SPLAT: A Framework for Optimised GPU Code-Generation for SParse reguLar ATtention

AHAN GUPTA, University of Illinois at Urbana-Champaign, USA
YUEMING YUAN, University of Illinois at Urbana-Champaign, USA
DEVANSH JAIN, University of Illinois at Urbana-Champaign, USA
YUHAO GE, University of Illinois at Urbana-Champaign, USA
DAVID APONTE, Microsoft, USA
YANQI ZHOU, Google DeepMind, USA
CHARITH MENDIS, University of Illinois at Urbana-Champaign, USA

Multi-head-self-attention (MHSA) mechanisms achieve state-of-the-art (SOTA) performance across natural language processing and vision tasks. However, their quadratic dependence on sequence lengths has bottlenecked inference speeds. To circumvent this bottleneck, researchers have proposed various sparse-MHSA models, where a subset of full attention is computed. Despite their promise, current sparse libraries and compilers do not support high-performance implementations for *diverse* sparse-MHSA patterns due to the underlying sparse formats they operate on. These formats are either too specialised, failing to cover a wide-range of sparse patterns, or too general, incurring high metadata overhead when computing on the *moderately* sparse (10-50% non-zeros) matrices present in sparse-MHSA.

We bridge this gap, achieving both generality and performance, by proposing a novel sparse format: affine-compressed-sparse-row (ACSR) and supporting code-generation scheme, SPLAT, that generates high-performance implementations for diverse sparse-MHSA patterns on GPUs. Core to our proposed format and code generation algorithm is the observation that common sparse-MHSA patterns have uniquely regular geometric properties. These properties, which can be analyzed just-in-time, expose novel optimizations and tiling strategies that SPLAT exploits to generate high-performance implementations for diverse patterns. To demonstrate SPLAT's efficacy, we use it to generate code for various sparse-MHSA models, achieving speedups of up-to 2.05x and 4.05x over hand-written kernels written in Triton and TVM respectively on A100 GPUs in single-precision.

CCS Concepts: • **Software and its engineering → Source code generation**; **Software performance**; • **General and reference → Performance**.

Additional Key Words and Phrases: GPU Code Generation, Self Attention, Large Language Models, Sparsity

ACM Reference Format:
Ahan Gupta, Yueming Yuan, Devansh Jain, Yuhao Ge, David Aponte, Yanqi Zhou, and Charith Mendis. 2025. SPLAT: A Framework for Optimised GPU Code-Generation for SParse reguLar ATtention. *Proc. ACM Program. Lang.* 9, OOPSLA1, Article 138 (April 2025), 29 pages. https://doi.org/10.1145/3720503

Authors' Contact Information: Ahan Gupta, University of Illinois at Urbana-Champaign, Champaign, USA, ag82@illinois.edu; Yueming Yuan, University of Illinois at Urbana-Champaign, Champaign, USA, yy28@illinois.edu; Devansh Jain, University of Illinois at Urbana-Champaign, Champaign, USA, devansh9@illinois.edu; Yuhao Ge, University of Illinois at Urbana-Champaign, Champaign, USA, yuhaoge2@illinois.edu; David Aponte, Microsoft, Seattle, USA, davidaponte@microsoft.com; Yanqi Zhou, Google DeepMind, Mountain View, USA, yanqiz@google.com; Charith Mendis, University of Illinois at Urbana-Champaign, Champaign, USA, charithm@illinois.edu.

ACM 2475-1421/2025/4-ART138
https://doi.org/10.1145/3720503

1 Introduction

Transformers have enjoyed widespread adoption in industry [5, 40]. However, to effectively train and serve models at scale, transformers must: (1) have high model quality, and (2) utilize manycore GPU architectures effectively. Nevertheless, achieving both simultaneously is challenging as datasets and tasks [48, 51] are demanding increasingly longer input sequences. This increases the memory consumption of multi-head-self-attention (MHSA) layers, which increases quadratically with respect to input sequence lengths, and reduces the largest permissible batch size (LPBS) a model can operate on. Since sequences across batches are independent and can be processed in parallel, large models operating on long contexts are forced to use small batches and do not realize their high-throughput potential despite being embarrassingly parallel.

To mitigate the memory bottleneck of MHSA, researchers have proposed several sparse-MHSA methods [8, 16, 24, 34, 35]. These methods compute a subset of the entire attention matrix using a statically fixed mask. However, unlike the sparsity levels encountered in widely studied scientific and high-performance computing applications [3, 33] which are extremely sparse (<10% of the values are non-zero), state-of-the-art (SOTA) sparse-MHSA methods are *moderately* sparse, computing 10-50% of the full attention matrix [8, 16, 34, 35, 52, 59]. Computing fewer values degrades model quality while computing more values consumes additional memory. These moderate sparsity ranges place unique challenges in adopting existing sparse formats to implement high-performance sparse-MHSA kernels.

On one end, general sparse libraries [36, 57] and optimizations usually operate on *general sparse formats* (GSFs) that incur high metadata overhead at the moderate sparsity levels present in sparse-MHSA. Such formats, like the compressed-sparse-row (CSR) and coordinate (COO) formats, contain metadata that represent the dense coordinates of each non-zero value, consuming memory in $O(nnzs)$. To extract performance out of sparse kernels operating on GSFs, sparse libraries and optimizations propose various strategies [15, 26, 32]. Nevertheless, every non-zero value read from a GSF must also read its respective metadata to uncover its dense coordinate. This *at least* doubles the data read from high-bandwidth memory within the inner loops of sparse primitives. Since sparse-MHSA layers are moderately sparse, producing megabytes of non-zero values per layer [24, 34], doubling the data read from high bandwidth memory (HBM) exacerbates contention of L1 caches and register-file resources, inhibiting performance. As we see in figure 1, even hand-optimized vendor libraries like cuSPARSE [2] that employ the CSR format are outperformed by their dense counterparts, cuBLAS [1], for density levels as low as 20% despite doing 1/5th of the compute.

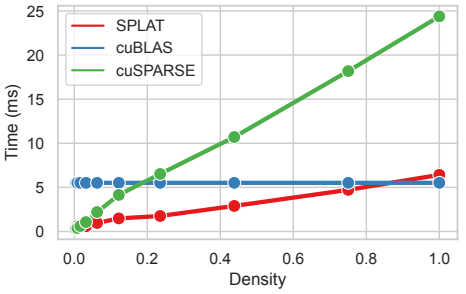


Fig. 1. Run-time results for a sparse primitive used in sparse-MHSA (R-SpMM) comparing cuSPARSE, cuBLAS and SPLAT. We vary the density of the sparse input across: [0.4, 0.8, 1.6, 3, 6, 12, 24, 44, 75, 100]. The sparse input takes the shape of the blocked pattern (figure 2 right).

On another end, hand-written kernels usually operate on *custom sparse formats* (CSFs) that are specialised to a single sparse-MHSA pattern and need to be redesigned to extract performance from different patterns. This specialisation permits format designs with lower metadata storage, and custom sparse-schedules with favorable thread access patterns. For example, triton's block-sparse kernels [29] are hand-written kernels that operate on a CSF curated to represent block-like sparsity patterns, giving up to a 9x speedup over using GSFs like CSR & COO [57]. However, naively

adopting its CSF to a non block-like sparsity pattern, such as the window pattern (figure 2 - 2^{nd} from left), results in redundant storage and compute, reducing the LPBS of models (see section 3 for an example). To recover this lost performance, practitioners need to rewrite hand-written kernels and the CSFs they operate on to specialise the indexing of sparse-structures to the sparse-MHSA pattern in question, like [8] which specialises to the windowed and strided patterns. Since indexing sparse structures requires non-trivial arithmetic involving nested layers of indirection, it is difficult to reason about which optimizations are effective, resulting in a concerted engineering effort to hand-write high-performance sparse-MHSA kernels for a variety of patterns.

We observe that no general data format facilitates high-performance implementations for various sparse-MHSA patterns. GSFs require $O(nnzs)$ metadata storage to permit generality at the cost of performance while CSFs reduce metadata storage to permit performance at the cost of generality. We plug this gap and propose a novel data-format: affine-compressed-sparse-row (ACSR) and supporting GPU code-generation scheme, SPLAT (**SP**arse regu**L**ar **AT**tention) that can cover a wide range of sparse-MHSA patterns while achieving good performance. However, in order to do so, we had to solve several challenges.

First, reducing metadata below $O(nnzs)$ typically reduces the generality of a format, potentially reducing its coverage of sparse-MHSA patterns. Fortunately, we observe that a variety of commonly used sparse-MHSA patterns are static with *regularly* repeating non-zero sub-structures. We introduce a novel geometric property that describes these regularly repeating sub-structures: affine-compressibility, and term sparse-MHSA patterns that are affine-compressible as *regular*. This allows us to lift their regularity into the design of the ACSR format, enabling compressed metadata storage in $O(rows)$, an asymptotic reduction of $O(nnzs)$ compared to GSFs. Importantly, our notion of affine-compressibility is general, enabling us to represent a *variety* of current and potential future affine-compressible sparse-MHSA structures without incurring extraneous padding & compute, unlike CSFs. For example, the ACSR format can precisely represent: longformer-strided and windowed [8] (12.5% density), gemma-two [24] (37.5% density), reformer [35] (25% density), and big-bird global (10.9% density) amongst others [16, 34, 42].

Second, with the introduction of our novel ACSR format, it is a challenge to code-generate high-performance sparse-MHSA kernels. Sparse-primitive schedules are intricately linked to the underlying sparse-format they operate on. Therefore the ACSR, in its unique layout of both its non-zero values and meta-data, requires novel optimizations and schedules to achieve high-performance. To overcome these challenges, we develop a GPU code-generation framework, SPLAT, that produces high-performance sparse-MHSA patterns with the ACSR as the underlying representation.

To demonstrate SPLAT's efficacy, we implement 4 widely used sparse-MHSA patterns at various sparsity levels. SPLAT-generated SDDMM and SpMM kernels, two core primitives of sparse-MHSA, outperform vendor libraries cuBLAS and cuSparse at moderate sparsity levels by 2.81 & 5.61x respectively. Moreover, SPLAT's end-to-end generated sparse-MHSA outperforms handwritten kernels in triton and TVM by up to 2.05x and 4.05x respectively in single-precision.

In summary, this paper makes the following contributions:

- We introduce a novel geometric property of sparse-MHSA patterns: *affine-compressibility* and leverage this to propose the ACSR format (Sections 4 & 5).
- We develop novel optimized GPU code-generation schemes for regularly sparse primitives, what we term as R-SDDMM and R-SpMM kernels, that use the ACSR (Sections 6 & 7).
- We use the optimized sparse operations to provide GPU code-generation strategies for end-to-end globally efficient sparse-MHSA models (Section 8).
- We implement these code generation schemes in a framework called SPLAT, and evaluate SPLAT against against hand-optimized kernels (Section 9).

2 Background

Full Attention. The backbone of the transformer is multi-head-self-attention (MHSA) [55]. MHSA computes the following matrix: $Concat(Head_1, Head_2, ..., Head_h)$ where $Head_i$ is: $\underbrace{softmax(QW_i^Q(KW_i^K)^T)}_{A_i} VW_i^V \in \mathbb{R}^{N \times d_m/h}$ and the concatenation happens across the columns of $A_i VW_i^V$. The matrices $Q, K \,\&\, V \in \mathbb{R}^{N \times d_m}$ are the input matrices consisting of N vectors of size $\mathbb{R}^{d_m}$, where N is the input sequence length. W_i^Q, W_i^K and $W_i^V \in \mathbb{R}^{d_m \times d_h}$ are linear transformations. The matrix A_i is known as the attention matrix and the softmax is taken row-wise in the product $QW_i^Q(KW_i^K)^T$. Self-attention is expensive due to the matrix A_i being of size $O(N^2)$.

Sparse Attention. To alleviate the quadratic computation in self-attention. Researchers have proposed a variety of *sparsification* techniques to reduce the size and memory of computing A_i. These techniques compute some subset of the values of A_i controlled by a mask matrix M, reducing the runtime of MHSA [8, 16] by computing:

$$\underbrace{[\underbrace{softmax(\overbrace{M \otimes QW_i^Q(KW_i^K)^T}^{R-SDDMM})}_{A_i^s}] VW_i^V}_{R-SpMM} \tag{1}$$

where mask M is a mask of 0s and 1s and $\otimes$ is a pair-wise product. The product: $M \otimes (QW_i^Q(KW_i^K)^T)$ in traditional sparse computing terminology is a sampled dense dense matrix multiplications (SDDMM), whilst the product: $A_i^s VW_i^V$ is a sparse matrix dense matrix multiplication (SpMM). However, compared to the sparsity levels studied in sparse computing literature, M is both moderately sparse and regular. Hence we term these operations appropriately prefixed with *regular* as R-SDDMM and R-SpMM respectively. We define regularity in section 4.

Sparse-MHSA Patterns. A variety of sparse transformers have been proposed in the literature [8, 16, 35, 61]. For example, the strided and windowed patterns (figure 2 far left and 2^{nd} from left) which implement Longformer (written in TVM) [8, 12], and the blocked pattern (figure 2 2^{nd} from right) which implements Reformer (written in JAX), and sparse-transformer (written in Triton) [10, 16, 53].

