Blocked Dynamic Sparse Attention

Mehul Goel (mehulg), Saransh Agrawal (saransh2)
Website: https://mehulgoel873.github.io/15418-final-project/

Summary

As modern Large Language Models scale to massive context lengths (now, millions of tokens), the standard attention mechanism is a severe computational bottleneck. Exploiting sparsity in the attention matrix, where many token interactions produce mathematically small outputs, reduces both training and inference costs. Research has demonstrated that transformers can maintain over 95% accuracy is maintained with 90% sparsity. However, efficiently parallelizing sparse operations on GPUs is non-trivial. Notable challenges include severe load imbalance, warp divergence, and uncoalesced memory accesses that perform poorly under the hardware's SIMT execution model.

We enforce sparsity at a block-level granularity (G) using a Blocked Compressed Sparse Row (BCSR) format to preserve contiguous memory accesses, vectorization, and cache locality. We implemented three core kernels in CUDA: Sampled Dense-Dense Matrix Multiplication (SDDMM), Sparse Softmax, and Sparse-Dense Matrix Multiplication (SpMM), focusing on dynamic warp-level active-coordinate gathering, static partitioned scheduling, and custom row-interleaved memory layouts to hide latency and maintain Streaming Multiprocessor (SM) occupancy.

Our end-to-end evaluations on an RTX 6000 demonstrate that at large context lengths ($N \geq 16384$) and coarse granularities ($G \geq 8$), our sparse implementation effectively approaches theoretical maximum speedups, scaling proportionally to $\frac{1}{1-p}$. Broken down by kernel, the dynamic SDDMM successfully eliminates wasted arithmetic but exhibits shared-memory bandwidth bottlenecks at larger scales; the heavily memory-bound Sparse Softmax achieves granularity-invariant speedups through optimized, coalesced reduction passes, especially at large context lengths; and the SpMM kernel scales cleanly by maximizing data reuse along aligned, contiguous block reads.

Background

Attention

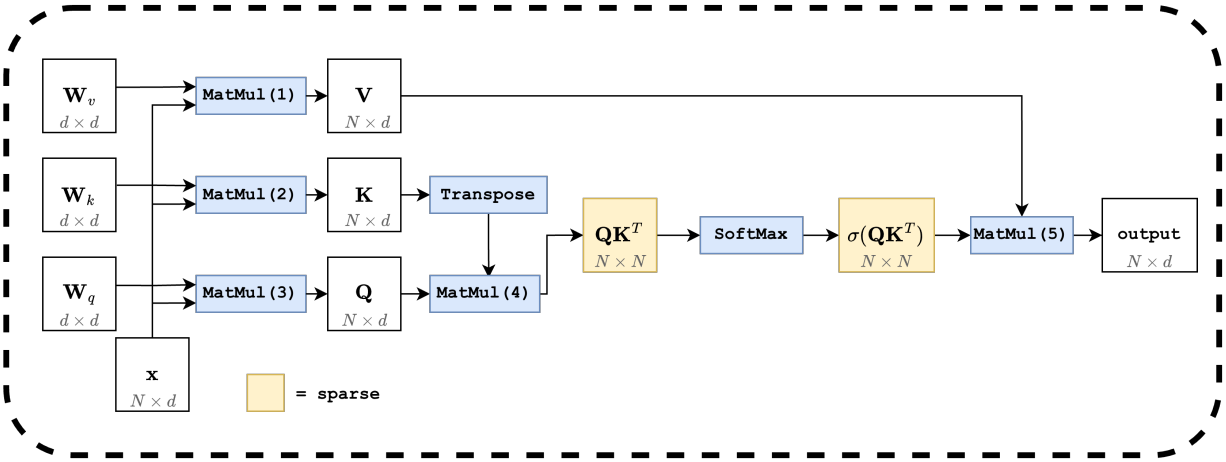


Figure 1: The attention block. Three projections (MatMuls (1)–(3)) produce Q, K, V; MatMul (4) and the softmax operate on an $N \times N$ matrix (highlighted), and MatMul (5) collapses back to $N \times d$.

Modern transformers process a sequence of tokens by repeatedly applying an *attention* operation. Each token is represented as a d-dimensional embedding vector, so the input to an attention layer is a matrix $\mathbf{x} \in \mathbb{R}^{N \times d}$, where N is the number of tokens (the *context length*) and d is the embedding dimension.

Figure 1 sketches the full attention block. The input $\mathbf{x}$ is first projected by three learned $d \times d$ weight matrices $\mathbf{W}_q, \mathbf{W}_k, \mathbf{W}_v$ (MatMuls (1)–(3)) into a query Q, a key K, and a value V, each of shape $N \times d$. The layer then computes

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d}}\right) V.$$

MatMul (4) forms the $N \times N$ score matrix QK^T, a row-wise softmax turns these scores into normalized weights, and MatMul (5) multiplies them by V.

The two highlighted stages in Figure 1 scale with the *square* of the sequence length. MatMul (4) materializes an $N \times N$ matrix, the softmax sweeps over it row by row, and MatMul (5) reads it again. In practical settings $N \gg d$. Today's frontier models target context lengths of a million tokens or more, while d is typically only 64 or 128 per head, so a single attention layer would touch on the order of 10^{12} entries in its intermediate matrix. Both the arithmetic and the memory traffic of storing and revisiting an $N \times N$ matrix become impractical at this scale.

Each block in the figure maps well to a GPU on its own. However, once you introduce sparsity to get rid of the operating on N^2 elements, the parallelization of the kernels becomes significantly more challenging.

Sparsity

Analysis of attention kernels in transformers has shown that the score matrix $\sigma(QK^T)$ produced by attention often has abundant sparsity: most of its entries are very small after the row-wise

softmax, and the output of MatMul (5) is dominated by a small fraction of the columns. This has motivated a broad family of *sparse attention* schemes that fall into two camps. Static patterns (such as local windows, strided, or block-diagonal masks) fix a sparsity structure ahead of time. Dynamic patterns (such as top-k or learned routing) choose which entries to keep based on the input itself, often giving better accuracy at a given sparsity level at the cost of more irregular computation [2]. In this paper, we study dynamic patterns as they often are harder to parallelize and are becoming increasingly important.

For our randomly-generated sparse masks, each cell is marked sparse with probability p, where p is the fraction of sparse entries. To match the granularity at which transformers in practice handle sparsity (which is in turn motivated by the GPU hardware architecture) [1], the mask is defined at a *block* level rather than per-element: a granularity of G means we partition the $N \times N$ matrix into $G \times G$ tiles and label each tile as fully dense or fully sparse. Although sparsity reduces the total amount of work, it gives up structure that dense kernels rely on. Coalesced memory accesses, vectorized loads, and predictable load balancing all assume regular tile traversal; once tiles are skipped, all of these assumptions are not provided. Recovering performance therefore requires both a sparse-friendly storage format and kernels optimized to handle sparsity masks in the three stages of attention.

