GPU Accelerated Structured Sparsity Matrix Multiplication

Jasper Zeng
qz2283@nyu.edu
New York University
New York, New York, USA

Abstract

Structured $N:M$ sparsity is a common way to reduce the cost of large neural networks, but turning the theoretical FLOP savings into real speedups on modern GPUs requires carefully engineered sparse matrix–multiply kernels. This project implements and studies a custom $2:4$ sparse SpMM on an RTX 4070, asking when a hand-written sparse kernel can outperform both a sequential CPU baseline and a dense cuBLAS GEMM across a wide range of matrix sizes.

The kernel is built around a compressed layout for the weight matrix, fixed 32×32 tiles with 4×4 per-thread register blocks, and an index–gather path for A, and is extended with two variants: a register-based prefetch design and a double-buffered design that uses `cp.async` and optional Split-K parallelization.

Experiments show that the baseline sparse kernel attains 10^2–$10^4\times$ speedup over the CPU and large speedups over cuBLAS for small and medium matrices under $2:4$ sparsity, but becomes bandwidth-bound at very large sizes where dense cuBLAS overtakes it. Among the variants, the double-buffered kernel converts asynchronous copies into higher SM/DRAM utilization and modest end-to-end speedups at larger sizes, while the register-prefetch kernel stays close to the baseline and mainly serves as a negative control for lightweight cache warming in this index–gather–dominated regime.

1 Introduction

Deep neural networks are now deployed in a wide range of applications, but their computational and memory costs remain a major bottleneck at inference time. Model pruning is one of the most common strategies to reduce these costs: small weights are removed to shrink the number of floating-point operations and the size of the parameter tensors, ideally without significantly hurting accuracy [9, 8]. After pruning, the dense weight matrices in fully connected and attention layers become sparse, and the dominant operation in many layers can be viewed as a sparse matrix–matrix multiplication (SpMM) between a sparse weight matrix and a dense activation matrix [6]. The efficiency of this SpMM kernel largely determines whether a pruned network actually runs faster on GPU hardware, so accelerating SpMM is a key step toward accelerating the end-to-end pruned model.

Unfortunately, sparsity does not automatically translate into speed. With unstructured (element-wise) pruning, non-zero entries are scattered throughout the weight matrix, leading to irregular memory access patterns that break coalescing, increase cache misses, and introduce significant control divergence. $N:M$ structured sparsity addresses this problem by enforcing that exactly N out of every M consecutive weights are non-zero (for example $2:4$ for 50% sparsity), providing a compromise between unstructured and coarse block sparsity [23]. This pattern preserves much of the accuracy benefit of fine-grained pruning while introducing regular structure that modern GPUs—and even recent hardware features such as $2:4$ sparse tensor cores—can exploit [14]. In practice, however, exploiting $N:M$ sparsity still requires specialized SpMM kernels that operate on a compressed representation with only $W = K \times (1 - \text{sparsity})$ non-zero entries along the original K dimension, stored as separate value and index matrices of size $W \times N$. This format forces a two-stage memory access pattern in which threads first read indices and then perform indirect gathers from the dense input matrix, introducing scattered global reads, possible uncoalesced transactions, and shared-memory bank conflicts that do not appear in dense GEMM.

The goal of this final project is to implement and study GPU kernels for $N:M$ sparse SpMM that can turn these theoretical FLOP savings into actual wall-clock speedups on an NVIDIA RTX 4070. To this end, the report implements and compares three CUDA kernel variants: a single-buffered baseline with synchronous tile loads, a prefetch-enhanced kernel that overlaps part of the next tile's memory traffic with computation on the current tile, and a double-buffered kernel that uses PTX-level asynchronous global-to-shared copies (`cp.async`) together with optional Split-K parallelization. All three variants use a 32×32 thread-block tile and a 4×4 per-thread tile while processing the compressed W-dimension in fixed-size chunks using index-based gathers. Using NVIDIA Nsight Compute, the project analyzes DRAM throughput, SM utilization, warp stall reasons, and shared-memory bank conflicts to evaluate when these increasingly sophisticated techniques deliver meaningful speedups over the baseline and when the overheads of additional complexity outweigh their benefits.

2 Literature Survey

The challenge of accelerating sparse matrix operations on GPUs lies in mapping irregular sparse computations to the regular, data-parallel architecture of GPUs. Existing work can be organized along two primary dimensions: the sparse representation format, which determines the structure of non-zero data, and the optimization strategies used to extract performance from GPU hardware. Within this two-dimensional taxonomy, different communities have proposed formats and kernels that emphasize different trade-offs between generality, memory efficiency, and raw throughput.

2.1 Sparse Representation Formats

The choice of sparse format fundamentally shapes what optimizations are possible. Traditional unstructured formats like Compressed Sparse Row (CSR) store arbitrary sparsity patterns with row pointers and column indices for each non-zero element [2, 1]. Libraries such as cuSPARSE [17] and Sputnik [6] leverage these formats effectively for highly irregular matrices, but they suffer from poor memory coalescing and control divergence on GPUs, often achieving only a fraction (on the order of a few tens of percent) of dense

GEMM throughput [6, 13]. Block-based formats like Block CSR improve this by storing small dense blocks instead of individual elements, amortizing index storage and enabling vectorized load [18, 3]. However, they require coarse-grained sparsity patterns and store unnecessary zeros when sparsity does not align with block boundaries.

N:M structured sparsity represents a middle ground, enforcing that exactly N out of every M consecutive elements are non-zero (e.g., $2:4$ for 50% sparsity) [4, 19]. This format stores values in a compressed array alongside index arrays or bit-masks indicating non-zero positions. The predictable fine-grained structure maintains better memory coalescing than unstructured formats while preserving model accuracy better than coarse block sparsity[23].

2.2 Optimization Strategies

Orthogonal to format selection, researchers have explored various optimization strategies to address irregular memory access and hardware underutilization. Dense GEMM kernels achieve high performance through hierarchical tiling that exploits the GPU memory hierarchy—registers, shared memory, L2 cache, and DRAM [5, 20]. Adapting these techniques to sparse operations is challenging because irregular access patterns break the regular striding that makes tiling effective. Prior work has explored auto-tuning and adaptive tiling based on sparsity patterns [11] and load-balancing schemes that split work across rows or tiles to reduce imbalance [10], but the fundamental issue remains: index-based gathering disrupts the predictable memory access patterns that dense operations rely upon.

Memory access optimization represents a critical bottleneck. The gather phase, where indices dictate which elements to load from dense inputs, introduces latency and hurts coalescing [15]. Tiling, blocking, and reordering techniques have been proposed to improve locality, and some works explore software or hardware-assisted prefetching to hide this latency [12]. However, prefetching effectiveness is limited by register pressure and the unpredictability of index-based patterns. Modern GPU architectures (Ampere and later) provide hardware-accelerated asynchronous memory copy instructions (`cp.async`) that enable double-buffering: while one tile computes, the next tile loads asynchronously [21]. Applying this to sparse operations introduces additional complexity because threads must first load indices synchronously before determining which addresses to prefetch, creating sequential dependencies that limit pipelining effectiveness.

Split-K parallelization partitions the reduction dimension across multiple thread blocks to improve occupancy when matrix dimensions are unbalanced[16]. While this can better utilize the GPU, it introduces synchronization overhead and additional memory traffic for partial sums. For sparse operations, effectiveness depends heavily on problem size, and the atomic accumulation cost can outweigh parallelism benefits [7]. The compressed dimension in structured formats adds further complexity in maintaining balanced workload distribution, since work must be divided along a dimension that is already irregular.