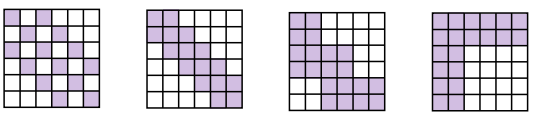
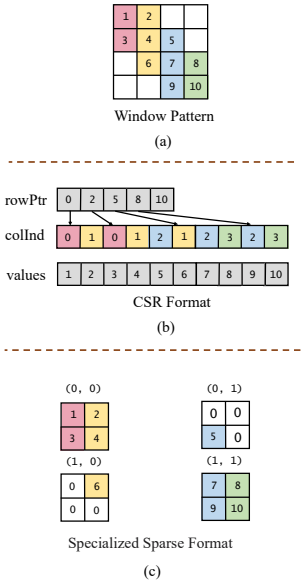


Fig. 2. Examples of 4 commonly occurring sparse-MHSA patterns in the literature. Strided (far left figure), Windowed (2^{nd} from left) [8], Blocked (2^{nd} from right) [16, 35], and Global (far right) [59]. Full attention computes all points

Sparse Formats. To obtain memory savings and performance benefits, sparse-kernels operate on data structures that only store the non-zero values in a sparse matrix. These data structures consist of non-zero values and their respective metadata. The metadata indicates the index of the trailing and leading dimension of a non-zero value. For example, the compressed-sparse-row (CSR) representation in figure 3 (b) contains the *rowPtr* and *colInd* arrays, indicating the leading and trailing dimensions of non-zero values in the *values* array. Many sparse formats have been proposed in the literature including: COO, CSC, BCSR, ELLPACK, DIA, CSF [17, 28], to name a few.

3 Motivation

In this section, we study the implications of using GSFs and CSFs in the moderately sparse context of sparse-MHSA for a particular pattern. Consider two sparse-formats: the CSR [23] general sparse-format and BCSR-like [21] custom sparse-format (used in triton blocksparse kernels [29]). The CSR




```
1 int x = blockIdx.x * blockDim.x + threadIdx.x;
2 int y = blockIdx.y * blockDim.y + threadIdx.y;
3 float answer = 0;
4 for (int k = row_ptr[x]; k < rowptr[x + 1]; k++) {
5     int col = col_idx[k]; // Metadata brought up memory hierarchy
6     answer += A_val[k] * B[col][y];
7 }
```

(d)

```
1  int tx = threadIdx.x; int ty = threadIdx.y;
2  int x = blockIdx.x * blockDim.x + tx;
3  int y = blockIdx.y * blockDim.y + ty;
4  float answer = 0;
5  int block_x = x / block_size;
6  for (int k = 0; k < K / block_size; k++) {
7      if (A_val[block_x][k]) {
8          float **tile = A_val[block_x][k];
9          for (int k_one = 0; k_one < block_size; k_one++) {
10             // Redundant compute
11             answer += tile[ty][k_one] * B[k_one][ty];
12         }
13     }
14 }
```

(e)

Fig. 3. A comparison between SpMM implementations that use the CSR format (b), and a specialized format (c). (d) and (e) are naive SpMM implementations of $C = AB$, when A is represented as a CSR and the specialized format of (c) (resembling a BCSR), respectively.

allows storage of *arbitrary* sparse patterns without extra zero-padding, while the BCSR is curated to store patterns with block-like sparsity. However, since sparse-MHSA has a variety of patterns, we investigate the performance characteristics of using the CSR and BCSR-like formats in figure 3 on the windowed sparsity pattern (figure 2 2^{nd} from left) at moderate sparsity (24% non-zero). Figures 3 (b) and 3 (c) show how the windowed pattern in figure 3 (a) is represented in the CSR and BCSR-like formats respectively. We focus on the SpMM kernel, $C = AB$ where A is a sparse tensor. For illustrative purposes, figures 3 (d) & 3 (e) show the naive implementations of the SpMM kernel when using the CSR and BCSR-like formats respectively, while figure 4 shows the performance profiles of *optimized* versions of the same kernel.

General sparse formats incur high metadata overhead. GSFs are designed to store random & extremely sparse patterns (with <10% of the values computed) by storing metadata for *each* non-zero value, occupying *O(nnzs)* space. However, their metadata storage results in considerable data moved through the GPU memory hierarchy at moderate sparsity levels. Consider the naive SpMM implementation operating on a CSR in figure 3 (d). When reading a value from sparse matrix A, its respective column index must be read (line 5) to multiply with the correct row from B which results in 3 loads of (1) *col_idx*, (2) A, and (3) B to L1-caches and register files. However, the sparsity levels of sparse-MHSA layers are moderate, with up to 2048 megabytes of data produced per layer in SOTA sparse architectures [24, 34], resulting in non-zero data (from A and B) and metadata (from *col_idx*) contending for space in caches and register files. Moreover, this contention occurs within every iteration of the inner loop (lines 4-6) resulting in frequent cache evictions and extraneous data moved from L2 to L1. This is not mitigated even in heavily optimized vendor-libraries for sparse computations like cuSPARSE, which operate on CSRs. As seen in figure 4, cuSPARSE's SpMM transfers 4.73x more data from L2 to L1 compared to cuBLAS's dense matrix-multiplication, despite executing 1/3rd of the compute instructions.

Custom sparse formats lack generality. CSFs are designed to store specific sparsity patterns, reducing the metadata storage required to represent the coordinates of non-zero values. However, using these formats to store sparsity patterns that these formats were not designed for results in redundant compute and storage. Consider the naive SpMM implementation operating on the specialized format that is curated to store block-like sparsity patterns in figure 3 (c). When naively adopting the same format to represent the window pattern, boundary conditions result in redundant storage of 0s in tiles *(0,1)* and *(1,0)*, incurring redundant compute in line 11 within the inner loop of lines 6-14. Moreover, more data is read than is necessary, resulting in extraneous traffic through the memory hierarchy. This is not mitigated even in hand-written kernels. As seen in figure 4, the highly optimized block-sparse kernels [29], hand-written kernels written in triton that operate on a similar CSF, execute 1.4x the floating point operations compared to cuSPARSE. To recover this lost performance, the sparse-kernel in figure 3 (e) needs to be rewritten and performance engineered with another CSF specialized to the windowed structure, resulting in a unique hand-written kernel for *each* sparse-MHSA pattern.

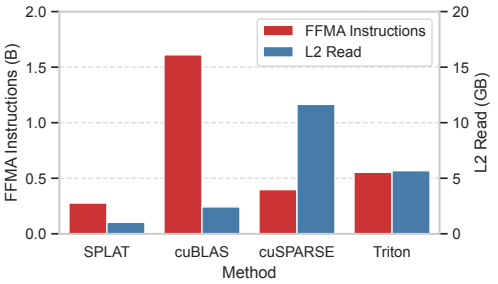


Fig. 4. Profile of R-SpMM sparse-primitive implemented in SPLAT, cuBLAS, cuSPARSE and Triton. Matrices are 1024x1024 with sparse matrices in the window format (see figure 2 - 2^{nd} from left) at 24% density. FFMA is an FP32 fused multiply-add instruction and L2 read is the amount of data-traffic (in GB) from L2 to L1 cache. Lower is better.

SPLAT. In this work, we recognize that an appropriate sparse-format for sparse-MHSA should ideally incur low metadata overhead and high coverage of a variety of patterns without redundant compute & storage. We bridge this gap by introducing a new sparse-format: affine-compressed sparse-row (ACSR). Core to its design is the observation that commonly used sparse-MHSA patterns are static with regularly repeating non-zero sub-structures, requiring metadata only in $O(rows)$ as opposed to $O(nnzs)$ like in GSFs, and without compromising generality like in CSFs. Moreover, we introduce a code-generation mechanism, SPLAT, that produces sparse-MHSA kernels which operate on the ACSR format, reducing the number of compute instructions and data traffic across the memory hierarchy as we observe in figure 4.

4 Overview

Figure 5 shows the workflow of SPLAT. SPLAT takes an input mask and code-generates high-performance sparse-MHSA implementations just-in-time. Its code-generation strategy proceeds in three phases. First, it proceeds with two analysis passes. The first pass analyzes the input mask and ensures that pre-conditions are met for the correctness of later code-generation passes. The second pass generates information required for certain optimisations later code-generation passes can exploit. Second, it proceeds with 3 kernel code-generation passes, producing the R-SDDMM (section 6), Softmax, and R-SpMM (see section 7) kernels used to implement sparse-MHSA. Third, it proceeds with an end-to-end code-generation pass (see section 8) that allocates the necessary memory and creates auxiliary objects required for the correctness of kernel optimizations. The output of the end-to-end code-generation phase is a compiled function that can be used in transformer models to implement the sparse-MHSA mechanism. Our code-generation scheme produces high-performance sparse-MHSA implementations that store sparsity in our novel custom format: affine-compressed-sparse-row - ACSR (see section 5) that leverages the regularity of these patterns.

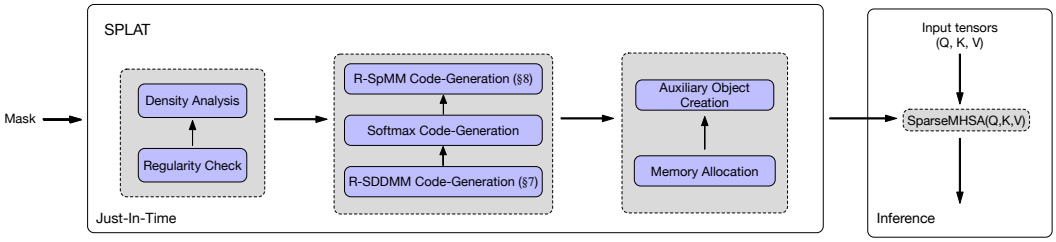


Fig. 5. An overview on SPLAT's inner mechanics and how its just-in-time strategy produces compiled sparse-MHSA kernels for inference.

Affine-Compressibility and Regularity. As observed in section 3, appropriate sparse formats for sparse-MHSA kernels should *compress* metadata, reducing the number of bytes used to store the indices of each non-zero value. Moreover, such a compression scheme should be able to precisely represent non-zero values' metadata across a variety of sparse-MHSA patterns without redundancy. We achieve both by observing that the *point-set* of commonly occurring sparse-MHSA structures (like in figure 2), consists of rows that are *affine-compressible*, and are therefore *regular*. This observation enables us to create a novel sparse-format that *symbolically* stores the metadata for each row of a regularly sparse structure through an affine function.

Point-Sets. To analyze the geometric properties of sparse-MHSA structures, we interpose their input-masks onto the cartesian coordinate system. For a mask, M, consisting of 0s and 1s, we map the point $M[i][j]$ to the point (j, i). We define the point-set of an input-mask as the set of all points that are 1, i.e. the set of all (j, i) such that $M[i][j] = 1$.

Affine-Compressibility. Affine-compressibility is a property of sets of points on the cartesian coordinate system. It states that a set of points can be compressed, such that they consecutively neighbor each other along the x-dimension (trailing-dimension of a matrix).

Definition 1. *Consider a set of points:* $P = \{(x_1, y_1), (x_2, y_2), ..., (x_k, y_k)\}$ *on the coordinate system. P is affine-compressible if and only if:*

$$\exists a, b \in \mathbb{N}_0, \textit{ such that,} \forall i \in [k-1], \frac{x_i - b}{a} + 1 = \frac{x_{i+1} - b}{a}$$

We denote a, b as the affine-indices of P.

For example, the set $P_1 = \{(0, 0), (2, 0), (4, 0), (6, 0)\}$ is affine-compressible with affine-indices: $a = 2, b = 0$, however the set $P_2 = \{(0, 0), (2, 0), (4, 0), (5, 0)\}$ is not.

Regularity. Regularity is a property of a sparse-MHSA mask, building upon the concept of affine-compressiblity. We define a sparse-MHSA mask, M, to be *regular* iff every row in its corresponding point-set, P, is affine-compressible. Hence, a regularly sparse mask is amenable to metadata compression by symbolically storing the dense indices of the trailing-dimension of a sparse matrix. For example, in figure 6 (b), we see for each row of the window pattern the respective linear-transformation (denoted as a) and translation (denoted as b).

5 Affine-Compressed Sparse-Row

We introduce the ACSR format to store regularly sparse matrices. The ACSR format leverages the regularity of sparse-MHSA matrices to store metadata in the order of number of rows with its metadata, the affine-indices, exposing various optimization opportunities. We detail the construction of the ACSR in section 5.1, and the optimizations its metadata exposes in section 5.2.

5.1 ACSR Construction

The ACSR comprises of two arrays: non-zero values, and metadata. The metadata symbolically records the index of the trailing dimension for each non-zero value in a particular row. ACSR represents a sparse matrix by computing the affine-indices per row and compressing data across the trailing dimension such that non-zero values consecutively neighbor each other. For example, in figure 6, the original 2-D matrix in figure 6 (a) is compressed across the trailing-dimension to 6 (b). Each row in 6 (b) has the triplet: a (linear-transformation), b (translation), and $nnzs$ (number of non-zero-values) as metadata. If $sparse_i$ is the index of a non-zero value's trailing dimension in the ACSR, then $(sparse_i \cdot a) + b$ is the index of the trailing dimension in the original sparse matrix. Consider the location with value 14 in figure 6 (b); it is at $sparse_i = 1$ and has $a = 1$, $b = 1$, and $nnzs = 4$ as metadata. The index of its trailing dimension in figure 6 (a) is thus $(1 \cdot 1) + 1 = 2$. The metadata consists of the $(a, b, nnzs)$ triplet per row, occupying $O(rows)$ rather than $O(nnzs)$ space.

However, to construct an ACSR, the affine-indices for each row of a sparse 2-D matrix need to be computed. This can be error-prone for a user to implement and can be avoided by observing that for a given row, y, in a regularly sparse 2-D matrix: $a = i_{1,y} - i_{0,y}$ and $b = i_{1,y}$, where $i_{0,y}$ & $i_{1,y}$ are the first two points' column indices in row y. Once the affine-indices for each row is computed, we can check to see if the entire pattern is then affine-compressible by computing: $\frac{i_{x,y}-b_y}{a_y} = \frac{i_{x-1,y}-b_y}{a_y} + 1$ and checking $(i_{x-1,y} - b_y) \mod a_y = 0 \wedge (i_{x,y} - b_y) \mod a_y = 0$, where a_y, b_y are the affine-indices of row y, and $i_{x,y}$ is the x^{th} non-zero value in row y.

5.2 ACSR Properties

The ACSR exposes novel optimization opportunities for R-SDDMM and R-SpMM kernels at the moderate sparsity levels observed in sparse-MHSA.

