THESIS

OPTIMIZING SPARSE COMPUTATIONS USING UNION OF Z-POLYHEDRA

Submitted by

Santoshkumar Tongli

Department of Computer Science

In partial fulfillment of the requirements

For the Degree of Master of Science

Colorado State University

Fort Collins, Colorado

Summer 2025

Master's Committee:

Advisor: Louis-Noël Pouchet

Shrideep Pallickara
Sudeep Pasricha

ABSTRACT

OPTIMIZING SPARSE COMPUTATIONS USING UNION OF Z-POLYHEDRA

Sparse matrices play a central role in a wide range of modern computational problems. They are especially common in domains such as scientific simulations, numerical methods, graph analytics, machine learning, and high-performance computing workloads, where data is often structured in a way that leads to a significant number of zero-valued elements. Instead of treating these zeros as meaningful data, sparse matrix techniques aim to exploit this sparsity to reduce both storage and computational cost, thereby improving scalability and efficiency. The Union of $\mathcal{Z}$-Polyhedra (UZP) sparse format models sparse structures as unions of integer polyhedra intersected with affine lattices, capturing both regular and irregular sparsity patterns in a unified form. Building on this abstraction, ur work introduces a suite of tuners that apply structural transformations to UZP representations without altering their mathematical semantics. These transformations improve data locality, Single Instruction Multiple Data (SIMD) vectorization, and parallelism, enabling performance tuning without modifying execution logic. Evaluated across 229 matrices from the SuiteSparse collection, the optimized UZP representations achieve highly competitive performance for sparse matrix-vector multiplication (SpMV) computations on multi-core CPUs, outperforming reference approaches such as Intel MKL's sparse implementation or formats dedicated to SIMD vectorization.

DEDICATION

I would like to dedicate this thesis to my Grandparents

TABLE OF CONTENTS

ABSTRACT ii
DEDICATION iii
LIST OF TABLES v
LIST OF FIGURES vi

Chapter 1 Introduction 1

Chapter 2 Prior work and Background 6
2.1 Sparse Matrices in Modern Computing and its Challenges 6
2.1.1 Related Work 7
2.1.2 Reconstructing Irregular Structures via $\mathcal{Z}$-Polyhedra 11
2.1.3 Custom Vector Code Generation for Fixed Sparse Structures 17
2.1.4 UZP Representation and its Background 19

Chapter 3 Understanding the Union of $\mathcal{Z}$-Polyhedra Format (UZP) 21
3.1 Design Principles of UZP 23
3.2 Implementation of UZP 24

Chapter 4 Tuning Mechanisms for UZP 29
4.1 Aggregator Tuner: Merging Regularly Strided Origin Points 30
4.1.1 Algorithm & Implementation Details 32
4.1.2 Impact Evaluation of Aggregator Tuner 32
4.2 Reordering Origins for Locality Optimization 37
4.2.1 Impact Evaluation of Favour Locality Tuner 39
4.3 Tile-Based Origin Reordering for Locality Optimization 42
4.4 Generic Executors for Polyhedral Sparse Representations 46
4.4.1 Design Principles of the Generic Executors 47
4.4.2 Further Optimization via Auto-Generated Shape-Specific Loops 49
4.4.3 Fine-Grained Loop Specialization Using Per-Origin Dispatch 52
4.4.4 Impact of Favour Locality Tuner with Macro-Specialized Executors . . 54

Chapter 5 Experiments and Results 57
5.1 Single-Threaded Performance 59
5.2 Parallel Scaling 65
5.3 Evaluation of UZP & TUNNERS Optimization Strategies 68

Chapter 6 Sparse Matrix Visualization Tool 77

Chapter 7 Conclusion and Future Work 79

Chapter 8 Personal Bibliography 81

LIST OF TABLES

2.1 Evolution of the number of pieces as a function of their maximal dimensionality (maxd) for matrix HB/nos1 (1,017 nonzero elements). 15

5.1 Comparison of Standard UZP and Aggregated UZP GFlops of SpMV operation on top 20 performing matrices 72

5.2 Top 10 Matrices with Significant GFlops Improvement across Aggregated UZP vs Favour Locality UZP (nnz > 50K) 74

LIST OF FIGURES

2.1 HB/can_1072: 1072×1072 matrix with 12,444 nonzeros 6
2.2 Figure showing a sample sparse matrix and its storage in (1) COO and (2) CSR formats. 8
2.3 Different sparse matrices from the HB group of the SuiteSparse Matrix Collection. Figure (a) shows an except of the accesses performed during SpMV of matrix `HB/nos1`. The nonzero elements in this matrix are shown in Figure (b), and a zoom of its main diagonal is provided in Figure (c). This is a 237×237 matrix with 1,017 nonzero elements, and its reconstructed code consists of a single statement inside an 8-dimensional loop. Figure (d) shows the nonzero elements in `HB/can_1072`, a $1,072 \times 1,072$ matrix with 12,444 nonzero elements, which does not exhibit any apparent regularity. Its reconstructed code includes 870 pieces of up to 8 dimensions. . 16

3.1 Different possible representations of the same sparse matrix: CSR, COO and two possible UZPs. 26

4.1 Schematic representation of tuner interaction pathways. 29
4.2 Performance comparison of Standard UZP vs Aggregator Tuner UZP across 229 sparse matrices ordered by non-zero count, highlighting gains from Aggregator tuner. 35
4.3 Plot showing Base shape count of Standard UZP vs Aggregator Tuner UZP across 229 sparse matrices ordered by non-zero. 36
4.4 Plot showing Origins count of Standard UZP vs Aggregator Tuner UZP across 229 sparse matrices ordered by non-zero. 36
4.5 Performance comparison of Standard UZP vs Favour Locality Tuner UZP across 229 sparse matrices sorted by increasing non-zero count, highlighting gains from locality-aware tuning. 39
4.6 Base shape count of Standard UZP vs Favour Locality Tuner UZP across 229 sparse matrices sorted by increasing non-zero count. 40
4.7 Origin count of Standard UZP vs Favour Locality Tuner UZP across 229 sparse matrices sorted by increasing non-zero count. 41
4.8 Illustrates average elapsed time for SpMV across Standard UZP and Tile Tuner UZP variants for matrices with nnz > 50K. 43
4.9 Illustrates average Gflops achieved for SpMV across Standard UZP and Tile Tuner UZP variants for matrices with nnz > 50K. 43
4.10 Illustrates average number of base shapes in SpMV across Standard UZP and Tile Tuner UZP variants for matrices with nnz > 50K. 45
4.11 Illustrates average number of origins in SpMV across Standard UZP and Tile Tuner UZP variants for matrices with nnz > 50K. 45
4.12 Performance comparison of SpMV operations on Standard UZP versus Favour Locality Tuned UZP with newgen/executor across 229 sparse matrices sorted by increasing non-zero count). 55

5.1 Sieve of the $2,754$ SuiteSparse matrices below 20M nonzeros. Selected matrices are marked with 'X's. 58

5.2 UZP generation time. 58
5.3 Performance (GFLOPS) of double-precision, cold cache kernels. MKL-BCSR in Fig. 6e shows the best performance for all block sizes between 2 and 32. 60
5.4 Performance (GFLOPS) of single-precision, cold cache kernels. 61
5.5 Joint performance (GFLOPS) plots of different kernel versions. 63
5.6 Scaling plots for cold cache executions with two and eight threads for matrices above 100K nonzeros. The plots show single-threaded (X axes) vs multi-threaded (Y axes) performance in GFLOPS. The solid lines show the linear regression models of the scaling. The dotted lines show the reference 2x/8x scalings. 67
5.7 Comparison of UZP variants under double-precision, cold-cache settings. Aggregation and favourdata tuners progressively improve performance over the non-aggregated baseline. Shuffling origin order reduces locality, leading to lower average throughput despite the same shape set. 68
5.8 Box plot comparing the normalized PAPI counter performance metrics for Standard Uzp versus Aggregated Uzp. 71
5.9 Stacked bar chart illustrating the relative space occupied by Standard UZP versus Aggregator UZP metadata size for multiple sparse matrices sorted in increasing order of non-zero elements. 73
5.10 Box plot comparing the normalized PAPI counter performance metrics for Aggregated Uzp versus Favour Locality Uzp. 75

6.1 Interactive sparse matrix visualization tool displaying a tiled view of the apache2.mtx matrix 78
6.2 Interactive sparse matrix visualization tool displaying a tiled view of the 3DSpectralwave2.mtx matrix. 78

Chapter 1

Introduction

Sparse computations sacrifice the structural regularity and optimization opportunities inherent in dense computations in order to reduce memory usage and computational cost by bypassing operations on zero-valued elements. While this may lead to a reduction in raw throughput, the overall efficiency is often improved due to the substantial decrease in the total number of executed operations.

Sparse computations depend on specialized data structures designed to capture arbitrary distributions of nonzero elements, inherently introducing irregularity in contrast to the structured loop nests and memory accesses typical of dense computations. Widely used storage formats—such as Compressed Sparse Row (CSR), Compressed Sparse Column (CSC), Coordinate (COO), and Blocked Compressed Sparse Row (BCSR)—explicitly encode the positions of nonzero entries, facilitating general-purpose representations of sparsity patterns without assuming any inherent regularity. However, the use of these formats necessitates the development of format-specific executors [19], which in turn must be carefully optimized to exploit the architectural features of target hardware platforms [37, 4, 29].

In contrast, dense computations benefit from regular loop structures and affine memory accesses, which enable a wide range of compiler optimizations, including loop tiling, parallelization [5], and effective Single Instruction Multiple Data (SIMD) vectorization [18]. A mature ecosystem of optimization frameworks has evolved to support dense linear algebra across diverse hardware architectures, frequently leveraging polyhedral compilation techniques that represent computations and data accesses as affine transformations. Recent advancements—such as the introduction of the Affine dialect in MLIR [21]—have further streamlined the specification and optimization of dense computations, building on the foundational principles established by earlier polyhedral frameworks [35, 5].

Efforts to enhance the performance of sparse computations span a broad spectrum of approaches. At one end are Inspector/Executor (I/E) frameworks, which operate over general-purpose sparse formats [28, 31]. These methods involve analyzing and restructuring the sparsity pattern—through partitioning, reordering, or other transformations—prior to execution in order to generate a more optimized executor. At the other end are sparsity-specialized techniques that generate code at compile time for a known set of nonzero coordinates. By eliminating indirection arrays, these approaches enable highly efficient SIMD vectorization [2, 18, 39, 11]. Between these two extremes lie hybrid solutions that leverage constrained structural regularity. Formats such as Diagonal (DIA) and Blocked Compressed Sparse Row (BCSR) strike a balance by partially mitigating the overheads inherent to fully general, structure-agnostic formats like CSR and COO.

A longstanding challenge in sparse computation lies in determining the optimal combination of storage format and executor organization that yields high performance for a given sparsity structure, particularly when targeting specific hardware architectures. In this work, we introduce an alternative strategy that bridges the gap between sparse and dense computation models, along with their respective optimization techniques.

Inspired by recent advances in reconstructing sparse sequences of integer tuples as unions of polyhedra and lattices [2, 27], we present a tunable and unified sparse representation known as the *Union of $\mathcal{Z}$-Polyhedra (UZP)* format [26]. UZP is a novel and highly flexible format based on a key insight: *any arbitrary set of sparse integer coordinates can be equivalently represented as a union of dense sets*.

In the most general case, a set of n sparse coordinates may be expressed as n singleton dense sets, each containing one coordinate—preserving complete generality. On the other hand, when regularity exists—such as in matrices with nonzeros along the diagonal—the entire set can be concisely represented by a single dense polyhedral set, e.g., $\{(i, j) \mid 0 \leq i < n \wedge i = j\}$. This principle naturally extends to higher-dimensional tensors, enabling UZP to represent sparsity patterns of arbitrary dimensionality in a unified, structured manner.

Given the structural flexibility of the UZP format, this thesis investigates the potential for post-generation optimization of UZP representations. This is realized through the development of a set of transformation tools, referred to as *tuners*, which leverage the decoupled design of shape definitions and origin lists within the UZP framework. These tuners enable the transformation of one valid UZP representation into another, facilitating explicit trade-offs between storage compression and runtime performance.

For example, certain tuners may favor shape configurations that enhance SIMD vectorization potential, while others may reorder origin coordinates to improve data locality and reduce control-flow divergence. Importantly, these transformations are applied at the representation level and do not require modifications to the executor logic or recompilation of low-level code. As such, tuners serve as a key enabler for flexible and architecture-aware performance tuning within the UZP ecosystem.

This thesis presents the design, implementation, and empirical evaluation of these tuners, emphasizing their impact on the execution efficiency of UZP-based sparse computations. Furthermore, a comprehensive performance analysis is conducted to examine the broader implications of tunable structured sparsity on sparse matrix representations.

Contributions

The key contributions of this thesis are as follows:

- **Development of Tuners:** Design and implementation of a suite of tuners-Aggregator, Favor Locality, and Tile-based Origin Reordering that transform UZP representations to improve storage compression, data locality, and computational performance.
- **Performance Profiling and Analysis:** Detailed profiling of hardware-level behaviors such as cache utilization, SIMD vectorization efficiency, and memory access patterns across different tuned UZP variants, using performance counters and systematic benchmarking.

- **Design of Generic Executors:** Creation of generic, parameterized executors capable of operating on UZP representations, including auto-generation of macro-specialized loops to improve compiler optimizations without sacrificing flexibility.
- **Experimental Validation:** Comprehensive evaluation over 229 matrices from the SuiteSparse collection, demonstrating significant improvements in SpMV runtime, memory footprint, and SIMD throughput compared to both traditional UZP and baseline formats.

Thesis Outline

This thesis is organized as follows:

- **Chapter 2** introduces the background and motivation for sparse matrix representations. It reviews the limitations of traditional formats, explores recent advances in structured sparse modeling using polyhedral techniques, and discusses foundational concepts behind Z-polyhedra and their use in capturing sparsity patterns.
- **Chapter 3** presents the design and mathematical foundations of the Union of Z-Polyhedra (UZP) format. It details how UZP generalizes sparse representations by modeling nonzeros as unions of structured polyhedra and lattices, and discusses the implementation strategy including integration with traditional CSR and COO formats.
- **Chapter 4** describes the tuning mechanisms developed for UZP, including the Aggregator Tuner for merging regularly strided origins, the Favor Locality Tuner for optimizing access locality, and the Tile-Based Tuner for spatial restructuring. It also presents the design of generic executors capable of operating over UZP representations, along with strategies for auto-generating shape-specific loop macros.
- **Chapter 5** details the experimental methodology, benchmark suite, and evaluation metrics. It presents comprehensive results analyzing the impact of different tuners on SpMV performance, memory behavior, SIMD utilization, and scaling. Comparisons between Standard UZP, Aggregated UZP, and Favor Locality-enhanced variants are discussed.

- **Chapter 6** introduces an interactive visualization tool developed to support UZP analysis. The tool enables graphical inspection of sparse matrix structures, shape placement, and transformations applied by the tuners.

- **Chapter 7** concludes the thesis by summarizing key findings, evaluating the broader implications of tunable structured sparse representations, and suggesting future research directions for extending the UZP framework and optimizing sparse computations further.

Chapter 2

Prior work and Background

2.1 Sparse Matrices in Modern Computing and its Challenges

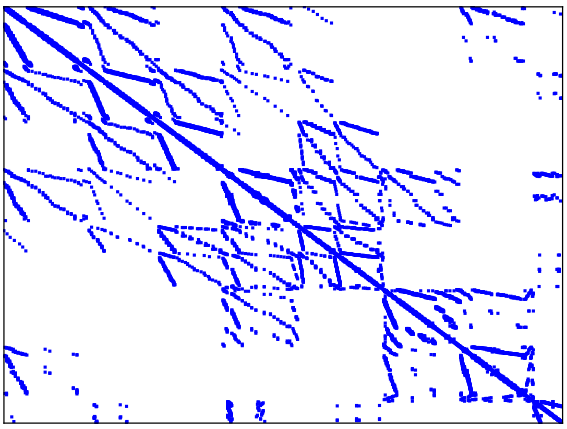

Figure 2.1: HB/can_1072: 1072×1072 matrix with 12,444 nonzeros

Sparse matrices play a central role in a wide range of modern computational problems. They are particularly common in domains such as scientific simulations, numerical methods, graph analytics, machine learning, and high-performance computing workloads, where data is often structured in a way that leads to a significant number of zero-valued elements. Rather than treating these zeros as meaningful data, sparse matrix techniques exploit sparsity to reduce both storage and computational costs, thereby improving scalability and efficiency.

Beyond traditional scientific computing, the role of sparse matrices has become increasingly critical in emerging fields such as deep learning model compression, natural language processing (e.g., sparse attention mechanisms), recommender systems, and large-scale knowledge graphs. Sparse representations enable systems to scale to billions of parameters or nodes while maintaining tractable memory footprints and computational demands. Efficient sparse data structures and algorithms are essential for enabling real-time inference on edge devices, accelerating graph-based learning models, and optimizing solvers in numerical weather prediction, finite element analysis, and quantum simulations. As hardware architectures evolve toward memory-centric and dataflow-oriented designs, the ability to exploit sparsity at multiple levels of the system-storage, memory access, and computation-continues to be a key factor in achieving performance, energy efficiency, and scalability at exascale and beyond.

Given the central role of sparse matrices across a wide range of computational domains, a natural question arises: how should sparse data be efficiently stored and organized to maximize computational performance? The choice of storage format significantly affects memory access patterns, computational throughput, and overall algorithmic efficiency. Sparse matrix representations must balance competing objectives, including minimizing storage overhead, enabling fast random or sequential access, and supporting efficient parallelization.

Common computational kernels such as Sparse Matrix-Vector Multiplication (SpMV) and Sparse Matrix-Matrix Multiplication (SpMM) are foundational operations in applications ranging from iterative solvers and eigenvalue computations to machine learning and graph processing. The performance of these kernels is highly sensitive to the underlying sparse data representation, motivating decades of research into storage schemes and execution strategies. Over time, the community has developed a variety of sparse matrix formats, including most commonly used one's i.e, Compressed Sparse Row (CSR), Compressed Sparse Column (CSC), Coordinate Format (COO), and more recent innovations aimed at hardware-specific optimizations.

In the section 2.1.1, we review some of the prominent and state-of-the-art work that has been done towards sparse matrix storage formats and computational frameworks. This survey highlights promising directions explored by the research community to address the challenges associated with sparse data representation and computation.

2.1.1 Related Work

Sparse matrix representations utilize specialized storage formats that encode only the nonzero elements and their corresponding positions, thereby eliminating the overhead associated with storing zero entries. Among the most widely adopted formats are Compressed Sparse Row (CSR), Compressed Sparse Column (CSC), and Coordinate (COO). The CSR format, for instance, maintains three arrays: one for nonzero values, one for the column indices of those values, and one for row pointers marking the start of each row. CSC is its column-major analog, while COO represents nonzeros as explicit $(\text{row}, \text{col}, \text{val})$ tuples.

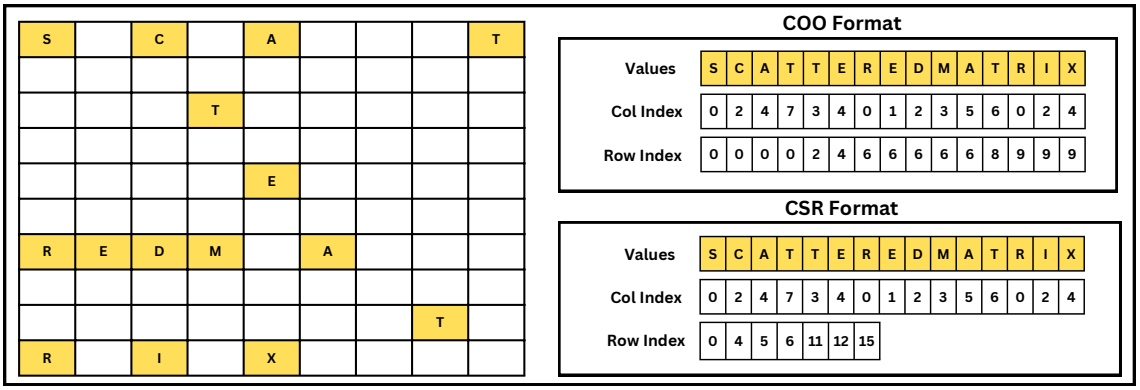

Figure 2.2: Figure showing a sample sparse matrix and its storage in (1) COO and (2) CSR formats.

These formats significantly reduce memory usage and enable efficient sparse matrix-vector (SpMV) and matrix-matrix (SpMM) operations at scale. Much work has been done to improve its shortcomings, particularly in tuning the storage format to specific computations. For example, Vuduc [38] presented an automated system for generating efficient implementations of SpMV on CPUs, while Williams et al. [41] moved toward multi-core platforms with the implementation of parallel SpMV kernels. Figure 2.2 illustrates the CSR and COO representations of a sample sparse matrix. Listing 2.2 shows the corresponding CSR-based SpMV executor.

Listing 2.1: CSR executor for SpMV.

```
1 for (i = 0; i < N; ++i)
2     for (j = row_ptr[i]; j < row_ptr[i+1]; ++j)
3         y[i] += A[j] * x[col_idx[j]];
```

While traditional formats such as CSR and COO are space-efficient and relatively simple to implement, they exhibit performance limitations on modern hardware. In particular, the CSR executor shown in Listing 2.2 relies on indirect memory accesses, such as `x[col_idx[j]]`, to fetch vector elements. These indirection arrays often cause irregular, non-contiguous memory access patterns, which are poorly suited for Single Instruction Multiple Data (SIMD) execution. Modern vector units require predictable and aligned memory access to fully exploit their through-

put capabilities. As a result, the irregularity inherent in formats like CSR and COO presents a bottleneck for compiler auto-vectorization, cache reuse, and hardware prefetching.

To mitigate these challenges, several high-performance libraries, including Intel MKL and cuSPARSE, provide architecture-specific optimized implementations for operations such as sparse matrix-vector multiplication (SpMV) and sparse matrix-matrix multiplication (SpMM). These libraries achieve high throughput on large matrices with regular sparsity patterns by applying preprocessing techniques such as format-specific tuning, blocking, and sparsity analysis. However, the preprocessing overhead can be significant, and its cost may dominate execution time for smaller or highly irregular matrices, resulting in diminished overall performance.

Another approach to address the limitations introduced by indirection-based memory access is to restructure sparse computations into regular forms to improve SIMD compatibility, cache efficiency, and hardware utilization. Recent research has proposed several methods to minimize or eliminate the dependency on indirection arrays through structured sparsity representations, polyhedral modeling, and code generation techniques. These efforts aim to enable predictable memory access patterns, facilitate compiler auto-vectorization, and reduce preprocessing overhead, particularly for matrices with irregular sparsity pattern.

Early work by Saltz et al. [28] proposed runtime techniques for distributed-memory parallelization of irregular applications [28, 23, 30]. These efforts were later augmented with compiler-based approaches that automatically generated parallel code from irregular structures [36, 15, 1]. More recent techniques aim to exploit runtime regularity to improve execution efficiency. For instance, Ravishankar et al. [25] leverage dynamic analysis to generate polyhedrally-optimizable executor code in specific cases. Similarly, Sukumaran-Rajam and Clauss [33] detect runtime regularity using linear interpolation and regression models, selecting optimizations speculatively based on detected patterns. Cheshmi et al. [12] further advance this direction by developing an inspection method that fuses two sparse kernels, enabling the generation of parallel code optimized for both memory locality and load balancing.

The Sparse Polyhedral Framework [20, 32, 34] employed *uninterpreted function symbols* to over-approximate irregular computations within a polyhedral model, facilitating the generation of inspector/executor code at compile time. The generated code is valid for any input sparse matrix but does not exploit opportunities for specialization based on the structure of a particular matrix.

Sympiler [8, 9] is an inspector/executor compiler that analyzes the sparsity structure of an input to generate ad-hoc executor code. But the problem with is, indirection arrays are not fully eliminated and may still appear in loop bounds and access functions. Similar work, the TACO compiler [19, 14] is a framework for generating optimized sparse tensor computations, supporting a wide range of sparse input formats and enabling the composition of different formats to generate the corresponding executor programs.

One of the promising work done in this approach is, Augustine et al. [2] DSCG work, they identify regularity in the sparsity structure of a matrix by computing polyhedral shapes that capture the nonzero coordinates, and generates data-specific code that eliminates the use of indirection arrays. The method employs $\mathcal{Z}$-polyhedra to represent nonzero coordinates and relies on compiler auto-vectorization to optimize the generated matrix-specific executor. A more detailed discussion of DSCG is presented in the next section, 2.1.2 as it forms part of the foundation for this thesis work.