2.3 Motivation

These representation and optimization choices have advanced the field considerably, providing diverse format options and a foundation of techniques adapted from dense GEMM. Recent hardware features like async copy and architectural improvements in memory hierarchies offer new opportunities to hide latency and increase effective bandwidth. However, several fundamental limitations persist. Index indirection overhead remains unavoidable—even with well-chosen tiles and reordering, gather operations introduce latency and hurt coalescing across all sparse formats [6]. Register pressure from storing indices, values, and accumulators limits occupancy compared to dense operations [15]. Most critically, the growing complexity of optimization techniques creates uncertain trade-offs: mechanisms that are highly effective for dense GEMM (double buffering, aggressive prefetching, large tiles) may hurt sparse operations due to index overhead and control divergence [22]. Furthermore, kernels optimized for one sparsity level often perform poorly at others as the memory-bound versus compute-bound balance shifts with changing sparsity and matrix shape [23].

These observations motivate the focus of this project. Prior work on N:M sparsity and its implementations has mostly targeted high-end datacenter GPUs and emphasized aggregate throughput, with limited systematic comparison of optimization strategies specifically for structured sparse SpMM on consumer hardware [23, 16]. Separate studies have examined prefetching [12], asynchronous copy [21], and Split-K [16] for various GPU workloads, but there is little controlled, apples-to-apples evaluation of how these techniques interact in the context of N:M sparse SpMM. Moreover, detailed profiling data—such as DRAM throughput, bank conflict rates, and warp stall breakdowns—are rarely presented in a way that explains why particular optimizations help or hurt [15], making it difficult to reason about failure modes and guide improvements.

This project addresses these gaps with two primary goals. First, it demonstrates that well-implemented N:M sparse SpMM kernels can achieve substantial practical speedups over both sequential CPU implementations and optimized dense libraries like cuBLAS by successfully translating the reduced FLOP count from sparsity into wall-clock performance gains. Second, through systematic comparison of three optimization strategies (baseline, prefetch, and double-buffer with Split-K) using Nsight Compute profiling on the RTX 4070, the project builds practical understanding of when each optimization provides consistent benefits, when effects are unpredictable or negative, and what performance characteristics explain the differences.

3 Proposed Idea

This section describes the implementation of the custom $N:M$ sparse SpMM. All kernels share a common compressed representation of the sparse weight matrix, a hierarchical tiling strategy adapted to the compressed dimension, and a register-level outer-product microkernel. On top of this baseline design, two increasingly aggressive optimization variants are explored: a register-based prefetch kernel and a double-buffered kernel that uses asynchronous copy instructions together with optional Split-K parallelization.

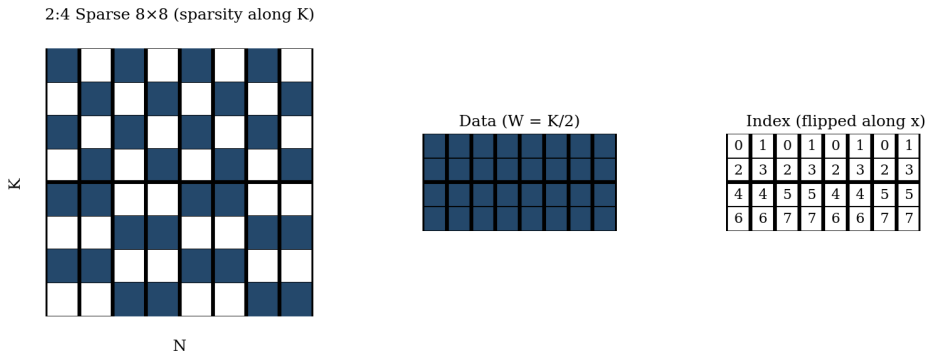


Figure 1: Illustration of $2:4$ structured sparsity along the K-dimension for an 8×8 matrix. The left panel shows the original sparse matrix with 2 non-zeros in every group of 4 entries along K. The middle panel is the compressed data tensor of shape $W \times N$ with $W = K/2$. The right panel stores the corresponding K-indices for each compressed row. Thick lines mark the $2:4$ blocks along K and column boundaries along N.

3.1 Compressed Sparse Representation and Index-Based Gathering

The sparse weight matrix $B \in \mathbb{R}^{K \times N}$ is stored in a compressed representation 1 that separates non-zero values from their positions along the original K-dimension. Rather than storing all $K \times N$ entries, the kernel operates on two arrays:

```
float* g_mat_data; // sparse values: W×N (row-major)
int*  g_mat_index; // sparse indices along K: W×N (row-major)
```

where $W = K \cdot (1 - \text{sparsity})$ is the number of non-zero positions per column in the compressed dimension. For example, under $2:4$ sparsity (50%), we have $W = K/2$, yielding a $2\times$ reduction in the number of stored values.

This representation directly affects the memory access pattern. In dense GEMM, the kernel accesses $A \in \mathbb{R}^{M \times K}$ and B using regular strided loads; each thread can compute the global address of the elements it needs from the tile indices alone. In the compressed $N:M$ setting, the kernel must perform a two-stage access for each compressed row: first load the index specifying which original K-position is active, and only then gather the corresponding row of A.

In the baseline kernel, this is implemented as:

```
// inside the loop over the compressed W dimension
int* B_index_global_ptr =
   g_mat_index + K_SPARSE_BLOCK_START * N + N_BLOCK_START;

for (int i = 0; i < BLOCK_SIZE_K_SPARSE; i += A_STRIDES) {
   if (K_SPARSE_BLOCK_START + i + A_BLOCK_ROW_START < W) {
        int idx = *(B_index_global_ptr +
                    (i + A_BLOCK_ROW_START) * N);
       FETCH_FLOAT4(A_shared[(i + A_BLOCK_ROW_START) *
                     BLOCK_SIZE_M + A_BLOCK_COL_START]) =
            FETCH_FLOAT4(A_global_ptr[idx * M +
            A_BLOCK_COL_START]);
    }
}
```

Here, the index `idx` identifies a row of A along the original K-dimension, and the kernel uses a vectorized `float4` load to gather a short contiguous segment into shared memory. In contrast, the corresponding values of B for the same compressed rows are laid out contiguously in `g_mat_data`, so loads into the B tile are regular.

This two-stage (index $\rightarrow$ gather) access introduces a sequential dependency (values in A cannot be requested until indices are known), irregular global memory access for A, and the possibility of shared-memory bank conflicts when many threads concurrently populate the same shared tile. The remainder of the design is driven by the need to mitigate these effects while still exploiting the reduced arithmetic cost from sparsity.

3.2 Efficient Hierarchical Tiling Strategy

All three kernels share the same hierarchical tiling structure in (M, N) and a loop over the compressed dimension W. At the block level, $C \in \mathbb{R}^{M \times N}$ is partitioned into fixed tiles of size 32×32. Each thread block computes one such tile, with the block origin determined by `blockIdx.x` and `blockIdx.y`:

```
const int BLOCK_SIZE_M        = 32;  // tile in M
const int BLOCK_SIZE_N        = 32;  // tile in N
const int BLOCK_SIZE_K_SPARSE = 64;
const int THREAD_SIZE_M     = 4;  // per-thread tile in M
const int THREAD_SIZE_N     = 4;  // per-thread tile in N;

const int THREADS_PER_BLOCK =
    (BLOCK_SIZE_M / THREAD_SIZE_M) *
   (BLOCK_SIZE_N / THREAD_SIZE_N);  // 8×8 = 64 threads
```

Within the block, threads are logically arranged in an 8×8 grid. Each thread owns a 4×4 sub-tile of the 32×32 output tile and accumulates it in registers. This design provides sufficient parallelism to populate the SM while keeping per-thread register usage manageable.