Reduction in Predicated Execution. Operating on sparse inputs may result in an imbalance of work across threads within a warp as certain input regions are potentially more dense than others. The ACSR, in storing the dense indices of the trailing dimension symbolically via affine-indices, exposes which regions of a sparse tensor are identical at the granularity of a row. For example, if different rows have identical linear-transformations, translations, and number of non-zero values, then they have data placed in identical trailing indices. Rows with identical affine-indices can be re-mapped to operate on threads within a warp to reduce predicated execution.

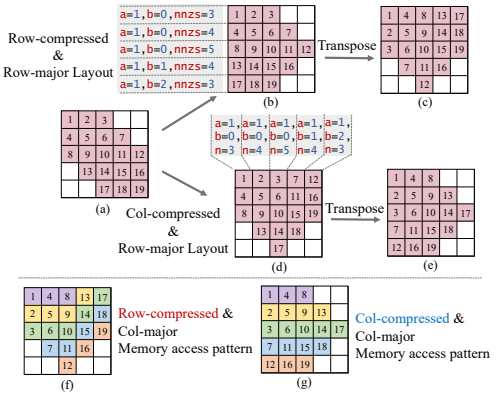


Fig. 6. The 4 different data-layouts an ACSR can take: (b) row-wise compressed row-major, (c) row-wise compressed col-major, (d) col-wise compressed row-major, (e) col-wise compressed col-major. For (f) and (g), colors represent elements of the same column. The a, b, and $nnzs$ represent a row's linear-transformation, translation, and number of non-zero values respectively, constituting the metadata. The a and b variables are the affine-indices of a row.

Favorable read/write access patterns. In R-SpMM kernels, memory accesses to conventional sparse-formats can be un-coalesced. To coalesce these accesses, contiguous elements in a column need to be laid out in contiguous memory addresses, which our construction in section 5.1 does not do. Fortunately, regularly sparse

kernels are also *column-wise* affine-compressible. When compressing data across the trailing dimension and laying out data in column-major, contiguous memory addresses in the ACSR contain contiguous elements in a column of a sparse matrix as shown in figure 6 (g).

Fast indexing. Certain sparse kernels check whether an index in a sparse 2-D matrix is non-zero by traversing a region of values in a sparse-format. For example, to identify if a point $(dense_i, dense_j)$ (leading, trailing dimensions resp.) is non-zero in a CSR requires a traversal of all the points $[rowPtr[dense_i], rowPtr[dense_i + 1]]$. However, the ACSR can compute the answer in $O(1)$ time and metadata accesses by computing: $dense_j\%(a) == 0 \wedge dense_j - b > 0$, where a and b are the affine-indices of row $dense_i$.

6 High-Performance R-SDDMM

An important optimization applied to GPU implementations of SDDMM kernels is tiling to improve reuse and reduce thread-divergence. This involves deciding a mapping of thread-blocks to outputs. Different tiling strategies have been explored within the context of random and extreme sparsity [27, 32, 38] (see section 10 for more details). However such strategies either target extremely sparse matrices or operate on specific sparse-formats.

Comparatively, we leverage the regular nature of the sparsity patterns and the ACSR format to provide a novel, inexpensive tiling strategy for the R-SDDMM kernel (see section 6.3). We show that our tiling approach increases cache reuse, and memory coalescing whilst reducing thread-divergence and redundant compute with *strong optimality guarantees* (see section 6.3).

6.1 Observations

The geometric diversity of sparse-MHSA patterns gives rise to several possible arrangements of thread-blocks over the output, C. Each arrangement trades off different factors that impact performance. We categorize each sparse pattern as either polygonal or strided. Polygonal patterns comprise of non-zero values that are clustered together, with no gaps between them like the windowed and blocked pattern. Strided patterns consist of non-zero values which have constant gaps between them, each non-zero value having no neighbor.

Polygonal Patterns. Figures 7 (c) and 7 (d) show two valid tiling arrangements for the same polygonal pattern. 7 (d) incurs more threads with divergent control-flow compared to 7 (c), as more threads within a thread-block exceed the boundary of the pattern and are predicated to terminate, diverging from the threads that compute output values. Instead 7 (c) incurs threads with redundant compute as thread-blocks that overlap (orange points) compute the same values. The more performant tiling arrangement between the two will depend on the relative costs associated with thread-divergence, redundant compute, and number of thread-blocks.

Strided Patterns. Figures 7 (g) and 7 (h) show two valid tiling arrangements for the same strided pattern. 7 (h) exhibits low spatial locality compared to 7 (g), as thread-blocks operate on outputs that do not re-use rows and columns from the input. Instead, thread-blocks in 7 (g) issue un-coalesced reads to input matrices as they operate on outputs with a constant stride. Additionally, 7 (h) exhibits increased divergent control-flow and uses more thread-blocks compared to 7 (g). The more performant tiling arrangement between the two will depend on the relative costs associated with un-coalesced memory accesses, divergent control flow, spatial locality, and number of thread-blocks.

Our observations indicate that 4 factors impact the performance of a tiling arrangement. (1) The amount of redundant compute between thread-blocks that overlap and compute the same output. (2) The amount of thread-divergence within a thread-block by being placed on irregular boundary conditions. (3) The amount of reuse within a thread-block by computing outputs that share either rows or columns of input matrices A and B. (4) The number of memory access/write

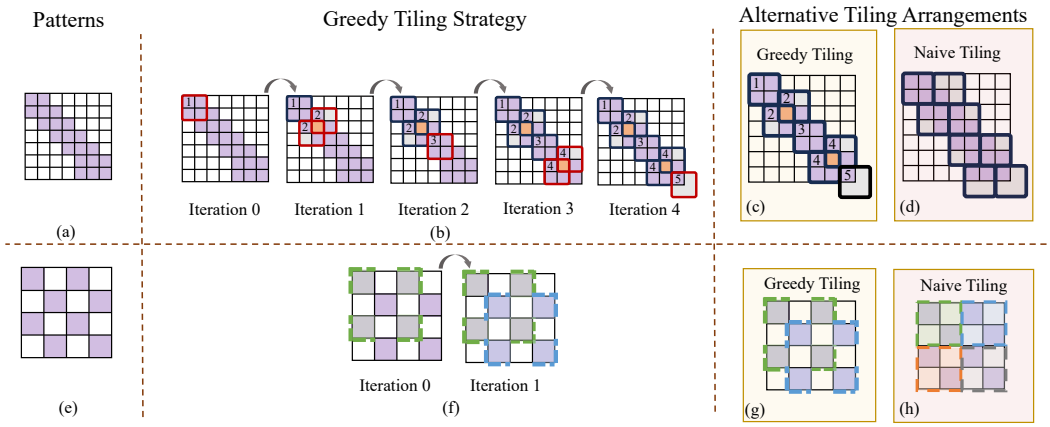


Fig. 7. Different ways thread-blocks can tile strided and polygonal patterns. (a) and (e) are the two patterns. (b) and (f) demonstrate our novel poset tiling strategy. (c) and (d) show two strategies to tile polygonal patterns. Gray represents thread-divergence, and orange represents redundant compute. (g) and (h) show two strategies to tile strided patterns. Numbers on the mask represent what iteration in the for-loop (line 4 of algorithm 1) the thread-block was placed.

requests issued by a warp that are coalesced by reading/writing to contiguous memory locations. A good code-generation scheme should generate a tiling arrangement that reduces the cost of each.

6.2 R-SDDMM Performance Characterisation

We first develop a cost model that explicitly reasons about each of the four factors that affect the performance of a tiling strategy. We achieve this by developing expressions to compute each of these factors as a function of the thread-block arrangement of a given tiling strategy.

Thread-Blocks. We define a thread-block to be a mapping between a logical rectangle of threads of size $m \times n$ to points on a mask, M.

Definition 2. *A thread-block TB_k consisting of $m \times n$ threads that partially covers a point-set P is defined by a tuple: $(t, s), t \in \mathbb{N} \times \mathbb{N}, s \in \mathbb{N}$. We further define the compute of a thread-block as $Comp(TB_k) = \{t + (i * s, j * s) | i \in \{0, \ldots, m-1\}, j \in \{0, \ldots, n-1\}\}$.*

Finally, we define its cover, anchor-point, and stretch factor as:

$$Cov(TB_k) = Comp(TB_k) \cap P,\ Anc(TB_k) = t,\ Str(TB_k) = s$$

The cover and anchor-point represent the points a thread-block computes and its top-left corner (its translation from the origin) respectively. For example, in figure 7 (g), the cover of the two thread-blocks is $Cov(TB_0) = \{(0,0), (0,2), (2,0), (2,2)\}$ (green thread-block), with $Anc(TB_0) = (0,0)$, and $Cov(TB_1) = \{(1,1), (3,1), (1,3), (3,3)\}$ (blue thread-block), with $Anc(TB_1) = (1,1)$. The stretch factor of a thread-block determines how far apart threads in neighboring rows and columns will be placed when covering a point-set. For example, the two thread-blocks in figure 7 (g) have a stretch factor of 2.

6.2.1 Factors affecting performance. Definition 3 gives the mathematical formulation of the four factors impacting the performance of a R-SDDMM kernel. We give intuitions for those definitions next. Note that thread-divergence and redundant compute are aggregate sums, while reuse and memory coalesced requests are averages across all thread-blocks.

Thread-divergence. Within a thread-block, threads that exceed the boundary conditions of a mask deviate control flow from threads that do not. Although thread-divergence happens within a warp, due to the irregular boundary conditions in regularly sparse masks, oftentimes threads that exceed the boundary of a mask exhibit thread-divergence. Therefore, we define the collective thread-divergence of an arrangement as the number of threads that do not cover a point in the point-set. (See ϕ_{TD} in definition 3).

Redundant Compute. An arrangement's redundant compute is the number of excess threads that do not do useful work across all thread-blocks. This amounts to a sum of all the threads in the arrangement subtracted by both the number of points in the point-set and the number of threads that have divergent control flow. (See ϕ_R in definition 3).

Reuse of Thread-block. Threads within a thread-block that do useful work usually reuse values of the input rows or columns. The threads that do not do useful work fall into two categories: (1) Divergent threads, (2) redundant threads. To compute the reuse of a thread-block, we compute the fraction of threads within a thread-block that are both not divergent and redundant. Hence, we define the reuse of an arrangement to be the average reuse across all thread-blocks. (See ϕ_{RU} in 3).

Degree memory requests are coalesced. The degree to which memory requests of a warp within a thread-block are coalesced is *inversely proportional* to a thread-block's stretch factor. Since both the inputs are dense, the larger the stretch factor, the larger the stride in reads issued to inputs, and writes issued to outputs. Hence, we define the amount of memory coalescing as the average stretch factor across all the thread-blocks in an arrangement. (See ϕ_{CMR} in 3).

Definition 3. *Consider an arrangement of thread-blocks, $TB = \{TB_1, TB_2, ...TB_\lambda\}$ each containing $m \times n$ threads, covering a point-set, P such that $\bigcup_{TB_i \in TB} Cov(TB_i) = P$. We define its collective thread-divergence (ϕ_{TD}), redundant compute (ϕ_R), reuse (ϕ_{RU}), and coalesced memory-requests (ϕ_{CMR}):*

$$\phi_{TD} = \left| \left(\bigcup_{TB_i \in TB} Comp(TB_i) \right) \backslash P \right| \qquad \phi_R = \lambda mn - |P| - \phi_{TD}$$

$$\phi_{RU} = \frac{mn - \frac{\phi_{TD}}{\lambda} - \frac{\phi_R}{\lambda}}{mn} = \frac{|P|}{\lambda mn} \qquad \phi_{CMR} = \frac{1}{\lambda} \sum_{TB_i \in TB} \frac{1}{Str(TB_i)}$$

We illustrate divergent threads and redundant threads in figure 7 (c) as gray and orange respectively, with $\phi_{TD} = |\{(1,3),(3,1),(6,4),(6,7),(7,7),(7,6)\}| = 6$, and $\phi_R = 7 * 4 - 20 - 6 = 2$. We compute the reuse of 7 (g): $\frac{2\times2-0-0}{2\times2} = 1$ and 7 (h): $\frac{2\times2-8/4-0}{2\times2} = \frac{1}{2}$, as well as the coalesced memory requests of 7 (g): $\frac{1}{2}(\frac{1}{2} + \frac{1}{2}) = \frac{1}{2}$, and 7 (h): 1.

Cost Model. A performant tiling arrangement will minimize redundant compute, thread-divergence, and stretch-factors of thread-blocks whilst maximizing reuse and coalesced memory requests. This amounts to minimizing ϕ_{TD}, and ϕ_R, whilst maximizing ϕ_{RU} and ϕ_{CMR}.

On one hand, minimizing both ϕ_{TD} and ϕ_R and maximising ϕ_{RU} corresponds to reducing λ, since the dimensions of thread-block: m, n and the size-of the point-set: P, are all fixed. Therefore, optimal tiling arrangements that reduce the costs of thread-divergence, redundant-compute, and low reuse will minimize the number of thread-blocks used. On the other hand, maximising ϕ_{CMR} corresponds to reducing $Str(TB_i)$ for all thread-blocks in the arrangement. Therefore, a good cost function will increase with the number of thread-blocks used, and decrease when ϕ_{CMR} increases.

Definition 4. *Consider an arrangement of thread-blocks, $TB = \{TB_1, TB_2, ...TB_\lambda\}$ each containing $m \times n$ threads covering a point-set, P. Its cost is denoted as $Cost(TB)$ and is computed as follows: $Cost(TB) = \frac{\lambda}{\phi_{CMR}}$*

6.3 Poset Tiling

We develop a tiling strategy - poset tiling - to tile patterns with *optimality* guarantees according to our cost model. Given a mask, M, whose point-set is P, it outputs an arrangement of thread-blocks that covers P by computing the anchor-points where thread-blocks should be placed. It computes these anchor-points by successively computing a set $\top$, using the comes-before (CB) relation.

