second level is the L2 cache, which is substantially smaller—typically by a factor of approximately 1000—but offers reduced access latency. The L1 cache, also known as shared memory, is directly integrated within each individual Streaming Multiprocessor (SM). While the L2 cache population occurs automatically through the GPU's memory hierarchy, shared memory must be explicitly managed by the programmer. Similar to CPU architecture, the lowest access latency is achieved through registers, which are stored in the register file on each SM. Table 1.2 is a summary of the A100 memory hierarchy and its characteristics.

This memory hierarchy creates opportunities for performance optimization through careful data movement strategies, allowing programmers to minimize high-latency global memory accesses and maximize the use of faster local storage. Effective management of this memory hierarchy is particularly crucial for memory-bound operations like sparse matrix multiplication, where computational performance is often limited by memory bandwidth rather than arithmetic throughput.

Figure 1.1: Architecture of an NVIDIA A100 SM [5]

Memory Type	Size	Latency	Location
Registers	256 KB per SM	1–2 cycles	SM
L1 Cache + shared mem.	192 KB per SM (combined)	30–60 cycles	SM
L2 Cache	40 MB	300-400 cycles	On-chip
HBM2 Memory	80 GB	400–1000 cycles	On-package

Figure 1.2: Memory hierarchy of an NVIDIA A100 80GB [4]

1.1.1 The GPU programming model

In CUDA, GPU programs are structured around a massive number of lightweight threads, organized in a hierarchical model. At the lowest level, each thread runs independently with its own registers and program counter. Threads are grouped into warps—typically 32 threads—which follow the Single Instruction, Multiple Threads (SIMT) execution model: all threads execute the same instructions in lockstep. These warps are further organized into thread blocks, also known as Cooperative Thread Arrays (CTAs). All warps of a thread block run on the same SM, so they all have access to the same shared memory, making it the primary mechanism for sharing memory between

threads. Threads within a thread block can synchronize with each other using barrier functions. Multiple blocks form a grid, which represents the full scope of a kernel's execution. The grid enables large-scale parallelism by allowing thousands or even millions of threads to work concurrently across the GPU. While threads within a block can cooperate via shared memory and synchronization, blocks in a grid execute independently, with no guaranteed order or direct communication—making them ideal for data-parallel tasks that can be decomposed into uniform, independent units of work.

1.1.2 Sparse Tensor Cores

As matrix multiplication has become a primary use case of GPUs, NVIDIA has introduced dedicated hardware for this problem starting with the Volta architecture in 2017. These Tensor Cores are dedicated units of each SM which operate on fixed-size matrix tiles of A, B and C to perform the calculation of AxB+C. Tensor Cores offer improved performance and compute density by minimizing control overhead; whereas thread-based matrix multiplication incurs per-thread overhead, Tensor Cores execute the operation with a single, shared control path. Therefore, more transistors can be dedicated to the computation itself rather than control. The trade-off is increased kernel complexity since tiling of larger matrices becomes necessary; however, such tiling is typically implemented in thread-based approaches as well to enhance memory efficiency. Additionally, the Tensor Core instruction breaks the previously described model of threads operating independently on different data, requiring explicit warp-level collaboration: all 32 threads in a single warp need to call the Tensor Core simultaneously, with each thread providing part of the information. This is done by calling an instruction in the PTX language, an intermediate representation that CUDA code compiles to. In our code, this instruction is embedded as *inline PTX* into the CUDA source code using a C++ macro.

Tensor Cores predominantly support lower-precision floating-point formats, such as 16-bit (FP16) or 8-bit (FP8) representations, as these reduced-precision formats have become increasingly prevalent in machine learning applications, offering an acceptable trade-off between numerical precision and computational efficiency.

With the Ampere architecture, released in 2020, NVIDIA introduced sparse Tensor Cores, expanding the functionality of Tensor Cores to accommodate sparse matrices. These specifically address the scenario where matrix A is sparse, while matrices B and C remain dense. The primary advantage of sparse Tensor Cores is twofold: they offer reduced runtime for computations with identical tile sizes compared to dense Tensor Cores, and they maintain equivalent runtime when processing tile sizes twice as large. For instance, with FP16

format, dense Tensor Cores perform multiplications with dimensions 16×8×16, whereas sparse Tensor Cores efficiently handle dimensions of 16×8×32.

However, sparse Tensor Cores impose significant constraints on the structure of sparsity: they exclusively support 2:4 sparsity patterns. This means that within groups of four consecutive values in the original matrix tile, at least two elements must be zero. Matrix A is supplied to the Tensor Core instruction as a dense tile of half the original size containing all non-zero values, accompanied by an additional metadata vector that indicates the positions of these values within each group of four elements. This format is visualized in Figure 1.3.

This architectural limitation stems from the underlying hardware implementation (see Figure 1.4): sparse and dense Tensor Cores utilize the same arithmetic logic units (ALUs) for multiplication operations, with a specialized decoder selecting the corresponding values from matrix B using the provided metadata vector prior to the operation. This approach requires only minimal additional decoder hardware and incurs little overhead due to the simplicity of the decoding mechanism. Additionally, the size of each memory load is halved for the same tile size, as only nonzero values of A and B are loaded. While this 2:4 sparsity format imposes structural constraints on compatible matrices, it represents an effective compromise between hardware complexity and performance enhancement for appropriately structured sparse matrices.

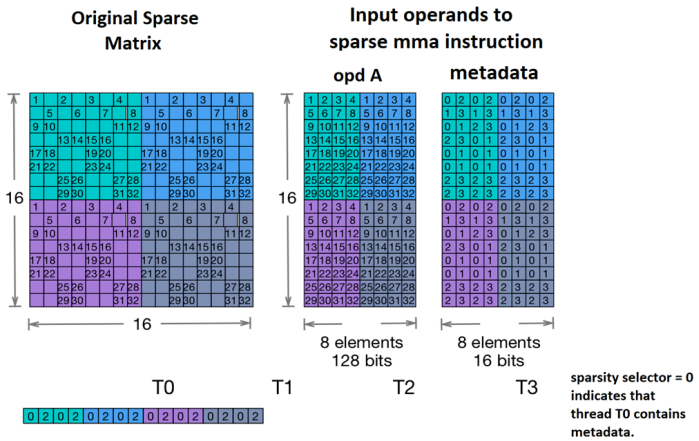

Figure 1.3: Input format of sparse Tensor Core instruction [6]

1.2 Sparse Matrix Formats

Many sparse matrix formats have been proposed in the past, such as Coordinate List (COO) or Compressed Sparse Row (CSR). While these formats achieve

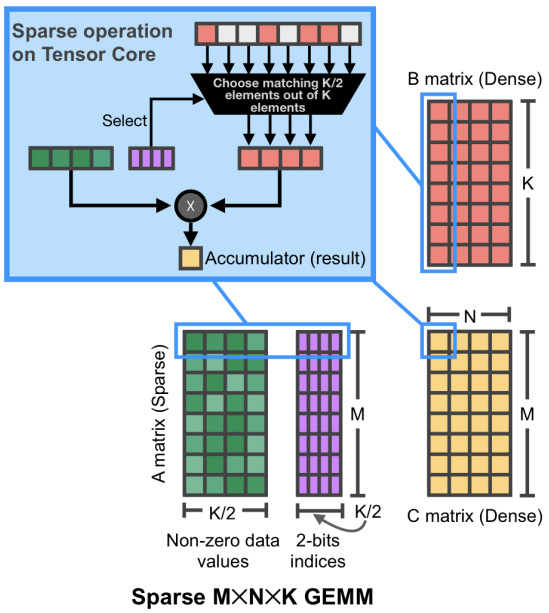

Figure 1.4: Hardware implementation of sparse Tensor Core [7]

optimal information density, they are unsuitable for high-performance GPU implementations. Exploiting cache hierarchies and leveraging Tensor Cores in modern GPUs requires blocking techniques, which these traditional storage formats generally do not support natively.

With sparse matrices, blocking introduces an inherent tradeoff between sparse compression effectiveness and tile size—as tile sizes increase, more zero elements must be stored explicitly, reducing the storage benefits of sparse representation. For our implementation, we utilize the Block Compressed Sparse Row (BCSR) format, which extends the traditional CSR format with blocking capabilities (see Figure 1.5). In BCSR, the row pointers and column indices reference blocks rather than individual elements, operating at the block level instead of the scalar level. Initially, we align our blocking factors with the dimensions of Tensor Core tiles, i.e. 16x16 in the dense and 16x32 in the sparse case.

