ASaP: Automatic Software Prefetching for Sparse Tensor Computations in MLIR

Konstantinos Sotiropoulos
ioanniss@chalmers.se
Chalmers University of Technology & University of Gothenburg
Department of Computer Science and Engineering
Gothenburg, Sweden

Jonas Skeppstedt
jonas.skeppstedt@cs.lth.se
Lund University
Department of Computer Science
Lund, Sweden

Per Stenström
pers@chalmers.se
Chalmers University of Technology & University of Gothenburg
Department of Computer Science and Engineering
Gothenburg, Sweden

Abstract

Sparse tensor computations suffer from irregular memory access patterns that degrade cache performance. While software prefetching can mitigate this, existing compiler approaches lack the semantic insight needed for effective optimization.

We present **ASaP**, an automatic software prefetching framework integrated within MLIR's sparse tensor dialect. By leveraging semantic information—tensor formats and loop structure—available during sparsification, ASaP determines accurate buffer bounds and injects prefetches in both innermost and outer loops, achieving broader coverage than prior work.

Evaluated on SuiteSparse matrices, ASaP demonstrates significant performance gains for unstructured matrices. For SpMV with innermost-loop prefetching, ASaP achieves 1.38× speedup over Ainsworth & Jones. For SpMM with outer-loop prefetching, ASaP achieves 1.28× speedup while Ainsworth & Jones fails to generate prefetches. Our experiments reveal that disabling inaccurate hardware prefetchers frees critical resources for software prefetching, suggesting future architectures should expose prefetcher control as an optimization interface.

CCS Concepts

• **Software and its engineering → Source code generation**.

Keywords

software prefetching, sparse data structures, sparse tensors

ACM Reference Format:
Konstantinos Sotiropoulos, Jonas Skeppstedt, and Per Stenström. 2025. ASaP: Automatic Software Prefetching for Sparse Tensor Computations in MLIR. In *Proceedings of The Eleventh Annual Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC 2025).* ACM, New York, NY, USA, 11 pages. https://doi.org/XXXXXXX.XXXXXXX

LLVM-HPC 2025, St. Louis, MO

ACM ISBN 978-x-xxxx-xxxx-x/YYYY/MM
https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

Sparse tensor computations, which involve multi-dimensional arrays with many zero elements, are crucial in a wide range of application domains such as scientific computing [16], machine learning [31], and graph analytics [14]. In scientific computing, discretizing partial differential equations into 3D grids leads to sparse matrices, as only key grid points are used. In graph analytics, adjacency matrices representing node connections are sparse since most nodes connect to only a few others. In machine learning, a neural network inference produces sparse weight matrices through pruning close-to-zero values after training, while recommendation systems result in sparse interaction matrices because users typically interact with a limited subset of items.

To manage the high proportion of zero elements, sparse tensors can be compressed using various formats [13] that store only non-zero values and encode their respective coordinates into a number of indirectly indexed buffers. The encoding method introduces irregular memory access patterns, and coupled with the fact that sparse tensor applications often exhibit a large reuse distance relative to the last-level cache, results in inefficient use of the cache-memory subsystems [8, 25].

Prefetching is a technique used to reduce memory latency by bringing data closer to the processor before it is actually needed. Contemporary hardware prefetchers [2], which predict memory accesses based on past patterns, struggle with the irregular access patterns [7] inherent in sparse tensor computations. Hardware prefetchers typically track regularity in access streams and lack the ability to adapt to the non-linear and non-contiguous memory accesses that sparse formats generate. Designing a hardware prefetcher capable of handling such complexity would require a significant degree of programmability that extends beyond mere configurability [37].

In this work, we argue that Multi-Level Intermediate Representation (MLIR) [24] is a compelling framework for software prefetching, that is, prefetching via explicit prefetch instructions, in sparse computations. MLIR provides a modular and extensible infrastructure for building optimizing compilers that operate across multiple levels of abstraction, and is a key enabler of modern domain-specific compilation flows [3, 11]. It achieves this flexibility through a system of dialects and progressive lowering, where high-level representations are gradually transformed into low-level Intermediate Representations (IRs) suitable for target-specific optimizations. This design

enables fine-grained control over transformations [27] and has already revitalized classic loop optimizations such as fusion, tiling, and vectorization [39] across a variety of platforms.

We introduce ASaP, the first automatic software prefetching framework built on MLIR's sparse tensor dialect. Our approach extends the dialect's sparsification transformation to inject prefetch instructions for indirect memory accesses of the form `B[A[i]]`. Previous software prefetching schemes, such as the universal, low-level compiler pass by Ainsworth & Jones [5, 6], target general-purpose code and treat indirect accesses in isolation. In contrast, ASaP specializes to the sparse tensor domain and leverages semantic information including sparse tensor formats, buffer structure, and loop hierarchy—the context in which these accesses are generated. This domain-specific insight enables more accurate prefetch generation, leading to better performance across memory-bound workloads.

Our **contributions** are:

(1) We present ASaP[1], the first software prefetching framework for sparse tensors built on MLIR. ASaP handles any sparse format expressible in the sparse tensor dialect

(2) We demonstrate how ASaP leverages sparsification-time knowledge to inject prefetches precisely where indirect accesses are generated, rather than detecting them post-hoc. Unlike Ainsworth & Jones, ASaP determines accurate buffer bounds using sparse tensor semantics instead of inferring them from loop bounds—a critical distinction since underestimation reduces prefetch coverage, creating the performance advantage we demonstrate in Section 5.3. ASaP generates runtime instructions to determine structure sizes when memory allocation sites are not visible during compilation, avoiding the need to trace pointer origins.

(3) We implement both innermost- and outer-loop prefetching strategies, showing how semantic understanding enables safe prefetch placement across different loop levels.

(4) We evaluate ASaP on SpMV and SpMM using the SuiteSparse matrix collection in the CSR format, achieving significant performance improvements over state-of-the-art approaches, particularly on unstructured matrices where indirect memory accesses create irregular access patterns with high memory latency. Our experiments reveal that disabling inaccurate hardware prefetchers frees critical resources—MSHRs and memory bandwidth—that software prefetching utilizes more effectively (Section 5.1).

The rest of this paper is organized as follows. Section 2 introduces MLIR's sparse tensor dialect through a motivational example, describes coordinate hierarchy trees, their memory storage, and the result of the sparsification transformation. Section 3 presents ASaP and explains how we detect indirect memory accesses and generate prefetch instructions. Section 4 outlines our evaluation setup. Section 5 analyzes SpMV and SpMM performance, and compares with prior work. Section 6 covers related efforts, and Section 7 concludes.

2 Background

The Multi-Level Intermediate Representation (MLIR) enables modular domain-specific compiler design through extensible IR levels called *dialects* [39]. Each dialect groups related types, operations, and attributes. Figure 1a illustrates this organization, showing operations from both `arith` and `linalg` dialects along with their associated types and attributes. MLIR's **sparse tensor** dialect treats sparse tensor types as first-class citizens. Following the MT1 Sparse Compiler approach [9], the dialect treats sparsity as a property of an operand, allowing operations to be described independently of storage formats.

We demonstrate the sparse tensor dialect through a motivating example (Section 2.1) showing how annotations encode sparse formats. We then explain the coordinate hierarchy tree abstraction underlying these encodings (Section 2.2) and their segmented buffer representation (Section 2.3), revealing why indirect memory accesses emerge inevitably from sparse storage. Finally, we examine sparsification (Section 2.4), which transforms declarative tensor operations into imperative code operating on these buffer representations. ASaP extends this transformation to inject prefetches precisely when indirect accesses materialize during code generation. Readers familiar with MLIR's sparse tensor dialect may proceed directly to Section 3.1.

2.1 Sparse Matrix-Vector Multiplication

Sparse Matrix-Vector Multiplication (SpMV) computes $a_i = B_{ij} \cdot c_j$ in tensor index notation. Figure 1a shows how `linalg.generic` captures this computation declaratively. The operation defines a 2D iteration space over dimensions `i` and `j`, with affine maps (lines 1-3, 5) relating operand indices to induction variables. Dimension `j` is marked as `reduction` (line 6), and the computation body appears in `bb0` (lines 13-16).

