Splyce: SIMD Vectorization of Sparse Coiteration

Kabilan Mahathevan
Virginia Tech
Blacksburg, USA
kabilan@vt.edu

Poorna Gunathilaka
Virginia Tech
Blacksburg, USA
poornag@vt.edu

Kirshanthan Sundararajah
Virginia Tech
Blacksburg, USA
kirshanthans@vt.edu

Abstract

Sparse tensor contractions are bottlenecked by sparse-sparse coiteration loops that resist standard loop vectorization. We present Splyce, an auto-vectorization framework in MLIR that overcomes this through a dual-path execution model. By decoupling coordinate intersection from pointer management via selective predication, Splyce inherently eliminates data-dependent branches as a side effect, allowing modern superscalar engines to maximize instruction-level parallelism and hide memory latency. Beyond simple branch elimination, our transformation exposes independent computation that can be executed concurrently, increasing functional-unit utilization that would otherwise be constrained by sequential dependencies. Evaluation across foundational sparse tensor kernels demonstrates performance ranging from 1.96× to 2.86× on synthetic inputs, with consistent speedups sustained across a vast majority of irregular real-world datasets from the SuiteSparse collection. Ultimately, Splyce demonstrates that by converting unpredictable control-flow into a predictable data stream, compiler-driven speculation can effectively reconcile the memory efficiency of compressed storage with the execution-unit throughput of modern superscalar architectures.

CCS Concepts

• **Software and its engineering → Compilers**.

Keywords

Sparse Tensor Compilation, Vectorization, Coiteration

ACM Reference Format:
Kabilan Mahathevan, Poorna Gunathilaka, and Kirshanthan Sundararajah. 2026. Splyce: SIMD Vectorization of Sparse Coiteration. In *International Conference on Parallel Architectures and Compilation Techniques (PACT '26), October 19–22, 2026, Chicago, IL, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3838684.3846860

1 Introduction

Dense tensor optimizations are well-established [9, 17, 18, 31, 35, 46, 46], but the proliferation of sparse data structures in modern workloads has shifted focus toward formats that compress data to save memory and reduce compute bounds [10, 12, 25, 34, 38, 43]. However, the complex memory layouts required for sparse compression complicate iteration and automated code generation, especially when compared to the straightforward, random-access nature of dense arrays. To address this, tensor compilers increasingly rely on the Compressed Sparse Fiber (CSF) format for its generalizability [38]. By modeling a tensor as a hierarchical tree, CSF allows each dimension to be annotated independently as dense, compressed, or singleton. Compressed dimensions explicitly track non-zero elements using position and coordinate arrays, while singleton levels represent unique coordinate entries without the overhead of pointers. Dense dimensions rely on implicit, fixed-size indexing, culminating in a unified contiguous array of non-zero values at the leaf level.

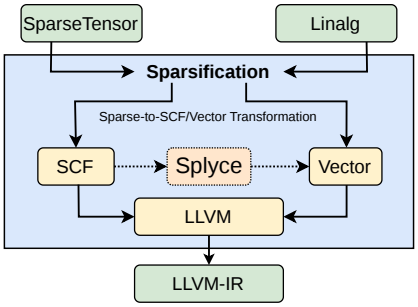


Figure 1: `SparseTensor` dialect lowering with Splyce.

To navigate the complexities of generating code for diverse sparse data layouts, advanced tensor compilers have been developed. Most notably, building upon the pioneering success of TACO [25], the MLIR framework integrated a dedicated sparse tensor dialect [6] that adapts TACO's principles for a modern, multi-level compilation ecosystem. Consequently, active research is heavily focused on designing and implementing novel optimization strategies across the various abstraction layers within this infrastructure.

The `SparseTensor` dialect operates as a compilation infrastructure that heavily leverages the progressive lowering ecosystem of MLIR, seamlessly integrating with the `Linalg`, `SCF`, and `Vector` dialects. Within this framework, core computations are abstractly defined using `linalg.generic` operations coupled with per-tensor sparse encoding annotations. As illustrated in Figure 1, the standard code generation pipeline progressively lowers these abstractions to achieve high performance, actively targeting the `Vector` dialect to exploit SIMD hardware whenever the underlying data layout permits. However, a major caveat of sparse tensor compilation is that the execution efficiency is inherently bound to the distribution of non-zeros in input data—a dynamic characteristic that the static compile-time optimizations largely fail to exploit [32].

A critical example of this limitation manifests in the *two-finger* `scf.while` loop structures (*i.e.,* sparse coiteration loop) generated during the lowering process. This coiteration loop is a fundamental requirement of sparse tensor computations when a shared or contracting dimension is compressed on both inputs. To traverse this contracting dimension, the generated code must simultaneously advance two cursors through the coordinate arrays of the

This work is licensed under a Creative Commons Attribution 4.0 International License.
PACT '26, Chicago, IL, USA
© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2915-7/2026/10
https://doi.org/10.1145/3838684.3846860

respective tensors that is similar to performing set intersection operation [8, 19, 30, 39, 49]. Crucially, for intersection operations, the inner computation is strictly predicated on a coordinate match; it executes exclusively when both cursors point to the same index. Because these pointers must advance dynamically based on runtime coordinate comparisons, this data-dependent control-flow inherently breaks standard loop optimizations, most notably vectorization [12, 25].

The introduction of divergent `if-else` control-flow within the `scf.while` loop and the data-dependent loop bounds structurally prevents standard SIMD auto vectorization. While compilers can circumvent this by evaluating the coiteration space as a dense kernel to eliminate branching, the resulting computational overhead negates the memory and performance benefits of the sparse representation overwhelmingly.

In this paper, we demonstrate that the throughput advantages of vectorized execution can actually outweigh the penalty of processing a calculated fraction of structural zeros. To achieve this, we introduce **Splyce**, a novel MLIR optimization pass that intercepts the sparsification pipeline between the `SCF` and `Vector` dialects (Figure 1). Splyce transforms the standard scalar sparse coiteration into a dual-path execution consisting of a vectorized fast-path and a scalar slow-path (Figure 3).

The fast-path iterates at the hardware vector width, utilizing predicated execution and vector masks to safely perform speculative SIMD operations. This transformation inherently eliminates data-dependent branches, allowing modern superscalar engines to maximize instruction-level parallelism (ILP) by overlapping coordinate matching with core tensor arithmetic. Specifically, the contributions of this work are as follows:

- **Splyce Vectorization:** We implement the Splyce pass in MLIR to natively vectorize sparse coiterations via a branchless fast-path, enabling SIMD execution on compressed formats.
- **Analysis of Execution Phases:** We analyze sparse coiteration phases to identify the optimal balance between vector and scalar execution. We establish that a hybrid approach prevents execution port monopolization.
- **Performance Trade-off Analysis:** We quantify the cost-benefit of speculative vectorization, achieving a 2.38× geometric mean speedup across five kernels on synthetic datasets with 5% sparsity.
- **Sparsity and Parallel Scaling:** We demonstrate that Splyce maintains robust performance across tensor sparsities from 0.01% to 10% and real-world distributions, while achieving near-linear scalability across parallel multi-core environments.

To improve readability throughout the paper, we present code snippets in a simplified, hybrid MLIR/C-like form. Additionally, for brevity, we use the term *coiteration* to refer specifically to *sparse coiteration* (formally detailed in Section 2.3) throughout the remainder of this paper. The remainder of this paper is structured as follows. Section 2 provides essential background and Section 3 details the design of the Splyce approach. We illustrate the evaluation of our approach in Section 4. Section 5 reviews related work, and we discuss future work in Section 6. Finally, we conclude the paper in Section 7.

2 Background

2.1 Sparse Tensor Compilers (STCs)

The proliferation of highly sparse tensors in modern applications, ranging from graph analytics to deep learning and quantum chemistry, has necessitated a fundamental shift in computational strategies for tensor contractions [22, 26]. Historically, the challenge of sparsity has been addressed by relying on manually crafted and tuned kernel libraries [1, 2, 15, 37]. While these libraries deliver near-peak hardware utilization for basic linear algebra routines like SpGEMM and SpMV, they suffer from a severe scalability crisis. Developing, optimizing, and maintaining hand-written kernels for every conceivable tensor expression across every permutation of compressed data formats, and targeting increasingly diverse hardware architectures, is fundamentally intractable [25].

Consequently, modern frameworks have shifted toward compiler-driven generation [5, 6, 25, 48], successfully enabling the generalized traversal of compressed sparse tensors without materializing them into memory-intensive dense formats. The advent of the Tensor Algebra Compiler (TACO) [25] marked a paradigm shift by introducing the iteration lattice—a mathematical framework that decouples the tensor expression from its underlying layout to automatically synthesize coiteration control-flow.

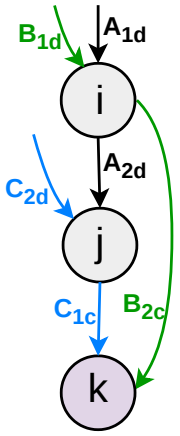
To achieve this generality, modern STCs rely on Compressed Sparse Fiber (CSF) abstractions. CSF models any sparse tensor as a hierarchical tree where each axis is defined independently, where each axis (or dimension) of a tensor is defined as either dense or compressed. By composing these level-wise annotations, an STC can natively represent and compile code for a vast spectrum of traditional data structures. For instance, a 2D matrix encoded with an outer-dense axis and an inner-compressed axis yields the standard Compressed Sparse Row (CSR) format. Reversing this to compressed-dense yields the Compressed Sparse Column (CSC) format, while a compressed-compressed annotation generates the Doubly Compressed Sparse Row (DCSR) format [12].

2.2 MLIR `SparseTensor` Dialect

Building upon these generalized STC frameworks, the Multi-Level Intermediate Representation (MLIR) infrastructure has emerged as the modern standard for compiler-driven tensor optimization [29]. Specifically, the MLIR `SparseTensor` dialect operationalizes the CSF abstraction, lowering these theoretical format annotations directly into executable iteration strategies.

Within this dialect, the physical memory access pattern for a given axis is dictated entirely by its sparsity annotation. If an axis is dense, its structural layout is implicitly defined by its fixed dimension size. To traverse it, MLIR generates a standard sequential loop iterating from zero to the dimension's upper bound. Because all elements along a dense axis are assumed to exist within the coordinate space, the physical location of any element in the underlying storage can be computed mathematically. This implicit structure allows MLIR to perform random-access lookups in constant time.