This approach allows us to balance the competing demands of sparse storage efficiency and computational performance on GPU architectures. By structuring our sparse data in block-oriented formats compatible with Tensor Core operations, we can more effectively harness the computational capabilities of modern GPUs while still benefiting from reduced memory requirements compared to dense representations.

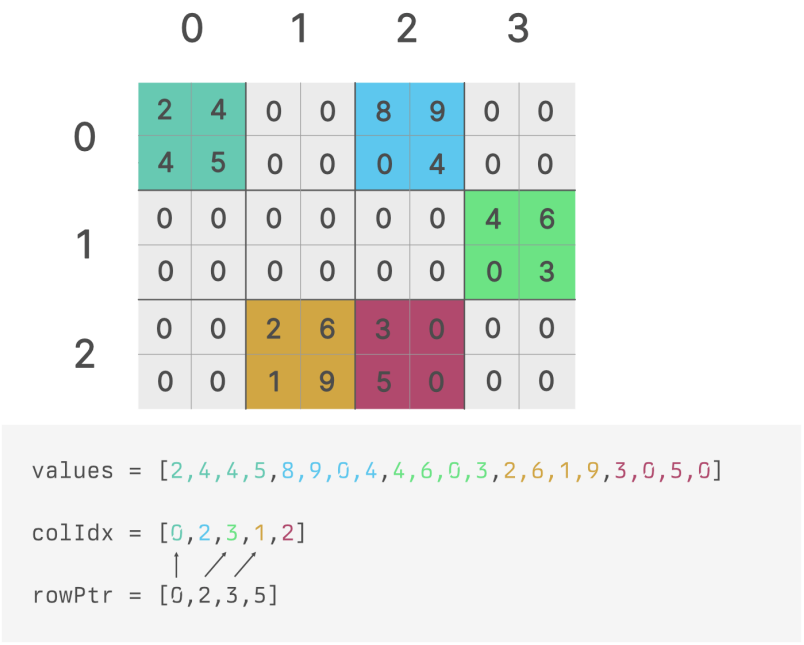

Figure 1.5: BCSR Format Visualization

1.3 Problem Setup

For the research problem addressed in this thesis, matrix A exhibits static or infrequent changes, allowing for comprehensive preprocessing operations including conversion to BCSR format and 2:4 sparsity pattern verification. This preprocessing phase is followed by multiple matrix multiplication operations using varying instances of matrix B.

The sparse matrix A features substantial dimensions, with performance evaluations spanning sizes from 2048×2048 to 16384×16384 elements. This matrix undergoes partitioning into tiles that fall into three distinct categories: fully sparse tiles containing exclusively zeros, 2:4 sparse tiles that conform to Tensor Core requirements, and dense tiles with predominantly non-zero values. The experimental framework incorporates various compositional arrangements with different proportions of these tile types to thoroughly assess performance across sparsity patterns.

The dense matrix B is characterized by its narrow profile, with benchmark configurations ranging from the width of a single Tensor Core tile up to 16 tiles. This configuration reflects common computational patterns found in scientific computing, where a large sparse matrix is multiplied by a relatively narrow dense matrix.

1.4 Previous Work

The usage of Tensor Cores for sparse matrices has been explored before, mainly focusing on specific matrix formats and usecases. Magicube [8] is a library designed specifically for deep learning. It therefore only supports lower-precision integers up to 8 bits. DASP [9] only supports matrix-vector multiplication.

1.4.1 Magicube

Magicube [8] is a specialized SpMM library developed specifically for deep learning applications. It employs a block-sparse matrix storage format, though with a strided variant that differentiates it from standard BCSR implementations. Reflecting its focus on deep learning workloads, Magicube operates exclusively with reduced-precision numerical formats that are prevalent in this domain, supporting only up to 8-bit integer precision as its highest-precision format. This specialization allows Magicube to achieve optimized performance for deep learning inference and training, though it limits its applicability to scientific computing domains that require higher numerical precision.

1.4.2 DASP

DASP [9] relies heavily on preprocessing, where it first divides the sparse matrix rows into three different categories depending on the number of nonzero values, after which it processes it using different strategies for each category. It supports FP16 and FP64 number formats and only supports Sparse Matrix-Vector Multiplication (SpMV).

1.4.3 NVIDIA cuSPARSE

NVIDIA cuSPARSE [10] is a general-purpose sparse matrix library that suffers from significant performance limitations [8] [9] [11], often failing to substantially outperform equivalent dense algorithms implemented in NVIDIA cuBLAS [12], particularly when processing highly sparse matrices (greater than 99% sparsity). These performance constraints can be attributed to several key factors, including its reliance on the non-blocked CSR format and its failure to leverage Tensor Cores, instead utilizing only CUDA cores for computation [13].

1.4.4 NVIDIA cuSPARSELt

NVIDIA cuSPARSELt [14] is specifically designed for structured sparsity in deep learning workloads. It utilizes sparse Tensor Cores, however it only works on fixed sparsity patterns.

1.4.5 SpInfer

SpInfer [15] is a library designed specifically for large language model inference using SpMM. It utilizes bitmaps for encoding the sparse matrix. Its performance falls behind other solutions at extreme sparsity levels (>=99.97%), as it does not optimize for these sparsity levels due to its focus on LLMs.

1.4.6 SMaT

SMaT [16] is a library for SpMM using Tensor Cores and is the starting point of the optimizations introduced in this thesis. It includes several kernels that iteratively add several optimization techniques which are detailed below.

mmaT

The first kernel employing Tensor Cores, `mmaT.cu`, utilizes a two-dimensional grid structure where each thread block is responsible for computing a single tensor-core tile of matrix C. As previously established, invoking a Tensor Core instruction requires a warp of 32 threads; consequently, we assign a single warp to each thread block. To calculate the values within its designated tile, each thread block must iterate through the corresponding tile row of matrix A, multiplying each tile with its matching tile from matrix B, accumulating the results, and ultimately writing the final values to matrix C. Figure 1.6 visualizes this process.

During each iteration, the kernel first determines whether the current tile in A consists entirely of zeros or contains non-zero values by consulting a provided metadata array. If the tile is completely zero, the iteration is bypassed. Otherwise, the calculation proceeds through a three-phase process: initially, the threads collaboratively transfer the tile from global memory to shared memory. Subsequently, each thread loads its specific portion of the tile—the data it will contribute to the Tensor Core instruction—into its thread-local registers. Following this, the threads collectively execute the Tensor Core instruction, with the resulting values being accumulated in the thread-local `C` registers. After completing all iterations, the accumulated values in the `C` registers are transferred back to shared memory, from which the entire shared memory tile is written to the corresponding location in global memory.

mmaBT

While the initial kernel skipped iterations when encountering tiles composed entirely of zeros, this kernel, `mmaBT`, fully leverages the row pointer and column index arrays provided by the BCSR matrix format. Rather than checking and potentially skipping zero tiles, it directly iterates through non-zero tiles by traversing the section of the column index array corresponding to the relevant row, obtaining the starting and ending positions from the row pointer array.

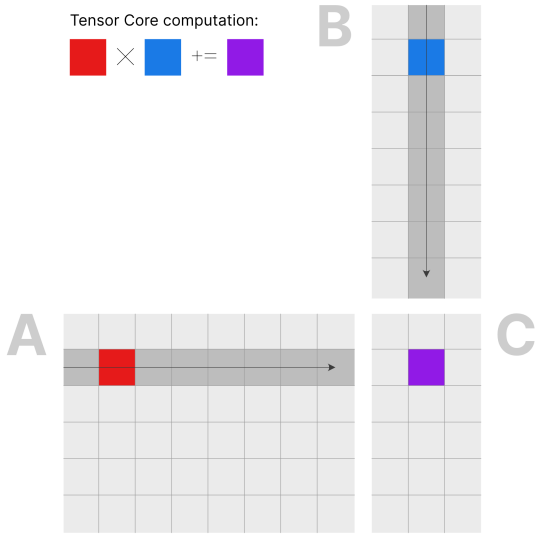

Figure 1.6: Matrix layout and computation - the arrows show the mainloop that is executed by a single warp

This approach eliminates unnecessary iteration attempts, focusing computational resources exclusively on tiles containing actual data, thereby improving efficiency.

mmaCBT

This kernel enhances memory transfer efficiency by implementing asynchronous memory copies. While the conventional process of global-to-shared memory loads requires an intermediate copy into registers, the asynchronous memory copy mechanism bypasses this step by utilizing Direct Memory Access (DMA). Furthermore, the asynchronous nature enables temporal overlapping of A and B matrix copies, further accelerating memory movement. It is important to note that this kernel does not facilitate the overlapping of memory operations with computation, as the subsequent shared-to-register loads and Tensor Core instruction execution remain dependent on the completion of these global-to-shared memory transfers.