BCSR storage. We store the block-level mask in *Blocked Compressed Sparse Row* (BCSR) format. BCSR is the block analogue of *Compressed Sparse Row* (CSR), a standard sparse-matrix layout that stores only the nonzero values along with the column index of each one. BCSR does the same thing at the granularity of fixed-size tiles: instead of recording the position of every individual nonzero, it records the position of every nonzero *block*. For each row of tiles, we store the column indices of the dense tiles, and the dense tiles themselves are laid out contiguously in memory in row-major order. This keeps the inner loop of every kernel operating on dense, contiguous tiles (so coalescing and vectorization still apply within a tile) while cutting work in proportion to the fraction of sparse tiles.

Sparsity in Kernels. The two matrix multiplications in Figure 1 change shape under block sparsity. MatMul (4) computes QK^T, but only at positions where the mask is dense. This is the *sampled dense-dense matrix multiplication* (SDDMM) pattern: Q and K are dense, and the mask selects which output tiles to compute. MatMul (5) computes $\sigma(QK^T) \cdot V$, where the left operand is now a block-sparse matrix in BCSR form and V is dense. This is the *sparse-dense matrix multiplication* (SpMM) pattern. The row-wise softmax sits between MatMul (4) and MatMul (5), and it has to respect the same mask. Sparse tiles can be treated as containing $-\infty$ scores, so they contribute zero to both the per-row sum and the output.

Workload characterization. At moderate granularities, all three sparse stages stay data-parallel at the tile level. SDDMM (MatMul (4)) and SpMM (MatMul (5)) decompose into independent per-tile dense matrix multiplications with no cross-tile dependencies, while the sparse softmax does a per-row reduction (max and exponential sum) over only the dense tiles in a row, and the rows themselves remain independent. Locality is strong inside a tile because operands are dense and contiguous in memory, and BCSR's row-stationary layout keeps each row's column indices and dense values close together, so a kernel sweeping a row sees a tight access pattern. SIMD execution is effective on the inner $G \times G$ matmul, where each warp runs a regular dense kernel.

This picture degrades sharply at small granularities. At $G = 1$ the workload is essentially a scatter-gather, and even at $G = 2, 4, 8$ each tile is smaller than a warp, so the inner dense

matmul cannot fill 32 SIMT lanes and tensor-core or vector instructions cannot reach their minimum operand sizes. A single warp ends up assigned to multiple adjacent tiles whose mask bits diverge, forcing serialized execution within the warp. BCSR's per-tile metadata (a column-index lookup and a row-pointer dereference) also becomes comparable in cost to the actual compute, so memory traffic is dominated by indexing rather than values. Even at larger granularities the scheduling unit is now a variable-length list of dense tiles rather than a fixed row of N elements, so load imbalance across thread blocks is the primary cost we have to contain.

Approach

The main development focus of the project was on the three functions: Sparse Matrix Multiplication (SpMM), Sampled Dense Dense Matrix Multiplication (SDDMM), and Sparse Softmax. The entirety of the attention layer was written in CUDA, targeting an architecture of an RTX Blackwell 6000, which has a 128 KB L1 cache for shared memory, 188 Streaming Multiprocessors (SMs), 24,064 CUDA cores, and supports up to 1024 threads in each thread block, performing best with around 5 block occupancy per SM with 128-512 threads per block. We implemented multiple versions of the three functions, starting with a best-effort dense matmul baseline that uses efficient shared memory tiling, then developing sparse mappings and scheduling approaches to leverage sparsity while preserving as much of the locality and parallelism of the dense case as possible.

Tiled MatMul

Before describing the sparse functions, we first describe our hand written baseline, Tiled Matrix Multiplication. Given $A \in \mathbb{R}^{M \times K}$ and $B \in \mathbb{R}^{K \times N}$, we compute $C = A \cdot B$.

The naive mapping launches one thread per output entry: thread (i, j) computes $C_{ij} = \sum_{k=0}^{K-1} A_{ik} B_{kj}$. Every thread streams a full row of A and a full column of B from global memory, even though threads in the same row of the output tile share the row of A and threads in the same column share the column of B. The reuse is large, but the naive kernel does not capture it.

The tiled version stages both operands through shared memory. We partition C into $T \times T$ tiles and assign one thread block of $T \times T$ threads to each tile, so every thread owns a single output entry. The block walks the shared K dimension in chunks of T: each iteration cooperatively loads a $T \times T$ stripe of A (rows $i_0{:}i_0 + T$, columns $k_0{:}k_0 + T$) and a $T \times T$ stripe of B (rows $k_0{:}k_0 + T$, columns $j_0{:}j_0 + T$) into shared buffers sA and sB, synchronizes, then has every thread accumulate its C entry as a dot product across the T columns of sA and rows of sB. The accumulator lives in a register across the entire K sweep and is written back once at the end. Each loaded entry of A is reused T times across the output tile's columns, and each entry of B is reused T times across its rows, which keeps the inner loop bandwidth-light. In practice we batch several T-wide stripes of K per iteration to amortize the cost of the per-chunk synchronization.

Algorithm 1 Tiled matrix multiplication for $C = A \cdot B$. One thread block per $T \times T$ output tile, one thread per output entry.

Block-level state: tile origin (i_0, j_0) in C; shared buffers $sA, sB \in \mathbb{R}^{T \times T}$
Thread-level state: local coords $(i, j) \in [0, T)^2$; register acc $\leftarrow 0$
for $k_0 = 0, T, 2T, \ldots, K - T$ **do**
  $sA[i, j] \leftarrow A[i_0 + i,\ k_0 + j]$
  $sB[i, j] \leftarrow B[k_0 + i,\ j_0 + j]$
  SYNCTHREADS
  for $k = 0$ **to** $T - 1$ **do**
    acc $\leftarrow$ acc $+ sA[i, k] \cdot sB[k, j]$
  end for
  SYNCTHREADS
end for
$C[i_0 + i,\ j_0 + j] \leftarrow$ acc

Sparse Matrix Multiplication (SpMM)

In SpMM we compute $C = AB$ where A is block-sparse (in BCSR form) and B is dense. The opportunity is that $C_{ij} = \sum_{k=0}^{K-1} A_{ik} B_{kj}$ only takes contributions from values of k where A_{ik} is dense, so most of the K-dimension reduction can be skipped.