This representation abstracts away imperative constructs—loops, conditionals, and side effects. Tensors remain immutable values, not yet lowered to memory buffers. Crucially, tensor `B` carries a sparse annotation (`#format`, line 10) defined in Figure 1b. Removing this annotation yields dense multiplication, demonstrating how the sparse tensor dialect treats sparsity as a tensor property rather than an algorithmic concern.

2.2 Coordinate Hierarchy Trees

Sparse tensor formats share a common characteristic: hierarchical organization of coordinates. This hierarchy forms a *coordinate hierarchy tree* where levels correspond to tensor dimensions, nodes contain coordinate values, and root-to-leaf paths encode non-zero element coordinates. Figure 2 illustrates such trees for three formats: Coordinate List (COO), Compressed Sparse Row (CSR), and Doubly Compressed Sparse Row (DCSR). The sparse tensor dialect describes formats through mappings from tensor dimensions to tree levels, annotated with level properties. Figure 1b shows these attributes for COO, CSR, and DCSR formats.

COO stores non-zero values with explicit row and column coordinates. Its first dimension uses `compressed` and `non-unique` levels; only rows with non-zeros appear, repeated for each non-zero they contain. In Figure 2(a), row 0 appears twice (two non-zeros) while row 1 is absent (empty). The second dimension is `singleton`, giving each first-level node exactly one child.

CSR replaces COO's first dimension with a `dense` level, storing all row coordinates including empty ones. Figure 2(b) shows row 1 present despite containing no non-zeros. The second dimension

[1] Artifacts available at https://github.com/kromancer/asap

```
#m_B = affine_map<(i, j) -> (i, j)>
#m_c = affine_map<(i, j) -> (j)>
#m_a = affine_map<(i, j) -> (i)>
#attributes = {
  indexing_maps  = [#m_B, #m_c, #m_a],
  iterator_types = ["parallel", "reduction"],
  sorted = true
}
%res = linalg.generic #attributes
   ins(%B : tensor<3x3xf64, #format>,
       %c : tensor<3xf64>)
  outs(%a : tensor<3xf64>) {
    ^bb0(%in: f64, %in_0: f64, %out: f64):
      %1 = arith.mulf %in, %in_0 : f64
      %2 = arith.addf %out, %1 : f64
      linalg.yield %2 : f64
  } -> tensor<3xf64>
```

(a) SpMV as a `linalg.generic` operation.

```
#COO = #sparse_tensor.encoding<{
         map = (i, j) -> (
           i: compressed(nonunique),
           j: singleton)}>

#CSR = #sparse_tensor.encoding<{
         map = (i, j) -> (
           i: dense,
           j: compressed)}>

#DCSR = #sparse_tensor.encoding<{
          map = (i, j) -> (
            i: compressed,
            j: compressed)}>
```

(b) Alternatives for `#format`, encoding attributes for COO, CSR and DCSR

Figure 1: Sparse matrix-vector multiplication (SpMV) expressed as a `linalg.generic` operation.

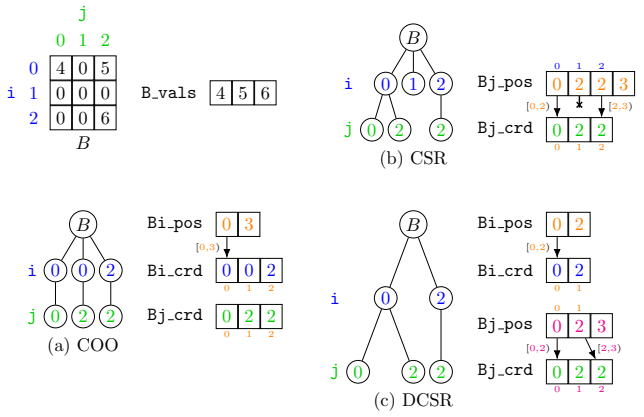

Figure 2: COO, CSR, and DCSR coordinate hierarchy trees for a 3×3 sparse matrix B with their associated in-memory buffers.

remains `compressed`, storing only columns with non-zeros. DCSR enhances CSR by compressing both dimensions. Unlike CSR's dense first level, DCSR eliminates empty rows entirely, as shown in Figure 2(c).

2.3 Storing Coordinate Hierarchy Trees

Figure 2 shows how coordinate hierarchy trees are serialized into memory buffers. Buffer requirements depend on level type: `dense` levels need no storage, while compressed levels use coordinate (`crd`) and position (`pos`) buffers. The `crd` buffer stores non-zero coordinates in a *segmented* layout; the `pos` buffer divides it into ranges, one per parent coordinate. Crucially, this segmentation introduces inherent indirection when accessing compressed levels.

Consider CSR in Figure 2(b). Buffer `Bj_crd` contains three segments for rows `i=0,1,2`, delimited by `Bj_pos`. Row 0's segment `[Bj_pos[0], Bj_pos[1])` stores columns `j=0,2`. Row 1's segment is empty (`Bj_pos[1] == Bj_pos[2]`). Row 2's segment `[Bj_pos[2], Bj_pos[3])` stores column `j=2`. DCSR in Figure 2(c) compresses both levels. Level `j` segments by parent coordinates `i=0,2` as in CSR. Level `i`, being topmost with only the root as parent, forms a single segment containing all non-empty rows.

Generally, for compressed level `l` of tensor `B`, the k^{th} segment spans `[Bl_pos[k], Bl_pos[k+1])` in `Bl_crd`. The position buffer has size `S+1` where `S` equals the parent level's node count, and `Bl_pos[S]` stores the total non-zero count at level `l`.

2.4 Sparsification

Sparsification lowers sparse tensor operations to imperative code operating directly on buffers. Figure 3 shows the sparsified SpMV from Figure 1a for three formats, using C-like syntax for clarity. The actual output uses MLIR's lower-level dialects: `scf` for control flow and `memref` for buffer operations.

Each listing traverses the coordinate hierarchy tree's leaf nodes, executing the computation from Figure 1a's `bb0` for non-zero values. The three formats differ in their loop structures: COO produces an imperfect loop nest requiring deduplication, while CSR and DCSR generate perfect loops. In COO (Figure 3a), the inner `while` (line 8) deduplicates non-unique row coordinates, creating segments for each unique row. Within segments, line 11 iterates over columns, accumulating products into the zero-initialized output. CSR (Figure 3b) yields a perfect loop nest: the outer loop traverses all rows including empty ones, while the inner loop processes each row's non-zeros. This explicit row representation eliminates deduplication overhead. DCSR (Figure 3c) combines benefits of both formats: like COO, it skips empty rows (retrieved at line 4), but like CSR, it avoids deduplication through unique row storage.

3 ASaP: Automatic Software Prefetching

ASaP extends MLIR's sparse tensor dialect with software prefetching for indirect accesses generated during sparsification. As demonstrated in Section 2.3, indirection is inherent to sparse tensor storage. Section 3.1 presents our analysis of when sparsification generates these indirect accesses, solving the challenge of determining precise injection sites for prefetch instructions without the need for post-hoc detection. Section 3.2 then details ASaP's prefetch generation scheme, which follows Ainsworth & Jones's approach but differs critically in bound determination for fault avoidance: ASaP leverages sparsification-time knowledge to extract accurate buffer bounds rather than inferring them from loop bounds, avoiding underestimation that would reduce prefetch coverage.

```
index ii = Bi_pos[0];
// Iterate over the non-empty rows of B
while (ii < Bi_pos[1]) {
  // Get the row coordinate
  index i = Bi_crd[ii];
  // Count duplicate row coordinates
  index segment_end = ii + 1;
  while (segment_end < Bi_pos[1] && Bi_crd[segment_end] == i)
    segend++;
  // For every non-zero of B in row i
  for (index jj = ii; jj < segment_end; jj++) {
    // Locate corresponding value in c
    f64 c_val = c[Bi_crd[jj]];
    a[i] += B_vals[jj] * c_val;
  }
  // Move to the next row's segment
  ii = segment_end;
}
```