The reduction over K is replaced by a reduction over the compressed dimension W, processed in tiles of size `BLOCK_SIZE_K_SPARSE`. For the baseline and prefetch kernels we use 64 compressed rows per tile, whereas the double-buffered kernel uses 32 rows to leave

shared memory for two B buffers. The loop over the compressed dimension has the form:

```
for (int K_SPARSE_BLOCK_START = 0;
         K_SPARSE_BLOCK_START < W;
        K_SPARSE_BLOCK_START += BLOCK_SIZE_K_SPARSE) {

    float* B_global_ptr =
     g_mat_data + K_SPARSE_BLOCK_START * N + N_BLOCK_START;
    int*   B_index_global_ptr =
     g_mat_index + K_SPARSE_BLOCK_START * N + N_BLOCK_START;

   // load A and B tiles into shared memory, then compute
}
```

For each tile along W, the kernel: (i) computes the correct offsets into the $W \times N$ compressed arrays, (ii) cooperatively loads the corresponding A and B tiles into shared memory, and (iii) applies the register-level microkernel to accumulate contributions to the local 32×32 output tile. The main departure from dense GEMM is that the reduction index ranges over compressed non-zero rows instead of all K positions and that the A tile is constructed via index-based gathers rather than simple strided loads.

3.3 Register-Level Computation and Accumulation

Within each sparse tile, computation is entirely register-resident. Each thread maintains:

- a short vector of A values of length `THREAD_SIZE_M` (here, 4),
- a short vector of B values of length `THREAD_SIZE_N` (also 4), and
- a 4×4 accumulator tile for C:

```
float A_reg[THREAD_SIZE_M]; // 4 entries per thread
float B_reg[THREAD_SIZE_N]; // 4 entries per thread
float C_reg[THREAD_SIZE_N][THREAD_SIZE_M] = {0.0f}; // 4×4
```

For each compressed row within the current tile, the thread first reads its portion of the A and B tiles from shared memory into registers and then computes a rank-one update:

```
for (int i = 0; i < tile_size; i++) {
    // load A and B fragments into registers
    for (int m = 0; m < THREAD_SIZE_M; m++) {
        A_reg[m] = A_shared[i * BLOCK_SIZE_M +
                            t_M * THREAD_SIZE_M + m];
    }
    for (int n = 0; n < THREAD_SIZE_N; n++) {
        B_reg[n] = B_shared[i * BLOCK_SIZE_N +
                            t_N * THREAD_SIZE_N + n];
    }

    // outer product: B_reg[n] * A_reg[m]
    for (int n = 0; n < THREAD_SIZE_N; n++) {
        for (int m = 0; m < THREAD_SIZE_M; m++) {
            C_reg[n][m] += B_reg[n] * A_reg[m];
        }
    }
}
```

This corresponds to a small outer-product microkernel:

$$C_{\text{local}} \mathrel{+}= b\, a^{\top},$$

where $a \in \mathbb{R}^4$ and $b \in \mathbb{R}^4$ are the per-thread fragments of A and B. Per thread and per iteration, this yields 16 fused multiply-adds using only eight scalar loads from shared memory, giving a reasonable arithmetic intensity at the thread level.

After all sparse tiles have been processed, each thread writes its 4×4 register tile back to global memory in column-major order:

```
for (int n = 0; n < THREAD_SIZE_N; n++) {
    for (int m = 0; m < THREAD_SIZE_M; m++) {
        int row = row_base + m;
        int col = col_base + n;
        g_data[col * M + row] = C_reg[n][m];
    }
}
```

This register blocking reduces the number of shared-memory accesses, at the cost of modestly higher register usage per thread, which is kept under control by fixing the 4×4 sub-tile size.

3.4 Leveraging $N{:}M$ Structure for Shared-Memory Access

Shared memory is used to stage tiles of both A and B for each sparse block. In the baseline and prefetch kernels, a single pair of tiles is allocated:

```
extern __shared__ float shared_mem[];
float* A_shared = shared_mem;
// A_shared size [BLOCK_SIZE_K_SPARSE][BLOCK_SIZE_M]
float* B_shared = A_shared +
    BLOCK_SIZE_M * BLOCK_SIZE_K_SPARSE;
```

The A tile is indexed by the compressed row in the current block along W and the local M coordinate in the 32×32 output tile. The B tile is indexed by the compressed row and the local N coordinate. Because `g_mat_data` is stored row-major in the compressed $W \times N$ layout, loads into the B tile are naturally coalesced and map cleanly to shared-memory banks if the leading dimension matches `BLOCK_SIZE_N`.

The irregular part lies in constructing the A tile. For each compressed row, the kernel first reads an index from `g_mat_index` and then gathers a short contiguous segment of the corresponding column of A into `A_shared`. In that sense, the $N : M$ pattern is helpful: it restricts which K-positions can appear in a local region of the compressed representation. Under $2{:}4$ sparsity, for example, each block of four consecutive K-positions contains exactly two non-zeros. When the compressed representation is laid out appropriately, the indices that appear within a given sparse tile must fall into a bounded range, which prevents the most pathological access patterns that fully unstructured sparsity could create.

In practice, this means that within a warp, many threads either access distinct banks or converge on the same address in shared memory. The latter case is serviced efficiently as a broadcast on recent NVIDIA architectures, avoiding full serialization. The choice of tile dimensions and leading dimensions in shared memory is therefore made to align the inner-most index with the M or N direction,

so that threads within a warp see regular strides in their shared-memory accesses even though the compressed-row sequence is determined by the index matrix.

3.5 Prefetch (Register-Based Latency Hiding)

The first optimization variant aims to hide part of the global-memory latency by register-based prefetching, without changing the shared-memory layout or the core microkernel. The idea is to speculatively access data for the next sparse tile so that the corresponding cache lines are likely to be resident when the tile is actually loaded into shared memory.

In the prefetch kernel, each iteration over the sparse tiles proceeds as follows:

(1) The current tile is loaded into `A_shared` and `B_shared` exactly as in the baseline kernel.
(2) If another tile remains, each thread uses the indices for the next tile to perform vectorized loads into temporary register arrays `prefetch_A` and `prefetch_B`, whose contents are never directly consumed by the computation:

```
float4 prefetch_A[16];
// enough for BLOCK_SIZE_K_SPARSE / A_STRIDES
float4 prefetch_B[16];
...
if (tile < total_tiles - 1) {
   int next_K_START = (tile + 1) * BLOCK_SIZE_K_SPARSE;
    float* next_A_ptr = g_vec + M_BLOCK_START;
    float* next_B_ptr = g_mat_data + next_K_START * N
                              + N_BLOCK_START;
  int*  next_B_index_ptr= g_mat_index + next_K_START * N
                              + N_BLOCK_START;
    int prefetch_count_A = 0, prefetch_count_B = 0;

    // prefetch A using indices for next tile
  for (int i = 0; i < BLOCK_SIZE_K_SPARSE; i += A_STRIDES) {
       if (next_K_START + i + A_BLOCK_ROW_START < W) {
            int idx = *(next_B_index_ptr + (i +
                A_BLOCK_ROW_START) * N);
            prefetch_A[prefetch_count_A++] =
                FETCH_FLOAT4(next_A_ptr[idx * M
                + A_BLOCK_COL_START]);
        }
    }

    // prefetch B contiguously
  for (int i = 0; i < BLOCK_SIZE_K_SPARSE; i += B_STRIDES) {
       if (next_K_START + i + B_BLOCK_ROW_START < W) {
            prefetch_B[prefetch_count_B++] =
                FETCH_FLOAT4(next_B_ptr
           [(i + B_BLOCK_ROW_START) * N +B_BLOCK_COL_START]);
        }
    }
}
```

(3) The kernel then executes the usual register-level outer-product microkernel for the current tile while the prefetched data resides in registers and, more importantly, the corresponding cache lines are warmed.