Chapter 2

Performance Optimizations

Kernel	Sparse TC	BCSR	Coop. Loads	Pipelining	Large Tiles	Tiling	Dense/Sparse Seg.
`mmaT`							
`mmaST`*	✓						
`mmaST_large`*	✓				✓		
`mmaBT`		✓					
`mmaCBT`		✓	✓				
`mmaOBT`*		✓		✓			
`mmaOBT_tiled`*		✓		✓		✓	
`mmaOBTS`*	✓	✓		✓			
`mmaOBTS_large`*	✓	✓		✓	✓		
`mmaOBTS_large_separate`*	✓	✓		✓	✓		✓

Table 2.1: Features of different kernel implementations (* denotes contribution of this thesis)

Starting from the baseline kernels `mmaT`, `mmaBT`, and `mmaCBT` from Okanovic et al. [1], this chapter presents a series of optimizations developed in this thesis. These optimizations are progressively integrated and, where appropriate, combined to explore their cumulative performance benefits.

2.1 Introduction of Sparse Tensor Cores

The first performance optimization strategy was the utilization of the aforementioned sparse Tensor Cores. As previously described, tiles must be supplied in a very specific format for the sparse Tensor Core instruction, necessitating

significant preprocessing for matrix A. A separate preprocessing CUDA kernel was created for this purpose. For each tile, the nonzero values are packed into tiles of half the original width, resulting in a `sparseMatrixA` array of 16×8 tiles. The metadata is supplied by one thread in a group of four as 16 2-bit vectors packed in a 32-bit integer value. To facilitate this approach, a separate `metadata` array is generated by the preprocessing kernel (see Listing 2.1).

To implement the kernel, `mmaST.cu`, the initial mmaT kernel was modified and enhanced with a branch in the main loop that depends on the sparsity of the tile. As in the previous implementation, loop iterations are skipped entirely for zero tiles, and the dense Tensor Core instruction is called for tiles that do not fulfill the 2:4 sparsity criteria. For tiles exhibiting 2:4 sparsity, the corresponding matrix and metadata values are loaded following the same two-step loading pattern. Subsequently, the sparse variant of the Tensor Core PTX instruction is invoked (see Listing 2.2). This conditional approach allows the kernel to adaptively select the most appropriate computation method based on the sparsity characteristics of each individual tile.

For the next optimization iteration, the focus shifted toward increasing throughput rather than reducing latency by exploiting the larger tile size capabilities of sparse Tensor Cores, resulting in the kernel `mmaST_large.cu`. In this implementation, the tile size was expanded to 16×32, effectively doubling the width compared to the standard Tensor Core dimensions. To handle dense tiles within this larger framework, the kernel executes the dense Tensor Core instruction twice per iteration.

```
1 half *src = ((half *)((int4 *)(&bcsrValuesA[(relativeIndex)*MMA_M * MMA_K +
2                                             (lane_id / 2) * MMA_K]) +
3                                              lane_id % 2));
4 half src_sparse[4];
5
6 char *cur_meta = (Meta_smem[lane_id / 2]) + (lane_id % 2);
7 for (int j = 0; j < 2; ++j) {
8   int cur_src_sparse = 0;
9   src_sparse[0 + (2 * j)] = 0;
10  src_sparse[1 + (2 * j)] = 0;
11  for (int i = 0; i < 4; ++i) {
12    if (src[i + (4 * j)] != (half)0.0f) {
13      src_sparse[cur_src_sparse + (2 * j)] = src[i + (4 * j)];
14      *cur_meta |= i << (6 - (2 * (1 - cur_src_sparse) + (4 * j)));
15
16      cur_src_sparse++;
17    }
18  }
19 }
```

Listing 2.1: Generation of 2:4 sparsity metadata vector in preprocessing kernel

```
#define HMMA16816_SPARSE(RD0, RD1, RA0, RA1, RB0, RB1, RC0, RC1, RE, CONST) \
  asm volatile(\
      "mma.sp::ordered_metadata.sync.aligned.m16n8k16.row.col.f16.f16.f16.f16 \
      {%0, %1}, {%2, %3}, {%4, %5}, {%6, %7}, %8, %9;" \
      : "=r"(RD0), "=r"(RD1) \
      : "r"(RA0), "r"(RA1), "r"(RB0), "r"(RB1), "r"(RC0), "r"(RC1), "r"(RE), \
        "n"(CONST))
```

Listing 2.2: mma.m16n8k16 Tensor Core instruction in FP16

2.2 Pipelining: Overlap of Memory and Compute

As a further optimization, the memory loading improvements developed by Okanovic et al. were expanded upon. By utilizing the `cuda::pipeline` feature, a two-stage pipeline was implemented to overlap memory operations and computation (see Figure 2.1). The capacity of each shared memory array was doubled to accommodate copies for both stages of the pipeline. The kernel initially populates both pipeline stages by loading the first two tiles from global memory to shared memory. Subsequently, the matrix tiles required for the next iteration of the main loop are loaded to shared memory concurrently with the shared-to-register loading and execution of the Tensor Core instruction for the current iteration. One limitation of this approach is the doubled shared memory consumption, as two complete pipeline stages must reside in memory simultaneously.

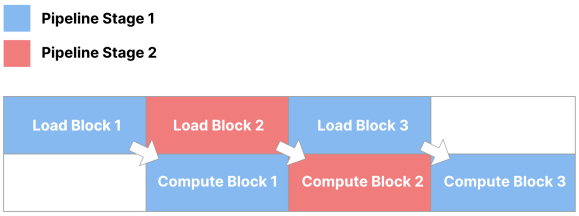

Figure 2.1: Pipeline, idealized with assumption of equal loading and processing latencies

2.3 Separation of Dense and Sparse Processing

Previous approaches relied on runtime branching within the main loop and load stage to choose between dense and sparse code paths. However, the presence of branching inhibits the compiler's ability to reorder instructions, limiting opportunities to overlap memory operations and computation. In our case, for example, the global memory load of the column index—which could otherwise be moved earlier—was constrained to occur after evaluating the branch condition. Furthermore, this condition itself was derived from the

sparsity information vector stored in global memory, introducing an additional global load and corresponding latency. Nsight Compute identified this as a stall during branch condition evaluation (see Figure 2.2). To eliminate this overhead, the kernel `mmaOBTS_large_separate.cu` segregates dense and 2:4 sparse tiles into separate arrays during preprocessing and processes them sequentially in distinct loops, allowing the compiler to fully optimize each path independently.

Figure 2.2: Nsight Compute profiler screenshot: high stall sampling result of 5.99% indicates frequent stalls during branch condition evaluation

2.4 Tiling

The fundamental advantage of tiling derives from a mathematical relationship between computation and memory access patterns. When computing a tile of dimensions `tile × tile` in matrix C, the algorithm must load corresponding rows from matrix A and columns from matrix B of length `tile`. This creates a favorable scaling relationship: the computational work increases quadratically (proportional to $\texttt{tile}^2$), while the memory access requirements increase only linearly (proportional to `2 × tile`).

More specifically, when computing a tile of size `tile × tile` in matrix C, the algorithm must load `tile` rows from matrix A and `tile` columns from matrix B. These loaded elements are reused multiple times during computation, with each loaded element from A being used for `tile` different calculations and each loaded element from B likewise being used for `tile` different calculations. Since memory loads typically constitute the majority of kernel runtime, increasing the arithmetic intensity—the amount of computation performed per memory load—generally yields performance benefits.

This advantageous relationship prompted the investigation of whether introducing another level of tiling might further enhance kernel performance. The kernel `mmaOBT_tiled.cu` implements this concept by adding a blocking factor `BLOCK` and loading larger tiles consisting of `BLOCK×BLOCK` Tensor Core tiles into shared memory. Instead of processing with a single warp, `BLOCK×BLOCK` warps collaboratively process these expanded tiles as part of a single thread block (see Figure 2.3). The larger tiles are created in the preprocessing stage by merging adjacent nonzero tiles.