Conversely, if an axis is compressed, it explicitly stores only the non-zero elements using auxiliary data structures, typically a position (or pointer) array and a coordinate (or index) array. Iteration along a compressed axis inherently requires indirect memory access. To traverse the non-zeros belonging to a specific parent node




```
for (int i = 0; i < dim_i; ++i){
  for (int j = 0; j < dim_j; ++j){
    // Load dim-k pointer
    double acc = 0.0;
    while (pB < ... && pC < ...){
        int kB = B_col[pB];
        int kC = C_row[pC];
        if (kB == kC) {
          acc += B_val[pB] *
                C_val[pC];
          pB++; pC++;
        } else
        (kB < kC) ? pB++ : pC++;
    }
    A[i][j] = acc;
}}
```

```
scf.for %i = %c0 to %dim_i step %c1
  scf.for %j = %c0 to %dim_j step %c1
  %res:3 = scf.while(%pB = ..., %pC = ..., %acc = ...){
      scf.condition(/* %pB < ... && %pC < ... */) %pB, %pC, %acc
    } do {
    ^bb0(%pB: index , %pC: index , %acc: f64):
      %kB, %kC = // memref.load indices for %pB and %pC
      %next_acc = scf.if (arith.cmpi eq, %kB, %kC) -> (f64) {
          // Predicated arithmetic: A(i,j) += B(i,k) * C(k,j)
          scf.yield %new_sum : f64
      } else { scf.yield %acc : f64 }
      // Pointer advancing: Increment %pB or %pC based on kB < kC
      scf.yield %next_pB, %next_pC, %next_acc : index, index, f64
    }
```

(a) Iteration graph. **(b) Two-finger loop in C.** **(c) MLIR coiteration lowering resulting in two-finger `scf.while` loop.**

Figure 2: For SpGEMM ($A_{ij} = \sum_k B_{ik} \cdot C_{kj}$) kernel, tensor dimensions are subscripted to indicate dense(d) vs. compressed(c).

in the CSF tree, the MLIR generates a loop that reads a bounding segment from the position array (*e.g.,* from $pos[p]$ to $pos[p+1]$) and iterates strictly within those bounds. During this traversal, the physical logical coordinate of each non-zero must be explicitly fetched from the coordinate array. Because the logical tensor coordinate is decoupled from its physical storage location, direct $O(1)$ random access is impossible. Instead, the MLIR must sequentially traverse, coiterate, or binary-search these auxiliary arrays to locate specific elements and perform computations.

2.3 Sparse Coiteration

The true complexity of sparse algebra arises during intersection operations, such as contracting ***k***-index in SpGeMM. As illustrated in Figure 2a, matrix ***B*** and ***C*** both access the ***k*** dimension. Crucially, $\boldsymbol{B}_{2c}$ and $\boldsymbol{C}_{1c}$ edge labels denote that this dimension is explicitly compressed in both input tensors. Because random access is impossible on compressed axes, the MLIR cannot emit standard, independent `for` loops. Instead it must synthesize a dynamic *coiteration* procedure—commonly referred to as a *two-finger* loop.

To traverse this compressed space, MLIR maintains two independent cursors that track the current non-zeros within their respective coordinate arrays. This dynamic *two-finger* coiteration is not an MLIR-specific construct, but rather a fundamental programming pattern universally required for sparse intersection, as mirrored in the equivalent handwritten `C` routine (Figure 2b). During each iteration of the compiler-generated `scf.while` loop (Figure 2c), the core tensor arithmetic is strictly predicated on a coordinate match, guarded by an `scf.if` condition. Following this predicated computation, the cursors must advance dynamically based on the relative values of their current coordinates. Rather than emitting complex, nested conditional branches to handle this pointer advancement logic, MLIR optimizes the step using data-dependent `select` operations. This ensures the cursors increment correctly while minimizing control-flow divergence within the critical path.

Despite these control-flow optimizations, the inherent memory dependencies and the conditional nature of the coiteration loop present a massive barrier to high performance. Because loop bounds and execution paths are dictated by runtime values that vary dynamically per cycle, classical static compiler transformations—such as loop tiling, array expansion, and, most critically, standard auto-vectorization—are severely restricted or rendered entirely inapplicable [33, 44].

2.4 Microarchitectural Bottlenecks

The performance of coiteration is fundamentally constrained by a three-fold conflict with modern microarchitecture. First, the irregular memory layout of compressed storage shatters the data-level regularity required for effective SIMD vectorization. Because matching coordinates must be discovered via runtime scalar comparisons, traditional compilers fail to safely populate wide vector registers, leaving high-throughput ALU resources underutilized [4, 13, 28].

Second, the data-dependent nature of sparse intersections creates a branch prediction wall. In deep Out-of-Order (OoO) engines, the irregular sparsity patterns result in frequent branch mispredictions, triggering pipeline flushes that incur penalties of 15-20 cycles per occurrence [16, 21, 36]. These penalties often dwarf the latency of the actual floating-point computation, rendering the arithmetic units secondary to the cost of control-flow recovery.

Finally, the Instruction-Level Parallelism (ILP) is restricted by the *two-finger* coiteration loop through inescapable data dependency chains. In this scalar model, pointer advancement is strictly coupled to intersection results, and arithmetic is predicated on pointer states (Refer Figure 2b). This sequentiality serializes execution and starves parallel hardware ports, crippling overall throughput [21, 40, 41].

Optimizing these kernels requires a fundamental decoupling of memory retrieval from intersection evaluation. To break these scalar dependency chains and reclaim unutilized SIMD capacity, we must move beyond incremental loop transformations toward a hardware-aware, branchless data-flow model. The following section introduces Splyce, an MLIR optimization pass that addresses these bottlenecks by prepending a vectorized, branchless *fast-lane* to the standard coiteration pipeline.

3 Design

The primary objective of Splyce is to safely vectorize the data-dependent `scf.while` coiteration loops generated during sparse tensor lowering. However, this vectorization opportunity is not universally applicable across all sparse tensor contractions. Generating a vectorizable two-finger coiteration loop necessitates that several strict structural preconditions are met.

3.1 Structural Preconditions

To successfully identify and optimize the dynamic two-finger coiteration bottleneck, our proposed MLIR pass explicitly targets loop nests that satisfy two strict structural preconditions:

- **Compression of the Contracting Index:** The shared index variable must be explicitly annotated as *compressed* across all interacting input tensors. A representative example is the index k in the SpGEMM kernel (Figure 2), which is compressed in both $\boldsymbol{B}[i,k]$ and $\boldsymbol{C}[k,j]$. If this shared dimension is uncompressed (dense) in even a single operand, the compiler avoids coiteration entirely. Instead, it generates a standard `scf.for` loop to iterate exclusively over the sparse non-zeros while performing direct, $O(1)$ random-access lookups into the dense tensor.
- **Innermost Loop Structure:** The generated `scf.while` coiteration must form the innermost structure of the loop nest. Topologically, this dictates that the shared compressed dimension must be the innermost (fastest-changing) index for all interacting tensors, a configuration clearly depicted in Figure 2. If the `scf.while` encloses an inner `scf.for` loop (*e.g.,* iterating over a dense innermost dimension), custom vectorization of the coiteration is unnecessary, as the standard compiler pipeline can readily auto-vectorize the innermost `for` loop.

3.2 The Dual-Path Vectorization Strategy

To address the microarchitectural bottlenecks inherent in dynamic scalar coiteration, we propose a novel MLIR optimization pass that fundamentally restructures the standard *two-finger* loop pattern. Specifically, the pass detects instances where `scf.while` loop forms the innermost structure and relies on divergent `if-else` control flow to evaluate coordinate intersections (as previously detailed in Section 3.1 and Figure 2). As illustrated on the left side of Figure 3, this baseline scalar loop is governed by a primary bounds check (`condition_a`). For a SpGEMM kernel with cursors p_B and p_C and segment bounds end_B and end_C, this global check is defined as $(p_B < end_B \wedge p_C < end_C)$, ensuring that both sparse iterators have remaining elements in their respective segments. Within the loop, actual computation is predicated on an internal coordinate match (`condition_b`). Figure 2c provides the complete MLIR lowering for the SpGEMM kernel, explicitly highlighting this critical *two-finger* loop.

Upon identifying this pattern, Splyce transforms this control flow into a dual-path execution model (Figure 3, right). First, it prepends a vectorized *fast lane*. Utilizing the MLIR `Vector` dialect, this new primary loop executes fixed-width SIMD operations. Based on the target hardware's vector width (n), the compiler loads n contiguous non-zero values from each input tensor into vector registers, using

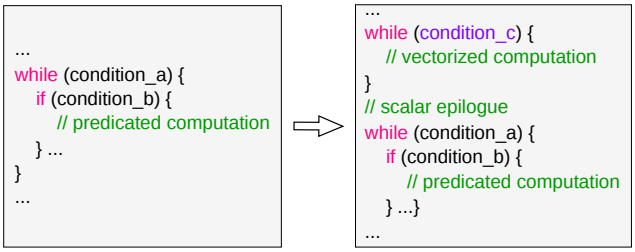


Figure 3: Transforming scalar code into Splyce dual-path.

vector masks to pad mismatched coordinates with explicit zero values. Crucially, this fast lane is guarded by a stricter, data-dependent runtime condition (`condition_c`). The augmented vectorization constraint, `condition_c`, ensures that a full hardware vector width of elements remains available for SIMD loading—specifically requiring that $(p_B + n \leq end_B \wedge p_C + n \leq end_C)$. This check guarantees that the fast lane only executes as long as both tensors possess at least n remaining non-zero elements along the active axis.

The original scalar `scf.while` loop is not entirely eliminated by Splyce; it is retained to serve as a slow lane scalar epilogue. Once the iterators advance near the end of the segment, and the remaining elements fall below the hardware vector width (n), `condition_c` evaluates to false. Execution then seamlessly falls through to the unmodified scalar loop, which processes the remaining tail elements using the standard coordinate-matching procedure. This guarantees correct algorithmic output without introducing vectorization overhead on small residual segments.

3.3 Execution Phases

Mechanically, the execution of the vectorized fast lane is divided into three distinct phases: loading the data, performing the arithmetic computation, and advancing the coordinate pointers. To precisely illustrate these phases, the following section continues to use SpGEMM as a running example, utilizing its mathematical formulation and MLIR sparse encoding map (the attribute that explicitly binds logical tensor axes to physical storage levels, such as dense or compressed) to construct and analyze the optimized `scf.while` innermost loop structure.

Consider the standard SpGEMM kernel computing $\boldsymbol{A}[i,j] = \boldsymbol{B}[i,k] \cdot \boldsymbol{C}[k,j]$. To force the compiler to generate the target coiteration, we define specific MLIR sparse encodings for the input tensors:

- **Matrix B** ($\mathbf{B}[i,k]$): Encoded in the standard Compressed Sparse Row (CSR) format. The outer dimension i is annotated as *dense*, while the inner contracting dimension k is annotated as *compressed*.
- **Matrix C** ($\mathbf{C}[k,j]$): Encoded in the Compressed Sparse Column (CSC) format, mapping j as the outer dense dimension and k as the inner compressed (fastest-changing) dimension. This column-wise access ensures the shared contracting dimension k is physically compressed in both operands, facilitating coiteration.

By configuring the compiler to schedule the loop nest in an (**i**, **j**, **k**) order, the workload decomposes into computing the intersection between the i-th row of **B** and the j-th column of **C**. Because the

```
// Original Scalar Data Load
...
%b0_coord = memref.load %B_col_index[%arg1]
%c0_coord = memref.load %C_row_index[%arg2]
%cond = // Calculates if %b0_coord == %c0_coord
%contr = scf.if %cond -> (f64) {
    %b0_val = memref.load %B_values[%arg1]
    %c0_val = memref.load %C_values[%arg2]
    // Tensor contraction
}
...
```

⇓ Lowering to Vector Data Loads

```
// After: 4-wide Vector Loads
...
%b_4_coords = vector.load %B_col_index[%arg1]
%c_4_coords = vector.load %C_row_index[%arg2]
%b_4_values = vector.load %B_values[%arg1]
%c_4_values = vector.load %C_values[%arg2]
...
```

Figure 4: Scalar memory loads from sparse tensors lowered to fixed-width `vector.load` operations.