Because the prefetch buffers are only used for their side effect on the memory subsystem, they are explicitly marked as unused in the code to suppress compiler warnings. The benefit of this approach is inherently limited: index loads are still required before any prefetch of A can be issued, and the additional registers used for `prefetch_A` and `prefetch_B` increase register pressure and can reduce occupancy. Nevertheless, this variant provides a useful intermediate design for evaluating how much latency hiding can be achieved without changing the core shared-memory structure.

3.6 Double-Buffer with Asynchronous Copy and Split-K

The second optimization variant makes more aggressive use of modern NVIDIA GPU features—specifically the `cp.async` asynchronous global-to-shared copy instructions introduced on Ampere and supported on Ada Lovelace (RTX 4070)—by introducing double-buffering for the B tiles, together with optional Split-K parallelization along the compressed dimension.

In this kernel, shared memory is partitioned into one tile for A and two tiles for B:

```
const int BLOCK_SIZE_K_SPARSE = 32;   // smaller sparse tile
const int A_TILE_SIZE = BLOCK_SIZE_M * BLOCK_SIZE_K_SPARSE;
const int B_TILE_SIZE = BLOCK_SIZE_N * BLOCK_SIZE_K_SPARSE;

float* A_shared = shared_mem;
float* B_shared[2];
B_shared[0] = A_shared + A_TILE_SIZE;
B_shared[1] = B_shared[0] + B_TILE_SIZE;
```

At any given time, one B tile is the "read" buffer used for computation, and the other is the "write" buffer being filled with data for the next sparse tile. Asynchronous copies are issued using PTX-level intrinsics:

```
#define CP_ASYNC_CG(addr, ptr, bytes) \
    asm volatile("cp.async.cg.shared.global [%0],
      [%1], %2;\n" : : "r"(addr), "l"(ptr), "n"(bytes))

...

// prefetch next B-tile into write_buf
for (int i = 0; i < BLOCK_SIZE_K_SPARSE; i += B_STRIDES) {
    int row = i + B_BLOCK_ROW_START;
    if (NEXT_K_START + row < W) {
     float* dst = &B_shared[write_buf][row * BLOCK_SIZE_N +
                                  B_BLOCK_COL_START];
        float* src = next_B_global_ptr + row * N
            + B_BLOCK_COL_START;
      unsigned int addr = __cvta_generic_to_shared(dst);
        CP_ASYNC_CG(addr, src, 16);  // float4
    }
}
CP_ASYNC_COMMIT_GROUP();
```

While the asynchronous copies are in flight, the kernel performs the usual register-level computation on the current tile using `B_shared[read_buf]`. Before advancing to the next tile, it waits for all outstanding copies to complete and synchronizes:

```
CP_ASYNC_WAIT_ALL();
__syncthreads();

// swap read and write buffers
int tmp    = read_buf;
read_buf   = write_buf;
write_buf  = tmp;
```

This ping–pong scheme overlaps most of the global-memory traffic for B with arithmetic, reducing the exposed memory latency. The A tiles are still populated via synchronous index-based gathers because their addresses depend on the indices for each sparse row.

The same kernel also supports Split-K parallelization along the compressed dimension by using the third grid dimension:

```
const int SPLIT_K = 2;  // example
dim3 dimGrid(M / BLOCK_SIZE_M,
             N / BLOCK_SIZE_N,
             SPLIT_K);
```

The total number of sparse tiles is divided across `SPLIT_K` slices, and `blockIdx.z` identifies the slice that each block handles:

```
const int tiles_total    = (W + BLOCK_SIZE_K_SPARSE - 1) /
                           BLOCK_SIZE_K_SPARSE;
const int tiles_per_split = (tiles_total + SPLIT_K - 1) /
                           SPLIT_K;
const int tile_start     = split_id * tiles_per_split;
const int tile_end       = min(tiles_total, tile_start
                               + tiles_per_split);

// some slices may be empty
if (tile_start >= tile_end) return;
```

Each z-slice accumulates a partial result into its register tile of C. When `SPLIT_K = 1`, the kernel writes directly to `g_data` as in the baseline. When `SPLIT_K > 1`, partial sums are combined via atomic addition:

```
if (SPLIT_K == 1) {
    g_data[col * M + row] = C_reg[nn][mm];
} else {
    atomicAdd(&g_data[col * M + row], C_reg[nn][mm]);
}
```

Split-K can increase effective parallelism when M and N are small by exposing more independent work along the compressed dimension, but it introduces additional global-memory traffic and contention on the atomic updates. In this project, the double-buffered and Split-K kernel is evaluated against the baseline and prefetch variants using Nsight Compute to determine when the additional complexity of asynchronous buffering and atomic accumulation translates into real wall-clock speedups and when it does not.

4 Experimental Setup

4.1 Hardware and Software Environment

All experiments are conducted on a dedicated 64-bit Linux server (`cuda5`) at the NYU Courant Institute. The machine is equipped with an NVIDIA GeForce RTX 4070 GPU with 12,282 MiB of GDDR6X memory, Ada Lovelace architecture (compute capability `sm_89`), 5,888 CUDA cores, and a theoretical peak memory bandwidth of 504 GB/s. The system runs an NVIDIA driver from the 580.xx series and the CUDA 12.4 toolkit. The host CPU is a dual-socket Intel Xeon E5-2650 v2 at 2.60 GHz, providing 16 physical cores (8 per socket, hyperthreading disabled), with an aggregate 40 MiB L3 cache and a two-node NUMA configuration. The operating system is a 64-bit Linux distribution on `x86_64`; all experiments are executed in the same environment to avoid cross-run variability.

CUDA kernels are compiled with `nvcc` V12.4.131 targeting the native Ada architecture using

```
-arch=sm_89 -O3 -Xcompiler -Wall
```

which enables aggressive device- and host-side optimizations and generates SASS for `sm_89` without relying on JIT compilation from PTX at runtime. Dense baselines are implemented using NVIDIA cuBLAS (as shipped with CUDA 12.4), which serves both as an optimized dense GEMM reference and, for large matrices, as a correctness oracle for the sparse kernels. Detailed performance profiling is performed with NVIDIA Nsight Compute 2024.1.1 (`ncu`), using the "full" metric set and CSV export for post-processing.

4.2 Sparse Matrix Configuration and Kernel Parameters

All sparse experiments use an $N:M$ structured sparsity pattern with $N = 2$ and $M = 4$, i.e., exactly two non-zero elements out of every four consecutive weights along the K dimension. For a dense weight matrix $B \in \mathbb{R}^{K\times N}$ this corresponds to a sparsity level of 50% and reduces the number of stored weights to $W = K/2$ per column. The compressed representation stores the non-zero values in a row-major array of size $W \times N$ and a matching integer index array of size $W \times N$ that encodes the original K-positions of the non-zeros. Indices are constructed to strictly respect the $2:4$ pattern so that each group of four consecutive positions in the original K dimension contains exactly two indices.

All sparse kernel variants share the same high-level tiling structure. At the block level, the output matrix $C \in \mathbb{R}^{M\times N}$ is partitioned into fixed tiles of size 32×32 in (M, N), with each thread block computing one such tile. At the thread level, each of the 64 threads in a block (arranged conceptually as an 8×8 grid) computes a 4×4 sub-tile of C and maintains all 16 accumulators in registers. The reduction over the compressed dimension W is performed in tiles of size `BLOCK_SIZE_K_SPARSE`. For the baseline and prefetch kernels, `BLOCK_SIZE_K_SPARSE` is set to 64, whereas the double-buffered kernel uses a smaller sparse tile of size 32 to leave sufficient shared memory for two B buffers in addition to the A tile. The grid configuration and shared-memory allocation are chosen so that all variants use identical (M, N) tiling and only differ in the way they schedule and overlap sparse tiles.