For dense matrices, this hierarchical tiling strategy almost universally improves performance, and the blocking factor should be maximized within the constraints of available shared memory. However, in sparse matrices, the tiling factor represents a critical trade-off: increasing the tile factor inevitably leads to a higher number of explicitly stored zero subtiles, which rapidly diminishes returns as the blocking factor increases (see Figure 2.4). This occurs because larger sparse tiles tend to encompass more empty regions, reducing the efficiency gained from the improved arithmetic intensity.

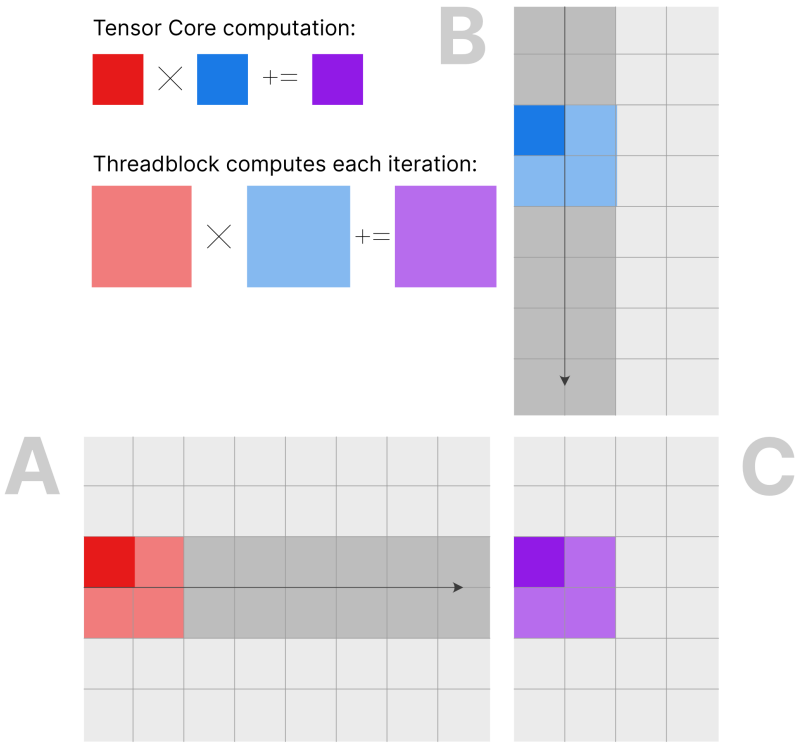

Figure 2.3: Illustration of mainloop for tiled kernel

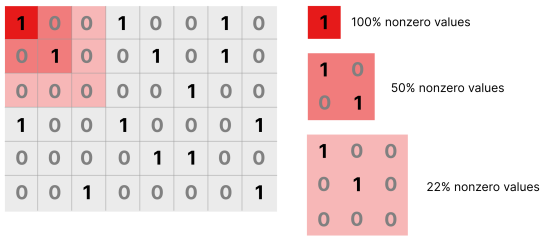

Figure 2.4: Explicitly stored zeros increase with tile size

Chapter 3

Benchmarks

3.1 Experimental Setup

Component	Specification
GPU	NVIDIA A100 80 GB 80 GB HBM2e ECC Memory 6912 CUDA Cores 432 3rd-Generation Tensor Cores
CPU	AMD EPYC 7742 64-Core Processor 4 Cores, 8 Threads dedicated to VM 2 MiB L2 Cache (4 instances) 16 MiB L3 Cache (1 instance)
System Memory	128 GB
Operating System	Ubuntu 24.04.1 LTS
CUDA Version	CUDA 12.7
Compiler	GCC 13.3.0 (Ubuntu 13.3.0-6ubuntu2~24.04) Compilation flags: -O3 -arch=sm_80

Table 3.1: Hardware and software configuration used for benchmarking

3.1.1 Benchmark Methodology

All experiments[1] were conducted on the hardware configuration detailed in Table 3.1. Each kernel was executed 10 times, with the median execution time reported. The throughput metric represents TFLOPS (trillion floating-point

[1]Source code available at: https://github.com/ajit283/sptc-smat

operations per second), calculated as

$$\text{Throughput (TFLOPS)} = \frac{2\,nnzb\,M_{\text{tile}}\,K_{\text{tile}}\,N_{\text{MULT}}}{t \times 10^{12}}$$

for our $N \times N$ matrices.

3.1.2 Matrix Variants

Synthetic matrices

To evaluate kernel performance under controlled circumstances, the benchmark suite utilizes a collection of synthetic matrices with defined sparsity patterns. Each matrix consists of 16x32 tiles characterized as either:

- Zero tiles (containing no non-zero elements)
- 2:4 structured sparse tiles (following the NVIDIA 2:4 sparsity pattern)
- Dense tiles (with randomly distributed non-zero elements)

Figure 3.1 visualizes the different tile types. The matrix variants are denoted using the nomenclature **XdYs**, where X represents the percentage of dense blocks and Y represents the percentage of structured sparse blocks, with the remaining blocks being zero.

In the second benchmark, performance is evaluated for a matrix consisting of entirely 2:4 sparse tiles to test the performance of sparse Tensor Cores in isolation. Additionally, the size of matrix B is varied by changing the factor `N_Mult` which determines the width of matrix B in blocks.

SuiteSparse collection

To evaluate performance in real-world applications, the kernels are additionally tested on matrices used in various scientific fields (see Table 3.2), sourced from the SuiteSparse matrix collection [17].

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 3.2: Selected matrices from the SuiteSparse matrix collection

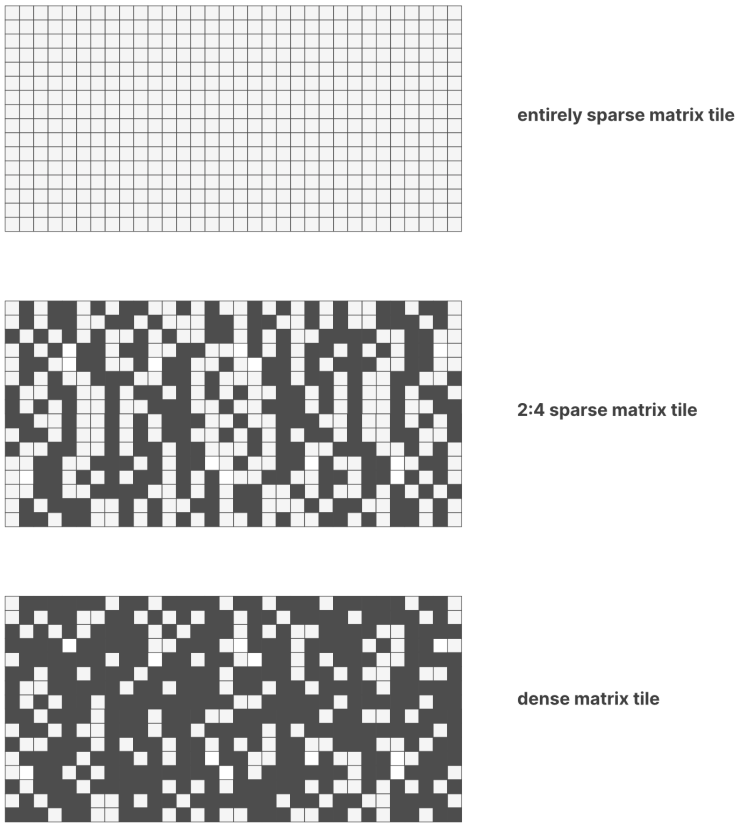

Figure 3.1: Visualization of matrix tiles that synthetic matrices are composed of (grey cells correspond to nonzero entries)