	$k_j^{(0)}$	$k_j^{(1)}$	$k_j^{(2)}$	$k_j^{(3)}$
$k_i^{(0)}$	$k_i^{(0)} = k_j^{(0)}$	$k_i^{(0)} = k_j^{(1)}$	$k_i^{(0)} = k_j^{(2)}$	$k_i^{(0)} = k_j^{(3)}$
$k_i^{(1)}$	$k_i^{(1)} = k_j^{(0)}$	$k_i^{(1)} = k_j^{(1)}$	$k_i^{(1)} = k_j^{(2)}$	$k_i^{(1)} = k_j^{(3)}$
$k_i^{(2)}$	$k_i^{(2)} = k_j^{(0)}$	$k_i^{(2)} = k_j^{(1)}$	$k_i^{(2)} = k_j^{(2)}$	$k_i^{(2)} = k_j^{(3)}$
$k_i^{(3)}$	$k_i^{(3)} = k_j^{(0)}$	$k_i^{(3)} = k_j^{(1)}$	$k_i^{(3)} = k_j^{(2)}$	$k_i^{(3)} = k_j^{(3)}$

Figure 5: Naive cartesian coiteration for SpGEMM: All-pairs of the nonzero indices in row B_i and column C_j are compared to identify matching k indices contributing to A_{ij}.

shared contracting index k resides at the innermost compressed level for both input tensors, $O(1)$ random access is impossible. Consequently, the MLIR sparsifier must emit our exact target code pattern: an innermost `scf.while` loop that advances two independent pointers to coiterate through the explicitly stored k-coordinate arrays. As explicitly illustrated in Figure 2, this dynamically bounded, branch-heavy scalar loop serves as the precise microarchitectural bottleneck that our optimization pass fundamentally restructures.

Phase 1: Contiguous Loading and Mask Generation. Because the fast lane is strictly guarded by the data-dependent `condition_c` (ensuring at least n non-zeros remain in both tensors), the compiler can safely bypass complex gather operations for the initial coordinate retrieval. Instead, the pass emits standard `vector.load` operations to fetch n contiguous coordinates from the explicitly stored k-index arrays of both **B** and **C**.

Figure 4 illustrates this load transformation. In the baseline scalar execution, the loop fetches individual coordinates from **B** and **C**, and the actual numerical values are loaded strictly after a conditional `if-else` branch confirms a coordinate match. In contrast, our vectorized fast lane unconditionally fetches n coordinates and n corresponding numerical values simultaneously from both matrices. By decoupling the memory loads from the intersection, this transformation entirely eliminates control-flow divergence within the program's hot-path.

Once these coordinate vectors populate the SIMD registers, the pass executes an all-pairs cross-comparison strategy, yielding an $n \times n$ boolean matrix representing all potential intersections (illustrated in Figure 5). Because the coordinate arrays in CSF formats are strictly increasing, this comparison guarantees at most one `true` value per row. This inherent property ensures that there are no duplicate coordinates within either loaded vector; therefore, a specific k-index from one tensor can match, at most, a single identical k-index in the other.

In the context of our running example SpGEMM (computing the intersection of the **i**-th row of **B** and the **j**-th column of **C**), this property ensures that for any specific **k**-coordinate in **B**, there can be at most one matching **k**-coordinate in **C**. If a row in the cross-comparison mask yields entirely `false` values, it indicates a strict coordinate mismatch, meaning the specific **k**-index exists in one operand but not the other, and therefore will not contribute to the final tensor contraction.

As illustrated in the detailed dataflow of Figure 6, the pass implements this all-pairs comparison natively within the SIMD pipeline. The n coordinate values from **B** are individually broadcast into full-width vector registers and checked for equality against the contiguous coordinate vector from **C**. The resulting boolean vectors serve as masks to filter the loaded values of **C**. Subsequently, a horizontal reduction extracts the single matched value (or a structural zero) from each masked vector, producing the necessary operands for the core tensor computation.

Phase 2: Predicated Tensor Computation. Following the cross-comparison and the subsequent extraction of matched values, the pass proceeds to the core arithmetic computation. As illustrated at the transition into Phase 2 of Figure 6, the horizontal reduction of the masked vectors yields discrete scalar values. To maintain a fully vectorized dataflow pipeline, our initial approach is to repack these discrete values—comprising both matched non-zeros (*e.g.,* `c0_v`) and explicit zero values—back into a cohesive SIMD register. The operation `vector.from_elements` helps to achieve repackaging within the MLIR `Vector` dialect (Figure 7b), effectively constructing a dense vector operand that represents the filtered values of **C**.

With both operand vectors fully populated, the core arithmetic is executed natively via fixed-width SIMD instructions. Because the loop's output accumulator (*i.e.,* `%Aij`) is a scalar value, the pass cannot utilize a standard fused multiply-add (`vector.fma`). Instead, the semi-ring multiply-accumulate (MAC) decomposes into a two-step vectorized operation. First, the pass emits an unconditional element-wise vector multiplication (`arith.mulf`) between the **B** and **C** registers. Subsequently, the resulting product vector undergoes a horizontal sum reduction (`vector.reduction <add>`) to collapse the array into a single scalar value (`%a_v`), which is then added to the loop's running accumulator.

Crucially, to sustain peak instruction throughput, this vectorized computation is executed entirely without branching. Unlike the baseline MLIR lowering (Figure 7a), which strictly guards scalar computation with an `scf.if` branch to process only true non-zeros, our transformed fast lane unconditionally executes the SIMD arithmetic across the entire vector width. While this approach

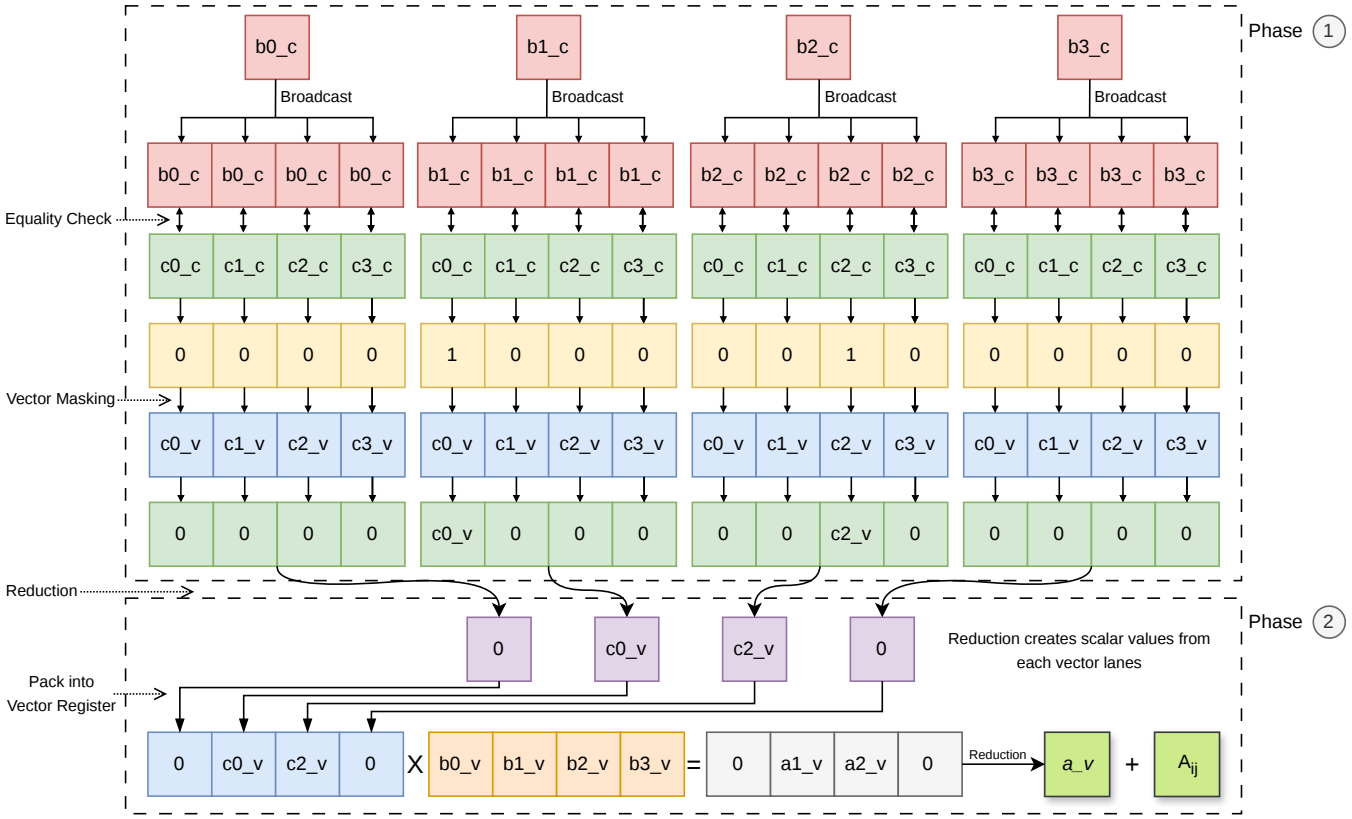


Figure 6: Dataflow architecture of a branchless 4×4 SIMD cross-compare algorithm for SpGEMM. The execution model bypasses scalar control-flow by broadcasting matrix B coordinates to perform parallel equality checks against matrix C coordinates. The resulting boolean masks filter matrix C values, which are then horizontally reduced to extract the matched non-zeros (or structural zeros) for the subsequent accumulate stage.

```
%contr = scf.if %cond -> (f64) {
    %b0_val = memref.load %B_v[%arg1]
    %c0_val = memref.load %C_v[%arg2]
    %mul = arith.mulf %b0_val, %c0_val
    %new_acc = arith.addf %Aij, %mul
}
```

(a) Baseline scalar execution generated from MLIR lowering.

```
// Pack C values into a vector
%c_vals_vec = vector.from_elements %c0_v, %c1_v, %c2_v, %c3_v
%contrib_vec = arith.mulf %b_vals_vec, %c_vals_vec
// Horizontal reduction
%a_v = vector.reduction <add>, %contrib_vec
%new_acc = arith.addf %Aij, %a_v
```

(b) Splyce-vectorized branchless SIMD computation and reduction.

Figure 7: Transform of the core computation in Phase 2.

inherently introduces redundant arithmetic—such as computing $b0_v \times 0 = 0$ on unmatched lanes, as depicted in the Phase 2 multiplication block—inserting conditional extraction logic to bypass these zero values would shatter the SIMD pipeline and induce frequent control-flow divergence. On modern superscalar architectures, the latency penalty for branch mispredictions and the resulting instruction pipeline flushes far outweighs the localized cost of executing dummy floating-point operations within the vector hardware.

Phase 3: Dynamic Pointer Advancement. After the predicated computation completes, the independent iteration cursors for Matrices **B** and **C** must be updated to prepare for the next iteration of the `scf.while` loop. Because the sparse coordinate distributions are irregular, the cursors cannot uniformly advance by the hardware vector width n. If one tensor's loaded coordinates vastly outpace the other's, aggressively advancing both pointers by n would skip valid intersections.