General approach. The natural starting point is to graft the sparse skip onto tiled matmul. A thread block still owns a $T \times T$ output tile and walks the K dimension in T-wide chunks, but it only visits chunks that could contribute to its tile. The trouble is that a single block covers T different rows of A, and those rows may have different sparsity patterns, so the block's column schedule has to be the *union*

$$\mathcal{K} = \bigcup_{i \in [i_0, i_0 + T)} \{\, k : A_{ik} \text{ is dense} \,\},$$

i.e., every column index that is dense in at least one of the T rows. The block loads the dense entries of A on rows $[i_0, i_0 + T)$ together with the rows of B indexed by $k \in \mathcal{K}$ into shared memory, and each thread then accumulates over only the columns that are dense in *its* own row.

This formulation has two problems:

- **Load imbalance.** Threads in the same block iterate over different per-row sparsity patterns. The densest row sets the pace for the whole block, since every thread has to wait at SYNCTHREADS before the block can advance to the next chunk.
- **Wasted bandwidth.** The loaded sB chunk covers every column in $\mathcal{K}$, but each individual row of A uses only its own subset. Threads load entries of B they will never multiply, and the contiguous-column reads that vectorized loads depend on are punctured by skips wherever a row's pattern disagrees with $\mathcal{K}$.

Both problems come from forcing T rows with potentially different sparsity patterns to share a single column schedule. The cleanest fix is to stop combining rows.

Optimization 1: one row at a time. We change the block's footprint from a $T \times T$ tile of C to a $1 \times T^2$ stripe: a single row of A, paired with T^2 output columns. The thread count per block is unchanged, but now only one row of A is in play, so $\mathcal{K}$ collapses to the sparsity pattern of that

Algorithm 2 Sparse–dense matmul with row-union column scheduling. One thread block per $T \times T$ tile of C.

Block state: tile origin (i_0, j_0); $\mathcal{K} \leftarrow \bigcup_{i \in [i_0, i_0+T)} \{k : A_{ik} \text{ dense}\}$
Thread state: local coords $(i, j) \in [0, T)^2$; register acc $\leftarrow 0$
for each T-sized chunk of $\mathcal{K}$ **do**
  cooperatively load the dense $A_{i_0+i,\,k}$ for k in this chunk into sA
  cooperatively load $B_{k,\,j_0+j}$ for k in this chunk into sB
  SYNCTHREADS
  for each k in this chunk **do**
    if $A_{i_0+i,\,k}$ is dense **then**
      acc $\leftarrow$ acc $+ sA[i, k] \cdot sB[k, j]$
    end if
  end for
  SYNCTHREADS
end for
$C[i_0 + i,\, j_0 + j] \leftarrow$ acc

single row. The union disappears, every thread that loads a column of B uses it, and every thread performs the same number of multiply-adds. Load imbalance and the vectorization problem both go away.

Optimization 2: align rows with the sparsity granularity. After implementing and testing the algorithm above, our measured speedups over the baseline were minimal. Profiling with Nsight Compute showed unexpectedly large load latencies in the kernel. We can see why by looking at $T = 16$ for example. In tiled matmul, a thread block outputs $T \times T = 256$ entries while loading $A_{i:i+T,\,k}$ and $B_{k,\,j:j+T}$ for all k, totaling $2TK$ elements. In the 1-row mapping above, the same block still outputs $T^2 = 256$ entries (now as a $1 \times T^2$ stripe), but loads $A_{i,\,k}$ and $B_{k,\,j:j+T^2}$ for all k, totaling $(T^2 + 1)K$ elements. That is roughly $16\times$ more memory traffic per output entry than the dense baseline, which matches the load times we saw in profiling.

We collapsed to a single row of A in Optimization 1 to guarantee a consistent sparsity pattern, but that constraint is softer than it looks: by construction, the mask is block-sparse at granularity G, so any consecutive G rows of A share a sparsity pattern by definition. We therefore widen the block back out, mapping each block to a $G \times (T^2/G)$ tile of C. The total load per block becomes $(G + T^2/G)\,K$. This allows us in large sparse patterns to increase our data reuse.

Sampled Dense Dense Matrix Multiplication (SDDMM)

SDDMM computes $C = (AB) \odot M$, where A and B are dense and M is the sparse mask (in BCSR form). Only the entries of C at dense positions of M need to be computed and stored in the output BCSR matrix. We implemented three broad approaches; the first one is a direct extension of tiled matmul which serves as our baseline, the second one is a gather-based approach that batches together dense column-blocks of M to recover some of the reuse of tiled matmul, and the third one is a gather-based approach that schedules at the level of individual active coordinates to eliminate all redundant work.

Naive: tiled matmul with a sparse output mask. This is the most direct approach; keeping the dense tiled matmul structure and skip writes for masked-out positions. Each thread block

computes a $T \times T$ tile of the output exactly as in dense matmul, but threads whose output position is sparse simply return without writing. The dense loads of sA and sB are unchanged, so cache locality and vectorization are preserved. The cost is wasted computation: as sparsity increases, the fraction of threads performing dot products whose results are never written also increases proportionally. Workload imbalance becomes a problem again, because the useful work per block depends on how many of its T^2 positions are dense, and at high sparsity an entire $T \times T$ tile can be launched with almost nothing to compute.

Row-batched dense Matrix Multiplication (`sddmm.cu`). Our first gather-based approach gathers at the BCSR block-row level. Each thread block is assigned one block-row b_i of C (the output matrix) together with a contiguous *batch* of dense column-blocks within that row, where the batch size is a per-granularity constant tuned to fill the warps and fit in shared memory. The block first reads its batch's column indices from the BCSR `col_idx` array; then, for each 32-wide chunk of the K dimension, it loads (i) the $T \times 32$ stripe of A corresponding to its block-row, once, and (ii) the $T \times 32$ stripe of B for each column-block in the batch, gathered into one shared-memory buffer. The inner loop is then a clean dense matmul: every thread accumulates a 32-element dot product per K-chunk with no mask checks, because the column-blocks were filtered to the dense ones during the gather. After the K sweep, the accumulators are written directly into C's BCSR storage. The cost of loading A amortizes across every column-block in the batch, and the mask is consulted once per batch rather than once per output entry.

The drawback is grid sizing: different block-rows have different numbers of dense column-blocks, so the grid is sized for the *worst* row. Block-rows with fewer dense blocks have leftover thread blocks that exit immediately, and at low granularity the per-row variance is large enough that this launch overhead grows noticeable.

Active-coordinate gather (`matmul.cu`). The version we ended up using inverts the order of operations: first gather the positions that need to be computed, then compute and store only those. This has the benefit of eliminating all redundant work, but it requires a more complex scheduling structure to keep the warps busy. The output is partitioned into 16×16 or 32×32 SDDMM tiles (we launch smaller SDDMM tiles for lower granularities), which sit on top of the smaller $T \times T$ BCSR blocks of the mask (with $T \leq 32$). One warp owns each SDDMM tile and processes it in three stages.

1. **Dynamic tile scheduling.** Warps pull the next SDDMM tile via an atomic increment of a global counter, rather than from a static grid. Warps that finish quickly grab another tile, and warps that land on a fully-sparse tile move on after almost no work. This dynamic scheduling is important for load balancing at all sparsity and granularity levels.