Definition 5. *Suppose we have a point-set, P that is partially tiled by $TB = \{TB_1, TB_2, ...TB_k\}$. Then given two points, $(x_1, y_1), (x_2, y_2) \in P$, we say that $x \leftarrow y$ (i.e. x CB y) iff $x_1 \leq x_2 \wedge y_1 \leq y_2$. Moreover, let P' be the set of un-covered points of P. We define $\top$ to be the set of points in P' such that: $\forall p_i \in \top, \nexists p_j \in P'$ such that $p_j \leftarrow p_i$.*

Intuitively, the set $\top$ defined over a partially covered point-set, P, represents a set of uncovered points that, when used as the anchor-points of thread-blocks, cover a large uncovered portion of P. For example, in figure 7 (b) (Iteration 2) when the points in $\top = \{(4, 5), (5, 4)\}$ are treated as the anchor-points of thread-blocks, we produce the arrangement in 7 (b) (Iteration 3). These additionally placed thread-blocks cover 6 uncovered points.

Algorithm 1: Poset Tiling

```
   Inputs : P, m, n
1  AncPt ← ∅;
2  Rem ← P;
3  s ← stretchFactorSelection(P);
4  while ∪_{TB_i ∈ TB} Cover(TB_i) ≠ P do
5  |  ⊤_curr ← f_⊤(Rem);
6  |  AncPt ← AncPt ∪ ⊤_curr;
7  |  for (x, y) ∈ ⊤_curr do
8  |  |  Rem ← P − {⋃_{i∈[m]} x + i × s} ×
   |  |    {⋃_{j∈[n]} y + j × s};
9  |  end
10 end
   Output: AncPt
```

Algorithm 1 demonstrates how poset tiling covers a point-set, P, using the set $\top$. It takes as input a point-set, P, to cover, and the dimensions of the thread-blocks ($m \times n$) used to cover P. It outputs a list of points representing the anchor-points of an arrangement of thread-blocks that covers P. It begins by initializing the answer (*AncPt*) to $\emptyset$ in line 1, and the set of uncovered points (*Rem*) to P in line 2. It decides a suitable stretch factor to apply to thread-blocks (line 3) by calling a sub-routine *stretchFactorSelection* (described in 6.3.1). The main loop in line 3 iterates until *Rem* is $\emptyset$, which occurs when we have a cover of P. At each loop iteration, line 4 computes the current $\top$ ($\top_{curr}$) over the uncovered points *Rem*, adding this to the solution set *AncPt*. Finally, using $\top_{curr}$ as the anchor-points of thread-blocks stretched by a factor of s, lines 6-8 remove points from *Rem* that will be covered.

Figure 7 (b) Iterations 0-4 represent each iteration of the main loop in poset tiling.

Theorem 1. *Given point-set P and an arrangement of thread-blocks $TB_{poset} = \{TB_1, TB_2, ..., TB_{\lambda_{poset}}\}$, each of size $m \times n$, generated by algorithm 1 to cover P. Let $TB_{opt} = \{TB_1, TB_2, ..., TB_{\lambda_{opt}}\}$ be the arrangement of lowest possible cost to cover P. For the definitions of the windowed, blocked & strided patterns in appendix B, we have for the windowed and blocked pattern that:*

$$\frac{Cost(TB_{poset})}{Cost(TB_{opt})} \leq 1 + \frac{m}{l} \tag{2}$$

Where l is the maximum number of points in a row of the mask. Moreover, for the strided pattern, the cost of the arrangement is optimal. See Appendix E for the proof.

6.3.1 Stretch Factor Selection. The stretch factor of thread-blocks may influence the number of thread-blocks used in a tiling arrangement. For example, take the strided pattern in figure 7 (g) and

7 (h). By increasing the stride from 1 to 2, we go from the arrangement in 7 (h) which uses 4 thread-blocks, to the arrangement in 7 (a) which uses 2 thread-blocks. Therefore, a good stretch factor will balance the cost of issuing un-coalesced memory requests with the number of thread-blocks used in an arrangement.

A naive stretch factor selection sub-routine will run poset tiling for every possible stretch factor, enumerating the cost of each arrangement and selecting the factor that corresponds to the lowest cost arrangement. Since the stretch-factor is bounded by the sequence length, N, this will return: $s = \text{argmin}_{i \in [N]} \frac{\lambda^i}{\phi^i_{CMR}}$. Where λ^i and ϕ^i_{CMR} are the number of thread-blocks and degree of coalesced memory-requests of an arrangement produced by poset tiling with stretch factor i, respectively. However, we make two observations that reduces the number of arrangements to search through.

```
1 rSDDMMKernel(map<int, coord> idxToOut,
2     int s, coord* metadata, float** A,
3     float** B,float**C) {
4   tx = threadIdx.x; ty = threadIdx.y;
5   bx = blockIdx.x; by = blockIdx.y;
6   blockId = by*gridDim.x+bx;
7   out_ix = idxToOut[blockId].x+tx*s;
8   out_iy = idxToOut[blockId].y+ty*s;
9   out = 0;
10  for (k = 0; k <= K; k++) {
11      out+=A[out_iy][k]*B[k][out_ix];
12  }
13  // fast indexing to place in ACSR
14  a = metadata[out_iy].a;
15  b = metadata[out_iy].b;
16  C[iy][(ix-b/a)] = out;
17 }
```

Listing 1. R-SDDMM naive Kernel

For polygonal patterns, stretching a thread-block will only reduce its cover (see Appendix C for the proof). Hence, for polygonal patterns, we return 1.

However, for strided patterns, stretching a thread-block may increase its cover. To aid us in cutting down the space of stretch factors to search through, we make the following observation. Consider a strided pattern with mask, M. Define its stride to be the number of points between two successive non-zero values in a row of M, denoted as X. Then we have that the cost of an arrangement is minimized when applying algorithm 1 with stretch factor $s = \text{argmin}_{d_i \in factors(X)} \frac{\lambda^{d_i}}{\phi^{d_i}_{CMR}}$ (See Appendix C for the proof).

```
1 cudaFunc codeGenRSDDMM(mask M) {
2   // Runs Algorithm 1.
3   int TBCount,
4   map<int, coord> anchorPoints,
5   int s = gen_map(M);
6   // Compile with number of TBs.
7   cudaFunc func =
8       compile(launcher<TBCount,s>);
9   return func,anchorPoints;}
10
11 template<int TBCount, int s>
12 void launcher(A,B,metadata,C,
13       map<int, coord> idxToOut) {
14   // Launch kernel.
15   Dim3 TBDim(/*TB Size.*/);
16   rSDDMMKernel<<<TBCount,
17           TBDim>>>(idxToOut,
18               s, A, B,C);}
```

Listing 2. R-SDDMM code-generator and launcher

6.4 R-SDDMM Kernel Code-Generation

We show the code-generation pass and a naive R-SDDMM kernel in listing 2 and listing 1 respectively. The code-generation pass takes in an input-mask (line 1) and first applies poset tiling to generate the thread-block count, anchor-points, and stretch factor (line 5). It then uses the thread-block count to instantiate the R-SDDMM launcher and compiles this function (lines 7-8), returning the function pointer, func. Finally, it returns the function pointer and anchor-points (line 9).

We illustrate a naive implementation of the R-SDDMM kernel for brevity. The R-SDDMM kernel takes as input the anchor-points (idxToOut), stretch-factor (s), left (A), and right (B) input matrices, and space to store the output (C). Lines 7-8 use the thread-block id to index the map to recover the

anchor-point of the thread-block the current thread belongs to. It uses this anchor-point to compute the row of A, out_{iy}, and column of B, out_{ix}, to dot-product in lines 10-12. Finally, it indexes the ACSR metadata to get the affine-indices for the out_{iy} row, applying the correct linear-transformation and translation to store the answer, out, at the correct index in the ACSR non-zero values array, C (in lines 14-16).

7 High-Performance R-SpMM

Similar to the R-SDDMM kernel, we need to devise novel techniques to generate high-performance R-SpMM code to leverage the full potential of the ACSR properties mentioned in Section 5. We first show how to construct a naive R-SpMM kernel that uses the ACSR format, then incrementally present two optimizations that enables SPLAT to generate high-performance R-SpMM implementations.

7.1 Observations

The Algorithm in listing 3 shows a naive implementation of the R-SpMM kernel $C = AB$, where A is a sparse matrix represented in the ACSR format and B is a dense matrix. Traditionally, SpMM kernels iterate over the non-zero values of A and multiply these with a corresponding value from B (see figure 3 (d) for an example). However, in R-SpMM kernels, up to 50-70% of A can contain non-zero values. Rather than iterating over only the non-zero values, we treat the R-SpMM kernel as a dense computation and iterate over the size of the entire trailing dimension (leading dimension of B) of matrix A. To reduce redundant computation, we place a guard condition (see listing 3 line 10) to skip iterations where values in A are 0. We can leverage the ACSR metadata to implement the guard condition in $O(1)$ through the observation that $A[dense_y][dense_x]$ exists iff:

$$\left(dense_x \% AffineIndices[dense_y].a == 0\right) \wedge \left(dense_x - AffineIndices[dense_y].b >= 0\right)$$

Nevertheless, listing 3 has two issues. (1) In SpMM kernels, non-zero values in the product of AB are produced only when non-zero values from A multiply with non-zero values from B. However, by letting the loop in line 4 iterate across the entire trailing dimension of A (leading dimension of B), we end up with identical loop counts regardless of the degree of sparsity in A. (2) The predicate in line 10 will result in control divergence between threads in a warp when corresponding attempts to read values from A are non-zero for certain threads, but zero for others. Moreover, this divergence is exacerbated by being placed within a loop that may run for 1000s of iterations. We propose optimizations to mitigate each issue in section 7.2.

7.2 Optimizations

Span specialisation. To mitigate issue 1, we observe that sparse-MHSA structures tend to have chunks of non-zero values. For example, the windowed pattern contains all its non-zero values in chunks surrounding the matrix's main diagonal. Therefore, a thread-block reading a selection of rows from a sparse matrix A in the window pattern need not iterate across all dense indices in the trailing dimension and can start at the first non-zero value and end at the last non-zero value: its *column-span*. Suppose a thread-block is reading a collection of rows, $r_1, r_2, ...r_k$ from sparse matrix A. Its column-span can be computed in $O(k)$ time through:

$$span(r_1, r_2, ...r_k) = [\min_{r_i \in [r_1, r_k]}(AI[r_i].b), \max_{r_i \in [r_1, r_k]}(AI[r_i].nnzs \cdot AI[r_i].a + AI[r_i].b)]$$

where AI are the $AffineIndices$. Figure 8 (a) demonstrates this *span-specialisation*. The thread-block that reads values from the first two rows of A iterates from index 0 to 2 in line 4 of listing 3.

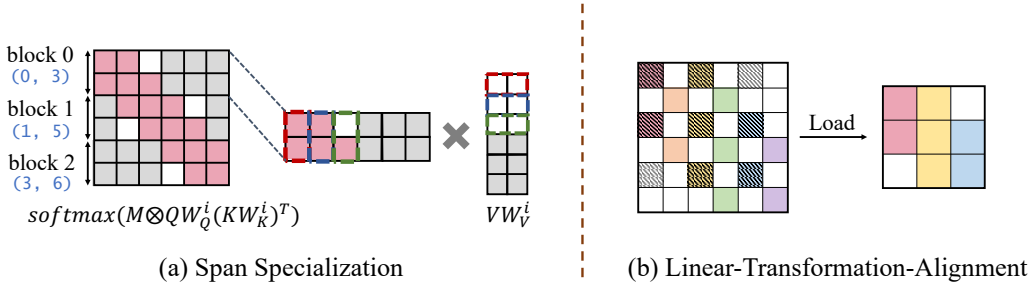


Fig. 8. Different SpMM optimizations. (a) span-specialisation, white are redundant reads. (b) - linear-transformation-alignment, colors represent points whose affine-indexes are identical (aligned), hashed boxes correspond to values loaded into the same thread-block.