(a) Imperative code for COO format

```
// Iterate over all rows of B
for (index i = 0; i < num_of_rows; i++) {
 // For every non-zero of B in row i
 for (index jj = Bj_pos[i]; jj < Bj_pos[i + 1]; jj++) {
   // Locate corresponding value in c
   f64 c_val = c[Bj_crd[jj]];
   a[i] += B_vals[jj] * c_val;
  }
}
```

(b) Imperative code for CSR format

```
// For every non-empty row of B
for (index ii = Bi_pos[0]; ii < Bi_pos[1]; ii++) {
 // Get the row index
 index i = Bi_crd[ii];
 // For every non-zero of B in row i
 for (index jj = Bj_pos[ii]; jj < Bj_pos[ii + 1]; jj++) {
   // Locate corresponding value in c
   f64 c_val = c[Bj_crd[jj]];
   a[i] += B_vals[jj] * c_val;
  }
}
```

(c) Imperative code for DCSR format

Figure 3: Sparsified version of Figure 1a

3.1 Detecting Indirect Accesses During Sparsification

Throughout Section 2, we have motivated our discussion of MLIR's sparsification transformation using the SpMV example presented in Figure 1a. This transformation builds on sparse iteration theory [22], first implemented in the Tensor Algebra Compiler (TACO) [20]. When sparse tensor `B` uses the DCSR format shown in Figure 2(c), the compiler generates the code in Figure 3c. Our analysis identifies when the indirect access `c[Bj_crd[jj]]` (line 8) emerges during compilation. Figure 4 shows a simplified representation of the elaboration stages during sparsification.

Sparsification begins by establishing iteration order across operand dimensions `i` and `j`. Since sparse tensors form coordinate hierarchy trees, iteration must preserve this hierarchical structure. The compiler captures dependencies using an *iteration graph* [20] (Figure 4a), where nodes represent operand dimensions and edges encode hierarchy constraints. The compiler constructs this graph by examining `affine_maps` (Figure 1a, lines 1-3) and respecting the `sorted = true` constraint (line 7), which prohibits reordering.

After establishing iteration order, the compiler resolves coordinates for each operand-dimension pair (`a,B,c` and `i,j`). Segment *iterators*—compile-time constructs that generate loops and conditionals—accomplish this resolution. Starting from root node `i`, the compiler recognizes this dimension belongs only to `B` and emits iterator `ii` as a `for`-loop. This loop enumerates coordinates `i` of all non-empty rows within `B`'s single top-level segment (Figure 4b, top node). Dereferencing this iterator through `Bi_crd` yields level coordinate `i`.

Node `j` in Figure 4a receives two edges: one from `B`'s parent coordinate level and one from dense vector `c`. Multiple incoming edges necessitate *coiteration*, requiring the compiler to select a traversal strategy for both operands' coordinate levels. The compiler chooses between merge-based traversal, for sorted coordinates, and *iterate-and-locate*, when a level supports constant-time membership checks via *locate* operations. For this example, the compiler selects iterate-and-locate. It generates an iterator for the $\texttt{ii}^{\text{th}}$ segment of `B`'s `j`-level, which resolves coordinate `j` and performs lookups in vector `c` (Figure 4c, bottom node).

The indirect access `c[Bj_crd[jj]]` (Figure 3c, line 10) emerges precisely when the compiler selects iterate-and-locate. ASaP injects prefetch instructions at this sparsification stage to mitigate memory latency from this access pattern. All other memory accesses, despite apparent complexity, follow regular streaming patterns that hardware prefetchers handle efficiently.

3.2 Prefetch Generation

ASaP generates prefetch instructions for indirect memory accesses following Ainsworth and Jones's three-step approach [5, 6], but with a critical distinction in bound calculation that significantly improves prefetch coverage. While prefetch instructions cannot cause access faults, the intermediate load for `Bj_crd[jj+distance]`, where `distance` is the prefetch lookahead offset in iterations, can. A suitable bound must be determined for fault avoidance. While prior approaches infer bounds from loop limits, ASaP leverages its knowledge of the sparse tensor format to determine actual buffer sizes, avoiding underestimation that reduces performance. Figure 5 illustrates our prefetch generation for `c[Bj_crd[jj]]`.

3.2.1 Step 1: Prefetching for `Bj_crd[jj+distance]`. Step 1 (lines 2-3) prefetches `Bj_crd[jj+2*distance]` to ensure the coordinate value is in cache when needed by Step 2.
Although `Bj_crd[jj+distance]` exhibits streaming behavior, hardware prefetchers often fail to detect this pattern. Off-core prefetchers monitoring higher-level cache misses lack program counter visibility and see conflicting access patterns: the same base address `Bj_crd` accessed at both offset `jj` (for coordinate resolution) and offset `jj+distance` (for prefetch operand loading). Even program-counter-aware prefetchers like L1_IPP [2] have limited tracking capacity—our evaluation platform sustains only two concurrent streams, insufficient for SpMV's multiple access streams. Omitting this step consistently degraded performance in our experiments.

3.2.2 Step 2: Loading `Bj_crd[jj+distance]`. Step 2 (lines 13-15) loads `Bj_crd[jj+distance]` with conditional bounds checking to prevent access faults. The choice of the bound can significantly impact performance by influencing the number of prefetches issued at

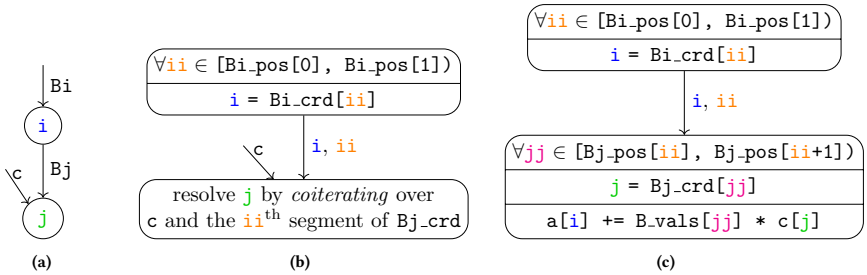

Figure 4: The iteration graph of SpMV at several stages of elaboration during the sparsification transformation. The initial iteration graph (a) depicts the order of iteration that follows the coordinate hierarchy tree of matrix B. In (b) the information that levels Bi and Bj are compressed is incorporated. Finally, (c) incorporates the information that vector c is dense, and an "iterate-and-locate" coiteration strategy is chosen. The color coding of the indices matches that of Figure 2.

```
1  // 1. Prefetch Bj_crd[jj + 2 * distance]
2  %0 = arith.addi %jj, %double_distance : index
3  memref.prefetch %Bj_crd[%0], read, locality<2>, data
4
5  // 2. Load Bj_crd[ min(jj + distance, bound) ]
6  // a. Determine bound as the size of %Bj_crd - 1
7  //    These instructions are loop-invariant and will be hoisted up
8  %1 = memref.load %Bi_pos[%c1] : memref<?xindex>
9  %2 = memref.load %Bj_pos[%1] : memref<?xindex>
10 %bound = arith.subi %2, %c1 : index
11
12 // b. Determine min(jj + distance, bound)
13 %2 = arith.addi %jj, %distance : index
14 %3 = arith.cmpi ult, %2, %bound : index
15 %4 = arith.select %3, %2, %bound : index
16
17 // c. Load Bj_crd[ min(jj + distance, bound) ]
18 %5 = memref.load %Bj_crd[%4] : memref<?xindex>
19
20 // 3. Prefetch c[Bj_crd[jj + distance]]
21 memref.prefetch %c[%5], read, locality<2>, data
```

Figure 5: Prefetching for `c[Bj_crd[jj]]` as seen in Figure 3a line 15, Figure 3b line 8 or Figure 3c line 10

runtime. Prior approaches derive bounds through use-def analysis on low-level IR, typically using the loop induction variable's upper limit. For our example, [5, 6] would bound prefetches by the `jj` iterator's limit, which corresponds to the number of coordinates in segment `ii`, as explained in lines 8-10 of page 8 in [6].