To theoretically maintain a purely vectorized pipeline across the entire coiteration cycle, our initial approach formulates pointer advancement as a branchless sequence of vector operations. The underlying formulation relies on establishing a safe execution boundary, or *pivot*, which represents the largest coordinate up to which both tensors have been fully inspected. As illustrated in Figure 8,

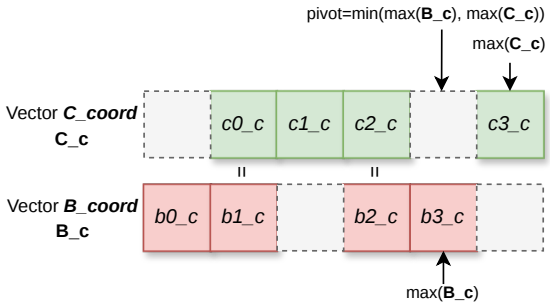


Figure 8: Branchless pointer advancement strategy.

this pivot is dynamically calculated as the minimum of the maximum coordinates currently loaded in the respective vector chunks (*i.e., pivot* = $min(max(B_coord), max(C_coord))$).

Rather than relying on conditional scalar loops to evaluate the next step, the compiler calculates the exact cursor advancement natively within the SIMD unit, detailed by the MLIR lowering in Figure 9. The scalar pivot is broadcast into a vector register and evaluated against the original loaded coordinate vectors via a parallel less-than-or-equal-to comparison. This yields a boolean mask in which true lanes indicate elements that have been safely consumed.

To quantify the final advancement step, these boolean results are zero-extended into integers (converting true to 1 and false to 0) and summed using a horizontal vector reduction. The resulting scalar unconditionally increments the tensor's base pointer for the next loop iteration. By formulating this advancement entirely through vector comparisons and reductions, the methodology achieves dynamic, data-dependent cursor updates without introducing a single control-flow branch.

3.4 Maximizing Parallel Throughput

While the purely vectorized, branchless architecture proposed above theoretically maximizes SIMD lane utilization, practical hardware constraints dictate that aggressive vectorization is not universally optimal. Specifically, the latency overheads associated with dynamically extracting matching values from vectors, repacking discrete values into new vector registers, and executing cross-lane shuffles can frequently eclipse the throughput gains of parallel execution.

To mitigate these potential microarchitectural bottlenecks, the n non-zeros processed in a single fast-lane cycle can alternatively be handled via highly unrolled scalar instructions. For all three phases discussed, it is theoretically possible to bypass the SIMD pipeline entirely and emit discrete scalar operations:

- **Phase 1**: Instead of vectorized masking and reduction to filter non-zeros, the $n \times n$ all-pairs cross-comparison can be executed directly via independent scalar equality checks and loading the matched values straight into scalar registers.
- **Phase 2**: Instead of repacking the extracted matching values back into SIMD registers, the arithmetic can be computed sequentially using unrolled scalar MAC instructions.
- **Phase 3**: The branchless pointer calculation, as shown in Figure 9, can bypass the vector unit entirely. As demonstrated in Figure 10, the pointer advancement can be executed using unrolled scalar comparisons and a tree-reduction.

Because of these competing microarchitectural trade-offs, the optimal configuration for the fast lane is a finely-tuned hybrid execution model. Rather than relying on theoretical assumptions or runtime auto-tuning, the architectural design of our final compiler pass is strictly informed by hardware realities—specifically, how the CPU's Out-of-Order execution engine and ILP interact with vector packing latencies. We conducted an ablation study to determine exactly which phases of our pass must be vectorized.

4 Evaluation

In this section, we evaluate the performance, scalability, and microarchitectural behavior of our proposed dual-path MLIR optimization pass. Our experimental analysis is designed to answer four primary research questions:

- **RQ1 (Phase Ablation):** Among data loading, tensor computation, and pointer advancement, does vectorizing all three execution phases yield the optimal microarchitectural performance?
- **RQ2 (Vectorization Overhead):** Does the performance benefit of vectorizing sparse coiteration via SIMD instructions outweigh the computational overhead of processing explicit zero values?
- **RQ3 (Sparsity Scaling):** Does the proposed dual-path architecture maintain its performance advantages across varying tensor dimensions and data densities?
- **RQ4 (Parallel Scalability):** Are the ILP gains achieved by Splyce orthogonal to macro-scale thread-level parallelism (TLP) when scaled across parallel multicore environments?

All experiments were conducted on a dual-socket Intel Xeon Gold 6430 (Sapphire Rapids) server. The system features a 2-node NUMA architecture with a total of 64 physical cores (32 per socket). To isolate microarchitectural behavior and ensure deterministic, cycle-accurate profiling, Simultaneous Multithreading (SMT) was disabled, and all benchmark threads were strictly pinned to a single NUMA node. The processors feature 120 MB of shared L3 cache and 2 MB of dedicated L2 cache per core, alongside native AVX-512 instruction support. The host operating system is Ubuntu 24.04.4 LTS (Linux kernel 6.x). The Splyce optimization pass is implemented and evaluated against the LLVM/MLIR `23.0.0-git` mainline.

All benchmark kernels were compiled with strict performance flags: `-O3 -march=native`. Although the `-Ofast` compiler flag enables floating-point reassociation—allowing the compiler to break the loop-carried dependency chain in multiply-accumulate operations that would otherwise negate vectorization gains—the default MLIR sparsifier pipeline does not enable *fast-math* optimizations. Consequently, Splyce inherits these conservative floating-point semantics by default throughout our evaluation.

4.1 Phase Ablation

To address **RQ1** and determine the optimal microarchitectural configuration, we conducted a rigorous ablation study on the SpGEMM fast lane. By selectively enabling SIMD vectorization across the three execution phases, we isolated the performance impact of each component. The results, detailed in Table 1 and Figure 11, reveal a critical insight: full-pipeline vectorization does not yield the optimal microarchitectural performance.

```
%pivot_v = vector.broadcast %pivot
%b_le_vec = arith.cmpi ule, %b_c, %pivot_v
%b_le_int = arith.extui %b_le_vec
%b_step_val = vector.reduction <add>, %b_le_int
%b_step = arith.index_cast %b_step_val
%new_ptr_B = arith.addi %ptr_B, %b_step
```

Figure 9: Branchless vectorized pointer advancement.

Table 1: Ablation study of SpGEMM coiteration phases (✓: SIMD, ×: scalar). Baseline (58.57s) represents the unmodified MLIR sparsifier.

#	Phases 1	2	3	Insn. (10^{11})	IPC	Br. Miss (10^7)	Time (s)	Speedup
–	Original MLIR			1.60	1.31	34.10	58.57	1.00×
1	×	×	×	2.72	3.95	5.32	33.01	1.77×
2	×	×	✓	2.97	3.14	5.34	45.36	1.29×
3	×	✓	×	2.93	4.54	5.31	30.91	1.89×
4	×	✓	✓	3.05	3.24	5.33	45.19	1.30×
5	✓	×	×	1.60	3.64	5.32	**21.10**	**2.78×**
6	✓	×	✓	1.45	1.73	5.32	40.25	1.46×
7	✓	✓	×	1.68	3.31	5.31	24.37	2.40×
8	✓	✓	✓	1.54	1.72	5.32	42.91	1.36×

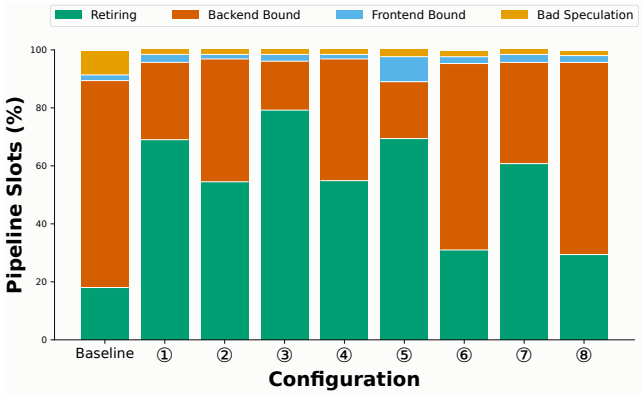


Figure 11: Top-Down Microarchitecture Analysis (TMA) pipeline slot utilization across different configurations.

Before analyzing the SIMD scaling, the data demonstrates the inherent structural advantage of the dual-path model itself. Compared to the unmodified MLIR baseline, every dual-path configuration achieves an approximate 6.41× reduction in branch mispredictions. This is the direct consequence of eliminating the divergent `if-else` coordinate matching logic from the `scf.while` hot path. By migrating from unpredictable data-dependent branches to bulk cross-comparisons, the architecture drastically reduces pipeline flushes. Figure 11, a Level 1 Top-Down Microarchitecture Analysis (TMA) [47] chart detailing all phase configurations alongside the MLIR baseline, visually corroborates this by demonstrating a strict reduction in *Bad Speculation* pipeline slots.

However, while this structural transformation mitigates control-flow penalties, attempting to vectorize the pointer advancement

```
%b0_le = arith.cmpi ule, %b0, %pivot : index ...
%b0_adv = arith.select %b0_le, %c1, %c0 : index ...
%b_s01 = arith.addi %b0_adv, %b1_adv : index
%b_s23 = arith.addi %b2_adv, %b3_adv : index
%b_step = arith.addi %b_s01, %b_s23 : index
%new_ptr_B = arith.addi %ptr_B, %b_step : index
```

Figure 10: Unrolled scalar variant of pointer advancement.

(Phase 3) introduces a fatal execution bottleneck. As explicitly shown in Table 1, the lowest overall speedups (2, 4, 6, and 8) occur precisely when Phase 3 is vectorized. Consequently, as illustrated in Figure 11, these specific configurations suffer a massive spike in *Backend Bound* stalls. This indicates that the CPU is stalling not due to memory latency or frontend starvation, but because its backend execution units are severely overwhelmed. Advancing dynamically bounded sparse pointers in SIMD not only requires expensive cross-lane shuffles, but it also triggers a severe microarchitectural hardware clash. The CPU's Address Generation Unit (AGU) requires memory addresses to reside in scalar General Purpose Registers (GPRs). When pointer arithmetic is forced into vector registers—either by design or by the LLVM SLP auto-vectorizer under `-O3`—the hardware must execute high-latency domain-crossing instructions (*e.g.,* `vpextrq`) or serialized Gather operations to feed the AGU for the subsequent iteration. To bypass this compiler cost-model myopia and preserve optimal AGU throughput, Phase 3 must remain strictly scalar, necessitating explicit `-fno-slp-vectorize` and `-fno-vectorize` compiler flags.

Beyond the penalties of Phase 3, the ablation data highlights the absolute necessity of vectorizing the memory loads and coordinate cross-comparisons (Phase 1). Configurations where Phase 1 remains scalar exhibit a massive inflation in total executed instructions. Notably, 3 achieves the highest Instructions Per Cycle (IPC) across all configurations, yet yields a suboptimal execution time. This empirical data demonstrates a classic SIMD IPC paradox: the scalar coiteration loop's high IPC consists of billions of lightweight scalar coordinate comparisons (an $n \times n$ matching complexity) that keep the pipeline highly active but yield little semantic progress towards the high-level tensor operation. Vectorizing Phase 1 collapses this complexity, mathematically deflating the total instruction count via bulk SIMD comparisons and proving mandatory for achieving significant speedup.