3.2 Benchmarks

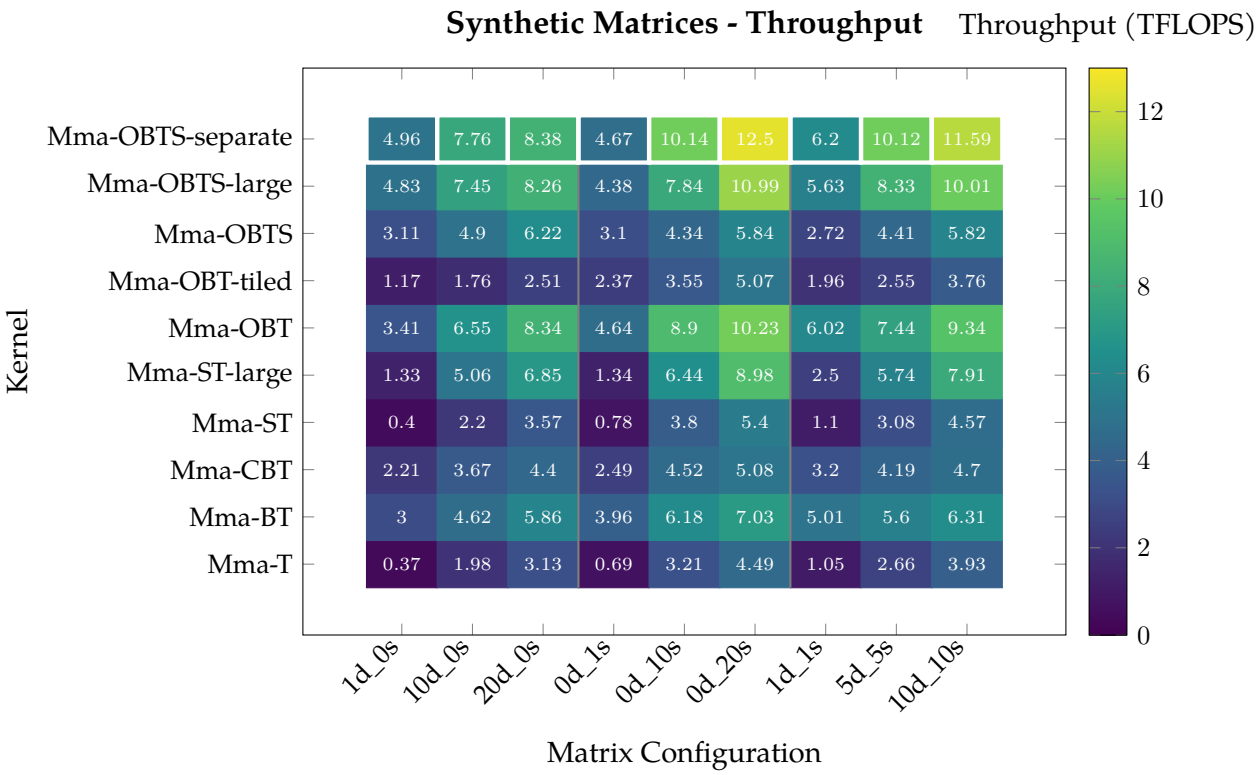

Figure 3.2: Throughput comparison of SpMM kernels on synthetic matrices. Matrices are grouped by sparsity pattern: only dense tiles, only 2:4 sparse tiles, and mixed sparsity patterns. White borders highlight the top performer for each matrix type.

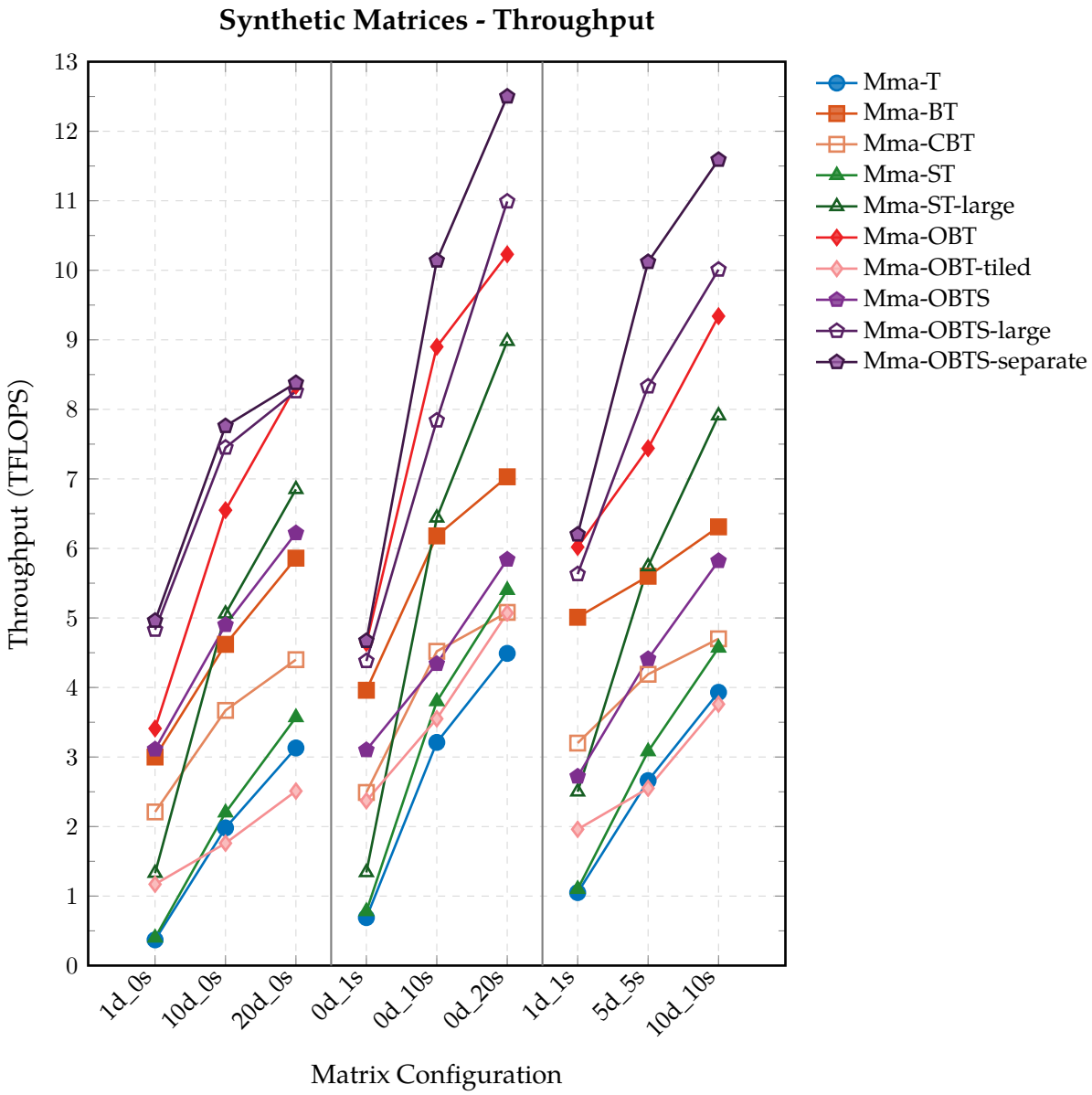

Figure 3.3: Throughput comparison of SpMM kernels on synthetic matrices. Matrices are grouped by sparsity pattern: only dense tiles, only 2:4 sparse tiles, and mixed sparsity patterns.