2. **Active-coordinate list.** The warp scans through its assigned SDDMM tile in the BCSR data structure to find the dense $T \times T$ blocks that intersect its tile, and appends the coordinates of every entry inside those blocks to a per-warp list in shared memory. If the list is empty after this pass, the warp moves on to the next tile.

3. **Dense load, sparse compute, sparse write-back.** Similar to tiled matmul, the warp strides through the N dimension of the dense input matrices A and B in chunks of 32 and load the corresponding stripes of A and B into shared memory with coalesced reads. Then the warp iterates through the active coordinate list in chunks and computes a dot product *only* at the coordinates in the active list. The reads performed while computing a tile are the

same as in tiled matmul, but the arithmetic is now sparse because the active coordinate list is a subset of the tile, and as a result the reads are not coalesced. After all chunks finish, the accumulators are written back into the BCSR storage of C at the active positions. Note that due to the BCSR layout which stores the non-zero blocks contiguously, the sparse write-back is coalesced within each tile.

This recovers the dense-load patterns of tiled matmul (the A and B tiles are read densely and contiguously, so coalescing and vectorization still apply) while avoiding the wasted arithmetic of the naive tiled variant: every multiply-add hits a position whose result is needed and will actually be stored. The dynamic counter also keeps warps busy regardless of how the dense tiles are distributed in output C, so the kernel behaves consistently as granularity G varies. The cost is the active-coordinate list and the per-tile scan to populate it, which are both proportional to the number of dense entries in the tile rather than to T^2.

Sparse Softmax

Softmax computes $\sigma(x_i) = \exp(x_i - m)/\sum_j \exp(x_j - m)$, where $m = \max_j(x_j)$, for each row of the score matrix. In the sparse regime, masked-out positions effectively have a score of $-\infty$ and contribute zero to both the row maximum and the exponential sum. Therefore, the sparse softmax kernel walks the BCSR structure of the mask directly, only visiting the dense tiles in each row. For each tile it computes the max and exponential sum across the tile's entries, then writes back the normalized values in place. The reduction structure remains the same as in dense softmax, but the length of each reduction is proportional to the number of dense tiles in the row rather than to N.

Sequentiality and Memory Intensiveness. Unlike SDDMM and SpMM, where output tiles can be computed largely independently of one another, Softmax has strict row-wide dependencies. It must make three distinct sequentially dependent passes over the active elements of a row: (1) a global row reduction to find the maximum value for numerical stability, (2) a reduction across the row to compute the sum of exponentials, and (3) a final pass to divide each valid exponential by the sum. With extremely low arithmetic intensity (very few FLOPs per byte loaded), sparse softmax is heavily bound by memory bandwidth.

Benefit of Row-Interleaved BCSR Format. Because the kernel is bandwidth bound, keeping memory accesses coalesced is the primary optimization target. In standard block-sparse formats, processing a single scalar row would typically require strided or jumping memory accesses as threads skip to the next dense tile. We resolve this using our *row-interleaved BCSR* data layout. Instead of storing entire $T \times T$ tiles contiguously, the values for all dense blocks in a given block-row are interleaved such that the elements of any scalar row t are laid out completely contiguously in memory. This allows the inner chunk-processing loops of the kernel to simply stride by `blockDim.x`, resulting in perfectly coalesced memory reads and writes across the entire active portion of the row.

Mapping and Scheduling. Because reductions are bound to rows, threads within a block cooperate using hardware `__shfl_down_sync` primitives for fast warp-level reductions, block-wide gather of the per-warp accumulator in shared memory, and sweeps by Warp 0 for block-wide reductions. To prevent variable sparsity from severely impacting load balance across SMs, we explored two row-to-block scheduling strategies:

1. **Dynamic (One-Row-Per-Block):** Maps exactly one scalar row to one block. At token lengths N longer than about 2^{10}, the number of rows is large enough to keep the GPU fully

occupied with blocks, and Cuda's dynamic scheduler handles load imbalance across rows. However, at shorter N or higher G, the number of blocks can become too small to fill the GPU, and load imbalance can lead to some SMs idling while others are still processing long rows.

2. **Static Partitioning:** To prevent load imbalance, we implemented a static partitioned scheduler. We precalculate the weights (number of non-zero elements) of every row and assign blocks to row intervals using a precomputed partition array. This ensures each thread block processes nearly identical amounts of data, maintaining high occupancy across the GPU regardless of the underlying sparsity distribution or granularity. The drawback is that the partitioning is static and does not adapt to runtime variability of compute and memory performance, but in practice we found it to be more effective than the dynamic scheduler at all sparsity levels and granularities.

Results

We measure performance by running one full attention iteration end-to-end and comparing our sparse implementation's wall-clock time against a dense baseline. The baseline runs the same attention computation, but uses our optimized tiled-matmul kernel for both QK^T and the multiplication with V, and a standard dense row-wise softmax in between. It does not exploit the mask in any way: it materializes the full $N \times N$ score matrix and processes every entry. Both implementations are written in CUDA, target the same RTX Blackwell 6000, and are timed end-to-end as well as kernel-by-kernel.

We sweep three hyperparameters:

- N, the context length (the problem size), $\in \{2^{10}, 2^{12}, 2^{14}, 2^{15}\}$.
- p, the fraction of sparse entries in the mask, $\in \{0.5, 0.9, 0.95, 0.99\}$.
- G, the sparsity granularity, $\in \{1, 2, 8, 32\}$.

Experimental setup. For each (N, p, G) configuration we run both the sparse and the dense baseline on three random seeds. The seed controls the input matrices and the sparsity pattern, so both implementations see identical inputs and an identical mask on each run. We record per-kernel wall-clock times alongside the end-to-end attention time, and report the mean across the three seeds.

Hyperparameter notes. Two observations frame the rest of this section:

- Increasing p reduces the amount of useful arithmetic. An ideally-parallel sparse implementation would therefore reach a speedup of $1/(1-p)$ over the dense baseline ($100\times$ at $p = 0.99$). That ratio is an upper bound; sequential portions and per-tile overhead always pull the realized speedup below it.
- Increasing G makes the sparsity pattern coarser and less expressive: at $G = 32$, every entry inside a 32×32 tile shares the same dense/sparse decision. Coarser patterns lose modeling fidelity, but they also let the kernels exploit larger contiguous regions, recovering vectorized loads and warp-aligned compute. The sweep over G exists to surface this tradeoff; in a production setting the granularity would be chosen empirically for the workload rather than from a uniform-random mask.

Our target machine was correct, with the hyper parallelization of threads available on a GPU, we were able to utilize the incredible speedup for massive Matrix Multiplications and Softmaxs that wouldn't have been possible otherwise.