Having established the necessity of a vectorized Phase 1 and a scalar Phase 3, the final architectural decision rests on Phase 2 (Predicated Tensor Computation). Table 1 confirms that 5, which executes Phase 2 using standard scalar instructions, achieves the absolute peak speedup (2.78×). This superiority stems from a microarchitectural trade-off between SIMD packing latencies and ILP. To execute Phase 2 in SIMD, the hardware must first repack the dynamically matched sparse values into contiguous vector registers—a process strictly gated by the Phase 1 vector mask. This dynamic permutation overhead severely degrades throughput. Conversely, emitting independent scalar MAC instructions allows the modern OoO execution engine to naturally exploit ILP, completely bypassing the SIMD repack latencies. Thus, avoiding vectorization overhead in Phase 2 in favor of hardware-driven ILP ultimately

Table 2: Ablation study of SpGEMM coiteration phases with associative math enabled in Splyce (✓: SIMD, ×: scalar). Baseline (58.57s) represents the unmodified MLIR sparsifier.

#	Phases 1	2	3	Insn. (10^{11})	IPC	Br. Miss (10^7)	Time (s)	Speedup
–	Original MLIR			1.60	1.31	34.10	58.57	1.00×
1-f	×	×	×	2.74	4.54	5.32	32.96	1.78×
2-f	×	×	✓	2.91	3.20	5.32	43.59	1.34×
3-f	×	✓	×	2.88	4.45	5.31	31.11	1.88×
4-f	×	✓	✓	3.01	3.17	5.34	45.52	1.29×
5-f	✓	×	×	1.37	3.21	5.31	**20.48**	**2.86×**
6-f	✓	×	✓	1.22	1.56	5.32	37.50	1.56×
7-f	✓	✓	×	1.47	3.43	5.32	20.62	2.84×
8-f	✓	✓	✓	1.33	1.68	5.33	37.91	1.54×

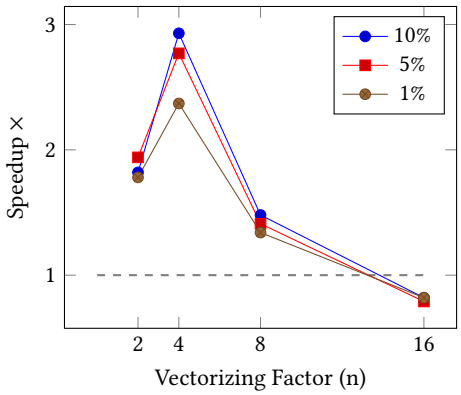


Figure 12: SpGEMM performance vs. vectorization factor (n).

yields the optimal performance. Because this optimal configuration is highly dependent on underlying hardware capabilities, Splyce exposes an optional compile-time flag to independently toggle the vectorization of each phase.

To preserve strict IEEE 754 semantics, the default MLIR sparsifier pipeline prohibits floating-point reassociation, which inherently restricts the potential for tree-height reduction. To explore this optimization space, Splyce introduces an optional flag that explicitly enables associative math transformations. As detailed in Table 2, activating this flag accelerates execution across all eight phase configurations relative to the original MLIR program. However, to uphold standardized floating-point guarantees and align with the default MLIR sparsifier, we isolate this ablation and omit the associative flag in all subsequent experiments.

Beyond the phase-level vectorization strategy, the efficiency of the dual-path model depends heavily on the hardware vector length (n), which dictates the number of elements processed per loop iteration. For 64-bit double-precision floating-point workloads, the native AVX-512 instruction set allows for a maximum vector width of $n = 8$. However, as detailed in Figure 12, maximizing the vector width to $n = 8$ empirically yields suboptimal performance across various sparsity densities.

This degradation reveals a fundamental microarchitectural bottleneck due to AVX-512 predicate register pressure. While the architecture natively supports 8 elements per vector register, it provisions only 7 usable mask registers (`k1`–`k7`) for conditional execution, as `k0` is hardwired to disable masking [3]. As the vectorization factor scales to $n = 8$, the intermediate boolean operations required to compute the Phase 1 coordinate cross-comparison masks exceed this 7-register limit. Consequently, the compiler is forced to spill active predicate masks to standard GPRs and execute costly `kmov` instructions to shuttle masks in and out of the execution units. Coupled with the inherently lower SIMD lane utilization that wider vectors suffer when processing sparse data, this register spilling severely degrades overall throughput.

Recognizing that the optimal vector width is dictated by a delicate balance between hardware register capacity, ISA constraints, and dataset sparsity, we designed our MLIR optimization pass to expose the vectorization factor n as a highly configurable compilation parameter rather than a rigid architectural constant. Based on our empirical profiling, setting $n = 4$ achieves the optimal microarchitectural trade-off for our target platform; consequently, this configuration is adopted as the standard vector width for all remaining experiments in this evaluation.

4.2 Vectorization Overhead

To address **RQ2** and quantify the computational trade-offs of our approach, we evaluated five sparse tensor contraction kernels: SpGEMM, SpMSpV, SpMTTKRP, SpMMH, and SpTTSpM. This initial evaluation used synthetic datasets with a uniform 5% non-zero values along the innermost compressed dimension. As detailed in Table 3, the proposed dual-path architecture yields end-to-end speedups ranging from 1.96× to 2.86×, achieving an overall geometric mean speedup of 2.38× against the standard MLIR baseline. This data explicitly proves our hypothesis: the immense throughput benefits of bulk SIMD coordinate matching and instruction deflation comprehensively outweigh the computational overhead of processing explicit zero-padding within the vector lanes. Furthermore, to validate the practical applicability of this optimization beyond uniformly distributed synthetic data, we extended our evaluation to highly irregular, real-world sparse tensors sourced from the SuiteSparse Matrix Collection [14, 27]. Across these production-grade datasets, the dual-path fast lane demonstrates improvements across the kernels.

4.3 Sparsity Scaling

Addressing **RQ3** is critical for establishing the operational boundaries of the optimization—specifically, determining the sparsity thresholds that dictate when this optimization pass yields profitable returns. To map this behavior, we conducted a sensitivity analysis on the SpGEMM kernel, sweeping the sparsity of the input matrices (uniform random distribution; identical for both operands). As illustrated in Figure 13, the performance delta between the baseline and the optimized fast lane scales positively with the sparsity of the matrix. Crucially, even at an extremely low 0.01% sparsity regime, the optimized kernel maintains strict execution parity with the unmodified baseline.

This trend extends to real-world datasets (Table 4). While many production tensors often possess much lower global sparsity than our synthetic benchmarks, the same convergence toward 1.0× speedup is observed in the limit, such as with the `internet` dataset

Table 3: Performance of sparse kernels on synthetic datasets (5% uniform sparsity). Boldface variables in the equations denote sparse tensors, whereas non-bold variables represent dense tensors.

Kernel	Equation	Input Dimension	Baseline (s)	Optimized (s)	Speedup
SpGEMM	$A_{ij} = \sum_k \mathbf{B}_{ik} \cdot \mathbf{C}_{kj}$	$\mathbf{B}(5k \times 5k)$, $\mathbf{C}(5k \times 5k)$	58.71	20.83	2.82×
SpMSpV	$y_i = \sum_j \mathbf{B}_{ij} \cdot \mathbf{x}_j$	$\mathbf{B}(100 \times 100M)$, $\mathbf{x}(100M)$	4.81	1.68	2.86×
SpMTTKRP	$A_{ik} = \sum_{k,l} \mathbf{B}_{ikl} \cdot \mathbf{C}_{lj} \cdot \mathbf{D}_{kj}$	$\mathbf{B}(1k \times 1k \times 1k)$, $\mathbf{C}(1k \times 1k)$, $\mathbf{D}(1k \times 1k)$	24.33	10.65	2.28×
SpMMH	$A_{ij} = \sum_k \mathbf{B}_{ik} \cdot C_{kj} \cdot \mathbf{D}_{ij}$	$\mathbf{B}(5k \times 5k)$, $C(5k \times 5k)$, $\mathbf{D}(5k \times 5k)$	63.82	32.63	1.96×
SpTTSpM	$A_{ijr} = \sum_k \mathbf{B}_{ijk} \cdot \mathbf{C}_{kr}$	$\mathbf{B}(500 \times 500 \times 500)$, $\mathbf{C}(500 \times 500)$	30.53	14.39	2.12×

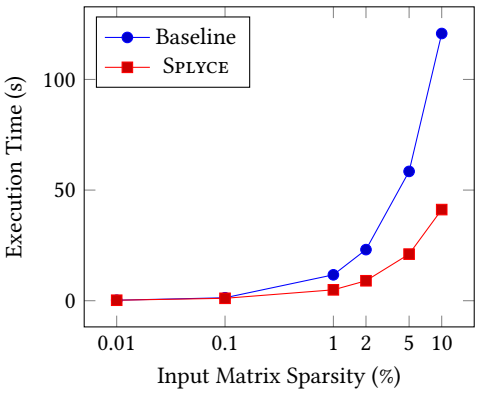


Figure 13: SpGEMM execution time vs. sparsity.

(1.33e-3% sparsity). Crucially, these results highlight the performance safety of our dual-path strategy. The vectorized fast-lane targets throughput gains within dense clusters, while the scalar epilogue ensures that execution remains consistent with the baseline in the sparse limit. In this context, global sparsity is not the sole determining factor, as globally sparse datasets can still contain highly dense local regions. The SpMSpV kernel results in Table 4 exemplify this: the highly sparse `stokes` matrix achieves a significant speedup, whereas `hugetrace-00020`, which falls in a similar range, shows almost no speedup. However, the *SIMD(%)* metric reveals that the portion of elements consumed by the SIMD fast-lane in the latter case is near zero, explaining the lack of speedup for that specific SuiteSparse matrix.

At the same time, the *SIMD(%)* metric alone is not a definitive performance predictor, since SIMD lanes may still process structural zero values within their wide execution width. Indeed, our extended evaluation across SpGEMM (Figure 14), SpMSpV (Figure 15), and SpMTTKRP (Figure 16) kernels reveals that while Splyce delivers significant speedups across the majority of diverse SuiteSparse matrices, a small subset incurs a negative speedup. Furthermore, this behaviour is highly kernel-dependent: a dataset that trigger a slowdown in one operation may still accelerate in another. For example, the `cdde1` dataset exhibits a performance regression during the SpGEMM kernel, yet achieves positive speedup in both SpMSpV and SpMTTKRP. Because we currently lack a definitive metric to capture these complex dataset-kernel interactions, identifying a priori which data properties trigger regression for specific sparse computations remains an open research problem. Nevertheless, by coordinating these paths, Splyce broadly preserves the performance benefits harvested in dense regions without suffering significant degradation in the vast majority of sparse environments. Because execution seamlessly falls through to the epilogue when the vector-width check (`condition_c`) is not met, the system avoids the speculative overhead and register-packing penalties that would otherwise occur where vectorization is not viable.

Table 4: Performance of Splyce on real-world sparse tensor datasets. Bold inputs are real-world matrix and the rest are synthetic with matching sparsity with a minimum of 0.001%.