ASaP recognizes that `jj` iterates through the entire buffer `Bj_crd` segment-by-segment (Figure 4c). Rather than limiting bounds to individual segments, ASaP uses the total buffer size. This distinction yields $S \cdot$ `distance` additional prefetches, where S is the number of segments at level l. Prior approaches miss the first `distance` elements in each segment because prefetching starts from the `distance`-th element. ASaP's approach enables prefetching during the last `distance` iterations of segment `ii-1` to cover the first `distance` elements of segment `ii`, achieving better memory operation overlap. This improvement proves particularly significant for sparse matrices with short segments (yielding short inner loops approaching the prefetch distance), common in graph adjacency matrices with low-degree vertices. Section 5.3 demonstrates this advantage experimentally.

In the general case, where all levels of a sparse tensor are compressed, as in the Compressed Sparse Fiber (CSF) format [34], determining the size of `Bj_crd` at compile time is not feasible. Instead, its size can be determined at runtime through a chain of indirect loads, as shown in lines 8–9 of Figure 5. Specifically, the size of `Bj_crd` corresponds to the last segment boundary, stored as the final element of the position buffer `Bj_pos`. Similarly, the size of `Bj_pos`—representing the number of segments at level `j`— is stored in the final element of `Bi_pos`. Since level `i` is the topmost level in the coordinate hierarchy, its size is stored in `Bi_pos[1]`.

ASaP determines the size of a coordinate buffer at an arbitrary level l_k using the following recursive formula:

$$\text{crd\_buf\_sz}(l_k) = \begin{cases} l_1\_pos[1], & k = 1 \\ l_k\_pos[\text{crd\_buf\_sz}(l_{k-1})], & \text{otherwise} \end{cases}$$

3.2.3 Step 3: Prefetching for `c[Bj_crd[jj]]`. The final step prefetches the address `c[Bj_crd[jj+distance]]`, where `distance` represents an offset in iterations relative to the current iteration. In other words, the prefetch anticipates the memory access to `c[Bj_crd[jj]]` `distance` iterations in advance. In practice, one picks `distance` large enough to cover the expected memory latency relative to the per-iteration work, but not so large that prefetches arrive too early and pollute the cache. ASaP leaves `distance` as a user- or profile-tunable parameter rather than fixing it at compile time.

4 Methodology

This section describes our experimental setup for evaluating ASaP. We first detail our target hardware platform and configuration of Intel Alder Lake E-cores (Section 4.1). We then describe our benchmark selection from the SuiteSparse collection for SpMV and SpMM kernels (Section 4.2). Next, we explain the three implementation variants compared in our evaluation (Section 4.3). Finally, we outline our performance measurement procedures and metrics (Section 4.4).

4.1 Experimental Platform

Our experiments target the Intel Alder Lake Core i9-12900K processor [33], specifically its efficiency cores (E-cores) based on the Gracemont microarchitecture. We focus on E-cores because software prefetching demonstrates greater benefits on in-order and small out-of-order cores [6], and these cores provide fine-grained documentation and control over hardware prefetchers [2]. This

control allows us to disable hardware prefetchers that typically produce inaccurate prefetches for irregular sparse accesses, thereby reserving Miss Status Holding Registers (MSHRs) and memory bandwidth for software prefetching. Table 1 summarizes our experimental platform.

Table 1: System configuration for the Intel Alder Lake Core i9-12900K experiments.

CPU Model	Intel Core i9-12900K (Alder Lake)
Microarchitecture (E-cores)	Gracemont
Number of E-cores	8, arranged in 2 clusters cores in each cluster share an L2 cache
E-Core Frequency	2.4 GHz, fixed using Intel's `pstate` scaling driver
L1 / L2 (E-cores)	32 KB / 2 MB per cluster (non-inclusive)
L3 Cache Size (System)	30 MB (inclusive)
Main Memory	128 GB DDR5 @ 4800 MT/s Dual Channel Mode
OS	Linux Kernel 6.8.0-52, intel-microcode 0x26, Ubuntu 22.04

Intel Alder Lake E-cores provide multiple hardware prefetchers configurable via Model Specific Registers (MSRs). Table 2 details each prefetcher's functionality and configuration. For irregular sparse access patterns, disabling specific prefetchers (L1 NLP, L2 NLP) reduces MSHR contention and bandwidth waste from inaccurate requests. Prefetchers that benefit regular streaming accesses (L1 IPP, MLC Streamer, LLC Streamer) remain enabled. The L2 AMP prefetcher requires workload-specific configuration: enabled for SpMM to exploit 2D stride detection, disabled for SpMV to prevent excessive misprediction.

Our configuration decisions are empirically validated through performance measurements. Figure 7 shows SpMV performance across matrix families, comparing baseline and ASaP implementations with both default and optimized prefetcher settings. The optimized configuration (L1 NLP and L2 AMP disabled) consistently improves ASaP performance, while showing negligible impact on the baseline. This demonstrates that while hardware prefetchers may generate some accurate requests, their net effect is negative when combined with software prefetching due to resource contention.

4.2 Benchmarks

We evaluate ASaP on two fundamental sparse BLAS kernels:

(1) **Sparse Matrix-Vector Multiplication (SpMV):** The operation detailed in previous sections, fundamental to scientific computing and machine learning applications.
(2) **Sparse Matrix-Dense Matrix Multiplication (SpMM):** Extends SpMV by replacing the vector with a dense matrix, significantly increasing memory footprint and access complexity.

We source sparse matrices from the non-complex portion of the SuiteSparse Matrix Collection [15], comprising 2,839 matrices with diverse dimensions, sparsity patterns, and memory footprints. The collection spans from trivially small matrices (e.g., 3×3) to massive datasets—the largest (AGATHA-2015) requires 1.4 GB for positions, 44 GB for coordinates, and 5.4 GB for values in CSC/CSR format, with even its dense SpMV vector approaching 175 MB.

For stable performance evaluation, we focus on the largest matrices by memory footprint: the top 5% for SpMV and top 10% for SpMM. This selection addresses a critical challenge in sparse matrix benchmarking—the SuiteSparse collection contains predominantly small matrices where input vectors for SpMV fit entirely within L1 cache (50.3% of matrices), L2 cache (91.3%), or L3 cache (98.7%). These cache-resident matrices yield highly variable performance measurements that would distort evaluation metrics when weighted equally with larger matrices. Our selected subsets achieve coefficient of variation below 1%, enabling meaningful performance comparisons while representing workloads that stress memory hierarchy beyond cache capacity.

We use 32-bit integer indices when non-zero counts permit, otherwise 64-bit indices. Binary matrices store single-byte values with

Table 2: Overview of data prefetchers on Alder Lake E-cores (Gracemont). "Default On" or "Default Off" refer to the out-of-box processor state.

Prefetcher	Description	Setting
L1 NLP	L1 Next-Line Prefetcher. On a miss, fetches the next cache line. In the presence of irregular accesses, it frequently produces inaccurate prefetches, and empirical results show better performance with it turned off.	***Off*** (Default On)
L1 IPP	L1 Instruction Pointer Prefetcher. Tracks regular strides on individual load instructions (capacity observed to be 2 concurrent load streams).	On (Default On)
L2 NLP	L2 Next-Line Prefetcher, functionally similar to L1 NLP but at the L2 level.	Off (Default Off)
MLC Streamer	Mid-Level Cache (L2) streamer for instructions and data, tracking streaming accesses from L1 to L2 queue.	On (Default On)
L2 AMP	L2 Adaptive Multipath Prefetcher. Detects complex access patterns and helps in 2D strides (e.g., SpMM). But in SpMV, it generates inaccurate prefetches and adds bandwidth pressure.	***Selective*** (Default On)
LLC Streamer	Last-Level Cache (L3) streamer for both instruction and data, analogous to the MLC streamer but at the L3 level.	On (Default On)

boolean operations (`arith.ori` for addition, `arith.andi` for multiplication). All other matrices use 64-bit floating-point arithmetic.