Overall Speedup

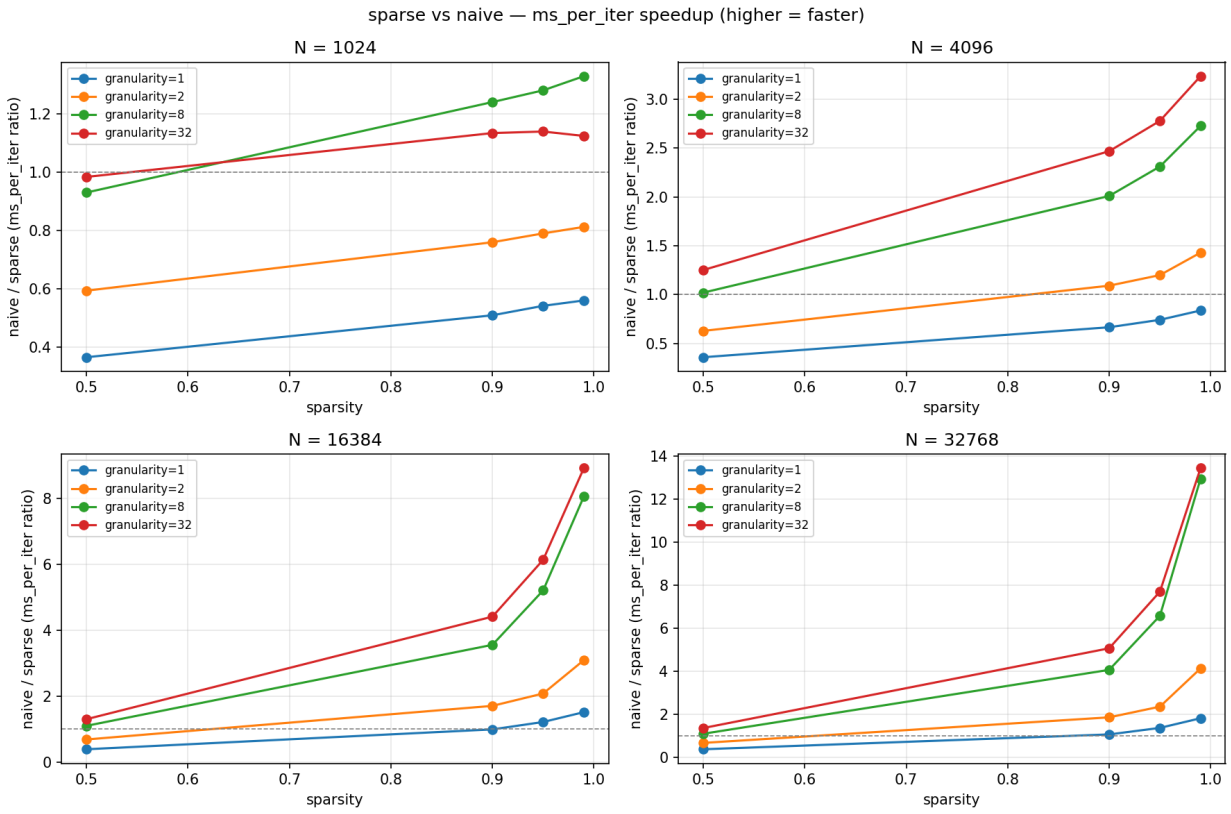


Figure 2: End-to-end attention speedup of our sparse implementation over the dense baseline, swept across N, G, and p. Each curve fixes (N, G) and varies p.

Two trends in Figure 2 stand out, neither of which is obvious from the algorithm alone:

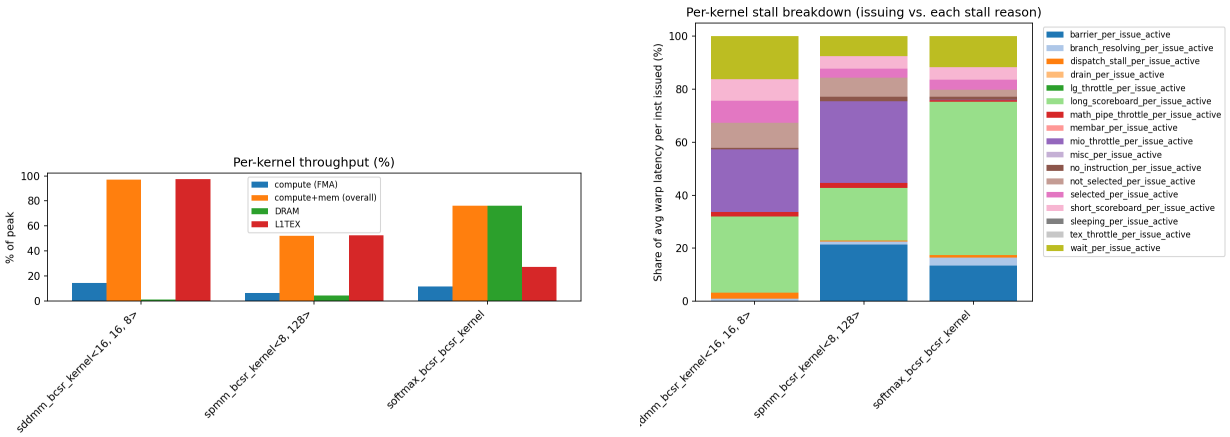
1. Speedup grows substantially with the problem size N. The curves shift up by a large margin between $N = 1024$ and $N = 32768$ at every sparsity level.

2. For a fixed p, coarser granularities G produce far larger speedups than finer ones, even though both have the same amount of useless computation.

Problem size. At small N, the dense projection matmuls that produce Q, K, V do work comparable to the sparse SpMM/SDDMM/softmax stage, since $d = 768$ is on the same order as N at the low end of the sweep. Amdahl's law then bounds the end-to-end speedup: even with infinitely fast sparse kernels, the projections still dominate the wall clock. As N grows, the sparse stage's $O(N^2 d)$ cost overtakes the projections' $O(Nd^2)$ cost, the projection share of the wall clock shrinks, and the per-kernel speedup our sparse implementation offers translates more directly into end-to-end speedup.

Granularity. Fine-grained sparsity breaks vectorization: a `float4` load fetches four contiguous values in one instruction, but only when all four are needed; once adjacent lanes diverge on the mask, the load decays into single-element accesses. Additionally, memory loads can be coalesced when loading from adjacent values, but when loading at fine-grained sparsities, loads become uncoalesced, significantly harming speedup times. Coarser G fixes both problems at once by guaranteeing warps stay aligned and contiguous loads survive. ML workloads in practice almost never use $G = 1$ for exactly these reasons, and our results confirm that fine-grained sparsity is the worst-case regime for any sparse-attention kernel.