Horro et al. [18] leverages DSCG work by developing a custom local rescheduling and advanced SIMD synthesis approach, including the fusion of independent reductions. MACVETH achieves systematic performance improvements over DSCG; however, these approaches lack the modularity and flexibility provided by UZP (our proposed work, explained in section 3). In particular, the executor code must be recompiled for each target sparse matrix, leading to binaries whose size can be proportional to the number of nonzero elements. In contrast, UZP generates *generic executors* whose size is independent of the number of nonzeros, while storing the polyhedral set information separately in a file that can be tuned and communicated efficiently.

Another work towards obtaining regularity from the sparse data was demonstrated by Cheshmi et al. [11] - Partially-Strided Codelets. PSC work propose to optimize sparse computation execu-

tion on multi-core CPUs [11]. PSCs are polyhedral shapes, but represent a more restricted subset compared to those considered by, e.g., Augustine et al. [2]. These structures can be encoded within the UZP framework. PSC targets a limited set of codelets for which efficient implementations are available, either through existing libraries or manual generation, and applies a partitioning strategy during inspection to map nonzero coordinates (nzc) to codelets, facilitating locality and load-balancing in parallel execution.

Where as UZP, by contrast, provides a generalized representation supporting arbitrary polyhedral shapes, enabling modular and decoupled reconstruction, tuning (e.g., locality optimization, storage compression), and execution. While PSC [11] is optimized for large sparse matrices on multi-core CPUs using a small number of predefined shapes, UZP focuses on scaling reconstruction across a larger set of candidate shapes and optimizing for matrices with fewer than 20 million nonzero elements, particularly in cold-cache, single-core environments. In single-core, cold-cache setup, UZP generic executor (more about in ssection 4.4 outperforms PSC on approximately 76% of evaluated matrices.

Block-based formats [13, 7] such as BCSR, BELLPACK, and CSB exploit structural regularity by representing contiguous coordinate sets through fixed-size blocks, often storing explicit zeros. These formats typically impose a uniform block size across the matrix. In contrast, UZP imposes no fixed block size or explicit zero storage.

DCSR, RPCSR, and DCSC [40, 6] compress index structures to reduce bandwidth overheads. CVR Xie et al. [42] targets SpMV by processing multiple rows concurrently. AlphaSparse Du et al. [17] optimizes SpMV on GPUs by generating sparsity-specific storage formats. Many of these formats can be encoded within the UZP framework.

2.1.2 Reconstructing Irregular Structures via $\mathcal{Z}$-Polyhedra

Irregular data structures are typically characterized by sets of discrete points, each of which must be uniquely identified within the computational domain. In this work, such elements are modeled as *integer tuples*, or *points*, forming the basis for a more structured representation.

These points are first captured in an ordered list called a *trace*. A trace formalizes the sparse layout as a sequence

$$T = \{\vec{p}^1, \vec{p}^2, \ldots, \vec{p}^n\},$$

where each $\vec{p}^i$ is an m-dimensional integer tuple. If the computational semantics allow, the trace may be considered *reorderable*, meaning that its elements can be grouped without preserving their original order. This is particularly relevant for operations such as SpMV, which are associative in nature and allow independent accumulation of nonzeros.

To provide a structured yet compact encoding of such irregular traces, this work leverages the mathematical abstraction of $\mathcal{Z}$*-polyhedra*. A $\mathcal{Z}$-polyhedron is defined as the image of an integer polyhedron $\mathcal{D}$ under an affine lattice mapping F, or equivalently as the intersection:

$$Z = \mathcal{D} \cap F,$$

where:

An *integer polyhedron* $\mathcal{D}$ can be defined as the set of integer points satisfying a system of affine inequalities, where each inequality takes the form:

$$\sum_{i=1}^{d} \alpha_i x_i + \beta \geq 0 \quad \text{with } \alpha_i, \beta \in \mathbb{Z}, \quad x_i \in \mathbb{Z},$$

In contrast, an *integer lattice* F is a multidimensional affine function that maps an integer input domain $\vec{I}$ to an output domain $\vec{O}$ using a set of equalities:

$$F = \left\{ [i_1, \ldots, i_{\dim(\vec{I})}] \to [o_1, \ldots, o_{\dim(\vec{O})}] : o_1 = f_1(\vec{I}),\, o_2 = f_2(\vec{I}), \ldots \right\},$$

where each output dimension can be defined as a linear combination of input indices with integer coefficients.

Example 1 (Example of a $\mathcal{Z}$-polyhedron)**.** *Consider a set of four 1D points:* $\{2, 4, 8, 10\}$. *These points can be captured by applying an integer lattice to the domain of a polyhedron.*

- *Define the **integer lattice**:* $F = \{[i, j] \to [x] : x = 6i + 2j\}$
- *Define the **integer polyhedron**:* $\mathcal{D}_2 = \{[i, j] : 0 \leq i \leq 1 \wedge 1 \leq j \leq 2\}$

Applying the lattice F to the integer polyhedron $\mathcal{D}_2$ yields a $\mathcal{Z}$-polyhedron:

$$Z = F(\mathcal{D}_2)$$

The image Z evaluates to the set $\{2, 4, 8, 10\}$, as the affine function $x = 6i + 2j$ maps the points in $\mathcal{D}_2$ to these values.

Coming back to the reconstruction, each $\mathcal{Z}$-polyhedron thus models a regular subset of points that satisfy both geometric constraints and affine transformation rules. The *origin* of a $\mathcal{Z}$-polyhedron is the lexicographically smallest point in $\mathcal{D} \cap F$ and serves as an anchor for instantiating the shape.

A reconstructed trace T_{poly} from an original trace T of n elements is defined as a finite collection of $\mathcal{Z}$-polyhedra D_i, denoted as

$$T_{\text{poly}} = \{D_1, \ldots, D_p\},$$

such that

$$\sum_i \#D_i = n \quad \text{and} \quad T \equiv T_{\text{poly}},$$

where T_{poly} captures exactly the same set of points as T. Each $\mathcal{Z}$-polyhedron D_i in T_{poly} may have a different dimensionality and a different number of points.

To illustrate the polyhedral expression presented in Example 1, the following code fragment demonstrates how the iteration space can be mapped into a nested loop structure, with p_x denoting the computed coordinate along the x-axis.

Listing 2.2: Generated loop structure corresponding to Example 1.

```
1 for (i = 0; i <= 1; ++i)
2     for (j = 1; j <= 2; ++j)
3         p_x = 6*i + 2*j;
```

Example 2 For the first 11 points from the Figure 2.3 if we want to construct $\mathcal{Z}$-polyhedron and show that non-contiguous points can be grouped into same polyhedra considering that the trace to be reorderable.

- $D_1 = \{[i, j, k] : 2 \leq i \leq 3 \wedge i = j \wedge k = 16i - 12\}$ models points p^6, p^{10}.
- $D_2 = \{[i, j, k] : 0 \leq i \leq 1 \wedge i = j \wedge k = 8i\}$ models points p^1, p^3.
- $D_3 = \{[i, j, k] : 1 \leq i \leq 2 \wedge 4 \leq j \leq 5 \wedge k = 12i + 4j - 16\}$ models points p^4, p^5, p^7, p^8.
- $D_4 = \{[i, j, k] : 0 \leq i \leq 1 \wedge j = 3i + 3 \wedge k = 36i + 4\}$, with lattice $F_4 = \{[i, j, k] \rightarrow [x, y, z] : x = 3i \wedge y = j \wedge z = k\}$, models p^2, p^{11}.
- $D_5 = \{[i, j, k] : i = 3 \wedge j = 0 \wedge k = 32\}$ models the singleton point p^9.

To illustrate the polyhedral expression corresponding to the points $p_6 = (2, 2, 20)$ and $p_{10} = (3, 3, 36)$. The following loop structure is generated by mapping the iteration space into a simple nested computation where i and j are equal, and k is computed as an affine function of i for polyhedron D_1.

Listing 2.3: Generated loop structure corresponding to polyhedron D_1.

```
for (int i = 2; i <= 3; ++i) {
    int j = i;
    int k = 16 * i - 12;
    // (i, j, k) represents the generated coordinates
    // example - perform spmv on them.
    y[i] = A_data[k] * x[j];
}
```

All polyhedra except D_4 use the identity lattice $F = \{[i, j, k] \to [x, y, z] : x = i \wedge y = j \wedge z = k\}$. The trace is thus reconstructed as the collection D_1, D_2, D_3, D_4, D_5, covering all 11 points.

Now to construct such T_{poly}, Augustine et al. [2] introduces an *Extended TRE algorithm* (eTRE), which incrementally builds polyhedra from the trace. The algorithm uses a window-based strategy, maintaining a list of candidate polyhedra sorted by heuristic fitness scores, and attempts to grow the best candidate at each step. When a candidate cannot be expanded further without incurring excessive dimensionality or violating structural constraints, the algorithm falls back to the next candidate or begins a new polyhedron.

Table 2.1: Evolution of the number of pieces as a function of their maximal dimensionality (maxd) for matrix HB/nos1 (1,017 nonzero elements).

$\mathbf{max}_d$	2	3	4	5	6	7	8
pieces	312	159	81	4	3	2	1
cycles	11373	11583	9938	35730	34116	39306	50371
LoC	772	1004	671	195	368	165	101

Note the Table 2.1 illustrates the trade-offs involved in reconstructing sparse traces using $\mathcal{Z}$-polyhedra at different maximum dimensions, for Fig. 2.3(c). While higher dimensions (e.g., $\max_d = 8$) can represent all points with a single polyhedral piece, they result in poor execution

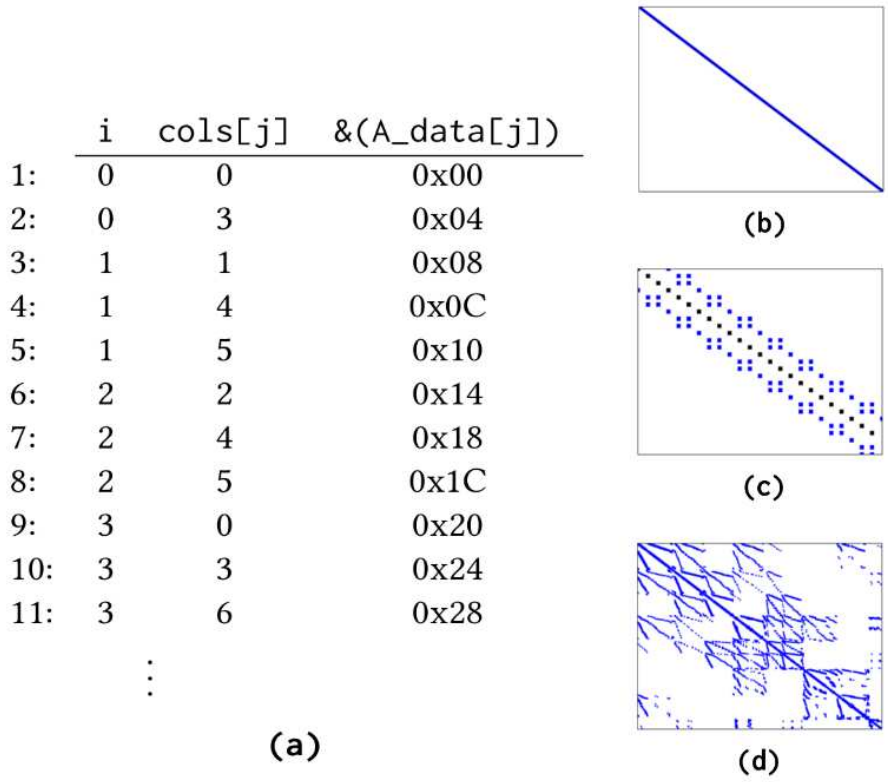

Figure 2.3: Different sparse matrices from the HB group of the SuiteSparse Matrix Collection. Figure (a) shows an except of the accesses performed during SpMV of matrix `HB/nos1`. The nonzero elements in this matrix are shown in Figure (b), and a zoom of its main diagonal is provided in Figure (c). This is a 237×237 matrix with 1,017 nonzero elements, and its reconstructed code consists of a single statement inside an 8-dimensional loop. Figure (d) shows the nonzero elements in `HB/can_1072`, a $1,072 \times 1,072$ matrix with 12,444 nonzero elements, which does not exhibit any apparent regularity. Its reconstructed code includes 870 pieces of up to 8 dimensions.

performance due to complex control flow and high cycle counts. Conversely, very low dimensions (e.g., $\max_d = 2$) produce many small pieces, introducing significant overhead. The optimal balance is observed around $\max_d = 4$, where the execution cost and code complexity are both minimized. This highlights the need to carefully tune the polyhedral modeling parameters based on application and hardware characteristics, rather than optimizing solely for structure or compactness.

Although effective, the eTRE approach is limited by its dependency on trace-order processing, which can result in suboptimal computational structures such as polyhedra with irregular strides or

poor data locality. To address this, the thesis introduces a complementary *pattern-matching-based* reconstruction strategy.

This pattern-matching approach operates on reorderable traces and searches for known polyhedral patterns across the trace, regardless of order. When a match is found, the corresponding points are grouped into a $\mathcal{Z}$-polyhedron. This method might enable better control over loop regularity, spatial locality and vectorization opportunities.

Together, these strategies enable a principled and tunable reconstruction of sparse traces into structured, reusable, and optimization-friendly polyhedral components. This serves as the foundation for efficient sparse representation and compiler-based code generation, enabling further tuning and execution-level optimization within the polyhedral framework.

Hence, this reconstruction approach proposed by Augustine et al. [2] enables the automatic generation of layout-aware, hardware-conscious code tailored to the static sparsity structure of the input. It generalizes across a wide range of sparse workloads, including tensors, and does not require manual format selection or tuning, thereby simplifying integration into compiler toolchains. However, a significant trade-off in this method is the increase in code size due to specialization. For each distinct polyhedral region identified in the trace, the compiler generates a corresponding loop nest, leading to potentially large and fragmented binaries. While these specialized loops are effective in exploiting locality and vectorization particularly on single-core CPUs this increase in code volume may hinder scalability and maintainability in larger applications. Again the scalability of this approach to multi-core architectures is not thoroughly explored, as the evaluation is limited to comparisons against Intel MKL's inspector-executor model in its single-core configuration. Consequently, the reported performance gains remain largely confined to single-core CPUs, leaving open questions regarding its applicability to modern multi-threaded environments.

2.1.3 Custom Vector Code Generation for Fixed Sparse Structures

Another notable limitation of the Augustine et al. [2] approach is its dependency on standard compiler heuristics for vectorization. While the framework generates affine loop nests amenable

to SIMD execution, it does not emit explicit SIMD intrinsics or low-level vectorized code. As a result, the extent of vectorization is assigned to the compiler, which may not fully exploit available vector units in the presence of non-contiguous or strided memory accesses. The regularization strategy thus enables, but does not ensure, efficient SIMD utilization.

To address this gap, subsequent work by Horro et al. [18] proposes custom vector code generation tailored to fixed sparse structures. This approach targets the generation of data-specific SIMD code that balances performance with code size and complexity, extending the capabilities of prior frameworks by addressing the irregularities that persist even after loop restructuring.

The [18] approach acknowledges that naive unrolling or aggressive vectorization may result in excessive code size, leading to instruction cache pressure and increased Translation Lookaside Buffer (TLB) overhead. Instead, the authors propose generating SIMD code in a data-specific manner-carefully selecting unrolling factors, vector widths, and memory layouts tailored to the structure of each sparse input. This enables fine-grained control over how vector instructions are applied, allowing the framework to extract more consistent performance across diverse inputs while avoiding the pitfalls of excessive specialization. Importantly, this work begins to bridge the gap between high-level structural regularization and low-level hardware aware code generation, moving toward truly end-to-end optimized execution pipelines for sparse computations.

To address the limitations of relying solely on compiler-driven vectorization, the authors of the Horro [18] paper propose a novel source-to-source compiler called *MACVETH* (Multi-Architectural C-VEcTorizer for HPC). Rather than depending on the auto-vectorizer within conventional compilers, MACVETH takes the output of data-specific inspectors such as those from the [2] approach and generates explicit SIMD-optimized C code tailored to the target architecture. Its core contribution lies in systematically exploring the design space of vectorization strategies for irregular memory access patterns commonly found in sparse computations.

MACVETH generates multiple candidate "codelets" for small computational units that involve gathering, reducing, or packing sparse data. These implementations leverage AVX2 intrinsics and range from using hardware gather instructions to handcrafted sequences involving shuffles,

blends, and load permutations. To manage the explosion of variants, the system integrates an SMT solver to prune instruction sequences that are functionally redundant, ensuring efficient coverage of meaningful vectorization strategies without exhaustive enumeration.

For each piecewise-regular loop obtained from the sparse matrix structure, MACVETH substitutes scalar memory accesses with the most efficient SIMD instruction sequence discovered during profiling. It also performs fusion of independent reductions to better utilize vector units and reduce instruction count. Recognizing the impact of code size on performance, particularly for instruction cache and TLB pressure, the system balances performance and binary footprint by preferring compact instruction recipes even at the cost of minor slowdowns overly unrolled versions. This hybrid strategy ensures both high performance and maintainability of the generated code.

Empirical results reported in the paper demonstrate that, across a diverse suite of 230 sparse matrices (with up to 20 million nonzeros), MACVETH achieves a geometric mean speedup of 1.33x over Intel MKL for matrices with more than 10,000 nonzeros. Its advantage diminishes for very large matrices, where memory bandwidth and cache pressure become the dominant bottlenecks. One notable limitation of MACVETH lies in its complexity: it requires architecture-specific profiling and exhaustive benchmarking to construct its microkernel database. This profiling step is resource-intensive, although amortized over repeated use. Moreover, its current scope is restricted to CPUs with AVX2/AVX-512 support, making generalization to other architectures an open challenge.

2.1.4 UZP Representation and its Background

The works of [2] and [18] collectively mark a significant progression in the field, from high-level structural regularization to low-level, vector-specific code generation. Despite these advancements, both approaches remain closely tied to the structure of individual matrices, relying on custom code generation pipelines and static analysis methods tailored to specific sparsity patterns. As a consequence, the generated binary code size can increase significantly for larger matrices, due to the need to generate and store extensive matrix-specific code. This leaves open the broader ques-

tion of how to generalize sparse matrix representations and executor design while maintaining a balance between performance, portability, and programmability.

To address this gap, we introduce the *Union of $\mathcal{Z}$-Polyhedra* (UZP) format a flexible, parametric representation that decouples the structural encoding of sparse data from executor logic. Unlike prior methods that generate dedicated loop nests per matrix, UZP supports a reusable, shape-aware execution model driven by a compact dictionary of polyhedral shapes and a list of origin points. Furthermore, rather than relying on aggressive static specialization, UZP enables post-optimization through tuners that transform the representation itself. These tuners restructure origin ordering, merge origins, or alter tiling layouts to improve locality, vectorization potential, and thread-level parallelism without requiring changes to the executor. As a result, UZP bridges the gap between static structural encoding and dynamic execution needs, offering a practical pathway toward performance-portable sparse computation.

A key feature of the UZP format is the separation between shape definitions and their instantiation points (list of origins). Rather than explicitly enumerating every occurrence of a polyhedral pattern which leads to redundancy when identical shapes recur, UZP stores a dictionary of polyhedral shapes and a corresponding list of origins where these shapes can be instantiated. This decoupling significantly reduces storage overhead and facilitates the reuse of structural patterns. To ensure representation profitability, UZP subsumes classical formats such as CSR and COO - use them for the set of unincorporated non zero points that are not part of the polyhedral dictionary. We will learn more about this section 3.

Chapter 3

Understanding the Union of $\mathcal{Z}$-Polyhedra Format (UZP)

The Union of $\mathcal{Z}$-Polyhedra (UZP) is a structured sparse matrix representation that captures regularity among nonzero coordinates to enable compact storage and efficient computation. Unlike traditional formats such as Compressed Sparse Row (CSR) and Coordinate List (COO), which represent each nonzero independently without exposing any higher-level structure, UZP organizes nonzeros into a dictionary of affine polyhedral shapes that can be instantiated across different locations within the matrix.

This chapter presents the mathematical foundations of UZP, formalizing the construction of $\mathcal{Z}$-polyhedra and illustrating how sets of nonzero coordinates are grouped into such representations. We begin by revisiting prior work of Augustine et al. [2] on polyhedral modeling for sparse data structures, then progressively build toward a unified framework that accommodates both structured and unstructured sparsity.

[2] demonstrated the use of polyhedra and affine lattices to compress sets of nonzero coordinates in sparse matrices. Each nonzero element in an n-dimensional sparse dataset can be represented as an integer tuple $(\vec{i}, \text{data})$, where $\vec{i}$ denotes the coordinate and data indicates the position within the data vector. Using this abstraction, sets of nonzeros—potentially non-consecutive—can be grouped into $\mathcal{Z}$-polyhedra

Formally, a $\mathcal{Z}$-polyhedron is defined as follows:

Definition 1 ($\mathcal{Z}$-polyhedron)**.** A $\mathcal{Z}$-polyhedron is the image of a k-dimensional integer polyhedron P under an $m \times k$ integer affine lattice L. Here, P is defined by a set of affine inequalities over its dimensions, and L is represented by an integer-coefficient matrix mapping from a k-dimensional space to an m-dimensional space.

In this definition, the dimension k of the polyhedron P is unrestricted, while the lattice dimension m equals $n + 1$ in the above encoding. Consequently, the image $L(P)$ equivalently the intersection of P and L compactly represents the sparse coordinates and their associated data positions within the sparse data structure.

For illustration, consider a sparse matrix containing six nonzero coordinates (nzcs): $(0, 0, 0)$, $(0, 71, 1)$, $(1, 42, 2)$, $(2, 44, 3)$, $(3, 46, 4)$, and $(42, 42, 5)$. In each 3-tuple, the last value represents the position in the data vector, as commonly used in CSR or COO storage schemes.

A possible polyhedral representation of this sparse structure might involve defining six separate $\mathcal{Z}$-polyhedra. For example, the first nonzero coordinate could be represented by:

$$P_1 : \{(i, j, d) : i = 0 \wedge j = 0 \wedge d = 0\}, \quad L_1 : \{(i, j, d) \to (i, j, d)\},$$

and similarly the second point as:

$$P_2 : \{(i, j, d) : i = 0 \wedge j = 71 \wedge d = 1\}, \quad L_2 = L_1,$$

with analogous definitions for the remaining points. However, such an approach provides limited practical benefit compared to traditional COO encoding, since it requires one distinct polyhedron for each individual nonzero element, offering no actual compression or structural simplification. Instead of representing each nonzero coordinate (nzc) individually, compression can be achieved by grouping multiple nzcs into a single polyhedral structure. For instance, the points $(1, 42, 2)$, $(2, 44, 3)$, and $(3, 46, 4)$ can be captured compactly using the following formulation:

$$P_3 : \{(l) : 1 \leq l \leq 3\}, \quad L_3 : \{(l) \to (i, j, d) : i = l,\ j = 40 + 2l,\ d = l + 1\}.$$

This single $\mathcal{Z}$-polyhedron represents all three points, thereby reducing the number of polyhedra required to express the structure by two, compared to an uncompressed representation.

This strategy proves effective even in sparse matrices that exhibit limited regularity. By identifying and modeling repetitive strided patterns, multiple nzcs can be grouped into structured polyhedra. As demonstrated in prior work Augustine et al. [2], complex sparsity structures can be reconstructed using this approach. For example, Figure 2.1 shows a reconstruction comprising 870 $\mathcal{Z}$-polyhedra, some of which utilize up to eight dimensions ($k = 8$) to encode high-dimensional patterns [16, 2].

3.1 Design Principles of UZP

The UZP format is a polyhedral-based sparse matrix representation that facilitates the implementation of generic executors. In contrast to previous approaches [2, 18], UZP explicitly decouples the parametric definition of polyhedral shapes from their instantiation coordinates, referred to as origins. This design choice allows the creation of executors independent of specific sparsity patterns while still benefiting from polyhedral compression. Generic executors that operate using the UZP format are described in Section 4.4.

As illustrated in Section 3, explicitly listing all occurrences of polyhedral shapes introduces redundancy, since identical shapes (e.g., a diagonal pattern of length three with stride two) frequently recur due to the shape-mining process [2, 18]. To mitigate this redundancy, UZP stores shape definitions separately in a dictionary of parametric shapes, distinct from the list of origins where these shapes apply. Consequently, storage overhead is substantially reduced. Furthermore, UZP integrates hybrid segments, allowing portions of the sparse structure to be represented using traditional formats such as CSR or COO. This hybrid capability is particularly useful for irregular sparsity patterns where polyhedral compression yields minimal benefit. Thus, UZP enables both flexible tuning of sparse representations and interoperability with established sparse matrix formats.

A complete UZP representation thus comprises five distinct components:

- A dictionary of parametric shapes represented as $\mathcal{Z}$-polyhedra;
- A list of origin coordinates specifying instantiation points for each polyhedral shape;

- An optional CSR-encoded segment for efficiently handling sparse regions with limited structure;
- An optional COO-encoded segment for explicitly representing irregular sparsity patterns;
- A data vector containing the actual nonzero values associated with these coordinates.