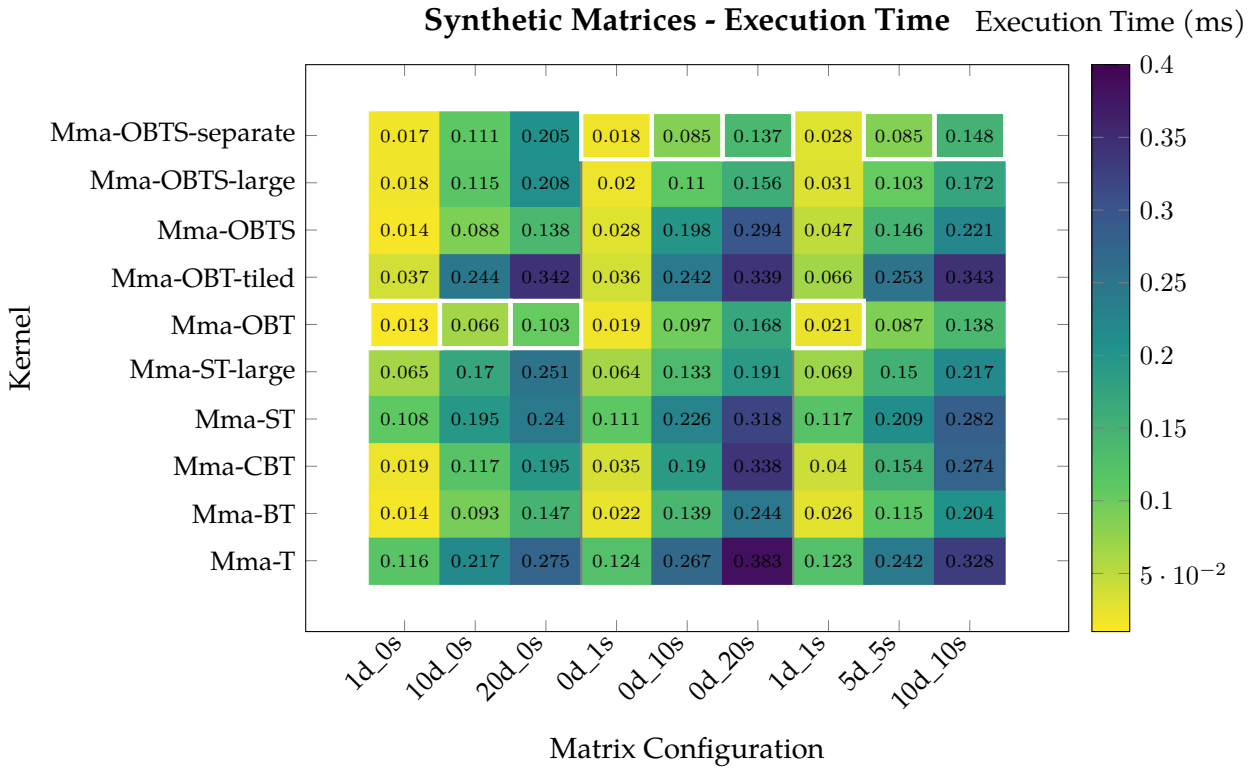

Figure 3.4: Duration comparison of SpMM kernels on synthetic matrices. Matrices are grouped by sparsity pattern: only dense tiles, only 2:4 sparse tiles, and mixed sparsity patterns. White borders highlight the top performer for each matrix type.

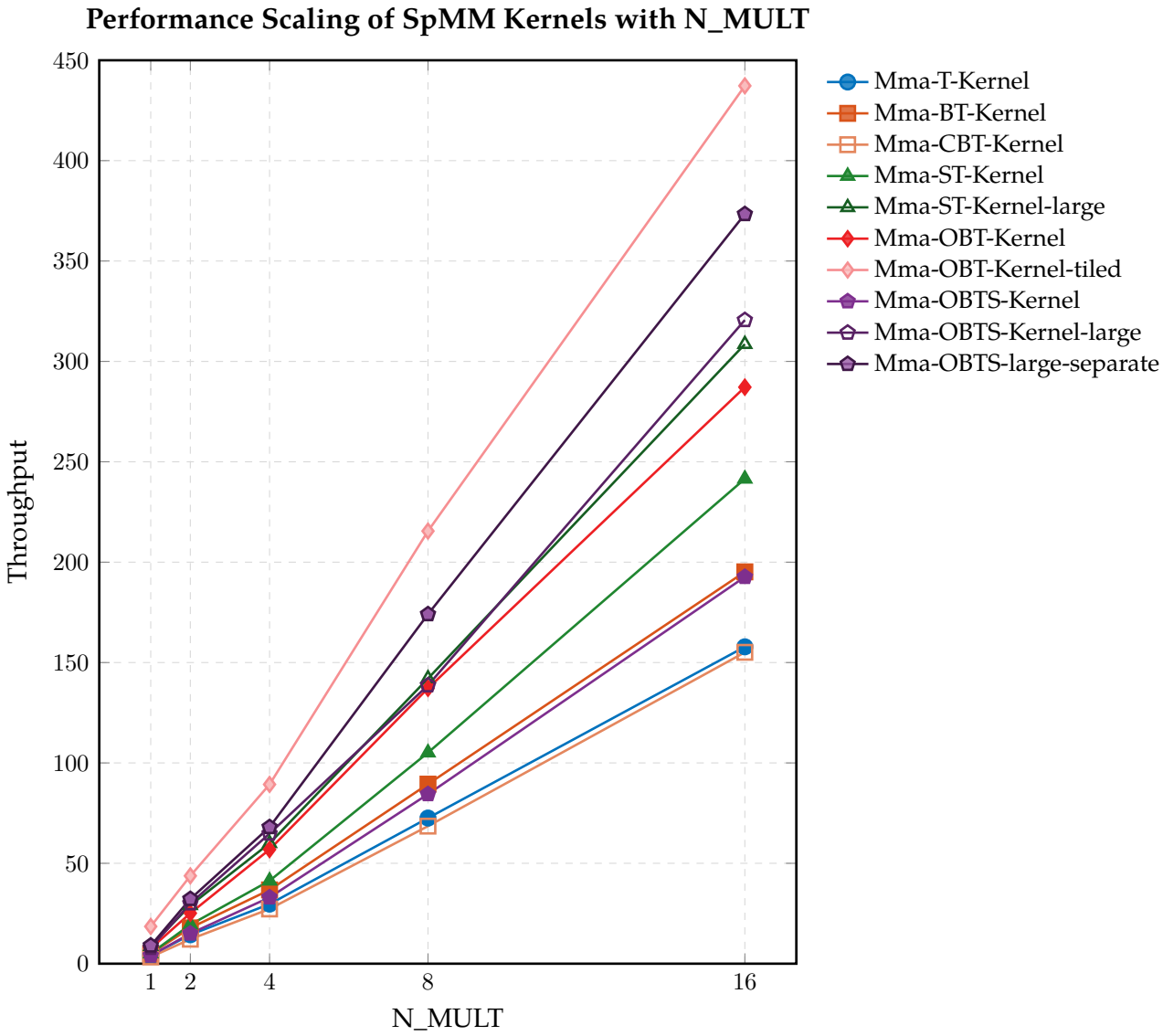

Figure 3.5: Throughput comparison of synthetic matrix consisting entirely of 2:4 sparse tiles and varying width of matrix B.

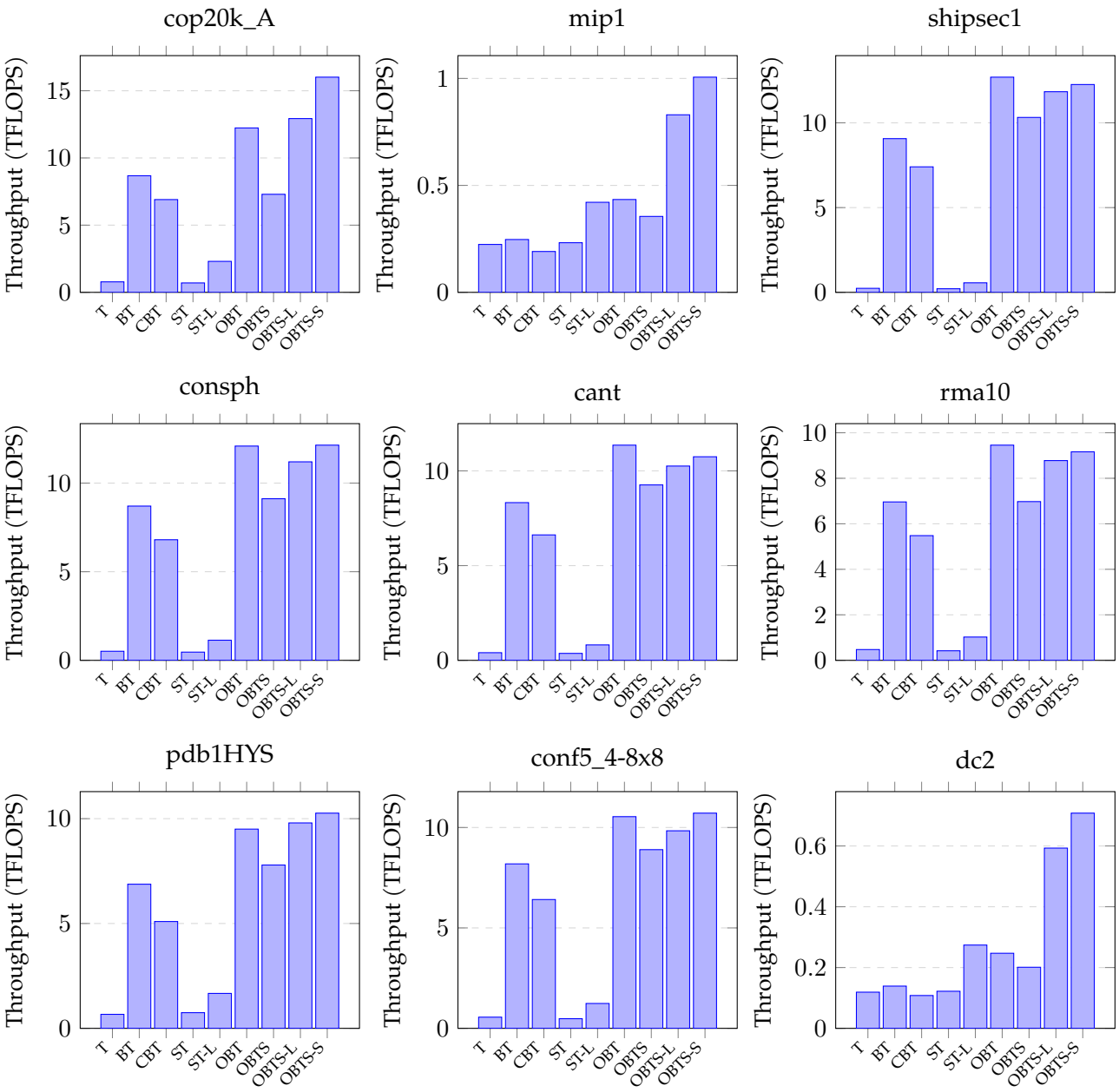

Figure 3.6: Throughput performance (TFLOPS) of different SpMM kernels across various real-world matrices. Kernels are abbreviated: T (Mma-T), BT (Mma-BT), CBT (Mma-CBT), ST (Mma-ST), ST-L (Mma-ST-large), OBT (Mma-OBT), OBTS (Mma-OBTS), OBTS-L (Mma-OBTS-large), OBTS-S (Mma-OBTS-large-separate).