4.3 Implementations and Compilation Flow

We compare the following three versions of each kernel, while configuring hardware prefetchers uniformly according to Table 2:

(1) **Baseline (no software prefetching):** Implemented in MLIR using the `linalg` dialect, with sparse tensor annotations in the `sparse_tensor` dialect. No software prefetching is performed.
(2) **ASaP (our approach):** The kernels are specified in MLIR as shown in Figure 1a, using the `linalg` dialect with sparse tensor annotations. We then enable our proposed software prefetching strategy for indirect memory accesses, making ASaP otherwise identical to the baseline version.
(3) **Ainsworth & Jones (low-level compiler pass):** Implemented using the TACO compiler [20] for high-level tensor index notation. We then apply the prefetching pass proposed by Ainsworth and Jones [5] on the generated `C` code at the LLVM IR level. For a fair comparison, we manually translate the resulting LLVM IR into MLIR (`scf` and `memref` dialects) so that all versions share the same final compilation flow to native code.

All final binaries are compiled at `-O3` optimization level using the LLVM backend. Parallel implementations use the sparse tensor dialect's `sparsifier` pipeline with the `parallelization-strategy` set to `dense-outer-loop`, which automatically emits OpenMP directives for outermost dense loops. Both prefetching variants use a fixed prefetch distance of 45 and the same locality hint (`locality<2>`).

4.4 Performance Measurements

We record the following during execution:

(1) **L2 Cache Misses**: For Gracemont E-cores, we approximate L2 misses by summing the following Perfomance Monitoring Unit (PMU) events: `MEM_LOAD_UOPS_RETIRED.DRAM_HIT` and `MEM_LOAD_UOPS_RETIRED.L3_HIT`.
(2) **Instruction Count**: Measured via the `INST_RETIRED.ANY` PMU event.
(3) **Execution Time**: Captured using high-precision timestamps from `clock_gettime(CLOCK_MONOTONIC)`.

To minimize measurement noise, we follow best-practice benchmarking guidelines as outlined in LLVM's benchmarking documentation [1]. We employ static core isolation by reserving all E-cores at boot time. Task affinity is explicitly set using `taskset`, and we optionally reinforce core isolation at runtime using `cset`.

Before each run, we flush hardware caches and the kernel's page cache to ensure a cold-cache starting state. We also preload all input data into main memory to eliminate page faults and prevent disk I/O from affecting measurements. Memory placement is configured with 2 MB huge pages for sparse matrix storage, and 1 GB pages for dense input operands to reduce TLB pressure from irregular accesses.

5 Experimental Results

Our primary metric is *throughput*, defined as the number of nonzero (nnz) elements processed per millisecond. We treat nnz as an asymptotic measure of work, capturing the total amount of useful computation performed by the kernel. We deliberately avoid normalizing by instruction count, as our prefetching scheme introduces additional instructions that do not perform useful work directly.

Using throughput as our work metric, we compute an *Equal-Work harmonic mean Speedup (EWS)* [17]: Throughputs are summarized using a harmonic mean, and speedup is then defined as a ratio of harmonic means. An implicit assumption underpinning EWS is that one conceptually weighs the amount of work done for each input matrix equally. By contrast, the geometric mean fails to capture this assumption and, as argued in [17], is essentially meaningless for comparing workloads that differ significantly in overall problem size.

The rest of this section is organized as follows: Section 5.1 analyzes ASaP on SpMV workloads, Section 5.2 extends this to SpMM kernels, and Section 5.3 compares ASaP against the prior work by Ainsworth & Jones.

5.1 ASaP on SpMV

Figure 6 shows that performance correlates strongly with memory pressure, as measured by L2 MPKI. For compute-bound matrices (low MPKI), ASaP can incur up to a 10% slowdown ($x = 0 \Rightarrow y \approx 0.9$) due to its instruction overhead. However, for memory-bound workloads, benefits increase linearly, with a break-even point near 4 L2 MPKI ($y = 1 \Rightarrow x \approx 4$), and speedups can exceed 2× when L2 MPKI approaches 50.

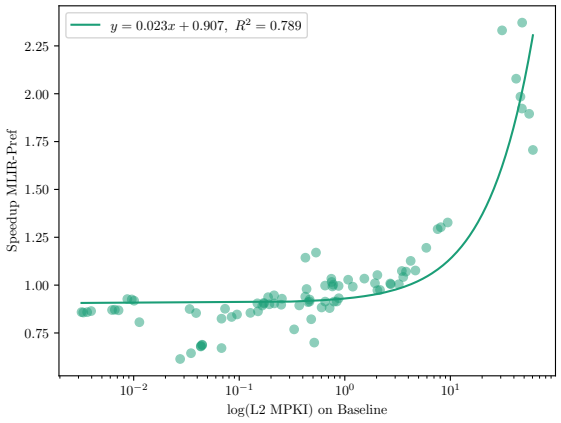

Figure 6: SpMV speedup (ASaP vs baseline) versus L2 MPKI for the largest 5% of matrices (single-threaded). Color intensity reflects point density: darker green regions indicate clusters of matrices with similar characteristics.

Figure 7 presents EWS speedups across matrix families, where configurations marked "-default" use default hardware prefetcher settings from Table 2 while unmarked configurations employ our

optimized settings (L1 NLP and L2 AMP disabled). ASaP achieves an average speedup of 1.42× for unstructured sparsity groups (Selected), where irregular access patterns defeat hardware prefetchers. The optimized prefetcher configuration consistently improves ASaP's performance over default settings, demonstrating that disabling inappropriate prefetchers amplifies software prefetching benefits. In contrast, the baseline benefits from the default hardware prefetcher configuration except for DIMACS10 where it is slightly slower. Mostly structured matrices (Others) show regression (0.8×) under ASaP, reflecting their lower memory-bound characteristics and effective utilization of hardware prefetching.

Insight: Compiler-controlled hardware prefetcher configuration could amplify software prefetching benefits, suggesting future architectures should expose prefetcher control as a first-class optimization interface.

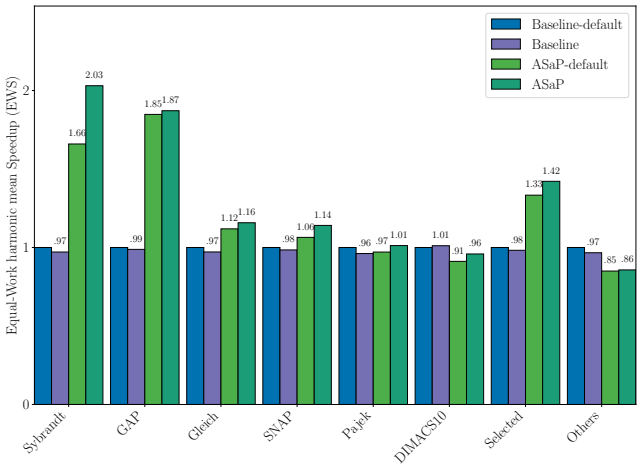

Figure 7: Equal-Work harmonic mean speedup (EWS) for SpMV across matrix groups (single-threaded). Configurations marked "-default" use default hardware prefetcher settings from Table 2, while unmarked configurations employ optimized settings (L1 NLP and L2 AMP disabled). "Selected" aggregates the first six groups while "Others" includes all remaining matrices not in those groups.

5.2 ASaP on SpMM

While our work metric is computed per non-zero element, this abstraction naturally extends from SpMV to SpMM, so long as the number of columns in the dense input remains small and bounded. To this end, the dense matrices used in our evaluation have as many columns as needed so that each row fits in a single cache line: 64 columns for binary matrices and 8 columns otherwise. Under these conditions, SpMM exhibits the same asymptotic behavior as SpMV, with total work remaining proportional to the number of non-zero elements in the sparse matrix.