Linear transformation Alignment. To mitigate issue 2, we observe that if rows in the ACSR have the same affine-indices, data is placed in identical indices across the trailing dimension. We can re-map threads within a warp to operate on rows with identical affine-indices, ensuring that threads execute the body of the main loop in tandem. Figure 8 (b) demonstrates this optimization; when loading from A, only 2 out of 9 threads diverge in control flow, as opposed to 4 out of 9 without this optimization.

```
1 template<int layout>
2 rSpMMKernel(A,B,metadata,metaOpt) {
3   ix = threadIdx.x + blockIdx.x
4          * blockDim.x;
5   iy = threadIdx.y + blockIdx.x
6          * blockDim.x;
7   out = 0;
8   for(dense_i=0;dense_i<=K;dense_i++){
9     // Leverage O(1) indexing
10    if (index(A,iy, dense_i,layout)){
11      out += value(A,iy,dense_i,
12         layout)*B[dense_i][ix];}}
13  C[iy][ix] = out;}
```

Listing 3. Naive SpMM Kernel With Guard Clause

7.3 R-SpMM Code-Generation

The R-SpMM code-generation pass, shown in listing 4, ingests ACSR metadata and code-generates a R-SpMM kernel. It returns an R-SpMM kernel (spmmFunc), thread-block count (TBCount), metadata required for optimizations (spmmMetaOpt), and layout of the ACSR (layout), (see lines 5-7). The optimizations metadata, spmmMetaOpt, contains a map of thread indices to 2 pieces of information. (1) The respective start and end loop indices, implementing span-specialisation. (2) The respective row of the ACSR to load, implementing linear-transformation-alignment. The boolean layout flag indicates the data-layout the ACSR should be for correct data indexing. For high-performance, the layout of the ACSR depends on the density of the input mask, see section 8 for more details.

```
1 void codeGenRSpMM(densityCls,metadata,
2                   metaOpt) {
3   // The spmm kernel and optimisations
4   // depend on the metadata.
5   spmmFunc spmmFunc,TBCount,
6   spmmMetaOpt,layout=genSpMM(metadata,
7                     densityCls);
8   // Compile the launcher
9   launchFunc func
10      = compile(launcher<TBCount,
11                spmmFunc,layout>);
12  return func,spmmMetaOpt;}
13
14 template<int TBSpmm,typename spmm,
15          int layout>
16 void launcher(A,B,metadata,C,metaOpt){
17   dim3 TBDim(/*TB dimensions*/);
18   spmm<layout><<<TBSpmm, TBDim>>>(AT,
19                       B,C,metadata);}
```

Listing 4. RSpMM code-generation

```
1  codeGenSparseMHSA(float** Mask, int sequenceLength) {
2      // Analsis passes
3      checkRegularity(Mask);
4      metadata = generateACSRMetadata(Mask);
5      densityCls = densityAnalysis(metadata, sequenceLength);
6      // Code-generation passes.
7      rsddmmFunc, anchorPoints = codeGenRSDDMM(Mask); // Section 7
8      softmaxFunc = codeGenSoftmax(Mask, metadata);
9      rspmmFunc,rspmmMetaOpt = codeGenRSpMM(densityCls, metadata); // Section 8
10     // Memory allocator & Auxiliary Data creation.
11     rsddmmOut, softmaxOut, transposeOut, rspmmOut = allocateMemory(metadata);
12     auxiliaryData.memoryAllocations={rsddmmOut,softmaxOut,transposeOut,rspmmOut
       };
13     ... // Store: metadata, anchorPoints and rspmmMetaOpt in auxiliaryData.
14     // data layout reordering optimisation
15     transposeFunc = codeGenTranspose(Mask, metadata, densityCls);
16     // Finally, sparseMHSA function generation.
17     sparseMHSA = compile(sparseMHSAlauncher<rsddmmFunc, softmxFunc,
18                                             transposeFunc, rspmmFunc>);
19     return sparseMHSA,auxiliaryData;}
20
21 template<typename rsddmm, typename softmax, typename transpose, typename rspmm>
22 sparseMHSAlauncher(Q,K,V,auxiliaryData) {
23     // Unpack memory allocations, metadata, anchor-points, and metadata
24     rsddmmOut,softmaxOut,transposeOut,rspmmOut=auxiliaryData.memoryAllocations;
25     ... // Unpack the rest.
26     // Launch all kernels in sequential order.
27     rsddmm(Q,K,metadata,rsddmmOut,anchorPoints);
28     softmax(out,metadata,softmaxOut);
29     transpose(out,metadata,transposeOut);
30     rspmm(out,V,metadata,rspmmOut);
31     return rspmmOut;
32 }
```

Listing 5. End-to-End Code-generator Psuedocode

8 Final Code-Generation of Sparse-MHSA

Tying everything together, SPLAT generates high-performance code for end-to-end implementations of sparse-MHSA. Its code-generation mechanism is shown in algorithm 5, proceeding in four passes. (1) An analysis pass (see lines 3-5) analyzes the input mask to generate information used by later code-generation passes and ensures the legality of optimizations and code-generation. (2) A code-generation pass (see lines 7-9), which ingests the information produced by the analysis pass to generate: R-SDDMM, Softmax, and R-SpMM kernels. (3) A memory allocation and auxiliary data creation pass (see lines 13-15) which allocates enough memory to hold output tensors and creates a data object required for any optimizations for R-SDDMM and R-SpMM kernels. (4) A data-layout reordering pass (see line 15), which reasons about the data-layout of the input tensor to the R-SpMM kernel, inserting a relevant transposition whenever necessary.

Analysis pass. The analysis pass first checks if the mask is regular (see line 3), terminating otherwise. It then generates the affine-indices (see line 4) as described in section 5. Finally, it analyzes the number of non-zero values in the mask (density analysis - see line 5), and if this number is greater than a threshold α, sets the classification returned to dense, else to sparse. This information is required for data-layout re-ordering optimizations for good end-to-end sparseMHSA performance.

Code-generation. The code-generation passes (see lines 7-9) produce high-performance implementations of the R-SDDMM (see section 6) and R-SpMM (see section 7) kernels, as well as objects required to implement optimizations correctly. We use cuDNN's softmax kernel to implement the softmax over the ACSR.

Memory Allocation & Auxiliary Data Creation. Lines 13-15 allocate the necessary amount of memory required to store the output tensors of each of the kernels: R-SDDMM, Softmax, and R-SpMM, and create an auxiliary data object. This data object contains information required for the correctness of the optimizations detailed in sections 6.3 and 7.1. It contains the metadata (the affine-indices and non-zero-values of the ACSR) required for fast-indexing, the anchor-points for poset-tiling, and a rspmmMetaOpt structure that contains thread-level mappings for linear-transformation-alignment and span-specialization.

Data-layout reordering. Reasoning about the data-layout of the ACSR in the R-SpMM kernel is important for a high-performance implementation of sparse-MHSA. At moderate sparsity levels, reading from an ACSR within the R-SpMM kernel is more expensive than writing to it within the R-SDDMM kernel. The R-SDDMM kernel only writes to each output value once, but the R-SpMM kernel reads each input multiple times (across multiple thread-blocks). We select the best format for the R-SpMM kernel by considering the global computations and their formats and inserting a transpose kernel before it whenever necessary according to the output of density analysis. For input-masks with high-density, we transpose the ACSR to a column-compressed & column-major layout, while for input-masks with low-density, we transpose the ACSR to a row-compressed & row-major layout before the R-SpMM kernel. The column-compressed & column-major allows threads to issue coalesced memory requests but requires complex arithmetic to index compared to the row-compressed & row-major layout. At higher density levels, these un-coalesced requests bottleneck kernels as the amount of data read is greater. Our ablations in section 9.5 illustrate this.

9 Evaluation

We evaluate SPLAT against state-of-the-art vendor-libraries (SOTA) and hand-optimized implementations across a *variety* of sparse patterns to demonstrate SPLAT's *generality* and *high-performance*. To this end, we conduct a series of run-time performance studies (section 9.3), memory & compute profile analysis (section 9.4), and ablations & sensitivity studies (section 9.5). We perform run-time performance studies at different granularities: individual sparse-kernels (R-SDDMM & R-SpMM), single layer sparse-MHSA, and end-to-end transformer to demonstrate the efficacy of SPLAT's code-generation framework.

In summary, our results show that SPLAT exhibits considerable speedups against vendor-libraries and hand-optimized implementations over a variety of sparse-MHSA patterns. SPLAT can achieve speedups of up to 2.07x and 5.68x over cuBLAS and cuSPARSE across desired sparsity ranges of [10%, 50%], respectively, further, it achieves up-to 2.05x and 4.05x over hand-optimized kernels in Triton and TVM, respectively.

9.1 Implementation

We implement SPLAT's GPU code-generation mechanism in C++ and Python. Currently, SPLAT is a CUDA code generation system compatible with JAX. SPLAT outputs CUDA implementations of sparse-MHSA that are just-in-time compiled through JAX's CUDA compatible foreign-function-interfaces (FFIs). We use the following software versions: cudatoolkit 11.6, jaxlib 0.4.6, triton 2.1.0, TVM 0.6.0, and pytorch 2.1.0.

9.2 Experimental Setup

We evaluate SPLAT on multiple sparsity patterns, comparing SPLAT generated kernels to real world sparse-MHSA implementations. For each sparsity pattern, we conduct experiments on different granularities in single-precision.

Sparsity Patterns. We evaluate SPLAT's effectiveness across 4 patterns: windowed, blocked, strided, and global. We select these patterns due to their popularity in the deep-learning community [8, 11, 16, 24, 34, 35, 59] as well as the availability of hand-optimized implementations of each sparse-MHSA mechanism. We compare SPLAT generated kernels against each baseline on the sparse-MHSA pattern the respective baseline was designed for.

Experimental Granularity. We compare SPLAT generated code to SOTA implementations at three different granularities: individual sparse-kernel primitives (comparing against cuSPARSE and cuBLAS), single-layer sparse-MHSA (comparing against TVM and triton) & end-to-end transformer (comparing against TVM and JAX), demonstrating speedups across each case.

Baselines. We compare SPLAT generated kernels to a variety of tensor-compilers, deep-learning frameworks and vendor libraries. All individual kernels are compared against cuBLAS & cuSPARSE. For single-layer sparse-MHSA and end-to-end transformer we give, for each pattern, the baseline we compare against.

Strided Pattern. Sparse-MHSA transformer: longformer-strided. Single-Layer sparse-MHSA and end-to-end transformer implementation: TVM[1].

Windowed Pattern. Sparse-MHSA transformer: longformer-windowed. Single-layer sparse-MHSA and end-to-end transformer implementation: TVM[1].

Blocked Pattern. Sparse-MHSA transformer: reformer & sparse-transformer. Single-layer sparse-MHSA implementation: triton[2] (sparse-transformer). End-to-end transformer implementation: JAX[3] (reformer).

Global Pattern. Sparse-MHSA transformer: big-bird. Single-layer sparse-MHSA and end-to-end transformer implementation: JAX[4].

We note that triton is a competitive baseline for block-sparse patterns, and is upto 5x faster than cuSPARSE (NVIDIA's sparse library) [57]. The TVM and JAX baselines are hand-written implementations of longformer and reformer & big-bird respectively, where the index arithmetic is specialised to a particular pattern.

Density Levels. Unless otherwise stated, all our comparisons are conducted at the density levels: [0.4, 0.8, 1.6, 3, 6, 12, 24, 44, 75, 100] except for the triton block-sparse baseline which imposes a lower limit on the block size of inputs (to 16×16). We tune the sparsity of each pattern by varying the width of the stride (strided-pattern), the size of the window (windowed pattern), the size of the block (blocked pattern), or the number of rows/columns computed (global pattern).

Matrix and Model Sizes. All the matrix sizes for the sparse-kernel primitives are 1024x64 corresponding to a sequence length of 1024 and a head dimension of 64. We set the batch size to 32 with 12 attention heads resulting each kernel computing 32×12=384 matrix multiplications. For the rest of the model (in the case of the end-to-end transformer baselines), we set the FFN hidden size to 3072 and set the number of layers to 12. These are common configurations for models such as BERT [20], GPT-1 [58] and GPT-2 [43] base models.

Gemma-2 2B Transformer. Due to the rising popularity of sparse-MHSA, billion parameter sparse-MHSA models have recently been pre-trained [24, 34]. Hence, we use SPLAT generated sparse-MHSA kernels to implement the gemma-2 2B variant [24]. Its architecture consists of alternating layers of dense full-attention and sparse-MHSA (the window pattern) with the window size set to half the sequence length. We note that gemma-2, despite being a sparse-MHSA model, is implemented as a dense computation with appropriate masking to simulate sparse-MHSA. Hence, we fix the density level at its original 37.5% and vary the sequence length instead. Given that

[1]https://github.com/allenai/longformer
[2]https://github.com/ptillet/triton/tree/triton-mlir/python/triton/ops/blocksparse
[3]https://github.com/google/trax/tree/master/trax/models/reformer
[4]https://github.com/huggingface/transformers

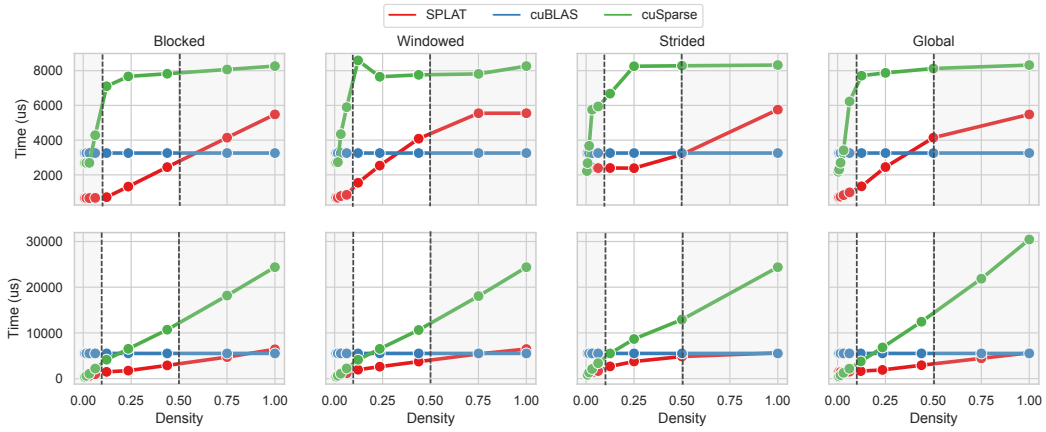


Fig. 9. Run-time performance of sparse-primitives: R-SDDMM and R-SpMM, comparing: SPLAT, cuBLAS and cuSPARSE. The top and bottom rows are the R-SDDMM and R-SpMM results, respectively. The desired density levels observed in sparse-MHSA are in the [10,50]% and are highlighted within the dashed lines.

gemma-2 is trained at a sequence length of 8192, we evaluate on the sequence lengths [2048, 4096] as 8192 results in an OOM on a single GPU. SPLAT only applies to the prefill stage of gemma-2 2B, see our limitations section (section 11) for further clarification.

9.3 Run-Time Performance Study

The primary motivation of the run-time performance study is to answer the question: Can SPLAT be used to accelerate end-to-end sparse-MHSA-based models, and is this speedup *as a result* of SPLAT's code-generation methodology? To answer the first part of the question, we evaluate SPLAT at three levels of granularity. To answer the second part of the question, we analyze the memory profiles of individual kernels and conduct a breakdown analysis of single layer sparse-MHSA showing this speedup is a result of SPLAT's code-generation mechanism.

9.3.1 Individual Kernels Speedups. Figure 9 shows our results for runtime performance. For the R-SDDMM, SPLAT experiences geomean speedups of 2.46x & 5.68x (blocked pattern), 1.29x & 3.17x (windowed pattern), 1.24x & 2.93x (strided pattern), 3.05x & 4.20 (global pattern) over cuBLAS and cuSPARSE respectively. For the R-SpMM, SPLAT experiences geomean speedups of 2.81x & 3.37x (blocked pattern), 2.07x & 2.47x (windowed pattern), 1.51x & 2.33x (strided pattern), 3.12x & 1.68 (global pattern) over cuBLAS and cuSPARSE respectively. All these speedups are reported in the 10%-50% density range.