3.2 Implementation of UZP

The implementation of both the UZP format and the associated generic executors, presented in the section 4.4, has been accomplished using C/C++ data structures. UZP is fully implemented, including conversion from/to CSR and COO, automatic mining of polyhedral shapes to build a UZP file, and very simple generic executors for SpMV and other computations.

Dictionary of Shapes.

A shape in UZP is defined as a parametric $\mathcal{Z}$-polyhedron, which is applied to a set of origins. $\mathcal{Z}$-Polyhedra are typically represented using matrices of integer coefficients [3] representing inequalities bounding the polyhedron, and the associated lattice. In the UZP format, columns of this matrix can be omitted when the shape is hyper-rectangular, as only extremal vertices are needed to represent the shape. For instance, a block of 8x8 dense points is simply represented as the pair of vertices $(0,0);(8,8)$ and a unit lattice. A dense block of nonzero coordinates located at $(42..50 \times 128..136)$ is therefore represented with the origin (42,128), on which the dense block prototype $(0,0);(8,8)$ with unit-stride is applied, resulting in the set $(42+0..42+8\times128+0..128+8)$.

List of Origins.

An origin in UZP is defined as an $m+2$-tuple of integer coefficients. This tuple includes the first m-dimensional vertex (e.g., $(42,128)$ in the previous example), the $data$ offset, and the shape identifier in the dictionary. The list of origins in UZP is constrained to never exceed the number of non-zero elements (nnz).

Modeling Data.

The `data` vector in UZP stores the actual data values for each non-zero coordinate (nzc). The reconstruction of the sparse structure into polyhedral shapes can be constrained to operate on a fixed data vector (e.g., the exact data vector of an existing CSR matrix representation, to facilitate integration in a full application flow of data-specific SpMV codes without any data conversion

Modeling Unincorporated Points.

Finally, UZP includes an optional section that is typically utilized to encode points that cannot be efficiently captured by $\mathcal{Z}$-polyhedral shapes. Rather than modeling these points using a single-point shape and the associated origins, UZP supports modeling these points using the classical CSR and COO formats, with their separate `data` segment.

Example 2. *Fig. 3.1 illustrates the representation of a sparse matrix using CSR, COO, and two possible UZP representations. In this example, the dictionary encodes hyper-rectangular shapes; however, UZP also supports more general polyhedral shapes.*

In the bottom-left example of Fig. 3.1, shape `s1` *represents a 1D structure. Shapes in UZP are encoded using a combination of a polyhedron and an integer affine lattice. The first two values define the vertices of the 1D polyhedron (e.g.,* $l : 0 \leq l \leq 2$*, where the vertices 0 and 2 are stored). The subsequent values define the coefficients of the integer affine lattice mapping* $l \mapsto (i, j, d)$*. For instance, the tuple* `2,0,0,0,1,0` *encodes the mappings* $i = 2l + 0$*,* $j = 0l + 0$*, and* $d = 1l + 0$*. Origins are applied to instantiate this* $\mathcal{Z}$*-polyhedral shape. For example, applying the origin* $(0, 0, 0, s1)$ *in the left UZP example yields the tuples* $(0, 0, 0)$*,* $(0, 2, 1)$*, and* $(0, 4, 2)$*, corresponding to the first three nonzero coordinates (nzc) labeled S, C, and A.*

Optionally, individual nonzero entries can be encoded using CSR or COO fragments. However, these entries can equivalently be represented as polyhedra of a single point, as demonstrated in the right UZP example with shape `s4`*.*

The UZP dictionary can accommodate shapes of varying dimensionalities. On the right side of Fig. 3.1, shape `s1` *is a 2D shape defined by two loop variables* l_1 *and* l_2 *such that* $0 \leq l_1 \leq 2$

and $9 \leq l_2 \leq 1$ *(encoded as* `0,2,0,1`*), along with an associated lattice* $(l_1, l_2) \mapsto (i, j, d)$*, where* $i = 0l_1 + 9l_2 + 0$*. This two-dimensional mapping requires three integer coefficients per output dimension instead of two, and applying the origin* $(0, 0, 0, s1)$ *instantiates the nonzero entries S, C, A, R, I, and X.*

To further clarify the instantiation process, consider again the 2D $\mathcal{Z}$-polyhedral shape `s1` described in Example 2. Listing 3.1 presents the generic executor code that can be used to perform the SpMV operation on such a 2D $\mathcal{Z}$-polyhedral shape. Given the dictionary values associated with `s1`, `(0,2,0,1,0,9,1,2,0,1,1,3,0)`, these entries can be directly mapped to the corresponding fields in the generic executor: `(shape.start[0], shape.end[0], shape.start[1], shape.end[1], orig.shape.lattice[0][0], orig.shape.lattice[0][1], orig.shape.lattice[0][2], orig.shape.lattice[1][0], orig.shape.lattice[1][1], orig.shape.lattice[1][2], _, _, _)`. Substituting these values into the executor loop structure shown in Listing 3.1 yields the specialized code presented in Listing 3.2.

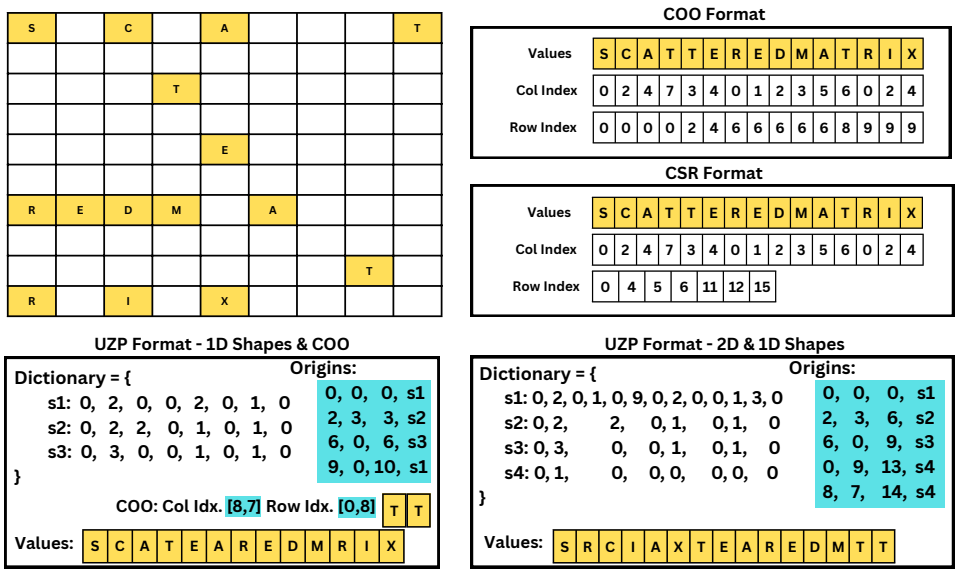

Figure 3.1: Different possible representations of the same sparse matrix: CSR, COO and two possible UZPs.

```
double* const data_vector = uzp_matrix->data;
int idx_i, idx_j;
// Retrieve data offset and shape information from origin:
int a_data_pos = orig.data_offset;
int lattice_0_0 = orig.shape.lattice[0][0];
int lattice_0_1 = orig.shape.lattice[0][1];
int lattice_1_0 = orig.shape.lattice[1][0];
int lattice_1_1 = orig.shape.lattice[1][1];
// Compute lattice increment for the inner loop:
int offset_idx_i = lattice_1_0 * shape.stride[1];
int offset_idx_j = lattice_1_1 * shape.stride[1];
// Loop nest scanning coordinates:
for (int i = shape.start[0]; i <= shape.end[0]; i += shape.stride[0]) {
    idx_i = orig.coordinates[0] + i * lattice_0_0
      + lattice_1_0 * shape.start[1] + orig.shape.lattice[0][2];
    idx_j = orig.coordinates[1] + i * lattice_0_1
      + lattice_1_1 * shape.start[1] + orig.shape.lattice[1][2];
    for (int j = shape.start[1]; j <= shape.end[1]; j += shape.stride[1]) {
        // Computation-specific operation:
        y[idx_i] += data_vector[a_data_pos] * x[idx_j];
        // Update coordinates and data offset:
        idx_i += offset_idx_i;
        idx_j += offset_idx_j;
        a_data_pos += 1;
    }
}
```

Listing 3.1: Parameterized executor for 2D shapes in SpMV.

```
offset_idx_y = 0 * 1;
offset_idx_x = 2 * 1;
for (int i = 0; i <= 1; i += 1) {
    idx_y = orig.coordinates[0] + (i * 9) + (0 * 0) + 0;
    idx_x = orig.coordinates[1] + (i * 0) + (2 * 0) + 0;
    for (int j = 0; j <= 2; j += 1) {
        y[idx_y] += data_vector[a_data_pos++] * x[idx_x];
        idx_y += offset_idx_y;
        idx_x += offset_idx_x;
    }
}
```

Listing 3.2: Executor loop after parameter substitution for shape `s1` for 2D form

Chapter 4

Tuning Mechanisms for UZP

The UZP format encodes sparse matrices through reusable $\mathcal{Z}$-polyhedral shapes and their associated origin points in the global coordinate space. Irregular regions are handled using embedded formats like CSR, CSC, COO, or BCSR. This separation of shape definition and instantiation creates a compact representation while preserving structural information.

While the construction of the UZP representation focuses on detecting and abstracting repeating patterns, it leaves the placement and ordering of shapes and irregular segments in a form that reflects the initial structure of the input matrix. As a result, there are several opportunities for post-processing the UZP representation to improve how the data is organized and later consumed by executors.

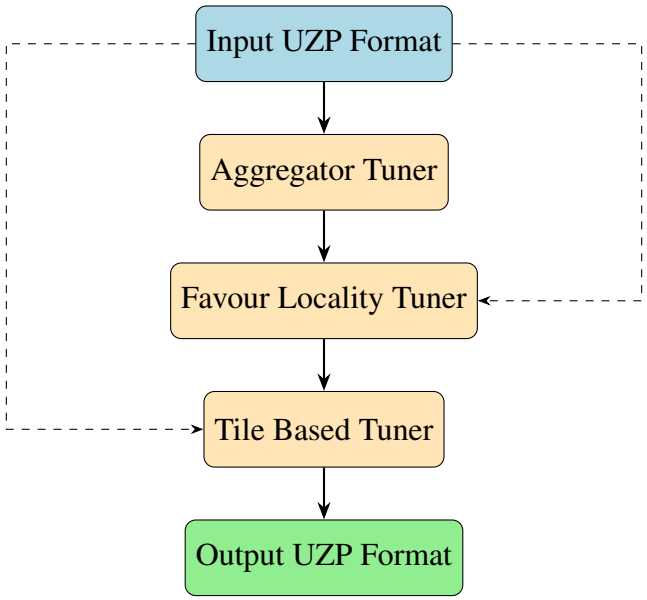

Figure 4.1: Schematic representation of tuner interaction pathways.

This section explores the space of such post-processing strategies, outlining specific transformation opportunities available within the UZP format. These may include reordering of origin lists, canonicalization of shape entries, layout normalization of irregular segments, and shape merging

when reuse is detected. Each tuning opportunity leverages the modular structure of the UZP format and aims to improve data organization or access efficiency, without changing the meaning or flexibility of the representation.

As illustrated in Figure 4.1, the UZP pipeline incorporates tuners such as `Aggregation`, `Favour Locality`, and `Tile Based`. These modular post-processing stages can operate independently or sequentially, enabling flexible optimization for various performance goals and hardware targets while maintaining the format's fundamental structure.

4.1 Aggregator Tuner: Merging Regularly Strided Origin Points

The UZP format represents sparse data structures using reusable $\mathcal{Z}$-polyhedral shapes instantiated at multiple origin coordinates. Although many origins share the same shape definition, they are not necessarily aligned in a regular geometric pattern. However, in practice, subsets of origin points often exhibit regular strides consistent with the underlying lattice structure of the shape. This observation creates an optimization opportunity: when origin points are regularly spaced according to the geometric properties of the lattice, they can be merged into a single, extended shape instance. Such merging reduces metadata redundancy and enhances spatial locality, thereby improving memory access efficiency and overall computational performance.

To exploit this property, we introduce the *Aggregator tuner*, a mechanism that identifies and merges regularly strided origin points. For each shape, the tuner iterates over the associated origin list and detects sequences of origins that follow the expected stride pattern defined by the lattice structure. When such a sequence is identified, the tuner merges the corresponding origins by extending the original shape's loop bounds, while preserving the lattice coefficients and stride parameters. This transformation effectively folds multiple shape instances into a single, larger polyhedral shape, reducing the total number of origins without modifying the semantics of the sparse representation.

To illustrate this process, consider a simple 1D $\mathcal{Z}$-polyhedral shape applied to a 2D sparse matrix. The shape iterates over four points using a loop variable $l \in [0, 3]$, with a lattice mapping

defined as:

$$(i, j) = (I_0 + 2l,\ J_0 + 3l),$$

where (I_0, J_0) denotes the origin. Instantiating the shape at an origin generates a set of nonzero coordinates determined by this affine relation.

Suppose two origins are observed at $(6, 9)$ and $(14, 21)$. Instantiating the shape at $(6, 9)$ yields the coordinates:

$$(6, 9),\ (8, 12),\ (10, 15),\ (12, 18),$$

while instantiating at $(14, 21)$ produces:

$$(14, 21),\ (16, 24),\ (18, 27),\ (20, 30).$$

It is evident that the sequence of points generated by the second origin continues the pattern initiated by the first. Rather than representing these as two distinct shape instances, the aggregator tuner merges them into a single shape by extending the loop domain to $l \in [0, 7]$, thereby generating all eight points under a unified polyhedral representation. Importantly, the underlying lattice structure and projection function remain unchanged during this merger.

While the preceding example is based on a 1D $\mathbb{Z}$-polyhedron function $(\mathcal{D}, F)$ over a 2D coordinate space, the aggregator tuner is not limited to such simple cases. The algorithm generalizes naturally to arbitrary multidimensional polyhedral domains with affine lattice projections. Merging decisions rely solely on detecting structural regularity under the shape's lattice semantics and affine iteration space. Furthermore, the merging process is inherently iterative: it continues consolidating origin points as long as new regular stride patterns are identified within the origin list, progressively compressing the overall representation.

This optimization significantly reduces the number of shape instantiations and their associated origin metadata. In addition, it improves the spatial regularity of nonzero coordinates, enhancing cache locality and enabling more effective vectorization during execution. The Aggregator tuner

thus acts as a structure-preserving refinement phase, applied after the initial UZP reconstruction, to compress and regularize the layout without modifying the semantics of the original sparse matrix.

4.1.1 Algorithm & Implementation Details

The implementation of the aggregator tuner follows the structure described in the preceding section and is realized through two modular procedures: the main tuner loop, presented in Algorithm 1, and the shape-specific origin consolidation routine, described in Algorithm 2.

Algorithm 1 performs the top-level traversal over all shapes in the input UZP representation $\mathcal{U}$. For each shape S, it constructs a spatial index over the associated origin list and computes a merge map M that identifies origin points exhibiting regular stride patterns aligned with the shape's lattice. Regularity is determined based on lattice-consistent translations, verified through spatial queries over the origin index. Once the merge map is populated, the shape S and its corresponding map M are passed to the MERGEUSINGMAP procedure for consolidation.

The merging logic is encapsulated in Algorithm 2. This procedure first iteratively applies the merge map to combine the data pointers associated with matched origin pairs. It then groups merged origin blocks that share identical lattice configurations and constructs new shapes with extended loop bounds reflecting the enlarged spatial extent covered by the combined origins. Origins that do not participate in any merge are preserved in their original form and included in the final output.

4.1.2 Impact Evaluation of Aggregator Tuner

We evaluate the impact of the Aggregator Tuner within the UZP framework using a benchmark suite of 229 sparse matrices drawn from the SuiteSparse collection [16]. The evaluation focuses on multiple performance metrics, including execution time, computational throughput (GFLOPs), cache efficiency, SIMD utilization, and memory footprint. Specifically, we assess the Aggregator Tuner's effect on SpMV performance by analyzing throughput variations, reductions in the number of origin points, and changes in the base polyhedral shape count. All results are reported relative

Algorithm 1 Aggregator Tuner for UZP

procedure AGGREGATORTUNER($\mathcal{U}$) ▷ $\mathcal{U}$ is the input UZP representation
  Initialize $\mathcal{U}_{\text{new}} \leftarrow \emptyset$ ▷ New UZP container
  for all $S \in \mathcal{U}$.shapes **do**
    Let $n \leftarrow |\text{origins}(S)|$
    Initialize merge map $M \in \mathbb{Z}^n,\ M[i] \leftarrow -1$
    Let $\mathcal{O} \leftarrow$ spatial index structure over S.origins ▷ e.g., hash map or kd-tree
    for $i \leftarrow 0$ to $n-1$ **do**
      Let $\mathbf{o}_i \leftarrow S.\text{origins}[i]$ ▷ $\mathbf{o}_i \in \mathbb{Z}^d$
      Let $\mathcal{L} \leftarrow S$.lattice ▷ $\mathcal{L} \in \mathbb{Z}^{d \times m}$
      Let $\mathbf{t} \leftarrow$ vector of loop counters from polyhedral domain
      Let $\hat{\mathbf{o}} \leftarrow \mathbf{o}_i + \mathcal{L} \cdot \mathbf{t}$
      Let $j \leftarrow$ SEARCHORIGIN($\mathcal{O}, \hat{\mathbf{o}}$) ▷ Multidimensional search
      if j exists **then**
        $M[i] \leftarrow j$
      end if
    end for
    Let $\texttt{listS}_{\text{new}} \leftarrow$ MERGEUSINGMAP(S, M) ▷ List of merged shapes
    $\mathcal{U}_{\text{new}} \leftarrow \mathcal{U}_{\text{new}} \cup \texttt{listS}_{\text{new}}$
  end for
  return $\mathcal{U}_{\text{new}}$
end procedure

to a baseline UZP configuration (*UZP Standard*), allowing a direct quantification of the structural compression and performance improvements introduced by the Aggregator Tuner.

Algorithm 2 MERGEUSINGMAP: Merge Origin Points Using Index Map

procedure MERGEUSINGMAP(S, M)
    Let $\mathcal{O} \leftarrow S$.origins, $n \leftarrow |\mathcal{O}|$ ▷ $\mathcal{O}_j$ is the j-th origin struct; $M[j]$ is its merge target index
    for $j \leftarrow n-1$ **down to** 0 **do**
        if $M[j] \neq -1$ **then**
            Let $k \leftarrow M[j]$
            $\mathcal{O}_j$.dataptr $\leftarrow \mathcal{O}_j$.dataptr $\cup$ $\mathcal{O}_k$.dataptr
            $\mathcal{O}_k$.dataptr $\leftarrow \emptyset$
        end if
    end for
    Initialize $\texttt{listS}_{\text{new}} \leftarrow \emptyset$
    for $j \leftarrow 0$ **to** $n-1$ **do**
        if $|\mathcal{O}_j$.dataptr$| > 0$ **then**
            Let $\mathcal{L} \leftarrow S.\mathcal{L}$
            Let $t \leftarrow |\mathcal{O}_j$.dataptr$|$
            Let $\vec{e}_{\text{new}} \leftarrow S.\vec{e} + (t-1) \cdot \mathcal{L}$
            Try to find $S' \in \texttt{listS}_{\text{new}}$ such that:
                $S'.\vec{e} = \vec{e}_{\text{new}}$ and $S'.\mathcal{L} = \mathcal{L}$
                if such S' exists **then**
                    Append $\mathcal{O}_j$ to S'.origins
                else
                    Create new shape $S_{\text{new}} \leftarrow$ clone of S
                    Set $S_{\text{new}}.\vec{e} \leftarrow \vec{e}_{\text{new}}$
                    Set S_{new}.origins $\leftarrow \{\mathcal{O}_j\}$
                    Add S_{new} to $\texttt{listS}_{\text{new}}$
                end if
        end if
    end for
    ▷ Handle origins that were not part of any merge
    Create shape $S_{\text{original}} \leftarrow$ clone of S with empty origin list
    for $j \leftarrow 0$ **to** $n-1$ **do**
        if $M[j] = -1$ **and** $|\mathcal{O}_j$.dataptr$| == 1$ **then**
            Append $\mathcal{O}_j$ to S_{original}.origins
        end if
    end for
    if $|S_{\text{original}}$.origins$| > 0$ **then**
        Add S_{original} to $\texttt{listS}_{\text{new}}$
    end if
    return $\texttt{listS}_{\text{new}}$
end procedure

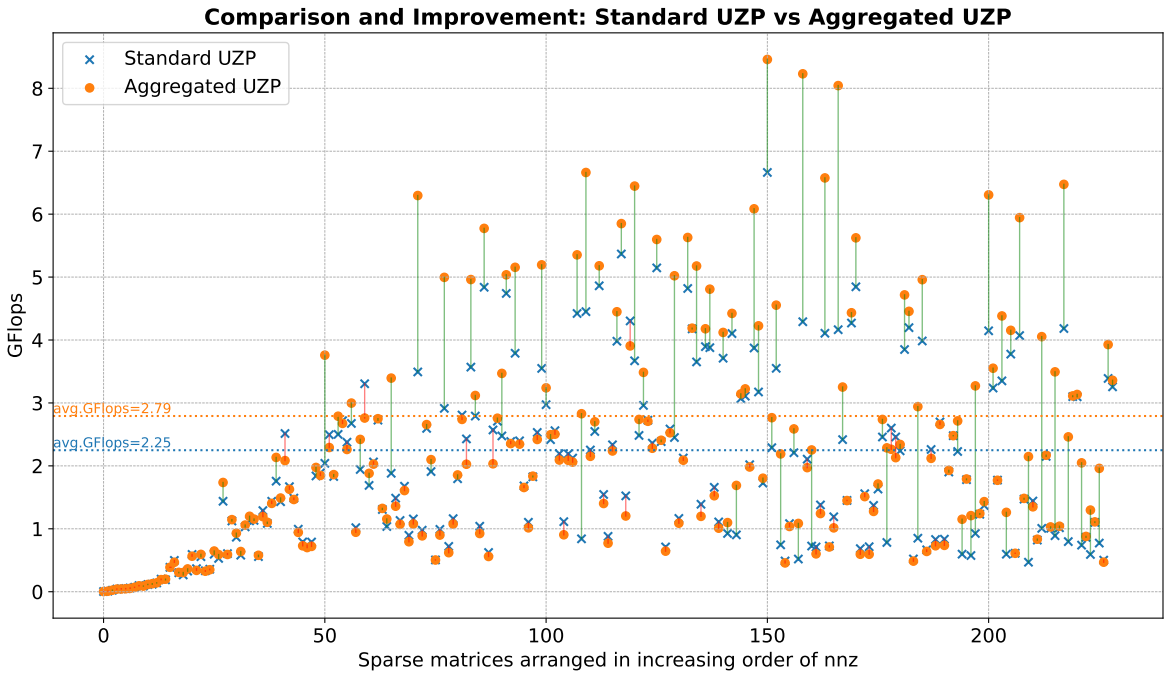

Figure 4.2: Performance comparison of Standard UZP vs Aggregator Tuner UZP across 229 sparse matrices ordered by non-zero count, highlighting gains from Aggregator tuner.

Figure 4.2 presents the comparative performance, measured in GFLOPs, for 229 sparse matrices ordered by increasing nonzero count (nnz). Note, the full experimental setup is described in Chapter 5

The analysis shows that the Aggregated UZP configuration consistently outperforms the Standard UZP baseline across a wide range of matrices. On average, Aggregated UZP achieves 2.79 GFLOPs compared to 2.25 GFLOPs for Standard UZP, corresponding to a 24% improvement in computational throughput. The performance gains are particularly pronounced for matrices with larger nnz values, where the benefits of origin consolidation and improved memory locality become more significant.

The primary factors contributing to the observed performance variation are the reduction in the number of origins and the increase in the size of the base polyhedral shapes. Figures 4.3 and 4.4 present the changes in the number of origins and base shapes per sparse matrix before and after applying the Aggregator Tuner.