Figure 8 presents SpMM results using the same scatter plot format as Figure 6, with ASaP speedup versus L2 MPKI. The linear relationship ($y = 0.706x + 0.995$, $R^2 = 0.776$) shows strong correlation between memory pressure and prefetching benefits, with

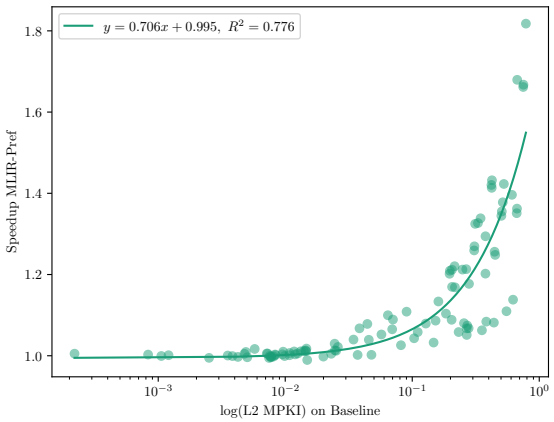

Figure 8: SpMM speedup (ASaP vs baseline) versus L2 MPKI for the top 10% of matrices (single-threaded).

```
1  // Iterate over all rows of B
2  for (index_t i = 0; i < M; i++) {
3    // For every non-zero of B in row i
4    for (index_t jj = Bj_pos[i]; jj < Bj_pos[i + 1]; jj++) {
5      index_t j = Bj_crd[jj];
6      // Opportunity to prefetch the first cache line
7      // of the next row in C: C[(j + 1) * N]
8      // For every column of C in row j
9      for (index_t k = 0; k < N; k++) {
10       A[i * N + k] += B[jj] * C[j * N + k];
11     }
12   }
13 }
```

Figure 9: Implementation of SpMM as a linear combination of rows using CSR format.

a much steeper slope than SpMV's 0.023x coefficient—indicating SpMM achieves higher speedups at lower memory pressure. ASaP overhead is negligible in compute-bound cases, with the regression line starting near 1.0. Compared to SpMV, where prefetching is performed in the innermost loop, SpMM benefits from generating prefetches in an outer (middle) loop, where instruction overhead is more easily amortized. Figure 9 shows this outer loop structure, where ASaP prefetches the first cache line of the next row in matrix C (line 6). Figure 10 shows an average speedup of 1.28× for sparse matrices from unstructured domains, and 1.02× for the rest. These gains, while smaller than for SpMV, confirm that outer-loop prefetching remains effective under constrained overhead. The gains from the optimized hardware prefetcher configuration (disabling L1 NLP) are negligible for SpMM and thus omitted from Figure 10.

Insight: Software prefetching schemes that understand tensor storage formats can extend beyond innermost loops to outer-loop prefetching.

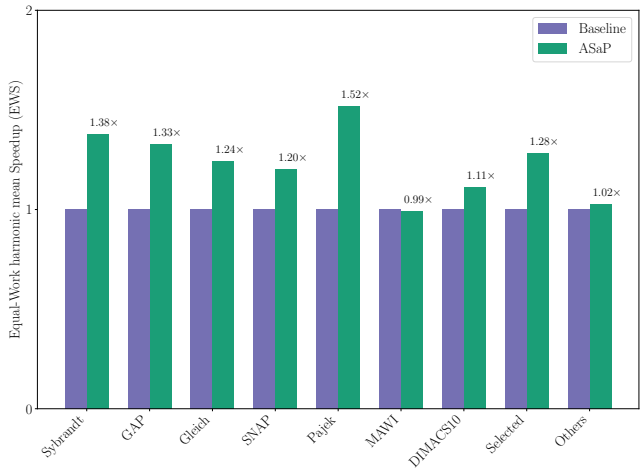

Figure 10: Equal-Work harmonic mean speedup (EWS) for SpMM across matrix groups (single-threaded). "Selected" aggregates the first seven groups while "Others" includes all remaining matrices not in those groups.

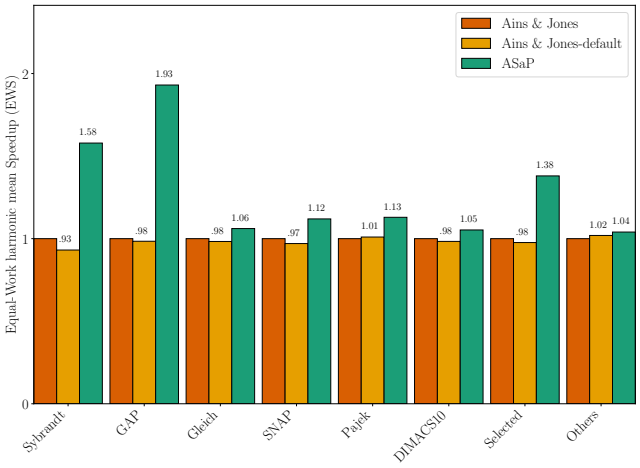

Figure 11: Equal-Work harmonic mean speedup (EWS) for SpMV across matrix groups (single-threaded). Configurations marked "-default" use default hardware prefetcher settings from Table 2, while unmarked configurations employ optimized settings (L1 NLP and L2 AMP disabled). "Selected" aggregates the first six groups while "Others" includes all remaining matrices not in those groups.

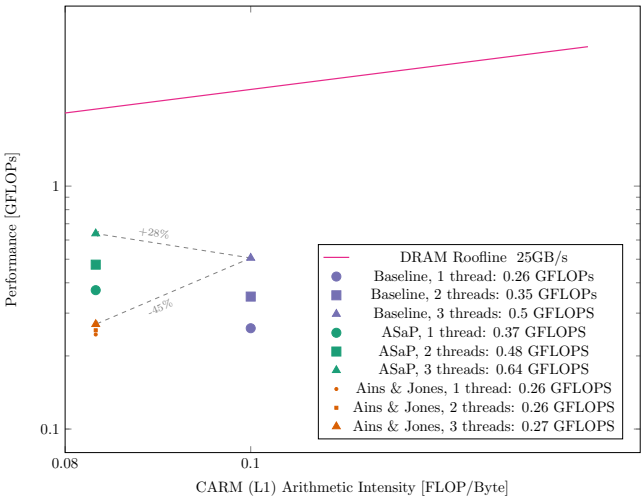

Figure 12: Roofline model showcasing the impact of software prefetching on SpMV performance for the GAP-twitter matrix on a multi-threaded scenario.

5.3 ASaP vs Ainsworth & Jones

We now compare ASaP to the prior state-of-the-art method [5, 6]. We note that the publicly available artifact[2] for Ainsworth & Jones' method would not generate prefetches for SpMM, limiting our comparison to SpMV. Figure 11 shows ASaP achieves an average SpMV speedup of 1.38× on unstructured matrix groups (Selected) and 1.04× on the remainder. The optimized hardware prefetcher configuration also improves Ainsworth and Jones' method marginally (1.02× for Selected), though far less than ASaP's gains. This performance gain stems from ASaP's ability to handle matrices with short inner loops, a common characteristic in sparse workloads where segment sizes approach the prefetch distance. While the prior method struggles when loop trip counts near the prefetch distance, ASaP's bound calculation for fault avoidance in the intermediate load (Section 3.2.2), maintains effective prefetching across segment boundaries.

Figure 12 visualizes performance within a cache-aware roofline model [18, 41]. ASaP consistently outperforms the baseline, with peak gains (28%) at 3 threads. The slight leftward shift in arithmetic intensity reflects increased memory operations from prefetching—a tradeoff that improves DRAM utilization. The GAP-twitter matrix exemplifies ASaP's advantage: its adjacency lists often contain fewer neighbors than the prefetch lookahead, causing the prior method to underperform significantly.

Insight: Domain-specific semantic knowledge enables more accurate prefetch bounds than generic low-level analysis, yielding 1.38× speedup over prior art on unstructured matrices.

6 Related Work

Software Prefetching. Software prefetching has a long history, dating back to the seminal works of Callahan et al. [12] and Mowry et al. [29]. More recent efforts have tackled the harder problem of indirect memory accesses, with the state-of-the-art being the compiler pass by Ainsworth and Jones [5, 6], which we build upon and improve in this work. Mehta et al. [28] extend prefetching to high-bandwidth memory systems like A64FX, which can tolerate large numbers of outstanding misses thanks to a high MLP.

Profile-guided approaches offer an orthogonal direction. APT-GET [19] dynamically profiles workloads to determine when and where to inject prefetches, including selecting optimal distances. RPG^2 [44] extends this idea by rolling back prefetches when they

[2]https://github.com/SamAinsworth/reproduce-tocs2019-paper

degrade performance. While our proposal does not rely on profiling, it could benefit from these strategies, particularly for tuning distances and adaptively disabling prefetches under low memory pressure.