Where the remaining gap comes from. Even at a favorable corner of the sweep ($N = 32768$, $G = 8$, $p = 0.95$), where the upper bound $1/(1-p) = 20$ should be close to attainable, we measure a 12× end-to-end speedup. Figure 3 explains the gap.



(a) Per-kernel compute and memory throughput.

(b) Per-kernel warp-stall reasons.

Figure 3: Nsight Compute traces for a $N = 32768, G = 8, p = 0.95$ run. SDDMM and SpMM under-use DRAM bandwidth because most of their loads land in shared memory, while the sparse softmax shows noticeably higher global-memory traffic. The right panel attributes per-kernel stalls: `long_scoreboard` flags warps waiting on global-memory loads, and `mio_throttle` flags warps waiting on shared-memory loads.

The takeaway from Figure 3 is that memory traffic, not arithmetic, is what bounds us in the high-sparsity regime. Skipping more arithmetic does not proportionally reduce loads: every tile we do compute still has to bring in its operand stripes, the BCSR metadata still has to be walked, and the softmax still has to sweep its row of dense tiles. The $20\times \rightarrow 12\times$ residual therefore lives in memory-system stalls rather than wasted compute, and would shrink under techniques aimed at the memory subsystem (better load pipelining, persistent kernels, smaller working sets).

SDDMM

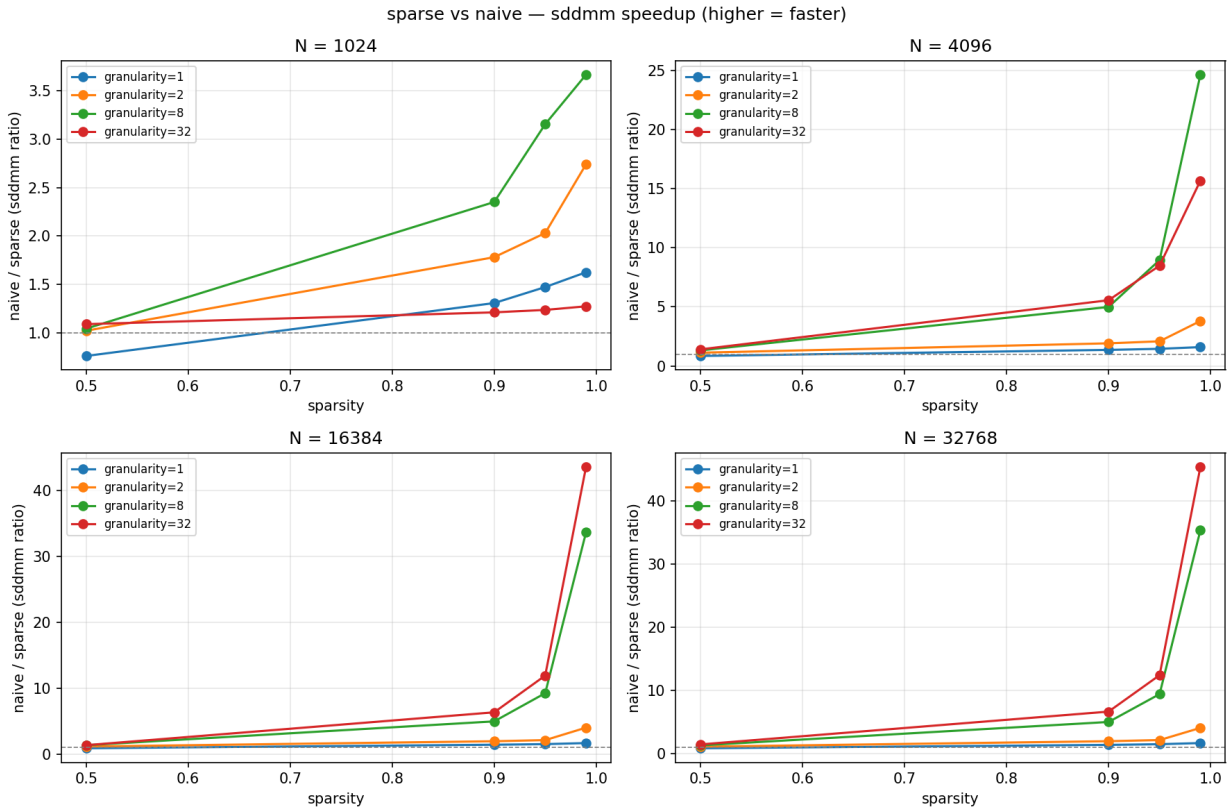


Figure 4: End-to-end speedup of the sparse SDDMM kernel over the dense baseline, swept across N, G, and p.

The SDDMM performance graph isolates the QK^T multiplication. The structural characteristics of the performance analysis are:

1. **Low occupancy at small scale ($N = 1024$):** Granularity 32 sees virtually no speedup at $N = 1024$. At $G = 32$, a 1024×1024 output matrix consists of only $32 \times 32 = 1024$ total tiles. Since our dynamic active-coordinate kernel assigns exactly one warp per SDDMM tile, this configuration launches too few warps to keep the SMs fully occupied or hide launch overheads, reducing sparsity benefits.

2. **Memory bottlenecks and stall profiles:** Figure 5 (ahead) shows that SDDMM has zero barrier stalls. This is because the dynamic gather algorithm operates entirely at the single warp level (using `__syncwarp()`), avoiding block-wide `__syncthreads()` completely.

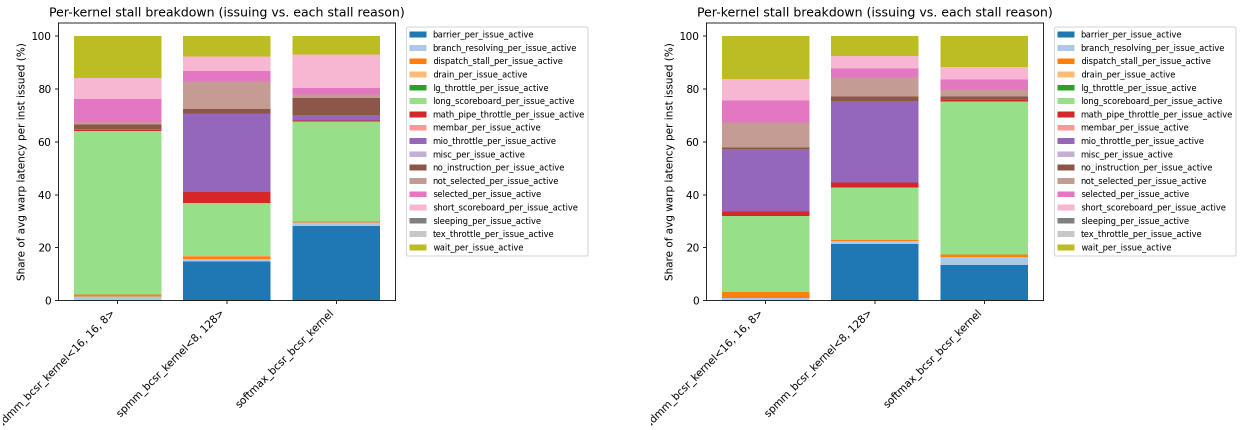
3. **Shifting bottlenecks with N:** At $N = 1024$, over 60% of warp stalls are `long_scoreboard` (waiting on global memory). The kernel is fully bound by fetching BCSR metadata and streaming the input matrices without enough active compute to hide the latency. As N scales up to 32768, the global memory stall fraction drops to roughly 30%, while `mio_throttle` (shared memory waits) jumps to 20%. This indicates that at large sizes, the inner loop, which densely loads A and B tiles into shared memory and then sparsely computes dot products

from those shared arrays, starts hitting the hardware shared memory bandwidth limits of the SM.