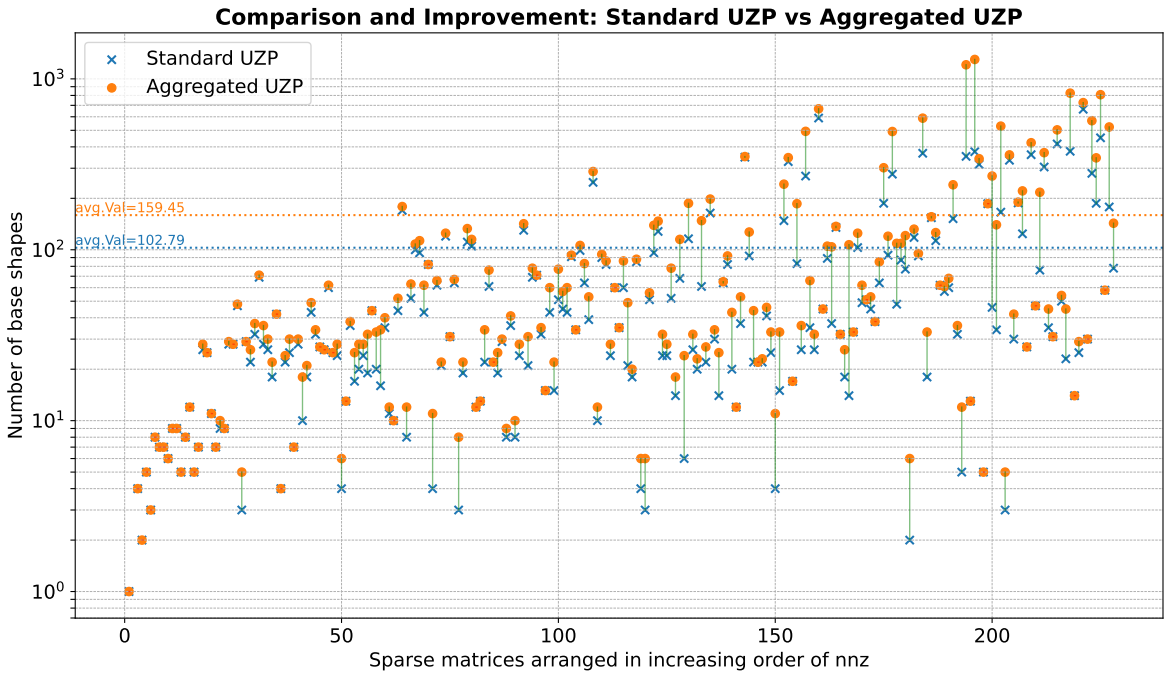

Figure 4.3: Plot showing Base shape count of Standard UZP vs Aggregator Tuner UZP across 229 sparse matrices ordered by non-zero.

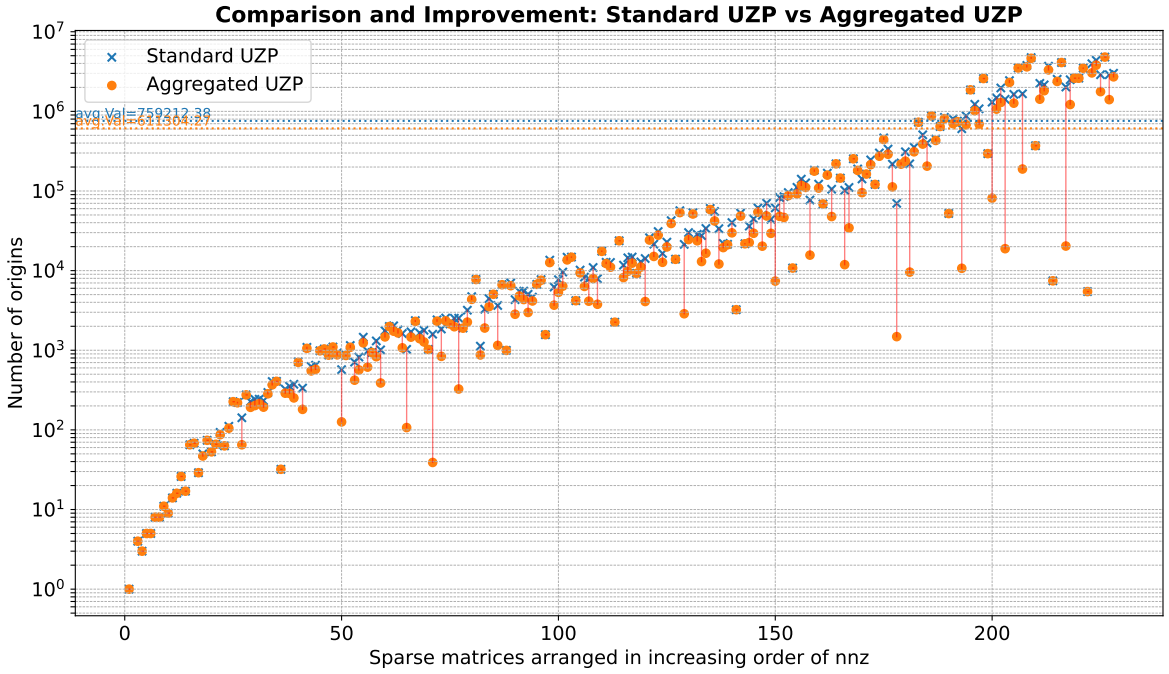

Figure 4.4: Plot showing Origins count of Standard UZP vs Aggregator Tuner UZP across 229 sparse matrices ordered by non-zero.

Figure 4.4 shows that, for most larger matrices, there is a substantial reduction in the number of origins after applying the Aggregator Tuner. This reduction decreases the associated metadata overhead and improves performance by enabling the merger of contiguous points. As a result, more iterations are executed within individual polyhedral shapes, leading to better loop regularity and improved computational efficiency.

4.2 Reordering Origins for Locality Optimization

Beyond structural compression achieved through origin merging, spatial and temporal data locality are critical factors influencing execution performance in sparse matrix operations. In particular, access locality directly affects cache utilization during memory-bound computations such as Sparse Matrix-Vector Multiplication (SpMV) and Sparse Matrix-Matrix Multiplication (SpMM).

The prior Aggregator Tuner 4.1 primarily targets structural contiguity by merging origin points into larger polyhedral shapes. While this strategy effectively reduces metadata overhead and improves computational regularity, it does not explicitly account for cache behavior. In some cases, aggressive merging can produce oversized shapes whose memory footprint exceeds cache capacity. Additionally, merged shapes may span non-contiguous memory regions if the origin points are not spatially clustered, potentially degrading memory access performance due to reduced locality.

To address the limitations of purely structural merging, we introduce a locality-aware reordering strategy. This tuner incorporates two mechanisms aimed at improving spatial data locality while maintaining manageable shape sizes:

1. **Merge Limiting via Thresholding:** A user-defined threshold is applied to restrict the number of origin points that can be aggregated into a single polyhedral shape. Once this threshold is reached, the current sequence of origins is finalized, and subsequent origins are processed separately. This constraint ensures that the spatial extent of each shape remains within a cache-friendly region.

2. **Block-wise Reordering by X-Axis Locality:** To further enhance memory coalescing, origin coordinates are grouped based on there polyhedral shapes and reordered based on their X-

axis position. For each polyhedral shape independently, the associated list of origin points is partitioned into spatial blocks of size G, defined as:

$$b_i = \left\lfloor \frac{x_i}{G} \right\rfloor$$

where x_i denotes the X-axis coordinate of origin $\mathbf{o}_i$ Algorithm 3. A stable sort is then applied over the block indices b_i, ensuring that origins belonging to the same spatial region are clustered in memory. This reordering is performed separately for the origins associated with each shape, rather than across all shapes globally. As a result, even if multiple shapes are processed sequentially during execution, the working set corresponding to each shape remains spatially compact, enabling better utilization of the CPU cache. Although full global contiguity is not guaranteed, limiting data movement to a small number of shape accesses significantly improves effective cache residency and reduces memory access overhead.

The complete reordering procedure is outlined in Algorithm 3. The method is lightweight, requires no matrix-specific tuning, and can be applied broadly across diverse sparse workloads.

Algorithm 3 Origin Reordering for X-Axis Locality

Require: Origin list $\mathcal{O} = \{\mathbf{o}_1, \mathbf{o}_2, \ldots, \mathbf{o}_n\}$, with $\mathbf{o}_i = (y_i, x_i)$

Require: Grouping threshold parameter $G \in \mathbb{Z}_{>0}$

Ensure: Reordered list $\mathcal{O}'$ optimized for X-axis locality

1: Assign block index: $b_i \leftarrow \lfloor x_i / G \rfloor\,, \quad \forall\, \mathbf{o}_i \in \mathcal{O}$

2: Construct block mapping: $\mathcal{B} = \{(b_i, \mathbf{o}_i) \mid \mathbf{o}_i \in \mathcal{O}\}$

3: Sort $\mathcal{B}$ by block index b_i using a stable sorting algorithm

4: Extract reordered origins: $\mathcal{O}' \leftarrow$ projection of $\mathcal{B}$ onto $\mathbf{o}_i$

5: **return** $\mathcal{O}'$

4.2.1 Impact Evaluation of Favour Locality Tuner

Building upon the previous evaluation setup, we assess the impact of the Favour Locality Tuner on SpMV performance using the same benchmark suite of 229 sparse matrices from the SuiteSparse collection [16]. For this the full experimental setup is described in Chapter 5. This evaluation focuses on improvements in memory access locality, computational throughput (GFLOPS), and changes in the structural organization of origins and base shapes. All results are reported relative to the baseline UZP configuration (*UZP Standard*).

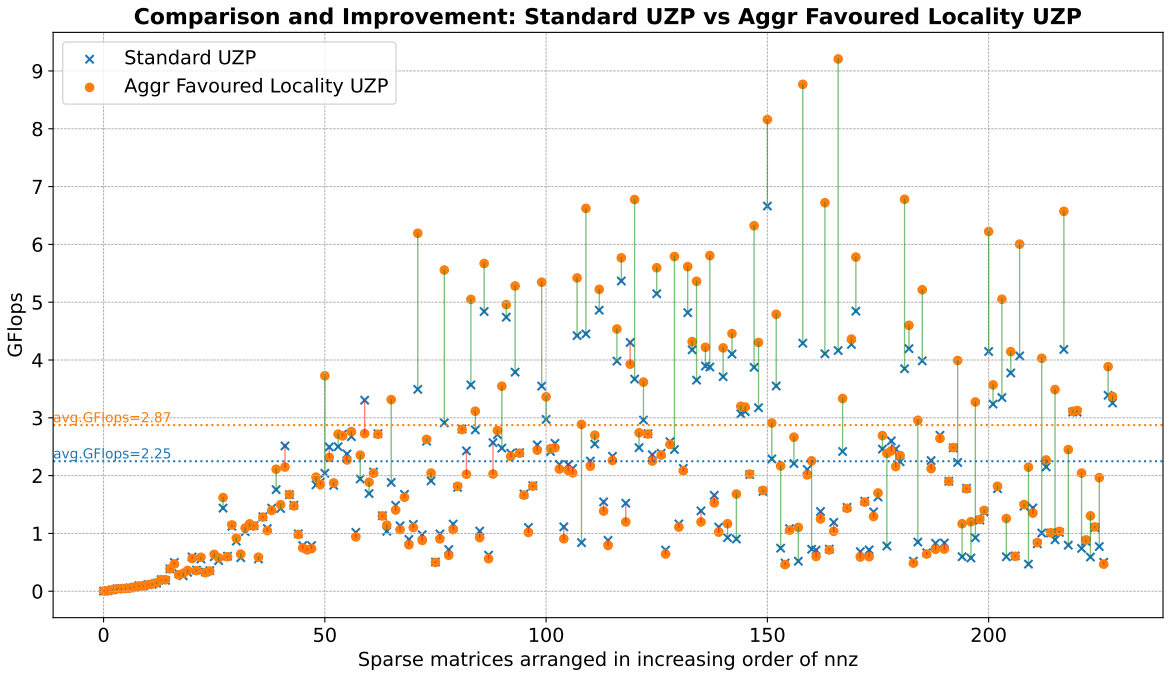

Figure 4.5: Performance comparison of Standard UZP vs Favour Locality Tuner UZP across 229 sparse matrices sorted by increasing non-zero count, highlighting gains from locality-aware tuning.

Figure 4.5 presents the comparative performance in GFLOPS for 229 sparse matrices, ordered by increasing number of non-zero elements.

The performance analysis shows that the Favour Locality Tuner achieves further improvements over the Standard UZP configuration by explicitly promoting spatial access locality. On average, the Favour Locality Tuner achieves 2.87 GFLOPS compared to 2.25 GFLOPS for Standard UZP, representing an average improvement of 28%. The performance gains are particularly notable for

matrices with larger nnz values, where access locality has a more pronounced impact on cache behavior and memory bandwidth utilization.

In addition to performance improvements, the structural impact of the Favour Locality Tuner is examined. Figures 4.6 and 4.7 present the variations in the number of base shapes and origins, respectively, before and after applying the locality-aware reordering. While from the Fig 4.6 we can see that the Avg. number of base shape per matrices is increased from 102.79 to 126.97 which better when compared to Aggregator tuner case 4.1, the number of origins shows a notable decrease compared to the Standard UZP configuration, reflecting improved aggregation efficiency while preserving spatial locality.

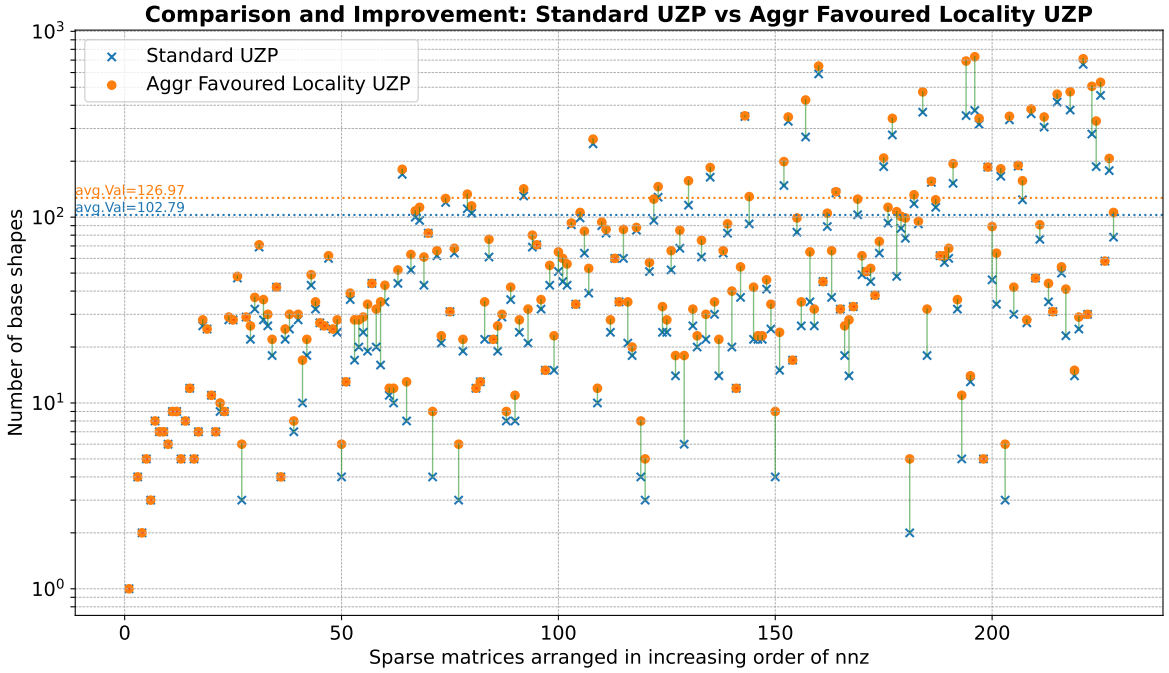

Figure 4.6: Base shape count of Standard UZP vs Favour Locality Tuner UZP across 229 sparse matrices sorted by increasing non-zero count.

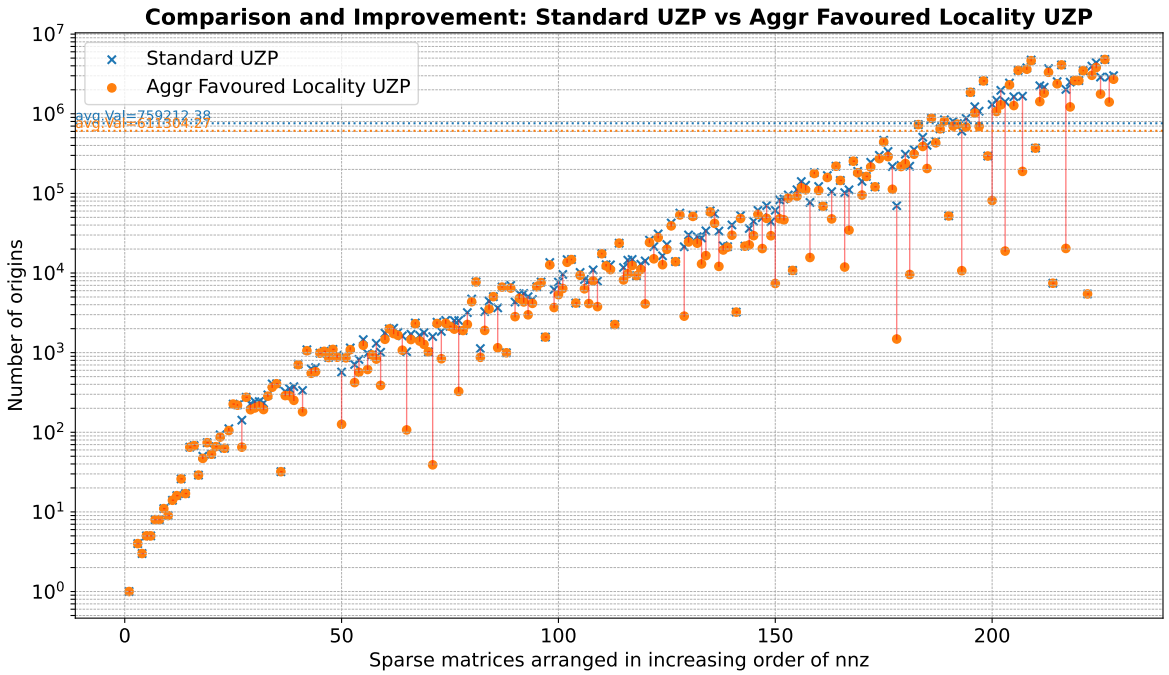

Figure 4.7: Origin count of Standard UZP vs Favour Locality Tuner UZP across 229 sparse matrices sorted by increasing non-zero count.

The observed performance improvement is attributed to two primary factors: (1) maintaining manageable number of shapes and also shape sizes through limiting merge process prevents cache overflow, and (2) improving the spatial clustering of origins listed under each shape, which enhances memory coalescing during loop execution. By ensuring that origins belonging to the same shape are reordered based on their X-axis locality, the tuner reduces cache misses, branch misprediction and improves effective cache residency, particularly in memory-bound scenarios.

4.3 Tile-Based Origin Reordering for Locality Optimization

To further enhance memory access locality in sparse matrix-vector multiplication (SpMV), we introduce a tiling-based origin reordering strategy within the UZP framework. The tiling-based reordering tuner aims to improve memory access locality by reorganizing origin points in the UZP format. Typically, for medium to large matrices where irregularity and sparsity are more pronounced, the list of origins often scales to millions of entries and must be efficiently managed to maintain runtime performance. For example, in the `msdoor/INPRO` matrix from the SuiteSparse collection, sized $415{,}863 \times 415{,}863$ with over 20 million nonzeros, the initial UZP representation of this matrix yields 138 distinct base polyhedral shapes that collectively capture 99.37% of the nonzero entries, yet the total number of mined origin points still stay at 2.7 million.

To improve memory locality, the tuner introduces a tiling mechanism that partitions the matrix domain into user-defined tile blocks. Each origin is assigned to a tile based on its (y, x) coordinates, and within each tile, origins are lexicographically sorted in the order of $\texttt{shape\_id}, y, \&x$. This localized grouping facilitates spatial clustering of nonzeros, allowing the corresponding data values to be laid out contiguously in memory. In contrast to the locality-favoring tuner, which primarily focuses on optimizing reuse of the x-vector without considering y-vector contention, the tiling-based reordering inherently accounts for both x- and y-axis access patterns. The row-major traversal of tiles enhances cache alignment, reduces data movement overhead, and improves hardware prefetching efficiency.

By preserving the semantics of the original sparse matrix while reorganizing its internal layout, the tiling-based transformation improves computational throughput on modern memory-bound architectures. Furthermore, this reordering provides a foundation for subsequent optimizations, such as loop tiling, SIMD vectorization, and NUMA-aware thread scheduling. Although the current tuner does not explicitly assign tiles to threads, the resulting memory-friendly layout is particularly well-suited for parallel SpMV kernels, where minimizing cache contention on the y-vector is critical for achieving scalability.

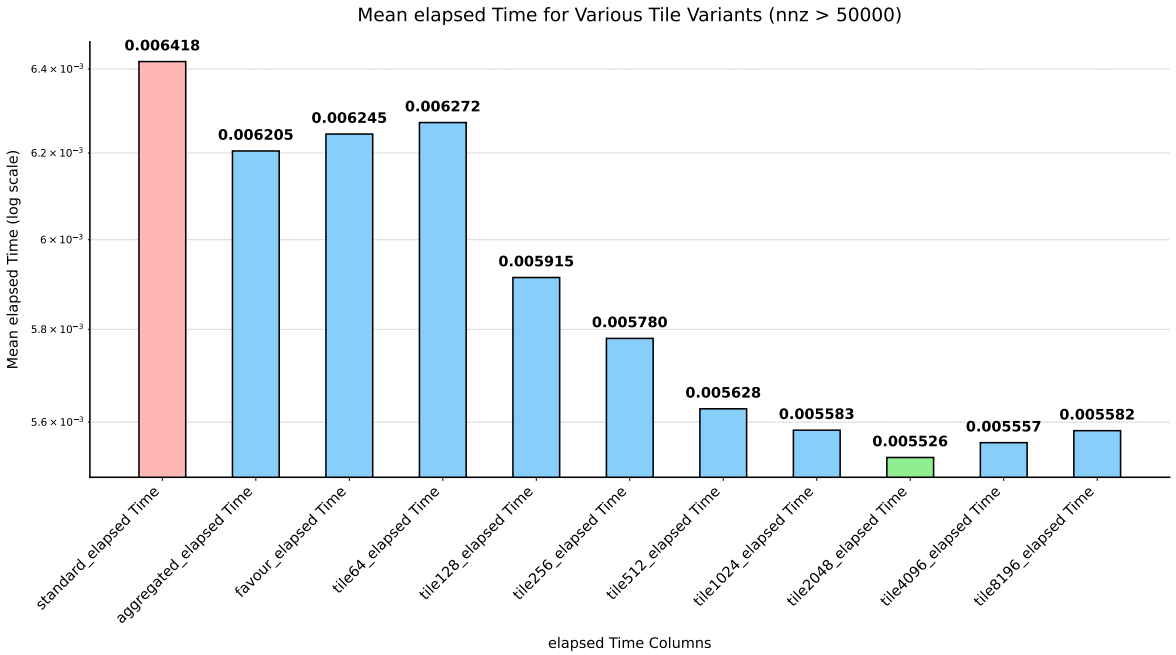

Figure 4.8: Illustrates average elapsed time for SpMV across Standard UZP and Tile Tuner UZP variants for matrices with nnz > 50K.

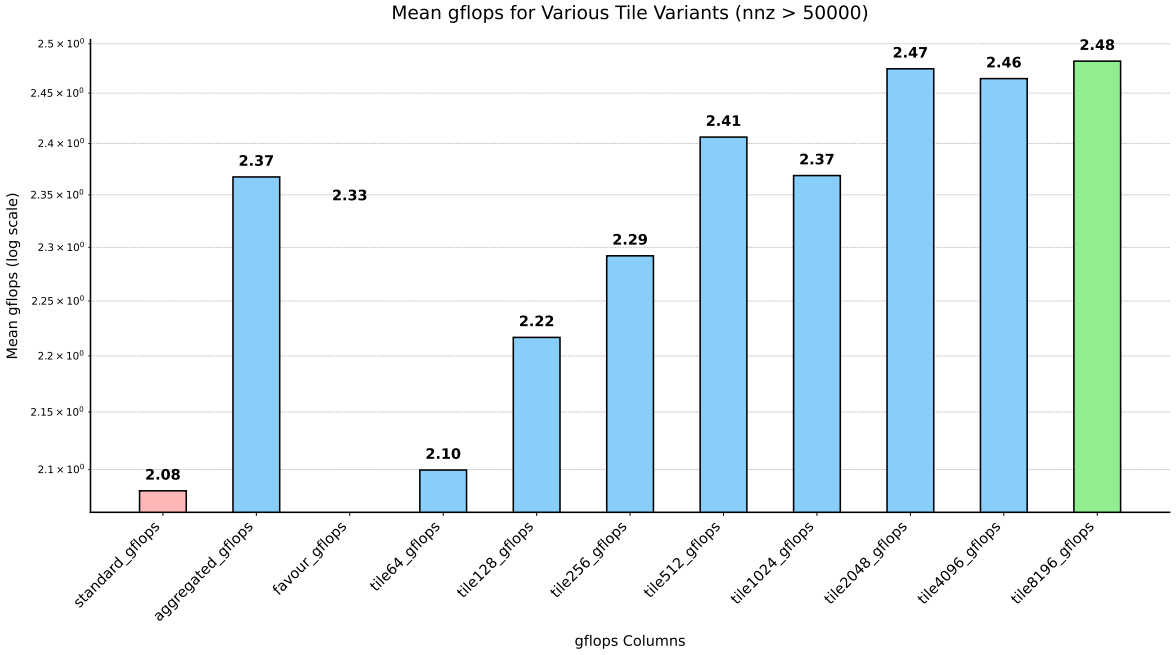

Figure 4.9: Illustrates average Gflops achieved for SpMV across Standard UZP and Tile Tuner UZP variants for matrices with nnz > 50K.