In contrast to prior art that implements prefetching at low-level IR such as LLVM, we are the first to implement software prefetching in an MLIR framework, leveraging high-level semantic information from the sparse tensor dialect to generate more effective prefetches.

Sparse Compilation. The difficulty of writing high-performance sparse code has long been recognized. Early work by Bik and Wijshoff [10] pioneered the automatic generation of sparse code from dense formulations, a line of thinking that would become foundational in sparse compilation. Pugh and Shpeisman [32] introduced the Sparse Intermediate Program Representation (SIPR), an early attempt to separate sparsity concerns from computation semantics—conceptually similar to what modern sparse tensor dialects aim to achieve. Kotlyar et al. [23] viewed sparse data structures as relations and computations over them as relational query evaluation. The Sparse Polyhedral Framework (SPF) [35, 36, 40, 45] advanced this line of work by extending polyhedral compilation to support sparse iteration spaces via symbolic analysis and the composition of inspector/executor transformations.

More recently, TACO [20], introduced by Kjølstad et al., established sparse iteration theory as a foundation for sparse tensor compilation. Building on this foundation, Chou et al. [13] introduced format abstraction to generalize code generation across tensor formats, while subsequent work has focused on optimizing computations with sparse results through workspace-based scheduling transformations [4, 21, 43]. Also influenced by TACO, SparseTIR [42] explores composable formats, where different regions of a tensor are stored using layouts best suited to their local sparsity patterns.

Prior to the introduction of MLIR's `sparse_tensor` dialect, Mutlu et al. [30] and Tian et al. [38] proposed COMET, an MLIR-based compiler infrastructure that defined an alternative sparse tensor dialect. Compared to COMET, MLIR's `sparse_tensor` dialect integrates more naturally with frameworks like TensorFlow [3] and JAX [11]. Liu et al. [26] extended this dialect to support sparse tensor convolutions, broadening its applicability to machine learning workloads. To the best of our knowledge, none of the aforementioned systems target software prefetching as a compiler optimization.

To the best of our knowledge, we are the first to build upon MLIR's sparsification dialect to implement software prefetching specifically for sparse data structures. By operating at a higher abstraction level than traditional LLVM-based approaches, we can exploit semantic knowledge about sparse iteration patterns and tensor formats

7 Conclusions

We presented ASaP, the first software prefetching framework built into MLIR's sparse tensor dialect, which leverages sparse tensor semantics to determine accurate buffer bounds (Section 3.2.2), achieving performance gains by maintaining prefetch coverage across segment boundaries—particularly for matrices with short inner loops where prior approaches fail (Section 5.3). The effectiveness of both innermost-loop prefetching (Section 5.1) and outer-loop prefetching (Section 5.2) validates our semantic-driven approach: while [5, 6] cannot generate prefetches for SpMM's multi-dimensional accesses, ASaP successfully places prefetches in outer loops by understanding loop-tensor storage relationships.

Our experiments revealed three key insights: First, domain-specific semantic knowledge enables more accurate prefetch bounds than generic low-level analysis, yielding 1.38× speedup over prior art on unstructured matrices (Section 5.3). Second, software prefetching schemes that understand tensor storage formats can extend beyond innermost loops to outer-loop prefetching (Section 5.2). Third, compiler-controlled hardware prefetcher configuration could amplify software prefetching benefits, suggesting future architectures should expose prefetcher control as a first-class optimization interface (Section 5.1).

Acknowledgments

This work was partially supported by the Wallenberg AI, Autonomous Systems and Sotware Program (WASP) funded by the Knut and Alice Wallenberg Foundation.

References

[1] [n. d.]. Benchmarking tips — LLVM 21.0.0git documentation. https://llvm.org/docs/Benchmarking.html

[2] 2023. Whitepaper: Hardware Prefetch Controls for Intel Atom Cores. https://www.intel.com/content/www/us/en/content-details/795247/hardware-prefetch-controls-for-intel-atom-cores.html

[3] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: A system for large-scale machine learning. doi:10.48550/ARXIV.1605.08695 Version Number: 2.

[4] Willow Ahrens, Fredrik Kjolstad, and Saman Amarasinghe. 2022. Autoscheduling for sparse tensor algebra with an asymptotic cost model. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. ACM, 269–285. doi:10.1145/3519939.3523442

[5] Sam Ainsworth and Timothy M. Jones. 2017. Software prefetching for indirect memory accesses. In *Proceedings of the 2017 International Symposium on Code Generation and Optimization (CGO '17)*. IEEE Press, 305–317.

[6] Sam Ainsworth and Timothy M. Jones. 2018. Software Prefetching for Indirect Memory Accesses: A Microarchitectural Perspective. *ACM Transactions on Computer Systems* 36, 3 (Aug. 2018), 1–34. doi:10.1145/3319393

[7] Grant Ayers, Heiner Litz, Christos Kozyrakis, and Parthasarathy Ranganathan. 2020. Classifying Memory Access Patterns for Prefetching. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, Lausanne Switzerland, 513–526. doi:10.1145/3373376.3378498

[8] Abanti Basak, Shuangchen Li, Xing Hu, Sang Min Oh, Xinfeng Xie, Li Zhao, Xiaowei Jiang, and Yuan Xie. 2019. Analysis and Optimization of the Memory Hierarchy for Graph Processing Workloads. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, Washington, DC, USA, 373–386. doi:10.1109/HPCA.2019.00051

[9] Aart Johannes Casimir Bik. 1997. *Compiler support for sparse matrix computations*.

[10] Aart J. C. Bik and Harry A. G. Wijshoff. 1993. Compilation techniques for sparse matrix computations. In *Proceedings of the 7th international conference on Supercomputing (ICS '93)*. Association for Computing Machinery, 416–424. doi:10.1145/165939.166023

[11] 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/jax-ml/jax

[12] David Callahan, Ken Kennedy, and Allan Porterfield. 1991. Software prefetching. In *Proceedings of the fourth international conference on Architectural support for programming languages and operating systems - ASPLOS-IV*. ACM Press, 40–52. doi:10.1145/106972.106979

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

[14] Timothy A. Davis. 2019. Algorithm 1000: SuiteSparse:GraphBLAS: Graph Algorithms in the Language of Sparse Linear Algebra. *ACM Trans. Math. Software* 45, 4 (Dec. 2019), 1–25. doi:10.1145/3322125

[15] Timothy A. Davis and Yifan Hu. 2011. The university of Florida sparse matrix collection. *ACM Trans. Math. Software* 38, 1 (Nov. 2011), 1–25. doi:10.1145/2049662.2049663

[16] Jack Dongarra, Michael A. Heroux, and Piotr Luszczek. 2016. A new metric for ranking high-performance computing systems. *National Science Review* 3, 1 (March 2016), 30–35. doi:10.1093/nsr/nwv084

[17] Lieven Eeckhout. 2024. R.I.P. Geomean Speedup Use Equal-Work (Or Equal-Time) Harmonic Mean Speedup Instead. *IEEE Computer Architecture Letters* 23, 1 (Jan. 2024), 78–82. doi:10.1109/LCA.2024.3361925

[18] Aleksandar Ilic, Frederico Pratas, and Leonel Sousa. 2014. Cache-aware Roofline model: Upgrading the loft. *IEEE Computer Architecture Letters* 13, 1 (Jan. 2014), 21–24. doi:10.1109/L-CA.2013.6

[19] Saba Jamilan, Tanvir Ahmed Khan, Grant Ayers, Baris Kasikci, and Heiner Litz. 2022. APT-GET: profile-guided timely software prefetching. In *Proceedings of the Seventeenth European Conference on Computer Systems*. ACM, Rennes France, 747–764. doi:10.1145/3492321.3519583

[20] Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, David Lugato, and Saman Amarasinghe. 2017. The tensor algebra compiler. *Proceedings of the ACM on Programming Languages* 1, OOPSLA (Oct. 2017), 1–29. doi:10.1145/3133901

[21] Fredrik Kjølstad, Willow Ahrens, Shoaib Kamil, and Saman Amarasinghe. 2019. Tensor Algebra Compilation with Workspaces. In *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 180–192. doi:10.1109/CGO.2019.8661185

[22] Fredrik Berg Kjølstad. 2020. *Sparse Tensor Algebra Compilation*. Ph.D. Dissertation. Massachusetts Institute of Technology, Cambridge, MA.