The double-buffered kernel further introduces a Split-K style partitioning along the compressed dimension: the grid's z-dimension is used to divide the sparse tiles evenly among `SPLIT_K` independent slices. In the experiments reported here, `SPLIT_K` is set to 2, so each logical (M, N) tile is computed by two thread blocks that accumulate partial results over disjoint subsets of W and then combine them via atomic additions to the global output matrix.

4.3 Data Generation and Correctness Verification

For each problem size, the dense input matrix $A \in \mathbb{R}^{M \times K}$ is initialized with pseudo-random floating-point values in $[0, 1)$ and stored in column-major order to match cuBLAS and the sparse kernels. The sparse weight matrix B is generated by first sampling random non-zero values in $[0, 1)$ and then applying a deterministic $2:4$ pruning rule along the K dimension to enforce exactly two non-zero elements in every consecutive group of four entries. The resulting non-zero values and their positions are written into the compressed value and index arrays of size $W \times N$ in row-major layout. The output matrix $C \in \mathbb{R}^{M \times N}$ is zero-initialized before each kernel launch to avoid residual contributions across runs or variants.

Correctness is verified against different references depending on the problem size. For matrices smaller than 4096×4096, a sequential CPU implementation of dense SpMM is used as the ground truth. For larger matrices, a dense cuBLAS GEMM with the logically equivalent dense weight matrix serves as reference, as a direct CPU computation would be prohibitively slow. After each run, the sparse kernel output is compared element-wise against the reference, and the result is accepted if the relative error for every element is below 10^{-3}; otherwise the configuration is flagged as incorrect and excluded from performance analysis.

4.4 Benchmarking Methodology

Performance is measured using a simple but standardized protocol based on CUDA events. For each matrix size, the benchmark first executes a warm-up phase of several kernel launches to bring the GPU into a steady thermal and power state, trigger any remaining JIT compilation, and populate page tables and caches. The number of warm-up iterations ranges from five to ten, with larger sizes using more iterations. After warm-up, a timing phase is executed in which the kernel is launched repeatedly and the elapsed time is measured via CUDA events placed immediately before and after the kernel. The measured time excludes memory allocation and host–device transfers; only the kernel execution is timed. The number of timed iterations varies between 20 and 100 depending on matrix size, with small sizes using more iterations to reduce variance.

Matrix dimensions are swept over a logarithmic grid to probe cache-resident and memory-bound regimes. The experiments cover "small" problems with $M = N = K \in \{64, 128, 256, 512\}$, where working sets largely fit into L2; "medium" problems with $M = N = K \in \{1024, 2048\}$, which emphasize the transition from cache to DRAM bandwidth; and "large" problems with $M = N = K \in \{4096, 8192\}$, which are strongly memory-bound and closer to realistic inference workloads. For each size, the average kernel time over the timed iterations is reported and converted to effective TFLOP/s and speedup factors relative to both a sequential CPU implementation and a dense cuBLAS baseline.

4.5 Profiling Methodology

To understand the performance behavior beyond wall-clock time, selected matrix sizes are profiled using NVIDIA Nsight Compute. A custom script invokes `ncu` with a regular-expression kernel filter that matches the three sparse kernels (`kernel_nmsparse_baseline`, `nmsparse_prefetch_kernel`, and `nmsparse_double_buffer_kernel`) and collects more than twenty hardware performance counters into CSV files. The metrics are chosen to cover five aspects of performance: memory throughput, compute utilization, memory access efficiency, shared-memory behavior, and warp stall reasons. Table 3 summarizes the main categories and example Nsight Compute counters used in the analysis.

A Python analysis script parses the raw CSV output, aggregates metrics across kernel launches, and produces summary tables that report means, minima, and maxima for each metric and variant. These summaries are then imported into plotting and spreadsheet tools for cross-kernel comparison.

4.6 Experiments Conducted

The evaluation comprises three complementary experiments. The first experiment establishes a baseline by measuring execution time and throughput for all sparse kernel variants across the full matrix-size sweep, together with the sequential CPU implementation and the dense cuBLAS GEMM. This experiment verifies correctness, quantifies raw speedups over the CPU, and measures how closely the sparse kernels approach (or surpass) dense cuBLAS performance as matrix size grows.

The second experiment focuses on detailed profiling at representative sizes ($M = N = K \in \{256, 2048\}$). These configurations are chosen because the benchmark results exhibit relatively pronounced performance differences between kernel variants at these sizes. For each of them, Nsight Compute is used to collect the full metric set for the baseline, prefetch, and double-buffered kernels. The resulting profiles are analyzed to identify bottlenecks such as DRAM saturation, poor cache utilization, shared-memory bank conflicts, or high fractions of warps stalled on memory dependencies. This experiment explains *why* some optimization variants succeed or fail rather than merely reporting their net speedup.

The third experiment is an optimization-effectiveness study that directly compares the prefetch and double-buffered kernels against the baseline. For each matrix size, the relative speedup or slowdown is computed, and the corresponding profiling data are examined to correlate performance changes with shifts in bottlenecks. In particular, this experiment investigates under which conditions register-based prefetching provides measurable latency hiding, when asynchronous double-buffering of B reduces exposed memory latency, and when Split-K parallelization is outweighed by the overhead of additional atomic accumulations.

4.7 Reproducibility

All results in this report can be reproduced using a simple make-based build and run system. The code is compiled with

```
make clean all
```

which produces a benchmark binary `test_sparse_kernels`. A single-size benchmark can then be run as

```
./build/test_sparse_kernels 1024 1024 1024 10 100
```

where the last two arguments specify the number of warm-up and timed iterations. A full size sweep is executed via

```
make run-all
```

Table 1: Nsight Compute metric groups used in profiling.

Aspect	Purpose	Example Nsight Compute metrics
Memory throughput	Quantify global and cache bandwidth utilization	`dram__throughput.avg.pct_of_peak_sustained_elapsed`, `l1tex__t_bytes.sum`, `lts__t_sectors_op_read.sum`
Compute utilization	Measure SM activity and instruction throughput	`sm__throughput.avg.pct_of_peak_sustained_elapsed`, `smsp__inst_executed.avg.per_cycle_active`, `sm__warps_active.avg.pct_of_peak_sustained_active`
Memory access efficiency	Assess global-load coalescing and transaction efficiency	`l1tex__average_t_sectors_per_request_pipe_lsu_mem_global_op_ld.ratio`, `smsp__sass_average_data_bytes_per_sector_mem_global_op_ld.pct`
Shared-memory behavior	Quantify bank conflicts for loads and stores	`l1tex__data_bank_conflicts_pipe_lsu_mem_shared_op_ld.sum`, `l1tex__data_bank_conflicts_pipe_lsu_mem_shared_op_st.sum`
Warp stall reasons	Attribute warp issue stalls to different bottlenecks	`smsp__average_warps_issue_stalled_long_scoreboard_per_issue_active.pct`, `smsp__average_warps_issue_stalled_barrier_per_issue_active.pct`, `smsp__average_warps_issue_stalled_drain_per_issue_active.pct`

For profiling, a wrapper script `profile_ncu.sh` invokes Nsight Compute with the appropriate metric set and kernel filters, for example:

```
./profile_ncu.sh 1024 1024 1024
```

which generates both raw metric CSV files and summarized statistics. All experiments reported in the following section are obtained using this workflow.