9.3.2 Single Layer Sparse-MHSA Speedups. Figure 10 (top row) shows our results for runtime performance. SPLAT realizes geomean speedups of 2.05x, 4.05x, 2.12x, and 2.78x over triton, TVM-windowed, TVM-strided, and JAX respectively across the *entire* density range.

9.3.3 End-to-End Sparse Transformer Speedups. Figure 10 (bottom row) shows our results for runtime performance. SPLAT experiences geomean speedups of 1.03x, 1.31x, 1.49x, and 1.78x over Reformer (blocked pattern) implemented in JAX, Longformer (windowed and strided pattern) implemented in TVM, and Big-bird (global pattern) implemented in JAX, respectively across the *entire* density range.

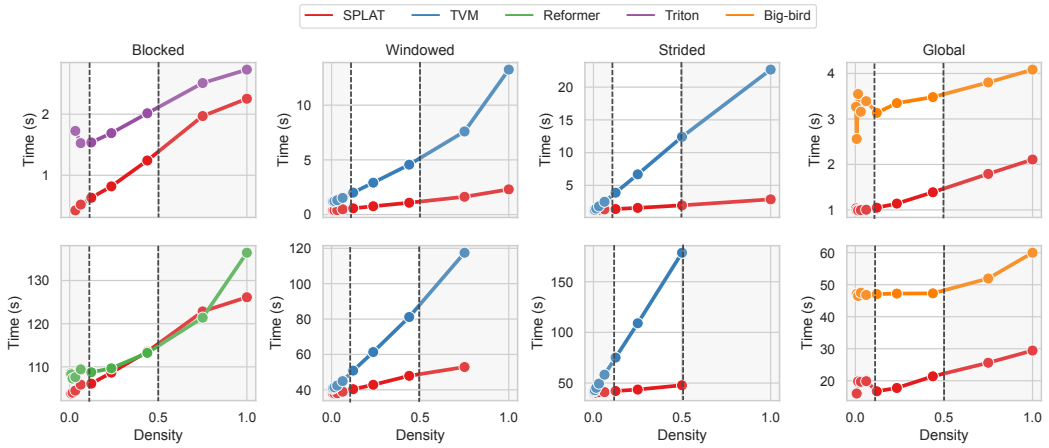


Fig. 10. Runtime performance of a single-layer MHSA and end-to-end sparse transformer, comparing SPLAT against sparse transformer (implemented in Triton), longformer (implemented in TVM), reformer (implemented in JAX), and big-bird (implemented in JAX). The **top** and **bottom** rows are the single-layer sparse-MHSA and end-to-end transformer implementations, respectively. Unplotted points are due to OOM issues, except the Triton baseline, which places a lower limit on the block size. The desired density levels are highlighted within the dashed lines.

9.3.4 Gemma-2 2B. We use SPLAT generated sparse-MHSA kernels to implement the gemma-2 2B variant on the sequence lengths [2048, 4096].

Speedups. Figure 11 shows our runtime performance results. For a single-layer sparse-MHSA SPLAT experiences speedups of 1.105x and 1.109x at sequence lengths of 2048 and 4096 respectively. For an end-to-end transformer SPLAT experiences speedups of 1.011x and 1.021x at sequence lengths of 2048 and 4096 respectively.

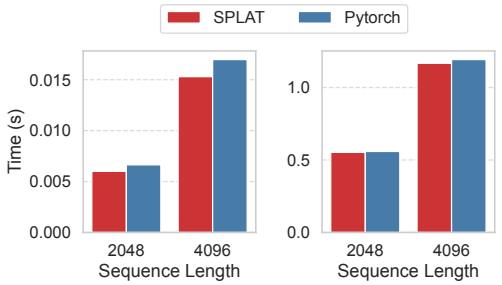


Fig. 11. A comparison of gemma-2 2B's original implementation (Pytorch) against when its sparse-MHSA mechanism is replaced with SPLAT generated kernels (SPLAT) at varying sequence lengths. Left is a single sparse-MHSA layer and Right is an end-to-end transformer.

9.4 Analysis of Performance Results

We now analyze how SPLAT's sparse-primitives achieve speedups over optimized vendor-libraries and hand-written kernels in Triton and TVM. We show that SPLAT's novel code-generation algorithms leverage the meta-data stored in the ACSR effectively to produce favorable memory access and write patterns balanced with enough inter-warp parallelism to hide read/write latencies. We show this by analyzing the memory profiles of all vendor-libraries, hand-written kernels, and SPLAT at a density level of 24% for the blocked pattern (except TVM, which is the windowed pattern). Favorable memory access patterns will read similar amounts of data from global memory to L2 cache, and from L2 to L1 cache, reducing extraneous data-movement through the memory hierarchy; we compute how much more data is transferred from L2 to L1 (denoted as L2 $\rightarrow$ L1), compared to

Table 1. Memory profiles of SPLAT, cuBLAS, cuSPARSE, Triton and TVM of the blocked pattern (except TVM which is the window pattern) at a density level of 24%. `Global` ⟶ `L2` and `L2` ⟶ `L1` is the amount of data transferred from global-memory to L2 cache, and L2 to L1 cache, respectively, as a result of memory reads. `L1` ⟶ `L2` and `L2` ⟶ `Global` is the amount of data transferred from L1 to L2 cache, and L2 to global-memory, respectively, as a result of memory writes. TVM spawns 32 kernels, hence the (x32) notation.

Kernel	Method	Threads/SM	Read (GB)		Write (GB)		Cache Hit Rate (%)	
			`Global` ⟶ `L2`	`L2` ⟶ `L1`	`L1` ⟶ `L2`	`L2` ⟶ `Global`	L1	L2
SDDMM	SPLAT	1152	0.190	3.170	0.377	0.346	44.25	92.91
	cuBLAS	512	0.201	1.610	1.610	1.590	00.41	92.34
	cuSPARSE	576	0.206	7.830	0.662	0.365	42.02	96.98
	Triton	128	0.201	0.432	0.681	0.360	59.21	82.25
	TVM (x32)	2048	0.006	0.358	0.035	0.001	75.23	98.19
SpMM	SPLAT	1152	0.101	0.602	0.805	0.084	57.53	91.10
	cuBLAS	512	1.720	2.420	0.100	0.970	00.10	43.43
	cuSPARSE	1536	0.203	11.330	1.100	0.089	53.71	97.57
	Triton	128	0.482	3.880	2.090	0.126	22.24	90.33
	TVM (x32)	2048	0.002	0.172	0.060	102 KB	89.94	92.35

global memory to L2 (denoted as `Global` ⟶ `L2`) for all hand-written kernels, libraries, and SPLAT. We apply a similar argument to memory write-back patterns, computing the excess data written from L1 to L2 (`L1` ⟶ `L2`), compared with L2 to global memory (`L2` ⟶ `Global`). Our results are in table 1. We systematically compare SPLAT to each vendor library and hand-written kernel.

Vendor-libraries. Analyzing thread access patterns, we report the excess data moved across the memory hierarchy due to the reading of data: 2.98GB (0.501GB), 1.409GB (0.7GB), and 7.624 (11.127GB) for SPLAT, cuBLAS, and cuSPARSE respectively for the R-SDDMM (R-SpMM) kernel. We observe SPLAT's thread-access patterns move significantly less data across the memory hierarchy compared to cuSPARSE, and slightly more compared to cuBLAS. We note cuBLAS, as a dense mat-mul, has regular thread access patterns indexing dense 2-D arrays (as opposed to complex sparse structures), and is thus amenable to favorable access patterns. Nevertheless, SPLAT spawns more threads per streaming-multiprocessor (SM), thus effectively latency hiding expensive memory read operations through inter-warp parallelism. Since cuBLAS's kernel is compute-bound, there is enough reuse to circumvent the need to latency hide memory reading costs.

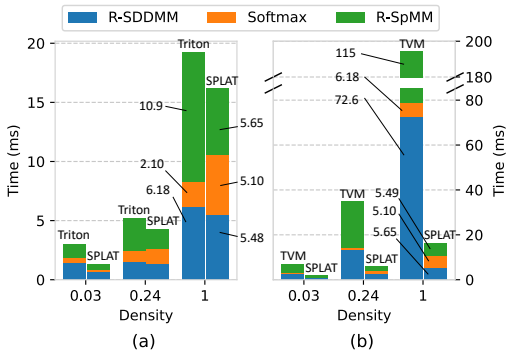


Fig. 12. A breakdown analysis of the three components: R-SDDMM, Softmax and R-SpMM of SPLAT, Triton and TVM's sparse-MHSA primitives. (a) is the blocked pattern, (b) is the windowed pattern.

We similarly report the excess data moved across the memory hierarchy as a result of write-backs: 0.031GB (0.721GB), 0.02GB (0.87GB), and 0.257GB (1.01GB) for SPLAT, cuBLAS, and cuSPARSE respectively for the R-SDDMM (R-SpMM) kernel. We observe that the write-back pattern profile of SPLAT is comparable to cuBLAS, and moves significantly less extraneous data compared to cuSPARSE. Overall, since SPLAT computes less than 1/4th of the values compared to cuBLAS, and has favorable access/write-back patterns it is the fastest of the three.

Hand-written Kernels. Analyzing thread access patterns, we report the excess data moved across the memory hierarchy: 2.98GB (0.501GB), 0.231 (3.398GB), and 0.352GB (0.17GB) for SPLAT, Triton and TVM respectively for the R-SDDMM (R-SpMM) kernel. We observe that SPLAT's access patterns are better than Triton's R-SpMM and both of TVM's kernels (TVM spawns 32 kernels, one for each batch, thus operates on 1/32nd the amount of data compared to SPLAT and Triton). Though Triton's R-SDDMM access patterns are slightly better than SPLAT's, it spawns 9x fewer threads per SM, inadequately hiding read/write latencies. A closer inspection of the kernel indicates this is a result of overusing shared-memory.

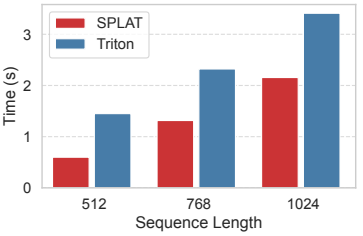


Fig. 13. A comparison of SPLAT generated single layer sparse-MHSA layer with triton's block-sparse kernels on randomly generated regularly sparse matrices at various sequence lengths.

Similarly, we report the excess data moved through the memory hierarchy as a result of write-backs: 0.031GB (0.721GB), 0.321GB (1.964GB), and 0.034GB (0.06GB) for SPLAT, Triton and TVM, respectively for the R-SDDMM (R-SpMM) kernel. We observe that SPLAT's write-back patterns are better than both Triton and TVM's.

Breakdown Analysis. To show that end-to-end sparse-transformers are accelerated due to SPLAT's high-performance code-generation mechanism, we break down the run-times of SPLAT's R-SDDMM, softmax, and R-SpMM kernels in a single sparse-MHSA layer and compare it to triton and TVM in 12. We breakdown these run-times across high, moderate, and low sparsity levels. We see that across all sparsity levels, the collective run-time of SPLAT's kernels is faster than Triton and TVM's.

9.5 Ablation & Sensitivity Studies

9.5.1 Randomly Generated Regular Matrices. We compare SPLAT generated sparse-MHSA kernels to triton's block-sparse kernels on randomly generated regularly sparse matrices at a density level of 37% for a variety of matrix sizes. We generate these matrices by randomly generating the affine-indices, fixing the number of non-zeros for each row to 37% of the size of the trailing dimension. We pick this density because the newest SOTA sparse-MHSA models have a density level of 37% [24, 34]. We vary the sequence length of the matrix in [512, 768, 1024] and fix the head hidden dimension to 64 with 12 attention heads and a batch size of 32 (resulting in matrices of size: [512, 768, 1024]×64). We compare a single layer sparse-MHSA only to triton because of: (1) growing popularity of triton [4, 57], and (2) our experiments indicate that they are the strongest baseline.

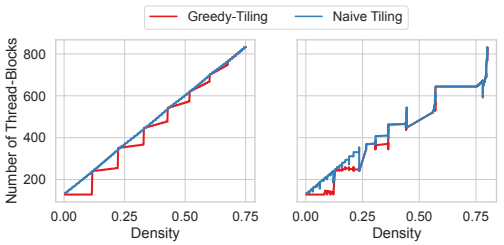


Fig. 14. A comparison of the number of thread-blocks used between different Tiling Strategies for the window (left) and blocked (right) pattern. Lower is better.

Speedups. Figure 13 shows our runtime performance results. SPLAT experiences speedups of 2.42x, 1.77x, and 1.58x on sequence lengths of 512, 768 and 1024 respectively.

9.5.2 R-SDDMM Tiling. High-performance arrangements use a minimal number of tiles. We compare the number of thread-blocks used in poset-tiling against a Naive tiling approach for the blocked and windowed pattern. We use a sequence length of 1024 and vary the density of each

pattern across all possible values. The results are in figure 14. Poset tiling reduces the number of thread-blocks by 1.098 and 1.095 for the window and blocked pattern respectively, on average. The maximum reduction is 1.83 and 1.72 which uses 13568 and 12544 fewer threads for the window and blocked pattern respectively.

9.5.3 R-SpMM Optimisations. We evaluate the benefit of the optimizations: span-specialization and linear-transformation alignment on R-SpMM kernels at varying density levels by comparing these optimizations to implementations where they are disabled.

Figure 15 shows the results for the effects of span-specialisation on the runtime of R-SpMM kernels. We fix a sequence length of 1024 and vary the density of the window and blocked pattern in [0.4, 0.8, 1.6, 3, 6, 12, 24, 44, 75, 100]. Across these density levels, span-specialization results in geomean speedups of: 3.4x and 3.96x for the windowed and blocked pattern respectively. This shows for the density ranges observed in sparse-MHSA [10,50]%, span-specialization achieves speedups. However, for extremely dense inputs where loop counts span the entire trailing dimension, span-specialization can be costly due to extra integer arithmetic to compute the loop start and end indices.