We evaluated the impact of tile-based tuning on sparse matrix-vector multiplication (SpMV) performance across different tile sizes ranging from 64 to 8196. The analysis primarily focuses on matrices with a nonzero count (nnz) greater than 50,000 to ensure sufficient computational workload.

Figure 4.8 presents the mean elapsed time across the variants. The standard and aggregated configurations exhibit the highest elapsed times, with the standard configuration reaching approximately 6.4×10^{-3} seconds. As the tile size increases, the elapsed time decreases consistently, with configurations such as `tile1024`, `tile2048`, and `tile4096` achieving the lowest times around 5.5×10^{-3} seconds. The results indicate that tiling improves memory access regularity and reduces overhead. However, beyond `tile2048`, further increases in tile size show minimal additional improvement, suggesting a performance saturation point.

Figure 4.9 shows the corresponding mean Gflops achieved for the same set of configurations. Tile-based variants achieve higher Gflops compared to the standard and aggregated baselines. In particular, `tile2048`, `tile4096`, and `tile8196` configurations reach mean Gflops values around 2.5, compared to approximately 2.1 for the standard case. The increase in Gflops with larger tile sizes indicates better computational throughput and improved utilization of available memory bandwidth. Similar to the elapsed time trends, Gflops performance stabilizes beyond `tile2048`, indicating that tiling beyond this point does not yield significant additional benefits. As a note, all these experiments were run for single thread double precision and cold cache configuration.

Overall, tile-based tuning is effective in improving SpMV performance for matrices with nnz greater than 50,000. Configurations with tile sizes of 512 and larger consistently outperform the baseline variants. Further improvements can be explored by refining tile sizes within the range of 512 to 2048 and profiling cache behavior to optimize memory hierarchy utilization. Adaptive tiling strategies based on matrix sparsity patterns could also be investigated to enhance performance across diverse matrices. The Figures 4.10 & 4.11 shows the average No. of shapes and origins count stored in the UZP across all variants.

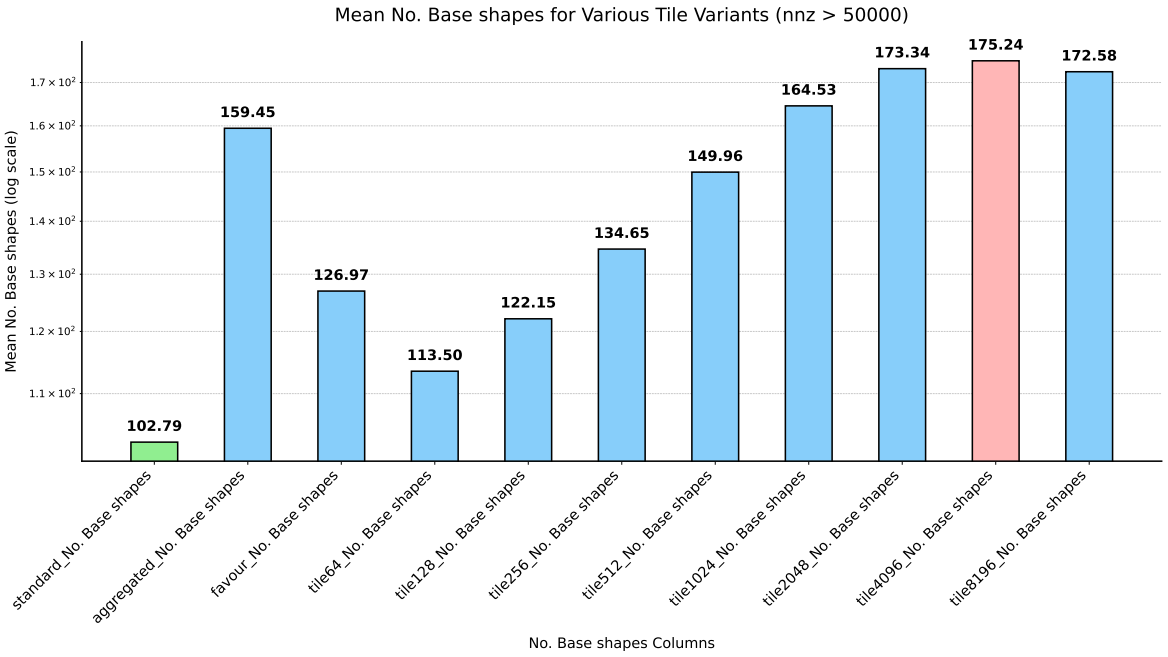

Figure 4.10: Illustrates average number of base shapes in SpMV across Standard UZP and Tile Tuner UZP variants for matrices with nnz > 50K.

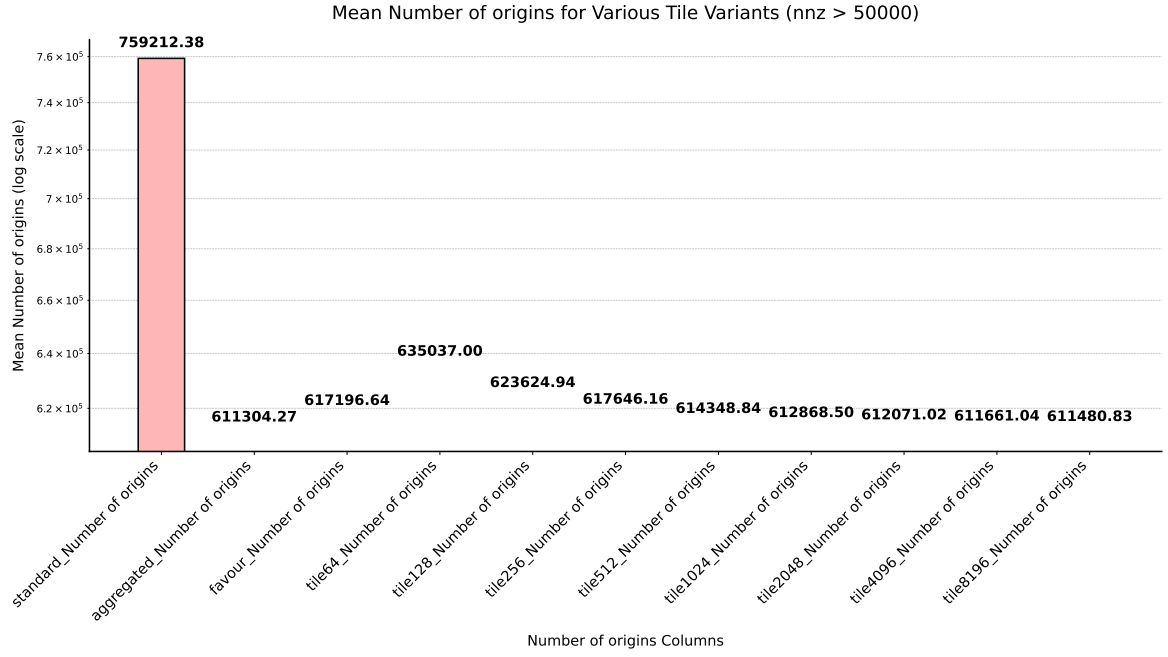

Figure 4.11: Illustrates average number of origins in SpMV across Standard UZP and Tile Tuner UZP variants for matrices with nnz > 50K.

4.4 Generic Executors for Polyhedral Sparse Representations

Prior work leveraging $\mathbb{Z}$-polyhedra to encode sparse structures primarily focused on generating sparsity-specific programs tailored to a fixed computation [2, 18]. In contrast, the UZP format seeks to overcome this limitation by introducing a more flexible approach: allowing the construction of *generic executors* that, similar to traditional sparse formats, are independent of the computation or sparsity pattern. These executors can be specialized or tuned for particular hardware targets or optimization goals.

A key feature of UZP is its ability to leverage coordinate compression through $\mathbb{Z}$-polyhedra, while supporting the construction of generic executors applicable to a range of computations, including SpMV as discussed in Sec. 5.3. These executors operate over all nonzero coordinates using (strided) dense loops and are amenable to optimization by polyhedral compilation frameworks. Unlike approaches that rely on computation-specific specialization, UZP-based executors perform a full traversal of the nonzero structure in a uniform and analyzable manner.

An important contrast with the generation of sparsity-specific code specialized for a particular computation is the independence of the executor binary size from the size of the sparse structure, since now the coordinates to operate on are externalized in a separate UZP data structure. It therefore alleviates the issues with potential binary size explosion and stress of the instruction cache [2, 18], however at the cost of some performance potential lost by lack of code specialization Horro et al. [18], as shown below.

In this section, we present generic executor framework, designed to perform sparse matrix-vector (SpMV) and sparse matrix-matrix (SpMM) operations on matrices represented in the UZP format. The executor demonstrates how UZP enables a structured and compiler-friendly code generation strategy, allowing general-purpose compilers to achieve effective SIMD vectorization with minimal manual intervention.

Our executor implements a specialized SpMV kernel tailored for UZP-encoded matrices, where the matrix is decomposed into a collection of structured polyhedral blocks. Each block defines a multidimensional iteration space characterized by lattice-based strides and origin points. The ex-

ecutor dynamically dispatches computation over these blocks by generating loop nests that match the dimensionality (1D, 2D, or 3D) of the underlying polyhedral shape.

The key innovation lies in our structured code generation approach, which explicitly targets the characteristics of UZP encoded sparse matrices. By preserving affine loop structures and exposing regular memory access patterns, our strategy enables generic compilers such as GCC and Clang to more effectively recognize and apply SIMD vectorization. Unlike traditional sparse formats, UZP facilitates the injection of minimal yet strategic structural hints, making it possible to generate high-performance code through standard compilation pipelines without extensive manual tuning or architecture-specific optimization.

4.4.1 Design Principles of the Generic Executors

The generic executor in UZP is designed to traverse all nonzero coordinates through a two-step process: (1) enumerating all origin points, and (2) scanning the corresponding polyhedral shape associated with each origin using a loop nest to reconstruct the original coordinates and apply the desired computation.

Although it is theoretically feasible to construct a fully generic parametric polyhedral loop nest capable of handling any shape, such an approach typically hinders the effectiveness of low-level compiler optimizations in C/C++. To address this, UZP adopts a more practical strategy: it instantiates a finite set of specialized loop nest templates, one per dimensionality case. This set remains tractable in practice, as the format only supports shapes up to 8 dimensions, and models sparse tensors of dimensionality ranging from 1D to 4D. This leads to a total of 80 loop nest cases.

The listing below illustrates a representative executor implementation for a 2D shape in the context of SpMV. This executor is parameterized based on the lattice structure and performs strength reduction to simplify induction variable generation, facilitating loop analysis and optimization by the compiler. For clarity, the example assumes that the data vector layout aligns with a unit lattice for computing data positions.

It is worth noting that for computations requiring a single traversal of the sparse matrix, without constraints on the order of access, the generic executor is directly applicable. The computation-specific operation can be encapsulated in an inlinable function or macro, utilizing `idx_i`, `idx_j`, and `a_data_pos` to access the appropriate coordinates and data values.

To improve performance, the executor design leverages the C/C++ compiler's ability to optimize regular loop structures. In particular, to facilitate compile-time analysis and enable auto-vectorization of the strided loops, the executor is *specialized* for distinct combinations of loop strides and lattice parameters. This approach exposes sufficient structural information to the compiler, allowing for more aggressive low-level optimizations.

An example of such a specialization is shown below for a 1D shape corresponding to a dense vertical vector of coordinates. This case assumes a unit stride and a lattice aligned along the row dimension:

```
1  // Specialized 1D shape #0:
2  // lattice[0] = 1, lattice[1] = 0, lattice[2] = 0,
3  // stride[0] = 1, start[0] = 0, arbitrary end[0] supported.
4  int idx_i = orig.coordinates[0];
5  int idx_j = orig.coordinates[1];
6  for (int i = 0; i <= shape.end[0]; i += 1) {
7    // Computation-specific operation:
8    y[idx_i] += data_vector[a_data_pos] * x[idx_j];
9    // Update coordinates and data offset:
10   idx_i += 1;
11   a_data_pos += 1;
12 }
```

Listing 4.1: Specialized executor for 1D vertical vector shape.

To accommodate performance-critical patterns, specialized executors have been developed for both 1D and 2D shapes, covering dense and strided cases across horizontal, vertical, and diagonal

directions. Each shape entry in the dictionary is associated with a unique `int` hash key that identifies the corresponding specialized implementation. At runtime, a simple `switch` statement on this key dispatches execution to the appropriate specialized routine when available; otherwise, it falls back to the generic executor described earlier.

To further enhance performance, versioning and semi-manual SIMD vectorization are employed to bypass the compiler's conservative cost models in some cases. In this work, we restrict ourselves to a limited set of explicitly versioned shapes, relying on GCC's auto-vectorization capabilities to optimize these routines. The primary focus is on evaluating the baseline performance of minimal loop forms necessary to implement correct UZP executors, thereby establishing a lower-bound performance reference that highlights the intrinsic limitations and optimization potential of the format.

4.4.2 Further Optimization via Auto-Generated Shape-Specific Loops

At the core of the UZP code generation framework lies the `run_o2dtype>` Listing 4.2 executor, a statically compiled, shape-dispatching function designed to support efficient computations over structured multidimensional iteration spaces. This executor serves as a runtime dispatcher that selects among pre-generated loop nests specialized for distinct polyhedral shapes captured in the UZP representation.

Each polyhedral shape mined during the UZP encoding process is assigned a unique `shape_id` and is associated with metadata encapsulated in the `shape_reg` structure. Where this structure holds a metadata that includes the shape's multidimensional lattice basis, iteration bounds, and stride values, which collectively define the affine iteration space required to reconstruct the sparse matrix's nonzero coordinates.

Building on this foundation, we implement an automated code generation system that emits statically defined loop macros for each unique polyhedral shape. When the matrix structure is known at compile time and is immutable, we exploit this static information to pre-generate loop-optimized code blocks for each shape instance. These loop nests are parameterized and inlined

into the executor via C macros, which are compiled into a centralized header file. The executor dispatches execution via a lightweight `switch`-based mechanism that selects the appropriate loop nest macro for a given `shape_id`. An example dispatcher definition is shown below:

```
#define fundecl_run_shape_o2d_multitype(datatypename) \
static inline \
void run_shape_o2d_##datatypename(s_spf_structure_t* restrict spf_matrix, \
                                  int shape_id, int start_offset, \
                                  int end_offset, \
                                  const int* restrict coord_y, \
                                  const int* restrict coord_x, \
                                  const int dataptr, \
                                  const datatypename* const restrict x, \
                                  datatypename* restrict y) \
{ \
    const datatypename* const data_vector = spf_matrix->data; \
    int a_data_pos = dataptr; \
    switch (shape_id) { \
        case 0: LOOP_SHAPE_0; break; \
        case 1: LOOP_SHAPE_1; break; \
        case 2: LOOP_SHAPE_2; break; \
        ... 
        case 11: LOOP_SHAPE_11; break; \
        default: break; \
    } \
}
```

Listing 4.2: Dispatcher definition for shape-specialized execution.

Each generated loop macro (e.g., `LOOP_SHAPE_0`, `LOOP_SHAPE_1`, etc.) represents a fully unrolled inner loop that performs fused multiply-accumulate operations over a fixed number of

lattice-based iterations. The outer loop iterates over origin entries using input coordinate arrays `coord_y[]` and `coord_x[]` while maintaining a linear update of destination and source indices. Here is an example of such a specialized loop nest:

```
#define LOOP_SHAPE_0 \
_Pragma("GCC ivdep") \
for (int j = start_offset; j < end_offset; j++) \
{ \
    int idx_y = coord_y[j]; \
    int idx_x = coord_x[j]; \
    _Pragma("unroll") \
    for (int i = 0; i <= 2; ++i) \
    { \
        y[idx_y] += data_vector[a_data_pos++] * x[idx_x]; \
        idx_y += 1; \
        idx_x += 1; \
    } \
} \
```

Listing 4.3: Macro definition for LOOP_SHAPE_0

By externalizing these shape-specialized loops, we reduce control-flow complexity and eliminate loop-carried dependencies that might hinder vectorization in the generic case. This design effectively transforms the SpMV executor into a shape-aware JIT-like system at compile time, where each shape is associated with a statically generated SIMD-friendly loop nest. Compiler directives such as `_Pragma("GCC ivdep")` and `_Pragma("unroll")` are strategically inserted to assist the compiler in performing aggressive vectorization and loop unrolling, bypassing cost model limitations.

This approach makes it easier for backends like GCC and Clang to detect vectorization opportunities, as the loops are fully predictable, stride-1, and aligned with the SIMD execution model of modern CPUs. Furthermore, this technique decouples the computation logic from runtime shape interpretation, thus allowing additional compiler-level tuning passes such as software pipelining,

register prefetching, and cross-loop fusion. Since each macro is generated with shape-specific iteration bounds and lattice directions, the generated code can be directly tailored for shape-locality grouping and cache-aware blocking strategies.

While this work focuses on generating a limited set of such specializations and relies on auto-vectorization through GCC, the overall framework is extensible to more advanced code generation strategies, including hardware-specific tuning, explicit SIMD intrinsics, or polyhedral scheduling tools. Our macro-driven strategy significantly enhances SIMD utilization for sparse kernels by enabling compilers to emit vector instructions even for sparsely structured loops, as long as the iteration space and memory accesses remain statically analyzable. This approach bridges the gap between domain-specific loop generation and low-level SIMD ISA exploitation, setting the stage for future extensions such as AVX-512 adaptation, instruction prefetch hints, and autovectorization report feedback loops integrated into the code generator.

4.4.3 Fine-Grained Loop Specialization Using Per-Origin Dispatch

As a refinement to our earlier shape-specialized code generation strategy, we introduce a more granular macro-based execution model in which loop bodies are now dispatched per-origin rather than over a batch of grouped origins. In the previous implementation, a fixed set of coordinates (`coord_x[]` and `coord_y[]`) and shape identifiers were processed in a range-based loop across all origins sharing the same shape ID. Each shape-specific macro (`LOOP_SHAPE_k`) was applied uniformly across this batch, relying on spatial locality within the group for optimization opportunities.

```
#define fundecl_run_shape_o2d_multitype(datatypename) \
static inline  \
void run_shape_o2d_##datatypename(s_spf_structure_t* restrict spf_matrix, \
                                  s_origin_2d_t orig, \
                                  datatypename* restrict x, \
                                  datatypename* restrict y) \
{ \
    const datatypename* const data_vector = spf_matrix->data; \
    int a_data_pos = dataptr; \
    switch (shape_id) { \
        case 0: LOOP_SHAPE_0; break; \
        case 1: LOOP_SHAPE_1; break; \
        case 2: LOOP_SHAPE_2; break; \
        ... 
        case 11: LOOP_SHAPE_11; break; \
        default: break; \
    } \
}
```

Listing 4.4: Dispatcher definition for shape-specialized execution.

In contrast, the updated design shifts to an **origin-level dispatch model**, where each origin is processed individually using its associated shape macro. The dispatcher, `run_shape_o2d_<type>` Listed 4.4, now accepts a single origin descriptor `s_origin_2d_t` containing the starting coordinate, data pointer offset, and shape ID. Based on this shape ID, the function invokes one of several statically defined macro templates (Example: for the sparse matrix named apache2/GHS_psdef we have 11 shapes that will generate 11 macro for loops `LOOP_SHAPE_0` to `LOOP_SHAPE_10`), each representing a loop nest fully unrolled and annotated with SIMD-friendly directives (e.g., `#pragma unroll`). The Listing 4.5 shows sample macro thats getting generated for a particular polyhedral shape. Leveraging the structured infor-

mation provided by the UZP format, we integrate C macro auto-generation techniques to enable compile-time optimizations. The iteration bounds and index stride computations are statically encoded within each macro, allowing the compiler to recognize and emit predictable affine loop structures that are highly suitable for SIMD vectorization.

This modification simplifies the loop control flow and increases the effectiveness of compiler autovectorization by eliminating runtime conditionals, loop bounds checks, and indirect indexing across batches. Additionally, per-origin dispatch enhances modularity and extensibility, allowing distinct prefetching, unrolling, or alignment strategies to be encoded on a per-shape basis. However, this finer-grained approach comes at the cost of increased function call overhead due to lack of loop fusion and fewer opportunities to amortize instruction fetch latency across similar origins. Furthermore, the lack of spatial grouping may degrade cache locality in cases where multiple origins from the same shape are adjacent in memory but are now processed independently.

Despite these trade-offs, this per-origin strategy represents an important shift toward **just-in-time shape specialization at compile time**. By tightly coupling shape metadata with its corresponding execution pattern, we enable future integration with adaptive runtime dispatch tables or even instruction-level tuning based on AVX-512 vector widths. This design also makes it feasible to explore compiler feedback loops and MLIR-based dynamic code regeneration in the future. In summary, the updated implementation sacrifices some locality aggregation in exchange for greater compiler friendliness, stronger inlining potential, and a scalable path toward dynamic shape-aware optimization in sparse matrix kernels.

4.4.4 Impact of Favour Locality Tuner with Macro-Specialized Executors

While the Favour Locality Tuner already demonstrated significant performance gains over the Standard UZP configuration using fully generic executors, its impact can be further amplified by introducing macro-specialized execution paths for common repeating patterns. Figure 4.12 illustrates the comparative performance results under this extended configuration. Note that the full experimental setup is described in Chapter 5.

```
#define LOOP_SHAPE_0 \
    _Pragma("unroll") \
    for (int i = 0; i <= 119; i++) \
    {  \
        /* Compute. */  \
        y[idx_y] += data_vector[a_data_pos++] * x[idx_x]; \
        idx_y += 1;  \
        idx_x += 1;  \
    }
```

Listing 4.5: Macro definition for `LOOP_SHAPE_0`.

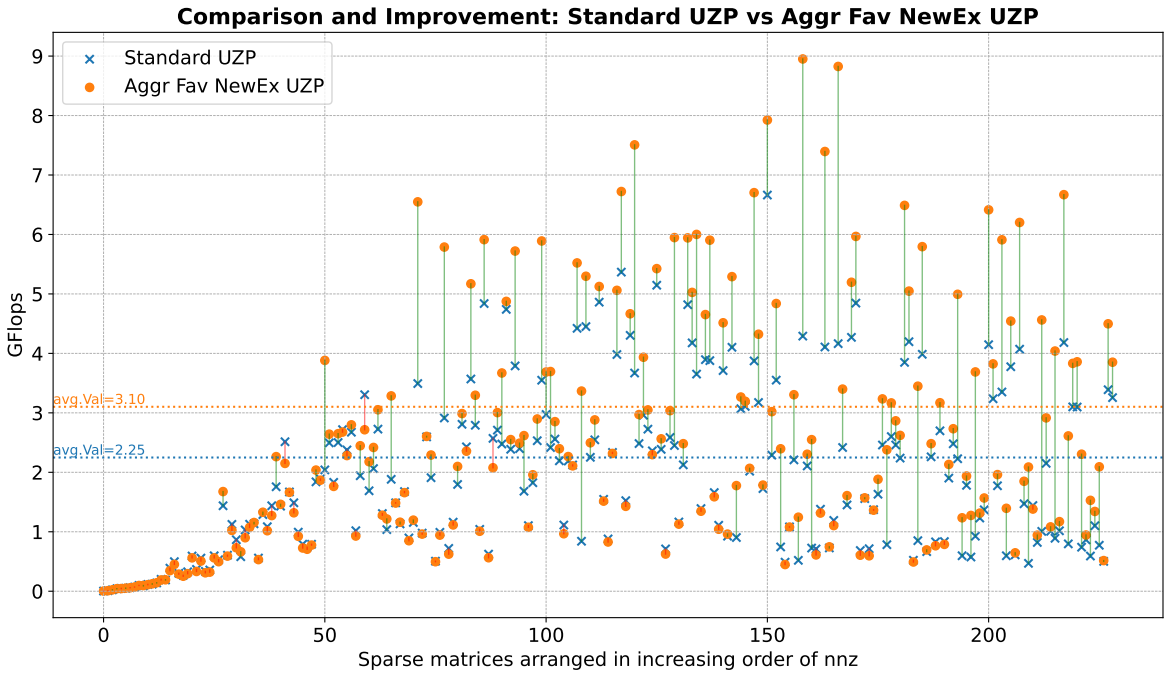

Figure 4.12: Performance comparison of SpMV operations on Standard UZP versus Favour Locality Tuned UZP with newgen/executor across 229 sparse matrices sorted by increasing non-zero count).