5 Results and Analysis

5.1 Scaling Across Matrix Sizes

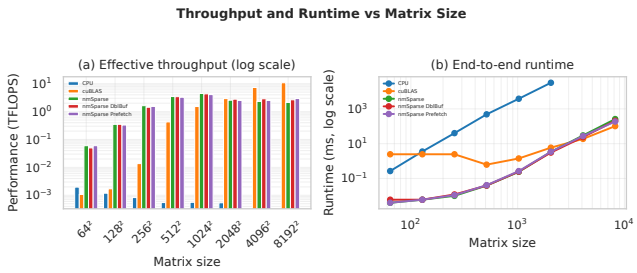
Figure 2a reports effective throughput (TFLOPS) and end-to-end runtime for all implementations across $M = N = K \in \{64, 128, \ldots, 8192\}$. Together with the speedup curves in Figure 2b, these results show a clear pattern: the custom $N:M$ sparse kernels deliver massive speedups over the CPU baseline across almost the entire sweep and outperform dense cuBLAS on small and medium matrices, but their advantage over cuBLAS disappears, and eventually reverses, once the problem size is large enough for dense GEMM to fully utilize the GPU.

At the smallest size (64^2), dense cuBLAS is still operating in a launch–overhead regime. Its effective throughput is only on the order of 10^{-3} TFLOPS and it is slower than the sequential CPU baseline. In contrast, the $N:M$ kernels already reach $\approx 6\times10^{-2}$ TFLOPS, about one to two orders of magnitude higher. Figure 2b(a) shows the same effect in relative terms: cuBLAS achieves a speedup of only $\approx 10^{-1}$ over the CPU at 64^2, while the sparse kernels are close to $10^2\times$ faster. This behavior follows directly from the design in Section 3: by working in the compressed dimension $W = K(1 - \text{sparsity})$ and executing a fixed 32×32 / 4×4 tiling strategy, the custom kernels perform roughly half as many floating-point operations as dense GEMM under $2:4$ sparsity and avoid any autotuning or complex launch setup. For such small matrices, the fixed launch cost of cuBLAS dominates the total runtime, while the simple, directly inlined sparse kernel turns its reduced FLOP count into real wall-clock speedups.

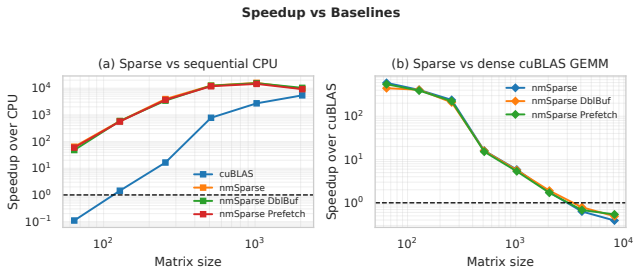
The Nsight Compute profile at 256^2 provides a more detailed view of this regime. The sparse kernel launches only 128 warps and keeps about 5.8% of warps active on average, with `inst_per_cycle` ≈ 0.12 and no observed store-side bank conflicts. The dominant stall reason at this size is long scoreboard stalls (≈ 421), reflecting the latency of global loads needed to construct the A tile via the index–gather pattern from Section 5.1, while `drain_stalls` remains modest (≈ 7.4). This matches the intended behavior of the register-level outer-product microkernel: each thread performs the rank-one update C_{local} += $ba^\top$ using fragments loaded from `A_shared` and `B_shared`, but the small matrix size means there are only a few W-tiles and thus relatively little work per launch. Even in this underutilized regime, skipping half the multiplies and avoiding heavyweight library machinery allows the sparse implementation to achieve dramatically higher throughput than both CPU and cuBLAS.

As matrix size grows into the medium regime (512^2–2048^2), the fixed launch cost becomes negligible for both dense and sparse implementations, and the hierarchical tiling described in Section 3.2 is fully exercised. The sparse kernels reach their peak effective throughput around 1024^2 (roughly 4–5 TFLOPS) and then flatten or slightly decline, whereas cuBLAS continues to scale from 0.43 TFLOPS at 512^2 to 2.9 TFLOPS at 2048^2 (Figure 2a(a)). The speedup-over-cuBLAS curve in Figure 2b(b) shrinks accordingly: the $N:M$ kernel is about 16× faster than cuBLAS at 512^2, roughly 5.9× faster at 1024^2, and only 1.7–1.9× faster at 2048^2. Across the same range, the speedup over the CPU baseline remains consistently large: the sparse kernels outperform the sequential implementation by roughly three to four orders of magnitude, confirming that the compressed representation and register-blocked microkernel successfully translate theoretical FLOP savings into practical speedups.

The Nsight profile at 2048^2 helps explain why the sparse curves start to bend while cuBLAS continues to climb. For the sparse kernel, SM throughput increases to 35.51 with `inst_per_cycle` ≈ 0.36 and $\approx 20\%$ of warps active, and the total number of warps launched jumps to 8192. At the same time, the kernel generates over 1.2×10^9 bytes of L1 traffic per run and incurs a large number of shared-memory bank conflicts on stores ($\approx 1.17 \times 10^6$). Both effects are consequences of the two-stage access pattern for A and the compressed layout of B from Section 5.1: for each compressed row, the kernel must load an index from `g_mat_index`, compute the corresponding row in A, then perform a vectorized gather into



(a) Effective throughput and runtime vs. matrix size.



(b) Speedup relative to CPU and cuBLAS.

Figure 2: Scaling behavior of dense cuBLAS and the custom $N:M$ sparse kernels on the RTX 4070. Panel (a) shows effective throughput in TFLOPS (left) and end-to-end runtime in milliseconds (right) on logarithmic axes for the CPU, cuBLAS, and the three sparse kernels. Panel (b) shows speedup over the sequential CPU baseline (left) and over cuBLAS (right), both on log–log scales with a horizontal $1\times$ reference line. The $N:M$ kernels provide large GPU speedups over the CPU and substantial gains over cuBLAS for small and medium matrices, then converge toward dense performance as matrix size grows.

Metric	256^3 Baseline	256^3 DblBuf	Ratio	2048^3 Baseline	2048^3 DblBuf	Ratio
SM throughput (%)	7.8	19.4	2.50×	35.5	81.0	2.28×
DRAM throughput (%)	9.4	22.3	2.38×	5.2	9.4	1.82×
Active warps (%)	5.8	10.4	1.79×	20.5	28.7	1.40×
Warps launched	128	256	2.0×	8192	16384	2.0×
Inst / cycle	0.123	0.162	1.31×	0.360	0.480	1.33×
Drain stalls (%)	7.4	68.3	9.2×	1.04	2.85	2.74×
Long scoreboard (%)	421.4	405.9	0.96×	264.4	290.5	1.10×
L1 traffic (MB)	3.41	4.85	1.42×	1275.1	1543.5	1.21×
Store bank conflicts	0.0	61.5	∞	1166393	834044	0.72×

Figure 3: Nsight Compute key metrics for the baseline and double-buffered $N:M$ sparse kernels at 256^3 and 2048^3. The table reports SM/DRAM throughput, occupancy, stall breakdown, L1 traffic, and store bank conflicts, together with Double-Buffer/Baseline ratios.

`A_shared`, before finally feeding the register-level outer-product. This extra metadata and indirection increases the number of bytes moved per useful FMA compared to dense GEMM, lowering the effective arithmetic intensity even though the kernel performs fewer FLOPs overall. The profiler shows this shift as a combination of high L1 traffic, moderate DRAM throughput, and long-scoreboard stalls that remain non-negligible even when `drain_stalls` drops to ≈ 1.