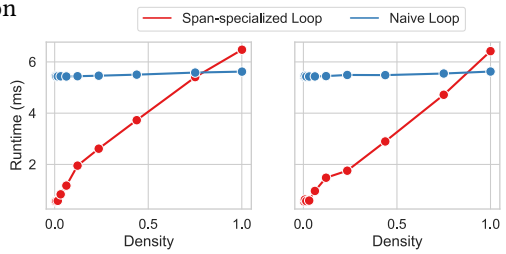


Fig. 15. A comparison between the runtimes of the R-SpMM kernel with and without span-specialisation. Left and right are the window and blocked patterns respectively.

Figure 16 shows the results for the effects of linear-transformation alignment on loads from a regularly sparse matrix in the R-SpMM kernel (algorithm 3 line 8). We fix a sequence length of 1024 and vary the density for the strided pattern across all possible values (varying the stride from 1 to 1024). We compare to a R-SpMM kernel without this optimization (naive loads). Linear-transformation alignment reduces control divergence of loads by 2.73, on average, with a maximum reduction of 8.1.

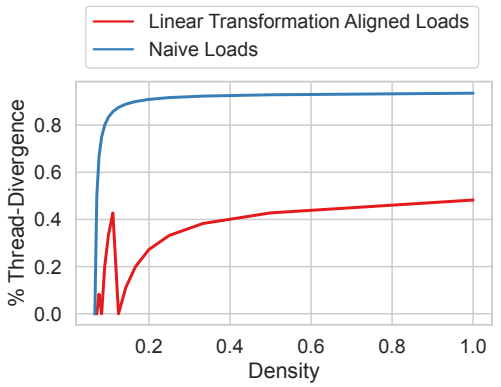


Fig. 16. A comparison between the percentage of threads that exhibit control divergence of loads to a regularly sparse matrix in the R-SpMM kernel for the strided pattern (figure 2 middle). Lower is better.

9.5.4 Data-Layout Exploration. We evaluate the benefits of different ACSR layouts in R-SpMM kernels in figure 17. We compare two layouts combined with the cost of transposing data into these layouts: row-compressed & row-major against column-compressed & column-major as these produce the fastest R-SpMM kernels across various density levels. We compare the runtimes of the R-SpMM kernels with these layouts across density levels [0.4, 0.8, 1.6, 3, 6, 12, 24, 44, 75, 100], categorizing these density levels into sparse inputs (density <10%) and dense inputs (density ≥10%).

Sparse Inputs. Across density levels lower than 10%, the R-SpMM kernel which uses the row-compressed & row-major layout experiences a geomean speedup of 1.37x compared to a column-compressed & column-major layout. The row-compressed & row-major layout requires

less complex index arithmetic to reference data at the cost of un-coalesced accesses to non-zero values which happens due to contiguous elements in a column being placed far apart in memory. At these sparsity levels, the bandwidth has not yet saturated, hence the cost of un-coalesced accesses is relatively low, resulting in faster R-SpMM kernels in this layout.

Dense Inputs. Across density levels greater than 10% the R-SpMM kernel which uses the column-compressed & column-major layout experiences a geomean speedup of 1.6x compared to a row-compressed & row-major layout. The column-compressed & column-major layout enables coalesced reads of data, since contiguous elements in a column are placed in contiguous memory addresses, at the cost of more complex arithmetic to reference this data. For dense inputs where we read a lot of data, the performance benefits of memory coalescing outweigh that of simplified arithmetic, resulting in faster R-SpMM kernels in this layout.

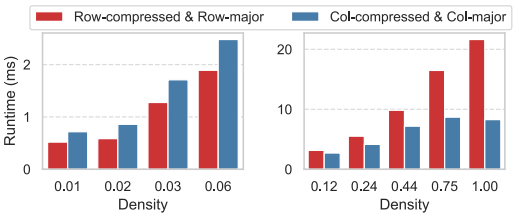


Fig. 17. A comparison of the combined runtimes of the transposition and R-SpMM kernels for different ACSR layouts at various density levels. We use the window pattern. Lower is better.

10 Related Work

Sparse Compilers TACO [36] uses iteration graphs and merge lattices to generate code for compositions of sparse kernels. SparTA [60] proposes sparse tensor annotations to capture the sparsity of moderately sparse pruned nerual networks. Sympiler [13] leverages symbolic analysis to guide an inspector to perform code transformations that are specific to the structure of non-zeros in a matrix. Parsy [14] extends this idea by proposing to inspect sparse kernel data dependencies to produce efficient parallel codes for sparse kernels. These compilers are effective and support a variety of data-formats, however do not support the ACSR format we introduced since it is challenging to leverage the symbolic affine-indices (that represent a variety of indexes in the trailing dimension of a matrix) in sparse-kernels.

Sparse Kernel Optimizations [32] proposes a lightweight tiling strategy for SDDMMs and SpMMs to enhance reuse but is tied to the CSR. [26] proposes a novel 1-d tiling and load-balancing strategy for moderately sparse inputs but is also tied to the CSR. [56] proposes a novel inspector-executor approach that uses an ILP formulation to produce high-performance SpMMs on CPUs. [15] proposes a novel inspection and code-transformation strategy to fuse two sparse kernels where at least one has loop-carried dependencies. [18] proposes a data-aware SpMM that considers input dynamics, targeting GPUs. Though effective, these optimizations do not target the ACSR with its unique metadata layout.

Compiling compositions of regular and irregular sparse programs. There has been a variety of work gone into exploiting the structure of irregular programs at compile time [6, 45, 47, 50], similar to SPLAT. [6] leverages polyhedral compilation to operate on sparse-immutable data-structures, proposing novel algorithms to uncover regular sub-structures within the non-zero coordinates of a sparse structure, however they target the sparse-matrix vector kernel, whereas SPLAT targets the R-SDDMM and R-SpMM kernels. [45] proposes a technique to compile loop nests that have both regular and irregular compute, using static analysis to compile the regular portion combined with an inspector for the irregular portion. [47] is a dynamic analysis technique that proposes a novel *folding-based analysis* which uses run-time information from instrumented binaries to build compact polyhedral program representations; their technique can support programs that are not fully affine. However, these compilation techniques do not support the ACSR, with its compact

representation of sparse-MHSA patterns, that we introduced. General polyhedral frameworks such as tiramisu [7], PLUTO [9] and others [30, 37, 54] that perform optimizations on regular loops are ineffective in performing the same on sparse-MHSA kernels with indirect array accesses.

Structured Sparsity NVIDIA introduced a new specialized data-path in the Ampere [39] architecture to compute *structured-sparsity* [41]. These instructions and special hardware units compute 1:2, 2:4 structured-sparsity where the largest absolute value of every 2 (or two largest values out of every 4) elements is kept, and the rest are pruned. Although these instructions target moderate sparsity levels, they can only represent the strided pattern with a stride of 2 (see figure 7 e for an example), and cannot represent the global, blocked or windowed patterns precisely. Adapting these instructions to represent the global, blocked, or windowed sparse-MHSA patterns will result in storing extraneous non-zeros as padding, similar to CSFs.

Sparse Formats There have been a wide variety of sparse-formats proposed in the literature, please look at the survey in [28] for further details (e.g. CSR, COO, ELLPACK, DCSR, DIA, BCSR, CSB, CSF to name a few). We categorize sparse formats into: general sparse-formats or custom sparse-formats. General sparse-formats (e.g. CSR, COO, CSF, DCSR [23, 46, 49]) incur high metadata in $O(nnzs)$, while custom sparse formats (e.g. BCSR, DIA [21, 22]) are specialized to a particular sparse-MHSA pattern. The ACSR format we introduce has low metadata in $O(rows)$, lower than GSFs, whilst having greater coverage of sparse-MHSA patterns over CSFs.

11 Limitations and Future Work

The ACSR and its supporting code-generation scheme, SPLAT, represent an advancement in code generating high-performance sparse-MHSA kernels. Though SPLAT is able to generate high-performance kernels for a variety of sparse-MHSA patterns, it has some limitations.

Regularity constraint. Input sparse-MHSA patterns need to be *regularly* sparse and therefore row-wise affine-compressible, placing a constraint on the types of sparse patterns the ACSR can represent. However, there are several sparse-MHSA patterns that do not fit this constraint [31, 44].

Static sparsity. SPLAT's code-generation scheme relies on a static sparsity pattern. However, several sparse-MHSA patterns are input dependent and dynamic like in [59].

Applications beyond sparse-MHSA. Leveraging statically structured sparsity to enhance the performance of deep learning algorithms has been explored beyond sparse-MHSA. For example, butterfly matrices [19] have been introduced to sparsify the dense feed-forward networks (post attention) as well as linear transformations (pre attention). Mixers [25] have been introduced to replace dense full-attention with a sub-quadratic variant using short and long convolutions. We leave an investigation into extracting performance from such methods to future work.

Autoregressive Decoding. SPLAT's code-generation scheme is specialised to the R-SDDMM and R-SpMM kernels present in the prefill stage of sparse-MHSA. However, autoregressive decoding operates on individual tokens resulting in sampled dense dense matrix-vector and sparse-matrix-dense-vector multiplications instead. We leave an investigation into extracting performance in these kernels to future work.

12 Conclusion

We have described SPLAT, an optimized code-generation framework that targets a variety of sparse-MHSA patterns. SPLAT exploits the regular nature of sparse-MHSA patterns, introducing a new sparse-format: ACSR, that enables SPLAT's code-generation schemes to have favorable memory-access patterns. We use SPLAT to implement a variety of sparse-MHSA patterns and transformers, demonstrating its generality and high-performance. Our experiments show that SPLAT realizes geomean speedups of 2.05x and 4.05x over hand written kernels written in Triton and TVM respectively in single-precision.

Data Availability Statement

Our artifact is a repository of code written in Python and C++ that implements a CUDA code generation system compatible with JAX. These pieces of code can then be just-in-time compiled through JAX's CUDA compatible foreign-function-interfaces (FFIs).

Acknowledgments

We would like to thank the anonymous reviewers for their constructive feedback that helped improve the paper. We would like to specifically thank Jai Arora and Wanyu Zhao for their feedback on early drafts of this paper. This work was supported in part by ACE, one of the seven centers in JUMP 2.0, a Semiconductor Research Corporation (SRC) program sponsored by DARPA, by NSF under grant CCF-2316233 and by the IBM-Illinois Discovery Accelerator Institute (IIDAI).

References

[1] [n. d.]. cuBLAS — developer.nvidia.com. https://developer.nvidia.com/cublas.

[2] [n. d.]. cuSPARSE — developer.nvidia.com. https://developer.nvidia.com/cusparse.

[3] [n. d.]. SuiteSparse. https://sparse.tamu.edu/.

[4] Jason Ansel, Edward Yang, Horace He, Natalia Gimelshein, Animesh Jain, Michael Voznesensky, Bin Bao, Peter Bell, David Berard, Evgeni Burovski, Geeta Chauhan, Anjali Chourdia, Will Constable, Alban Desmaison, Zachary DeVito, Elias Ellison, Will Feng, Jiong Gong, Michael Gschwind, Brian Hirsh, Sherlock Huang, Kshiteej Kalambarkar, Laurent Kirsch, Michael Lazos, Mario Lezcano, Yanbo Liang, Jason Liang, Yinghai Lu, C. K. Luk, Bert Maher, Yunjie Pan, Christian Puhrsch, Matthias Reso, Mark Saroufim, Marcos Yukio Siraichi, Helen Suk, Shunting Zhang, Michael Suo, Phil Tillet, Xu Zhao, Eikan Wang, Keren Zhou, Richard Zou, Xiaodong Wang, Ajit Mathews, William Wen, Gregory Chanan, Peng Wu, and Soumith Chintala. 2024. PyTorch 2: Faster Machine Learning Through Dynamic Python Bytecode Transformation and Graph Compilation. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2* (La Jolla, CA, USA) *(ASPLOS '24)*. Association for Computing Machinery, New York, NY, USA, 929–947. doi:10.1145/3620665.3640366

[5] Anthropic. 2023. *Introducing 100K Context Windows.* https://www.anthropic.com/index/100k-context-windows

[6] Travis Augustine, Janarthanan Sarma, Louis-Noël Pouchet, and Gabriel Rodríguez. 2019. Generating piecewise-regular code from irregular structures. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Phoenix, AZ, USA) *(PLDI 2019)*. Association for Computing Machinery, New York, NY, USA, 625–639. doi:10.1145/3314221.3314615

[7] Riyadh Baghdadi, Jessica Ray, Malek Ben Romdhane, Emanuele Del Sozzo, Abdurrahman Akkas, Yunming Zhang, Patricia Suriana, Shoaib Kamil, and Saman P. Amarasinghe. 2018. Tiramisu: A Polyhedral Compiler for Expressing Fast and Portable Code. *CoRR* abs/1804.10694 (2018). arXiv:1804.10694 http://arxiv.org/abs/1804.10694

[8] Iz Beltagy, Matthew E. Peters, and Arman Cohan. 2020. Longformer: The Long-Document Transformer. arXiv:2004.05150 [cs.CL]

[9] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. 2008. A Practical Automatic Polyhedral Parallelizer and Locality Optimizer. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Tucson, AZ, USA) *(PLDI '08)*. Association for Computing Machinery, New York, NY, USA, 101–113. doi:10.1145/1375581.1375595

[10] James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, and Qiao Zhang. 2018. *JAX: composable transformations of Python+NumPy programs.* http://github.com/google/jax

[11] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language Models are Few-Shot Learners. *CoRR* abs/2005.14165 (2020). arXiv:2005.14165 https://arxiv.org/abs/2005.14165

[12] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Meghan Cowan, Haichen Shen, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation* (Carlsbad, CA, USA) *(OSDI'18)*. USENIX Association, USA, 579–594.