Experimental results show that under this configuration, the Favour Locality Tuner achieves a further uplift in GFLOPS compared to the fully generic execution case. On average, through-

put increases from 2.87 GFLOPS Section 4.2 (generic executors) to 3.10 GFLOPS with macro specialization, reflecting an additional improvement of approximately 8%. The benefits are more pronounced for matrices with larger numbers of nonzeros, where repeated structural patterns are more common and can be effectively captured by specialized macros.

The observed improvements are attributed to two main factors: (1) the Favour Locality Tuner continues to enhance spatial locality through controlled aggregation and X-axis reordering, and (2) the introduction of macro-specialized loops reduces runtime branching, improves loop unrolling, and enables more effective SIMD vectorization for the frequent cases. Together, these effects lower instruction overhead and improve cache and memory subsystem utilization.

Overall, combining the Favour Locality Tuner with macro-specialized execution paths provides a scalable and effective optimization strategy within the UZP framework. It leverages sparsity structure regularization and common-pattern specialization to achieve higher performance without requiring extensive auto-generation infrastructure or full manual tuning.

Chapter 5

Experiments and Results

This chapter presents the experimental evaluation of the UZP framework and the developed tuners. We analyze single-threaded performance improvements for both double-precision and single-precision kernels, and evaluate parallel scaling behavior. We compare UZP performance against various state-of-the-art representations, including CSR, CSR5, BCSR, and Intel MKL, for SpMV computations. Additionally, we compare baseline and tuned UZP configurations across key metrics such as computational throughput, storage compression, and memory efficiency. The results highlight the effectiveness of structured sparse representations and tuning strategies in optimizing sparse computations.

We evaluate our approach on a set of 229 sparse matrices, following the selection methodology of Horro et al. [18]. The matrices are obtained by filtering the SuiteSparse Matrix Collection Davis and Hu [16] to retain instances with fewer than 20 million nonzero entries, using a sieving strategy similar to that employed by Augustine et al. [2]. It is worth mentioning that the complexity of the UZP generation process is approximately linear in the number of nonzeros. The corresponding generation times are reported in Figure 5.2.

Matrices are classified according to the decile they belong to in terms of the number of nonzeros and the percentage of them captured in polyhedral patterns. This process yields 100 categories, and inside each category, k-means clustering is used to select representative matrices. The number of representatives per cluster is chosen so that the probability density of the sample matches the original distribution. Figure 5.1 displays this selection process.

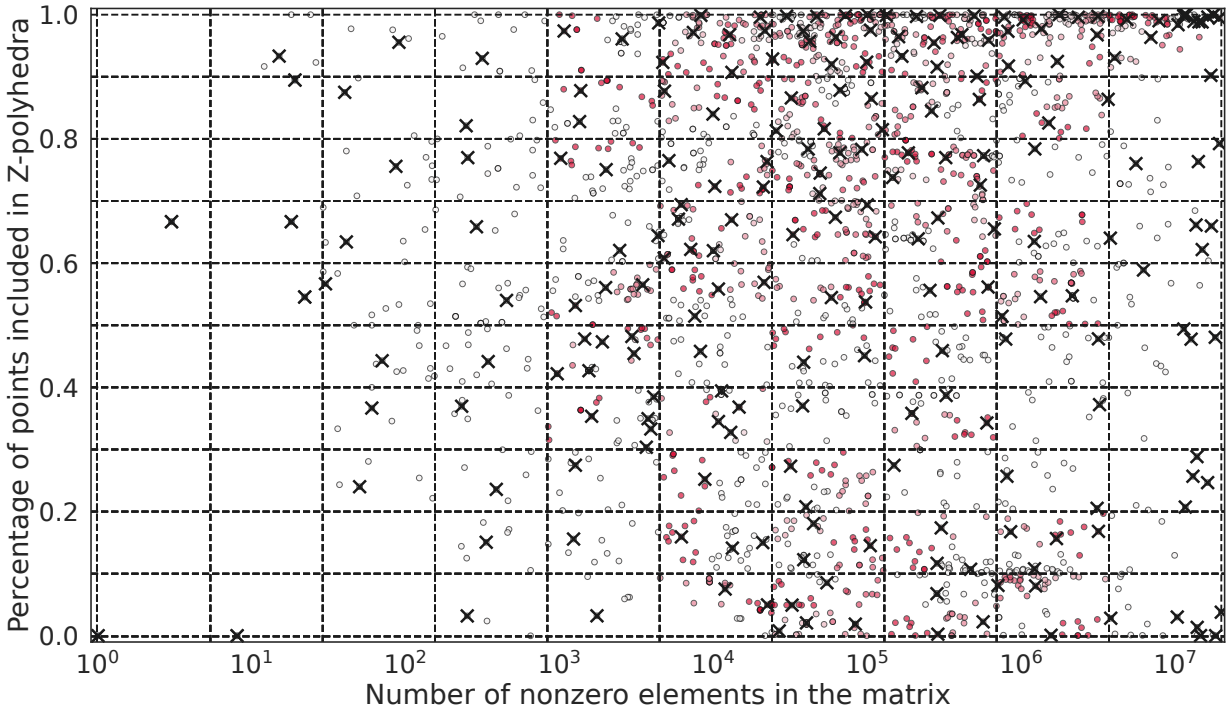

Figure 5.1: Sieve of the $2,754$ SuiteSparse matrices below 20M nonzeros. Selected matrices are marked with 'X's.

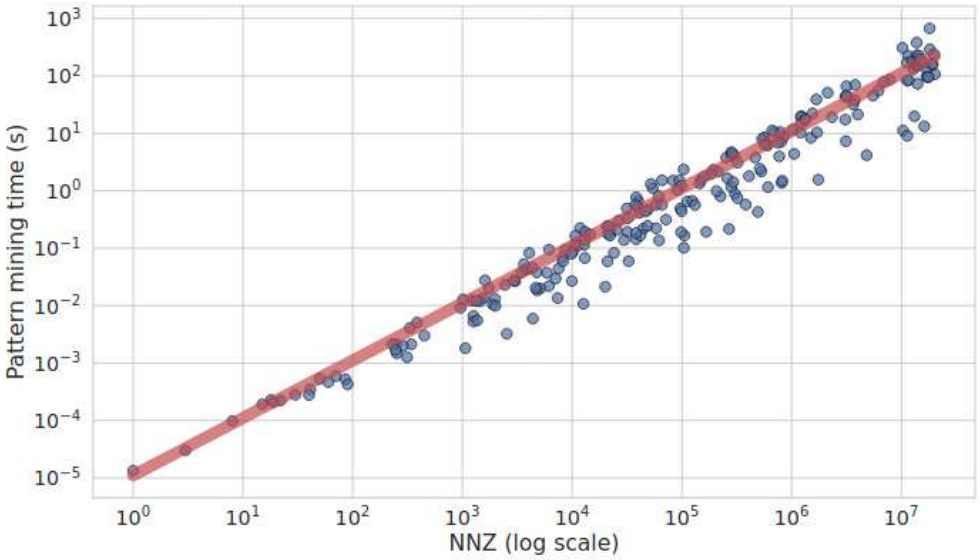

Figure 5.2: UZP generation time.

In this section, we analyze the experimental performance of SpMV executed on double-precision floating-point data. Experiments were conducted on an Intel Core i9-12900K processor with

128 GiB of RAM. Each run was repeated 10 times; we report the best performance achieved for each experiment after removing outliers, identified as measurements deviating more than 3σ from the mean. The CPU frequency was fixed at the nominal base of 3.2 GHz to minimize experimental variability due to thermal throttling. Both data and code segments were allocated using 2 MiB hugepages for all experimental versions. All codes were compiled with GCC 11.4.0, using the flags `-O2 -ffast-math -ftree-vectorize -floop-unroll-and-jam -march=native`. The PolyBench benchmarking harness [24] was used for performance measurements. Prefetching of the text segment [2] was included during linking for data-specific code versions. Under the same setup, using the 16 logical P-cores of the processor, Intel Linpack reports an average raw performance of 364.2 GFLOPS, with a peak of 499.6 GFLOPS. The memory-bound Intel HPCG benchmark reports a peak single-threaded SpMV performance of 5.5 GFLOPS.

We evaluate six different versions of the SpMV computation: (1) a baseline CSR implementation shown in Listing 2.2; (2) Intel MKL 2024.1.0 using the `mkl_sparse_s_mv()` routine; (3) the same MKL version using BCSR encoding through `mkl_sparse_convert_bsr()`; (4) our generic executor described in Section 4.4, labeled UZP GenEx; (5) a CSR5-based SpMV kernel [22]; and (6) the partially-strided codelets approach [11, 10], using publicly available GitHub sources.

5.1 Single-Threaded Performance

The performance of these kernels under cold-cache, single-threaded conditions is shown in Figures 5.3 and 5.4. In these experiments, each SpMV kernel is executed once after data loading and cache flushing, measuring raw performance. UZP GenEx achieves the best overall results, averaging 2.29 GFLOPS across the full experimental set (CSR: 1.63, CSR5: 1.39, MKL: 1.83). While CSR and CSR5 consistently underperform relative to UZP, MKL begins to close the gap for matrices with more than 500K nonzeros. A closer analysis shows that GenEx's advantage over MKL correlates with the percentage of unincorporated points and the number of shapes required to capture the majority of incorporated points. Considering matrices with more than 10K nonzeros,

MKL outperforms GenEx with 72% probability when more than 20% of points are unincorporated. Conversely, when fewer than 10% of points are unincorporated, GenEx is more likely to outperform MKL.

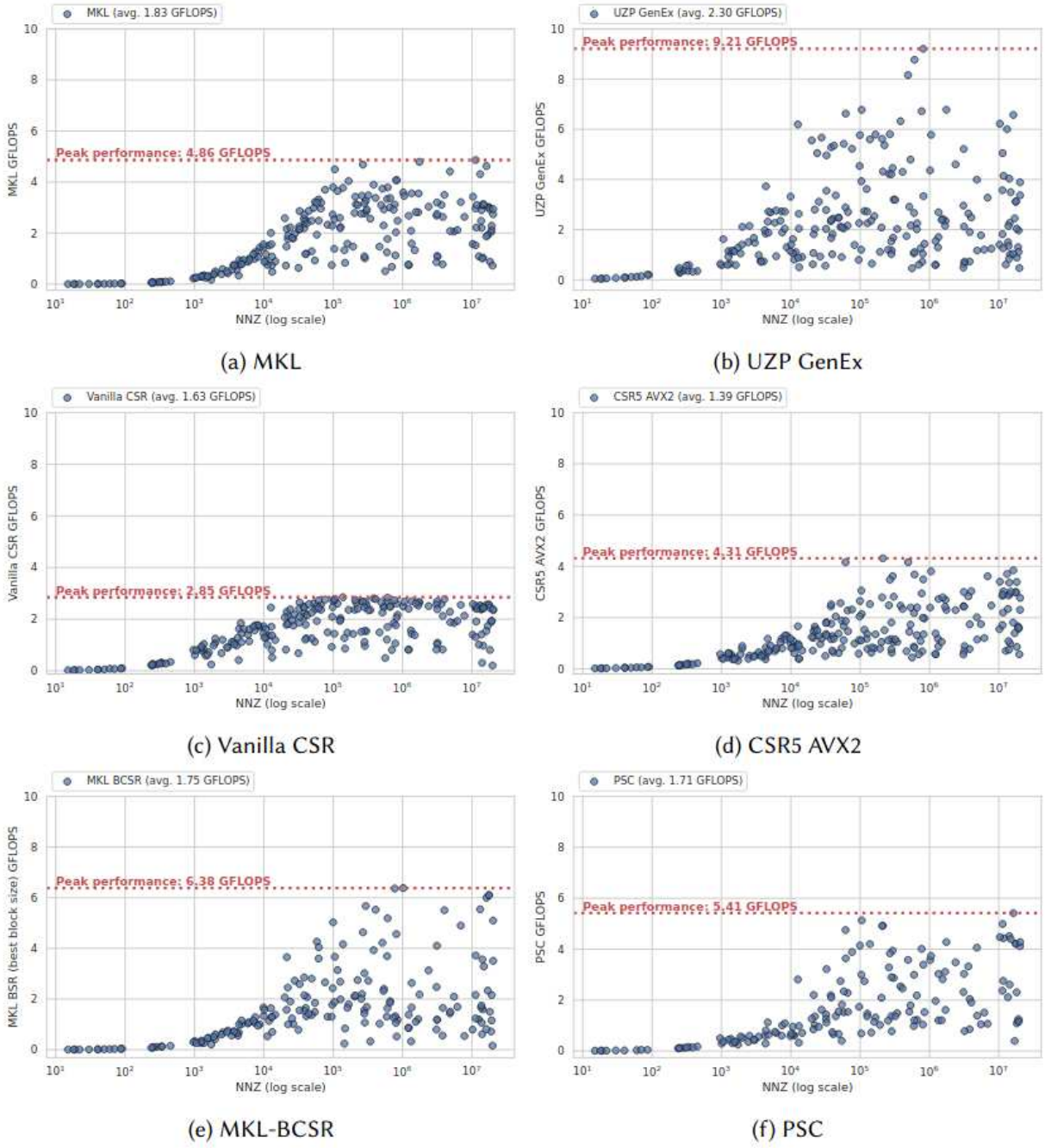

Figure 5.3: Performance (GFLOPS) of double-precision, cold cache kernels. MKL-BCSR in Fig. 6e shows the best performance for all block sizes between 2 and 32.

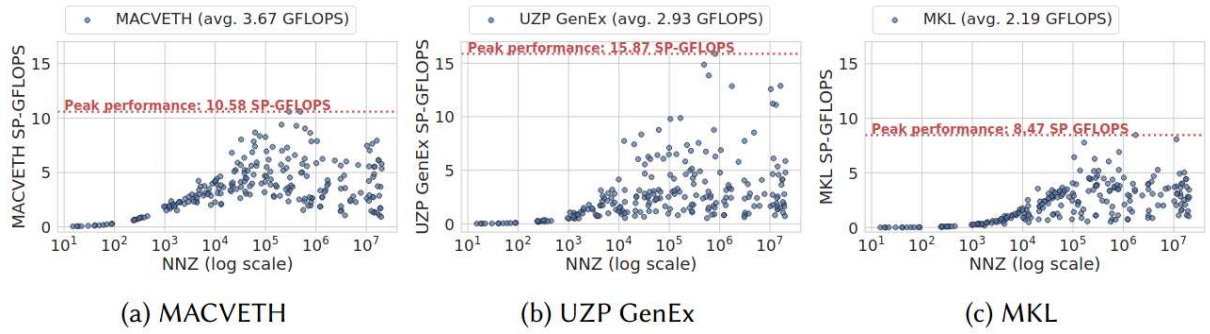

Figure 5.4: Performance (GFLOPS) of single-precision, cold cache kernels.

Results for MKL-BCSR and PSC are reported separately, as these kernels do not produce outputs across the full experimental set. Averages are computed over the corresponding valid subset for each kernel. The MKL-BCSR implementation requires that the block size divides both the number of rows and columns; when these dimensions are coprime, no feasible block size exists. For each matrix, all feasible block sizes between 2 and 32 were evaluated, and the configuration achieving the best raw performance is reported in Figure 5.3. Across the matrices for which results are available (187 matrices), MKL-BCSR achieves an average performance of 1.75 GFLOPS, compared to 1.89 GFLOPS for MKL and 2.37 GFLOPS for UZP over the same subset. MKL-BCSR outperforms both UZP and MKL on 30 matrices, where the spurious computation overhead (i.e., computations on explicit zeros introduced by BCSR padding) remains below a $2\times$ factor (average overhead: $1.91\times$). On this subset, MKL-BCSR achieves an average performance of 2.91 GFLOPS, compared to 2.02 GFLOPS for MKL and 1.98 GFLOPS for UZP. For the remaining 157 matrices, MKL-BCSR introduces a higher spurious overhead, averaging $5.27\times$, and achieves an average performance of 1.52 GFLOPS (MKL: 1.86, UZP: 2.44).

For PSC, we modified the publicly available source code to normalize the experimental setup by introducing the PolyBench/C harness. We reproduced a cold-cache environment in our tests: the matrix is read, the cache is flushed, and a single repetition of the SpMV kernel is executed. Symmetric matrices were expanded to ensure fair comparison with all other kernel versions in our

setup. This approach produced results for a total of 166 matrices,[1] yielding an average performance of 1.71 GFLOPS (MKL: 1.97, UZP: 2.49). Performance is primarily driven by how well the sparsity structure of a matrix conforms to the predefined codelet shapes. PSC outperforms both MKL and UZP GenEx on 16 matrices, for which it emits only 2.15 instructions per FLOP (MKL: 2.43, UZP: 2.52) and achieves an average performance of 3.51 GFLOPS (MKL: 2.64, UZP: 2.75). For the remaining 150 matrices, PSC emits 22.83 instructions per FLOP (MKL: 14.20, UZP: 5.03) and achieves an average performance of 1.52 GFLOPS (MKL: 1.90, UZP: 2.46).

We draw general performance considerations by analyzing selected hardware counters. The superior performance of UZP GenEx relative to MKL is attributed to both improved memory behavior and more efficient vectorization. On the memory side, UZP incurs 0.29/0.016 L2/L3 cache misses per FLOP (MKL: 0.45/0.031, MKL-BCSR: 0.65/0.072). Regarding vectorization, MKL performs on-the-fly zero-filling when traversing the sparse structure, whereas GenEx relies on statically provided UZP metadata. Consequently, MKL executes 19% more FLOPs than strictly required across the experimental set, emitting 1.38 billion FLOPs versus 1.13 billion useful FLOPs. In contrast, GenEx emits 1.16 billion FLOPs, much closer to the theoretical minimum.

CSR, CSR5, and PSC also exhibit degraded memory performance compared to GenEx, with L2/L3 cache misses per FLOP of 0.33/0.021 (CSR), 0.30/0.017 (CSR5), and 0.52/0.023 (PSC), respectively. Additionally, these kernels execute approximately 1.5$\times$ more instructions than UZP GenEx and MKL. In CSR, all FLOPs are scalar, while CSR5 relies almost exclusively on 256-bit AVX2 vector instructions, resulting in a total of 1.44 billion FLOPs, corresponding to a 27% excess over the theoretical operation count.

Single-Precision Kernels.

We repeated these experiments for single-precision floating point inputs, except for CSR5 and PSC, whose implementations are not compatible with floats. We observe the same trends as for double-precision computations: UZP GenEx achieves the best overall performance, with an aver-

[1]We suspect possible incompatibilities with certain MTX file formats from the SuiteSparse Matrix Collection.

age of 2.93 GFLOPS (CSR: 1.73, MKL: 2.19, MKL-BCSR: 2.05). Using single-precision inputs the L2/L3 misses per FLOP experiment a reduction across all kernels (UZP: 0.25/0.008, CSR: 0.28/0.014, MKL: 0.42/0.020, MKL-BCSR: 0.46/0.033). The number of FLOPs issued by both approaches also grows, to 1.39 and 1.54 billion, respectively, for GenEx and MKL. This signals a less efficient utilization of the available vector lanes when their numbers increase. Other execution parameters remain qualitatively similar to the double-precision case. Selected performance plots are shown in Fig. 5.4.

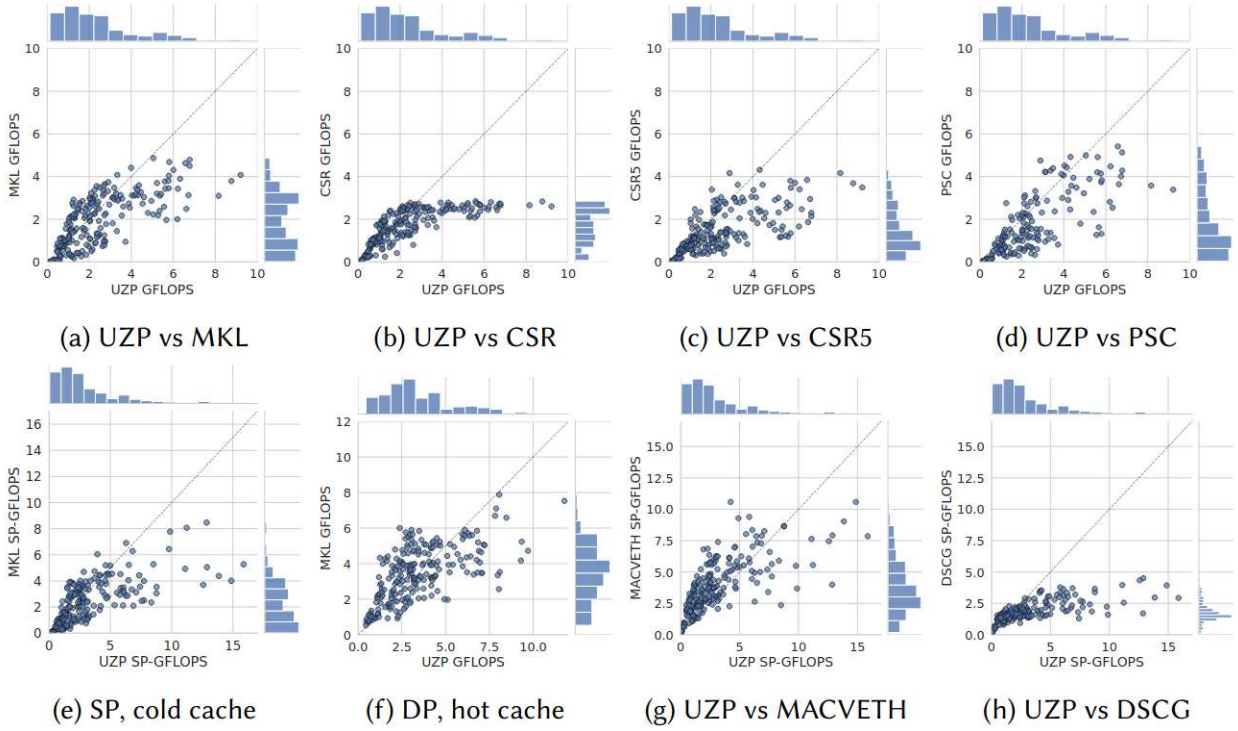

Figure 5.5: Joint performance (GFLOPS) plots of different kernel versions.

Repeating the same reasoning on attainable peak performance as above, we now obtain a peak range between 6.3 and 18.8 GFLOPS, derived from the change in arithmetic intensity due to the new datatype sizes. In this case, the maximum observed performances increase to 8.47 and 15.87 GFLOPS for MKL and GenEx, respectively. The joint performance plot for this case is shown in Fig. 5.5.

Hot Cache Setting.

We executed the SpMV kernels in a hot cache, double-precision setting, repeating the sparse multiplication 100 times for each matrix while varying the input vectors and without flushing the cache in between each repetition. This configuration is similar to, e.g., neural network inference. UZP GenEx remains the best performing approach on average, achieving 3.37 GFLOPS (CSR: 2.13, CSR5: 2.02, MKL: 3.32, PSC: 2.58[2]). We observe the same trends as for cold cache experiments in terms of L2/L3 misses per FLOP, which appears to indicate that all approaches benefit from the cache in a similar way. In this case, the maximum performance obtained by UZP GenEx increases to 12.79 GFLOPS (CSR: 3.09, CSR5: 4.88, MKL: 7.88 GFLOPS). The joint performance plot for this case is shown in Fig. 5.5.

Data-Specific Codes.

Generating highly-specialized data-specific and hardware-specific kernels has the potential to provide significant performance advantages, by fine-tuning the code to the specific characterics of both data and hardware. Fig. 5.5 shows the joint plot of the performance of GenEx vs MACVETH. This experiment uses a single-precision, cold cache setting, as the MACVETH compiler by Horro et al. [18] is not compatible with double-precision data. For completeness, experimentation was also conducted for the original data-specific codes as generated by Augustine et al. [2], i.e., data-specific codes compiled using GCC directly, without MACVETH optimization. This version, referred to below as DSCG, never outperforms MACVETH in our experimental set.

The MACVETH-optimized data-specific codes have a clear performance advantage over other kernels, achieving an average of 3.67 GFLOPS (GenEx: 2.93, MKL: 2.19, DSCG: 1.72). The fundamental driver of performance in this case is a 5.8x reduction in the number of instructions executed (3.7x fewer microoperations). This is due to the fully linear nature of MACVETH codes, which have been fully unrolled (no loops exist in the code) and then aggressively vectorized using AVX2 primitives specifically designed for the target machine. However, the clear disadvantage is

[2]For the subset of matrices for which PSC produces results, the GFLOP count is: UZP: 3.59, MKL: 3.54, CSR: 2.26, CSR5: 2.16.

the explosion of code sizes: while the generic executor binary is only 80 KiB, MACVETH binaries for the largest matrices in the experimental set have sizes in the order of hundreds of MiB. This imposes high pressures both on the memory subsystem, featuring a 1.5x increase in the number of L3 misses, and on the processor front-end, which routinely becomes the performance bottleneck. As a result, even if this approach presents the best average performance, it achieves a more limited peak performance of 10.58 GFLOPS (UZP GenEx: 15.87). Joint plots of both MACVETH and DSCG vs UZP are shown in Fig. 5.5.