4. **Divergence in $1/(1-p)$ scaling:** As sparsity p increases toward 0.99, the theoretical arithmetic speedup approaches 100×. For fine granularities ($G = 1$ and 2), the observed SDDMM speedup plateaus significantly below this upper bound. At $G = 1$ and high sparsity, almost every structural 32×32 block has at least one active entry, forcing the warp to redundantly load the dense stripes of A and B from global memory just to compute a scattered handful of values. While the active coordinate list perfectly scales down the actual *FLOPs*, it introduces severe metadata indexing overhead and heavily uncoalesced memory accesses. Conversely, coarser granularities like $G = 8$ and $G = 32$ exhibit the exact concave-up, sharply increasing behavior we expect from theoretical scaling, by skipping large, hardware-aligned contiguous blocks and leveraging coalesced load and vectorized compute potential.

Overall, our active-coordinate gather successfully eliminated the wasted computation of a masked matmul, making it so that every FLOP computed was necessary. However, the plateauing behavior at lower granularities means that the true cost of sparse structures is often metadata overhead, uncoalesced memory accesses, and inefficient mapping. The big difference between $G = 1$ and $G = 32$ suggests that tuning/specialization based on tiling models and GPU hardware strengths, specifically contiguous memory and warp alignment, would be helpful to attain ideal scaling on lower granularities. While our implementation proves this is possible at coarse granularities, further tuning, varying mapping approaches, and specializing code for specific sparsity-granularity models could help achieve more consistent performance across the full spectrum.

Potential improvements for SDDMM would involve decoupling the SDDMM tile size from the problem dimension at small N to artificially inflate warp counts and restore occupancy. At large N, the shared memory bottleneck (`mio_throttle`) could be addressed by increasing register-level accumulation or tuning the tile aspect ratio to improve data reuse from each shared memory fetch, although the latter is a tradeoff between shared memory size (which affects block-SM occupancy) and memory bandwidth (which affects the stall profile).

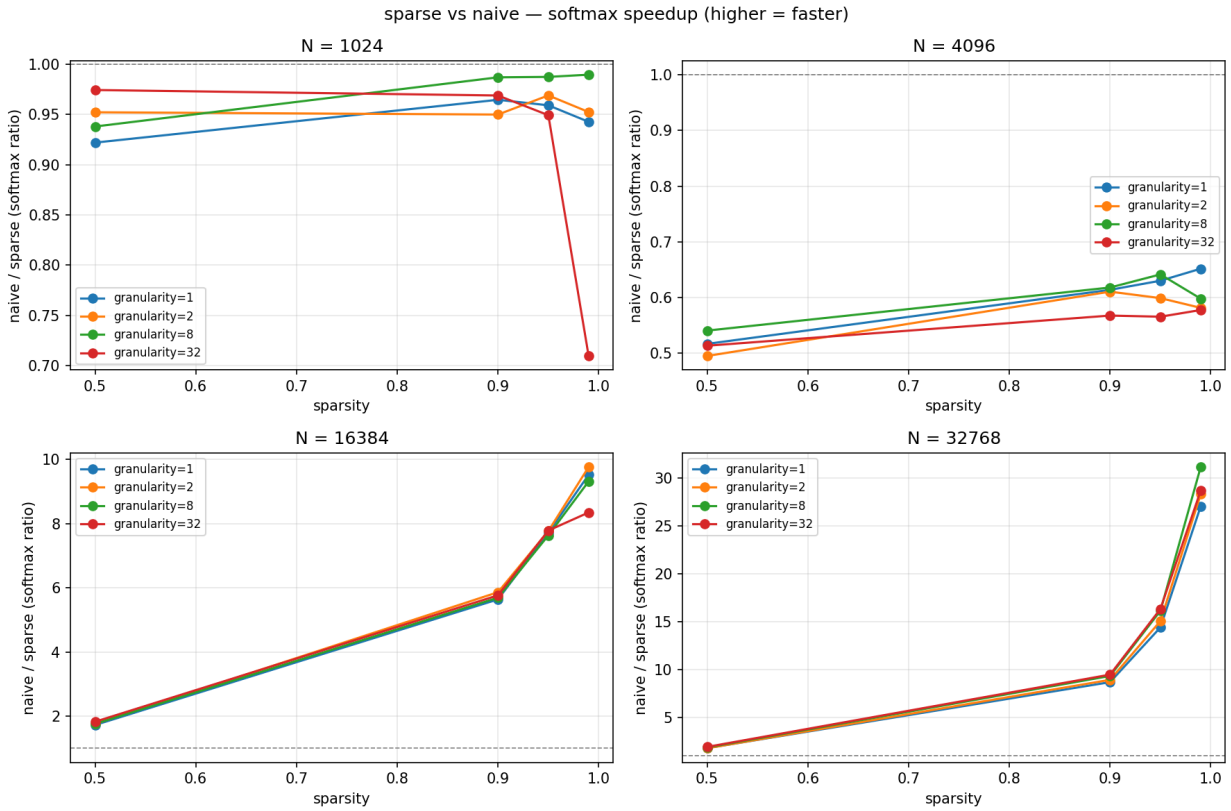


(a) Per-kernel warp stall reasons for $N = 1024$.

(b) Per-kernel warp stall reasons for $N = 32768$.

Figure 5: Nsight Compute traces for the $N = 32768$, $N = 1024$ run. Per-kernel stalls: `long_scoreboard` flags warps waiting on global-memory loads, `mio_throttle` flags warps waiting on shared-memory loads, `barrier` flags warps waiting on a synchronization call.

Sparse Softmax



The sparse softmax speedup characteristics look different from SDDMM and SpMM:

1. **Granularity invariance:** The curves for $G = 1, 2, 8, 32$ overlap almost entirely. This matches our implementation in `softmax.cu`: the softmax kernel maps a thread block to a single matrix row and processes all items within that row as a flattened, contiguous 1D array of length `num_blks * T`. It does not matter whether those non-zero elements came from two 16×16 blocks or thirty-two 1×1 blocks; the row-interleaved BCSR layout ensures the memory access pattern is identical (as discussed in the implementation approach above). Speedup is therefore strictly a function of total sparsity p, not granularity G.