For the largest sizes (4096^2 and 8192^2), the plots in Figure 2a indicate that all of the sparse implementations saturate between 2.1 and 2.9 TFLOPS, while cuBLAS continues to scale up to 7.3 TFLOPS at 4096^2 and 10.8 TFLOPS at 8192^2. The speedup-over-cuBLAS curve in Figure 2b(b) drops below 1, meaning that dense GEMM now executes the *full* dense FLOP count faster than the $N:M$ kernels can process the reduced workload. By this point, the 32×32 / 4×4 tiling and register microkernel from Section 3 are doing exactly what they were designed to do: the SMs are well utilized and the outer products are computed from register-resident tiles. The limiting factor is no longer parallelism, but memory traffic. The compressed representation doubles the logical footprint of B by storing both values and indices and introduces irregular gathers from A, so each useful FMA is supported by more bytes of data movement than in the dense case. Once M, N, and W are large enough that the A and B tiles can no longer benefit from strong L2 reuse, the sparse kernels become effectively bandwidth-bound, while cuBLAS continues to increase its arithmetic intensity through large, regular tiles and stride-based access patterns.

Overall, the scaling results confirm the goals of the project. For small and medium matrices, the custom $N:M$ kernels exploit reduced FLOP counts and a lightweight tiling strategy to achieve enormous speedups over the CPU baseline and substantial gains over cuBLAS. For very large matrices, the same compressed representation and index-based gathering that enable those gains at small scale introduce enough extra memory traffic to push the kernels against the bandwidth ceiling, at which point a highly tuned dense GEMM eventually catches up and surpasses them.

5.2 Double Buffer vs. Baseline

Figure 4(a) plots the performance of the double-buffered kernel relative to the nmSparse baseline across matrix sizes, and Figure 4(b) highlights which sparse variant achieves the best runtime at each size. Up to 1024^2, the double-buffered kernel closely tracks or slightly lags the baseline; it only starts to pull ahead at 2048^2 and becomes clearly faster at 4096^2 and 8192^2. This is consistent with the raw TFLOPS and runtime curves: at 2048^2 the double-buffered kernel reaches 2.79 TFLOPS versus 2.56 TFLOPS for the baseline and reduces runtime from 3.35 ms to 3.08 ms ($\approx 8\%$ speedup), with even larger relative gains at 4096^2 and 8192^2.

Table 3 explains this behavior. At the smaller size (256^2), the double-buffered kernel shows much higher instantaneous activity than the baseline: DRAM throughput increases from 9.36 to 22.27, SM throughput from 7.78 to 19.43, active warps from 5.79% to 10.38%, and `inst_per_cycle` from 0.12 to 0.16. However, these gains come with a nearly 9× increase in `drain_stalls` (7.44 to 68.31) and additional store-bank conflicts. With only a few sparse tiles per block, the pipeline set-up and `cp.async` commit/wait phases are not amortized, so the kernel spends a disproportionate fraction of time draining the asynchronous copy machinery instead of exploiting its overlap, and end-to-end runtime is slightly worse than the baseline.

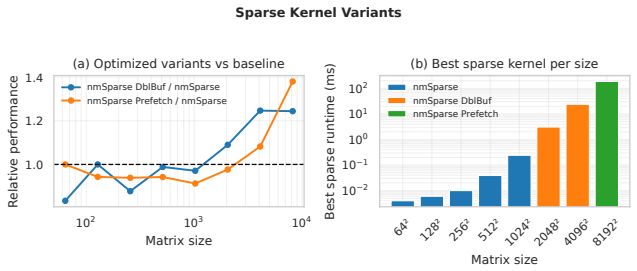


Figure 4: Sparse-kernel variants on the RTX 4070. The left panel (generated by `plot_sparse_variants`) reports the performance of the double-buffered and prefetch kernels relative to the nmSparse baseline across matrix sizes, and the right panel shows the best sparse runtime per size, with bar colors indicating which variant wins at each point.

At 2048^2, the same mechanism finally pays off. Relative to the baseline, the double-buffered kernel roughly doubles the number of warps launched (from 8192 to 16384) *because* the configuration uses `SPLIT_K` = 2, effectively replicating the grid along the compressed W dimension. On top of this extra parallelism from Split-K, the asynchronous double-buffering raises SM throughput from 35.51 to 81.02, DRAM throughput from 5.16 to 9.38, active warps from 20.46% to 28.72%, and `inst_per_cycle` from 0.36 to 0.48. The cost of managing asynchronous copies is still visible in `drain_stalls` (which increase from 1.04 to 2.85), but that increase is modest compared to the 2×–2.5× rise in SM and DRAM utilization. Notably, store-bank conflicts *decrease* from 1.17×10^6 to 8.34×10^5, indicating that the ping–pong layout for the B tiles produces slightly more regular shared-memory access patterns than the single-buffer baseline. In other words, once the problem is large enough to keep many sparse tiles in flight, the double-buffered kernel does exactly what it was designed to do: overlap global loads of B with the register-level outer-product microkernel, increase the number of resident warps, and convert that extra concurrency into higher sustained throughput and shorter runtime.

5.3 Prefetch vs. Baseline

The register-prefetch variant was intended as a lighter-weight attempt at latency hiding: for each sparse tile, it speculatively loads data for the next tile into temporary register arrays `prefetch_A` and `prefetch_B`, relying on the side effect that the corresponding cache lines are warmed when the next tile is actually loaded into shared memory. Figures 4(a) and (b) show that this design remains very close to the baseline across all sizes: throughput differs by at most a few percent up to 2048^2, and the prefetch kernel only becomes clearly better at the very largest sizes, where it slightly outperforms the baseline but still trails the double-buffered kernel except at 8192^2.

The profiler confirms that these differences are small and mostly unfavorable. At 256^2, the prefetch kernel has slightly *lower* SM throughput (7.25 vs. 7.78), slightly lower DRAM throughput (8.24 vs. 9.36), the same active-warp percentage, and a marginally reduced IPC (0.11 vs. 0.12). `drain_stalls` stays essentially unchanged (about 7.4), while `long_scoreboard_stalls` increase from 421.35 to 483.99, indicating more time waiting on long-latency memory operations. Importantly, the reported L1 traffic is *identical* to the baseline at this size, which suggests that register prefetching neither reduces the exposed latency nor meaningfully changes the amount of data moved per useful FMA.

The same pattern appears at 2048^2. SM and DRAM throughput, active-warp percentage, and IPC for the prefetch kernel all trail the baseline slightly, while `long_scoreboard_stalls` rise from 264.42 to 323.48 and `drain_stalls` and L1 traffic remain effectively the same. Given the index–gather dependency, the kernel still has to fetch index entries before any prefetch for A can be issued, so prefetching does not fully decouple memory access from computation. In practice, the variant increases latency-related stalls without increasing useful work per cycle, which explains why its wall-clock behavior closely tracks, and occasionally underperforms, the simpler single-buffer design.

The scaling curves and Nsight metrics point to a clear division of labor among the variants. The baseline kernel is the right choice for tiny matrices, where any additional control flow is pure overhead. For small to medium sizes, the double-buffered kernel is the only variant that consistently turns extra complexity into higher sustained throughput and modest but real runtime gains. The register-prefetch kernel, by contrast, largely preserves the memory traffic and utilization characteristics of the baseline while slightly worsening long-latency stalls, so its main value is as a negative control: it shows that modest cache warming alone is not sufficient to overcome the index–gather latency in this $N:M$ sparse setting.