5.2 Parallel Scaling

The *UZP* framework introduces a structured approach to sparse matrix-vector multiplication (SpMV) by leveraging polyhedral shape representations of nonzero entries. Unlike traditional formats that rely on flat coordinate-based encodings, UZP enables a shape-centric view that can be exploited for improved computational efficiency and parallelism. In the context of parallel execution, the UZP generic executor partitions the matrix into independent polyhedral regions, each corresponding to a distinct instance of a reusable pattern. These regions are then distributed across threads using OpenMP-based static scheduling. Specifically, the executor employs `#pragma omp parallel for` directives to dispatch processing tasks across multiple cores, allowing for concurrent evaluation of the matrix-vector product over disjoint shape-origin mappings. This design facilitates scalable parallel execution by aligning computational granularity with structural regularity, reducing thread synchronization overhead and improving cache locality.

To evaluate the effectiveness of parallelization within the UZP-based SpMV executor, we conducted a set of cold-cache, double-precision experiments across four competing backends: UZP GenEx, Intel MKL, CSR5, and PSC. All implementations were executed using multithreaded configurations, with the MKL backend utilizing the threaded variant of the Intel MKL library. For UZP GenEx, parallelism was achieved by statically distributing loop iterations via OpenMP directives, enabling concurrent processing of disjoint polyhedral regions in the matrix.

Our results, summarized in Figure 5.6, focus on matrices with more than 100,000 nonzeros to ensure meaningful performance trends. The UZP executor demonstrated strong scaling behavior, achieving average parallel performances of 3.94 GFLOPS and 6.46 GFLOPS under varying thread configurations. This surpasses the performance of MKL (3.73 and 5.53 GFLOPS), CSR5 (2.94 and 6.06 GFLOPS), and PSC (3.97 and 5.18 GFLOPS) on average. Furthermore, UZP GenEx attained peak throughput of 11.61 GFLOPS and 29.19 GFLOPS-outperforming all baselines, including MKL (7.33 and 12.13 GFLOPS) and CSR5 (6.48 and 17.72 GFLOPS). These results indicate that the structured parallelization strategy employed by UZP not only scales effectively with increasing thread counts but also capitalizes on the inherent regularity of polyhedral regions to deliver superior performance across a range of sparse matrix workloads.

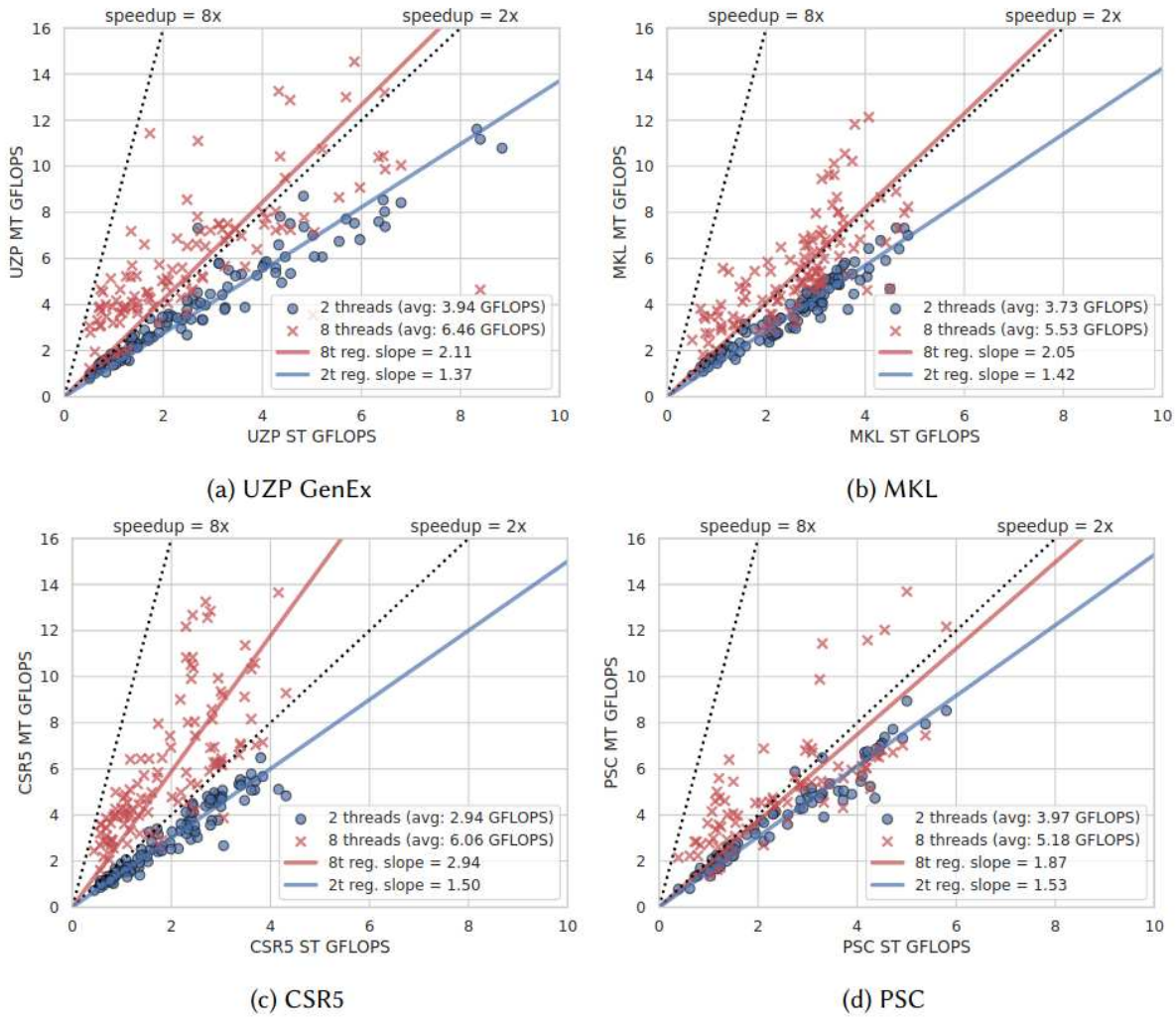

Figure 5.6: Scaling plots for cold cache executions with two and eight threads for matrices above 100K nonzeros. The plots show single-threaded (X axes) vs multi-threaded (Y axes) performance in GFLOPS. The solid lines show the linear regression models of the scaling. The dotted lines show the reference 2x/8x scalings.

5.3 Evaluation of UZP & TUNNERS Optimization Strategies

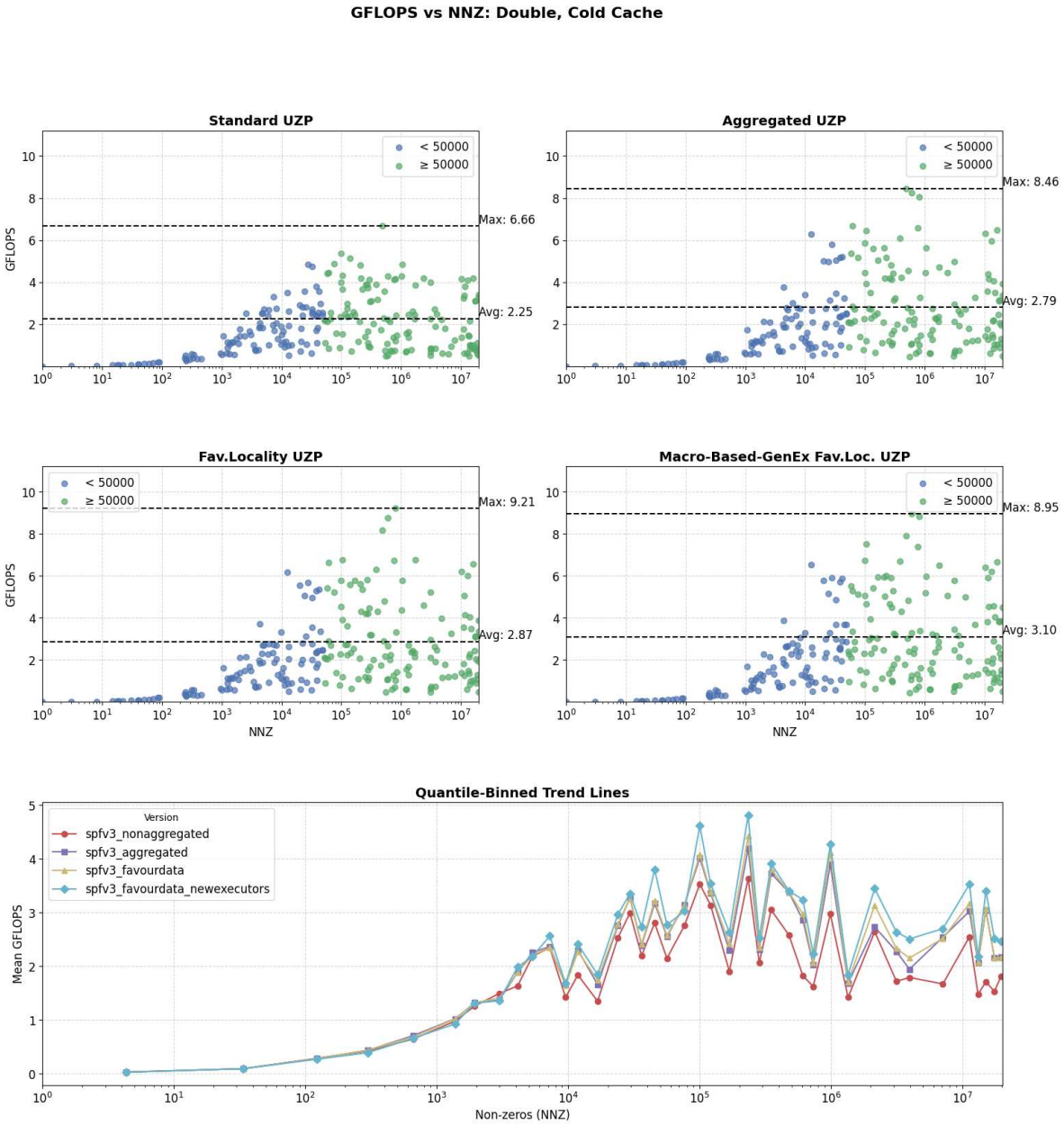

Figure 5.7: Comparison of UZP variants under double-precision, cold-cache settings. Aggregation and favourdata tuners progressively improve performance over the non-aggregated baseline. Shuffling origin order reduces locality, leading to lower average throughput despite the same shape set.

This section presents a comprehensive evaluation of the UZP representation and the optimization strategies enabled by the developed tuners. All experiments are conducted using a benchmark

suite of 229 sparse matrices drawn from the SuiteSparse collection [16], covering a wide range of matrix sizes, sparsity patterns, and structural complexities.

We compare the performance impact of each tuner- Aggregator Tuner, Favor Locality Tuner, and Auto-generated macro-specialized loops based Executor optimizations—focusing on key metrics such as computational throughput (GFLOPS), memory access behavior, and scalability across different sparsity profiles. The evaluation highlights how selective aggregation, locality-aware reordering, and executor specialization contribute to overall improvements in sparse matrix computation efficiency, particularly for operations such as SpMV and SpMM.

In this section, we also further analyze the impact of the tuners developed, focusing on both their improvements and potential degradations. We begin with the Aggregator Tuner, emphasizing a deeper analysis based on hardware counter data collected using PAPI (Performance Application Programming Interface). PAPI enables precise measurement of machine cycles spent on fundamental operations during execution, providing detailed insights into low-level performance characteristics.

The Figure 5.7 consolidates the GFLOPS versus NNZ performance results across all major UZP tuning configurations: Standard UZP (Nonaggregated), Aggregated UZP, Favour Locality UZP, and Favour Locality combined with New Executors. Each subplot reports both the maximum and average GFLOPS achieved across the 229 matrices, with points color-coded based on matrix density.

The overall trend clearly illustrates the progressive impact of each tuning stage. Aggregated UZP achieves a substantial improvement over the Standard UZP baseline, raising the average GFLOPS from 2.25 to 2.79, primarily through the reduction of origin metadata and improved computational efficiency. The Favour Locality Tuner builds upon this by enhancing memory access regularity, achieving an average of 2.87 GFLOPS. Further, the introduction of New Executors—optimizing loop structures to better align with modern SIMD architectures—pushes the average performance to 3.10 GFLOPS.

These results demonstrate that while Aggregation alone provides significant benefits, additional layers of tuning focused on memory access locality and executor specialization can deliver further improvements. Importantly, the modularity of the UZP framework enables each optimization to be applied independently or in combination, without altering the fundamental matrix structure.

Now let's dive deeper into understanding the cause and reasons behind the improvements in performance due to the Aggregator Tuner. Before that its important to note that presenting PAPI variations across all 229 matrices would be impractical and not particularly informative. Given the diversity of the dataset, aggregate statistics can obscure meaningful trends and make it difficult to draw actionable conclusions. Moreover, failures or limited improvements for certain matrices often arise due to intrinsic properties such as sparsity patterns, highlighting the importance of understanding the structural characteristics of each matrix class.

To address this, we focus our detailed analysis on the top 20 matrices—approximately the top 10% of the dataset—ranked by the performance improvements achieved with the Tuner. This selection enables a more targeted evaluation of the tuner's effectiveness under favorable conditions, while still maintaining generality. Although the tuners are applied independently across all matrices, focusing on best-performing cases provides clearer insights into the types of sparsity structures where the tuner yields substantial benefits. An overly generalized view across the full dataset would risk diluting these insights, as variations would be averaged out and potentially mask important optimization effects.

To evaluate the impact of the Aggregator Tuner within the UZP framework as part of the UZP & TUNERS Optimization Strategies work, we conduct an experimental study using 229 sparse matrices from the SuiteSparse collection. This evaluation analyzes multiple performance dimensions, including execution time, computational throughput, cache behavior, SIMD utilization, and memory footprint, providing a comprehensive view of system performance. Results are compared against the baseline standard UZP representation to quantify both the structural and computational improvements achieved by the Aggregator Tuner.

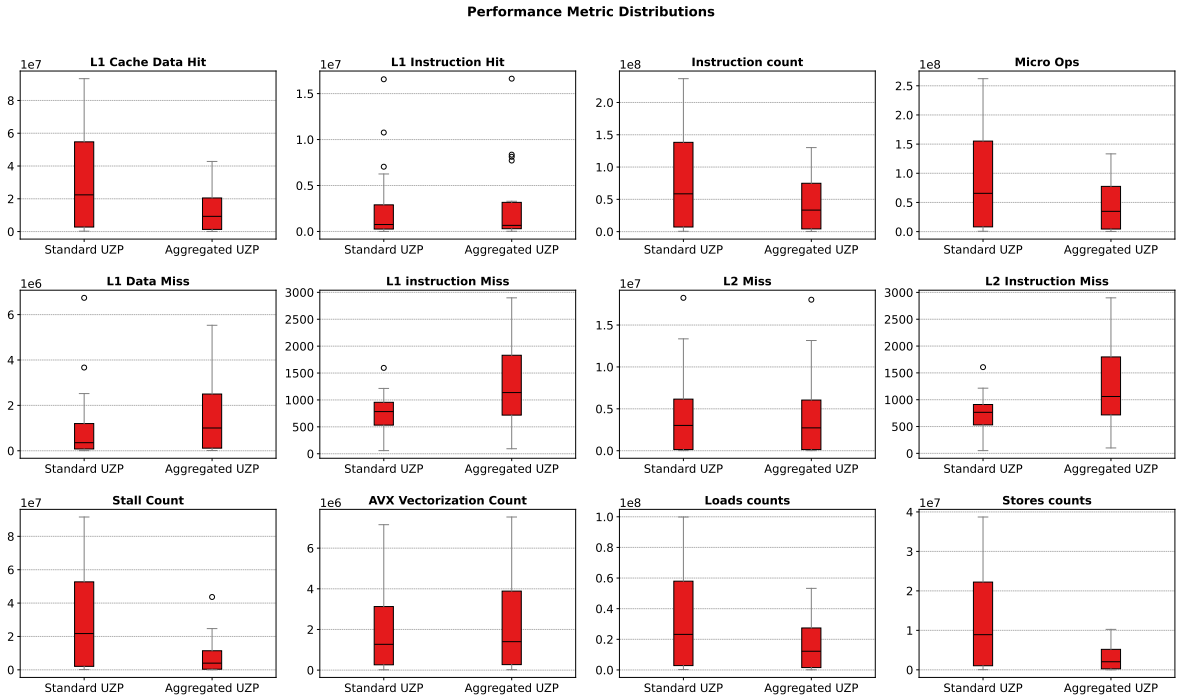

Figure 5.8: Box plot comparing the normalized PAPI counter performance metrics for Standard Uzp versus Aggregated Uzp.

Figures 4.2, 4.3, and 4.4 illustrate the key structural and performance impacts of the Aggregator Tuner. To further investigate the underlying causes of the observed performance improvements, Figure 5.8 presents a detailed analysis of hardware counter variations (PAPI metrics) across the experimental dataset of top 20 best performing samples listed in the Table 5.1 for both the Standard UZP and Aggregated UZP configurations. The results show that the performance gains achieved by the Aggregator Tuner are not primarily due to cache behavior improvements, but rather then enhanced control flow regularity and reduced computational overhead. With the aggregation of contiguous origin points, we tend to reduce the instruction count, which in turn reduces significant load and store operations.

In Sparse Matrix-Vector Multiplication (SpMV) experiments, the Aggregated UZP format demonstrates consistently better performance compared to the Standard UZP format, despite exhibiting slightly higher cache miss rates. A detailed analysis reveals that the Aggregated format reduces the total number of micro-operations issued, decreases memory load and store operations,

and lowers stall cycles within the processor pipeline. These reductions indicate that Aggregated UZP not only improves memory access regularity but also minimizes computational work per nonzero matrix element.

Additionally, modest improvements in AVX vectorization rates were observed, suggesting enhanced SIMD utilization. Collectively, these factors show that the performance improvement results from a combination of reduced instruction overhead and more efficient memory bandwidth utilization. Although cache miss rates increased slightly, the Aggregated format enables more contiguous memory accesses, allowing hardware prefetchers and DRAM controllers to sustain higher effective throughput. Thus, the performance gains can be attributed to both increased computational efficiency (fewer μops and improved vectorization) and better memory streaming behavior, with memory bandwidth optimization playing the dominant role.

Table 5.1: Comparison of Standard UZP and Aggregated UZP GFlops of SpMV operation on top 20 performing matrices

File Names	nrows	ncols	nnz	Inc. (%)	Std GFLOPS	Agg GFLOPS	Diff (%)
JP	87616	67320	13734559	96.63	0.47	2.15	357.90
crankseg_2	63838	63838	14148858	99.70	1.01	4.05	302.64
pkustk14	151926	151926	14836504	99.47	0.89	3.49	292.27
nd6k	18000	18000	6897316	97.27	0.93	3.27	253.29
cfd2	123440	123440	3087898	98.12	0.85	2.94	245.86
air02	50	6774	61555	94.68	0.84	2.83	236.47
Zd_Jac6_db	22835	22835	663643	89.79	0.73	2.25	210.21
in-2004	1382908	1382908	16917053	94.63	0.80	2.46	208.90
Maragal_6	21255	10152	537694	85.65	0.75	2.19	193.91
web-NotreDame	325729	325729	1497134	91.75	0.78	2.29	192.04
human_gene2	14340	14340	18068388	69.53	0.74	2.05	176.55
eu-2005	862664	862664	19235140	91.30	0.77	1.96	153.51
Freescale1	3428755	3428755	18920347	83.74	0.59	1.30	119.14
tp-6	142752	1014301	11537419	66.73	0.60	1.26	111.43
rajat30	643994	643994	6175377	86.78	0.58	1.21	110.18
rajat20	86916	86916	605045	79.69	0.52	1.09	108.83
Zhao2	33861	33861	166453	100.00	2.45	5.02	104.87
TSOPF_RS_b162_c4	20374	20374	812749	99.89	4.16	8.04	93.15
rajat29	643994	643994	4866270	85.49	0.60	1.15	92.85
TSOPF_FS_b162_c1	10798	10798	608540	99.55	4.29	8.23	91.82

Beyond performance improvements, the Aggregator Tuner also has a significant impact on storage efficiency particularly in compressing the metadata required to represent sparse matrices.

As discussed earlier, the UZP format is composed of reusable polyhedral shapes and an associated list of origin points. By reducing the number of origins through merging, the tuner directly reduces the volume of metadata stored per matrix, which can be substantial for large, sparse datasets.

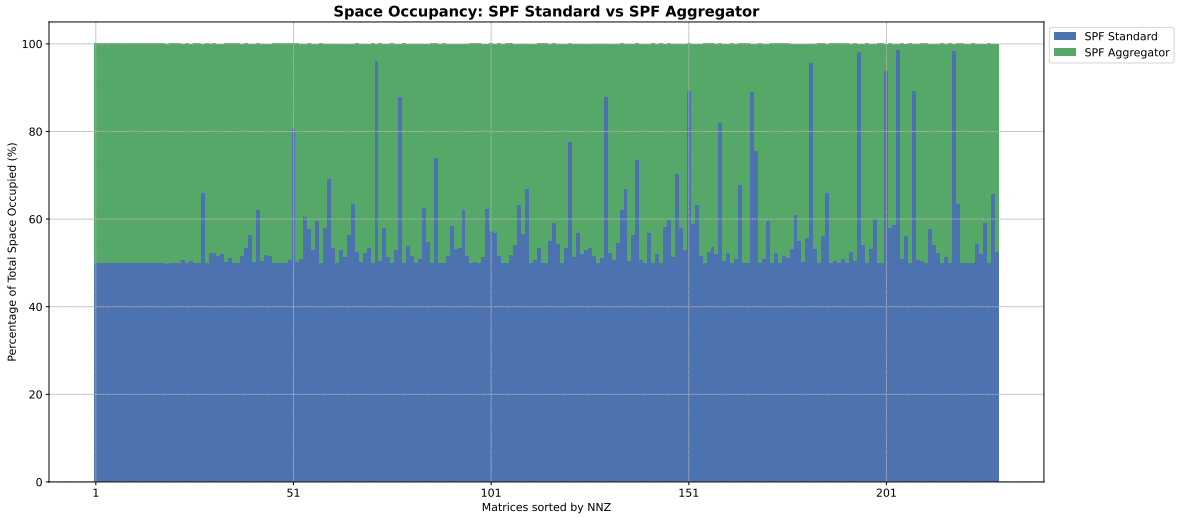

Figure 5.9: Stacked bar chart illustrating the relative space occupied by Standard UZP versus Aggregator UZP metadata size for multiple sparse matrices sorted in increasing order of non-zero elements.

To quantify this effect, we compared the metadata size of the standard UZP representation with that of the Aggregated UZP across all 229 matrices in our evaluation set. We observed that approximately half of the matrices benefited from aggregation in terms of storage space. In many cases, especially for those matrices that exhibited strong regular patterns, the memory footprint got reduced because of the significant reduction in the number of origin entries and compressing the output size. The Figure 5.9 illustrates the comparison of storage space utilization between the Standard UZP and Aggregated UZP formats across 229 sparse matrices. The y-axis represents the percentage of total storage space, with the sum of the blue and green regions normalized to 100% for each matrix.

The primary cause of this behavior is the reduction in the number of origin points, as previously observed in Figure 4.4 and discussed in Section 4.1. By consolidating multiple origins into larger

polyhedral regions, the Aggregator UZP strategy reduces metadata overhead and improves storage efficiency, while fully preserving the completeness of the sparse matrix representation.

Table 5.2: Top 10 Matrices with Significant GFlops Improvement across Aggregated UZP vs Favour Locality UZP (nnz > 50K)

File Name	nnz	GFlops (Aggregated)	GFlops (Favour Locality)	GFlops Diff (%)
apache2	4,817,870	2.7142	3.9916	47.06
crashbasis	1,750,416	4.7169	6.7799	43.73
rat	268,908	4.8079	5.8063	20.77
Zhao2	166,453	5.0209	5.7903	15.32
kim2	11,330,020	4.3824	5.0508	15.25
TSOPF_RS_b162_c4	812,749	8.0429	9.2072	14.48
sparsine	1,548,988	2.2630	2.4284	7.31
TSOPF_FS_b162_c1	608,540	8.2301	8.7686	6.54
fome13	285,056	1.0986	1.1687	6.38
gridgena	512,084	2.7655	2.9109	5.26

Table 5.2 lists the top 10 matrices where the Favour Locality Tuner achieves the highest GFLOPS improvement over the Aggregated UZP configuration, with gains reaching up to 47%. These results demonstrate that locality-aware reordering and selective aggregation strategies are critical for optimizing sparse matrix performance, particularly for matrices with large and irregular sparsity patterns.