Kernel	Dataset	Sparsity(%)	Base.(s)	Optim.(s)	Speedup	SIMD(%)
SpGEMM	internet (**B,C**)	1.33e-3	210.12	210.09	1.00×	3.45
	exdata_1 (**B,C**)	3.16	107.77	99.61	1.08×	24.85
	c8_mat11 (**B**)	9.37	115.47	51.79	2.23×	97.90
	heart1 (**B,C**)	10.97	33.86	17.73	1.91×	98.01
SpMSpV	stokes (**B**)	2.66e-4	7.94	4.07	1.71×	67.77
	mycielskian18 (**B**)	0.78	2.79	1.66	1.68×	76.63
	arabic-2005 (**B**)	1.24e-4	14.87	8.84	1.68×	81.37
	hugetrace-00020 (**B**)	1.87e-5	7.07	6.84	1.03×	0.00
SpMTTKRP	heart1 (**D**)	10.97	1385.74	636.46	2.18×	98.57
	CAG_mat364 (**C,D**)	10.25	5.06	2.46	2.06×	87.13
	struct4 (**C,D**)	1.26	78.28	49.51	1.58×	95.86
	cavity26 (**C,D**)	0.66	20.26	14.48	1.40×	92.03
SpMMH	bayer01 (**B,D**)	8.33e-3	51.84	54.60	0.95×	21.75
	msc23052 (**B,D**)	0.22	193.47	114.05	1.70×	98.30
	smt (**B,D**)	0.57	526.06	438.67	1.20×	99.38
	mark3jac020 (**B,D**)	6.74e-2	1.98	1.90	1.04×	41.03
SpTTSpM	barth4 (**C**)	6.48e-2	25.07	24.14	1.04×	45.71
	rdist1 (**C**)	0.55	129.13	86.21	1.50×	84.43
	psmigr_1 (**C**)	0.51	1256.14	562.02	2.24×	95.25
	EX6 (**C**)	0.69	620.91	288.39	2.15×	95.46

4.4 Parallel Scalability

To investigate the interaction between our vectorization pass and TLP, we evaluated the scalability of the proposed transformation on a 32-core NUMA node. For this analysis, we selected the SpMTTKRP kernel using the same data and configurations as defined in Table 4. This kernel was chosen as a representative stress test because its computational intensity provides a measurable throughput footprint even at high core counts, whereas lower-order kernels often collapse into microsecond execution ranges that are susceptible to measurement noise.

We parallelized the outermost `scf.parallel` loop while maintaining the Splyce-optimized inner coiteration loop. As illustrated in Figure 17, the results demonstrate that Splyce is inherently orthogonal to TLP, with synthetic data. The implementation achieves near-linear self-scaling, reaching a 31.18× speedup on 32 cores relative to the single-core Splyce execution. More importantly, the absolute speedup over the original unoptimized baseline scales proportionally, reaching 67.90× on 32 cores. This confirms that the branchless, dual-path architecture of Splyce operates strictly within the instruction-level parallelism (ILP) domain. By resolving the structural conflicts of sparse traversal at the register level, Splyce saturates execution units without introducing resource contention

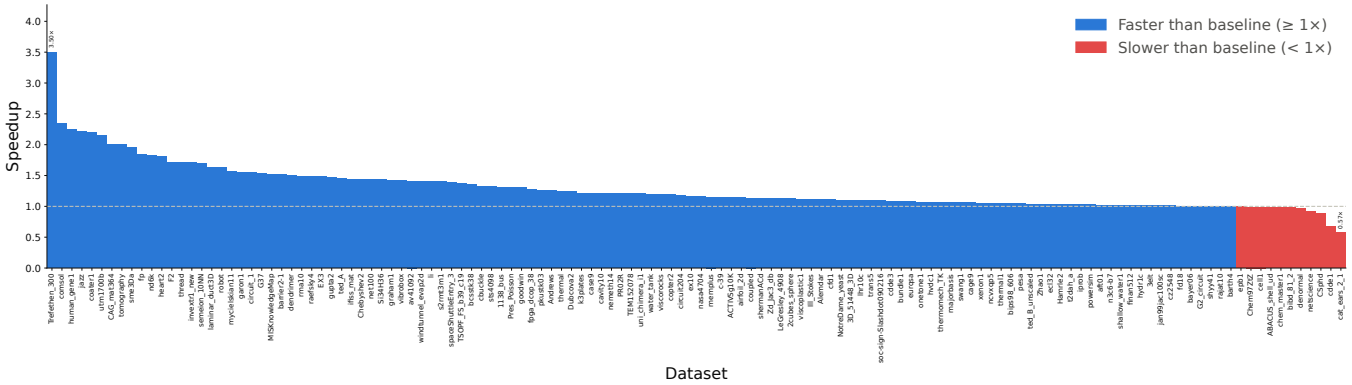


Figure 14: Speedup of Splyce for SpGEMM ($A_{ij} = \sum_k \mathbf{B}_{ik} \cdot \mathbf{C}_{kj}$). The same SuiteSparse matrix is used for both input matrices B and C. To represent the SuiteSparse Matrix Collection, we selected one matrix from each group, specifically choosing the one with the median non-zero density.

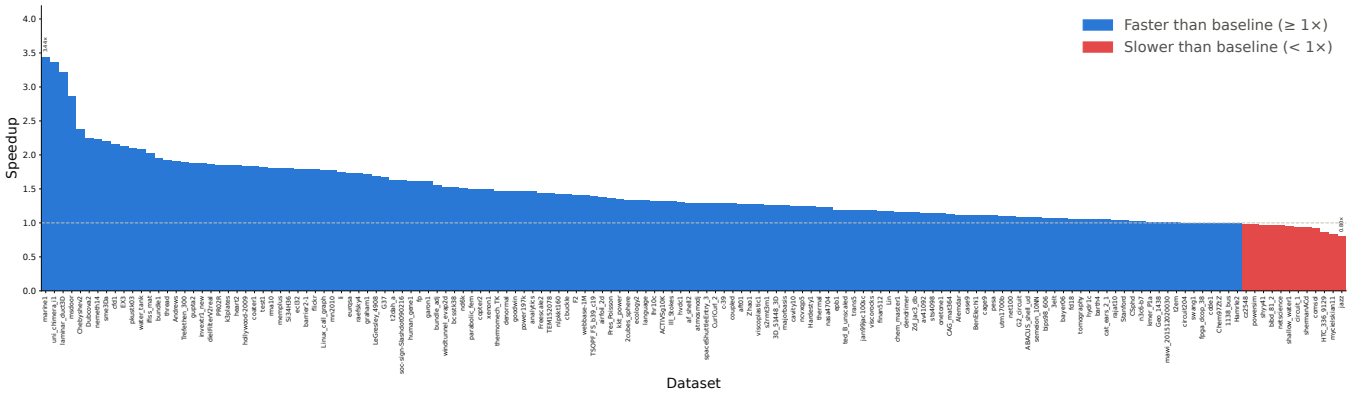


Figure 15: Speedup of Splyce for SpMSpV ($y_i = \sum_j \mathbf{B}_{ij} \cdot \mathbf{x}_j$). Input matrices follow the same selection strategy used for SpGEMM. The synthetic input vector is generated to match the sparsity factor of its corresponding matrix, subject to a minimum sparsity floor of 0.001%.

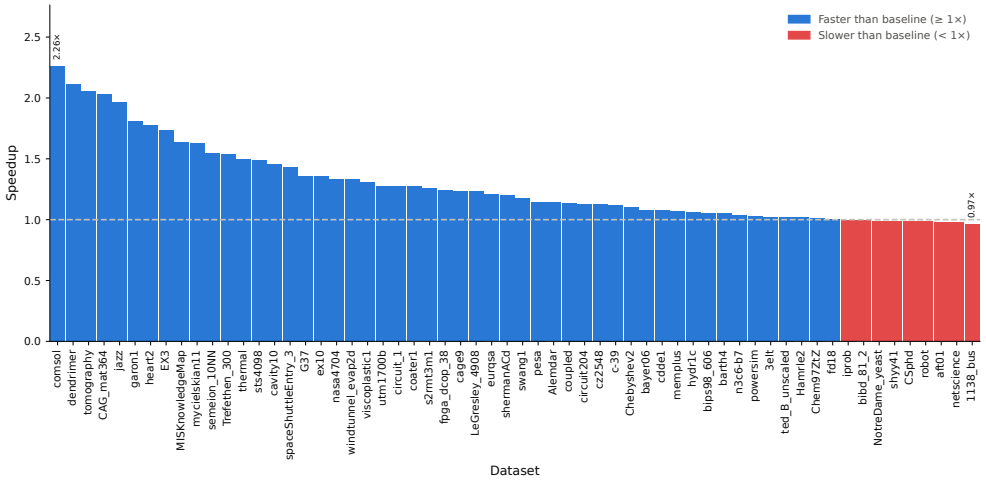


Figure 16: Speedup of Splyce for SpMTTKRP ($A_{ik} = \sum_{k,l} \mathbf{B}_{ikl} \cdot \mathbf{C}_{lj} \cdot \mathbf{D}_{kj}$). The 2D matrices follow the same selection strategy used for SpGEMM, with the same matrix used for both inputs C and D. The synthetic 3D input tensor is generated to match the sparsity factor of these matrices, subject to a minimum sparsity floor of 0.001%.

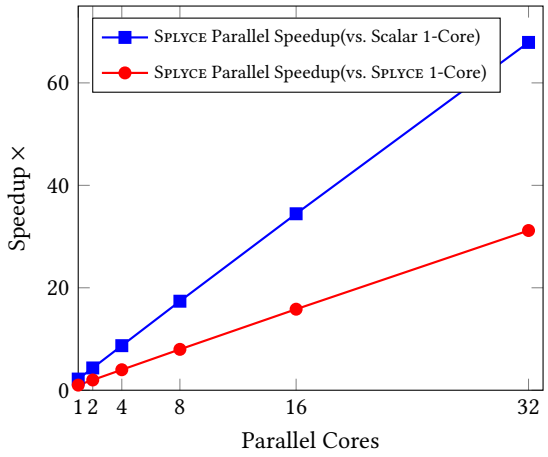


Figure 17: Scaling of the parallelized SpMTTKRP.

or synchronization overhead, thereby preserving macro-scale multicore efficiency. However, it should be noted that this near-linear scalability relies on the uniform workload distribution of synthetic data. Real-world sparse matrices typically exhibit irregular non-zero distributions across rows, which inherently introduces thread-level load imbalance and sub-linear scaling in practice.

5 Related Work

Early efforts in sparse code generation [7, 24, 42] evolved into the formalized merge lattice theory of TACO, which rivaled library-based approaches. This was subsequently expanded by the level-format abstraction, which modularized tensor dimensions into per-level types (*e.g.,* dense, compressed). By decoupling storage representation from the iteration engine, this abstraction enabled unified code generation across disparate layouts like CSR, CSF, and COO. Contemporary frameworks—including Finch [5], SparseTIR [48], and the MLIR `SparseTensor` [6] dialect—build upon this modular blueprint to facilitate deeper integration with general-purpose ecosystems.

While vectorization is a standard optimization [11, 20, 23, 45], existing strategies predominantly accelerate dense-sparse interactions (e.g., SpMV). These methods typically enforce regular access patterns through blocking or explicit memory-level zero-padding, which compromises compression benefits and inflates memory bandwidth requirements. While hand-tuned libraries (Intel MKL, cuSPARSE) and specialized hardware accelerators deliver peak performance, they inherently sacrifice the flexibility and format-agnostic composability provided by a generalized compiler infrastructure.