2. **Reduction scaling:** At a small problem size ($N = 1024$), almost 30% of warp stalls are `barrier` stalls. Because each row is short, the chunk-processing phase is fast and load imbalance has the tendency to be higher, so threads spend a disproportionate amount of time waiting at block-wide `__syncthreads()` barriers during the shared-memory max and sum reductions.

3. **Bandwidth bottleneck:** As N grows to 32768, the fraction of `barrier` stalls drops by nearly half (to 15%), and `long_scoreboard` stalls increase to 60%. This is likely because the kernel is now streaming large amounts of data from global memory (completely bound by DRAM bandwidth) just to use it once. The reduction phases are still present, but they are now a smaller fraction of the total runtime and therefore cause fewer stalls. This matches our analysis above that the softmax kernel is heavily memory-bound.

4. **Scaling response tied to problem size:** Because softmax operates on 1D vectors rather than multidimensional matrices, its baseline arithmetic intensity is negligible compared to

the $O(N^3)$ matmuls. At lower token lengths ($N = 1024, 4096$), increasing sparsity reduces the total work available so much that launch overheads, metadata resolution, and sequential reduction operations completely dominate, resulting in a flattened speedup curve. However, at larger context lengths ($N = 16384, 32768$), the kernel reaches an inflection point where the amount of saved computation overcomes the fixed costs, and the speedup cleanly follows the ideal concave-up $1/(1-p)$ theoretical scaling response.

Ultimately, while our custom row-interleaved BCSR format and statically partitioned mappers successfully balanced the SM workloads and kept memory accesses perfectly coalesced, the absolute speedups depend heavily on maintaining enough active work per block. We initially hypothesized that avoiding the traversal of a fully dense $N \times N$ matrix would grant order-of-magnitude improvements across the board. In practice, at small regimes, the fundamental memory-bound nature combined with low active work meant our gains were inherently capped. But at context lengths where these optimizations actually matter most ($N = 16384+$), the architectural scaling matched our theoretical maximums (hopes).

Because softmax requires multi-pass reads, the `long_scoreboard` penalty at large N could benefit from fusing operations. If the scaling of Q and the softmax itself could be fused directly into the tail of the SDDMM kernel, the intermediate sparse scores would never need to be written to and read back from global memory, removing this bottleneck. This would require on-the-fly softmax computation as the SDDMM tile is processed, which is a non-trivial extension of the current algorithm, but it would have had a lot of potential to improve the end-to-end speedup.

SpMM

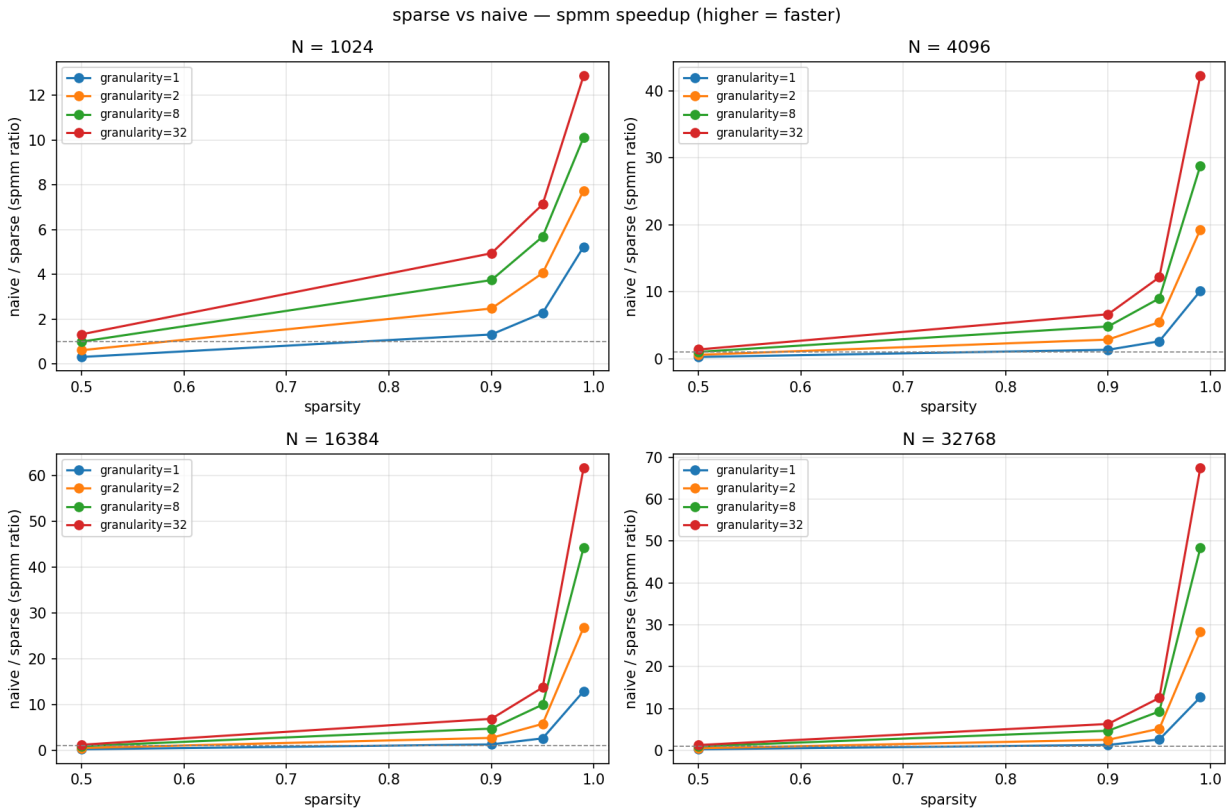


Figure 6: End-to-end speedup of the sparse SpMM kernel over the dense baseline, swept across N, G, and p.

SpMM is the cleanest case for our optimizations: the trends in Figure 6 match the predictions from the general-speedup analysis above, and the per-G ordering follows the same explanation. Looking at Figure 5, the stall composition does not change in character as N grows; the ratio of `long_scoreboard` to `mio_throttle` stays roughly constant. The gains from increasing N therefore come from raising arithmetic intensity against a roughly fixed memory-loading cost, rather than from changing what the kernel is bottlenecked on.

References

[1] Trevor Gale, Matei Zaharia, Cliff Young, and Erich Elsen. Sparse GPU kernels for deep learning. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2020.

[2] Tianyang Lin, Yuxin Wang, Xiangyang Liu, and Xipeng Qiu. A survey of transformers, 2021.

List of work

The total credit for the project was 50-50 between Saransh and Mehul.