The Favor Locality Tuner consistently improves memory access regularity by reorganizing origin points based on spatial proximity, which enhances cache utilization and reduces memory latency. While Aggregated UZP reduces metadata and computational overhead by merging origins, Favor Locality further refines performance by aligning data layouts with the memory hierarchy.

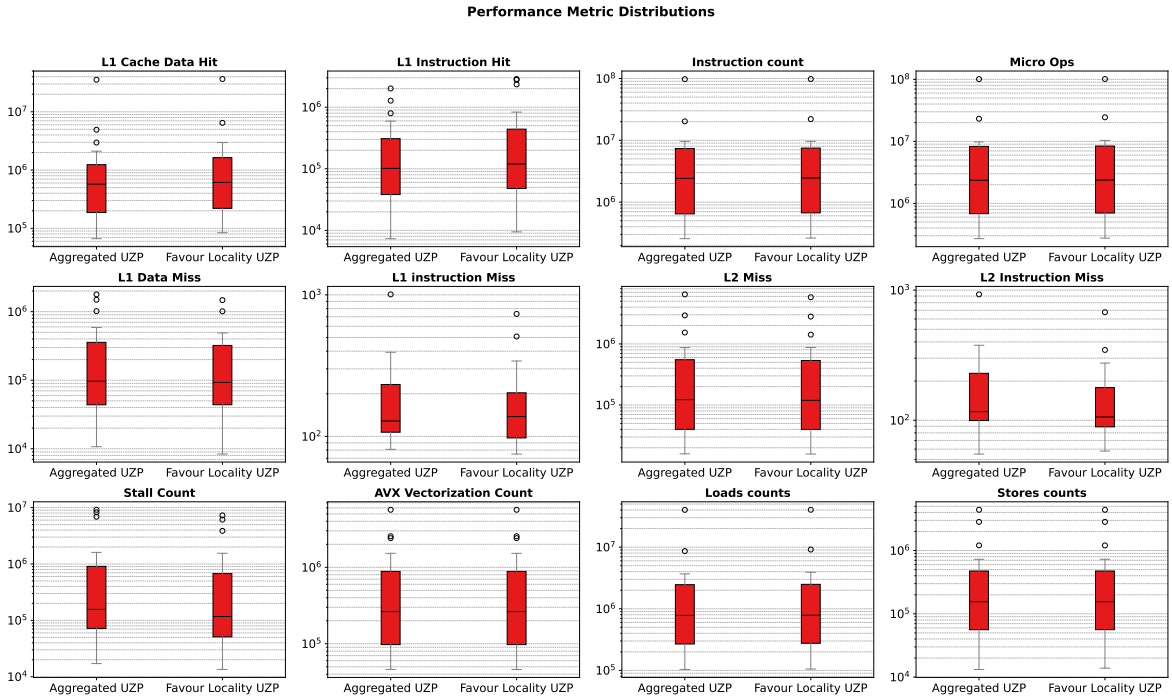

Figure 5.10: Box plot comparing the normalized PAPI counter performance metrics for Aggregated Uzp versus Favour Locality Uzp.

This combined approach ensures that the tuners not only compress the representation but also expose more predictable memory access patterns, critical for achieving high throughput on modern architectures.

Although the Favor Locality Tuner improves memory access regularity, which is also evident from the Figure 5.10 the overall performance gains compared to the Aggregator Tuner remain modest. The observed improvements are incremental, with only marginal differences between the two configurations. Further enhancements, particularly in tile-based tuning strategies, could unlock greater potential. Reordering origin locations after polyhedral shapes have been expanded through aggregation becomes increasingly challenging, as it requires careful consideration of the shape bounds used for instantiation. While additional tile-based reordering experiments were explored, they did not yield significant improvements in the current implementation.

Nevertheless, these experiments demonstrate the flexibility of the UZP framework and its potential for further optimization. The ability to modify origin organization, aggregation strategies,

and data layout — without altering or corrupting the underlying sparse structure information — offers a powerful foundation for continued development. Overall, the tuners validate the adaptability of the UZP representation and highlight many promising opportunities to enhance compression and performance in sparse computations such as SpMV and SpMM.

Chapter 6

Sparse Matrix Visualization Tool

In this chapter, we introduce a custom software tool developed as part of this research to visualize and analyze very large-scale sparse matrices. Sparse matrices are ubiquitous across scientific computing, machine learning, numerical simulations, and graph analysis, where datasets often contain millions or billions of elements, with only a small fraction being nonzero. Traditional matrix visualization tools frequently fail to handle such high-dimensional sparse data due to performance limitations or lack of interactivity, making it difficult to inspect structural properties at scale.

To address this gap, we developed an efficient matrix visualization tool capable of loading, rendering, and interacting with extremely large sparse matrices stored in Matrix Market (.mtx) format. The tool supports scalable matrix traversal, coordinate inspection, and interactive pattern extraction, enabling researchers to efficiently explore sparsity patterns without incurring high computational overhead. To show how tool user interface look we have added the figures 6.1 & 6.2 that load portion of selected large-scale matrices such as `apache2.mtx` & *3DSpectralwave2.mtx*. The `apache2.mtx` which contains over 715,000 rows and columns and nearly 5 million nonzeros, showcasing its ability to load and render datasets that exceed the practical limits of traditional visualization methods.

The primary motivation for building this tool stems from the need to visually inspect matrix structures, identify locality, and detect patterns in sparsity that are critical for algorithmic optimization. Structural features such as diagonal dominance, banded sparsity, or block patterns can inform the design of more efficient solvers, preconditioners, or storage schemes. In the context of graph-based algorithms, where sparse matrices often represent adjacency matrices, visualization enables the intuitive detection of communities, isolated nodes, and connectivity patterns.

The tool also facilitates hands-on pattern mining, manual pattern selection, and exploratory data analysis in domains where direct inspection of sparsity is critical yet traditionally infeasible. Overall, this contribution supports a wide range of research activities that require insight into spar-

sity structure, providing a practical foundation for both algorithm development and performance optimization.

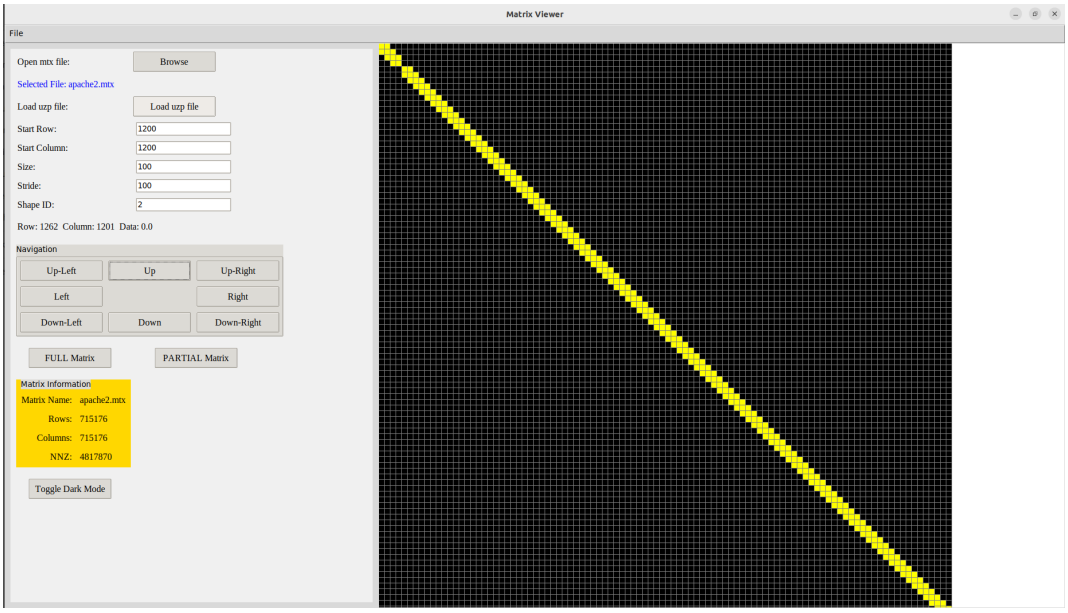

Figure 6.1: Interactive sparse matrix visualization tool displaying a tiled view of the apache2.mtx matrix

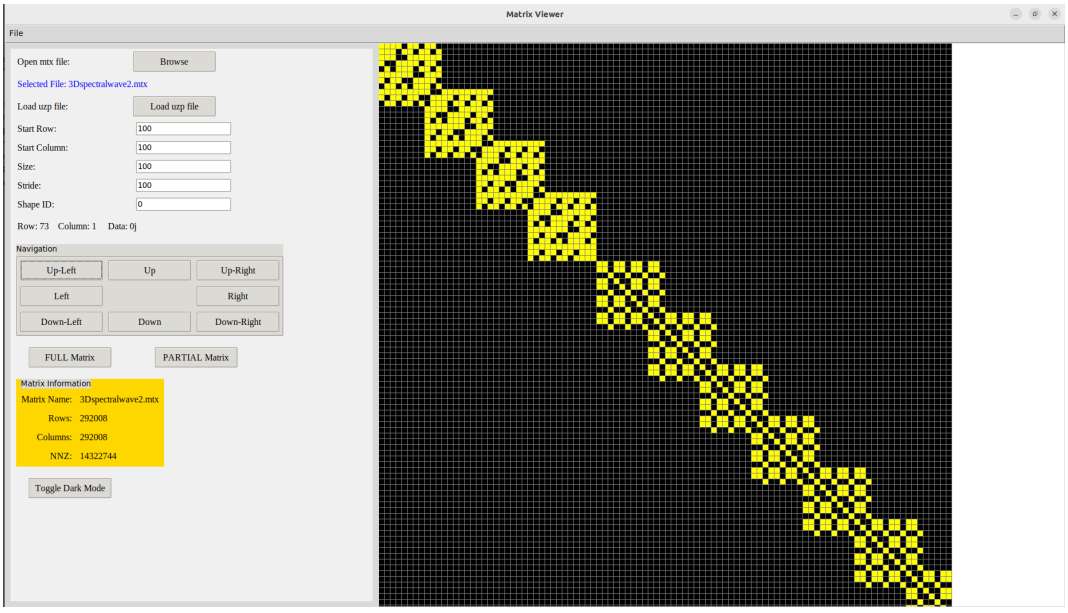

Figure 6.2: Interactive sparse matrix visualization tool displaying a tiled view of the 3DSpectralwave2.mtx matrix.

Chapter 7

Conclusion and Future Work

This thesis presented an in-depth exploration of optimizing Union of $\mathcal{Z}$-Polyhedra (UZP) [26] representation towards improving the sparse computations performance for SpMV on CPU. Sparse matrices, fundamental to many computational domains such as scientific simulation, machine learning, and graph analytics, present unique challenges in storage efficiency and computational performance due to their inherent irregularity. Traditional sparse formats like CSR, CSC, and COO efficiently represent arbitrary sparsity, but their reliance on indirect memory accesses often limits performance on modern architectures.

The UZP format was introduced as a structured, highly flexible representation that models sparse structures as unions of integer polyhedra intersected with affine lattices. This approach unifies the representation of regular and irregular sparsity patterns and enables efficient executor design through decoupling shape descriptions from their instantiation points.

Building on the core capabilities of UZP, this work developed a suite of tuners-Aggregator, Favor Locality, and Tile-based-that transform UZP representations to further optimize performance without altering the underlying sparse structure. These tuners exploit opportunities to merge regularly strided origin points, improve access locality, and restructure spatial layout to better suit hardware characteristics. A lightweight generic executor framework was also developed, supporting flexible operation over tuned UZP representations, with optional shape specific macro specialized loop next generation to enhance compiler optimizations.

Experimental evaluations conducted over a suite of 229 matrices from the SuiteSparse collection demonstrated that the tuners significantly improve storage compression, SIMD utilization, and computational throughput for Sparse Matrix-Vector Multiplication (SpMV) operations. In particular, the Aggregator tuner was shown to reduce the number of origins and improve execution locality, while the Favor Locality tuner further enhanced memory behavior through reordering strategies. Together, these techniques demonstrated the effectiveness of post-optimization in the

UZP ecosystem and validated the viability of tunable, structured sparse representations for practical high-performance computing.

Future Work. While this work establishes the effectiveness of tunable transformations within UZP, several exciting research directions remain open. Future efforts could explore adaptive tuning strategies that automatically select or combine tuners based on matrix characteristics or hardware profiles. Additionally, extending the UZP framework to fully support multi-threaded execution, heterogeneous compute platforms (e.g., GPUs and accelerators), and integration with modern compiler toolchains such as MLIR presents promising avenues.

Chapter 8

Personal Bibliography

Alonso Rodríguez-Iglesias, Santoshkumar T. Tongli, Emily Tucker, Louis-Noël Pouchet, Gabriel Rodríguez, and Juan Touriño. 2025. Modular Construction and Optimization of the UZP Sparse Format for SpMV on CPUs. Proc. ACM Program. Lang. 9, PLDI, Article 232 (June 2025), 25 pages. https://doi.org/10.1145/3729335

Bibliography

[1] G. Agrawal, J. Saltz, and R. Das. Interprocedural partial redundancy elimination and its application to distributed memory compilation. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI*, pages 258–269, La Jolla, CA, USA, 1995. doi: 10.1145/223428.207157.

[2] Travis Augustine, Janarthanan Sarma, Louis-Noël Pouchet, and Gabriel Rodríguez. Generating piecewise-regular code from irregular structures. In *40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI*, page 625–639, Phoenix, AZ, USA, 2019. ISBN 9781450367127. doi: 10.1145/3314221.3314615.

[3] C. Bastoul. Code generation in the polyhedral model is easier than you think. In *13th International Conference on Parallel Architectures and Compilation Techniques, PACT*, pages 7–16, Antibes, France, 2004. doi: 10.1109/PACT.2004.1342537.

[4] N. Bell and M. Garland. Implementing sparse matrix-vector multiplication on throughput-oriented processors. In *ACM/IEEE Conference on High Performance Computing, SC*, Portland, OR, USA, 2009. doi: 10.1145/1654059.1654078.

[5] Uday Bondhugula, Albert Hartono, J. Ramanujan, and P. Sadayappan. A practical automatic polyhedral program optimization system. In *29th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI*, pages 101–113, Tucson, AZ, USA, 2008. doi: 10.1145/1375581.1375595.

[6] Aydin Buluç and John R. Gilbert. On the representation and multiplication of hypersparse matrices. In *2008 IEEE International Symposium on Parallel and Distributed Processing, PDP*, Miami, FL, USA, 2008. doi: 10.1109/IPDPS.2008.4536313.

[7] Aydin Buluç, Jeremy T. Fineman, Matteo Frigo, John R. Gilbert, and Charles E. Leiserson. Parallel sparse matrix-vector and matrix-transpose-vector multiplication using compressed

sparse blocks. In *21st Annual Symposium on Parallelism in Algorithms and Architectures, SPAA*, page 233–244, Calgary, AB, Canada, 2009. doi: 10.1145/1583991.1584053.

[8] K. Cheshmi, S. Kamil, M. M. Strout, and M. M. Dehnavi. Sympiler: transforming sparse matrix codes by decoupling symbolic analysis. In *International Conference for High Performance Computing, Networking, Storage and Analysis, SC*, Denver, CO, USA, 2017. doi: 10.1145/3126908.3126936.

[9] K. Cheshmi, S. Kamil, M. M. Strout, and M. M. Dehnavi. Parsy: inspection and transformation of sparse matrix computations for parallelism. In *International Conference for High Performance Computing, Networking, Storage, and Analysis, SC*, Dallas, TX, USA, 2018. doi: 10.1109/SC.2018.00065.

[10] Kazem Cheshmi. Partially strided codelet github repository, 2023. URL https://github.com/sparse-specialize/partially-strided-codelet. Commit: c03d0593411c8afc9c6861de152695c453358a04.

[11] Kazem Cheshmi, Zachary Cetinic, and Maryam Mehri Dehnavi. Vectorizing sparse matrix computations with partially-strided codelets. In *International Conference for High Performance Computing, Networking, Storage and Analysis, SC*, Dallas, TX, USA, 2022. IEEE. doi: 10.1109/SC41404.2022.00037.

[12] Kazem Cheshmi, Michelle Strout, and Maryam Mehri Dehnavi. Runtime composition of iterations for fusing loop-carried sparse dependence. In *International Conference for High Performance Computing, Networking, Storage and Analysis, SC*, Denver, CO, USA, 2023. ISBN 9798400701092. doi: 10.1145/3581784.3607097.

[13] J.W. Choi, A. Singh, and R.W. Vuduc. Model-driven autotuning of sparse matrix-vector multiply on GPUs. In *15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP*, pages 115–126, Bangalore, India, January 2010. doi: 10.1145/1837853.1693471.

[14] S. Chou, F. Kjolstad, and S. Amarasinghe. Format abstraction for sparse tensor algebra compilers. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA), 2018. doi: https://doi.org/10.1145/3276493.

[15] R. Das, P. Havlak, J. Saltz, and K. Kennedy. Index array flattening through program transformation. In *ACM/IEEE Supercomputing Conference, SC*, San Diego, CA, USA, 1995. doi: 10.1145/224170.224420.

[16] Timothy A. Davis and Yifan Hu. The university of florida sparse matrix collection. *ACM Trans. Math. Softw.*, 38(1), 2011. doi: 10.1145/2049662.2049663.

[17] Zhen Du, Jiajia Li, Yinshan Wang, Xueqi Li, Guangming Tan, and Ninghui Sun. Alphasparse: Generating high performance spmv codes directly from sparse matrices. In *International Conference for High Performance Computing, Networking, Storage and Analysis, SC*, Dallas, TX, USA, 2022. doi: 10.1109/SC41404.2022.00071.

[18] Marcos Horro, Louis-Noël Pouchet, Gabriel Rodríguez, and Juan Touriño. Custom high-performance vector code generation for data-specific sparse computations. In *International Conference on Parallel Architectures and Compilation Techniques, PACT*, page 160–171, Chicago, IL, USA, 2022. ISBN 9781450398688. doi: 10.1145/3559009.3569668.

[19] F. Kjolstad, S. Kamil, S. Chou, D. Lugato, and S. Amarasinghe. The tensor algebra compiler. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA), 2017. doi: 10.1145/3133901.

[20] A. LaMielle and M. Strout. Enabling code generation within the sparse polyhedral framework. Technical Report CS-10-102, Colorado State University, 2010. URL https://www.cs.colostate.edu/TechReports/Reports/2010/tr10-102.pdf.

[21] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. Mlir: Scaling compiler infrastructure for domain specific computation. In *2021 IEEE/ACM International*

Symposium on Code Generation and Optimization, CGO, pages 2–14, Seoul, South Korea, 2021. doi: 10.1109/CGO51591.2021.9370308.

[22] Weifeng Liu and Brian Vinter. Csr5: An efficient storage format for cross-platform sparse matrix-vector multiplication. In *29th ACM on International Conference on Supercomputing, ICS*, pages 339–350, Newport Beach, CA, USA, 2015. doi: 10.1145/2751205.2751208.

[23] R. Ponnusamy, J.H. Saltz, and A.N. Choudhary. Runtime compilation techniques for data partitioning and communication schedule reuse. In *ACM/IEEE Conference on Supercomputing, SC*, pages 361–370, Portland, OR, USA, 1993. doi: 10.1145/169627.169752.

[24] L.-N. Pouchet. PolyBench: The Polyhedral Benchmarking suite, version PolyBench/C 4.2.1, 2011. URL http://polybench.sf.net.

[25] M. Ravishankar, R. Dathathri, V. Elango, L.-N. Pouchet, J. Ramanujam, A. Rountev, and P. Sadayappan. Distributed memory code generation for mixed irregular/regular computations. In *20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP*, pages 65–75, San Francisco, CA, USA, 2015. doi: 10.1145/2688500.2688515.

[26] Alonso Rodríguez-Iglesias, Santoshkumar T. Tongli, Emily Tucker, Louis-Noël Pouchet, Gabriel Rodríguez, and Juan Touriño. Modular construction and optimization of the uzp sparse format for spmv on cpus. *Proc. ACM Program. Lang.*, 9(PLDI):25, June 2025. doi: 10.1145/3729335.

[27] G. Rodríguez, M. T. Kandemir, and J. Touriño. Affine modeling of program traces. *IEEE Trans. Comput.*, 68(2):294–300, 2019. doi: 10.1109/TC.2018.2853747.

[28] J. Saltz, K. Crowley, R. Mirchandaney, and H. Berryman. Run-time scheduling and execution of loops on message passing machines. *J. Parallel Distrib. Comput.*, 8(4):303–312, 1990. doi: 10.1016/0743-7315(90)90129-D.

[29] Naser Sedaghati, Te Mu, Louis-Noel Pouchet, Srinivasan Parthasarathy, and P. Sadayappan. Automatic selection of sparse matrix representation on gpus. In *29th ACM on International*

Conference on Supercomputing, ICS, pages 99–108, Newport Beach, CA, USA, 2015. doi: 10.1145/2751205.2751244.

[30] S. Sharma, R. Ponnusamy, B. Moon, Y.-S. Hwang, R. Das, and J. Saltz. Run-time and compile-time support for adaptive irregular problems. In *ACM/IEEE Conference on Supercomputing, SC*, pages 97–106, Washington, DC, USA, 1994. doi: 10.1109/SUPERC.1994.344269.

[31] Michelle Mills Strout, Mary Hall, and Catherine Olschanowsky. The sparse polyhedral framework: Composing compiler-generated inspector-executor code. *Proceedings of the IEEE*, 106(11):1921–1934, 2018. doi: 10.1109/JPROC.2018.2857721.

[32] M.M. Strout, G. George, and C. Olschanowsky. Set and relation manipulation for the sparse polyhedral framework. In *25th International Workshop on Languages and Compilers for Parallel Computing, LCPC*, pages 61–75, Tokyo, Japan, September 2012. doi: 10.1007/978-3-642-37658-0_5.

[33] A. Sukumaran-Rajam and P. Clauss. The polyhedral model of nonlinear loops. *ACM Trans. Archit. Code Optim.*, 12(4), 2016. doi: 10.1145/2838734.

[34] A. Venkat, M.S. Mohammadi, J. Park, H. Rong, R. Barik, M.M. Strout, and M. Hall. Automating wavefront parallelization for sparse matrix computations. In *International Conference for High Performance Computing, Networking, Storage and Analysis, SC*, Salt Lake City, UT, USA, 2016. doi: 10.1109/SC.2016.40.

[35] Sven Verdoolaege. ISL: An integer set library for the polyhedral model. In *3rd International Congress on Mathematical Software, ICMS*, pages 299–302. Kobe, Japan, 2010. doi: 10.1007/978-3-642-15582-6_49.

[36] R. von Hanxleden, K. Kennedy, C. Koelbel, R. Das, and J. Saltz. Compiler analysis for irregular problems in Fortran D. In *6th International Workshop on Languages and Compilers*

for Parallel Computing, LCPC, pages 97–111, New Haven, CT, USA, 1992. doi: 10.1007/3-540-57502-2_42.

[37] Rich Vuduc, James W Demmel, Katherine A Yelick, Shoaib Kamil, Rajesh Nishtala, and Benjamin Lee. Performance optimizations and bounds for sparse matrix-vector multiply. In *Proceedings of the 2002 ACM/IEEE Conference on Supercomputing, SC*, Baltimore, MD, USA, 2002. doi: 10.1109/SC.2002.10025.

[38] R.W. Vuduc. *Automatic Performance Tuning of Sparse Matrix Kernels*. PhD thesis, University of California, January 2004.

[39] Lucas Wilkinson, Kazem Cheshmi, and Maryam Mehri Dehnavi. Register tiling for unstructured sparsity in neural network inference. *Proceedings of the ACM on Programming Languages*, 7(PLDI):1995–2020, 2023. doi: 10.1145/3591302.

[40] Jeremiah Willcock and Andrew Lumsdaine. Accelerating sparse matrix computations via data compression. In *20th Annual International Conference on Supercomputing, ICS*, page 307–316, Cairns, QLD, Australia, 2006. ISBN 1595932828. doi: 10.1145/1183401.1183444.

[41] S. Williams, L. Oliker, R.W. Vuduc, J. Shalf, K.A. Yelick, and J. Demmel. Optimization of sparse matrix-vector multiplication on emerging multicore platforms. *Parallel Comput.*, 35 (3):178–194, 2009. doi: 10.1145/1362622.1362674.

[42] Biwei Xie, Jianfeng Zhan, Xu Liu, Wanling Gao, Zhen Jia, Xiwen He, and Lixin Zhang. Cvr: efficient vectorization of spmv on x86 processors. In *International Symposium on Code Generation and Optimization, CGO*, page 149–162, Vienna, Austria, 2018. doi: 10.1145/3168818.