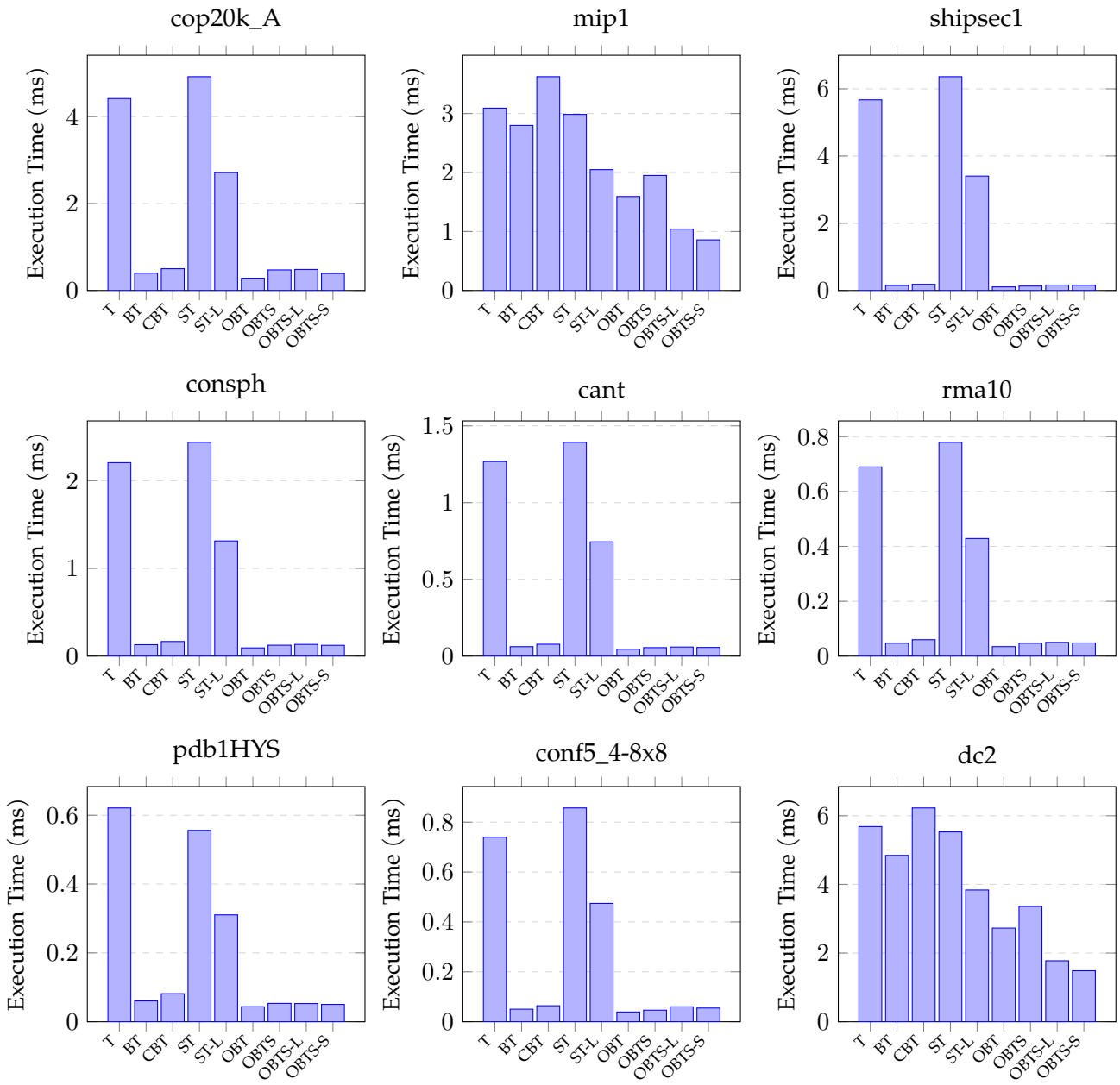

Figure 3.7: Execution time (ms) of different SpMM kernels across various real-world matrices. Kernels are abbreviated: T (Mma-T), BT (Mma-BT), CBT (Mma-CBT), ST (Mma-ST), ST-L (Mma-ST-large), OBT (Mma-OBT), OBTS (Mma-OBTS), OBTS-L (Mma-OBTS-large), OBTS-S (Mma-OBTS-large-separate). Lower values indicate better performance.

Chapter 4

Discussion of Results

4.1 Sparse Tensor Cores

Overall, the utilization of Tensor Core instructions substantially increases computational throughput. The `mmaST` kernel demonstrates only marginal performance improvements over its dense counterpart, the `mmaT` kernel, and even exhibits lower performance than the `mmaT` kernel in scenarios with very low density. This limited improvement can be attributed to the fact that computation constitutes only a small fraction of the kernel's overall latency; consequently, a reduction in computational latency yields only a minimal decrease in total kernel latency. Performance improvements become significantly more pronounced when leveraging the doubled tile size in the `mmaST_large` kernel, where 16x32 tiles of matrix A are processed in the Tensor Core instead of 16x16, resulting in double the throughput compared to the baseline `mmaT` kernel in the `0d_10s` case. Interestingly, the synthetic benchmarks reveal noticeable throughput improvements even when processing dense tiles exclusively. One possible explanation for this behavior is what could be described as a form of *thread tiling*: recall that in the dense case, two 16×16 tiles are processed sequentially to align with the expanded 16×32 layout used for sparse tiles. To support this, two shared-to-register loads are issued, which the hardware may be able to fuse into a single wider memory load. As a result, the same amount of data transfer can serve a larger amount of computation, increasing arithmetic intensity and potentially improving overall kernel efficiency. However, this remains a hypothesis, and further low-level profiling would be required to confirm whether such memory load coalescing is indeed occurring.

4.2 Pipelining

The pipelining optimization provides a consistent performance improvement, with the `mmaOBT` kernel outperforming its baseline `mmaBT` kernel by approxi-

mately 1.4x across all synthetic and real-world matrices. This improvement can be explained easily, as apart from a constant pipeline initialization and synchronization overhead, the overlap of memory and compute reduces execution time to the shorter of the two stages.

4.3 Tiling

The performance metrics for the tiled kernel illustrate how the substantial improvements achieved in dense matrix multiplication with this method do not necessarily extend to sparse matrices. With the high-sparsity matrices benchmarked in Figure 3.2, the problem of explicitly stored zeros becomes evident as the `mmaOBT_tiled` kernel significantly underperforms against its baseline `mmaOBT` kernel. In the ideal case of entirely nonzero tiles (presented in Figure 3.5), however, this issue is mitigated since all tiles contain nonzero elements. Consequently, performance improvements similar to those observed when applying this optimization to dense matrix multiplication kernels become apparent, with the `mmaOBT_tiled` outperforming all other kernels in this case. This clearly indicates that tiling optimization is primarily beneficial for matrices with very low sparsity.