[23] Vladimir Kotlyar, Keshav Pingali, and Paul Stodghill. 1997. A relational approach to the compilation of sparse matrix programs. In *Euro-Par'97 Parallel Processing*. Vol. 1300. Springer Berlin Heidelberg, Berlin, Heidelberg, 318–327. doi:10.1007/BFb0002751

[24] 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 *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, Seoul, Korea (South), 2–14. doi:10.1109/CGO51591.2021.9370308

[25] Jiajia Li, Mahesh Lakshminarasimhan, Xiaolong Wu, Ang Li, Catherine Olschanowsky, and Kevin Barker. 2020. A Sparse Tensor Benchmark Suite for CPUs and GPUs. In *2020 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, Beijing, China, 193–204. doi:10.1109/IISWC50251.2020.00027

[26] Peiming Liu, Alexander J Root, Anlun Xu, Yinying Li, Fredrik Kjolstad, and Aart J.C. Bik. 2024. Compiler Support for Sparse Tensor Convolutions. *Proceedings of the ACM on Programming Languages* 8, OOPSLA2 (Oct. 2024), 275–303. doi:10.1145/3689721

[27] Martin Paul Lücke, Oleksandr Zinenko, William S. Moses, Michel Steuwer, and Albert Cohen. 2025. The MLIR Transform Dialect: Your Compiler Is More Powerful Than You Think. In *Proceedings of the 23rd ACM/IEEE International Symposium on Code Generation and Optimization*. ACM, Las Vegas NV USA, 241–254. doi:10.1145/3696443.3708922

[28] Sanyam Mehta, Gary Elsesser, and Terry Greyzck. 2022. Software pre-execution for irregular memory accesses in the HBM era. In *Proceedings of the 31st ACM SIGPLAN International Conference on Compiler Construction*. ACM, Seoul South Korea, 231–242. doi:10.1145/3497776.3517783

[29] Todd C. Mowry, Monica S. Lam, and Anoop Gupta. 1992. Design and evaluation of a compiler algorithm for prefetching. *ACM SIGPLAN Notices* 27, 9 (Sept. 1992), 62–73. doi:10.1145/143371.143488

[30] Erdal Mutlu, Ruiqin Tian, Bin Ren, Sriram Krishnamoorthy, Roberto Gioiosa, Jacques Pienaar, and Gokcen Kestor. 2020. COMET: A Domain-Specific Compilation of High-Performance Computational Chemistry. In *Languages and Compilers for Parallel Computing: 33rd International Workshop, LCPC 2020, Virtual Event, October 14-16, 2020, Revised Selected Papers*. Springer-Verlag, 87–103. doi:10.1007/978-3-030-95953-1_7

[31] Maxim Naumov, Dheevatsa Mudigere, Hao-Jun Michael Shi, Jianyu Huang, Narayanan Sundaraman, Jongsoo Park, Xiaodong Wang, Udit Gupta, Carole-Jean Wu, Alisson G. Azzolini, Dmytro Dzhulgakov, Andrey Mallevich, Ilia Cherniavskii, Yinghai Lu, Raghuraman Krishnamoorthi, Ansha Yu, Volodymyr Kondratenko, Stephanie Pereira, Xianjie Chen, Wenlin Chen, Vijay Rao, Bill Jia, Liang Xiong, and Misha Smelyanskiy. 2019. Deep Learning Recommendation Model for Personalization and Recommendation Systems. doi:10.48550/ARXIV.1906.00091 Version Number: 1.

[32] William Pugh and Tatiana Shpeisman. 1998. SIPR: A New Framework for Generating Efficient Code for Sparse Matrix Computations. In *Proceedings of the 11th International Workshop on Languages and Compilers for Parallel Computing (LCPC '98)*. Springer-Verlag, 213–229.

[33] Efraim Rotem, Adi Yoaz, Lihu Rappoport, Stephen J. Robinson, Julius Yuli Mandelblat, Arik Gihon, Eliezer Weissmann, Rajshree Chabukswar, Vadim Basin, Russell Fenger, Monica Gupta, and Ahmad Yasin. 2022. Intel Alder Lake CPU Architectures. *IEEE Micro* 42, 3 (May 2022), 13–19. doi:10.1109/MM.2022.3164338

[34] 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*. ACM, Austin Texas, 1–7. doi:10.1145/2833179.2833183

[35] Michelle Mills Strout, Mary Hall, and Catherine Olschanowsky. 2018. The Sparse Polyhedral Framework: Composing Compiler-Generated Inspector-Executor Code. *Proc. IEEE* 106, 11 (Nov. 2018), 1921–1934. doi:10.1109/JPROC.2018.2857721

[36] Michelle Mills Strout, Alan LaMielle, Larry Carter, Jeanne Ferrante, Barbara Kreaseck, and Catherine Olschanowsky. 2016. An approach for code generation in the Sparse Polyhedral Framework. *Parallel Comput.* 53, C (April 2016), 32–57. doi:10.1016/j.parco.2016.02.004

[37] Nishil Talati, Kyle May, Armand Behroozi, Yichen Yang, Kuba Kaszyk, Christos Vasiladiotis, Tarunesh Verma, Lu Li, Brandon Nguyen, Jiawen Sun, John Magnus Morton, Agreen Ahmadi, Todd Austin, Michael O'Boyle, Scott Mahlke, Trevor Mudge, and Ronald Dreslinski. 2021. Prodigy: Improving the Memory Latency of Data-Indirect Irregular Workloads Using Hardware-Software Co-Design. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, Seoul, Korea (South), 654–667. doi:10.1109/HPCA51647.2021.00061

[38] Ruiqin Tian, Luanzheng Guo, Jiajia Li, Bin Ren, and Gokcen Kestor. 2021. A High Performance Sparse Tensor Algebra Compiler in MLIR. In *2021 IEEE/ACM 7th Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC)*. 27–38. doi:10.1109/LLVMHPC54804.2021.00009

[39] Nicolas Vasilache, Oleksandr Zinenko, Aart J. C. Bik, Mahesh Ravishankar, Thomas Raoux, Alexander Belyaev, Matthias Springer, Tobias Gysi, Diego Caballero, Stephan Herhut, Stella Laurenzo, and Albert Cohen. 2022. Composable and Modular Code Generation in MLIR: A Structured and Retargetable Approach to Tensor Compiler Construction. arXiv:2202.03293 [cs].

[40] Anand Venkat, Mary Hall, and Michelle Strout. 2015. Loop and data transformations for sparse matrix code. *ACM SIGPLAN Notices* 50, 6 (Aug. 2015), 521–532. doi:10.1145/2813885.2738003

[41] 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

[42] 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*. ACM, 660–678. doi:10.1145/3582016.3582047

[43] Genghan Zhang, Olivia Hsu, and Fredrik Kjolstad. 2024. Compilation of Modular and General Sparse Workspaces. *Proceedings of the ACM on Programming Languages* 8, PLDI (June 2024), 1213–1238. doi:10.1145/3656426

[44] Yuxuan Zhang, Nathan Sobotka, Soyoon Park, Saba Jamilan, Tanvir Ahmed Khan, Baris Kasikci, Gilles A Pokam, Heiner Litz, and Joseph Devietti. 2024. RPG^2: Robust Profile-Guided Runtime Prefetch Generation. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. ACM, 999–1013. doi:10.1145/3620665.3640396

[45] Tuowen Zhao, Tobi Popoola, Mary Hall, Catherine Olschanowsky, and Michelle Strout. 2023. Polyhedral Specification and Code Generation of Sparse Tensor Contraction with Co-iteration. *ACM Transactions on Architecture and Code Optimization* 20, 1 (March 2023), 1–26. doi:10.1145/3566054