[13] Kazem Cheshmi, Shoaib Kamil, Michelle Mills Strout, and Maryam Mehri Dehnavi. 2017. Sympiler: transforming sparse matrix codes by decoupling symbolic analysis. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 1–13. doi:10.1145/3126908.3126936

[14] Kazem Cheshmi, Shoaib Kamil, Michelle Mills Strout, and Maryam Mehri Dehnavi. 2018. ParSy: Inspection and Transformation of Sparse Matrix Computations for Parallelism. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*. 779–793. doi:10.1109/SC.2018.00065

[15] Kazem Cheshmi, Michelle Strout, and Maryam Mehri Dehnavi. 2023. Runtime Composition of Iterations for Fusing Loop-carried Sparse Dependence. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Denver, CO, USA) *(SC '23)*. Association for Computing Machinery, New York, NY, USA, Article 89, 15 pages. doi:10.1145/3581784.3607097

[16] Rewon Child, Scott Gray, Alec Radford, and Ilya Sutskever. 2019. Generating Long Sequences with Sparse Transformers. *CoRR* abs/1904.10509 (2019). arXiv:1904.10509 http://arxiv.org/abs/1904.10509

[17] Stephen Chou, Fredrik Kjolstad, and Saman P. Amarasinghe. 2018. Unified Sparse Formats for Tensor Algebra Compilers. *CoRR* abs/1804.10112 (2018). arXiv:1804.10112 http://arxiv.org/abs/1804.10112

[18] Guohao Dai, Guyue Huang, Shang Yang, Zhongming Yu, Hengrui Zhang, Yufei Ding, Yuan Xie, Huazhong Yang, and Yu Wang. 2022. Heuristic Adaptability to Input Dynamics for SpMM on GPUs. arXiv:2202.08556 [cs.AR] https://arxiv.org/abs/2202.08556

[19] Tri Dao, Beidi Chen, Kaizhao Liang, Jiaming Yang, Zhao Song, Atri Rudra, and Christopher Ré. 2022. Pixelated Butterfly: Simple and Efficient Sparse training for Neural Network Models. arXiv:2112.00029 [cs.LG] https://arxiv.org/abs/2112.00029

[20] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. *CoRR* abs/1810.04805 (2018). arXiv:1810.04805 http://arxiv.org/abs/1810.04805

[21] Jack Dongarra. 1995. Block Compressed Row Storage (BCRS) — netlib.org. https://netlib.org/linalg/html_templates/node93.html. [Accessed 15-10-2024].

[22] Jack Dongarra. 1995. Compressed Diagonal Storage (CDS) — netlib.org. https://netlib.org/linalg/html_templates/node94.html. [Accessed 15-10-2024].

[23] Jack Dongarra. 1995. Compressed Row Storage (CRS) — netlib.org. https://netlib.org/linalg/html_templates/node91.html. [Accessed 15-10-2024].

[24] Clement Farabet and Tris Warkentin. [n. d.]. Gemma 2 is now available to researchers and developers — blog.google. https://blog.google/technology/developers/google-gemma-2/. [Accessed 15-07-2024].

[25] Daniel Y. Fu, Simran Arora, Jessica Grogan, Isys Johnson, Sabri Eyuboglu, Armin W. Thomas, Benjamin Spector, Michael Poli, Atri Rudra, and Christopher Ré. 2023. Monarch Mixer: A Simple Sub-Quadratic GEMM-Based Architecture. arXiv:2310.12109 [cs.LG] https://arxiv.org/abs/2310.12109

[26] Trevor Gale, Matei Zaharia, Cliff Young, and Erich Elsen. 2020. Sparse GPU Kernels for Deep Learning. arXiv:2006.10901 [cs.LG]

[27] 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* (Atlanta, Georgia) *(SC '20)*. IEEE Press, Article 17, 14 pages.

[28] Jianhua Gao, Weixing Ji, Fangli Chang, Shiyu Han, Bingxin Wei, Zeming Liu, and Yizhuo Wang. 2023. A Systematic Survey of General Sparse Matrix-Matrix Multiplication. *ACM Comput. Surv.* 55, 12, Article 244 (mar 2023), 36 pages. doi:10.1145/3571157

[29] Scott Gray, Alec Radford, and Durk Kingma. 2017. Block-Sparse GPU Kernels. https://openai.com/research/block-sparse-gpu-kernels. https://openai.com/research/block-sparse-gpu-kernels

[30] Tobias Grosser, Armin Größlinger, and Christian Lengauer. 2012. Polly - Performing Polyhedral Optimizations on a Low-Level Intermediate Representation. *Parallel Process. Lett.* 22 (2012). https://api.semanticscholar.org/CorpusID:18533155

[31] Chi Han, Qifan Wang, Hao Peng, Wenhan Xiong, Yu Chen, Heng Ji, and Sinong Wang. 2024. LM-Infinite: Zero-Shot Extreme Length Generalization for Large Language Models. arXiv:2308.16137 [cs.CL] https://arxiv.org/abs/2308.16137

[32] Changwan Hong, Aravind Sukumaran-Rajam, Israt Nisa, Kunal Singh, and P. Sadayappan. 2019. Adaptive sparse tiling for sparse matrix multiplication. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming* (Washington, District of Columbia) *(PPoPP '19)*. Association for Computing Machinery, New York, NY, USA, 300–314. doi:10.1145/3293883.3295712

[33] Victor Eijkhout Jack Dongarra and Henk van der Vorst. [n. d.]. SparseBench. https://www.netlib.org/benchmark/sparsebench/.

[34] Albert Q. Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Florian Bressand, Gianna Lengyel, Guillaume Lample, Lucile Saulnier, Lélio Renard Lavaud, Marie-Anne Lachaux, Pierre Stock, Teven Le Scao, Thibaut Lavril, Thomas Wang, Timothée Lacroix, and William El Sayed. 2023. Mistral 7B.

arXiv:2310.06825 [cs.CL] https://arxiv.org/abs/2310.06825

[35] Nikita Kitaev, Lukasz Kaiser, and Anselm Levskaya. 2020. Reformer: The Efficient Transformer. *CoRR* abs/2001.04451 (2020). arXiv:2001.04451 https://arxiv.org/abs/2001.04451

[36] Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, David Lugato, and Saman Amarasinghe. 2017. The Tensor Algebra Compiler. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 77 (Oct. 2017), 29 pages. doi:10.1145/3133901

[37] Ravi Teja Mullapudi, Vinay Vasista, and Uday Bondhugula. 2015. PolyMage: Automatic Optimization for Image Processing Pipelines. *SIGPLAN Not.* 50, 4 (mar 2015), 429–443. doi:10.1145/2775054.2694364

[38] Israt Nisa, Aravind Sukumaran-Rajam, Sureyya Emre Kurt, Changwan Hong, and P. Sadayappan. 2018. Sampled Dense Matrix Multiplication for High-Performance Machine Learning. In *2018 IEEE 25th International Conference on High Performance Computing (HiPC)*. 32–41. doi:10.1109/HiPC.2018.00013

[39] NVIDIA. 2020. *Ampere Architecture.* https://www.nvidia.com/en-us/data-center/ampere-architecture/

[40] OpenAI. 2023. GPT-4 Technical Report. arXiv:2303.08774 [cs.CL]

[41] Jeff Pool, Abhishek Sawarkar, and Jay Rodge. 2023. Accelerating inference with sparsity using the Nvidia ampere architecture and NVIDIA TENSORRT. https://developer.nvidia.com/blog/accelerating-inference-with-sparsity-using-ampere-and-tensorrt/

[42] Jiezhong Qiu, Hao Ma, Omer Levy, Scott Wen tau Yih, Sinong Wang, and Jie Tang. 2020. Blockwise Self-Attention for Long Document Understanding. arXiv:1911.02972 [cs.CL] https://arxiv.org/abs/1911.02972

[43] Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. 2019. Language Models are Unsupervised Multitask Learners. https://api.semanticscholar.org/CorpusID:160025533

[44] Aditya Ramesh, Mikhail Pavlov, Gabriel Goh, Scott Gray, Chelsea Voss, Alec Radford, Mark Chen, and Ilya Sutskever. 2021. Zero-Shot Text-to-Image Generation. arXiv:2102.12092 [cs.CV] https://arxiv.org/abs/2102.12092

[45] Mahesh Ravishankar, Roshan Dathathri, Venmugil Elango, Louis-Noël Pouchet, J. Ramanujam, Atanas Rountev, and P. Sadayappan. 2015. Distributed memory code generation for mixed Irregular/Regular computations. *SIGPLAN Not.* 50, 8 (Jan. 2015), 65–75. doi:10.1145/2858788.2688515

[46] Yousef Saad. 2003. *Iterative methods for sparse linear systems.* SIAM. I–XVIII, 1–528 pages.

[47] Manuel Selva, Fabian Gruber, Diogo Sampaio, Christophe Guillon, Louis-Noël Pouchet, and Fabrice Rastello. 2019. Building a Polyhedral Representation from an Instrumented Execution: Making Dynamic Analyses of Nonaffine Programs Scalable. *ACM Trans. Archit. Code Optim.* 16, 4, Article 45 (Dec. 2019), 26 pages. doi:10.1145/3363785

[48] Uri Shaham, Elad Segal, Maor Ivgi, Avia Efrat, Ori Yoran, Adi Haviv, Ankit Gupta, Wenhan Xiong, Mor Geva, Jonathan Berant, and Omer Levy. 2022. SCROLLS: Standardized CompaRison Over Long Language Sequences. In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, Abu Dhabi, United Arab Emirates, 12007–12021. https://aclanthology.org/2022.emnlp-main.823

[49] Shaden Smith and George Karypis. 2015. Tensor-matrix products with a compressed sparse tensor. In *Proceedings of the 5th Workshop on Irregular Applications: Architectures and Algorithms* (Austin, Texas) *(IA³ '15)*. Association for Computing Machinery, New York, NY, USA, Article 5, 7 pages. doi:10.1145/2833179.2833183

[50] Aravind Sukumaran-Rajam and Philippe Clauss. 2015. The Polyhedral Model of Nonlinear Loops. *ACM Trans. Archit. Code Optim.* 12, 4, Article 48 (Dec. 2015), 27 pages. doi:10.1145/2838734

[51] Yi Tay, Mostafa Dehghani, Samira Abnar, Yikang Shen, Dara Bahri, Philip Pham, Jinfeng Rao, Liu Yang, Sebastian Ruder, and Donald Metzler. 2020. Long Range Arena: A Benchmark for Efficient Transformers. arXiv:2011.04006 [cs.LG] https://arxiv.org/abs/2011.04006

[52] Yi Tay, Mostafa Dehghani, Dara Bahri, and Donald Metzler. 2020. Efficient Transformers: A Survey. *CoRR* abs/2009.06732 (2020). arXiv:2009.06732 https://arxiv.org/abs/2009.06732

[53] Philippe Tillet, H. T. Kung, and David Cox. 2019. Triton: An Intermediate Language and Compiler for Tiled Neural Network Computations. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages* (Phoenix, AZ, USA) *(MAPL 2019)*. Association for Computing Machinery, New York, NY, USA, 10–19. doi:10.1145/3315508.3329973

[54] Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary DeVito, William S. Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. 2018. Tensor Comprehensions: Framework-Agnostic High-Performance Machine Learning Abstractions. *CoRR* abs/1802.04730 (2018). arXiv:1802.04730 http://arxiv.org/abs/1802.04730

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

[56] Lucas Wilkinson, Kazem Cheshmi, and Maryam Mehri Dehnavi. 2023. Register Tiling for Unstructured Sparsity in Neural Network Inference. *Proc. ACM Program. Lang.* 7, PLDI, Article 188 (jun 2023), 26 pages. doi:10.1145/3591302

[57] Zihao Ye, Ruihang Lai, Junru Shao, Tianqi Chen, and Luis Ceze. 2023. SparseTIR: Composable Abstractions for Sparse Compilation in Deep Learning. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3* (Vancouver, BC, Canada) *(ASPLOS 2023)*. Association for Computing Machinery, New York, NY, USA, 660–678. doi:10.1145/3582016.3582047

[58] Gokul Yenduri, Ramalingam M, Chemmalar Selvi G, Supriya Y, Gautam Srivastava, Praveen Kumar Reddy Maddikunta, Deepti Raj G, Rutvij H Jhaveri, Prabadevi B, Weizheng Wang, Athanasios V. Vasilakos, and Thippa Reddy Gadekallu. 2023. Generative Pre-trained Transformer: A Comprehensive Review on Enabling Technologies, Potential Applications, Emerging Challenges, and Future Directions. arXiv:2305.10435 [cs.CL] https://arxiv.org/abs/2305.10435

[59] Manzil Zaheer, Guru Guruganesh, Avinava Dubey, Joshua Ainslie, Chris Alberti, Santiago Ontañón, Philip Pham, Anirudh Ravula, Qifan Wang, Li Yang, and Amr Ahmed. 2020. Big Bird: Transformers for Longer Sequences. *CoRR* abs/2007.14062 (2020). arXiv:2007.14062 https://arxiv.org/abs/2007.14062

[60] Ningxin Zheng, Bin Lin, Quanlu Zhang, Lingxiao Ma, Yuqing Yang, Fan Yang, Yang Wang, Mao Yang, and Lidong Zhou. 2022. SparTA: Deep-Learning Model Sparsity via Tensor-with-Sparsity-Attribute. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. USENIX Association, Carlsbad, CA, 213–232. https://www.usenix.org/conference/osdi22/presentation/zheng-ningxin

[61] Chen Zhu, Wei Ping, Chaowei Xiao, Mohammad Shoeybi, Tom Goldstein, Anima Anandkumar, and Bryan Catanzaro. 2021. Long-Short Transformer: Efficient Transformers for Language and Vision. *CoRR* abs/2107.02192 (2021). arXiv:2107.02192 https://arxiv.org/abs/2107.02192

Received 2024-10-15; accepted 2025-02-18