4.4 Separation of Dense and Sparse processing

One might reasonably anticipate that the combination of pipelining and sparse Tensor Cores would yield cumulative performance gains from both optimizations, thereby surpassing kernels that implement only a single optimization. However, empirical observations reveal that the combined `mmaOBTS` and `mmaOBTS_large` kernels fail to exceed the performance of the `mmaOBT` kernel and, in certain instances, also underperform relative to the `mmaST` kernel. This performance deficit can be attributed to the branching mechanism employed to select between dense and 2:4 sparse code paths during the loading and processing phases. The disparate latencies of these two code paths induce pipeline imbalances, diminishing the degree of parallelism achieved through pipelining. As discussed in the Optimizations chapter, this branching mechanism additionally constrains compiler optimization opportunities and potential performance gains through instruction-level parallelism.

The `mmaOBTS_large_separate` kernel eliminates branching by segregating dense and sparse tiles during the preprocessing phase. This enables the effective integration of both optimizations, establishing it as the highest-performing kernel across all 2:4 sparse synthetic matrices, with throughput being up to 1.83x higher than the `mmaBT` kernel and up to 13.4x higher than the `mmaT` kernel. In the case of the SuiteSparse matrices, however, it does not consistently achieve optimal performance, with the `mmaOBT` kernel demonstrating superior performance in numerous instances. The underlying cause becomes evident

upon detailed examination of the respective matrices' structural properties. Although these matrices exhibit high overall sparsity, they frequently contain patterns that render most blocks incompatible with the 2:4 sparse format. For instance, the `cant` matrix contains only 7.8% of nonzero blocks that conform to the 2:4 sparse pattern, with the remainder being completely dense. Conversely, in the `mip1` matrix, where `mmaOBTS_large_separate` outperforms `mmaOBT`, 33.8% of all nonzero blocks exhibit 2:4 sparsity.

Examination of the respective densities of these matrices reveals that overall sparsity demonstrates minimal correlation with the proportion of 2:4 sparse blocks, as even a single violation of the 2:4 sparse property, such as three nonzero values positioned adjacently, results in the entire tile being classified as dense. Under these circumstances, the overhead introduced by the sparse main loop diminishes the performance of `mmaOBTS_large_separate` relative to `mmaOBT`, relegating it to second-best performance in numerous instances.

4.5 Conclusion

In this thesis, we investigated whether 2:4-sparse Tensor Cores, in combination with additional kernel optimizations, can achieve a $\geq$1.5× speed-up over state-of-the-art dense-core BCSR SpMM for matrices with at least 30% sparse-compatible tiles. Our results show that such speed-ups are indeed possible under the right conditions, demonstrating the potential of sparse Tensor Cores to improve computational efficiency in real-world applications. However, the strict 2:4 sparsity constraint significantly limits the range of matrices that benefit from this optimization and adds considerable kernel complexity. In contrast, pipelining proved effective across a much wider set of matrices, delivering consistent performance improvements. Tiling offered limited gains due to the cost of handling explicitly stored zeros, and is most useful for dense or uniformly sparse matrices, which may be better served by existing GeMM approaches. Overall, sparse Tensor Cores offer a promising but narrowly applicable acceleration path, best leveraged in conjunction with broader, more generally effective techniques like pipelining.

Chapter 5

Further Areas of Improvement

An additional optimization would be to combine sparsity, pipelining, and tiling in an `mmaOBTS_large_tiled` kernel. Each tile could also utilize a sparse representation such as CSR to only specify nonzero internal blocks, circumventing the problem of explicitly stored zeros. However, it remains unclear whether this approach would effectively address the challenges of tiling in highly sparse matrices, as a low density of nonzero blocks per tile would diminish the expected memory loading improvements. Furthermore, we must consider that the performance gains observed in our current implementation primarily arise when target blocks in a single row or column reuse the same row or column of the A and B matrices. As sparsity increases, the probability of such alignment decreases significantly, potentially negating any performance advantages.

Additionally, it would be valuable to exploit the capabilities of newer tensor core hardware found in NVIDIA's Hopper and Blackwell generation GPUs. The Hopper architecture introduced sparse and dense tensor core instructions that operate across multiple warps rather than a single warp, enabling larger tile sizes up to $64 \times 256 \times 16$. This generation also introduced the Tensor Memory Accelerator (TMA), which significantly enhances memory throughput between global and shared memory—addressing a key bottleneck in our current kernel implementations.

Lastly, as reduced-precision number formats such as FP8 or FP4 become increasingly relevant in machine learning applications, adapting these kernels to such highly quantized formats represents a promising direction for future research. These formats could potentially offer further performance improvements while maintaining acceptable accuracy for many machine learning workloads.

Bibliography

[1] P. Okanovic, G. Kwasniewski, P. S. Labini, M. Besta, F. Vella, and T. Hoefler, "High performance unstructured spmm computation using tensor cores," 2024. [Online]. Available: https://arxiv.org/abs/2408.11551

[2] Advanced Micro Devices, Inc., "Amd instinct mi300 series instruction set architecture," AMD, Tech. Rep., 2024, available at: https://www.amd.com/content/dam/amd/en/documents/instinct-tech-docs/instruction-set-architectures/amd-instinct-mi300-cdna3-instruction-set-architecture.pdf.

[3] NVIDIA, *CUDA C Programming Guide*, version 12.8 ed., NVIDIA Corporation, 2025. [Online]. Available: https://docs.nvidia.com/cuda/cuda-c-programming-guide/

[4] NVIDIA Corporation, "Nvidia a100 tensor core gpu architecture," https://images.nvidia.com/aem-dam/en-zz/Solutions/data-center/nvidia-ampere-architecture-whitepaper.pdf, 2020, accessed: 2025-04-29.

[5] NVIDIA, "Nvidia ampere architecture in-depth," NVIDIA Technical Blog, 2020, accessed: April 29, 2025. [Online]. Available: https://developer.nvidia.com/blog/nvidia-ampere-architecture-in-depth/

[6] NVIDIA Corporation, *PTX ISA Version 8.7: Parallel Thread Execution ISA*, NVIDIA Corporation, 2025, accessed: April 5, 2025. [Online]. Available: https://docs.nvidia.com/cuda/parallel-thread-execution/

[7] A. Mishra, J. A. Latorre, J. Pool, D. Stosic, D. Stosic, G. Venkatesh, C. Yu, and P. Micikevicius, "Accelerating sparse deep neural networks," 2021. [Online]. Available: https://arxiv.org/abs/2104.08378

[8] S. Li, K. Osawa, and T. Hoefler, "Efficient quantized sparse matrix operations on tensor cores," ser. SC '22. IEEE Press, 2022.

[9] 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.

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

[11] 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

[12] NVIDIA Corporation, "cublas: The NVIDIA CUDA basic linear algebra subroutines library," https://developer.nvidia.com/cublas, 2024, accessed: April 6, 2025.

[13] Y. Wang, B. Feng, Z. Wang, G. Huang, and Y. Ding, "TC-GNN: Bridging sparse GNN computation and dense tensor cores on GPUs," in *2023 USENIX Annual Technical Conference (USENIX ATC 23)*. Boston, MA: USENIX Association, Jul. 2023, pp. 149–164. [Online]. Available: https://www.usenix.org/conference/atc23/presentation/wang-yuke

[14] NVIDIA Corporation, "cusparselt: The NVIDIA CUDA structured sparse matrix library," https://docs.nvidia.com/cuda/cusparselt/, 2024, accessed: April 6, 2025.

[15] R. Fan, X. Yu, P. Dong, Z. Li, G. Gong, Q. Wang, W. Wang, and X. Chu, "Spinfer: Leveraging low-level sparsity for efficient large language model inference on gpus," in *Proceedings of the Twentieth European Conference on Computer Systems*, ser. EuroSys '25. New York, NY, USA: Association for Computing Machinery, 2025, p. 243–260. [Online]. Available: https://doi.org/10.1145/3689031.3717481

[16] P. Okanovic, G. Kwasniewski, P. S. Labini, M. Besta, F. Vella, and T. Hoefler, "High performance unstructured spmm computation using tensor cores," in *SC24: International Conference for High Performance Computing, Networking, Storage and Analysis*, 2024, pp. 1–14.

[17] T. A. Davis and Y. Hu, "The university of florida sparse matrix collection," *ACM Transactions on Mathematical Software*, vol. 38, no. 1, pp. 1:1–1:25, 2011.

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor's thesis, Master's thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

Using Sparse Tensor Cores for Sparse Matrix-Matrix Multiplication

Authored by (in block letters):

For papers written by groups the names of all authors are required.

Name(s):	**First name(s):**
Mistry	Ajit Jens Nathan

With my signature I confirm that

- I have committed none of the forms of plagiarism described in the 'Citation etiquette' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date	**Signature(s)**
Zürich, 30.04.2025	

For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.