Traditional auto-vectorizers successfully handle interleaved data with constant strides, but conservatively assume branches are divergent. To address this, VecRC [32] introduces an auto-vectorizer that uses run-time checks to identify dynamic uniformity—cases where all SIMD lanes follow the same control path at run-time. VecRC leverages this uniformity to relax constraints on loops with control-dependent loop-carried dependencies, enabling vectorization without the overhead of speculation. While VecRC's probability-based cost model effectively mitigates predication overhead in uniform scenarios, standard auto-vectorizers remain structurally incapable of handling the dynamic, data-dependent intersections inherent in sparse *two-finger* loops.

The problem of two-finger loops is a well-known algorithmic bottleneck across various domains, including data compression [30], relational databases [8, 19], and sequence alignment in bioinformatics [39]. While extensive SIMD vectorization efforts have successfully accelerated these co-iteration patterns in compression and database sort-merge joins, these optimizations do not trivially translate to sparse tensor compilers. Unlike standard applications that traverse contiguous arrays to materialize matching identifiers, sparse tensor coiteration navigates coordinate arrays—imposing heavy memory indirection—and tightly fuses this unpredictable intersection logic directly with floating-point arithmetic. This limits the applicability of traditional SIMD techniques and necessitates the specialized vectorization pass proposed in this work.

6 Discussion and Future Work

Although Splyce supports compressed outputs, current evaluation materializes dense outputs to reduce measurement overhead. The current transformation targets two-way intersection coiteration while the union-based one is left for future work.

Multiway coiteration presents another extension of the dual-path approach. When the underlying operation permits associative decomposition, a multiway coiteration can be expressed as a sequence of two-way coiterations. This introduces additional choices in decomposition order and execution strategy. It may also change the phase-level trade-offs observed in Section 4.1. Multiway coiteration can increase the amount of computation performed per traversal step, potentially amortizing the overheads associated with value extraction, packing, and reduction. Consequently, more extensive vectorization may become beneficial for such workloads.

Blocked-sparse formats are particularly relevant because they introduce fixed-size dense regions within a sparse contraction. These dense sub-blocks expose regular computation that can be mapped to SIMD execution while retaining sparse contraction at the block level. This may increase the amount of useful computation performed within each fast-lane iteration and improve the amortization of coiteration overheads.

Finally, the current implementation targets CPU SIMD execution. However, the underlying idea of converting irregular, branch-dependent sparse coiteration into a more regular, parallel execution pattern is not inherently CPU-specific. This suggests an opportunity to adapt Splyce to heterogeneous accelerators, where similar speculative and predicated execution strategies could be used to expose more parallelism and improve hardware utilization.

7 Conclusion

In this paper, we presented Splyce, an MLIR optimization pass designed to resolve the structural conflict between compressed sparse storage and modern parallel hardware. By transforming the divergent scalar coiteration loop into a branchless, dual-path execution model, we enable modern superscalar engines to maximize ILP and saturate parallel execution units previously starved by sequential dependencies.

References

[1] 2026. 1. Introduction — cuSPARSE 13.2 documentation. https://docs.nvidia.com/cuda/cusparse/index.html [Online; accessed 16. Apr. 2026].

[2] 2026. Accelerate Fast Math with Intel® oneAPI Math Kernel Library. https://www.intel.com/content/www/us/en/developer/tools/oneapi/onemkl.html [Online; accessed 16. Apr. 2026].

[3] 2026. Intel® Xeon® Processor Scalable Family Technical Overview. https://www.intel.com/content/www/us/en/developer/articles/technical/xeon-processor-scalable-family-technical-overview.html [Online; accessed 1. May 2026].

[4] 2026. SIMD vectorization in LLVM and GCC for Intel® CPUs and GPUs. https://www.intel.com/content/www/us/en/developer/articles/technical/vectorization-llvm-gcc-cpus-gpus.html [Online; accessed 17. Apr. 2026].

[5] Willow Ahrens, Teodoro Fields Collin, Radha Patel, Kyle Deeds, Changwan Hong, and Saman Amarasinghe. 2025. Finch: Sparse and structured tensor programming with control flow. *Proceedings of the ACM on Programming Languages* 9, OOPSLA1 (2025), 1042–1072.

[6] Aart Bik, Penporn Koanantakool, Tatiana Shpeisman, Nicolas Vasilache, Bixia Zheng, and Fredrik Kjolstad. 2022. Compiler Support for Sparse Tensor Computations in MLIR. 19, 4, Article 50 (Sept. 2022), 25 pages. doi:10.1145/3544559

[7] Aart JC Bik and Harry AG Wijshoff. 1993. Compilation techniques for sparse matrix computations. In *Proceedings of the 7th international conference on Supercomputing*. 416–424.

[8] Peter Boncz, Marcin Zukowski, and Niels Nes. 2005. MonetDB/X100: Hyper-Pipelining Query Execution.. In *Cidr*, Vol. 5. 225–237.

[9] Tianqi Chen, Lianmin Zheng, Eddie Q. Yan, Ziheng Jiang, T. Moreau, L. Ceze, Carlos Guestrin, and A. Krishnamurthy. 2018. Learning to Optimize Tensor Programs. (2018), 3393–3404.

[10] Yuedan Chen, Kenli Li, Wangdong Yang, Guoqing Xiao, Xianghui Xie, and Tao Li. 2019. Performance-Aware Model for Sparse Matrix-Matrix Multiplication on the Sunway TaihuLight Supercomputer. *IEEE Transactions on Parallel and Distributed Systems* 30 (2019), 923–938. doi:10.1109/tpds.2018.2871189

[11] Kazem Cheshmi, Zachary Cetinic, and Maryam Mehri Dehnavi. 2022. Vectorizing sparse matrix computations with partially-strided codelets. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis* (Dallas, Texas) *(SC '22)*. IEEE Press, Article 32, 15 pages.

[12] Stephen Chou, Fredrik Kjolstad, and Saman P. Amarasinghe. 2018. Format abstraction for sparse tensor algebra compilers. *Proceedings of the ACM on Programming Languages* 2 (2018), 1 – 30. doi:10.1145/3276493

[13] Shail Dave, Riyadh Baghdadi, Tony Nowatzki, Sasikanth Avancha, Aviral Shrivastava, and Baoxin Li. 2021. Hardware Acceleration of Sparse and Irregular Tensor Computations of ML Models: A Survey and Insights. *Proc. IEEE* 109, 10 (2021), 1706–1752. doi:10.1109/JPROC.2021.3098483

[14] Timothy A. Davis and Yifan Hu. 2011. The university of Florida sparse matrix collection. *ACM Trans. Math. Softw.* 38, 1, Article 1 (Dec. 2011), 25 pages. doi:10.1145/2049662.2049663

[15] J. J. Dongarra, Jeremy Du Croz, Sven Hammarling, and I. S. Duff. 1990. A set of level 3 basic linear algebra subprograms. *ACM Trans. Math. Softw.* 16, 1 (March 1990), 1–17. doi:10.1145/77626.79170

[16] Agner Fog. 2023. The Microarchitecture of Intel, AMD, and VIA CPUs. https://www.agner.org/optimize/microarchitecture.pdf.

[17] Mahdi Ghorbani, Emilien Bauer, Tobias Grosser, and Amir Shaikhha. 2025. Compressed and Parallelized Structured Tensor Algebra. *Proc. ACM Program. Lang.* 9, OOPSLA1, Article 141 (April 2025), 29 pages. doi:10.1145/3720506

[18] Mahdi Ghorbani, Mathieu Huot, Shideh Hashemian, and Amir Shaikhha. 2023. Compiling Structured Tensor Algebra. 7, OOPSLA2, Article 229 (Oct. 2023), 30 pages. doi:10.1145/3622804

[19] Goetz Graefe. 1993. Query evaluation techniques for large databases. *ACM Computing Surveys (CSUR)* 25, 2 (1993), 73–169.

[20] Xianhao He, Haotian Wang, Jiapeng Zhang, Wangdong Yang, Anthony Theodore Chronopoulos, and Kenli Li. 2025. An Input-Aware Sparse Tensor Compiler Empowered by Vectorized Acceleration. In *Proceedings of the 62nd Annual ACM/IEEE Design Automation Conference* (San Francisco, California, United States) *(DAC '25)*. IEEE Press, Article 437, 7 pages. doi:10.1109/DAC63849.2025.11133371

[21] John L. Hennessy and David A. Patterson. 2011. *Computer Architecture, Fifth Edition: A Quantitative Approach* (5th ed.). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[22] So Hirata. 2003. Tensor Contraction Engine: Abstraction and Automated Parallel Implementation of Configuration-Interaction, Coupled-Cluster, and Many-Body Perturbation Theories. *The Journal of Physical Chemistry A* 107, 46 (2003), 9887–9897. arXiv:https://doi.org/10.1021/jp034596z doi:10.1021/jp034596z

[23] Marcos Horro, Louis-Noël Pouchet, Gabriel Rodríguez, and Juan Touriño. 2023. Custom High-Performance Vector Code Generation for Data-Specific Sparse Computations. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques* (Chicago, Illinois) *(PACT '22)*. Association for Computing Machinery, New York, NY, USA, 160–171. doi:10.1145/3559009.3569668

[24] George Karypis and Vipin Kumar. 1997. METIS: A Software Package for Partitioning Unstructured Graphs, Partitioning Meshes, and Computing Fill-Reducing Orderings of Sparse Matrices. https://conservancy.umn.edu/items/2f610239-590c-45c0-bcd6-321036aaad56 [Online; accessed 28. Apr. 2026].

[25] 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

[26] Tamara G. Kolda and Brett W. Bader. 2009. Tensor Decompositions and Applications. *SIAM Rev.* 51, 3 (2009), 455–500. arXiv:https://doi.org/10.1137/07070111X doi:10.1137/07070111X

[27] Scott P. Kolodziej, Mohsen Aznaveh, Matthew Bullock, Jarrett David, Timothy A. Davis, Matthew Henderson, Yifan Hu, and Read Sandstrom. 2019. The SuiteSparse Matrix Collection Website Interface. *Journal of Open Source Software* 4, 35 (2019), 1244. doi:10.21105/joss.01244

[28] Martin J. Kühn, Johannes Holke, Annette Lutz, Jonas Thies, Melven Röhrig-Zöllner, Alexander Bleh, Jan Backhaus, and Achim Basermann. 2023. SIMD vectorization for simultaneous solution of locally varying linear systems with multiple right-hand sides. *J. Supercomput.* 79, 13 (Sept. 2023), 14684–14706. doi:10.1007/s11227-023-05220-4

[29] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. 2021. MLIR: scaling compiler infrastructure for domain specific computation. In *Proceedings of the 2021 IEEE/ACM International Symposium on Code Generation and Optimization* (Virtual Event, Republic of Korea) *(CGO '21)*. IEEE Press, 2–14. doi:10.1109/CGO51591.2021.9370308

[30] D. Lemire and L. Boytsov. 2015. Decoding billions of integers per second through vectorization. *Softw. Pract. Exper.* 45, 1 (Jan. 2015), 1–29. doi:10.1002/spe.2203

[31] Rui Li, Aravind Sukumaran-Rajam, R. Veras, Tze Meng Low, F. Rastello, A. Rountev, and P. Sadayappan. 2019. Analytical Cache Modeling and Tile-size Optimization for Tensor Contractions. *SC19: International Conference for High Performance Computing, Networking, Storage and Analysis* (2019), 1–13. doi:10.1145/3295500.3356218