6 Conclusion

This project implements a custom $N:M$ sparse SpMM kernel together with two optimization variants—a register-prefetch design and a cp.async double-buffered design—and evaluates them against a sequential CPU baseline and dense cuBLAS on an RTX 4070 using Nsight Compute and got following key conclusion:

- A carefully engineered $N:M$ sparse SpMM that combines compressed storage, hierarchical 32×32 tiling, and a register-level outer-product microkernel can turn reduced FLOP counts into *real* end-to-end gains: the implemented nmSparse baseline achieves 10^2–$10^4\times$ speedup over the CPU and strong speedups over cuBLAS for small and medium matrices, as long as sparsity is high (e.g., $2:4$) and the problem fits the fixed-tile regime.
- These gains are strongly regime-dependent. As matrix size grows, the compressed representation and index–gather pattern increase metadata and memory traffic per useful FMA, pushing the sparse kernels into a bandwidth-bound regime where dense cuBLAS eventually matches and then surpasses them in absolute throughput despite performing the full dense FLOP count.
- Among the implemented variants, only the cp.async double-buffered kernel consistently converts extra complexity into higher sustained SM/DRAM utilization and measurable runtime improvements at medium and large sizes; the register-prefetch kernel leaves the memory-traffic profile essentially unchanged and slightly worsens long-latency stalls, serving

mainly as a negative control that highlights the limits of lightweight cache warming in this index–gather–dominated $N:M$ sparse setting.

References

[1] Arash Ashari, Naser Sedaghati, John Eisenlohr, Srinivasan Parthasarath, and P. Sadayappan. 2014. 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*, 781–792. doi:10.1109/SC.2014.69.

[2] Nathan Bell and Michael Garland. 2008. Efficient Sparse Matrix-Vector Multiplication on CUDA. NVIDIA Technical Report NVR-2008-004. NVIDIA Corporation, (Dec. 2008).

[3] Aydin Buluç and John R. Gilbert. 2012. Parallel sparse matrix-matrix multiplication and indexing: implementation and experiments. *SIAM Journal on Scientific Computing*, 34, 4, C170–C191. eprint: https://doi.org/10.1137/110848244. doi:10.1137/110848244.

[4] Jack Choquette, Wishwesh Gandhi, Olivier Giroux, Nick Stam, and Ronny Krashinsky. 2021. Nvidia a100 tensor core gpu: performance and innovation. *IEEE Micro*, 41, 2, 29–35. doi:10.1109/MM.2021.3061394.

[5] Jack J. Dongarra, Piotr Luszczek, and Antoine Petitet. 2003. The linpack benchmark: past, present and future. *Concurrency and Computation: Practice and Experience*, 15, 9, 803–820. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/cpe.728. doi:https://doi.org/10.1002/cpe.728.

[6] Trevor Gale, Matei Zaharia, Cliff Young, and Erich Elsen. 2020. Sparse gpu kernels for deep learning. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (SC '20) Article 17. IEEE Press, Atlanta, Georgia, 14 pages. ISBN: 9781728199986.

[7] Juan Gomez-Luna, Jose Maria Gonzalez-Linares, Jose Ignacio Benavides Benitez, and Nicolas Guil Mata. 2013. Performance Modeling of Atomic Additions on GPU Scratchpad Memory. *IEEE Transactions on Parallel & Distributed Systems*, 24, 11, (Nov. 2013), 2273–2282. doi:10.1109/TPDS.2012.319.

[8] Song Han, Huizi Mao, and William Dally. 2016. Deep compression: compressing deep neural networks with pruning, trained quantization and huffman coding. In (Oct. 2016).

[9] Song Han, Jeff Pool, John Tran, and William J. Dally. 2015. Learning both weights and connections for efficient neural networks. In *Proceedings of the 29th International Conference on Neural Information Processing Systems - Volume 1* (NIPS'15). MIT Press, Montreal, Canada, 1135–1143.

[10] Sunpyo Hong and Hyesoon Kim. 2010. An integrated gpu power and performance model. *SIGARCH Comput. Archit. News*, 38, 3, (June 2010), 280–289. doi:10.1145/1816038.1815998.

[11] Jiajia Li, Guangming Tan, Mingyu Chen, and Ninghui Sun. 2013. Smat: an input adaptive auto-tuner for sparse matrix-vector multiplication. *SIGPLAN Not.*, 48, 6, (June 2013), 117–126. doi:10.1145/2499370.2462181.

[12] Weifeng Liu and Brian Vinter. 2014. An efficient gpu general sparse matrix-matrix multiplication for irregular data. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, 370–381. doi:10.1109/IPDPS.2014.47.

[13] Weifeng Liu and Brian Vinter. 2015. Csr5: an efficient storage format for cross-platform sparse matrix-vector multiplication. In *Proceedings of the 29th ACM on International Conference on Supercomputing* (ICS '15). Association for Computing Machinery, Newport Beach, California, USA, 339–350. ISBN: 9781450335591. doi:10.1145/2751205.2751209.

[14] Asit Mishra, Jorge Albericio Latorre, Jeff Pool, Darko Stosic, Dusan Stosic, Ganesh Venkatesh, Chong Yu, and Paulius Micikevicius. 2021. Accelerating sparse deep neural networks. (2021). https://arxiv.org/abs/2104.08378 arXiv: 2104.08378 [cs.LG].

[15] Yusuke Nagasaka, Akira Nukada, and Satoshi Matsuoka. 2016. Adaptive multi-level blocking optimization for sparse matrix vector multiplication on gpu. *Procedia Computer Science*, 80, 131–142. International Conference on Computational Science 2016, ICCS 2016, 6-8 June 2016, San Diego, California, USA. doi:https://doi.org/10.1016/j.procs.2016.05.304.

[16] NVIDIA Corporation. 2024. Cublas library. https://docs.nvidia.com/cuda/cublas/. CUDA Toolkit Documentation, accessed 2025-11-19. (2024).

[17] NVIDIA Corporation. 2024. Cusparse library. https://docs.nvidia.com/cuda/cusparse/index.html. CUDA Toolkit Documentation, accessed 2025-11-19. (2024).

[18] Juan C. Pichel, Francisco F. Rivera, Marcos Fernández, and Aurelio Rodríguez. 2012. Optimization of sparse matrix–vector multiplication using reordering techniques on gpus. *Microprocessors and Microsystems*, 36, 2, 65–77. SPECIAL ISSUE -EXPLOITATION OF HARDWARE ACCELERATORS. doi:https://doi.org/10.1016/j.micpro.2011.05.005.

[19] Jeff Pool and Chong Yu. 2021. Channel permutations for n:m sparsity. In *Advances in Neural Information Processing Systems*. M. Ranzato, A. Beygelzimer, Y. Dauphin, P.S. Liang, and J. Wortman Vaughan, (Eds.) Vol. 34. Curran Associates, Inc., 13316–13327. https://proceedings.neurips.cc/paper_files/paper/2021/file/6e8404c3b93a9527c8db241a1846599a-Paper.pdf.

[20] Vasily Volkov and James W. Demmel. 2008. Benchmarking gpus to tune dense linear algebra. In *SC '08: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, 1–11. doi:10.1109/SC.2008.5214359.

[21] Yaqi Xia, Weihu Wang, Donglin Yang, Xiaobo Zhou, and Dazhao Cheng. 2025. Voltrix: sparse matrix-matrix multiplication on tensor cores with asynchronous and balanced kernel optimization. In *Proceedings of the 2025 USENIX Conference on Usenix Annual Technical Conference* (USENIX ATC '25) Article 42. USENIX Association, Boston, MA, USA, 16 pages. ISBN: 978-1-939133-48-9.

[22] Xiuxia Zhang, Guangming Tan, Shuangbai Xue, Jiajia Li, Keren Zhou, and Mingyu Chen. 2017. Understanding the gpu microarchitecture to achieve bare-metal performance tuning. *SIGPLAN Not.*, 52, 8, (Jan. 2017), 31–43. doi:10.1145/3155284.3018755.

[23] Aojun Zhou, Yukun Ma, Junnan Zhu, Jianbo Liu, Zhijie Zhang, Kun Yuan, Wenxiu Sun, and Hongsheng Li. 2021. Learning n:m fine-grained structured sparse neural networks from scratch. (2021). https://arxiv.org/abs/2102.04010 arXiv: 2102.04010 [cs.CV].