[32] Bangtian Liu, Avery Laird, Wai Hung Tsang, Bardia Mahjour, and Maryam Mehri Dehnavi. 2023. Combining Run-Time Checks and Compile-Time Analysis to Improve Control Flow Auto-Vectorization. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques* (Chicago, Illinois) *(PACT '22)*. Association for Computing Machinery, New York, NY, USA, 439–450. doi:10.1145/3559009.3569663

[33] Dorit Nuzman, Ira Rosen, and Ayal Zaks. 2006. Auto-vectorization of interleaved data for SIMD. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Ottawa, Ontario, Canada) *(PLDI '06)*. Association for Computing Machinery, New York, NY, USA, 132–143. doi:10.1145/1133981.1133997

[34] Eric Qin, Geonhwa Jeong, William Won, Sheng-Chun Kao, Hyoukjun Kwon, S. Srinivasan, Dipankar Das, G. Moon, S. Rajamanickam, and T. Krishna. 2021. Extending Sparse Tensor Accelerators to Support Multiple Compression Formats. *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)* (2021), 1014–1024. doi:10.1109/ipdps49936.2021.00110

[35] Ryan Senanayake, Changwan Hong, Ziheng Wang, Amalee Wilson, Stephen Chou, Shoaib Kamil, Saman P. Amarasinghe, and Fredrik Kjolstad. 2020. A sparse iteration space transformation framework for sparse tensor algebra. *Proceedings of the ACM on Programming Languages* 4 (2020), 1 – 30. doi:10.1145/3428226

[36] André Seznec and Pierre Michaud. 2006. A case for (partially) tagged geometric history length branch prediction. *The Journal of Instruction-Level Parallelism* 8 (Feb. 2006), 23. https://inria.hal.science/hal-03408381

[37] Yang Shi, U. Niranjan, Anima Anandkumar, and C. Cecka. 2016. Tensor Contractions with Extended BLAS Kernels on CPU and GPU. *2016 IEEE 23rd International Conference on High Performance Computing (HiPC)* (2016), 193–202. doi:10.1109/hipc.2016.031

[38] Shaden Smith and G. Karypis. 2015. Tensor-matrix products with a compressed sparse tensor. *Proceedings of the 5th Workshop on Irregular Applications: Architectures and Algorithms* (2015). doi:10.1145/2833179.2833183

[39] Temple F Smith, Michael S Waterman, et al. 1981. Identification of common molecular subsequences. *Journal of molecular biology* 147, 1 (1981), 195–197.

[40] J. A. Stankovic, K. Ramamritham, P. Shiah, and W. Zhao. 1989. *Real-Time Scheduling Algorithms for Multiprocessors.* Technical Report. USA.

[41] David W. Wall. 1991. Limits of instruction-level parallelism. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems* (Santa Clara, California, USA) *(ASPLOS IV)*. Association for Computing Machinery, New York, NY, USA, 176–188. doi:10.1145/106972.106991

[42] J.B. White and P. Sadayappan. 1997. On improving the performance of sparse matrix-vector multiplication. In *Proceedings Fourth International Conference on High-Performance Computing*. 66–71. doi:10.1109/HIPC.1997.634472

[43] Jeremiah Willcock and A. Lumsdaine. 2006. Accelerating sparse matrix computations via data compression. (2006), 307–316. doi:10.1145/1183401.1183444

[44] Samuel Williams, Andrew Waterman, and David Patterson. 2009. Roofline: an insightful visual performance model for multicore architectures. *Commun. ACM*

52, 4 (April 2009), 65–76. doi:10.1145/1498765.1498785
[45] Biwei Xie, Jianfeng Zhan, Xu Liu, Wanling Gao, Zhen Jia, Xiwen He, and Lixin Zhang. 2018. Cvr: Efficient vectorization of spmv on x86 processors. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization*. 149–162.
[46] Rohan Yadav, A. Aiken, and Fredrik Kjolstad. 2022. DISTAL: the distributed tensor algebra compiler. *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (2022). doi:10.1145/3519939.3523437
[47] Ahmad Yasin. 2014. A Top-Down method for performance analysis and counters architecture. In *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 35–44. doi:10.1109/ISPASS.2014.6844459
[48] 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*. 660–678.
[49] Tuowen Zhao, Tobi Popoola, Mary Hall, Catherine Olschanowsky, and Michelle Strout. 2022. Polyhedral specification and code generation of sparse tensor contraction with co-iteration. *ACM Transactions on Architecture and Code Optimization* 20, 1 (2022), 1–26.

A Artifact Appendix

A.1 Abstract

This artifact contains the implementation of Splyce, an MLIR pass that vectorizes sparse coiteration (`scf.while` two-finger loops) by transforming them into a dual-path version of branchless fast-lane and scalar-epilogue. The artifact includes the Splyce MLIR pass (built against LLVM/MLIR mainline), the five benchmark kernels evaluated in the paper (SpGEMM, SpMSpV, SpMTTKRP, SpMMH, SpTTSpM), synthetic sparse tensor generators, scripts to download and preprocess the SuiteSparse matrices used in Table 4, and analysis scripts that reproduce the phase-ablation study (Table 1, Figure 11), the speedup results (Table 3, Table 4), the sparsity scaling sweep (Figure 13), the vector width experiment (Figure 12), and the parallel scalability study (Figure 17). Reproducing these results validates the following central claims of the paper:

- Splyce reduces branch misprediction.
- In Splyce, phase config [5] is the best performing one on the particular hardware.
- Splyce achieves a 2.38× geomean speedup on synthetic data.
- Splyce sustains speedups on real-world SuiteSparse inputs.
- Splyce constrains its optimization within a single core.

Full instructions for environment configuration, compilation, and experiment reproduction are detailed in `README.md` of the repository. This appendix directs readers to specific sections of that online documentation for detailed execution steps.

A.2 Artifact checklist

- **Algorithm:** Dual-path (vectorized fast-lane + scalar epilogue) SIMD coiteration transform for sparse tensor contractions.
- **Program:** Splyce MLIR pass (C++/MLIR), benchmark drivers for SpGEMM, SpMSpV, SpMTTKRP, SpMMH, SpTTSpM.
- **Compilation:** LLVM/MLIR 23; `-O3 -mavx512f -mavx512vl`
- **Transformations:** `SparseTensor` → `SCF/Vector` → `LLVM-IR` intercepted by Splyce between `SCF/Vector` dialect stages.
- **Binary:** `splyce-opt` MLIR pass plugin, benchmark executables (pre-built on the server; source also provided for inspection/rebuild).
- **Data set:** Synthetic uniform-random sparse tensors; real-world matrices from the SuiteSparse Matrix Collection (*e.g., internet, exdata_1, c8_mat11, heart1, stokes, bayer01, barth4*)
- **Run-time environment:** Ubuntu 24.04.4 LTS, Linux kernel 6.x (as configured on the provided server)
- **Hardware:** Provided server—dual-socket Intel Xeon Gold 6430 (Sapphire Rapids), 64 physical cores (32/socket), AVX-512, 120 MB shared L3, 2 MB L2/core; SMT disabled, threads pinned to a single NUMA node.
- **Execution:** Single-threaded for Section 4.1- 4.3; multi-threaded (1-32 cores) for Section 4.4.
- **Metrics:** Wall-clock execution time, speedup vs. the pre-built unmodified MLIR baseline, IPC, branch misprediction count, TMA pipeline-slot breakdown.
- **Output:** CSV files with timings, plots matching Figures 11-17
- **Experiments:** Phase ablation (Table 1, Figure 11), vectorization overhead across 5 kernels (Table 3), real-world dataset evaluation (Table 4), Vector width ablation (Figure 12), Sparsity Scaling (Figure 13), Multicore Scaling (Figure 17).
- **How much disk space required (approximately)?:** ~116 GiB (Mostly for SuiteSparse and Synthetic matrices)
- **Publicly available?:** Yes
- **Code licenses (if publicly available)?:** Apache 2.0 (matching LLVM/MLIR licensing)
- **Data licenses (if publicly available)?:** SuiteSparse matrices are publicly available under their respective collection terms.
- **Workflow automation framework used?:** Shell Scripts, Python Scripts, CMake
- **Archived (provide DOI)?:** https://doi.org/10.5281/zenodo.21878752
- **Archived (Trovi):** https://trovi.chameleoncloud.org/dashboard/artifacts/4ad43ccf-564d-450a-abf2-25a99d5d3d70
- **Repository:** https://github.com/KabilanMA/Splyce

A.3 Description

A.3.1 Access. **Zenodo**: https://doi.org/10.5281/zenodo.21878752

A.3.2 Hardware dependencies. Reproducing the absolute timings reported in this work requires an *x86-64* processor with *AVX-512F* and *AVX-512VL* support. All experiments were conducted on a dual-socket Intel Xeon Gold 6430 (Sapphire Rapids) server running Ubuntu 24.04.4 LTS. The system features 64 total physical cores (32 per socket), 120 MB of shared L3 cache, and 2 MB of dedicated L2 cache per core. To isolate microarchitectural behavior and ensure deterministic, cycle-accurate profiling, Simultaneous Multithreading (SMT) was disabled, and all execution threads were pinned to a single NUMA node. Consequently, multi-socket hardware is not strictly required for reproduction. The Splyce optimization pass is implemented and evaluated against the LLVM/MLIR `23.0.0-git` mainline.

Regarding core counts, the single-thread evaluations in Sections 4.1 through 4.3 require only a single core. Conversely, the scalability evaluation in Section 4.4 requires a dense multi-core environment. At least 32 physical cores on a single NUMA node are necessary to reproduce the full sweep shown in Figure 17, although the general scaling trend remains observable on hardware with fewer cores.

A.3.3 Software dependencies. We defer to the `README.md` within the artifact repository for complete and up-to-date specifications regarding software dependencies and environment configuration.

A.3.4 Containerized Evaluation. In addition to the pre-configured server, we supply a `Dockerfile` and supporting documentation to facilitate local execution. Reviewers electing to use this method should be advised of two caveats. First, building the Docker image

requires compiling LLVM from source, a process that can be highly time-consuming depending on the host machine. Second, while this environment is fully capable of functional validation, we cannot guarantee the fidelity of the absolute execution timings or scaling metrics due to the inherent overhead of containerization.

A.3.5 Data sets. The evaluation utilizes both synthetic and real-world datasets. Synthetic tensors, featuring uniform random sparsity, are generated on-demand via the `experiments/gen_data.sh` script. Real-world matrices are retrieved dynamically from the SuiteSparse Matrix Collection using automated scripts included in the repository.

A.4 Installation

Detailed procedures for resolving the LLVM dependency–either by compiling from source or configuring a pre-existing build–are provided in the `README.md` file. The documentation subsequently outlines the compilation steps for Splyce, along with a *Getting Started* section designed to simply validate the environment setup through a minimal working example.

A.5 Experiment workflow

To facilitate reproducibility, the artifact provides six automated scripts capable of generating all plots and corresponding CSV data files discussed in the Evaluation section. We also supply reference plots and baseline CSV output against which reviewers can validate their generated results. Comprehensive documentation detailing each experiment is maintained in the `README.md` of the repository.

A.6 Evaluation and expected results

Every experimental workflow is isolated within a specific folder under the `experiments` directory. These folders contain the baseline execution numbers and reference plots. Upon executing the provided scripts, reviewers can validate their setup by ensuring their newly generated data and figures closely match these reference files, which were used to populate the tables and figures in the paper.